Experiment report: Stream Processing and Flink

Done by student: **Bylkova Kristina (伯汀娜), 1820249049**
Course: **Big Data Analysis Technology**
Date: **October 2024**

Summary

This report shows the MindMap of stream processing and flink. Here is also the experiment screenshots.

Stream Processing MindMap

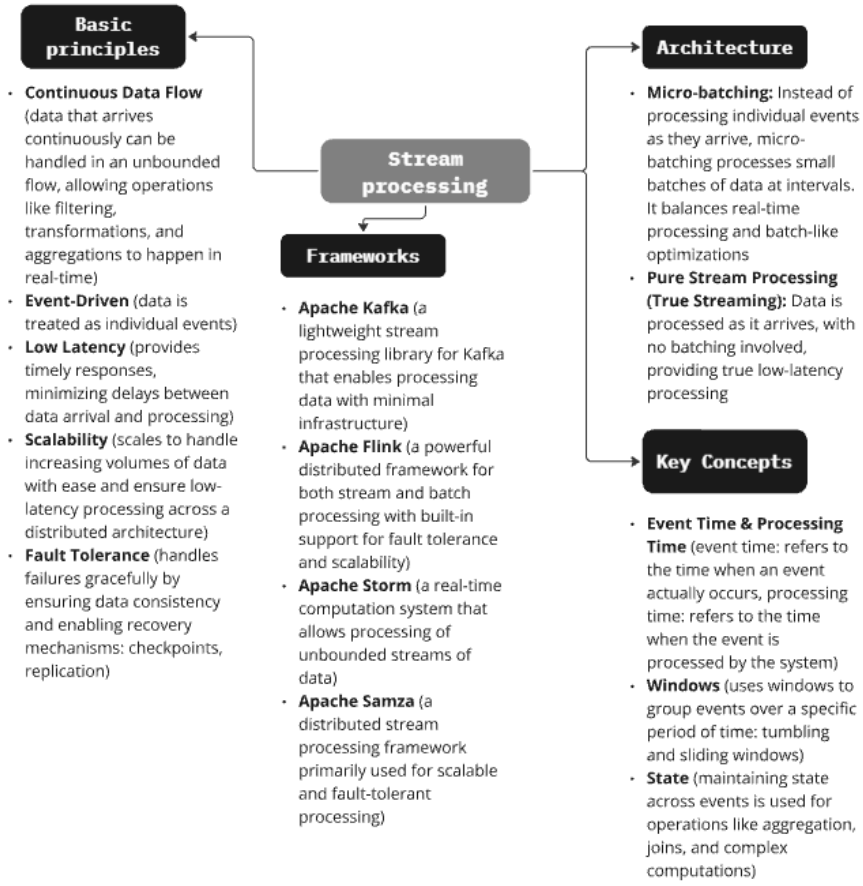

Figure 1: Stream Processing Mindmap

Flink MindMap

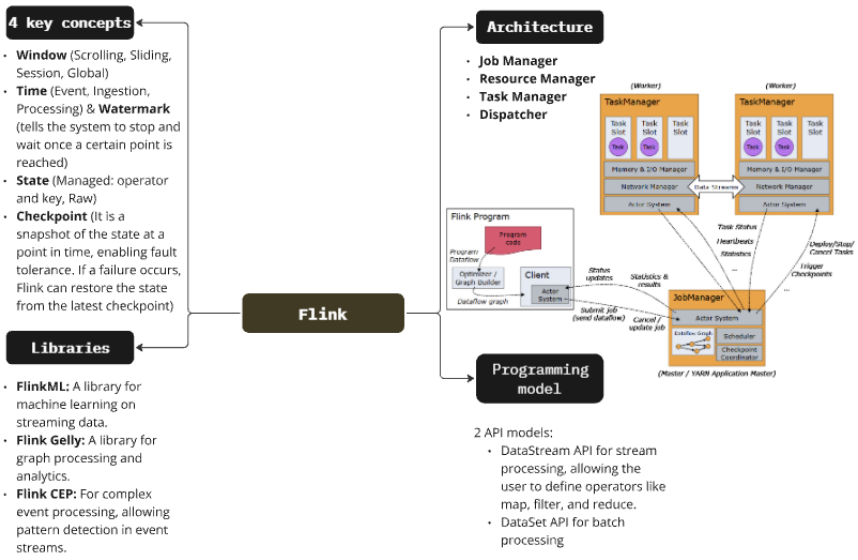

Figure 2: Flink MindMap

Task

1. Explain and illustrate the concepts of State and Checkpoint, as well as their working principles, and how the fault recovery can be achieved through State and Checkpoint. (using text and the diagram)

2. With reference to the Flink architecture diagram, explain the components and working principles of two architectures in Flink's YARN mode: the task submission architecture and the runtime architecture. (Flink runtime architectures include Local mode, Standalone cluster mode, and YARN mode).

