**Hadoop 4th Edition Notes**

* MapReduce is fundamentally a batch processing system, and is not suitable for interactive analysis.
* YARN is a cluster resource management system, which allows any distributed program (not just MapReduce) to run on data in a Hadoop cluster.
* Iterative processing not supported by Map-Reduce but it supported by spark.
* Streaming systems like Storm, Spark Streaming, or Samza make it possible to run realtime,distributed computations on unbounded streams of data and emit results to Hadoop storage or external systems.
* The Solr search platform can run on a Hadoop cluster, indexing documents as they are added to HDFS, and serving search queries from indexes stored in HDFS.

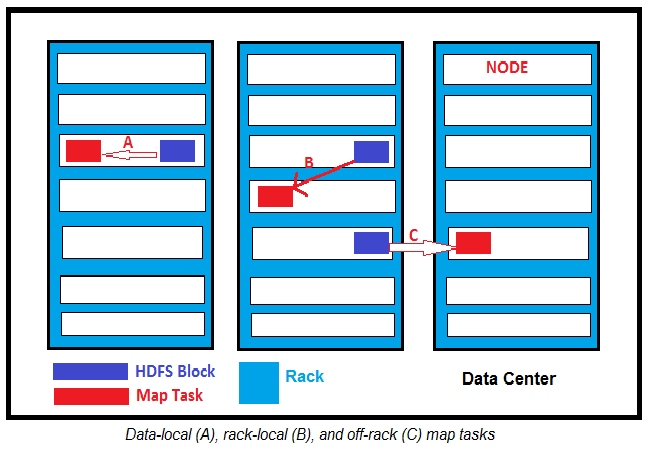
1. Why can’t we use databases with lots of disks to do large-scale analysis? Why is Hadoop needed?

**A1)** Several regions like seek time in disk, Semistructured and unstructured data not supported, suitable for ad-hoc query processing.

* **Data Locality:** Hadoop tries to co-locate the data with the compute nodes, so data access is fast because it is local.This feature, known as *data locality*, is at the heart of data processing in Hadoop and is the reason for its good performance.
* For most jobs, a good split size tends to be the size of an DFS block, which is 128 MB by default, although this can be changed for the cluster (for all newly created files) or specified when each file is created

It should now be clear why the optimal split size is the same as the block size: it is the largest size of input that can be guaranteed to be stored on a single node. If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks, so some of the split would have to be transferred across the network to the node running the map

task, which is clearly less efficient than running the whole map task using local data.

****

* Map tasks write their output to the local disk, not to HDFS. Why is this? Map output is intermediate output: it’s processed by reduce tasks to produce the final output, and once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output.
* Reduce tasks don’t have the advantage of data locality; the input to a single reduce task is normally the output from all mappers. In the present example, we have a single reduce task that is fed by all of the map tasks. Therefore, the sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function. The output of the reduce is normally stored in HDFS for reliability. For each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes for reliability. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline consumes.
* Combiner function is an optimization used by Map-Reduce Job. Before Map output goes to reduce task the combiner runs.Let’s take one example:

Example: Consider there are two splits = 2 Mapers will run.

Mapper1 produce output = {1950,2} (year,temp)

{1950,35}

{1950,20}

Mapper2 pro duce output = {1950,-2}

{1950,30}

If we process without combiner ,each mapper output will go to each reducer and reducer will perform below task to find out maximum temperature as: {1950,(2,35,20,-2,30)} which will result to {1950,35}.

But if I will use combiner then each mapper output will change like this: Mapper1 = {1950,35}

Mapper2 = {1950,30}

Now reduce task will perform maximum operation like {1950,(35,30)} which yields to {1950,35}

* Combiner function helps to cut down the data shuffle between map and reduce tasks. But combiner can’t replace reduce function.

