Tanmay Deshpande

Learning Apache Flink

Discover the definitive guide to crafting lightning-fast data processing for distributed systems with Apache Flink

Packt

Learning Apache Flink

Discover the definitive guide to crafting lightning-fast data processing for distributed systems with Apache Flink

Tanmay Deshpande

Packt

BIRMINGHAM - MUMBAI

Learning Apache Flink

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: February 2017

Production reference: 1160217

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78646-622-8

www.packtpub.com

Credits

Author
Tanmay Deshpande

Reviewers
Ivan Mushketyk
Chiwan Park

Commissioning Editor
Wilson D'souza

Acquisition Editor
Manish Nainani

Content Development Editor
Sumeet Sawant

Technical Editor
Deepti Tuscano

Copy Editor
Safis Editing

Project Coordinator
Shweta H Birwatkar

Proofreader
Safis Editing

Indexer
Francy Puthiry

Production Coordinator
Aparna Bhagat

About the Author

Tanmay Deshpande is a Hadoop and big data evangelist. He currently works with Schlumberger as a Big Data Architect in Pune, India. He has interest in a wide range of technologies, such as Hadoop, Hive, Pig, NoSQL databases, Mahout, Sqoop, Java, cloud computing, and so on. He has vast experience in application development in various domains, such as oil and gas, finance, telecom, manufacturing, security, and retail. He enjoys solving machine-learning problems and spends his time reading anything that he can get his hands on. He has great interest in open source technologies and has been promoting them through his talks. Before Schlumberger, he worked with Symantec, Lumiata, and Infosys. Through his innovative thinking and dynamic leadership, he has successfully completed various projects.

He regularly blogs on his website `http://hadooptutorials.co.in`. You can connect with him on LinkedIn at `https://in.linkedin.com/in/deshpandetanmay`.

He has also authored *Mastering DynamoDB*, published in August 2014, *DynamoDB Cookbook*, published in September 2015, *Hadoop Real World Solutions Cookbook-Second Edition*, published in March 2016, *Hadoop: Data Processing and Modelling*, published in August, 2016, and *Hadoop Blueprints*, published in September 2016, all by Packt Publishing.

About the Reviewers

Ivan Mushketyk is an experienced software development engineer and certified AWS specialist with a vast experience in different domains, including cloud computing, networking, artificial intelligence, and monitoring. Outside of work he is a big Open Source enthusiast and has made several contributions, most notably to Gatling and Apache Flink. Ivan also enjoys writing for technical blogs such as SitePoint, DZone, and his own blog Brewing Codes.

I would like to thank my wife, Svitlana, for her support, patience, encouragement, and love.

Chiwan Park is a master's student majoring in data mining in Korea. His research interests include large-scale graph processing and scalable machine learning. He has contributed to big data related open source projects such as Apache Sqoop and Apache Flink for 3 years. He is a committer of Apache Flink since June 2015.

I would like to thank my girlfriend, Sunmi Yoon, and my lovely dog Reo for standing by me. I would like to thank The Flink community for their great contributions. Apache Flink couldn't be improved without their valuable contributions.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

Mapt

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://goo.gl/iDhRaq`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

With the advent of massive computer systems, organizations in different domains generate large amounts of data at a real-time basis. The latest entrant to big data processing, Apache Flink, is designed to process continuous streams of data at a lightning fast pace.

This book will be your definitive guide to batch and stream data processing with Apache Flink. The book begins by introducing the Apache Flink ecosystem, setting it up and using the DataSet and DataStream API for processing batch and streaming datasets. Bringing the power of SQL to Flink, this book will then explore the Table API for querying and manipulating data. In the latter half of the book, readers will get to learn the remaining ecosystem of Apache Flink to achieve complex tasks such as event processing, machine learning, and graph processing. The final part of the book would consist of topics such as scaling Flink solutions, performance optimization, and integrating Flink with other tools such as Hadoop, ElasticSearch, Cassandra, and Kafka.

Whether you want to dive deeper into Apache Flink, or investigate how to get more out of this powerful technology, you'll find everything inside. This book covers a lot of real-world use cases, which will help you connect the dots.

What this book covers

Chapter 1, *Introduction to Apache Flink*, introduces you to the history, architecture, features and installation of Apache Flink on single node and multinode clusters.

Chapter 2, *Data Processing Using the DataStream API*, provides you with the details of Flink's streaming first concept. You will learn details about data sources, transformation, and data sinks available with DataStream API.

Chapter 3, *Data Processing Using the Batch Processing API*, enlightens you with the batch processing API, that is, DataSet API. You will learn about data sources, transformations, and sinks. You will also learn about the connectors available with the API.

Chapter 4, *Data Processing Using the Table API*, helps you understand how to use SQL concepts with Flink data processing frameworks. You will also learn how to apply these concepts to the real-world use case.

Chapter 5, *Complex Event Processing*, provides insights to you on how to solve complex event processing problems using Flink CEP library. You will learn details about the pattern definition, detection, and alert generation.

`Chapter 6`, *Machine Learning Using Flink ML*, covers details on machine learning concepts and how to apply various algorithms to the real-life use cases.

`Chapter 7`, *Flink Graph API - Gelly*, introduces you to the graph concepts and what Flink Gelly offers us to solve real-life use cases. It enlightens you on iterative graph processing capabilities provided by Flink.

`Chapter 8`, *Distributed Data Processing Using Flink and Hadoop*, covers details on how to use existing Hadoop-YARN clusters to submit Flink jobs. It talks about how Flink works on YARN in detail.

`Chapter 9`, *Deploying Flink on Cloud*, provides details on how to deploy Flink on Cloud. It talks in detail about how to use Flink on Google Cloud and AWS.

`Chapter 10`, *Best Practices*, covers various best practices developers should follow in order to use Flink in an efficient manner. It also talks about logging, monitoring best practices to control the Flink environment.

What you need for this book

You would need a laptop or desktop with any OS such as Windows, Mac, or UNIX. It's good to have an IDE such as Eclipse or IntelliJ and, of course, you would need a lot of enthusiasm.

Who this book is for

This book is for the big data developers who are looking to process batch and real-time data on distributed systems and for data scientists who look for industrializing analytic solutions.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, path names, dummy URLs, user input, and Twitter handles are shown as follows: "This will generate the public and private keys in the `/flinkuser/.ssh` folder."

A block of code is set as follows:

```
CassandraSink.addSink(input)
  .setQuery("INSERT INTO cep.events (id, message) values (?, ?);")
  .setClusterBuilder(new ClusterBuilder() {
    @Override
    public Cluster buildCluster(Cluster.Builder builder) {
      return builder.addContactPoint("127.0.0.1").build();
    }
  })
  .build();
```

Any command-line input or output is written as follows:

```
$sudo tar -xzf flink-1.1.4-bin-hadoop27-scala_2.11.tgz
$cd flink-1.1.4
$bin/start-local.sh
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Once all our work is done, it is important to shut down the cluster. To do this, we again need to go to AWS console and click on **Terminate** button".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support`and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Apache-Flink`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/MasteringApacheFlink_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support`and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Introduction to Apache Flink

With distributed technologies evolving all the time, engineers are trying to push those technologies to their limits. Earlier, people were looking for faster, cheaper ways to process data. This need was satisfied when Hadoop was introduced. Everyone started using Hadoop, started replacing their ETLs with Hadoop-bound ecosystem tools. Now that this need has been satisfied and Hadoop is used in production at so many companies, another need arose to process data in a streaming manner, which gave rise to technologies such as Apache Spark and Flink. Features, such as fast processing engines, the ability to scale in no time, and support for machine learning and graph technologies, are popularizing these technologies among the developer community.

Some of you might have been already using Apache Spark in your day-to-day life and might have been wondering if I have Spark why I need to use Flink? The question is quite expected and the comparison is natural. Let me try to answer that in brief. The very first thing we need to understand here is Flink is based on the **streaming first principle** which means it is real streaming processing engine and not a fast processing engine that collects streams as mini batches. Flink considers batch processing as a special case of streaming whereas it is vice-versa in the case of Spark. Likewise we will discover more such differences throughout this book.

This book is about one of the most promising technologies–Apache Flink. In this chapter we are going to talk about the following topics:

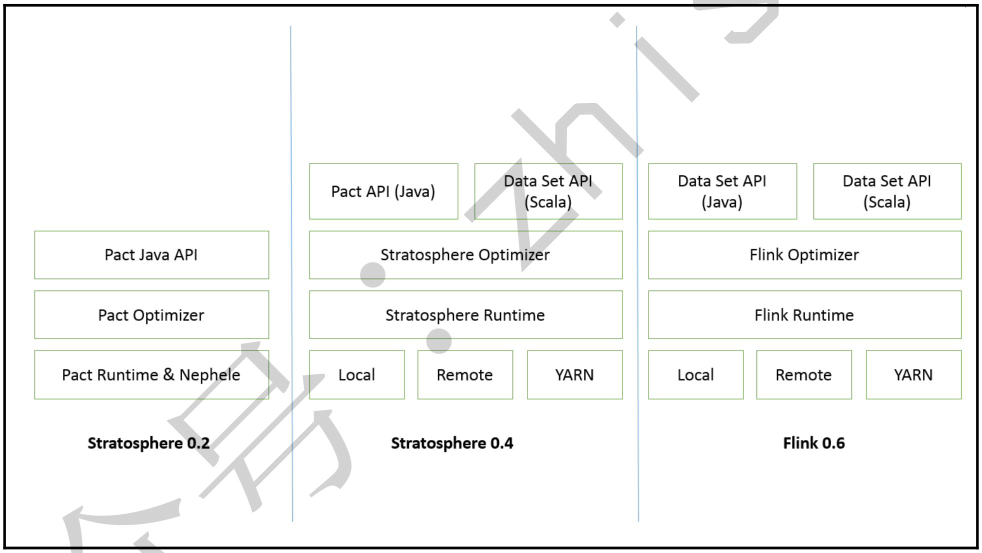
- History
- Architecture
- Distributed execution
- Features
- Quick start setup

- Cluster setup
- Running a sample application

History

Flink started as a research project named *Stratosphere* with the goal of building a next generation big data analytics platform at universities in the Berlin area. It was accepted as an Apache Incubator project on April 16, 2014. Initial versions of Stratosphere were based on a research paper by Nephele at `http://stratosphere.eu/assets/papers/Nephele_09.pdf`.

The following diagram shows how the evolution of Stratosphere happened over time:

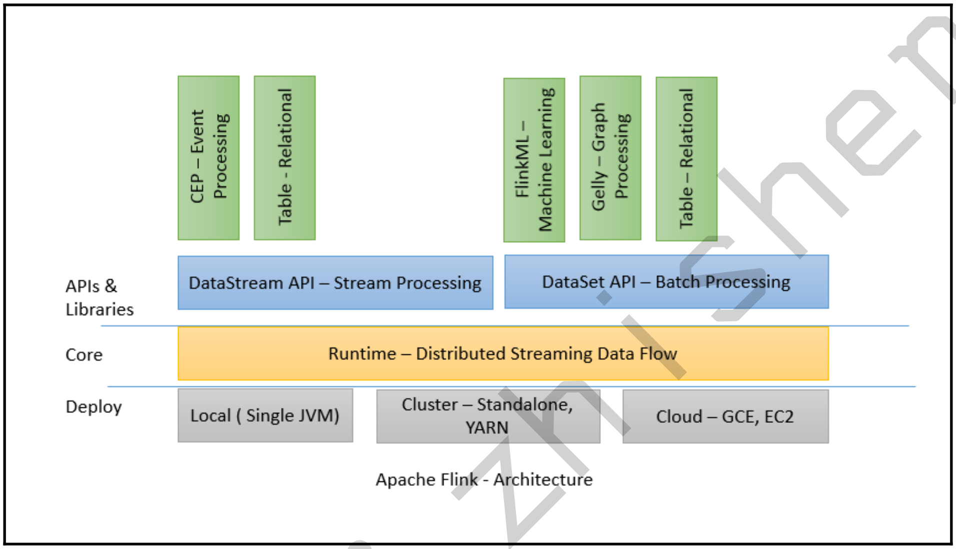


The very first version of Stratosphere was focused on having a runtime, optimizer, and the Java API. Later, as the platform got more mature, it started supporting execution on various local environments as well as on **YARN**. From version 0.6, Stratosphere was renamed Flink. The latest versions of Flink are focused on supporting various features such as batch processing, stream processing, graph processing, machine learning, and so on.

Flink 0.7 introduced the most important feature of Flink that is, Flink's streaming API. Initially release only had the Java API. Later releases started supporting Scala API as well. Now let's look the current architecture of Flink in the next section.

Architecture

Flink 1.X's architecture consists of various components such as deploy, core processing, and APIs. We can easily compare the latest architecture with Stratosphere's architecture and see its evolution. The following diagram shows the components, APIs, and libraries:


CEP – Event Processing
Table - Relational
FlinkML – Machine Learning
Gelly – Graph Processing
Table – Relational
APIs & Libraries
DataStream API – Stream Processing
DataSet API – Batch Processing
Core
Runtime – Distributed Streaming Data Flow
Deploy
Local (Single JVM)
Cluster – Standalone, YARN
Cloud – GCE, EC2
Apache Flink - Architecture

Flink has a layered architecture where each component is a part of a specific layer. Each layer is built on top of the others for clear abstraction. Flink is designed to run on local machines, in a YARN cluster, or on the cloud. Runtime is Flink's core data processing engine that receives the program through APIs in the form of JobGraph. **JobGraph** is a simple parallel data flow with a set of tasks that produce and consume data streams.

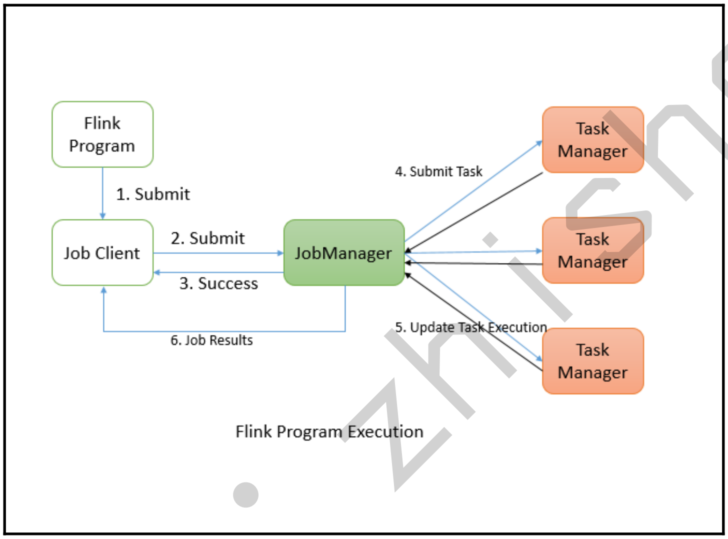
The DataStream and DataSet APIs are the interfaces programmers can use for defining the Job. JobGraphs are generated by these APIs when the programs are compiled. Once compiled, the DataSet API allows the optimizer to generate the optimal execution plan while DataStream API uses a stream build for efficient execution plans.

The optimized JobGraph is then submitted to the executors according to the deployment model. You can choose a local, remote, or YARN mode of deployment. If you have a Hadoop cluster already running, it is always better to use a YARN mode of deployment.

Distributed execution

Flink's distributed execution consists of two important processes, master and worker. When a Flink program is executed, various processes take part in the execution, namely **Job Manager**, **Task Manager**, and **Job Client**.

The following diagram shows the Flink program execution:



The Flink program needs to be submitted to a **Job Client**. The **Job Client** then submits the job to the **Job Manager**. It's the **Job Manager's** responsibility to orchestrate the resource allocation and job execution. The very first thing it does is allocate the required resources. Once the resource allocation is done, the task is submitted to the respective the **Task Manager**. On receiving the task, the **Task Manager** initiates a thread to start the execution. While the execution is in place, the Task Managers keep on reporting the change of states to the **Job Manager**. There can be various states such as starting the execution, in progress, or finished. Once the job execution is complete, the results are sent back to the client.

Job Manager

The **master** processes, also known as **Job Managers**, coordinate and manage the execution of the program. Their main responsibilities include scheduling tasks, managing checkpoints, failure recovery, and so on.

There can be many Masters running in parallel and sharing these responsibilities. This helps in achieving high availability. One of the masters needs to be the leader. If the leader node goes down, the master node (standby) will be elected as leader.

The Job Manager consists of the following important components:

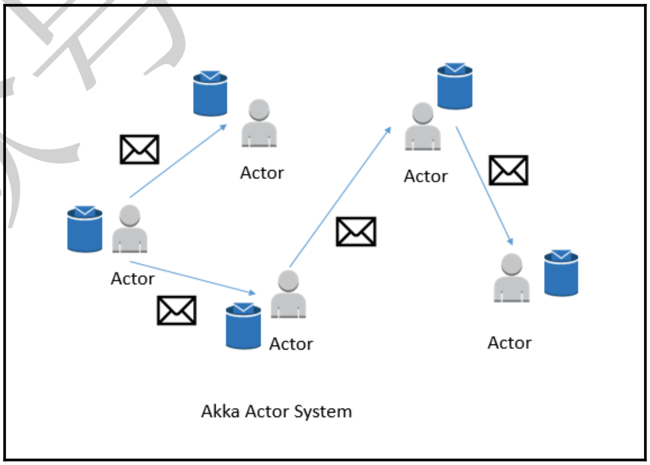
- Actor system
- Scheduler
- Check pointing

Flink internally uses the **Akka** actor system for communication between the Job Managers and the Task Managers.

Actor system

An actor system is a container of actors with various roles. It provides services such as scheduling, configuration, logging, and so on. It also contains a thread pool from where all actors are initiated. All actors reside in a hierarchy. Each newly created actor would be assigned to a parent. Actors talk to each other using a messaging system. Each actor has its own mailbox from where it reads all the messages. If the actors are local, the messages are shared through shared memory but if the actors are remote then messages are passed thought RPC calls.

Each parent is responsible for the supervision of its children. If any error happens with the children, the parent gets notified. If an actor can solve its own problem then it can restart its children. If it cannot solve the problem then it can escalate the issue to its own parent:



In Flink, an actor is a container having state and behavior. An actor's thread sequentially keeps on processing the messages it will receive in its mailbox. The state and the behavior are determined by the message it has received.

Scheduler

Executors in Flink are defined as task slots. Each Task Manager needs to manage one or more task slots. Internally, Flink decides which tasks needs to share the slot and which tasks must be placed into a specific slot. It defines that through the **SlotSharingGroup** and **CoLocationGroup**.

Check pointing

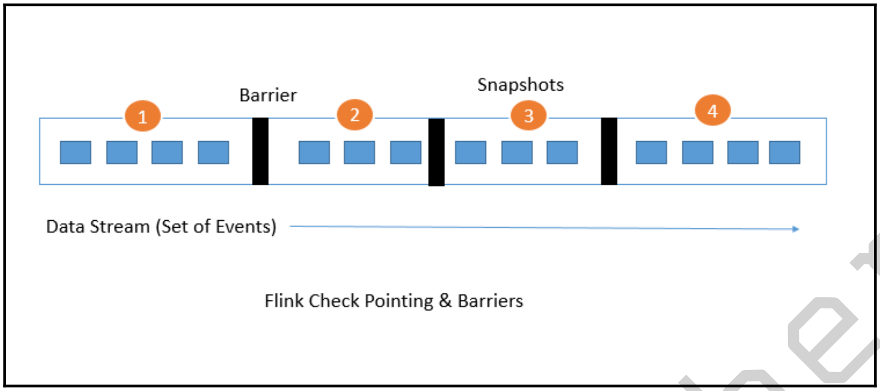
Check pointing is Flink's backbone for providing consistent fault tolerance. It keeps on taking consistent snapshots for distributed data streams and executor states. It is inspired by the Chandy-Lamport algorithm but has been modified for Flink's tailored requirement. The details about the Chandy-Lamport algorithm can be found at: `http://research.microsoft.com/en-us/um/people/lamport/pubs/chandy.pdf`.

The exact implementation details about snapshotting are provided in the following research paper: *Lightweight Asynchronous Snapshots for Distributed Dataflows* (`http://arxiv.org/abs/1506.08603`).

The fault-tolerant mechanism keeps on creating lightweight snapshots for the data flows. They therefore continue the functionality without any significant over-burden. Generally the state of the data flow is kept in a configured place such as HDFS.

In case of any failure, Flink stops the executors and resets them and starts executing from the latest available checkpoint.

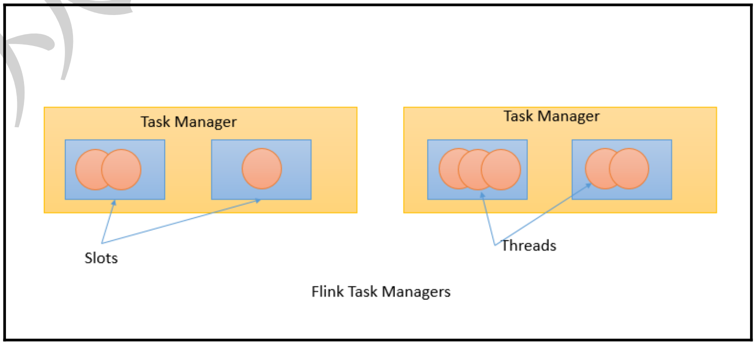
Stream barriers are core elements of Flink's snapshots. They are ingested into data streams without affecting the flow. Barriers never overtake the records. They group sets of records into a snapshot. Each barrier carries a unique ID. The following diagram shows how the barriers are injected into the data stream for snapshots:



Each snapshot state is reported to the Flink **Job Manager's** checkpoint coordinator. While drawing snapshots, Flink handles the alignment of records in order to avoid re-processing the same records because of any failure. This alignment generally takes some milliseconds. But for some intense applications, where even millisecond latency is not acceptable, we have an option to choose low latency over exactly a single record processing. By default, Flink processes each record exactly once. If any application needs low latency and is fine with at least a single delivery, we can switch off that trigger. This will skip the alignment and will improve the latency.

Task manager

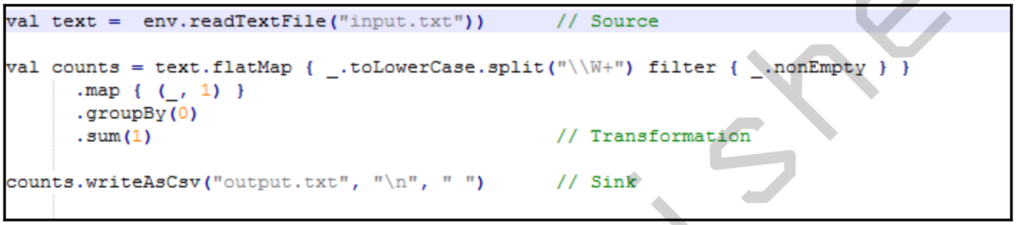
Task managers are worker nodes that execute the tasks in one or more threads in JVM. Parallelism of task execution is determined by the task slots available on each Task Manager. Each task represents a set of resources allocated to the task slot. For example, if a Task Manager has four slots then it will allocate 25% of the memory to each slot. There could be one or more threads running in a task slot. Threads in the same slot share the same JVM. Tasks in the same JVM share TCP connections and heart beat messages:



Job client

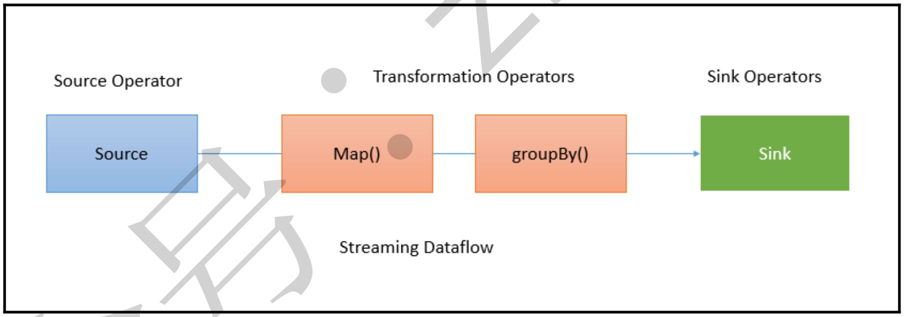
The job client is not an internal part of Flink's program execution but it is the starting point of the execution. The job client is responsible for accepting the program from the user and then creating a data flow and then submitting the data flow to the Job Manager for further execution. Once the execution is completed, the job client provides the results back to the user.

A data flow is a plan of execution. Consider a very simple word count program:



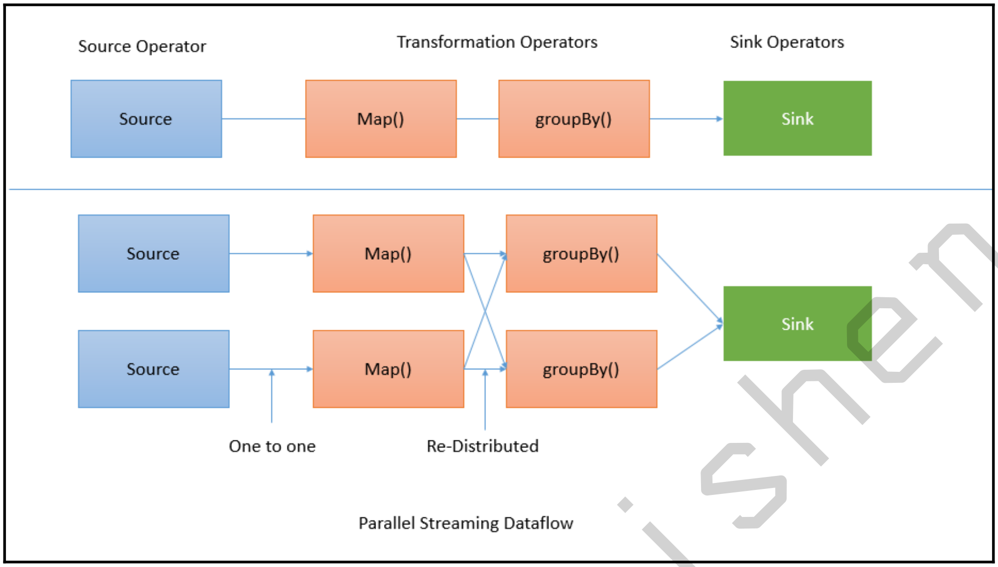
```
val text =  env.readTextFile("input.txt"))        // Source

val counts = text.flatMap { _.toLowerCase.split("\\W+") filter { _.nonEmpty } }
      .map { (_, 1) }
      .groupBy(0)
      .sum(1)                                   // Transformation

counts.writeAsCsv("output.txt", "\n", " ")      // Sink
```

When a client accepts the program from the user, it then transforms it into a data flow. The data flow for the aforementioned program may look like this:



The preceding diagram shows how a program gets transformed into a data flow. Flink data flows are parallel and distributed by default. For parallel data processing, Flink partitions the operators and streams. Operator partitions are called sub-tasks. Streams can distribute the data in a one-to-one or a re-distributed manner.

The data flows directly from the source to the map operators as there is no need to shuffle the data. But for a GroupBy operation Flink may need to redistribute the data by keys in order to get the correct results:



Features

In the earlier sections, we tried to understand the Flink architecture and its execution model. Because of its robust architecture, Flink is full of various features.

High performance

Flink is designed to achieve high performance and low latency. Unlike other streaming frameworks such as Spark, you don't need to do many manual configurations to get the best performance. Flink's pipelined data processing gives better performance compared to its counterparts.

Exactly-once stateful computation

As we discussed in the previous section, Flink's distributed checkpoint processing helps to guarantee processing each record exactly once. In the case of high-throughput applications, Flink provides us with a switch to allow at least once processing.

Flexible streaming windows

Flink supports data-driven windows. This means we can design a window based on time, counts, or sessions. A window can also be customized which allows us to detect specific patterns in event streams.

Fault tolerance

Flink's distributed, lightweight snapshot mechanism helps in achieving a great degree of fault tolerance. It allows Flink to provide high-throughput performance with guaranteed delivery.

Memory management

Flink is supplied with its own memory management inside a JVM which makes it independent of Java's default garbage collector. It efficiently does memory management by using hashing, indexing, caching, and sorting.

Optimizer

Flink's batch data processing API is optimized in order to avoid memory-consuming operations such as shuffle, sort, and so on. It also makes sure that caching is used in order to avoid heavy disk IO operations.

Stream and batch in one platform

Flink provides APIs for both batch and stream data processing. So once you set up the Flink environment, it can host stream and batch processing applications easily. In fact Flink works on Streaming first principle and considers batch processing as the special case of streaming.

Libraries

Flink has a rich set of libraries to do machine learning, graph processing, relational data processing, and so on. Because of its architecture, it is very easy to perform complex event processing and alerting. We are going to see more about these libraries in subsequent chapters.

Event time semantics

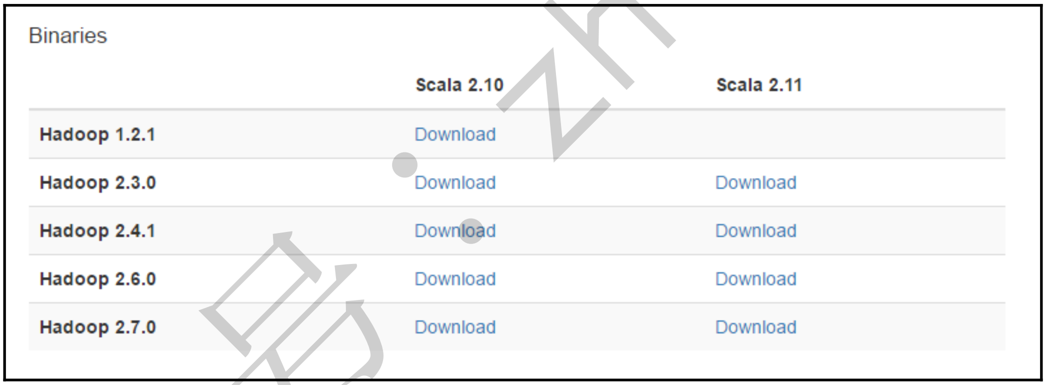
Flink supports event time semantics. This helps in processing streams where events arrive out of order. Sometimes events may come delayed. Flink's architecture allows us to define windows based on time, counts, and sessions, which helps in dealing with such scenarios.

Quick start setup

Now that we understand the details about Flink's architecture and its process model, it's time to get started with a quick setup and try out things on our own. Flink works on both Windows and Linux machines.

The very first thing we need to do is to download Flink's binaries. Flink can be downloaded from the Flink download page at: `http://flink.apache.org/downloads.html`.

On the download page, you will see multiple options as shown in the following screenshot:



Binaries

	Scala 2.10	Scala 2.11
Hadoop 1.2.1	Download	
Hadoop 2.3.0	Download	Download
Hadoop 2.4.1	Download	Download
Hadoop 2.6.0	Download	Download
Hadoop 2.7.0	Download	Download

In order to install Flink, you don't need to have Hadoop installed. But in case you need to connect to Hadoop using Flink then you need to download the exact binary that is compatible with the Hadoop version you have with you.

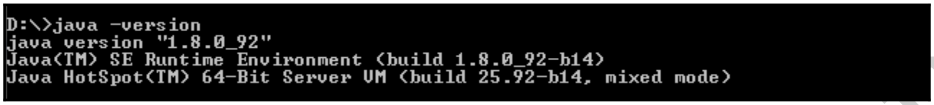
As I have latest version of **Hadoop 2.7.0** installed with me, I am going to download the Flink binary compatible with Hadoop 2.7.0 and built on Scala 2.11.

Here is direct link to download:

```
http://www-us.apache.org/dist/flink/flink-1.1.4/flink-1.1.4-bin-hadoop27-scal
a_2.11.tgz
```

Pre-requisite

Flink needs Java to be installed first. So before you start, please make sure Java is installed. I have JDK 1.8 installed on my machine:



```
D:\>java -version
java version "1.8.0_92"
Java(TM) SE Runtime Environment (build 1.8.0_92-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.92-b14, mixed mode)
```

Installing on Windows

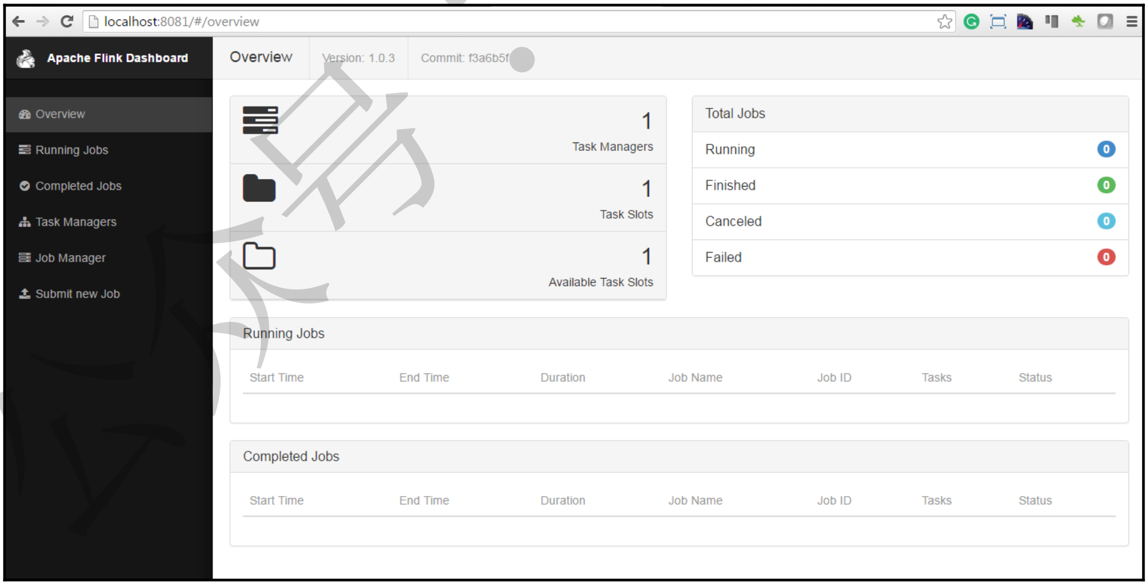
Flink installation is very easy to install. Just extract the compressed file and store it on the desired location.

Once extracted, go to the folder and execute `start-local.bat`:

```
>cd flink-1.1.4
>bin\start-local.bat
```

And you will see that the local instance of Flink has started.

You can also check the web UI on `http://localhost:8081/`:



localhost:8081/#/overview

Apache Flink Dashboard

Overview | Version: 1.0.3 | Commit: f3a6b5f

- Overview
- Running Jobs
- Completed Jobs
- Task Managers
- Job Manager
- Submit new Job

1 Task Managers

1 Task Slots

1 Available Task Slots

Total Jobs	
Running	0
Finished	0
Canceled	0
Failed	0

Running Jobs

Start Time	End Time	Duration	Job Name	Job ID	Tasks	Status

Completed Jobs

Start Time	End Time	Duration	Job Name	Job ID	Tasks	Status

You can stop the Flink process by pressing *Cltr* + *C*.

Installing on Linux

Similar to Windows, installing Flink on Linux machines is very easy. We need to download the binary, place it in a specific folder, extract, and finish:

```
$sudo tar -xzf flink-1.1.4-bin-hadoop27-scala_2.11.tgz
$cd flink-1.1.4
$bin/start-local.sh
```

As in Windows, please make sure Java is installed on the machine.

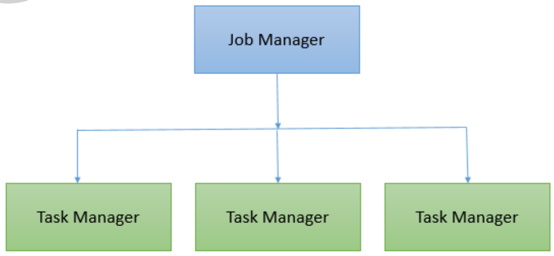
Now we are all set to submit a Flink job. To stop the local Flink instance on Linux, execute following command:

```
$bin/stop-local.sh
```

Cluster setup

Setting up a Flink cluster is very simple as well. Those who have a background of installing a Hadoop cluster will be able to relate to these steps very easily. In order to set up the cluster, let's assume we have four Linux machines with us, each having a moderate configuration. At least two cores and 4 GB RAM machines would be a good option to get started.

The very first thing we need to do this is to choose the cluster design. As we have four machines, we will use one machine as the **Job Manager** and the other three machines as the **Task Managers**:



Flink Cluster Setup

SSH configurations

In order to set up the cluster, we first need to do password less connections to the Task Manager from the Job Manager machine. The following steps needs to be performed on the Job Manager machine which creates an SSH key and copies it to `authorized_keys`:

```
$ssh-keygen
```

This will generate the public and private keys in the `/home/flinkuser/.ssh` folder. Now copy the public key to the Task Manager machine and perform the following steps on the Task Manager to allow password less connection from the Job Manager:

```
sudo mkdir -p /home/flinkuser/.ssh
sudo touch /home/flinkuser/authorized_keys
sudo cp /home/flinkuser/.ssh
sudo sh -c "cat id_rsa.pub >> /home/flinkuser/.ssh/authorized_keys"
```

Make sure the keys have restricted access by executing the following commands:

```
sudo chmod 700 /home/flinkuser/.ssh
sudo chmod 600 /home/flinkuser/.ssh/authorized_keys
```

Now you can test the password less SSH connection from the Job Manager machine:

```
sudo ssh <task-manager-1>
sudo ssh <task-manager-2>
sudo ssh <task-manager-3>
```

TIP

If you are using any cloud service instances for the installations, please make sure that the ROOT login is enabled from SSH. In order to do this, you need to login to each machine: `open file /etc/ssh/sshd_config`. Then change the value to `PermitRootLogin yes`. Once you save the file, restart the SSH service by executing the command: `sudo service sshd restart`

Java installation

Next we need to install Java on each machine. The following command will help you install Java on Redhat/CentOS based UNIX machines.

```
wget --no-check-certificate --no-cookies --header "Cookie:
oraclelicense=accept-securebackup-cookie"
http://download.oracle.com/otn-pub/java/jdk/8u92-b14/jdk-8u92-
linux-x64.rpm
sudo rpm -ivh jdk-8u92-linux-x64.rpm
```

Next we need to set up the `JAVA_HOME` environment variable so that Java is available to access from everywhere.

Create a `java.sh` file:

```
sudo vi /etc/profile.d/java.sh
```

And add following content in it and save it:

```
#!/bin/bash
JAVA_HOME=/usr/java/jdk1.8.0_92
PATH=$JAVA_HOME/bin:$PATH
export PATH JAVA_HOME
export CLASSPATH=.
```

Make the file executable and source it:

```
sudo chmod +x /etc/profile.d/java.sh
source /etc/profile.d/java.sh
```

You can now check if Java is installed properly:

```
$ java -version
java version "1.8.0_92"
Java(TM) SE Runtime Environment (build 1.8.0_92-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.92-b14, mixed mode)
```

Repeat these installations steps on the Job Manager and Task Manager machines.

Flink installation

Once SSH and Java installation is done, we need to download Flink binaries and extract them into a specific folder. Please make a note that the installation directory on all nodes should be same.

So let's get started:

```
cd /usr/local
sudo wget  http://www-eu.apache.org/dist/flink/flink-1.1.4/flink-
1.1.4-bin-hadoop27-scala_2.11.tgz
sudo tar -xzf flink-1.1.4-bin-hadoop27-scala_2.11.tgz
```

Now that the binary is ready, we need to do some configurations.

Configurations

Flink's configurations are simple. We need to tune a few parameters and we are all set. Most of the configurations are same for the Job Manager node and the Task Manager node. All configurations are done in the `conf/flink-conf.yaml` file.

The following is a configuration file for a Job Manager node:

```
jobmanager.rpc.address: localhost
jobmanager.rpc.port: 6123
jobmanager.heap.mb: 256
taskmanager.heap.mb: 512
taskmanager.numberOfTaskSlots: 1
```

You may want to change memory configurations for the Job Manager and Task Manager based on your node configurations. For the Task Manager, `jobmanager.rpc.address` should be populated with the correct Job Manager hostname or IP address.

So for all Task Managers, the configuration file should be like the following:

```
jobmanager.rpc.address: <jobmanager-ip-or-host>
jobmanager.rpc.port: 6123
jobmanager.heap.mb: 256
taskmanager.heap.mb: 512
taskmanager.numberOfTaskSlots: 1
```

We need to add the `JAVA_HOME` details in this file so that Flink knows exactly where to look for Java binaries:

```
export JAVA_HOME=/usr/java/jdk1.8.0_92
```

We also need to add the slave node details in the `conf/slaves` file, with each node on a separate new line.

Here is how a sample `conf/slaves` file should look like:

```
<task-manager-1>
<task-manager-2>
<task-manager-3>
```

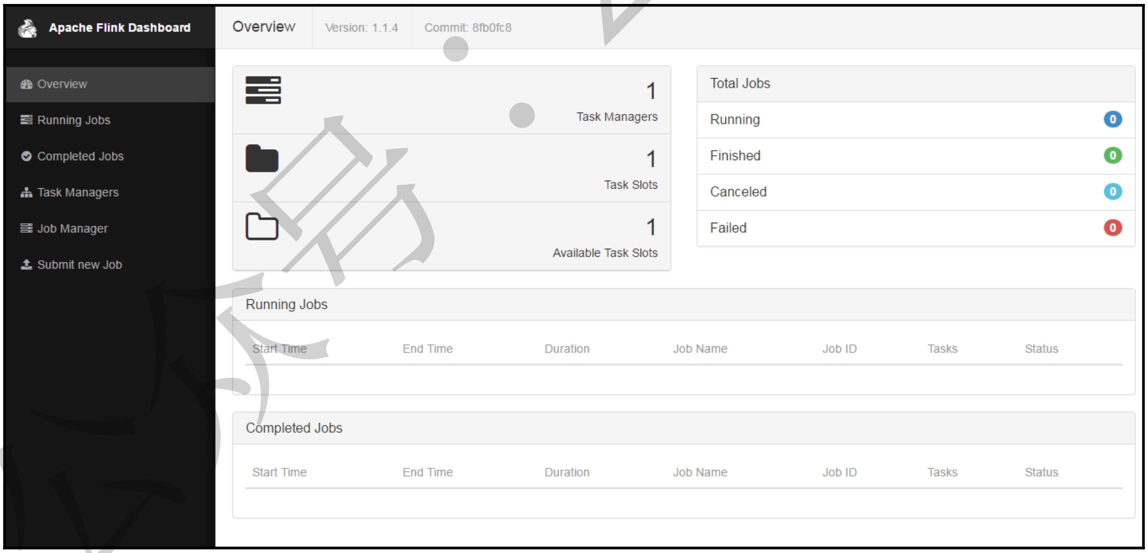
Starting daemons

Now the only thing left is starting the Flink processes. We can start each process separately on individual nodes or we can execute the `start-cluster.sh` command to start the required processes on each node:

```
bin/start-cluster.sh
```

If all the configurations are good, then you would see that the cluster is up and running. You can check the web UI at `http://<job-manager-ip>:8081/`.

The following are some snapshots of the Flink Web UI:



Overview Version: 1.1.4 Commit: 8fb0fc8

1 Task Managers

1 Task Slots

1 Available Task Slots

Total Jobs	
Running	0
Finished	0
Canceled	0
Failed	0

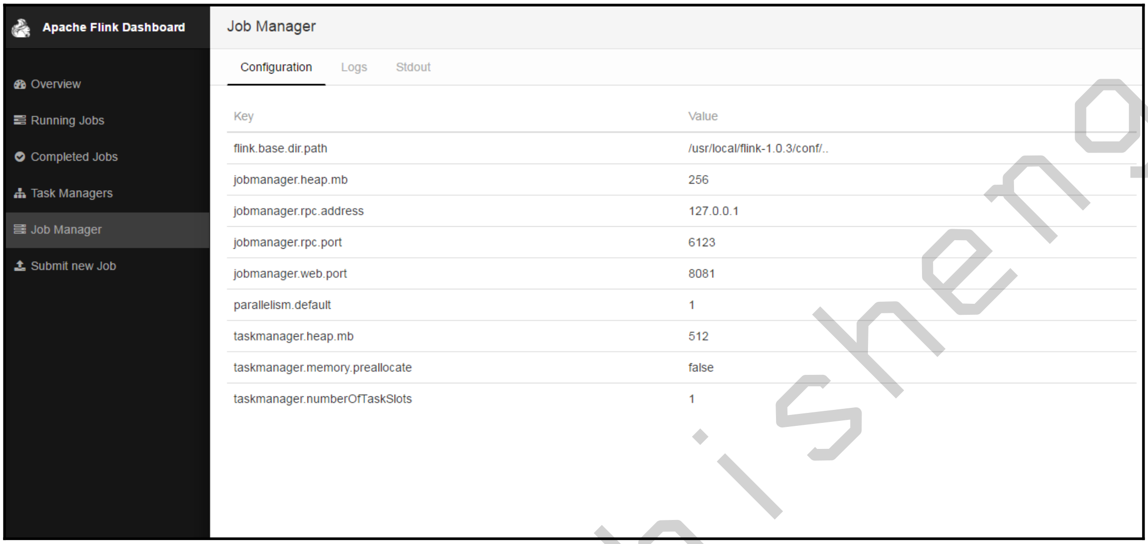
Running Jobs

Start Time	End Time	Duration	Job Name	Job ID	Tasks	Status

Completed Jobs

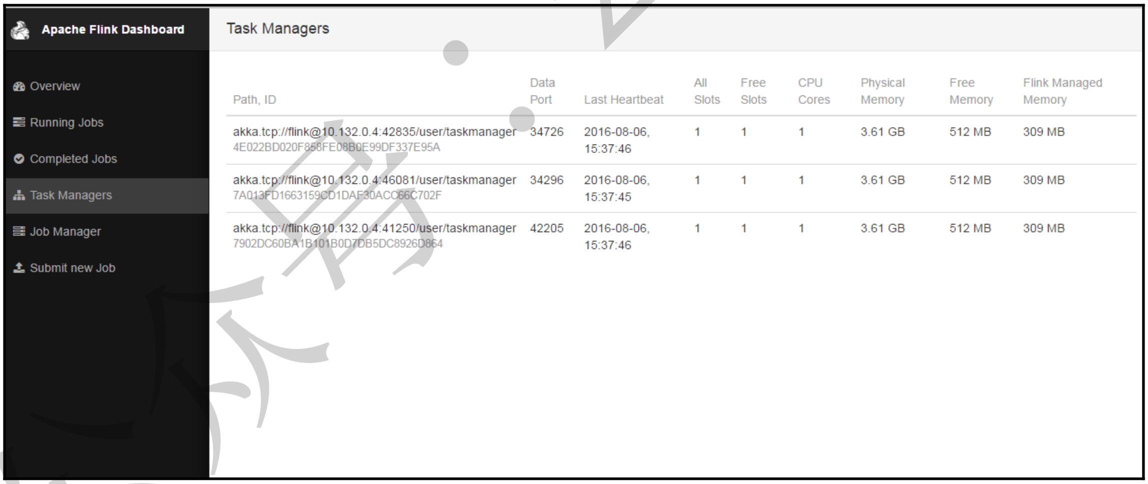
Start Time	End Time	Duration	Job Name	Job ID	Tasks	Status

You can click on the **Job Manager** link to get the following view:



Key	Value
flink.base.dir.path	/usr/local/flink-1.0.3/conf/..
jobmanager.heap.mb	256
jobmanager.rpc.address	127.0.0.1
jobmanager.rpc.port	6123
jobmanager.web.port	8081
parallelism.default	1
taskmanager.heap.mb	512
taskmanager.memory.preallocate	false
taskmanager.numberOfTaskSlots	1

Similarly, you can check out the **Task Managers** view as follows:



Path, ID	Data Port	Last Heartbeat	All Slots	Free Slots	CPU Cores	Physical Memory	Free Memory	Flink Managed Memory
akka.tcp://flink@10.132.0.4:42835/user/taskmanager 4E022BD020F858FE08B0E99DF337E95A	34726	2016-08-06, 15:37:46	1	1	1	3.61 GB	512 MB	309 MB
akka.tcp://flink@10.132.0.4:46081/user/taskmanager 7A013FD1663159CD1DAF30ACC66C702F	34296	2016-08-06, 15:37:45	1	1	1	3.61 GB	512 MB	309 MB
akka.tcp://flink@10.132.0.4:41250/user/taskmanager 7902DC60BA1B101B0D7DB5DC8926D864	42205	2016-08-06, 15:37:46	1	1	1	3.61 GB	512 MB	309 MB

Adding additional Job/Task Managers

Flink provides you with the facility to add additional instances of Job and Task Managers to the running cluster.

Before we start the daemon, please make sure that you have followed the steps given previously.

To add an additional Job Manager to the existing cluster, execute the following command:

```
sudo bin/jobmanager.sh start cluster
```

Similarly, we need to execute the following command to add an additional Task Manager:

```
sudo bin/taskmanager.sh start cluster
```

Stopping daemons and cluster

Once the job execution is completed, you want to shut down the cluster. The following commands are used for that.

To stop the complete cluster in one go:

```
sudo bin/stop-cluster.sh
```

To stop the individual Job Manager:

```
sudo bin/jobmanager.sh stop cluster
```

To stop the individual Task Manager:

```
sudo bin/taskmanager.sh stop cluster
```

Running sample application

Flink binaries come with a sample application which can be used as it is. Let's start with a very simple application, word count. Here we are going try a streaming application which reads data from the netcat server on a specific port.

So let's get started. First start the netcat server on port `9000` by executing the following command:

```
nc -l 9000
```

Now the netcat server will be start listening on port 9000 so whatever you type on the command prompt will be sent to the Flink processing.

Next we need to start the Flink sample program to listen to the netcat server. The following is the command:

```
bin/flink run examples/streaming/SocketTextStreamWordCount.jar --
hostname localhost --port 9000
08/06/2016 10:32:40     Job execution switched to status RUNNING.
08/06/2016 10:32:40     Source: Socket Stream -> Flat Map(1/1)
switched to SCHEDULED
08/06/2016 10:32:40     Source: Socket Stream -> Flat Map(1/1)
switched to DEPLOYING
08/06/2016 10:32:40     Keyed Aggregation -> Sink: Unnamed(1/1)
switched to SCHEDULED
08/06/2016 10:32:40     Keyed Aggregation -> Sink: Unnamed(1/1)
switched to DEPLOYING
08/06/2016 10:32:40     Source: Socket Stream -> Flat Map(1/1)
switched to RUNNING
08/06/2016 10:32:40     Keyed Aggregation -> Sink: Unnamed(1/1)
switched to RUNNING
```

This will start the Flink job execution. Now you can type something on the netcat console and Flink will process it.

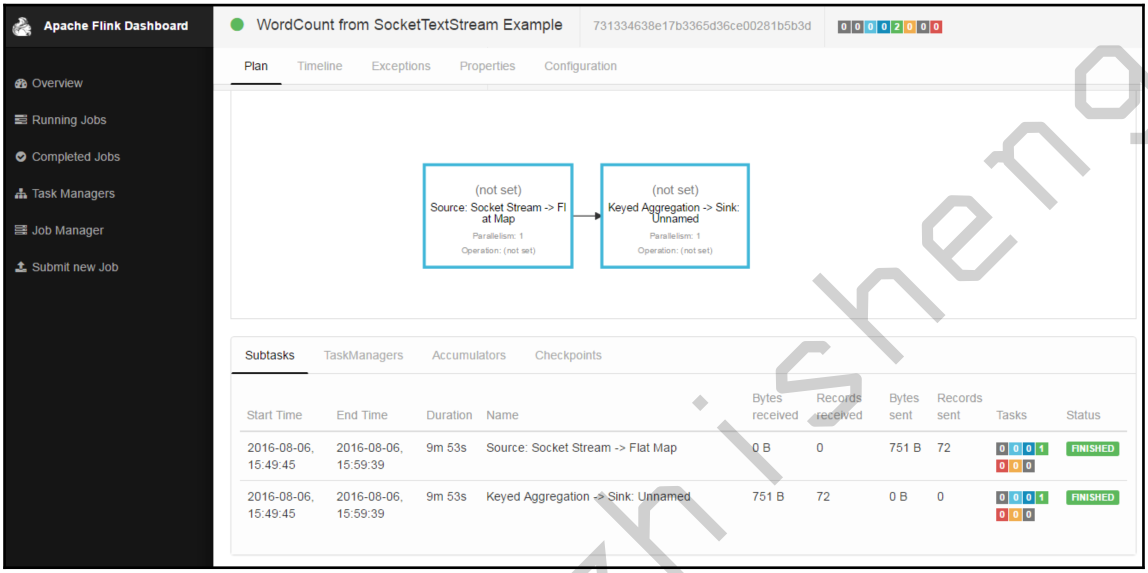
For example, type the following on the netcat server:

```
$nc -l 9000
hi Hello
Hello World
This distribution includes cryptographic software.  The country in
which you currently reside may have restrictions on the import,
possession, use, and/or re-export to another country, of
encryption software.  BEFORE using any encryption software, please
check your country's laws, regulations and policies concerning the
import, possession, or use, and re-export of encryption software,
to see if this is permitted.  See <http://www.wassenaar.org/> for
more information.
```

You can verify the output in logs:

```
$ tail -f flink-*-taskmanager-*-flink-instance-*.out
==> flink-root-taskmanager-0-flink-instance-1.out <==
(see,2)
(http,1)
(www,1)
(wassenaar,1)
(org,1)
(for,1)
(more,1)
(information,1)
(hellow,1)
(world,1)

==> flink-root-taskmanager-1-flink-instance-1.out <==
(is,1)
(permitted,1)
(see,2)
(http,1)
(www,1)
(wassenaar,1)
(org,1)
(for,1)
(more,1)
(information,1)

==> flink-root-taskmanager-2-flink-instance-1.out <==
(hello,1)
(worlds,1)
(hi,1)
(how,1)
(are,1)
(you,1)
(how,2)
(is,1)
(it,1)
(going,1)
```

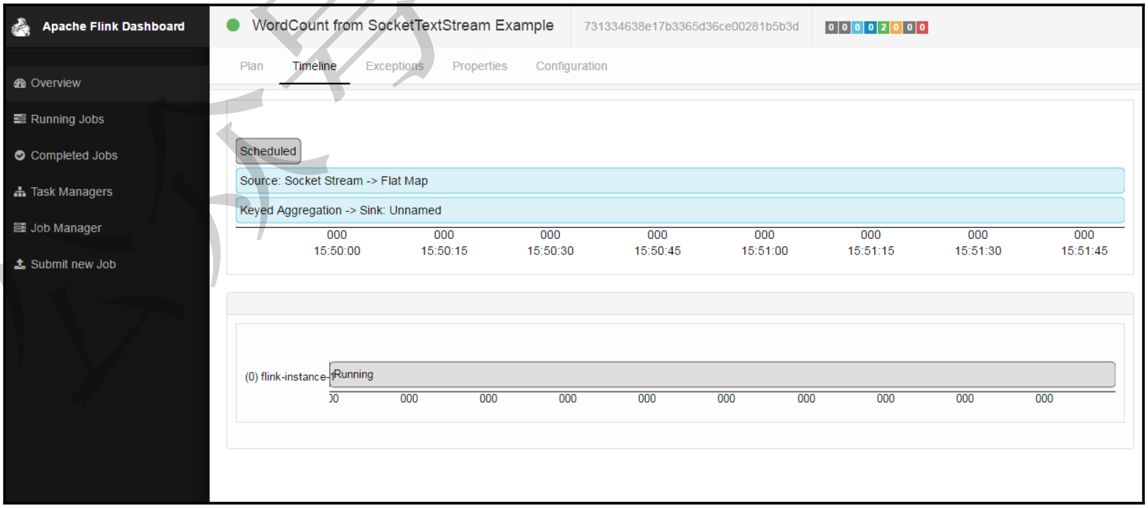
You can also checkout the Flink Web UI to see how your job is performing. The following screenshot shows the data flow plan for the execution:



Start Time	End Time	Duration	Name	Bytes received	Records received	Bytes sent	Records sent	Tasks	Status
2016-08-06, 15:49:45	2016-08-06, 15:59:39	9m 53s	Source: Socket Stream -> Flat Map	0 B	0	751 B	72	0 0 0 1 0 0 0	FINISHED
2016-08-06, 15:49:45	2016-08-06, 15:59:39	9m 53s	Keyed Aggregation -> Sink: Unnamed	751 B	72	0 B	0	0 0 0 1 0 0 0	FINISHED

Here for the job execution, Flink has two operators. The first is the source operator which reads data from the Socket stream. The second operator is the transformation operator which aggregates counts of words.

We can also look at the timeline of the job execution:



Summary

In this chapter, we talked about how Flink started as a university project and then became a full-fledged enterprise-ready data processing platform. We looked at the details of Flink's architecture and how its process model works. We also learnt how to run Flink in local and cluster modes.

In the next chapter, we are going to learn about Flink's Streaming API and look at its details and how can we use that API to solve our data streaming processing problems.

2

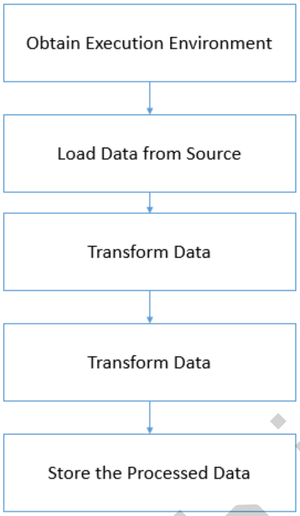
Data Processing Using the DataStream API

Real-time analytics is currently an important issue. Many different domains need to process data in real time. So far there have been multiple technologies trying to provide this capability. Technologies such as Storm and Spark have been on the market for a long time now. Applications derived from the **Internet of Things** (**IoT**) need data to be stored, processed, and analyzed in real or near real time. In order to cater for such needs, Flink provides a streaming data processing API called DataStream API.

In this chapter, we are going to look at the details relating to DataStream API, covering the following topics:

- Execution environment
- Data sources
- Transformations
- Data sinks
- Connectors
- Use case – sensor data analytics

Any Flink program works on a certain defined anatomy as follows:



Anatomy of a Flink Program

We will be looking at each step and how we can use DataStream API with this anatomy.

Execution environment

In order to start writing a Flink program, we first need to get an existing execution environment or create one. Depending upon what you are trying to do, Flink supports:

- Getting an already existing Flink environment
- Creating a local environment
- Creating a remote environment

Typically, you only need to use `getExecutionEnvironment()`. This will do the right thing based on your context. If you are executing on a local environment in an IDE then it will start a local execution environment. Otherwise, if you are executing the JAR then the Flink cluster manager will execute the program in a distributed manner.

If you want to create a local or remote environment on your own then you can also choose do so by using methods such as `createLocalEnvironment()` and `createRemoteEnvironment(String host, int port, String`, and `.jar` files).

Data sources

Sources are places where the Flink program expects to get its data from. This is a second step in the Flink program's anatomy. Flink supports a number of pre-implemented data source functions. It also supports writing custom data source functions so anything that is not supported can be programmed easily. First let's try to understand the built-in source functions.

Socket-based

DataStream API supports reading data from a socket. You just need to specify the host and port to read the data from and it will do the work:

```
socketTextStream(hostName, port);
```

You can also choose to specify the delimiter:

```
socketTextStream(hostName,port,delimiter)
```

You can also specify the maximum number of times the API should try to fetch the data:

```
socketTextStream(hostName,port,delimiter, maxRetry)
```

File-based

You can also choose to stream data from a file source using file-based source functions in Flink. You can use `readTextFile(String path)` to stream data from a file specified in the path. By default it will read `TextInputFormat` and will read strings line by line.

If the file format is other than text, you can specify the same using these functions:

```
readFile(FileInputFormat<Out> inputFormat, String path)
```

Flink also supports reading file streams as they are produced using the `readFileStream()` function:

```
readFileStream(String filePath, long intervalMillis,
FileMonitoringFunction.WatchType watchType)
```

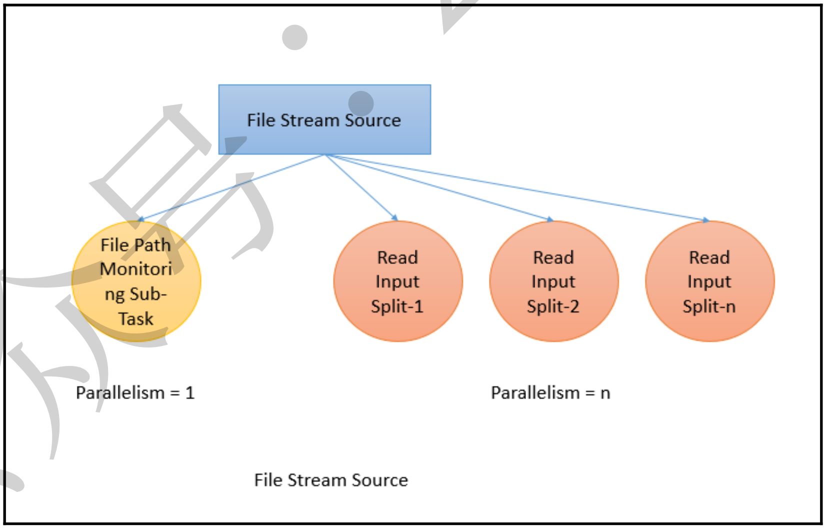
You just need to specify the file path, the polling interval in which the file path should be polled, and the watch type. Watch types consist of three types:

- `FileMonitoringFunction.WatchType.ONLY_NEW_FILES` is used when the system should process only new files
- `FileMonitoringFunction.WatchType.PROCESS_ONLY_APPENDED` is used when the system should process only appended contents of files
- `FileMonitoringFunction.WatchType.REPROCESS_WITH_APPENDED` is used when the system should re-process not only the appended contents of files but also the previous content in the file

If the file is not a text file, then we do have an option to use following function, which lets us define the file input format:

```
readFile(fileInputFormat, path, watchType, interval, pathFilter, typeInfo)
```

Internally, it divides the reading file task into two sub-tasks. One sub task only monitors the file path based on the `WatchType` given. The second sub-task does the actual file reading in parallel. The sub-task which monitors the file path is a non-parallel sub-task. Its job is to keep scanning the file path based on the polling interval and report files to be processed, split the files, and assign the splits to the respective downstream threads:



Transformations

Data transformations transform the data stream from one form into another. The input could be one or more data streams and the output could also be zero, or one or more data streams. Now let's try to understand each transformation one by one.

Map

This is one of the simplest transformations, where the input is one data stream and the output is also one data stream.

In Java:

```
inputStream.map(new MapFunction<Integer, Integer>() {
  @Override
  public Integer map(Integer value) throws Exception {
        return 5 * value;
      }
    });
```

In Scala:

```
inputStream.map { x => x * 5 }
```

FlatMap

FlatMap takes one record and outputs zero, one, or more than one record.

In Java:

```
inputStream.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public void flatMap(String value, Collector<String> out)
        throws Exception {
        for(String word: value.split(" ")){
            out.collect(word);
        }
    }
});
```

In Scala:

```
inputStream.flatMap { str => str.split(" ") }
```

Filter

Filter functions evaluate the conditions and then, if they result as true, only emit the record. Filter functions can output zero records.

In Java:

```
inputStream.filter(new FilterFunction<Integer>() {
    @Override
    public boolean filter(Integer value) throws Exception {
        return value != 1;
    }
});
```

In Scala:

```
inputStream.filter { _ != 1 }
```

KeyBy

KeyBy logically partitions the stream-based on the key. Internally it uses hash functions to partition the stream. It returns `KeyedDataStream`.

In Java:

```
inputStream.keyBy("someKey");
```

In Scala:

```
inputStream.keyBy("someKey")
```

Reduce

Reduce rolls out the `KeyedDataStream` by reducing the last reduced value with the current value. The following code does the sum reduce of a `KeyedDataStream`.

In Java:

```
keyedInputStream. reduce(new ReduceFunction<Integer>() {
    @Override
    public Integer reduce(Integer value1, Integer value2)
    throws Exception {
        return value1 + value2;
    }
});
```

In Scala:

```
keyedInputStream. reduce { _ + _ }
```

Fold

Fold rolls out the `KeyedDataStream` by combining the last folder stream with the current record. It emits a data stream back.

In Java:

```
keyedInputStream keyedStream.fold("Start", new FoldFunction<Integer,
String>() {
    @Override
    public String fold(String current, Integer value) {
        return current + "=" + value;
    }
  });
```

In Scala:

```
keyedInputStream.fold("Start")((str, i) => { str + "=" + i })
```

The preceding given function when applied on a stream of (1,2,3,4,5) would emit a stream like this: `Start=1=2=3=4=5`

Aggregations

DataStream API supports various aggregations such as `min`, `max`, `sum`, and so on. These functions can be applied on `KeyedDataStream` in order to get rolling aggregations.

In Java:

```
keyedInputStream.sum(0)
keyedInputStream.sum("key")
keyedInputStream.min(0)
keyedInputStream.min("key")
keyedInputStream.max(0)
keyedInputStream.max("key")
keyedInputStream.minBy(0)
keyedInputStream.minBy("key")
keyedInputStream.maxBy(0)
keyedInputStream.maxBy("key")
```

In Scala:

```
keyedInputStream.sum(0)
keyedInputStream.sum("key")
keyedInputStream.min(0)
keyedInputStream.min("key")
keyedInputStream.max(0)
keyedInputStream.max("key")
keyedInputStream.minBy(0)
keyedInputStream.minBy("key")
keyedInputStream.maxBy(0)
keyedInputStream.maxBy("key")
```

The difference between `max` and `maxBy` is that max returns the maximum value in a stream but `maxBy` returns a key that has a maximum value. The same applies to `min` and `minBy`.

Window

The `window` function allows the grouping of existing `KeyedDataStreams` by time or other conditions. The following transformation emits groups of records by a time window of 10 seconds.

In Java:

```
inputStream.keyBy(0).window(TumblingEventTimeWindows.of(Time.seconds(10)));
```

In Scala:

```
inputStream.keyBy(0).window(TumblingEventTimeWindows.of(Time.seconds(10)))
```

Flink defines slices of data in order to process (potentially) infinite data streams. These slices are called windows. This slicing helps processing data in chunks by applying transformations. To do windowing on a stream, we need to assign a key on which the distribution can be made and a function which describes what transformations to perform on a windowed stream.

To slice streams into windows, we can use pre-implemented Flink window assigners. We have options such as, tumbling windows, sliding windows, global and session windows. Flink also allows you to write custom window assigners by extending `WindowAssginer` class. Let's try to understand how these various assigners work.

Global windows

Global windows are never-ending windows unless specified by a trigger. Generally in this case, each element is assigned to one single per-key global Window. If we don't specify any trigger, no computation will ever get triggered.

Tumbling windows

Tumbling windows are created based on certain times. They are fixed-length windows and non over lapping. Tumbling windows should be useful when you need to do computation of elements in specific time. For example, tumbling window of 10 minutes can be used to compute a group of events occurring in 10 minutes time.

Sliding windows

Sliding windows are like tumbling windows but they are overlapping. They are fixed-length windows overlapping the previous ones by a user given window slide parameter. This type of windowing is useful when you want to compute something out of a group of events occurring in a certain time frame.

Session windows

Session windows are useful when windows boundaries need to be decided upon the input data. Session windows allows flexibility in window start time and window size. We can also provide session gap configuration parameter which indicates how long to wait before considering the session in closed.

WindowAll

The `windowAll` function allows the grouping of regular data streams. Generally this is a non-parallel data transformation as it runs on non-partitioned streams of data.

In Java:

```
inputStream.windowAll(TumblingEventTimeWindows.of(Time.seconds(10)));
```

In Scala:

```
inputStream.windowAll(TumblingEventTimeWindows.of(Time.seconds(10)))
```

Similar to regular data stream functions, we have window data stream functions as well. The only difference is they work on windowed data streams. So window reduce works like the `Reduce` function, Window fold works like the `Fold` function, and there are aggregations as well.

Union

The `Union` function performs the union of two or more data streams together. This does the combining of data streams in parallel. If we combine one stream with itself then it outputs each record twice.

In Java:

```
inputStream. union(inputStream1, inputStream2, ...);
```

In Scala:

```
inputStream. union(inputStream1, inputStream2, ...)
```

Window join

We can also join two data streams by some keys in a common window. The following example shows the joining of two streams in a Window of `5` seconds where the joining condition of the first attribute of the first stream is equal to the second attribute of the other stream.

In Java:

```
inputStream. join(inputStream1)
   .where(0).equalTo(1)
    .window(TumblingEventTimeWindows.of(Time.seconds(5)))
    .apply (new JoinFunction () {...});
```

In Scala:

```
inputStream. join(inputStream1)
    .where(0).equalTo(1)
    .window(TumblingEventTimeWindows.of(Time.seconds(5)))
    .apply { ... }
```

Split

This function splits the stream into two or more streams based on the criteria. This can be used when you get a mixed stream and you may want to process each data separately.

In Java:

```
SplitStream<Integer> split = inputStream.split(new
OutputSelector<Integer>() {
    @Override
    public Iterable<String> select(Integer value) {
        List<String> output = new ArrayList<String>();
        if (value % 2 == 0) {
            output.add("even");
        }
        else {
            output.add("odd");
        }
        return output;
    }
});
```

In Scala:

```
val split = inputStream.split(
  (num: Int) =>
    (num % 2) match {
      case 0 => List("even")
      case 1 => List("odd")
    }
)
```

Select

This function allows you to select a specific stream from the split stream.

In Java:

```
SplitStream<Integer> split;
DataStream<Integer> even = split.select("even");
DataStream<Integer> odd = split.select("odd");
DataStream<Integer> all = split.select("even","odd");
```

In Scala:

```
val even = split select "even"
val odd = split select "odd"
val all = split.select("even","odd")
```

Project

The `Project` function allows you to select a sub-set of attributes from the event stream and only sends selected elements to the next processing stream.

In Java:

```
DataStream<Tuple4<Integer, Double, String, String>> in = // [...]
DataStream<Tuple2<String, String>> out = in.project(3,2);
```

In Scala:

```
val in : DataStream[(Int,Double,String)] = // [...]
val out = in.project(3,2)
```

The preceding function selects the attribute numbers 2 and 3 from the given records. The following is the sample input and output records:

```
(1,10.0, A, B )=> (B,A)
(2,20.0, C, D )=> (D,C)
```

Physical partitioning

Flink allows us to perform physical partitioning of the stream data. You have an option to provide custom partitioning. Let us have a look at the different types of partitioning.

Custom partitioning

As mentioned earlier, you can provide custom implementation of a partitioner.

In Java:

```
inputStream.partitionCustom(partitioner, "someKey");
inputStream.partitionCustom(partitioner, 0);
```

In Scala:

```
inputStream.partitionCustom(partitioner, "someKey")
inputStream.partitionCustom(partitioner, 0)
```

While writing a custom partitioner you need make sure you implement an efficient hash function.

Random partitioning

Random partitioning randomly partitions data streams in an evenly manner.

In Java:

```
inputStream.shuffle();
```

In Scala:

```
inputStream.shuffle()
```

Rebalancing partitioning

This type of partitioning helps distribute the data evenly. It uses a round robin method for distribution. This type of partitioning is good when data is skewed.

In Java:

```
inputStream.rebalance();
```

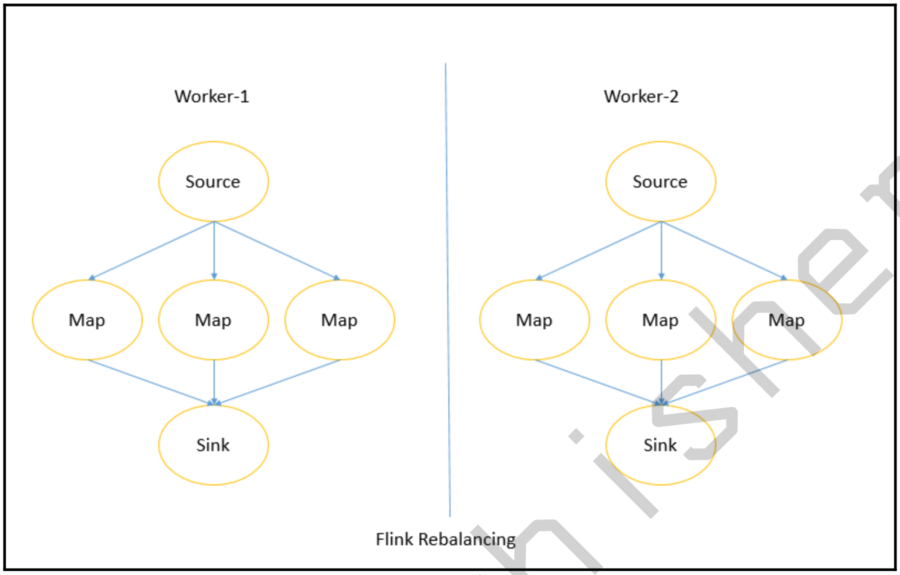
In Scala:

```
inputStream.rebalance()
```

Rescaling

Rescaling is used to distribute the data across operations, perform transformations on sub-sets of data and combine them together. This rebalancing happens over a single node only, hence it does not require any data transfer across networks.

The following diagram shows the distribution:



In Java:

```
    inputStream.rescale();
```

In Scala:

```
    inputStream.rescale()
```

Broadcasting

Broadcasting distributes all records to each partition. This fans out each and every element to all partitions.

In Java:

```
    inputStream.broadcast();
```

In Scala:

```
    inputStream.broadcast()
```

Data sinks

After the data transformations are done, we need to save results into some place. The following are some options Flink provides us to save results:

- `writeAsText()`: Writes records one line at a time as strings.
- `writeAsCsV()`: Writes tuples as comma separated value files. Row and fields delimiter can also be configured.
- `print()/printErr()`: Writes records to the standard output. You can also choose to write to the standard error.
- `writeUsingOutputFormat()`: You can also choose to provide a custom output format. While defining the custom format you need to extend the `OutputFormat` which takes care of serialization and deserialization.
- `writeToSocket()`: Flink supports writing data to a specific socket as well. It is required to define `SerializationSchema` for proper serialization and formatting.

Event time and watermarks

Flink Streaming API takes inspiration from Google Data Flow model. It supports different concepts of time for its streaming API. In general, there three places where we can capture time in a streaming environment. They are as follows

Event time

The time at which event occurred on its producing device. For example in IoT project, the time at which sensor captures a reading. Generally these event times needs to embed in the record before they enter Flink. At the time processing, these timestamps are extracted and considering for windowing. Event time processing can be used for out of order events.

Processing time

Processing time is the time of machine executing the stream of data processing. Processing time windowing considers only that timestamps where event is getting processed. Processing time is simplest way of stream processing as it does not require any synchronization between processing machines and producing machines. In distributed asynchronous environment processing time does not provide determinism as it is dependent on the speed at which records flow in the system.

Ingestion time

This is time at which a particular event enters Flink. All time based operations refer to this timestamp. Ingestion time is more expensive operation than processing but it gives predictable results. Ingestion time programs cannot handle any out of order events as it assigs timestamp only after the event is entered the Flink system.

Here is an example which shows how to set event time and watermarks. In case of ingestion time and processing time, we just need to the time characteristics and watermark generation is taken care automatically. Following is a code snippet for the same.

In Java:

```
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime);
//or
env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime);
```

In Scala:

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime)
//or
env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime)
```

In case of event time stream programs, we need to specify the way to assign watermarks and timestamps. There are two ways of assigning watermarks and timestamps:

- Directly from data source attribute
- Using a timestamp assigner

To work with event time streams, we need to assign the time characteristic as follows

In Java:

```
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime;
```

In Scala:

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
```

It is always best to store event time while storing the record in source. Flink also supports some pre-defined timestamp extractors and watermark generators. Refer to `https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/event_timestamp_extractors.html`.

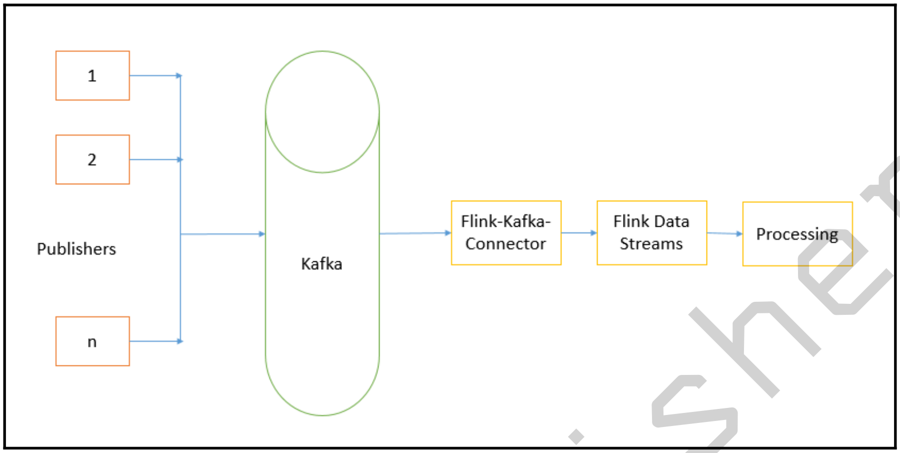
Connectors

Apache Flink supports various connectors that allow data read/writes across various technologies. Let's learn more about this.

Kafka connector

Kafka is a publish-subscribe, distributed, message queuing system that allows users to publish messages to a certain topic; this is then distributed to the subscribers of the topic. Flink provides options to define a Kafka consumer as a data source in Flink Streaming. In order to use the Flink Kafka connector, we need to use a specific JAR file.

The following diagram shows how the Flink Kafka connector works:



We need to use the following Maven dependency to use the connector. I have been using Kafka version 0.9 so I will be adding the following dependency in pom.xml:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka-0.9_2.11/artifactId>
  <version>1.1.4</version>
</dependency>
```

Now let's try to understand how to use the Kafka consumer as the Kafka source:

In Java:

```
Properties properties = new Properties();
  properties.setProperty("bootstrap.servers", "localhost:9092");
  properties.setProperty("group.id", "test");
DataStream<String> input  = env.addSource(new
FlinkKafkaConsumer09<String>("mytopic", new SimpleStringSchema(),
properties));
```

In Scala:

```
val properties = new Properties();
properties.setProperty("bootstrap.servers", "localhost:9092");
// only required for Kafka 0.8
properties.setProperty("zookeeper.connect", "localhost:2181");
properties.setProperty("group.id", "test");
stream = env
    .addSource(new FlinkKafkaConsumer09[String]("mytopic", new
SimpleStringSchema(), properties))
    .print
```

In the preceding code, we first set the properties of the Kafka host and the zookeeper host and port. Next we need to specify the topic name, in this case `mytopic`. So if any messages get published to the `mytopic` topic, they will be processed by the Flink streams.

If you get data in a different format, then you can also specify your custom schema for deserialization. By default, Flink supports string and JSON deserializers.

In order to enable fault tolerance, we need to enable checkpointing in Flink. Flink is keen on taking snapshots of the state in a periodic manner. In the case of failure, it will restore to the last checkpoint and then restart the processing.

We can also define the Kafka producer as a sink. This will write the data to a Kafka topic. The following is a way to write data to a Kafka topic:

In Scala:

```
stream.addSink(new FlinkKafkaProducer09<String>("localhost:9092",
"mytopic", new SimpleStringSchema()));
```

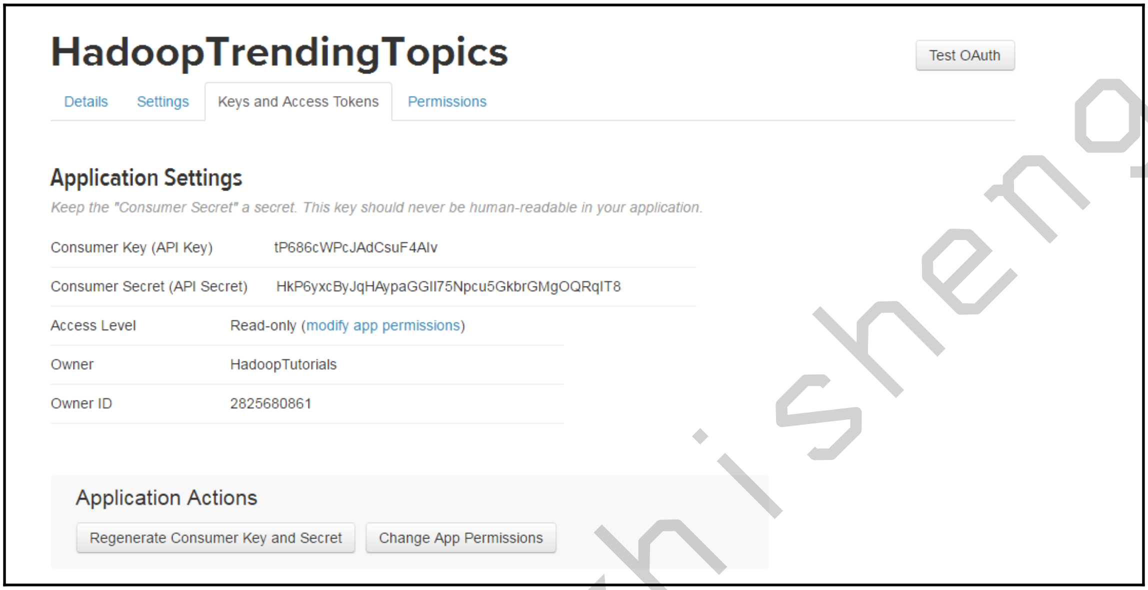
In Java:

```
stream.addSink(new FlinkKafkaProducer09[String]("localhost:9092",
"mytopic", new SimpleStringSchema()))
```

Twitter connector

These days, it is very important to have the ability to fetch data from Twitter and process it. Many companies use Twitter data for doing sentiment analytics for various products, services, movies, reviews, and so on. Flink provides the Twitter connector as one data source. To use the connector, you need to have a Twitter account. Once you have a Twitter account, you need to create a Twitter application and generate authentication keys to be used by the connector. Here is a link that will help you to generate tokens: `https://dev.twitter.com/oauth/overview/application-owner-access-tokens`.

The Twitter connector can be used through the Java or Scala API:



Application Settings

Keep the "Consumer Secret" a secret. This key should never be human-readable in your application.

Consumer Key (API Key)	tP686cWPcJAdCsuF4Alv
Consumer Secret (API Secret)	HkP6yxcByJqHAypaGGIl75Npcu5GkbrGMgOQRqIT8
Access Level	Read-only (modify app permissions)
Owner	HadoopTutorials
Owner ID	2825680861

Application Actions

Once tokens are generated, we can start writing the program to fetch data from Twitter. First we need to add a Maven dependency:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-twitter_2.11/artifactId>
  <version>1.1.4</version>
</dependency>
```

Next we add Twitter as a data source. The following is the sample code:

In Java:

```
Properties props = new Properties();
props.setProperty(TwitterSource.CONSUMER_KEY, "");
props.setProperty(TwitterSource.CONSUMER_SECRET, "");
props.setProperty(TwitterSource.TOKEN, "");
props.setProperty(TwitterSource.TOKEN_SECRET, "");
DataStream<String> streamSource = env.addSource(new TwitterSource(props));
```

In Scala:

```
val props = new Properties();
props.setProperty(TwitterSource.CONSUMER_KEY, "");
props.setProperty(TwitterSource.CONSUMER_SECRET, "");
props.setProperty(TwitterSource.TOKEN, "");
props.setProperty(TwitterSource.TOKEN_SECRET, "");
DataStream<String> streamSource = env.addSource(new TwitterSource(props));
```

In the preceding code, we first set properties for the token we got. And then we add the `TwitterSource`. If the given information is correct then you will start fetching the data from Twitter. `TwitterSource` emits the data in a JSON string format. A sample Twitter JSON looks like the following:

```
{
...
"text": ""Loyalty 3.0: How to Revolutionize Customer &amp; Employee
Engagement with Big Data &amp; #Gamification" can be ordered here:
http://t.co/1XhqyaNjuR",
  "geo": null,
  "retweeted": false,
  "in_reply_to_screen_name": null,
  "possibly_sensitive": false,
  "truncated": false,
  "lang": "en",
    "hashtags": [{
      "text": "Gamification",
      "indices": [90,
      103]
    }],
  },
  "in_reply_to_status_id_str": null,
  "id": 330094515484508160
...
}
```

`TwitterSource` provides various endpoints. By default it uses `StatusesSampleEndpoint`, which returns a set of random tweets. If you need to add some filters and don't want to use the default endpoint, you can implement the `TwitterSource.EndpointInitializer` interface.

Now that we know how to fetch data from Twitter, we can then decide what to do with this data depending upon our use case. We can process, store, or analyze the data.

RabbitMQ connector

RabbitMQ is a widely used distributed, high-performance, message queuing system. It is used as a message delivery system for high throughput operations. It allows you to create a distributed message queue and include publishers and subscribers in the queue. More reading on RabbitMQ can be done at following link `https://www.rabbitmq.com/`

Flink supports fetching and publishing data to and from RabbitMQ. It provides a connector that can act as a data source of data streams.

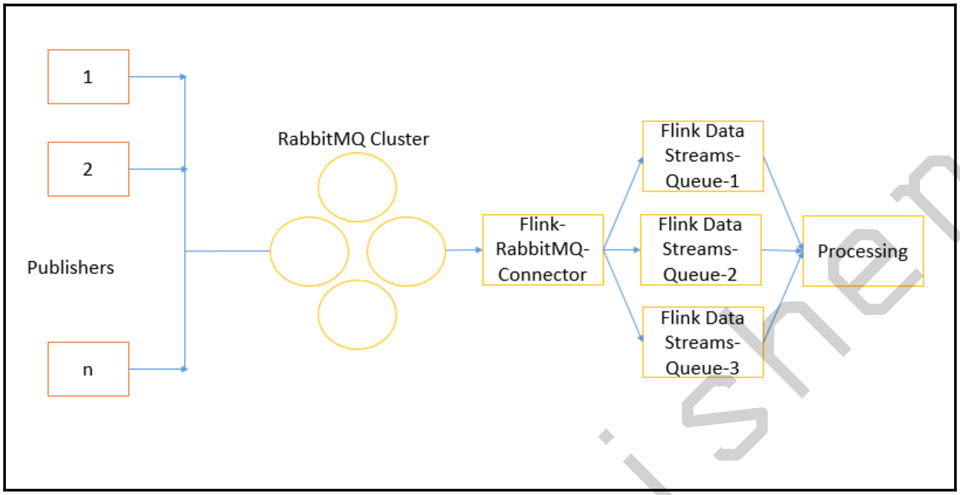
For the RabbitMQ connector to work, we need to provide following information:

- RabbitMQ configurations such as host, port, user credentials, and so on.
- Queue, the RabbitMQ queue name which you wish to subscribe.
- Correlation IDs is a RabbitMQ feature used for correlating the request and response by a unique ID in a distributed world. The Flink RabbitMQ connector provides an interface to set this to true or false depending on whether you are using it or not.
- Deserialization schema–RabbitMQ stores and transports the data in a serialized manner to avoid network traffic. So when the message is received, the subscriber should know how to deserialize the message. The Flink connector provides us with some default deserializers such as the string deserializer.

RabbitMQ source provides us with the following options on stream deliveries:

- Exactly once: Using RabbitMQ correlation IDs and the Flink check-pointing mechanism with RabbitMQ transactions
- At-least once: When Flink checkpointing is enabled but RabbitMQ correlation IDs are not set
- No strong delivery guarantees with the RabbitMQ auto-commit mode

Here is a diagram to help you understand the RabbitMQ connector in better manner:


1
2
Publishers
n
RabbitMQ Cluster
Flink-RabbitMQ-Connector
Flink Data Streams-Queue-1
Flink Data Streams-Queue-2
Flink Data Streams-Queue-3
Processing

Now let's look at how to write code to get this connector working. Like other connectors, we need to add a Maven dependency to the code:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-rabbitmq_2.11/artifactId>
  <version>1.1.4</version>
</dependency>
```

The following snippet shows how to use the RabbitMQ connector in Java:

```
//Configurations
RMQConnectionConfig connectionConfig = new RMQConnectionConfig.Builder()
.setHost(<host>).setPort(<port>).setUserName(..)
.setPassword(..).setVirtualHost("/").build();

//Get Data Stream without correlation ids
DataStream<String> streamWO = env.addSource(new
RMQSource<String>(connectionConfig, "my-queue", new SimpleStringSchema()))
  .print
//Get Data Stream with correlation ids
DataStream<String> streamW = env.addSource(new
RMQSource<String>(connectionConfig, "my-queue", true, new
SimpleStringSchema()))
  .print
```

Similarly, in Scala the code can written as follows:

```
val connectionConfig = new RMQConnectionConfig.Builder()
.setHost(<host>).setPort(<port>).setUserName(..)
.setPassword(..).setVirtualHost("/").build()
streamsWOIds = env
    .addSource(new RMQSource[String](connectionConfig, " my-queue", new
SimpleStringSchema))
    .print

streamsWIds = env
    .addSource(new RMQSource[String](connectionConfig, "my-queue", true,
new SimpleStringSchema))
    .print
```

We can also use the RabbitMQ connector as a Flink sink. If you want to send processes back to some different RabbitMQ queue, the following is a way to do so. We need to provide three important configurations:

- RabbitMQ configurations
- Queue name–Where to send back the processed data
- Serialization schema–Schema for RabbitMQ to convert the data into bytes

The following is sample code in Java to show how to use this connector as a Flink sink:

```
RMQConnectionConfig connectionConfig = new RMQConnectionConfig.Builder()
.setHost(<host>).setPort(<port>).setUserName(..)
.setPassword(..).setVirtualHost("/").build();
stream.addSink(new RMQSink<String>(connectionConfig, "target-queue", new
StringToByteSerializer()));
```

The same can be done in Scala:

```
val connectionConfig = new RMQConnectionConfig.Builder()
.setHost(<host>).setPort(<port>).setUserName(..)
.setPassword(..).setVirtualHost("/").build()
stream.addSink(new RMQSink[String](connectionConfig, "target-queue", new
StringToByteSerializer
```

ElasticSearch connector

ElasticSearch is a distributed, low-latency, full text search engine system that allows us to index documents of our choice and then allows us to do a full text search over the set of documents. More on ElasticSearch can be read here at `https://www.elastic.co/`.

In many use cases, you may want to process data using Flink and then store it in ElasticSearch. To enable this, Flink supports the ElasticSearch connector. So far, ElasticSearch has had two major releases. Flink supports them both.

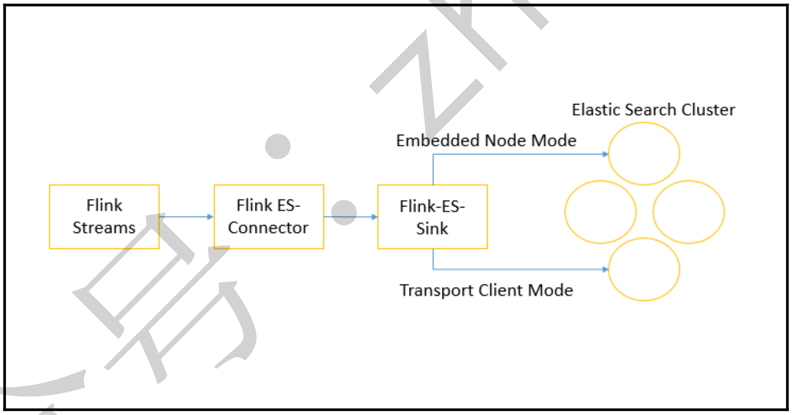
For ElasticSearch 1.X, the following Maven dependency needs to be added:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-elasticsearch_2.11</artifactId>
  <version>1.1.4</version>
</dependency>
```

The Flink connector provides a sink to write data to ElasticSearch. It uses two methods to connect to ElasticSearch:

- Embedded node
- Transport client

The following diagram illustrates this:



Embedded node mode

In the embedded node mode, the sink uses BulkProcessor to send the documents to ElasticSearch. We can configure how many requests to buffer before sending documents to ElasticSearch.

The following is the code snippet:

```
DataStream<String> input = ...;

Map<String, String> config = Maps.newHashMap();
config.put("bulk.flush.max.actions", "1");
config.put("cluster.name", "cluster-name");

input.addSink(new ElasticsearchSink<>(config, new
IndexRequestBuilder<String>() {
    @Override
    public IndexRequest createIndexRequest(String element, RuntimeContext
ctx) {
        Map<String, Object> json = new HashMap<>();
        json.put("data", element);

        return Requests.indexRequest()
                .index("my-index")
                .type("my-type")
                .source(json);
    }
}));
```

In the preceding code snippet, we create a hash map with configurations such as the cluster name and how many documents to buffer before sending the request. Then we add the sink to the stream, specifying the index, type, and the document to store. Similar code in Scala follows:

```
val input: DataStream[String] = ...

val config = new util.HashMap[String, String]
config.put("bulk.flush.max.actions", "1")
config.put("cluster.name", "cluster-name")

text.addSink(new ElasticsearchSink(config, new IndexRequestBuilder[String]
{
  override def createIndexRequest(element: String, ctx: RuntimeContext):
IndexRequest = {
    val json = new util.HashMap[String, AnyRef]
    json.put("data", element)
    Requests.indexRequest.index("my-index").`type`("my-type").source(json)
  }
}))
```

Transport client mode

ElasticSearch allows connections through the transport client on port 9300. Flink supports connecting using those through its connector. The only thing we need to mention here is all the ElasticSearch nodes present in the cluster in configurations.

The following is the snippet in Java:

```
DataStream<String> input = ...;

Map<String, String> config = Maps.newHashMap();
config.put("bulk.flush.max.actions", "1");
config.put("cluster.name", "cluster-name");

List<TransportAddress> transports = new ArrayList<String>();
transports.add(new InetSocketTransportAddress("es-node-1", 9300));
transports.add(new InetSocketTransportAddress("es-node-2", 9300));
transports.add(new InetSocketTransportAddress("es-node-3", 9300));

input.addSink(new ElasticsearchSink<>(config, transports, new
IndexRequestBuilder<String>() {
    @Override
    public IndexRequest createIndexRequest(String element, RuntimeContext
ctx) {
        Map<String, Object> json = new HashMap<>();
        json.put("data", element);

        return Requests.indexRequest()
                .index("my-index")
                .type("my-type")
                .source(json);
    }
}));
```

Here as well we provide the details about the cluster name, nodes, ports, maximum requests to send in bulk, and so on. Similar code in Scala can be written as follows:

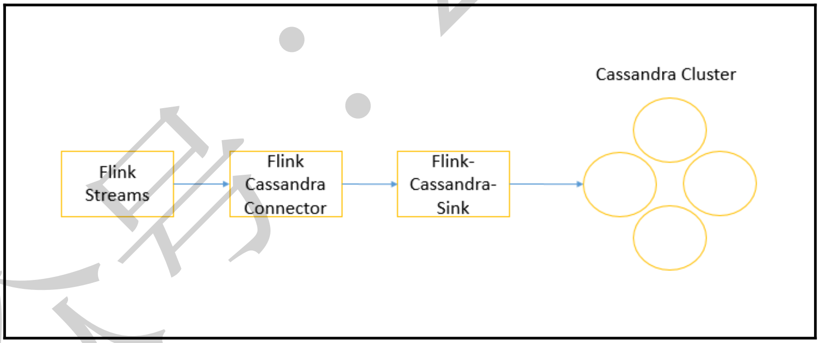
```
val input: DataStream[String] = ...

val config = new util.HashMap[String, String]
config.put("bulk.flush.max.actions", "1")
config.put("cluster.name", "cluster-name")

val transports = new ArrayList[String]
transports.add(new InetSocketTransportAddress("es-node-1", 9300))
transports.add(new InetSocketTransportAddress("es-node-2", 9300))
transports.add(new InetSocketTransportAddress("es-node-3", 9300))
```

```
text.addSink(new ElasticsearchSink(config, transports, new
IndexRequestBuilder[String] {
  override def createIndexRequest(element: String, ctx: RuntimeContext):
IndexRequest = {
    val json = new util.HashMap[String, AnyRef]
    json.put("data", element)
    Requests.indexRequest.index("my-index").`type`("my-type").source(json)
  }
}))
```

Cassandra connector

Cassandra is a distributed, low latency, NoSQL database. It is a key value-based database. Many high throughput applications use Cassandra as their primary database. Cassandra works with a distributed cluster mode, where there is no master-slave architecture. Reads and writes can be felicitated by any node. More on Cassandra can be found at: `http://cassandra.apache.org/`.

Apache Flink provides a connector which can write data to Cassandra. In many applications, people may want to store streaming data from Flink into Cassandra. The following diagram shows a simple design of the Cassandra sink:



Like other connectors, to get this we need to add it as a maven dependency:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-cassandra_2.11</artifactId>
  <version>1.1.4</version>
</dependency>
```

Once the dependency is added, we just need to add the Cassandra sink with its configurations, as follows:

In Java:

```
CassandraSink.addSink(input)
  .setQuery("INSERT INTO cep.events (id, message) values (?, ?);")
  .setClusterBuilder(new ClusterBuilder() {
    @Override
    public Cluster buildCluster(Cluster.Builder builder) {
      return builder.addContactPoint("127.0.0.1").build();
    }
  })
  .build()
```

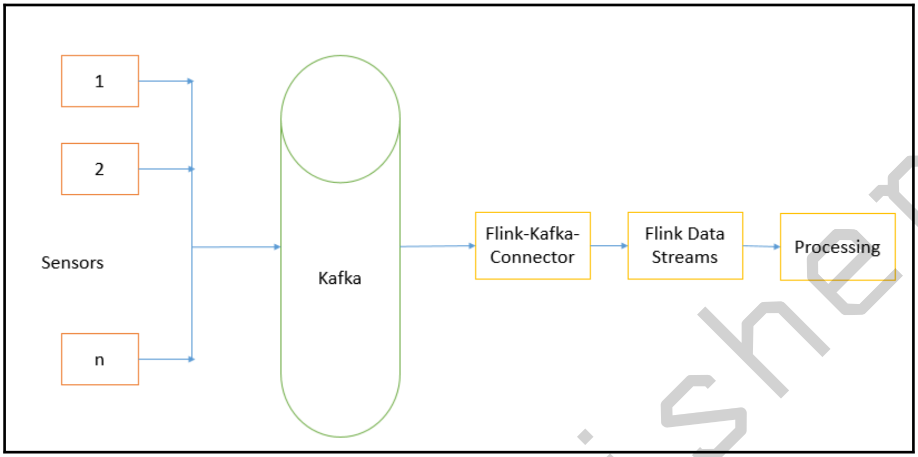
The preceding code writes stream of data into a table called **events**. The table expects an event ID and a message. Similarly in Scala:

```
CassandraSink.addSink(input)
  .setQuery("INSERT INTO cep.events (id, message) values (?, ?);")
  .setClusterBuilder(new ClusterBuilder() {
    @Override
    public Cluster buildCluster(Cluster.Builder builder) {
      return builder.addContactPoint("127.0.0.1").build();
    }
  })
  .build();
```

Use case – sensor data analytics

Now that we have looked at various aspects of DataStream API, let's try to use these concepts to solve a real world use case. Consider a machine which has sensor installed on it and we wish to collect data from these sensors and calculate average temperature per sensor every five minutes.

Following would be the architecture:



In this scenario, we assume that sensors are sending information to Kafka topic called **temp** with information as (timestamp, temperature, sensor-ID). Now we need to write code to read data from Kafka topics and processing it using Flink transformation.

Here important thing to consider is as we already have timestamp values coming from sensor, we can use Event Time computations for time factors. This means we would be able to take care of events even if they reach out of order.

We start with simple streaming execution environment which will be reading data from Kafka. Since we have timestamps in events, we will be writing a custom timestamp and watermark extractor to read the timestamp values and do window processing based on that. Here is code snippet for the same.

```
// set up the streaming execution environment
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
// env.enableCheckpointing(5000);
nv.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
Properties properties = new Properties();
properties.setProperty("bootstrap.servers", "localhost:9092");

properties.setProperty("zookeeper.connect", "localhost:2181");
properties.setProperty("group.id", "test");

FlinkKafkaConsumer09<String> myConsumer = new
FlinkKafkaConsumer09<>("temp", new SimpleStringSchema(),
                      properties);
myConsumer.assignTimestampsAndWatermarks(new CustomWatermarkEmitter());
```

Here we assume that we receive events in Kafka topics as strings and in the format:

```
Timestamp,Temperature,Sensor-Id
```

The following an example code to extract timestamp from record:

```
public class CustomWatermarkEmitter implements
AssignerWithPunctuatedWatermarks<String> {
    private static final long serialVersionUID = 1L;

    @Override
    public long extractTimestamp(String arg0, long arg1) {
        if (null != arg0 && arg0.contains(",")) {
           String parts[] = arg0.split(",");
           return Long.parseLong(parts[0]);
           }
          return 0;
    }
    @Override
    public Watermark checkAndGetNextWatermark(String arg0, long arg1) {
        if (null != arg0 && arg0.contains(",")) {
            String parts[] = arg0.split(",");
            return new Watermark(Long.parseLong(parts[0]));
        }
        return null;
    }
}
```

Now we simply created keyed data stream and perform average calculation on temperature values as shown in the following code snippet:

```
DataStream<Tuple2<String, Double>> keyedStream =
env.addSource(myConsumer).flatMap(new Splitter()).keyBy(0)
.timeWindow(Time.seconds(300))
.apply(new WindowFunction<Tuple2<String, Double>, Tuple2<String, Double>,
Tuple, TimeWindow>() {
    @Override
    public void apply(Tuple key, TimeWindow window,
    Iterable<Tuple2<String, Double>> input,
    Collector<Tuple2<String, Double>> out) throws Exception {
        double sum = 0L;
            int count = 0;
            for (Tuple2<String, Double> record : input) {
                sum += record.f1;
                count++;
            }
     Tuple2<String, Double> result = input.iterator().next();
     result.f1 = (sum/count);
```

```
        out.collect(result);
      }
    });
```

When execute the preceding given code, and if proper sensor events are published on Kafka topics then we will get the average temperature per sensor every five minutes.

The complete code is available on GitHub at `https://github.com/deshpandetanmay/mastering-flink/tree/master/chapter02/flink-streaming`.

Summary

In this chapter, we started with Flink's most powerful API: DataStream API. We looked at how data sources, transformations, and sinks work together. Then we looked at various technology connectors such as ElasticSearch, Cassandra, Kafka, RabbitMQ, and so on.

At the end, we also tried to apply our learning to solve a real-world sensor data analytics use case.

In the next chapter, we are going to learn about another very important API from Flink's ecosystem point of view the DataSet API.

3

Data Processing Using the Batch Processing API

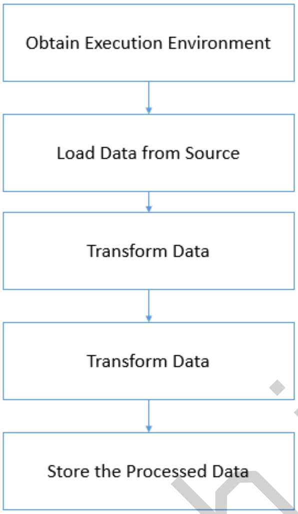
Even though many people appreciate the potential value of streaming data processing in most industries, there are many use cases where people don't feel it is necessary to process the data in a streaming manner. In all such cases, batch processing is the way to go. So far Hadoop has been the default choice for data processing. However, Flink also supports batch data processing by DataSet API.