Answer:

1. State is updated as events are processed. Each operator maintains some internal state as it processes each incoming event. Checkpointing occurs at regular intervals (configured by the user), where the system records the state of each operator. This means that at each checkpoint, a snapshot of the current state is written to a durable store.

If a failure occurs, the system will recover by using the most recent checkpoint (rolling back the system to the last successful checkpoint).

The fault recovery process works:

- Failure Detection (when a failure occurs, the system detects the failure)

- State Restoration (the system will attempt to recover by restoring the state from the last successful checkpoint)
- Reprocessing (after restoring the state, the system can resume processing from where it left off, reprocessing the unprocessed records)

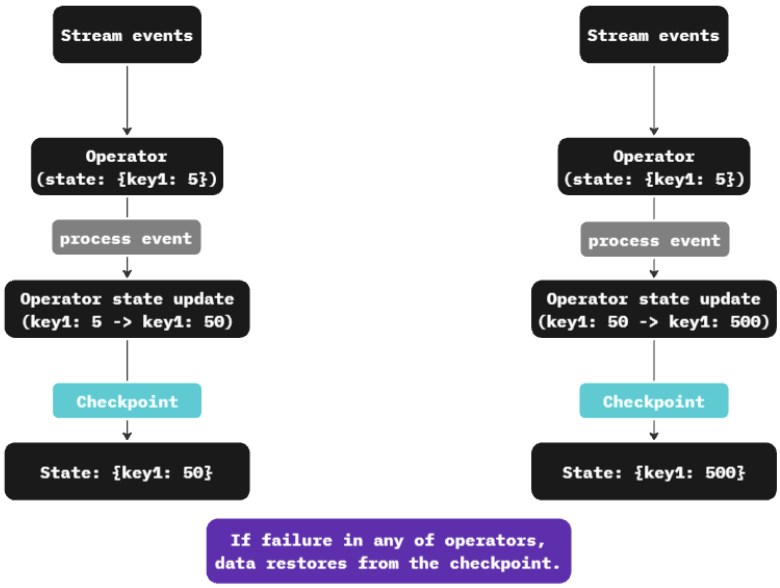

Figure 3: State and Checkpoint

2. **Task Submission Architecture:**
Components:

- Client (submit the Flink job to the YARN cluster)
- ResourceManager (manages resources across all available nodes in the cluster and allocates them for various applications)
- ApplicationMaster (manage the life cycle of the Flink job, including resource allocation and monitoring)
- JobManager (schedule tasks, distribute them to the TaskManagers, and manage job execution)

The client submits a Flink job to YARN using the JobManager. The ResourceManager assigns an ApplicationMaster to the job. The ApplicationMaster requests resources (containers) for the TaskManagers from the ResourceManager. Once resources are allocated, the JobManager coordinates the execution of tasks on the available TaskManagers.

Flink runtime architecture:

Components:

- JobManager coordinates the job execution. It is responsible for scheduling tasks, managing state, and recovering from failures
- TaskManagers are the worker nodes responsible for executing the tasks assigned by the JobManager
- The ResourceManager is responsible for managing the resources in the YARN cluster

In Local Mode, Flink runs on a single machine for development and testing purposes. In Standalone Cluster Mode, Flink is deployed across multiple machines or containers, where it runs in a distributed mode, but without any external resource manager like YARN or Mesos. In YARN Mode, Flink runs as an application within a YARN (Yet Another Resource Negotiator) cluster. YARN manages the resources across the cluster, and Flink leverages it to allocate resources for job execution.

