ALMA MATER STUDIORUM
UNIVERSITY OF BOLOGNA

The power of Apache Flink

Apache Flink as the fastest growing engine in the Big Data ecosystem

Distributed Systems
Master Degree in Computer Science

Sokol Guri (*sokol.guri@studio.unibo.it*)

AA 2019-2020
Cesena

Abstract

According to a recent report by IBM Marketing cloud, "90 per cent of the data in the world today has been created in the last two years alone, creating 2.5 quintillion bytes of data every day and with new devices, sensors and technologies emerging, the data growth rate will likely accelerate even more". The amount of data is growing significantly over the past few years, therefore, the need for distributed data processing frameworks is growing.

It all started back in 2011 when the first version of Apache Hadoop was released. Here is where Distributed Data processing frameworks come into play. Apache Flink (released in March 2016) is a new face in the field of distributed data processing. Flink has been considered as the next distributed data processing revolution.

Apache Flink is a top-level Apache project that allows unifying distributed stream and batch processing. Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments, perform computations at in-memory speed and any scale. In the core of Apache Flink is a streaming data-flow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams.

Contents

Chapter 1

Introduction

In daily life, we continuously interact with an enormous quantity of information, coming from a different source in different ways. Handling, scaling and analyzing this data quantity until nowadays has been done through various methods, without having a truly effective and standard approach. In the last year, people have begun to work with data at very large scare across a wide variety of sectors. Systems should fulfil the exact consistency and certain knowledge of the order of events and these were and remain, the most important challenges to getting better at.

The requirement for real-time response is high in many scenarios as a trading system, fraud analysis system, delivery system, monitoring system or social networks interactions. These data sources are continuous and the new data arrival must be processed immediately, otherwise, the processing will never end. For this reason, highly scalable streaming computing solutions are needed to prevent serious consequences in case of not doing streaming well.

Among the numerous open-source frameworks developed for Big Data, one of the most important and known is Hadoop[1]. It is a distributed file system allows concurrent processing and fault tolerance. It allows to analyze enormous data quantity set distributed to a cluster of machines. This approach may be ideal for processing data block, but this solution does not perform well working with real-time data, introducing high latency.

To find a fair and stable relation between continuous data production and data consumption, was developed Apache Storm[2]. It made possible to

[1]Apache Hadoop doc: `https://hadoop.apache.org/docs/stable/`

[2]Apache Storm doc: `http://storm.apache.org/releases/current/index.html`

process streams with very low latency, but high throughput was hard to achieve. Storm did not prove the level of correctness that often is needed.

Apache Spark[3] was the next fault-tolerant streaming architecture based on the idea of micro-batching. To overcome the complexity and overhead that come from processing and buffering records, a continuous computation is broken down in a series of small batch jobs called micro-batches. These small batches may either succeed or fail.

The previous framework examples and many others, evince that is hard to define a standard solution approach for handling the real-time data streaming and working with real-time data involves tradeoffs between high throughput, low-latency, consistency and accurate state of events.

"Than why not Flink?" Removing many of these tradeoffs and reducing the complexity that has been faced by other data-driven engines, Apache Flink[4] is considered one of the most powerful data streaming engines. Though Flink is compatible with Hadoop, still it is the most demanding technology now because of the following main features of Flink.

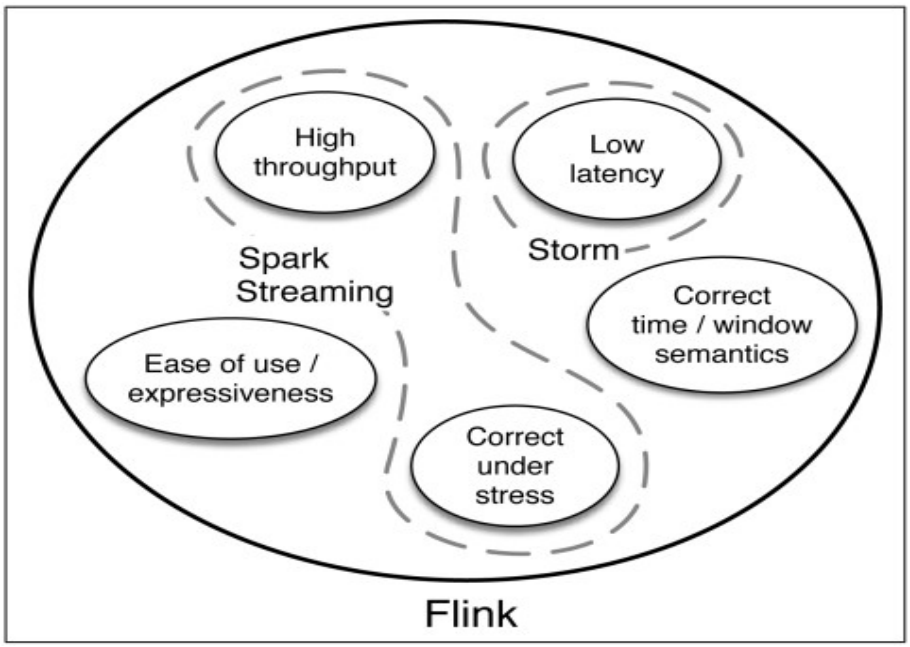

Figure 1.1: Flink combining tradeoffs[5]

[3]Apache Spark doc: `https://spark.apache.org/documentation.html`
[4]Flink: `https://ci.apache.org/projects/flink/flink-docs-stable/`
[5]Source: [9], Figure 1.2

1.1 Goals

This project aims to reveal through theoretical and practical explanation the most important features and advantages Apache Flink provides. The central point of the theoretical side is to focus on the mechanisms that differentiate Apache Flink from other technologies and how Apache Flink implements these mechanisms. The practical approach will concentrate to design and implement a basic event-driven application with multiple testing entry-points, to demonstrate pragmatically the usage of Flink DataStream API. The practical approach includes also testing scenarios that aim to verify the correctness of Flink's mechanisms. The objectives set for this project are the following:

- The theoretical approach of the project will focus to study the Flink's architecture and describe how its main components interact to execute applications. Through basic examples will be highlighted the most important components and their function.
- The project aims to point out the most important mechanisms and features that differentiate Apache Flink from other technologies on the market that are used to resolve and handle the challenges of distributed systems: fault-tolerance, low-latency, etc.
- Since its very early days, Apache Flink has followed the philosophy of taking a unified approach to batch and streaming. The core building block is the "*continuous processing of unbounded data streams, with batch as a special, bounded set of those streams.*" For this reason, the project will concentrate on streaming Processor and the **Data Streaming API**[6] Flink offers. Firstly a comprehensive overview of the DataStream API will be exposed and analyzed, and in the second part will be used in practical scenarios.
- The project will include a "*quick roundup*" regarding the Flink Command Line interface with usage explanation and various examples.
- Regarding the practical approach, to test the Flink's main streaming APIs, will be implemented event-driven applications to consume tweets from Twitter. They will ingest events from one or more event streams and reacts to incoming events by triggering computations and operations. These application will point out the most important features focusing on **Data Stream API**.

[6] API or Application Programming Interface is a set of programming code that enables data transmission between one software product and another.

- Other than real-time tweets ingestion, will be implemented some testing scenarios to test how Flink handles out-of-order event for a particular type of window. The project aims to test also how Flink will handle failure scenarios by randomly removing or killing available resources and analyzing the environment state reaction.

- For more efficient and rich documentation, we aim to use external technologies, other than Flink's metric feature, to visualize environment and resources state during the DataStream and other scenarios testing.

- The basic application will consist of different entry points, indicating the testing scenarios. Associating the documentation of these tests with representative charts[7] will help to archive a good performance valuation.

- After studying and practicing comprehensively Apache Flink features, the most important advantage and disadvantages of this technology will be pointed out. To conclude the project, a confrontation of *pros* and *cons* of this technology will give as a complete panorama of the main characteristics of Flink data-streaming framework.

[7]Representative charts include all the visual elements used to represent data analysis: graphs, diagrams, trees, etc.

Chapter 2

Background

In this chapter will take a closer look at how the Apache Flink architecture is organized and will explain in detail the most important Apache Flink functionalities. An important section of the chapter is dedicated to DataStream API explanation.

2.1 Architecture

Flink's architecture consists of various components and each component is part of a specific layer built one on top of the other. The most important components are: *deploy, core processing, APIs & libraries.*

Flink is designed to **run** in a cluster, as a YARN[1] application or in a Mesos[2] cluster within a single machine. The core of Flink is the **distributed dataflow engine** which has the function to execute dataflow programs. It can be seen as a streaming dataflow engine and both the DataSet and DataStream API create runtime programs executable by the engine. Flink offers a developer-friendly API that layers on top of the runtime and generates streaming dataflow programs. The **DataStream** API is used for stream processing and **DataSet** API for batch processing. DataStream is a fluent API for defining analysis on infinite streams and it is available for Java or Scala. On top of this layer, there are the domain-specific **libraries** that generate DataSet and DataStream programs, FlinkML for machine learning, Table for SQL-like operations (see Fig 2.1).

[1] a system for managing distributed application

[2] an open-source project to manage computer clusters

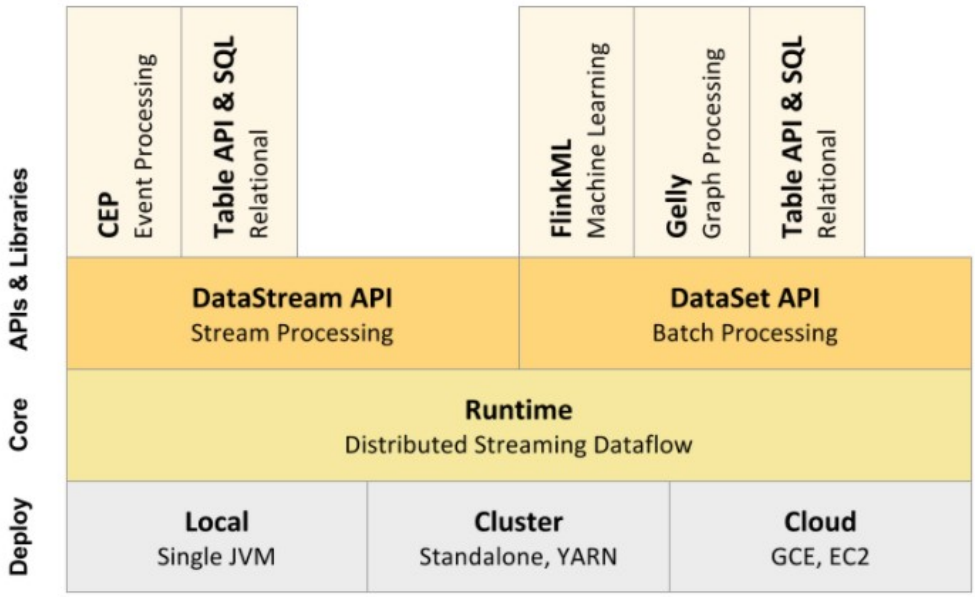

Figure 2.1: The Flink software stack [3]

2.2 Execution pattern

As many other distributed systems, for the program execution, Flink implements the **master-worker** pattern. The master process is called *JobManager* and the worker process is called *TaskManager*. Flink cluster consists of at least one JobManger and one or more TaskManager. FlinkClient takes the program code, transforms it into a dataflow graph and submits that to JobManager. The JobManager coordinates the distributed execution of the dataflow and tracks the state and progress of each operator or stream. JobManger handles also the scheduling of new possible operators and coordinates checkpoints. The data processing happens in the TaskManager executing one or more operators that produce streams and report the status to the JobManager [5].

