UNIT 4

# HDFS FEDERATION

## 4.1 HDFS FEDERATION

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 0.23 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.

Each Namenode Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and furthermore the failure of one namenode does not affect the availability of the namespaces managed by other namenodes. Block pool storage is not partitioned, however, so datanodes register with each namenode in the cluster and store blocks from multiple block pools.

## 4.2 HDFS HIGH-AVAILABILITY

The namenode is still a single point of failure (SPOF), since 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. In such an event the whole Hadoop system would effectively be out of service until a new namenode could be brought online. 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.

The 0.23 release series of Hadoop remedies this situation by adding support for HDFS highavailability (HA). In this implementation there is a pair of namenodes in an

activestandby configuration. In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant interruption.

A few architectural changes are needed to allow this to happen:

1. The namenodes must use highly-available shared storage to share the edit log.
2. Datanodes must send block reports to both namenodes since the block mappings are stored in a namenode's memory, and not on disk.
3. Clients must be configured to handle namenode failover, which uses a mechanism that is transparent to users.

**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. Failover controllers are pluggable, but the first implementation uses ZooKeeper to ensure that only one namenode is active.

Failover may also be initiated manually by an adminstrator, in the case of routine maintenance, for example. This is known as a graceful failover, since the failover controller arranges an orderly transition for both namenodes to switch roles. In the case of an ungraceful failover, 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.

## 4.3 THE COMMAND-LINE INTERFACE

We're going to have a look at HDFS by interacting with it from the command line. There are many other interfaces to HDFS, but the command line is one of the simplest and, to many developers, the most familiar. We are going to run HDFS on one machine, so first follow the instructions for setting up Hadoop in pseudo-distributed mode in Appendix A. Later you'll see how to run on a cluster of machines to give us scalability and fault tolerance.

There are two properties that we set in the pseudo-distributed configuration that deserve further explanation. The first is fs.default.name, set to hdfs://localhost/, which is used to set a default filesystem for Hadoop. Filesystems are specified by a URI, and here we have used an hdfs URI to configure Hadoop to use HDFS by default. The HDFS daemons

will use this property to determine the host and port for the HDFS namenode. We'll be running it on localhost, on the default HDFS port, 8020. And HDFS clients will use this property to work out where the namenode is running so they can connect to it.

We set the second property, dfs.replication, to 1 so that HDFS doesn't replicate filesystem blocks by the default factor of three. When running with a single datanode, HDFS can't replicate blocks to three datanodes, so it would perpetually warn about blocks being under-replicated. This setting solves that problem.

## 4.4 BASIC FILE SYSTEM OPERATIONS

The filesystem is ready to be used, and we can do all of the usual filesystem operations such as reading files, creating directories, moving files, deleting data, and listing directories. You can type hadoop fs -help to get detailed help on every command. Start by copying a file from the local filesystem to HDFS:

hadoop fs -copyFromLocal input/docs/quangle.txt hdfs://localhost/user/tom/quangle.txt

This command invokes Hadoop's filesystem shell command fs, which supports a number of subcommands—in this case, we are running -copyFromLocal. The local file quangle.txt is copied to the file /user/tom/quangle.txt on the HDFS instance running on localhost. In fact, we could have omitted the scheme and host of the URI and picked up the default, hdfs://localhost, as specified in core-site.xml:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt /user/tom/quangle.txt
```

We could also have used a relative path and copied the file to our home directory in HDFS, which in this case is */user/tom*:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt quangle.txt
```

Let's copy the file back to the local filesystem and check whether it's the same:

```
% hadoop fs -copyToLocal quangle.txt quangle.copy.txt
% md5 input/docs/quangle.txt quangle.copy.txt
MD5 (input/docs/quangle.txt) = a16f231da6b05e2ba7a339320e7dacd9
MD5 (quangle.copy.txt) = a16f231da6b05e2ba7a339320e7dacd9
```

The MD5 digests are the same, showing that the file survived its trip to HDFS and is back intact. Finally, let's look at an HDFS file listing. We create a directory first just to see how it is displayed in the listing:

```
% hadoop fs -mkdir books
% hadoop fs -ls .
Found 2 items
drwxr-xr-x   - tom supergroup          0 2009-04-02 22:41 /user/tom/books
-rw-r--r--   1 tom supergroup        118 2009-04-02 22:29 /user/tom/quangle.txt
```

The information returned is very similar to the Unix command ls -l, with a few minor differences. The first column shows the file mode. The second column is the replication factor of the file (something a traditional Unix filesystem does not have). Remember we set the default replication factor in the site-wide configuration to be 1, which is why we see the same value here.