Running

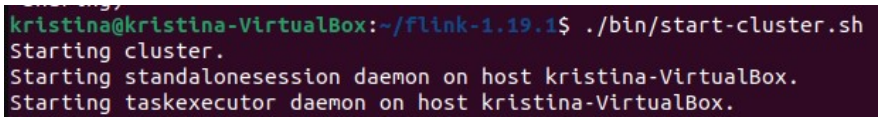

Figure 4: In terminal

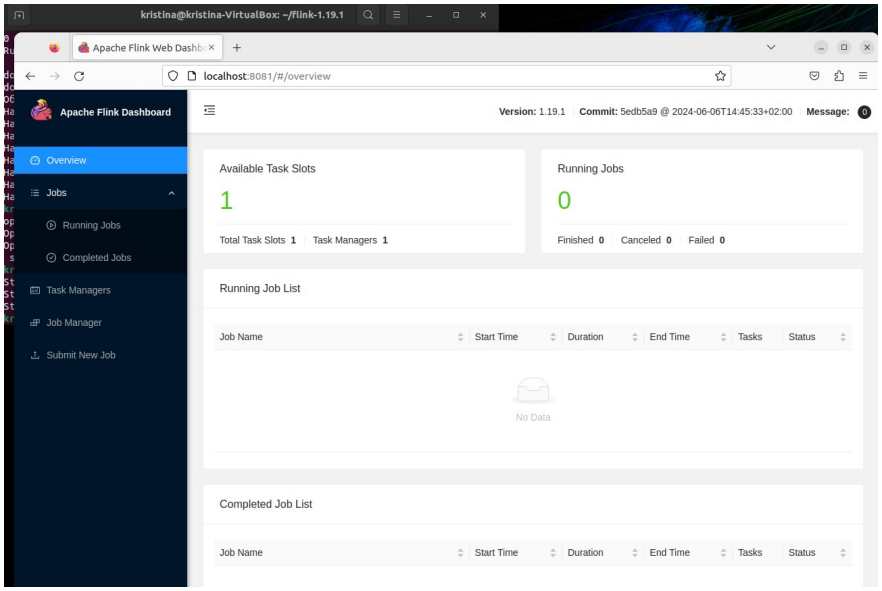

Figure 5: Web page

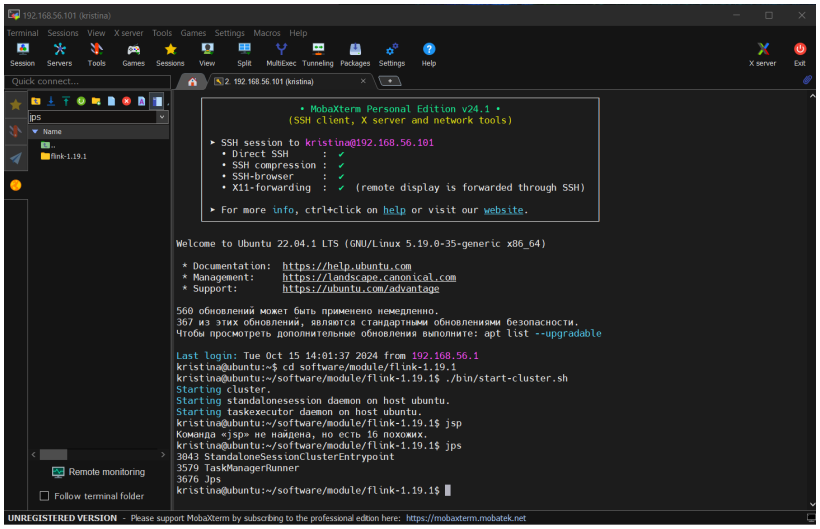

Figure 6: Connection via MobaXTerm

Flink operations

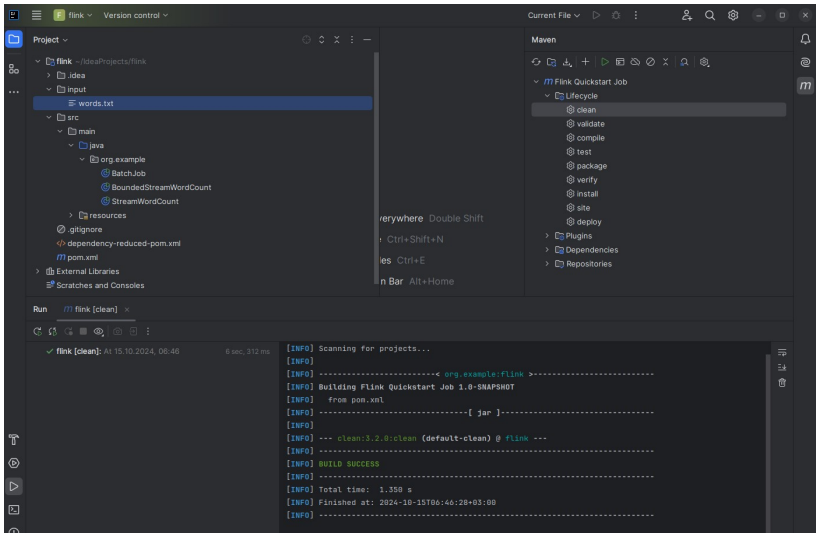

Figure 7: Clean

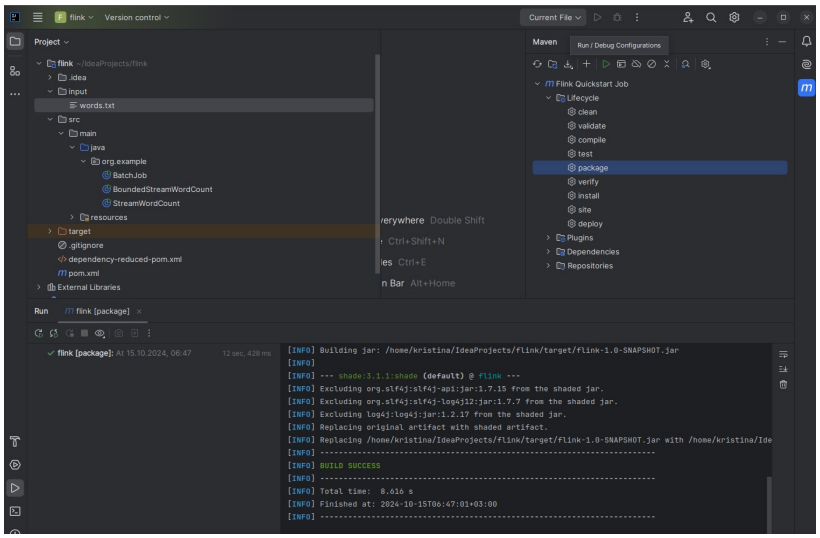

Figure 8: Package

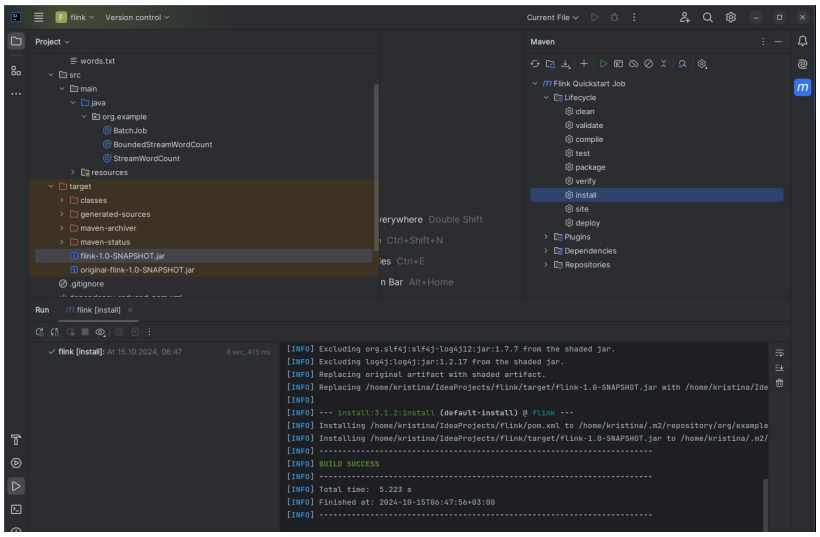

Figure 9: Install

Experiment

BatchJob experiment

```java
package org.example;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.operators.DataSource;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.util.Collector;


public class BatchJob {

  public static void main(String[] args) throws Exception {
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
    DataSource<String> stringDataSource = env.readTextFile("input/words.txt");

    DataSet<Tuple2<String, Integer>> counts =
        stringDataSource.flatMap(new Tokenizer())
            .groupBy(0)
            .sum(1);
    counts.print();

  }

  public static class Tokenizer implements FlatMapFunction<String, Tuple2<String, Integer>> {

    @Override
    public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {
      String[] tokens = value.toLowerCase().split("\\W+");

      for (String token : tokens) {
        if (token.length() > 0) {
          out.collect(new Tuple2<>(token, 1));
        }
      }
    }
  }
}
```

