Chapter 2. Introduction to TensorFlow Extended

In the previous chapter, we introduced the concept of machine learning pipelines and discussed the components that make up a pipeline. In this chapter, we introduce *TensorFlow Extended* (TFX). The TFX library supplies all the components we will need for our machine learning pipelines. We define our pipeline tasks using TFX, and they can then be executed with a pipeline orchestrator such as Airflow or Kubeflow Pipelines. Figure 2-1 gives an overview of the pipeline steps and how the different tools fit together.

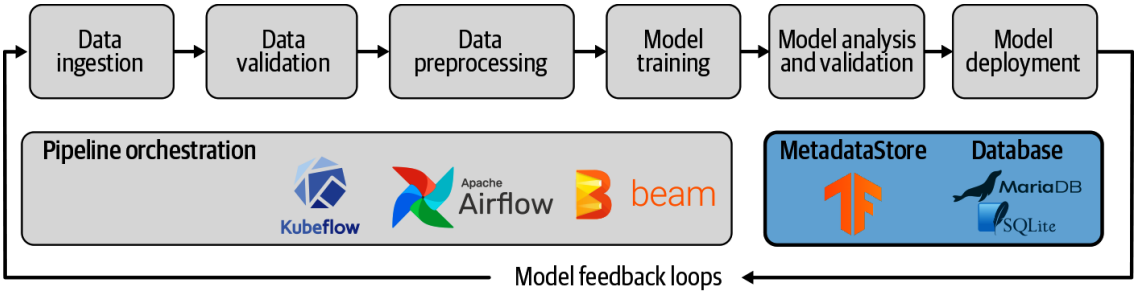

Figure 2-1. TFX as part of ML pipelines

In this chapter, we will guide you through the installation of TFX, explaining basic concepts and terminology that will set the stage for the following chapters. In those chapters, we take an in-depth look at the individual components that make up our pipelines. We also introduce Apache Beam in this chapter. Beam is an open source tool for defining and executing data-processing jobs. It has two uses in TFX pipelines: first, it runs under the hood of many TFX components to carry out processing steps like data validation or data preprocessing. Second, it can be used as a pipeline orchestrator, as we discussed in Chapter 1. We introduce Beam here because it will help you understand TFX components, and it is essential if you wish to write custom components, as we discuss in Chapter 10.

What Is TFX?

Machine learning pipelines can become very complicated and consume a lot of overhead to manage task dependencies. At the same time, machine learning pipelines can include a variety of tasks, including tasks for data validation, preprocessing, model training, and any post-training tasks. As we discussed in Chapter 1, the connections between tasks are often brittle and cause the pipelines to fail. These connections are also known as the glue code from the publication "Hidden Technical Debt in Machine Learning Systems". Having brittle connections ultimately means that production models will be updated infrequently, and data scientists and machine learning engineers loathe updating *stale* models. Pipelines also require well-managed distributed processing, which is why TFX leverages Apache Beam. This is especially true for large workloads.

Google faced the same problem internally and decided to develop a platform to simplify the pipeline definitions and to minimize the amount of task boilerplate code to write. The open source version of Google's internal ML pipeline framework is TFX.

Figure 2-2 shows the general pipeline architecture with TFX. Pipeline orchestration tools are the foundation for executing our tasks. Besides the orchestration tools, we need a data store to keep track of the intermediate pipeline results. The individual components communicate with the data store to receive their inputs, and they return the results to the data store. These results can then be inputs to following tasks. TFX provides the layer that combines all of these tools, and it provides the individual components for the major pipeline tasks.

