**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN**

**KHOA CÔNG NGHỆ THÔNG TIN**

# Course seminar

# Topic: Apache Flink

**COURSE:** Introduction to Big Data

**Lecturer:**

Nguyễn Ngọc Thảo

Đoàn Đình Toàn

Nguyễn Khải Phú (20127062)

Thái Cẩm Phong (20127406)

Nguyễn Duy Thịnh (20127333)

Đỗ Đạt Thành (20127411)

**Năm:** 2023

# Contents

# 1. Introduction

Apache Flink is an open-source software, designed to perform computations and analysis over unbounded and bounded data streams. Historically, processing streams of data on a very large scale, across multiple sectors, has always been a challenge. But now, with the introduction of new technologies and architecture such as Flink, one can wish to analyze an extensive amount of real-time data in a timely fashion.

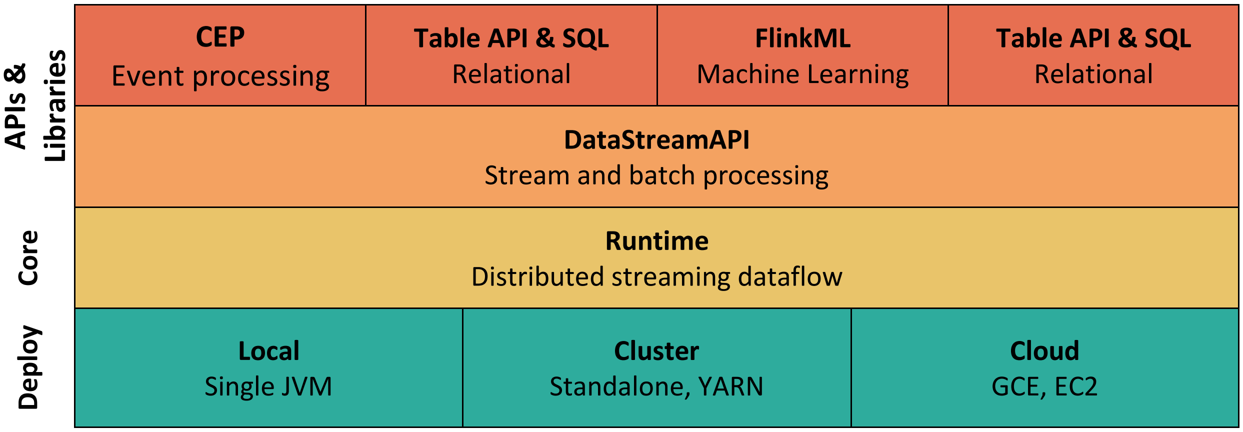
Apache Flink originated from Stratosphere, a collaboration of Technical University of Berlin, Humboldt University and Hasso Plattner Institute (all located in Berlin, Germany). In March 2014, Stratosphere was adopted by Apache Incubator, and got accepted as an Apache Software Foundation's top level project in December 2014. Stratosphere's main goal was to advance the state-of-the-art distributed, fault-tolerant data processing. In its 0.6 release (August 2014), Stratosphere was renamed to Flink, a general-purpose data processing engine for clusters. Throughout the years, more and more programmers joined Flink's open-source community and have greatly contributed to its development. Flink is integrated by multiple big companies and enterprises around the world, such as Amazon, Alibaba, ebay, HUAWEI,…

# 2. A deeper insight to the selected solution

## *2.1. Major components and main functionalities:*

### Major components

The figure below shows the technology stack of Flink with **3 layers**:



As illustrated in the figure, Flink does not come with a storage system. Users can integrate external databases/streaming systems (for example: HDFS, MongoDB, Hbase,…) and use Flink to collect and process data from those sources.

The lowest level of Flink is the **Deploy layer**. Flink can be deployed in 3 forms:

- Local – single node, single JVM
- Cluster – cluster of multiple nodes, can integrate with all common cluster resource managers, such as YARN, Apache Mesos, Kubernetes, or as a stand-alone
- Cloud – on Google's GCE or Amazon's EC2

The middle level of Flink is the **Runtime layer**, also the kernel of Apache Flink. This layer provides distributed processing with fault-tolerance and high reliability. Flink runtime consists of 2 types of processes:

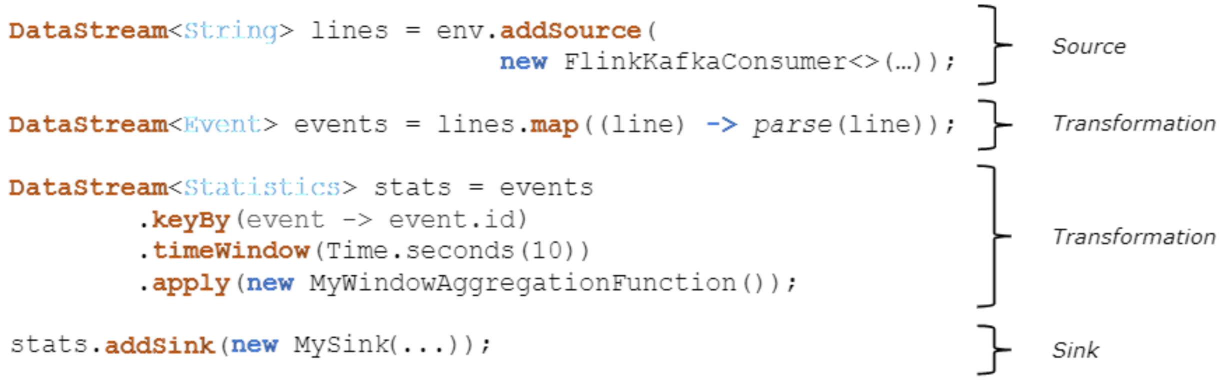
- JobManagers: schedule tasks, coordinate checkpoints and failure recovery
- TaskManagers: execute the tasks

The client is not a part of the Runtime layer, but is used to submit jobs to the JobManager.