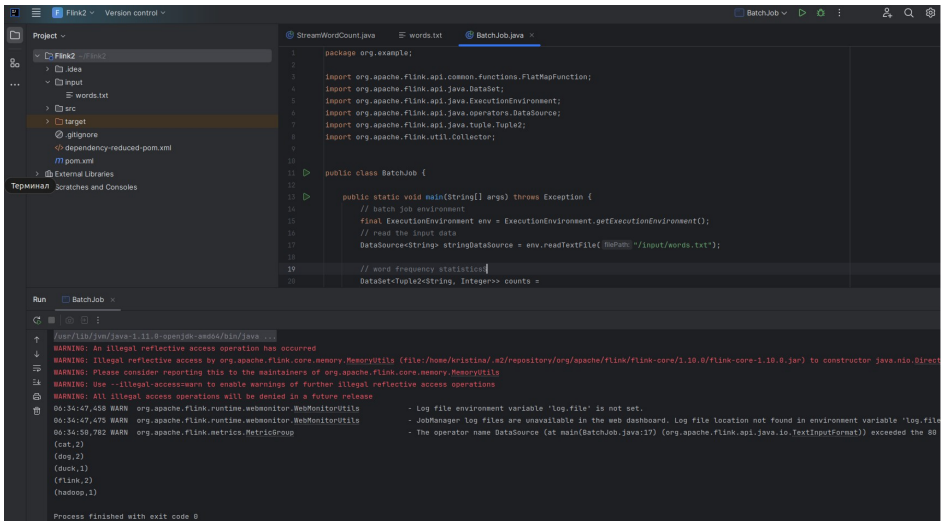

Figure 10: BatchJob execution

BoundedStreamWordCount experiment

```java
package org.example;

import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.datastream.KeyedStream;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.util.Collector;

import java.lang.reflect.Type;


public class BoundedStreamWordCount {

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

    DataStreamSource<String> lineDataStreamSource = env.readTextFile("input/words.txt");

    SingleOutputStreamOperator<Tuple2<String, Long>> wordAndOneTuple =  lineDataStreamSource.flatMap((
     String line, Collector<Tuple2<String, Long>> out) -> {
      String[] words = line.split(" ");
      for (String word: words){
        out.collect(Tuple2.of(word, 1L));

      }
    })
            .returns(Types.TUPLE(Types.STRING, Types.LONG));

```

```
    KeyedStream<Tuple2<String, Long>, String> wordAndOneKeyedStream = wordAndOneTuple.keyBy(data ->
     data.f0);

    SingleOutputStreamOperator<Tuple2<String, Long>> sum = wordAndOneKeyedStream.sum(1);

    sum.print();

    env.execute();
  }
}
```

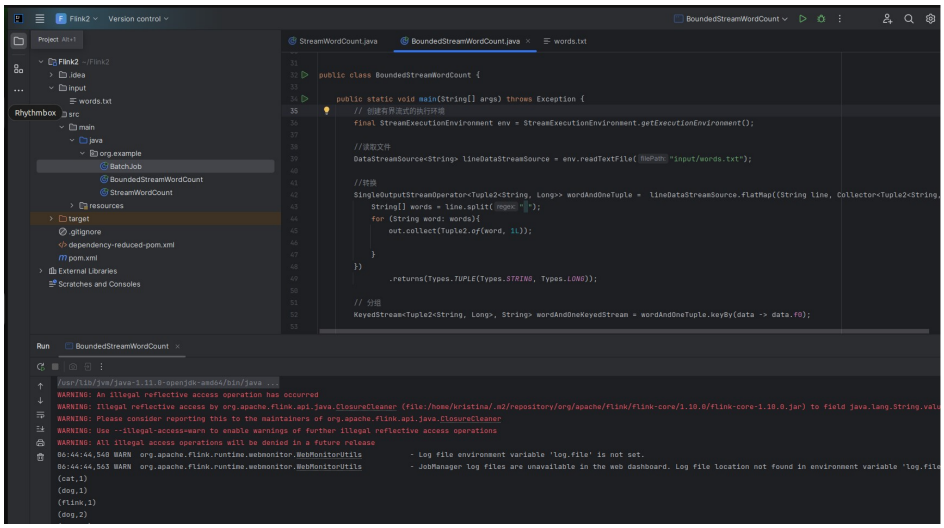

Figure 11: BoundedStreamWordCount execution

InfiniteStreamingJob (StreamWordCount) experiment

In this experiment after running the code we can see the process of the real-time streaming data statistics updates. In the end we get output file: "output1.csv".

```
package org.example;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.util.Collector;

public class InfiniteStreamingJob {

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

    if (args.length != 1) {
      System.out.println("Usage: StreamingJob <output path>");
      return;
    }
    String outputPath = args[0];

    DataStream<String> textStream = env.socketTextStream("localhost", 9999);

    DataStream<Tuple2<String, Integer>> counts = textStream
        .flatMap(new Tokenizer())
        .keyBy(value -> value.f0)
        .sum(1);

    counts.print();

    counts.writeAsText(outputPath).setParallelism(1);

    env.execute("Infinite Streaming Word Count Job");
  }

  public static class Tokenizer implements FlatMapFunction<String, Tuple2<String, Integer>> {
    @Override
    public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {
      String[] tokens = value.toLowerCase().split("\\W+");
      for (String token : tokens) {
        if (token.length() > 0) {
          out.collect(new Tuple2<>(token, 1));
        }
      }
    }
  }
}
```

