Solutions Streaming Assignment

Wouter Zorgdrager

December 2017

Task 1: Parsing JSON into Flink tuples (10 pts).

I used the json4s library to parse the raw data into Scala case classes. I added the following dependencies to my **pom.xml** to use this library:

```
		<dependency>
			<groupId>org.json4s</groupId>
			<artifactId>json4s-native_2.10</artifactId>
			<version>3.6.0-M2</version>
		</dependency>
		<dependency>
			<groupId>org.json4s</groupId>
			<artifactId>json4s-jackson_2.10</artifactId>
			<version>3.6.0-M2</version>
		</dependency>
```

To parse the JSON, I first created some case classes to store parts of the data. Not every field is necessary to store, since we won't use it in later exercises.

```
//set default formats e.g. dates
  implicit val formats = DefaultFormats
//case classes used to parse, type with backticks because type is a reserved keyword
//payload is still set to a parsed type, since that one differs per event (so we can parse it
    after a filter)
  case class Event(id: String, `type`: String, actor: Actor, repo: Repo, payload: JObject,
      created_at : Date, public : Boolean);
  case class Actor(id: Integer, login: String, gravatar_id: String, url: String, avatar_url:
      String);
  case class Repo(id: Integer, name: String, url: String);
```

Then I created a simple method to parse each line into the Event case class:

```
  /**
    * Converts JSON (event) string to event.
    * @param in input string.
    * @return event class
    */
  def strToEvent(in : String) : Event = {
    val json = parse(in); //parse to json
    val event : Event = json.extract[Event]; //extract to case class

    return event;
  }
```

Finally I set up the Flink streaming environment and parsed all the lines into their case classes.

```
def main(args: Array[String]) {
  // set up the execution environment
  val env = StreamExecutionEnvironment.getExecutionEnvironment

  // read file
  val file:DataStream[String] = env.readTextFile("github-okt1-5.10.2017.gz");

  //parse all the lines
  val streamer = file.map(strToEvent(_));

  //start the environment
  env.execute();
  }
```

Task 2: Filtering events of interest (10pts)

This task is quite simple, basically you just use the 'filter' method. For example if you want to filter on the *IssuesEvent*, you use the following code:

```
//parse all the lines and filter
  val streamer = file.map(strToEvent(_))
      .filter(event => event.`type` == "IssuesEvent");
```

Task 3: Defining Event-time (20pts)

I just realized the assignment stated that there is no guarantee the stream comes in-order. However I did assume this while making the solutions by using the assignAscendingTimestamps method. In practice it turned out only a few events are out of order. So, assignAscending-Timestamps can be seen as correct as long as the student understands that this might be a problem if the whole stream is out of order.

For some useful information on (event)times and (sliding) windows in Flink, check the following resources: event time and windows. First of all we have to set the time characteristic in the main. There are several options: processing, ingestion and event time. We will need the latter.

```
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
```

Then from the events we need to 'extract' the timestamps we want to use as event-time. In our case this is the **created_at** field which is a default field in each event. The code for that looks like this:

```
//parse all the lines and filter
  val streamer = file.map(strToEvent(_)).
    filter(event => event.`type` == "IssuesEvent"). //filter on particular event
    assignAscendingTimestamps(event => event.created_at.getTime); //get time of the date timestamp
```

Task 4: Computing Aggregates over (Sliding) Windows (30pts)

Every hour, report the count the unique issues that have been opened per repository during the last 2 days (10pts).

I started by creating a method which extracts the payload from the event.

```
//issue case class
case class Issue(action: String, issue: IssueId);

//issue id case class
case class IssueId(id: Integer)

/**
  * Extract issue from the payload.
  *
  * @param event the input event.
  * @return tuple with (repo_name, Issue)
  */
def toIssue(event : Event) : (String, Issue) = {
  (event.repo.name, event.payload.extract[Issue])
}
```

I assume that each opened issue is unique. As for the window I used a sliding window. Then the actual code to execute this task:

```
  val streamer = file.map(strToEvent(_)).
    filter(event => event.`type` == "IssuesEvent"). //filter on particular event
    assignAscendingTimestamps(event => event.created_at.getTime). //get time of the date timestamp
    map(x => toIssue(x)). //map to issue
    filter(issue => issue._2.action == "opened"). //filter on opened
    map(x => (x._1, 1)). //map to a simple map reduce form
    keyBy(0). //key by repo name
    window(SlidingEventTimeWindows.of(Time.days(2), Time.hours(1))). //window 2 days, slide 1hour
    sum(1); //sum the issues per repo
```

Part of the output:

```
(HawkRidgeSystems/HRS-Main,2)
(idno/idno,1)
(silverstripe/silverstripe-framework,1)
(SimpleLance/simplelance.github.io,1)
(angular-ui/bootstrap,1)
(WayofTime/BloodMagic,1)
(mbasaglia/Melanobot,1)
(miwarin/netbsd-gnats-memo,1)
(NodeBB/NodeBB,1)
(EKGAPI/webAppEKGAPI,3)
```

Every day, report the number of commits per that repository that have been pushed that very day (10pts).

