CHAPTER 11

Google Cloud Dataflow

A few times in this book, we have introduced chapters by reflecting on how much technology has changed over the past few years and how that has shaped our understanding of concepts like security, information channels, mobility, and social exchange. But possibly one of the most groundbreaking advances in recent years is related to how we understand, analyze, and process large subsets of information. The advances in processing power have set the perfect stage for giant amounts of information to be generated. Regrettably, in the early days, this information generally was not compatible or accessible; and when it was, the computational needs required to work with it were out of reach for most individuals and entities.

Cloud computing has not only opened the door to providing sufficient power to process large subsets of data but also contributed to raising awareness of the potential of cross-connecting different sources of information. Take the example of sequencing your own genome. As recently as ten years ago, this was only within reach for a handful of people in the world. Today, you can do this for a couple thousand dollars; and in a few years, almost every living human will be able to know their inner source code and use it to help prevent future illnesses before they manifest.

Google Cloud Dataflow was conceived by Google to simplify the construction of batch or streaming data processing pipelines simply by providing SDKs and a fully managed and elastic infrastructure optimized for parallel execution of pipelines. To do that, it makes the best possible use of Google Cloud Platform internally, outsourcing computing power to Google Compute Engine or using Google BigQuery and Google Cloud Storage as two of the options for data sources and sinks. Cloud Dataflow can also use streaming data channels like Google Cloud Pub/Sub to communicate with other services in a real time fashion.

As you see in this chapter, one of the main strengths of this technology resides on its unified programming model, which enables you to work with bounded and unbounded sets of information using the same pipeline by simply exchanging the steps that define where the information is taken from and put once processed. Also, its modularity allows for more effective testing, as well as flexibility in the replacement of different parts of your pipeline.

A unit of work in Cloud Dataflow is referred to as a *pipeline*. This is a group of tasks, including operations like reading, extraction, transformation, and writing of information. You can construct pipelines using the SDKs provided (an SDK is only available for Java as of this writing). Once your pipeline is ready, you can run it locally or directly in the cloud. In the latter case, your pipeline is analyzed and optimized before creating a job representation of it. During the execution of this job, you can monitor its status and progress and access the logs generated from Google Developers Console.

In the following sections, you see how to set up, design, and execute a job using Cloud Dataflow.

Setup

It all starts with your project in Google Developers Console: `https://console.developers.google.com`. If you do not have a project yet, go ahead and create it.

As mentioned earlier, Cloud Dataflow uses other services from Cloud Platform. You need to enable the APIs for Cloud Dataflow and the associated services. You do that by accessing APIs & Auth ➤ APIs from the left side panel in your newly created project in Developers Console. These are the APIs that you need to enable in order to work with Cloud Dataflow:

- Google Cloud Dataflow API
- Google Compute Engine
- Google Cloud Logging API
- Google Cloud Storage
- Google Cloud Storage JSON API
- BigQuery API
- Google Cloud Pub/Sub
- Google Cloud Datastore API

You can use the search box at the top of the page to find APIs quickly. To enable an API, access its detail page and click Enable API.

■ **Note** As mentioned earlier, Cloud Dataflow can use other services in Google Cloud Platform like Compute Engine and Cloud Storage. Some of these services are capped to a certain quota in the free trial mode. To go beyond these quotas you must enable billing. To do so, click the Preferences icon next to your profile at upper right on screen. If a project is selected, you see a Project Billing Settings options. From there, you can see the details of the billing account associated with that project. To see all the billing accounts you have registered, click Billing Accounts on the same Preferences menu.

If you have not done so yet, install the Google Cloud SDK: `https://cloud.google.com/sdk/#Quick_Start`.

This SDK comes with a command-line tool (`gcloud`) that you can use to operate services in Cloud Platform like Compute Engine, Cloud DNS, Cloud SQL, Cloud Storage, and others. You can also use Cloud Dataflow, in alpha phase as of this writing. Once the SDK is installed, you can log in using the command `gcloud auth login` and explore the possibilities of this tool by listing the associated components (`gcloud components list`) and checking the help section (`gcloud -h`).

In the example used in this chapter, you use Cloud Storage as a data source and sink. Therefore, you need to create a bucket that you later reference from your task in Cloud Dataflow. You can create a new bucket in your project using the Developers Console: go to Storage ➤ Cloud Storage ➤ Storage Browser. Click Create Bucket, give it a name, and choose the standard storage class and the region that fits your location. You also need to create a bucket in Cloud Storage to persist the files that your pipeline uses when executed. Remember that every bucket name must be globally unique. To read more about Cloud Storage and the specifics of creating buckets, see Chapter 8.

■ **Note** Thanks to the way Cloud Dataflow is designed, exchanging your data sources and is very simple. To see different examples using Cloud Storage to Cloud Pub/Sub or BigQuery, check the examples in the Cloud Dataflow SDK for Java under `https://github.com/GoogleCloudPlatform/DataflowJavaSDK`.

You are now ready to begin designing and coding your pipeline using the Java SDK. Java is the only option available as of this writing, but alternatives for other programming languages are planned. The first step is to create a Java project with a class in charge of defining your pipeline, with the source and target storage and the transformations to apply to the data set.

■ **Note** To run the examples in this chapter and those included in the Cloud Dataflow SDK, you need at least version 1.7 of the Java Development Kit. You can download and install this version from the Java downloads page at `www.oracle.com/technetwork/java/javase/downloads/index.html`.