For Flink, batch processing is a special case of stream processing. Here is a very interesting article explaining this thought in detail at `http://data-artisans.com/batch-is-a-special-case-of-streaming/`.

In this chapter, we are going to look at the details regarding DataSet API. This includes the following topics:

- Data sources
- Transformations
- Data sinks
- Connectors

As we learnt in the previous chapter, any Flink program works on a certain defined anatomy as follows:



Anatomy of a Flink Program

The DataSet API is not an exception to this flow. We will look at each step in detail. We already discussed in the previous chapter how to obtain the execution environment. So we will directly move to the details of data sources supported by DataSet API.

Data sources

Sources are places where the DataSet API expects to get its data from. It could in the form of a file or from Java collections. This is the second step in the Flink program's anatomy. DataSet API supports a number of pre-implemented data source functions. It also supports writing custom data source functions so anything that is not supported can be programmed easily. First let's try to understand the built-in source functions.

File-based

Flink supports reading data from files. It reads data line by line and returns it as strings. The following are built-in functions you can use to read data:

- `readTextFile(Stringpath)`: This reads data from a file specified in the path. By default it will read `TextInputFormat` and will read strings line by line.
- `readTextFileWithValue(Stringpath)`: This reads data from a file specified in the path. It returns `StringValues`. `StringValues` are mutable strings.
- `readCsvFile(Stringpath)`: This reads data from comma separated files. It returns the Java POJOs or tuples.
- `readFileofPremitives(path, delimiter, class)`: This parses the new line into primitive data types such as strings or integers.
- `readHadoopFile(FileInputFormat, Key, Value, path)`: This reads files from a specified path with the given `FileInputFormat`, `Key` class and `Value` class. It returns the parsed values as tuples `Tuple2<Key,Value>`.
- `readSequenceFile(Key, Value, path)`: This reads files from a specified path with the given `SequenceFileInputFormat`, `Key` class and `Value` class. It returns the parsed values as tuples `Tuple2<Key,Value>`.

> For file-based inputs, Flink supports the recursive traversal of folders specified in a given path. In order to use this facility, we need to set an environment variable and pass it as a parameter while reading the data. The variable to set is `recursive.file.enumeration`. We need to set this variable to `true` in order to enable recursive traversal.

Collection-based

With Flink DataSet API, we can also read data from Java-based collections. The following are some functions we can use to read the data:

- `fromCollection(Collection)`: This creates a dataset from Java-based collections.
- `fromCollection(Iterator, Class)`: This creates a dataset from an iterator. The elements of the iterator are of a type given by the class parameter.
- `fromElements(T)`: This creates a dataset of a sequence of objects. The object type is specified in the function itself.

- `fromParallelCollection(SplittableIterator, Class)`: This creates a dataset from the iterator in parallel. Class represents the object types.
- `generateSequence(from, to)`: This generates the sequence of numbers between given limits.

Generic sources

DataSet API supports a couple of generic functions to read data:

- `readFile(inputFormat, path)`: This creates a dataset of the type `FileInputFormat` from a given path
- `createInput(inputFormat)`: This creates a dataset of the generic input format

Compressed files

Flink supports the decompression of files while reading if they are marked with proper extensions. We don't need to do any different configurations to read the compressed files. If a file with a proper extension is detected then Flink automatically decompresses it and sends it for further processing.

One thing to note here is that decompression of files cannot be done in parallel so this might take a bit of time before the actual data processing starts.

At this stage, it is recommended to avoid using compressed files as decompression is not a scalable activity in Flink.

The following are supported algorithms:

Compression algorithm	**Extension**	**Is parallel?**
Gzip	`.gz`, `.gzip`	No
Deflate	`.deflate`	No

Transformations

Data transformations transform the dataset from one form into another. The input could be one or more datasets and the output could also be zero, or one or more data streams. Now let's try to understand each transformation one by one.

Map

This is one of the simplest transformations where the input is one dataset and output is also one dataset.

In Java:

```
inputSet.map(new MapFunction<Integer, Integer>() {
  @Override
  public Integer map(Integer value) throws Exception {
        return 5 * value;
      }
    });
```

In Scala:

```
inputSet.map { x => x * 5 }
```

In Python:

```
inputSet.map { lambda x : x * 5 }
```

Flat map

The flat map takes one record and outputs zero, or one or more than one records.

In Java:

```
inputSet.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public void flatMap(String value, Collector<String> out)
        throws Exception {
        for(String word: value.split(" ")){
            out.collect(word);
        }
    }
});
```

In Scala:

```
inputSet.flatMap { str => str.split(" ") }
```

In Python:

```
inputSet.flat_map {lambda str, c:[str.split() for line in str }
```

Filter

Filter functions evaluate the conditions and then if returned `true` only emit the record. Filter functions can output zero records.

In Java:

```
inputSet.filter(new FilterFunction<Integer>() {
    @Override
    public boolean filter(Integer value) throws Exception {
        return value != 1;
    }
});
In Scala:
inputSet.filter { _ != 1 }
```

In Python:

```
inputSet.filter {lambda x: x != 1 }
```

Project

Project transformations remove or move the elements of a tuple into another. This can be used to do selective processing on specific elements.

In Java:

```
DataSet<Tuple3<Integer, String, Double>> in = // [...]
DataSet<Tuple2<String, Integer>> out = in.project(1,0);
```

In Scala, this transformation is not supported.

In Python:

```
inputSet.project(1,0)
```

Reduce on grouped datasets

Reduce transformations reduce each group into a single element based on the user-defined reduce function.

In Java:

```
public class WC {
  public String word;
  public int count;
}

//Reduce function
public class WordCounter implements ReduceFunction<WC> {
  @Override
  public WC reduce(WC in1, WC in2) {
    return new WC(in1.word, in1.count + in2.count);
  }
}

// [...]
DataSet<WC> words = // [...]
DataSet<WC> wordCounts = words
                         // grouping on field "word"
                         .groupBy("word")
                         // apply ReduceFunction on grouped DataSet
                         .reduce(new WordCounter());
```

In Scala:

```
class WC(val word: String, val count: Int) {
  def this() {
    this(null, -1)
  }
}

val words: DataSet[WC] = // [...]
val wordCounts = words.groupBy("word").reduce {
  (w1, w2) => new WC(w1.word, w1.count + w2.count)
}
```

In Python, the code is not supported.

Reduce on grouped datasets by field position key

For datasets with tuples, we can also group by the field positions. The following is an example.

In Java:

```
DataSet<Tuple3<String, Integer, Double>> reducedTuples = tuples
                           // group by on second and third field
                            .groupBy(1, 2)
                            // apply ReduceFunction
                            .reduce(new MyTupleReducer());
```

In Scala:

```
val reducedTuples = tuples.groupBy(1, 2).reduce { ... }
```

In Python:

```
reducedTuples = tuples.group_by(1, 2).reduce( ... )
```

Group combine

In some applications, it is important to do intermediate operations before doing some more transformations. Group combine operations can be very handy in this case. Intermediate transformations could be reducing the size and so on.

This is performed in memory with a greedy strategy that gets performed in multiple steps.

In Java:

```
DataSet<String> input = [..]

  DataSet<Tuple2<String, Integer>> combinedWords = input
  .groupBy(0); // group similar words
  .combineGroup(new GroupCombineFunction<String, Tuple2<String,
   Integer>() {

    public void combine(Iterable<String> words,
    Collector<Tuple2<String, Integer>>) { // combine
        String key = null;
        int count = 0;

        for (String word : words) {
            key = word;
            count++;
        }
        // emit tuple with word and count
        out.collect(new Tuple2(key, count));
    }
});
```

In Scala:

```
val input: DataSet[String] = [..]

val combinedWords: DataSet[(String, Int)] = input
  .groupBy(0)
  .combineGroup {
    (words, out: Collector[(String, Int)]) =>
        var key: String = null
        var count = 0

        for (word <- words) {
            key = word
            count += 1
        }
        out.collect((key, count))
}
```

In Python, this code is not supported.

Aggregate on a grouped tuple dataset

Aggregate transformations are very common. We can easily perform common aggregations such as `sum`, `min`, and `max` on tuple datasets. The following is the way we do it.

In Java:

```
DataSet<Tuple3<Integer, String, Double>> input = // [...]
DataSet<Tuple3<Integer, String, Double>> output = input
             .groupBy(1)        // group DataSet on second field
             .aggregate(SUM, 0) // compute sum of the first field
             .and(MIN, 2);      // compute minimum of the third field
```

In Scala:

```
val input: DataSet[(Int, String, Double)] = // [...]
val output = input.groupBy(1).aggregate(SUM, 0).and(MIN, 2)
```

In Python:

```
input = # [...]
output = input.group_by(1).aggregate(Sum, 0).and_agg(Min, 2)
```

Please note here that in DataSet API, if we need to apply multiple aggregations, we need to use the `and` keyword.

MinBy on a grouped tuple dataset

The `minBy` function selects a single tuple from each group of tuple datasets for which the value is the minimum. The fields used for comparison must be comparable.

In Java:

```
DataSet<Tuple3<Integer, String, Double>> input = // [...]
DataSet<Tuple3<Integer, String, Double>> output = input
                    .groupBy(1)   // group by on second field
                    .minBy(0, 2); // select tuple with minimum values for
first and third field.
```

In Scala:

```
val input: DataSet[(Int, String, Double)] = // [...]
val output: DataSet[(Int, String, Double)] = input
            .groupBy(1)
            .minBy(0, 2)
```

In Python, this code is not supported.

MaxBy on a grouped tuple dataset

The `MaxBy` function selects a single tuple from each group of tuple datasets for which the value is the maximum. The fields used for comparison must be comparable.

In Java:

```
DataSet<Tuple3<Integer, String, Double>> input = // [...]
DataSet<Tuple3<Integer, String, Double>> output = input
                    .groupBy(1)   // group by on second field
                    .maxBy(0, 2); // select tuple with maximum values for
                                  /*first and third field. */
```

In Scala:

```
val input: DataSet[(Int, String, Double)] = // [...]
val output: DataSet[(Int, String, Double)] = input
.groupBy(1)
.maxBy(0, 2)
```

In Python, this code is not supported.

Reduce on full dataset

The reduce transformation allows for the application of a user-defined function on a full dataset. Here is an example.

In Java:

```
public class IntSumReducer implements ReduceFunction<Integer> {
  @Override
  public Integer reduce(Integer num1, Integer num2) {
    return num1 + num2;
  }
}

DataSet<Integer> intNumbers = // [...]
DataSet<Integer> sum = intNumbers.reduce(new IntSumReducer());
```

In Scala:

```
val sum = intNumbers.reduce (_ + _)
```

In Python:

```
sum = intNumbers.reduce(lambda x,y: x + y)
```

Group reduce on a full dataset

The group reduce transformation allows for the application of a user-defined function on a full dataset. Here is an example.

In Java:

```
DataSet<Integer> input = // [...]
DataSet<Integer> output = input.reduceGroup(new MyGroupReducer());
```

In Scala:

```
val input: DataSet[Int] = // [...]
val output = input.reduceGroup(new MyGroupReducer())
```

In Python:

```
output = data.reduce_group(MyGroupReducer())
```

Aggregate on a full tuple dataset

We can run common aggregation functions on full datasets. So far Flink supports `MAX`, `MIN`, and `SUM`.

In Java:

```
DataSet<Tuple2<Integer, Double>> output = input
.aggregate(SUM, 0) // SUM of first field
.and(MIN, 1); // Minimum of second
```

In Scala:

```
val input: DataSet[(Int, String, Double)] = // [...]
val output = input.aggregate(SUM, 0).and(MIN, 2)
```

In Python:

```
output = input.aggregate(Sum, 0).and_agg(Min, 2)
```

MinBy on a full tuple dataset

The `MinBy` function selects a single tuple from the full dataset for which the value is the minimum. The fields used for comparison must be comparable.

In Java:

```
DataSet<Tuple3<Integer, String, Double>> input = // [...]
DataSet<Tuple3<Integer, String, Double>> output = input
                  .minBy(0, 2); // select tuple with minimum values for
                                first and third field.
```

In Scala:

```
val input: DataSet[(Int, String, Double)] = // [...]
val output: DataSet[(Int, String, Double)] = input
.minBy(0, 2)
```

In Python, this code is not supported.

MaxBy on a full tuple dataset

`MaxBy` selects a single tuple full dataset for which the value is maximum. The fields used for comparison must be comparable.

In Java:

```
DataSet<Tuple3<Integer, String, Double>> input = // [...]
DataSet<Tuple3<Integer, String, Double>> output = input
                 .maxBy(0, 2); // select tuple with maximum values for
first and third field.
```

In Scala:

```
val input: DataSet[(Int, String, Double)] = // [...]
val output: DataSet[(Int, String, Double)] = input
                                   .maxBy(0, 2)
```

In Python, this code is not supported.

Distinct

The distinct transformation emits distinct values from the source dataset. This is used for removing duplicate values from the source.

In Java:

```
DataSet<Tuple2<Integer, Double>> output = input.distinct();
```

In Scala:

```
val output = input.distinct()
```

In Python, this code is not supported.

Join

The join transformation joins two datasets into one dataset. The joining condition can be defined as one of the keys from each dataset.

In Java:

```
public static class Student { public String name; public int deptId; }
public static class Dept { public String name; public int id; }
DataSet<Student> input1 = // [...]
DataSet<Dept> input2 = // [...]
DataSet<Tuple2<Student, Dept>>
            result = input1.join(input2)
.where("deptId")
.equalTo("id");
```

In Scala:

```
val input1: DataSet[(String, Int)] = // [...]
val input2: DataSet[(String, Int)] = // [...]
val result = input1.join(input2).where(1).equalTo(1)
```

In Python

```
result = input1.join(input2).where(1).equal_to(1)
```

There are various other ways in which two datasets can be joined. Here is a link where you can read more about all such joining options: https://ci.apache.org/projects/flink/flink-docs-master/dev/batch/dataset_transformations.html#join.

Cross

The cross transformation does the cross product of two datasets by applying a user-defined function.

In Java:

```
DataSet<Class> input1 = // [...]
DataSet<class> input2 = // [...]
DataSet<Tuple3<Integer, Integer, Double>>
            result =
            input1.cross(input2)
                   // applying CrossFunction
                   .with(new MyCrossFunction());
```

In Scala:

```
val result = input1.cross(input2) {
//custom function
}
```

In Python:

```
result = input1.cross(input2).using(MyCrossFunction())
```

Union

The union transformation combines two similar datasets. We can also union multiple datasets in one go.

In Java:

```
DataSet<Tuple2<String, Integer>> input1 = // [...]
DataSet<Tuple2<String, Integer>> input2 = // [...]
DataSet<Tuple2<String, Integer>> input3 = // [...]
DataSet<Tuple2<String, Integer>> unioned =
input1.union(input2).union(input3);
```

In Scala:

```
val input1: DataSet[(String, Int)] = // [...]
val input2: DataSet[(String, Int)] = // [...]
val input3: DataSet[(String, Int)] = // [...]
val unioned = input1.union(input2).union(input3)
```

In Python:

```
unioned = input1.union(input2).union(input3)
```

Rebalance

This transformation evenly rebalances parallel partitions. This helps in achieving better performance as it helps in removing data skews.

In Java:

```
DataSet<String> in = // [...]
DataSet<Tuple2<String, String>> out = in.rebalance();
```

In Scala:

```
val in: DataSet[String] = // [...]
val out = in.rebalance()
```

In Python, this code is not supported.

Hash partition

This transformation partitions the dataset on a given key.

In Java:

```
DataSet<Tuple2<String, Integer>> in = // [...]
DataSet<Tuple2<String, String>> out = in.partitionByHash(1);
```

In Scala:

```
val in: DataSet[(String, Int)] = // [...]
val out = in.partitionByHash(1)
```

In Python, this code is not supported.

Range partition

This transformation range partitions the dataset on a given key.

In Java:

```
DataSet<Tuple2<String, Integer>> in = // [...]
DataSet<Tuple2<String, String>> out = in.partitionByRange(1);
```

In Scala:

```
val in: DataSet[(String, Int)] = // [...]
val out = in.partitionByRange(1)
```

In Python, this code is not supported.

Sort partition

This transformation locally sorts the partitions dataset on a given key and in the given order.

In Java:

```
DataSet<Tuple2<String, Integer>> in = // [...]
DataSet<Tuple2<String, String>> out = in.sortPartition(1,Order.ASCENDING);
```

In Scala:

```
val in: DataSet[(String, Int)] = // [...]
val out = in.sortPartition(1, Order.ASCENDING)
```

In Python, this code is not supported.

First-n

This transformation arbitrarily returns the first-n elements of the dataset.

In Java:

```
DataSet<Tuple2<String, Integer>> in = // [...]
// Returns first 10 elements of the data set.
DataSet<Tuple2<String, String>> out = in.first(10);
```

In Scala:

```
val in: DataSet[(String, Int)] = // [...]
val out = in.first(10)
```

In Python, this code is not supported.

Broadcast variables

Broadcast variables allow user to access certain dataset as collection to all operators. Generally, broadcast variables are used when we you want to refer a small amount of data frequently in a certain operation. Those who are familiar with Spark broadcast variables will be able use the same feature in Flink as well.

We just need to broadcast a dataset with a specific name and it will be available on each executors handy. The broadcast variables are kept in memory so we have to be cautious in using them. The following code snippet shows how to broadcast a dataset and use it as needed.

```
// Get a data set to be broadcasted
DataSet<Integer> toBroadcast = env.fromElements(1, 2, 3);
DataSet<String> data = env.fromElements("India", "USA", "UK").map(new
RichMapFunction<String, String>() {
    private List<Integer> toBroadcast;
    // We have to use open method to get broadcast set from the context
    @Override
    public void open(Configuration parameters) throws Exception {
    // Get the broadcast set, available as collection
    this.toBroadcast =
    getRuntimeContext().getBroadcastVariable("country");
    }

    @Override
    public String map(String input) throws Exception {
          int sum = 0;
          for (int a : toBroadcast) {
                sum = a + sum;
          }
          return input.toUpperCase() + sum;
    }
}).withBroadcastSet(toBroadcast, "country"); // Broadcast the set with name
data.print();
```

Broadcast variables are useful when we have look up conditions to be used for transformation and the lookup dataset is comparatively small.

Data sinks

After the data transformations are done, we need to save the results somewhere. The following are some options that Flink DataSet API provides to save the results:

- `writeAsText()`: This writes records one line at a time as strings.
- `writeAsCsV()`: This writes tuples as comma-separated value files. Row and field delimiters can also be configured.

- `print()`/`printErr()`: This writes records to the standard output. You can also choose to write to a standard error.
- `write()`: This supports writing data in a custom `FileOutputFormat`.
- `output()`: This is used for datasets which are not file-based. This can be used where we want to write data to some database.

Connectors

Apache Flink's DataSet API supports various connectors, allowing data read/writes across various systems. Let's try to explore more on this.

Filesystems

Flink allows connecting to various distributed filesystems such as HDFS, S3, Google Cloud Storage, Alluxio, and so on, by default. In this section, we will see how to connect to these filesystems.

In order to connect to these systems, we need to add the following dependency in `pom.xml`:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-hadoop-compatibility_2.11</artifactId>
  <version>1.1.4</version>
</dependency>
```

This allows us to use Hadoop data types, input formats, and output formats. Flink supports writable and writable comparable out-of-the-box, so we don't need the compatibility dependency for that.

HDFS

To read data from an HDFS file, we create a data source using the `readHadoopFile()` or `createHadoopInput()` method. In order to use this connector, we first need to configure `flink-conf.yaml` and set `fs.hdfs.hadoopconf` to the proper Hadoop configuration directory.

The resulting dataset would be a tuple of the type that matches with the HDFS data types. The following code snippet shows how to do this.

In Java:

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
DataSet<Tuple2<LongWritable, Text>> input =
    env.readHadoopFile(new TextInputFormat(), LongWritable.class,
Text.class, textPath);
```

In Scala:

```
val env = ExecutionEnvironment.getExecutionEnvironment
val input: DataSet[(LongWritable, Text)] =
  env.readHadoopFile(new TextInputFormat, classOf[LongWritable],
classOf[Text], textPath)
```

We can also use this connector to write back the processed data to HDFS. The `OutputFormat` wrapper expects the dataset to be in `Tuple2` format. The following code snippet shows how to write back processed data to HDFS.

In Java:

```
// Get the processed data set
DataSet<Tuple2<Text, IntWritable>> results = [...]

// Set up the Hadoop Output Format.
HadoopOutputFormat<Text, IntWritable> hadoopOF =
  // create the Flink wrapper.
  new HadoopOutputFormat<Text, IntWritable>(
    // set the Hadoop OutputFormat and specify the job.
    new TextOutputFormat<Text, IntWritable>(), job
  );
hadoopOF.getConfiguration().set("mapreduce.output.textoutputformat.separato
r", " ");
TextOutputFormat.setOutputPath(job, new Path(outputPath));

// Emit data
result.output(hadoopOF);
```

In Scala:

```
// Get the processed data set
val result: DataSet[(Text, IntWritable)] = [...]

val hadoopOF = new HadoopOutputFormat[Text,IntWritable](
  new TextOutputFormat[Text, IntWritable],
  new JobConf)
```

```
hadoopOF.getJobConf.set("mapred.textoutputformat.separator", " ")
FileOutputFormat.setOutputPath(hadoopOF.getJobConf, new Path(resultPath))

result.output(hadoopOF)
```

Amazon S3

As stated earlier, Flink supports reading data from Amazon S3 by default. But we need to do some configurations in Hadoop's `core-site.xml`. We need to set the following properties:

```
<!-- configure the file system implementation -->
<property>
  <name>fs.s3.impl</name>
  <value>org.apache.hadoop.fs.s3native.NativeS3FileSystem</value>
</property>
<!-- set your AWS ID -->
<property>
  <name>fs.s3.awsAccessKeyId</name>
  <value>putKeyHere</value>
</property>
<!-- set your AWS access key -->
<property>
  <name>fs.s3.awsSecretAccessKey</name>
  <value>putSecretHere</value>
</property>
```

Once done, we can access the S3 filesystem as shown here:

```
// Read from S3 bucket
env.readTextFile("s3://<bucket>/<endpoint>");
// Write to S3 bucket
stream.writeAsText("s3://<bucket>/<endpoint>");
```

Alluxio

Alluxio is an open source, memory speed virtual distributed storage. Many companies have been using Alluxio for high-speed data storage and processing. You can read more about Alluxio at: `http://www.alluxio.org/`.

Flink supports reading data from Alluxio by default. But we need to do some configurations in Hadoop `core-site.xml`. We need to set the following properties:

```
<property>
  <name>fs.alluxio.impl</name>
  <value>alluxio.hadoop.FileSystem</value>
</property>
```

Once done, we can access the Alluxio filesystem as shown here:

```
// Read from Alluxio path
env.readTextFile("alluxio://<path>");

// Write to Alluxio path
stream.writeAsText("alluxio://<path>");
```

Avro

Flink has built-in support for Avro files. It allows easy reads and writes to Avro files. In order to read Avro files, we need to use `AvroInputFormat`. The following code snippet shows how to read Avro files:

```
AvroInputFormat<User> users = new AvroInputFormat<User>(in, User.class);
DataSet<User> userSet = env.createInput(users);
```

Once the dataset is ready we can easily perform various transformations, such as:

```
userSet.groupBy("city")
```

Microsoft Azure storage

Microsoft Azure Storage is a cloud-based storage that allows storing data in a durable and scalable manner. Flink supports managing data stored on Microsoft Azure table storage. The following explains how we do this.

First of all, we need to download the `azure-tables-hadoop` project from `git` and then compile it:

```
git clone https://github.com/mooso/azure-tables-hadoop.git
cd azure-tables-hadoop
mvn clean install
```

Next we add the following dependencies in `pom.xml`:

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-hadoop-compatibility_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
<dependency>
  <groupId>com.microsoft.hadoop</groupId>
  <artifactId>microsoft-hadoop-azure</artifactId>
  <version>0.0.4</version>
</dependency>
```

Next we write the following code to access Azure storage:

```
final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();

    // create a  AzureTableInputFormat, using a Hadoop input format wrapper
    HadoopInputFormat<Text, WritableEntity> hdIf = new
HadoopInputFormat<Text, WritableEntity>(new AzureTableInputFormat(),
Text.class, WritableEntity.class, new Job());

// set account URI
hdIf.getConfiguration().set(AzureTableConfiguration.Keys.ACCOUNT_URI.getKey
(), "XXXX");
    // set the secret storage key
hdIf.getConfiguration().set(AzureTableConfiguration.Keys.STORAGE_KEY.getKey
(), "XXXX");
    // set the table name
hdIf.getConfiguration().set(AzureTableConfiguration.Keys.TABLE_NAME.getKey(
), "XXXX");

 DataSet<Tuple2<Text, WritableEntity>> input = env.createInput(hdIf);
```

Now we are all set do any processing of the dataset.

MongoDB

Through open source contributions, developers have been able to connect Flink to MongoDB. In this section, we are going to talk about one such project.

The project is open source and can be downloaded from GitHub:

```
git clone https://github.com/okkam-it/flink-mongodb-test.git
cd flink-mongodb-test
mvn clean install
```

Next we use the preceding connector in the Java program to connect to MongoDB:

```
// set up the execution environment
    final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();

// create a MongodbInputFormat, using a Hadoop input format wrapper
HadoopInputFormat<BSONWritable, BSONWritable> hdIf =
        new HadoopInputFormat<BSONWritable, BSONWritable>(new
MongoInputFormat(),
       BSONWritable.class, BSONWritable.class, new JobConf());

// specify connection parameters
hdIf.getJobConf().set("mongo.input.uri",
                "mongodb://localhost:27017/dbname.collectioname");

DataSet<Tuple2<BSONWritable, BSONWritable>> input = env.createInput(hdIf);
```

Once the data is available as a dataset, we can easily do the desired transformation. We can also write back the data to the MongoDB collection as shown here:

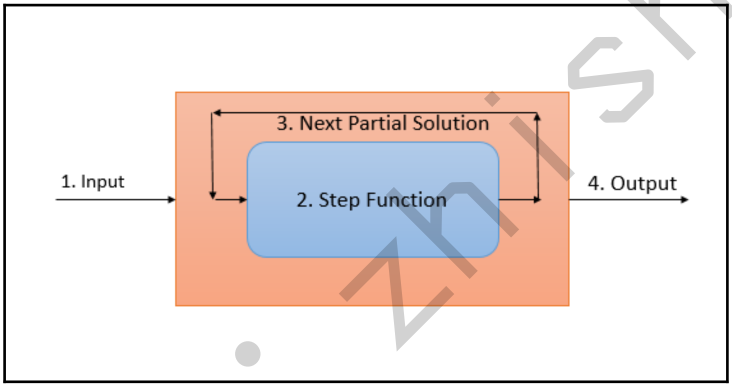
```
MongoConfigUtil.setOutputURI( hdIf.getJobConf(),
                "mongodb://localhost:27017/dbname.collectionname ");
 // emit result (this works only locally)
 result.output(new HadoopOutputFormat<Text,BSONWritable>(
                new MongoOutputFormat<Text,BSONWritable>(),
hdIf.getJobConf()));
```

Iterations

One of the unique features Flink supports is iterations. These days a lot developers want to run iterative machine-learning and graph-processing algorithms using big data technologies. To cater to these needs, Flink supports running iterative data processing by defining a step function.

Iterator operator

An iterator operator consists of the following components:



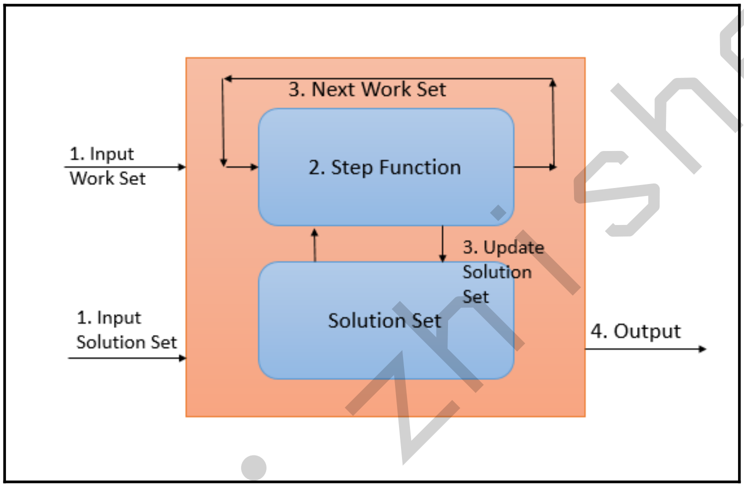
- **Iteration Input**: This is either the initial dataset received or the output of the previous iteration
- **Step Function**: This is the function that needs to be applied on input dataset
- **Next Partial Solution**: This is the output of the step function which needs to be fed back to the next iteration
- **Iteration Result**: After all iterations are completed, we get the result of iterations

The number of iterations can be controlled by various ways. One way could be setting up the number of iterations to perform or we can also put conditional termination.

Delta iterator

The delta operator iterates over the set of elements for incremental iteration operations. The main difference between that the delta iterator and regular iterator is, delta iterator works on updating the solution set rather than fully re-computing it every iteration.

This leads to more efficient operations as it allows us to focus on the important parts of the solution in less time. The following diagram shows flow of delta iterator in Flink.



- **Iteration Input**: We have to read the work set and solution set for the delta iterator from some files
- **Step Function**: Step function is the function that needs to be applied on the input dataset
- **Next Work Set/ Update Solution**: Here after every iteration solution set it is updated with the latest results and the next work set is fed to the next iteration
- **Iteration Result**: After all iterations are completed, we get the result of the iterations in the form of a solution set

Since delta iterators work on hot dataset itself, the performance and efficiency are great. Here is a detailed article that talks about using Flink iterations for the PageRank algorithm. `http://data-artisans.com/data-analysis-with-flink-a-case-study-and-tutorial/`.

Use case – Athletes data insights using Flink batch API

Now that we have learnt the details of DataSet API, let's try to apply this knowledge to a real-life use case. Let's say we have a dataset with us, which has information about the Olympics athletes and their performance in various games. The sample data looks like the following table:

Player	Country	Year	Game	Gold	Silver	Bronze	Total
Yang Yilin	China	2008	Gymnastics	1	0	2	3
Leisel Jones	Australia	2000	Swimming	0	2	0	2
Go Gi-Hyeon	South Korea	2002	Short-Track Speed Skating	1	1	0	2
Chen Ruolin	China	2008	Diving	2	0	0	2
Katie Ledecky	United States	2012	Swimming	1	0	0	1
Ruta Meilutyte	Lithuania	2012	Swimming	1	0	0	1
DAiniel Gyurta	Hungary	2004	Swimming	0	1	0	1
Arianna Fontana	Italy	2006	Short-Track Speed Skating	0	0	1	1
Olga Glatskikh	Russia	2004	Rhythmic Gymnastics	1	0	0	1
Kharikleia Pantazi	Greece	2000	Rhythmic Gymnastics	0	0	1	1
Kim Martin	Sweden	2002	Ice Hockey	0	0	1	1
Kyla Ross	United States	2012	Gymnastics	1	0	0	1
Gabriela Dragoi	Romania	2008	Gymnastics	0	0	1	1
Tasha Schwikert-Warren	United States	2000	Gymnastics	0	0	1	1

Now we want to get answers to the questions such as, How many players per country participated in the games? Or how many players participated for each game? As the data is at rest, we will be using Flink Batch API to analyze it.

The data available is in the CSV format. So we will be using a CSV reader provided by Flink API as shown in the following code snippet.

```
final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();
DataSet<Record> csvInput = env.readCsvFile("olympic-athletes.csv")
                     .pojoType(Record.class, "playerName", "country",
"year", "game", "gold", "silver", "bronze", "total");
```

Once the data is parse properly, it is easy to move ahead and use it as required. The following code snippet shows how to get information of no. of players per country:

```
DataSet<Tuple2<String, Integer>> groupedByCountry = csvInput
.flatMap(new FlatMapFunction<Record, Tuple2<String, Integer>>() {
private static final long serialVersionUID = 1L;
@Override
public void flatMap(Record record, Collector<Tuple2<String, Integer>> out)
throws Exception {
out.collect(new Tuple2<String, Integer>(record.getCountry(), 1));
}
}).groupBy(0).sum(1);
groupedByCountry.print();
```

In the preceding code snippet, we are first creating datasets with the key as the player's country and value as `1` and then we group it and sum up the values to get the total count. Once we execute the code, here is how the output looks:

```
(Australia,11)
(Belarus,7)
(China,25)
(France,3)
(Germany,2)
(Italy,4)
(Turkey,1)
(United States,22)
(Cameroon,2)
(Hungary,1)
(Kenya,1)
(Lithuania,1)
(Russia,23)
(Spain,2)
(Ukraine,1)
(Chinese Taipei,2)
(Great Britain,1)
(Romania,14)
(Switzerland,1)
(Bulgaria,3)
(Finland,1)
```

```
(Greece,7)
(Japan,1)
(Mexico,1)
(Netherlands,2)
(Poland,1)
(South Korea,6)
(Sweden,6)
(Thailand,1)
```

Similarly we can apply the same logic to find the number of players per game as shown in the following code snippet:

```
DataSet<Tuple2<String, Integer>> groupedByGame = csvInput
.flatMap(new FlatMapFunction<Record, Tuple2<String, Integer>>() { private
static final long serialVersionUID = 1L;
@Override
public void flatMap(Record record, Collector<Tuple2<String, Integer>> out)
throws Exception {
out.collect(new Tuple2<String, Integer>(record.getGame(), 1));
}
}).groupBy(0).sum(1);
groupedByGame.print();
```

The output of the preceding code snippet would look as follows:

```
(Basketball,1)
(Gymnastics,42)
(Ice Hockey,7)
(Judo,1)
(Swimming,33)
(Athletics,2)
(Fencing,2)
(Nordic Combined,1)
(Rhythmic Gymnastics,27)
(Short-Track Speed Skating,5)
(Table Tennis,1)
(Weightlifting,4)
(Boxing,3)
(Taekwondo,3)
(Archery,3)
(Diving,14)
(Figure Skating,1)
(Football,2)
(Shooting,1)
```

This way you can run various other transformations to get the desired output. The complete code for this use case is available at `https://github.com/deshpandetanmay/mastering-flink/tree/master/chapter03/flink-batch`.

Summary

In this chapter, we learnt about DataSet API. It enabled us to do the batch processing. We learnt various transformations in order to do data processing. Later we also explored the various file-based connectors to read/write data from HDFS, Amazon S3, MS Azure, Alluxio, and so on.

In the last section, we looked a use case where we applied the knowledge learnt in the earlier sections.

In the next chapter, we are going to learn another very important API from Flink's ecosystem point of view that is, Table API.

4

Data Processing Using the Table API

In the earlier chapters, we talked about batch and stream data processing APIs provided by Apache Flink. In this chapter, we are going to talk about Table API which is a SQL interface for data processing in Flink. Table API operates on a table interface which can be created from a dataset and datastream. Once the dataset/datastream is registered as a table, we are free to apply relational operations such as aggregations, joins, and selections.

Tables can also be queried like regular SQL queries. Once the operations are performed, we need to convert the table back to either a dataset or datastream. Apache Flink internally uses another open source project called Apache Calcite `https://calcite.apache.org/` for optimizing these query transformations.

In this chapter, we are going to cover the following topics:

- Registering tables
- Accessing the registered table
- Operators
- Data types
- SQL

Now let's get started.

In order to use Table API, the very first thing we need to do is to create a Java Maven project and add the following dependency in it:

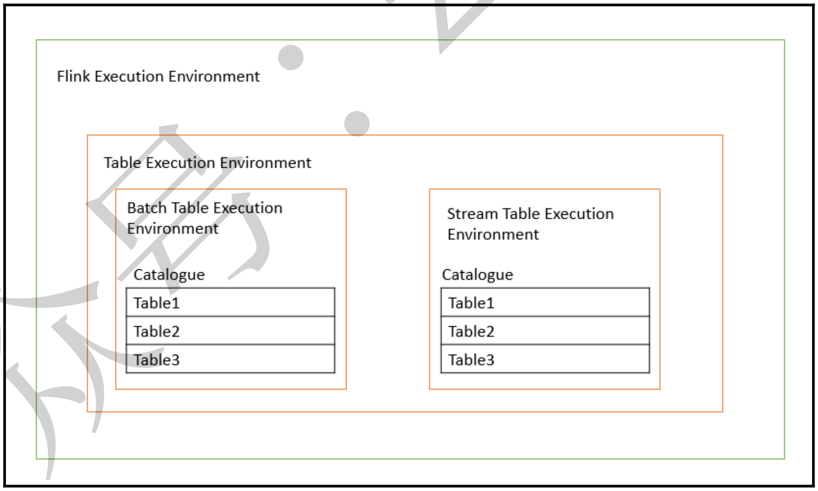
```
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-table_2.11</artifactId>
        <version>1.1.4</version>
      </dependency>
```

This dependency will download all the required JARs in your class path. Once the download is complete, we are all good to use Table API.

Registering tables

In order to operate on datasets/datastreams, first we need to register a table in `TableEnvironment`. Once the table is registered with a unique name, it can be easily accessed from `TableEnvironment`.

`TableEnvironment` maintains an internal table catalogue for table registration. The following diagram shows the details:


Flink Execution Environment
Table Execution Environment
Batch Table Execution Environment
Catalogue
Table1
Table2
Table3
Stream Table Execution Environment
Catalogue
Table1
Table2
Table3

It is very important to have unique table names, otherwise you will get an exception.

Registering a dataset

In order to perform SQL operations on a dataset, we need to register it as a table in `BatchTableEnvironment`. We need to define a Java POJO class while registering the table.

For instance, let's say we need to register a dataset called Word Count. Each record in this table will have word and frequency attributes. The Java POJO for the same would look like the following:

```
public static class WC {
    public String word;
    public long frequency;
    public WC(){
    }

    public WC(String word, long frequency) {
      this.word = word;
      this.frequency = frequency;
    }

    @Override
    public String toString() {
      return "WC " + word + " " + frequency;
    }
  }
```

The same class in Scala can be defined as follows:

```
case class WordCount(word: String, frequency: Long)
```

Now we can register this table.

In Java:

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

BatchTableEnvironment tEnv = TableEnvironment.getTableEnvironment(env);

DataSet<WC> input = env.fromElements(new WC("Hello", 1), new WC("World",
1), new WC("Hello", 1));

// register the DataSet as table "WordCount"
tEnv.registerDataSet("WordCount", input, "word, frequency");
```

In Scala:

```
val env = ExecutionEnvironment.getExecutionEnvironment

val tEnv = TableEnvironment.getTableEnvironment(env)

val input = env.fromElements(WordCount("hello", 1), WordCount("hello", 1),
WordCount("world", 1), WordCount("hello", 1))

//register the dataset
tEnv.registerDataSet("WordCount", input, 'word, 'frequency)
```

Please make a note that the name of the dataset table must not match the `^_DataSetTable_[0-9]+` pattern as it is reserved for internal memory use.

Registering a datastream

Similar to a dataset, we can also register a datastream in `StreamTableEnvironment`. We need to define a Java POJO class while registering the table.

For instance, let's say we need to register a datastream called Word Count. Each record in this table will have a word and frequency attributes. The Java POJO for the same would look as follows:

```
public static class WC {
    public String word;
    public long frequency;

    public WC() {
    }s
    public WC(String word, long frequency) {
      this.word = word;
      this.frequency = frequency;
    }

    @Override
    public String toString() {
      return "WC " + word + " " + frequency;
    }
  }
```

The same class in Scala can be defined as shown here:

```
case class WordCount(word: String, frequency: Long)
```

Now we can register this table.

In Java:

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
    StreamTableEnvironment tEnv =
TableEnvironment.getTableEnvironment(env);

    DataStream<WC> input = env.fromElements(new WC("Hello", 1), new
WC("World", 1), new WC("Hello", 1));

    // register the DataStream as table "WordCount"
    tEnv.registerDataStream("WordCount", input, "word, frequency");
```

In Scala:

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

val tEnv = TableEnvironment.getTableEnvironment(env)


val input = env.fromElements(WordCount("hello", 1), WordCount("hello", 1),
WordCount("world", 1), WordCount("hello", 1))

//register the dataset
tEnv.registerDataStream("WordCount", input, 'word, 'frequency)
```

Please make a note that the name of the datastream table must not match the `^_DataStreamTable_[0-9]+` pattern as it is reserved for internal memory use.

Registering a table

Similar to a dataset and a datastream, we can also register a table originating from Table API.

In Java:

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
BatchTableEnvironment tEnv = TableEnvironment.getTableEnvironment(env);
```

```
DataSet<WC> input = env.fromElements(new WC("Hello", 1), new WC("World",
1), new WC("Hello", 1));

tEnv.registerDataSet("WordCount", input, "word, frequency");

Table selectedTable = tEnv
        .sql("SELECT word, SUM(frequency) as frequency FROM WordCount GROUP
BY word having word = 'Hello'");

tEnv.registerTable("selected", selectedTable);
```

In Scala:

```
val env = ExecutionEnvironment.getExecutionEnvironment

val tEnv = TableEnvironment.getTableEnvironment(env)

val input = env.fromElements(WordCount("hello", 1), WordCount("hello", 1),
WordCount("world", 1), WordCount("hello", 1))

tEnv.registerDataSet("WordCount", input, 'word, 'frequency)

val table = tEnv.sql("SELECT word, SUM(frequency) FROM WordCount GROUP BY
word")

val selected = tEnv.sql("SELECT word, SUM(frequency) FROM WordCount GROUP
BY word where word = 'hello'")
    tEnv.registerTable("selected", selected)
```

Registering external table sources

Flink allows us to register an external table from sources using a `TableSource`. A table source can allow us to access data stored in databases such as MySQL and Hbase, in. filesystems such as CSVs, Parquet, and ORC, or you can also read messaging systems such as RabbitMQ and Kafka.

Currently, Flink allows reading data from CSV files using CSV sources and JSON data from Kafka topics using Kafka sources.

CSV table source

Now let's look at how to directly read data using a CSV source and then register the source in a table environment.

A CSV source is by default available in the `flink-table` API JAR so there is no need to add any other extra Maven dependency. The following dependency is good enough:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table_2.11</artifactId>
  <version>1.1.4</version>
</dependency>
```

The following code snippet shows how to read CSV files and register the table source.

In Java:

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
BatchTableEnvironment tableEnv = TableEnvironment.getTableEnvironment(env);

TableSource orders = new CsvTableSource("/path/to/file", ...)

// register a TableSource as external table "orders"
tableEnv.registerTableSource("orders", orders)
```

In Scala:

```
val env = ExecutionEnvironment.getExecutionEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

val orders: TableSource = new CsvTableSource("/path/to/file", ...)

// register a TableSource as external table "orders"
tableEnv.registerTableSource("orders", orders)
```

Kafka JSON table source

We can also register the Kafka JSON table source in the table environment. In order to use this API we need to add the following two dependencies:

The first one is for Table API:

```
<dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-table_2.11</artifactId>
      <version>1.1.4</version>
    </dependency>
```

The second dependency would be for the Kafka Flink connector:

- If you are using Kafka 0.8, apply:

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka-0.8_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

- If you are using Kafka 0.9, apply:

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka-0.9_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

Now we need to write the code as shown in the following code snippet:

```
String[] fields =  new String[] { "id", "name", "price"};
Class<?>[] types = new Class<?>[] { Integer.class, String.class,
Double.class };

KafkaJsonTableSource kafkaTableSource = new Kafka08JsonTableSource(
    kafkaTopic,
    kafkaProperties,
    fields,
    types);

tableEnvironment.registerTableSource("kafka-source", kafkaTableSource);
Table result = tableEnvironment.ingest("kafka-source");
```

In the preceding code, we define the Kafka source for Kafka 0.8 and then register the source in the table environment.

Accessing the registered table

Once the table is registered, we can access it very easily from `TableEnvironment` as shown here:

```
tableEnvironment.scan("tableName")
```

The preceding statement scans the table registered with the name "tableName" in BatchTableEnvironment:

```
tableEnvironment.ingest("tableName")
```

The preceding statement ingests the table registered with the name "tableName" in StreamTableEnvironment:

Operators

Flink's Table API provides various operators as part of its domain-specific language. Most of the operators are available in Java and Scala APIs. Let's look at those operators one by one.

The select operator

The select operator is like a SQL select operator which allows you to select various attributes/columns in a table.

In Java:

```
Table result = in.select("id, name");
Table result = in.select("*");
```

In Scala:

```
val result = in.select('id, 'name);
val result = in.select('*);
```

The where operator

The where operator is used for filtering out results.

In Java:

```
Table result = in.where("id = '101'");
```

In Scala:

```
val result = in.where('id == "101");
```

The filter operator

The `filter` operator can be used as a replacement for the `where` operator.

In Java:

```
Table result = in.filter("id = '101'");
```

In Scala:

```
val result = in.filter('id == "101");
```

The as operator

The `as` operator is used for renaming fields:

In Java:

```
Table in = tableEnv.fromDataSet(ds, "id, name");
Table result = in.as("order_id, order_name");
```

In Scala:

```
val in = ds.toTable(tableEnv).as('order_id, 'order_name )
```

The groupBy operator

This is similar to SQL `groupBy` operations which aggregate the results according to a given attribute.

In Java:

```
Table result = in.groupBy("company");
```

In Scala:

```
val in = in.groupBy('company)
```

The join operator

The `join` operator is used to join tables. It is compulsory that we specify at least one equality joining condition.

In Java:

```
Table employee = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table dept = tableEnv.fromDataSet(dept, "d_id, d_name");

Table result = employee.join(dept).where("deptId = d_id").select("e_id,
e_name, d_name");
```

In Scala:

```
val employee = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId);

val dept = deptDS.toTable(tableEnv, 'd_id, 'd_name);

Table result = employee.join(dept).where('deptId == 'd_id).select('e_id,
'e_name, 'd_name);
```

The leftOuterJoin operator

The `leftOuterJoin` operator joins two tables by getting all the values from the table specified on the left side and selects only the matching values from the right side table. It is compulsory that we specify at least one equality joining condition.

In Java:

```
Table employee = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table dept = tableEnv.fromDataSet(dept, "d_id, d_name");

Table result = employee.leftOuterJoin(dept).where("deptId =
d_id").select("e_id, e_name, d_name");
```

In Scala:

```
val employee = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId);

val dept = deptDS.toTable(tableEnv, 'd_id, 'd_name);

Table result = employee.leftOuterJoin(dept).where('deptId ==
'd_id).select('e_id, 'e_name, 'd_name);
```

The rightOuterJoin operator

The `rightOuterJoin` operator joins two tables by getting all values from the table specified on the right side and selects only matching values from the left side table. It is compulsory that we specify at least one equality joining condition.

In Java:

```
Table employee = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table dept = tableEnv.fromDataSet(dept, "d_id, d_name");

Table result = employee.rightOuterJoin(dept).where("deptId =
d_id").select("e_id, e_name, d_name");
```

In Scala:

```
val employee = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId);

val dept = deptDS.toTable(tableEnv, 'd_id, 'd_name);

Table result = employee.rightOuterJoin(dept).where('deptId ==
'd_id).select('e_id, 'e_name, 'd_name);
```

The fullOuterJoin operator

The `fullOuterJoin` operator joins two tables by getting all the values from both tables. It is compulsory that we specify at least one equality joining condition.

In Java:

```
Table employee = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table dept = tableEnv.fromDataSet(dept, "d_id, d_name");

Table result = employee.fullOuterJoin(dept).where("deptId =
d_id").select("e_id, e_name, d_name");
```

In Scala:

```
val employee = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId);

val dept = deptDS.toTable(tableEnv, 'd_id, 'd_name);

Table result = employee.fullOuterJoin(dept).where('deptId ==
'd_id).select('e_id, 'e_name, 'd_name);
```

The union operator

The `union` operator merges two similar tables. It removes duplicate values in the resulting table.

In Java:

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table employee2 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table result = employee1.union(employee2);
```

In Scala:

```
val employee1 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)

val employee2 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)

Table result = employee1.union(employee2)
```

The unionAll operator

The `unionAll` operator merges two similar tables.

In Java:

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table employee2 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table result = employee1.unionAll(employee2);
```

In Scala:

```
val employee1 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)

val employee2 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)

Table result = employee1.unionAll(employee2)
```

The intersect operator

The `intersect` operator returns matching values from both tables. It makes sure that the resultant table does have any duplicates.

In Java:

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table employee2 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table result = employee1.intersect(employee2);
```

In Scala:

```
val employee1 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)

val employee2 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)

Table result = employee1.intersect(employee2)
```

The intersectAll operator

The `intersectAll` operator returns matching values from both tables. The resultant table might have duplicate records.

In Java:

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table employee2 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");


Table result = employee1.intersectAll(employee2);
```

In Scala:

```
val employee1 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)

val employee2 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)

Table result = employee1.intersectAll(employee2)
```

The minus operator

The `minus` operator returns records from the left table which do not exist in the right table. It makes sure that the resultant table does not have any duplicates.

In Java:

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table employee2 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table result = employee1.minus(employee2);
```

In Scala:

```
val employee1 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)

val employee2 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)

Table result = employee1.minus(employee2)
```

The minusAll operator

The `minusAll` operator returns records from the left table which do not exist in the right table. The resultant table might have duplicate records.

In Java:

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table employee2 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table result = employee1.minusAll(employee2);
```

In Scala:

```
val employee1 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)

val employee2 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)

Table result = employee1.minusAll(employee2)
```

The distinct operator

The `distinct` operator returns only unique value records from the table.

In Java:

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table result = employee1.distinct();
```

In Scala:

```
val employee1 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)

Table result = employee1.distinct()
```

The orderBy operator

The `orderBy` operator returns records sorted across globally parallel partitions. You can choose the order as ascending or descending.

