CHAPTER 24
MapReduce

The future has already arrived. It's just not evenly distributed yet.
—William Gibson

MapReduce is a programming model for performing parallel processing on large data sets. Although it is a powerful technique, its basics are relatively simple.

Imagine we have a collection of items we'd like to process somehow. For instance, the items might be website logs, the texts of various books, image files, or anything else. A basic version of the MapReduce algorithm consists of the following steps:

1. Use a `mapper` function to turn each item into zero or more key-value pairs. (Often this is called the `map` function, but there is already a Python function called `map` and we don't need to confuse the two.)
2. Collect together all the pairs with identical keys.
3. Use a `reducer` function on each collection of grouped values to produce output values for the corresponding key.

This is all sort of abstract, so let's look at a specific example. There are few absolute rules of data science, but one of them is that your first MapReduce example has to involve counting words.

Example: Word Count

DataSciencester has grown to millions of users! This is great for your job security, but it makes routine analyses slightly more difficult.

For example, your VP of Content wants to know what sorts of things people are talking about in their status updates. As a first attempt, you decide to count the words that appear, so that you can prepare a report on the most frequent ones.

When you had a few hundred users this was simple to do:

```
def word_count_old(documents):
    """word count not using MapReduce"""
    return Counter(word
        for document in documents
        for word in tokenize(document))
```

With millions of users the set of `documents` (status updates) is suddenly too big to fit on your computer. If you can just fit this into the MapReduce model, you can use some "big data" infrastructure that your engineers have implemented.

First, we need a function that turns a document into a sequence of key-value pairs. We'll want our output to be grouped by word, which means that the keys should be words. And for each word, we'll just emit the value `1` to indicate that this pair corresponds to one occurrence of the word:

```
def wc_mapper(document):
    """for each word in the document, emit (word,1)"""
    for word in tokenize(document):
        yield (word, 1)
```

Skipping the "plumbing" step 2 for the moment, imagine that for some word we've collected a list of the corresponding counts we emitted. Then to produce the overall count for that word we just need:

```
def wc_reducer(word, counts):
    """sum up the counts for a word"""
    yield (word, sum(counts))
```

Returning to step 2, we now need to collect the results from `wc_mapper` and feed them to `wc_reducer`. Let's think about how we would do this on just one computer:

```
def word_count(documents):
    """count the words in the input documents using MapReduce"""

    # place to store grouped values
    collector = defaultdict(list)

    for document in documents:
        for word, count in wc_mapper(document):
            collector[word].append(count)

    return [output
            for word, counts in collector.iteritems()
            for output in wc_reducer(word, counts)]
```

Imagine that we have three documents `["data science", "big data", "science fiction"]`.

Then `wc_mapper` applied to the first document yields the two pairs `("data", 1)` and `("science", 1)`. After we've gone through all three documents, the `collector` contains

```
{ "data" : [1, 1],
  "science" : [1, 1],
  "big" : [1],
  "fiction" : [1] }
```

Then `wc_reducer` produces the count for each word:

```
[("data", 2), ("science", 2), ("big", 1), ("fiction", 1)]
```

Why MapReduce?

As mentioned earlier, the primary benefit of MapReduce is that it allows us to distribute computations by moving the processing to the data. Imagine we want to word-count across billions of documents.

Our original (non-MapReduce) approach requires the machine doing the processing to have access to every document. This means that the documents all need to either live on that machine or else be transferred to it during processing. More important, it means that the machine can only process one document at a time.

Possibly it can process up to a few at a time if it has multiple cores and if the code is rewritten to take advantage of them. But even so, all the documents still have to *get to* that machine.

Imagine now that our billions of documents are scattered across 100 machines. With the right infrastructure (and glossing over some of the details), we can do the following:

- Have each machine run the mapper on its documents, producing lots of (key, value) pairs.
- Distribute those (key, value) pairs to a number of "reducing" machines, making sure that the pairs corresponding to any given key all end up on the same machine.
- Have each reducing machine group the pairs by key and then run the reducer on each set of values.
- Return each (key, output) pair.

What is amazing about this is that it scales horizontally. If we double the number of machines, then (ignoring certain fixed-costs of running a MapReduce system) our computation should run approximately twice as fast. Each mapper machine will only need to do half as much work, and (assuming there are enough distinct keys to further distribute the reducer work) the same is true for the reducer machines.

