Author: Xiaoyan Ge

Date: 04/29/2020

Class: CS 591 K1

Submit to: Vasiliki Kalavri

CS 591 Final Project Report

Contents

Design

Pre-requisite

Something you should know before looking at my program. First, the parameters that was required to pass in was implemented by commandlineparser from import org.apache.commons.cli.*. All the parameters were parsed and pass in to the main program. If you run the jar file/ run in Intellij without these parameters, it will raise an error of "missing parameters" and present help message.

One other thing you probably want to know is the watermark interval was set to 1000ms for reading data from Kafka for fault tolerance. The code for reading from local data source was commented. The kafka producer is just the FilterTaskEventsToKafka.java file and I made some change to it. So you must run this file first.

Rule Anomaly Query

First thing I want to talk about is how I implemented rule anomaly query in my program. So the query followed a process of getting event from source (local source or Kafka), **filter** them by event types (e1, e2, e3) passed from command line, **keyed** by jobId and taskIndex, and followed by the **process** function where the store information and actual calculation was done. In the storing process, I have a Map State where it takes the event type as key and task event as value. Basically what is does is store information of latest task event of each event type. I tried to store a list of task event for each type but I got stuck on MapStateDescriptor on how to pass List<Taskevent> in. I need to save this store information until the time constrain was passed and I know it is then safe to do calculation about the anomaly. For example, the onTimer was set at current timestamp + time constraint whenever there is a e1 event. In the onTimer, if the e1 happened before e2, e2 happened before e3, and e3 –e1 > time constraint, we output this anomaly.

Time Anomaly Query

For the time anomaly query, first I filtered all the event of type 1(schedule), and keyed them by machineID. Then I used SlidingEventTimeWindow of *length* and *slide* that was passed in from command line, and did a process window function. In the process window function, I calculated the CPU average of each window and output a tuple 3 of <machineId, TimeStamp, and Average> for further process. Next, I keyed the window event by machineId again, and create a map State of average that keep track of current average. If the number of mapState = number of windows, I started to calculate the moving average by adding the previous average together and divide by size. To decide which is "legal", I checked the timestamp of each and if it is with in the *slide* * (*window* – 1) * 60000 time it was considered as legal. Now we have moving average and current average, we calculate the %diff and output if is larger than *threshold diff* we defined.

Invariant Anomaly Query

For this one basically after I got the data (filter by event type and keyed by priority), I assigned an onTimer that equals to the end of training period (For example time = 10). Before time 10, we made 2 map State the keep tracked of min and maximum of schedule event and submit event. When time = 10, the on Timer was triggered and in the on Timer, I used another value State that store information of boundaries. After cross comparing the minimum and maximum, store the boundaries and the time moved forward to detection period (time > 10). In detection, we checked whether the current Watermark is greater than end of training period, if it is, it means all record are now legal and ready to compute, I used another value state to keep track of all the submit state. And I subtract current task event timestamp to all submit state. If the latency is outside the bound of minMixSchState, we output this as an alert.

Discussion

Rule Anomaly Query

So implemented in a fashion that time constrain period does need to be exhausted for the computation to be done. Like I mentioned in the previous, I hold the state information until CurrentTime + Time constraint and compute it in onTimer. In the case of out of order event, I simply discarded them as illegal events if the timestamp of the event is less than the current watermark. I did this because my approach is only detecting one pair of anomaly event instead all of them. I really doubt this as a huge problem since my watermark interval is set at 1000ms and if it is not too out of order we should be fine. I hope this classify as handling out of order event as well. In the end, my clean up happened at the end of the onTimer so it will be really for the next series of event in the "future". It is only safe to drop the event at end of currentTime + time constraint for the task events of e1.

Time Anomaly Query

So in general, if you increase the threshold, it tends to generate less alerts as it goes. For my test cases, a lot of them has a % difference of 0.9~. Also, towards the end of data stream, I always get a lot of time window that only consist one data record, which contributed to the second part of this problem. If the sliding is not much smaller than the length, I think it will: 1. Output put a lot more windows will mostly unique records in it and waste a lot of RAM and corrupt the usefulness of moving avg as the result, and 2. Getting a lot of 1 record time window which probably will always output alert for such big average CPU.

On the other hand, if the sliding window is small, it will creating a lot of window with duplicate records, hence wastes of CPU and RAMs. So one thing I could think of is instead of output the average each windows, we output the total amount of CPU and done the calculation later. This way we could find of how duplicate the previous window is with the current window, and do something with the different. As the result, the computation power was saved.

Invariant Anomaly Query

One of the way to retrain and adjust the learned by if when the previous bound is generating too much output. This could mean that either the training we chose was an anomaly itself, or the training period is too short. We can implemented this by maintaining the number of output in a period of time by using state(MapState most likely). If the number gets too high, we assign a new timer and update the minimum and maximum boundaries.

One of the reason that storing everything is a bad approach is that you are not doing anything with the data until the training period is done. In my approach, we only storing 2 sets of data: min/max for scheduling and min/max of submitting. The main improvement is that we safe RAM. In the end, in the presence of out of order events, we simply save the data and do nothing until the next event arrives and then do the computation by comparing to the min Max schedule state.