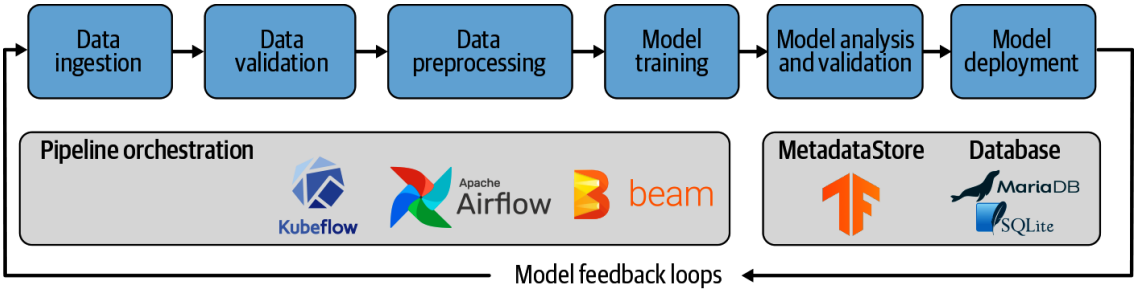

Figure 2-2. ML pipeline architecture

Initially, Google released some of the pipeline functionality as open source TensorFlow libraries (e.g., TensorFlow Serving is discussed in Chapter 8) under the umbrella of TFX libraries. In 2019, Google then pub-

lished the open source glue code containing all the required pipeline components to tie the libraries together and to automatically create pipeline definitions for orchestration tools like Apache Airflow, Apache Beam, and Kubeflow Pipelines.

TFX provides a variety of pipeline components that cover a good number of use cases. At the time of writing, the following components were available:

- Data ingestion with `ExampleGen`
- Data validation with `StatisticsGen`, `SchemaGen`, and the `ExampleValidator`
- Data preprocessing with `Transform`
- Model training with `Trainer`
- Checking for previously trained models with `ResolverNode`
- Model analysis and validation with `Evaluator`
- Model deployments with `Pusher`

Figure 2-3 shows how the components of the pipeline and the libraries fit together.

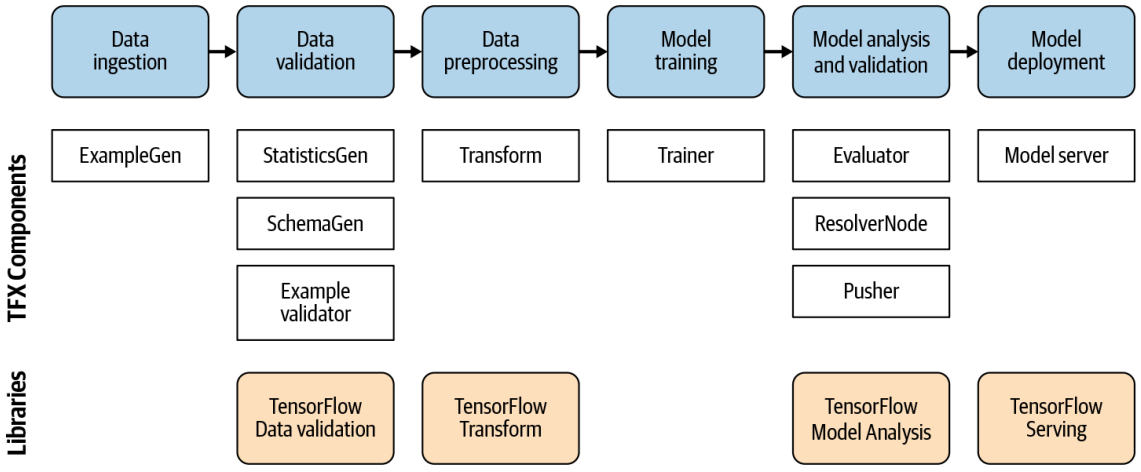

Figure 2-3. TFX components and libraries

We will discuss the components and libraries in greater detail in the following chapters. In case you need some nonstandard functionality, in Chapter 10 we discuss how to create custom pipeline components.

STABLE RELEASE OF TFX

At the time of writing this chapter, a stable 1.X version of TFX hasn't been released. The TFX API mentioned in this and the following chapters might be subject to future updates. To the best of our knowledge, all the examples in this book will work with TFX version 0.22.0.

