4.7 Distributed Data Processing

Distributed systems are often used to collect, access, and manipulate large data sets. For example, the database systems described earlier in the chapter can operate over datasets that are stored across multiple machines. No single machine may contain the data necessary to respond to a query, and so communication is required to service requests.

This section investigates a typical big data processing scenario in which a data set too large to be processed by a single machine is instead distributed among many machines, each of which process a portion of the dataset. The result of processing must often be aggregated across machines, so that results from one machine's computation can be combined with others. To coordinate this distributed data processing, we will discuss a programming framework called MapReduce.

Creating a distributed data processing application with MapReduce combines many of the ideas presented throughout this text. An application is expressed in terms of pure functions that are used to *map* over a large dataset and then to *reduce* the mapped sequences of values into a final result.

Familiar concepts from functional programming are used to maximal advantage in a MapReduce program. MapReduce requires that the functions used to map and reduce the data be pure functions. In general, a program expressed only in terms of pure functions has considerable flexibility in how it is executed. Sub-expressions can be computed in arbitrary order and in parallel without affecting the final result. A MapReduce application evaluates many pure functions in parallel, reordering computations to be executed efficiently in a distributed system.

The principal advantage of MapReduce is that it enforces a separation of concerns between two parts of a distributed data processing application:

1. The map and reduce functions that process data and combine results.
2. The communication and coordination between machines.

The coordination mechanism handles many issues that arise in distributed computing, such as machine failures, network failures, and progress monitoring. While managing these issues introduces some complexity in a MapReduce application, none of that complexity is exposed to the application developer. Instead, building a MapReduce application only requires specifying the map and reduce functions in (1) above; the challenges of distributed computation are hidden via abstraction.

4.7.1 MapReduce

The MapReduce framework assumes as input a large, unordered stream of input values of an arbitrary type. For instance, each input may be a line of text in some vast corpus. Computation proceeds in three steps.

1. A map function is applied to each input, which outputs zero or more intermediate key-value pairs of an arbitrary type.
2. All intermediate key-value pairs are grouped by key, so that pairs with the same key can be reduced together.
3. A reduce function combines the values for a given key `k`; it outputs zero or more values, which are each associated with `k` in the final output.

To perform this computation, the MapReduce framework creates tasks (perhaps on different machines) that perform various roles in the computation. A *map task* applies the map function to some subset of the input data and outputs intermediate key-value pairs. A *reduce* task sorts and groups key-value pairs by key, then applies the reduce function to the values for each key. All communication between map and reduce tasks is handled by the framework, as is the task of grouping intermediate key-value pairs by key.

In order to utilize multiple machines in a MapReduce application, multiple mappers run in parallel in a *map phase*, and multiple reducers run in parallel in a *reduce phase*. In between these phases, the *sort phase* groups together key-value pairs by sorting them, so that all key-value pairs with the same key are adjacent.

Consider the problem of counting the vowels in a corpus of text. We can solve this problem using the MapReduce framework with an appropriate choice of map and reduce functions. The map function takes as input a line of text and outputs key-value pairs in which the key is a vowel and the value is a count. Zero counts are omitted from the output:

```
def count_vowels(line):
    """A map function that counts the vowels in a line."""
    for vowel in 'aeiou':
        count = line.count(vowel)
        if count > 0:
            emit(vowel, count)
```

The reduce function is the built-in sum functions in Python, which takes as input an iterator over values (all values for a given key) and returns their sum.

4.7.2 Local Implementation

To specify a MapReduce application, we require an implementation of the MapReduce framework into which we can insert map and reduce functions. In the following section, we will use the open-source Hadoop implementation. In this section, we develop a minimal implementation using built-in tools of the Unix operating system.

The Unix operating system creates an abstraction barrier between user programs and the underlying hardware of a computer. It provides a mechanism for programs to communicate with each other, in particular by allowing one program to consume the output of another. In their seminal text on Unix programming, Kernigham and Pike assert that, ""The power of a system comes more from the relationships among programs than from the programs themselves."