Conveniently, the Cloud Dataflow Java SDK is available in Maven Central—a repository of libraries and other dependencies that are publicly available in the Internet so that applications can reference them transparently without needing to download and set them up manually. If you are using Maven as your build system, you can reference the SDK in your Java project by adding the following dependency to your `pom.xml` file:

```
<dependency>
    <groupId>com.google.cloud.dataflow</groupId>
        <artifactId>google-cloud-dataflow-java-sdk-all</artifactId>
    <version>LATEST</version>
</dependency>
```

In the examples used in this chapter, you use Gradle to build your application. This approach brings independence to your code because Gradle[1] provides the option to embed the build system in your project in the form of a wrapper.[2] You can see the files that make up this wrapper in the repository for the chapter examples.[3] The `gradlew` command takes care of the execution; the contents of the wrapper are located under `gradle/`.

This is how you add a dependency for the Cloud Dataflow Java SDK in Gradle:

```
dependencies {
    compile 'com.google.cloud.dataflow:google-cloud-dataflow-java-sdk-all:0.3.+'
}
```

[1]"Installing Gradle": `http://gradle.org/docs/current/userguide/installation.html`.
[2]"The Gradle Wrapper": `http://gradle.org/docs/current/userguide/gradle_wrapper.html`.
[3]The Gradle wrapper is in the repository of the example used in this chapter at `https://github.com/GoogleCloudPlatformBook/code-snippets/tree/master/cloud-dataflow`.

■ **Note** In both examples, the targeted version of the dependency is not strictly specified. In Maven, setting the version to `LATEST` fetches the latest available version of the SDK. Similarly, in Gradle, the targeted version is the latest minor update available. For example, if versions 0.3.150109, 0.3.150606 and 0.4 are available, the dependency in Maven fetches version 0.4 and the alternative in Gradle chooses 0.3.15.0606. This is useful during development so that you always use the most up-to-date snapshot of your dependencies; but it is strongly discouraged in production scenarios, because an update in one of your dependencies could break your business logic, affecting your users. The code in this book is written using version 0.3.150109 of the Cloud Dataflow SDK for Java.

From this point on, you can use your favorite IDE or text editor to write your code. If you ask us, IntelliJ IDEA features a great integration with Gradle, and Sublime Text is a fantastic text editor. For the sake of simplicity, in this chapter the Java code is built and executed using Gradle directly through the command line.

The Building Blocks of Cloud Dataflow

The idea behind Cloud Dataflow is to help you abstract the details of the system that you use to perform processing operations, taking it to a level that resembles the way your brain operates when thinking about a specific problem. For example, data from source A is transformed into states 1, 2, and 3 and stored in data sink B.

Four main concepts will help you construct your jobs and understand the way Cloud Dataflow operates: pipelines, `PCollections`, transforms, and data sources and sinks.

In the following sections you get an introduction to these concepts. To dive into the details of each of them, check the reference for the SDK in Java – `https://cloud.google.com/dataflow/release-notes/java`.

■ **Note** As of this writing this service is in beta phase, thus it is possible that backward-incompatible changes are introduced. Because of that, it is important to keep an eye on the development of the service if you have bigger plans using Cloud Dataflow.

Pipelines

A pipeline represents a single unit of processing in Cloud Dataflow. Pipelines are constructed by writing logic on top of the Cloud Dataflow SDK and generally define where to read the data from and where to write the results back, as well as all the intermediate steps applied to the data, referred to as *transforms*. Pipelines are designed to run on independent services or runners, and the Cloud Dataflow service is one of them. The steps in your pipeline are executed as a single unit of work that cannot share information with other jobs.

This is an example of a dummy pipeline written in Java:

```
public class MyFirstPipeline{

    public static interface Options extends PipelineOptions {
        ...
    }

    public static void main(String[] args) {

        // Creates the options with the arguments coming from the command line
        Options options = PipelineOptionsFactory.fromArgs(args)
                                                .withValidation().as(Options.class);

        Pipeline pipeline = Pipeline.create(options);

        // Reads from source and writes to sink using SDK provided PTransforms
        pipeline.apply(TextIO.Read.named("Read Input")
                                 .from("gs://lunchmates_logs/access.log"))
                .apply(TextIO.Write.named("WriteOutput")
                                 .to("gs://lunchmates_logs/output/results.txt"));

        pipeline.run();
    }
}
```

This is the one most basic versions of a pipeline possible. All it does is read content from the input and then write it to the output. Normally you would add other transformations between these two steps. The names specified using the method `.named()` are used by the system to describe the steps involved in the process.

PCollection

A `PCollection` is the canonical representation of your data in a pipeline. Every step involved in the process takes and/or returns a `PCollection` object on its own or contained in another collection or container object. `PCollections` can hold any type of object—from integers or strings to composite objects like key-value pairs or `TableRows` from BigQuery—without any size restrictions. It is more appropriate to think of `PCollections` as data containers than regular collections, because they differ from collections substantially. For example, `PCollections` are immutable and only allow sequential access to their content.