Installing TFX

TFX can easily be installed by running the following Python installer command:

```
$ pip install tfx
```

The `tfx` package comes with a variety of dependencies that will be installed automatically. It installs not only the individual TFX Python packages (e.g., TensorFlow Data Validation), but also their dependencies like Apache Beam.

After installing TFX, you can import the individual Python packages. We recommend taking this approach if you want to use the individual TFX packages (e.g., you want to validate a dataset using TensorFlow Data Validation, see Chapter 4):

```
import tensorflow_data_validation as tfdv
import tensorflow_transform as tft
import tensorflow_transform.beam as tft_beam
...
```

Alternatively, you can import the corresponding TFX component (if using the components in the context of a pipeline):

```
from tfx.components import ExampleValidator
from tfx.components import Evaluator
from tfx.components import Transform
...
```

Overview of TFX Components

A component handles a more complex process than just the execution of a single task. All machine learning pipeline components read from a channel to get input artifacts from the metadata store. The data is then loaded from the path provided by the metadata store and processed. The output of the component, the processed data, is then provided to the next pipeline components. The generic internals of a component are always:

- Receive some input
- Perform an action
- Store the final result

In TFX terms, the three internal parts of the component are called the *driver*, *executor*, and *publisher*. The driver handles the querying of the metadata store. The executor performs the actions of the components. And the publisher manages the saving of the output metadata in the MetadataStore. The driver and the publisher aren't moving any data. Instead, they read and write references from the MetadataStore. Figure 2-4 shows the structure of a TFX component.

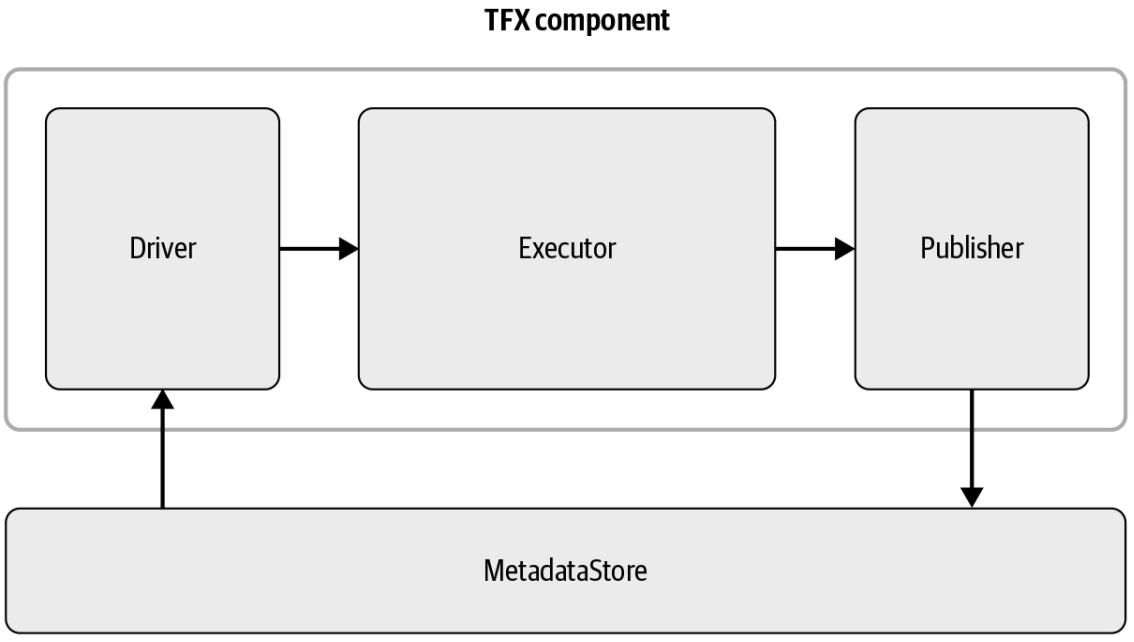

Figure 2-4. Component overview

The inputs and outputs of the components are called *artifacts*. Examples of artifacts include raw input data, preprocessed data, and trained mod-

els. Each artifact is associated with metadata stored in the MetadataStore. The artifact metadata consists of an artifact type as well as artifact properties. This artifact setup guarantees that the components can exchange data effectively. TFX currently provides ten different types of artifacts, which we review in the following chapters.