The entry in this column is empty for directories since the concept of replication does not apply to them—directories are treated as metadata and stored by the namenode, not the datanodes. The third and fourth columns show the file owner and group. The fifth column is the size of the file in bytes, or zero for directories. The sixth and seventh columns are the last modified date and time. Finally, the eighth column is the absolute name of the file or directory.

## 4.5 HADOOP FILES SYSTEMS

Hadoop has an abstract notion of filesystem, of which HDFS is just one implementation. The Java abstract class org.apache.hadoop.fs.FileSystem represents a filesystem in Hadoop, and there are several concrete implementations, which are described in Table.

**Table : Hadoop filesystems**

| Filesystem | URI scheme | Java implementation (all under org.apache.hadoop) | Description |
|---|---|---|---|
| Local | *file* | `fs.LocalFileSystem` | A filesystem for a locally connected disk with client-side checksums. Use `RawLocalFileSystem` for a local filesystem with no checksums. See "LocalFileSystem" on page 84. |
| HDFS | *hdfs* | `hdfs.DistributedFileSystem` | Hadoop's distributed filesystem. HDFS is designed to work efficiently in conjunction with MapReduce. |
| HFTP | *hftp* | `hdfs.HftpFileSystem` | A filesystem providing read-only access to HDFS over HTTP. (Despite its name, HFTP has no connection with FTP.) Often used with *distcp* (see "Parallel Copying with distcp" on page 76) to copy data between HDFS clusters running different versions. |

| | | | |
|---|---|---|---|
| HSFTP | *hsftp* | `hdfs.HsftpFileSystem` | A filesystem providing read-only access to HDFS over HTTPS. (Again, this has no connection with FTP.) |
| WebHDFS | *webhdfs* | `hdfs.web.WebHdfsFileSystem` | A filesystem providing secure read-write access to HDFS over HTTP. WebHDFS is intended as a replacement for HFTP and HSFTP. |
| HAR | *har* | `fs.HarFileSystem` | A filesystem layered on another filesystem for archiving files. Hadoop Archives are typically used for archiving files in HDFS to reduce the namenode's memory usage. See "Hadoop Archives" on page 78. |
| KFS (CloudStore) | *kfs* | `fs.kfs.KosmosFileSystem` | CloudStore (formerly Kosmos filesystem) is a distributed filesystem like HDFS or Google's GFS, written in C++. Find more information about it at *http://kosmosfs.sourceforge.net/*. |
| FTP | *ftp* | `fs.ftp.FTPFileSystem` | A filesystem backed by an FTP server. |
| S3 (native) | *s3n* | `fs.s3native.NativeS3FileSystem` | A filesystem backed by Amazon S3. See *http://wiki.apache.org/hadoop/AmazonS3*. |
| S3 (block-based) | *s3* | `fs.s3.S3FileSystem` | A filesystem backed by Amazon S3, which stores files in blocks (much like HDFS) to overcome S3's 5 GB file size limit. |
| Distributed RAID | *hdfs* | `hdfs.DistributedRaidFileSystem` | A "RAID" version of HDFS designed for archival storage. For each file in HDFS, a (smaller) parity file is created, which allows the HDFS replication to be reduced from three to two, which reduces disk usage by 25% to 30%, while keeping the probability of data loss the same. Distributed RAID requires that you run a `RaidNode` daemon on the cluster. |
| View | *viewfs* | `viewfs.ViewFileSystem` | A client-side mount table for other Hadoop filesystems. Commonly used to create mount points for federated namenodes (see "HDFS Federation" on page 49). |

Hadoop provides many interfaces to its filesystems, and it generally uses the URI scheme to pick the correct filesystem instance to communicate with. For example, the filesystem shell that we met in the previous section operates with all Hadoop filesystems. To list the files in the root directory of the local filesystem, type:

```
% hadoop fs -ls file:///
```

Although it is possible (and sometimes very convenient) to run MapReduce programs that access any of these filesystems, when you are processing large volumes of data, you should choose a distributed filesystem that has the data locality optimization, notably HDFS.