[3] `http://ci.apache.org/projects/flink/flink-docs-release-1.1/internals/general_arch.html`

[4] Fig. source: Learning Apache Flink, page 27, [4]

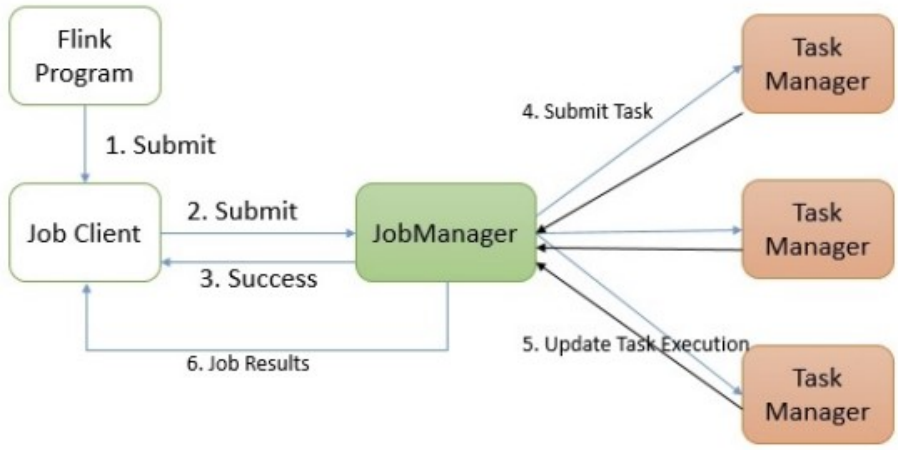

Figure 2.2: Flink application execution[4]

2.3 Flow control with backpressure

In the world of data, ingesting streams with high volume will lead the application to a point where it is impossible to process input data at the rate at which it arrives. This typically happens when the input data stream is too high for the resources allocated or the input rate varies and causes spikes of high load. Despite the causes of why an operator can not handle the data as input, this situation should never be a reason for a stream processor to terminate. Backpressure, if not dealt with correctly, can lead to exhaustion of resources, or even, in the worst case, data loss.

In Flink, TaskManagers exchange data among themselves and it can be done in local or remote-over-network. When the input data rate is high, the receiving TaskManager start to queue buffers with the received data (see Fig 2.3). At some point the buffer pool has no more available buffers, that means it can no longer continue to buffer data. The sending TaskManager starts queuing buffers until its own buffer pool ha no more buffers available. With both buffer pools empty, the tasks are blocked and eventually the whole application is processing to a slow rate [10].

Apache Flink does not send the records one-by-one as it would be the cause of overhead. It packs together records into its network buffer and sends them together. By this operation, as a performance result, the cost

per records will be reduced and the throughput will be higher. But still a high number of buffers on the queue leads to backpressure and to avoid this Flink introduced **Cred-based Flow Control**.

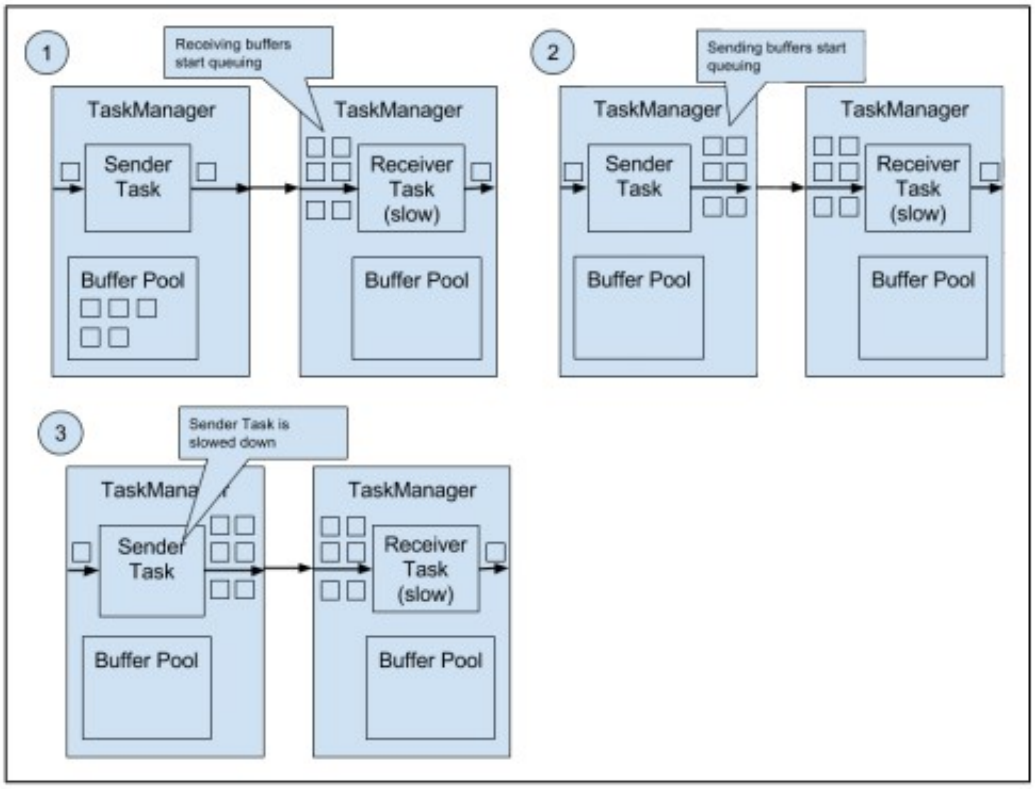

Figure 2.3: Flink's backpressure scenario[5]

2.3.1 Handling backpressure

The central observation is the following: *For records to progress through Flink, buffers need to be available* [3]. When a buffer is withdrawn from the pool, after it is consumed, it will be recycled to the pool for further operations. The size of the pools varies as it depends on the run time. The buffering rate is defined by the amount of memory buffers in the network stack. There are two types of buffers, *exclusive one*, buffers allocated to a specific channel and *floating buffers*, buffers shared among all channels. The floating buffers are distributed in the base of the number of buffers waiting in the queue, helping to relieve backpressure.

[5]Fig. source: Stream Processing with Apache Flink, Fig 2.4, [10]

For an easy explanation, we will consider buffers as credits. So, 1 credit means 1 buffer. In this example, exclusive buffers are set to 2 and floating buffers are 5.

1. Receiving TaskManager sends 2 buffers(credits) to sender TaskManager. At this point, the buffer pool of available buffers at the receiver is empty.
2. The sender task sends 2 buffers (batched data) along with the backlog of size 5 to the receiver.
3. Receivers request the floating buffer based on the number of buffers waiting in the queue, because different senders may have different batch size and priority.
4. The request may be accepted, rejected or partially accepted. In this example, it will ask for 3 floating buffers as it already has 2 exclusive buffers.
5. The receiver will send an exclusive buffer and new floating buffers to the sender task for further processing. At this point, there are 5 available buffers for the receiver.

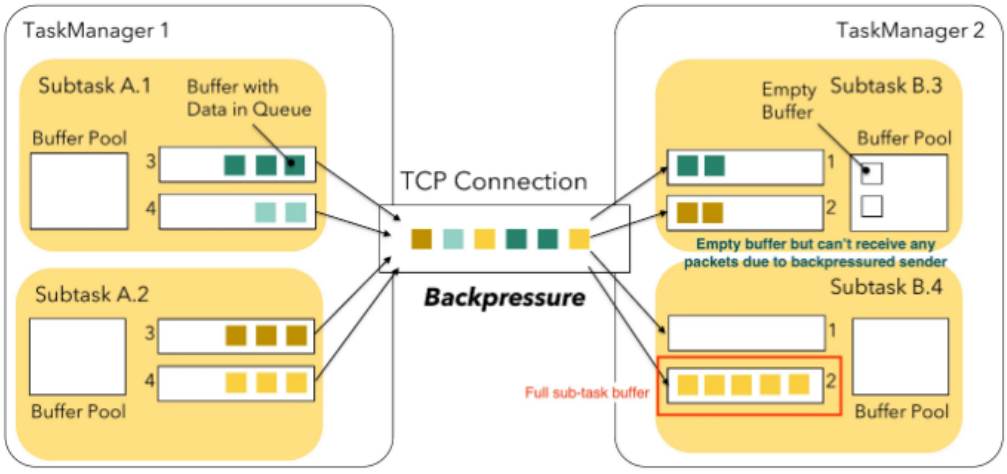

Figure 2.4: Credit-based Flow Control[6]

When the sending TaskManager sends a buffer to the receiver, 1 credit is subtracted from the total credits sent by the receiver. Receiver already told

[6] https://www.kharekartik.dev/2019/12/12/what-makes-apache-flink-scale/

the sender about the amount it can process and helps to eliminate backpressure at the receiver's end. In different situations, the credits may go to `0`, but using the network stack, buffers can be kept there so that they can hold the system failure or data loss for some time.

2.4 Exactly-once semantic

Most challenging type of guarantee to achieve is **exactly-once semantic**. Exactly-once leads to no event loss and the updates on internal states will be applied exactly once for each event. When we say "*exactly-once semantics*", what we mean is that each incoming event affects the final results exactly once. Even in case of a machine or software failure, there's no duplicate data and no data that goes unprocessed, as the failure never happened.

To satisfy this type of guarantee, Flink introduces a feature known as "***checkpoints***" as a way to handle the failures and to reset the state of the application. A checkpoint is a consistent snapshot of the current state and the position of the application in an input stream.

Checkpoints act as regular records and are processed by operators, but they trigger actions related to checkpointing. When a data source, that is reading data from the input stream, detects a checkpoint barrier, it saves the position of this record and it is used as an offset. This offset will be used whenever the stream is served from the message transport to a stable storage mechanism. The JobManager holds pointers to the location where the checkpoints of every task are stored [7].