What Is ML Metadata?

The components of TFX "communicate" through *metadata*; instead of passing artifacts directly between the pipeline components, the components consume and publish references to pipeline artifacts. An artifact could be, for example, a raw dataset, a transform graph, or an exported model. Therefore, the metadata is the backbone of our TFX pipelines. One advantage of passing the metadata between components instead of the direct artifacts is that the information can be centrally stored.

In practice, the workflow goes as follows: when we execute a component, it uses the ML Metadata (MLMD) API to save the metadata corresponding to the run. For example, the component driver receives the reference for a raw dataset from the metadata store. After the component execution, the component publisher will store the references of the component outputs in the metadata store. MLMD saves the metadata consistently to a MetadataStore, based on a storage backend. Currently, MLMD supports three types of backends:

- In-memory database (via SQLite)
- SQLite
- MySQL

Because the TFX components are so consistently tracked, ML Metadata provides a variety of useful functions. We can compare two artifacts from the same component. For example, we see this in Chapter 7 when we discuss model validation. In this particular case, TFX compares the model analysis results from a current run with the results from the previous run. This checks whether the more recently trained model has a better accuracy or loss compared to the previous model. The metadata can also be

used to determine all the artifacts that have been based on another, previously created artifact. This creates a kind of audit trail for our machine learning pipelines.

Figure 2-5 shows that each component interacts with the MetadataStore, and the MetadataStore stores the metadata on the provided database backend.

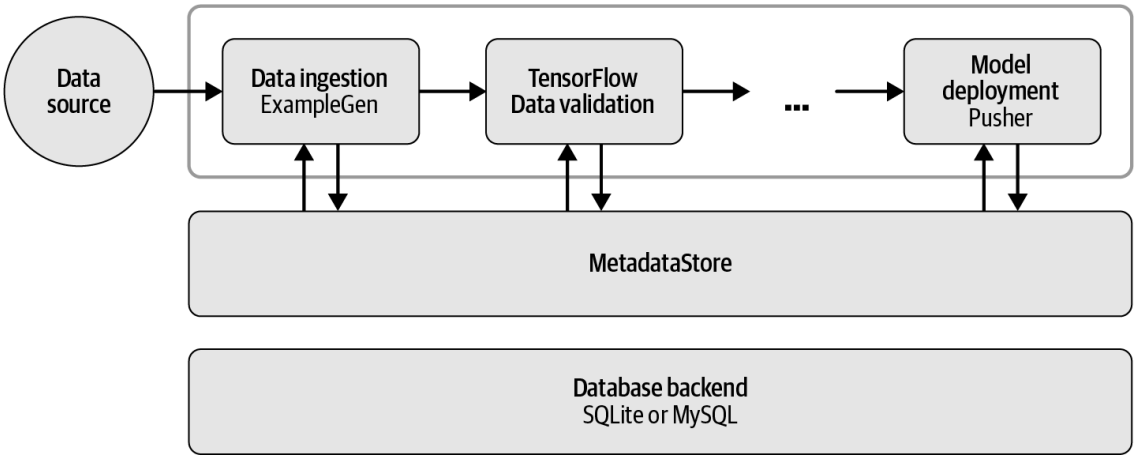

Figure 2-5. Storing metadata with MLMD

Interactive Pipelines

Designing and implementing machine learning pipelines can be frustrating at times. It is sometimes challenging to debug components within a pipeline, for example. This is why the TFX functionality around interactive pipelines is beneficial. In fact, in the following chapters, we will implement a machine learning pipeline step by step and demonstrate its implementations through an interactive pipeline. The pipeline runs in a Jupyter Notebook, and the components' artifacts can be immediately reviewed. Once you have confirmed the full functionality of your pipeline, in Chapters 11 and 12, we discuss how you can convert your interactive pipeline to a production-ready pipeline, for example, for execution on Apache Airflow.