For this taskes I first created some case classes and a parse method for a **PushEvent**:

```
  //push case class, size is amount of commits
```

```
 case class Push(size: Integer);

   /**
   * Extract push event from the payload.
   *
   * @param event the input event.
   * @return tuple with (repo_name, Push)
   */
 def toPush(event: Event) : (String, Push) = {
   (event.repo.name, event.payload.extract[Push]);
 }
```

The code for the actual stream is quite similar to the first task. Filtering and parsing on a push event, then a simple map reduce using a tumbling window. The actual code:

```
   //parse all the lines and filter
   val streamer = file.map(strToEvent(_)).
     filter(event => event.`type` == "PushEvent"). //filter on particular event
     assignAscendingTimestamps(event => event.created_at.getTime). //get time of the date timestamp
     map(x => toPush(x)). //map to push event
     map(push => (push._1, push._2.size)). //map to (repo_name, amount_of_commits)
     keyBy(0). //key by repo name
     window(TumblingEventTimeWindows.of(Time.days(1))). //tumbling window of 1 day
     sum(1); //sum the commits per repo
```

Part of the output:

```
(davidjhulse/davesbingrewardsbot,2)
(xndcn/d-statistics,3)
(ingydotnet/zilla-dist-pm,3)
(su-github-machine-user/github-nagios-check,24)
(350dotorg/megamap-data,1)
(JordanMussi/IRC,1)
(jsr-software/openspace-android-sdk,6)
(iamandrebulatov/BC-Category-Page-Color-Swatch,11)
(WSCU/Robotics,1)
(prabhash1785/DataStructures,1)
```

Count the number of issues per project, which have not received any updates (i.e., closed, opened, etc.) for more than one day. Is this a session window?(10pts)

Note: This question is a bit ambiguous, I assumed that this query should output if a issue had no activity for one day after a certain action. Multiple interpretations are correct (as long they give a good reasoning).

We will re-use the issue case classed defined above. First we parse everything to a (repo_name, issue_id, 1) to count the amount of 'updates' per issue within 1 day. We indeed use a session window for that, because that helps us to check for a gap of 1 day. Finally we do a summation on the amount of events per issue per day, filter out the ones with only 1 event and count the amount of issues per repo. The actual code:

```
//parse all the lines and filter
   val streamer = file.map(strToEvent(_)).
     filter(event => event.`type` == "IssuesEvent"). //filter on particular event
```

```
    assignAscendingTimestamps(event => event.created_at.getTime). //get time of the date timestamp
    map(x => toIssue(x)). //map to push event
    map(issue => (issue._1, issue._2.issue.id, 1)). //map to (repo_name, issue_id, 1)
    keyBy(0, 1). //key by repo name and issue id
    window(EventTimeSessionWindows.withGap(Time.days(1))). //session window of 1 day
    sum(2). //sum the amount of events per repo_name/issue_id combination
    filter(issue => issue._3 == 1). //filter out the ones with only 1 event
    map(issue => (issue._1, 1)). //remove the issue_id
    keyBy(0).sum(1); //simple map reduce
```

Part of the output (it also outputs intermediate results):

```
(HawkRidgeSystems/HRS-Main,1)
(HawkRidgeSystems/HRS-Main,2)
(silverstripe/silverstripe-framework,1)
(SimpleLance/simplelance.github.io,1)
(minillinim/sammy,1)
(CTC-CompTech/delivery,1)
(antonioortegajr/beerfind.me,1)
(MiYa-Solutions/sbcx,1)
(RainLoop/rainloop-webmail,1)
(WebDevStudios/custom-post-type-ui,1)
```

Task 5: Extracting Patterns (30pts)

For this tasks you need the CEP library.

Output all the repository/pull-request combinations that have been opened and closed. (15pts).