At the verification of any failure situation, Flink recovers the application's state by resetting the state to the most recently completed checkpoint.

2.4.1 Checkpointing without stopping the world

Stopping the world is an expression used in distributed systems to represent those situations where a streaming application should "be paused" and then "resumed" to take a checkpoint. This method to take checkpoints is not acceptable even in those applications having moderate latency requirements.

Flink implements a solution to this problem based on a well-known algorithm, Chandy-Lamport algorithm[7] for distributed snapshots[8]. Using this

[7] a snapshot algorithm that is used in distributed systems for recording a consistent global state of an asynchronous system

[8] see `http://lamport.azurewebsites.net/pubs/chandy.pdf`

algorithm, Flink does not need to stop the world to take the checkpoints.

2.4.1.1 Checkpointing barriers

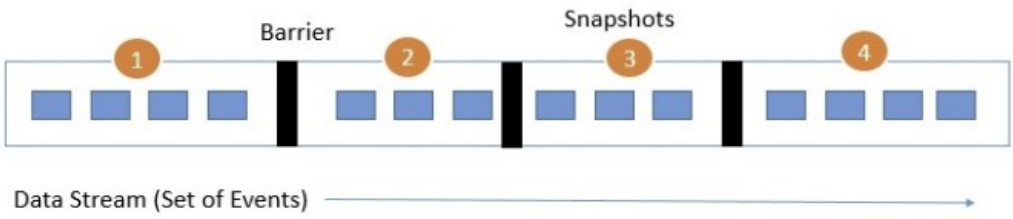

Figure 2.5: Checkpointing & barriers

The core elements in Flink's distributed snapshotting are stream **barriers** and are injected into the data as part of the data stream. The checkpointing barriers split a stream into two parts, the set of records that goes into the current snapshot and the records that go into the next snapshot. The position of the barrier defines the extent of a checkpoint. All state modifications due to records that precede a barrier are included in the checkpoint while the records following the barrier do not modify the checkpointed state.

A JobManager initiates a checkpoint by sending a message with a new checkpoint ID to each data source task. When a data source task receives a message with a new checkpoint, it pauses its processing operation, checkpoint its local state to the remote storage and emits a checkpoint barrier into its regular output stream. Emitting the checkpoint barrier into the output data stream defines the stream position where the checkpoint was taken. The barrier is broadcasted to all the tasks to ensure that each task received a checkpoint from each of its input streams. After these steps, the task waits for the arrival of all barriers for the checkpoint, meantime continues processing records that did not contain a barrier yet. The records that forward an injected barrier should be buffered and not proceed [10].

Once all barriers arrive, the task checkpoints its operator state to the remote storage and forward a checkpoint barriers to each of its downstream connected tasks. After the checkpoint barrier has been emitted, the task starts processing the buffered records.

The task should acknowledge the reception of the barrier to JobManager and it records the checkpoint as completed once it receives a barrier acknowledgement from each sink task.

2.4.1.2 Savepoints

Savepoints are one of the mechanisms that distinguish Apache Flink from other stream processing frameworks. This mechanism offers the possibility of reprocessing and going back in time at a determinant point of the stream input that was a user's choice [11]. A savepoint is a globally consistent snapshot[9] of the positions of all data sources and the state of all operators.

There is a similarity between checkpoints and savepoint, but these two mechanisms have different functionalities. Checkpoints are an internal mechanism of Apache Flink used to recover the system in case of failure by copying the application state and reading positions of the input. Flink savepoints main goal is to act as the way to restart, continue or reopen a paused application after manual backup and resume activity by the user. Checkpoints are automatic and periodic in nature because they are created and dropped by Flink without any user interaction. On the contrary, savepoints are managed manually by the user.

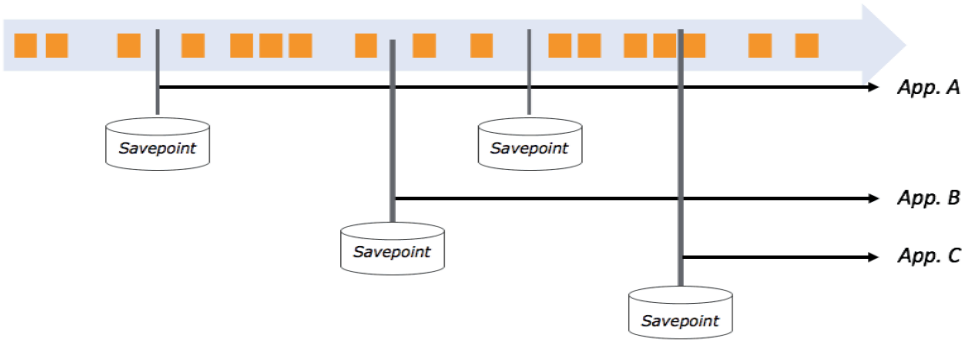

Figure 2.6: Savepointing in Flink [10]

2.4.1.3 Watermarks

Watermarks are an instrument offered by Apache Flink to configure event-time window behaviour. To keep track of event time, we need a clock that

[9] the state of all parallel operators is checkpointed at exactly the same well-defined position in all inputs

[10] https://www.ververica.com/blog/turning-back-time-savepoints

is driven by the data instead of the clock used by machines. Watermarks provide a logical clock[11] which informs the system about the current time. It represents a certain point in time, where we are confident that no more delayed events will arrive. Receiving a watermark `T`, the operator assumes that no further event with timestamp less than `T` will be received [1]. This is an essential concept to handle out-of-order events for event-time windows and operators.

Using watermarks we notice a trade-off between confidence and latency because watermarks that ensure low latency, provide lower confidence. The same thing happens when providing high confidence, we might unnecessarily increase the processing latency [2].

2.5 Data Stream API

Data stream programs in Flink are regular programs that implement transformations on data streams. A Flink program is compound by the following basic steps:

- Execution environment,
- Load & create the initial data,
- Transformations on this data,
- Result of the computation,
- Program execution

`DataStream` [6] class is used to represent a collection of data in a Flink program where data can be bounded or unbounded. The data consists of immutable collections that may contain duplicates. In order to apply transformations on this data, we should use the operations offered by DataStream API.

[11] mechanism for capturing chronological and causal relationships in a distributed system

```
1  /** Set the runtime environment **/
2  StreamExecutionEnvironment env = StreamExecutionEnvironment
       .getExecutionEnvironment();
3
4  /** Configure the data source to read data */
5  DataStream<String> text = env.readTextFile ("input");
6
7  /** Perform a series of transformations */
8  DataStream<Tuple2<String, Integer>> counts =
9                text.flatMap(new Tokenizer()).keyBy(0).sum(1);
10
11 /** Configure the Sink to write data out */
12 counts.writeAsText("output");
13
14 /** Submit for execution */
15 env.execute("Streaming WordCount");
```

This simple code example of `WordCount` provides the basic structure for developing programs based on the Flink DataStream API.
LINE 2: The first step is to obtain a `StreamExecutionEnvironment` object that represents the context object of building the graph.
LINE 5: Using the built-in data source for reading files in the `Environment` object we obtain a `DataStream` object with an unbounded data set.
LINE 8-9: We can apply a sequence of different operations on this data set, for example, each record[12] is first separated into words and implemented through the `FlatMap` operation. To obtain a stream of words, using `keyBy(...)` we group the words in the stream and compute the data of each word applying `sum(...)` operation.
LINE 12: The computed word data forms a new stream and is written into the output file.
LINE 15: Finally, by calling `execute(...)` method we start the program execution. The logic will not be executed if the method will not be called.

After seeing the previous example and what operations Flink offers to build a simple program structure, now we will give a comprehensive description of the most important elements of DataStream API.

[12] one row in the file

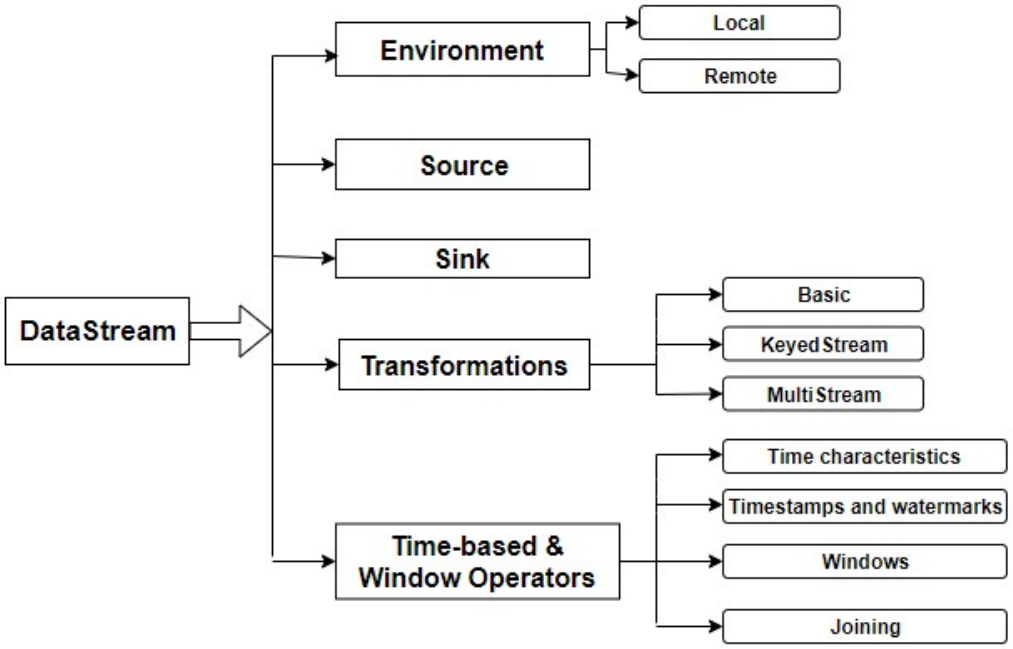

Figure 2.7: Summary of DataStream API structure

2.5.1 Environment

To start writing a Flink program, we first need to get an existing execution environment or create a new one. The `StreamExecutionEnvironment` is the core of all Flink programs and can be obtained using static methods on `StreamExecutionEnvironment` as:
- `getExecutionEnvironment()` to get an already existing one,
- `createLocalEnvironment()` to create a local environment,
- `createRemoteEnvironment(host, port, jarFiles)` to create a remote environment.

2.5.2 Source

Sources represent places where the Flink program expect to get its data from. This is the second step in Flink's program anatomy (see sec 2.5). Flink provides pre-implemented data source functions, but we can write our custom sources by implementing `SourceFunction`[13] or `ParallelSourceFunction`. We can attach a source to the program by calling `addSource(sourceFunction)`. One of the predefined sources accessible from `StreamExecutionEnvironment`

[13]for non-parallel sources

is `readTextFile`[14] to read text files line-by-line and returns them as strings.