Any interactive pipeline is programmed in the context of a Jupyter Notebook or a Google Colab session. In contrast to the orchestration tools we will discuss in Chapters 11 and 12, interactive pipelines are orchestrated and executed by the user.

You can start an interactive pipeline by importing the required packages:

```
import tensorflow as tf
from tfx.orchestration.experimental.interactive.interactive_context import \
    InteractiveContext
```

Once the requirements are imported, you can create a `context` object. The context object handles component execution and displays the component's artifacts. At this point, the `InteractiveContext` also sets up a simple in-memory ML MetadataStore:

```
context = InteractiveContext()
```

After setting up your pipeline component(s) (e.g., `StatisticsGen`), you can then execute each component object through the `run` function of the `context` object, as shown in the following example:

```
from tfx.components import StatisticsGen

statistics_gen = StatisticsGen(
    examples=example_gen.outputs['examples'])
context.run(statistics_gen)
```

The component itself receives the outputs of the previous component (in our case, the data ingestion component `ExampleGen`) as an instantiation argument. After executing the component's tasks, the component automatically writes the metadata of the output artifact to the metadata store. The output of some components can be displayed in your notebook. The immediate availability of the results and the visualizations is very convenient. For example, you can use the `StatisticsGen` component to inspect the features of the dataset:

```
context.show(statistics_gen.outputs['statistics'])
```

After running the previous `context` function, you can see a visual overview of the statistics of the dataset in your notebook, as shown in

Figure 2-6.

Sometimes it can be advantageous to inspect the output artifacts of a component programmatically. After a component object has been executed, we can access the artifact properties, as shown in the following example. The properties depend on the specific artifact:

```
for artifact in statistics_gen.outputs['statistics'].get():
    print(artifact.uri)
```

This gives the following result:

```
'/tmp/tfx-interactive-2020-05-15T04_50_16.251447/StatisticsGen/statistics/2'
```

