5 Advanced MapReduce

This chapter covers

- Chaining multiple MapReduce jobs
- Performing joins of multiple data sets
- Creating Bloom filters

As your data processing becomes more complex you'll want to exploit different Hadoop features. This chapter will focus on some of these more advanced techniques.

When handling advanced data processing, you'll often find that you can't program the process into a single MapReduce job. Hadoop supports *chaining* MapReduce programs together to form a bigger job. You'll also find that advanced data processing often involves more than one data set. We'll explore various *joining* techniques in Hadoop for simultaneously processing multiple data sets. You can code certain data processing tasks more efficiently when processing a group of records at a time. We've seen how Streaming natively supports the ability to process a whole split at a time, and the Streaming implementation of the maximum function takes advantage of this ability. We'll see that the same is true for Java programs. We'll discover the Bloom filter and implement it with a mapper that keeps state information across records.

5.1 Chaining MapReduce jobs

You've been doing data processing tasks which a single MapReduce job can accomplish. As you get more comfortable writing MapReduce programs and take on more ambitious data processing tasks, you'll find that many complex tasks need to be broken down into simpler subtasks, each accomplished by an individual MapReduce job. For example, from the citation data set you may be interested in finding the ten most-cited patents. A sequence of two MapReduce jobs can do this. The first one creates the "inverted" citation data set and counts the number of citations for each patent, and the second job finds the top ten in that "inverted" data.

5.1.1 Chaining MapReduce jobs in a sequence

Though you can execute the two jobs manually one after the other, it's more convenient to automate the execution sequence. You can chain MapReduce jobs to run sequentially, with the output of one MapReduce job being the input to the next. Chaining MapReduce jobs is analogous to Unix pipes.

```
mapreduce-1 | mapreduce-2 | mapreduce-3 | ...
```

Chaining MapReduce jobs sequentially is quite straightforward. Recall that a driver sets up a `JobConf` object with the configuration parameters for a MapReduce job and passes the `JobConf` object to `JobClient.runJob()` to start the job. As `JobClient.runJob()` blocks until the end of a job, chaining MapReduce jobs involves calling the driver of one MapReduce job after another. The driver at each job will have to create a new `JobConf` object and set its input path to be the output path of the previous job. You can delete the intermediate data generated at each step of the chain at the end.

5.1.2 Chaining MapReduce jobs with complex dependency

Sometimes the subtasks of a complex data processing task don't run sequentially, and their MapReduce jobs are therefore not chained in a linear fashion. For example, mapreduce1 may process one data set while mapreduce2 independently processes another data set. The third job, mapreduce3, performs an inner join of the first two jobs' output. (We'll discuss data joining in the next sections.) It's dependent on the other two and can execute only after both mapreduce1 and mapreduce2 are completed. But mapreduce1 and mapreduce2 aren't dependent on each other.

Hadoop has a mechanism to simplify the management of such (nonlinear) job dependencies via the `Job` and `JobControl` classes. A `Job` object is a representation of a MapReduce job. You instantiate a `Job` object by passing a `JobConf` object to its constructor. In addition to holding job configuration information, `Job` also holds dependency information, specified through the `addDependingJob()` method. For `Job` objects `x` and `y`,

```
x.addDependingJob(y)
```

means x will not start until y has finished. Whereas Job objects store the configuration and dependency information, JobControl objects do the managing and monitoring of the job execution. You can add jobs to a JobControl object via the addJob() method. After adding all the jobs and dependencies, call JobControl's run() method to spawn a thread to submit and monitor jobs for execution. JobControl has methods like allFinished() and getFailedJobs() to track the execution of various jobs within the batch.

5.1.3 Chaining preprocessing and postprocessing steps

A lot of data processing tasks involve record-oriented preprocessing and postprocessing. For example, in processing documents for information retrieval, you may have one step to remove *stop words* (words like *a*, *the*, and *is* that occur frequently but aren't too meaningful), and another step for *stemming* (converting different forms of a word into the same form, such as *finishing* and *finished* into *finish.*) You can write a separate MapReduce job for each of these pre- and postprocessing steps and chain them together, using IdentityReducer (or no reducer at all) for these steps. This approach is inefficient as each step in the chain takes up I/O and storage to process the intermediate results. Another approach is for you to write your mapper such that it calls all the preprocessing steps beforehand and the reducer to call all the postprocessing steps afterward. This forces you to architect the pre- and postprocessing steps in a modular and composable manner. Hadoop introduced the ChainMapper and the ChainReducer classes in version 0.19.0 to simplify the composition of pre- and postprocessing.

You can think of chaining MapReduce jobs, as explained in section 5.1.1, symbolically using the pseudo-regular expression:

```
[MAP | REDUCE]+
```

where a reducer REDUCE comes after a mapper MAP, and this [MAP | REDUCE] sequence can repeat itself one or more times, one right after another. The analogous expression for a job using ChainMapper and ChainReducer would be

```
MAP+ | REDUCE | MAP*
```

The job runs multiple mappers in sequence to preprocess the data, and after running reduce it can optionally run multiple mappers in sequence to postprocess the data. The beauty of this mechanism is that you write the pre- and postprocessing steps as standard mappers. You can run each one of them individually if you want. (This is useful when you want to debug them individually.) You call the addMapper() method in ChainMapper and ChainReducer to compose the pre- and postprocessing steps, respectively. Running all the pre- and postprocessing steps in a single job leaves no intermediate file and there's a dramatic reduction in I/O.

Consider the example where there are four mappers (Map1, Map2, Map3, and Map4) and one reducer (Reduce), and they're chained into a single MapReduce job in this sequence:

```
Map1 | Map2 | Reduce | Map3 | Map4
```

In this setup, you should think of `Map2` and `Reduce` as the core of the MapReduce job, with the standard partitioning and shuffling applied between the mapper and reducer. You should consider `Map1` as a preprocessing step and `Map3` and `Map4` as postprocessing steps. The number of processing steps can vary. This is only an example.

You can specify the composition of this sequence of mappers and reducer with the driver. See listing 5.1. You need to make sure the key and value outputs of one task have matching types (classes) with the inputs of the next task.

Listing 5.1 Driver for chaining mappers within a MapReduce job

```
Configuration conf = getConf();
JobConf job = new JobConf(conf);

job.setJobName("ChainJob");
job.setInputFormat(TextInputFormat.class);
job.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(job, in);
FileOutputFormat.setOutputPath(job, out);

JobConf map1Conf = new JobConf(false);
ChainMapper.addMapper(job,
                      Map1.class,
                      LongWritable.class,
                      Text.class,                  Add Mapl step to job
                      Text.class,
                      Text.class,
                      true,
                      map1Conf);

JobConf map2Conf = new JobConf(false);
ChainMapper.addMapper(job,
                      Map2.class,
                      Text.class,
                      Text.class,                  Add Map2 step to job
                      LongWritable.class,
                      Text.class,
                      true,
                      map2Conf);

JobConf reduceConf = new JobConf(false);
ChainReducer.setReducer(job,
                        Reduce.class,
                        LongWritable.class,
                        Text.class,                Add Reduce step to job
                        Text.class,
                        Text.class,
                        true,
                        reduceConf);

JobConf map3Conf = new JobConf(false);
ChainReducer.addMapper(job,
                       Map3.class,
                       Text.class,                 Add Map3 step to job
                       Text.class,
                       LongWritable.class,
                       Text.class,
                       true,                      Add Map3 step to job
                       map3Conf);

JobConf map4Conf = new JobConf(false);
ChainReducer.addMapper(job,
                       Map4.class,
                       LongWritable.class,
                       Text.class,                Add Map4 step to job
                       LongWritable.class,
                       Text.class,
                       true,
                       map4Conf);

JobClient.runJob(job);
```

The driver first sets up the "global" `JobConf` object with the job's name, input path, output path, and so forth. It adds the five steps of the chained job one at a time, in the sequence of the steps' execution. It adds all the steps before `Reduce` using the static `ChainMapper.addMapper()` method. It sets the reducer with the static `ChainReducer.setReducer()` method. Using the `ChainReducer.addMapper()` method, it adds the last steps. The global `JobConf` object (`job`) is passed through all five `add*` methods. In addition, each mapper and the reducer have a local `JobConf` object (`map1Conf`, `map2Conf`, `map3Conf`, `map4Conf`, and `reduceConf`) that takes precedence over the global one in configuring the individual mapper/reducer. The recommended local `JobConf` object is a new `JobConf` object initiated without defaults — `new JobConf(false)`.