The last layer is **APIs and libraries**, providing 2 core APIs along with a lot more tools for different kinds of needs.

- DataSet and DataStream API: implement transformations on data sets, batch (using DataSet) or data streams (using DataStream) and return the output via sinks, which write the data to files or standard output, like command line terminal.
- TableAPI: SQL-like expression language for relational stream and batch processing that can be embedded in DataSet and DataStream APIs.
- FlinkML: new product from Flink's open-source community, providing scalable ML algorithms and intuitive APIs to handle ML applications in Flink
- FlinkCEP: stands for complex event processing. This library allows easy detection of complex event patterns in a stream of endless data

## Stream processing with streaming dataflows



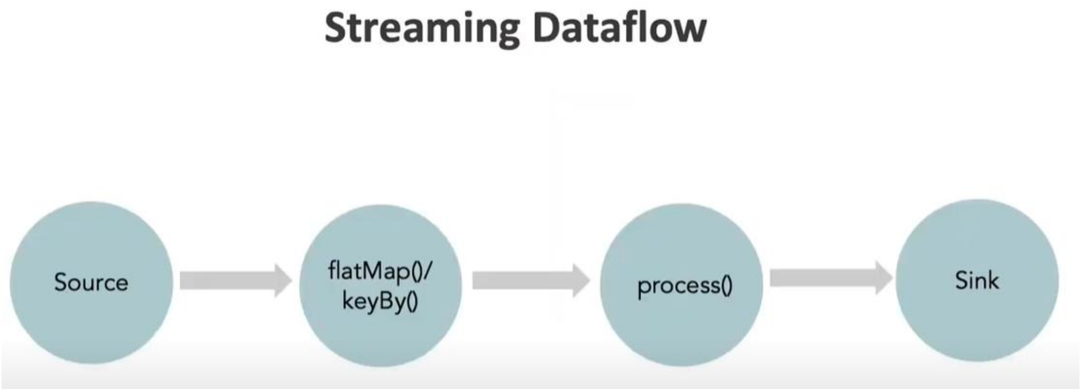
```
DataStream<String> lines = env.addSource(
                               new FlinkKafkaConsumer<>(…));

DataStream<Event> events = lines.map((line) -> parse(line));

DataStream<Statistics> stats = events
        .keyBy(event -> event.id)
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());

stats.addSink(new MySink(...));
```

*Example of a Flink program's code*

Let's say we have a code snippet of a Flink program as shown in the image above.

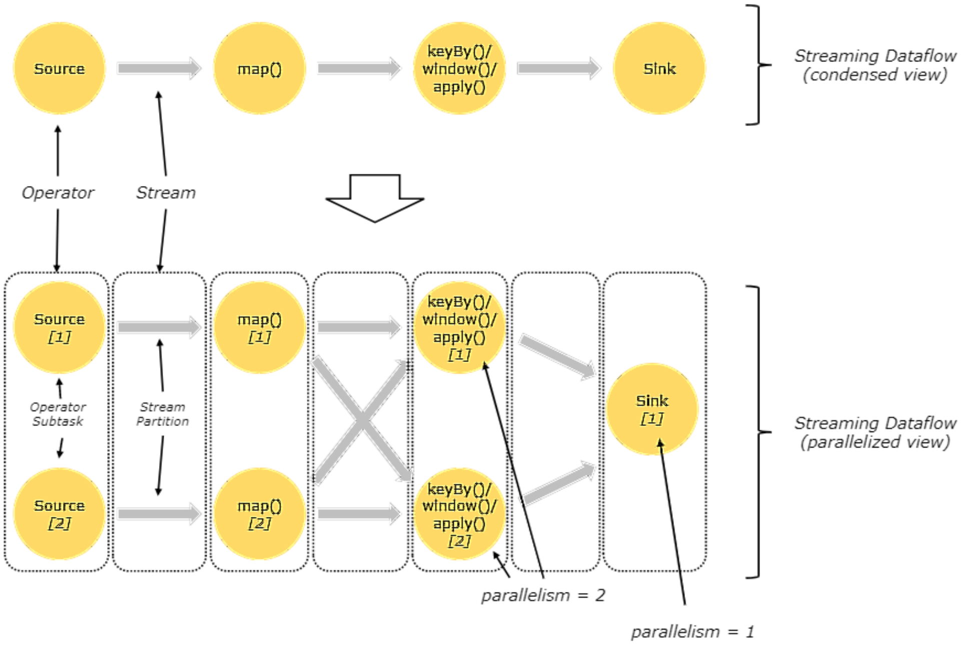
***Step 1:*** The jar file of this program is distributed amongst all nodes inside the Flink cluster.

***Step 2:*** Flink converts all the user-defined transformations into a **streaming dataflow**, which is a data structure that encloses the order of operations applied to the input stream. This dataflow can be represented as a directed graph that always starts with source(s) and ends at sink(s).