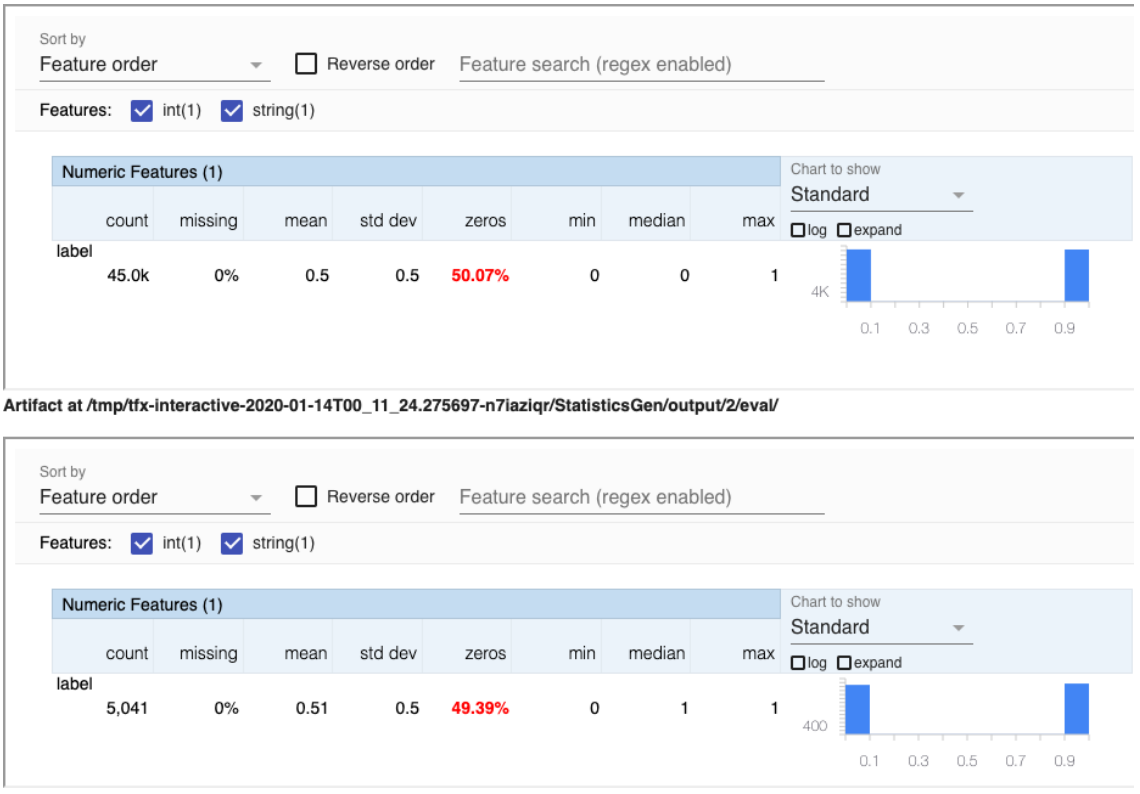

Figure 2-6. Interactive pipelines allow us to visually inspect our dataset

Throughout the following chapters, we will show how each component can be run in an interactive context. Then in Chapters 11 and 12, we will show the full pipeline and how it can be orchestrated by both Airflow and Kubeflow.

Alternatives to TFX

Before we take a deep dive into TFX components in the following chapters, let's take a moment to look at alternatives to TFX. The orchestration of machine learning pipelines has been a significant engineering challenge in the last few years, and it should come as no surprise that many major Silicon Valley companies have developed their own pipeline frameworks. In the following table, you can find a small selection of frameworks:

Company	Framework	Link
AirBnb	AeroSolve	*https://github.com/airbnb/aerosolve*
Stripe	Railyard	*https://stripe.com/blog/railyard-training-models*
Spotify	Luigi	*https://github.com/spotify/luigi*
Uber	Michelangelo	*https://eng.uber.com/michelangelo-machine-learning-platform/*
Netflix	Metaflow	*https://metaflow.org/*

Since the frameworks originated from corporations, they were designed with specific engineering stacks in mind. For example, AirBnB's AeroSolve focuses on Java-based inference code, and Spotify's Luigi focuses on efficient orchestration. TFX is no different in this regard. At this point, TFX architectures and data structures assume that you are using TensorFlow (or Keras) as your machine learning framework. Some TFX components can be used in combination with other machine learning frameworks. For example, data can be analyzed with TensorFlow Data Validation and later consumed by a scikit-learn model. However, the TFX framework is closely tied to TensorFlow or Keras models. Since TFX is backed by the TensorFlow community and more companies like Spotify

are adopting TFX, we believe it is a stable and mature framework that will ultimately be adopted by a broader base of machine learning engineers.

Introduction to Apache Beam

A variety of TFX components and libraries (e.g., TensorFlow Transform) rely on Apache Beam to process pipeline data efficiently. Because of the importance for the TFX ecosystem, we would like to provide a brief introduction into how Apache Beam works behind the scenes of the TFX components. In Chapter 11, we will then discuss how to use Apache Beam for a second purpose: as a pipeline orchestrator tool.

Apache Beam offers you an open source, vendor-agnostic way to describe data processing steps that then can be executed on various environments. Since it is incredibly versatile, Apache Beam can be used to describe batch processes, streaming operations, and data pipelines. In fact, TFX relies on Apache Beam and uses it under the hood for a variety of components (e.g., TensorFlow Transform or TensorFlow Data Validation). We will discuss the specific use of Apache Beam in the TFX ecosystem when we talk about TensorFlow Data Validation in Chapter 4 and TensorFlow Transform in Chapter 5.

