Data Stream Processing and Analytics
Spring Semester 2020

Prof. Vasiliki Kalavri
vkalavri@bu.edu

Final Project

Submission deadline: April 30 2020, 11:59 pm (**no extensions**)

The rest of this assignment assumes a UNIX-based setup. If you are a Windows user, you are advised to use Windows subsystem for Linux (WSL), Cygwin, or a Linux virtual machine to run Flink in a UNIX environment.

1. Introduction

In this final project, you will use Apache Flink to build a real-time monitoring and anomaly detection framework for data centers. The tasks you will implement are inspired by the anomaly detection queries described in the SAQL paper. It is highly recommended to read Sections 1 and 2 of the paper to understand how stream processing can enable continuous system monitoring and low-latency anomaly detection in the real world.

Your cluster monitoring framework will provide users with the option to run one of following three (3) anomaly queries:

1. **Rule-based query**: The user can specify a sequence of events and the application should output an alert if a pattern matching the sequence is found.
2. **Time-series query**: The user can specify a sliding window, a number of consecutive windows to consider, and a percentage difference threshold. The application should output an alert whenever a task scheduling event requests an abnormally high or low number of CPU resources.
3. **Invariant-based query**: The user can specify a training model period, during which the application ingests events and learns the normal bounds of task scheduling delays. During the detection period, the application generates an alert for any anomalous task scheduling delays detected.

2. General requirements

To receive full points, your solutions must meet the following general requirements:

1. Programs should work correctly with parallelism > 1. Source parallelism is always set to 1.

2. Programs can correctly handle out-of-order input data. The `TaskEventSource` generates watermarks automatically and can be configured to produce slightly out-of-order events by setting the `maxEventDelaySecs` parameter in the constructor. Note that watermarks will always be correct. You can use the code provided in Assignment #1 to assign watermarks and timestamps in the `FlinkKafkaConsumer`.

3. Programs use the managed state API for all stateful processing.

If any of the above requirements is not met, the solution will only be eligible for 50% of its corresponding score. If multiple requirements are not met, the score will decrease proportionally. For instance, a solution to Q1 that works correctly for parallelism = 1 only, handles out-of-order data correctly, and uses managed state, will be eligible for 50% of 20 points, i.e. 10 points. If the solution works with parallelism = 1 and only with in-order data but uses managed state, then it is eligible for up to 50% of 10 points, i.e. 5 points. If the solution uses in-memory state and provides correct results only with parallelism = 1 and in-order data, then it can score up to 2.5 points.

3. Q1: Rule-based anomaly (20/100)

The simplest way to detect anomalies is to specify explicit rules that describe what constitutes an anomaly. These rules are specified as a sequence of events that occur in temporal order. The anomaly detection program continuously monitors the stream and generates an alert when a rule is detected, i.e.when a sequence of events in the stream match the specified pattern. For example, we could be interested in anomalies that occur when an update causes a task to fail. We could specify a rule as the sequence `Task.SCHEDULE → Task.UPDATE_RUNNING → Task.FAIL` and raise an alert every time a scheduled task fails after an update.

For the purpose of this project, we will consider fixed sequences of three (3) events and allow task events only. A sequence must always correspond to the same task (same `jobId` and `taskIndex`).

a. Input

Your program should read events from the `TaskEventSource` and accept the following parameters:

- **Event sequence**: The event sequence (rule) must be provided as 3 integer arguments, `--e1`, `--e2`, `--e3`, whose values correspond to the `EventType` as described in the format and schema document. The arguments must be given in temporal order. For example, the inputs ``` ``--e1 0 --e2 1 --e3 5” ``` define the sequence `SUBMIT(0) → SCHEDULE(1) → KILL(5)`. Any combination of task events is considered a valid rule even if it does not *make sense*, e.g., the sequence `FINISH → SUMBIT → SCHEDULE` will never occur in the given dataset, however, one might want to check for such sequences to detect bugs in the scheduler.