PCollections can be classified according to their boundaries:

- Bounded PCollections hold datasets of a limited and known size that do not change. Data sources and sinks for the classes TextIO, BigQueryIO, and DatastoreIO, and those created using the custom source/sink API, operate with bounded PCollections.
- Unbounded PCollections represent information that is being added continuously, such as real-time or streaming information. This type of PCollection does not have known boundaries. The class PubSubIO works with unbounded PCollections on both the source and sink sides. BigQueryIO accepts unbounded PCollection objects as sink information: that is, you can write the resulting unbounded PCollections in BigQuery using BigQueryIO.

Unbounded PCollections are virtually unlimited; therefore, you need to define interruption points that allow Cloud Dataflow to work with chunks of transformed information coming from this type of PCollection. Attempting to apply grouping transformations—for example, GroupByKey—to unbounded PCollections will cause your pipeline to fail to construct, making the job fail. In other words, it is impossible to retrieve the values for all possible keys in a collection that is continually growing.

You can tune the factors that determine this division of information by using *windowing*—that is, setting the boundaries of originally unbounded collections based on a set of parameters or functions that you determine. This decision can be made based on data size, fixed or dynamic time intervals, or discrete conditions. For more information about windows[4] and triggers,[5] check the Google Cloud Platform documentation.

In addition to the aforementioned, other data sources and sinks may be added in the near future.

Transforms

Transforms represent each of the steps you use to mutate your data from the original to a modified state, operating with PCollections directly (taking and returning PCollection objects, or a list of them). PTransform objects must implement the apply method, which is where transformations are defined. In this method, you can add any logic necessary to obtain the desired results for your data.

If many steps are involved in a single transformation, it is recommended that you break the operation into smaller versions that you can group together. This not only improves readability and possibilities for testability, but also helps you visualize your pipeline when in action. The transformations applied are represented in hierarchically organized boxes that you can graphically interact with from the Developers Console by expanding and collapsing them to dive into the details.

Figure 11-1 shows an example pipeline in the Cloud Dataflow section in Developers Console when it's executed.

[4]Windows in Cloud Dataflow: https://cloud.google.com/dataflow/model/windowing.
[5]Triggers in Cloud Dataflow: https://cloud.google.com/dataflow/model/triggers.

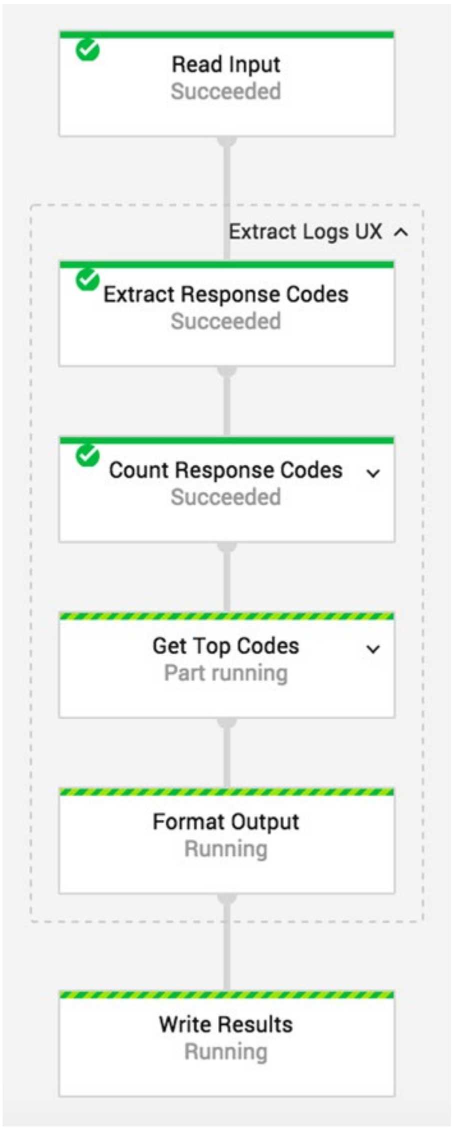

Figure 11-1. *Graphical representation of your pipeline in the Developers Console*

Using the example pipeline, here is how you apply a single transformation between the input read and the output write:

```
// Reads from source and writes to sink using SDK provided PTransforms
pipeline.apply(TextIO.Read.named("Read Input").from(options.getInput()))
        .apply(new ExtractLogExperience().withName("Extract Logs UX"))
        .apply(TextIO.Write.named("Write Output").to(options.getOutput()));
```

ExtractLogExperience is a custom composite transform that groups several transforms into a single unit. You can see that in Figure 11-1, where there is an expanded box named after the composite transformation class that groups four other subtransforms: ExtractResponseCodes, CountResponseCodes, GetTopCodes, and FormatOutput.

You can group functionality using as many levels of nested composite transforms as you need, but in the end, the core of your composite transforms is formed by core transforms. Core transforms are the foundation of Cloud Dataflow's parallel-processing model. These types of transformations apply a set of rules that complement the logic specific to the transformation. This logic is defined by a set of function classes that are either defined by the system or that you can implement yourself.

Cloud Dataflow SDK provides four fundamental core transformations, as discussed in the following sections.