While Apache Beam abstracts away the data processing logic from its supporting runtime tools, it can be executed on multiple distributed processing runtime environments. This means that you can run the same data pipeline on Apache Spark or Google Cloud Dataflow without a single change in the pipeline description. Also, Apache Beam was not just developed to describe batch processes but to support streaming operations seamlessly.

Setup

The installation of Apache Beam is straightforward. You can install the latest version with:

```
$ pip install apache-beam
```

If you plan to use Apache Beam in the context of Google Cloud Platform—for example, if you want to process data from Google BigQuery or run our data pipelines on Google Cloud Dataflow (as described in "Processing Large Datasets with GCP")—you should install Apache Beam as follows:

```
$ pip install 'apache-beam[gcp]'
```

If you plan to use Apache Beam in the context of Amazon Web Services (AWS) (e.g., if you want to load data from S3 buckets), you should install Apache Beam as follows:

```
$ pip install 'apache-beam[boto]'
```

If you install TFX with the Python package manager `pip`, Apache Beam will be automatically installed.

Basic Data Pipeline

Apache Beam's abstraction is based on two concepts: collections and transformations. On the one hand, Beam's collections describe operations where data is being read or written from or to a given file or stream. On the other hand, Beam's transformations describe ways to manipulate the data. All collections and transformations are executed in the context of a pipeline (expressed in Python through the context manager command `with`). When we define our collections or transformations in our following example, no data is actually being loaded or transformed. This only happens when the pipeline is executed in the context of a runtime environment (e.g., Apache Beam's DirectRunner, Apache Spark, Apache Flink, or Google Cloud Dataflow).

Basic collection example

Data pipelines usually start and end with data being read or written, which is handled in Apache Beam through collections, often called

`PCollections`. The collections are then transformed, and the final result can be expressed as a collection again and written to a filesystem.

The following example shows how to read a text file and return all lines:

```
import apache_beam as beam

with beam.Pipeline() as p: ❶
    lines = p | beam.io.ReadFromText(input_file) ❷
```

❶ Use the context manager to define the pipeline.

❷ Read the text into a `PCollection`.

Similar to the `ReadFromText` operation, Apache Beam provides functions to write collections to a text file (e.g., `WriteToText`). The write operation is usually performed after all transformations have been executed:

```
with beam.Pipeline() as p:
    ...
    output | beam.io.WriteToText(output_file) ❶
```

❶ Write the `output` to the file *output_file*.

Basic transformation example

In Apache Beam, data is manipulated through transformations. As we see in this example and later in Chapter 5, the transformations can be chained by using the pipe operator `|`. If you chain multiple transformations of the same type, you have to provide a name for the operation, noted by the string identifier between the pipe operator and the right-angle brackets. In the following example, we apply all transformations sequentially on our lines extracted from the text file:

```
counts = (
    lines
```

```
    | 'Split' >> beam.FlatMap(lambda x: re.findall(r'[A-Za-z\']+', x))
    | 'PairWithOne' >> beam.Map(lambda x: (x, 1))
    | 'GroupAndSum' >> beam.CombinePerKey(sum))
```

Let's walk through this code in detail. As an example, we'll take the phrases *"Hello, how do you do?"* and *"I am well, thank you."*

The `Split` transformation uses `re.findall` to split each line into a list of tokens, giving the result:

```
["Hello", "how", "do", "you", "do"]
["I", "am", "well", "thank", "you"]
```

`beam.FlatMap` maps the result into a `PCollection`:

```
"Hello" "how" "do" "you" "do" "I" "am" "well" "thank" "you"
```