2.5.3 Transformations

Data transformations transform one or more data streams from one form into another form. The most important transformation operations Flink offers are represented in Fig. 2.8 as an extension of Fig. 2.5.

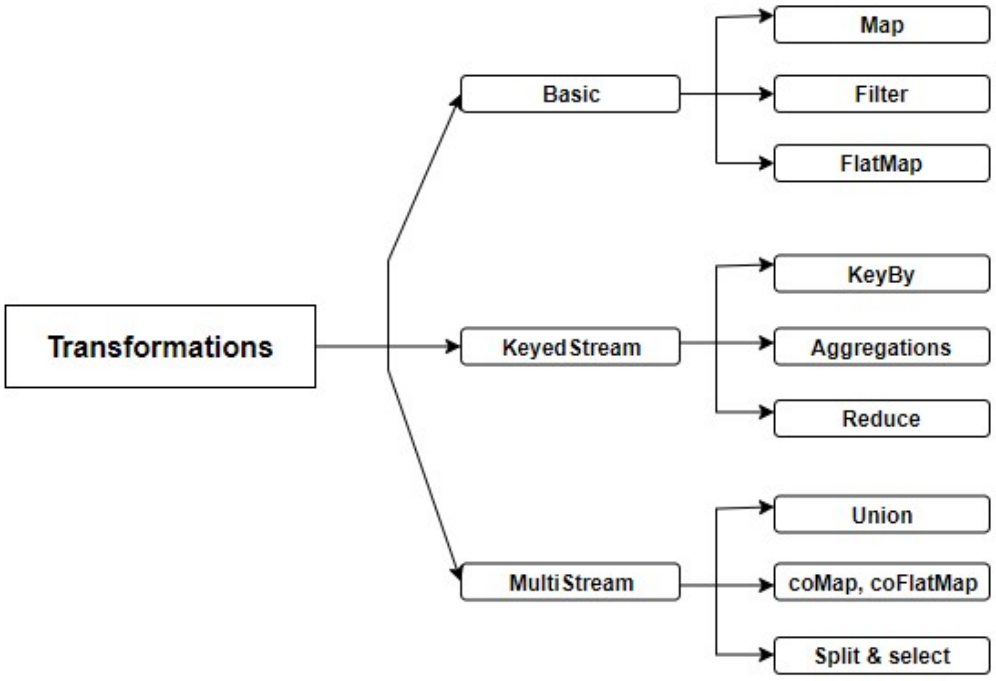

Figure 2.8: Data stream transformation operations

The first category, the *basic* one, `map, filter, flatMap` take one or more streams as input to transform it into another stream. This type of transformations processes individual events, meaning that each output record was produced from a single input record.

The *keyedStream* category takes one or more streams as input to transform it into a `KeyedStream`. This feature provided by DataStream API, group the events that share a certain property together. The `keyBy` converts a `DataStream` into a `KeyedStream` by specifying a key. The aggregations are applied on a `KeyedStream` to produced a `DataStream` of aggregates. Some of the most important aggregation operations are: `sum, min,`

[14] used in the previous example (see code example 2.5, line 5)

`max, minBy, maxBy`. `Reduce` operations are used to combine the current element with the last reduced element and to emit the new value.

The main function of the *multistream* category of transformation is to merge different streams or separate stream into a different stream. The most used operators are `union` used to merge two or more streams of the same type to produce a stream of the same type and `split` that has the inverse function of the union transformation.

2.5.4 Sink

After the data transformations are done, we need to save results in someplace. Data sinks consume data streams and forwarded them to files, sockets, external system or print them. Flink offers built-in sinks by calling various methods as `writeAsText()`, to write elements as strings, or `writeAsCsv()`, to write tuples a comma-separated value files. By calling `print()` method we can print the string value on the standard output and by calling `addSink(...)` method we can add our custom data consumer.

2.5.5 Time-Based and Window Operators

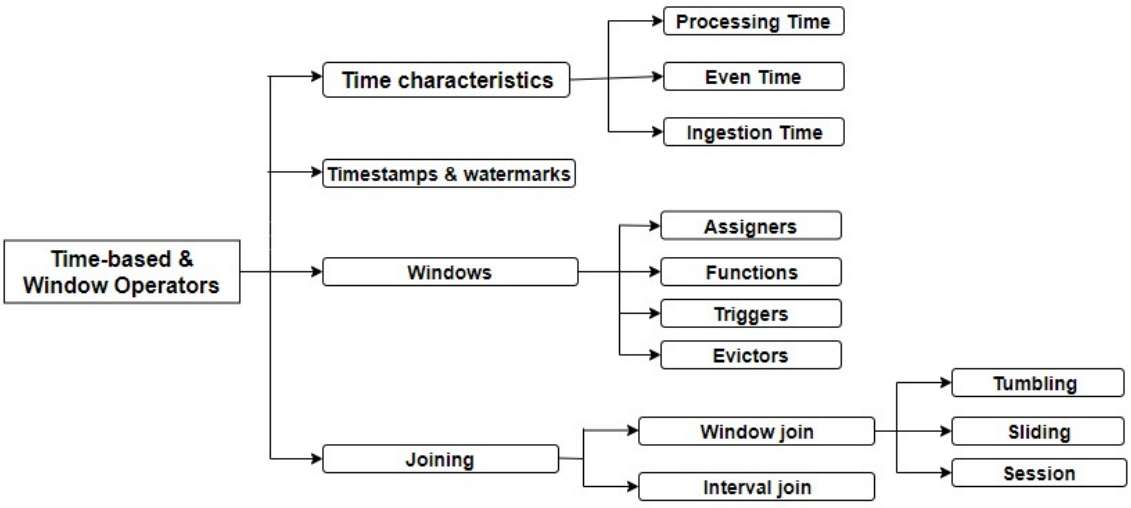

Figure 2.9: Time-based and Window operators

2.5.5.1 Time characteristics

To define and apply time operations in a stream processing program is important to understand the meaning of time. DataStream API provides the possibility to configure *time characteristics* on window creation. The time characteristics is a property of `StreamExecutionEnvironment` and take the following values:
- *Ingestion Time*: the time that events enter Flink,
- *Event Time*: the time that event occurred on its producing device,
- *Processing Time*: the system time of the machine that is executing the respective operation.

2.5.5.2 Windowing

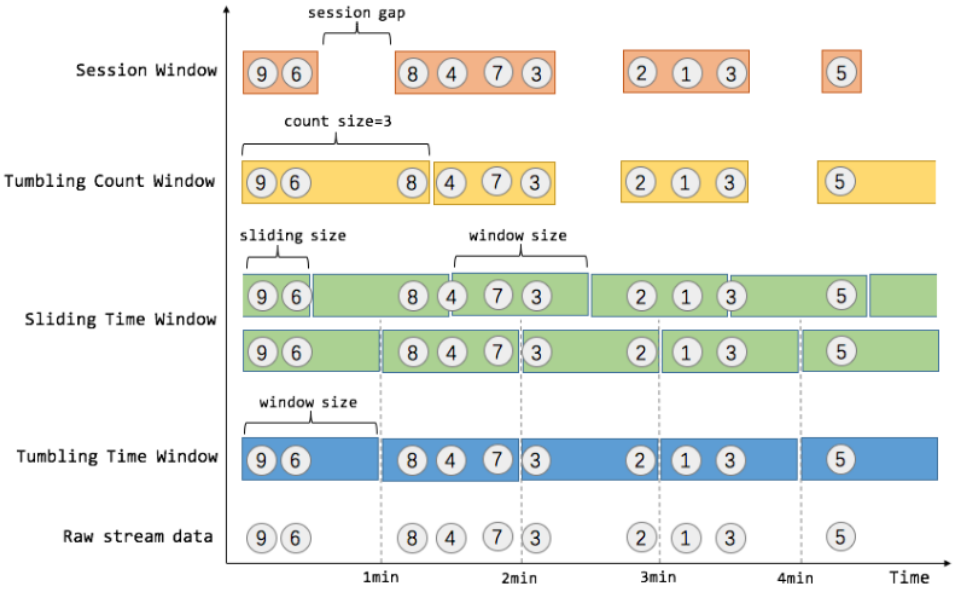

Figure 2.10: Flink different window types [15]

Windows are a mechanism that Flink offers, to group and collect a bunch of events by time or by other characteristics to compute their result. A window is created as soon as the first element that should belong to this window arrives, and it is removed when the time passes its end timestamp plus the allowed lateness if it specified.

[15] `https://datafibers-community.github.io/blog/2019/10/05/2019-10-05-flink-windows-explained/`

Each window has a `Trigger` and a function attached to it. The function defines the computation to be applied to the elements of the window, while `Trigger` defines the conditions under which the window is considered ready for the function to be applied. `Evicitors` are used to remove elements from the window after the trigger fires and before and/or after the function is applied [8].

Window assigners
A `WindowAssigner` assigns each incoming event to one or more windows. We can extend `WindowAssigner` to implement a custom window assigner. Flink offers pre-defined window assigners as (see Fig. 2.10):
- *Tumbling*: windows assign events into non-overlapping buckets of fixed size,
- *Sliding*: windows assign events into overlapping buckets of fixed size. The sliding is provided by defining the length and the slide,
- *Session*: windows assign events in the base of user activity or active sessions.

Window functions
A *window function* is used to specify the computation that we want to apply on each of the windows. Once the system determines that window is ready for processing, the function will be computed at each element. The most important window functions that Flink offers are:
- `ReduceFunction`: specifies how two elements from the input are combined to produce an output element of the same type,
- `AggregateFunction`: a generalized version of a ReduceFunction that an input type, accumulator type, and an output type. It has methods for adding one input element to an accumulator. The interface offers method to create an accumulator or to merge the existing ones.
- `FoldFunction`: specifies how an input element of the window is combined with an element of the output type.

Triggers A `Trigger` determines when a window is ready to be processed by the window function. If no trigger will be specified, `WindowAssigner` has their own default `Trigger`. We can specify a custom trigger by calling `trigger(...)` method.

2.6 Process function

Apache Flink offers a lot of high-level functionalities for data stream processing and transformation as we have seen in the previous sections. Besides the high-level functionalities, Flink provides an API for low-level stream processing operations. This API is embedded in the DataStream API through `ProcessFunction`.