MapReduce More Generally

If you think about it for a minute, all of the word-count-specific code in the previous example is contained in the `wc_mapper` and `wc_reducer` functions. This means that with a couple of changes we have a much more general framework (that still runs on a single machine):

```
def map_reduce(inputs, mapper, reducer):
    """runs MapReduce on the inputs using mapper and reducer"""
    collector = defaultdict(list)

    for input in inputs:
        for key, value in mapper(input):
            collector[key].append(value)

    return [output
            for key, values in collector.iteritems()
            for output in reducer(key,values)]
```

And then we can count words simply by using:

```
word_counts = map_reduce(documents, wc_mapper, wc_reducer)
```

This gives us the flexibility to solve a wide variety of problems.

Before we proceed, observe that `wc_reducer` is just summing the values corresponding to each key. This kind of aggregation is common enough that it's worth abstracting it out:

```
def reduce_values_using(aggregation_fn, key, values):
    """reduces a key-values pair by applying aggregation_fn to the values"""
    yield (key, aggregation_fn(values))

def values_reducer(aggregation_fn):
    """turns a function (values -> output) into a reducer
    that maps (key, values) -> (key, output)"""
    return partial(reduce_values_using, aggregation_fn)
```

after which we can easily create:

```
sum_reducer = values_reducer(sum)
max_reducer = values_reducer(max)
min_reducer = values_reducer(min)
count_distinct_reducer = values_reducer(lambda values: len(set(values)))
```

and so on.

Example: Analyzing Status Updates

The content VP was impressed with the word counts and asks what else you can learn from people's status updates. You manage to extract a data set of status updates that look like:

```
{"id": 1,
 "username" : "joelgrus",
 "text" : "Is anyone interested in a data science book?",
 "created_at" : datetime.datetime(2013, 12, 21, 11, 47, 0),
 "liked_by" : ["data_guy", "data_gal", "mike"] }
```

Let's say we need to figure out which day of the week people talk the most about data science. In order to find this, we'll just count how many data science updates there are on each day of the week. This means we'll need to group by the day of week, so that's our key. And if we emit a value of `1` for each update that contains "data science," we can simply get the total number using `sum`:

```
def data_science_day_mapper(status_update):
    """yields (day_of_week, 1) if status_update contains "data science" """
    if "data science" in status_update["text"].lower():
        day_of_week = status_update["created_at"].weekday()
        yield (day_of_week, 1)

data_science_days = map_reduce(status_updates,
                               data_science_day_mapper,
                               sum_reducer)
```

As a slightly more complicated example, imagine we need to find out for each user the most common word that she puts in her status updates. There are three possible approaches that spring to mind for the `mapper`:

- Put the username in the key; put the words and counts in the values.
- Put the word in key; put the usernames and counts in the values.
- Put the username and word in the key; put the counts in the values.

If you think about it a bit more, we definitely want to group by `username`, because we want to consider each person's words separately. And we don't want to group by `word`, since our reducer will need to see all the words for each person to find out which is the most popular. This means that the first option is the right choice:

```
def words_per_user_mapper(status_update):
    user = status_update["username"]
    for word in tokenize(status_update["text"]):
        yield (user, (word, 1))
```

```
def most_popular_word_reducer(user, words_and_counts):
    """given a sequence of (word, count) pairs,
    return the word with the highest total count"""

    word_counts = Counter()
    for word, count in words_and_counts:
        word_counts[word] += count

    word, count = word_counts.most_common(1)[0]

    yield (user, (word, count))

user_words = map_reduce(status_updates,
                        words_per_user_mapper,
                        most_popular_word_reducer)
```

Or we could find out the number of distinct status-likers for each user:

```
def liker_mapper(status_update):
    user = status_update["username"]
    for liker in status_update["liked_by"]:
        yield (user, liker)

distinct_likers_per_user = map_reduce(status_updates,
                                      liker_mapper,
                                      count_distinct_reducer)
```

Example: Matrix Multiplication

Recall from "Matrix Multiplication" on page 260 that given a $m \times n$ matrix A and a $n \times k$ matrix B, we can multiply them to form a $m \times k$ matrix C, where the element of C in row i and column j is given by:

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + ... + A_{in}B_{nj}$$

As we've seen, a "natural" way to represent a $m \times n$ matrix is with a `list` of `lists`, where the element A_{ij} is the jth element of the ith list.