Let's look at the signature of the `ChainMapper.addMapper()` method to understand in detail how to add each step to the chained job. The signature and function of `ChainReducer.setReducer()` and `ChainReducer.addMapper()` are analogous and we'll skip them.

```
public static <K1,V1,K2,V2> void
                            addMapper(JobConf job,
                                      Class<? extends Mapper<K1,V1,K2,V2>> klass,
                                      Class<? extends K1> inputKeyClass,
                                      Class<? extends V1> inputValueClass,
                                      Class<? extends K2> outputKeyClass,
                                      Class<? extends V2> outputValueClass,
                                      boolean byValue,
                                      JobConf mapperConf)
```

This method has eight arguments. The first and last are the global and local `JobConf` objects, respectively. The second argument (`klass`) is the `Mapper` class that will do the data processing. The four arguments `inputValueClass`, `inputKeyClass`, `outputKeyClass`, and `outputValueClass` are the input/output class types of the `Mapper` class.

The argument `byValue` will need a little explanation. In the standard `Mapper` model, the output key/value pairs are serialized and written to disk,[1] prepared to be shuffled

[1] The key and value's ability to be cloned and serialized is provided by them being implemented as `Writables`.

to a reducer that may be at a completely different node. Formally this is considered to be *passed by value*, as a copy of the key/value pair is sent over. In the current case where we can chain one `Mapper` to another, we can execute the two in the same JVM thread. Therefore, it's possible for the key/value pairs to be *passed by reference*, where the output of the initial `Mapper` stays in place in memory and the following `Mapper` refers to it directly in the same memory location. When `Map1` calls `OutputCollector.collect (K k, V v)`, the objects `k` and `v` pass directly to `Map2`'s `map()` method. This improves performance by not having to clone a potentially large volume of data between the mappers. But doing this can violate one of the more subtle "contracts" in Hadoop's MapReduce API. The call to `OutputCollector.collect(K k, V v)` is guaranteed to not alter the content of `k` and `v`. `Map1` can call `OutputCollector.collect(K k, V v)` and then use the objects `k` and `v` afterward, fully expecting their values to stay the same. But if we pass those objects by reference to `Map2`, then `Map2` may alter them and violate the API's guarantee. If you're sure that `Map1`'s `map()` method doesn't use the content of `k` and `v` after calling `OutputCollector.collect(K k, V v)`, or that `Map2` doesn't change the value of its `k` and `v` input, you can achieve some performance gains by setting `byValue` to false. If you're not sure of the `Mapper`'s internal code, it's best to play safe and let `byValue` be true, maintaining the pass-by-value model, and be certain that the `Mappers` will work as expected.

5.2 Joining data from different sources

It's inevitable that you'll come across data analyses where you need to pull in data from different sources. For example, given our patent data sets, you may want to find out if certain countries cite patents from another country. You'll have to look at citation data (`cite75_99.txt`) as well as patent data for country information (`apat63_99.txt`). In the database world it would just be a matter of joining two tables, and most databases automagically take care of the join processing for you. Unfortunately, joining data in Hadoop is more involved, and there are several possible approaches with different trade-offs.

We use a couple toy data sets to better illustrate joining in Hadoop. Let's take a comma-separated Customers file where each record has three fields: Customer ID, Name, and Phone Number. We put four records in the file for illustration:

```
1,Stephanie Leung,555-555-5555
2,Edward Kim,123-456-7890
3,Jose Madriz,281-330-8004
4,David Stork,408-555-0000
```

We store Customer orders in a separate file, called Orders. It's in CSV format, with four fields: Customer ID, Order ID, Price, and Purchase Date.

```
3,A,12.95,02-Jun-2008
1,B,88.25,20-May-2008
```

```
2,C,32.00,30-Nov-2007
3,D,25.02,22-Jan-2009
```

If we want an inner join of the two data sets above, the desired output would look a listing 5.2.

Listing 5.2 Desired output of an inner join between Customers and Orders data

```
1,Stephanie Leung,555-555-5555,B,88.25,20-May-2008
2,Edward Kim,123-456-7890,C,32.00,30-Nov-2007
3,Jose Madriz,281-330-8004,A,12.95,02-Jun-2008
3,Jose Madriz,281-330-8004,D,25.02,22-Jan-2009
```

Hadoop can also perform outer joins, although to simplify explanation we focus on inner joins.

5.2.1 Reduce-side joining

Hadoop has a contrib package called *datajoin* that works as a generic framework for data joining in Hadoop. Its jar file is at contrib/datajoin/hadoop-*-datajoin.jar. To distinguish it from other joining techniques, it's called the *reduce-side join*, as we do most of the processing on the reduce side. It's also known as the *repartitioned join* (or the *repartitioned sort-merge join*), as it's the same as the database technique of the same name. Although it's not the most efficient joining technique, it's the most general and forms the basis of some more advanced techniques (such as the semijoin).

Reduce-side join introduces some new terminologies and concepts, namely, data source, tag, and group key. A *data source* is analogous to a table in relational databases. We have two data sources in our toy example: Customers and Orders. A data source can be a single file or multiple files. The important point is that all the records in a data source have the same structure, analogous to a schema.

The MapReduce paradigm calls for processing each record one at a time in a stateless manner. If we want some state information to persist, we have to *tag* the record with such state. For example, given our two files, a record may look to a mapper like this:

```
3,Jose Madriz,281-330-8004
```

or:

```
3,A,12.95,02-Jun-2008
```

where the record type (Customers or Orders) is dissociated from the record itself. Tagging the record will ensure that specific metadata will always go along with the record. For the purpose of data joining, we want to tag each record with its *data source*.

The *group key* functions like a join key in a relational database. For our example, the group key is the Customer ID. As the datajoin package allows the group key to be any user-defined function, group key is more general than a join key in a relational database.

Before explaining how to use the contrib package, let's go through all the major steps in a repartitioned sort-merge join of our toy datasets. After seeing how those steps fit together, we'll see which steps are done by the datajoin package, and which ones we program. We'll have code to see the hooks for integrating our code with the datajoin package.

DATA FLOW OF A REDUCE-SIDE JOIN

Figure 5.1 illustrates the data flow of a repartitioned join on the toy data sets Customers and Orders, up to the reduce stage. We'll go into more details later to see what happens in the reduce stage.

First we see that mappers receive data from two files, Customers and Orders. Each mapper knows the filename of the data stream it's processing. The `map()` function is called with each record, and the main goal of `map()` is to package each record such that joining on the reduce side is possible.

Recall that in the MapReduce framework, `map()` outputs records as key/value pairs that are partitioned on the key, and all records of the same key will end up in a single reducer and be processed together. For joining, we would want the `map()` function to output a record package where the key is the group key for joining—the Customer ID in this case. The value in this key/value package will be the original record, tagged with the data source (i.e., filename). For example, for the record

```
3,A,12.95,02-Jun-2008
```

from the Orders file, `map()` will output a key/value pair where the key is `"3"`, the Customer ID that will be used to join with records from the Customers file. The value output by `map()` is the entire record wrapped by a tag `"Orders"`.

After `map()` packages each record of the inputs, MapReduce's standard partition, shuffle, and sort takes place. Note that as the group key is set to the join key, `reduce()` will process all records of the same join key together. The function `reduce()` will unwrap the package to get the original record and the data source of the record by its tag. We see that for group keys (Customer IDs) `"1"` and `"2"`, the `reduce()` function gets two values. One value is tagged with `"Customers"` and the other value is tagged with `"Orders"`. For the map output with (group) key `"4"`, `reduce()` will only see one value, which is tagged with `"Customers"`. This is expected as there is no record in Orders with a Customer ID of `"4"`. On the other hand, `reduce()` will see three values for the (group) key `"3"`. This is due to one record from Customers and two more from Orders.