### Interfaces

Hadoop is written in Java, and all Hadoop filesystem interactions are mediated through the Java API. The filesystem shell, for example, is a Java application that uses the Java FileSystem class to provide filesystem operations. The other filesystem interfaces are discussed briefly in this section. These interfaces are most commonly used with HDFS, since the other filesystems in Hadoop typically have existing tools to access the underlying filesystem (FTP clients for FTP, S3 tools for S3, etc.), but many of them will work with any Hadoop filesystem.

## 4.6 INFORMATION MANAGEMENT

Information management is the entire range of technical, operational, and social functions of a system that is used to handle information. Individuals, social networks of individuals, organizations, businesses, and governments all engage in some form of information management.

### 4.6.1 The Big Data Foundation

The Big Data foundation is composed of two major systems. The first stores the data and the second processes it.

1. **Big Data storage**

Big Data storage is often synonymously interchanged with the Hadoop File System (HDFS), but traditional data warehouses can also house Big Data. HDFS is distributed data storage that has become the de facto standard because you can store any type of data without limitations on the type or amount of data. One of the reasons HDFS has become so popular is that you don't have to do any "set up" to store the data. In traditional databases, you need to do quite a bit of "set up" in order to store data.

2. **Big Data processing**

Big Data processing involves the manipulation and calculations that occur on the Big Data. Traditional databases have differing abilities to process Big Data sets effectively and efficiently. Additionally, there is a wide degree of variance in how database software exploits the underlying hardware architecture. Database software that is hardware agnostic doesn't take full advantage of the underlying hardware architecture. But some database

software is tightly coupled with the hardware architecture so as to exploit the unique hardware processing capabilities to get throughput processing.

Data warehouse appliances fall into this category and have varying degrees of performance advantages due to this tight coupling.

Today, the de facto processing software for HDFS is MapReduce, which was discussed in the previous chapter. MapReduce is a fault-tolerant parallel programming framework that was designed to harness distributed processing capabilities. MapReduce automatically divides the processing workload into smaller workloads that are distributed. When working with HDFS, data manipulation and calculations are programmed using the MapReduce framework in the language of choice (typically Java programming language).

To maximize the processing throughput, MapReduce assumes that the workloads being distributed are independent tasks and the workload is equally divided just like the division of labor to the 10 assembly lines to produce 50 trucks each. However, if there are dependencies in the processing workload, the MapReduce framework is unaware of those dependencies.

The programmer has to be aware of those dependencies and has to specifically divide the workload up in the program understanding that MapReduce will automatically distribute tasks. This type of programming is called parallel programming . Just like the production planning that has to occur in dividing the workload between assembly lines that only produce engines and all the other assembly lines was more complicated, parallel programming is more complicated. One of the benefi ts of MapReduce and some data warehouse appliances is that the easier independent processing is automatically handled by the framework or date warehouse appliance.

MapReduce was designed to be fault tolerant because, when using unknown hardware whose reliability is unknown, there needs to be a way to gracefully handle processing failure. Fault tolerant software is designed to automaticallyrecover and handle processing failures, which makes it highly reliable.

MapReduce along with many data warehouse appliances are fault tolerant.

There are a few different typical workfl ows that become processing bottlenecks as data gets increasingly larger. The fi rst one is the speed of ingesting or loading the data. The second one is the speed of analytics computational processing (or "number crunching" as it 's often referred to). The third is the speed at which you can perform your analytics on-demand and respond to the business.

### 4.6.2 Big Data Computing Platforms

A big data platform works to wrangle this amount of information, storing it in a manner that is organized and understandable enough to extract useful insights. Big data platforms utilize a combination of data management hardware and software tools to aggregate data on a massive scale, usually onto the cloud.

There are many different computing techniques available today, parallel computing platforms are the only platforms suited to handle the speed and volume of data being produced today. There are three prominent parallel computing options available today

1. Clusters or grids
2. Massively parallel processing (MPP)
3. High performance computing (HPC)

**1. Clusters or grids**

Clusters and grids are types of computing where servers are loosely coupled and networked together for distributing workloads. Clusters or grid environments can be either homogeneous or heterogeneous commodity hardware environments. People tend to use cluster or grid approaches because the perceived total cost of ownership is negligible, since they can buy commodity hardware and pull it together. Grid environments do provide lower-cost storage but the price differential between a grid and other parallel alternatives is not as significant when you take a total cost of ownership perspective, as there are human costs to factor into setting up and maintaining the grid.