**HADOOP STREAMING**

* **hadoop jar $HADOOP\_HOME/share/hadoop/tools/lib/hadoop-streaming-\*.jar \**

**-files ch02-mr-intro/src/main/ruby/max\_temperature\_map.rb,\**

**ch02-mr-intro/src/main/ruby/max\_temperature\_reduce.rb \**

**-input input/ncdc/all \**

**-output output \**

**-mapper ch02-mr-intro/src/main/ruby/max\_temperature\_map.rb \**

**-combiner ch02-mr-intro/src/main/ruby/max\_temperature\_reduce.rb \**

**-reducer ch02-mr-intro/src/main/ruby/max\_temperature\_reduce.rb**

**Chapter 3. The Hadoop Distributed Filesystem**

* HBase is currently a better choice for low-latency access.

**WHY IS A BLOCK IN HDFS SO LARGE ?**

* HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks. If the block is large enough, the time it takes to transfer the data from the disk can be significantly longer than the time to seek to the start of the block. Thus, transferring a large file made of multiple blocks operates at the disk transfer rate.A quick calculation shows that if the seek time is around 10 ms and the transfer rate is 100 MB/s, to make the seek time 1% of the transfer time, we need to make the block size around 100 MB. The default is actually 128 MB, although many HDFS installations use larger block sizes. This figure will continue to be revised upward as transfer speeds grow with new generations of disk drives.This argument shouldn’t be taken too far, however. Map tasks in MapReduce normally operate on one block at a time,so if you have too few tasks (fewer than nodes in the cluster), your jobs will run slower than they could otherwise.
* Blocks are just chunks of data to be stored, file metadata such as permissions information does not need to be stored with the blocks, so another system can handle metadata separately.
* **hdfs fsck / -files –blocks** will list the blocks that make up each file in the file system
* The namenode manages the file system namespace. It maintains the file system tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located; however, it does not store block locations persistently, because this information is reconstructed from datanodes when the system starts.
* It is important to make the namenode resilient to failure, and Hadoop provides

two mechanisms for this.

* The first way is to back up the files that make up the persistent state of the filesystem metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic. The usual configuration choice is to write to local disk as well as a remote NFS mount.
* It is also possible to run a secondary namenode, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine because it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing. However, the state of the secondary namenode lags that of the primary, so in the event of total failure of the primary, data loss is almost certain. The usual course of action in this case is to copy the namenode’s metadata files that are on NFS to the secondary and run it as the new primary.
* Hadoop 2 remedied this situation by adding support for HDFS high availability (HA). In this implementation, there are a pair of namenodes in an active-standby configuration. Inthe event of the failure of the active namenode, the standby takes over its duties tocontinue servicing client requests without a significant interruption. A few architectural changes are needed to allow this to happen:
* The namenodes must use highly available shared storage to share the edit log. When a standby namenode comes up, it reads up to the end of the shared edit log to synchronize its state with the active namenode, and then continues to read new entriesas they are written by the active namenode.
* Datanodes must send block reports to both namenodes because the block mappings are stored in a namenode’s memory, and not on disk.
* Clients must be configured to handle namenode failover, using a mechanism that is transparent to users.
* The secondary namenode’s role is subsumed by the standby, which takes periodic checkpoints of the active namenode’s namespace.

There are two choices for the highly available shared storage: an NFS filer, or a quorum journal manager (QJM). The QJM is a dedicated HDFS implementation, designed for the sole purpose of providing a highly available edit log, and is the recommended choice for most HDFS installations. The QJM runs as a group of journal nodes, and each edit must be written to a majority of the journal nodes. Typically, there are three journal nodes, so the system can tolerate the loss of one of them. This arrangement is similar to the way ZooKeeper works, although it is important to realize that the QJM implementation does not use ZooKeeper.

Hadoop File System

-------------------

1. Different kind of hadoop file systems are : Locale, HDFS, webHdfs, swebHDFS, HAR, view, s3, Azure, FTP, Swift.
2. The java interface used to communicate different file system is: org.apache.haddop.fs.FileSystem.
3. FSDataInputStream, Prograssable,FSDataOutputStream,FileStatus (Metadata about file or directory),FileUtil, PathFilter (Used in FileStatus to retrieve particular obect),FilePatterns (two methods used In FileSystem i.e. **public** FileStatus[] globStatus(Path pathPattern) **throws** IOException and **public** FileStatus[] globStatus(Path pathPattern, PathFilter filter)**throws** IOException ),

Data Flow:

---------