Lesson 2

MapReduce Basics

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.

Execution Framework: Responsibilities

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).

Execution Framework: Responsibilities

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.

Execution Framework: Responsibilities

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 $m \times r$ distinct copy operations, since each mapper may have intermediate output going to every reducer.

Execution Framework: Responsibilities

Error and fault handling. The MapReduce execution framework must accomplish all the tasks above 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.).

PARTITIONERS AND COMBINERS

- Partitioners are responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers. In other words, the partitioner specifies the task to which an intermediate key-value pair must be copied. Within each reducer, keys are processed in sorted order (which is how the “group by" is implemented). The simplest partitioner involves computing the hash value of the key and then taking the mod of that value with the number of reducers. This assigns approximately the same number of keys to each reducer.

PARTITIONERS AND COMBINERS

The partitioner only considers the key and ignores the value therefore, a roughly-even partitioning of the key space may nevertheless yield large differences in the number of key-values pairs sent to each reducer (since different keys may have different numbers of associated values). This imbalance in the amount of data associated with each key is relatively common in many text processing applications due to the Zipfian distribution of word occurrences.

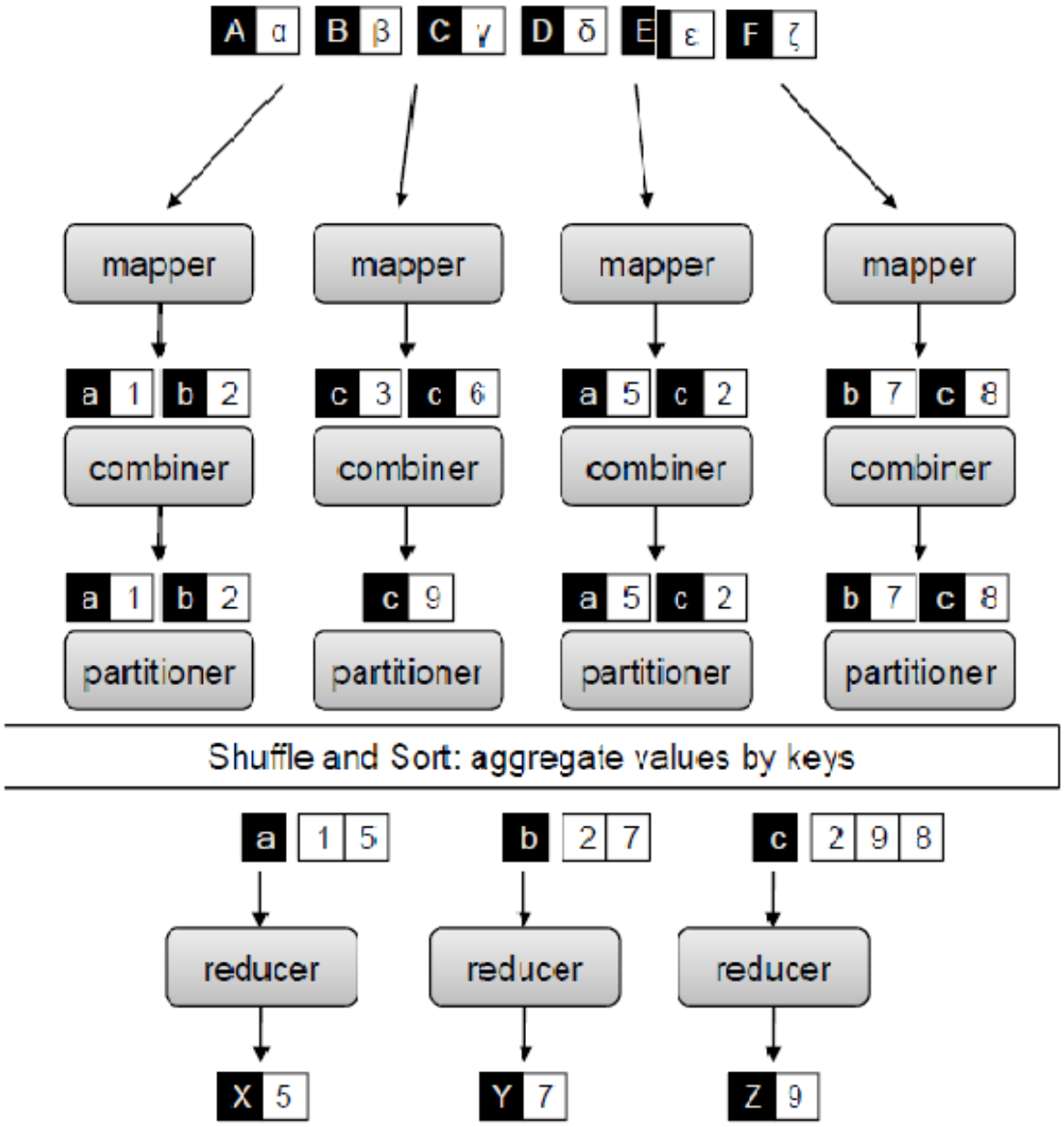
Combiners

Combiners are an optimization in MapReduce that allow for local aggregation before the shuffle and sort phase. We can motivate the need for combiners by considering the word count algorithm, which emits a key-value pair for each word in the collection. These key-value pairs need to be copied across the network, and so the amount of intermediate data will be larger than the input collection itself. This is clearly inefficient. One solution is to perform local aggregation on the output of each mapper, i.e., to compute a local count for a word over all the documents processed by the mapper. With this modification, the number of intermediate key-value pairs will be at most the number of unique words in the collection times the number of mappers (and typically far smaller because each mapper may not encounter every word).

Combiners

One can think of combiners as “mini-reducers" that take place on the output of the mappers, prior to the shuffle and sort phase. Each combiner operates in isolation and therefore does not have access to intermediate output from other mappers. The combiner is provided keys and values associated with each key (the same types as the mapper output keys and values).