In Java:

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

Table result = employee1.orderBy("e_id.asc");
```

In Scala:

```
val employee1 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)

Table result = employee1.orderBy('e_id.asc)
```

The limit operator

The `limit` operator limits records sorted across globally parallel partitions from a given offset.

In Java:

```
Table employee1 = tableEnv.fromDataSet(emp, "e_id, e_name, deptId");

//returns records from 6th record
Table result = employee1.orderBy("e_id.asc").limit(5);
```

```
//returns 5 records from 4th record
Table result1 = employee1.orderBy("e_id.asc").limit(3,5);
```

In Scala:

```
val employee1 = empDS.toTable(tableEnv, 'e_id, 'e_name, 'deptId)
//returns records from 6th record
Table result = employee1.orderBy('e_id.asc).limit(5)
//returns 5 records from 4th record
Table result = employee1.orderBy('e_id.asc).limit(3,5)
```

Data types

Table API supports common SQL data types which can be used easily. Internally, it uses `TypeInformation` to identify various data types. It currently does not support all Flink data types:

Table API	SQL	Java type
`Types.STRING`	`VARCHAR`	`java.lang.String`
`Types.BOOLEAN`	`BOOLEAN`	`java.lang.Boolean`
`Types.BYTE`	`TINYINT`	`java.lang.Byte`
`Types.SHORT`	`SMALLINT`	`java.lang.Short`
`Types.INT`	`INTEGER`, `INT`	`java.lang.Integer`
`Types.LONG`	`BIGINT`	`java.lang.Long`
`Types.FLOAT`	`REAL`, `FLOAT`	`java.lang.Float`
`Types.DOUBLE`	`DOUBLE`	`java.lang.Double`
`Types.DECIMAL`	`DECIMAL`	`java.math.BigDecimal`
`Types.DATE`	`DATE`	`java.sql.Date`
`Types.TIME`	`TIME`	`java.sql.Time`
`Types.TIMESTAMP`	`TIMESTAMP(3)`	`java.sql.Timestamp`
`Types.INTERVAL_MONTHS`	INTERVAL YEAR TO MONTH	`java.lang.Integer`
`Types.INTERVAL_MILLIS`	INTERVAL DAY TO SECOND(3)	`java.lang.Long`

With continuous development and support from the community, more data types will be supported soon.

SQL

Table API also allows us to write free form SQL queries using the `sql()` method. The method internally also uses Apache Calcite for SQL syntax verification and optimization. It executes the query and returns results in the table format. Later the table can be again transformed into either a dataset or datastream or `TableSink` for further processing.

One thing to note here is that, for the SQL method to access the tables, they must be registered with `TableEnvironment`.

More support is being added to the SQL method continuously so if any syntax is not supported, it will error out with `TableException`.

Now let's look at how to use the SQL method on a dataset and datastream.

SQL on datastream

SQL queries can be executed on datastreams registered with `TableEnvironment` using the `SELECT STREAM` keyword. Most of the SQL syntax is common between datasets and datastreams. To know more about stream syntax, the Apache Calcite's Streams documentation would be helpful. It can be found at: `https://calcite.apache.org/docs/stream.html`.

Let's say we want to analyze the product schema defined as (`id`, `name`, `stock`). The following code needs to be written using the `sql()` method.

In Java:

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
StreamTableEnvironment tableEnv =
TableEnvironment.getTableEnvironment(env);

DataStream<Tuple3<Long, String, Integer>> ds = env.addSource(...);
// register the DataStream as table "Products"
tableEnv.registerDataStream("Products", ds, "id, name, stock");
// run a SQL query on the Table and retrieve the result as a new Table
Table result = tableEnv.sql(
  "SELECT STREAM * FROM Products WHERE name LIKE '%Apple%'");
```

In Scala:

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tEnv = TableEnvironment.getTableEnvironment(env)

val ds: DataStream[(Long, String, Integer)] = env.addSource(...)
// register the DataStream under the name "Products"
tableEnv.registerDataStream("Products", ds, 'id, 'name, 'stock)
// run a SQL query on the Table and retrieve the result as a new Table
val result = tableEnv.sql(
  "SELECT STREAM * FROM Products WHERE name LIKE '%Apple%'")
```

Table API uses a lexical policy similar to Java in order to define queries properly. This means the case of the identifiers is preserved and they are matched case sensitively. If any of your identifiers contain non-alpha numeric characters then you can quote those using back ticks.

For instance, if you want to define a column with the name `'my col'` then you need to use back ticks as shown here:

```
"SELECT col as `my col` FROM table "
```

Supported SQL syntax

As stated earlier, Flink uses Apache Calcite for validating and optimizing SQL queries. With the current version, the following **Backus Naur Form** (**BNF**) is supported:

```
query:
  values
  | {
      select
      | selectWithoutFrom
      | query UNION [ ALL ] query
      | query EXCEPT query
      | query INTERSECT query
    }
    [ ORDER BY orderItem [, orderItem ]* ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start { ROW | ROWS } ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY]

orderItem:
  expression [ ASC | DESC ]

select:
  SELECT [ STREAM ] [ ALL | DISTINCT ]
  { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
  [ GROUP BY { groupItem [, groupItem ]* } ]
  [ HAVING booleanExpression ]

selectWithoutFrom:
  SELECT [ ALL | DISTINCT ]
  { * | projectItem [, projectItem ]* }

projectItem:
  expression [ [ AS ] columnAlias ]
  | tableAlias . *

tableExpression:
  tableReference [, tableReference ]*
  | tableExpression [ NATURAL ] [ LEFT | RIGHT | FULL ] JOIN
tableExpression [ joinCondition ]

joinCondition:
  ON booleanExpression
  | USING '(' column [, column ]* ')'

tableReference:
  tablePrimary
  [ [ AS ] alias [ '(' columnAlias [, columnAlias ]* ')' ] ]

tablePrimary:
  [ TABLE ] [ [ catalogName . ] schemaName . ] tableName

values:
  VALUES expression [, expression ]*

groupItem:
  expression
  | '(' ')'
  | '(' expression [, expression ]* ')'
```

Scalar functions

Table API and SQL support various built-in scalar functions. Let's try to understand those one by one.

Scalar functions in the table API

The following is the list of supported scalar functions in the table API:

Java function	Scala function	Description
`ANY.isNull`	`ANY.isNull`	Returns `true` if the given expression is null.
`ANY.isNotNull`	`ANY.isNotNull`	Returns `true` if the given expression is not null.
`BOOLEAN.isTrue`	`BOOLEAN.isTrue`	Returns `true` if the given Boolean expression is `true`. `False` otherwise.
`BOOLEAN.isFalse`	`BOOLEAN.isFalse`	Returns `true` if given Boolean expression is false. `False` otherwise.
`NUMERIC.log10()`	`NUMERIC.log10()`	Calculates the base 10 logarithm of given value.
`NUMERIC.ln()`	`NUMERIC.ln()`	Calculates the natural logarithm of given value.
`NUMERIC.power(NUMERIC)`	`NUMERIC.power(NUMERIC)`	Calculates the given number raised to the power of the other value.
`NUMERIC.abs()`	`NUMERIC.abs()`	Calculates the absolute value of given value.
`NUMERIC.floor()`	`NUMERIC.floor()`	Calculates the largest integer less than or equal to a given number.
`NUMERIC.ceil()`	`NUMERIC.ceil()`	Calculates the smallest integer greater than or equal to a given number.
`STRING.substring(INT, INT)`	`STRING.substring(INT, INT)`	Creates a substring of the given string at the given index for the given length
`STRING.substring(INT)`	`STRING.substring(INT)`	Creates a substring of the given string beginning at the given index to the end. The start index starts at 1 and is inclusive.
`STRING.trim(LEADING, STRING)STRING.trim(TRAILING, STRING)STRING.trim(BOTH, STRING)STRING.trim(BOTH)STRING.trim()`	`STRING.trim(leading = true, trailing = true, character = " ")`	Removes leading and/or trailing characters from the given string. By default, whitespaces at both sides are removed.
`STRING.charLength()`	`STRING.charLength()`	Returns the length of a string.
`STRING.upperCase()`	`STRING.upperCase()`	Returns all of the characters in a string in upper case using the rules of the default locale.
`STRING.lowerCase()`	`STRING.lowerCase()`	Returns all of the characters in a string in lower case using the rules of the default locale.
`STRING.initCap()`	`STRING.initCap()`	Converts the initial letter of each word in a string to uppercase. Assumes a string containing only `[A-Za-z0-9]`, everything else is treated as whitespace.
`STRING.like(STRING)`	`STRING.like(STRING)`	Returns true, if a string matches the specified LIKE pattern. For example, `"Jo_n%"` matches all strings that start with `"Jo(arbitrary letter)n"`.

`STRING.similar(STRING)`	`STRING.similar(STRING)`	Returns `true`, if a string matches the specified SQL regex pattern. For example, `"A+"` matches all strings that consist of at least one `"A"`.
`STRING.toDate()`	`STRING.toDate`	Parses a date string in the form `"yy-mm-dd"` to a SQL date.
`STRING.toTime()`	`STRING.toTime`	Parses a time string in the form `"hh:mm:ss"` to a SQL time.
`STRING.toTimestamp()`	`STRING.toTimestamp`	Parses a timestamp string in the form `"yy-mm-dd hh:mm:ss.fff"` to a SQL timestamp.
`TEMPORAL.extract(TIMEINTERVALUNIT)`	NA	Extracts parts of a time point or time interval. Returns the part as a long value. For example, `2006-06-05 .toDate.extract(DAY)` leads to `5`.
`TIMEPOINT.floor(TIMEINTERVALUNIT)`	`TIMEPOINT.floor(TimeIntervalUnit)`	Rounds a time point down to the given unit. For example, `"12:44:31".toDate.floor(MINUTE)` leads to `12:44:00`.
`TIMEPOINT.ceil(TIMEINTERVALUNIT)`	`TIMEPOINT.ceil(TimeIntervalUnit)`	Rounds a time point up to the given unit. For example, `"12:44:31".toTime.floor(MINUTE)` leads to `12:45:00`.
`currentDate()`	`currentDate()`	Returns the current SQL date in UTC time zone.
`currentTime()`	`currentTime()`	Returns the current SQL time in UTC time zone.
`currentTimestamp()`	`currentTimestamp()`	Returns the current SQL timestamp in UTC time zone.
`localTime()`	`localTime()`	Returns the current SQL time in local time zone.
`localTimestamp()`	`localTimestamp()`	Returns the current SQL timestamp in local time zone.

Scala functions in SQL

The following is the list of supported scalar functions in the `sql()` method:

Function	**Description**
`EXP(NUMERIC)`	Calculates the Euler's number raised to the given power.
`LOG10(NUMERIC)`	Calculates the base 10 logarithm of the given value.
`LN(NUMERIC)`	Calculates the natural logarithm of the given value.

`POWER(NUMERIC, NUMERIC)`	Calculates the given number raised to the power of the other value.
`ABS(NUMERIC)`	Calculates the absolute value of the given value.
`FLOOR(NUMERIC)`	Calculates the largest integer less than or equal to a given number.
`CEIL(NUMERIC)`	Calculates the smallest integer greater than or equal to a given number.
`SUBSTRING(VARCHAR, INT, INT) SUBSTRING(VARCHAR FROM INT FOR INT)`	Creates a substring of the given string at the given index for the given length. The index starts at 1 and is inclusive, that is, the character at the index is included in the substring. The substring has the specified length or less.
`SUBSTRING(VARCHAR, INT) SUBSTRING(VARCHAR FROM INT)`	Creates a substring of the given string beginning at the given index to the end. The start index starts at 1 and is inclusive.
`TRIM(LEADING VARCHAR FROM VARCHAR) TRIM(TRAILING VARCHAR FROM VARCHAR) TRIM(BOTH VARCHAR FROM VARCHAR) TRIM(VARCHAR)`	Removes leading and/or trailing characters from the given string. By default, whitespaces at both sides are removed.
`CHAR_LENGTH(VARCHAR)`	Returns the length of a string.
`UPPER(VARCHAR)`	Returns all of the characters in a string in upper case using the rules of the default locale.
`LOWER(VARCHAR)`	Returns all of the characters in a string in lower case using the rules of the default locale.
`INITCAP(VARCHAR)`	Converts the initial letter of each word in a string to uppercase. Assumes a string containing only `[A-Za-z0-9]`, everything else is treated as whitespace.
`VARCHAR LIKE VARCHAR`	Returns true if a string matches the specified LIKE pattern. For example, "`Jo_n%`" matches all strings that start with "`Jo(arbitrary letter)n`".
`VARCHAR SIMILAR TO VARCHAR`	Returns true if a string matches the specified SQL regex pattern. For example, "`A+`" matches all strings that consist of at least one "`A`".

`DATE VARCHAR`	Parses a date string in the form `"yy-mm-dd"` to a SQL date.
`TIME VARCHAR`	Parses a time string in the form `"hh:mm:ss"` to a SQL time.
`TIMESTAMP VARCHAR`	Parses a timestamp string in the form `"yy-mm-dd hh:mm:ss.fff"` to a SQL timestamp.
`EXTRACT(TIMEINTERVALUNIT FROM TEMPORAL)`	Extracts parts of a time point or time interval. Returns the part as a long value. For example, `EXTRACT(DAY FROM DATE '2006-06-05')` leads to `5`.
`FLOOR(TIMEPOINT TO TIMEINTERVALUNIT)`	Rounds a time point down to the given unit. For example, `FLOOR(TIME '12:44:31' TO MINUTE)` leads to `12:44:00`.
`CEIL(TIMEPOINT TO TIMEINTERVALUNIT)`	Rounds a time point up to the given unit. For example, `CEIL(TIME '12:44:31' TO MINUTE)` leads to `12:45:00`.
`CURRENT_DATE`	Returns the current SQL date in UTC timezone.
`CURRENT_TIME`	Returns the current SQL time in UTC timezone.
`CURRENT_TIMESTAMP`	Returns the current SQL timestamp in UTC timezone.
`LOCALTIME`	Returns the current SQL time in local timezone.
`LOCALTIMESTAMP`	Returns the current SQL timestamp in local timezone.

Use case – Athletes data insights using Flink Table API

Now that we have learnt details of Table API, let's try to apply this knowledge to a real life use case. Consider we have a dataset with us, which has information about the Olympic athletes and their performance in various games.

The sample data looks like that shown in the following table:

Player	Country	Year	Game	Gold	Silver	Bronze	Total
Yang Yilin	China	2008	Gymnastics	1	0	2	3
Leisel Jones	Australia	2000	Swimming	0	2	0	2
Go Gi-Hyeon	South Korea	2002	Short-Track Speed Skating	1	1	0	2
Chen Ruolin	China	2008	Diving	2	0	0	2
Katie Ledecky	United States	2012	Swimming	1	0	0	1
Ruta Meilutyte	Lithuania	2012	Swimming	1	0	0	1
DÃ¡niel Gyurta	Hungary	2004	Swimming	0	1	0	1
Arianna Fontana	Italy	2006	Short-Track Speed Skating	0	0	1	1
Olga Glatskikh	Russia	2004	Rhythmic Gymnastics	1	0	0	1
Kharikleia Pantazi	Greece	2000	Rhythmic Gymnastics	0	0	1	1
Kim Martin	Sweden	2002	Ice Hockey	0	0	1	1
Kyla Ross	United States	2012	Gymnastics	1	0	0	1
Gabriela Dragoi	Romania	2008	Gymnastics	0	0	1	1
Tasha Schwikert-Warren	United States	2000	Gymnastics	0	0	1	1

Now we want to get answers to the questions like, how many medals were won by country or by game. As the data we have in structured data, we can use Table API to query data in a SQL way. So let's get started.

The data available is in the CSV format. So we will be using a CSV reader provided by Flink API as shown in the following code snippet:

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
BatchTableEnvironment tableEnv = TableEnvironment.getTableEnvironment(env);
DataSet<Record> csvInput = env
          .readCsvFile("olympic-athletes.csv")
          .pojoType(Record.class, "playerName", "country", "year",
                        "game", "gold", "silver", "bronze", "total");
```

Next we need to create a Table with this dataset and register it in Table Environment for further processing:

```
Table atheltes = tableEnv.fromDataSet(csvInput);
tableEnv.registerTable("athletes", atheltes);
```

Next we can write a regular SQL query to get more insights from the data. Or else we can use Table API operators to manipulate the data, as shown in the following code snippet:

```
Table groupedByCountry = tableEnv.sql("SELECT country, SUM(total) as
frequency FROM athletes group by country");
DataSet<Result> result = tableEnv.toDataSet(groupedByCountry,
Result.class);
result.print();
Table groupedByGame = atheltes.groupBy("game").select("game, total.sum as
frequency");
DataSet<GameResult> gameResult = tableEnv.toDataSet(groupedByGame,
GameResult.class);
gameResult.print();
```

This way we can analyse such data in a much more simpler way using Table API. The complete code for this use case is available on GitHub at `https://github.com/deshpandetanmay/mastering-flink/tree/master/chapter04/flink-table`.

Summary

In this chapter, we learned about a SQL-based API supported by Flink called Table API. We also learned how to transform a dataset/stream into a table, registering a table, datasets, and datastreams with `TableEnvironment` and then using the registered tables to perform various operations. For people coming from a SQL databases background, this API is bliss.

In the next chapter, we are going to talk about a very interesting library called **Complex Event Processing** and how to use it for solving various business use cases.

5

Complex Event Processing

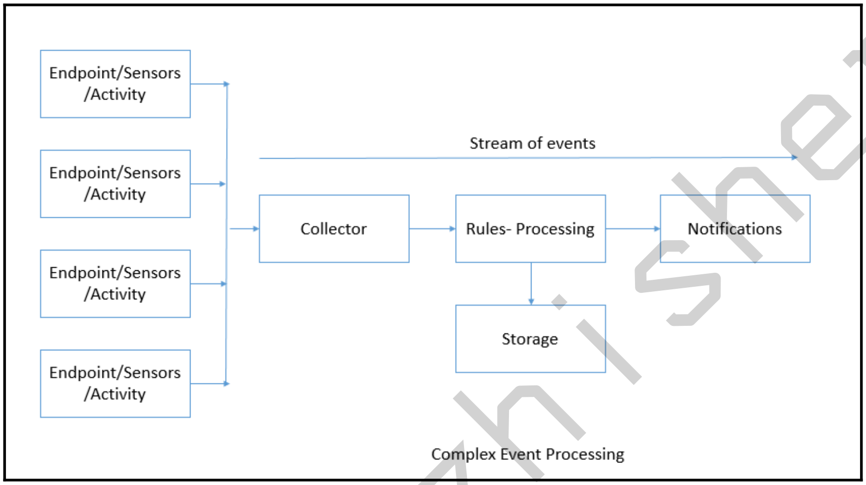
In the previous chapter, we talked about the Table API provided by Apache Flink and how we can use it to process relational data structures. This chapter onwards, we will start learning more about the libraries provided by Apache Flink and how we can use them for specific use cases. To start with, let's try to understand a library called **Complex Event Processing** (**CEP**). CEP is a very interesting but complex topic that has its value in various industries. Wherever there is a stream of events expected, naturally people want to perform complex event processing in all such use cases. Let's try to understand what CEP is all about.

What is complex event processing?

CEP analyzes streams of disparate events occurring with high frequency and low latency. These days, streaming events can be found in various industries, for example:

- In the oil and gas domain, sensor data comes from various drilling tools or from upstream oil pipeline equipment
- In the security domain, activity data, malware information, and usage pattern data come from various end points
- In the wearable domain, data comes from various wrist bands with information about your heart beat rate, your activity, and so on
- In the banking domain, data comes from credit card usage, banking activities, and so on

It is very important to analyze variation patterns to get notified in real time about any change in the regular assembly. CEP can understand patterns across the streams of events, sub-events, and their sequences. CEP helps to identify meaningful patterns and complex relationships among unrelated events, and sends notifications in real and near real time to prevent damage:


Endpoint/Sensors /Activity
Endpoint/Sensors /Activity
Endpoint/Sensors /Activity
Endpoint/Sensors /Activity
Stream of events
Collector
Rules- Processing
Notifications
Storage
Complex Event Processing

The preceding diagram shows how the CEP flow works. Even though the flow looks simple, CEP has various abilities such as:

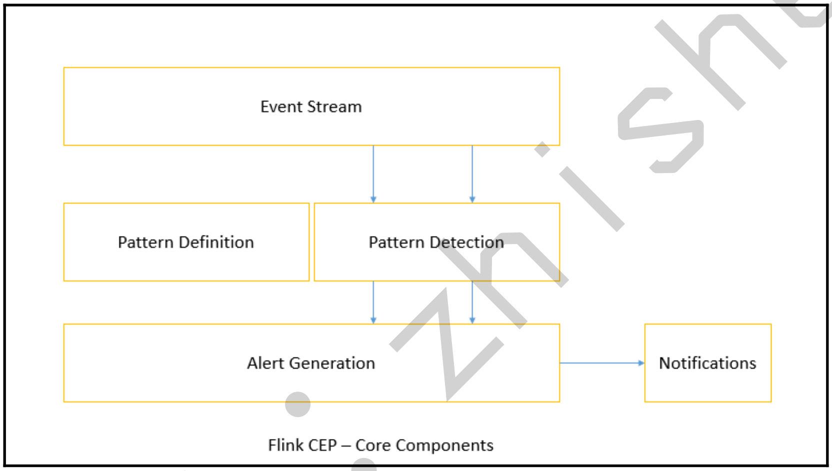
- The ability to produce results as soon as the input event stream is available
- The ability to provide computations such as aggregation over time and timeout between two events of interest
- The ability to provide real-time/near real-time alerts and notifications on detection of complex event patterns
- The ability to connect and correlate heterogeneous sources and analyze patterns in them
- The ability to achieve high-throughput, low-latency processing

There are various solutions available on the market. With big data technology advancements, we have multiple options like Apache Spark, Apache Samza, Apache Beam, among others, but none of them have a dedicated library to fit all solutions. Now let us try to understand what we can achieve with Flink's CEP library.

Flink CEP

Apache Flink provides the Flink CEP library, which provides APIs to perform complex event processing. The library consists of the following core components:

- Event stream
- Pattern definition
- Pattern detection
- Alert generation



Flink CEP works on Flink's streaming API called DataStream. A programmer needs to define the pattern to be detected from the stream of events and then Flink's CEP engine detects the pattern and takes the appropriate action, such as generating alerts.

In order to get started, we need to add the following Maven dependency:

```
<!--
https://mvnrepository.com/artifact/org.apache.flink/flink-cep-scala_2.10 -
->
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-cep-scala_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

Event streams

A very important component of CEP is its input event stream. In earlier chapters, we have seen details of the DataStream API. Now let's use that knowledge to implement CEP. The very first thing we need to do is define a Java POJO for the event. Let's assume we need to monitor a temperature sensor event stream.

First we define an abstract class and then extend this class.

While defining the event POJOs we need to make sure that we implement the `hashCode()` and `equals()` methods, because while comparing the events, compile will make use of them.

The following code snippets demonstrate this.

First, we write an abstract class as shown here:

```
package com.demo.chapter05;

public abstract class MonitoringEvent {

  private String machineName;

  public String getMachineName() {
    return machineName;
  }

  public void setMachineName(String machineName) {
    this.machineName = machineName;
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((machineName == null) ? 0 :
machineName.hashCode());
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    MonitoringEvent other = (MonitoringEvent) obj;
    if (machineName == null) {
      if (other.machineName != null)
        return false;
    } else if (!machineName.equals(other.machineName))
      return false;
    return true;
  }

  public MonitoringEvent(String machineName) {
    super();
    this.machineName = machineName;
  }

}
```

Then we create a POJO for the actual temperature event:

```
  package com.demo.chapter05;

  public class TemperatureEvent extends MonitoringEvent {

    public TemperatureEvent(String machineName) {
      super(machineName);
    }

    private double temperature;

    public double getTemperature() {
      return temperature;
    }

    public void setTemperature(double temperature) {
      this.temperature = temperature;
    }

    @Override
    public int hashCode() {
      final int prime = 31;
      int result = super.hashCode();
      long temp;
      temp = Double.doubleToLongBits(temperature);
      result = prime * result + (int) (temp ^ (temp >>> 32));
      return result;
    }
    @Override
    public boolean equals(Object obj) {
      if (this == obj)
        return true;
      if (!super.equals(obj))
        return false;
      if (getClass() != obj.getClass())
        return false;
      TemperatureEvent other = (TemperatureEvent) obj;
      if (Double.doubleToLongBits(temperature) !=
Double.doubleToLongBits(other.temperature))
        return false;
      return true;
    }

    public TemperatureEvent(String machineName, double temperature) {
      super(machineName);
      this.temperature = temperature;
    }

    @Override
    public String toString() {
      return "TemperatureEvent [getTemperature()=" + getTemperature() + ",
getMachineName()=" + getMachineName()
          + "]";
    }

  }
```

Now we can define the event source as follows:

In Java:

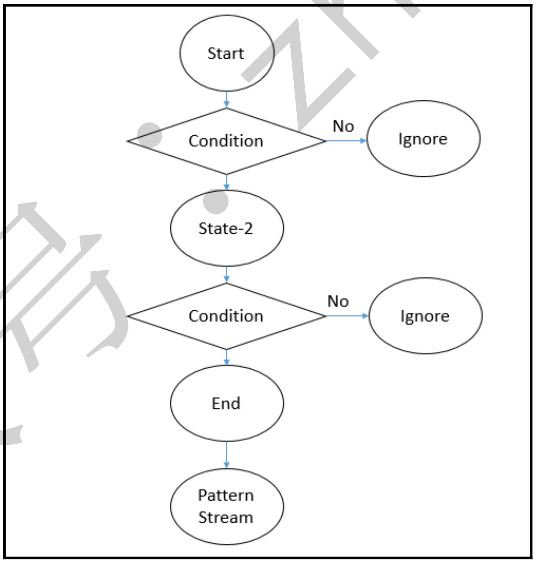
```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
    DataStream<TemperatureEvent> inputEventStream = env.fromElements(new
TemperatureEvent("xyz", 22.0),
        new TemperatureEvent("xyz", 20.1), new TemperatureEvent("xyz",
21.1), new TemperatureEvent("xyz", 22.2),
        new TemperatureEvent("xyz", 22.1), new TemperatureEvent("xyz",
22.3), new TemperatureEvent("xyz", 22.1),
        new TemperatureEvent("xyz", 22.4), new TemperatureEvent("xyz",
22.7),
        new TemperatureEvent("xyz", 27.0));
```

In Scala:

```
val env: StreamExecutionEnvironment =
StreamExecutionEnvironment.getExecutionEnvironment
    val input: DataStream[TemperatureEvent] = env.fromElements(new
TemperatureEvent("xyz", 22.0),
      new TemperatureEvent("xyz", 20.1), new TemperatureEvent("xyz", 21.1),
new TemperatureEvent("xyz", 22.2),
      new TemperatureEvent("xyz", 22.1), new TemperatureEvent("xyz", 22.3),
new TemperatureEvent("xyz", 22.1),
      new TemperatureEvent("xyz", 22.4), new TemperatureEvent("xyz", 22.7),
      new TemperatureEvent("xyz", 27.0))
```

Pattern API

The Pattern API allows you to define complex event patterns very easily. Each pattern consists of multiple states. To go from one state to another state, generally we need to define the conditions. The conditions could be continuity or filtered out events.



Let's try to understand each pattern operation in detail.

Begin

The initial state can be defined as follows:

In Java:

```
Pattern<Event, ?> start = Pattern.<Event>begin("start");
```

In Scala:

```
val start : Pattern[Event, _] = Pattern.begin("start")
```

Filter

We can also specify the filter condition for the initial state:

In Java:

```
start.where(new FilterFunction<Event>() {
    @Override
    public boolean filter(Event value) {
        return ... // condition
    }
});
```

In Scala:

```
start.where(event => ... /* condition */)
```

Subtype

We can also filter out events based on their sub-types, using the `subtype()` method:

In Java:

```
start.subtype(SubEvent.class).where(new FilterFunction<SubEvent>() {
    @Override
    public boolean filter(SubEvent value) {
        return ... // condition
    }
});
```

In Scala:

```
start.subtype(classOf[SubEvent]).where(subEvent => ... /* condition */)
```

OR

Pattern API also allows us define multiple conditions together. We can use `OR` and `AND` operators.

In Java:

```
pattern.where(new FilterFunction<Event>() {
    @Override
    public boolean filter(Event value) {
        return ... // condition
    }
}).or(new FilterFunction<Event>() {
    @Override
    public boolean filter(Event value) {
        return ... // or condition
    }
});
```

In Scala:

```
pattern.where(event => ... /* condition */).or(event => ... /* or condition
*/)
```

Continuity

As stated earlier, we do not always need to filter out events. There can always be some pattern where we need continuity instead of filters.

Continuity can be of two types – strict continuity and non-strict continuity.

Strict continuity

Strict continuity needs two events to succeed directly which means there should be no other event in between. This pattern can be defined by `next()`.

In Java:

```
Pattern<Event, ?> strictNext = start.next("middle");
```

In Scala:

```
val strictNext: Pattern[Event, _] = start.next("middle")
```

Non-strict continuity

Non-strict continuity can be stated as other events are allowed to be in between the specific two events. This pattern can be defined by `followedBy()`.

In Java:

```
Pattern<Event, ?> nonStrictNext = start.followedBy("middle");
```

In Scala:

```
val nonStrictNext : Pattern[Event, _] = start.followedBy("middle")
```

Within

Pattern API also allows us to do pattern matching based on time intervals. We can define a time-based temporal constraint as follows.

In Java:

```
next.within(Time.seconds(30));
```

In Scala:

```
next.within(Time.seconds(10))
```

Detecting patterns

To detect patterns against a stream of events, we need to run the stream though the pattern. The `CEP.pattern()` returns `PatternStream`.

The following code snippet shows how we can detect a pattern. First the pattern is defined to check if the temperature value is greater than `26.0` degrees in `10` seconds.

In Java:

```
Pattern<TemperatureEvent, ?> warningPattern = Pattern.<TemperatureEvent>
begin("first")
        .subtype(TemperatureEvent.class).where(new
FilterFunction<TemperatureEvent>() {
          public boolean filter(TemperatureEvent value) {
            if (value.getTemperature() >= 26.0) {
              return true;
            }
            return false;
          }
        }).within(Time.seconds(10));

    PatternStream<TemperatureEvent> patternStream =
CEP.pattern(inputEventStream, warningPattern);
```

In Scala:

```
val env: StreamExecutionEnvironment =
StreamExecutionEnvironment.getExecutionEnvironment

val input = // data

val pattern: Pattern[TempEvent, _] = Pattern.begin("start").where(event =>
event.temp >= 26.0)
val patternStream: PatternStream[TempEvent] = CEP.pattern(input, pattern)
```

Selecting from patterns

Once the pattern stream is available, we need to select the pattern from it and then take appropriate actions based on it. We can use the `select` or `flatSelect` method to select data from the pattern.

Select

The select method needs `PatternSelectionFunction` implementation. It has a select method which would be called for each event sequence. The `select` method receives a map of string/event pairs of matched events. The string is defined by the name of the state. The `select` method returns exactly one result.

To collect the results, we need to define the output POJO. In our case, let's say we need to generate alerts as output. Then we need to define POJO as follows:

```
package com.demo.chapter05;

public class Alert {

  private String message;

  public String getMessage() {
    return message;
  }
  public void setMessage(String message) {
    this.message = message;
  }

  public Alert(String message) {
    super();
    this.message = message;
  }

  @Override
  public String toString() {
    return "Alert [message=" + message + "]";
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((message == null) ? 0 :
    message.hashCode());
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Alert other = (Alert) obj;
    if (message == null) {
      if (other.message != null)
        return false;
    } else if (!message.equals(other.message))
      return false;
    return true;
  }

}
```

Next we define the select functions.

In Java:

```
class MyPatternSelectFunction<IN, OUT> implements PatternSelectFunction<IN,
OUT> {
    @Override
    public OUT select(Map<String, IN> pattern) {
        IN startEvent = pattern.get("start");
        IN endEvent = pattern.get("end");
        return new OUT(startEvent, endEvent);
    }
}
```

In Scala:

```
def selectFn(pattern : mutable.Map[String, IN]): OUT = {
    val startEvent = pattern.get("start").get
    val endEvent = pattern.get("end").get
    OUT(startEvent, endEvent)
}
```

flatSelect

The `flatSelect` method is similar to the `select` method. The only difference between the two is that `flatSelect` can return an arbitrary number of results. The `flatSelect` method has an additional `Collector` parameter which is used for output element.

The following example shows how we can use the `flatSelect` method.

In Java:

```
class MyPatternFlatSelectFunction<IN, OUT> implements
PatternFlatSelectFunction<IN, OUT> {
    @Override
    public void select(Map<String, IN> pattern, Collector<OUT> collector) {
        IN startEvent = pattern.get("start");
        IN endEvent = pattern.get("end");

        for (int i = 0; i < startEvent.getValue(); i++ ) {
            collector.collect(new OUT(startEvent, endEvent));
        }
    }
}
```

In Scala:

```
def flatSelectFn(pattern : mutable.Map[String, IN], collector :
Collector[OUT]) = {
    val startEvent = pattern.get("start").get
    val endEvent = pattern.get("end").get
    for (i <- 0 to startEvent.getValue) {
        collector.collect(OUT(startEvent, endEvent))
    }
}
```

Handling timed-out partial patterns

Sometimes we may miss out certain events if we have constrained the patterns with a time boundary. It is possible that events may be discarded because they exceed the length. In order to take actions on the timed out events, the `select` and `flatSelect` methods allow a timeout handler. This handler is called for each timeout event pattern.

In this case, the select method contains two parameters: `PatternSelectFunction` and `PatternTimeoutFunction`. The return type for a timed out function can be different from the select pattern function. The timed out event and select event are wrapped in the class `Either.Right` and `Either.Left`.

The following code snippets shows how we do things in practice.

In Java:

```
PatternStream<Event> patternStream = CEP.pattern(input, pattern);

DataStream<Either<TimeoutEvent, ComplexEvent>> result =
patternStream.select(
    new PatternTimeoutFunction<Event, TimeoutEvent>() {...},
    new PatternSelectFunction<Event, ComplexEvent>() {...}
);

DataStream<Either<TimeoutEvent, ComplexEvent>> flatResult =
patternStream.flatSelect(
    new PatternFlatTimeoutFunction<Event, TimeoutEvent>() {...},
    new PatternFlatSelectFunction<Event, ComplexEvent>() {...}
);
```

In Scala, the select API:

```
val patternStream: PatternStream[Event] = CEP.pattern(input, pattern)

DataStream[Either[TimeoutEvent, ComplexEvent]] result =
patternStream.select{
    (pattern: mutable.Map[String, Event], timestamp: Long) =>
TimeoutEvent()
} {
    pattern: mutable.Map[String, Event] => ComplexEvent()
}
```

The `flatSelect` API is called with the `Collector` as it can emit an arbitrary number of events:

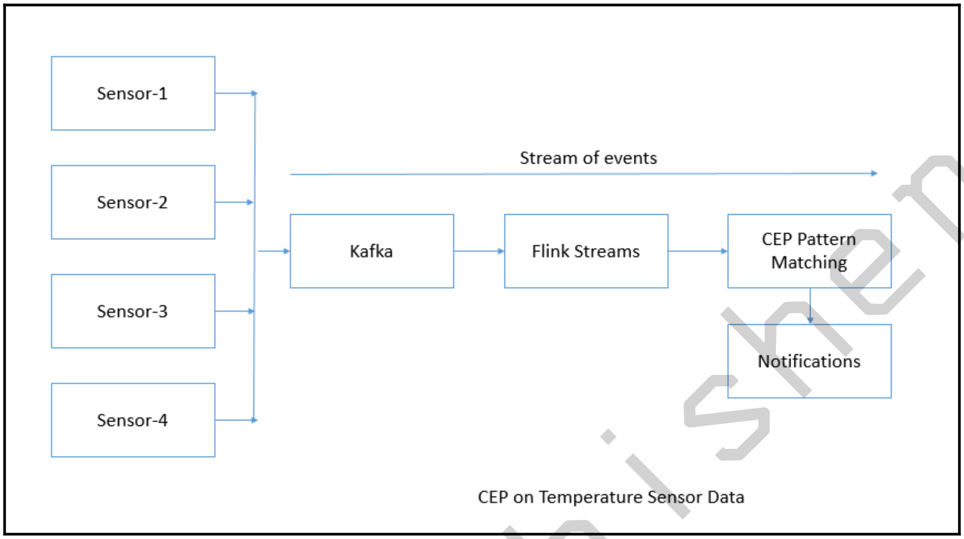
```
val patternStream: PatternStream[Event] = CEP.pattern(input, pattern)

DataStream[Either[TimeoutEvent, ComplexEvent]] result =
patternStream.flatSelect{
    (pattern: mutable.Map[String, Event], timestamp: Long, out:
Collector[TimeoutEvent]) =>
        out.collect(TimeoutEvent())
} {
    (pattern: mutable.Map[String, Event], out: Collector[ComplexEvent]) =>
        out.collect(ComplexEvent())
}
```

Use case – complex event processing on a temperature sensor

In earlier sections, we learnt about various features provided by the Flink CEP engine. Now it's time to understand how we can use it in real-world solutions. For that, let's assume we work for a mechanical company which produces some products. In the product factory, there is a need to constantly monitor certain machines. The factory has already set up the sensors which keep on sending the temperature of the machines at a given time.

Now we will be setting up a system that constantly monitors the temperature value and generates an alert if the temperature exceeds a certain value.

We can use the following architecture:



Here we will be using Kafka to collect events from sensors. In order to write a Java application, we first need to create a Maven project and add the following dependency:

```
  <!--
https://mvnrepository.com/artifact/org.apache.flink/flink-cep-scala_2.11 -
->
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-cep-scala_2.11</artifactId>
      <version>1.1.4</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.apache.flink/flink-
streaming-java_2.11 -->
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-streaming-java_2.11</artifactId>
      <version>1.1.4</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.apache.flink/flink-
streaming-scala_2.11 -->
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-streaming-scala_2.11</artifactId>
      <version>1.1.4</version>
    </dependency>
    <dependency>
```

```
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-connector-kafka-0.9_2.11</artifactId>
      <version>1.1.4</version>
    </dependency>
```

Next we need to do following things for using Kafka.

First we need to define a custom Kafka deserializer. This will read bytes from a Kafka topic and convert it into `TemperatureEvent`. The following is the code to do this.

`EventDeserializationSchema.java`:

```
package com.demo.chapter05;

import java.io.IOException;
import java.nio.charset.StandardCharsets;

import org.apache.flink.api.common.typeinfo.TypeInformation;
import org.apache.flink.api.java.typeutils.TypeExtractor;
import org.apache.flink.streaming.util.serialization.DeserializationSchema;

public class EventDeserializationSchema implements
DeserializationSchema<TemperatureEvent> {

  public TypeInformation<TemperatureEvent> getProducedType() {
    return TypeExtractor.getForClass(TemperatureEvent.class);
  }

  public TemperatureEvent deserialize(byte[] arg0) throws IOException {
    String str = new String(arg0, StandardCharsets.UTF_8);

    String[] parts = str.split("=");
    return new TemperatureEvent(parts[0], Double.parseDouble(parts[1]));
  }

  public boolean isEndOfStream(TemperatureEvent arg0) {
    return false;
  }

}
```

Next we create topics in Kafka called `temperature`:

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-
factor 1 --partitions 1 --topic temperature
```

Now we move to Java code which would listen to these events in Flink streams:

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

    Properties properties = new Properties();
    properties.setProperty("bootstrap.servers", "localhost:9092");
    properties.setProperty("group.id", "test");

DataStream<TemperatureEvent> inputEventStream = env.addSource(
        new FlinkKafkaConsumer09<TemperatureEvent>("temperature", new
EventDeserializationSchema(), properties));
```

Next we will define the pattern to check if the temperature is greater than `26.0` degrees Celsius within `10` seconds:

```
Pattern<TemperatureEvent, ?> warningPattern = Pattern.<TemperatureEvent>
begin("first").subtype(TemperatureEvent.class).where(new
FilterFunction<TemperatureEvent>() {
          private static final long serialVersionUID = 1L;

          public boolean filter(TemperatureEvent value) {
            if (value.getTemperature() >= 26.0) {
              return true;
            }
            return false;
          }
        }).within(Time.seconds(10));
```

Next match this pattern with the stream of events and select the event. We will also add up the alert messages into results stream as shown here:

```
DataStream<Alert> patternStream = CEP.pattern(inputEventStream,
warningPattern)
        .select(new PatternSelectFunction<TemperatureEvent, Alert>() {
          private static final long serialVersionUID = 1L;

          public Alert select(Map<String, TemperatureEvent> event) throws
Exception {

            return new Alert("Temperature Rise Detected:" +
event.get("first").getTemperature()
                + " on machine name:" +
event.get("first").getMachineName());
          }

});
```

In order to know what the alerts were generated, we will print the results:

```
patternStream.print();
```

And we execute the stream:

```
env.execute("CEP on Temperature Sensor");
```

Now we are all set to execute the application. As and when we get messages in Kafka topics, the CEP will keep on executing.

The actual execution will looks like the following. Here is how we can provide sample input:

```
xyz=21.0
xyz=30.0
LogShaft=29.3
Boiler=23.1
Boiler=24.2
Boiler=27.0
Boiler=29.0
```

Here is how the sample output will look like:

```
Connected to JobManager at Actor[akka://flink/user/jobmanager_1#1010488393]
10/09/2016 18:15:55  Job execution switched to status RUNNING.
10/09/2016 18:15:55  Source: Custom Source(1/4) switched to SCHEDULED
10/09/2016 18:15:55  Source: Custom Source(1/4) switched to DEPLOYING
10/09/2016 18:15:55  Source: Custom Source(2/4) switched to SCHEDULED
10/09/2016 18:15:55  Source: Custom Source(2/4) switched to DEPLOYING
10/09/2016 18:15:55  Source: Custom Source(3/4) switched to SCHEDULED
10/09/2016 18:15:55  Source: Custom Source(3/4) switched to DEPLOYING
10/09/2016 18:15:55  Source: Custom Source(4/4) switched to SCHEDULED
10/09/2016 18:15:55  Source: Custom Source(4/4) switched to DEPLOYING
10/09/2016 18:15:55  CEPPatternOperator(1/1) switched to SCHEDULED
10/09/2016 18:15:55  CEPPatternOperator(1/1) switched to DEPLOYING
10/09/2016 18:15:55  Map -> Sink: Unnamed(1/4) switched to SCHEDULED
10/09/2016 18:15:55  Map -> Sink: Unnamed(1/4) switched to DEPLOYING
10/09/2016 18:15:55  Map -> Sink: Unnamed(2/4) switched to SCHEDULED
10/09/2016 18:15:55  Map -> Sink: Unnamed(2/4) switched to DEPLOYING
10/09/2016 18:15:55  Map -> Sink: Unnamed(3/4) switched to SCHEDULED
10/09/2016 18:15:55  Map -> Sink: Unnamed(3/4) switched to DEPLOYING
10/09/2016 18:15:55  Map -> Sink: Unnamed(4/4) switched to SCHEDULED
10/09/2016 18:15:55  Map -> Sink: Unnamed(4/4) switched to DEPLOYING
10/09/2016 18:15:55  Source: Custom Source(2/4) switched to RUNNING
10/09/2016 18:15:55  Source: Custom Source(3/4) switched to RUNNING
10/09/2016 18:15:55  Map -> Sink: Unnamed(1/4) switched to RUNNING
10/09/2016 18:15:55  Map -> Sink: Unnamed(2/4) switched to RUNNING
```

```
10/09/2016 18:15:55  Map -> Sink: Unnamed(3/4) switched to RUNNING
10/09/2016 18:15:55  Source: Custom Source(4/4) switched to RUNNING
10/09/2016 18:15:55  Source: Custom Source(1/4) switched to RUNNING
10/09/2016 18:15:55  CEPPatternOperator(1/1) switched to RUNNING
10/09/2016 18:15:55  Map -> Sink: Unnamed(4/4) switched to RUNNING
1> Alert [message=Temperature Rise Detected:30.0 on machine name:xyz]
2> Alert [message=Temperature Rise Detected:29.3 on machine name:LogShaft]
3> Alert [message=Temperature Rise Detected:27.0 on machine name:Boiler]
4> Alert [message=Temperature Rise Detected:29.0 on machine name:Boiler]
```

We can also configure a mail client and use some external web hook to send e-mail or messenger notifications.

The code for the application can be found on GitHub: `https://github.com/deshpandetanmay/mastering-flink`.

Summary

In this chapter, we learnt about CEP. We discussed the challenges involved and how we can use the Flink CEP library to solve CEP problems. We also learnt about Pattern API and the various operators we can use to define the pattern. In the final section, we tried to connect the dots and see one complete use case. With some changes, this setup can be used as it is present in various other domains as well.

In the next chapter, we will see how to use Flink's built-in Machine Learning library to solve complex problems.

6

Machine Learning Using FlinkML

In the previous chapter, we talked about how to solve complex event-processing problems using the Flink CEP library. In this chapter, we are going to see how to do machine learning using Flink's machine learning library, called FlinkML. FlinkML consists of a set of supported algorithms, which can be used to solve real-life use cases. Throughout this chapter, we will look at what algorithms are available in FlinkML and how to apply them.

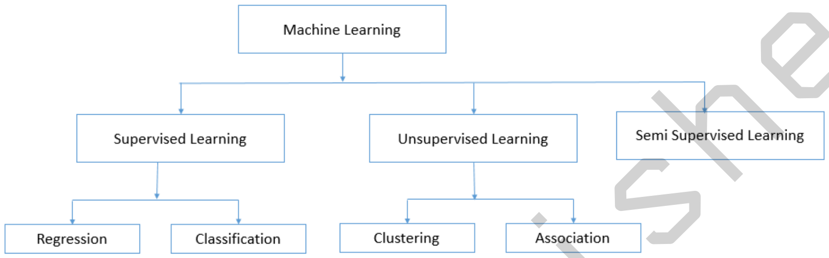
Before diving deep into FlinkML, let us first try to understand basic machine learning principles.

What is machine learning?

Machine learning is a stream of engineering which uses mathematics to allow machines to make classifications, predictions, recommendations, and so on, based on the data provided to them. This area is vast, and we could spend years discussing it. But in order to keep our discussion focused, we will discuss only what is required for the scope of this book.

Very broadly, machine learning can be divided into three big categories:

- Supervised learning
- Unsupervised learning
- Semi supervised learning



Overview of Machine Learning

The preceding diagram shows a broad classification of machine learning algorithms. Now let's discuss these in detail.

Supervised learning

In supervised learning, we are generally given an input dataset, which is a historical record of actual events. We are also given what the expected output should look like. Using the historical data, we choose which factors contributed to the results. Such attributes are called features. Using the historical data, we understand how the previous results were calculated and apply that same understanding to the data on which we want to make predictions.

Supervised learning can be again subdivided into:

- Regression
- Classification

Regression

In regression problems, we try to predict results using inputs from a continuous function. Regression means predicting the score of one variable based on the scores of another variable. The variable we will be predicting is called the criterion variable, and the variable from which we will be doing our predictions is called the predictor variable. There can be more than one predictor variable; in this case, we need to find the best fitting line, called the regression line.

> You can read more about regression at `https://en.wikipedia.org/wiki/Regression_analysis`.

Some very common algorithms used for solving regression problem are as follows:

- Logistic regression
- Decision trees
- Support Vector Machine (SVM)
- Naive Bayes
- Random forest
- Linear regression
- Polynomial regression

Classification

In classification, we predict the output in discrete results. Classification, being a part of supervised learning, also needs the input data and sample output to be given. Here, based on the features, we try to classify the results into sets of defined categories. For instance, based on the features given, classify records of people into male or female. Or, based on customer behavior, predict if he/she would buy a product or not. Or based on the e-mail content and sender, predict if the e-mail is spam or not. Refer to `https://en.wikipedia.org/wiki/Statistical_classification`.

In order to understand the difference between regression and classification, consider the example of stock data. Regression algorithms can help to predict the value of stock in upcoming days, while classification algorithms can help decide whether to buy the stock or not.

Unsupervised learning

Unsupervised learning does not give us any idea about how our results should look. Instead, it allows us to group data based on the features of the attributes. We derive the clustering based on the relationships among the records.

Unlike supervised learning, there is no validation we can do to verify our results, which means there is no feedback method to teach us whether we did right or wrong. Unsupervised learning is primarily based on clustering algorithms.

Clustering

In order to understand clustering more easily, let's consider an example; let's say we have 20,000 news articles on various topics and we have to group them based on their content . In this case, we can use clustering algorithms, which would group set of articles into small groups.

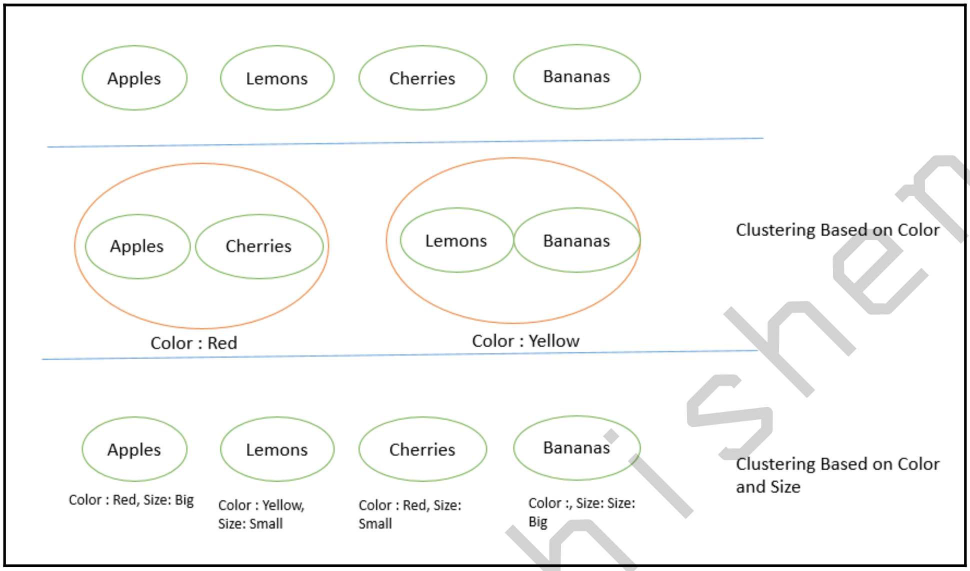
We can also consider the basic example of fruits. Let's say we have apples, bananas, lemons, and cherries in a fruit basket and we need to classify them into groups. If we look at their colors, we can classify them into two groups:

- **Red color group**: Apples and cherries
- **Yellow color group**: Bananas and lemons

Now we can do more grouping based on another feature, its size:

- **Red color and large size**: Apples
- **Red color and small size**: Cherries
- **Yellow color and large size**: Banana
- **Yellow color and small size**: Lemons

The following diagram shows a representation of clustering:



This way, by looking at more features, we can also do more clustering. Here, we don't have any training data and a variable to be predicted, unlike in supervised learning. Our only task is to learn more about the features and cluster the records based on inputs.

The following are some of the algorithms commonly used for clustering:

- K-means clustering
- Hierarchical clustering
- Hidden Markov models

Association

Association problems are more about learning, and making recommendations by defining association rules. Association rules could, for example, refer to the assumption that people who bought an iPhone are more likely to buy an iPhone case.

These days, many retail companies use these algorithms to make personalized recommendations. For instance, on `www.amazon.com`, if I tend to purchase product *X* and then Amazon recommends me product *Y* as well, there must be some association between the two.

Some of the algorithms based on these principles are as follows:

- Apriori algorithm
- Eclat algorithm
- FDP growth algorithm

Semi-supervised learning

Semi-supervised learning is a sub-class of supervised learning that considers unlabeled data for training. Generally, while training, it has a good amount of unlabeled data and only a very small amount of labeled data. Many researchers and machine learning practitioners have found that, when labeled data is used in conjunction with unlabeled data, the results are likely to be more accurate.

More details on semi-supervised learning can be found at `https://en.wikipedia.org/wiki/Semi-supervised_learning`.

FlinkML

FlinkML is a library of sets of algorithms supported by Flink that can be used to solve real-life use cases. The algorithms are built so that they can use the distributed computing power of Flink and make predictions or do clustering and so on with ease. Right now, there are only a few sets of algorithms supported, but the list is growing.

FlinkML is being built with the focus on ML developers needing to write minimal glue code. Glue code is code that helps bind various components together. Another goal of FlinkML is to keep the use of algorithms simple.

Flink exploits in-memory data streaming and executes iterative data processing natively. FlinkML allows data scientists to test their models locally, with a subset of data, and then execute them in cluster mode on bigger data.

FlinkML is inspired by scikit-learn and Spark's MLlib, which allows you to define data pipelines cleanly and solve machine learning problems in a distributed manner.

The following is the road map Flink's development team is aiming to build:

- Pipelines of transformers and learners
- Data pre-processing:
  - Feature scaling
  - Polynomial feature base mapper
  - Feature hashing
  - Feature extraction for text
  - Dimensionality reduction
- Model selection and performance evaluation:
  - Model evaluation using a variety of scoring functions
  - Cross-validation for model selection and evaluation
  - Hyper-parameter optimization
- Supervised learning:
  - Optimization framework
  - Stochastic Gradient Descent
  - L-BFGS
  - Generalized Linear Models
  - Multiple linear regression
  - LASSO, Ridge regression
  - Multi-class Logistic regression
  - Random forests
  - Support Vector Machines
  - Decision trees
- Unsupervised learning:
  - Clustering
  - K-means clustering
  - Principal Components Analysis
- Recommendation:
  - ALS
- Text analytics:
  - LDA
- Statistical estimation tools
- Distributed linear algebra
- Streaming ML

The algorithms highlighted are already part of the existing Flink source code. In the following section, we will look at how we can use those in practice.

Supported algorithms

To get started with FlinkML, we first need to add the following Maven dependency:

```
<!-- https://mvnrepository.com/artifact/org.apache.flink/flink-ml_2.11 -->
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-ml_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

Now let's try to understand the supported algorithms and how to use those.