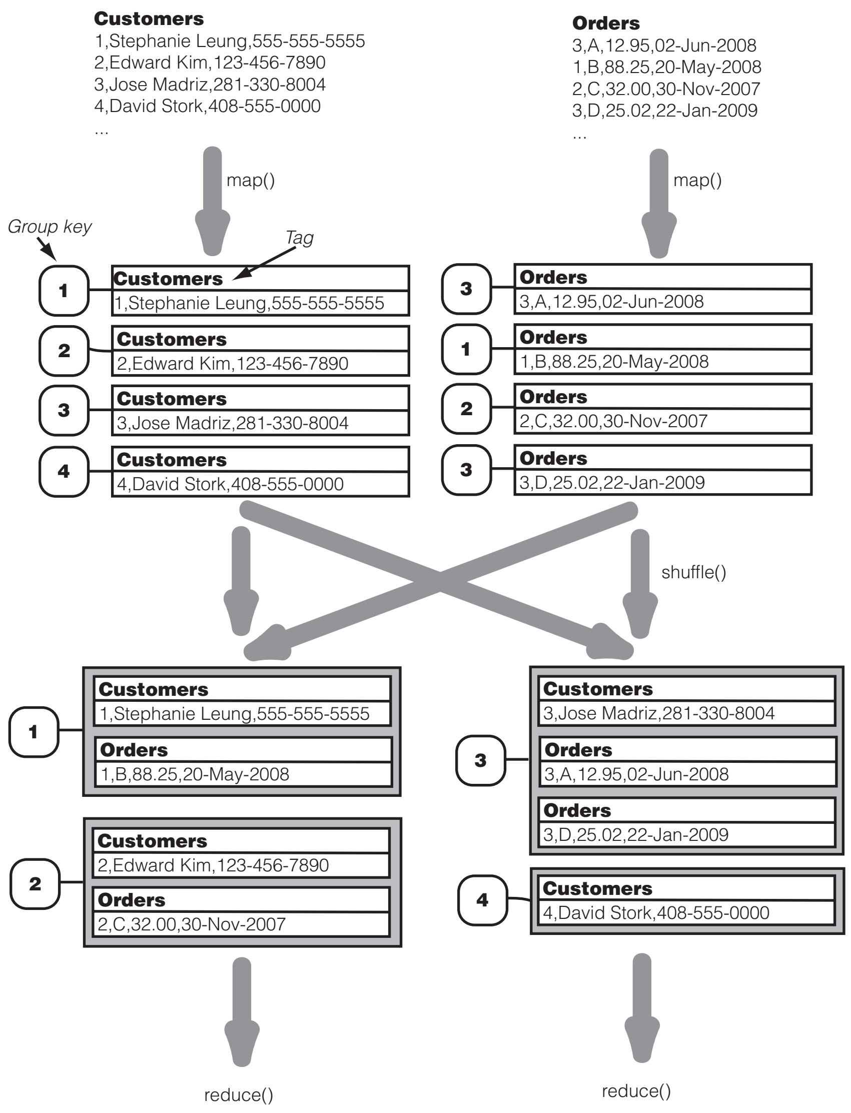
Customers
1,Stephanie Leung,555-555-5555
2,Edward Kim,123-456-7890
3,Jose Madriz,281-330-8004
4,David Stork,408-555-0000
...
Orders
3,A,12.95,02-Jun-2008
1,B,88.25,20-May-2008
2,C,32.00,30-Nov-2007
3,D,25.02,22-Jan-2009
...
map()
map()
Group key
Tag
1 Customers 1,Stephanie Leung,555-555-5555
2 Customers 2,Edward Kim,123-456-7890
3 Customers 3,Jose Madriz,281-330-8004
4 Customers 4,David Stork,408-555-0000
3 Orders 3,A,12.95,02-Jun-2008
1 Orders 1,B,88.25,20-May-2008
2 Orders 2,C,32.00,30-Nov-2007
3 Orders 3,D,25.02,22-Jan-2009
shuffle()
1 Customers 1,Stephanie Leung,555-555-5555 Orders 1,B,88.25,20-May-2008
2 Customers 2,Edward Kim,123-456-7890 Orders 2,C,32.00,30-Nov-2007
3 Customers 3,Jose Madriz,281-330-8004 Orders 3,A,12.95,02-Jun-2008 Orders 3,D,25.02,22-Jan-2009
4 Customers 4,David Stork,408-555-0000
reduce()
reduce()

Figure 5.1 In repartitioned join, the mapper first wraps each record with a group key and a tag. The group key is the joining attribute, and the tag is the data source (*table* in SQL parlance) of the record. The partition and shuffle step will group all the records with the same group key together. The reducer is called on the set of records with the same group key.

The function `reduce()` will take its input and do a full *cross-product* on the values. `Reduce()` creates all combinations of the values with the constraint that a combination will not be tagged more than once. In cases where `reduce()` sees values of distinct tags, the cross-product is the original set of values. In our example, this is the case for group

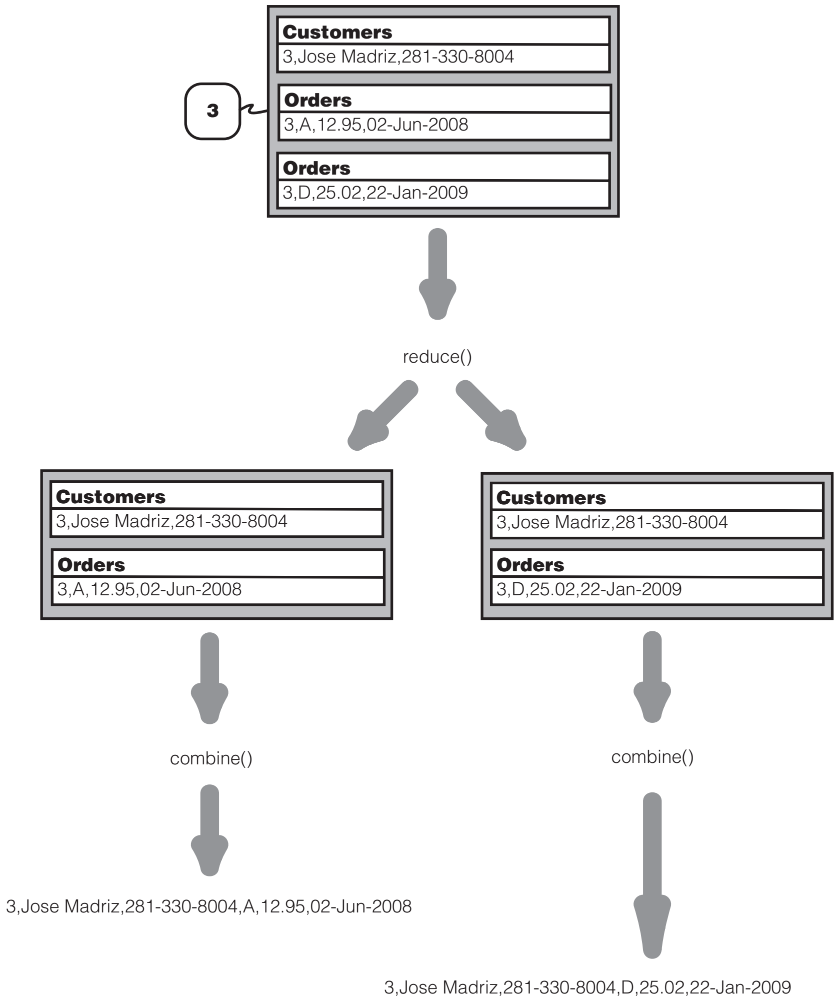
Customers
3,Jose Madriz,281-330-8004
3
Orders
3,A,12.95,02-Jun-2008
Orders
3,D,25.02,22-Jan-2009
reduce()
Customers
3,Jose Madriz,281-330-8004
Orders
3,A,12.95,02-Jun-2008
Customers
3,Jose Madriz,281-330-8004
Orders
3,D,25.02,22-Jan-2009
combine()
combine()
3,Jose Madriz,281-330-8004,A,12.95,02-Jun-2008
3,Jose Madriz,281-330-8004,D,25.02,22-Jan-2009

Figure 5.2 The reduce side of a repartitioned join. For a given join key, the reduce task performs a full cross-product of values from different sources. It sends each combination to *combine()* to create an output record. The *combine()* function can choose to not output any particular combination.

keys `1`, `2`, and `4`. Figure 5.2 illustrates the cross product for group key `3`. We have three values, one tagged with Customers and two tagged with Orders. The cross-product creates two combinations. Each combination consists of the Customers value and one of the Orders value.

NOTE Our toy example has an implicit schema that Customer ID identifies a unique record in Customers, making the number of combinations in a cross-product always the number of Orders records with the Customer ID (except when it's zero, in which case the cross-product is the Customers record itself). For more complicated settings, the number of combinations generated by the cross-product is the product of the number of records under each tag. If `reduce()` sees two Customers records and three Orders records together, then the cross-product will create six (2 * 3) combinations. If there's a third data source (Accounts) with two records, then the cross-product will create twelve (2 * 2 * 3) combinations.

It feeds each combination from the cross-product into a function called `combine()`. (Don't confuse with combiners as explained in section 4.5.) Due to the nature of the cross-product, `combine()` is guaranteed to see at most one record from each of the data sources (tags), and all the records it sees have the same join key. It's the `combine()` function that determines whether the whole operation is an inner join, outer join, or another type of join. In an inner join, `combine()` drops all combinations where not all tags are present, such as our case with group key `"4"`. Otherwise `combine()` merges the records from different sources into a single output record.

Now you see why we call this joining process the repartitioned sort-merge join. The records in the original input sources can be in random order. They are *repartitioned* onto the reducers in the right grouping. The reducer can then *merge* records of the same join key together to create the desired join output. (The *sort* happens but it's not critical to understanding the operation.)

IMPLEMENTING JOIN WITH THE DATAJOIN PACKAGE

Hadoop's datajoin package implements the dataflow of a join as described previously. We have certain hooks to handle the details of our particular data structure and a special hook for us to define the exact function of `combine()`.

Hadoop's datajoin package has three abstract classes that we inherit and make concrete: `DataJoinMapperBase`, `DataJoinReducerBase`, and `TaggedMapOutput`. As the names suggest, our `MapClass` will extend `DataJoinMapperBase`, and our `Reduce` class will extend `DataJoinReducerBase`. The datajoin package has already implemented the `map()` and `reduce()` methods in these respective base classes to perform the join dataflow describe in the last section. Our subclass will only have to implement a few new methods to configure the details.

