SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE MATERIAL

Subject Name : DATA ANALYTICS **UNIT V** **Subject Code : SIT1303**

5.1 Distributed File System
5.2 Physical Organization of Computer nodes
5.3 Large Scale File System Organization
5.4 Map Reduce
5.5 The Map Task
5.6 Grouping and Aggregation
5.7 The reduce task
5.8 Combiners
5.9 Details of Map Reduce Execution
5.10 Coping with node failures

5.1 Distributed File System

Modern Internet applications have created a need to manage immense amounts of data quickly. In many of these applications, the data is extremely regular, and there is ample opportunity to exploit parallelism. Important examples are:

1. The ranking of Web pages by importance, which involves an iterated matrix- vector multiplication where the dimension is in the tens of billions, and

2. Searches in "friends" networks at social-networking sites, which involve graphs with hundreds of millions of nodes and many billions of edges.

To deal with applications such as these, a new software stack has developed. It begins with a new form of file system, which features much larger units than the disk blocks in a conventional operating system and also provides replication of data to protect against the frequent media failures that occur when data is distributed over thousands of disks.

On top of these file systems, we find higher-level programming systems developing. Central to many of these is a programming system called map-reduce. Implementations of map-reduce enable many of the most common calculations on large-scale data to be performed on large collections of computers, efficiently and in a way that is tolerant of hardware failures during the computation.

Map-reduce systems are evolving and extending rapidly. We include in this chapter a discussion of generalizations of map-reduce, first to acyclic workflows and then to recursive algorithms. We conclude with a discussion of commu- nication cost and what it tells us about the most efficient algorithms in this modern computing environment.

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 pro- gramming 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. In this section, we discuss both the characteristics of these computing installations and the specialized file systems that have been developed to take advantage of them.

5.2 Physical Organization of Computer nodes

The new parallel-computing architecture, sometimes called cluster computing, is organized as follows. Compute nodes are stored on racks, perhaps 8–64 on 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 intrarack Ethernet, but given the number of pairs of nodes that might need to communicate between racks, this bandwidth may be essential. Figure 2.1 suggests the architecture of a large- scale computing system. 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. For systems such as Fig. 2.1, the principal failure modes are the loss of a single node (e.g., the disk at that node crashes) and the loss of an entire rack (e.g., the network connecting its nodes to each other and to the outside world fails).

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

Switch

5.3 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 has 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. For example, an airline reservation system would not be suitable for a DFS, even if the data were very large, because the data is changed so frequently.

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

DFS Implementations

There are several distributed file systems of the type we have described that are used in practice. Among these:

1. The Google File System (GFS), the original of the class.

2. Hadoop Distributed File System (HDFS), an open-source DFS used with Hadoop, an implementation of map-reduce and distributed by the Apache Software Foundation.

3. CloudStore, an open-source DFS originally developed by Kosmix.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.

5.4 The Map Reduce

Map-reduce is a style of computing that has been implemented several times. You can 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, and also deals with the possibility that one of these tasks will fail to execute. 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 con- troller 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 val- ues associated with that key in some way. The manner of combinationof values is determined by the code written by the user for the Reduce function.

Figure : Schematic of a map-reduce computation

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.

5.5 Map Task

The Map task reads a document and breaks it into its sequence of words $w_1, w_2, \ldots, w_n$. It then emits a sequence of key-value pairs where the value is always 1. That is, the output of the Map task for this document is the sequence of key-value pairs:

$$(w_1, 1), (w_2, 1), \ldots, (w_n, 1)$$

Note that a single Map task will typically process many documents – all the documents in one or more chunks. Thus, its output will be more than the sequence for the one document suggested above. Note also that if a word w appears m times among all the documents assigned to that process, then there will be m key-value pairs (w, 1) among its output. An option, which we discuss in Section 2.2.4, is to combine these m pairs into a single pair (w, m), but we can only do that because, as we shall see, the Reduce tasks apply an associative and commutative operation, addition, to the values. ✷

5.6 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 tasks there 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.[1]

After 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 $(k, [v_1, v_2, \ldots, v_n])$, where (k, v_1), (k, v_2), . . . , (k, v_n) are all the key-value pairs with key k coming from all the Map tasks.

5.7 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.

Example 2.2 : Let us continue with the word-count example of Example 2.1. The Reduce function simply adds up all the values. Thus, the output of the

Reduce tasks is a sequence of (w, m) pairs, where w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents.

Implementations of Map-Reduce

The original implementation of map-reduce was as an internal and propri- etary system at Google. It was called simply "Map-Reduce." There is an open-source implementation called Hadoop. It can be downloaded, along with the HDFS distributed file system, from the Apache Foundation.

5.8 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. The addition performed in Example 2.2 is an example of an associative and commutative operation. It doesn't matter how we group a list of numbers $v_1, v_2, \ldots$

v_n; 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.

5.9 Details of Map-Reduce Execution

Let us now consider in more detail how a program using map-reduce is executed. Figure 2.3 offers an outline of how processes, tasks, and files interact. 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 (a Map worker) or Reduce tasks (a Reduce worker), 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

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

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.

The Master keeps track of the status of each Map and Reduce task (idle, executing at a particular Worker, or completed). 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.

5.10 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 fail- ure 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.