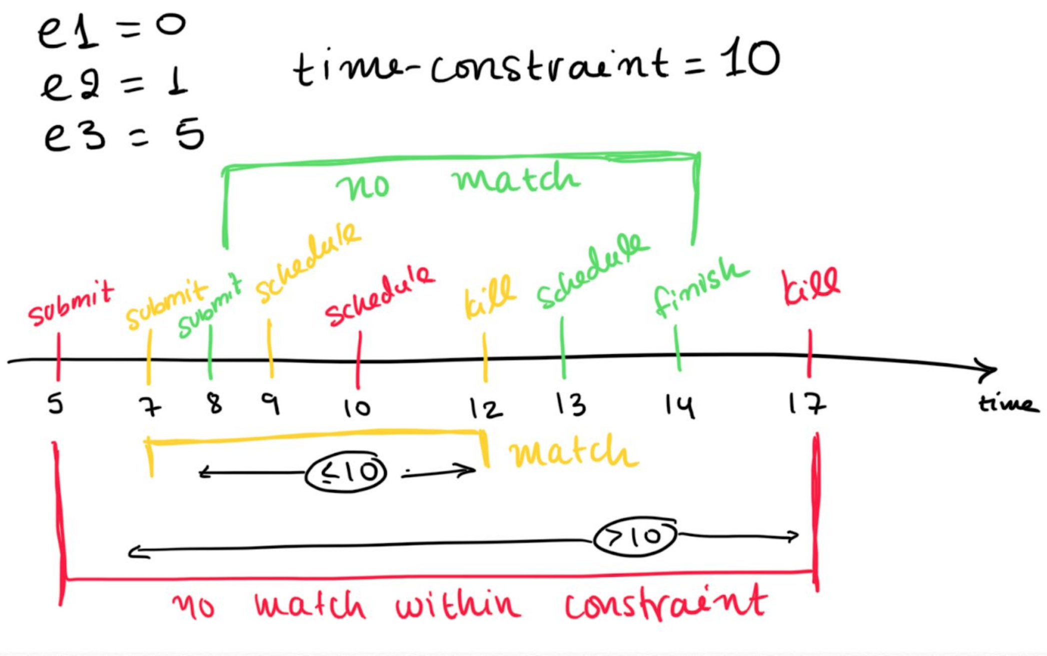
- **Time constraint**: The maximum time difference (in milliseconds) between the first and last events in a matching sequence. To generate an alert, events in a matching sequence must happen within the time bounds specified by the time constraint, otherwise they can be discarded. A time constraint of 0 means no constraint: events can match the rule no matter how much time apart they occur. The time constraint must be provided as a long argument `--time-constraint`.

b. Output

The program must output a `Tuple3<TaskEvent, TaskEvent, TaskEvent>` for every sequence that matches the provided rule and satisfies the time constraint.

c. Example

Consider the example below, where events corresponding to the same task are shown in the same color and timestamps are shown on the time axis. The program needs to generate an alert for any task whose events match the pattern `SUBMIT(0) → SCHEDULE(1) → KILL(5)` within the specified constraint of 10ms. In this example, only the yellow task is a match and the program must output a `Tuple3` with the events `SUBMIT` at t=7, `SCHEDULE` at t=9, and `KILL` at t=12.



d. Questions

In your report, provide brief answers to the following questions:

1. Can your implementation detect anomalies as soon as they happen or does the time constraint period need to be exhausted? Do you use timers? How?
2. How do you handle out-of-order events? In the example above, if the yellow `KILL` arrives before the yellow `SCHEDULE`, can your solution still detect the match? How?
3. How do you handle state clean-up? When is it safe to drop an event and how quickly can your program decide that?

4. Q2: Time-series anomaly (20/100)

A time-series is a sequence of measurements over time. Time-series analysis is useful when monitoring a large system, such as a cluster or a datacenter, because it can be used to detect workload and network spikes. For instance, SAQL Query 2 monitors data exchange and computes the average amount of data transferred in 10-minute sliding windows. To detect an anomaly, the query maintains a **moving average**, the mean of the averages across three consecutive windows. The query compares the current window average with the moving average and generates an alert if their difference is too high.