Once again, we create some case classes and an extractor method:

```
//pr case class
case class PR(action: String, number: Integer);

  /**
  * Extract PR event from the payload.
  *
  * @param event the input event.
  * @return tuple with (repo_name, PR)
  */
def toPR(event: Event) : (String, PR) = {
  (event.repo.name, event.payload.extract[PR]);
}
```

We use 3-tuple for the pattern; (repo_name, pr_id, pr_action). The pattern checks for an *opened* event followed by an *closed* event using **Non-Deterministic Relaxed Contiguity** (see documentation). The actual pattern in code:

```
    val pattern = Pattern.
    begin[(String, Integer, String)]("start").
    where(_._3 == "opened").
    followedByAny("end").
    where(_._3 == "closed");
```

Then we use this code to setup the correct stream:

Note: in order for this stream query to work, I had to remove the timestamp assignment + removing the stream time characteristic.

```
  //parse all the lines and filter
  val streamer = file.map(strToEvent(_)).
    filter(event => event.`type` == "PullRequestEvent"). //filter on particular event
    map(x => toPR(x)). //map to PR event
    map(pr => (pr._1, pr._2.number, pr._2.action)). //map to (repo_name, pr_id, pr_action)
    keyBy(0, 1); //key by repo and pr combinations
```

Finally I created a PatternStream on which I selected and collected the results.

```
  val patternStream = CEP.pattern(streamer, pattern); //create the patternstream
  val result = patternStream.select[(String, String)]{ //select from the patternstream

    (map : scala.collection.Map[String, Iterable[(String, Integer, String)]]) =>
      val startEvent = map.get("start").get.head; //get the start event
      val endEvent = map.get("end").get.head; //get the end event

      (startEvent._1, endEvent._2.toString); //return a (repo_name, pr_id) tuple
  };
```

Part of the output:

```
(TeamGabriel/gabriel,1)
(uw-it-aca/jira-hook,2)
(SageMantis/SeniorProjectGrouple,71)
(vanceavalon/cassandra,1)
(fishulla/Torque3D,41)
(SirCmpwn/ChatSharp,14)
(raorao/groceries,3)
(captainkirkby/Gears,29)
(DamageStudios/rims,83)
(veravera/dots,6)
```

Count all issues that have been opened, closed, and re-opened in less than 48 hours (15pts).

We will re-use the Issue case classes we already set up before. For the pattern we again use a 3-way tuple (repo_name, issue_id, issue_action). We will use the same contiguity as for the pattern in the last task, but this time we also add a time constraint of 48 hours.

```
  val pattern = Pattern.begin[(String, Integer, String)]("start").
    where(_._3 == "opened").
    followedByAny("middle").
    where(_._3 == "closed").
    followedByAny("end").
    where(_._3 == "reopened").
    within(Time.hours(48));
```

Then we use this code to setup the correct stream:
Note: we add the stream time characteristic and timestamp assignment again.

```
      //parse all the lines and filter
    val streamer = file.map(strToEvent(_)).
      filter(event => event.`type` == "IssuesEvent"). //filter on particular event
      assignAscendingTimestamps(x => x.created_at.getTime).
      map(x => toIssue(x)). //map to issue event
      map(i => (i._1, i._2.issue.id, i._2.action)). //map to (repo_name, issue_id, issue_action)
      keyBy(0, 1); //key by repo and issue combinations
```

Finally we create a PatternStream on which we select and collect the results as we did in the previous task. Finally we do a simple map reduce, to get the count per repository (this part is a bit open to interpretation since the assignment asks for just a count).

```
val patternStream = CEP.pattern(streamer, pattern); //create the patternstream
    val result = patternStream.select[(String, String)]{ //select from the patternstream

      (map : scala.collection.Map[String, Iterable[(String, Integer, String)]]) =>
        val startEvent = map.get("start").get.head; //get the start event
        val middleEvent = map.get("middle").get.head; //get the middle event
        val endEvent = map.get("end").get.head; //get the end event

        //it doesnt really matter which event we pick, since repo_name and id will be the same
        (startEvent._1, endEvent._2.toString); //return a (repo_name, issue_id) tuple
    }.map(x => (x._1, 1)).keyBy(0).sum(1); //map reduce
```

Keep in mind that since we are doing a pattern matching on windows and then a 'global' map-reduce it outputs intermediate results. Part of the output is:

```
(emberjs/data,1)
(magento/magento2,1)
(yoyoHoneyS1ngh/OgniDon-tGoInHere-,1)
(yoyoHoneyS1ngh/OgniDon-tGoInHere-,2)
(yoyoHoneyS1ngh/OgniDon-tGoInHere-,3)
(yoyoHoneyS1ngh/OgniDon-tGoInHere-,4)
(yoyoHoneyS1ngh/OgniDon-tGoInHere-,5)
(yoyoHoneyS1ngh/OgniDon-tGoInHere-,6)
(yoyoHoneyS1ngh/OgniDon-tGoInHere-,7)
(yoyoHoneyS1ngh/OgniDon-tGoInHere-,8)
```