MapReduce Execution Framework

One of the most important idea behind MapReduce is separating the what of distributed processing from the how.

A MapReduce program, referred to as a job, consists of code for mappers and reducers (as well as combiners and partitioners to be discussed in the next section) packaged together with configuration parameters (such as where the input lies and where the output should be stored).

The developer submits the job to the submission node of a cluster (in Hadoop, this is called the jobtracker) and execution framework (sometimes called the "runtime") takes care of everything else: it transparently handles all other aspects of distributed code execution, on clusters ranging from a single node to a few thousand nodes.

Each MapReduce job is divided into smaller units called tasks. For example, a map task may be responsible for processing a certain block of input key-value pairs (called an input split in Hadoop); similarly, a reduce task may handle a portion of the intermediate key space. It is not uncommon for MapReduce jobs to have thousands of individual tasks that need to be assigned to nodes in the cluster. In large jobs, the total number of tasks may exceed the number of tasks that can be run on the cluster concurrently, making it necessary for the scheduler to maintain some sort of a task queue and to track the progress of running tasks so that waiting tasks can be assigned to nodes as they become available. Another aspect of scheduling involves coordination among tasks belonging to different jobs (e.g., from different users).

Data/code co-location. In order for computation to occur, we need to somehow feed data to the code. In MapReduce, this issue is inexplicably intertwined with scheduling and relies heavily on the design of the underlying distributed file system. To achieve data locality, the scheduler starts tasks on the node that holds a particular block of data (i.e., on its local drive) needed by the task. This has the effect of moving code to the data. If this is not possible (e.g., a node is already running too many tasks), new tasks will be started elsewhere, and the necessary data will be streamed over the network. An important optimization here is to prefer nodes that are on the same rack in the datacenter as the node holding the relevant data block, since inter-rack bandwidth is significantly less than intra-rack bandwidth.

Synchronization. In general, synchronization refers to the mechanisms by which multiple concurrently running processes "join up", for example, to share intermediate results or otherwise exchange state information. In MapReduce, synchronization is accomplished by a barrier between the map and reduce phases of processing. Intermediate key-value pairs must be grouped by key, which is accomplished by a large distributed sort involving all the nodes that executed map tasks and all the nodes that will execute reduce tasks. This necessarily involves copying intermediate data over the network, and therefore the process is commonly known as "shuffle and sort". A MapReduce job with m mappers and r reducers involves up to m * r distinct copy operations, since each mapper may have intermediate output going to every reducer.

Error and fault handling. The MapReduce execution f/w must accomplish all these tasks in an environment where errors and faults are the norm, not the exception.

Since MapReduce was explicitly designed around low-end commodity servers, the runtime must be especially resilient. In large clusters, disk failures are common and RAM experiences more errors than one might expect. Datacenters suffer from both planned outages (e.g., system maintenance and hardware upgrades) and unexpected outages (e.g., power failure, connectivity loss, etc.).

MapReduce Advantages

MapReduce provides an abstraction that hides many system-level details from the programmer. Therefore, a developer can focus on what computations need to be performed, as opposed to how those computations are actually carried out or how to get the data to the processes that depend on them.

MapReduce provides a means to distribute computation without burdening the programmer with the details of distributed computing.

Large-data processing by definition requires bringing data and code together for computation to occur. MapReduce addresses this challenge by providing a simple abstraction for the developer, transparently handling most of the details behind the scenes in a scalable, robust, and efficient manner.

Instead of moving large amounts of data around, it is far more efficient, if possible, to move the code to the data. This is operationally realized by spreading data across the local disks of nodes in a cluster and running processes on nodes that hold the data. The complex task of managing storage in such a processing environment is typically handled by a distributed file system that sits underneath MapReduce.

Gracefully handling partial failures: MapReduce spares the programmer from having to think about failure, since the implementation detects failed map or reduce tasks and reschedules with suitable replacements. MapReduce is able to do this since it’s a shared-nothing architecture, meaning that tasks have no dependence on the other.

Word count algorithm

```
class Mapper
    method Map(lineNum a; line l)
        for all term t in line l do
            Emit(term t; count 1)
class Reducer
    method Reduce(term t; counts [c1; c2; …])
        sum  = 0
        for all count c in counts [c1; c2; …] do
            sum =  sum + c
        Emit(term t; count sum)
```

This algorithm counts the number of occurrences of every word in a text collection, which may be the first step in, for example, building a unigram language model (i.e., probability distribution over words in a collection).