2.6.1 The ProcessFunction

It is a low-level streaming operation that provides access to the basic building blocks builds by streaming applications corresponding to the following three concepts:

1. **events**: stream element or stream data
2. **state**: fault tolerance
3. **timer**: event and processing time

This mechanism gives direct access and control of the Flink's state and context, where Flink keeps the information of the current partition key, current timestamp and the timer service. State and timer service are only available in process function on a keyed stream partitioned by a specific key. `stream.keyBy('id').process(new MyProcessFunction())`

The `ProcessFunction` can be considered as a `FlatMapFunction` with access to keyed state and timers. It will be invoked for each received event in the input stream. To implement low-level operations on two input can be used `CoProcessFunction` or `KeyedCoProcessFunction`. This function is to bound two different inputs and gets individual calls to `processElemt1(...)` and `processElement2(...)` for records from the two different inputs.

2.6.2 Timers

Timers allow reacting to events in processing time and event time. Each call to `processElelemt(...)` gets a Context object which gives access to the element's event time timestamp and to the `TimerService`. The timer service is used for registering/unregistering timers that are used for performing an action when a timer is triggered by the current time. When a timer's particular times is reached, the `onTimer(...)` method is called. It is worth to mention that only one timer can be registered per key and timestamp.

2.7 Flink command-line interface

Flink runs on all *UNIX-like environments*, e.g Linux, Mac OS X, and Cygwin. Make sure you have installed **Java 1.8.x** or higher and correctly configured `JAVA_HOME` environment variable.

Now we are going to show some of the most important Flink command-line commands to start/stop a cluster, to run an application in various ways, to manage the job execution and to handle savepoints.

2.7.1 Cluster commands

The following command starts a Flink cluster on the local node and now is ready to accept jobs.

```
$ ./bin/start-cluster.sh
Starting cluster.
Starting standalonesession daemon on host cokle-HP.
Starting taskexecutor daemon on host cokle-HP.
```

After correctly executing this command, Flink offers a rich customized web interface to visualize the state, information and metrics about our cluster. It is accessible thought `http://localhost:8081/#/overview` address.

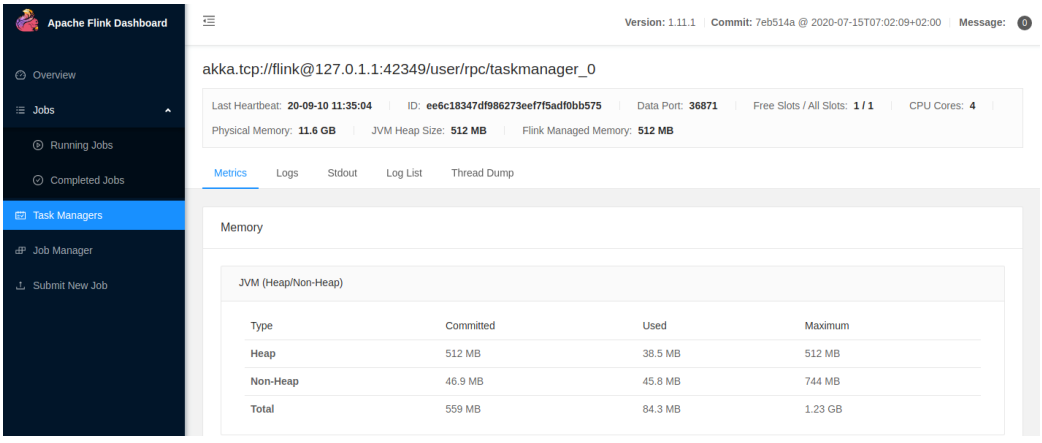

Figure 2.11: Flink dashboard interface

In order to stop the running Flink cluster we execute the command:

```
$ ./bin/stop-cluster.sh
Stopping taskexecutor daemon (pid: 15159) on host cokle-HP.
Stopping standalonesession daemon (pid: 14845) on host
   cokle-HP.
```

2.7.2 Job submission

To run an application with no arguments we can use:

```
$ ./bin/flink run ./examples/MyApplication.jar
```

Adding `-q` flag we will run the program with log output disabled.

To specify program arguments for input and output files:

```
$ ./bin/flink run ./examples/MyApplication.jar \
                    --input file:///home/user/dummy.txt \
                    --output file:///home/user/dummy_out
```

Adding `-p 16` flag we will be able to run the program with parallelism of `16`.

To run the program specifying a specific class of the application as an entry point we can use:

```
$ ./bin/flink run -c org.apache.flink.examples.EntryPoint \
                    ./examples/MyApplication.jar \
                    --input file:///home/user/dummy.txt \
                    --output file:///home/user/dummy_out
```

2.7.3 Job Management

Flink offers a rich command-line interface to manage the jobs, but we will list the most important and used one. To list all the scheduled and running jobs, including their `JobID` use:

```
$ ./bin/flink list
```

Specifying `-s` flag we will list only the scheduled one, in the other side `-r` will show only the running ones.

To cancel a running job run:

```
$ $./bin/flink cancel <jobID>
```

To make use of the savepointing mechanism when we want to a job:

```
./bin/flink stop [-p targetDirectory] [-d] <jobID>
```

2.7.4 Savepoints

Flink makes it possible to control the savepoints by client command line. We can trigger a savepoint by:

```
$ ./bin/flink savepoint <jobId> [savepointDirectory]
```

This command will trigger a savepoint for the job with ID `jobID` and returns the path of the created savepoint. The path is used to restore the savepoints. We can optionally specify the target file system directory to store the savepoint in which will be accessible by the `JobManager`.

To restore a savepoint, run:

```
$ ./bin/flink run -s <savepointPath>
```

To dispose a savepoint, run:

```
$ ./bin/flink savepoint -d <savepointPath>
```

2.8 Flink environment configuration

Flink configuration is done thought `conf/flink-conf.yaml` files which is parsed and evaluated when the Flink processes are started. We can customize the environment modifying the values in `YAML` file.

2.8.1 Memory sizes

The default memory size values are efficient for simple streaming/batch applications, but we can customize their values by:

- `jobmanager.memory.process.size`: Total size of the `JobManager`,
- `taskmanager.memory.process.size`: Total size of the `TaskManager` process.

These values are configured as memory sizes, for example *1536m* or *2g*.

2.8.2 Parallelism

Configuring the parallel execution of programs in the `YAML` file:

- `taskmanager.numberOfTaskSlots`: The number of slots that a `TaskManager` offers where each slod take one task.
- `parallelism.default`: The default parallelism used when no parallelism is specified anywhere (*default 1*).

2.8.3 Checkpointing

The checkpointing values can be configured directly in the code with Flink job, but specifying the values in this file defines them as defaults.

- `state.backend`: This defines the data structure mechanism for taking snapshots, *state backend*. Common values are `filesystem` or `rocksdb`.
- `state.checkpoints.dir`: The directory to write checkpoints to. This takes a path URI like *s3://mybucket/flink-app/checkpoints* or *hdfs://namenode:port/flink/checkpoints*.
- `state.savepoints.dir`: The default directory for savepoints. Takes a path URI, similar to `state.checkpoints.dir`.

2.8.4 Host and ports

Configuring the host names and ports of different Flink components:

- `jobmanager.rpc.address`: The network address to connect to for communication with the job manager (*default none*).
- `jobmanager.rpc.port`: The network port to connect to for communication with the job manager (*default 6123*).
- `taskmanager.data.port`: External port used for data exchange operations (*default 0*.
- `taskmanager.rpc.port`: The external RPC[16] port where the TaskManager is exposed (*default "0"*).

[16]Remote Procedure Call

Chapter 3

Using Apache Flink

In this chapter, we will describe some experiment testing scenarios to highlight the common usage of the DataStream API and the features offered by Apache Flink. The practical side will be based on `TwitterFlink`, an event-driven application that ingests tweets from a twitter data source and reacts to incoming tweets by triggering other computations. Different application entry-points will represent different testing scenario. The testing scenarios aim to highlight the most important features of DataStream API.

Other than testing scenarios based on Twitter tweets, will be implemented other test working with non-real-time streams to accurately describe the advantages of checkpointing mechanism to recover from failures and watermark mechanism to handle out-of-order events.

3.1 TwitterSource

TwitterStreaming API offers access to the stream of tweets made available by Twitter and `TwitterSource` class is used to establish a connection to this stream. To establish this connection, we should register to their program and acquire the *tokens*[1] for the authentication. The tokens to be defined are:

```
TwitterSource.CONSUMER_KEY, "myKey")
TwitterSource.CONSUMER_SECRET, "mySecret")
TwitterSource.TOKEN, "myToken")
TwitterSource.TOKEN_SECRET, "mySecretToken")
```

[1] the tokens provided on the project source will be disabled after project valuation.

3.1.1 Tweets format on JSON

The `TwitterSource` emits strings containing a JSON object representing a Tweet. The JSON objects offered by Twitter have various information fields, but for easy testing scenarios, we will be considering only some of them.

```
"text" : "this is a tweet text",
"lang": "en",
"entities": {
    "hashtags": ["this", "is", "a", "hashtag", "list"],
    "urls": [],
    "user_mentions": []
 }
```

3.2 DataStream

3.2.1 EnglishTweetsFiltering

This test scenario filters all the tweets in the Twitter data source by the language, in this case, the English language. The example shows how the `TweetMapper` [2] operator applies the mapping function on a `jsonTweet` string object to obtain a Tweet object. After the mapping is done, applying a filter operator with a customized `FilterFunction` we can filter all the tweets by the language.

```
0  /* Set the runtime environment */
1  StreamExecutionEnvironment env = StreamExecutionEnvironment
       .getExecutionEnvironment();
2
3  /** Initialize twitter source */
4  TwitterSource source = new TwitterSource(...);
5
6  /** Apply the transformation operators */
7  env.addSource(source)
8      .map(new TweetMapper())
9      .filter(filterByLanguage(Language.ENGLISH))
10     .print();
11
12 /** Execute the application */
13 env.execute("Filtering tweets by english language");
```

[2] see package flink.twitter.streaming.utils.TweetMapper

```
0 Tweet{language='en', text='...', tags=[], createdAt
     =1599640457000}
1 Tweet{language='en', text='...', tags=[Akhil5, Akhil,
     AkhilAkkineni], createdAt=1599640457000}
2 Tweet{language='en', text='...', tags=[], createdAt
     =1599640457000}
3 Tweet{language='en', text='...', tags=[], createdAt
     =1599640457000}
```

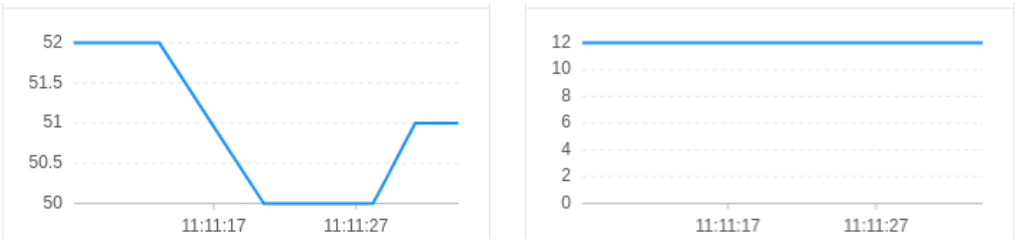

Figure 3.1: Data visualization
To the left: Map records in per second and to the right: Filter records out per second. The map operator ingests around 52 tweets per second and in the others side, filter operator outputs around 12 tweets per second.

