Abstractions for Distributed Data Processing

Speaker: Bogdan Alexe
Scribe: Karl Schnaitter

Notes for CMPS 253, April 24, 2007

1 MapReduce

Inside Google, there is often a need to process very large data sets. This processing is performed by a large cluster of machines. In order to perform these tasks, it was originally necessary to write a program from scratch. A distributed processing application needs to properly parallelize the work in a balanced way while handling machine and network failures. At the same time, the application needs to be tuned to minimize network I/O, which is the main bottleneck. These issues can be a major distraction, and can obscure the logic in the code. Even worse, sometimes these issues would not be addressed because of the extreme amout of necessary effort. Google ultimately decided that it was necessary to design an abstraction that separated the logic of the computation from the details of distributed processing.

This problem was solved with the MapReduce abstraction. In order to process some data, a user can specify two functions, `map` and `reduce`. These functions have the following polymorphic types.

```
   map : k1 v1 -> list <k2, v2>
reduce : <k2, list v2> -> list v2
```

The semantics of these functions are inspired from functional programming. The `map` function takes an input key/value pair and produces a list of intermediate key/value pairs. The `reduce` function gets called once for each intermediate key. It sees the list of values for the intermediate key, and produces another list of values as output. The result of the reduce step is usually a small number of values, and most commonly a single value. An example of a reduce task that returns a list would be a "top-10" operator.

Many powerful tasks can be expressed with MapReduce. One example is an application that counts the term frequencies for a collection of documents. The `map` phase outputs key/value pairs where the key is a term and the value is 1. The `reduce` phase simply adds the 1's together. A more complex application of MapReduce is web indexing. Google actually implemented their indexing procedure in terms of about ten MapReduce tasks. This is a very interesting point because it is one of the most important tasks at Google, yet it is implemented using such a high-level abstraction rather than being hand tuned at a low level. In theory, it should be possible to get better performance by implementing the indexing procedure from scratch, but MapReduce seems to work better in practice. The reason for this seems to be that the performance tuning is so difficult that it can't be done as effectively as it can with MapReduce.

All MapReduce tasks are executed using a similar workflow. The root program starts by spawning mapper workers, reducer workers, and a master machine. The input is divided into M splits and each split is processed by some mapper. The map results are stored on the mapper's local disk, and the master is notified when each map task completes. The map results are divided into R partitions and each reducer processes one partition. The reduce results are stored in the global file system and the master is notified when the task completes. The master is primarily responsible for communication with the mappers and reducers. It decides how to assign input data to mappers in a way that minimizes network traffic. In addition, the master may resolve problems, such as restarting computations that are lost due to a failure, or replicating computations that are going too slowly.

In class, we discussed how the MapReduce abstraction might be extended. We noticed that some tasks are defined with a pipeline of MapReduce steps. In the current implementation, the separate tasks are performed one at a time. It might be possible for a program to express the steps as a MapReduce sequence. Then the implementation could process the steps all at once in a more streaming fashion. This would make the job of the master more complex, but in theory it should be possible.

2 Sawzall

MapReduce offers an excellent abstraction for processing data on a large scale, but the employees at Google noticed that the mechanism is overly general and flexible for many common tasks. In order to simplify the work needed to implement these tasks, a new language was designed to act as an interface over MapReduce. This language is named Sawzall.

The primary function of a Sawzall program is to aggregate a collection of records. The result of the aggregation usually summarizes the records in some way. For example, the aggregation could be the sum or maximum of values from the records. Another possibility would be to aggregate the records into a small random sample. An aggregator is referred to as a *table*. A table may only be updated based on one record at a time. This can be a limitation. For instance, there is no way to detect whether two records exist with the same value within the execution of the program.

The following is a small, but complete, Sawzall program.

```
total: table sum of float;
x: float = input;
emit total <- x;
```

Each record is a `float` and fetched by accessing the special `input` variable. The `total` table stores the sum of the records. A subtle point is that `table` variables are initialized once before the records are processed, but other variables, such as `float`s are initialized before each record is processed. In this example, the aggregator is just a single number, but there are more complex possibilities. Other aggregators maintain a set of values. It is also possible to create an array of aggregators of the same type, or to create various aggregators of different types.

There are many other features in Sawzall, such as special array iterators, but most of these features are unrelated to the issues of distributed data processing. One important feature relates to "undefined" records. Sawzall operates on huge data sets, which may be unreliable. If there is an unexpected problem with a few records, it is often okay to ignore these records. A Sawzall program can avoid failures on bad data by using an operation that tests whether a variable is defined. Undefined records are usually just skipped.

Despite the highly specialized features, and some clumsy syntax, the language is very small and easy to learn. It has enjoyed widespread use at Google for many important tasks. It has also given the Google admins a very powerful method for access control. For instance, if there are files whose contents are private, a user might be granted access to process the files with Sawzall, but with the sensitive information hidden. This ability has added to the popularity of the language.