Supervised learning

Flink supports three algorithms in the supervised-learning category. They are as follows:

- Support Vector Machine (SVM)
- Multiple linear regression
- Optimization framework

Let's get started learning about them one at a time.

Support Vector Machine

Support Vector Machines (**SVMs**) are supervised learning models, which analyze the data solving classification and regression problems. It helps classify objects into one category or another. It is non-probabilistic linear classification. There are various examples in which SVM can be used, such as the following:

- Regular data classification problems
- Text and hypertext classification problems
- Image classification problems
- Biological and other science problems

Flink supports SVM based on a soft-margin using a communication-efficient distributed dual-coordinate ascent algorithm.

Details on this algorithm are available at `https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/ml/svm.html#description`.

Flink uses **Stochastic Dual Coordinate Ascent** (**SDCA**) to solve the minimization problem. To make this algorithm efficient in a distributed environment, Flink uses the CoCoA algorithm, which calculates the SDCA on a local data block and then merges it into global state.

The implementation of this algorithm is based on the following paper: `https://arxiv.org/pdf/1409.1458v2.pdf`.

Now let's look at how we can solve a real-life problem using this algorithm. We will take the example of the Iris dataset (`https://en.wikipedia.org/wiki/Iris_flower_data_set`), consisting of four attributes which decide the species of Iris. The following is some sample data:

Sepal length	Sepal width	Petal length	Petal width	Species
5.1	3.5	1.4	0.2	1
5.6	2.9	3.6	1.3	2
5.8	2.7	5.1	1.9	3

Here, it is important to use categories in number format to be used as input to SVM:

Species code	Species name
1	Iris Setosa
2	Iris Versicolor
3	Iris Virginica

One more thing we need to do before using data for Flink's SVM algorithm is to convert this CSV data into LibSVM data.

LibSVM data is a special format used for specifying SVM datasets. More information on LibSVM is available at `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`.

To convert CSV data to LibSVM data, we will use some open-source Python code available at `https://github.com/zygmuntz/phraug/blob/master/csv2libsvm.py`.

To convert CSV to LibSVM, we need to execute the following command:

```
csv2libsvm.py <input file> <output file> [<label index = 0>] [<skip
headers = 0>]
```

Now let's get started with writing the program:

```
package com.demo.chapter06

import org.apache.flink.api.scala._
import org.apache.flink.ml.math.Vector
import org.apache.flink.ml.common.LabeledVector
import org.apache.flink.ml.classification.SVM
import org.apache.flink.ml.RichExecutionEnvironment

object MySVMApp {
  def main(args: Array[String]) {
    // set up the execution environment
    val pathToTrainingFile: String = "iris-train.txt"
    val pathToTestingFile: String = "iris-train.txt"
    val env = ExecutionEnvironment.getExecutionEnvironment

    // Read the training dataset, from a LibSVM formatted file
    val trainingDS: DataSet[LabeledVector] =
    env.readLibSVM(pathToTrainingFile)

    // Create the SVM learner
    val svm = SVM()
      .setBlocks(10)

    // Learn the SVM model
    svm.fit(trainingDS)

    // Read the testing dataset
    val testingDS: DataSet[Vector] =
    env.readLibSVM(pathToTestingFile).map(_.vector)

    // Calculate the predictions for the testing dataset
    val predictionDS: DataSet[(Vector, Double)] =
    svm.predict(testingDS)
```

```
      predictionDS.writeAsText("out")

      env.execute("Flink SVM App")
    }
  }
```

So, now we are all set to run the program, and you will the see the predicted output in the output folder.

The following is the code:

```
(SparseVector((0,5.1), (1,3.5), (2,1.4), (3,0.2)),1.0)
(SparseVector((0,4.9), (1,3.0), (2,1.4), (3,0.2)),1.0)
(SparseVector((0,4.7), (1,3.2), (2,1.3), (3,0.2)),1.0)
(SparseVector((0,4.6), (1,3.1), (2,1.5), (3,0.2)),1.0)
(SparseVector((0,5.0), (1,3.6), (2,1.4), (3,0.2)),1.0)
(SparseVector((0,5.4), (1,3.9), (2,1.7), (3,0.4)),1.0)
(SparseVector((0,4.6), (1,3.4), (2,1.4), (3,0.3)),1.0)
(SparseVector((0,5.0), (1,3.4), (2,1.5), (3,0.2)),1.0)
(SparseVector((0,4.4), (1,2.9), (2,1.4), (3,0.2)),1.0)
(SparseVector((0,4.9), (1,3.1), (2,1.5), (3,0.1)),1.0)
(SparseVector((0,5.4), (1,3.7), (2,1.5), (3,0.2)),1.0)
(SparseVector((0,4.8), (1,3.4), (2,1.6), (3,0.2)),1.0)
(SparseVector((0,4.8), (1,3.0), (2,1.4), (3,0.1)),1.0)
```

We can also fine-tune the results by setting various parameters:

Parameter	Description
`Blocks`	Sets the number of blocks into which the input data will be split. It is ideal to set this number equal to the parallelism you want to achieve. On each block, local stochastic dual-coordinate ascent is performed. The default value is `None`.
`Iterations`	Sets the number of iterations of the outer loop method, for example, the amount of time the SDCA method should applied on blocked data. The default value is `10`.
`LocalIterations`	Defines the maximum number of SDCA iterations that need to be executed locally. The default value is `10`.
`Regularization`	Sets the regularization constant of the algorithm. The higher you set the value, the smaller the 2 norm of the weighted vector be. The default value is `1`.

`StepSize`	Defines the initial step size for the updates of weight vector. This value needs to be set up in case the algorithm becomes unstable. The default value is `1.0`.
`ThresholdValue`	Defines the limiting value for the decision function. The default value is `0.0`.
`OutputDecisionFunction`	Setting this to true will return the hyperplane distance for each example. Setting it to false will return the binary label.
`Seed`	Sets the random long integer. This will be used to initialize the random number generator.

Multiple Linear Regression

Multiple Linear Regression (**MLR**) is an extension of simple linear regression where more than one independent variable (*X*) is used to determine the single independent variable (*Y*). The predicted value is a linear transformation of input variables such that the sum of squared deviations of the observed and predicted is minimum.

MLR tries to model the relationship between multiple explanatory variables and response variables by fitting a linear equation.

A more detailed explanation of MLR can be found on this link `http://www.stat.yale.edu/Courses/1997-98/101/linmult.htm`.

Let's try solving the same classification problem of Iris dataset using MLR now. First we need the training dataset on which we can train our mode.

Here we will be using the same data files we used in previous section on SVM. So now we have `iris-train.txt` and `iris-test.txt` which are converted into LibSVM format.

The following code snippet shows how MLR can be used:

```
package com.demo.flink.ml

import org.apache.flink.api.scala._
import org.apache.flink.ml._
import org.apache.flink.ml.common.LabeledVector
import org.apache.flink.ml.math.DenseVector
import org.apache.flink.ml.math.Vector
import org.apache.flink.ml.preprocessing.Splitter
import org.apache.flink.ml.regression.MultipleLinearRegression
```

```
object MLRJob {
  def main(args: Array[String]) {
    // set up the execution environment
    val env = ExecutionEnvironment.getExecutionEnvironment
    val trainingDataset = MLUtils.readLibSVM(env, "iris-train.txt")
    val testingDataset = MLUtils.readLibSVM(env, "iris-test.txt").map {
    lv => lv.vector }
    val mlr = MultipleLinearRegression()
      .setStepsize(1.0)
      .setIterations(5)
      .setConvergenceThreshold(0.001)

    mlr.fit(trainingDataset)

    // The fitted model can now be used to make predictions
    val predictions = mlr.predict(testingDataset)

    predictions.print()

  }
}
```

The complete code and the data files are available for download on `https://github.com/deshpandetanmay/mastering-flink/tree/master/chapter06`. We can also fine-tune the results by setting various parameters:

Parameter	**Description**
`Iterations`	Sets the maximum number of iterations. The default value is `10`.
`Stepsize`	The step size of the gradient descent method. This value controls how far the gradient descent method can move in the opposite direction. Tuning this parameter is very important to get better results. The default value is `0.1`.
`Convergencethreshold`	The threshold for the relative change of the sum of squared residuals until the iteration is stopped. The default value is `None`.
`Learningratemethod`	`Learningratemethod` is used to calculate the learning rate of each iteration.

Optimization framework

Optimization framework in Flink is a developer-friendly package which can be used to solve optimization problems. This is not a specific algorithm to solve exact problems, but it is the basis of every machine learning problem.

Generally, it is about finding the model, with a set of parameters, with a minimization function. FlinkML supports **Stochastic Gradient Descent** (**SGD**), with the following types of regularizations:

Regularization function	Class name
L1 regularization	`GradientDescentL1`
L2 regularization	`GradientDescentL2`
No regularization	`SimpleGradient`

The following code snippet shows how to use SGD using FlinkML:

```
// Create SGD solver
val sgd = GradientDescentL1()
  .setLossFunction(SquaredLoss())
  .setRegularizationConstant(0.2)
  .setIterations(100)
  .setLearningRate(0.01)
  .setLearningRateMethod(LearningRateMethod.Xu(-0.75))


// Obtain data
val trainingDS: DataSet[LabeledVector] = ...

// Optimize the weights, according to the provided data
val weightDS = sgd.optimize(trainingDS)
```

We can also use parameters to fine-tune the algorithm:

Parameter	Description
`LossFunction`	Flink supports the following loss functions: • Squared Loss • Hinged Loss • Logistic Loss • The default is `None`

`RegularizationConstant`	The weight of regularization to be applied. The default value is `0.1`.
`Iterations`	The maximum number of iterations to be performed. The default is `10`.
`ConvergenceThreshold`	The threshold for relative change of the sum of squared residuals until the iteration is stopped. The default value is `None`.
`LearningRateMethod`	This method is used to calculate the learning rate of each iteration.
`LearningRate`	This is the initial learning rate for the gradient descent method.
`Decay`	The default value is `0.0`.

Recommendations

Recommendation engines are one of the most interesting and heavily used machine learning techniques to provide user-based and item-based recommendations. E-commerce companies such as Amazon use recommendation engines to personalize recommendations based on the purchasing patterns and review ratings of its customers.

Flink also supports ALS-based recommendations. Let's look at ALS in more detail.

Alternating Least Squares

The **Alternating Least Squares** (**ALS**) algorithm factorizes a given matrix, *R*, into two factors, *U* and *V*, such that $R \approx U^T V$

In order to better understand the application of this algorithm, let's assume that we have a dataset which contains the rating, *r*, provided by user *u* for book *b*.

Here is a sample data format (`user_id`, `book_id`, `rating`):

```
1  10 1
1  11 2
1  12 5
1  13 5
1  14 5
1  15 4
1  16 5
1  17 1
```

```
1  18 5
2  10 1
2  11 2
2  15 5
2  16 4.5
2  17 1
2  18 5
3  11 2.5
3  12 4.5
3  13 4
3  14 3
3  15 3.5
3  16 4.5
3  17 4
3  18 5
4  10 5
4  11 5
4  12 5
4  13 0
4  14 2
4  15 3
4  16 1
4  17 4
4  18 1
```

Now we can feed this information to the ALS algorithm and start getting recommendations from it. The following is a code snippet for using ALS:

```
package com.demo.chapter06

import org.apache.flink.api.scala._
import org.apache.flink.ml.recommendation._
import org.apache.flink.ml.common.ParameterMap

object MyALSApp {
  def main(args: Array[String]): Unit = {

    val env = ExecutionEnvironment.getExecutionEnvironment
    val inputDS: DataSet[(Int, Int, Double)] = env.readCsvFile[(Int,
    Int, Double)]("input.csv")

    // Setup the ALS learner
    val als = ALS()
      .setIterations(10)
      .setNumFactors(10)
      .setBlocks(100)
      .setTemporaryPath("tmp")
    // Set the other parameters via a parameter map
    val parameters = ParameterMap()
      .add(ALS.Lambda, 0.9)
      .add(ALS.Seed, 42L)

    // Calculate the factorization
    als.fit(inputDS, parameters)

    // Read the testing dataset from a csv file
    val testingDS: DataSet[(Int, Int)] = env.readCsvFile[(Int, Int)]
    ("test-data.csv")

    // Calculate the ratings according to the matrix factorization
    val predictedRatings = als.predict(testingDS)

    predictedRatings.writeAsCsv("output")

    env.execute("Flink Recommendation App")
  }
}
```

Once you execute the application, you'll get the results as recommendations. Like with other algorithms, you can fine-tune the parameters to get better results:

Parameter	Description
`NumFactors`	The number of latent factors to use for the underlying model. The default value is `10`.
`Lambda`	This is a regularization factor; we can tune this parameter for better results. The default is `1`.
`Iterations`	The maximum number of iterations to be performed. The default is `10`.
`Blocks`	The number of blocks in which user and item matrix are grouped. The fewer the blocks, the less data is sent redundantly. The default value is `None`.
`Seed`	The seed value to initiate the item matrix generator. The default is `0`.
`TemporaryPath`	This is a path to be used for storing intermediate results.

Unsupervised learning

Now let's try to understand what FinkML offers for unsupervised learning. For now, it supports only one algorithm, called the k Nearest Neighbor join algorithm.

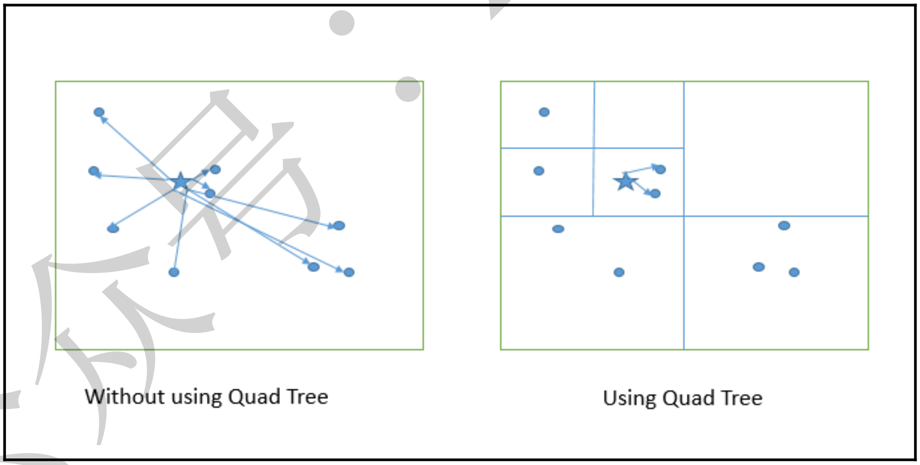
k Nearest Neighbour join

The **k Nearest Neighbor** (**kNN**) algorithm is designed to find the k Nearest Neighbour from a dataset for every object in another dataset. It is one of the most widely used solutions in many data-mining algorithms. The kNN is an expensive operation, as it is a combination of finding the k nearest neighbor and performing a join. Considering the volume of the data, it is very difficult to perform this operation on a centralized single machine, hence it is always good to have solutions that can work on distributed architecture. The FlinkML algorithm provides kNN on a distributed environment.

A research paper describing the implementation of kNN on a distributed environment can be found here: `https://arxiv.org/pdf/1207.0141v1.pdf`.

Here, the idea is to compute the distance between every training and testing point and then find the nearest points for a given point. Computing the distance between each point is a time-consuming activity, which is eased out in Flink by implementing quad trees.

Using quad trees reduces the computation by partitioning the dataset. This reduces the computation to the subset of data only. The following diagram shows computation with and without quad trees:



Without using Quad Tree

Using Quad Tree

You can find a detailed discussion on using quad trees to calculate the nearest neighbors here: `http://danielblazevski.github.io/assets/player/KeynoteDHTMLPlayer.html`.

It's not always the case that quad trees will perform better. If the data is spatial, the quad trees might be the worst choice. But as a developer, we don't need to worry about it as FlinkML takes care of deciding whether to use quad tree or not based on the data available.

The following code snippet shows how to use kNN join in FlinkML:

```
import
org.apache.flink.api.common.operators.base.CrossOperatorBase.CrossHint
import org.apache.flink.api.scala._
import org.apache.flink.ml.nn.KNN
import org.apache.flink.ml.math.Vector
import org.apache.flink.ml.metrics.distances.SquaredEuclideanDistanceMetric

val env = ExecutionEnvironment.getExecutionEnvironment

// prepare data
val trainingSet: DataSet[Vector] = ...
val testingSet: DataSet[Vector] = ...

val knn = KNN()
  .setK(3)
  .setBlocks(10)
  .setDistanceMetric(SquaredEuclideanDistanceMetric())
  .setUseQuadTree(false)
  .setSizeHint(CrossHint.SECOND_IS_SMALL)

// run knn join
knn.fit(trainingSet)
val result = knn.predict(testingSet).collect()
```

The following are some parameters we can use to fine-tune the results:

Parameter	**Description**
`K`	The number of nearest neighbours to search for. The default is `5`.
`DistanceMetric`	Sets the distance metric to be used to calculate the distance between two points. By default, Euclidian Distance Metric is used.
`Blocks`	The number of blocks into which the input data should be split. It is ideal to set this number equal to the degree of parallelism.
`UseQuadTree`	Sets whether to use quad tree for processing or not. The default value is `None`. If nothing is specified, the algorithm decides on its own.

Utilities

FlinkML supports various extensible utilities, which can be handy while doing data analysis and predictions. One such utility is distance metrics. Flink supports a set of distance metrics which can be used. The following link shows Flink-supported distance metrics: https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/ml/distance_metrics.html.

If any of the previously mentioned algorithms do not satisfy your needs, you can think about writing your own custom distance algorithm. The following code snippet shows how to do so:

```
class MyDistance extends DistanceMetric {
  override def distance(a: Vector, b: Vector) = ... // your implementation
}

object MyDistance {
  def apply() = new MyDistance()
}

val myMetric = MyDistance()
```

A good application of using distance metrics is the kNN join algorithm, where you can set the distance metric to use.

Another important utility is `Splitter`, which can be used for cross validation. In some cases, we may not have a test dataset to validate our results. In such cases, we can split the training dataset using `Splitter`.

The following is an example:

```
// A Simple Train-Test-Split
val dataTrainTest: TrainTestDataSet = Splitter.trainTestSplit(data, 0.6,
true)
```

In the preceding example, we are splitting the training dataset into portions of 60% and 40% of the actual data.

There is another method to fetch better results, called `TrainTestHoldout` split. Here, we use some portion of the data for training, some for testing, and another set for final result validations. The following snippet shows how to do it:

```
// Create a train test holdout DataSet
val dataTrainTestHO: trainTestHoldoutDataSet =
Splitter.trainTestHoldoutSplit(data, Array(6.0, 3.0, 1.0))
```

We can use another strategy, called K fold splits. In this method, the training set is split into *k* equal size folds. Here, an algorithm is created for each fold and then validated against its testing set. The following code shows how to do k-fold splits:

```
// Create an Array of K TrainTestDataSets
val dataKFolded: Array[TrainTestDataSet] =  Splitter.kFoldSplit(data, 10)
```

We can also use **Multi Random Splits**; here we can specify how many datasets to create and of what portion of the original:

```
// create an array of 5 datasets of 1 of 50%, and 5 of 10% each
val dataMultiRandom: Array[DataSet[T]] = Splitter.multiRandomSplit(data,
Array(0.5, 0.1, 0.1, 0.1, 0.1))
```

Data pre processing and pipelines

Flink supports Python scikit-learn style pipeline. A pipeline in FlinkML is feature to chain multiple transformers and predictors in one go. In general, many data scientists would like to see and build the flow of machine learning application with ease. Flink allows them to do so using the concept of pipelines.

In general, there are three building blocks of ML pipelines:

- **Estimator:** Estimator performs the actual training of a model using a `fit` method. For example, finding correct weights in a linear regression model.
- **Transformer:** Transformer as the name suggests have a `transform` method which can help in scaling the input.
- **Predictor:** Predictors have the `predict` method which applies the algorithm for generating predictions, for example, SVM or MLR.

A pipeline is a chain of estimator, transformers and predictor. The predictor is the end of a pipeline and nothing can be chained after that.

Flink supports various data pre-processing tools which would help us advance the results. Let's start understanding the details.

Polynomial features

The polynomial feature is a transformer which maps a vector into the polynomial feature space of degree *d*. Polynomial feature helps in solving classification problems by changing the graph of the function. Let's try to understand this by an example:

- Consider a linear formula: *F(x,y) = 1*x + 2*y;*
- Imagine we have two observations:
    - *x=12* and *y=2*
    - *x=5* and *y =5.5*

In both cases, we get *f() = 16*. If these observations belong to two different classes then we cannot differentiate between the two. Now if we add one more feature called *z* which is combination of previous two features *z = x+y*.

So now *f(x,y,z) = 1*x + 2*y + 3*z*

Now the same observations would be

- *(1*12)+ (2*2) + (3*24) = 88*
- *(1*5)+ (2*5.5) + (3*27.5) = 98.5*

This way adding a new feature using existing features can help us get better results. Flink polynomial features allows us to do the same with pre-build functions.

In order to use polynomial features in Flink, we have the following code:

```
val polyFeatures = PolynomialFeatures()
      .setDegree(3)
```

Standard scaler

Standard scaler helps scale the input data using the user-specified mean and variance. If the user does not specify any values then default mean is `0` and standard deviation would be `1`. Standard scaler is a transformer which has `fit` and `transform` method.

First we need to define the values for mean and standard deviation as shown in the following code snippet:

```
  val scaler = StandardScaler()
      .setMean(10.0)
      .setStd(2.0)
```

Next we need to let it learn about mean and standard deviation of training dataset as shown in the following code snippet:

```
scaler.fit(trainingDataset)
```

And finally we scale the provided data using the user-defined mean and standard deviation as shown in the following code snippet:

```
val scaledDS = scaler.transform(trainingDataset)
```

Now we can use this scaled input data to do further transformation and analysis.

MinMax scaler

MinMax scaler is like standard scaler but the only difference is it makes sure that scaling of each feature lies between user-defined `min` and `max` values.

The following code snippet shows how to use this:

```
val minMaxscaler = MinMaxScaler()
.setMin(1.0)
.setMax(3.0)
minMaxscaler.fit(trainingDataset)
val scaledDS = minMaxscaler.transform(trainingDataset)
```

Thus, we can use these data pre-processing operations to enhance the results. These can also be combined in pipelines to create the workflow.

The following code snippet shows how to use these data pre-processing operations in pipeline:

```
// Create pipeline PolynomialFeatures -> MultipleLinearRegression
val pipeline = polyFeatures.chainPredictor(mlr)
// train the model
pipeline.fit(scaledDS)
// The fitted model can now be used to make predictions
val predictions = pipeline.predict(testingDataset)
predictions.print()
```

The complete code is available on GitHub at `https://github.com/deshpandetanmay/mastering-flink/tree/master/chapter06`.

Summary

In this chapter, we learned about the different types of machine learning algorithm. We looked at various supervised and unsupervised algorithms, and their respective examples. We also looked at various utilities provided by FlinkML, which can be very handy during data analysis. Later we looked at data pre-processing operations and how to use them in pipelines.

In the following chapter, we will look at the graph-processing capabilities of Flink.

7
Flink Graph API - Gelly

We are living in the era of social media where everyone is connected to each other by some means. Every single object is in a relationship with another. Facebook and Twitter are excellent examples of social graphs, where *x* is friends with *y* and *p* is following *q*, and so on. These graphs are so huge that we need an engine which can process them efficiently. If we are surrounded by such graphs, it is very important to analyze them in order to get more insights about their relationships and next-level relationships.

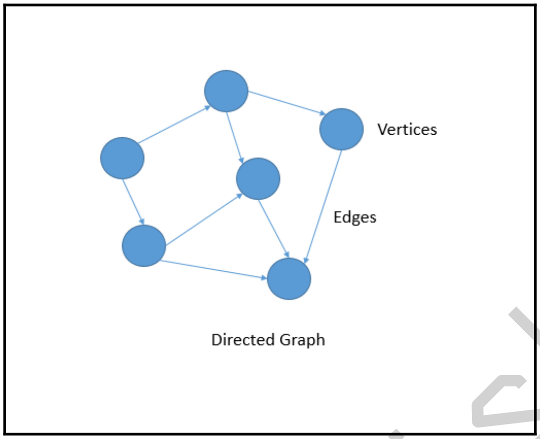
There are various technologies in the market which help us analyze such graphs, for example, graph databases such as Titan and Neo4J, graph processing libraries such as Spark GraphX and Flink Gelly, and so on. In this chapter, we are going to understand the details of graphs and how we can use Flink Gelly to analyze graph data.

So let's get started.

What is a graph?

In the computer science field, a graph is a means of representing relationships amongst the object. It consists of a set of vertices connected via edges. **Vertices** are objects on a plane, identified by co-ordinates or some unique id/name while **Edges** are the connecting links between the vertices having certain weights or the relationship. A graph can be directed or undirected. In a directed graph, the edges are directed from one vertex to other while there is no direction for edges in undirected graph.

The following diagram shows the basic representation of a directed graph:



A graph structure can be used for various purposes, such as finding the shortest path to a certain destination, or it could be used for finding out the degree of relationship between certain vertices, or it could be used for finding out the nearest neighbor.

Now let's dive deep into Flink's Graph API – Gelly.

Flink graph API – Gelly

Flink provides a graph processing library called Gelly to simplify the development of graph analysis. It provides data structures to store and represent graph data and it provides methods to analyze the graphs. In Gelly, we can transform graphs from one state to another using Flink's higher-level functions. It also provides a set of algorithms used for detailed graph analysis.

Gelly is currently available as a part of the Flink libraries, so we need to add a Maven dependency in our programs to use it.

Java dependency:

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-gelly_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

Scala dependency:

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-gelly-scala_2.11</artifactId>
    <version>1.1.4</version>
</dependency>
```

Now let's look at various options we have in order to use Gelly effectively.

Graph representation

In Gelly, a graph is represented as dataset of nodes and datasets of edges.

Graph nodes

A graph node is represented by a `Vertex` data type. A `Vertex` data type consists of a unique ID and an optional value. A unique ID should implement a comparable interface because, while doing graph processing, we compare the nodes by their IDs. A `Vertex` can have a value or it can have a null value as well. A null-valued vertex is defined by the type, `NullValue`.

The following code snippets show how to create nodes:

In Java:

```
// A vertex with a Long ID and a String value
Vertex<Long, String> v = new Vertex<Long, String>(1L, "foo");

// A vertex with a Long ID and no value
Vertex<Long, NullValue> v = new Vertex<Long, NullValue>(1L,
NullValue.getInstance());
```

In Scala:

```
// A vertex with a Long ID and a String value
val v = new Vertex(1L, "foo")

// A vertex with a Long ID and no value
val v = new Vertex(1L, NullValue.getInstance())
```

Graph edges

Similarly, an edge can be defined by the type, Edge. An Edge has a source node ID, a destination node ID, and an optional value. The value represents the degree or weight of relationship. Source and Target Ids need to be of same type. Edges with no value can be defined using NullValue.

The following code snippets shows Edge definitions in Java and Scala:

In Java:

```
// Edge connecting Vertices with Ids 1 and 2 having weight 0.5

Edge<Long, Double> e = new Edge<Long, Double>(1L, 2L, 0.5);

Double weight = e.getValue(); // weight = 0.5
```

In Scala:

```
// Edge connecting Vertices with Ids 1 and 2 having weight 0.5

val e = new Edge(1L, 2L, 0.5)

val weight = e.getValue // weight = 0.5
```

In Gelly, a graph is always directed from a source to a destination vertex. In order to show an undirected graph, we should add another edge representing a connection from the destination to the source and back.

The following code snippet represents a directed graph in Gelly:

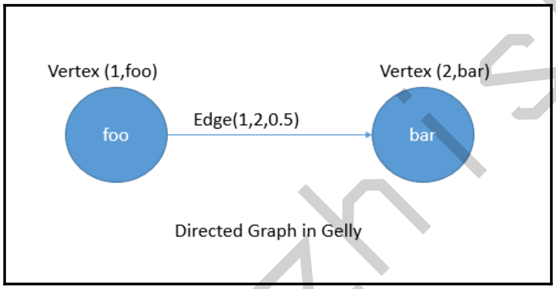
In Java:

```
// A vertex with a Long ID and a String value
Vertex<Long, String> v1 = new Vertex<Long, String>(1L, "foo");

// A vertex with a Long ID and a String value
Vertex<Long, String> v2 = new Vertex<Long, String>(2L, "bar");

// Edge connecting Vertices with Ids 1 and 2 having weight 0.5

Edge<Long, Double> e = new Edge<Long, Double>(1L, 2L, 0.5);
```

In Scala:

```
// A vertex with a Long ID and a String value
val v1 = new Vertex(1L, "foo")

// A vertex with a Long ID and a String value
val v2 = new Vertex(1L, "bar")

// Edge connecting Vertices with Ids 1 and 2 having weight 0.5

val e = new Edge(1L, 2L, 0.5)
```

The following is its visual representation for the same:



The following code snippet represents the vertex and edge definitions for an undirected graph in Gelly:

In Java:

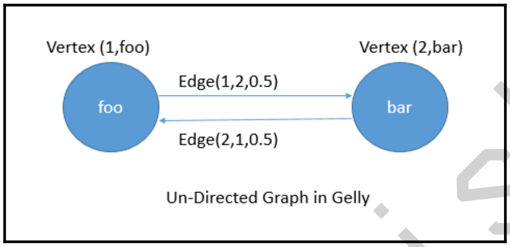
```
// A vertex with a Long ID and a String value
Vertex<Long, String> v1 = new Vertex<Long, String>(1L, "foo");

// A vertex with a Long ID and a String value
Vertex<Long, String> v2 = new Vertex<Long, String>(2L, "bar");

// Edges connecting Vertices with Ids 1 and 2 having weight 0.5

Edge<Long, Double> e1 = new Edge<Long, Double>(1L, 2L, 0.5);

Edge<Long, Double> e2 = new Edge<Long, Double>(2L, 1L, 0.5);
```

In Scala:

```
// A vertex with a Long ID and a String value
val v1 = new Vertex(1L, "foo")

// A vertex with a Long ID and a String value
```

```
val v2 = new Vertex(1L, "bar")

// Edges connecting Vertices with Ids 1 and 2 having weight 0.5

val e1 = new Edge(1L, 2L, 0.5)

val e2 = new Edge(2L, 1L, 0.5)
```

The following is its visual representation for the same:



Graph creation

In Flink Gelly, graphs can be created in multiple ways. The following are some examples.

From dataset of edges and vertices

The following code snippets represent how we create graphs using the dataset of edges and optional vertices:

In Java:

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

DataSet<Vertex<String, Long>> vertices = ...

DataSet<Edge<String, Double>> edges = ...

Graph<String, Long, Double> graph = Graph.fromDataSet(vertices, edges,
env);
```

In Scala:

```
val env = ExecutionEnvironment.getExecutionEnvironment

val vertices: DataSet[Vertex[String, Long]] = ...

val edges: DataSet[Edge[String, Double]] = ...

val graph = Graph.fromDataSet(vertices, edges, env)
```

From dataset of tuples representing edges

The following code snippets represent how we create graphs using the dataset of Tuple2 representing edges. Here Gelly automatically converts Tuple2 into edges having source and destination vertices IDs and null value.

In Java:

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

DataSet<Tuple2<String, String>> edges = ...

Graph<String, NullValue, NullValue> graph = Graph.fromTuple2DataSet(edges,
env);
```

In Scala:

```
val env = ExecutionEnvironment.getExecutionEnvironment

val edges: DataSet[(String, String)] = ...

val graph = Graph.fromTuple2DataSet(edges, env)
```

The following code snippets represent how we create graphs using the dataset of Tuple3 representing edges. Here vertices are represented using Tuple2 while edge using Tuple3 are represented with information about source, destination vertex, and weight. We can also read the set of values from CSV files:

In Java:

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

DataSet<Tuple2<String, Long>> vertexTuples =
env.readCsvFile("path/to/vertex/input.csv").types(String.class,
Long.class);
```

```
DataSet<Tuple3<String, String, Double>> edgeTuples =
env.readCsvFile("path/to/edge/input.csv").types(String.class, String.class,
Double.class);

Graph<String, Long, Double> graph = Graph.fromTupleDataSet(vertexTuples,
edgeTuples, env);
```

In Scala:

```
val env = ExecutionEnvironment.getExecutionEnvironment

val vertexTuples = env.readCsvFile[String,
Long]("path/to/vertex/input.csv")

val edgeTuples = env.readCsvFile[String, String,
Double]("path/to/edge/input.csv")

val graph = Graph.fromTupleDataSet(vertexTuples, edgeTuples, env)
```

From CSV files

The following code snippets represent how we create graphs using the CSV file reader. CSV files should have data represented in the form vertices and edges.

The following code snippet creates a graph from CSV files of the format, source, target, weight, for edges and ID, name for vertices:

```
val env = ExecutionEnvironment.getExecutionEnvironment

// create a Graph with String Vertex IDs, Long Vertex values and Double
Edge values
val graph = Graph.fromCsvReader[String, Long, Double](
    pathVertices = "path/to/vertex/input.csv",
    pathEdges = "path/to/edge/input.csv",
    env = env)
```

We can use a vertex value initializer as well by defining a `map` function while creating the graph:

```
val simpleGraph = Graph.fromCsvReader[Long, Double, NullValue](
        pathEdges = "path/to/edge/input.csv",
        vertexValueInitializer = new MapFunction[Long, Double]() {
            def map(id: Long): Double = {
                id.toDouble
            }
        },
        env = env)
```

From collection lists

We can also create graphs from a collection of list. The following code snippet show how we create graphs from a list of edges and vertices:

In Java:

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

List<Vertex<Long, Long>> vertexList = new ArrayList...

List<Edge<Long, String>> edgeList = new ArrayList...

Graph<Long, Long, String> graph = Graph.fromCollection(vertexList,
edgeList, env);
```

In Scala:

```
val env = ExecutionEnvironment.getExecutionEnvironment

val vertexList = List(...)

val edgeList = List(...)

val graph = Graph.fromCollection(vertexList, edgeList, env)
```

If no vertex input is provided, then we can think of providing a `map` initialization function as shown here:

```
val env = ExecutionEnvironment.getExecutionEnvironment

// initialize the vertex value to be equal to the vertex ID
val graph = Graph.fromCollection(edgeList,
    new MapFunction[Long, Long] {
       def map(id: Long): Long = id
    }, env)
```

Graph properties

The following table shows the set of methods available to retrieve graph properties:

Property	In Java	In Scala
`getVertices` dataset	`DataSet<Vertex<K, VV>> getVertices()`	`getVertices: DataSet[Vertex[K, VV]]`
`getEdges` dataset	`DataSet<Edge<K, EV>> getEdges()`	`getEdges: DataSet[Edge[K, EV]]`
`getVertexIds`	`DataSet<K> getVertexIds()`	`getVertexIds: DataSet[K]`
`getEdgeIds`	`DataSet<Tuple2<K, K>> getEdgeIds()`	`getEdgeIds: DataSet[(K, K)]`
Get dataset of vertex IDs and `inDegrees` for all vertices	`DataSet<Tuple2<K, LongValue>> inDegrees()`	`inDegrees: DataSet[(K, LongValue)]`
Get dataset of vertex IDs and `outDegrees` for all vertices	`DataSet<Tuple2<K, LongValue>> outDegrees()`	`outDegrees: DataSet[(K, LongValue)]`
Get dataset of vertex IDs and in, `getDegree` for all vertices	`DataSet<Tuple2<K, LongValue>> getDegrees()`	`getDegrees: DataSet[(K, LongValue)]`
Get `numberOfVertices`	`long numberOfVertices()`	`numberOfVertices: Long`
Get `numberOfEdges`	`long numberOfEdges()`	`numberOfEdges: Long`
`getTriplets` provides triplets consisting of source vertex, target vertex and the edge	`DataSet<Triplet<K, VV, EV>> getTriplets()`	`getTriplets: DataSet[Triplet[K, VV, EV]]`

Graph transformations

Gelly provides various transformation operations which helps transforming graphs from one form to another. The following are some transformations we can do by using Gelly.

Map

Gelly provides map transformations which keeps the vertices and edge IDs intact and transforms the values as per given in the function. This operation always returns a new graph. The following code snippet shows how to use it.

In Java:

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
Graph<Long, Long, Long> graph = Graph.fromDataSet(vertices, edges, env);

// increment each vertex value by 5
Graph<Long, Long, Long> updatedGraph = graph.mapVertices(
        new MapFunction<Vertex<Long, Long>, Long>() {
          public Long map(Vertex<Long, Long> value) {
            return value.getValue() + 5;
          }
        });
```

In Scala:

```
val env = ExecutionEnvironment.getExecutionEnvironment
val graph = Graph.fromDataSet(vertices, edges, env)

// increment each vertex value by 5
val updatedGraph = graph.mapVertices(v => v.getValue + 5)
```

Translate

Translate is a special function that allows translating vertex IDs, vertex values, edge IDs, and so on. Translation is performed using a custom map function provided by the user. The following code snippet shows how we use the translate function.

In Java:

```
// translate each vertex and edge ID to a String
Graph<String, Long, Long> updatedGraph = graph.translateGraphIds(
        new MapFunction<Long, String>() {
          public String map(Long id) {
            return id.toString();
          }
        });

// translate vertex IDs, edge IDs, vertex values, and edge values to
LongValue
Graph<LongValue, LongValue, LongValue> updatedGraph = graph
```

```
                .translateGraphIds(new LongToLongValue())
                .translateVertexValues(new LongToLongValue())
                .translateEdgeValues(new LongToLongValue())
```

In Scala:

```
// translate each vertex and edge ID to a String
val updatedGraph = graph.translateGraphIds(id => id.toString)
```

Filter

`FilterFunction` can be used to filter out vertices and edges based on certain conditions. `filterOnEdges` will create a sub-graph of the original one. In this operation, the vertices dataset remains unchanged. Similarly, `filterOnVertices` applies the filter on vertices values. In this case, edges that do not find target nodes are removed. The following code snippet shows how we use `FilterFunction` in Gelly.

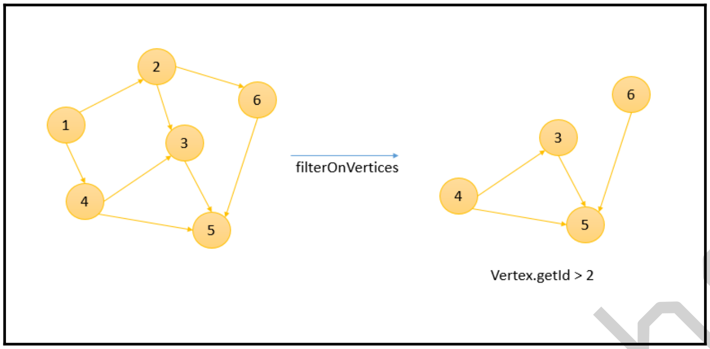
In Java:

```
Graph<Long, Long, Long> graph = ...

graph.subgraph(
    new FilterFunction<Vertex<Long, Long>>() {
           public boolean filter(Vertex<Long, Long> vertex) {
          // keep only vertices with positive values
          return (vertex.getValue() > 2);
         }
       },
    new FilterFunction<Edge<Long, Long>>() {
        public boolean filter(Edge<Long, Long> edge) {
          // keep only edges with negative values
          return (edge.getTarget() == 3);
        }
    })
```

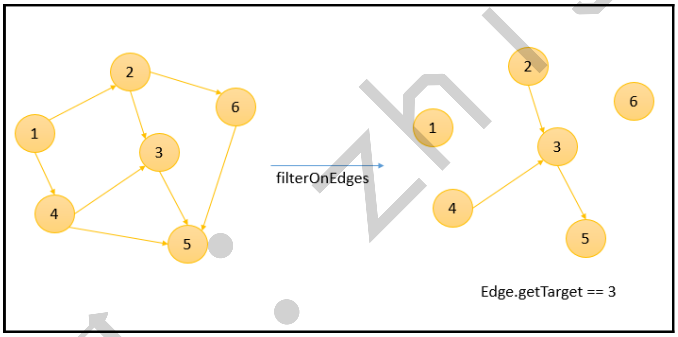
In Scala:

```
val graph: Graph[Long, Long, Long] = ...
graph.subgraph((vertex => vertex.getValue > 2), (edge => edge.getTarget ==
3))
```

The following is a graphical representation of the preceding code:



Similarly, the following diagram shows `filterOnEdges`:



Join

The `join` operation helps in joining vertices and edge datasets with other datasets. The `joinWithVertices` method joins with vertex IDs and the first field of Tuple2. The `join` method returns a new graph. Similarly, input datasets can be joined with edges. There are three ways we can join edges:

- `joinWithEdges`: Joins graphs with the Tuple3 dataset on composite keys of both source and target vertex IDs
- `joinWithEdgeOnSource`: Joins with the Tuple2 dataset on the source key and first attribute of the Tuple2 dataset
- `joinWithEdgeOnTarget`: Joins with the Tuple2 dataset on the target key and first attribute of the Tuple2 dataset

The following code snippet shows how to use joins in Gelly:

In Java:

```
Graph<Long, Double, Double> network = ...

DataSet<Tuple2<Long, LongValue>> vertexOutDegrees = network.outDegrees();

// assign the transition probabilities as the edge weights
Graph<Long, Double, Double> networkWithWeights =
network.joinWithEdgesOnSource(vertexOutDegrees,
        new VertexJoinFunction<Double, LongValue>() {
          public Double vertexJoin(Double vertexValue, LongValue
inputValue) {
            return vertexValue / inputValue.getValue();
          }
        });
```

In Scala:

```
val network: Graph[Long, Double, Double] = ...

val vertexOutDegrees: DataSet[(Long, LongValue)] = network.outDegrees
// assign the transition probabilities as the edge weights

val networkWithWeights = network.joinWithEdgesOnSource(vertexOutDegrees,
(v1: Double, v2: LongValue) => v1 / v2.getValue)
```

Reverse

The `reverse` method returns a graph with edge direction reverted.

The following code snippet shows how to use the same:

In Java:

```
Graph<Long, Double, Double> network = ...;
Graph<Long, Double, Double> networkReverse  = network.reverse();
```

In Scala:

```
val network: Graph[Long, Double, Double] = ...
val networkReversed: Graph[Long, Double, Double] = network.reverse
```

Undirected

The `undirected` method returns a new graph with additional edges opposite to the original ones.

The following code snippet shows how to use the same:

In Java:

```
Graph<Long, Double, Double> network = ...;
Graph<Long, Double, Double> networkUD  = network.undirected();
```

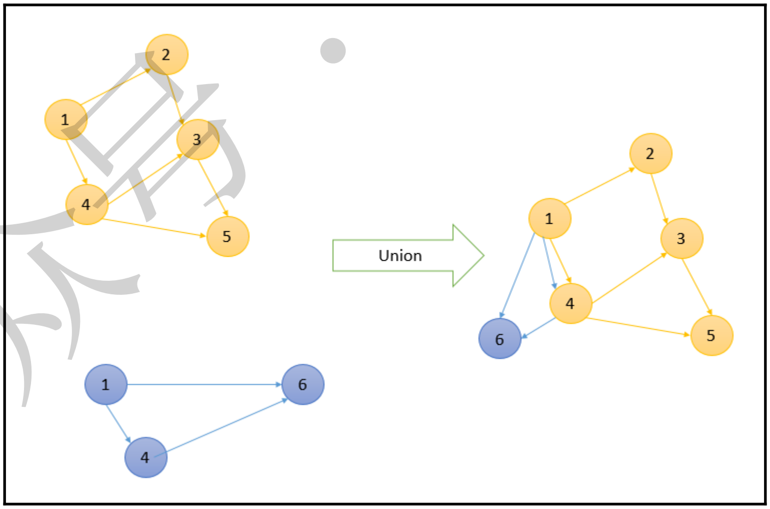
In Scala:

```
val network: Graph[Long, Double, Double] = ...
val networkUD: Graph[Long, Double, Double] = network.undirected
```

Union

The `union` operation returns a graph combining vertices and edges of two graphs. It joins the vertices on vertex IDs. Duplicate vertices are removed while edges are preserved.

The following is a graphical representation of the `union` operation:



Intersect

The `intersect` method performs the intersection of edges from given graph datasets. Two edges are considered equal if they have the same source and target vertices. The method also contains distinct parameter; if set to `true`, it only returns distinct graphs. The following are some code snippets show casing the `intersect` method usage.

In Java:

```
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

// create first graph from edges {(1, 2, 10) (1, 2, 11), (1, 2, 10)}
List<Edge<Long, Long>> edges1 = ...
Graph<Long, NullValue, Long> graph1 = Graph.fromCollection(edges1, env);

// create second graph from edges {(1, 2, 10)}
List<Edge<Long, Long>> edges2 = ...
Graph<Long, NullValue, Long> graph2 = Graph.fromCollection(edges2, env);

// Using distinct = true results in {(1,2,10)}
Graph<Long, NullValue, Long> intersect1 = graph1.intersect(graph2, true);

// Using distinct = false results in {(1,2,10),(1,2,10)} as there is one
edge pair
Graph<Long, NullValue, Long> intersect2 = graph1.intersect(graph2, false);
```

In Scala:

```
val env = ExecutionEnvironment.getExecutionEnvironment

// create first graph from edges {(1, 2, 10) (1, 2, 11), (1, 2, 10)}
val edges1: List[Edge[Long, Long]] = ...
val graph1 = Graph.fromCollection(edges1, env)

// create second graph from edges {(1, 2, 10)}
val edges2: List[Edge[Long, Long]] = ...
val graph2 = Graph.fromCollection(edges2, env)


// Using distinct = true results in {(1,2,10)}
val intersect1 = graph1.intersect(graph2, true)

// Using distinct = false results in {(1,2,10),(1,2,10)} as there is one
edge pair
val intersect2 = graph1.intersect(graph2, false)
```

Graph mutations

Gelly provides methods to add/remove edges and vertices to existing graphs. Let's try to understand these mutations one by one.

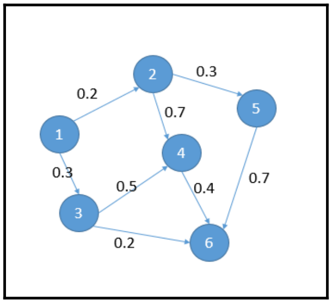
Mutation	In Java	In Scala
Add vertex.	`Graph<K, VV, EV> addVertex(final Vertex<K, VV> vertex)`	`addVertex(vertex: Vertex[K, VV])`
Add list of vertices.	`Graph<K, VV, EV> addVertices(List<Vertex<K, VV>> verticesToAdd)`	`addVertices(verticesToAdd: List[Vertex[K, VV]])`
Add edges to the graph. This adds new edges and vertices if they don't exist already.	`Graph<K, VV, EV> addEdge(Vertex<K, VV> source, Vertex<K, VV> target, EV edgeValue)`	`addEdge(source: Vertex[K, VV], target: Vertex[K, VV], edgeValue: EV)`
Add edges, if vertices do not exist then the edge is considered invalid.	`Graph<K, VV, EV> addEdges(List<Edge<K, EV>> newEdges)`	`addEdges(edges: List[Edge[K, EV]])`
Remove vertex, removes edges and vertices from the given graph.	`Graph<K, VV, EV> removeVertex(Vertex<K, VV> vertex)`	`removeVertex(vertex: Vertex[K, VV])`

Remove multiple vertices from the given graph.	Graph<K, VV, EV> removeVertices(List<Vertex<K, VV>> verticesToBeRemoved)	removeVertices(verticesToBeRemoved: List[Vertex[K, VV]])
Removes all edges that match the given edge.	Graph<K, VV, EV> removeEdge(Edge<K, EV> edge)	removeEdge(edge: Edge[K, EV])
Removes edges that match the given list of edges.	Graph<K, VV, EV> removeEdges(List<Edge<K, EV>> edgesToBeRemoved)	removeEdges(edgesToBeRemoved: List[Edge[K, EV]])

Neighborhood methods

Neighborhood methods help in performing operations related to its first hop neighborhood. Methods such as `reduceOnEdges()` and `reduceOnNeighbours()` can be used to perform aggregate operations. The first one is used to compute aggregation on neighboring edges of the vertex while the latter is used to compute aggregation on neighboring vertices. The neighbor scope can be defined by providing edge directions, and we have options such as `IN`, `OUT`, or `ALL`.

Consider an example where we need to get the maximum weight of all vertices for `OUT` direction edges:



Now we want to find out the maximum weighted `OUT` edge for each vertex. Gelly provides us with neighborhood methods with which we can find the desired result. The following is the code snippet for the same:

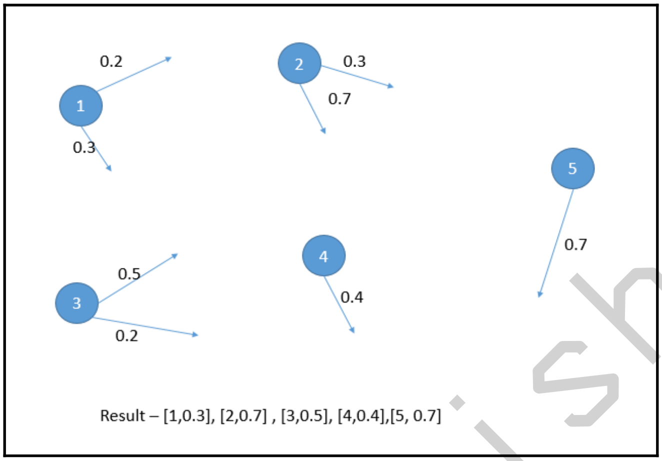
In Java:

```
Graph<Long, Long, Double> graph = ...

DataSet<Tuple2<Long, Double>> maxWeights = graph.reduceOnEdges(new
SelectMaxWeight(), EdgeDirection.OUT);

// user-defined function to select the max weight
static final class SelectMaxWeight implements ReduceEdgesFunction<Double> {

    @Override
    public Double reduceEdges(Double firstEdgeValue, Double
secondEdgeValue) {
      return Math.max(firstEdgeValue, secondEdgeValue);
    }
}
```

In Scala:

```
val graph: Graph[Long, Long, Double] = ...

val minWeights = graph.reduceOnEdges(new SelectMaxWeight,
EdgeDirection.OUT)

// user-defined function to select the max weight
final class SelectMaxWeight extends ReduceEdgesFunction[Double] {
  override def reduceEdges(firstEdgeValue: Double, secondEdgeValue:
Double): Double = {
    Math.max(firstEdgeValue, secondEdgeValue)
  }
 }
```

Gelly solves this by first segregating each vertex and finding out the maximum weighted edge for each vertex. The following is a graphical representation for the same:



Similarly, we can also write one more function to sum the values of incoming edges in all neighborhoods.