Before explaining how to use `DataJoinMapperBase` and `DataJoinReducerBase`, you need to understand a new abstract data type `TaggedMapOutput` that is used throughout the code. Recall from the dataflow description that the mapper outputs a package with a (group) key and a value that is a tagged record. The datajoin package specifies the (group) key to be of type `Text` and the value (i.e., the tagged record) to be of type `TaggedMapOutput`. `TaggedMapOutput` is a data type for wrapping our records with a `Text` tag. It trivially implements a `getTag()` and a `setTag(Text tag)` method. It specifies an abstract method `getData()`. Our subclass will implement that method to handle the type of the record. There's no explicit requirement for the subclass to have a `setData()` method but we must pass in the record data. The subclass can implement such a `setData()` method for the sake of symmetry or take in a record in the constructor. In addition, as the output of a mapper, `TaggedMapOutput` needs to be `Writable`. Therefore, our subclass has to implement the `readFields()` and `write()` methods. We created `TaggedWritable`, a simple subclass for handling any `Writable` record type.

```
public static class TaggedWritable extends TaggedMapOutput {

    private Writable data;

    public TaggedWritable(Writable data) {
        this.tag = new Text("");
        this.data = data;
    }

    public Writable getData() {
        return data;
    }

    ...
}
```

Recall from the join dataflow that the mapper's main function is to package a record such that it goes to the same reducer as other records with the same join key. `DataJoinMapperBase` performs all the packaging, but the class specifies three abstract methods for our subclass to fill in:

```
protected abstract Text generateInputTag(String inputFile);
protected abstract TaggedMapOutput generateTaggedMapOutput(Object value);
protected abstract Text generateGroupKey(TaggedMapOutput aRecord);
```

The `generateInputTag()` is called at the start of a map task to globally specify the tag for all the records this map task will process. The tag is defined to be of type `Text`. Note that we call the `generateInputTag()`with the filename of the records. The mapper working on the Customers file will receive the string `"Customers"` as the argument to `generateInputTag()`. As we're using the tag to signify the data source, and our filename is set up to reflect the data source, `generateInputTag()` is

```
protected Text generateInputTag(String inputFile) {
    return new Text(inputFile);
}
```

If a data source is spread out over several files (part-0000, part-0001, etc.), you would not want the tag to be the complete filename, rather some prefix of it. For example, the tag (data source) can be the filename before the dash (-) sign.

```
protected Text generateInputTag(String inputFile) {
    String datasource = inputFile.split('-')[0];
    return new Text(datasource);
}
```

We store the result of `generateInputTag()`in the `DataJoinMapperBase` object's `inputTag` variable for later use. We can also store the filename in `DataJoinMapperBase`'s `inputFile` variable if we want to refer to it again.

After the map task's initialization, `DataJoinMapperBase`'s `map()` is called for each record. It calls the two abstract methods that we have yet to implement.

```
public void map(Object key, Object value,
                OutputCollector output, Reporter reporter) throws IOException
{
    TaggedMapOutput aRecord = generateTaggedMapOutput(value);
    Text groupKey = generateGroupKey(aRecord);
    output.collect(groupKey, aRecord);
}
```

The `generateTaggedMapOutput()` method wraps the record value into a `TaggedMapOutput` type. Recall the concrete implementation of `TaggedMapOutput` that we're using is called `TaggedWritable`. The method `generateTaggedMapOutput()` can return a `TaggedWritable` with any `Text` tag that we want. In principle, the tag can even be different for different records in the same file. In the standard case, we want the tag to stand for the data source that our `generateInputTag()` had computed earlier and stored in `this.inputTag`.

```
protected TaggedMapOutput generateTaggedMapOutput(Object value) {
    TaggedWritable retv = new TaggedWritable((Text) value);
    retv.setTag(this.inputTag);
    return retv;
}
```

The `generateGroupKey()` method takes a tagged record (of type `TaggedMapOutput`) and returns the group key for joining. For our current purpose, we unwrap the tagged record and take the first field in the CSV-formatted value as the join key.

```
protected Text generateGroupKey(TaggedMapOutput aRecord) {
    String line = ((Text) aRecord.getData()).toString();
    String[] tokens = line.split(",");
    String groupKey = tokens[0];
    return new Text(groupKey);
}
```

In a more general implementation, the user will be able to specify which field should be the joining key and if the record separator may be some character other than a comma.

`DataJoinMapperBase` is a simple class, and much of the mapper code is in our subclass. `DataJoinReducerBase`, on the other hand, is the workhorse of the datajoin package, and it simplifies our programming by performing a full outer join for us. Our reducer subclass only has to implement the `combine()` method to filter out unwanted combinations to get the desired join operation (inner join, left outer join, etc.). It's also in the `combine()` method that we format the combination into the appropriate output format.

We give the `combine()` method *one combination of the cross product of the tagged records with the same join (group) key*. This may sound complicated, but recall from the dataflow diagrams in figures 5.1 and 5.2 that the cross-product is simple for the canonical case of two data sources. Each combination will have either two records (meaning there's at least one record in each data source with the join key) or one (meaning only one data source has that join key).

Let's look at the signature of `combine()`:

```
protected abstract TaggedMapOutput
                                ➥ combine(Object[] tags, Object[] values);
```

An array of tags and an array of values represent the combination. The size of those two arrays is guaranteed to be the same and equal to the number of tagged records in the combination. The first tagged record in the combination is represented by `tags[0]` and `values[0]`, the second one is `tags[1]` and `values[1]`, and so forth. Furthermore, the tags are always in sorted order.

As tags correspond to the data sources, in the canonical case of joining two data sources, the `tags` array to `combine()` won't be longer than two. Figure 5.2 shows `combine()` being called twice. For the left side, the `tags` and `values` arrays are like this:[2]

```
tags = {"Customers", "Orders"};
values = {"3,Jose Madriz,281-330-8004", "A,12.95,02-Jun-2008"};
```

For an inner join, `combine()` will ignore combinations where not all tags are present. It does so by returning null. Given a legal combination, the role of `combine()` is to concatenate all the values into one single record for output. The order of concatenation is fully determined by `combine()`. In the case of an inner join, the length of `values[]` is always the number of data sources available (two in the canonical case), and the tags are always in sorted order. It's a sensible choice to loop through the `values[]` array to get the default alphabetical ordering based on data source names.

`DataJoinReducerBase`, like any reducer, outputs key/value pairs. For each legal combination, the key is always the join key and the value is the output of `combine()`. Note that the join key is still present in each element of the `values[]` array. The `combine()` method should strip out the join key in those elements before concatenating them. Otherwise the join key will be shown multiple times in one output record.

Finally, `DataJoinReducerBase` expects `combine()` to return a `TaggedMapOutput`. It's unclear why as `DataJoinReducerBase` ignores the tag in the `TaggedMapOutput` object.

Listing 5.3 shows the complete code, including our reduce subclass.

Listing 5.3 Inner join of data from two files using reduce-side join

```
public class DataJoin extends Configured implements Tool {

    public static class MapClass extends DataJoinMapperBase {

        protected Text generateInputTag(String inputFile) {
            String datasource = inputFile.split("-")[0];
            return new Text(datasource);
        }

        protected Text generateGroupKey(TaggedMapOutput aRecord) {
            String line = ((Text) aRecord.getData()).toString();
            String[] tokens = line.split(",");
            String groupKey = tokens[0];
            return new Text(groupKey);
```

[2] The `tags` array is of type `Text[]` and `values` is of type `TaggedWritable[]`. We ignore those details to focus on the their contents.

```
        }

        protected TaggedMapOutput generateTaggedMapOutput(Object value) {
            TaggedWritable retv = new TaggedWritable((Text) value);
            retv.setTag(this.inputTag);
            return retv;
        }
    }

    public static class Reduce extends DataJoinReducerBase {

        protected TaggedMapOutput combine(Object[] tags, Object[] values) {
            if (tags.length < 2) return null;
            String joinedStr = "";
            for (int i=0; i<values.length; i++) {
                if (i > 0) joinedStr += ",";
                TaggedWritable tw = (TaggedWritable) values[i];
                String line = ((Text) tw.getData()).toString();
                String[] tokens = line.split(",", 2);
                joinedStr += tokens[1];
            }
            TaggedWritable retv = new TaggedWritable(new Text(joinedStr));
            retv.setTag((Text) tags[0]);
            return retv;
        }
    }

    public static class TaggedWritable extends TaggedMapOutput {

        private Writable data;

        public TaggedWritable(Writable data) {
            this.tag = new Text("");
            this.data = data;
        }

        public Writable getData() {
            return data;
        }

        public void write(DataOutput out) throws IOException {
            this.tag.write(out);
            this.data.write(out);
        }

        public void readFields(DataInput in) throws IOException {
            this.tag.readFields(in);
            this.data.readFields(in);
        }
    }

    public int run(String[] args) throws Exception {
        Configuration conf = getConf();

        JobConf job = new JobConf(conf, DataJoin.class);

        Path in = new Path(args[0]);
        Path out = new Path(args[1]);
        FileInputFormat.setInputPaths(job, in);
        FileOutputFormat.setOutputPath(job, out);
        job.setJobName("DataJoin");
        job.setMapperClass(MapClass.class);
        job.setReducerClass(Reduce.class);

        job.setInputFormat(TextInputFormat.class);
        job.setOutputFormat(TextOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(TaggedWritable.class);
        job.set("mapred.textoutputformat.separator", ",");

        JobClient.runJob(job);
        return 0;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(),
                                 new DataJoin(),
                                 args);

        System.exit(res);
    }
}
```

Next we'll look at another way of doing joins that is more efficient in some common applications.

