**Reading Assignment – Big Data**

**Aparna PAvithran (axp161730)**

**2.1 Functional Programming Roots**

**a. What is a higher order function? How is it used in MapReduce? Give your own example.**

A higher order function is a function that can accept other functions as arguments. In Mapreduce, two built in higher order functions commonly used are map and fold. Map takes a function f as an argument and function applies to every element in a list. Fold takes function g as an argument and iteratively applies function g to aggregate results. Function g is first applied to initial value and first item in the list and result is stored in an intermediate variable. Function g is iteratively applies to this intermediate variable and next item in the list and this repeats until all items in the list are done. Fold will return the final result of the intermediate variable.

For example, compute multiplication of squares of odd integers of list and cube of even integers of list. Map function will check the input list and compute square or cube of integer based on even or odd (x: if x odd : x^2 else x^3). Fold will do the multiplication function (x, y : x \* y) of the resulting list by taking initial value of one.

**2.2 Mappers and Reducers understand the purpose of mappers and reducers. (No need to write anything)**

**a. What are the restrictions on mappers and reducers? Can they have side effects? Explain.**

**Restrictions**: - While using external resources in mappers and reducers, must be very careful because multiple mappers or reducers may be struggling for those resources and this can in turn create bottleneck.

Mappers and reducers can have **side effects**. Preserving state across multiple inputs is central to the design of many MapReduce algorithms. Such algorithms can have internal side effects which change state that is internal to the mapper or reducer. Correctness of these algorithms is difficult to guarantee because functions output depends on current and previous inputs. MapReduce can have external side effects as well. In MapReduce many mappers and reducers are run in parallel and DFS is shared to all as a global resource. For example, while writing a file to distributed file system, special care should be taken to avoid synchronization conflicts. To avoid this we can write a temporary file that can be renamed after successful completion of mapper or reducer.

**2.3 The Execution Framework**

**a. Write brief summary of the various responsibilities of the execution framework**

**Scheduling**: - 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 and a reduce task may handle a portion of the intermediate key space. If the job is large enough then there will be thousands of individual tasks that needs to be assigned to nodes in the cluster. In that case total number of tasks that can run on the cluster concurrently w might exceed. Now scheduler have to maintain a task queue to track progress of running tasks so that waiting tasks can be assigned to nodes when they are available. Also coordination among tasks belong to different jobs are necessary and is done by scheduling.

**Data/code co-location: -** Key idea behind MapReduce is to move code but not the data. But then for computation to occur we need to give data to code. In MapReduce scheduling and design of DFS achieve this. Scheduler starts tasks on the node that holds a particular block of data that needed by the particular task. If the node is already overloaded with many tasks and not able to move to that node, then necessary data will be streamed over the network. It prefers nodes that are on same rack in the datacenter as the node holding the relevant data block. This is good because inter rack bandwidth is less than intra rack bandwidth.

**Synchronization:-** In MapReduce, there is a barrier between map and reduce phases of processing to achieve synchronization. Intermediate key value pairs must be grouped by key. It is accomplished a large distributed sort. It is known as shuffle and sort. It copies intermediate data over the network. If a MapReduce job has m mappers and r reducers then it involces up to mXr distinct copy operations. This is because each mapper may have intermediate outpur going to every reducer.

**Error and fault handling: -** Since MapReduce was designed in low-end commodity servers; its runtime must be resilient. Disk failures are very common and RAM experiences more errors than expected. So data centers suffer from both expected and unexpected outages. From software point of view, its never bug free. Hence excpetions must be appropriately trapped, logged and recovered from. Any large dataset will contain corrupted data, which has to be cleaned before transformation.

**2.4 Partitioners and Combiners**

**a. Write a brief summary of the roles of partitioners and combiners. Where are they executed and what are their input and output?**

**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. 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. Partitioner is executed before the reducer and after the combiner.

**Combiners** are an optimization in MapReduce that allow for local aggregation before the shuffle and sort phase. Combiners are nothing but mini-reducers 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 combiner can emit any number of key-value pairs, but the keys and values must be of the same type as the mapper output. In cases where an operation is both associative and commutative (e.g., addition or multiplication), reducers can directly serve as combiners. Combiners are executed after mapper phase and before partitioner phase.

**2.5 The Distributed File System**

**a. What are the responsibilities of the NameNode?**

HDFS namenode has the following **responsibilities**:

**1. 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.

**2. 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.

**3. 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, 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 more even disk utilization.

**b.What assumptions about the operational environment of HDFS were made by the designers?**

* The file system stores a relatively modest number of large files. The definition of modest varies by the size of the deployment, but in HDFS multi-gigabyte files are common. 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. In addition, mappers in a MapReduce job use individual files as a basic unit for splitting input data. 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.
* 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.
* 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.
* 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.

**c. What are the advantages and disadvantages of the single-master design in HDFS?**

Finally, some discussion is necessary to understand the single-master design of HDFS

and GFS. It has been demonstrated that in large-scale distributed systems, simultaneously

providing consistency, availability, and partition tolerance is impossible|this is

Brewer's so-called CAP Theorem [58]. Since partitioning is unavoidable in large-data

systems, the real tradeo\_ is between consistency and availability. A single-master design

trades availability for consistency and signi\_cantly simpli\_es implementation. If the

master (HDFS namenode or GFS master) goes down, the entire \_le system becomes

unavailable, which trivially guarantees that the \_le system will never be in an inconsistent

state. An alternative design might involve multiple masters that jointly manage

the \_le namespace|such an architecture would increase availability (if one goes down,

another can step in) at the cost of consistency, not to mention requiring a more complex

implementation (cf. [4, 105]).

The single-master design of GFS and HDFS is a well-known weakness, since if

the master goes o\_ine, the entire \_le 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 \_le 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 loadinduced

crashes. 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. The open source environment and the fact

that many organizations already depend on Hadoop for production systems virtually

guarantees that more e\_ective solutions will be developed over time.

**2.6 Hadoop Cluster Architecture**

**a. Which of the following can a programmer control more precisely: the number of mappers (M) or the number of reducers (R)?**

The number of reduce tasks is equal to the number of reducers specified by the programmer.

**b. On what factors does the number of map tasks (M) depend?**

The number of map tasks 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.

**c. How does the framework try to optimize the following: - Number of input splits - Location of map tasks.**

Each map task is assigned a sequence of input key-value pairs, called an input split in Hadoop. 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 job tracker 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.

**d. What guarantee does the framework make about the order of intermediate keys presented to the reduce method?**

The frameworkguarantees 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.