In our context, the time-series consists of CPU resource requests in the Task events table. When a task is scheduled, it requests a number of CPU cores (also some amount of RAM and some amount of disk space). The program must first compute the average number of CPU cores requested per machine in a sliding window fashion and then use the values of a configurable number of consecutive windows to compute the moving average. For every window, the program must compare the current average with the moving average and generate an alert if their difference is *significant*, according to a configurable threshold.

To parallelize the computation, we will consider the set of tasks scheduled **on each machine** as a separate time-series to be monitored for anomalies. The difference between a given window value, `curAvg`, and a moving average, `movAvg`, will be computed as their percentage difference:

$$\%diff = \frac{|curAvg - movAvg|}{\frac{(curAvg + monAvg)}{2}}$$

If the percentage difference is above the threshold, the program must generate an alert signifying that an anomaly has been detected. Note that you do not need to process all events per task to extract the resource requests. You can filter out all events except `SCHEDULE` events. Also note that some resource request values are zero. You can also filter those out.

a. Input

The program must accept the following parameters:

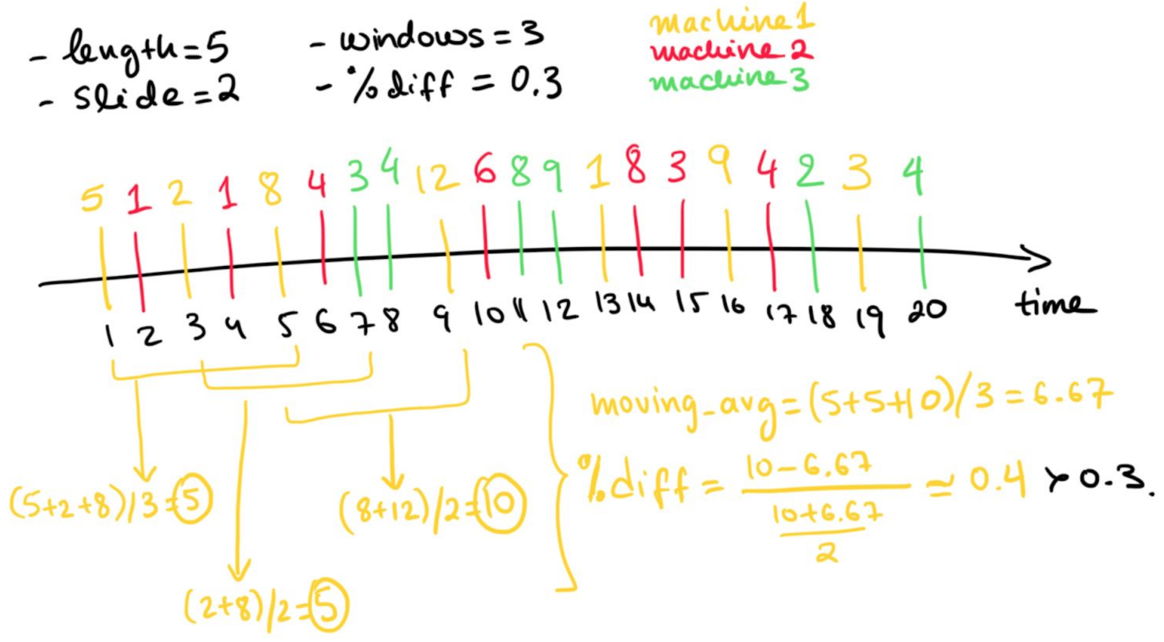
- The sliding window **length** and **slide** in milliseconds, as two long arguments `--length` and `--slide`.
- The **number of consecutive windows** to consider in the moving average calculation, as an integer argument `--windows`.
- The **%diff threshold**, above which an anomaly is detected, as a double argument `--diff`.

b. Output

For every window, the program must output an alert if the percentage difference is above the specified threshold. Otherwise, it must output nothing. The alert must be a record of type `Tuple3<Integer, Double, Double>`, where the first field is the machine ID, and the next fields are the current and moving average values, respectively.

c. Example

Consider the example below where events scheduled on the same machine are shown in the same color and timestamps are shown on the time axis. The event numbers correspond to CPU resource requests. The figure shows the computation of a sliding window moving average over three consecutive windows for machine 1 (yellow). This computation will generate an alert as the percentage difference of the last window (10) and the moving average (6.67) is above the specified threshold.



d. Questions

In your report, provide brief answers to the following questions:

1. What happens to the number of generated alerts when:
    a. you increase the threshold?
    b. the window slide is not much smaller than the length?