In Java:

```
Graph<Long, Long, Double> graph = ...

DataSet<Tuple2<Long, Long>> verticesWithSum = graph.reduceOnNeighbors(new
SumValues(), EdgeDirection.IN);

static final class SumValues implements ReduceNeighborsFunction<Long> {

        @Override
        public Long reduceNeighbors(Long firstNeighbor, Long
secondNeighbor) {
          return firstNeighbor + secondNeighbor;
      }
}
```

In Scala:

```
val graph: Graph[Long, Long, Double] = ...

val verticesWithSum = graph.reduceOnNeighbors(new SumValues,
EdgeDirection.IN)

final class SumValues extends ReduceNeighborsFunction[Long] {
     override def reduceNeighbors(firstNeighbor: Long, secondNeighbor:
```

```
Long): Long = {
      firstNeighbor + secondNeighbor
    }
}
```

Graph validation

Gelly provides us with a utility to validate the input graph before sending it for processing. In various situations, we first need to validate the graph with certain conditions and only then send it for further processing. Validations could be checking if the graph contains duplicate edges or checking if the graph structure is bipartite.

> Bipartite or Bigraph is a graph whose vertices can be divided into two distinct sets such that every vertex in each set has an edge connecting to vertex in another set. A simple example of bipartite graph is a graph of Basket Ball Players and the teams they play for. Here we will have two sets as players and teams and each vertex in player set would have edge to team set. More details on Bipartite graph can be read here `https://en.wikipedia.org/wiki/Bipartite_graph`.

We can also define custom validate methods to get the desired output. Gelly also provides a built-in validator called `InvalidVertexValidator`. This checks if the edge set contains validate vertex IDs. The following are some code snippets showcasing its usage.

In Java:

```
Graph<Long, Long, Long> graph = Graph.fromCollection(vertices, edges, env);

// Returns false for invalid vertex id.
graph.validate(new InvalidVertexIdsValidator<Long, Long, Long>());
```

In Scala:

```
val graph = Graph.fromCollection(vertices, edges, env)

// Returns false for invalid vertex id.
graph.validate(new InvalidVertexIdsValidator[Long, Long, Long])
```

Iterative graph processing

Gelly enhances Flink's iterative processing capabilities to support large scale graph processing. Currently it supports implementation of the following models:

- Vertex-Centric
- Scatter-Gather
- Gather-Sum-Apply

Let's start by understanding these models in the context of Gelly.

Vertex-Centric iterations

As the name suggest, these iterations are built thinking the vertex is in the center. Here each Vertex processes the same user-defined function in parallel. Each step of execution is called a **superset**. A vertex can send a message to another vertex as long as it knows its unique ID. This message would be used as input to the next superset.

To use Vertex-Centric iterations, the user needs to provide a `ComputeFunction`. We can also define an optional `MessageCombiner` to reduce the cost of communication. We can solve problems, such as Single Source Shortest Path in which we need to find the shortest path from source vertex to all other vertices.

> Single Source Shortest Path is where we try to minimize the sum of weights joining two distinct vertices. A very simple example could be a graph of cities and flight routes. In this case, SSSP algorithm will try to find the shortest distance connecting two cities considering the available flight routes. More details on SSSP can be found at `https://en.wikipedia.org/wiki/Shortest_path_problem`.

The following code snippets show how we solve the Single Source Shortest Path problem using Gelly.

In Java:

```
// maximum number of iterations
int maxIterations = 5;

// Run vertex-centric iteration
Graph<Long, Double, Double> result = graph.runVertexCentricIteration(
            new SSSPComputeFunction(), new SSSPCombiner(), maxIterations);

// Extract the vertices as the result
```

```
DataSet<Vertex<Long, Double>> singleSourceShortestPaths =
result.getVertices();


//User defined compute function to minimize the distance between //the
vertices

public static final class SSSPComputeFunction extends ComputeFunction<Long,
Double, Double, Double> {

public void compute(Vertex<Long, Double> vertex, MessageIterator<Double>
messages) {

    double minDistance = (vertex.getId().equals(srcId)) ? 0d :
Double.POSITIVE_INFINITY;

    for (Double msg : messages) {
        minDistance = Math.min(minDistance, msg);
    }

    if (minDistance < vertex.getValue()) {
        setNewVertexValue(minDistance);
        for (Edge<Long, Double> e: getEdges()) {
            sendMessageTo(e.getTarget(), minDistance + e.getValue());
        }
    }
}

// message combiner helps in optimizing the communications
public static final class SSSPCombiner extends MessageCombiner<Long,
Double> {

    public void combineMessages(MessageIterator<Double> messages) {

        double minMessage = Double.POSITIVE_INFINITY;
        for (Double msg: messages) {
           minMessage = Math.min(minMessage, msg);
        }
        sendCombinedMessage(minMessage);
    }
}
```

In Scala:

```
// maximum number of iterations
val maxIterations = 5

// Run the vertex-centric iteration
```

```
val result = graph.runVertexCentricIteration(new SSSPComputeFunction, new
SSSPCombiner, maxIterations)

// Extract the vertices as the result
val singleSourceShortestPaths = result.getVertices


//User defined compute function to minimize the distance between //the
vertices

final class SSSPComputeFunction extends ComputeFunction[Long, Double,
Double, Double] {

    override def compute(vertex: Vertex[Long, Double], messages:
    MessageIterator[Double]) = {

    var minDistance = if (vertex.getId.equals(srcId)) 0 else
    Double.MaxValue

    while (messages.hasNext) {
        val msg = messages.next
        if (msg < minDistance) {
            minDistance = msg
        }
    }

    if (vertex.getValue > minDistance) {
        setNewVertexValue(minDistance)
        for (edge: Edge[Long, Double] <- getEdges) {
            sendMessageTo(edge.getTarget, vertex.getValue +
            edge.getValue)
        }
    }
}

// message combiner helps in optimizing the communications
final class SSSPCombiner extends MessageCombiner[Long, Double] {

    override def combineMessages(messages: MessageIterator[Double]) {

        var minDistance = Double.MaxValue

        while (messages.hasNext) {
          val msg = inMessages.next
          if (msg < minDistance) {
            minDistance = msg
          }
        }
```

```
            sendCombinedMessage(minMessage)
        }
    }
```

We can use following configurations in Vertex-Centric iterations.

Parameter	**Description**
Name: `setName()`	Sets the name of Vertex-Centric iteration. Can be seen in logs.
Parallelism: `setParallelism()`	Sets the parallelism for parallel execution.
Broadcast variables: `addBroadcastSet()`	Adds broadcast variables to the computation function.
Aggregator: `registerAggregator()`	Registers custom defined aggregator function to be used by the computation function.
Solution set in unmanaged memory: `setSolutionSetUnmanagedMemory()`	Defines whether the solution set is kept in managed memory.

Scatter-Gather iterations

Scatter-Gather iterations also works in superset iterations and also have a vertex at its center and we also define a function that are executed in parallel. Here each vertex has two important things to do:

- **Scatter**: Scatter produces the message it needs to send to other vertices
- **Gather**: Gather updates the vertex value from the messages it received

Gelly provides methods for scatter and gather. The user needs to implement only these two functions to make use of these iterations. `ScatterFunction` produces messages for the rest of the vertices while `GatherFunction` computes the updated value for the vertex based on the messages it has received.

The following code snippet shows how we can solve the Single Source Shortest Path problem using Gelly-Scatter-Gather iterations:

In Java:

```
// maximum number of iterations
int maxIterations = 5;

// Run the scatter-gather iteration
Graph<Long, Double, Double> result = graph.runScatterGatherIteration(
      new MinDistanceMessenger(), new VertexDistanceUpdater(),
maxIterations);

// Extract the vertices as the result
DataSet<Vertex<Long, Double>> singleSourceShortestPaths =
result.getVertices();


// Scatter Gather function definition

// Through scatter function, we send distances from each vertex
public static final class MinDistanceMessenger extends
ScatterFunction<Long, Double, Double, Double> {

  public void sendMessages(Vertex<Long, Double> vertex) {
    for (Edge<Long, Double> edge : getEdges()) {
      sendMessageTo(edge.getTarget(), vertex.getValue() + edge.getValue());
    }
  }
}

// In gather function, we gather messages sent in previous //superstep to
find out the minimum distance.
public static final class VertexDistanceUpdater extends
GatherFunction<Long, Double, Double> {

  public void updateVertex(Vertex<Long, Double> vertex,
MessageIterator<Double> inMessages) {
    Double minDistance = Double.MAX_VALUE;

    for (double msg : inMessages) {
      if (msg < minDistance) {
        minDistance = msg;
      }
    }

    if (vertex.getValue() > minDistance) {
      setNewVertexValue(minDistance);
    }
  }
}
```

In Scala:

```
// maximum number of iterations
val maxIterations = 5

// Run the scatter-gather iteration
val result = graph.runScatterGatherIteration(new MinDistanceMessenger, new
VertexDistanceUpdater, maxIterations)

// Extract the vertices as the result
val singleSourceShortestPaths = result.getVertices


// Scatter Gather definitions

// Through scatter function, we send distances from each vertex
final class MinDistanceMessenger extends ScatterFunction[Long, Double,
Double, Double] {

  override def sendMessages(vertex: Vertex[Long, Double]) = {
    for (edge: Edge[Long, Double] <- getEdges) {
      sendMessageTo(edge.getTarget, vertex.getValue + edge.getValue)
    }
  }
}

// In gather function, we gather messages sent in previous //superstep to
find out the minimum distance.
final class VertexDistanceUpdater extends GatherFunction[Long, Double,
Double] {

  override def updateVertex(vertex: Vertex[Long, Double], inMessages:
MessageIterator[Double]) = {
    var minDistance = Double.MaxValue

    while (inMessages.hasNext) {
      val msg = inMessages.next
      if (msg < minDistance) {
      minDistance = msg
      }
    }

    if (vertex.getValue > minDistance) {
      setNewVertexValue(minDistance)
```

```
        }
      }
    }
```

We can configure Scatter-Gather iterations using the following parameters:

Parameter	Description
Name: `setName()`	Sets the name of scatter-gather iteration. Can be seen in logs.
Parallelism: `setParallelism()`	Sets the parallelism for parallel execution.
Broadcast variables: `addBroadcastSet()`	Adds broadcast variables to the computation function.
Aggregator: `registerAggregator()`	Registers the custom defined aggregator function to be used by the computation function.
Solution set in unmanaged memory: `setSolutionSetUnmanagedMemory()`	Defines whether the solution set is kept in managed memory.
Number of vertices: `setOptNumVertices()`	Accesses the total no. of vertices in an iteration.
Degrees: `setOptDegrees()`	Sets the number of in/out degrees to be reached within an iteration.
Messaging directions: `setDirection()`	By default we only consider out degrees for processing but we can change that by setting this property. Options are `in`, `out`, and `all`.

Gather-Sum-Apply iterations

Like the previous two models, **Gather-Sum-Apply** (**GSA**) iterations are also synchronized in iterative steps. Each superset consists of the following steps:

1. **Gather**: A user defined function executed on edges and each neighbor, producing a partial value.
2. **Sum**: Partial values calculated in earlier step would be aggregated in this step.
3. **Apply**: Each vertex value is updated by applying the function on the aggregated value from the previous step and the current value.

We will try to solve the Single Source Shortest Path using the GSA iteration. To use this, we need to define custom functions for gather, sum, and apply.

In Java:

```
// maximum number of iterations
int maxIterations = 5;

// Run the GSA iteration
Graph<Long, Double, Double> result = graph.runGatherSumApplyIteration(
        new CalculateDistances(), new ChooseMinDistance(), new
UpdateDistance(), maxIterations);

// Extract the vertices as the result
DataSet<Vertex<Long, Double>> singleSourceShortestPaths =
result.getVertices();


// Functions for GSA

// Gather
private static final class CalculateDistances extends
GatherFunction<Double, Double, Double> {

  public Double gather(Neighbor<Double, Double> neighbor) {
    return neighbor.getNeighborValue() + neighbor.getEdgeValue();
  }
}

// Sum
private static final class ChooseMinDistance extends SumFunction<Double,
Double, Double> {

  public Double sum(Double newValue, Double currentValue) {
    return Math.min(newValue, currentValue);
  }
}

// Apply
private static final class UpdateDistance extends ApplyFunction<Long,
Double, Double> {

  public void apply(Double newDistance, Double oldDistance) {
    if (newDistance < oldDistance) {
      setResult(newDistance);
    }
  }
}
```

In Scala:

```
// maximum number of iterations
val maxIterations = 10

// Run the GSA iteration
val result = graph.runGatherSumApplyIteration(new CalculateDistances, new
ChooseMinDistance, new UpdateDistance, maxIterations)

// Extract the vertices as the result
val singleSourceShortestPaths = result.getVertices


// Custom function for GSA

// Gather
final class CalculateDistances extends GatherFunction[Double, Double,
Double] {

  override def gather(neighbor: Neighbor[Double, Double]): Double = {
    neighbor.getNeighborValue + neighbor.getEdgeValue
  }
}

// Sum
final class ChooseMinDistance extends SumFunction[Double, Double, Double] {

  override def sum(newValue: Double, currentValue: Double): Double = {
    Math.min(newValue, currentValue)
  }
}

// Apply
final class UpdateDistance extends ApplyFunction[Long, Double, Double] {

  override def apply(newDistance: Double, oldDistance: Double) = {
    if (newDistance < oldDistance) {
      setResult(newDistance)
    }
  }
}
```

We can configure GSA iterations using the following parameters:

Parameter	**Description**
Name: `setName()`	Sets the name of the GSA iteration. Can be seen in logs.
Parallelism: `setParallelism()`	Sets the parallelism for parallel execution.
Broadcast variables: `addBroadcastSet()`	Adds broadcast variables to the computation function.
Aggregator: `registerAggregator()`	Registers the custom defined aggregator function to be used by the computation function.
Solution set in unmanaged memory: `setSolutionSetUnmanagedMemory()`	Defines whether the solution set is kept in managed memory.
Number of vertices: `setOptNumVertices()`	Accesses the total number of vertices in an iteration.
Neighbour directions: `setDirection()`	By default we only consider `OUT` degrees for processing but we can change that by setting this property. Options are `IN`, `OUT`, and `ALL`.

Use case – Airport Travel Optimization

Let's consider a use case where we have data for the airports and the distance between them. In order to travel to certain destination from a particular airport, we have to find the shortest path between the two. Our airport data looks like as shown in the following table:

Id	**Airport name**
s01	A
s02	B
s03	C
s04	D
s05	E

The distance information between the airport looks like as shown in the following table:

From	To	Distance
s01	s02	10
s01	s02	12
s01	s03	22
s01	s04	21
s04	s11	22
s05	s15	21
s06	s17	21
s08	s09	11
s08	s09	12

Now let's use Gelly to find the Single Source Shortest Path.

Here we have options to choose among the three algorithms we learnt in previous section. In this example, we will use Vertex-Centric iterations method.

In order to solve the Single Source Shortest Path, we have to first load the data from CSV files as shown in the following code:

```
// set up the batch execution environment
final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();

// Create graph by reading from CSV files
DataSet<Tuple2<String, Double>> airportVertices = env
            .readCsvFile("nodes.csv").types(String.class, Double.class);

DataSet<Tuple3<String, String, Double>> airportEdges = env
            .readCsvFile("edges.csv")
            .types(String.class, String.class, Double.class);

Graph<String, Double, Double> graph =
Graph.fromTupleDataSet(airportVertices, airportEdges, env);
```

Next we run Vertex-Centric iteration as discussed in the previous section on the graph we created:

```
// define the maximum number of iterations
int maxIterations = 10;

// Execute the vertex-centric iteration
Graph<String, Double, Double> result = graph.runVertexCentricIteration(new
SSSPComputeFunction(), new SSSPCombiner(), maxIterations);

// Extract the vertices as the result
DataSet<Vertex<String, Double>> singleSourceShortestPaths =
result.getVertices();
singleSourceShortestPaths.print();
```

The implementation of Compute function and Combiner is similar to what we looked in the earlier section. When we run this code, we will get the answer for SSSP from given source vertex.

The complete code and sample data for this use case is available at `https://github.com/deshpandetanmay/mastering-flink/tree/master/chapter07/flink-gelly`

In general, all three iterations ways look similar but they have minute differences. One needs to really think which algorithm to use based on use case. Here is some good reading about the this thought `https://ci.apache.org/projects/flink/flink-docs-release-1.1/apis/batch/libs/gelly.html#iteration-abstractions-comparison`.

Summary

In this chapter, we explored various aspects of the graph processing API provided by the Flink Gelly library. We learnt how to define graphs, load data, and process it. We also looked at various transformations one can do on a graph. Finally we learnt details of iterative graph processing options that Gelly provides.

In the next chapter, we will see how to execute Flink applications on Hadoop and YARN.

8

Distributed Data Processing with Flink and Hadoop

Apache Hadoop has become a core and essential part of data processing and analytics infrastructures over the last couple of years. With Hadoop 1.X, the community learnt the distributed data processing using the MapReduce framework, whereas the next version of Hadoop, 2.X taught us the efficient use of resources and scheduling using the YARN framework. The YARN framework is a core part of Hadoop data processing, where it handles complex tasks such as job executions, distribution, resource allocation, scheduling, and so on. It allows for multi-tenancy, scalability, and high availability.

The best part about YARN is that it is not just a framework but more like a complete operating system where developers are free to develop and execute applications of their choice. It gives abstraction by letting developers only focus on application development and forget the pain of data and execution distribution in parallel. YARN sits on top of the Hadoop Distributed File System and can also read data from filesystems such as AWS S3.

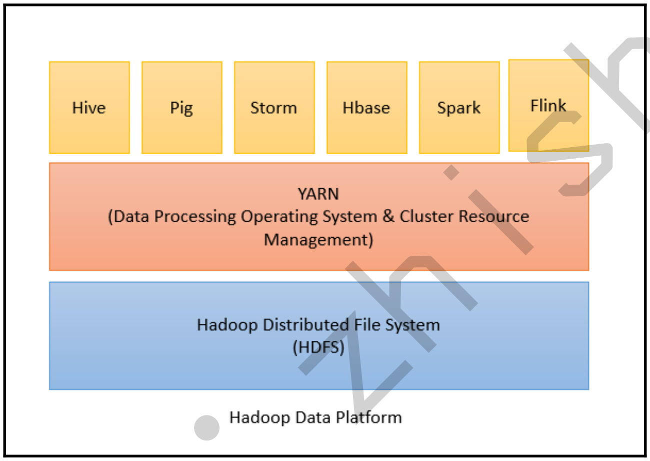
The YARN application framework has been built so well that it can host any distributed processing engine. In recent times, there has been a significant rise in new distributed data processing engines such as Spark, Flink, and so on. As they are built to be executed on a YARN cluster, it becomes very easy for people to try new things in parallel on the same YARN cluster. This means we can run Spark as well as Flink jobs on the same cluster using YARN. In this chapter, we are going to see how we can make use of existing Hadoop/YARN clusters to execute our Flink job in parallel.

So let's get started.

Quick overview of Hadoop

Most of you will be already aware of Hadoop and what it does but for those who are new to the world of distributed computing, let me try to give a brief introduction to Hadoop.

Hadoop is a distributed, open source data processing framework. It consists of two important parts: one data storage unit, **Hadoop Distributed File System** (**HDFS**) and the resource management unit, **Yet Another Resource Negotiator** (**YARN**). The following diagram shows a high-level overview of the Hadoop ecosystem:


Hive
Pig
Storm
Hbase
Spark
Flink
YARN
(Data Processing Operating System & Cluster Resource Management)
Hadoop Distributed File System
(HDFS)
Hadoop Data Platform

HDFS

HDFS, as the name suggests, is a highly available, distributed filesystem used for data storage. These days, this is one of the core frameworks of most companies. HDFS consists of a master-slave architecture, with daemons such as NameNode, secondary NameNode, and DataNode.

In HDFS, NameNode stores metadata about the files to be stored while DataNode stores the actual block comprising a file. Data blocks are by default three-fold replicated in order to achieve high availability. A secondary NameNode is used for backing up the filesystem metadata stored on NameNode.

Here is a link where you can read more about HDFS at `http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html`.

YARN

Prior to YARN, MapReduce was the data processing framework which ran on top of HDFS. But people started realizing its limitation of number of task trackers a job tracker can handle. This gave rise to YARN. The fundamental idea behind YARN is to separate the resource management and scheduling tasks. YARN has the global resource manager and per application–the application master. The resource manager works on master nodes whereas it has a per worker node agent–the node manager, which is responsible for managing containers, monitoring their usage (CPU, disk, memory) and reporting back to the resource manager.

The resource manager has two important components–**scheduler** and **applications manager**. Scheduler is responsible for scheduling applications in the queue while applications manager takes care of accepting job submissions, negotiating first container to the application specific application master. It is also responsible for restarting the **application master** in case of failure.

Because of operating systems like nature, YARN provides API which can be extended to build applications. **Spark** and **Flink** are great examples of this.

Here is a link where you can read more about YARN at `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`.

Now let's look at how we can use Flink on YARN.

Flink on YARN

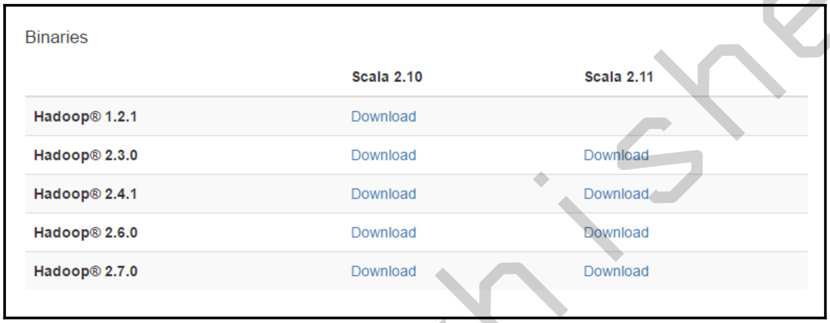
Flink has been built-in supported to be execution-ready on YARN. Any application build using Flink APIs can be executed on YARN without much effort. Users don't need to set up or install anything if they already have a YARN cluster. Flink expects the following requirements to be met:

- Hadoop version should be 2.2 or above
- HDFS should be up-and-running

Configurations

In order to run Flink on YARN, the following configurations needs to be done. First of all, we need to download the Hadoop compatible version of the Flink distribution.

The binaries are available for download at `http://flink.apache.org/downloads.html`. You have to choose from the following options.



Binaries

	Scala 2.10	Scala 2.11
Hadoop® 1.2.1	Download	
Hadoop® 2.3.0	Download	Download
Hadoop® 2.4.1	Download	Download
Hadoop® 2.6.0	Download	Download
Hadoop® 2.7.0	Download	Download

Let's assume we are running Hadoop 2.7 and Scala 2.11. We will download the specific binary and store it on a node where Hadoop is installed and running.

Once downloaded, we need to extract the `tar` files as shown here:

```
$tar -xzf flink-1.1.4-bin-hadoop27-scala_2.11.tgz
$cd flink-1.1.4
```

Starting a Flink YARN session

Once the binaries are extracted, we can start the Flink session. A Flink session is a session which starts all required Flink services (Job Manager and Task Managers) on respective nodes so that we can start executing Flink jobs. To start the Flink session, we have the following executable with the given options:

```
# bin/yarn-session.sh
Usage:
   Required
     -n,--container <arg>            Number of YARN container to
                                     allocate (=Number of Task
                                     Managers)
   Optional
```

```
     -D <arg>                        Dynamic properties
     -d,--detached                   Start detached
     -id,--applicationId <arg>       Attach to running YARN session
     -j,--jar <arg>                  Path to Flink jar file
     -jm,--jobManagerMemory <arg>    Memory for JobManager
                                     Container [in MB]
     -n,--container <arg>            Number of YARN container to
                                     allocate (=Number of Task
                                     Managers)
     -nm,--name <arg>                Set a custom name for the
                                     application on YARN
     -q,--query                      Display available YARN
                                     resources (memory, cores)
     -qu,--queue <arg>               Specify YARN queue.
     -s,--slots <arg>                Number of slots per
                                     TaskManager
     -st,--streaming                 Start Flink in streaming mode
     -t,--ship <arg>                 Ship files in the specified
                                     directory (t for transfer)
     -tm,--taskManagerMemory <arg>   Memory per TaskManager
                                     Container [in MB]
     -z,--zookeeperNamespace <arg>   Namespace to create the
                                     Zookeeper sub-paths for high
                                     availability mode
```

We have to make sure that the `YARN_CONF_DIR` and `HADOOP_CONF_DIR` environment variables are set so that Flink can find the required configurations. Now let's start the Flink session by providing information.

Here is how we start the Flink session by giving the details about the number of task managers, memory for each task manager, and slots to be used:

```
# bin/yarn-session.sh -n 2 -tm 1024 -s 10
2016-11-14 10:46:00,126 WARN
org.apache.hadoop.util.NativeCodeLoader
- Unable to load native-hadoop library for your platform... using
builtin-java classes where applicable
2016-11-14 10:46:00,184 INFO
org.apache.flink.yarn.YarnClusterDescriptor
- The configuration directory ('/usr/local/flink/flink-1.1.3/conf')
contains both LOG4J and Logback configuration files. Please delete
or rename one of them.
2016-11-14 10:46:01,263 INFO  org.apache.flink.yarn.Utils
- Copying from file:/usr/local/flink/flink-
1.1.3/conf/log4j.properties to
hdfs://hdpcluster/user/root/.flink/application_1478079131011_0107/
log4j.properties
2016-11-14 10:46:01,463 INFO  org.apache.flink.yarn.Utils
```

```
- Copying from file:/usr/local/flink/flink-1.1.3/lib to
hdfs://hdp/user/root/.flink/application_1478079131011_0107/lib
2016-11-14 10:46:02,337 INFO  org.apache.flink.yarn.Utils
- Copying from file:/usr/local/flink/flink-1.1.3/conf/logback.xml
to hdfs://hdpcluster/user/root/.flink/
application_1478079131011_0107/logback.xml
2016-11-14 10:46:02,350 INFO  org.apache.flink.yarn.Utils
- Copying from file:/usr/local/flink/flink-1.1.3/lib/flink-
dist_2.11-1.1.3.jar to hdfs://hdpcluster/user/root/.flink/
application_1478079131011_0107/flink-dist_2.11-1.1.3.jar
2016-11-14 10:46:03,157 INFO  org.apache.flink.yarn.Utils
- Copying from /usr/local/flink/flink-1.1.3/conf/flink-conf.yaml to
hdfs://hdpcluster/user/root/.flink/application_1478079131011_0107/
flink-conf.yaml
org.apache.flink.yarn.YarnClusterDescriptor
- Deploying cluster, current state ACCEPTED
2016-11-14 10:46:11,976 INFO
org.apache.flink.yarn.YarnClusterDescriptor
- YARN application has been deployed successfully.
Flink JobManager is now running on 10.22.3.44:43810
JobManager Web Interface:
http://myhost.com:8088/proxy/application_1478079131011_0107/
2016-11-14 10:46:12,387 INFO  Remoting
- Starting remoting
2016-11-14 10:46:12,483 INFO  Remoting
- Remoting started; listening on addresses :
[akka.tcp://flink@10.22.3.44:58538]
2016-11-14 10:46:12,627 INFO
org.apache.flink.yarn.YarnClusterClient
- Start application client.
2016-11-14 10:46:12,634 INFO
org.apache.flink.yarn.ApplicationClient
- Notification about new leader address
akka.tcp://flink@10.22.3.44:43810/user/jobmanager with session ID
null.
2016-11-14 10:46:12,637 INFO
org.apache.flink.yarn.ApplicationClient
- Received address of new leader
akka.tcp://flink@10.22.3.44:43810/user/jobmanager
with session ID null.
2016-11-14 10:46:12,638 INFO
org.apache.flink.yarn.ApplicationClient
- Disconnect from JobManager null.
2016-11-14 10:46:12,640 INFO
org.apache.flink.yarn.ApplicationClient
- Trying to register at JobManager
akka.tcp://flink@10.22.3.44:43810/user/jobmanager.
2016-11-14 10:46:12,649 INFO
```

```
org.apache.flink.yarn.ApplicationClient
- Successfully registered at the ResourceManager using JobManager
Actor[akka.tcp://flink@10.22.3.44:43810/user/jobmanager#-862361447]
```

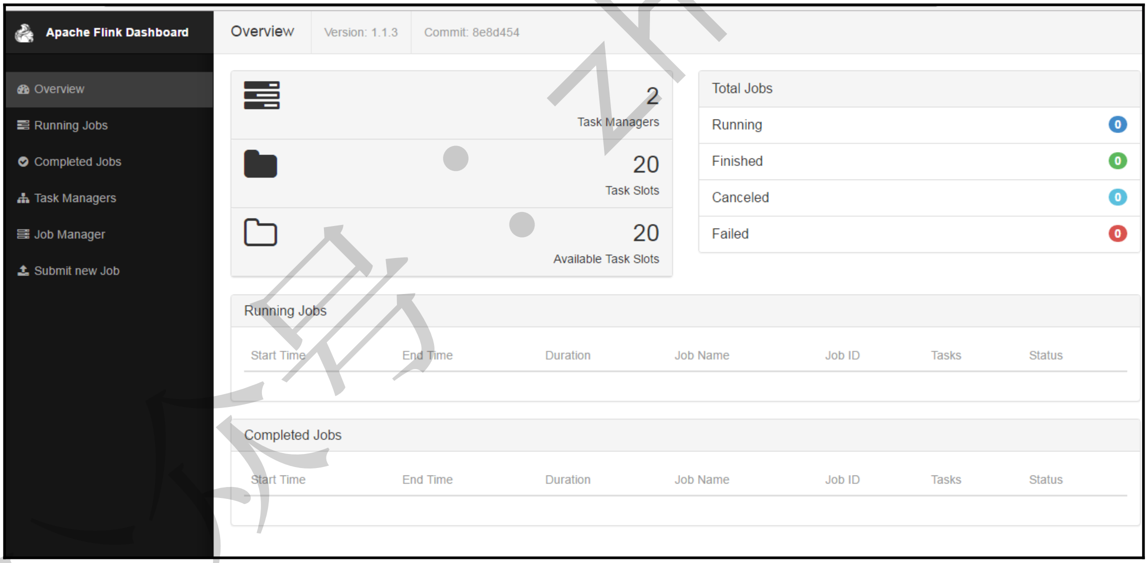
If the configuration directories are not set properly, you will get an error mentioning the same. In that case, first you can set the configuration directories and then start the Flink YARN session.

The following command sets the configuration directories:

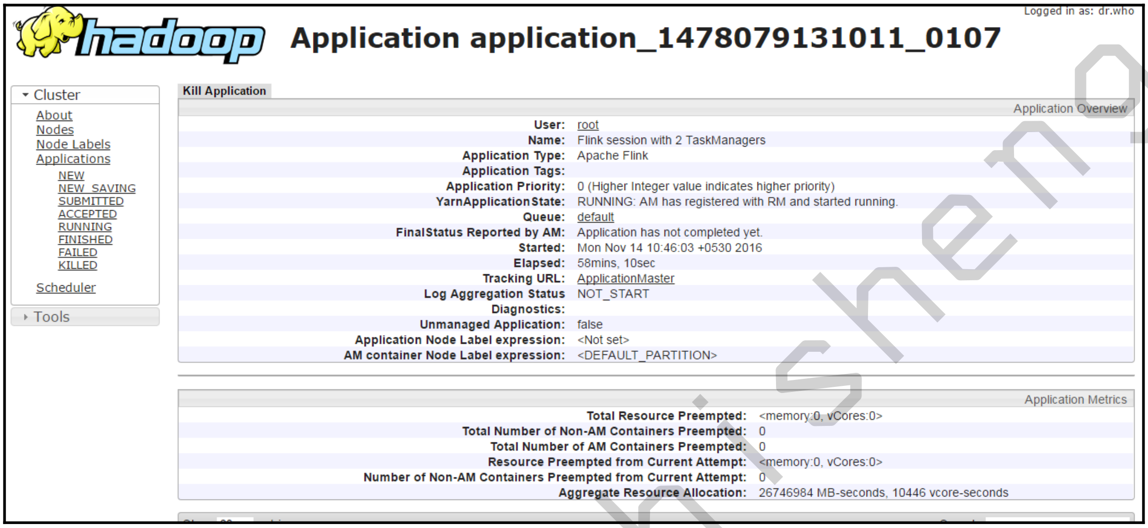
```
export HADOOP_CONF_DIR=/etc/hadoop/conf
export YARN_CONF_DIR=/etc/hadoop/conf
```

We can also check the Flink Web UI as by hitting the following URL: `http://host:8088/proxy/application_<id>/#/overview`.

Here is the screenshot of the same:



Similarly, we can also check the YARN application UI at `http://myhost:8088/cluster/app/application_1478079131011_0107`.



Logged in as: dr.who

Application application_1478079131011_0107

Application Overview

User: root
Name: Flink session with 2 TaskManagers
Application Type: Apache Flink
Application Tags:
Application Priority: 0 (Higher Integer value indicates higher priority)
YarnApplicationState: RUNNING: AM has registered with RM and started running.
Queue: default
FinalStatus Reported by AM: Application has not completed yet.
Started: Mon Nov 14 10:46:03 +0530 2016
Elapsed: 58mins, 10sec
Tracking URL: ApplicationMaster
Log Aggregation Status NOT_START
Diagnostics:
Unmanaged Application: false
Application Node Label expression: <Not set>
AM container Node Label expression: <DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted: <memory:0, vCores:0>
Total Number of Non-AM Containers Preempted: 0
Total Number of AM Containers Preempted: 0
Resource Preempted from Current Attempt: <memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt: 0
Aggregate Resource Allocation: 26746984 MB-seconds, 10446 vcore-seconds

Submitting a job to Flink

Now that we have a Flink session connected to YARN, we are all set to submit a Flink job to YARN.

We can use the following command with options to submit the Flink job:

```
#./bin/flink
./flink <ACTION> [OPTIONS] [ARGUMENTS]
```

We can execute the Flink job using the run action. We have the following options in run:

Option	Description
`-c`, `--class <classname>`	Class with the program entry point (`main()` method or `getPlan()` method). Only needed if the JAR file does not specify the class in its manifest.
`-C`, `--classpath <url>`	Adds a URL to each user code classloader on all nodes in the cluster. The paths must specify a protocol (for example, `file://`) and be accessible on all nodes (for example, by means of an NFS share). You can use this option multiple times for specifying more than one URL. The protocol must be supported by `{@link java.net.URLClassLoader}`. This can be used in case you wish to use certain third-party libraries with Flink YARN session.
`-d`, `--detached`	If present, runs the job in detached mode. Detached mode can be useful when you don't want to keep running the Flink YARN session all the time. In this case, Flink client will only submit the job and will detach itself. We cannot stop detached Flink YARN session using Flink commands. To do so, we have to kill the application using YARN commands `yarn application -kill <appId>`

`-m, --jobmanager <host:port>`	Address of the Job Manager (master) in which to connect. Use this flag to connect to a different Job Manager than the one specified in the configuration.
`-p, --parallelism <parallelism>`	The parallelism with which to run the program. Optional flag to override the default value specified in the configuration.
`-q, --sysoutLogging`	If present, suppresses logging output to standard `OUT`.
`-s, --fromSavepoint <savepointPath>`	Path to a save point to reset the job back to, for example `file:///flink/savepoint-1537`. Savepoints are externally stored states of a Flink program. They are snapshots which are stored to a certain location. If Flink program fails, we can resume it from its last stored save point. More details on save points `https://ci.apache.org/projects/flink/flink-docs-release-1.2/setup/savepoints.html`
`-z, --zookeeperNamespace <zookeeperNamespace>`	Namespace to create the Zookeeper sub-paths for high availability mode

The following options are available for the `yarn-cluster` mode:

Option	**Description**
`-yD <arg>`	Dynamic properties
`yd, --yarndetached`	Start detached
`-yid, --yarnapplicationId <arg>`	Attach to running YARN session
`-yj, --yarnjar <arg>`	Path to Flink jar file
`-yjm, --yarnjobManagerMemory <arg>`	Memory for Job Manager container (in MB)
`-yn, --yarncontainer <arg>`	Number of YARN containers to allocate (= number of task managers)
`-ynm, --yarnname <arg>`	Sets a custom name for the application on YARN
`-yq, --yarnquery`	Displays available YARN resources (memory, cores)
`-yqu, --yarnqueue <arg>`	Specifies YARN queue
`-ys, --yarnslots <arg>`	Number of slots per Task Manager
`-yst, --yarnstreaming`	Starts Flink in streaming mode
`-yt, --yarnship <arg>`	Ships files in the specified director (t for transfer)
`-ytm, --yarntaskManagerMemory <arg>`	Memory per TaskManager ontainer (in MB)
`-yz, --yarnzookeeperNamespace <arg>`	Namespace to create the Zookeeper sub-paths for high availability mode

Now let's try to run a sample word count example on YARN. The following are the steps to do so.

First let's have input file stored on HDFS as input to the word count program. Here we are going to run the word count on Apache License text. The following is the way we download and store it on HDFS:

```
wget -O LICENSE-2.0.txt http://www.apache.org/licenses/LICENSE-
2.0.txt
hadoop fs -mkdir in
hadoop fs -put LICENSE-2.0.txt in
```

Now we will submit the example word count job:

```
./bin/flink run ./examples/batch/WordCount.jar
hdfs://myhost/user/root/in  hdfs://myhost/user/root/out
```

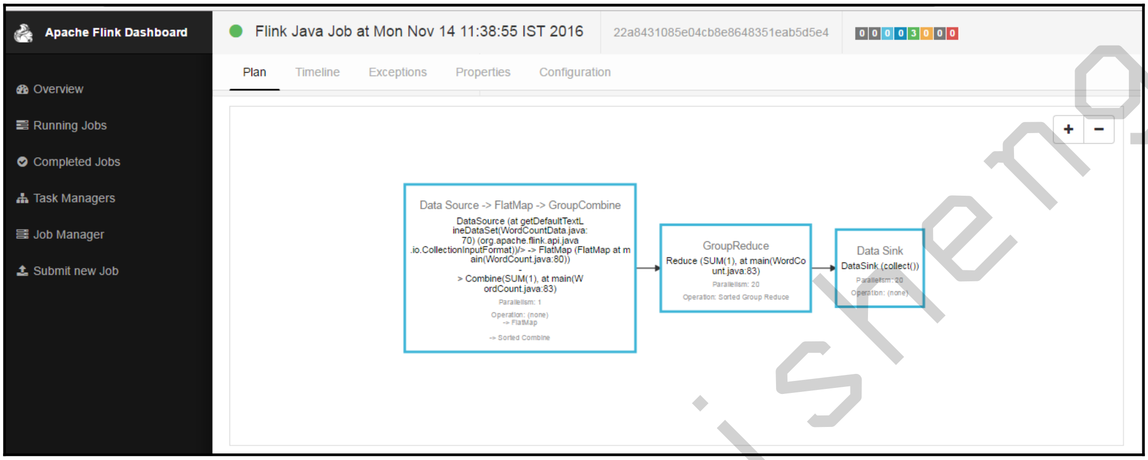
This will invoke the Flink job which would get executed on the YARN cluster. You should see the console as:

```
  # ./bin/flink run ./examples/batch/WordCount.jar
2016-11-14 11:26:32,603 INFO
org.apache.flink.yarn.cli.FlinkYarnSessionCli
- YARN properties set default parallelism to 20
2016-11-14 11:26:32,603 INFO
org.apache.flink.yarn.cli.FlinkYarnSessionCli
- YARN properties set default parallelism to 20
YARN properties set default parallelism to 20
2016-11-14 11:26:32,603 INFO
org.apache.flink.yarn.cli.FlinkYarnSessionCli
- Found YARN properties file /tmp/.yarn-properties-root
2016-11-14 11:26:32,603 INFO
org.apache.flink.yarn.cli.FlinkYarnSessionCli
- Found YARN properties file /tmp/.yarn-properties-root
Found YARN properties file /tmp/.yarn-properties-root
2016-11-14 11:26:32,603 INFO
org.apache.flink.yarn.cli.FlinkYarnSessionCli
- Using Yarn application id from YARN properties
application_1478079131011_0107
2016-11-14 11:26:32,603 INFO
org.apache.flink.yarn.cli.FlinkYarnSessionCli
- Using Yarn application id from YARN properties
application_1478079131011_0107
Using Yarn application id from YARN properties
application_1478079131011_0107
2016-11-14 11:26:32,604 INFO
org.apache.flink.yarn.cli.FlinkYarnSessionCli
- YARN properties set default parallelism to 20
```

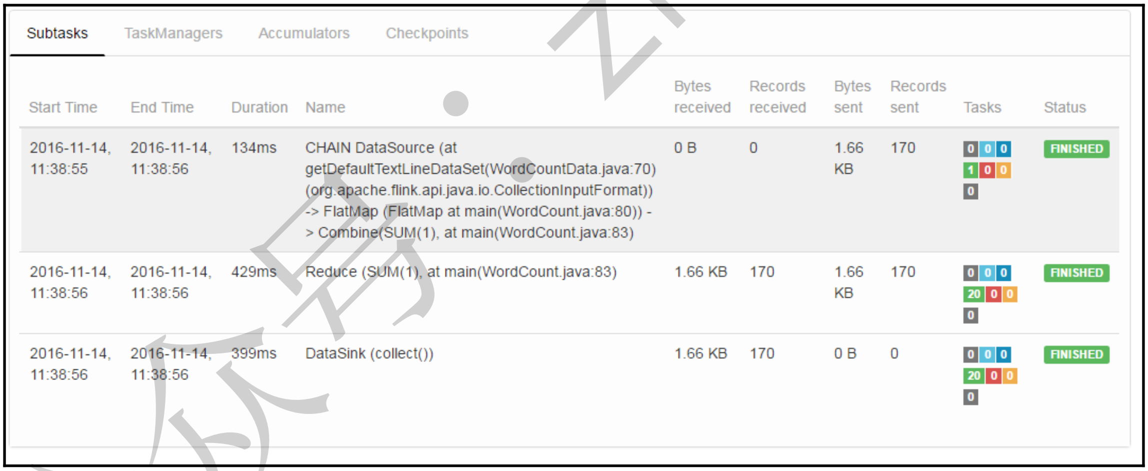
```
2016-11-14 11:26:32,604 INFO
org.apache.flink.yarn.cli.FlinkYarnSessionCli
- YARN properties set default parallelism to 20
YARN properties set default parallelism to 20
2016-11-14 11:26:32,823 INFO
org.apache.hadoop.yarn.client.api.impl.TimelineClientImpl
- Timeline service address: http://hdpdev002.pune-
in0145.slb.com:8188/ws/v1/timeline/
2016-11-14 11:26:33,089 INFO
org.apache.flink.yarn.YarnClusterDescriptor
- Found application JobManager host name myhost.com' and port
'43810' from supplied application id
'application_1478079131011_0107'
Cluster configuration: Yarn cluster with application id
application_1478079131011_0107
Using address 163.183.206.249:43810 to connect to JobManager.
Starting execution of program
2016-11-14 11:26:33,711 INFO
org.apache.flink.yarn.YarnClusterClient
- TaskManager status (2/1)
TaskManager status (2/1)
2016-11-14 11:26:33,712 INFO
org.apache.flink.yarn.YarnClusterClient
- All TaskManagers are connected
All TaskManagers are connected
2016-11-14 11:26:33,712 INFO
org.apache.flink.yarn.YarnClusterClient
- Submitting job with JobID: b57d682dd09f570ea336b0d56da16c73.
Waiting for job completion.
Submitting job with JobID: b57d682dd09f570ea336b0d56da16c73.
Waiting for job completion.
Connected to JobManager at
Actor[akka.tcp://flink@163.183.206.249:43810/user/
jobmanager#-862361447]
11/14/2016 11:26:33     Job execution switched to status RUNNING.
11/14/2016 11:26:33     CHAIN DataSource (at
getDefaultTextLineDataSet(WordCountData.java:70)
(org.apache.flink.api.java.io.CollectionInputFormat)) -> FlatMap
(FlatMap at main(WordCount.java:80)) -> Combine(SUM(1), at
main(WordCount.java:83)(1/1) switched to RUNNING
11/14/2016 11:26:34     DataSink (collect())(20/20) switched to
FINISHED
...
11/14/2016 11:26:34     Job execution switched to status FINISHED.
(after,1)
(coil,1)
(country,1)
(great,1)
```

```
(long,1)
(merit,1)
(oppressor,1)
(pangs,1)
(scorns,1)
(what,1)
(a,5)
(death,2)
(die,2)
(rather,1)
(be,4)
(bourn,1)
(d,4)
(say,1)
(takes,1)
(thy,1)
(himself,1)
(sins,1)
(there,2)
(whips,1)
(would,2)
(wrong,1)
...
  Program execution finished
  Job with JobID b57d682dd09f570ea336b0d56da16c73 has finished.
  Job Runtime: 575 ms
  Accumulator Results:
  - 4950e35c195be901e0ad6a8ed25790de (java.util.ArrayList) [170
  elements]
  2016-11-14 11:26:34,378 INFO
  org.apache.flink.yarn.YarnClusterClient
  - Disconnecting YarnClusterClient from ApplicationMaster
```

The following are the screenshots of the job execution from the Flink application master UI. Here is the screenshot of the Flink execution plan:

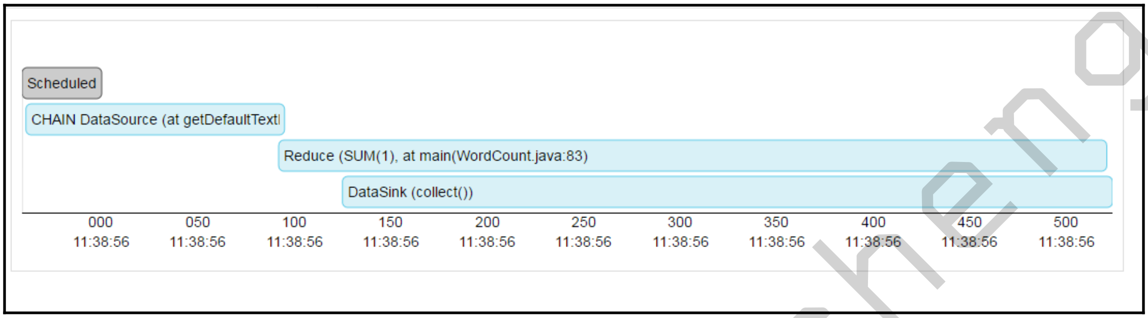

Apache Flink Dashboard
Overview
Running Jobs
Completed Jobs
Task Managers
Job Manager
Submit new Job
Flink Java Job at Mon Nov 14 11:38:55 IST 2016
22a8431085e04cb8e8648351eab5d5e4
0 0 0 0 3 0 0 0
Plan
Timeline
Exceptions
Properties
Configuration
+
−
Data Source -> FlatMap -> GroupCombine
DataSource (at getDefaultTextLineDataSet(WordCountData.java:70) (org.apache.flink.api.java.io.CollectionInputFormat))/> -> FlatMap (FlatMap at main(WordCount.java:80)) -> Combine(SUM(1), at main(WordCount.java:83)
Parallelism: 1
Operation: (none) -> FlatMap -> Sorted Combine
GroupReduce
Reduce (SUM(1), at main(WordCount.java:83)
Parallelism: 20
Operation: Sorted Group Reduce
Data Sink
DataSink (collect())
Parallelism: 20
Operation: (none)

Next we can see the screenshot of the steps getting executed for this job:



Start Time	End Time	Duration	Name	Bytes received	Records received	Bytes sent	Records sent	Tasks	Status
2016-11-14, 11:38:55	2016-11-14, 11:38:56	134ms	CHAIN DataSource (at getDefaultTextLineDataSet(WordCountData.java:70) (org.apache.flink.api.java.io.CollectionInputFormat)) -> FlatMap (FlatMap at main(WordCount.java:80)) -> Combine(SUM(1), at main(WordCount.java:83)	0 B	0	1.66 KB	170	0 0 0 1 0 0 0	FINISHED
2016-11-14, 11:38:56	2016-11-14, 11:38:56	429ms	Reduce (SUM(1), at main(WordCount.java:83)	1.66 KB	170	1.66 KB	170	0 0 0 20 0 0 0	FINISHED
2016-11-14, 11:38:56	2016-11-14, 11:38:56	399ms	DataSink (collect())	1.66 KB	170	0 B	0	0 0 0 20 0 0 0	FINISHED

And at last we have the screenshot of the timeline of the Flink job execution. The timeline shows all the steps that can be executed in parallel and what needs to be executed sequentially:



Stopping Flink YARN session

Once the processing is done, you can stop the Flink YARN session in two ways. First you can simple do a *Cltr+C* on the console where you started the YARN session. This will send the termination signal and stop the YARN session.

The second way is to execute the following command to stop the session:

```
./bin/yarn-session.sh -id application_1478079131011_0107 stop
```

We can see immediately the Flink YARN application get killed:

```
2016-11-14 11:56:59,455 INFO
org.apache.flink.yarn.YarnClusterClient
Sending shutdown request to the Application Master
2016-11-14 11:56:59,456 INFO
org.apache.flink.yarn.ApplicationClient
Sending StopCluster request to JobManager.
2016-11-14 11:56:59,464 INFO
org.apache.flink.yarn.YarnClusterClient
- Deleted Yarn properties file at /tmp/.yarn-properties-root
2016-11-14 11:56:59,464 WARN
org.apache.flink.yarn.YarnClusterClient
Session file directory not set. Not deleting session files
2016-11-14 11:56:59,565 INFO
org.apache.flink.yarn.YarnClusterClient
- Application application_1478079131011_0107 finished with state
FINISHED and final state SUCCEEDED at 1479104819469
2016-11-14 11:56:59,565 INFO
```

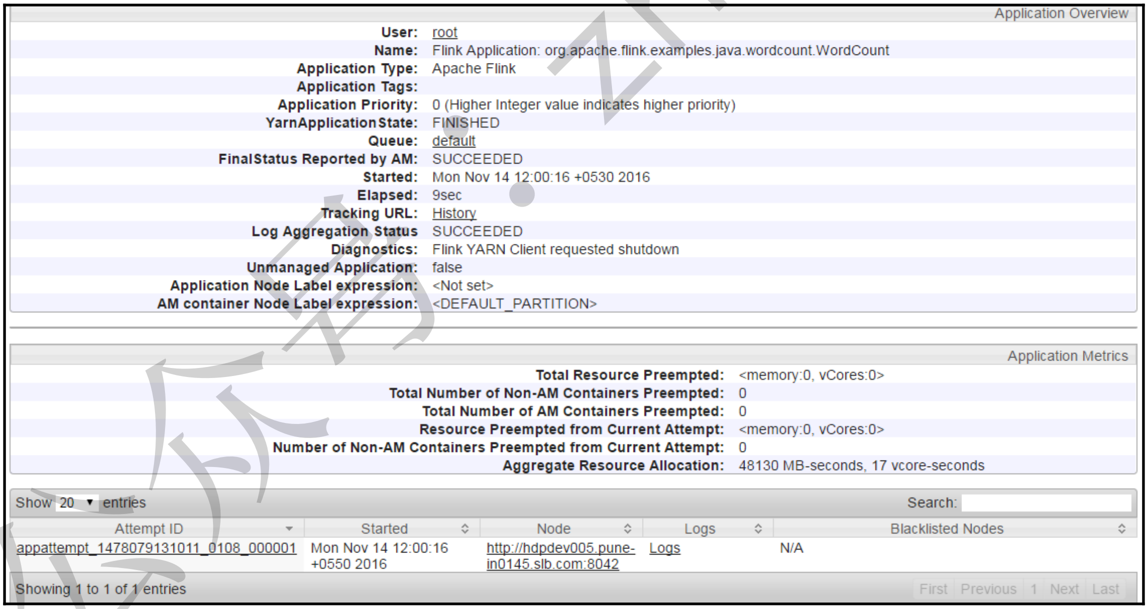
```
org.apache.flink.yarn.YarnClusterClient
- YARN Client is shutting down
```

Running a single Flink job on YARN

We can also run a single Flink job on YARN without blocking the resources for the YARN session. This is a good option if you only wish to run a single Flink job on YARN. In the earlier case, when we start the Flink session on YARN, it blocks the resources and cores until we stop the session whereas in this case the resources are blocked till the job is executing and they are freed up as soon as the job is complete. The following command shows how we execute a single Flink job on YARN without a session:

```
./bin/flink run -m yarn-cluster -yn 2
./examples/batch/WordCount.jar
```

We can see similar results as we saw in the earlier case. We can also track its progress and debugging using the YARN application UI. The following is a sample screenshot of the same:



Application Overview

User:	root
Name:	Flink Application: org.apache.flink.examples.java.wordcount.WordCount
Application Type:	Apache Flink
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Mon Nov 14 12:00:16 +0530 2016
Elapsed:	9sec
Tracking URL:	History
Log Aggregation Status	SUCCEEDED
Diagnostics:	Flink YARN Client requested shutdown
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	48130 MB-seconds, 17 vcore-seconds

Show 20 entries Search:

Attempt ID	Started	Node	Logs	Blacklisted Nodes
appattempt_1478079131011_0108_000001	Mon Nov 14 12:00:16 +0550 2016	http://hdpdev005.pune-in0145.slb.com:8042	Logs	N/A

Showing 1 to 1 of 1 entries

Recovery behavior for Flink on YARN

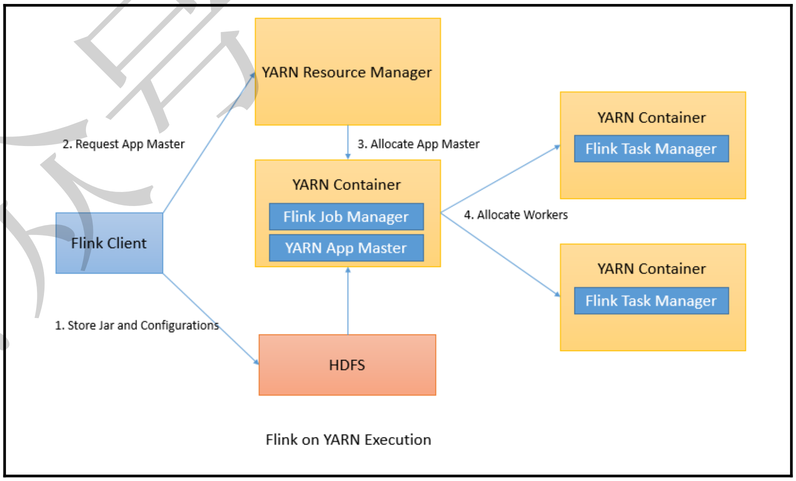
Flink on YARN provides the following configuration parameters for tuning the recovery behavior:

Parameter	Description
`yarn.reallocate-failed`	Sets whether Flink should reallocate failed task manager containers. The default is `true`.
`yarn.maximum-failed-containers`	Sets the maximum number of failed containers the application master accepts before failing the YARN session. The default is number of task managers requested during initiation.
`yarn.application-attempts`	Sets the number of application master attempts. The default is `1`, which means if complete, the YARN session will fail if the application master fails.