Input key-values pairs take the form of (lineNum, line) pairs stored on the distributed file system, where the former is a unique identifier for the line, and the latter is the text of the line itself.

The mapper takes an input key-value pair, tokenizes the document, and emits an intermediate key-value pair for every word: the word itself serves as the key, and the integer one serves as the value (denoting that we've seen the word once).

The MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer. Therefore, in our word count algorithm, we simply need to sum up all counts (ones) associated with each word.

The reducer does exactly this and emits final key-value pairs with the word as the key, and the count as the value. Final output is written to the distributed file system, one file per reducer.

Words within each file will be sorted by alphabetical order, and each file will contain roughly the same number of words.

The partitioner controls the assignment of words to reducers.

The output can be examined by the programmer or used as input to another MapReduce program.

Mappers and reducers are objects that implement the Map and Reduce methods, respectively. In Hadoop, a mapper object is initialized for each map task (associated with a particular sequence of key-value pairs called an input split) and the Map method is called on each key-value pair by the execution framework. In a MapReduce job, the programmer provides a hint on the number of map tasks to run, but the execution framework makes the final determination based on the physical layout of the data. The situation is similar for the reduce phase: a reducer object is initialized for each reduce task, and the Reduce method is called once per intermediate key. In contrast with the number of map tasks, the programmer can precisely specify the number of reduce tasks.

Restrictions on Mappers and Reducers

Mappers and reducers can express arbitrary computations over their inputs. However, one must generally be careful about use of external resources since multiple mappers or reducers may be contending for those resources. For example, it may be unwise for a mapper to query an external SQL database.

Mappers can emit an arbitrary number of intermediate key-value pairs, and they need not be of the same type as the input key-value pairs. Similarly, reducers can emit an arbitrary number of final key-value pairs, and they can differ in type from the intermediate key-value pairs.

Side effects

Mappers and reducers can have side effects. It may be useful for mappers or reducers to have external side effects, such as writing files to the distributed file system. Since many mappers and reducers are run in parallel, and the distributed file system is a shared global resource, special care must be taken to ensure that such operations avoid synchronization conflicts. One strategy is to write a temporary file that is renamed upon successful completion of the mapper or reducer.

Shared Data in MapReduce?

- MapReduce programs are designed to compute large volumes of data in a parallel fashion. This requires dividing the workload across a large number of machines.
- This model would not scale to large clusters (hundreds or thousands of nodes) if the components were allowed to share data arbitrarily. The communication overhead required to keep the data on the nodes synchronized at all times would prevent the system from performing reliably or efficiently at large scale.
- Instead, all data elements in MapReduce are immutable, meaning that they cannot be updated. If in a mapping task you change an input (key, value) pair, it does not get reflected back in the input files; communication occurs only by generating new output (key, value) pairs which are then forwarded by the Hadoop system into the next phase of execution.

So MapReduce cannot process any program which has globally shared data. So keys or shared data cannot be used in MapReduce.

Hadoop vs. Google

In Hadoop, the reducer is presented with a key and an iterator over all values associated with the particular key. The values are arbitrarily ordered.

Google's implementation allows the programmer to specify a secondary sort key for ordering the values (if desired) in which case values associated with each key would be presented to the developer's reduce code in sorted order.

In Google's implementation the programmer is not allowed to change the key in the reducer. That is, the reducer output key must be exactly the same as the reducer input key.

In Hadoop, there is no such restriction, and the reducer can emit an arbitrary number of output key-value pairs (with different keys).

Anatomy of a MR job

- http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html (http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html)

MapReduce is the most successful abstraction over large-scale computational resources we have seen to date. However, as anyone who has taken an introductory computer science course knows, abstractions manage complexity by hiding details and presenting well-defined behaviors to users of those abstractions. They, inevitably, are imperfect - making certain tasks easier but others more difficult, and sometimes, impossible (in the case where the detail suppressed by the abstraction is exactly what the user cares about). This critique applies to MapReduce: it makes certain large-data problems easier, but suffers from limitations as well. This means that MapReduce is not the final word, but rather the first in a new class of programming models that will allow us to more effectively organize computations at a massive scale.