Critically, one cannot assume that a combiner will have the opportunity to process all values associated with the same key. The combiner can emit any number of key-value pairs, but the keys and values must be of the same type as the mapper output (same as the reducer input). In cases where an operation is both associative and commutative (e.g., addition or multiplication), reducers can directly serve as combiners. In general, reducers and combiners are not interchangeable.


A α
B β
C γ
D δ
E ε
F ζ
mapper
mapper
mapper
mapper
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combiner
combiner
combiner
combiner
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partitioner
partitioner
partitioner
partitioner
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8
reducer
reducer
reducer
X 5
Y 7
Z 9

Job

A complete MapReduce job consists of code for the mapper, reducer, combiner, and partitioner, along with job conguration parameters. The execution framework handles everything else.

Distributed File System (DFS)

The main idea is to divide user data into blocks and replicate those blocks across the local disks of nodes in the cluster.

Blocks are significantly larger than block sizes in typical single-machine file systems (64MB by default).

The distributed file system adopts a master-slave architecture in which the master maintains the file namespace (metadata, directory structure, file to block mapping, location of blocks, and access permissions) and the slaves manage the actual data blocks. In GFS, the master is called the GFS master, and the slaves are called GFS chunkservers. In Hadoop, the same roles are filled by the namenode and datanodes, respectively.

Distributed File System (DFS)

- In HDFS, an application client wishing to read a file must first contact the namenode to determine where the actual data is stored.
- The namenode returns the relevant block id and the location where the block is held (i.e., which datanode).
- The client contacts the datanode to retrieve the data. Blocks are themselves stored on standard single-machine file systems, so HDFS lies on top of the standard OS stack (e.g., Linux).
- An important feature of the design is that data is never moved through the namenode. All data transfer occurs directly between clients and datanodes; communications with the namenode only involves transfer of metadata.

Distributed File System (DFS)

- By default, HDFS stores three separate copies of each data block to ensure both reliability, availability, and performance. In large clusters, the three replicas are spread across two different physical racks, so HDFS is resilient towards two common failure scenarios: individual datanode crashes and failures in networking equipment that bring an entire rack offline. Replicating blocks across physical machines also increases opportunities to co-locate data and processing in the scheduling of MapReduce jobs, since multiple copies yield more opportunities to exploit locality.