Next, the `PairWithOne` transformation uses `beam.Map` to create a tuple out of every token and the count (1 for each result):

```
("Hello", 1) ("how", 1) ("do", 1) ("you", 1) ("do", 1) ("I", 1) ("am", 1)
("well", 1) ("thank", 1) ("you", 1)
```

Finally, the `GroupAndSum` transformation sums up all individual tuples for each token:

```
("Hello", 1) ("how", 1) ("do", 2) ("you", 2) ("I", 1) ("am", 1) ("well", 1)
("thank", 1)
```

You can also apply Python functions as part of a transformation. The following example shows how the function `format_result` can be applied to earlier produced summation results. The function converts the resulting tuples into a string that then can be written to a text file:

```
def format_result(word_count):
    """Convert tuples (token, count) into a string"""
```

```
    (word, count) = word_count
    return "{}: {}".format(word, count)

output = counts | 'Format' >> beam.Map(format_result)
```

Apache Beam provides a variety of predefined transformations. However, if your preferred operation isn't available, you can write your own transformations by using the `Map` operators. Just keep in mind that the operations should be able to run in a distributed way to fully take advantage of the capabilities of the runtime environments.

Putting it all together

After discussing the individual concepts of Apache Beam's pipelines, let's put them all together in one example. The previous snippets and following examples are a modified version of the Apache Beam introduction. For readability, the example has been reduced to the bare minimum Apache Beam code:

```
import re

import apache_beam as beam
from apache_beam.io import ReadFromText
from apache_beam.io import WriteToText
from apache_beam.options.pipeline_options import PipelineOptions
from apache_beam.options.pipeline_options import SetupOptions

input_file = "gs://dataflow-samples/shakespeare/kinglear.txt" ❶
output_file = "/tmp/output.txt"

# Define pipeline options object.
pipeline_options = PipelineOptions()

with beam.Pipeline(options=pipeline_options) as p: ❷
    # Read the text file or file pattern into a PCollection.
    lines = p | ReadFromText(input_file) ❸

    # Count the occurrences of each word.
    counts = ( ❹
        lines
```

```
            | 'Split' >> beam.FlatMap(lambda x: re.findall(r'[A-Za-z\']+', x))
            | 'PairWithOne' >> beam.Map(lambda x: (x, 1))
            | 'GroupAndSum' >> beam.CombinePerKey(sum))

    # Format the counts into a PCollection of strings.
    def format_result(word_count):
        (word, count) = word_count
        return "{}: {}".format(word, count)

    output = counts | 'Format' >> beam.Map(format_result)

    # Write the output using a "Write" transform that has side effects.
    output | WriteToText(output_file)
```

❶ The text is stored in a Google Cloud Storage bucket.

❷ Set up the Apache Beam pipeline.

❸ Create a data collection by reading the text file.

❹ Perform the transformations on the collection.

The example pipeline downloads Shakespeare's *King Lear* and performs the token count pipeline on the entire corpus. The results are then written to the text file located at */tmp/output.txt*.

Executing Your Basic Pipeline

As an example, you can run the pipeline with Apache Beam's DirectRunner by executing the following command (assuming that the previous example code was saved as `basic_pipeline.py`). If you want to execute this pipeline on different Apache Beam runners like Apache Spark or Apache Flink, you will need to set the pipeline configurations through the `pipeline_options` object:

```
python basic_pipeline.py
```

The results of the transformations can be found in the designated text file:

```
$ head /tmp/output.txt*
KING: 243
LEAR: 236
DRAMATIS: 1
PERSONAE: 1
king: 65
...
```

Summary

In this chapter, we presented a high-level overview of TFX and discussed the importance of a metadata store as well as the general internals of a TFX component. We also introduced Apache Beam and showed you how to carry out a simple data transformation using Beam.

Everything we discussed in this chapter will be useful to you as you read through Chapters 3–7 on the pipeline components and the pipeline orchestration expalined in Chapters 11 and 12. The first step is to get your data into the pipeline, and we will cover this in Chapter 3.