*A streaming dataflow*

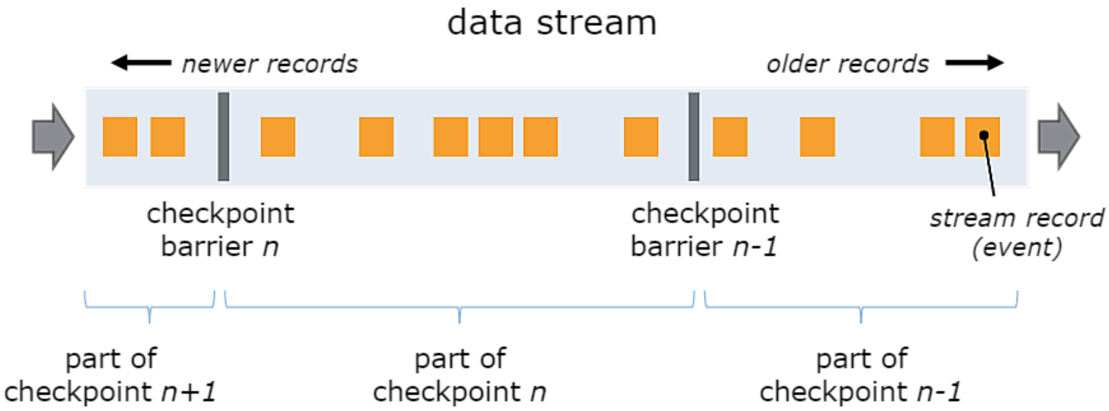
***Step 3:*** Flink converts the dataflow into an execution graph. This graph not only depicts how the abstract streaming dataflow is mapped to the system's available resources, it also shows the level of parallelism of the job's execution process.



*How a streaming dataflow is converted into an execution graph*

## Fault-tolerance with Snapshots

Flink periodically inserts "markers" into the stream, flowing along with the data. These markers create barriers that split the data stream into 2 parts, containing either old records or new ones. Using this technique, Flink can distinguish between different states of an application at different points in time.



*Checkpoint barriers are inserted into the data stream to separate different checkpoints.*

In Flink's terminology, markers are *checkpoint barriers* and parts are *checkpoints*. Checkpoint n encapsulates the state of the operators resulted from having consumed all events happening **only before** checkpoint barrier n. It works like loading a computer game from a saved version. These **snapshots** are stored inside a "fault-tolerant storage", which can be built from a HDFS or Amazon S3. This mechanism provides the abilities to, literally, *time travel*. By accessing the storage, Flink users can go back in time and recover an earlier but consistent version, or experiment with different implementations at a desired point in time.

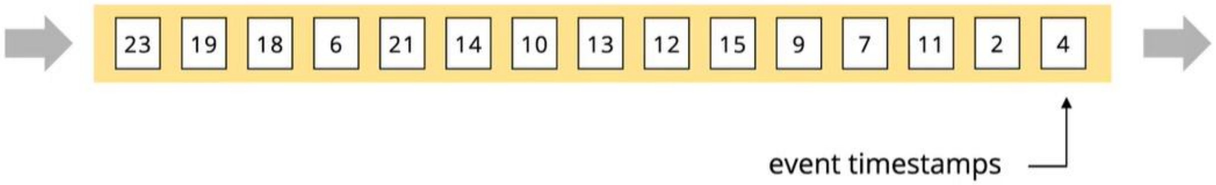
### Event time and Watermark

Flink supports different notions of time:

- Event time: the time when an event actually occurred, recorded by the edge device producing the event.
- Ingestion time: the time when an event is ingested by Flink. If Flink needs to reprocess the data from the message broker due to failure, this type of timestamp cannot be reproduced.
- Processing time: the system clock time when a specific operator processes the event.

As seen from these descriptions, event time is superior for performing most kind of computations and analytics. However, it does come with a cost of introducing complexity since the events can arrive at the operators **out of order**. The solution for this problem is **Watermark**.

Let's look at an example of a stream of events with timestamps.

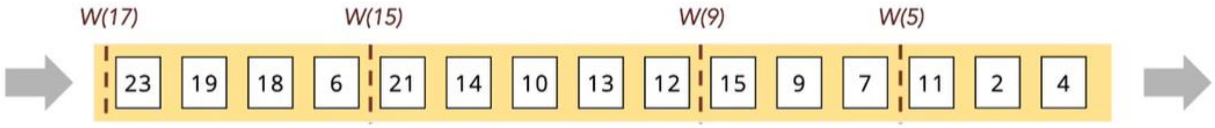


*A stream of elements that receives input out of chronological order*

We can see from the image above that the stream is out of order. Imagine that we are trying to output the same stream of events, but sorted by timestamps. The first element is a 4, and it needs to be delayed at least until the 2 arrives. After these 2 elements are emitted, 11 comes in. From this view provided by the image, we can see *ahead in time* that Flink needs to wait for 7, 9, 10 and 6 to arrive first. But in hindsight, it is very diffcult to know when the results are ready after a while.

If there is a chance that 3, 5 or 8 comes in later, Flink will have to decide whether to wait for those elements, or just skip them to maintain low latency, since they are not likely to exist.

Flink uses watermark as the solution for this complex problem. Watermarks define when to stop waiting for more events. Watermark generators inject watermarks into the stream that flow along with the data.



*Watermarks injected into the stream with maximum delay = 6*

Flink views the out-of-order problem as if each event can arrive after a certain amount of delay, and these delays vary between each element. Developers can configure the maximum threshold on this delay. In the example above, the maximum delay is set to 6. The watermark coming right after element 15 will assert that all following events is probably higher than 15-6=9, thus the name W(9). Element 6 arrives long after W(9), and will be marked as *late*.

Flink refers to this approach as *bounded-out-of-orderness*. Developers use bounded-out-of-orderness to find the optimal compromise between latency and precision. Flink also provides methods to handle late events. More of this will be introduced in the next part.

## Windows

When processing streams of data, we often face the tasks of doing aggregate analytics on different parts of the stream, such as: count total items viewed per hour, find the maximum temperature per minute, number of users per week,… In order to calculate these numbers, an analytics software needs to have the ability to slice the data stream into buckets with varying sizes based on the users' needs. In Flink, this concept is referred to as *Windows.* Each windowed Flink program needs to include 2 main components: *Window Assigner* and *Window functions.*

A Window Assigner defines how its elements are assigned to windows. Flink introduces some pre-implemented window assigner types:

- *Global windows*: assign all elements with the same key to the same window. This kind of window does not have a predetermined end, thus no computation can be done on these global windows, unless developers have already specifed some kind of triggers (discussed later in the section).