Distributed File System (DFS)

The namenode is in periodic communication with the datanodes to ensure proper replication of all the blocks: if there aren't enough replicas (e.g., due to disk or machine failures or to connectivity losses due to networking equipment failures), the namenode directs the creation of additional copies; if there are too many replicas (e.g., a repaired node rejoins the cluster), extra copies are discarded.

Distributed File System (DFS)

- To create a new file and write data to HDFS, the application client first contacts the namenode, which updates the file namespace after checking permissions and making sure the file doesn't already exist.
- The namenode allocates a new block on a suitable datanode, and the application is directed to stream data directly to it.
- From the initial datanode, data is further propagated to additional replicas.
- In the most recent release of Hadoop (release 0.20.2), files are immutable, they cannot be modified after creation. There are current plans to officially support file appends in the near future, which is a feature already present in GFS.

HDFS namenode responsibilities

Namespace management. The namenode is responsible for maintaining the file namespace, which includes metadata, directory structure, file to block mapping, location of blocks, and access permissions. These data are held in memory for fast access and all mutations are persistently logged.

Coordinating file operations. The namenode directs application clients to datanodes for read operations, and allocates blocks on suitable datanodes for write operations. All data transfers occur directly between clients and datanodes. When a file is deleted, HDFS does not immediately reclaim the available physical storage; rather, blocks are lazily garbage collected.

HDFS namenode responsibilities

Maintaining overall health of the file system. The namenode is in periodic contact with the datanodes via heartbeat messages to ensure the integrity of the system. If the namenode observes that a data block is under-replicated (fewer copies are stored on datanodes than the desired replication factor), it will direct the creation of new replicas. Finally, the namenode is also responsible for rebalancing the file system. During the course of normal operations, certain datanodes may end up

holding more blocks than others; rebalancing involves moving blocks from datanodes with more blocks to datanodes with fewer blocks. This leads to better load balancing and better disk utilization.

HDFS (or GFS) environment

Stores a relatively modest number of large files. In HDFS multi-gigabyte files are common (and even encouraged). There are several reasons why lots of small files are to be avoided. Since the namenode must hold all file metadata in memory, this presents an upper bound on both the number of files and blocks that can be supported. Large multi-block files represent a more efficient use of namenode memory than many single-block files (each of which consumes less space than a single block size). In addition, mappers in a MapReduce job use individual files as a basic unit for splitting input data.

HDFS (or GFS) environment

At present, there is no default mechanism in Hadoop that allows a mapper to process multiple files. As a result, mapping over many small files will yield as many map tasks as there are files. This results in two potential problems: first, the startup costs of mappers may become significant compared to the time spent actually processing input key-value pairs; second, this may result in an excessive amount of across-the-network copy operations during the “shuffle and sort" phase (recall that a MapReduce job with m mappers and r reducers involves up to $m \times r$ distinct copy operations).

HDFS (or GFS) environment

Workloads are batch oriented, dominated by long streaming reads and large sequential writes. As a result, high sustained bandwidth is more important than low latency. This exactly describes the nature of MapReduce jobs, which are batch operations on large amounts of data. Due to the common-case workload, both HDFS and GFS do not implement any form of data caching.

HDFS (or GFS) environment

Applications are aware of the characteristics of the distributed file system. Neither HDFS nor GFS present a general POSIX-compliant API, but rather support only a subset of possible file operations. This simplifies the design of the distributed file system, and in essence pushes part of the data management onto the end application. One rationale for this decision is that each application knows best how to handle data specific to that application, for example, in terms of resolving inconsistent states and optimizing the layout of data structures.

HDFS (or GFS) environment

The file system is deployed in an environment of cooperative users. There is no discussion of security in the original GFS paper, but HDFS explicitly assumes a datacenter environment where only authorized users have access. File permissions in HDFS are only meant to prevent unintended operations and can be easily circumvented.

The system is built from unreliable but inexpensive commodity components. As a result, failures are the norm rather than the exception. HDFS is designed around a number of self-monitoring and self-healing mechanisms to robustly cope with common failure modes.

