
Distributed File Systems - Large-Scale FileSystem Organization – HDFS concepts – MapReduce Execution, Algorithms using MapReduce, Matrix-Vector Multiplication – Hadoop YARN

2.1Distributed File Systems

Most computing is done on a single processor, with its main memory, cache, and local disk (a compute node). In the past, applications that called for parallel processing, such as large scientific calculations, were done on special-purpose parallel computers with many processors and specialized hardware. However, the prevalence of large-scale Web services has caused more and more computing to be done on installations with thousands of compute nodes operating more or less independently. In these installations, the compute nodes are commodity hardware, which greatly reduces the cost compared with special-purpose parallel machines. These new computing facilities have given rise to a new generation of programming systems. These systems take advantage of the power of parallelism and at the same time avoid the reliability problems that arise when the computing hardware consists of thousands of independent components, any of which could fail at any time.

Distributed File System Requirements:

In DFS first we need: access transparency and location transparency. Distributed File System Requirements are performance, scalability, concurrency control, fault tolerance and security requirements emerged and tolerance and security requirements emerged and were met in the later phases of DFS development.

- ✓ **Access transparency:** Client programs should be unaware of the distribution of files.
- ✓ **Location transparency:** Client program should see a uniform namespace. Files should be able to be relocated without changing their path name.
- ✓ **Mobility transparency:** Neither client programs nor system admin program tables in the client nodes should be changed when files are moved either automatically or by the system admin.
- ✓ **Performance transparency:** Client programs should continue to perform well on load within a specified range.
- ✓ **Scaling transparency:** increase in size of storage and network size should be transparent.

The following are the characteristics of these computing installations and the specialized file systems that have been developed to take advantage of them.

Physical Organization of Compute Nodes

The new parallel-computing architecture, sometimes called cluster computing, is organized as follows. Compute nodes are stored on racks, perhaps 8–64on a rack. The nodes on a single rack are connected by a network, typically gigabit Ethernet. There can be many racks of compute nodes, and racks are connected by another level of network or a switch. The bandwidth of inter-rack communication is somewhat greater than the intra rack Ethernet, but given the number of pairs of nodes that might need to communicate between racks, this bandwidth may be essential.

However, there may be many more racks and many more compute nodes per rack. It is a fact of life that components fail, and the more components, such as compute nodes and interconnection networks, a system has, the more frequently something in the system will not be working at any given time. Some important calculations take minutes or even hours on thousands of compute nodes. If we had to abort and restart the computation every time one component failed, then the computation might never complete successfully.

The solution to this problem takes two forms:

1. Files must be stored redundantly. If we did not duplicate the file at several compute nodes, then if one node failed, all its files would be unavailable until the node is replaced. If we did not back up the files at all, and the disk crashes, the files would be lost forever.

2. Computations must be divided into tasks, such that if any one task fails to execute to completion, it can be restarted without affecting other tasks. This strategy is followed by the map-reduce programming system

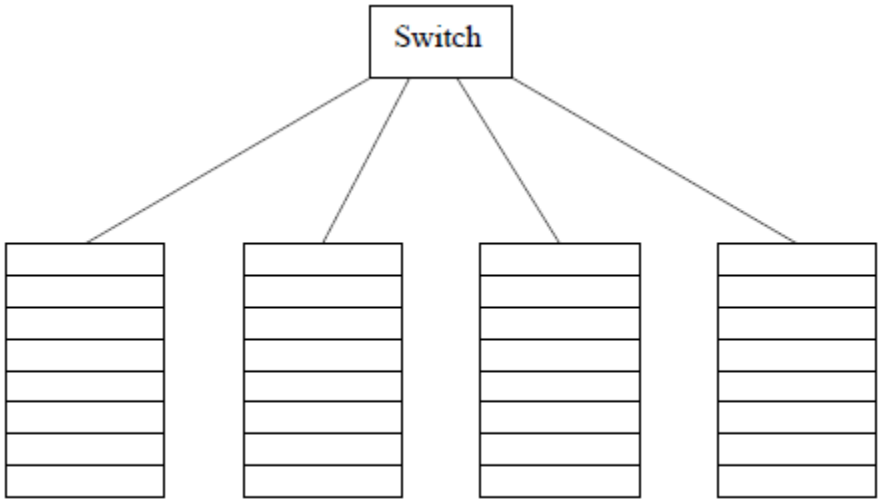

Fig: architecture of a large-scale computing system

2.2 Large-Scale File-System Organization

To exploit cluster computing, files must look and behave somewhat differently from the conventional file systems found on single computers. This new file system, often called a distributed file system or DFS (although this term had other meanings in the past), is typically used as follows.