3.2.2 TweetsPerLanguage

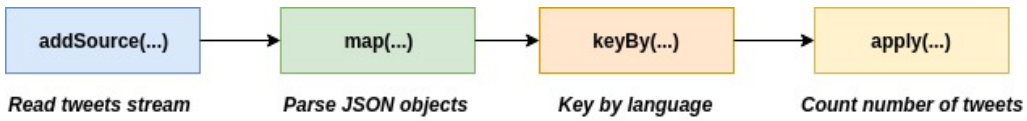

Figure 3.2: Application scenario logic showing the sequence of operators

This application counts how many tweets users write in different languages per 1 minute interval. To implement this application we will use keyed windows. We will start with the tweets stream and then we will group them by language so that we will have one logical stream for each language. Later we will process them in parallel and count how many tweets do we have in each language per selected time interval.

```
0 ...
1 /** Apply the transformation operators */
2  env.addSource(source)
3     .map(tweetMapper())
4     .keyBy(languageSelector())
5     .timeWindow(Time.minutes(1))
6     .apply(countingWindow())
7     .print();
8
9 env.execute("Tweets per language");
```

`TimeWindow` at line `10` will define a non-overlapping `1`-minute window on the tweets stream. Using the `apply(...)` method with a `WindowFunction` as an argument we process to all window elements the same computation, producing the following output:

```
0 /** ( Language code, nr of records, timestamp) */
1 (zh,100,Thu Sep 10 15:27:00 CEST 2020)
2 (fa,105,Thu Sep 10 15:27:00 CEST 2020)
3 (tl,127,Thu Sep 10 15:27:00 CEST 2020)
4 ...
5 (es,328,Thu Sep 10 15:27:00 CEST 2020)
6 (ko,211,Thu Sep 10 15:27:00 CEST 2020)
7 (en,370,Thu Sep 10 15:27:00 CEST 2020)
8 (en,967,Thu Sep 10 15:27:00 CEST 2020)
```

3.2.3 TopHashtag in 5 minutes

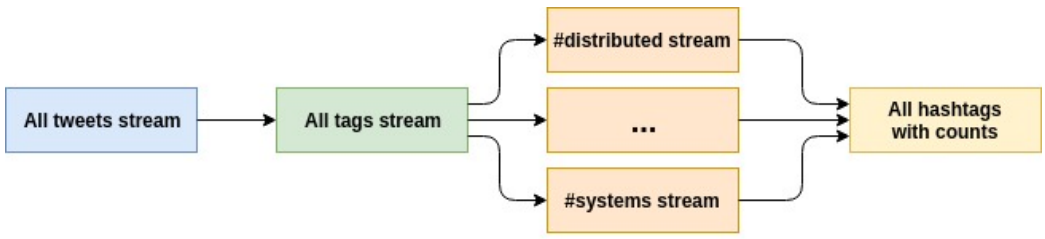

Figure 3.3: Combining both stream types

`TopHashtag` application will find the most used hashtag in a defined interval of time to understand which is more trending topic of Twitter tweets.

This is a slightly more complicated example than the previous one, we will use both keyed and non-keyed windows.

First of all, we will have a stream of tweets as before and then we will extract all hashtags to have a new stream of hashtags. We will group the elements of this stream by hashtag value and we will have a logical stream where every hashtag will have its own logical stream. The will apply count function to find out how many times we saw a specific hashtag and this will produce a stream of hashtags that contains the value and the occurrences. We will use a non-keyed window to find the highest count per a 5 minutes interval.

```
0  ...
1  /** Apply the transformation operators */
2  env.addSource(twitterSource)
3      .map(tweetMapper())
4      .flatMap(toHashtagConvertor())
5      .keyBy(k -> k.f0)
6      .timeWindow(Time.minutes(5))
7      .sum(1)
8      .timeWindowAll(Time.minutes(5))
9      .apply(mostUsedHashtagWindow())
10     .print();
11
12 /** Execute the application */
13 env.execute("Top Hashtag");
```

We will notice an output string with the most used hashtag every 5 minutes

```
0 (Thu Sep 10 16:20:00 CEST 2020, BTS, 464)
1 (Thu Sep 10 16:25:00 CEST 2020, TrumpLiedAmericans, 178)
2 (Thu Sep 10 16:30:00 CEST 2020, COVID19, 143)
3 (Thu Sep 10 16:35:00 CEST 2020, NFL, 93)
4 (Thu Sep 10 16:40:00 CEST 2020, DianaRigg, 72)
5 (Thu Sep 10 16:45:00 CEST 2020, BarbosaGolpeador, 85)
```

3.2.4 TerminalTweetsFiltering

This testing scenario is an extension of the previous example (see Section 3.2.1), but now the tweets filtering will be done through another `DataStream`.

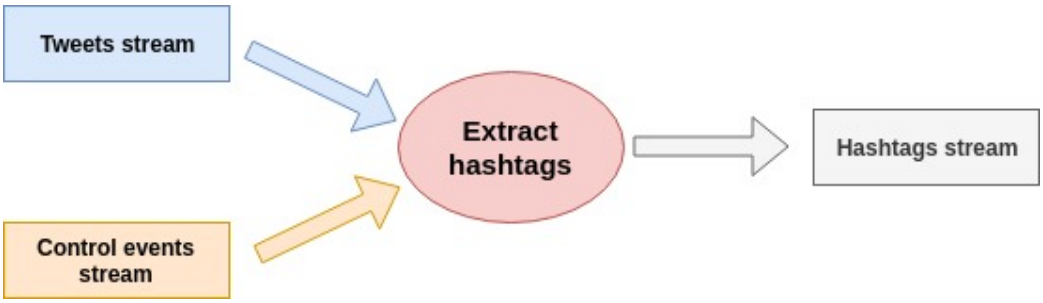

Figure 3.4: Control stream pattern
Application stream logic showing the control stream pattern. In addition to our main tweet stream, we have a control events stream that controls how our application should behave.

Firstly, we will have our regular tweets stream that we are going to parse and group by language key. We will use `socketTextStream(...)` method offered by `StreamExecutionEnvironment` to read commands from the socket and to parse them to Java objects. We will group these commands by a language key. The `connect(...)` function will be used to combine both streams together. The last step extracts the hashtags from our stream of tweets.[3].

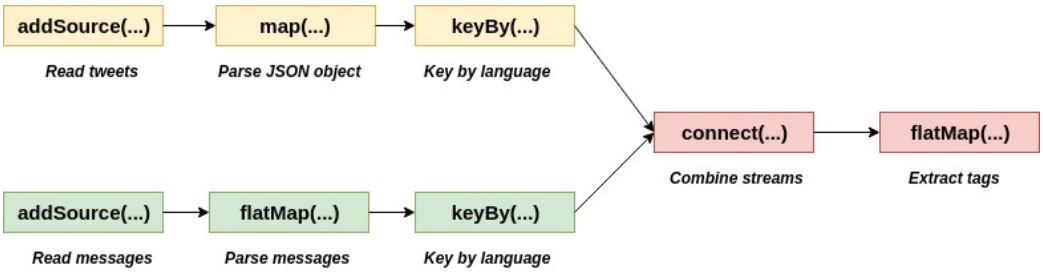

Figure 3.5: Application transformation operators sequence

[3]For better code visualization and readability, the repetitive code lines have been removed and the input parameter's name have been changed.
See the example in package `flink.twitter.streaming.TerminalTweetsFiltering`

```
/** Listen on a socket port and create the stream */
DataStream<LanguageConfig> controlStream =
        env.socketTextStream(HOST, port)
           .flatMap(terminalFlatMap());

/** Apply the transformation operators */
env.addSource(twitterSource)
    .map(tweetMapper())
    .keyBy(languageSelector())
    .connect(controlStream.
                keyBy(terminalLanguageSelector()))
    .flatMap(terminalRichMap())
    .print();

/** Execute the application */
env.execute("Filtering tweets by terminal input");
```

To run this application example, we should firstly, open a socket where we will write the filter commands. The commands will be in form of the ISO language code and a boolean value, for example, `en=true` or `en=false`. To open the socket connection we will use `nc -l PORT` command.

```
$ nc -l 9876
en=true       // firstly we will notice only english tweets
es=true       // after we will notice spanish tweet too
```

```
(en,HongKong)
(en,StopPrivatisation_SaveGovtJob)
(en,COVID19)
...
(es,EspanaDirecto)
(en,Chainlink)
(en,UdhavThackeray)
(es,TomeloEnCuenta)
(en,StopPrivatisation_SaveGovtJob)
(en,DeathOfDemocracy)
```

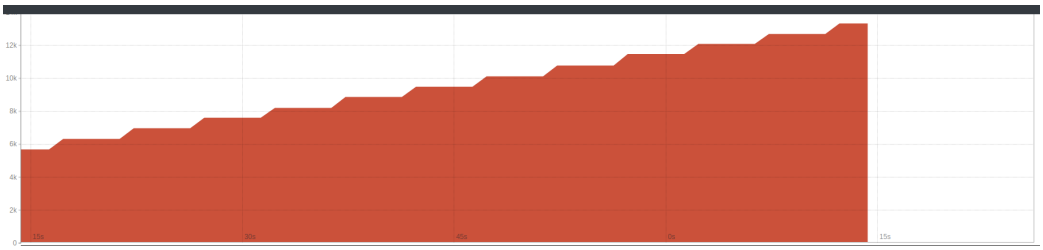

Figure 3.6: TaskManager records in
Graph visualizing the number input records, after enabling and disabling some language filters.

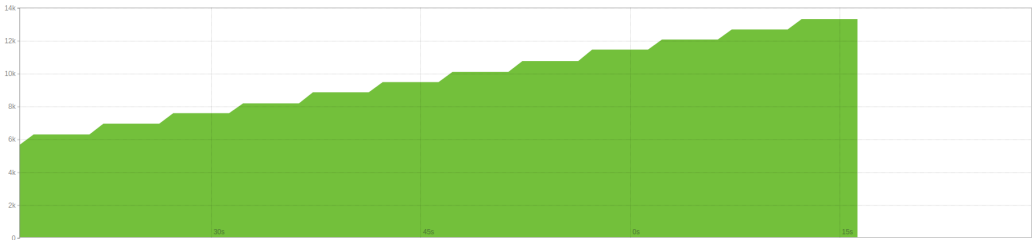

Figure 3.7: TaskManager records out
Graph visualizing the number output records, after enabling and disabling some language filters.

