Count distinct elements per time window problem report

Author: Mohamed Taman, Monday, November 18, 2018

The Problem

Is to solve a business requirement for calculating unique users per minute, day, week, month, or year.

For the first version of the application, business wants us to provide just the *unique users per minute*.

Assumptions

- The data consists of (Log)-Frames of JSON data that are streamed into/from apache Kafka.
- Each frame has a timestamp property which is Unix time, the name of the property is `**ts**`.
- Each frame has a user id property called `**uid**`.
- You can assume that 99.9% of the frames arrive with a maximum latency of 5 seconds.
- You want to display the results as soon as possible.
- The results should be forwarded to a new Kafka topic (again as JSON) choose a suitable structure.
- For an advanced solution, you should assume that you cannot guarantee that events are always strictly ordered.

Requirements:

1. Provide a Readme that shows how to build and run the code on Linux or OS X.
2. Write a report: what did you do? What were the reasons you did it like that?
   2.1. Measure at least one performance metric (e.g. frames per second).
   2.2. Document your approach on how you decide **when** to output the data.
   2.3. Document the estimated error in counting.
   2.4. It should be possible to ingest historical data. e.g. from the last 2 years.

Solution Abstract

Here are the breakdown of the requirements and initial detailed explanations:

1. From the assumptions, we can conclude that the counter window is based on the **ts** property in the JSON message, which should be used for that and not the default timestamp assigned by the producer or Kafka to the message.

2. And this will help in calculating historical data easily.

3. For a number of unique users per a specific time window, we can rely on the user "**uid"** property in message alongside the "**ts"** timestamp of this event to be a unique key for a user per window in a specific time window to not be counted more than once.

4. Such user uniqueness state needs to be stored in a data structure to be queried for maintaining the unique data to be not to be passed to the window count calculator.

5. The calculated window needs to be maintained also to have the final updated result of counts to cover late arriving message of this window.

6. Need to maintain the timestamp correctness.

7. All the final window/user counts should be printed to console and/or sent finally as JSON formatted message to a new topic.

The Solution

The requirements suggestions also contain to solutions **basic** and **advanced**.

Basic solution

Is covered in project readme.md as the following:

1. Install Kafka, JDK 1.8, and Maven 3.6. (Done)
2. Running Kafka cluster components. (Done)
3. Create two topics for in and out data processing (**log-frames-topic**, ***users-count-topic***) respectively. (Done)
4. Use the Kafka producer from Kafka itself to send our test data to your topic **log-frames-topic**. (Done)
5. Create a small app that reads this data from Kafka topic **log-frames-topic** and prints it to stdout. (Done)
6. Find a suitable data structure for counting and implement a simple counting mechanism, output the results to stdout. (Done). (*Explained next*)

Advanced Solution

7. Benchmark (Done). (*Explained next*)
8. Output to a new out Kafka topic (***users-count-topic***) instead of stdout. (Done) (Covered in the readme.md file).
9. Try to measure performance and optimize. (*Explained next performance measurements*) (Done)
10. Write about how you could scale. (*Explained next at section: Scalability / Increase throughput*) (Done)
11. Only now think about the edge cases, options and other things. (*Explained next*) (Done)

Project Code

You can clone the project from this link and open it from your favourite IDE as it is a *maven-based project*.

The code is very well documented to understand the working mechanism and how it works.

I am going to explain each component and the main critical and important features it provides and why.

Check the code and it is organized as the following:

1- All classes reside in a parent package `rs.com.sirius.xi.kafka.tm`

2- There are four packages under this parent package.

   a. *Config* package: *contains all common configurations used by all other components.*
   b. *Producer* package: *Contain the code for the producer application*.
   c. *Stream* package: *Contains all stream application and its components*.
   d. *Util* package: *Contains an Utils class that holds all the common functions used by the system.*

Producer application

Bounce question: Make it accept data also from std-in (instead of Kafka) for rapid prototyping. (This might be helpful to have for testing anyway)? *Next is the solution.*

Solution:

The Producer is capable of receiving a file from the command line to process it and send all messages sync/async via provided configurations to a specific topic.

For Async calls it is implemented using callback handler implemented in class ProducerCallBackHandler.

It also calculates the time taken for the message to be sent to the topic and get an acknowledgment back, print it for each message key/value to which topic partition in how many milliseconds.