• Files can be enormous, possibly a terabyte in size. If you have only small files, there is no point using a DFS for them.

• Files are rarely updated. Rather, they are read as data for some calculation, and possibly additional data is appended to files from time to time.

Files are divided into chunks, which are typically 64 megabytes in size. Chunks are replicated, perhaps three times, at three different compute nodes. Moreover, the nodes holding copies of one chunk should be located on different racks, so we don't lose all copies due to a rack failure. Normally, both the chunk size and the degree of replication can be decided by the user.

To find the chunks of a file, there is another small file called the master node or name node for that file. The master node is itself replicated, and a directory for the file system as a whole knows where to find its copies. The directory itself can be replicated, and all participants using the DFS know where the directory copies are.

2.4 Map-Reduce

Map-reduce is a style of computing that has been implemented several times. Use an implementation of map-reduce to manage many large-scale computations in a way that is tolerant of hardware faults. All you need to write are two functions, called Map and Reduce, while the system manages the parallel execution, coordination of tasks that execute Map or Reduce. In brief, a map-reduce computation executes as follows:

1. Some number of Map tasks each are given one or more chunks from a distributed file system. These Map tasks turn the chunk into a sequence of key-value pairs. The way key-value pairs are produced from the input data is determined by the code written by the user for the Map function.

2. The key-value pairs from each Map task are collected by a master controller and sorted by key. The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Re-duce task.

3. The Reduce tasks work on one key at a time, and combine all the values associated with that key in some way. The manner of combination of values is determined by the code written by

the user for the Reduce function.