A Python source file can be converted into a Unix program by adding a comment to the first line indicating that the program should be executed using the Python 3 interpreter. The input to a Unix program is an iterable object called *standard input* and accessed as `sys.stdin`. Iterating over this object yields string-valued lines of text. The output of a Unix program is called *standard output* and accessed as `sys.stdout`. The built-in `print` function writes a line of text to standard output. The following Unix program writes each line of its input to its output, in reverse:

```
#!/usr/bin/env python3

import sys

for line in sys.stdin:
    print(line.strip('\n')[::-1])
```

If we save this program to a file called `rev.py`, we can execute it as a Unix program. First, we need to tell the operating system that we have created an executable program:

```
$ chmod u+x rev.py
```

Next, we can pass input into this program. Input to a program can come from another program. This effect is achieved using the | symbol (called "pipe") which channels the output of the program before the pipe into the program after the pipe. The program `nslookup` outputs the host name of an IP address (in this case for the New York Times):

```
$ nslookup 170.149.172.130 | ./rev.py
moc.semityn.www
```

The `cat` program outputs the contents of files. Thus, the `rev.py` program can be used to reverse the contents of the `rev.py` file:

```
$ cat rev.py | ./rev.py
3nohtyp vne/nib/rsu/!#

sys tropmi

:nidts.sys ni enil rof
)]1-::[)'n\'(pirts.enil(tnirp
```

These tools are enough for us to implement a basic MapReduce framework. This version has only a single map task and single reduce task, which are both Unix programs implemented in Python. We run an entire MapReduce application using the following command:

```
$ cat input | ./mapper.py | sort | ./reducer.py
```

The `mapper.py` and `reducer.py` programs must implement the map function and reduce function, along with some simple input and output behavior. For instance, in order to implement the vowel counting application described above, we would write the following `count_vowels_mapper.py` program:

```
#!/usr/bin/env python3

import sys
from mr import emit

def count_vowels(line):
    """A map function that counts the vowels in a line."""
    for vowel in 'aeiou':
        count = line.count(vowel)
        if count > 0:
            emit(vowel, count)

for line in sys.stdin:
    count_vowels(line)
```

In addition, we would write the following `sum_reducer.py` program:

```
#!/usr/bin/env python3

import sys
```

```
from mr import values_by_key, emit

for key, value_iterator in values_by_key(sys.stdin):
    emit(key, sum(value_iterator))
```

The mr module is a companion module to this text that provides the functions `emit` to emit a key-value pair and `group_values_by_key` to group together values that have the same key. This module also includes an interface to the Hadoop distributed implementation of MapReduce.

Finally, assume that we have the following input file called `haiku.txt`:

```
Google MapReduce
Is a Big Data framework
For batch processing
```

Local execution using Unix pipes gives us the count of each vowel in the haiku:

```
$ cat haiku.txt | ./count_vowels_mapper.py | sort | ./sum_reducer.py
'a'   6
'e'   5
'i'   2
'o'   5
'u'   1
```

4.7.3 Distributed Implementation

Hadoop is the name of an open-source implementation of the MapReduce framework that executes MapReduce applications on a cluster of machines, distributing input data and computation for efficient parallel processing. Its streaming interface allows arbitrary Unix programs to define the map and reduce functions. In fact, our `count_vowels_mapper.py` and `sum_reducer.py` can be used directly with a Hadoop installation to compute vowel counts on large text corpora.

Hadoop offers several advantages over our simplistic local MapReduce implementation. The first is speed: map and reduce functions are applied in parallel using different tasks on different machines running simultaneously. The second is fault tolerance: when a task fails for any reason, its result can be recomputed by another task in order to complete the overall computation. The third is monitoring: the framework provides a user interface for tracking the progress of a MapReduce application.

In order to run the vowel counting application using the provided `mapreduce.py` module, install Hadoop, change the assignment statement of `HADOOP` to the root of your local installation, copy a collection of text files into the Hadoop distributed file system, and then run:

```
$ python3 mr.py run count_vowels_mapper.py sum_reducer.py [input] [output]
```

where `[input]` and `[output]` are directories in the Hadoop file system.

For more information on the Hadoop streaming interface and use of the system, consult the Hadoop Streaming Documentation.

Continue: 4.8 Parallel Computing