HowManyMapsAndReduces

Partitioning your job into maps and reduces

Picking the appropriate size for the tasks for your job can radically change the performance of Hadoop. Increasing the number of tasks increases the framework overhead, but increases load balancing and lowers the cost of failures. At one extreme is the 1 map/1 reduce case where nothing is distributed. The other extreme is to have 1,000,000 maps/ 1,000,000 reduces where the framework runs out of resources for the overhead.

Number of Maps

The number of maps is usually driven by the number of DFS blocks in the input files. Although that causes people to adjust their DFS block size to adjust the number of maps. The right level of parallelism for maps seems to be around 10-100 maps/node, although we have taken it up to 300 or so for very cpu-light map tasks. Task setup takes awhile, so it is best if the maps take at least a minute to execute.

Actually controlling the number of maps is subtle. The mapred.map.tasks parameter is just a hint to the InputFormat for the number of maps. The default InputFormat behavior is to split the total number of bytes into the right number of fragments. However, in the default case the DFS block size of the input files is treated as an upper bound for input splits. A lower bound on the split size can be set via mapred.min.split.size. Thus, if you expect 10TB of input data and have 128MB DFS blocks, you'll end up with 82k maps, unless your mapred.map.tasks is even larger. Ultimately the InputFormat determines the number of maps.

The number of map tasks can also be increased manually using the JobConf's conf.setNumMapTasks(int num). This can be used to increase the number of map tasks, but will not set the number below that which Hadoop determines via splitting the input data.

Number of Reduces

The ideal reducers should be the optimal value that gets them closest to:

- A multiple of the block size
- A task time between 5 and 15 minutes
- Creates the fewest files possible

Anything other than that means there is a good chance your reducers are less than great. There is a tremendous tendency for users to use a REALLY high value ("More parallelism means faster!") or a REALLY low value ("I don't want to blow my namespace quota!"). Both are equally dangerous, resulting in one or more of:

- Terrible performance on the next phase of the workflow
- Terrible performance due to the shuffle
- Terrible overall performance because you've overloaded the namenode with objects that are ultimately useless
- Destroying disk IO for no really sane reason
- Lots of network transfers due to dealing with crazy amounts of CFIF/MFIF work

Now, there are always exceptions and special cases. One particular special case is that if following that advice makes the next step in the workflow do ridiculous things, then we need to likely 'be an exception' in the above general rules of thumb.

Currently the number of reduces is limited to roughly 1000 by the buffer size for the output files (io.buffer.size * 2 * numReduces << heapSize). This will be fixed at some point, but until it is it provides a pretty firm upper bound.

The number of reduce tasks can also be increased in the same way as the map tasks, via JobConf's conf.setNumReduceTasks(int num).