- *Tumbling windows*: assign elements to non-overlapping windows of a fixed size. These windows do not overlap

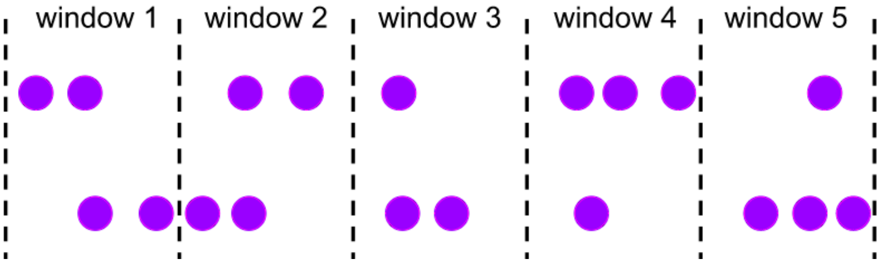


*Illustration of tumbling windows*

- *Slide windows*: very similar to tumbling windows, but these windows overlap. An additional *window slide* parameter is responsible for deciding how much these windows overlap with each other. For example, say we have windows of size 10 minutes and slide of 5 minutes. This means we get a window containing events in the last 10 minutes every 5 minutes

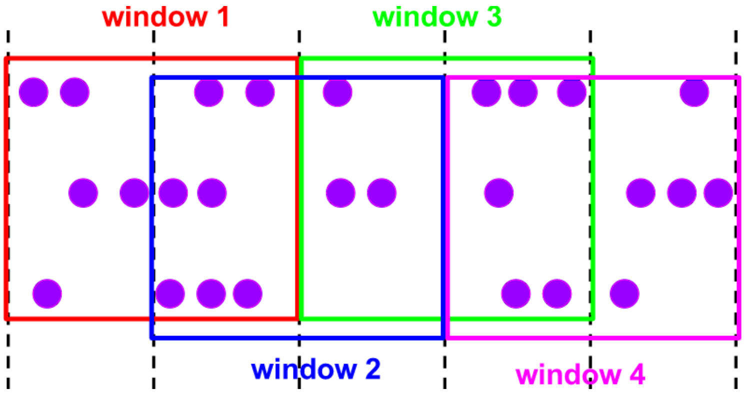


*Illustration of slide windows*

- *Session windows*: assign elements by sessions of activity. A session window closes when it stops receiving elements after a certain amount of time.

Window functions define the type of computations to be performed on each window. Flink gives 3 main types of window functions:

- *ProcessWindowFunction*: receives an iterable of all elements of the window, along with a Context object containing time and state information,

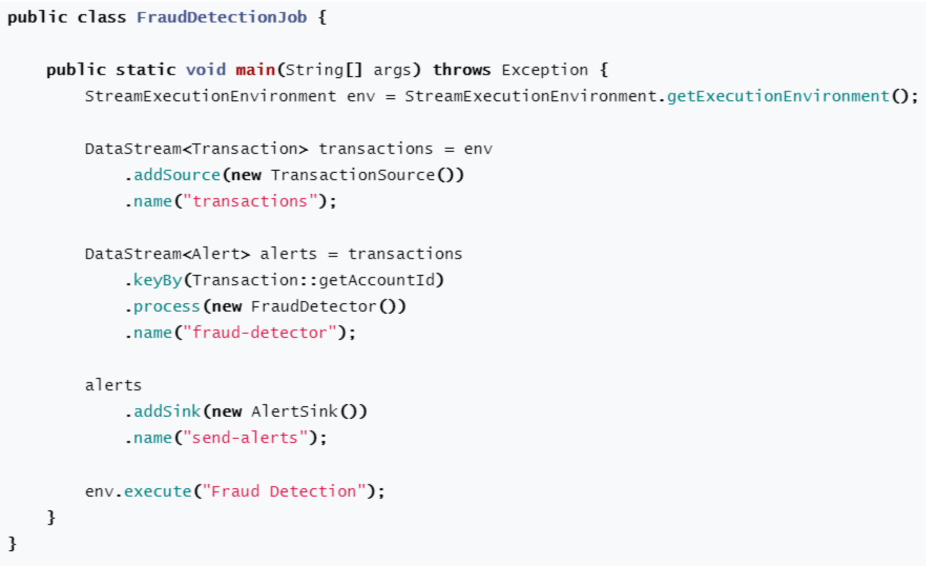
- *ReduceFunction* and *AggregateFunction*: incrementally aggregate the elements of a window.
- Mixture of both types

Flink also provides *Triggers* and *Evictors* in addition to these components. Triggers can react to certain events in a window and starts a window function. Flink comes with 4 basic types of Triggers: *onElement(), onEventTime(), onProcessingTime(), onMerge()*, but users can also design custom Triggers based on their tasks. These methods will be called when their respective invocation events happen, and trigger some computations specified by the users. Evictors can remove elements from a window after a trigger fires and before/after a window function has run.

When working with Event-time window, sometimes elements arrive after a big delay, and is marked as late by the Watermark's policy, mentioned in the Watermark section. These late elements will be dropped by default, but users can configure a "timer" method called *AllowedLateness* to make Flink hold the state of the window until the timer expires. Late data that arrives during the allowed lateness can be collected by the *sideOutputLateData* and pushed to another output stream.

## Event-driven application

Let's use an example of a fraud detection system built on Flink. These code snippets are provided by Apache Flink's documentation.