5.2.2 Replicated joins using DistributedCache

The reduce-side join technique discussed in the last section is flexible, but it can also be quite inefficient. Joining doesn't take place until the reduce phase. We shuffle all data across the network first, and in many situations we drop the majority of this data during the joining process. It would be more efficient if we eliminate the unnecessary data right in the map phase. Even better would be to perform the entire joining operation in the map phase.

The main obstacle to performing joins in the map phase is that a record being processed by a mapper may be joined with a record not easily accessible (or even located) by that mapper. If we can guarantee the accessibility of all the necessary data when joining a record, joining on the map side can work. For example, if we know that the two sources of data are partitioned into the same number of partitions *and* the partitions are all sorted on the key *and* the key is the desired join key, then each mapper (with the proper `InputFormat` and `RecordReader`) can deterministically locate and retrieve all the data necessary to perform joining. In fact, Hadoop's org.apache.hadoop.mapred.join package contains helper classes to facilitate this map-side join. Unfortunately, situations where we can naturally apply this are limited, and running extra MapReduce jobs to repartition the data sources to be usable by this package seems to defeat the efficiency gain. Therefore, we'll not pursue this package further.

All hope is not lost though. There's another data pattern that occurs quite frequently that we can take advantage of. When joining big data, often only one of the sources is big; the second source may be orders of magnitude smaller. For example, a local phone company's Customers data may have only tens of millions of records (each record containing basic information for one customer), but its transaction log can have billions of records containing detailed call history. When the smaller source can fit in memory of a mapper, we can achieve a tremendous gain in efficiency by copying the smaller source to all mappers and performing joining in the map phase. This is called *replicated join* in the database literature as one of the data tables is replicated across all nodes in the cluster. (The next section will cover the case when the smaller source doesn't fit in memory.)

Hadoop has a mechanism called *distributed cache* that's designed to distribute files to all nodes in a cluster. It's normally used for distributing files containing "background" data needed by all mappers. For example, if you're using Hadoop to classify documents, you may have a list of keywords for each class. (Or better yet, a probabilistic model for each class, but we digress…) You would use distributed cache to ensure all mappers have access to the lists of keywords, the "background" data. For executing replicated joins, we consider the smaller data source as background data.

Distributed cache is handled by the appropriately named class `DistributedCache`. There are two steps to using this class. First, when configuring a job, you call the static method `DistributedCache.addCacheFile()` to specify the files to be disseminated to all nodes. These files are specified as URI objects, and they default to HDFS unless a different filesystem is specified. The JobTracker will take this list of URIs and create a local copy of the files in all the TaskTrackers when it starts the job. In the second step, your mappers on each individual TaskTracker will call the static method `DistributedCache.getLocalCacheFiles()` to get an array of local file Paths where the local copy is located. At this point the mapper can use standard Java file I/O techniques to read the local copy.

Replicated joins using `DistributedCache` are simpler than reduce-side joins. Let's begin with our standard Hadoop template.

```
public class DataJoinDC extends Configured implements Tool {

    public static class MapClass extends MapReduceBase
        implements Mapper<Text, Text, Text, Text> {

        ...
    }

    public int run(String[] args) throws Exception {
        ...
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(),
                                 new DataJoinDC(),
                                 args);

        System.exit(res);
    }
}
```

Note that we've taken out the Reduce class. We plan on performing the joining in the map phase and will configure this job to have no reducers. You'll find our driver method familiar too.

```
public int run(String[] args) throws Exception {
   Configuration conf = getConf();
   JobConf job = new JobConf(conf, DataJoinDC.class);

   DistributedCache.addCacheFile(new Path(args[0]).toUri(), conf);

   Path in = new Path(args[1]);
   Path out = new Path(args[2]);
   FileInputFormat.setInputPaths(job, in);
   FileOutputFormat.setOutputPath(job, out);

   job.setJobName("DataJoin with DistributedCache");
   job.setMapperClass(MapClass.class);
   job.setNumReduceTasks(0);

   job.setInputFormat(KeyValueTextInputFormat.class);
   job.setOutputFormat(TextOutputFormat.class);
   job.set("key.value.separator.in.input.line", ",");

   JobClient.runJob(job);

   return 0;
}
```

The crucial addition here is where we take the file specified by the first argument and add it to DistributedCache. When we run the job, each node will create a local copy of that file from HDFS. The second and third arguments denote the input and output paths of the standard Hadoop job. Note that we've limited the number of data sources to two. This is not an inherent limitation of the technique, but a simplification that makes our code easier to follow.

Up to now our MapClass has only had to define one method, map(). In fact, the Mapper interface (and also the Reducer interface) has two more abstract methods, configure() and close(). We call the method configure() when we first instantiate the MapClass, and the method close() when the mapper finishes processing its split. The MapReduceBase class provides default no-op implementations for these methods. Here we want to override configure() to load our join data into memory when a mapper is first initialized. This way we can have the data available each time we call map() to process a new record.

```
public static class MapClass extends MapReduceBase
    implements Mapper<Text, Text, Text, Text> {

    private Hashtable<String, String> joinData =
                                 new Hashtable<String, String>();

    @Override
    public void configure(JobConf conf) {
        try {
            Path [] cacheFiles = DistributedCache.getLocalCacheFiles(conf);
            if (cacheFiles != null && cacheFiles.length > 0) {
                String line;
                String[] tokens;

                BufferedReader joinReader = new BufferedReader(
                                new FileReader(cacheFiles[0].toString()));
                try {
                    while ((line = joinReader.readLine()) != null) {
                        tokens = line.split(",", 2);
                        joinData.put(tokens[0], tokens[1]);
                    }
                } finally {
                    joinReader.close();
                }
            }
        } catch (IOException e) {
            System.err.println("Exception reading DistributedCache: " + e);
        }
    }

    public void map(Text key, Text value,
                    OutputCollector<Text, Text> output,
                    Reporter reporter) throws IOException {

        String joinValue = joinData.get(key);
        if (joinValue != null) {
            output.collect(key,
                           new Text(value.toString() + "," + joinValue));
        }
    }
}
```

When we call `configure()`, we get an array of file paths to the local copy of files pushed by `DistributedCache`. As our driver method has only pushed one file (given by our first argument) into `DistributedCache`, this should be an array of size one. We read that file using standard Java file I/O. For our purpose, the program assumes each line is a record, the key/value pair is comma separated, and the key is unique and will be used for joining. The program reads this source file into a Java `Hashtable` called `joinData` that's available throughout the mapper's lifespan.

The joining takes place in the `map()` method and is straightforward now that one of the sources resides in memory in the form of `joinData`. If we don't find the join key in `joinData`, we drop the record. Otherwise, we match the (join) key to the value in `joinData` and concatenate the values. The result is outputted directly into HDFS as we don't have any reducer for further processing.

A not-infrequent situation in using `DistributedCache` is that the background data (the smaller data source in our data join case) is in the local filesystem of the client rather than stored in HDFS. One way to handle this is to add code to upload the local file on the client to HDFS before calling `DistributedCache.addCacheFile()`. Fortunately, this process is natively supported as one of the generic Hadoop command line arguments in `GenericOptionsParser`. The option is `-files` and it automatically copies a comma-separated list of files to all the task nodes. Our command line statement is

```
bin/hadoop jar -files small_in.txt DataJoinDC.jar big_in.txt output
```

Now that we don't need to call `DistributedCache.addCacheFile()` ourselves anymore, we no longer have to take in the filename of the smaller data source as one of the arguments. The index to the arguments has shifted.

```
Path in = new Path(args[0]);
Path out = new Path(args[1]);
```

With these minor changes our `DistributedCache` join program will take a local file on the client machine as one of the input sources.