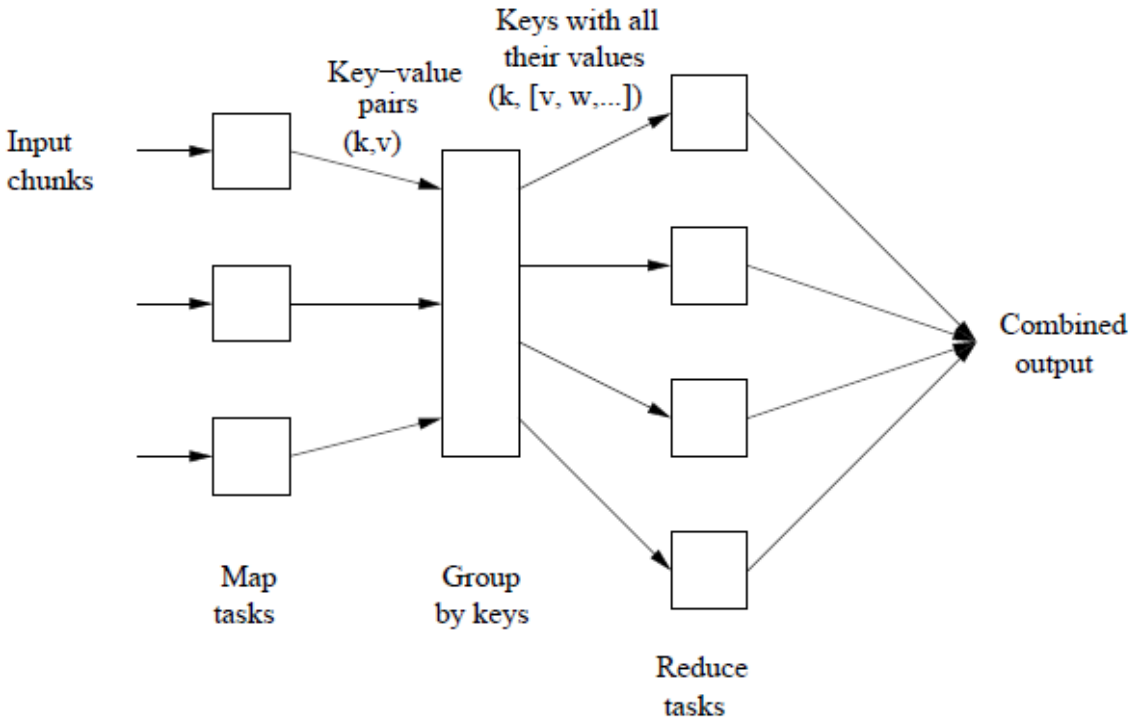

Fig: Map-Reduce computation

The Map Tasks

We view input files for a Map task as consisting of elements, which can be any type: a tuple or a document, for example. A chunk is a collection of elements, and no element is stored across two chunks. Technically, all inputs to Map tasks and outputs from Reduce tasks are of the key-value-pair form, but normally the keys of input elements are not relevant and we shall tend to ignore them. Insisting on this form for inputs and outputs is motivated by the desire to allow composition of several map-reduce processes.

A Map function is written to convert input elements to key-value pairs. The types of keys and values are each arbitrary. Further, keys are not "keys" in the usual sense; they do not have to be unique. Rather a Map task can produce several key-value pairs with the same key, even from the same element.

Grouping and Aggregation

Grouping and aggregation is done the same way, regardless of what Map and Reduce tasks do. The master controller process knows how many Reduce tasksthere will be, say r such

tasks. The user typically tells the map-reduce system what r should be. Then the master controller normally picks a hash function that applies to keys and produces a bucket number from 0 to r − 1. Each key that is output by a Map task is hashed and its key-value pair is put in one of r local files. Each file is destined for one of the Reduce tasks.1After all the Map tasks have completed successfully, the master controller merges the file from each Map task that are destined for a particular Reduce task and feeds the merged file to that process as a sequence of key-list-of-value pairs. That is, for each key k, the input to the Reduce task that handles key k is a pair of the form (k, [v1, v2, . . . , vn]), where (k, v1), (k, v2), . . . , (k, vn) are all the key-value pairs with key k coming from all the Map tasks.

The Reduce Tasks

The Reduce function is written to take pairs consisting of a key and its list of associated values and combine those values in some way. The output of a Reduce task is a sequence of key-value pairs consisting of each input key k that the Reduce task received, paired with the combined value constructed from the list of values that the Reduce task received along with key k. The outputs from all the Reduce tasks are merged into a single file.

Combiners

It is common for the Reduce function to be associative and commutative. That is, the values to be combined can be combined in any order, with the same result. It doesn't matter how we group a list of numbersv1, v2, . . . , vn; the sum will be the same. When the Reduce function is associative and commutative, it is possible to push some of what Reduce does to the Map tasks. For example, instead of the Map tasks in Example 2.1 producing many pairs (w, 1), (w, 1), . . ., we could apply the Reduce function within the Map task, before the output of the Map tasks is subject to grouping and aggregation. These key-value pairs would thus be replaced by one pair with key w and value equal to the sum of all the 1's in all those pairs. That is, the pairs with key w generated by a single Map task would be combined into a pair (w,m), where m is the number of times that w appears among the documents handled by this Map task. Note that it is still necessary to do grouping and aggregation and to pass the result to the Reduce tasks, since there will typically be one key-value pair with key w coming from each of the Map tasks.

Details of Map-Reduce Execution

Let us now consider in more detail how a program using map-reduce is executed. Taking advantage of a library provided by a map-reduce system such as Hadoop, the user program forks a Master controller process and some number of Worker processes at different compute nodes. Normally, a Worker handles either Map tasks or Reduce tasks, but not both. The Master has many responsibilities. One is to create some number of Map tasks and some number of Reduce tasks, these numbers being selected by the user program. These tasks will be assigned to Worker processes by the Master. It is reasonable to create one Map task for every chunk of the input file(s), but we may wish to create fewer Reduce tasks. The reason for limiting the number of Reduce tasks is that it is necessary for each Map task to create an intermediate file for each Reduce task, and if there are too many Reduce tasks the number of intermediate files explodes. A Worker process reports to the Master when it finishes a task, and a new task is scheduled by the

Master for that Worker process. Each Map task is assigned one or more chunks of the input file(s) and executes on it the code written by the user. The Map task creates a file for each Reduce task on the local disk of the Worker that executes the Map task. The Master is informed of the location and sizes of each of these files, and the Reduce task for which each is destined. When a Reduce task is assigned by the master to a Worker process, that task is given all the files that form its input. The Reduce task executes code written by the user and writes its output to a file that is part of the surrounding distributed file system.