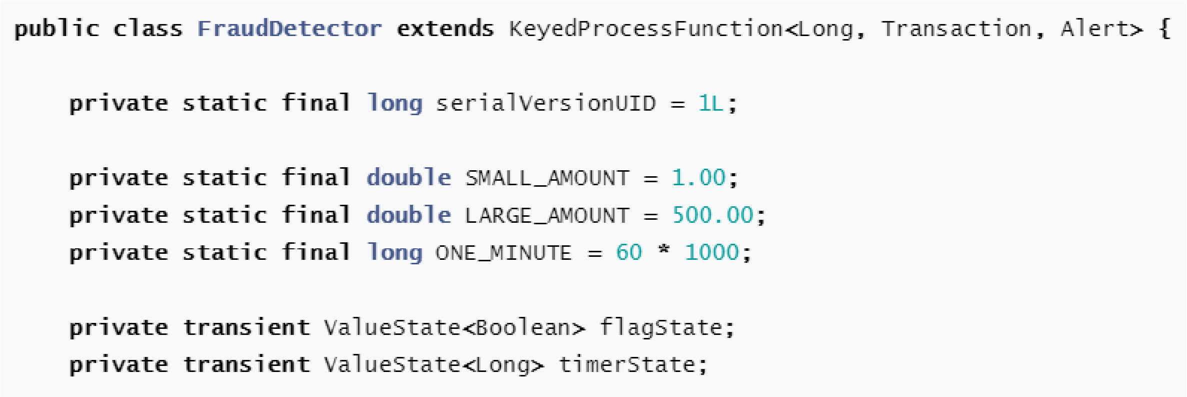


```
public class FraudDetectionJob {

    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        DataStream<Transaction> transactions = env
            .addSource(new TransactionSource())
            .name("transactions");

        DataStream<Alert> alerts = transactions
            .keyBy(Transaction::getAccountId)
            .process(new FraudDetector())
            .name("fraud-detector");

        alerts
            .addSink(new AlertSink())
            .name("send-alerts");

        env.execute("Fraud Detection");
    }
}
```

*Method main() in class FraudDetectionJob*

**The first line** in this *main()* function creates a *StreamExecutionEnvironment* object, which is going to set the properties and sources for our Job. **The second line** defines the data source from which our system can consume the events. Most common external systems used as data sources for Flink include Apache Kafka, Rabbit MQ or Apache Pulsar. **The third line** is usually referred to as *data transformation*. The *keyBy()* operator specifies the key for every element. The *process()* method calls the function from its parameter (*FraudDetector()* in this case) on the keyed data stream. **The fourth line** sets the *sink*, which is the destination that the ouputs go to.

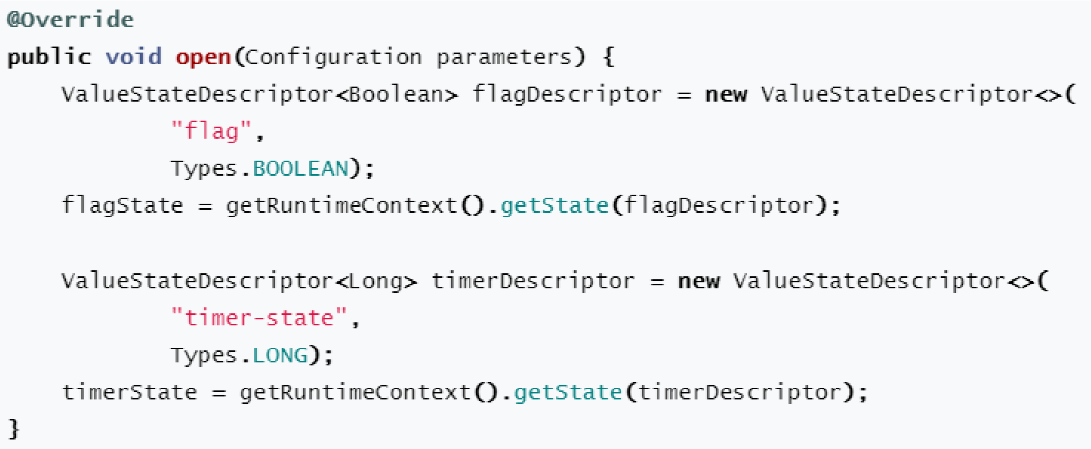
Now we take a dig into *FraudDetector()* to see how the business logic of this system is implemented. In this example, an account is marked as a fraud when it makes a small transaction (less than $1.00) immediately followed by a big one (more than $500).



```
public class FraudDetector extends KeyedProcessFunction<Long, Transaction, Alert> {

    private static final long serialVersionUID = 1L;

    private static final double SMALL_AMOUNT = 1.00;
    private static final double LARGE_AMOUNT = 500.00;
    private static final long ONE_MINUTE = 60 * 1000;

    private transient ValueState<Boolean> flagState;
    private transient ValueState<Long> timerState;
```