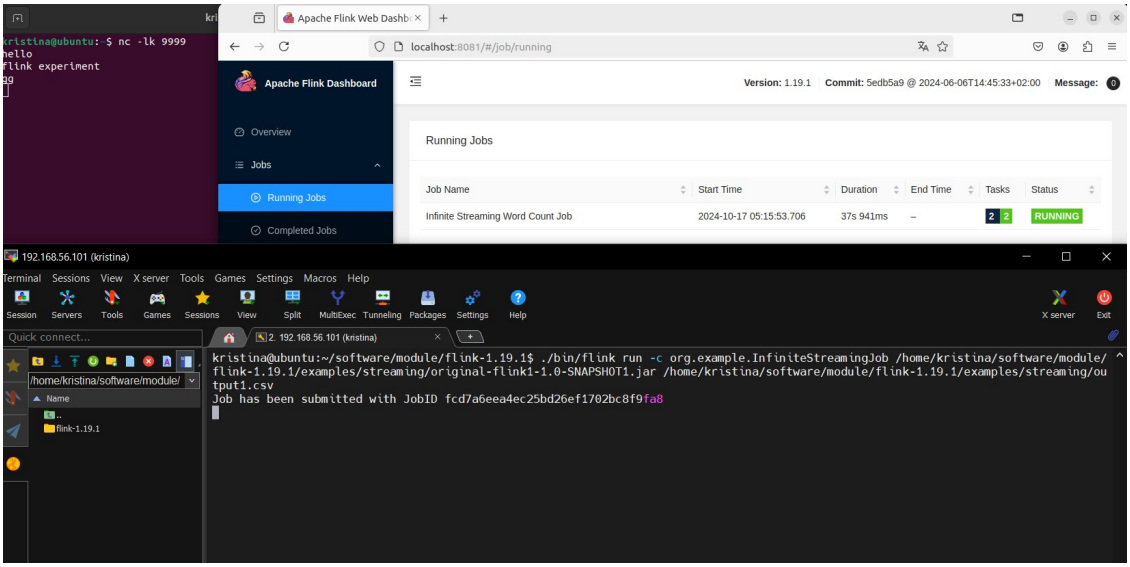

Figure 12: Streaming job execution

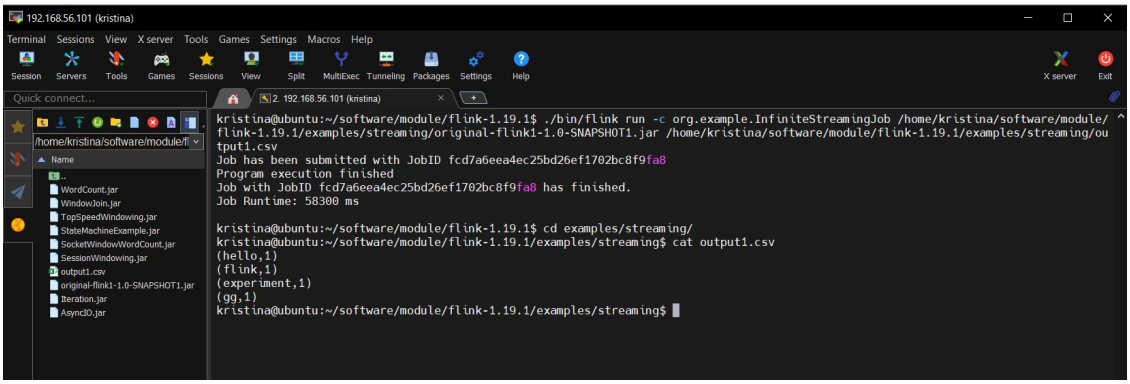

Figure 13: Streaming job result

Optional experiment

TaxiTripDuration experiment

Here we read the train.csv file and output the duration of each trip

```
package org.example;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.api.java.io.TextOutputFormat;

public class TaxiTripDuration {
  public static void main(String[] args) throws Exception {
```

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

    String inputPath = args[0];
    String outputPath = args[1];

    DataSet<String> source = env.readTextFile(inputPath);

    DataSet<Tuple2<String, Integer>> result = source
        .map(new MapFunction<String, Tuple2<String, Integer>>() {
          @Override
          public Tuple2<String, Integer> map(String line) throws Exception {
            String[] fields = line.split(",");

            if (fields[0].equals("id")) {
              return null;


            try {
              String id = fields[0].trim();
              Integer tripDuration = Integer.parseInt(fields[10].trim());
              System.out.println("Processing ID: " + id + ", Trip Duration: " + tripDuration);
              return new Tuple2<>(id, tripDuration);
            } catch (NumberFormatException | ArrayIndexOutOfBoundsException e) {
              System.err.println("Error processing line: " + line + " - " + e.getMessage());
              return null;
            }
          }
        })
        .filter(tuple -> tuple != null);