5.2.3 Semijoin: reduce-side join with map-side filtering

One of the limitations in using replicated join is that one of the join tables has to be small enough to fit in memory. Even with the usual asymmetry of size in the input sources, the smaller one may still not be small enough. You can solve this problem by rearranging the processing steps to make them more efficient. For example, if you're looking for the order history of all customers in the 415 area code, it's correct but inefficient to join the Orders and the Customers tables first before filtering out records where the customer is in the 415 area code. Both the Orders and Customers tables may be too big for replicated join and you'll have to resort to the inefficient reduce-side join. A better approach is to first filter out customers living in the 415 area code. We store this in a temporary file called Customers415. We can arrive at the same end result by joining Orders with Customers415, but now Customers415 is small enough that a replicated join is feasible. There is some overhead in creating and distributing the Customers415 file, but it's often compensated by the overall gain in efficiency.

Sometimes you may have a lot of data to analyze. You can't use replicated join no matter how you rearrange your processing steps. Don't worry. We still have ways to make reduce-side joining more efficient. Recall that the main problem with reduce-side joining is that the mapper only tags the data, all of which is shuffled across the network but most of which is ignored in the reducer. The inefficiency is ameliorated if the mapper has an extra prefiltering function to eliminate most or even all the unnecessary data before it is shuffled across the network. We need to build this filtering mechanism.

Continuing our example of joining Customers415 with Orders, the join key is Customer ID and we would like our mappers to filter out any customer not from the 415 area code rather than send those records to reducers. We create a data set CustomerID415 to store all the Customer IDs of customers in the 415 area code. CustomerID415 is smaller than Customers415 because it only has one data field. Assuming CustomerID415 can now fit in memory, we can improve reduce-side join by using distributed cache to disseminate CustomerID415 across all the mappers. When processing records from Customers and Orders, the mapper will drop any record whose key is not in the set CustomerID415. This is sometimes called a *semijoin*, taking the terminology from the database world.

Last but not least, what if the file CustomerID415 is still too big to fit in memory? Or maybe CustomerID415 does fit in memory but it's size makes replicating it across all the mappers inefficient. This situation calls for a data structure called a Bloom filter. A Bloom filter is a compact representation of a set that supports only the *contain* query. ("Does this set contain this element?") Furthermore, the query answer is not completely accurate, but it's guaranteed to have no false negatives and a small probability of false positives. The slight inaccuracy is the trade-off for the data structure's compactness. By using a Bloom filter representation of CustomerID415, the mappers will pass through all customers in the 415 area code. It still guarantees the correctness of the data join algorithm. The Bloom filter will also pass a small portion of customers not in the 415 area code to the reduce phase. This is fine because those will be ignored in the reduce phase. We'll still have improved performance by reducing dramatically the amount of traffic shuffled across the network. The use of Bloom filters is in fact a standard technique for joining in distributed databases, and it's used in commercial products such as Oracle 11g. We'll describe Bloom filter and its other applications in more details in the next section.

5.3 Creating a Bloom filter

If you use Hadoop for batch processing of large data sets, your data-intensive computing needs probably include transaction-style processing as well. We won't cover all the techniques for running real-time distributed data processing (caching, sharding, etc.). They aren't necessarily Hadoop-related and are well beyond the scope of this book. One lesser-known tool for real-time data processing is the Bloom filter, which is a summary of a data set whose usage makes other data processing techniques more efficient. When that data set is big, Hadoop is often called in to generate the Bloom filter representation. As we mentioned earlier, a Bloom filter is also sometimes used for data joining within Hadoop itself. As a data processing expert, you'll be well rewarded to have the Bloom filter in your bag of tricks. In this section we'll explain this data structure in more detail and we'll go through an online ad network example that will build a Bloom filter using Hadoop.

5.3.1 What does a Bloom filter do?

At its most basic, a Bloom filter object supports two methods: `add()` and `contains()`. These two methods work in a similar way as in the Java `Set` interface. The method `add()` adds an object to the set, and the method `contains()` returns a Boolean true/false value denoting whether an object is in the set or not. But, for a Bloom filter, `contains()` doesn't always give an accurate answer. It has no *false negatives*. If `contains()` returns false, you can be sure that the set doesn't have the object queried. It does have a small probability of *false positives* though. `contains()` can return true for some objects not in the set. The probability of false positives depends on the number of elements in the set and some configuration parameters of the Bloom filter itself.

The major benefit of a Bloom filter is that its size, in number of bits, is constant and is set upon initialization. Adding more elements to a Bloom filter doesn't increase its size. It only increases the false positive rate. A Bloom filter also has another configuration parameter to denote the number of hash functions it uses. We'll discuss the reason for this parameter and how the hash functions are used later when we discuss the Bloom filter's implementation. For now, its main implication is that it affects the false positive rate. The false positive rate is approximated by the equation

$$(1 - \exp(-kn/m))k$$

where *k* is the number of hash functions used, *m* is the number of bits used to store the Bloom filter, and *n* is the number of elements to be added to the Bloom filter. In practice, *m* and *n* are determined by the requirement of the system, and therefore, *k* is chosen to minimize the false positive rate given *m* and *n*, which (after a little calculus) is

$$k = \ln(2) * (m/n) \approx 0.7 * (m/n)$$

The false positive rate with the given *k* is 0.6185*m/n*, and *k* has to be an integer. The false positive rate will only be an approximation. From a design point of view, one should think in terms of (*m/n*), number of bits per element, rather than *m* alone. For example, we have to store a set containing ten million URLs (*n*=10,000,000). Allocating 8 bits per URL (*m*/*n*=8) will require a 10 MB Bloom filter (*m* = 80,000,000 bits). This Bloom filter will have a false positive rate of (0.6185)8, or about 2 percent. If we were to implement the `Set` class by storing the raw URLs, and let's say the average URL length was 100 bytes, we would have to use 1 GB. Bloom filter has shrunk the storage requirement by 2 orders of magnitude at the expense of only a 2 percent false positive rate! A slight increase in storage allocated to the Bloom filter will reduce the false positive rate further. At 10 bits per URL, the Bloom filter will take up 12.5 MB and have a false positive rate of only 0.8 percent.

In summary, the signature of our Bloom filter class will look like the following:

```
class BloomFilter<E> {
    public BloomFilter(int m, int k) { ... }
    public void add(E obj) { ... }
    public boolean contains(E obj) { ... }
}
```

More applications of the Bloom filter

The Bloom filter found its first successful applications back when memory was scarce. One of its first uses was in spellchecking. Not being able to store a whole dictionary in memory, spellcheckers used a Bloom filter representation (of the dictionary) to catch most misspellings. As memory size grew and became cheaper, such space consideration waned. Bloom filter usage is finding a resurgence in large-scale data-intensive operations as data is fast outgrowing memory and bandwidth.

We've already seen commercial products, such as Oracle 11g, using Bloom filters to join data across distributed databases. In the networking world, one successful

(*continued*)

product using Bloom filters is the open source distributed web proxy called Squid. Squid caches frequently accessed web content to save bandwidth and give users a faster web experience. In a cluster of Squid servers, each one can cache a different set of content. An incoming request should be routed to the Squid server holding a copy of the requested content, or in case of a cache miss, the request is passed on to the originating server. The routing mechanism needs to know what each of the Squid servers contains. As sending a list of URLs for each Squid server and storing it in memory is expensive, Bloom filters are used. A false positive means a request is forwarded to the wrong Squid server, but that server would recognize it as a cache miss and pass it on to the originating server, ensuring the correctness of the overall operation. The small performance hit from a false positive is far outweighed by the overall improvement.

Sharding systems are a similar application but more advanced. In a nutshell, database sharding is the partitioning of a database across multiple machines such that each machine only has to deal with a subset of records. Each record has some ID that determines which machine it's assigned to. In more basic designs, the ID is hashed statically to one of a fixed number of database machines. This approach is inflexible to adding more shards or rebalancing existing ones. To add flexibility, it uses a dynamic look-up for each record ID, but unfortunately that adds processing delay if the look-up is done through a database (i.e., using disk). Like Squid, more advanced shard systems use in-memory Bloom filters as a fast look-up. It needs some mechanism to handle false positives, but the occurrence is small enough to not impact the overall performance improvement.

For online display ad networks, it's important to be able to target an ad from the right category to a visitor. Given the volume of traffic a typical ad network receives and the latency requirements, one can end up spending a lot of money on hardware to have the capability of retrieving the category in real time. A design based on Bloom filters can dramatically decrease that cost. Use an offline process to tag web pages (or visitors) on a limited number of categories (sports, family-oriented, music, etc.). Build a Bloom filter for each category and store it in memory at the ad servers. When an ad request arrives, the ad servers can quickly and cheaply determine which category of ads to show. The amount of false positives is negligible.