ParDo: Processing Data in Parallel

ParDo is a core transformation for parallel operation that executes any logic specified in its function class (DoFn) and applies it to each of the elements in the associated PCollection. Here is an example:

```
PCollection<String> responseCodes = linesCollection
.apply(ParDo.named("Extract Response Codes").of(new GetResponseCodeFn()));
```

GetResponseCodeFn inherits from the class expected by ParDo (DoFn) and implements the method that processes each of the elements, processElement:

```
private static class GetResponseCodeFn extends DoFn<String, String> {

    private static final String RESPONSE_CODE_PATTERN = "HTTP[0-9./]*\" ([2345][0-9][0-9])";
    private static final Pattern pattern = Pattern.compile(RESPONSE_CODE_PATTERN);

    @Override
    public void processElement(ProcessContext context) {

        // Find matches for the specified regular expression
        Matcher matcher = pattern.matcher(context.element());

        // Output each response code into the resulting PCollection
        if (matcher.find()) {
            context.output(matcher.group(1));
        }
    }
}
```

The logic in processElement is scoped to the processing of a single object because the execution of the transformation may happen in parallel across different instances. That is, the collection may be split into subcollections that are processed by different machines at the same time.

ParDo(s) can take multiple input data through side inputs. These inputs are represented by PCollectionViews, computed by earlier steps in the pipeline that add information on how to process the main input PCollection. Similarly, ParDo(s) can return multiple output PCollections using PCollectionList if all the collections in the list contain the same data type or PCollectionTuple otherwise.

GroupByKey

This transformation allows you to group key-value pair collections in parallel using their key. GroupByKey is useful in collections containing a set of pairs with duplicated keys holding different values. The resulting dataset is a collection of pairs with unique keys holding multiple values each.

Combine

This transformation allows you to aggregate a collection of values into a single value—Combine.globally—or key-value pairs into key grouped collections—Combine.perKey. In addition to the aggregation type, this transformation accepts an Fn class referencing a function that is used to aggregate the items in the collection. For example, the following snippet calculates the mean of all the elements in a PCollection into a single result:

```
PCollection<Integer> collection = ...;
PCollection<Double> mean = collection.apply(Combine.globally(new Mean.MeanFn<Integer>()));
```

The classes that hold these mathematical operations also contain helper methods that you can use to reference the same operations with much cleaner syntax. For example, the previous method can be replaced with

```
PCollection<Integer> collection = ...;
PCollection<Double> mean = collection.apply(Mean.<Integer>globally());
```

Now, suppose you have a set of key-value pairs in which each element holds a list of values. This can happen, for example, after a GroupByKey transformation:

```
book, 4,2,4
song, 1,2
movie, 8,3
```

As in the previous snippet, you can extract the maximum value of the list of elements for each of the keys in a set of key-value pairs:

```
PCollection<Integer> collection = ...;
PCollection<KV<String, Long>> max = collection
.apply(Combine.<String, Long>perKey(new Max.MaxLongFn()));
```

Just like before, there is a helper method to generate the same Combine directive:

```
PCollection<Integer> collection = ...;
PCollection<KV<String, Long>> max = collection.apply(Max.<String>longsPerKey());
```

This function returns the same pairs, including only the maximum value for each entry:

```
book, 4
song, 2
movie, 8
```

Flatten: Merging Data Collections

Flatten helps you merge a list of PCollection-holding objects of the same type into a single PCollection:

```
PCollection<String> collection1 = ...
PCollection<String> collection2 = ...

PCollectionList<String> collections = PCollectionList.of(collection1).and(collection2);
PCollection<String> mergedResults = collections.apply(Flatten.<String>pCollections());
```

When you use unbounded PCollections with windows, all collections must have equal windowing functions—WindowingFn—to ensure that all elements have the same window strategy and sizing.

Data Sources and Sinks

Data sources define where your data is read from, whereas sinks specify where the data is written after the processing is done. Reading and writing the information you're working with are usually the first and last steps of a processing job, respectively. Cloud Dataflow makes it simple to treat different sources and sinks in a similar way by providing you with a set of interfaces, each representing different types of storage systems and services. That way, you can flexibly decide where your information comes from and where it goes.

Data sources and sinks work with PCollection objects; sources generate them, and sinks accept them as input when performing the final write operation. As of this writing, the Cloud Dataflow Java SDK includes the following transforms to operate with data sources and sinks.

Text Files

The class TextIO works with files stored in your local file system or in Google Cloud Storage. To reference files in Cloud Storage, you need to specify the appropriate gs scheme: gs://<bucket>/<filepath>.

This is how you read a file from Cloud Storage:

```
PipelineOptions options = PipelineOptionsFactory.create();
Pipeline pipeline = Pipeline.create(options);

PCollection<String> lines = pipeline.apply(TextIO.Read.named("ReadMyInputLogs")
                                    .from("gs://path-to-your-file/file.ext"));
```

Similarly, here is how you write the contents of a PCollection object into a file in Cloud Storage:

```
lines.apply(TextIO.Write.named("WriteMyInputLogs")
      .to("gs://path-to-your-file/file.ext"));
```

Note that Cloud Dataflow shards the execution of your job by default; because of that, output files can be generated with suffixes specifying shard numbers. If you want to control the final part of the file name, you can append a suffix using the method .withSuffix(".icontrolmyownextensions").

Avro Files

The class AvroIO operates similarly to TextIO, with the main difference being that AvroIO operates with Avro files with an associated schema instead of regular text files. Here is an example of reading Avro files as the input of your pipeline:

```
PipelineOptions options = PipelineOptionsFactory.create();
Pipeline pipeline = Pipeline.create(options);

PCollection<AvroType> lines = pipeline.apply(AvroIO.Read.named("ReadAvroFromGCS")
                                  .from("gs://path-to-your-file/file.avro")
                                  .withSchema(AvroType.class);
```

BigQuery Tables

You can use the class BigQueryIO to operate with tables in BigQuery as a means of reading and writing data. To do that, you need to specify the fully qualified name of the table you want to work with. This consists of three parts, concatenated together as follows: <project-id>:<dataset-id>.<table-id>.