A variation on the cluster or grid is a public cloud environment, where a vendor such as Amazon or EC2 has set up a cluster or grid and sells space and computing power to customers via the Internet. Cloud environments have become popular for a few reasons. One is that they are "elastic," which means that I pay as I grow/shrink or only pay for the space/processing that I need today and, as my data/processing needs increase/decrease, I buy more or less from the vendor.

The downside to cloud environments for Big Data is that at some point, the time it takes to get data to the cloud becomes longer and that may not be acceptable for the particular business requirement. Private clouds are a grid internal to an organization. Typically, private clouds are a "shared" environment where several business units share the cost of the Big Data infrastructure and support.

### 2. Massively parallel processing (MPP)

Massively parallel processing platforms are essentially a grid inside of a box. MPP platforms combine storage, memory, and compute to create a platform. Typically, MPP platforms are used for known high-value use cases. EMC Greenplum and ParAccel are examples of MPP platforms.

MPP appliances place the storage, memory, and compute into a single machine that is optimized for performance and scalability. In an appliance, the network that connects the storage and compute is designed for optimal throughput. MPP appliances go one step further to design the software for the specific MPP platform, which allows the software to maximize the storage and computational capabilities of the MPP device.

MPP appliances have been used in enterprises for 101 years and with that maturity the software is designed to be easier to use for analytics. While you can typically write your own parallel programs to embed analytics into the platform (the equivalent of writing Mappers and Reducers) you don 't have to because there are a variety of easier to use tools available on MPP appliances. Typically, these tools automatically perform the parallel processing under the covers. This makes it easier to use and delivers on performance and scalability.

MPP appliances are designed for price performance. Examples of MPP appliances are IBM Netezza (just recently renamed IBM PureData System for Analytics) and Teradata.

### 3. High performance computing (HPC)

The next option is a high-performance computing option, or HPC environment. HPC environments are designed for high-speed fl oating point processing and much of the calculations are done in memory, which renders the highest computational performance possible. Cray Computers and IBM Blue Gene are examples of HPC environments.

HPC environments are predominantly used by research organizations and by business units that demand very high scalability and computational performance, where the value being created is so huge and strategic that cost is not the most important consideration. While HPC environments have been around for quite some time, they are used for specialty applications and primarily provide a programming environment for custom application development.

### 4.6.3 Big Data Computation

Big Data computing platforms are all parallel computing environments that require some type of parallel programming. Parallel programming is another order of magnitude more difficult than standard programming. In a parallel computing environment, you have to think about how to you structure your code in a way that when the workload is distributed to the parallel computing processors, the results are combined to produce the correct result.

The various Big Data computing platform vendors have varied approaches to the level of effort involved in parallel programming. Some environments provide software layers or tools to make it easier while others simply provide standard programming tools, and the hard work to do the parallel programming is left to the skill and experience of the programmer. The typical parallel computing environments include:

1. MapReduce
2. In-database analytics
3. Message passing interface (MPI)

**1. MapReduce**

Let's take a look at how MapReduce, a common parallelization technique used with Big Data, works. MapReduce is typically used in conjunction with HDFS but there are also implementations of MapReduce for data warehouses. In the MapReduce framework, a master controller distributes work via a Mapper function to all of the available processors. Each of the processors independently performs the Mapper task and the results are fed into a Reducer task, which summarizes the work from Mappers.

Today, the Mappers and Reducers are custom code written in Java. However, there are higherlevel tools and analytics available that use MapReduce under the covers. For example, Pig and Hive are two open-source tools that provide a SQL-like interface to HDFS because sometimes it 's easier to use SQL than to custom code the same capability that is inherent in SQL. There are other open-source projects such as Mahout and Lucene, which are evolving libraries of analytic functions that are typically written in MapReduce.

**2. In-database analytics**

The second software parallelization technique is an overloaded term called in-database analytics. There are two ways the term is used. The first is to refer to the collection of SQL

language extension capabilities called user defined extensions that take custom code and execute the code in database. The second are prebuilt, parallelized analytics that execute in the database. Both of these methods move the computational processing next to the data where the data resides so that you avoid all of the latency and performance bottlenecks associated with moving Big Data.