2. If the slide is small and the number of consecutive windows is large, Flink will be creating and then deleting many window objects continuously. This might lead to garbage collection pauses and increased burden for the state backend. How could you implement the moving average so that consecutive windows can share partial computation? When do you think that would help performance?

5. Q3: Invariant-based anomaly (20/100)

An invariant-based anomaly detection query learns invariants (system properties) during a training period and then continuously verifies these properties during an online detection period.

In our context, what we are going to "learn" during the training period is the normal task scheduling time (the difference between the timestamp of the `SUBMIT` and the subsequent `SCHEDULE` event) **per priority**. Tasks with higher priority are supposed to get preference for resources over tasks with smaller priority values. During the **training period**, the program needs to maintain the current minimum and maximum task scheduling delays, for each priority. During the **online detection** period, the program needs to generate an alert if a task scheduling duration is outside the bounds defined by the learned minimum and maximum values.

a. Input

The program must accept the following parameter:

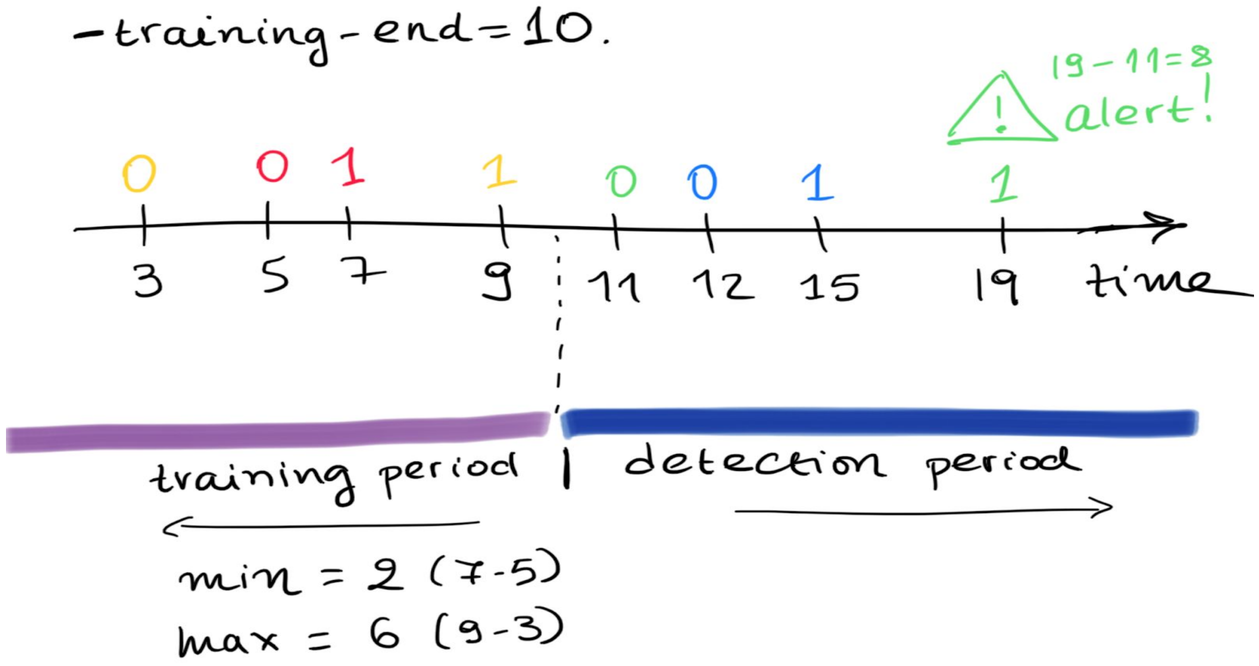
- The timestamp (in milliseconds) marking the end of the **training period**, as a long argument `--training-end`.