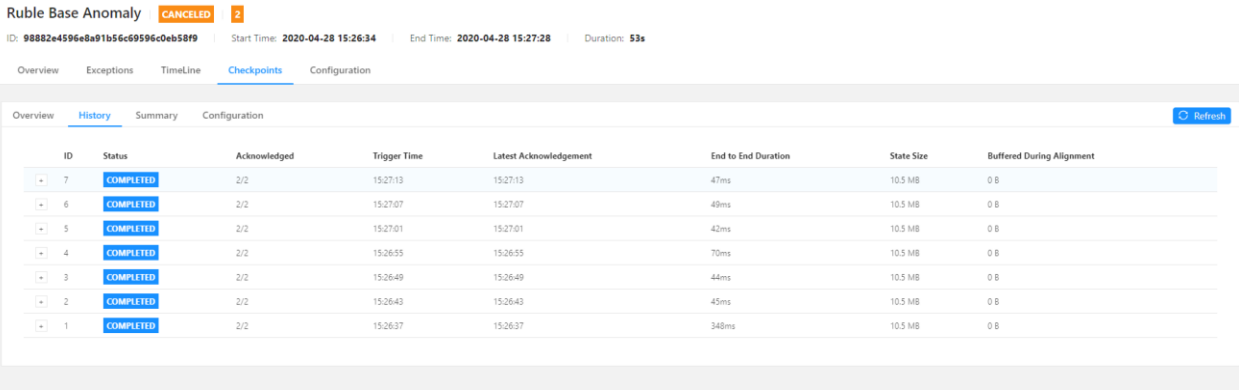
Fault Tolerance

After change backend to RocksDB and I implemented Exactly once checkpoint in my RuleBaseAnomaly Program. The Result are listed in the follow:

Rocksdb Incremental = false
Interval = 60000

ID	Status	Acknowledged	Trigger Time	Latest Acknowledgement	End to End Duration	State Size	Buffered During Alignment
5	COMPLETED	2/2	15:37:26	15:37:26	84ms	19.8 MB	0 B
4	COMPLETED	2/2	15:36:26	15:36:26	88ms	19.8 MB	0 B
3	COMPLETED	2/2	15:35:26	15:35:26	87ms	19.8 MB	0 B
2	COMPLETED	2/2	15:34:26	15:34:26	75ms	19.8 MB	0 B
1	COMPLETED	2/2	15:33:26	15:33:26	92ms	19.8 MB	0 B

Interval = 6000



ID	Status	Acknowledged	Trigger Time	Latest Acknowledgement	End to End Duration	State Size	Buffered During Alignment
7	COMPLETED	2/2	15:27:13	15:27:13	47ms	10.5 MB	0 B
6	COMPLETED	2/2	15:27:07	15:27:07	49ms	10.5 MB	0 B
5	COMPLETED	2/2	15:27:01	15:27:01	42ms	10.5 MB	0 B
4	COMPLETED	2/2	15:26:55	15:26:55	70ms	10.5 MB	0 B
3	COMPLETED	2/2	15:26:49	15:26:49	44ms	10.5 MB	0 B
2	COMPLETED	2/2	15:26:43	15:26:43	45ms	10.5 MB	0 B
1	COMPLETED	2/2	15:26:37	15:26:37	348ms	10.5 MB	0 B

Interval = 600

29	COMPLETED	2/2	15:32:32	15:32:32	90ms	23.8 MB	0 B
28	COMPLETED	2/2	15:32:31	15:32:31	96ms	23.8 MB	0 B
27	COMPLETED	2/2	15:32:31	15:32:31	90ms	23.8 MB	0 B
26	COMPLETED	2/2	15:32:30	15:32:30	88ms	23.8 MB	0 B
25	COMPLETED	2/2	15:32:30	15:32:30	110ms	23.8 MB	0 B
24	COMPLETED	2/2	15:32:29	15:32:29	98ms	23.8 MB	0 B
23	COMPLETED	2/2	15:32:28	15:32:28	94ms	23.8 MB	0 B
22	COMPLETED	2/2	15:32:28	15:32:28	90ms	23.8 MB	0 B
21	COMPLETED	2/2	15:32:27	15:32:27	93ms	23.8 MB	0 B

So in General, the latency on checkpoint was minimum at interval = 6000. In a broad term, I think the latency increases as the time interval decreases. But for some reason, the Intveral = 60000 is a lot of longer than I expected. So I considered that data set as outliers.
On the other hand, when I turned incremental checkpoint on, I ran into a bug of path resolution problem that was addressed in : https://issues.apache.org/jira/browse/FLINK-10918. After communicating with professor, it is due to the version issue and resolved into newer versions. Due to time constraints, I did not accomplish these experiments.

However, I don't think incrementing checkpoint is suitable in every scenario. Incrementing make sense when constraint data flowing in and too costly to use full checkpoint. On the other hand, if the data traffic is not too high and the difference between 2 checkpoint is not significant, we should disable it.

In the end, about the save point. After implementing save point in Flink as instructed, several things as been noticed. When I canceled it and restart the query, there weren't any alert raised. The original task was still marked as canceled. In the web UI dashboard, another task with a different ID was created, but the parallelism stays the same. I'm not quite sure what you mean by duplicates but I think I didn't observe any.