HDFS (or GFS) environment

The single-master design of GFS and HDFS is a well-known weakness, since if the master goes offline, the entire file system and all MapReduce jobs running on top of it will grind to a halt. This weakness is mitigated in part by the lightweight nature of file system operations. Recall that no data is ever moved through the namenode and that all communication between clients and datanodes involve only metadata. Because of this, the namenode rarely is the bottleneck, and for the most part avoids load induced crashes.

HDFS (or GFS) environment

In practice, this single point of failure is not as severe a limitation as it may appear with diligent monitoring of the namenode, mean time between failure measured in months are not uncommon for production deployments. Furthermore, the Hadoop community is well-aware of this problem and has developed several reasonable workarounds for example, a warm standby namenode that can be quickly switched over when the primary namenode fails.

HADOOP CLUSTER ARCHITECTURE

The HDFS **namenode** runs the namenode daemon.

The job submission node runs the **jobtracker**, which is the single point of contact for a client wishing to execute a MapReduce job. The job tracker monitors the progress of running MapReduce jobs and is responsible for coordinating the execution of the mappers and reducers. Typically, these services run on two separate machines, although in smaller clusters they are often co-located.

The bulk of a Hadoop cluster consists of slave nodes that run both a **tasktracker**, which is responsible for actually running user code, and a **datanode** daemon, for serving HDFS data.

HADOOP CLUSTER ARCHITECTURE

A Hadoop MapReduce job is divided up into a number of map tasks and reduce tasks. Tasktrackers periodically send heartbeat messages to the jobtracker that also doubles as a vehicle for task allocation. If a tasktracker is available to run tasks (in Hadoop parlance, has empty task slots), the return acknowledgment of the tasktracker heartbeat contains task allocation information. The number of reduce tasks is equal to the number of reducers specified by the programmer. The number of map tasks, on the other hand, depends on many factors: the number of mappers specified by the programmer serves as a hint to the execution framework, but the actual number of tasks depends on both the number of input files and the number of HDFS data blocks occupied by those files. Each map task is assigned a sequence of input key-value pairs, called an **input split** in Hadoop.

HADOOP CLUSTER ARCHITECTURE

Input splits are computed automatically and the execution framework strives to align them to HDFS block boundaries so that each map task is associated with a single data block. In scheduling map tasks, the jobtracker tries to take advantage of data locality, if possible, map tasks are scheduled on the slave node that holds the input split, so that the mapper will be processing local data. The alignment of input splits with HDFS block boundaries simplifies task scheduling. If it is not possible to run a map task on local data, it becomes necessary to stream input key-value pairs across the network. Since large clusters are organized into racks, with far greater intra-rack bandwidth than inter-rack bandwidth, the execution framework strives to at least place map tasks on a rack which has a copy of the data block.

HADOOP CLUSTER ARCHITECTURE

In Hadoop, mappers are Java objects with a Map method (among others). A mapper object is instantiated for every map task by the tasktracker. The life-cycle of this object begins with instantiation, where a hook is provided in the API to run programmer-specified code.

HADOOP CLUSTER ARCHITECTURE

This means that mappers can read in “side data", providing an opportunity to load state, static data sources, dictionaries, etc. After initialization, the Map method is called (by the execution framework) on all key-value pairs in the input split. Since these method calls occur in the context of the same Java object, it is possible to preserve state across multiple input key-value pairs within the same map task. This is an important property to exploit in the design of MapReduce algorithms, as we will see in the next chapter. After all key-value pairs in the input split have been processed, the mapper object provides an opportunity to run programmer-specified termination code. This, too, will be important in the design of MapReduce algorithms.

HADOOP CLUSTER ARCHITECTURE

The actual execution of reducers is similar to that of the mappers. Each reducer object is instantiated for every reduce task. The Hadoop API provides hooks for programmer-specified initialization and termination code. After initialization, for each intermediate key in the partition (defined by the partitioner), the execution framework repeatedly calls the Reduce method with an intermediate key and an iterator over all values associated with that key. The programming model also guarantees that intermediate keys will be presented to the Reduce method in sorted order. Since this occurs in the context of a single object, it is possible to preserve state across multiple intermediate keys (and associated values) within a single reduce task.