These configurations need to be in either `conf/flink-conf.yaml` or can be set during the session initiation using the `-D` parameter.

Working details

In the previous sections, we looked at how we can use Flink on YARN. Now let's try to understand how it works internally:



Flink on YARN Execution

The preceding diagram shows the internal workings of Flink on YARN. It goes through the following steps:

1. Checks if the Hadoop and YARN configuration directories are set.
2. If yes, contacts HDFS and stores the JAR and configuration on HDFS.
3. Contacts the node manager for allocating the application master.
4. Once the application master is allocated, initiates the Flink Job Manager.
5. Later, initiates the Flink task managers based on the configuration parameters given.

Now we are all set to submit the Flink job on YARN.

Summary

In this chapter, we talked about how to use existing YARN clusters to execute Flink jobs in a distributed mode. We looked at the step-by-step details and understood some practical examples for the same.

In the next chapter, we are going to see how to execute Flink jobs in the cloud environment.

9
Deploying Flink on Cloud

In the recent times, more and more companies have been investing in Cloud-based solutions and which is justified looking at the cost and efficiency we achieve through the Cloud. **Amazon Web Services** (**AWS**), **Google Cloud Platform** (**GCP**) and Microsoft Azure are the clear leaders so far in this business. Almost all of them provide big data solutions which are quite handy to use. The cloud provides efficient solutions in a timely manner where people don't need to worry about hardware purchases, networking, and so on.

In this chapter, we are going to see how we can deploy Flink on cloud. We will see a detailed approach to installing and deploying applications on AWS and Google Cloud. So let's get started.

Flink on Google Cloud

Flink can be deployed on Google Cloud using one utility called BDUtil. It is an open source utility available for everyone to use `https://cloud.google.com/hadoop/bdutil`. The very first step we need to do is to install **Google Cloud SDK**.

Installing Google Cloud SDK

Google Cloud SDK is an executable utility that can be installed on the Windows, Mac, or UNIX operating systems. You can choose the mode of installation based on your operating system. Here is a link that directs users about detailed installations at `https://cloud.google.com/sdk/downloads`.

Here I assume that you are already familiar with Google Cloud concepts and the terminologies; if not I would recommend reading `https://cloud.google.com/docs/`.

In my case, I will be using UNIX machine to launch a Flink-Hadoop cluster. So let's get started with the installation.

First, we need to download the installer for the Cloud SDK.

```
wget
https://dl.google.com/dl/cloudsdk/channels/rapid/downloads/google-
cloud-sdk-135.0.0-linux-x86_64.tar.gz
```

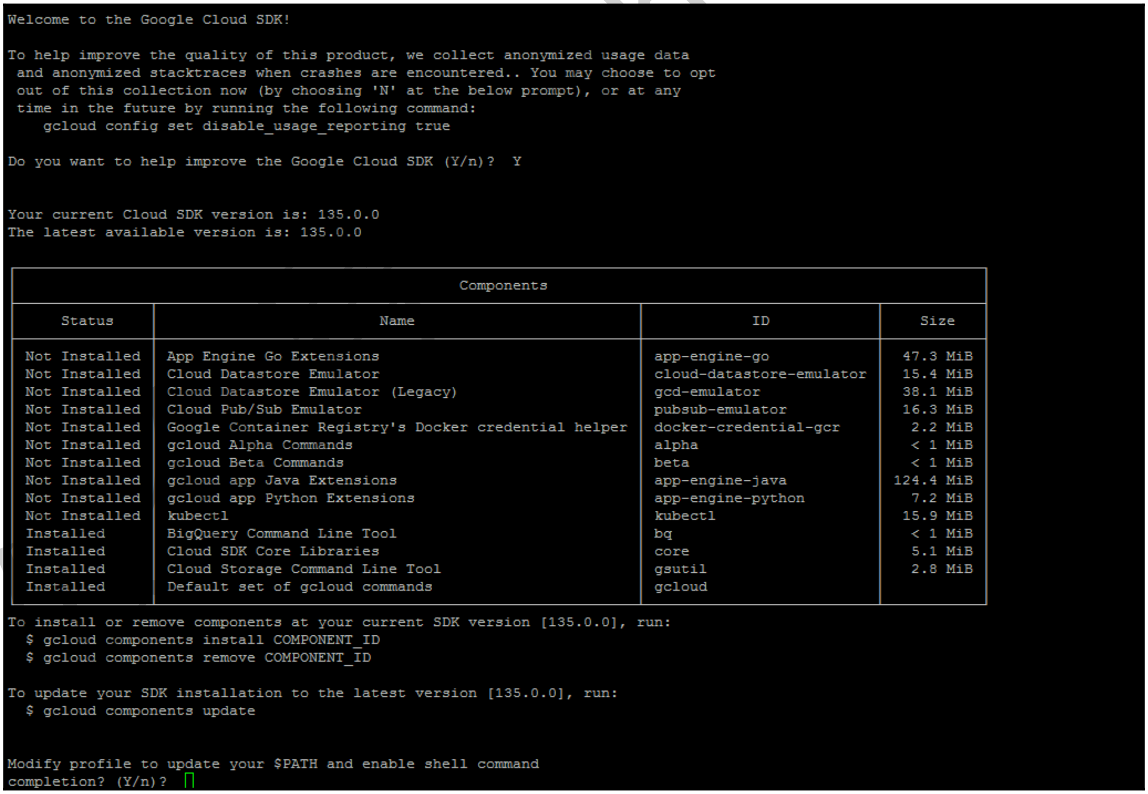
Next we un-tar the files by the following command:

```
tar -xzf google-cloud-sdk-135.0.0-linux-x86_64.tar.gz
```

Once done, we need to initialize the SDK:

```
cd google-cloud-sdk
bin/gcloud init
```

This will start an interactive installation process and will require you to provide input as and when needed. The following screenshot shows the process:



```
Welcome to the Google Cloud SDK!

To help improve the quality of this product, we collect anonymized usage data
 and anonymized stacktraces when crashes are encountered.. You may choose to opt
 out of this collection now (by choosing 'N' at the below prompt), or at any
 time in the future by running the following command:
    gcloud config set disable_usage_reporting true

Do you want to help improve the Google Cloud SDK (Y/n)?  Y


Your current Cloud SDK version is: 135.0.0
The latest available version is: 135.0.0

┌──────────────────────────────────────────────────────────────────────────────────────────────────────────────┐
│                                                  Components                                                  │
├───────────────┬──────────────────────────────────────────────────────┬──────────────────────────┬────────────┤
│     Status    │                         Name                         │            ID            │    Size    │
├───────────────┼──────────────────────────────────────────────────────┼──────────────────────────┼────────────┤
│ Not Installed │ App Engine Go Extensions                             │ app-engine-go            │   47.3 MiB │
│ Not Installed │ Cloud Datastore Emulator                             │ cloud-datastore-emulator │   15.4 MiB │
│ Not Installed │ Cloud Datastore Emulator (Legacy)                    │ gcd-emulator             │   38.1 MiB │
│ Not Installed │ Cloud Pub/Sub Emulator                               │ pubsub-emulator          │   16.3 MiB │
│ Not Installed │ Google Container Registry's Docker credential helper │ docker-credential-gcr    │    2.2 MiB │
│ Not Installed │ gcloud Alpha Commands                                │ alpha                    │    < 1 MiB │
│ Not Installed │ gcloud Beta Commands                                 │ beta                     │    < 1 MiB │
│ Not Installed │ gcloud app Java Extensions                           │ app-engine-java          │  124.4 MiB │
│ Not Installed │ gcloud app Python Extensions                         │ app-engine-python        │    7.2 MiB │
│ Not Installed │ kubectl                                              │ kubectl                  │   15.9 MiB │
│ Installed     │ BigQuery Command Line Tool                           │ bq                       │    < 1 MiB │
│ Installed     │ Cloud SDK Core Libraries                             │ core                     │    5.1 MiB │
│ Installed     │ Cloud Storage Command Line Tool                      │ gsutil                   │    2.8 MiB │
│ Installed     │ Default set of gcloud commands                       │ gcloud                   │            │
└───────────────┴──────────────────────────────────────────────────────┴──────────────────────────┴────────────┘
To install or remove components at your current SDK version [135.0.0], run:
  $ gcloud components install COMPONENT_ID
  $ gcloud components remove COMPONENT_ID

To update your SDK installation to the latest version [135.0.0], run:
  $ gcloud components update


Modify profile to update your $PATH and enable shell command
completion? (Y/n)?
```

It is also recommended to get authenticated by executing the following command:

```
gcloud auth login
```

This will give you a URL to be opened in your machine's browser. On hitting that URL, you will a get code which will be used for authentication.

Once the authentication is done, we are all set for the BDUtil installation.

Installing BDUtil

As we said earlier, BDUtil is a utility developed by Google to facilitate hiccup-free big data installations on Google Cloud. You can install the following services:

- Hadoop – HDP and CDH
- Flink
- Hama
- Hbase
- Spark
- Storm
- Tajo

The following steps are required to install BDUtil. First of all, we need to download the source code:

```
wget
https://github.com/GoogleCloudPlatform/bdutil/archive/master.zip
```

Unzip the code by the following command:

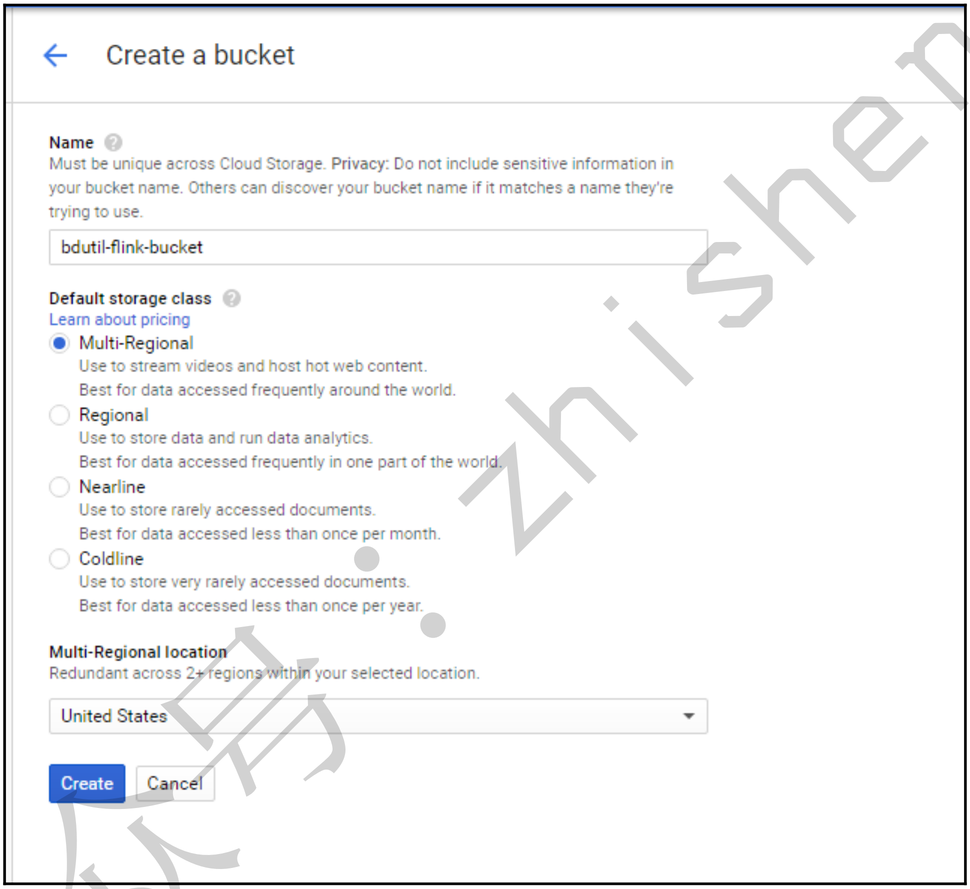
```
unzip master.zip
cd bdutil-master
```

It is recommended to use a **non-root account** for BDUtil operations if you are using it on one the Google Compute machine. Generally root logins are by default disabled on all compute engine machines.

Now we are all set with the BDUtil installation and ready for deployment.

Launching a Flink cluster

BDUtil needs at least one project in which we will do our installations and a bucket where temporary files can be kept. To create a bucket, you can go to the **Cloud Storage** sections and choose to create a bucket as shown in the following screenshot:



Create a bucket

Name

Must be unique across Cloud Storage. **Privacy:** Do not include sensitive information in your bucket name. Others can discover your bucket name if it matches a name they're trying to use.

bdutil-flink-bucket

Default storage class

Learn about pricing

● Multi-Regional
Use to stream videos and host hot web content.
Best for data accessed frequently around the world.

○ Regional
Use to store data and run data analytics.
Best for data accessed frequently in one part of the world.

○ Nearline
Use to store rarely accessed documents.
Best for data accessed less than once per month.

○ Coldline
Use to store very rarely accessed documents.
Best for data accessed less than once per year.

Multi-Regional location

Redundant across 2+ regions within your selected location.

United States ▾

We have named this bucket at **bdutil-flink-bucket**. Next we need to edit the `bdutil_env.sh` file to configure information about the project name, bucket name and Google Cloud zone to be used. We can also set other things such as the machine type and Operating System. `bdutil_env.sh` looks as shown in the following:

```
 # A GCS bucket used for sharing generated SSH keys and GHFS configuration.
CONFIGBUCKET="bdutil-flink-bucket"

# The Google Cloud Platform text-based project-id which owns the GCE
resources.
PROJECT="bdutil-flink-project"


###################### Cluster/Hardware Configuration #########
# These settings describe the name, location, shape and size of your
cluster,
# though these settings may also be used in deployment-configuration--for
# example, to whitelist intra-cluster SSH using the cluster prefix.

# GCE settings.
GCE_IMAGE='https://www.googleapis.com/compute/v1/projects/debian-cloud/glob
al/images/backports-debian-7-wheezy-v20160531'
GCE_MACHINE_TYPE='n1-standard-4'
GCE_ZONE="europe-west1-d"
# When setting a network it's important for all nodes be able to
communicate
# with eachother and for SSH connections to be allowed inbound to complete
# cluster setup and configuration.
```

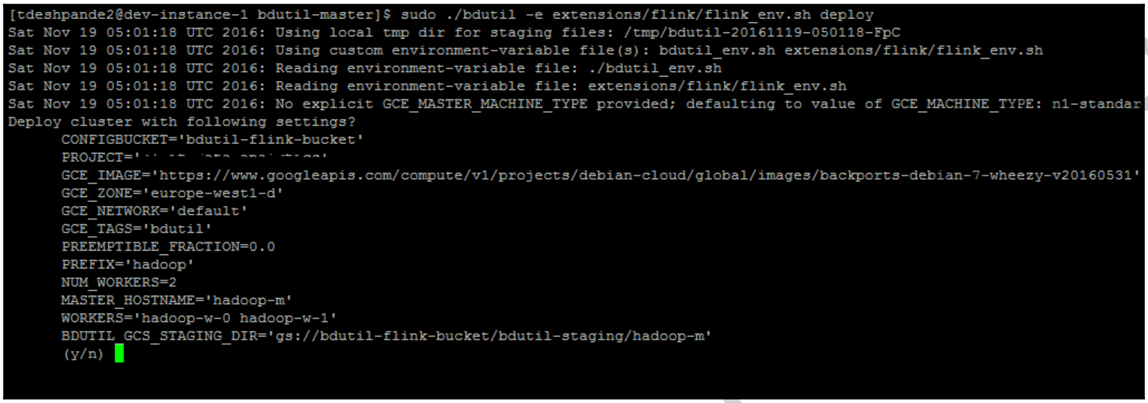
By default, the configuration launches three node, Hadoop/Flink cluster with one master and two worker nodes.

> If you are using the trial version of GCP, then it is recommended to use machine type as **n1-standard-2**. This will restrict the CPU and storage of the node type.

Now we are all set to launch the cluster, with the following command:

```
./bdutil -e extensions/flink/flink_env.sh deploy
```

This will start creating machines and will deploy required software on it. It generally takes 10-20 minutes of time to get the cluster up and running if everything works well. Before starting the executing, you should review what the screen shot tell us.



```
[tdeshpande2@dev-instance-1 bdutil-master]$ sudo ./bdutil -e extensions/flink/flink_env.sh deploy
Sat Nov 19 05:01:18 UTC 2016: Using local tmp dir for staging files: /tmp/bdutil-20161119-050118-FpC
Sat Nov 19 05:01:18 UTC 2016: Using custom environment-variable file(s): bdutil_env.sh extensions/flink/flink_env.sh
Sat Nov 19 05:01:18 UTC 2016: Reading environment-variable file: ./bdutil_env.sh
Sat Nov 19 05:01:18 UTC 2016: Reading environment-variable file: extensions/flink/flink_env.sh
Sat Nov 19 05:01:18 UTC 2016: No explicit GCE_MASTER_MACHINE_TYPE provided; defaulting to value of GCE_MACHINE_TYPE: n1-standar
Deploy cluster with following settings?
      CONFIGBUCKET='bdutil-flink-bucket'
      PROJECT='[illegible]'
      GCE_IMAGE='https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/backports-debian-7-wheezy-v20160531'
      GCE_ZONE='europe-west1-d'
      GCE_NETWORK='default'
      GCE_TAGS='bdutil'
      PREEMPTIBLE_FRACTION=0.0
      PREFIX='hadoop'
      NUM_WORKERS=2
      MASTER_HOSTNAME='hadoop-m'
      WORKERS='hadoop-w-0 hadoop-w-1'
      BDUTIL_GCS_STAGING_DIR='gs://bdutil-flink-bucket/bdutil-staging/hadoop-m'
      (y/n)
```

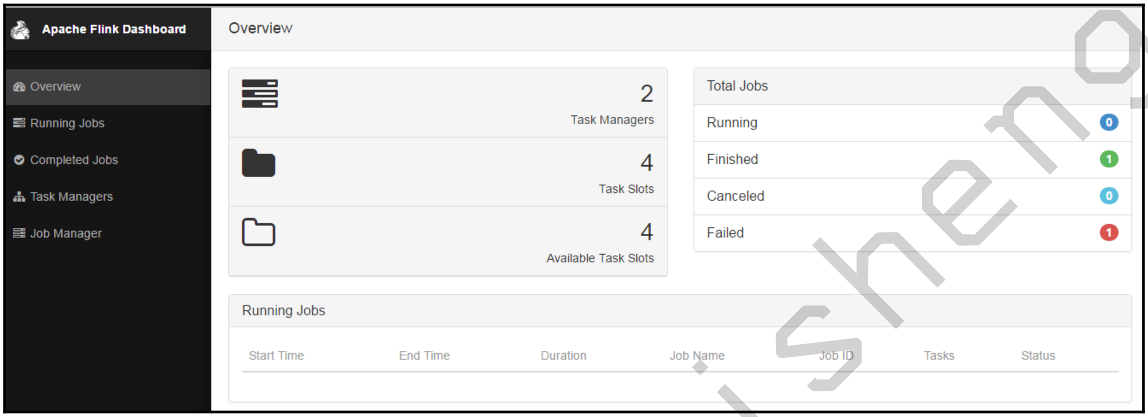
Once complete, you will see some messages as shown in the following:

```
gcloud --project=bdutil ssh --zone=europe-west1-c hadoop-m
Sat Nov 19 06:12:27 UTC 2016: Staging files successfully deleted.
Sat Nov 19 06:12:27 UTC 2016: Invoking on master: ./deploy-ssh-master-
setup.sh
.Sat Nov 19 06:12:27 UTC 2016: Waiting on async 'ssh' jobs to finish. Might
take a while...
.
Sat Nov 19 06:12:29 UTC 2016: Step 'deploy-ssh-master-setup,*' done...
Sat Nov 19 06:12:29 UTC 2016: Invoking on workers: ./deploy-core-setup.sh
..Sat Nov 19 06:12:29 UTC 2016: Invoking on master: ./deploy-core-setup.sh
.Sat Nov 19 06:12:30 UTC 2016: Waiting on async 'ssh' jobs to finish. Might
take a while...
...
Sat Nov 19 06:13:14 UTC 2016: Step 'deploy-core-setup,deploy-core-setup'
done...
Sat Nov 19 06:13:14 UTC 2016: Invoking on workers: ./deploy-ssh-worker-
setup.sh
..Sat Nov 19 06:13:15 UTC 2016: Waiting on async 'ssh' jobs to finish.
Might take a while...
..
Sat Nov 19 06:13:17 UTC 2016: Step '*,deploy-ssh-worker-setup' done...
Sat Nov 19 06:13:17 UTC 2016: Invoking on master: ./deploy-master-nfs-
setup.sh
.Sat Nov 19 06:13:17 UTC 2016: Waiting on async 'ssh' jobs to finish. Might
take a while...
.
```

```
Sat Nov 19 06:13:23 UTC 2016: Step 'deploy-master-nfs-setup,*' done...
Sat Nov 19 06:13:23 UTC 2016: Invoking on workers: ./deploy-client-nfs-
setup.sh
..Sat Nov 19 06:13:23 UTC 2016: Invoking on master: ./deploy-client-nfs-
setup.sh
.Sat Nov 19 06:13:24 UTC 2016: Waiting on async 'ssh' jobs to finish. Might
take a while...
...
Sat Nov 19 06:13:33 UTC 2016: Step 'deploy-client-nfs-setup,deploy-client-
nfs-setup' done...
Sat Nov 19 06:13:33 UTC 2016: Invoking on master: ./deploy-start.sh
.Sat Nov 19 06:13:34 UTC 2016: Waiting on async 'ssh' jobs to finish. Might
take a while...
.
Sat Nov 19 06:13:49 UTC 2016: Step 'deploy-start,*' done...
Sat Nov 19 06:13:49 UTC 2016: Invoking on workers: ./install_flink.sh
..Sat Nov 19 06:13:49 UTC 2016: Invoking on master: ./install_flink.sh
.Sat Nov 19 06:13:49 UTC 2016: Waiting on async 'ssh' jobs to finish. Might
take a while...
...
Sat Nov 19 06:13:53 UTC 2016: Step 'install_flink,install_flink' done...
Sat Nov 19 06:13:53 UTC 2016: Invoking on master: ./start_flink.sh
.Sat Nov 19 06:13:54 UTC 2016: Waiting on async 'ssh' jobs to finish. Might
take a while...
.
Sat Nov 19 06:13:55 UTC 2016: Step 'start_flink,*' done...
Sat Nov 19 06:13:55 UTC 2016: Command steps complete.
Sat Nov 19 06:13:55 UTC 2016: Execution complete. Cleaning up temporary
files...
Sat Nov 19 06:13:55 UTC 2016: Cleanup complete.
```

In case of any failures in between, please check what logs say. You can visit the Google Cloud Compute Engine Console to get the exact IPs of the master and slave machines.

Now if you check the Job Manager UI, you should have two task managers and four task slots available for the use. You can hit URL `http://<master-node-ip>:8081`. The following is sample screenshot for the same:



2 Task Managers

4 Task Slots

4 Available Task Slots

Total Jobs	
Running	0
Finished	1
Canceled	0
Failed	1

Running Jobs

Start Time	End Time	Duration	Job Name	Job ID	Tasks	Status

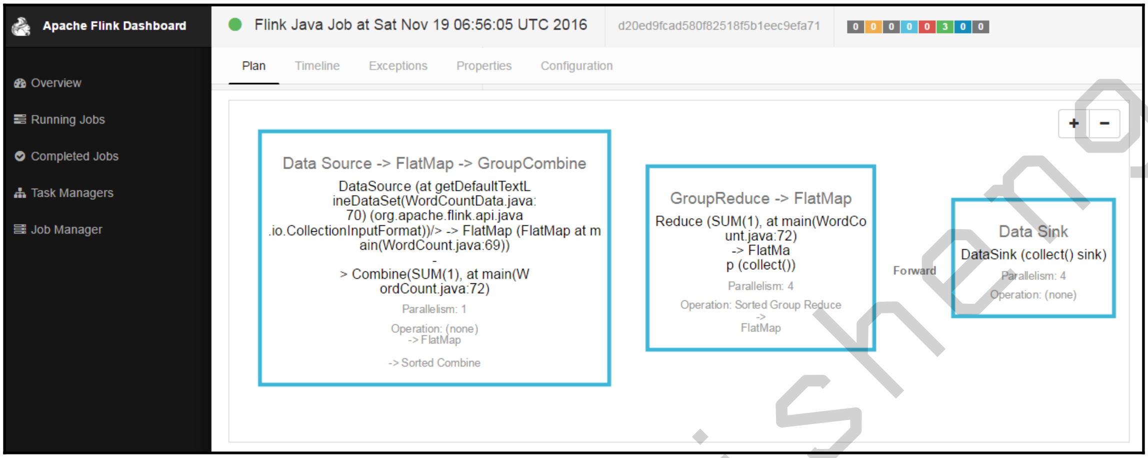
Executing a sample job

You can check if everything is working fine by launching a sample word count program. In order to do so, we first need to log in to Flink Master node. The following command starts a sample word count program provided by Flink installation.

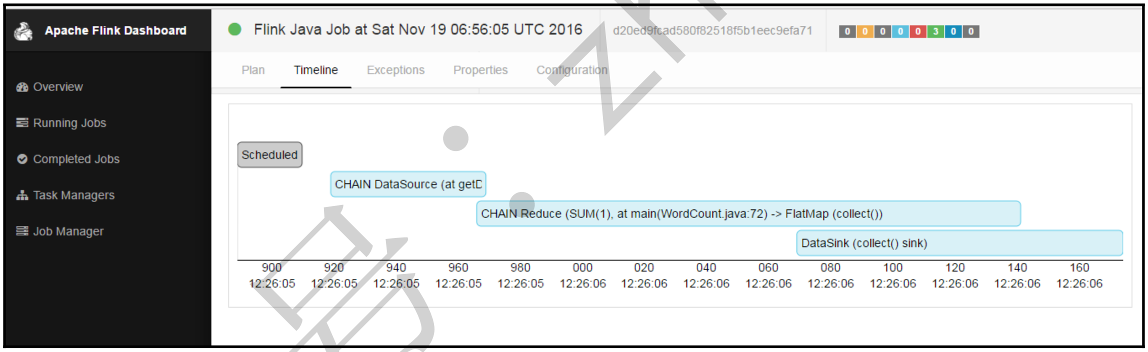
```
      /home/hadoop/flink-install/bin$ ./flink run
      ../examples/WordCount.jar
   11/19/2016 06:56:05    Job execution switched to status RUNNING.
   11/19/2016 06:56:05    CHAIN DataSource (at
   getDefaultTextLineDataSet(WordCountData.java:70)
   (org.apache.flink.api.java.io.CollectionInputFormat)) -> FlatMap (FlatMap
   at main(WordCount.java:69)) -> Combine(SUM(1), at
   main(WordCount.java:72)(1/1) switched to SCHEDULED
   11/19/2016 06:56:05    CHAIN DataSource (at
   getDefaultTextLineDataSet(WordCountData.java:70)
   (org.apache.flink.api.java.io.CollectionInputFormat)) -> FlatMap (FlatMap
   at main(WordCount.java:69)) -> Combine(SUM(1), at
   main(WordCount.java:72)(1/1) switched to DEPLOYING
   11/19/2016 06:56:05    CHAIN DataSource (at
   getDefaultTextLineDataSet(WordCountData.java:70)
   (org.apache.flink.api.java.io.CollectionInputFormat)) -> FlatMap (FlatMap
   at main(WordCount.java:69)) -> Combine(SUM(1), at
   main(WordCount.java:72)(1/1) switched to RUNNING
   11/19/2016 06:56:05    CHAIN Reduce (SUM(1), at main(WordCount.java:72) ->
   FlatMap (collect())(1/4) switched to SCHEDULED
```

```
11/19/2016 06:56:05     CHAIN DataSource (at
getDefaultTextLineDataSet(WordCountData.java:70)
(org.apache.flink.api.java.io.CollectionInputFormat)) -> FlatMap (FlatMap
at main(WordCount.java:69)) -> Combine(SUM(1), at
main(WordCount.java:72)(1/1) switched to FINISHED
...
RUNNING
11/19/2016 06:56:06     DataSink (collect() sink)(3/4) switched to
SCHEDULED
11/19/2016 06:56:06     DataSink (collect() sink)(3/4) switched to
DEPLOYING
11/19/2016 06:56:06     DataSink (collect() sink)(1/4) switched to
SCHEDULED
11/19/2016 06:56:06     DataSink (collect() sink)(1/4) switched to
DEPLOYING
11/19/2016 06:56:06     CHAIN Reduce (SUM(1), at main(WordCount.java:72) ->
FlatMap (collect())(1/4) switched to FINISHED
11/19/2016 06:56:06     CHAIN Reduce (SUM(1), at main(WordCount.java:72) ->
FlatMap (collect())(3/4) switched to FINISHED
11/19/2016 06:56:06     DataSink (collect() sink)(3/4) switched to
11/19/2016 06:56:06     CHAIN Reduce (SUM(1), at
11/19/2016 06:56:06     DataSink (collect() sink)(2/4) switched to FINISHED
11/19/2016 06:56:06     Job execution switched to status FINISHED.
(after,1)
(arms,1)
(arrows,1)
(awry,1)
(bare,1)
(be,4)
(coil,1)
(consummation,1)
(contumely,1)
(d,4)
(delay,1)
(despis,1)
...
```

The following screenshot shows the execution map of the job:


Apache Flink Dashboard
Overview
Running Jobs
Completed Jobs
Task Managers
Job Manager
Flink Java Job at Sat Nov 19 06:56:05 UTC 2016
d20ed9fcad580f82518f5b1eec9efa71
0 0 0 0 0 3 0 0
Plan
Timeline
Exceptions
Properties
Configuration
+
−
Data Source -> FlatMap -> GroupCombine
DataSource (at getDefaultTextLineDataSet(WordCountData.java:70) (org.apache.flink.api.java.io.CollectionInputFormat)) -> FlatMap (FlatMap at main(WordCount.java:69)) -> Combine(SUM(1), at main(WordCount.java:72)
Parallelism: 1
Operation: (none) -> FlatMap -> Sorted Combine
GroupReduce -> FlatMap
Reduce (SUM(1), at main(WordCount.java:72) -> FlatMap (collect())
Parallelism: 4
Operation: Sorted Group Reduce -> FlatMap
Forward
Data Sink
DataSink (collect() sink)
Parallelism: 4
Operation: (none)

Here is another screenshot of the timeline with which all tasks got executed:



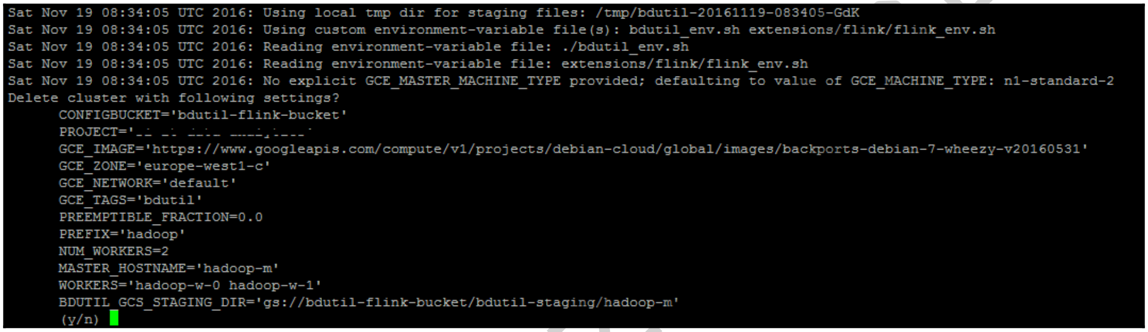
Shutting down the cluster

Once we are done with all our executions and if we no longer wish to do any further use of the cluster then it is better to shut it down.

The following is a command, we need to execute to shut down the cluster we started:

```
./bdutil -e extensions/flink/flink_env.sh delete
```

Please make sure to confirm the configurations before deleting the cluster. The following is a screenshot which shows what it is going to delete and the complete process:



```
Sat Nov 19 08:34:05 UTC 2016: Using local tmp dir for staging files: /tmp/bdutil-20161119-083405-GdK
Sat Nov 19 08:34:05 UTC 2016: Using custom environment-variable file(s): bdutil_env.sh extensions/flink/flink_env.sh
Sat Nov 19 08:34:05 UTC 2016: Reading environment-variable file: ./bdutil_env.sh
Sat Nov 19 08:34:05 UTC 2016: Reading environment-variable file: extensions/flink/flink_env.sh
Sat Nov 19 08:34:05 UTC 2016: No explicit GCE_MASTER_MACHINE_TYPE provided; defaulting to value of GCE_MACHINE_TYPE: n1-standard-2
Delete cluster with following settings?
      CONFIGBUCKET='bdutil-flink-bucket'
      PROJECT='[redacted]'
      GCE_IMAGE='https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/backports-debian-7-wheezy-v20160531'
      GCE_ZONE='europe-west1-c'
      GCE_NETWORK='default'
      GCE_TAGS='bdutil'
      PREEMPTIBLE_FRACTION=0.0
      PREFIX='hadoop'
      NUM_WORKERS=2
      MASTER_HOSTNAME='hadoop-m'
      WORKERS='hadoop-w-0 hadoop-w-1'
      BDUTIL_GCS_STAGING_DIR='gs://bdutil-flink-bucket/bdutil-staging/hadoop-m'
      (y/n)
```

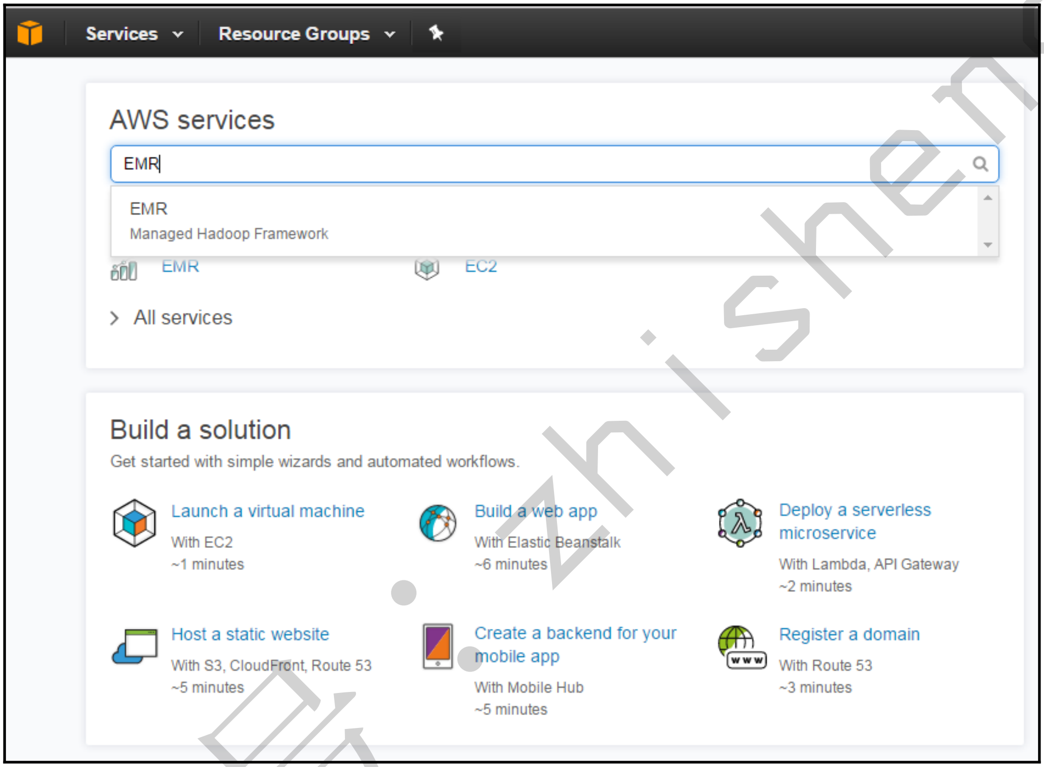
Flink on AWS

Now let's look at how we can use Flink on **Amazon Web Services** (**AWS**). Amazon provides a hosted Hadoop service called **Elastic Map Reduce** (**EMR**). We can use and Flink in combination. We can do reading on EMR at `https://aws.amazon.com/documentation/elastic-mapreduce/`.

Here I assume that you already have AWS account and knows basics of AWS.

Launching an EMR cluster

The very first thing we need to do is launch EMR cluster. We first need to log in to AWS account and choose **EMR** service from the console as shown in the following screenshot:



AWS services

EMR

EMR
Managed Hadoop Framework

Build a solution

Get started with simple wizards and automated workflows.

Launch a virtual machine
With EC2
~1 minutes

Build a web app
With Elastic Beanstalk
~6 minutes

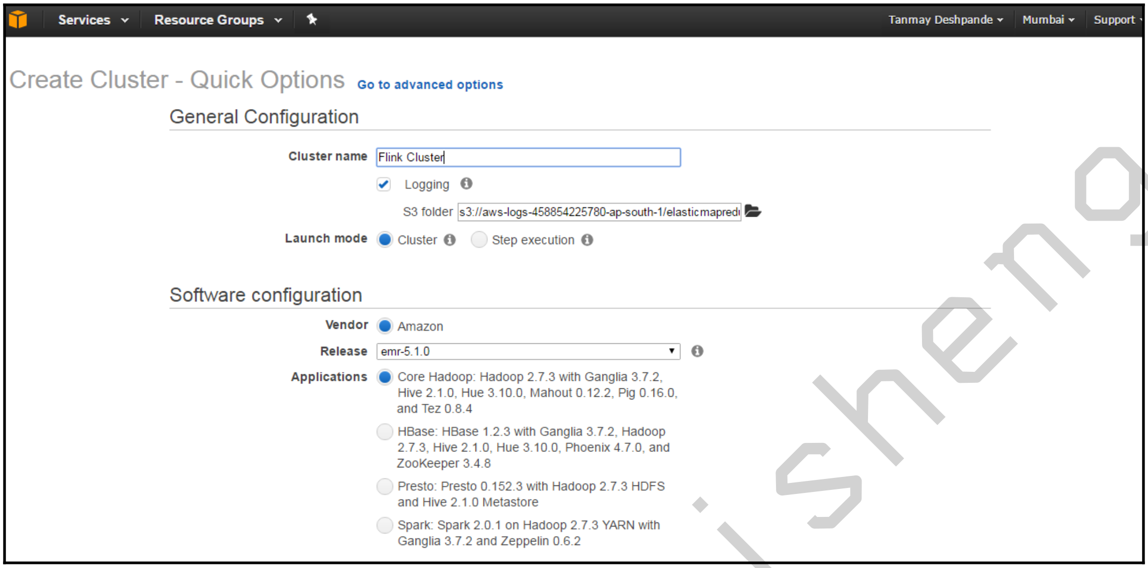
Deploy a serverless microservice
With Lambda, API Gateway
~2 minutes

Host a static website
With S3, CloudFront, Route 53
~5 minutes

Create a backend for your mobile app
With Mobile Hub
~5 minutes

Register a domain
With Route 53
~3 minutes

Next we go to EMR console and launch a three-node cluster with one master and two slave nodes. Here we choose minimum cluster size to avoid surprise billing. The following screenshot shows the EMR cluster creation screen:



Create Cluster - Quick Options Go to advanced options

General Configuration

Cluster name: Flink Cluster

☑ Logging

S3 folder: s3://aws-logs-458854225780-ap-south-1/elasticmapredi

Launch mode: ● Cluster ○ Step execution

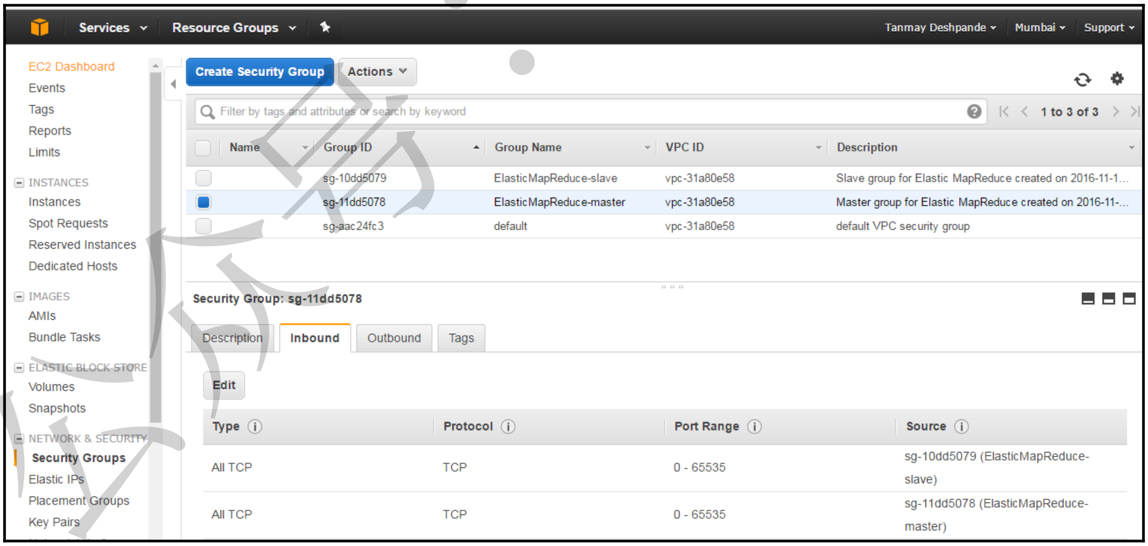
Software configuration

Vendor: ● Amazon

Release: emr-5.1.0

Applications:
● Core Hadoop: Hadoop 2.7.3 with Ganglia 3.7.2, Hive 2.1.0, Hue 3.10.0, Mahout 0.12.2, Pig 0.16.0, and Tez 0.8.4
○ HBase: HBase 1.2.3 with Ganglia 3.7.2, Hadoop 2.7.3, Hive 2.1.0, Hue 3.10.0, Phoenix 4.7.0, and ZooKeeper 3.4.8
○ Presto: Presto 0.152.3 with Hadoop 2.7.3 HDFS and Hive 2.1.0 Metastore
○ Spark: Spark 2.0.1 on Hadoop 2.7.3 YARN with Ganglia 3.7.2 and Zeppelin 0.6.2

Generally it takes 10-15 minutes for cluster to be up and running. Once the cluster is ready, we can do SSH to the cluster. For that we first need to click on **Create Security Group** section and add rule to add SSH port 22 rule. The following screen shows the security group section in which we need to edit **In Bound** traffic rule for SSH:

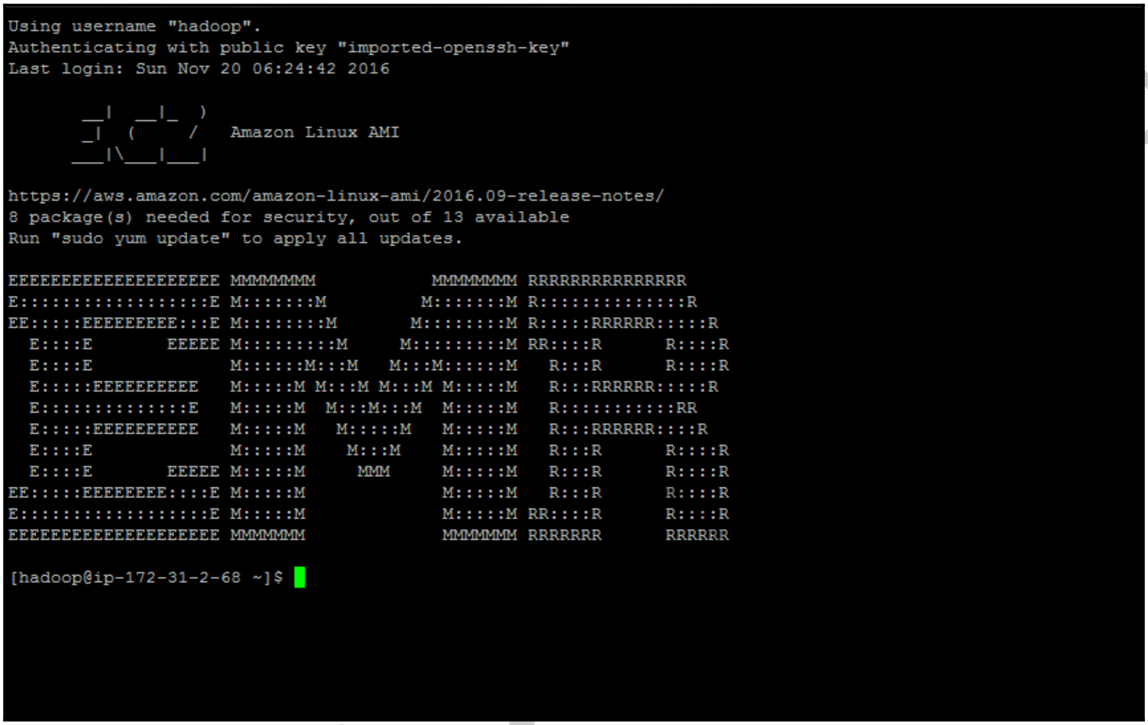


Name	Group ID	Group Name	VPC ID	Description
	sg-10dd5079	ElasticMapReduce-slave	vpc-31a80e58	Slave group for Elastic MapReduce created on 2016-11-1...
	sg-11dd5078	ElasticMapReduce-master	vpc-31a80e58	Master group for Elastic MapReduce created on 2016-11-...
	sg-aac24fc3	default	vpc-31a80e58	default VPC security group

Security Group: sg-11dd5078

Type	Protocol	Port Range	Source
All TCP	TCP	0 - 65535	sg-10dd5079 (ElasticMapReduce-slave)
All TCP	TCP	0 - 65535	sg-11dd5078 (ElasticMapReduce-master)

Now we are all set to login to the master node using SSH with the private key. You will see the following screen once you login with user name Hadoop:



```
Using username "hadoop".
Authenticating with public key "imported-openssh-key"
Last login: Sun Nov 20 06:24:42 2016

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2016.09-release-notes/
8 package(s) needed for security, out of 13 available
Run "sudo yum update" to apply all updates.

EEEEEEEEEEEEEEEEEEEE MMMMMMMM           MMMMMMMM RRRRRRRRRRRRRRR
E::::::::::::::::::E M:::::::M         M:::::::M R::::::::::::::R
EE:::::EEEEEEEEE:::E M::::::::M       M::::::::M R:::::RRRRRR:::::R
  E::::E       EEEEE M:::::::::M     M:::::::::M RR::::R      R::::R
  E::::E             M::::::M:::M   M:::M::::::M   R:::R      R::::R
  E:::::EEEEEEEEEE   M:::::M M:::M M:::M M:::::M   R:::RRRRRR:::::R
  E::::::::::::::E   M:::::M  M:::M:::M  M:::::M   R:::::::::::RR
  E:::::EEEEEEEEEE   M:::::M   M:::::M   M:::::M   R:::RRRRRR::::R
  E::::E             M:::::M    M:::M    M:::::M   R:::R      R::::R
  E::::E       EEEEE M:::::M     MMM     M:::::M   R:::R      R::::R
EE:::::EEEEEEEE::::E M:::::M             M:::::M   R:::R      R::::R
E::::::::::::::::::E M:::::M             M:::::M RR::::R      R::::R
EEEEEEEEEEEEEEEEEEEE MMMMMMM             MMMMMMM RRRRRRR      RRRRRR

[hadoop@ip-172-31-2-68 ~]$
```

Installing Flink on EMR

Installing Flink is very easy once we have our EMR cluster ready. We need to do the following steps:

1. Download the Flink compatible to right Hadoop Version from link – `http://flink.apache.org/downloads.html`. I am downloading Flink compatible with Hadoop 2.7 version:

   ```
   wget http://www-eu.apache.org/dist/flink/flink-1.1.4/flink-
   1.1.4-bin-hadoop27-scala_2.11.tgz
   ```

2. Next, we need to un-tar the installer:

```
tar -xzf flink-1.1.4-bin-hadoop27-scala_2.11.tgz
```

3. And that is it, just go the un-tarred folder and set following environment variables and we are all set:

```
cd flink-1.1.4
export HADOOP_CONF_DIR=/etc/hadoop/conf
export YARN_CONF_DIR=/etc/hadoop/conf
```

Executing Flink on EMR-YARN

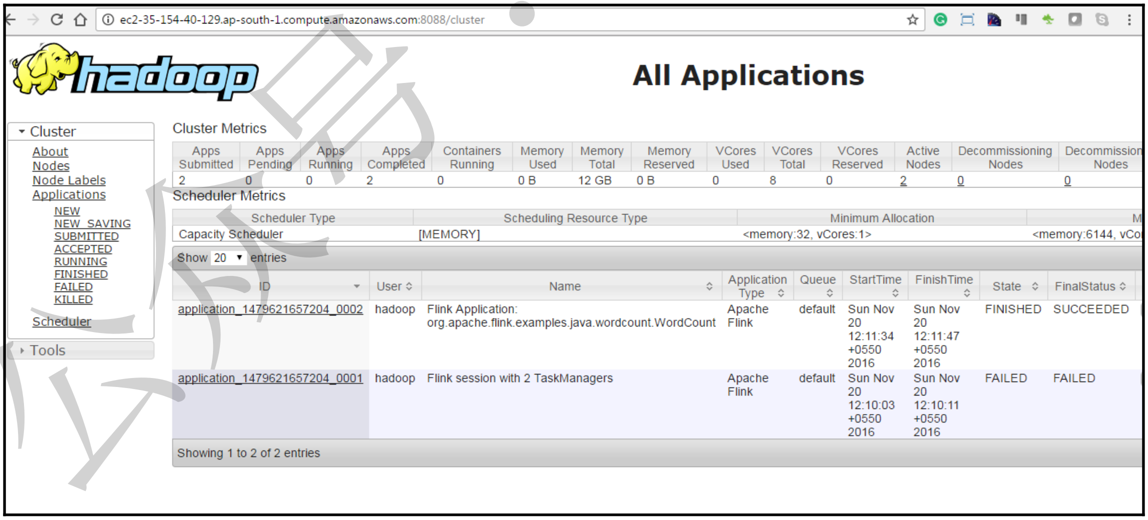
Executing Flink on YARN is very easy. We have already learnt the details on Flink on YARN in the previous chapter. The following steps shows a sample job execution. This would submit a single Flink job to YARN:

```
./bin/flink run -m yarn-cluster -yn 2
./examples/batch/WordCount.jar
```

You will see immediately Flink executing will start and on completion, you will see the word count results:

```
2016-11-20 06:41:45,760 INFO  org.apache.flink.yarn.YarnClusterClient
- Submitting job with JobID: 0004040e04879e432365825f50acc80c. Waiting for
job completion.
Submitting job with JobID: 0004040e04879e432365825f50acc80c. Waiting for
job completion.
Connected to JobManager at
Actor[akka.tcp://flink@172.31.0.221:46603/user/jobmanager#478604577]
11/20/2016 06:41:45     Job execution switched to status RUNNING.
11/20/2016 06:41:46     CHAIN DataSource (at
getDefaultTextLineDataSet(WordCountData.java:70)
(org.apache.flink.api.java.io.CollectionInputFormat)) -> FlatMap (FlatMap
at main(WordCount.java:80)) -> Combine(SUM(1), at
main(WordCount.java:83)(1/1) switched to RUNNING
11/20/2016 06:41:46     Reduce (SUM(1), at
getDefaultTextLineDataSet(WordCountData.java:70)
(org.apache.flink.api.java.io.CollectionInputFormat)) -> FlatMap (FlatMap
at main(WordCount.java:80)) -> Combine(SUM(1), at
main(WordCount.java:83)(1/1) switched to FINISHED
11/20/2016 06:41:46     Reduce (SUM(1), at main(WordCount.java:83)(1/2)
switched to DEPLOYING
11/20/2016 06:41:46     Reduce (SUM(1), at main(WordCount.java:83)(1/2)
switched to RUNNING
11/20/2016 06:41:46     Reduce (SUM(1), at main(WordCount.java:83)(2/2)
switched to RUNNING
1/20/2016 06:41:46     Reduce (SUM(1), at main(WordCount.java:83)(1/2)
switched to FINISHED
11/20/2016 06:41:46     DataSink (collect())(2/2) switched to DEPLOYING
11/20/2016 06:41:46     Reduce (SUM(1), at main(WordCount.java:83)(2/2)
switched to FINISHED
11/20/2016 06:41:46     DataSink (collect())(2/2) switched to RUNNING
11/20/2016 06:41:46     DataSink (collect())(2/2) switched to FINISHED
11/20/2016 06:41:46     Job execution switched to status FINISHED.
 (action,1)
 (after,1)
 (against,1)
 (and,12)
 (arms,1)
 (arrows,1)
 (awry,1)
 (ay,1)
 (bare,1)
 (be,4)
 (bodkin,1)
 (bourn,1)
 (calamity,1)
 (cast,1)
 (coil,1)
 (come,1)
```

We can also look at YARN cluster UI as shown in the following screenshot:



All Applications

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioning Nodes	Decommission Nodes
2	0	0	2	0	0 B	12 GB	0 B	0	8	0	2	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	M
Capacity Scheduler	[MEMORY]	<memory:32, vCores:1>	<memory:6144, vCo

Show 20 entries

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus
application_1479621657204_0002	hadoop	Flink Application: org.apache.flink.examples.java.wordcount.WordCount	Apache Flink	default	Sun Nov 20 12:11:34 +0550 2016	Sun Nov 20 12:11:47 +0550 2016	FINISHED	SUCCEEDED
application_1479621657204_0001	hadoop	Flink session with 2 TaskManagers	Apache Flink	default	Sun Nov 20 12:10:03 +0550 2016	Sun Nov 20 12:10:11 +0550 2016	FAILED	FAILED

Showing 1 to 2 of 2 entries

Starting a Flink YARN session

Alternatively we can also start a YARN session by blocking the resources which we have already seen in the previous chapter. A Flink YARN session will create a continuously running YARN session which can be used to execute multiple Flink jobs. This session keeps on running until we stop it.

