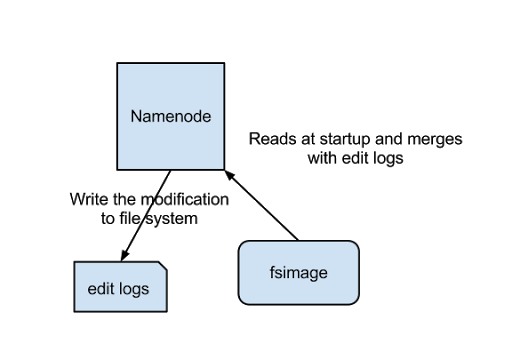
## Secondary Namenode

Namenode holds the meta data for the HDFS like Namespace information, block information etc. When in use, all this information is stored in main memory. But this information also stored in disk for persistence storage.



The above image shows how Name Node stores information in disk.

Two different files are

fsimage - Its the snapshot of the filesystem when namenode started

Edit logs - Its the sequence of changes made to the filesystem after namenode started

Only in the restart of namenode , edit logs are applied to fsimage to get the latest snapshot of the file system. But namenode restart are rare in production clusters which means edit logs can grow very large for the clusters where namenode runs for a long period of time. The following issues we will encounter in this situation.

Editlog become very large , which will be challenging to manage it

Namenode restart takes long time because lot of changes has to be merged

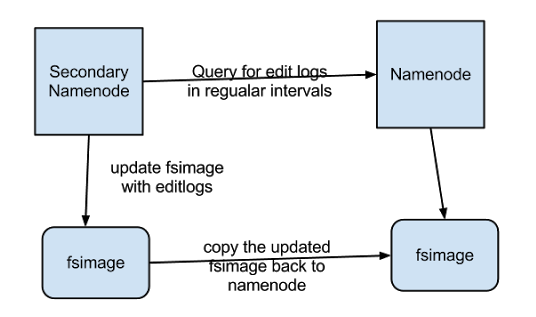
In the case of crash, we will lost huge amount of metadata since fsimage is very old

So to overcome this issues we need a mechanism which will help us reduce the edit log size which is manageable and have up to date fsimage ,so that load on namenode reduces . It’s very similar to Windows Restore point, which will allow us to take snapshot of the OS so that if something goes wrong , we can fallback to the last restore point.

So now we understood NameNode functionality and challenges to keep the meta data up to date.So what is this all have to with Seconadary Namenode?

### Secondary Namenode

Secondary Namenode helps to overcome the above issues by taking over responsibility of merging editlogs with fsimage from the namenode.



The above figure shows the working of Secondary Namenode

It gets the edit logs from the namenode in regular intervals and applies to fsimage

Once it has new fsimage, it copies back to namenode

Namenode will use this fsimage for the next restart,which will reduce the startup time

Secondary Namenode whole purpose is to have a checkpoint in HDFS. Its just a helper node for namenode.That’s why it also known as checkpoint node inside the community.

So we now understood all Secondary Namenode does puts a checkpoint in filesystem which will help Namenode to function better. Its not the replacement or backup for the Namenode. So from now on make a habit of calling it as a checkpoint node.

## [Which node sort/shuffle the keys in Hadoop?](http://stackoverflow.com/questions/19674843/which-node-sort-shuffle-the-keys-in-hadoop)

MapReduce makes the guarantee that the input to every reducer is sorted by key. The process by which the system performs the sort—and transfers the map outputs to the reducers as inputs—is known as the shuffle.

Sort And Shuffle Phase is divided among the Mappers and Reducers. That is the reason we seen the Reduce % increasing(Usually till 33%) while the Mapper is still Running.

Increasing the sort buffer memory and the performance gain from that will depend on:

a)The size/total Number of the Keys being emitted by the mapper

b) The Nature of the Mapper Tasks : (IO intensive, CPU intensive)

c) Available Primary Memory, Map/Reduce Slots(occupied) in the given Node

d) Data skewness

## Partitioning

Partitioning is the process of determining which reducer instance will receive which intermediate keys and values. Each mapper must determine for all of its output (key, value) pairs which reducer will receive them. It is necessary that for any key, regardless of which mapper instance generated it, the destination partition is the same Problem: How does hadoop make it? Use a hash function? what is the default function?

The default partitioner in Hadoop is the HashPartitioner which has a method called getPartition. It takes key.hashCode() & Integer.MAX\_VALUE and finds the modulus using the number of reduce tasks.

For example, if there are 10 reduce tasks, getPartition will return values 0 through 9 for all keys.

Here is the code:

public class HashPartitioner<K, V> extends Partitioner<K, V> {

public int getPartition(K key, V value, int numReduceTasks) {

return (key.hashCode() & Integer.MAX\_VALUE) % numReduceTasks;

}

}

To create a custom partitioner, you would extend Partitioner, create a method getPartition, then set your partitioner in the driver code (job.setPartitionerClass(CustomPartitioner.class);). This is particularly helpful if doing secondary sort operations, for example.

## Number of mappers and reducers

The optimal number of mappers and reducers has to do with a lot of things.

The main thing to aim for is the balance between the used CPU power, amount of data that is transported (in mapper, between mapper and reducer, and out the reducers) and the disk 'head movements'.

Each task in a mapreduce job works best if it can read/write the data 'with minimal disk head movements'. Usually described as "sequential reads/writes". But if the task is CPU bound the extra diskhead movements do not impact the job.

It seems to me that in this specific case you have

* a mapper that does quite a bit of CPU cycles (i.e. more mappers make it go faster because the CPU is the bottle neck and the disks can keep up in providing the input data).
* a reducer that does almost no CPU cycles and is mostly IO bound. This causes that with a single reducer you are still CPU bound, yet with 4 or more reducers you seem to be IO bound. So 4 reducers cause the disk head to move 'too much'.

Possible ways to handle this kind of situation:

First do exactly what you did: Do some test runs and see which setting performs best given this specific job and your specific cluster.

Then you have three options:

* Accept the situation you have
* Shift load from CPU to disk or the other way around.
* Get a bigger cluster: More CPUs and/or more disks.

Suggestions for shifting the load:

* If CPU bound and all CPUs are fully loaded then reduce the CPU load:
  + Check for needless CPU cycles in your code.
  + Switch to a 'lower CPU impact' compression codec: I.e. go from GZip to Snappy or to 'no compression'.
  + Tune the number of mappers/reducers in your job.
* If IO bound and you have some CPU capacity left:
  + Enable compression: This makes the CPUs work a bit harder and reduces the work the disks have to do.
  + Experiment with various compression codecs (I recommend sticking with either Snappy or Gzip ... I very often go with Gzip).
  + Tune the number of mappers/reducers in your job.