*Variables defined in class FraudDetector*

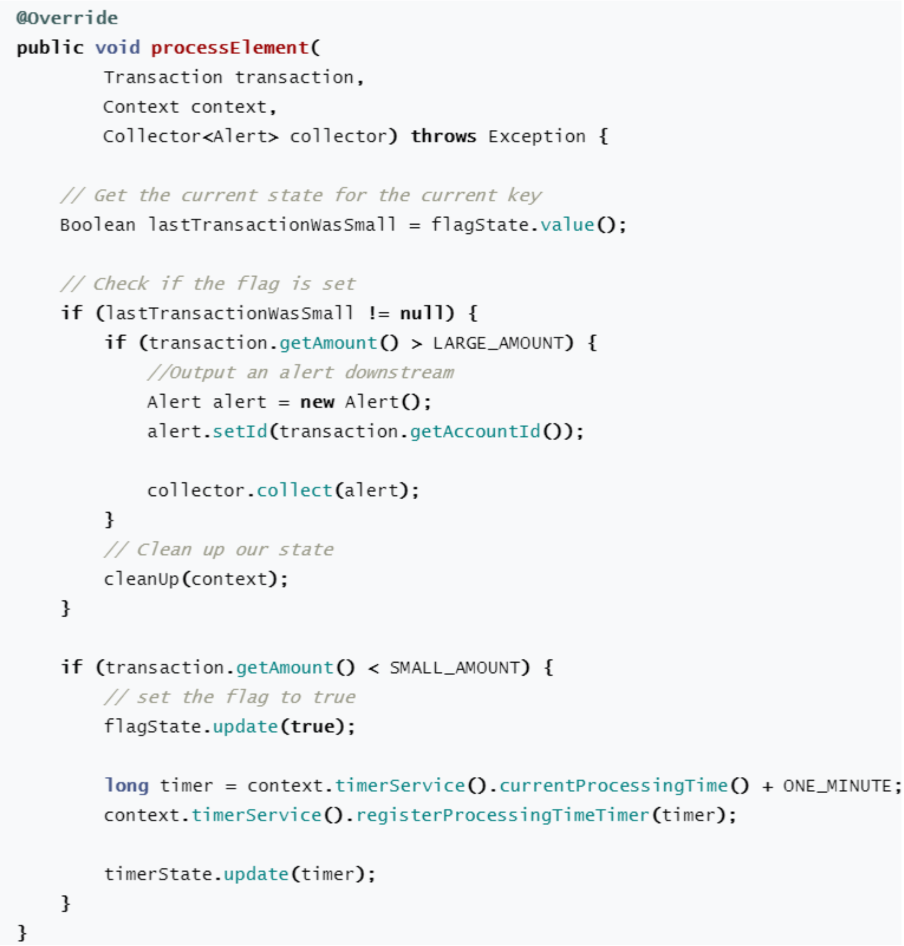
Usually, boolean flags are used for this kind of task. The flag is set to True when a small transaction comes in, and the flag is checked whenever a big one arrives. However, this system has to process transactions from multiple accounts, so a variable simply won't work. To address these problems, Flink provides *ValueState*, a fault-tolerant data type that will be automatically scoped to the key of the elements being processed. Our *flagState* variable from the image above not only maintains an independent state for each account, it can also be recovered by Flink in the case of component failures. Another variable called *timerState* specifies the timeout window which, if expires, will reset the flag to null.



```
@Override
public void open(Configuration parameters) {
    ValueStateDescriptor<Boolean> flagDescriptor = new ValueStateDescriptor<>(
            "flag",
            Types.BOOLEAN);
    flagState = getRuntimeContext().getState(flagDescriptor);

    ValueStateDescriptor<Long> timerDescriptor = new ValueStateDescriptor<>(
            "timer-state",
            Types.LONG);
    timerState = getRuntimeContext().getState(timerDescriptor);
}
```

*Registering flagState and timerState*

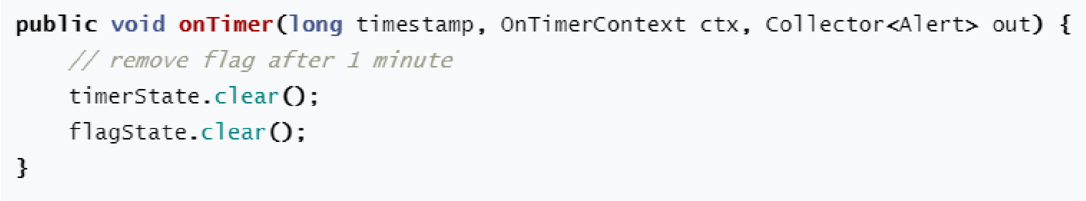
Flink always looks for the *open()* method when starting the Job. Here, we register *flagState* and *timerState* with the framework using *getState()* coupled with a *ValueStateDescriptor*. This way, Flink knows that these variables should be managed as a *ValueState*.



```
@Override
public void processElement(
        Transaction transaction,
        Context context,
        Collector<Alert> collector) throws Exception {

    // Get the current state for the current key
    Boolean lastTransactionWasSmall = flagState.value();

    // Check if the flag is set
    if (lastTransactionWasSmall != null) {
        if (transaction.getAmount() > LARGE_AMOUNT) {
            //Output an alert downstream
            Alert alert = new Alert();
            alert.setId(transaction.getAccountId());

            collector.collect(alert);
        }
        // Clean up our state
        cleanUp(context);
    }

    if (transaction.getAmount() < SMALL_AMOUNT) {
        // set the flag to true
        flagState.update(true);

        long timer = context.timerService().currentProcessingTime() + ONE_MINUTE;
        context.timerService().registerProcessingTimeTimer(timer);

        timerState.update(timer);
    }
}
```

*Method processElement()*

*processElement()* specifies what should be done with each element coming in. Here we implement the aforementioned flag logic. *timerService().registerProcessingTime()* registers a one-minute timer, and the method *onTimer()* in the image below will reset the flag once the timer fires.



```
public void onTimer(long timestamp, OnTimerContext ctx, Collector<Alert> out) {
    // remove flag after 1 minute
    timerState.clear();
    flagState.clear();
}
```

*Method onTimer()*

## 2.2. Its applications in academic and/or industry activities

Apache Flink excels in high-concurrency stream-processing with millisecond-level latency, high throughput, fault-tolerance. Flink is currently a top open-source stream processing engine in the industry. Dozens of companies use Flink to handle their processes involving real-time data, such as anomaly detection, business process monitoring, financial analytics, log aggregation,… Here are some examples of business-critical applications powered by Apache Flink:

- Amazon Kinetic Data Analysis: Amazon’s cloud service for stream processing. It uses Flink to power its Java application capability
- Gojek: uses Flink to improve data-driven decisions across functions
- OPPO: integrates Flink into its real-time data warehouse in order to “analyze the effects of operating activities and short-term interests of users”
- SK Telecom: uses Flink for several applications, such as smart factory, mobility applications
- Uber: built AthenaX, its internal streaming analytics platform, on Flink

Not only companies, many universities and research institutes around the world use Flink for research and educational purposes. For example: Technical University of Berlin, Leipzig University, University of Zagreb, Big Data Europe,…

A good way to learn what we can achieve from Apache Flink’s excellent features set is to look into how **companies and organizations** are **utilizing Flink’s** capabilities in their business.

### *2.3. Popularity*

Over 50 companies around the globe, including big names such as Amazon, Alibaba, HUAWEI, choose Flink to facilitate their business operations. A lot more universities and institutes are using Flink in their lectures, as well as doing research on Flink's capabilities in the Big Data field.

### *2.4. Other similar solutions*

#### Apache Spark

Spark is an open-source, multi-language engine designed to process large amounts of real-time data, execute large-scale data analytics. It is deployed in a stand-alone mode or on top of many other distributed computing frameworks.

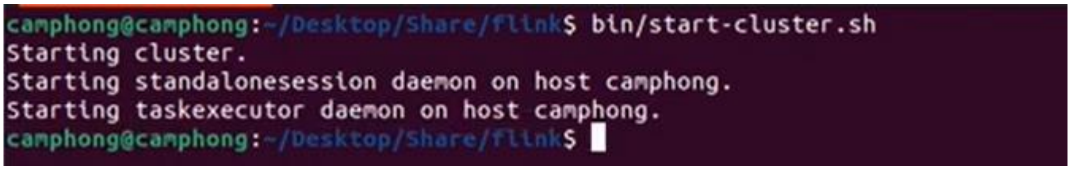
Spark is well-developed, has wider usage than Flink and is used by a lot more companies and enterprises. Flink, while being less mature than Spark, is superior in terms of performance, latency, scalability and automaticity. Both technologies are compatible with locally adapted applications in one unified region.

#### Apache Storm

Storm is an open-source, distributed, stream-processing system. Storm's pipelined engine looks a bit similar to Flink, and they both aim for low-latency stream-processing. However, Storm does not come with batch capabilities, unlike Flink being suitable for both forms. Moreover, Flink offers higher level APIs, which means a lot of Flink's APIs must be manually implemented when using Storm.

## 3. Demonstration

This demonstration includes a batch-processing job and a stream-processing job for the Word Count problem. In order to execute this job, we will deploy Flink on local mode by running *"bin/start-cluster.sh"* on the terminal

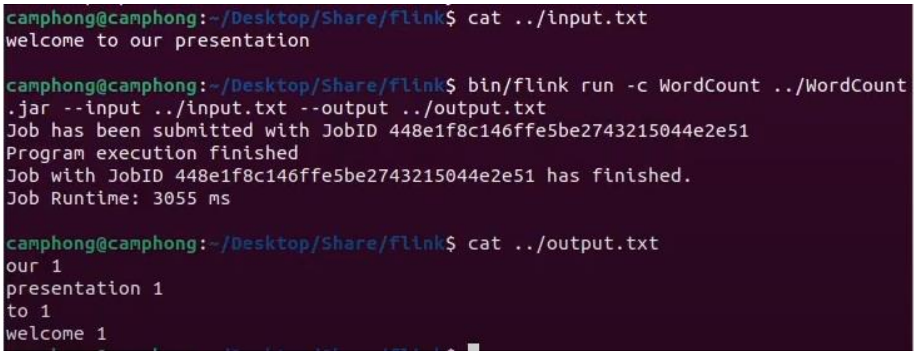


```
camphong@camphong:~/Desktop/Share/flink$ bin/start-cluster.sh
Starting cluster.
Starting standalonesession daemon on host camphong.
Starting taskexecutor daemon on host camphong.
camphong@camphong:~/Desktop/Share/flink$ 
```

*Deploying Flink*

### 3.1. Batch-processing

Run “*bin/flink run*” and specify the main class, .jar file, input and output path. The code for this can be found in the *WordCount batch* folder in this submission.



```
camphong@camphong:~/Desktop/Share/flink$ cat ../input.txt
welcome to our presentation

camphong@camphong:~/Desktop/Share/flink$ bin/flink run -c WordCount ../WordCount
.jar --input ../input.txt --output ../output.txt
Job has been submitted with JobID 448e1f8c146ffe5be2743215044e2e51
Program execution finished
Job with JobID 448e1f8c146ffe5be2743215044e2e51 has finished.
Job Runtime: 3055 ms

camphong@camphong:~/Desktop/Share/flink$ cat ../output.txt
our 1
presentation 1
to 1
welcome 1
```