To start the Flink YARN session, we need to execute the following command:

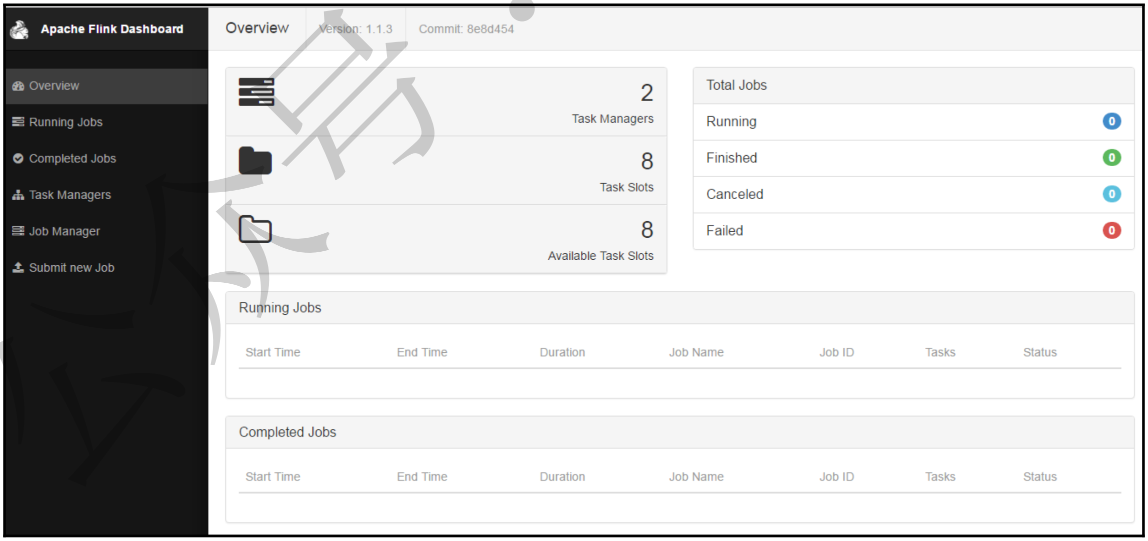
```
$ bin/yarn-session.sh -n 2 -tm 768 -s 4
```

Here we start two Task Managers with 768 MB memory each and 4 slots. You will see the YARN session ready as shown in the console logs:

```
2016-11-20 06:49:09,021 INFO  org.apache.flink.yarn.YarnClusterDescriptor
- Using values:
2016-11-20 06:49:09,023 INFO  org.apache.flink.yarn.YarnClusterDescriptor
-   TaskManager count = 22016-11-20 06:49:09,023 INFO
org.apache.flink.yarn.YarnClusterDescriptor
-   JobManager memory = 10242016-11-20 06:49:09,023 INFO
org.apache.flink.yarn.YarnClusterDescriptor
-   TaskManager memory = 768
2016-11-20 06:49:09,488 INFO
org.apache.hadoop.yarn.client.api.impl.TimelineClientImpl
- Timeline service address:
http://ip-172-31-2-68.ap-south-1.compute.internal:8188/ws/v1/timeline/
2016-11-20 06:49:09,613 INFO  org.apache.hadoop.yarn.client.RMProxy
- Connecting to ResourceManager at ip-172-31-2-68.ap-
south-1.compute.internal/172.31.2.68:8032
2016-11-20 06:49:10,309 WARN  org.apache.flink.yarn.YarnClusterDescriptor
- The configuration directory ('/home/hadoop/flink-1.1.3/conf') contains
both LOG4J and Logback configuration files. Please delete or rename one of
them.
2016-11-20 06:49:10,325 INFO  org.apache.flink.yarn.Utils
- Copying from file:/home/hadoop/flink-1.1.3/conf/log4j.properties to
hdfs://ip-172-31-2-68.ap-
south-1.compute.internal:8020/user/hadoop/.flink/application_1479621657204_
0004/log4j.properties
2016-11-20 06:49:10,558 INFO  org.apache.flink.yarn.Utils
- Copying from file:/home/hadoop/flink-1.1.3/lib to
hdfs://ip-172-31-2-68.ap-
south-1.compute.internal:8020/user/hadoop/.flink/application_1479621657204_
0004/lib
2016-11-20 06:49:12,392 INFO  org.apache.flink.yarn.Utils
- Copying from /home/hadoop/flink-1.1.3/conf/flink-conf.yaml to
hdfs://ip-172-31-2-68.ap-
south-1.compute.internal:8020/user/hadoop/.flink/application_1479621657204_
```

```
0004/flink-conf.yaml
2016-11-20 06:49:12,825 INFO  org.apache.flink.yarn.YarnClusterDescriptor
- Submitting application master application_1479621657204_0004
2016-11-20 06:49:12,893 INFO
org.apache.hadoop.yarn.client.api.impl.YarnClientImpl
- Submitted application application_1479621657204_0004
2016-11-20 06:49:12,893 INFO  org.apache.flink.yarn.YarnClusterDescriptor
- Waiting for the cluster to be allocated
2016-11-20 06:49:17,929 INFO  org.apache.flink.yarn.YarnClusterDescriptor
- YARN application has been deployed successfully.
Flink JobManager is now running on 172.31.0.220:45056
JobManager Web Interface:
http://ip-172-31-2-68.ap-south-1.compute.internal:20888/proxy/application_1
479621657204_0004/
2016-11-20 06:49:18,117 INFO  org.apache.flink.yarn.YarnClusterClient
- Starting client actor system.
2016-11-20 06:49:18,591 INFO  akka.event.slf4j.Slf4jLogger
- Slf4jLogger started
2016-11-20 06:49:18,671 INFO  Remoting
akka.tcp://flink@172.31.0.220:45056/user/jobmanager.
2016-11-20 06:49:19,343 INFO  org.apache.flink.yarn.ApplicationClient
- Successfully registered at the ResourceManager using JobManager
Actor[akka.tcp://flink@172.31.0.220:45056/user/jobmanager#1383364724]
Number of connected TaskManagers changed to 2. Slots available: 8
```

Here is a screenshot of the Flink Job Manager UI, where we can see two Task Managers and eight task slots:



Overview Version: 1.1.3 Commit: 8e8d454

2 Task Managers

8 Task Slots

8 Available Task Slots

Total Jobs	
Running	0
Finished	0
Canceled	0
Failed	0

Running Jobs

Start Time	End Time	Duration	Job Name	Job ID	Tasks	Status

Completed Jobs

Start Time	End Time	Duration	Job Name	Job ID	Tasks	Status

Executing Flink job on YARN session

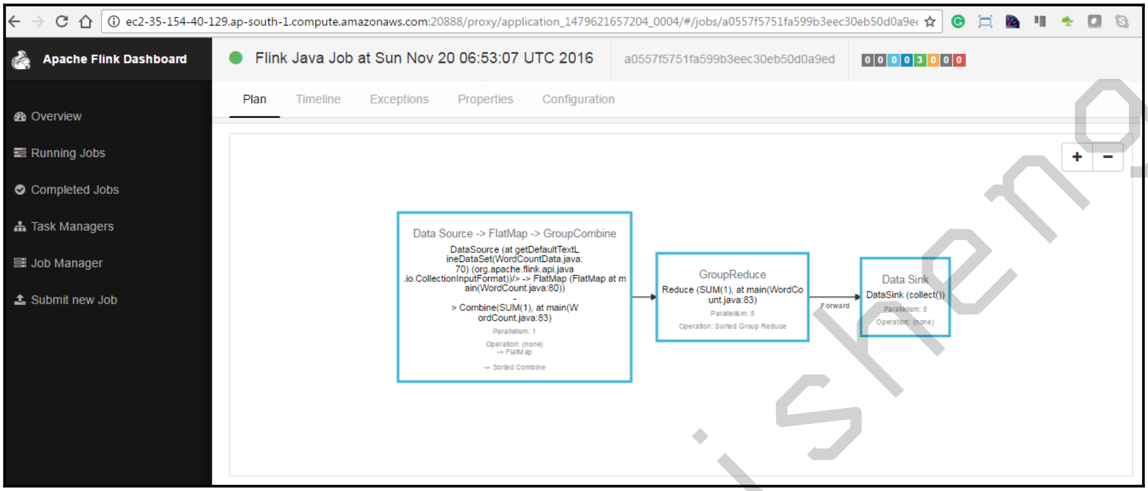
Now we can use this YARN session to submit Flink Jobs by executing the following command:

```
$./bin/flink run ./examples/batch/WordCount.jar
```

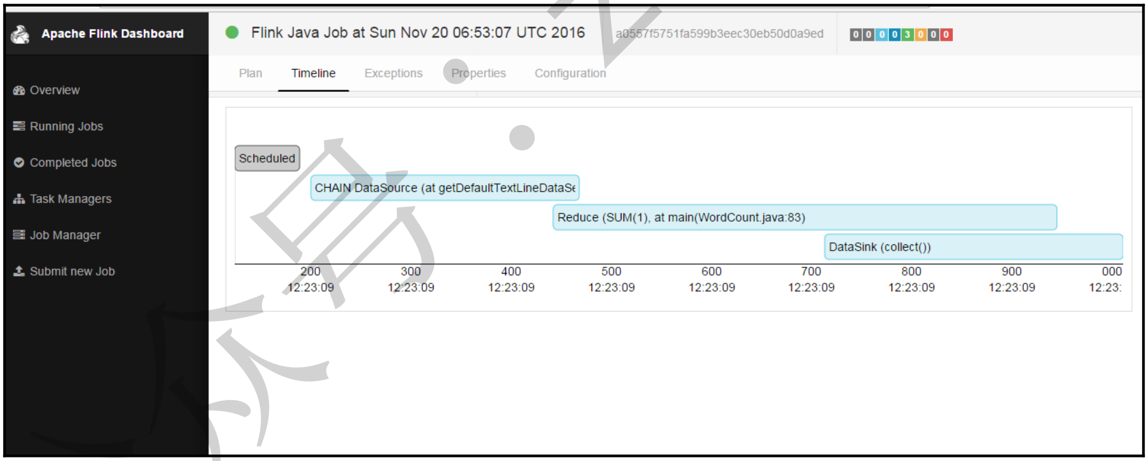
You will see word count job getting executed as shown in the following code:

```
2016-11-20 06:53:06,439 INFO  org.apache.flink.yarn.cli.FlinkYarnSessionCli
- Found YARN properties file /tmp/.yarn-properties-hadoop
2016-11-20 06:53:06,439 INFO  org.apache.flink.yarn.cli.FlinkYarnSessionCli
- Found YARN properties file /tmp/.yarn-properties-hadoop
Found YARN properties file /tmp/.yarn-properties-hadoop
2016-11-20 06:53:06,508 INFO  org.apache.flink.yarn.cli.FlinkYarnSessionCli
-
org.apache.flink.yarn.cli.FlinkYarnSessionCli
- YARN properties set default parallelism to 8
YARN properties set default parallelism to 8
2016-11-20 06:53:06,510 INFO  org.apache.flink.yarn.cli.FlinkYarnSessionCli
- Found YARN properties file /tmp/.yarn-properties-hadoop
2016-11-20 06:53:07,069 INFO
org.apache.hadoop.yarn.client.api.impl.TimelineClientImpl
- Timeline service address:
http://ip-172-31-2-68.ap-south-1.compute.internal:8188/ws/v1/timeline/
Executing WordCount example with default input data set.
Use --input to specify file input.
Printing result to stdout. Use --output to specify output path.
2016-11-20 06:53:07,728 INFO  org.apache.flink.yarn.YarnClusterClient
- Waiting until all TaskManagers have connected
Waiting until all TaskManagers have connected
2016-11-20 06:53:07,729 INFO  org.apache.flink.yarn.YarnClusterClient
Submitting job with JobID: a0557f5751fa599b3eec30eb50d0a9ed. Waiting for
job completion.
Connected to JobManager at
Actor[akka.tcp://flink@172.31.0.220:45056/user/jobmanager#1383364724]
11/20/2016 06:53:09     Job execution switched to status RUNNING.
11/20/2016 06:53:09     CHAIN DataSource (at
getDefaultTextLineDataSet(WordCountData.java:70)
(org.apache.flink.api.java.io.CollectionInputFormat)) -> FlatMap (FlatMap
at main(WordCount.java:80)) -> Combine(SUM(1), at
main(WordCount.java:83)(1/1) switched to SCHEDULED
11/20/2016 06:53:09     CHAIN DataSource (at
getDefaultTextLineDataSet(WordCountData.java:70)
(org.apache.flink.api.java.io.CollectionInputFormat)) -> FlatMap (FlatMap
at main(WordCount.java:80)) -> Combine(SUM(1), at
main(WordCount.java:83)(1/1) switched to DEPLOYING
11/20/2016 06:53:09     CHAIN DataSource (at
getDefaultTextLineDataSet(WordCountData.java:70)
(org.apache.flink.api.java.io.CollectionInputFormat)) -> FlatMap (FlatMap
at main(WordCount.java:80)) -> Combine(SUM(1), at
11/20/2016 06:53:10     DataSink (collect())(7/8) switched to FINISHED
11/20/2016 06:53:10     DataSink (collect())(8/8) switched to FINISHED
11/20/2016 06:53:10     Job execution switched to status FINISHED.
(bourn,1)
(coil,1)
(come,1)
(d,4)
(dread,1)
(is,3)
(long,1)
(make,2)
(more,1)
(must,1)
(no,2)
(oppressor,1)
(pangs,1)
(perchance,1)
(sicklied,1)
(something,1)
(takes,1)
(these,1)
(us,3)
(what,1)
Program execution finished
Job with JobID a0557f5751fa599b3eec30eb50d0a9ed has finished.
Job Runtime: 903 ms
Accumulator Results:
- f895985ab9d76c97aba23bc6689c7936 (java.util.ArrayList) [170 elements]
```

Here is a screenshot of the job execution details and task breakup:


ec2-35-154-40-129.ap-south-1.compute.amazonaws.com:20888/proxy/application_1479621657204_0004/#/jobs/a0557f5751fa599b3eec30eb50d0a9e
Apache Flink Dashboard
Flink Java Job at Sun Nov 20 06:53:07 UTC 2016
a0557f5751fa599b3eec30eb50d0a9ed
0 0 0 0 3 0 0 0
Plan
Timeline
Exceptions
Properties
Configuration
Overview
Running Jobs
Completed Jobs
Task Managers
Job Manager
Submit new Job
+
−
Data Source -> FlatMap -> GroupCombine
DataSource (at getDefaultTextLineDataSet(WordCountData.java:70) (org.apache.flink.api.java.io.CollectionInputFormat))/> -> FlatMap (FlatMap at main(WordCount.java:80)) -> Combine(SUM(1), at main(WordCount.java:83)
Parallelism: 1
Operation: (none) -> FlatMap -> Sorted Combine
GroupReduce
Reduce (SUM(1), at main(WordCount.java:83)
Parallelism: 8
Operation: Sorted Group Reduce
Forward
Data Sink
DataSink (collect())
Parallelism: 8
Operation: (none)

We can also see the timeline details on which all task were executed in parallel and which are in sequential manner. Here is screenshot of the same:



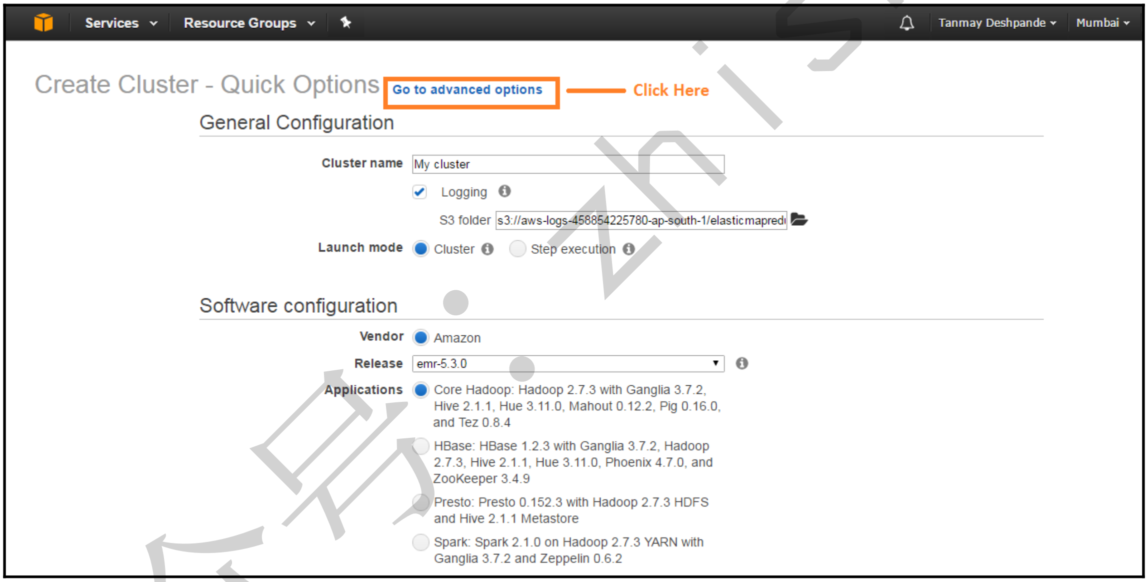
Shutting down the cluster

Once all our work is done, it is important to shut down the cluster. To do this, we again need to go to AWS console and click on the **Terminate** button.

Flink on EMR 5.3+

AWS has now started supporting Flink by default in its EMR cluster. In order to get that we have to follow these instructions.

First of all, we have to go to AWS EMR create cluster screen and then click on **Go to advanced options link** as heighted in the following screenshot:



Create Cluster - Quick Options

General Configuration

Cluster name: My cluster

☑ Logging

S3 folder: s3://aws-logs-458854225780-ap-south-1/elasticmapred

Launch mode: ● Cluster ○ Step execution

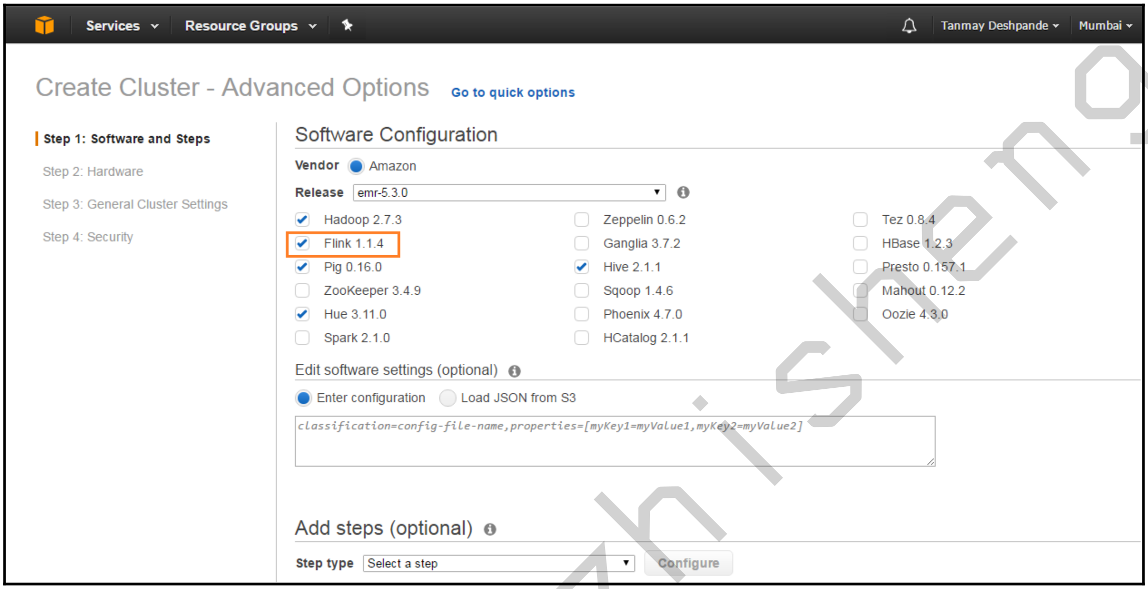
Software configuration

Vendor: ● Amazon

Release: emr-5.3.0

Applications:
- ● Core Hadoop: Hadoop 2.7.3 with Ganglia 3.7.2, Hive 2.1.1, Hue 3.11.0, Mahout 0.12.2, Pig 0.16.0, and Tez 0.8.4
- ○ HBase: HBase 1.2.3 with Ganglia 3.7.2, Hadoop 2.7.3, Hive 2.1.1, Hue 3.11.0, Phoenix 4.7.0, and ZooKeeper 3.4.9
- ○ Presto: Presto 0.152.3 with Hadoop 2.7.3 HDFS and Hive 2.1.1 Metastore
- ○ Spark: Spark 2.1.0 on Hadoop 2.7.3 YARN with Ganglia 3.7.2 and Zeppelin 0.6.2

Next you will have a screen which will all you to choose additional services you wish to have. There you need to check Flink 1.1.4:



Create Cluster - Advanced Options   Go to quick options

Step 1: Software and Steps
Step 2: Hardware
Step 3: General Cluster Settings
Step 4: Security

Software Configuration

Vendor ● Amazon
Release emr-5.3.0

☑ Hadoop 2.7.3   □ Zeppelin 0.6.2   □ Tez 0.8.4
☑ Flink 1.1.4   □ Ganglia 3.7.2   □ HBase 1.2.3
☑ Pig 0.16.0   ☑ Hive 2.1.1   □ Presto 0.157.1
□ ZooKeeper 3.4.9   □ Sqoop 1.4.6   □ Mahout 0.12.2
☑ Hue 3.11.0   □ Phoenix 4.7.0   □ Oozie 4.3.0
□ Spark 2.1.0   □ HCatalog 2.1.1

Edit software settings (optional)
● Enter configuration ○ Load JSON from S3

```
classification=config-file-name,properties=[myKey1=myValue1,myKey2=myValue2]
```

Add steps (optional)
Step type Select a step   Configure

And then click on the **Next** button to continue the rest of the setup. The remaining steps would be same as we saw in the previous sections. Once the cluster is up and running, you are all set to use Flink directly.

Using S3 in Flink applications

Amazon Simple Storage Service (**S3**) is a Software-as-a-Service provided by AWS to store in the AWS Cloud. Many companies use S3 for cheap data storage. It is a hosted filesystem as a service. S3 can be used as alternative to the HDFS. One can think of using S3 over HDFS if he/she does not want to invest in complete Hadoop cluster. Flink provides you API to allow reading data stored on S3.

We can use S3 objects like simple files. The following code snippet shows how to use S3 object in Flink:

```
// Read data from S3 bucket
env.readTextFile("s3://<bucket>/<endpoint>");

// Write data to S3 bucket
stream.writeAsText("s3://<bucket>/<endpoint>");

// Use S3 as FsStatebackend
env.setStateBackend(new FsStateBackend("s3://<your-bucket>/<endpoint>"));
```

S3 is treated like any other filesystem by Flink. It uses S3 client for Hadoop.

To access S3 objects, Flink needs authentication. This can be provided by using AWS IAM service. This method helps maintaining the security as we don't need to distribute the access and secret keys.

Summary

In this chapter, we learnt how we can deploy Flink on AWS and GCP. This is very handy for faster deployments and installations. We can spawn and delete Flink cluster with minimum efforts.

In the next chapter, we are going to learn about the best practice one should follow in order to efficiently use Flink.

10
Best Practices

So far in this book, we have learned various things about Flink. We started with Flink's architecture and the various APIs it supports. We also learned how we use graph and machine learning APIs provided by Flink. Now in this concluding chapter, we are going to talk about some best practices you should follow in order to create production quality maintainable Flink applications.

We will be discussing about the following topics:

- Logging best practices
- Using custom serializers
- Using and monitoring the REST API
- Back pressure monitoring

So let's get started.

Logging best practices

It is very important to have logs configured in any software application. Logs help in debugging the issues. We don’t follow these logging practices, it would be very diffuclt to understand the progress of the job or if any issues with it. There are couple of libraries we can use for better logging experience.

Configuring Log4j

Log4j, as we know, is one the most widely used logging libraries. We can configure it in any Flink application with very little effort. We have only to include a `log4j.properties` file. We can pass the `log4j.properties` file by passing it as an `Dlog4j.configuration=/path/to/log4j.properties` argument.

Flink supports the following default property files:

- `log4j-cli.properties`: This file is used by the Flink command line tool. Here is the exact file at `https://github.com/apache/flink/blob/master/flink-dist/src/main/flink-bin/conf/log4j-cli.properties`.
- `log4j-yarn-session.properties`: This file is used by the Flink YARN session. Here is the exact file at `https://github.com/apache/flink/blob/master/flink-dist/src/main/flink-bin/conf/log4j-yarn-session.properties`.
- `log4j.properties`: This file is used by the Flink Job Manager and Task Manager. Here is the exact file at `https://github.com/apache/flink/blob/master/flink-dist/src/main/flink-bin/conf/log4j.properties`.

Configuring Logback

These days a lot of people prefer using Logback over Log4j because of it features. Logback provides faster I/O, thoroughly tested libraries, extensive documentation etc. Flink also supports configuring Logback for an application.

We need to use the same property to configure `logback.xml`. `Dlogback.configurationFile=<file>`, or we can also put the `logback.xml` file in the class path. A sample `logback.xml` would look like this:

```
<configuration>
    <appender name="file" class="ch.qos.logback.core.FileAppender">
        <file>${log.file}</file>
        <append>false</append>
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level
            %logger{60} %X{sourceThread} - %msg%n</pattern>
        </encoder>
    </appender>

    <!-- This affects logging for both user code and Flink -->
    <root level="INFO">
        <appender-ref ref="file"/>
    </root>

    <!-- Uncomment this if you want to only change Flink's logging -->
    <!--<logger name="org.apache.flink" level="INFO">-->
        <!--<appender-ref ref="file"/>-->
    <!--</logger>-->

    <!-- The following lines keep the log level of common
    libraries/connectors on
         log level INFO. The root logger does not override this. You
         have to manually
         change the log levels here. -->
    <logger name="akka" level="INFO">
        <appender-ref ref="file"/>
    </logger>
    <logger name="org.apache.kafka" level="INFO">
        <appender-ref ref="file"/>
    </logger>
    <logger name="org.apache.hadoop" level="INFO">
        <appender-ref ref="file"/>
    </logger>
    <logger name="org.apache.zookeeper" level="INFO">
        <appender-ref ref="file"/>
    </logger>

    <!-- Suppress the irrelevant (wrong) warnings from the Netty
     channel handler -->
    <logger name="org.jboss.netty.channel.DefaultChannelPipeline"
    level="ERROR">
        <appender-ref ref="file"/>
    </logger>
</configuration>
```

We can always change the `logback.xml` file and set the logging level according to our preferences.

Logging in applications

While using SLF4J in any Flink application, we need to import the following package and classes, and initiate the logger with the class name:

```
import org.slf4j.LoggerFactory
import org.slf4j.Logger

Logger LOG = LoggerFactory.getLogger(MyClass.class)
```

It is also a best practice to use a placeholder mechanism for logging instead of using a string formatter. The placeholder mechanism helps to avoid unnecessary string formations instead it only does string concatenation. The following code snippet shows how to use a placeholder:

```
LOG.info("Value of a = {}, value of b= {}", myobject.a, myobject.b);
```

We can also use placeholder logging in exception handling:

```
catch(Exception e){
  LOG.error("Error occurred {}",  e);
}
```

Using ParameterTool

Since Flink 0.9, we have a built-in `ParameterTool` in Flink, which helps to get parameters from external sources such as arguments, system properties, or from property files. Internally, it is a map of strings which keeps the key as the parameter name and the value as the parameter value.

For example, we can think of using ParameterTool in our DataStream API example, where we need to set Kafka properties:

```
String kafkaproperties = "/path/to/kafka.properties";
ParameterTool parameter = ParameterTool.fromPropertiesFile(propertiesFile);
```

From system properties

We can read properties defined in system variables. We need to pass the system properties file before initializing them by setting `Dinput=hdfs://myfile`.

Now we can read all those properties in `ParameterTool` as follows:

```
ParameterTool parameters = ParameterTool.fromSystemProperties();
```

From command line arguments

We can also read the parameters from command line arguments. We have to set `--elements` before invoking the application.

The following code shows how to read parameters from command line arguments:

```
ParameterTool parameters = ParameterTool.fromArgs(args);
```

From .properties file

We can also read the parameters from the `.properties` file. The following is the code for this:

```
String propertiesFile = /my.properties";
ParameterTool parameters =
ParameterTool.fromPropertiesFile(propertiesFile);
```

We can read the parameters in the Flink program. The following shows how we get the parameters:

```
parameter.getRequired("key");
parameter.get("paramterName", "myDefaultValue");
parameter.getLong("expectedCount", -1L);
parameter.getNumberOfParameters()
```

Naming large TupleX types

As we know, a tuple is a complex data type used to represent complex data structures. It is a combination of various primitive data types. Generally, it is recommended not to use large tuples; instead it is recommended to use Java POJOs. If you want to use a tuple, it is recommended to name it with some custom POJO type.

It is very easy to create a custom type for a large tuple. For example, if we want to use `Tuple8` then we can define it as follows:

```
//Initiate Record Tuple
RecordTuple rc = new RecordTuple(value0, value1, value2, value3, value4,
value5, value6, value7);

// Define RecordTuple instead of using Tuple8
public static class RecordTuple extends Tuple8<String, String, Integer,
String, Integer, Integer, Integer, Integer> {
```

```
            public RecordTuple() {
                  super();
            }

            public RecordTuple(String value0, String value1, Integer value2,
String value3, Integer value4, Integer value5,
                        Integer value6, Integer value7) {
                  super(value0, value1, value2, value3, value4, value5,
value6, value7);
            }
         }
```

Registering a custom serializer

In the distributed computing world, it is very important to take care of each and every small thing. Serialization is one of them. By default, Flink uses the Kryo serializer. Flink also allows us to write custom serializers in case you think the default one is not good enough. We need to register the custom serializer in order for Flink to understand it. Registering the custom serializer is very simple; we just need to register its class type in the Flink execution environment. The following code snippet shows how we do that:

```
final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();

// register the class of the serializer as serializer for a type
env.getConfig().registerTypeWithKryoSerializer(MyCustomType.class,
MyCustomSerializer.class);

// register an instance as serializer for a type
MySerializer mySerializer = new MySerializer();
env.getConfig().registerTypeWithKryoSerializer(MyCustomType.class,
mySerializer);
```

Here is a complete sample class for Custom Serializer at `https://github.com/deshpandetanmay/mastering-flink/blob/master/chapter10/flink-batch-adv/src/main/java/com/demo/flink/batch/RecordSerializer.java`.

And the custom type at `https://github.com/deshpandetanmay/mastering-flink/blob/master/chapter10/flink-batch-adv/src/main/java/com/demo/flink/batch/Record.java`.

We need to make sure that the custom serializer has to extend the Kryo's serializer class. With Google Protobuf and Apache Thrift, this has been done already.

> You can read more about Google Protobuf at `https://github.com/google/protobuf`.
> Details on Apache Thrift can be read at `https://thrift.apache.org/`.

In order to use Google Protobuf, you can add the following Maven dependency:

```
<dependency>
  <groupId>com.twitter</groupId>
  <artifactId>chill-protobuf</artifactId>
  <version>0.5.2</version>
</dependency>
<dependency>
  <groupId>com.google.protobuf</groupId>
  <artifactId>protobuf-java</artifactId>
  <version>2.5.0</version>
</dependency>
```

Metrics

Flink supports a metrics system which allows users to know more about the Flink setup and the applications running on it. This would be very useful if you are using Flink in a very big production system where a huge number of jobs are running and we need to get details of each. We can also use these to feed external monitoring systems. So let's try to understand what is available and how to use them.

Registering metrics

Metric functions are available for use from any user function which extends `RichFunction` by calling `getRuntimeContext().getMetricGroup()`. These methods return a `MetricGroup` object, which can be used to create and register a new metric.

Flink supports various metrics types, such as:

- Counters
- Gauges

- Histograms
- Meters

Counters

A counter can be used to count certain things while processing. A simple use of a counter can be to count invalid records in the data. You can choose to either increment or decrement the counter, based on the conditions. The following code snippet shows this:

```
public class TestMapper extends RichMapFunction<String, Integer> {
  private Counter errorCounter;

  @Override
  public void open(Configuration config) {
    this.errorCounter = getRuntimeContext()
      .getMetricGroup()
      .counter("errorCounter");
  }

  @public Integer map(String value) throws Exception {
    this.errorCounter.inc();
  }
}
```

Gauges

A gauge can provide any value whenever required. In order to use a gauge, first we need to create a class that implements `org.apache.flink.metrics.Gauge`. Later, you can register that with `MetricGroup`.

The following code snippet shows the use of a gauge in the Flink application:

```
public class TestMapper extends RichMapFunction<String, Integer> {
  private int valueToExpose;

  @Override
  public void open(Configuration config) {
    getRuntimeContext()
      .getMetricGroup()
      .gauge("MyGauge", new Gauge<Integer>() {
        @Override
        public Integer getValue() {
          return valueToReturn;
        }
      });
  }
}
```

Histograms

A histogram provides for the distribution of long values over a metric. This can be used to monitor certain metrics over time. The following code snippet shows how to use this:

```
public class TestMapper extends RichMapFunction<Long, Integer> {
  private Histogram histogram;

  @Override
  public void open(Configuration config) {
    this.histogram = getRuntimeContext()
      .getMetricGroup()
      .histogram("myHistogram", new MyHistogram());
  }

  @public Integer map(Long value) throws Exception {
    this.histogram.update(value);
  }
}
```

Meters

A meter is used for monitoring a specific parameter's average throughput. The occurrence of an event is registered using the `markEvent()` method. We can register a meter using the `meter(String name, Meter meter)` method on `MeterGroup`:

```
public class MyMapper extends RichMapFunction<Long, Integer> {
  private Meter meter;

  @Override
  public void open(Configuration config) {
    this.meter = getRuntimeContext()
      .getMetricGroup()
      .meter("myMeter", new MyMeter());
  }

  @public Integer map(Long value) throws Exception {
    this.meter.markEvent();
  }
}
```

Reporters

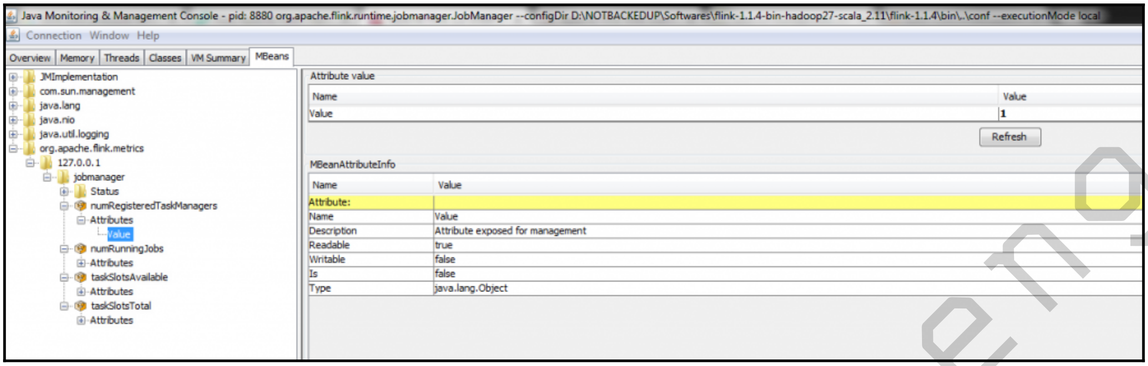
Metrics can be displayed to the external system by configuring one or more reporters in the `conf/flink-conf.yaml` file. Most of you might be aware of systems such as JMX, which help in monitoring many systems. We can consider configuring JMX reporting in Flink. A reporter should have certain properties, as listed in the following table:

Configuration	**Description**
`metrics.reporters`	The list of named reporters
`metrics.reporter.<name>.<config>`	Configuration for reporter with `<name>`
`metrics.reporter.<name>.class`	Reporter class used for reporter named `<name>`
`metrics.reporter.<name>.interval`	Interval time for reporter with name `<name>`
`metrics.reporter.<name>.scope.delimiter`	Scope of reporter with name `<name>`

The following is an example of a reported configuration for the JMX reporter:

```
metrics.reporters: my_jmx_reporter

metrics.reporter.my_jmx_reporter.class:
org.apache.flink.metrics.jmx.JMXReporter
metrics.reporter.my_jmx_reporter.port: 9020-9040
```

Once we add the preceding given configuration in `config/flink-conf.yaml`, we need to start Flink Job Manager process. Now Flink will start exposing these variables to JMX port `8789`. We can use JConsole to monitor the reports published by Flink. JConsole comes by default with JDK installation. We just need to go to JDK installation directory and start `JConsole.exe`. Once the JConsole is running, we need to select the Flink Job Manager process to monitor and we can see various values that can be monitored. Following is a sample screenshot of a JConsole screen monitoring Flink.



Java Monitoring & Management Console - pid: 8880 org.apache.flink.runtime.jobmanager.JobManager --configDir D:\NOTBACKEDUP\Softwares\flink-1.1.4-bin-hadoop27-scala_2.11\flink-1.1.4\bin\..\conf --executionMode local

Name	Value
Value	1

Name	Value
Attribute:	
Name	Value
Description	Attribute exposed for management
Readable	true
Writable	false
Is	false
Type	java.lang.Object

Apart from JMX, Flink supports reporters such as Ganglia, Graphite and StasD. More information on those can be found at `https://ci.apache.org/projects/flink/flink-docs-release-1.2/monitoring/metrics.html#reporter`.

Monitoring REST API

Flink supports the monitoring of the status of running and completed apps. These APIs are also used by Flink's own job dashboard. The status APIs support the `get` method which returns JSON objects giving information of the job. Currently, monitoring APIs is by default started within the Flink Job Manager dashboard. This information can also be accessed with Job Manager Dashboard.

There are many APIs available in Flink. Let's start understanding some of them.

Config API

This gives configuration details of the API:

```
http://localhost:8081/config
```

The following is the response:

```
{
    "refresh-interval": 3000,
    "timezone-offset": 19800000,
    "timezone-name": "India Standard Time",
    "flink-version": "1.0.3",
    "flink-revision": "f3a6b5f @ 06.05.2016 @ 12:58:02 UTC"
}
```

Overview API

This gives an overview of the Flink cluster:
`http://localhost:8081/overview`

The following is the response:

```
{
    "taskmanagers": 1,
    "slots-total": 1,
    "slots-available": 1,
    "jobs-running": 0,
    "jobs-finished": 1,
    "jobs-cancelled": 0,
    "jobs-failed": 0,
    "flink-version": "1.0.3",
    "flink-commit": "f3a6b5f"
}
```

Overview of the jobs

This gives an overview of the jobs which have run recently and are currently running:
`http://localhost:8081/jobs`

The following is the response:

```
{
    "jobs-running": [],
    "jobs-finished": [
        "cd978489f5e76e5988fa0e5a7c76c09b"
    ],
    "jobs-cancelled": [],
    "jobs-failed": []
}
```

`http://localhost:8081/joboverview` API gives the complete overview of a Flink job. It contains job ID, start and end times, duration of run, no. of tasks and their states. A state could be started, running, killed or finished.

The following is the response:

```
{
    "running": [],
    "finished": [
        {
            "jid": "cd978489f5e76e5988fa0e5a7c76c09b",
            "name": "Flink Java Job at Sun Dec 04 16:13:16 IST 2016",
            "state": "FINISHED",
            "start-time": 1480848197679,
            "end-time": 1480848198310,
            "duration": 631,
            "last-modification": 1480848198310,
            "tasks": {
                "total": 3,
                "pending": 0,
                "running": 0,
                "finished": 3,
                "canceling": 0,
                "canceled": 0,
                "failed": 0
            }
        }
    ]
}
```

Details of a specific job

This gives details of the specific job. We need to provide the job ID returned by the previous API. When a job is submitted, Flink creates a Directed Acyclic Job (DAG) for that job. This graph contains vertices as the tasks of the job and the execution plan. Following output shows the same details.
`http://localhost:8081/jobs/<jobid>`

The following is the response:

```
{
    "jid": "cd978489f5e76e5988fa0e5a7c76c09b",
    "name": "Flink Java Job at Sun Dec 04 16:13:16 IST 2016",
    "isStoppable": false,
    "state": "FINISHED",
    "start-time": 1480848197679,
    "end-time": 1480848198310,
    "duration": 631,
    "now": 1480849319207,
    "timestamps": {
        "CREATED": 1480848197679,
        "RUNNING": 1480848197733,
        "FAILING": 0,
        "FAILED": 0,
        "CANCELLING": 0,
        "CANCELED": 0,
        "FINISHED": 1480848198310,
        "RESTARTING": 0
    },
    "vertices": [
        {
            "id": "f590afd023018e19e30ce3cd7a16f4b1",
            "name": "CHAIN DataSource (at
             getDefaultTextLineDataSet(WordCountData.java:70)
             (org.apache.flink.api.java.io.CollectionInputFormat)) ->
             FlatMap (FlatMap at main(WordCount.java:81)) ->
             Combine(SUM(1), at main(WordCount.java:84)",
            "parallelism": 1,
            "status": "FINISHED",
            "start-time": 1480848197744,
            "end-time": 1480848198061,
            "duration": 317,
            "tasks": {
                "CREATED": 0,
                "SCHEDULED": 0,
                "DEPLOYING": 0,
                "RUNNING": 0,
                "FINISHED": 1,
                "CANCELING": 0,
                "CANCELED": 0,
                "FAILED": 0
            },
            "metrics": {
                "read-bytes": 0,
                "write-bytes": 1696,
                "read-records": 0,
                "write-records": 170
            }
        },
        {
            "id": "c48c21be9c7bf6b5701cfa4534346f2f",
            "name": "Reduce (SUM(1), at main(WordCount.java:84)",
            "parallelism": 1,
            "status": "FINISHED",
            "start-time": 1480848198034,
            "end-time": 1480848198190,
            "duration": 156,
            "tasks": {
                "CREATED": 0,
                "SCHEDULED": 0,
                "DEPLOYING": 0,
                "RUNNING": 0,
                "FINISHED": 1,
                "CANCELING": 0,
                "CANCELED": 0,
                "FAILED": 0
            },
            "metrics": {
                "read-bytes": 1696,
                "write-bytes": 1696,
                "read-records": 170,
                "write-records": 170
            }
        },
        {
            "id": "ff4625cfad1f2540bd08b99fb447e6c2",
            "name": "DataSink (collect())",
            "parallelism": 1,
            "status": "FINISHED",
            "start-time": 1480848198184,
            "end-time": 1480848198269,
            "duration": 85,
            "tasks": {
                "CREATED": 0,
                "SCHEDULED": 0,
                "DEPLOYING": 0,
                "RUNNING": 0,
                "FINISHED": 1,
                "CANCELING": 0,
                "CANCELED": 0,
                "FAILED": 0
            },
            "metrics": {
                "read-bytes": 1696,
                "write-bytes": 0,
                "read-records": 170,
                "write-records": 0
            }
        }
    ],
    "status-counts": {
        "CREATED": 0,
        "SCHEDULED": 0,
        "DEPLOYING": 0,
        "RUNNING": 0,
        "FINISHED": 3,
        "CANCELING": 0,
        "CANCELED": 0,
        "FAILED": 0
    },
    "plan": {
//plan details
    }
}
```

User defined job configuration

This gives the user defined job configuration used by a specific job:

```
http://localhost:8081/jobs/<jobid>/config
```

The following is the response:

```
    {
        "jid": "cd978489f5e76e5988fa0e5a7c76c09b",
        "name": "Flink Java Job at Sun Dec 04 16:13:16 IST 2016",
        "execution-config": {
            "execution-mode": "PIPELINED",
            "restart-strategy": "default",
            "job-parallelism": -1,
            "object-reuse-mode": false,
            "user-config": {}
        }
    }
```

Similarly, you can explore all the following listed APIs on your own setup:

```
    /config
    /overview
    /jobs
    /joboverview/running
    /joboverview/completed
    /jobs/<jobid>
    /jobs/<jobid>/vertices
    /jobs/<jobid>/config
    /jobs/<jobid>/exceptions
    /jobs/<jobid>/accumulators
    /jobs/<jobid>/vertices/<vertexid>
```

```
/jobs/<jobid>/vertices/<vertexid>/subtasktimes
/jobs/<jobid>/vertices/<vertexid>/taskmanagers
/jobs/<jobid>/vertices/<vertexid>/accumulators
/jobs/<jobid>/vertices/<vertexid>/subtasks/accumulators
/jobs/<jobid>/vertices/<vertexid>/subtasks/<subtasknum>
/jobs/<jobid>/vertices/<vertexid>/subtasks/<subtasknum>/attempts/<attempt>
/jobs/<jobid>/vertices/<vertexid>/subtasks/<subtasknum>/attempts/<attempt>/
accumulators
/jobs/<jobid>/plan
```

Back pressure monitoring

Back pressure is a special situation in Flink applications where the downstream operators are not able to consume data with the same speed of the upstream operator that is pushing the data. This starts building pressure on the pipeline and the data flow starts in the opposite direction. Generally, if this happens, Flink gives us warnings in the logs.

In a source sink scenario, if we see a warning to the source, then it means sink is consuming data slower than the source is producing it.

It is very important to monitor back pressure in all streaming jobs, as a high back pressuring job may fail or give the wrong results. The backpressure can be monitored from the Flink dashboard.

Flink handles back pressure monitoring continuously, taking sample stack traces of the running tasks. If the sample shows that the task is stuck in an internal method, this indicates that there is a back pressure.

On an average, the Job Manager triggers 100 stack traces every 50 milliseconds. Based on the number of tasks stuck in the internal process, the back pressure warning level is decided, as shown in the following table:

Ratio	**Back pressure level**
0 to 0.10	ok
0.10 to 0.5	low
0.5 to 1	high

You can also configure the number of samples and their intervals by setting the following parameters:

Parameter	Description
`jobmanager.web.backpressure.refresh-interval`	Refresh interval to reset available stats. Default is `60,000`, 1 min.
`jobmanager.web.backpressure.delay-between-samples`	Interval for delay between the samples. Default is `50` ms.
`jobmanager.web.backpressure.num-samples`	Number of samples to determine the back pressure. Default is `100`.

Summary

In this final chapter, we looked at some best practices you should follow in order to achieve the best of Flink's performance. We also looked at various monitoring APIs and metrics which can be used for the detailed monitoring of Flink applications.

For Flink, I would say the journey has just started and I am sure over the years, the community and support is going to get stronger and better. After all, Flink is called the **fourth generation** (**4G**) of big data!

Index