If you obviate the project ID, the default project specified in PipelineOptions is used. You can access this value through the PipelineOptions.getProject method. In this instance, the table name is as follows: <dataset-id>.<table-id>.

Here is an example of a pipeline that reads information from a table in BigQuery:

```
PipelineOptions options = PipelineOptionsFactory.create();
Pipeline pipeline = Pipeline.create(options);

PCollection<TableRow> habitablePlanets = pipeline.apply(BigQueryIO.Read
                   .named("ReadPlanetInformation")
                         .from("earth-is-a-strange-place:planets.habitable-planets"));
```

Cloud Pub/Sub Messages

You can read from and write data to Cloud Pub/Sub messages using the class PubSubIO. You do that by specifying either a subscription that has already been created in Cloud Pub/Sub or the topic you are interested in.

■ **Note** Due to the nature of Cloud Pub/Sub messages, this data source can only be used in streaming pipelines. Therefore, you need to enable the streaming flag in PipelineOptions. For the same reason, PubSubIO operates with unbounded collections; thus you must specify a window strategy for the resulting PCollection before applying transformations that expect data with defined boundaries, such as GroupByKey and Combine.

The following snippet uses a topic called `projects/earth-is-a-strange-place/topics/habitable-planets` in Cloud Pub/Sub as a data source:

```
PipelineOptions options = PipelineOptionsFactory.create();
Pipeline pipeline = Pipeline.create(options);

String topicName = "projects/earth-is-a-strange-place/topics/habitable-planets";
PCollection<String> dataFromTopic = pipeline.apply(PubsubIO.Read
                                                .named("ReadFromPubsub")
                                                .topic(topicName));
```

Similarly, you can write `PCollections` into Cloud Pub/Sub channels by doing the following:

```
PCollection<String> dataFromTopic = ...;
dataFromTopic.apply(PubsubIO.Write.named("WriteToPubsub").topic(topicName));
```

To learn more about Cloud Pub/Sub, see Chapter 13.

Constructing and Executing Jobs in Cloud Dataflow

Now you have a basic idea of how Cloud Dataflow operates and how to model the blocks that construct a job or pipeline using the SDK in Java. In this section, you apply these concepts to a practical example that you program and execute using the service in the cloud.

MAKING USERS AS HAPPY AS A PIE

In this example, the idea is to make users of your service happy. Sure, that is obvious, but how do you do it? Access logs not only provide valuable information to system administrators and developers, but also give you an idea of what your clients—and thus your users—see when they exchange information with your service or API.

For example, if your logs show a significantly greater presence of any response codes other than 200–299, there may be a need to fix some things. In this example, you process the `access.log` file that you uploaded to Google Cloud Storage in Chapter 8 and generate an output file that lists the occurrences of each response code in the logs.

■ **Pro Tip** If you have not read Chapter 8 yet, don't worry. You can use any of the thousands of `access.log` files indexed by Internet search engines.

Before getting your hands on the keyboard, it is important to design your system. Doing so helps you to identify potential issues in your strategy and speeds up the developing part.

As mentioned, the goal is to measure the impact of response codes from incoming requests to your server. The starting point is an `access.log` file, and the result should be something like a list of response codes and the number of occurrences for each of them. One approach to get that result is the following:

1. Read the file from the input.
2. Extract the response codes from the log lines.
3. Count the number of occurrences for each response code.
4. Get a list with the five most common response codes, to discard noise.
5. Format the results in an appropriate way to output.
6. Write the results to the data sink.

■ **Note** Before executing your pipeline, Cloud Dataflow automatically optimizes the steps involved. Because of that, the steps you define may not be executed in the same order you specify in your code.

Now that you know how your system should operate, you can start the fun part. The first thing to do is create the pipeline with which you will operate. A pipeline holds all the necessary information about the processing job—for example, the origin and destination of the data, the location of staging files for execution, the number of workers to execute the job, the project identifier in Google Cloud, and so on—as well as the transformations applied to it.

To create your pipeline, you call the `create` method, passing `PipelineOptions` as a parameter:

```
PipelineOptions options = PipelineOptionsFactory.create();
Pipeline pipeline = Pipeline.create(options);
```

The `PipelineOptions` object holds the information about the environment that the pipeline needs. You can take this approach one step further by loading the parameters with the arguments provided directly from the command line:

```
PipelineOptions options = PipelineOptionsFactory.fromArgs(arguments);
Pipeline pipeline = Pipeline.create(options);
```

Finally, you can also extend the functionality of `PipelineOptions` using inheritance. Suppose you want to control the data source and sink locations as parameters in your pipeline options object. You can do that by creating a new object that inherits from `PipelineOptions` and defining new properties that you can add as arguments whenever you run your job from the command line. In this example, the class `AllowedOptions` adds two more parameters:

```
public static interface AllowedOptions extends PipelineOptions {

   @Description("Default path to logs file")
   @Default.String("gs://lunchmates_logs/access.log")
   String getInput();
   void setInput(String value);
   @Description("Path of the file to write the results to")
   @Default.String("gs://lunchmates_logs/output/results.txt")
   String getOutput();
   void setOutput(String value);
}

AllowedOptions options = PipelineOptionsFactory.fromArgs(args)
                                              .withValidation()
                                              .as(AllowedOptions.class);

Pipeline pipeline = Pipeline.create(options);
```

This new class allows you to include two more arguments in the running command: `--input` and `--output`. These names are extracted from the getters and setters specified in your class. Note that you can specify default options in case the arguments are not set. Finally, the creation of the `PipelineOptions` object now includes a validation method according to the target class you just created. This validates the arguments obtained, based on the annotations and methods configured in your class.

