MapReduce

How Map and Reduce operations are actually carried out

Introduction

This document describes how MapReduce operations are carried out in Hadoop. If you are not familiar with the Google MapReduce programming model you should get acquainted with it first.

"Excerpt from TomWhite's blog: MapReduce"

- In essence, it allows massive data sets to be processed in a distributed fashion by breaking the processing into many small computations of two types:
    1. A Map operation that transforms the input into an intermediate representation.
    2. A Reduce function that recombines the intermediate representation into the final output.
- This processing model is ideal for the operations a search engine indexer like Nutch or Google needs to perform - like computing inlinks for URLs, or building inverted indexes - and it will mapred.pdf into a scalable, distributed search engine.

Map

As the Map operation is parallelized the input file set is first split to several pieces called FileSplits. If an individual file is so large that it will affect seek time it will be split to several Splits. The splitting does not know anything about the input file's internal logical structure, for example line-oriented text files are split on arbitrary byte boundaries. Then a new map task is created per FileSplit.

When an individual map task starts it will open a new output writer per configured reduce task. It will then proceed to read its FileSplit using the RecordReader it gets from the specified InputFormat. InputFormat parses the input and generates key-value pairs. InputFormat must also handle records that may be split on the FileSplit boundary. For example TextInputFormat will read the last line of the FileSplit past the split boundary and, when reading other than the first FileSplit, TextInputFormat ignores the content up to the first newline.

It is not necessary for the InputFormat to generate both meaningful keys *and* values. For example the default output from TextInputFormat consists of input lines as values and somewhat meaninglessly line start file offsets as keys - most applications only use the lines and ignore the offsets.

As key-value pairs are read from the RecordReader they are passed to the configured Mapper. The user supplied Mapper does whatever it wants with the input pair and calls OutputCollector.collect with key-value pairs of its own choosing. The output it generates must use one key class and one value class. This is because the Map output will be written into a SequenceFile which has per-file type information and all the records must have the same type (use subclassing if you want to output different data structures). The Map input and output key-value pairs are not necessarily related typewise or in cardinality.

When Mapper output is collected it is partitioned, which means that it will be written to the output specified by the Partitioner. The default HashPartitioner uses the hashcode function on the key's class (which means that this hashcode function must be good in order to achieve an even workload across the reduce tasks). See MapTask for details.

N input files will generate M map tasks to be run and each map task will generate as many output files as there are reduce tasks configured in the system. Each output file will be targeted at a specific reduce task and the map output pairs from all the map tasks will be routed so that all pairs for a given key end up in files targeted at a specific reduce task.

Combine

When the map operation outputs its pairs they are already available in memory. For efficiency reasons, sometimes it makes sense to take advantage of this fact by supplying a combiner class to perform a reduce-type function. If a combiner is used then the map key-value pairs are not immediately written to the output. Instead they will be collected in lists, one list per each key value. When a certain number of key-value pairs have been written, this buffer is flushed by passing all the values of each key to the combiner's reduce method and outputting the key-value pairs of the combine operation as if they were created by the original map operation.

For example, a word count MapReduce application whose map operation outputs (*word*, 1) pairs as words are encountered in the input can use a combiner to speed up processing. A combine operation will start gathering the output in in-memory lists (instead of on disk), one list per word. Once a certain number of pairs is output, the combine operation will be called once per unique word with the list available as an iterator. The combiner then emits (*word*, count-in-this-part-of-the-input) pairs. From the viewpoint of the Reduce operation this contains the same information as the original Map output, but there should be far fewer pairs output to disk and read from disk.

Reduce

When a reduce task starts, its input is scattered in many files across all the nodes where map tasks ran. If run in distributed mode these need to be first copied to the local filesystem in a *copy phase* (see ReduceTaskRunner).

Once all the data is available locally it is appended to one file in an *append phase*. The file is then merge sorted so that the key-value pairs for a given key are contiguous (*sort phase*). This makes the actual reduce operation simple: the file is read sequentially and the values are passed to the reduce method with an iterator reading the input file until the next key value is encountered. See ReduceTask for details.

At the end, the output will consist of one output file per executed reduce task. The format of the files can be specified with JobConf.setOutputFormat. If SequentialOutputFormat is used then the output key and value classes must also be specified.