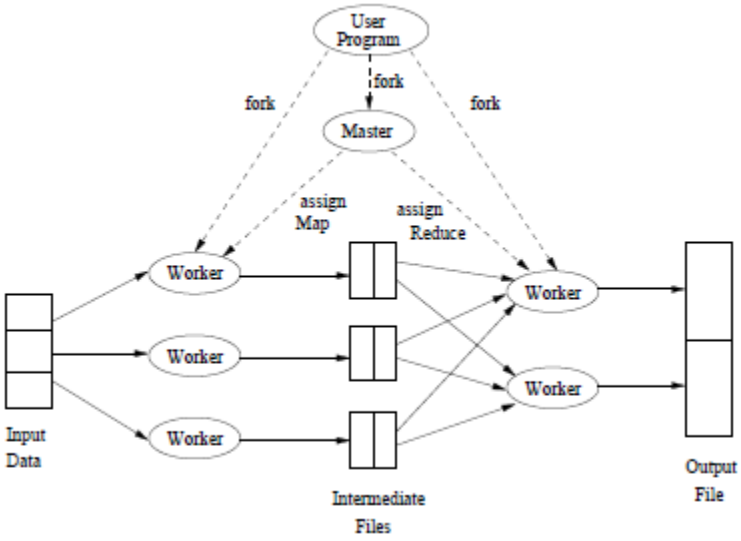

Fig: Overview of the execution of a map-reduce program

Coping With Node Failures

The worst thing that can happen is that the compute node at which the Master is executing fails. In this case, the entire map-reduce job must be restarted. But only this one node can bring the entire process down; other failures will be managed by the Master, and the map-reduce job will complete eventually. Suppose the compute node at which a Map worker resides fails. This failure will be detected by the Master, because it periodically pings the Worker processes. All the Map tasks that were assigned to this Worker will have to be redone, even if they had completed. The reason for redoing completed Map tasks is that their output destined for the Reduce tasks resides at that compute node, and is now unavailable to the Reduce tasks. The Master sets the status of each of these Map tasks to idle and will schedule them on a Worker when one becomes available. The Master must also inform each Reduce task that the location of its input from that Map task has changed. Dealing with a failure at the node of a Reduce worker is simpler. The Master simply sets the status of its currently executing Reduce tasks to idle. These will be rescheduled on another reduce worker later.

2.5 Algorithms Using Map-Reduce

The MapReduce algorithm contains two important tasks, namely Map and Reduce.

- The map task is done by means of Mapper Class
- The reduce task is done by means of Reducer Class.

Mapper class takes the input, tokenizes it, maps and sorts it. The output of Mapper class is used as input by Reducer class, which in turn searches matching pairs and reduces them. MapReduce implements various mathematical algorithms to divide a task into small parts and assign them to multiple systems. In technical terms, MapReduce algorithm helps in sending the Map & Reduce tasks to appropriate servers in a cluster.

Map-reduce is not a solution to every problem, not even every problem that profitably can use many compute nodes operating in parallel. Thus, we would not expect to use either a DFS or an implementation of map-reduce for managing on-line retail sales, even though a large on-line retailer such as Amazon.com uses thousands of compute nodes when processing requests over the Web. The reason is that the principal operations on Amazon data involve responding to searches for products, recording sales, and so on, processes that involve relatively little calculation and that change the database.2 On the other hand, Amazon might use map-reduce to perform certain analytic queries on large amounts of data, such as finding for each user those users whose buying patterns were most similar. The original purpose for which the Google implementation of map-reduce was created was to execute very large matrix-vector multiplications as are needed in the calculation of Page Rank. We shall see that matrix-vector and matrix-matrix calculations fit nicely into the map-reduce style of computing. Another important class of operations that can use map-reduce effectively are the relational-algebra operations. We shall examine the map-reduce execution of these operations as well.