5.3.2 Implementing a Bloom filter

Conceptually the implementation of a Bloom filter is quite straightforward. We describe its implementation in a single system first before implementing it using Hadoop in a distributed way. The internal representation of a Bloom filter is a bit array of size m. We have k independent hash functions, where each hash function takes an object as input and outputs an integer between 0 and m-1. We use the integer output as an index into the bit array. When we "add" an element to the Bloom filter, we use the hash functions to generate k indexes into the bit array. We set the k bits to 1. Figure 5.3 shows what happens when we add several objects (`x`, `y`, and `z`) over time, in a Bloom filter that uses three hash functions. Note that a bit will be set to 1 regardless of its previous state. The number of 1s in the bit array can only grow.

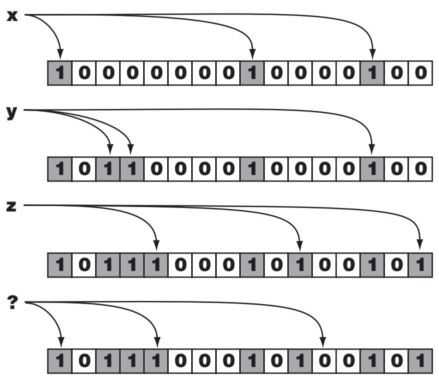

Figure 5.3 A Bloom filter is a bit array that represents a set with some probability of false positives. Objects (such as x, y, and z) are deterministically hashed into positions in the array, and those bits are set to 1. You can check whether an object is in the set or not by hashing and checking the values of those bit positions.

When an object comes in and we want to check whether it has been previously added to the Bloom filter, we use the same *k* hash functions to generate the bit array indexes as we would do in adding the object. Now we check whether *all* those *k* bits in the bit array are 1s. If all *k* bits are 1, we return true and claim that the Bloom filter contains the object. Otherwise we return false. We see that if the object has in fact been added before, then the Bloom filter will necessarily return true. There are no false negatives (returning false when the object is truly in the set). The *k* bits corresponding to the queried object can all be set to 1 even though the object has never been added to the set. It may happen that adding other objects set those bits leading to false positives.[3]

Our implementation of a Bloom filter in Java would use the Java `BitSet` class as its internal representation. We have a function `getHashIndexes(obj)` that takes an object and returns an integer array of size *k*, containing indexes into the `BitSet`. The main functions of the Bloom filter, `add()` and `contains()`, are quite straightforward:

```
class BloomFilter<E> {

    private BitSet bf;

    public void add(E obj) {
        int[] indexes = getHashIndexes(obj);

        for (int index : indexes) {
            bf.set(index);
        }
    }

    public boolean contains(E obj) {
        int[] indexes = getHashIndexes(obj);

        for (int index : indexes) {
            if (bf.get(index) == false) {
                return false;
            }
        }
```

[3] For an accessible introduction to Bloom Filters, see http://en.wikipedia.org/wiki/Bloom_filter.

```
        return true;
    }

    protected int[] getHashIndexes(E obj) { ... }
}
```

To implement `getHashIndexes()` such that it works truly as *k* independent hash functions is nontrivial. Instead, in our Bloom filter implementation in listing 5.4, we use a hack to generate *k* indexes that are roughly independent and uniformly distributed. The `getHashIndexes()` method seeds the Java `Random` number generator with an MD5 hash of the object and then takes *k* "random" numbers as indexes. The Bloom filter class would benefit from a more rigorous implementation of `getHashIndexes()`, but our hack suffices for illustration purposes.

An ingenious way of creating a Bloom filter for the union of two sets is by OR'ing the (bit array of the) Bloom filters of each individual set. As adding an object is setting certain bits in a bit array to 1, it's easy to see why this union rule is true:

```
public void union(BloomFilter<E> other) {
    bf.or(other.bf);
}
```

We'll be exploiting this union trick to build Bloom filters in a distributed fashion. Each mapper will build a Bloom filter based on its own data split. We'll send the Bloom filters to a single reducer, which will take a union of them and record the final output.

As the Bloom filter will be shuffled around as the mappers' output, the `BloomFilter` class will have to implement the `Writable` interface, which consists of methods `write()` and `readFields()`. For our purpose these methods transform between the internal `BitSet` representation and a byte array such that the data can be serialized to `DataInput`/`DataOutput`. The final code is in listing 5.4.

Listing 5.4 Basic Bloom filter implementation

```
class BloomFilter<E> implements Writable {

    private BitSet bf;
    private int bitArraySize = 100000000;
    private int numHashFunc = 6;

    public BloomFilter() {
        bf = new BitSet(bitArraySize);
    }

    public void add(E obj) {
        int[] indexes = getHashIndexes(obj);

        for (int index : indexes) {
            bf.set(index);
        }
    }

    public boolean contains(E obj) {
        int[] indexes = getHashIndexes(obj);

        for (int index : indexes) {
            if (bf.get(index) == false) {
                return false;
            }
        }
        return true;
    }

    public void union(BloomFilter<E> other) {
        bf.or(other.bf);
    }

    protected int[] getHashIndexes(E obj) {
        int[] indexes = new int[numHashFunc];

        long seed = 0;
        byte[] digest;
        try {
            MessageDigest md = MessageDigest.getInstance("MD5");
            md.update(obj.toString().getBytes());
            digest = md.digest();

            for (int i = 0; i < 6; i++) {
                seed = seed ^ (((long)digest[i] & 0xFF))<<(8*i);
            }
        } catch (NoSuchAlgorithmException  e) {}

        Random gen = new Random(seed);

        for (int i = 0; i < numHashFunc; i++) {
            indexes[i] = gen.nextInt(bitArraySize);
        }

        return indexes;
    }

    public void write(DataOutput out) throws IOException {
        int byteArraySize = (int)(bitArraySize / 8);

        byte[] byteArray = new byte[byteArraySize];
        for (int i = 0; i < byteArraySize; i++) {
            byte nextElement = 0;
            for (int j = 0; j < 8; j++) {
                if (bf.get(8 * i + j)) {
                    nextElement |= 1<<j;
                }
            }

            byteArray[i] = nextElement;
        }
        out.write(byteArray);
    }

    public void readFields(DataInput in) throws IOException {
        int byteArraySize = (int)(bitArraySize / 8);
        byte[] byteArray = new byte[byteArraySize];

        in.readFully(byteArray);

        for (int i = 0; i < byteArraySize; i++ ) {
            byte nextByte = byteArray[i];
```

```
                for (int j = 0; j < 8; j++) {
                    if (((int)nextByte & (1<<j)) != 0) {
                        bf.set(8 * i + j);
                    }
                }
            }
        }
    }
```

Next we'll create the MapReduce program to make a Bloom filter using Hadoop. As we said earlier, each mapper will instantiate a `BloomFilter` object and add the key of each record in its split into its `BloomFilter` instance. (We're using the key of the record to follow our data joining example.) We'll create a union of the `BloomFilters` by collecting them into a single reducer.

The driver for the MapReduce program is straightforward. Our mappers will output a key/value pair where the value is a `BloomFilter` instance.

```
job.setOutputValueClass(BloomFilter.class);
```

The output key will not matter in terms of partitioning because we only have a single reducer.

```
job.setNumReduceTasks(1);
```

We want our reducer to output the final `BloomFilter` as a binary file. Hadoop's `OutputFormats` outputs either text files or assumes a key/value pair. Our reducer, therefore, won't use Hadoop's MapReduce output mechanism and instead we'll write the result out to a file ourselves.

```
job.setOutputFormat(NullOutputFormat.class);
```

WARNING In general life gets a little more dangerous when you deviate from MapReduce's input/output framework and start working with your own files. Your tasks are no longer guaranteed to be idempotent and you'll need to understand how various failure scenarios can affect your tasks. For example, your files may only be partially written when some tasks are restarted. Our example here is safe(r) because all the file operations take place together only once in the `close()` method and in only one reducer. A more careful/paranoid implementation would check each individual file operation more closely.

Recall that our strategy for the mapper is to build a single Bloom filter on the entire split and output it *at the end of the split* to the reducer. Given that the `map()` method of the `MapClass` has no state information about which record in the split it's processing, we should output the `BloomFilter` in the `close()` method to ensure that all the records in the split have been read. Although the `map()` method is passed an `OutputCollector` to collect the mapper's outputs, the `close()` method is not given one. The standard pattern in Hadoop to get around this situation is for `MapClass` itself to hold on to a reference to the `OutputCollector` when it's passed into `map()`. This `OutputCollector` is known to function even in the `close()` method. The `MapClass` looks like

```
public static class MapClass extends MapReduceBase
        implements Mapper<K1, V1, K2, V2> {

    OutputCollector<K2, V2> oc = null;

    public void map(K1 key, V1 value,
                    OutputCollector<K2,V2> output,
                    Reporter reporter) throws IOException {

        if (oc == null) oc = output;
        ...
    }

    public void close() throws IOException {
        oc.collect(k, v);
    }
}
```

The `BloomFilters` generated by all the mappers are sent to a single reducer. The `reduce()` method in the `Reduce` class will do a Bloom filter union of all of them.

```
while (values.hasNext()) {
    bf.union((BloomFilter<String>)values.next());
}
```

As we mentioned earlier, we want the final `BloomFilter` to be written out in a file of our own format rather than one of Hadoop's `OutputFormats`. We had already set the reducer's `OutputFormat` to `NullOutputFormat` in the driver to turn off that output mechanism. Now the `close()` method will have to handle the file output itself. It will have to know the output path as specified by the user, which can be found in the `mapred.output.dir` property of the `JobConf` object. But the `close()` is not given the job configuration object. We handle this oversight the same way we handled `Output-Collector` in the mapper. We keep a reference to the `JobConf` object in the `Reduce` class to be used by the `close()` method. The rest of the `close()` method will use Hadoop's file I/O to write out our `BloomFilter` in binary to a file in HDFS. The complete code is in listing 5.5.