    DataSet<String> header = env.fromElements("id,trip_duration");
    DataSet<String> output = header.union(result.map(tuple -> tuple.f0 + "," + tuple.f1));

    output.writeAsText(outputPath);

    env.execute("Taxi Trip Duration");
  }
}
```

```
... 9 more
kristina@ubuntu:~/software/module/flink-1.19.1$ ./bin/flink run -c org.example.TaxiTripDuration /home/kristina/software/module/flink-1.19.1/examples/streaming/original-flink1-1.0-SNAPSHOT.jar /home/kristina/software/module/train.csv /home/kristina/software/module/flink-1.19.1/examples/streaming/output2.csv
Job has been submitted with JobID 5bf863d4eac925f3db7d021f00bcb867
Program execution finished
Job with JobID 5bf863d4eac925f3db7d021f00bcb867 has finished.
Job Runtime: 13949 ms

kristina@ubuntu:~/software/module/flink-1.19.1$
```

Figure 14: Taxi trip Duration execution

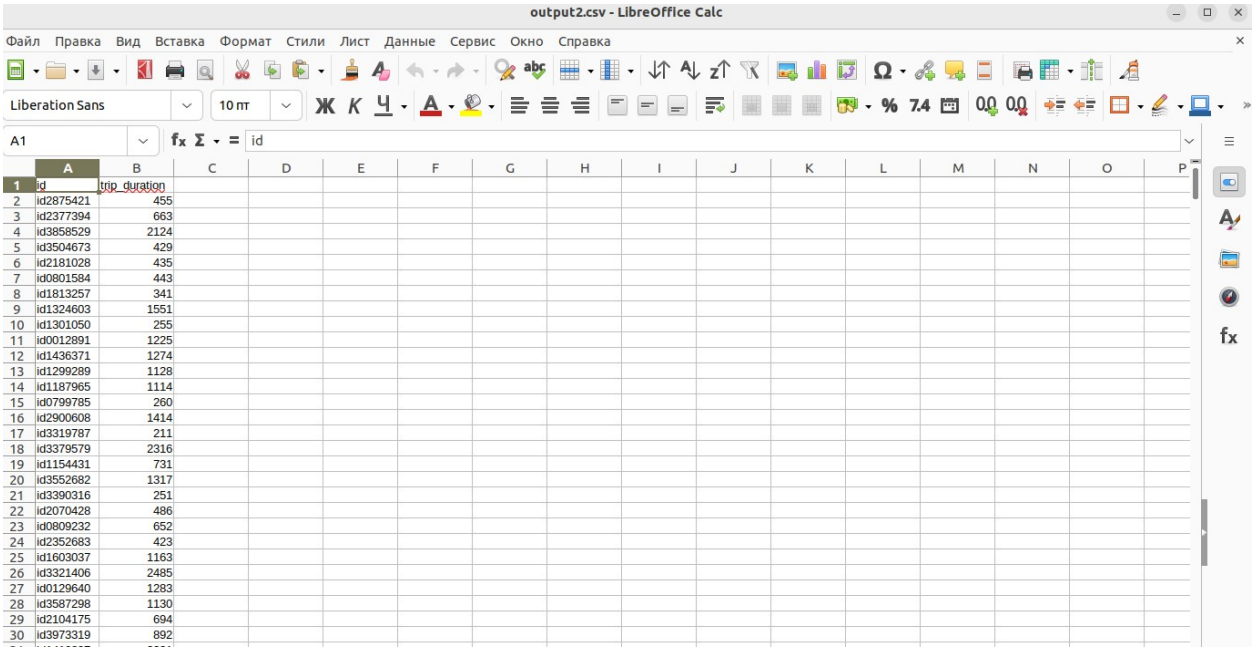

Figure 15: Taxi trip duration result

PassengerCount experiment

In this code we read train.csv file and output passengers' count.

```
package org.example;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.functions.ReduceFunction;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple2;

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

    String inputPath = args[0];
    String outputPath = args[1];

    DataStream<String> source = env.readTextFile(inputPath);

    DataStream<Tuple2<Integer, Integer>> result = source
        .map(new MapFunction<String, Tuple2<Integer, Integer>>() {
          @Override
          public Tuple2<Integer, Integer> map(String line) throws Exception {
            String[] fields = line.split("\\s*,\\s*");
            if (fields.length < 6) {
              return null;
            }
```