As mentioned, in addition to the default arguments, you can set values for input and output locations:

```
$ ./gradlew run -Pargs="--project=lunch--mates
                        --runner=BlockingDataflowPipelineRunner
                        --stagingLocation=gs://lunchmates_logs/staging
                        --input=gs://lunchmates_logs/access.log
                        --output=gs://lunchmates_logs/output/results.txt"
```

Later, when the pipeline is ready for execution, we discuss the possible values for the arguments in this command.

Now that your pipeline is configured, you need to get some data. In this case, the `access.log` file you want to process is located in Cloud Storage at the location specified in the `PipelineOptions` object: `gs://lunchmates_logs/access.log` by default. Because the content is plain text, you use `TextIO` to read it:

```
pipeline.apply(TextIO.Read.named("Read Input").from(options.getInput()));
```

The result is written back to Cloud Storage in a text file, placed in the output folder specified in the `PipelineOptions` object: `gs://lunchmates_logs/output/results.txt` by default. Your pipeline now has entry and exit points:

```
pipeline.apply(TextIO.Read.named("Read Input").from(options.getInput()))
            .apply(new ExtractLogExperience().withName("Extract Logs UX"))
            .apply(TextIO.Write.named("Write Results")
                                  .to(options.getOutput())
                                  .withSuffix(".txt"));
```

All the steps are named so that the entire flow is easier to understand when it is being executed and represented in Cloud Dataflow.

Notice that there is only one transformation in this pipeline, named `Extract Logs UX`. This is a custom composite transform that contains subtransforms. When defining your own composite transforms, the system uses that to represent the flow diagram and group modules—which are shown as boxes—based on this hierarchy.

As expected, the class ExtractLogExperience contains all the necessary steps according to the design of this flow:

```
public static class ExtractLogExperience
extends PTransform<PCollection<String>, PCollection<String>> {

    @Override
    public PCollection<String> apply(PCollection<String> lines) {

        //1. Filter log line to extract response code
        PCollection<String> responseCodes = lines.apply(ParDo.named("Extract Response Codes")
                                                              .of(new GetResponseCodeFn()));

        //2. Counts occurrences for each response code found
        PCollection<KV<String, Long>> responseCodeResults = responseCodes
                                   .apply(Count.<String>perElement()
                                                   .withName("Count Response Codes"));

        //3. Get the top five response codes
        PCollection<List<KV<String, Long>>> topThreeResponseCodes = responseCodeResults
                                    .apply(new TopCodes().withName("Get Top Codes"));

        //4. Format response codes and counts into a printable string
        return topThreeResponseCodes.apply(ParDo.named("Format Output")
                                                .of(new FormatResultsFn()));
    }
}
```

When you create a class that inherits from PTransform. you specify the type of the input and output objects. As you can see in the definition of the class, this transform receives a PCollection<String> representing a collection with each of the lines of the log file and returns the same type of object; this time, each line represents the response code and the number of occurrences in the entire log.

1. Filter Log to Extract Response Code

Notice that because you are working with PCollections throughout the pipeline, the same apply method is used to add other transformations to it.

A filtering operation is a common task for ParDos. Remember that this is simply a container that executes some logic in parallel (you define the logic to execute in a class that inherits from DoFn). Other common tasks that fit parallel execution are formatting data and processing items in the collection in an independent fashion (for example, on a per-key basis). In this case, this is specified in the class GetResponseCodeFn:

```
private static class GetResponseCodeFn extends DoFn<String, String> {

    private static final String RESPONSE_CODE_PATTERN = "HTTP[0-9./]*\" ([2345][0-9][0-9])";
    private static final Pattern pattern = Pattern.compile(RESPONSE_CODE_PATTERN);
    @Override
    public void processElement(ProcessContext context) {

        // Find matches for the specified regular expression
        Matcher matcher = pattern.matcher(context.element());

        // Output each response code into the resulting PCollection
        if (matcher.find()) {
            context.output(matcher.group(1));
        }
    }
}
```

Similar to `PTransforms`, `DoFn` defines two types of arguments, corresponding to the input and output types, respectively.

The method `processElement` is called for every element in the collection. In this example, the regular expression looks for a common pattern in log files—`HTTP/[version]" [response-code]`. If the pattern is matched, the first group—the one containing the response code—is returned.

