Topic 3: MapReduce

- Explain the MapReduce paradigm and programming model
- Explain the system architecture
- Explain a concrete example application and how it is executed
- Explain how to optimize the performance and how worker failures can be handled

Material: CC book chapter 14, MapReduce slides, and MapReduce paper

MapReduce Paradigm & Programming Model

MapReduce is a computation model whose goal is to make it possible to parallelize computation simply, using two computation primitives: Map and Reduce. The actual logic for making these primitives run in parallel on distributed systems should then be abstracted away by the library.

The computation takes a set of input key/value pairs and produces a set out output key/value pairs.

Map

Map is the first function which the user must define. Its responsibility is to produce an intermediate set of key/value pairs.

Between Map and Reduce

The MapReduce library aggregates all values for a given key before passing it on to the Reducer.

Reduce

Reduce is the second function which the user must define. It receives a key and the set of its values, and produces the output key/value pairs.

System Architecture

There is one master node, many workers, M Map tasks and R Reduce tasks. They share access to a global file system, such as GFS or HDFS, etc. The input data is split into M segments, and workers are assigned Map or Reduce tasks by the master. The Map worker stores the intermediate data locally, and the master assigns a Reduce worker to read it and compute the final result. The result is stored in the global file system.

Example Application & Execution

WordCount is a classic example, where we want to count the occurrence of words in large or many documents.

The Map function receives a list of words, and emits `<word>/1` for each word. The Reduce function receives the word, and a list of 1's, emitting `<word>/<list.length()>` for each word, where `list.length()` is the length of the list of 1's.

Execution

1. Split the data into M pieces. Fork the program.
2. The master assigns M map tasks and R reduce tasks to available workers.
3. Each Map worker performs the Map function, storing the result in memory.
4. Periodically, these values are written to disk, and their location is reported to the master node.
5. A reduce worker is notified by the master about the locations, and uses RPC to read the data. It then sorts the data.
6. For each unique key, the reduce function passes they key and its values to the user's reduce function. The result is emitted to the final output.
7. Once all tasks are completed, the master returns.

Optimizing Performance

- Specifying a custom partitioning function for mapping intermediate key values to the reduce shards.
- Custom combiner functions after mapping tasks to reduce the intermediate data sent from a mapping worker to a reducer.
- Enabling backup tasks to reduce slowdown by stragglers.

Handling Worker Failures

The master marks a worker as failed if it does not respond to pings within a certain amount of time. All in-progress tasks are reset to idle, and become available for scheduling on other workers. All mapping tasks completed by the failed worker are marked as idle as well, since their result is stored on the worker and not the global file system.