Streamer Application

Streamer application contains the following components:

- *LogFrameTimestampExtractor* class is a custom implementation to extract timestamp to be used by internal stream API method *windowedBy()* for grouping by the window which in our case per minute.

  It is used by stream API to intersect each message comes from topic after serialization to be processed especially for timestamp values if required any further calculation, in our case we interested in **ts** element of the JSON message to be returned as the message timestamp.

  This way we can calculate any historical data.

- *LogFrameTransformerSupplier* is used to pre-process the message and maintain the count uniqueness. *Explained in Data Structure section*.

- In the Streamer application *LogFrameStreamer* after the pre-processed data is filtered, after using the stream API I *groupBy()* the records with one dummy key called "***window***" for applying the *count()* aggregate function. But we need count users per one minute window (or whatever the window) so I used the *windowedBy()* stream function to partition the grouped users by a specific window.

  With window grouping I used a tumbling window which is the best case here since tumbling windows never overlap, a data record will belong to one and only one window.

  And here is also the trick to managing the late arriving frames (messages) I save the final Window-users count in a state store called "*window-users-count-store*", which keeps the count and update the counter if there is any new late arriving message and send its update.

- The results should be forwarded to a new Kafka topic (again as JSON) choose a suitable structure.
  The output is sent to topic "***users-count-topic***" as the following format:

```
{"Window":"From: Tuesday,November 13,2018 10:40, To: Tuesday,November 13,2018 10:41 ","users":23}
{"Window":"From: Tuesday,November 13,2018 10:41, To: Tuesday,November 13,2018 10:42 ","users":27}
```

Data structure used

Actually, we can use any kind of the following:

Linear List

Search runtime complexity: O(n) **Space Complexity:** O(n)

Hash Set

The HashSet is a good starting point, with optimum speed to search data.
Search runtime complexity: O(1) **amortized insert time:** O(1)) **Space Complexity:** O(n)
where n is the number of unique elements.

HyperLogLog

The HyperLogLog has the same search time complexity as the HashSet, but with really lower memory usage.

Search runtime complexity: O(1) **Space Complexity:** O(ε−2 log log n + log n)
A disadvantage of the HyperLogLog against the HashSet is that the number of uniques is only an estimative so there could be errors in counting, while the HashSet provides the exact value.

Or we can use the implementation of Eclipse Collections which is optimized memory footprint and search for a large number of elements.

Finally, I didn't use any of these. As they need handling across streamer instances and maintain thread access management and more extra work care.

My solution:

I used Interactive Queries with state store. Interactive Queries allows you to get more than just processing from streaming. It allows you to treat the stream processing layer as a lightweight, embedded database and directly query the state of your stream processing application, without needing to materialize that state to external databases or storage.

Apache Kafka maintains and manages that state and guarantees ***high availability and fault tolerance.*** As such, this feature enables the hyper-convergence of processing and storage into one easy-to-use application that uses the Apache Kafka's Streams API which I used in my application.

I have Implemented a *LogFrameTransformerSupplier* class that is plugged-in into Kafka Stream API, to manage the uniqueness of the users per windows in a local store "*unique-users-store*", and check if user already exists in the store and if yes then return to the stream API null, otherwise add the key to the store then forward the message to the Stream processing to be counted per the next window grouping phase.

Have a look at the *LogFrameTransformerSupplier* and *LogFrameStreamer* Application and how they integrated together.

Pros of using Interactive Queries with embedded databases:

- There are fewer moving pieces; just your application and the Kafka cluster. You don't have to deploy, maintain and operate an external database.
- It enables faster and more efficient use of the application state. Data is local to your application (in memory or possibly on SSDs); you can access it very quickly.
- It provides better isolation; the state is within the application. One rogue application cannot overwhelm a central data store shared by other stateful applications.
- It allows for flexibility; the internal application state can be optimized for the query pattern required by the application.
- You can plug in any embedded database of choice (RocksDB comes as default) and still have the database's backend storage kept fault-tolerant and available in Apache Kafka.
- This state store is managed by Kafka Streams internally.
- It is also replicated to Kafka (for fault tolerance and elasticity) topic – this is log compacted topic and nothing but a *changelog* of the local state store contents (this is the default behaviour which is also configurable using the enableLogging() method or can be turned off using disablelogging() .

