CS 591 K1: Data Stream Processing and Analytics Final Project Report

Haoyu Xu

April 28, 2020

1 Abstract

The Final Report mainly focuses on three sections: Design, Discussion and Misc.

2 Design

First, for how to retrieve parameters from command line, we can just use `params.getInt()` or some other related APIs. Define a global variable so that we can use it in operator functions.

I. `RuleBasedAnomaly.java`

This question want to find a specific matched pattern under some time constraint. **We take when we find the pattern for the first time as our result**. First we use `filter()` to filter the event type we want, which is the type in the parameters, then `keyBy()` by the same task, finally we can use `KeyedProcessFunction()` to do some operations. The process function's logic is that for `processElement()` we want to find when it's the first time meet the starting type(smallest value), then we set timer at current timestamp plus the value of timeconstraint. Then for `onTimer()` we can use MapState to retrieve the earliest, mid and the latest. The state primitives we use is MapState because we want to store all three elements in the list so that we can find the time diff. What's more, we can identify these three events by the key which is event's type value. When we see all three types exist and for the first time we meet, we can calculate the diff.

II. `TimeSeriesAnomaly.java`

First we only consider SCHEDULE event and CPU greater than 0 so we use `filter()`. Then we consider on each machine so we `keyBy()` machineId. After that we can pass parameters to the `timeWindow()`. `apply()` is an important function, we use it to generate a custom object that stores key, current average and slidng window's end. Then we can use `flatMap()` and `keyBy()` machineId again. The process function's logic is that for

`processElement()` we always put the window's end and task object in the MapState and register a timer at sliding window's end + $(n-1) * slide$(where n is number Of windows). Then in `onTimer()` we can check if elements from t to $t-(n-1)*slide$ exist in the MapState, if all exist, then we can calculate the result to see if it meets the threshold. So that's why we use MapState here. The key is window's end and value is object. We can check if all the keys in the range exist, which means that it is a continuous sliding window.

III. `InvariantBasedAnomaly.java`

This question is similar to `MaxTaskCompletionTimeFromKafka.java` in Assignment 1. We want to compute the task duration per priority. So first `filter()` only leave SUBMIT and SCHEDULE events. Then, `keyBy()` for the same task. After `process()`, `flatMap()` and `keyBy()` we can get `Tuple` that store priority and info object InvTask. Finally, `ProcessFunction()` here we have two ValueStates, which are used to keep updating the max and min. Then there are two ListStates, the first one stores the current state in detection period, the second one is to store some ahead periods happened after watermark belongs to detection period. Finally we use `onTimer()` first to calculate the ahead state for once and then to check if the duration exceeds the upper or lower bound and will generate an alert.

3 Discussion

3.1 3D

I. 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?

No. My implementation needs the time constraint period to be exhausted. Because I register the `onTimer()` at the end of the constraint. When we meet the smallest type, we set timer at current timestamp plus constraint, and we can check if all three types exist in the MapState and then to choose if it meets the standard. However, we can switch to "as soon as they happen" mode if we first check the map's size in `processElement()`.

II. 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?

Yes. As I said, because I use MapState to store all three matched elements, so I don't care which one arrives first. We just find a match with all these three elements, then we can get these elements in `onTimer()` and compare with their timestamps to decide whether output or not. In this way, we handle out-of-order events by the way.

III. How do you handle state clean-up? When is it safe to drop an event and how quickly can your program decide that?

In my program, I clean it after getting these three TastEvent objects. It's safe after your program not using the state again and my program actually works in this way.

3.2 4D

I. What happens to the number of generated alerts when:

a. you increase the threshold?

The number of alerts will be decreased because it generate alerts when result goes beyond threshold.

b. the window slide is not much smaller than the length?

The number of alerts will be decreased because we generate less data than previous in the view of longer slide length.

II. 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?

I think we can use some state sharing technique, like the stream slicing technique which was introduced in class, which means that we can separate the stream into non-overlapped slices so that we can combine all the windows from the slices. By now, flink creates bucket for each sliding window that has overlapping part, when we receive a tuple, we put it into all these buckets, but when we use stream slicing technique, it means that we only put the tuple into only one slice, which you can consider a smaller bucket, and we compute the partial aggregation for each slice so that the slices can make up to a whole sliding window to compute the result.

3.3 5D

I. 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?

We can define a periodical bounds generator in the `onTimer()` function, which means that we keep updating the training start and end. We can just compare new training start and end with watermark to determine whether we are in the training period. The state is ValueState, which stores next period start and end. If watermark is less than start, do nothing, if watermark go through the end, then we can update the ValueState with next period's start and end.

II. 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?

Because we are storing large number of elements in the state backend if the training period is very long. This might lead to garbage collection pauses and increased burden for the state backend, which can probably shut down the service.

III. 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?

We can compare watermark with training end to check whether events contribute to training period or not because watermark ensures that all events before watermark happened. I use kind of cache like ListState to store periods happens after watermark and training period in advance. And in `onTimer()` we output these periods in one time.

3.4 6B

I. 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.

Here, we use `InvariantBasedAnomaly.java` as our example, we take the 5th checkpont in the history and the argument is:

```
--e1 0 --e2 1 --e3 5 --time-constraint 100000000
```

Increment Off/On	Checkpoint Period	Checkpoint size	Time
Off	5000	57.5 KB	78ms
On	5000	65.4 KB	18ms
Off	1000	15.5 MB	279ms
On	1000	4.07 MB	155ms

II. Is the incremental checkpointing approach always better? When does it make sense? When would taking complete checkpoints make more sense?

The incremental checkpointing is not always better. Incremental checkpoints can dramatically reduce overhead for large state. It would be needed if you have a large state (GB-TB), but between two checkpoints only little changes happen (KB-MB).

There are two reasons for large state: large user state or large operator state coming from joins, windows, or grouping. If it's small and check-

pointing duration is low, there is absolutely no way to go incremental, just complete checkpoints would have excellent performance.

However, there are some trade-off in choosing type of checkpoints. We cannot assert that for example, incremental checkpoint is always best in large state. If the cluster has frequent failures, Flink's TaskManager needs to download the required state files from multiple checkpoints (these files contain some states that have been deleted), and the overall time for job recovery may be longer than without using incremental checkpoints. In addition, we cannot delete the files generated by the old checkpoint, because the new checkpoint will continue to refer to them, which may result in the demand for more storage and may cost more bandwidth when restored.

III. 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?

It just restores the program from a savepoint and specifies a new parallelism, but if increased parallelism is too high, it will not start as there are not enough TaskSlots to execute it. My program continues generating alerts from the point it left off because I set acceptable parallelism. Since Kafka is in exactly-once mode, so I don't see any duplicate when start from savepoints. If we can't guarantee re-settable resources, then it will have duplicates after triggering savepoints.

4 Misc

I would say that the most annoying thing is testing. Although it's not compulsory to write unit test, but I write it in order to prove the rightness of my code. This process is pretty time consuming because I need to pre-calculate the result and what's more, when it related to watermark, there is no watermark set in unit test, as we talked in office hour, you had to add more frequent watermarks so that you can see the result. So I just wander if in industry, how engineer tests their code to prove their correctness in the view of not easy to write test in Flink.
However, there is some positive aspects when I do the project. I enjoy doing some research of benchmark on different checkpointings. It's practical and useful that we can learn some real pros and cons after we do it by our own. Also, the way to set `onTimer()` provides me with a reverse thinking method, not restricted in traditional way which is really interesting.