Listing 5.5 A MapReduce program to create a Bloom filter

```
public class BloomFilterMR extends Configured implements Tool {

    public static class MapClass extends MapReduceBase
        implements Mapper<Text, Text, Text, BloomFilter<String>> {

        BloomFilter<String> bf = new BloomFilter<String>();
        OutputCollector<Text, BloomFilter<String>> oc = null;

        public void map(Text key, Text value,
                        OutputCollector<Text, BloomFilter<String>> output,
                        Reporter reporter) throws IOException {
```

```
            if (oc == null) oc = output;
            bf.add(key.toString());
        }

        public void close() throws IOException {
            oc.collect(new Text("testkey"), bf);
        }
    }

    public static class Reduce extends MapReduceBase
        implements Reducer<Text, BloomFilter<String>, Text, Text> {

        JobConf job = null;
        BloomFilter<String> bf = new BloomFilter<String>();

        public void configure(JobConf job) {
            this.job = job;
        }

        public void reduce(Text key, Iterator<BloomFilter<String>> values,
                           OutputCollector<Text, Text> output,
                           Reporter reporter) throws IOException {

            while (values.hasNext()) {
                bf.union((BloomFilter<String>)values.next());
            }
        }

        public void close() throws IOException {
            Path file = new Path(job.get("mapred.output.dir") +
                                 "/bloomfilter");
            FSDataOutputStream out = file.getFileSystem(job).create(file);
            bf.write(out);
            out.close();
        }
    }

    public int run(String[] args) throws Exception {
        Configuration conf = getConf();
        JobConf job = new JobConf(conf, BloomFilterMR.class);

        Path in = new Path(args[0]);
        Path out = new Path(args[1]);
        FileInputFormat.setInputPaths(job, in);
        FileOutputFormat.setOutputPath(job, out);

        job.setJobName("Bloom Filter");
        job.setMapperClass(MapClass.class);
        job.setReducerClass(Reduce.class);
        job.setNumReduceTasks(1);

        job.setInputFormat(KeyValueTextInputFormat.class);
        job.setOutputFormat(NullOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(BloomFilter.class);
        job.set("key.value.separator.in.input.line", ",");

        JobClient.runJob(job);

        return 0;
    }
```

```
        public static void main(String[] args) throws Exception {
            int res = ToolRunner.run(new Configuration(),
                                     new BloomFilterMR(),
                                     args);

            System.exit(res);
        }
    }
```

5.3.3 Bloom filter in Hadoop version 0.20+

Hadoop version 0.20 has a Bloom filter class in it. It plays a support role to some of the new classes introduced in version 0.20, and it will likely stay around for future versions as well. It functions much like our `BloomFilter` class in listing 5.4, although it's much more rigorous in its implementation of the hashing functions. As a built-in class, it can be a good choice for semijoin within Hadoop. But it's not easy to separate this class from the Hadoop framework to use it as a standalone class. If you're building a Bloom filter for non-Hadoop applications, Hadoop's built-in `BloomFilter` may not be appropriate.

5.4 Exercising what you've learned

You can test your understanding of more advanced MapReduce techniques through these exercises:

1. *Anomaly detection*—Take a web server log file. Write a MapReduce program to aggregate the number of visits for each IP address. Write another MapReduce program to find the top *K* IP addresses in terms of visits. These frequent visitors may be legitimate ISP proxies (shared among many users) or they may be scrapers and fraudsters (if the server log is from an ad network). Chain these two MapReduce jobs together such that they can be easily run on a daily basis.
2. *Filter out records in input*—In both patent data sets we've used (`cite75_99.txt` and `apat63_99.txt`), the first row is metadata (column names). So far we've had to explicitly or implicitly filter out that row in our mappers, or interpret our results knowing that the metadata record has some deterministic influence. A more permanent solution is to remove the metadata row from the input data and keep track of it elsewhere. Another solution is to write a mapper as a preprocessor that filters all records that look like metadata. (For example, records that don't start with a numeric patent number.) Write such a mapper and use `ChainMapper/ChainReducer` to incorporate it into your MapReduce programs.
3. *Disjoint selection*—Using the same Customers and Orders example for the datajoin package, how will you change the code to output customers *not* in the Orders data source? Perhaps the Orders data only contains orders made within the last *N* months, and these customers haven't purchased anything in that time period.

A business may choose to re-engage these customers with discounts or other incentives.

4 *Calculating ratios*—Ratios are often a better unit of analysis than raw numbers. For example, say you have a data set of today's stock prices and another data set for stock prices from yesterday. You may be more interested in each stock's growth rate than its absolute price. Use the datajoin framework to write a program that takes two data sources and output the ratio.

5 *Product of a vector with a matrix*—Look up your favorite linear algebra text on the definition of matrix multiplication. Implement a MapReduce job to take the product of a vector and a matrix. You should use `DistributedCache` to hold the value of the vector. You may assume the matrix is in sparse representation.

6 *Spatial join*—Let's get more adventurous. Consider a two-dimensional space where both the x and y coordinates range from -1,000,000,000 to +1,000,000,000. You have one file with the location of *foos*, and another file with the location of *bars*. Each record in those files is a comma-separated (x,y) coordinate. For example, a couple lines may look like
145999.32455,888888880.001
834478899.2,5656.87660922
Write a MapReduce job to find all *foos* that are less than 1 unit distance from a *bar*. Distance is measured using the familiar Euclidean distance, sqrt[$(x1-x2)^2 + (y1-y2)^2$]. Although both *foos* and *bars* are relatively sparse in this 2D space, their respective files are too big to be stored in memory. You can't use `DistributedCache` for this spatial join.
Hint: The datajoin package as it's currently implemented doesn't work that well for this problem either, but you can solve it with your own mapper and reducer that have a similar data flow as the datajoin package.
Hint #2: In all the MapReduce programs we've discussed up till now, the keys are only extracted and passed around, whereas the values go through various computations. You should consider computing the key for the mapper's output.

7 *Spatial join, enhanced with Bloom filter*—After you've answered the last question, figure out how you can use a Bloom filter to speed up the join operation. Assume *bars* are much fewer in number than *foos*, but still too many to fit all their locations in memory.

5.5 Summary

We can often write the basic MapReduce programs as one job operating on one data set. We may need to write the more advanced programs as multiple jobs or we may operate them on multiple data sets. Hadoop has several different ways of coordinating multiple jobs together, including sequential chaining and executing them according to predefined dependencies. For the frequent case of chaining map-only jobs around a full MapReduce job, Hadoop has special classes to do it efficiently.

Joining is the canonical example for processing data from multiple data sources. Though Hadoop has a powerful datajoin package for doing arbitrary joins, its generality comes at the expense of efficiency. A couple other joining methods can provide faster joins by exploiting the relative asymmetry in data source sizes typical of most data joins. One of these methods leverages the Bloom filter, a data structure that's useful in many data processing tasks.

At this point, your knowledge of MapReduce programming should enable you to start writing your own programs. As all programmers know, programming is more than writing code. You have various techniques and processes—from development to deployment and testing and debugging. The nature of MapReduce programming and distributed computing adds complexity and nuance to these processes, which we'll cover in the next chapter.

5.6 Further resources

http://portal.acm.org/citation.cfm?doid=1247480.1247602—MapReduce's lack of simple support for joining datasets is well-known. Many of the tools to enhance Hadoop (such as Pig, Hive, and CloudBase) offer data joins as a first-class operation. For a more formal treatment, Hung-chih Yang and coauthors have published a paper "Map-reduce-merge: simplified relational data processing on large clusters" that proposes a modified form of MapReduce with an extra "merge" step that supports data joining natively.

http://umiacs.umd.edu/~jimmylin/publications/Lin_etal_TR2009.pdf—Section 5.2.2 describes the use of distributed cache to provide side data to tasks. The limitation of this technique is that the side data is replicated to every TaskTracker, and the side data must fit into memory. Jimmy Lin and colleagues explore the use of memcached, a distributed in-memory object caching system, to provide global access to side data. Their experience is summarized in the paper "Low-Latency, High-Throughput Access to Static Global Resources within the Hadoop Framework."