MapReduce

Jim Harner

1/6/2020

4.5 MapReduce

MapReduce is a simple but powerful programming model for breaking a task into pieces and operating on those pieces in an embarrassingly parallel manner across a cluster.

MapReduce operates on key-value pairs. The input, output, and intermediate data are all key-value pairs. A MapReduce job consists of three phases that operate on these key-value pairs: the map, the shuffle/sort, and the reduce.

- *Map*: A map function is applied to each input key-value pair, which does some user-defined processing and emits new key-value pairs to intermediate storage to be processed by the reduce function (after shuffling).
- *Shuffle/Sort*: The map output values are collected for each unique map output key and passed to a reduce function.
- *Reduce*: A reduce function is applied in parallel to all values corresponding to each unique map output key and emits output key-value pairs.

MapReduce cluster computing uses a linear dataflow on distributed programs. In its canonical form, it reads input data from HDFS, maps a function across the data, reduces the results of the map, and stores the reduction results in HDFS. See the schematic here (scroll down to see the figure).

The `map` function and `reduce` function are user-defined. The MapReduce engine takes care of everything else. We will get a better feel for how things work by looking at some examples in this section.

`map()` and `reduce()` functions are stables of functional programming languages, but take on new importance within Hadoop.

The MapReduce model outlines a way to perform work across a cluster of inexpensive, commodity machines.

Two phases:

- Map phase: divides input data and groups the pieces into smaller, independent piles of related material;
- Reduce phase: perform some action on each pile.

MapReduce is a "divide-and-conquer" model. The piles can be reduced in parallel because they do not rely on one another.

Map Phase:

1. Each cluster node runs a part of the initial big data and runs a Map task on each record (item) of input.
2. The Map task runs in parallel and creates a *key/value* pair for each record. The key identifies the items pile for the reduce operation. The value is often the record itself.

The Shuffle:

Each key/value pair is assigned a pile based on the key.

Reduce Phase:

1. The cluster nodes then run the Reduce task on each pile.

2. The Reduce task typically emits output for each pile.

See Figure 5.1. in the Parallel R book.

Pseudocode Examples

For these examples, we will use a fictitious text input format in which each record is a comma-separated line that describes a phone call:

{id}, {date}, {caller_num}, {caller_carrier}, {dest_num}, {dest_carrier}, {length}

Calculate the average call length for each date

The Map task groups the records by `date`, and then calculates the mean (average) call length in the Reduce task.

Map task:

- Receives a single line of input (that is, one input record)

- Uses text manipulation to extract the {date} and {length} fields

- Emits key: {date}, value: {length}

Reduce task:

- Receives the key: {date}, value:{length1 … lengthN} i.e., each reduce task receives all of the call lengths for a single date based on the shuffle

- Loops through {length1 … lengthN} to calculate total call length, and also to note the number of calls

- Calculates the mean (divides the total call length by the number of calls)

- Outputs the date and the mean call length

Number of Calls by Each User, on Each Date

The goal is to get a breakdown of each caller for each date. The Map phase will define the keys to group the inputs, and the Reduce task will perform the calculations. Notice that the Map task emits a dummy value (the number 1) as its value because we use the Reduce task for a simple counting operation.

Map task:

- Receives single line of input

- Uses text manipulation to extract {date}, {caller num}

- Emits key: {date}{caller num}, value: 1

Reduce task:

- Receives key: {date}{caller num}, value: {1 … 1}

- Loops through each item, to count total number of items (calls)

- Outputs {date}, {caller num} and the number of calls

Binary and Whole-File Data: SequenceFiles

You're in a different situation if you plan to use Hadoop with binary data (sound files, image files, proprietary data formats) or if you want to treat an entire text file (e.g., XML document) as a record.

By default, when you point Hadoop to an input file, it will assume it is a text document and treat each line as a record. There are times when this is not what you want: maybe you're performing feature extraction on sound files, or you wish to perform sentiment analysis on text documents.

Use a special archive called a *SequenceFile*. A SequenceFile is similar to a zip or tar file, in that it's just a container for other files. Hadoop considers each file in a SequenceFile to be its own record.

To manage zip files, you use the `zip` command. Tar file? Use `tar`. SequenceFiles? Hadoop doesn't ship with any tools for this, but you still have options: you can write a Hadoop job using the Java API; or you can use the `forqlift` command-line tool.

`forqlift` strives to be simple and straightforward. For example, to create a SequenceFile from a set of MP3s, you would run:

```
forqlift create --file=/path/to/file.seq *.mp3
```

Then, in a Hadoop job, the Map task's key would be an MP3's filename and the value would be the file's contents.

Running MapReduce Jobs

MapReduce jobs are run on a Hadoop cluster, which can be built locally using contaners, or by using a cloud service, e.g., Amazon Web Service (AWS). In the next subsection we will run several MapReduce jobs using Hadoop Streaming.

AWS provides computing resources such as virtual servers and storage in *metered* (pay-per-use) way. You can hand-build your cluster using virtual servers on Elastic Compute Cloud (EC2), or you can leverage the Hadoop-on-demand service called Elastic MapReduce (EMR).

An EMR-based cluster is designed to be ephemeral: by default, AWS tears down the cluster as soon as your job completes. All of the cluster nodes and resources disappear. That means you can't leverage HDFS for long-term storage.