There are three types of user-defined extensions defined by the SQL language:

1. User-defi ned function (UDF), which performs a task and returns one value
2. User-defi ned aggregate (UDA), which summarizes a group of data and returns one value
3. User-defi ned table function (UDTF), which takes data of one shape (i.e., 10 columns and 100 rows) and reshapes the output (i.e., 2 columns and 552 rows)

In a parallel data warehouse, SQL along with user-defined extensions are executed in parallel automatically without having to explicitly perform the parallel programming. However, just like in MapReduce, a programmer who is writing a user-defined extension needs to understand how to program calculations such as median even though SQL automatically distributes the workload. Mappers are equivalent to UDTFs and Reducers are equivalent to UDAs.

Prebuilt in-database analytics leverage user-defined extensions to encapsulate analytic functions such as linear regression, decision trees, and many other analytic functions. In this method, the parallel programming has already been performed and the analytic function can simply be invoked to perform the specified task. Examples of out-of-box, prebuilt, in-database analytics are included in IBM Netezza, DB Lytix by Fuzzy Logix, and Alpine Data Miner.

Parallel-compute environments are shared nothing environments whereby design, each processor, and its associated memory are unaware of the other processors.

### 3. Message passing interface (MPI)

The third software parallelization technique is message passing interface, which is interprocess communication software to facilitate sharing of information between processors in a shared nothing compute environment. This is a collection of software processes that allow a distributed environment to share information between the processors. For the median calculation, MPI is used to share interim processing results between the processors while attempting to find the middle number.

Each parallel-compute platform uses one or more of these software parallelization techniques. It's very typical for HPC environments to use MPI, it 's very typical for MPP environments to use in-database analytics, and it 's very typical for grids to use MapReduce. But there are hybrid approaches as well.

For example, IBM Netezza has an in-database MapReduce capability, MPI capability, and both in-database analytic capabilities while Teradata Aster has a SQLMR capability.

## 4.7 MORE ON BIG DATA STORAGE

As mentioned earlier, Big Data storage is often used interchangeably with HDFS. However, data warehouse have been used for Big Data storage of known value for quite some time and have evolved to several storage techniques. By far the most common and popular is the relational database that stores information in rows and columns.

In relational databases the primary access is via the row. Another storage technique is a columnar storage scheme. While a columnar database has rows and columns, information is predominantly accessed via the columns. There are also hybrid approaches that allow for some information to be easily accessed via row and other data to be easily accessed via columns. Examples of relational data warehouses include IBM Netezza, Teradata, and Oracle Exadata. Vertica [an HP company] is the most prevalent columnar storage data warehouse.

### 4.7.1 Big Data Computational Limitations

With all the Big Data buzz and gee whiz bang technology available today, you might conclude that there are no limitations to processing Big Data but laws of physics still apply. The typical Big Data computing platform limitations encountered are:

1. Disk bound
2. I/O bound
3. Memory bound
4. CPU bound

**1. Disk bound**

Disk bound means you simply don 't have enough storage capacity (e.g., have 2 PB of data and there is only 500 TB of storage capacity).

2. **I/O bound**

I/O bound is when there isn't enough bandwidth to move around the data to meet the needs of the business. Think about this as having a pipe with a 10-inch diameter and trying to squeeze 50 TB of Big Data through the pipe in five minutes in order to provide the service level agreements (SLAs) that your business requires when it would take a 100-inch pipe to meet the SLAs.

3. **Memory bound and CPU bound**

Computational platforms use a combination of memory and CPUs to process Big Data analytics. Memory is like the scrap of paper we use when doing a more complicated math problem. Most analytic software today attempts to load all the data needed for a calculation into memory and then uses the CPU to perform the calculation. This technique is very fast but is bounded by the available memory. Memory is limited on various computing platforms because memory is still fairly expensive. There are heavy in-memory systems, such as SAP Hana, which are quickly becoming cost effective solutions. Here is the typical scenario when an analytic process become memory bound:

Similarly, some analytics are more computationally intensive, which means that they use more of the capacity of the CPU. An example of a mathematically intensive analytic is Monte Carlo simulation. When executing a Monte Carlo simulation, a calculation or set of calculations are being performed with different starting conditions or parameters. In this huge "what-if" process, there are many computational intensive calculations being performed concurrently that can simply overwhelm and exhaust the CPU capacity.

## IMPORTANT QUESTIONS

1. What is hdfs federation? explain about hdfs high availability and the command-line interface.
2. Discuss briefly about basic file system operations and hadoop files stems.
3. What is information management? Explain about the big data foundation and big data computing platforms.
4. Discuss briefly about big data computation and more on big data storage.
5. Explain about big data computational limitations.