But large matrices are sometimes *sparse*, which means that most of their elements equal zero. For large sparse matrices, a list of lists can be a very wasteful representation. A more compact representation is a list of tuples `(name, i, j, value)` where `name` identifies the matrix, and where `i, j, value` indicates a location with nonzero value.

For example, a billion × billion matrix has a *quintillion* entries, which would not be easy to store on a computer. But if there are only a few nonzero entries in each row, this alternative representation is many orders of magnitude smaller.

Given this sort of representation, it turns out that we can use MapReduce to perform matrix multiplication in a distributed manner.

To motivate our algorithm, notice that each element A_{ij} is only used to compute the elements of C in row i, and each element B_{ij} is only used to compute the elements of C in column j. Our goal will be for each output of our `reducer` to be a single entry of C, which means we'll need our mapper to emit keys identifying a single entry of C. This suggests the following:

```
def matrix_multiply_mapper(m, element):
    """m is the common dimension (columns of A, rows of B)
    element is a tuple (matrix_name, i, j, value)"""
    name, i, j, value = element

    if name == "A":
        # A_ij is the jth entry in the sum for each C_ik, k=1..m
        for k in range(m):
            # group with other entries for C_ik
            yield((i, k), (j, value))
    else:
        # B_ij is the i-th entry in the sum for each C_kj
        for k in range(m):
            # group with other entries for C_kj
            yield((k, j), (i, value))

def matrix_multiply_reducer(m, key, indexed_values):
    results_by_index = defaultdict(list)
    for index, value in indexed_values:
        results_by_index[index].append(value)

    # sum up all the products of the positions with two results
    sum_product = sum(results[0] * results[1]
                      for results in results_by_index.values()
                      if len(results) == 2)

    if sum_product != 0.0:
        yield (key, sum_product)
```

For example, if you had the two matrices

```
A = [[3, 2, 0],
     [0, 0, 0]]

B = [[4, -1, 0],
     [10, 0, 0],
     [0, 0, 0]]
```

you could rewrite them as tuples:

```
entries = [("A", 0, 0, 3), ("A", 0, 1,  2),
           ("B", 0, 0, 4), ("B", 0, 1, -1), ("B", 1, 0, 10)]
mapper = partial(matrix_multiply_mapper, 3)
```

```
reducer = partial(matrix_multiply_reducer, 3)

map_reduce(entries, mapper, reducer) # [((0, 1), -3), ((0, 0), 32)]
```

This isn't terribly interesting on such small matrices, but if you had millions of rows and millions of columns, it could help you a lot.

An Aside: Combiners

One thing you have probably noticed is that many of our mappers seem to include a bunch of extra information. For example, when counting words, rather than emitting `(word, 1)` and summing over the values, we could have emitted `(word, None)` and just taken the length.

One reason we didn't do this is that, in the distributed setting, we sometimes want to use *combiners* to reduce the amount of data that has to be transferred around from machine to machine. If one of our mapper machines sees the word "data" 500 times, we can tell it to combine the 500 instances of `("data", 1)` into a single `("data", 500)` before handing off to the reducing machine. This results in a lot less data getting moved around, which can make our algorithm substantially faster still.

Because of the way we wrote our reducer, it would handle this combined data correctly. (If we'd written it using `len` it would not have.)

For Further Exploration

- The most widely used MapReduce system is Hadoop, which itself merits many books. There are various commercial and noncommercial distributions and a huge ecosystem of Hadoop-related tools.

  In order to use it, you have to set up your own *cluster* (or find someone to let you use theirs), which is not necessarily a task for the faint-hearted. Hadoop mappers and reducers are commonly written in Java, although there is a facility known as "Hadoop streaming" that allows you to write them in other languages (including Python).
- Amazon.com offers an Elastic MapReduce service that can programmatically create and destroy clusters, charging you only for the amount of time that you're using them.
- mrjob is a Python package for interfacing with Hadoop (or Elastic MapReduce).
- Hadoop jobs are typically high-latency, which makes them a poor choice for "real-time" analytics. There are various "real-time" tools built on top of Hadoop, but there are also several alternative frameworks that are growing in popularity. Two of the most popular are Spark and Storm.

- All that said, by now it's quite likely that the flavor of the day is some hot new distributed framework that didn't even exist when this book was written. You'll have to find that one yourself.