*Run the WordCount batch job on the terminal. The last line is the result.*

### 3.2. Stream-processing

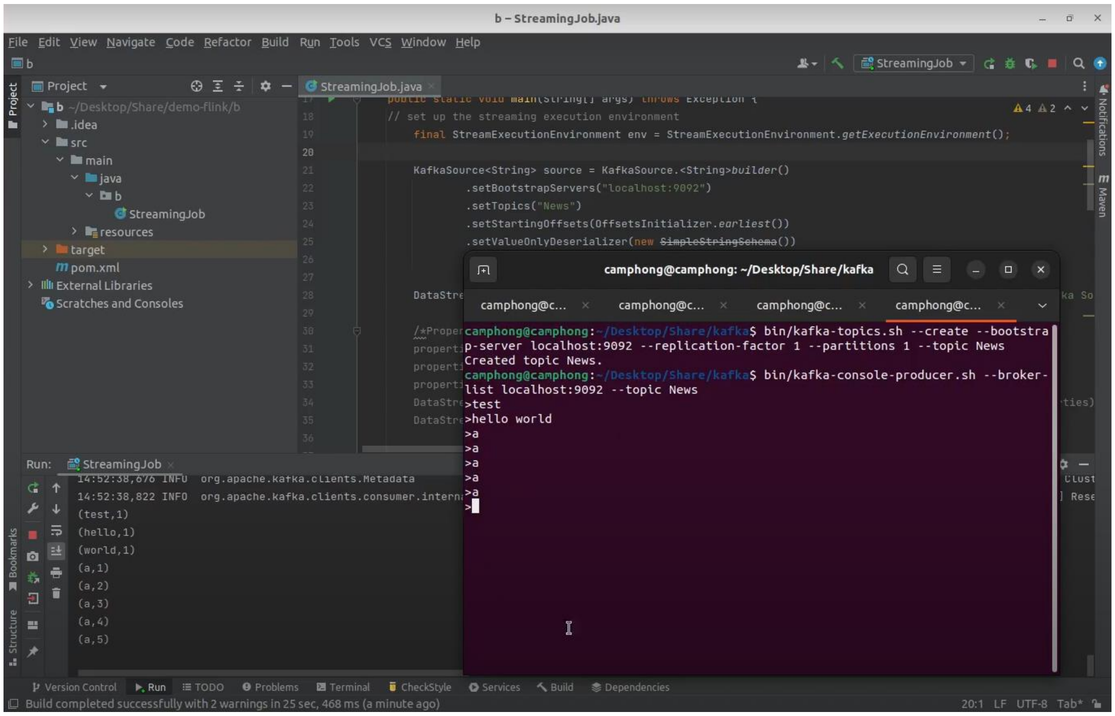
Apache Kafka will be used as the data source for our stream-processing demonstration. Follow these steps to set up our data stream:

1) Open terminal in Kafka folder and run *“bin/zookeeper-server-start.sh config/zookeeper.properties”*
2) Open another terminal and run *“bin/kafka-server-start.sh config/server.properties”* to start the kafka server
3) Open another terminal and run *“bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic News”*. This line create a *topic* with the name *News*
4) Open another terminal and run *“bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic News”*. Now we can start typing messages on Kafka

For the sake of simplicity, we are not going any further into Kakfa’s installation and execution.

Next, to connect Flink with Kafka, we need to specify the dependencies inside *pom.xml*. Our submission also includes this file in the same folder as the *StreamingJob.java*, which is the source code for our stream-processing WordCount. The correct dependencies can be found on Maen repository[16].

Finally, open an IDE, import the project and run. In this example, we will be using IntelliJ IDEA.



```java
    public static void main(String[] args) throws Exception {
    // set up the streaming execution environment
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        KafkaSource<String> source = KafkaSource.<String>builder()
                .setBootstrapServers("localhost:9092")
                .setTopics("News")
                .setStartingOffsets(OffsetsInitializer.earliest())
                .setValueOnlyDeserializer(new ~~SimpleStringSchema~~())
```

```
camphong@camphong:~/Desktop/Share/kafka$ bin/kafka-topics.sh --create --bootstra
p-server localhost:9092 --replication-factor 1 --partitions 1 --topic News
Created topic News.
camphong@camphong:~/Desktop/Share/kafka$ bin/kafka-console-producer.sh --broker-
list localhost:9092 --topic News
>test
>hello world
>a
>a
>a
>a
>a
>
```

```
14:52:38,676 INFO  org.apache.kafka.clients.Metadata
14:52:38,822 INFO  org.apache.kafka.clients.consumer.intern
(test,1)
(hello,1)
(world,1)
(a,1)
(a,2)
(a,3)
(a,4)
(a,5)
```

```
Build completed successfully with 2 warnings in 25 sec, 468 ms (a minute ago)
```

*Running our stream-processing WordCount. Words typed on kafka is counted and display as key-value pairs in real-time on IntelliJ IDEA's terminal.*

## 4. References

[1] https://nightlies.apache.org/flink/flink-docs-release-1.17/

[2] https://flink.apache.org – Flink

[3] https://www.youtube.com/@ververica1483 – Ververica

[4] https://nexocode.com/blog/posts/what-is-apache-flink/ - nexocode

[5] https://www.oreilly.com/library/view/introduction-to-apache/9781491977132/ch01.html - O'Reilly

[6] https://github.com/apache/flink - Apache Flink Github

[7] https://en.wikipedia.org/wiki/Apache_Flink - Wikipedia

[8] http://stratosphere.eu – Stratosphere

[9] https://data-flair.training/blogs/apache-flink-ecosystem-components/ - DataFlair

[10] https://forum.huawei.com/enterprise/en/fi-components-basic-principle-of-flink/thread/599282-893 - HUAWEI Forum

[11] https://cwiki.apache.org/confluence/display/FLINK/Powered+by+Flink – Atlassian Confluence

[12] https://spark.apache.org – Apache Spark

[13] https://cloudinfrastructureservices.co.uk/apache-spark-vs-flink-whats-the-difference/ - Cloud Infrasturcture Services

[14] https://www.tutorialspoint.com/apache_flink/apache_flink_conclusion - tutorials-point

[15] https://github.com/mnm1331/flink-streaming-batch-example - Github user mnm1331

[16] https://mvnrepository.com – Maven repository