3.3 Advanced mechanisms

3.3.1 Out-of-order: TweetsWatermarked

This testing scenario we will not work with real-time tweets offered from the `TwitterSource` but we will generate *dummy* tweets with a defined timestamp to show the potential of watermarking mechanism in Apache Flink.

```
Tweet("en", "1-First  window", 1525132800000L);
Tweet("en", "2-First  window", 1525132800000L);
Tweet("en", "3-Second window", 1525132920000L);
Tweet("en", "4-First  window", 1525132800000L));
```

As you may have noticed, the third tweet has a higher timestamp and should finish in the second window. The fourth tweet has a lower timestamp, but should finish at the first window.

Using a periodic watermark generator that will inject a `Watermark` into the stream every `n` milliseconds, where `n` is defined by `ExecutionConfig.`

`setAutoWatermarkingInterval(...)` (*default 200ms)*, the fourth tweet finished into the first window. The result of the application will be:

```
(1525132800000,1525132860000, 1-First | 2-First | 4-First)
(1525132920000,1525132980000, 3-Second )
```

To ignore the late events, in this case the fourth tweets, we should implement a punctuated `WatermarkGenerator` configured to generate a watermark every event:

```
...
TimestampAssigner<Tweet> createTimestampAssigner(Context
   context) {
    return new TimestampAssigner<Tweet>() {
        public long extractTimestamp(Tweet tweet) {
            return tweet.getCreatedAt();
        }
    };
}

void onEvent(Tweet tweet, WatermarkOutput output) {
    output.emitWatermark(
        new Watermark(tweet.getCreatedAt()));
}
```

Now the fourth tweet will be dropped because it is considered as a late event and we will have this output:

```
(1525132800000,1525132860000, 1-First | 2-First)
(1525132920000,1525132980000, 3-Second )
```

This could be adjusted by relaxing the watermark generator to accommodate some amout of out-of-orderness.

```
..
void onEvent(Tweet tweet, WatermarkOutput output) {
    output.emitWatermark(
        new Watermark(tweet.getCreatedAt()) -
           MAX_OUT_OF_ORDERNESS);
}
```

Injecting watermarks at each occurred event will cause overhead and in a real-time data source, we would not use punctuated watermarks in this way.

For the most application periodic watermarks based on `BoundedOutOfOrdernessTimestampExtractor` is a better choice. This was only a demonstration how Flink handles out-of-order events with watermarks mechanism.

3.3.2 Parallelism: EnglishTweetsFiltering

This scenario will be an extension of the example of `EnglishTweetsFiltering` (see sec 3.2.1), but this time we aim to demonstrate how Flink handles scalability and parallelism through rebalancing of the available resource for more efficient work.

Our source pipeline is composed of `4` operators and it will be executed in a cluster with 3 TaskManagers having only one task slot. For the operators has been specified different parallelism values, having a total of `5` tasks, where max parallelism defined is `3`.

```
...
/** Operators parallelism */
env.addSource(source).setParallelism(1)
    .map(mapper()).setParallelism(1)
    .filter(byLang()).setParallelism(3)
    .print().setParallelism(1);
```

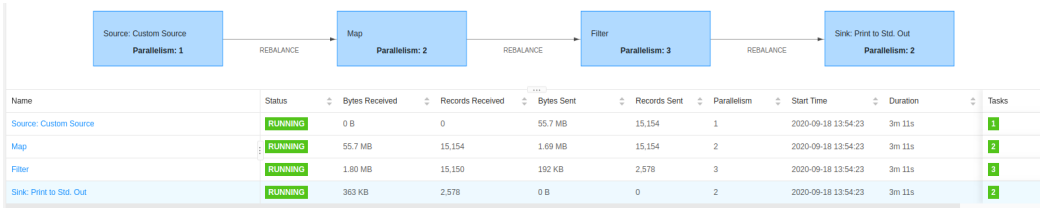

Figure 3.8: Flink UI dashboard
Pipeline running configuration having the max parallelism 3 and total tasks 5. The cluster has used all available resources to run the job, 3 TaskManagers.

To be able to run our job, Flink cluster should have at least `3` TaskManagers. If the number of TaskManagers is `3`, everything is working fine and Flink finds a way to balance data between task slots. It seems that the number of TaskManagers must be equal or grated than the max number of parallelism used in the pipeline, in our case `3`. IF we change the parallelism of one operator to `4`, we must have 4 TaskManagers to be able to run the job.

We decided to add one more `TaskManager`, after the job was still running without changing the initial parallelism configuration (max was still `3`), and noticed that the already added resource was not being used by Flink cluster.

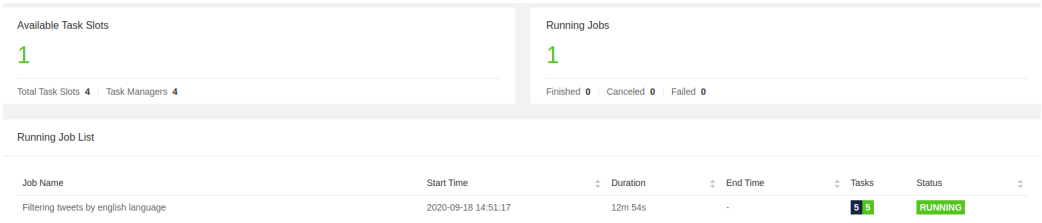

Figure 3.9: Environment resources
After the job was running with the initial configuration, after adding a new resource, Flink cluster is not utilizing it.

We realised that each task needs a TaskManager to run, but Flink enables slot sharing by default so that one `TaskManager` slot can host one parallel instance of each task in the job. That's why our job can start with `3` task slots. However different parallel instances of the same task can not share a `TaskManager`, that's why we need at least 3 TaskManagers to run the job. We can set tasks to be in different slot sharing group via `slotSharingGroup(...)` to force certain `TaskManager` to not share slots. This allows the task to bot burden each other, but in this way, the job will need more slots to start.

Flink does not currently auto-scale a job as a new resource become available. In order to use the already needed resource, the job must be stopped via savepoint and restarted with new parallelism. Flink offered a rescale CLI but was temporarily removed.

[4] `https://ci.apache.org/projects/flink/flink-docs-release-1.9/concepts/runtime.html#task-slots-and-resources`

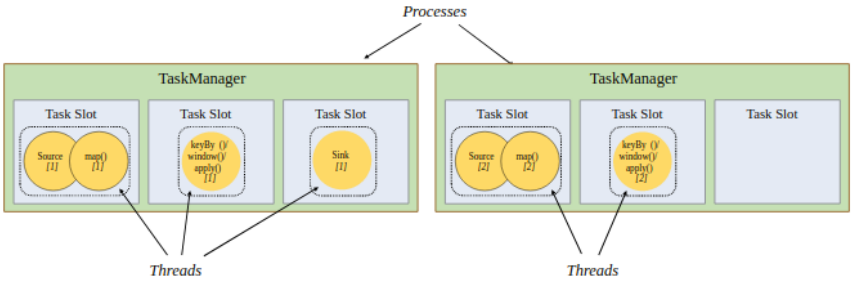

Figure 3.10: TaskManager internal subset of resources. [4]

3.3.3 Failure recovery: JobRecovery

This testing scenario aims to point out the advantages of the checkpointing mechanism, used to recover running or scheduled jobs from failures. The failures will be generated by killing the resource associated to with a `TaskManager`. After a defined number of attempts, the `JobManager` will declare it as a finished job.

The main stream source will implement `CheckpointedFunction` to successfully generate checkpoints and it will continuously generate `Long` values. After the data is generated, it will be passed to a mapping operator which at some point will kill the task that is handling that operation. The mapping function will handle the failure generation and will capture the information regarding the failure by passing into a downstream as the output result.

Flink supports various approaches for storing and checkpointing state in other state backends and it can be configured via by passing the state backend to `StreamExecutionEnvironment.setStateBackend(...)`. The application will be using a `RocksDB` as a distributed state storage which scales well beyond main memory and reliably stores large keyed state. To reduce the time taken from a checkpoint, `RocksDB` constructor takes a boolean flag as input to enable incremental checkpointing. Incremental checkpoints can dramatically reduce the checkpointing time in comparison to fully checkpoints, because it records only the changes compared to the previously completed checkpoint.

Firstly the job will be in the *created* state and then will switch to *running*. In case of failures, the job switches to *failing* where it cancels all the running tasks. If the job can be restarted, it will enter the *restarting* state. The `JobManager`, opens the latest checkpoint object and sends back to the

corresponding task, which then can restore its state from the distributed storage. This storage is fault-tolerant and the state can be easily redistributed. Once the job has been completely restarted, it will reach the *created* state and will continue the execution.

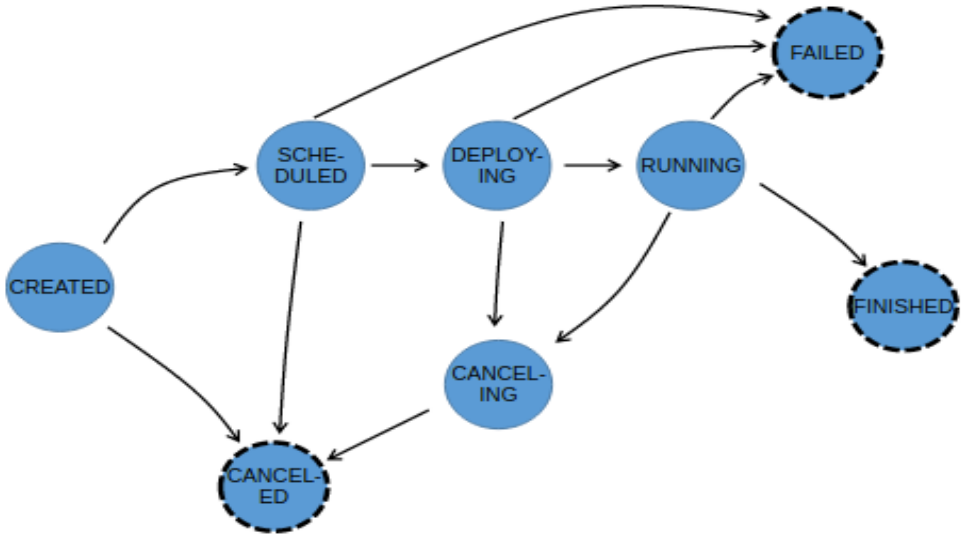

Figure 3.11: This diagram represents all the possible states of a cluster job.