The output of this step should look something similar to the following (replacing new lines with bars "|"):

```
200 | 201 | 500 | 500 | 404 | 401 | 201 | 401 | 404 | 302 | 401 | 401 | 401 | 204 | 200 | ...
```

2. Count Occurrences for Each Response Code Found

In this step, the goal is to count the number of occurrences for each unique response code in the collection provided by the previous step. For this purpose, you can use the `Count` transform provided by the SDK, which does exactly what you are looking for:

```
PCollection<KV<String, Long>> responseCodeResults = responseCodes
                   .apply(Count.<String>perElement()
                               .withName("Count Response Codes"));
```

As you can see, the response object is a collection of key-value pairs of String and Long types, holding the unique response code and number result of the count transformation. The expected output is as follows:

```
[Response Code]: [# of occurrences]
200: 1248
201: 256
404: 512
401: 192
500: 64
503: 96
422: 1024
```

3. Get the Top Five Response Codes

In this step, you get the top five most common response codes. Similar to step 2, you use a transform provided by the Cloud Dataflow Java SDK, but in this instance the `Top` class requires a comparator that you need to define. For readability, modularity, and testability reasons, you create a custom `PTransform` class that simply applies the transformation from the `Top` class. As you can see, this new class is called `TopCodes`. Here is its definition:

```
private static class TopCodes
extends PTransform<PCollection<KV<String, Long>>, PCollection<List<KV<String, Long>>>> {

    @Override
    public PCollection<List<KV<String, Long>>>
    apply(PCollection<KV<String, Long>> responseCodes) {

        return responseCodes.apply(Top
        .of(5, new SerializableComparator<KV<String, Long>>() {

            @Override
            public int compare(KV<String, Long> o1, KV<String, Long> o2) {
                return Long.compare(o1.getValue(), o2.getValue());
            }
        }));
    }
}
```

Once again, this transform defines the input and output types. In this case, the output is a list of key-value pairs containing as many results as specified in the `Top` class.

In the `apply` method, you call the `Top` transform and specify the number of results you want to return, along with the anonymous inner comparator class. This class is used to let `Top` know how to determine which value is greater than the other.

Based on the result from the previous step, the output is as follows:

```
[Response Code]: [# of occurrences]
200: 1248
422: 1024
404: 512
201: 256
401: 192
```

4. Format Response Codes and Counts into a Printable String

In this step, you prepare and convert the processed content into the desired output format. For this example, you use the nomenclature `<response-code>|<number-of-occurrences>` so that you can read and parse the content quickly from other systems.

Just as in step 1, you perform this operation in parallel with a ParDo. In this case, the class defining the logic to execute is called FormatResultsFn, which once again inherits from DoFn:

```
private static class FormatResultsFn extends DoFn<List<KV<String, Long>>, String> {

    @Override
    public void processElement(ProcessContext context) {
        for (KV<String, Long> item : context.element()) {
            context.output(item.getKey() + "|" + item.getValue());
        }
    }
}
```

The final result of the processing job should look like this:

```
200|1266
302|99
404|75
301|31
304|20
```

This completes the pipeline job. As shown in the definition part, this content is now written into the desired output determined by the PipelineOptions configuration.

All that is left is running your pipeline. For reference purposes, here is the entire main method:

```
public static void main(String[] args) {

    AllowedOptions options = PipelineOptionsFactory.fromArgs(args)
                                                   .withValidation()
                                                   .as(AllowedOptions.class);
    Pipeline pipeline = Pipeline.create(options);

    pipeline.apply(TextIO.Read.named("Read Input").from(options.getInput()))
            .apply(new ExtractLogExperience().withName("Extract Logs UX"))
            .apply(TextIO.Write.named("Write Results").to(options.getOutput())
                                                      .withSuffix(".txt"));

    pipeline.run();
}
```

You can find the complete source for this pipeline in the code-snippets repository under the folder named cloud-dataflow —https://github.com/GoogleCloudPlatformBook/code-snippets/tree/master/cloud-dataflow.

Executing your pipeline

Your pipeline is now ready to run. Pipelines run in *runners* that execute your job locally, in Google's infrastructure or in third-party services.

To run the pipeline, you use the Gradle command that you saw earlier. Before getting into that, let's look at the contents of the `build.gradle` file. This file, located in the root folder of your project, is in charge of executing your code and determining the type of application, versions, and dependencies:

```
apply plugin: 'java'
apply plugin: 'application'

sourceCompatibility = JavaVersion.VERSION_1_7
targetCompatibility = JavaVersion.VERSION_1_7

mainClassName = 'com.lunchmates.LogAnalyzer'

repositories {
    mavenCentral()
}

dependencies {
    compile 'com.google.cloud.dataflow:google-cloud-dataflow-java-sdk-all:0.3.+'
    testCompile 'junit:junit:4.11'
}

task resources {
    def resourcesDir = new File('build/resources/main')
    resourcesDir.mkdirs()
}

run {
    if (project.hasProperty('args')) {
        args project.args.split('\\s')
    }
}

run.mustRunAfter 'resources'
```

This works as follows, from top to bottom:

- The `plugins` define the type of application to execute. This parameter allows Gradle to conveniently set up conventions and extensions.
- `sourceCompatibility` and `targetCompatibility` determine the target Java version to use for this application.
- The `mainClassName` specifies the class that is executed when you run the build script. This is always the class that contains the pipeline you intend to run.
- In the `repositories` section, you specify the different local and remote repositories from which you will fetch dependencies.
- `dependencies` is a list of libraries that your project needs to work. In this case, the Java SDK for Google Dataflow is added in this section. This makes dependency management simple and flexible.

The rest of the build script is specific to Google Dataflow. task `resources` creates the folder `/resources/main`, which Dataflow expects to find in the `directory` folder. Finally, the `run` method preprocesses the arguments string, turning it into a list that can be accessed in your code.

