Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Spring Term 2019 Dr. Vasiliki Kalavri

Data Stream Processing and Analytics

Assignment 1 – Sample Solution

1. Setup up Apache Flink and run an example

Tasks

#1: The Flink distribution includes a default configuration for a local single-node cluster along with all the required startup scripts. You should hopefully not have experienced any significant troubles getting this running, other than perhaps incompatibilities with JDK 9/10[1]. In general, it's worth browsing to the dashboard (http://localhost:8081/) to be sure all the task managers successfully registered themselves and, in the case you need to troubleshoot further, the log files are an invaluable aid and can be found using:

```
tail log/flink-*-standalonesession-*.log
```

Aside from running the basic word count topology, you should also have tinkered with the configuration and set the following options:

- `parallelism.default`: defines how many physical instances of a task are instantiated during execution. Note that whilst this controls the system-wide default, the actual level of parallelism may be overridden in several ways – for instance by specifying the `-p` flag when running your streaming job or within the code with `.setParallelism(N)`, either across the execution environment or at the level of a specific operator.
- `taskmanager.numberOfTaskSlots`: this setting allows use of multiple CPU cores since each worker can execute multiple sub-tasks in separate threads. Something worth nothing is that each task slot only represents a fixed division of the *memory* resources but does not imply CPU isolation.

If you're still wondering about how logical tasks are executed at runtime or have questions like 'what is the role of a job vs. task manager?' then we recommend having a look at this section of Flink's documentation entitled "Distributed Runtime Environment" which goes into the terminology and core concepts in more detail.

1 https://issues.apache.org/jira/browse/FLINK-8033

#2: As already mentioned above there are several possibilities to adjust parallelism, either statically in the configuration, by embedding appropriate calls in the code or passing the `--parallelism` flag. Note that the order of arguments matters and so it is important to put the `-p` argument directly after the "run" action (otherwise it would be passed as a program argument to your Java program)!

2. Ingest and analyze a stream of Wikipedia Edits

Tasks

#1: The edit stream describes both additions and removals of content. Since we'd only like to retain the former we can add an operator to the chain which looks at the diff count (second element of the tuple):

```
--- wikiedits/src/main/java/wikiedits/WikipediaAnalysis.java
+++ wikiedits/src/main/java/wikiedits/WikipediaAnalysis1.java
@@ -41,10 +41,16 @@
             public Tuple2 reduce(Tuple2<String, Integer> e1,
                                  Tuple2<String, Integer> e2)
             {
               return new Tuple2<>(e1.f0, e1.f1 + e2.f1);
             }
+         })
+         .filter(new FilterFunction<Tuple2<String, Integer>>() {
+           @Override
+           public boolean filter(Tuple2<String, Integer> e1){
+             return e1.f1 >= 0;
+           }
          });
```

#2: We can bound the state stored by our computation by using a window to restrict the infinite stream. In Flink's API there are two mandatory components that need to be specified: (i) an *assigner* which places each incoming tuple into a window and (ii) a *window function* which specifies the computation we'd like to perform. For the first, we'd like to limit along the time axis and can therefore use a tumbling window as shown below; for the latter, there is no change required because nothing has changed about our desired output and so we can retain the same aggregation as before.

```
--- wikiedits/src/main/java/wikiedits/WikipediaAnalysis.java
+++ wikiedits/src/main/java/wikiedits/WikipediaAnalysis.java
@@ -34,17 +34,27 @@
          })
          // group by user
          .keyBy(0)
+         // split stream into distinct buckets along the time axis
+         .timeWindow(Time.seconds(10))
+         // (or this equivalent formulation which specifies the
+         // desired window type more clearly)
+         //.window(TumblingProcessingTimeWindows.of(Time.seconds(10)))
          // aggregate changes per user
```

```
.reduce(new ReduceFunction<Tuple2<String, Integer>>() {
  @Override
```