At the end of this testing scenario, we will notice how Flink recovers the job execution from failure by restoring the last successfully saved checkpoint that was stored in `RocksDB`. Flink will try for e defined number of times to recover a job, otherwise, it will successfully tag the job as finished and exit the execution.

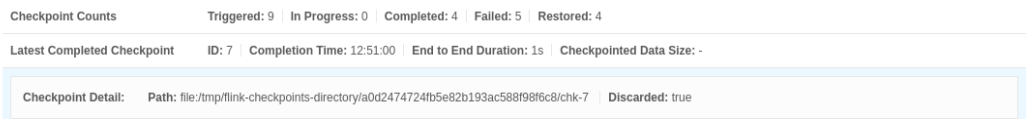

Figure 3.12: Restoring checkpoints
After continuously killing the resources by the application, having the max number of attempts 3, Flink has successfully completed 4 state restores and then it will label the job execution as finished.

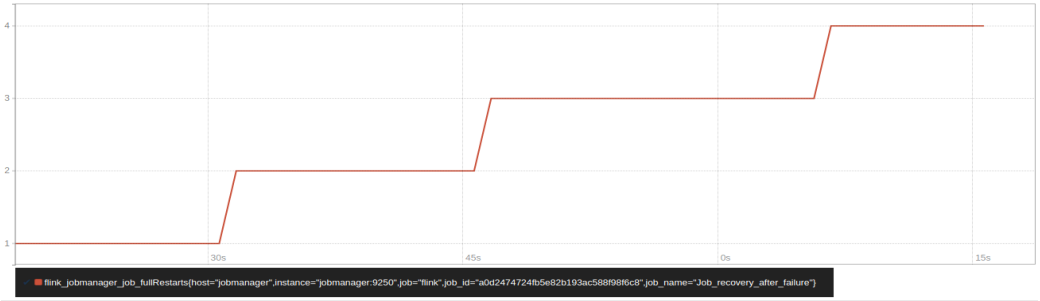

Figure 3.13: Job number of restored states
This diagram represents the number of recovered states from the failures using the last successfully saved checkpoint and a RocksDB state backend.

Figure 3.14: JobManager threads
After generating the failures by the map operator, we will notice changes to the number of JobManager available threads due to rescheduling policy.

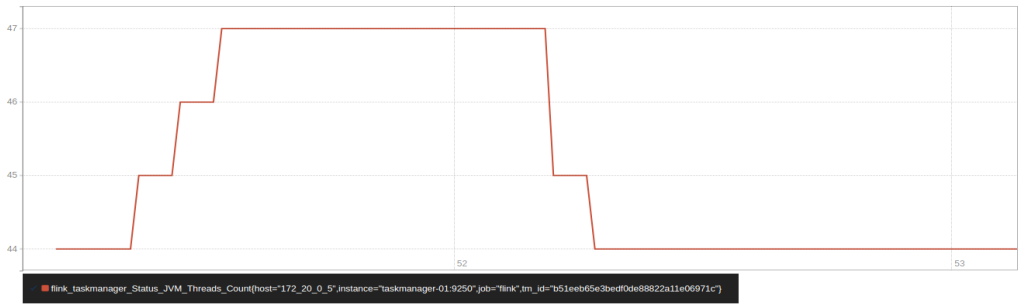

Figure 3.15: TaskManager threads
After generating the failures by the map operator, we will notice changes to the number of TaskManager available threads due to rescheduling policy.

Chapter 4

Deployment

The project was developed using Gradle and deployed using Docker to facilitate the execution in different environments. Running Flink in Docker will not have an impact on the performance in itself and it does not employ virtualization. The project application is deployed on docker containers that are isolated from one another. Using docker-compose as a container orchestration framework, it is possible to automatically configure the environment.

4.1 Application

Firstly we will build our `twitter-flink` images using the Dockerfile. Our cluster will be composed by a `JobManager` and a `TaskManager`. We can add more cluster resources by modifying `docker-compose` file. To build and run the containers, the following commands should be executed from the root of the project:

```
$ docker-compose build --no-cache
$ TESTING_CLASS='MyMainTestClass' docker-compose up -d --
   remove-orphans
```

Using the flag `TESTING_CLASS` will be possible to specify the entry point of the testing scenario. It will run the job by choosing the main class from the flag value. Due to environment configuration with only one TaskManager, is recommended to firstly stop a job before uploading a new one.

To access our image container and to run commands on its bash, the following command should be used:

```
$ docker exec -it sdprojectguriaa1920_client_1 bash
```

Running Flink on Docker containers has the disadvantages of the logs that can not be observed from the web UI dashboard, but in a new terminal window we can run the following commands in the foreground to track the job logs:

```
$ docker logs sdprojectguriaa1920_taskmanager-01_1 -f
```

A more detailed deployment and running specification will be added to `README` file on the project source repository with all the commands to generate each testing scenario.

4.2 Metric visualizzation

Apache Flink through its Web UI dashboard offers a set of metrics to visualize the statics regarding the environment and the resources. For a more interesting and rich visualization has been used Prometheous, It has multiple offers multiple modes for visualizing data with a built-in expression browser or Grafana integration.

The Prometheus[1] container has bee defined in the docker-compose file and it will be built and running with other containers too when the previous docker commands will be executed. The Prometheous web dashboard will be accessible on `http://localhost:9090/graph`.

[1]Prometheous doc: `https://prometheus.io/docs/`

Chapter 5

Conclusions

5.1 Evaluation

5.1.1 Pros

Apache Flink distinguishes from other technologies by many features and mechanisms it offers. The most important advantages Flink offers regarding other technologies out in the market are:

- **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 any manual configurations to get the best performance. Flink's pipelined data processing gives better performance compared to its counterparts.
- **Exactly-once guarantees**: 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. After a failure, a state in stateful operators should be correctly restored.
- **Flow control**: Backpressure from slow operators should be naturally absorbed by the system and the data sources to avoid crashes or degrading performance due to slow consumers
- **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.
- **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.

- **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.
- **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.
- **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.
- **Native closed-loop iteration operators**: Flink has dedicated support for iterative computations. It iterates data by using streaming Architecture. The concept of an iterative algorithm is tightly bounded into Flink query optimizer. Flink's pipelined architecture allows processing the streaming data faster with lower latency.
- **Unified Framework**: Flink is a unified framework which allows building a single data workflow that holds streaming, batch, SQL and Machine learning.

5.1.2 Cons

As new technology in the market, Flink has some disadvantages making it difficult to start working with.

- **Less community and forums for discussion**: Flink may be difficult to understand starting as a beginner because there are not many active communities and forums to exchange problems and doubt about Flink features.
- **Less open-source projects**: There are not many open-source projectd to study and practice Flink. The lack of code source makes it very difficult the familiarization with the most innovative features and mechanisms it offers.
- **Immaturity**: Immaturity in the industry is a disadvantage for Apache Flink, because is a new technology and many features are constantly being updated and modified. We should avoid Apache Flink, if we need a more matured framework compared to other competitors in the same space.
- **API support**: Apache Flink is not the best choice if we need more API support apart from Java and Scala languages.
- **Data representation**: Flink uses raw bytes as internal data representation, which if needed, can be hard to program.

5.2 What did we learn?

During this project we learned about:

- the Flink **architecture**
- innovative **mechanisms** as:
  1. Checkpoints
  2. Savepoints
  3. Watermarks
- most important **features** that distinguish Flink from other competitors
- most important **DataStream API** functionalities
- configuring the Flink **execution environment**
- using the Flink **Command-Line Interface**
- implementing tweets **streaming applications** with DataStream API
- testing Flink's **watermarking** and **checkpointing** mechanism

5.3 Future work

This project was the *"tip of the iceberg"* of Apache Flink technology and there so many aspects to be discovered and studied. Even if it's a immature technology, it totally grabbed our attention. Shortly we will focus on:

- integrating Apache Flink with external technologies like Apache Hadoop to deploy Flink of top of Hadoop's resource manager or Apache Kafka to generate the data.
- exploring `Metrics` for more efficient debugging and monitoring. Flink exposes a metric system that allows gathering and exposing metrics to external systems.
- exploring Table API & SQL. This is a language-integrated query API that allows the composition of queries in a very intuitive way.
- getting used with the important libraries offered by Flink:
  - *Event Processing (CEP)*: implemented to detect event patterns in an endless stream of events.
  - *State Processing API*: provides powerful functionality to reading, writing and modifying savepoints and checkpoints.
  - *Graphs Gelly*: contains a set of methods and utilities which aim to simplify the development of graph analysis applications in Flink.

List of Figures

Glossary

cluster of machines Set of connected computers that work together and can be viewed as a single system where each node perform s the same task.

dataflow Movement of data through a system comprised of software, hardware or a combination of both.

micro-batching Technique to group tasks into small batches to achieve some of the performance advantage of batch processing, without increasing the latency for each task completion too much. Typically is applied in system with a variable amount of incoming tasks.

Bibliography

[1] P. Carbone, A. Katsifodimos, Kth, S. Sweden, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. State management in apache flink: Consistent stateful distributed stream processing. *IEEE Data Engineering Bulletin*, 2015.

[2] P. Carbone, A. Katsifodimos, Kth, S. Sweden, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 2015.

[3] U. Celebi. How apache flink handles backpressure. 2015. URL `http://www.ververica.com/blog/how-flink-handles-backpressure`.

[4] T. Deshpande. *Learning Apache Flink*. Packt Publishing Ltd., 2017.

[5] A. F. Doc. Why apache flink – best guide for apache flink features. 2018. URL `https://ci.apache.org/projects/flink/flink-docs-master/concepts/flink-architecture.html`.

[6] A. F. Doc. Flink datastream api programming guide. 2018. URL `https://ci.apache.org/projects/flink/flink-docs-stable/dev/datastream_api.html`.

[7] A. F. Doc. Data streaming fault tolerance. 2018. URL `https://ci.apache.org/projects/flink/flink-docs-release-1.1/internals/stream_checkpointing.html`.

[8] A. F. Doc. Apache flink: Windows. 2018. URL `https://ci.apache.org/projects/flink/flink-docs-release-1.1/apis/streaming/windows.html`.

[9] E. W. Friedman and K. Tzoumas. Introduction to apache flink: Stream processing for real time and beyond. 2016.

[10] F. Hueske and V. Kalavri. *Stream Processing with Apache Flink*. O'Reilly Media, 2017.

[11] S. Richter. 3 differences between savepoints and checkpoints in apache flink. 2018. URL `https://www.ververica.com/blog/differences-between-savepoints-and-checkpoints-in-flink`.