```
26
27            if (fields[0].equals("id")) {
28              return null;
29            }
30
31            try {
32              int passengerCount = Integer.parseInt(fields[4].trim());
33              System.out.println("Processing Passenger Count: " + passengerCount);
34              return new Tuple2<>(passengerCount, 1);
35            } catch (NumberFormatException e) {
36              System.err.println("Error processing line: " + line + " - " + e.getMessage());
37              return null;
38            }
39          }
40        })
41        .filter(tuple -> tuple != null)
42        .keyBy(tuple -> tuple.f0)
43        .reduce(new ReduceFunction<Tuple2<Integer, Integer>>() {
44          @Override
45          public Tuple2<Integer, Integer> reduce(Tuple2<Integer, Integer> value1, Tuple2<Integer,
      Integer> value2) throws Exception {
46            return new Tuple2<>(value1.f0, value1.f1 + value2.f1);
47          }
48        });
49
50    result.map(tuple -> "passenger_count,count\n" + tuple.f0 + "," + tuple.f1)
51        .writeAsText(outputPath);
52
53    env.execute("Passenger Count");
54  }
55 }
```

```
kristina@ubuntu:~/software/module/flink-1.19.1$ ./bin/flink run -c org.example.PassengerCount /home/kristina/software/module/flink-
1.19.1/examples/streaming/flink1-1.0-SNAPSHOT.jar /home/kristina/software/module/train.csv /home/kristina/software/module/flink-1.1
9.1/examples/streaming/output3.csv
Job has been submitted with JobID 506b8b2e1baa64d8dfc910de9f95876e
Program execution finished
Job with JobID 506b8b2e1baa64d8dfc910de9f95876e has finished.
Job Runtime: 19166 ms
```

Figure 16: Passenger Count execution

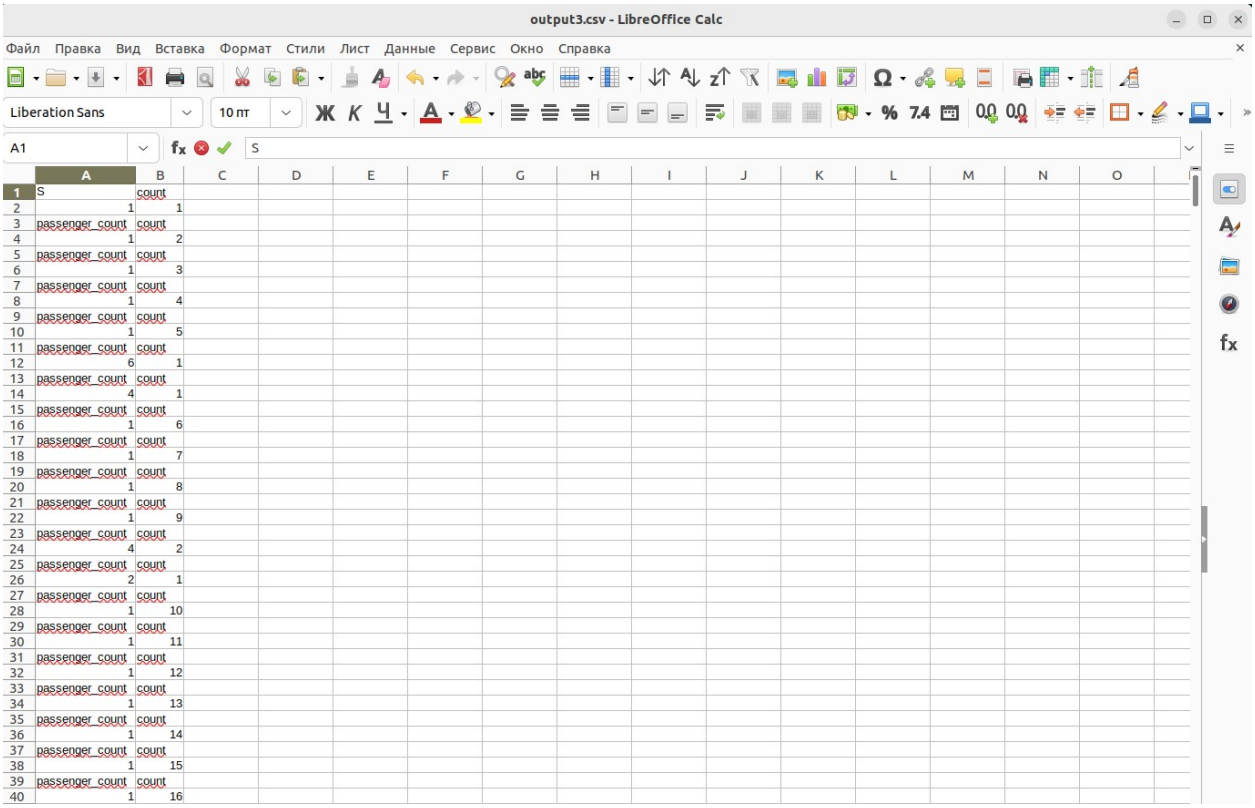

Figure 17: Passenger Count result