To run the pipeline, execute the `run` command in Gradle:

```
$ ./gradlew run -Pargs="--project=lunch--mates
                        --runner=DirectPipelineRunner
                        --stagingLocation=gs://lunchmates_logs/staging
                        --input=gs://lunchmates_logs/access.log
                        --output=gs://lunchmates_logs/output/results.txt"
```

The `runner` parameter determines whether your project runs using the Cloud Dataflow service or locally. To run your project locally, you can omit the `runner` parameter or use `DirectPipelineRunner`. Conversely, if you want to run it using Cloud Dataflow, you need to use `DataflowPipelineRunner` or `BlockingDataflowPipelineRunner`. The first value executes your pipeline asynchronously, whereas if you use `BlockingDataflowPipelineRunner`, the job is executed synchronously, so the console does not return until the work is done. During this time, the logs and execution information about the project are output in the console.

Here is a list of other relevant arguments for the configuration of your pipeline:

- `--project` selects the project in Google Cloud used to run your pipeline. The expected value is the identifier of your project.
- `--streaming` is a `boolean` value that determines whether streaming mode is enabled or disabled.
- `--stagingLocation` determines the location the service uses to store local binaries needed for distributed execution. It must be a valid Cloud Storage URL.
- `--tempLocation` specifies the location that the service uses to store temporary files. It must be a valid Cloud Storage URL.

In addition, you can specify the arguments of your custom `PipelineOptions` class if you specified one. In this example, remember that you added `input` and `output` to specify the data source and sink to operate with.

Running your project locally is encouraged before bumping instances. This way, you can troubleshoot and debug your code before using production resources. When you run your project locally, it is recommended that you work with a data set that fits in the computer's memory. Remember that you can use local files as data sources. And keep in mind that running your job locally also interacts with cloud services like Cloud Storage or BigQuery.

When you are ready to execute your job using the Cloud Dataflow service, you can specify one of the runners mentioned earlier. This creates a JSON representation of your module that is uploaded to Cloud Dataflow, optimized, and finally executed. Once your pipeline starts running, you can monitor its progress from the Cloud Dataflow panel in the Developers Console, in the Big Data section of the left bar. It shows you a list of the recently executed and in-progress jobs. Clicking one of them takes you to the detail page for that job. Cloud Dataflow features an intuitive representation of your pipeline, using collapsible, connected boxes, according to the structure of the steps involved in the process. This graph is updated in real time and shows information such as logs, summary of the task, and the number of records currently being processed. This is another great way to debug your pipeline logic, because you have an expectation about which steps reduce or increase the number of potential output records.

The following command runs your project synchronously using the Cloud Dataflow service for the project lunch--mates, places local and temporary files in the bucket lunchmates_logs/staging in Cloud Storage, takes the input data from the default input defined in PipelineOptions gs://lunchmates_logs/access.log, and writes the result into the default output gs://lunchmates_logs/output/results.txt:

```
$ ./gradlew run -Pargs="--project=lunch--mates
                        --runner=BlockingDataflowPipelineRunner
                        --stagingLocation=gs://lunchmates_logs/staging"
```

Showing Results

Choosing an appropriate data sink for your pipeline depends primarily on the purpose of the generated information. Here is a list of the options and their typical applications:

- BigQuery, if you intend to perform further analysis or aggregations with data from other sources.
- Cloud Storage, if the generated data is intended to be static and independent. For example, suppose you build an intelligent pipeline that extracts the essence of encyclopedias and reduces them to a few pages of text. You can store the result in Cloud Storage, because you do not need to further analyze or combine the outcome.
- Cloud Datastore, if you want to expose the data directly on an application you develop.
- Cloud Pub/Sub, is very powerful when your data is not constrained and changes continuously. For example real-time scenarios.

In this example, the ideal use of the results would be to show them in a simple web site that performs a very basic analysis of the top response codes and delivers a conclusion. Cloud Datastore and Cloud Storage are great data sinks for this purpose, because you can load information from these two sources easily from an application hosted in Google App Engine, for example.

Summary

In this chapter, you have been introduced to a service that makes processing big data a less cumbersome duty by abstracting the infrastructural details of the system away from you so that you can focus on the details of your problem. You learned about the building blocks of the system, constructed a pipeline using Cloud Dataflow components, and ran it locally and using Google Cloud's infrastructure.

Remember that as of this writing, this service is in beta phase, so there is a lot to look forward to in such areas as working with other programming languages, having more options to run pipelines in different providers, adding functions that help you process data, and so on. At the same time, some things may be subject to change, such as specific syntax or procedures. Because of that, if this service is crucial to you, we recommend that you keep an eye of the development of the platform and pair that with your learning process.

Independent of your final outcome and whether Cloud Dataflow is the right solution for it, Google and other companies are very interested in providing tools to help you handle and analyze big data in as quick and straightforward a manner as possible. This is a great time to catch the train with these technologies, which represent one of the biggest advances in computing in the last decade.

Finally, the Cloud Dataflow team is engaging in open source activities: the Java SDK for Cloud Dataflow is open and available in GitHub under the following repository: https://github.com/GoogleCloudPlatform/DataflowJavaSDK. This is a great opportunity for the technology itself and also for you to get a better understanding of how your logic operates and to learn and contribute as the technology evolves.