Cons:

- It may involve moving away from a datastore you know and trust.

- You might want to scale storage independently of processing.
- You might need customized queries, specific to some data stores.

Bounce question: You may want count things for different time frames but only do JSON parsing once. Next is the answer.

Here I have used the default Kafka JSON serdes org.apache.kafka.connect.json.JsonDeserializer/ JsonSerializer to just return the JsonNode directly without further processing.

With the previous solution, we can have parsed JSON once and only extract the required information and returned it as Key/value pair for stream processing.

Frames with a random timestamp (e.g. hit by a bitflip)

Frames with a bitflip could be solved by adding an error correction section protocol on top of the frame (e.g. https://en.wikipedia.org/wiki/Error_detection_and_correction)

Performance measurements:

For checking the memory usage (e.g. use JVisualVM).

For performance, load test and measurements there are great tools actually shipped with Kafka we can use to test the whole Kafka system and it provides a great metrics statistic.

Follow the test commands under the folder benchmark.

This presentation (Producer Performance Tuning for Apache Kafka) is good for explaining the tool and how to measure the important metrics then tune.

Scalability / Increase throughput

We can run multiple instances from our application Streamer that is relevant to the same group, which increase topic processing power. But we have to consider how many instances we can run, to avoid ideal instances. And the following question is very important to answer.

What is the maximum parallelism of my application? And the maximum number of app instances I can run?
Slightly simplified, the maximum parallelism at which your application may run is determined by the maximum number of partitions of the input topic(s) the application is reading from in its processing topology. The number of input partitions determines how many stream tasks Kafka Streams will create for the application, and the amount of stream tasks is the upper bound on the application's parallelism.

Let's take an example. Imagine your application is reading from an input topic that has 5 partitions. How many app instances can we run here?

The short answer is that we can run up to 5 instances of this application because the application's maximum parallelism is 5. If we run more than 5 app instances, then the "excess" app instances will successfully launch but remain idle. If one of the busy instances goes down, one of the idle instances will resume the former's work.

The longer, more detailed answer for this example would be:

- **5 stream tasks** will be created, each of which will process one of the input partitions. And it's actually the number of stream tasks that determines the maximum parallelism of an application – the number of input partitions is simply the main parameter from which the number of stream tasks is computed.

Now that we know the application's theoretical maximum parallelism, the next question is how to actually run the application at its maximum parallelism? To do so, we must ensure that all 5 stream tasks are running in parallel, i.e. there should be 5 processing threads, with each thread executing one task. There are several options at your disposal:

- **Option 1 is to scale horizontally (scaling out):** You run five single-threaded instances of your application, each of which executes one thread/task (i.e., num.stream.threads would be set to 1; see optional configuration parameters). This option is used, for example, when you have many low-powered machines for running the app instances.

- **Option 2 is to scale vertically (scaling up):** You run a single multi-threaded instance of your application that will execute all threads/tasks (i.e., num.stream.threads would be set to 5). This option is useful, for example, when you have a very powerful machine to run the app instance.

- **Option 3 combines options 1 and 2:** You run multiple instances of your application, each of which is running multiple threads. This option is used, for example, when your application runs on a large scale. Here, you may even choose to run multiple app instances per machine.

There are many other factors related to ***Running Kafka in Production*** ranging from Hardware/Software/JVM/Kafka configs and others, those links I use as guides:

1- https://docs.confluent.io/current/kafka/deployment.html
2- https://www.cloudera.com/documentation/kafka/latest/topics/kafka_performance.html

Explain how you would cope with failure if the app crashes mid-day / mid-year.

We will run the application again and it will start from the last offset it stopped at, also if we run multiple instances from our application consumers/streamers in the same group, then if one instance is failed the other will take place, and Kafka broker will manage the assignment of the partition.

Edge cases and further improvements.

1. The Streamer Transformer and Timestamp extractors check for ***poison bill messages*** by eliminating them from further processing. and we can improve this case by sending such messages to a quarantine topic or count the number of corrupted messages and output them to alert topic.

2. The producer could work to take command line data not just a file to process, and it is already handling large data file processing. Also, could handle zipped files.

3. Develop a consumer it is an easy task but needs more time.

4. The streamer could have a configuration to specify how many instances could run at once.