b. Output

After the end of the training period and during the online detection period, the program must output an alert whenever a task scheduling duration is outside the learned bounds for its priority. The alert will be a record of `Tuple2<Integer, Long>`, where the first field is the task priority and the second field is the time difference between the `SUBMIT` and `SCHEDULE` events that raised the alert.

c. Example

Consider the example below where events for the same task are shown in the same color. For illustration purposes, let's assume that all task events shown in the figure belong to the same priority class. (However, your solution must learn the bounds for each priority separately). The events are indicated by their number ids (0 denotes a `SUBMIT` and 1 denotes a `SCHEDULE`). When the training period is over, the program has learned the scheduling time bounds [2, 6]. During the detection period, the blue task has a scheduling duration within bounds and does not generate an alert, while the green task has a scheduling duration that exceeds the upper bound learned and will generate an alert.



d. Questions

In your report, provide brief answers to the following questions:

1. Can you think of a way to periodically retrain and adjust the learned bounds? How would you choose when to retrain? What state would you need to maintain in that case?
2. A naive way to implement the control switch between the training and detection period would be to store everything in state until the watermark arrives to the end of the training period and compute the minimum and maximum task scheduling duration at that point. Why is that a bad approach?
3. In the presence of out-of-order events, how did you check which events contribute to the training period? What does your program do if it receives a task duration that belongs to the online detection phase while the watermark is still behind the training period end?

6. Fault-tolerance (20/100)

Great job implementing an anomaly detection framework but don't rush into deploying it to production just yet! Let's take a few steps to ensure that your applications can tolerate failures without losing data. You wouldn't want to miss any alerts, would you?

a. Use a Kafka source

If you have already used the state management API, the only additional thing you need to make your Flink applications provide exactly-once fault tolerance is a re-playable source. The easiest way to achieve that is by using Kafka.

Write a simple application that pushes task events to a Kafka topic (this will be very similar to the **FilterTaskEventsToKafka** program you wrote in Assignment #1) and then ingest events using a `FlinkKafkaConsumer` (this will be very similar to **MaxTaskCompletionTimeFromKafka** from Assignment #1.). If you now enable Flink checkpoints, Kafka will automatically remember the latest offset it read from in a case of failure and start reading from that point on recovery. Consult the Flink documentation to make sure you configure Kafka properly.

b. Questions

1. Configure Flink to use RocksDB as the state backend and enable checkpointing. Choose one of the queries you implemented above and experiment with different checkpointing periods and with turning incremental checkpointing on and off. Report the checkpoint sizes and the time it takes for each checkpoint to complete.
2. Is the incremental checkpointing approach always better? When does it make sense? When would taking complete checkpoints make more sense?

3. Review the Flink documentation page on Savepoints and prepare your program for reconfiguration (e.g., assign operator IDs). Now, choose one of the queries you have implemented, start it, trigger a savepoint and restart the program with different parallelism. What happens? Does your program continue generating alerts from the point it left off? Do you see duplicates?

7. Report (20/100)

Write a short report (up to 5 pages) to discuss your solutions and results. Structure your report using the following 3 sections:

1. **Design**: Provide a short overview of your solution for each anomaly query. Describe and justify the dataflows you built, the operators you used, the state primitives you chose.
2. **Discussion:** Provide answers to the questions listed in the corresponding section of each of the tasks above (3d, 4d, 5d, 6b).
3. **Misc:** Mention anything you found fun, interesting, surprising, or annoying while working on this project.

8. Tests

As in previous assignments, you can use the `JobEventCountTest` class as an example to write tests. For the test base classes to work properly, you will need to use the `printOrTest()` method as the dataflow sink. Tests will not be graded but you are welcome to submit them together with your code.

9. Deliverables

This project has two (2) deliverables to be uploaded to **Blackboard**:

1. Your code for Tasks 3, 4, 5, and 6. As in previous assignments, **upload your `src` folder to Blackboard**. If you have edited the `pom.xml` file in any way, make sure to provide the updated file as well. Note that **well-documented code is always easier to understand and grade**, so please make sure your code is clean, readable, and has comments.
2. A brief report (maximum 5 pages) as described in Section 7.

Happy coding and good luck 🍀