2.6 MATRIX-VECTOR MULTIPLICATION BY MAP-REDUCE

Suppose we have an n×n matrix M, whose element in row i and column j will be denoted mij . Suppose we also have a vector v of length n, whose jth element is vj . Then the matrix-vector product is the vector x of length n, whose ith element xi is given by

$$x_i = \sum_{j=1}^{n} m_{ij} v_j$$

If n = 100, we do not want to use a DFS or map-reduce for this calculation. But this sort of calculation is at the heart of the ranking of Web pages that goes on at search engines, and there, n is in the tens of billions. Let us first assume that n is large, but not so large that vector v cannot fit in main memory, and be part of the input to every Map task. It is useful to observe at this time that there is nothing in the definition of map-reduce that forbids providing the same input to more than one Map task. The matrix M and the vector v each will be stored in a file of the DFS. We assume that the row-column coordinates of each matrix element will be discoverable, either from its position in the file, or because it is stored with explicit coordinates, as a triple (i, j, mij). We also assume the position of element vj in the vector v will be discoverable in the analogous way.

- The Map Function: Each Map task will take the entire vector v and a chunk of the matrix M. From each matrix element mij it produces the key-value pair (i, mijvj). Thus, all terms of the sum that make up the component xi of the matrix-vector product will get the same key.
- The Reduce Function: A Reduce task has simply to sum all the values associated with a given key i. The result will be a pair (i, xi).

2.7 HADOOP YARN:

Apache Hadoop YARN (Yet Another Resource Negotiator) is a cluster management technology. YARN is one of the key features in the second-generation Hadoop 2 version of the Apache Software Foundation's open source distributed processing framework. Originally described by Apache as a redesigned resource manager, YARN is now characterized as a large-scale, distributed operating system for big data applications.

Sometimes called MapReduce 2.0, YARN is a software rewrite that decouples MapReduce's resource management and scheduling capabilities from the data processing component, enabling Hadoop to support more varied processing approaches and a broader array of applications. For example, Hadoop clusters can now run interactive querying and streaming data applications simultaneously with MapReduce batch jobs.

YARN combines a central resource manager that reconciles the way applications use Hadoop system resources with node manager agents that monitor the processing operations of individual cluster nodes. Running on commodity hardware clusters, Hadoop has attracted particular interest as a staging area and data store for large volumes of structured and unstructured data intended for use in analytics applications. Separating HDFS from MapReduce

with YARN makes the Hadoop environment more suitable for operational applications that can't wait for batch jobs to finish.

Apache Hadoop Yarn – Concepts & Applications:

As previously described, YARN is essentially a system for managing distributed applications. It consists of a central **ResourceManager**, which arbitrates all available cluster resources, and a per-node **NodeManager**, which takes direction from the ResourceManager and is responsible for managing resources available on a single node.

Resource Manager

In YARN, the ResourceManager is, primarily, a pure scheduler. In essence, it's strictly limited to arbitrating available resources in the system among the competing applications – a market maker if you will. It optimizes for cluster utilization against various constraints such as capacity guarantees, fairness, and SLAs. To allow for different policy constraints the ResourceManager has a pluggable scheduler that allows for different algorithms such as capacity and fair scheduling to be used as necessary.

ApplicationMaster

The ApplicationMaster is, in effect, an instance of a framework-specific library and is responsible for negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the containers and their resource consumption. It has the responsibility of negotiating appropriate resource containers from the ResourceManager, tracking their status and monitoring progress.

The ApplicationMaster allows YARN to exhibit the following key characteristics:

- **Scale:** The Application Master provides much of the functionality of the traditional ResourceManager so that the entire system can scale more dramatically. In tests, we've already successfully simulated 10,000 node clusters composed of modern hardware without significant issue. This is one of the key reasons that we have chosen to design the ResourceManager as a pure scheduler i.e. it doesn't attempt to provide fault-tolerance for resources. We shifted that to

become a primary responsibility of the ApplicationMaster instance. Furthermore, since there is an instance of an ApplicationMaster per application, the ApplicationMaster itself isn't a common bottleneck in the cluster.

- **Open:** Moving all application framework specific code into the ApplicationMaster generalizes the system so that we can now support multiple frameworks such as MapReduce, MPI and Graph Processing.

