Debugging

The MapReduce framework provides a facility to run user-provided scripts for debugging. When a MapReduce task fails, a user can run a debug script, to process task logs for example. The script is given access to the task's stdout and stderr outputs, syslog and jobconf. The output from the debug script's stdout and stderr is displayed on the console diagnostics and also as part of the job UI.

In the following sections we discuss how to submit a debug script with a job. The script file needs to be distributed and submitted to the framework.

How to distribute the script file:

The user needs to use DistributedCache to *distribute* and *symlink* the script file.

How to submit the script:

A quick way to submit the debug script is to set values for the properties `mapred.map.task.debug.script` and `mapred.reduce.task.debug.script`, for debugging map and reduce tasks respectively. These properties can also be set by using APIs JobConf.setMapDebugScript(String) and JobConf.setReduceDebugScript(String) . In streaming mode, a debug script can be submitted with the command-line options –`mapdebug` and `-reducedebug`, for debugging map and reduce tasks respectively.

The arguments to the script are the task's stdout, stderr, syslog and jobconf files. The debug command, run on the node where the MapReduce task failed, is:
`$script $stdout $stderr $syslog $jobconf`

Pipes programs have the c++ program name as a fifth argument for the command. Thus for the pipes programs the command is
`$script $stdout $stderr $syslog $jobconf $program`

Default Behavior:

For pipes, a default script is run to process core dumps under gdb, prints stack trace and gives info about running threads.

JobControl

JobControl is a utility which encapsulates a set of MapReduce jobs and their dependencies.

Data Compression

Hadoop MapReduce provides facilities for the application-writer to specify compression for both intermediate map-outputs and the job-outputs i.e. output of the reduces. It also comes bundled withCompressionCodec implementation for the zlib compression algorithm.
The gzip file format is also supported.

Hadoop also provides native implementations of the above compression codecs for reasons of both performance (zlib) and non-availability of Java libraries. More details on their usage and availability are available here.

Intermediate Outputs

Applications can control compression of intermediate map-outputs via the JobConf.setCompressMapOutput(boolean) api and the `CompressionCodec` to be used via theJobConf.setMapOutputCompressorClass(Class) api.

Job Outputs

Applications can control compression of job-outputs via the FileOutputFormat.setCompressOutput(JobConf, boolean) api and the `CompressionCodec` to be used can be specified via theFileOutputFormat.setOutputCompressorClass(JobConf, Class) api.

If the job outputs are to be stored in the SequenceFileOutputFormat, the required `SequenceFile.CompressionType` (i.e. `RECORD` / `BLOCK` - defaults to `RECORD`) can be specified via theSequenceFileOutputFormat.setOutputCompressionType(JobConf, SequenceFile.CompressionType) api.

Skipping Bad Records

Hadoop provides an option where a certain set of bad input records can be skipped when processing map inputs. Applications can control this feature through the SkipBadRecords class.

This feature can be used when map tasks crash deterministically on certain input. This usually happens due to bugs in the map function. Usually, the user would have to fix these bugs. This is, however, not possible sometimes. The bug may be in third party libraries, for example, for which the source code is not available. In such cases, the task never completes successfully even after multiple attempts, and the job fails. With this feature, only a small portion of data surrounding the bad records is lost, which may be acceptable for some applications (those performing statistical analysis on very large data, for example).

By default this feature is disabled. For enabling it, refer to SkipBadRecords.setMapperMaxSkipRecords(Configuration, long) and SkipBadRecords.setReducerMaxSkipGroups(Configuration, long).

With this feature enabled, the framework gets into 'skipping mode' after a certain number of map failures. For more details, see SkipBadRecords.setAttemptsToStartSkipping(Configuration, int). In 'skipping mode', map tasks maintain the range of records being processed. To do this, the framework relies on the processed record counter. See SkipBadRecords.COUNTER_MAP_PROCESSED_RECORDSand SkipBadRecords.COUNTER_REDUCE_PROCESSED_GROUPS. This counter enables the framework to know how many records have been processed successfully, and hence, what record range caused a task to crash. On further attempts, this range of records is skipped.

The number of records skipped depends on how frequently the processed record counter is incremented by the application. It is recommended that this counter be incremented after every record is processed. This may not be possible in some applications that typically batch their processing. In such cases, the framework may skip additional records surrounding the bad record. Users can control the number of skipped records through SkipBadRecords.setMapperMaxSkipRecords(Configuration, long) and SkipBadRecords.setReducerMaxSkipGroups(Configuration, long). The framework tries to narrow the range of skipped records using a binary search-like approach. The skipped range is divided into two halves and only one half gets executed. On subsequent failures, the framework figures out which half contains bad records. A task will be re-executed till the acceptable skipped value is met or all task attempts are exhausted. To increase the number of task attempts, useJobConf.setMaxMapAttempts(int) and JobConf.setMaxReduceAttempts(int).

Skipped records are written to HDFS in the sequence file format, for later analysis. The location can be changed through SkipBadRecords.setSkipOutputPath(JobConf, Path).