Resource Model

YARN supports a very general resource model for applications. An application can request resources with highly specific requirements such as:

- Resource-name (hostname, rackname – we are in the process of generalizing this further to support more complex network topologies with YARN-18).
- Memory (in MB)
- CPU (cores, for now)
- In future, expect us to add more resource-types such as disk/network I/O, GPUs etc.

Hadoop YARN architecture:

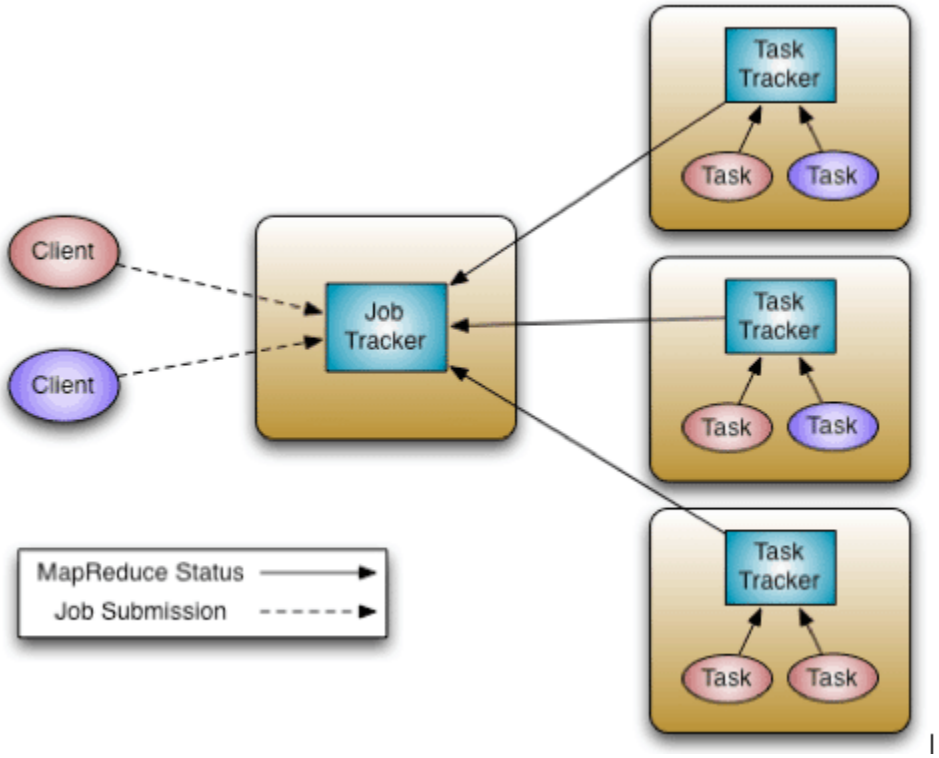

Fig: Hadoop YARN Architecture

- The JobTracker is responsible for resource management (managing the worker nodes i.e. TaskTrackers), tracking resource consumption/availability and also job life-cycle management (scheduling individual tasks of the job, tracking progress, providing fault-tolerance for tasks etc).

- The TaskTracker has simple responsibilities – launch/teardown tasks on orders from the JobTracker and provide task-status information to the JobTracker periodically.

How Yarn Works

YARN's original purpose was to split up the two major responsibilities of the JobTracker/TaskTracker into separate entities:

- a global ResourceManager
- a per-application ApplicationMaster
- a per-node slave NodeManager
- a per-application Container running on a NodeManager

The ResourceManager and the NodeManager formed the new generic system for managing applications in a distributed manner. The ResourceManager is the ultimate authority that arbitrates resources among all applications in the system. The ApplicationMaster is a framework-specific entity that negotiates resources from the ResourceManager and works with the NodeManager(s) to execute and monitor the component tasks.

The ResourceManager has a scheduler, which is responsible for allocating resources to the various applications running in the cluster, according to constraints such as queue capacities and user limits. The scheduler schedules based on the resource requirements of each application.

Each ApplicationMaster has responsibility for negotiating appropriate resource containers from the scheduler, tracking their status, and monitoring their progress. From the system perspective, the ApplicationMaster runs as a normal container.

The NodeManager is the per-machine slave, which is responsible for launching the applications' containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager.