Auditing and replaying billions of streaming events with AWS Athena

Author: Seed Zeng

Claps: 45

Date: May 2, 2018

Klaviyoâ€™s event processing pipeline(AKA ðŸ¤”**the thinking pipe**), tasked with ingesting billions of events from various sources every week, faces data consistency challenge like any other stream processing pipeline. While occasional failures in the streaming process are nicely handled by a general purpose wrapper where failed tasks are retried in place and then automatically replayed (from Elasticsearch) if failure persists, peculiar failure cases that lead to data inconsistency between data storages could still happen.

Those ðŸ‘¹anomalous failure cases can even happen anterior to entering the pipe. As a result, they often cannot be caught by the normal error handling. To preclude *the pipe* from any data inconsistency, we have built a high-performance generic trace logging system for the event processing pipeline. This stand-alone system allows us to log **per-event contextual information** for offline analysis/debugging or raw task logging with negligent amount of overhead. By reading this post, you will understand how we have designed/implemented this subsystem leveraging **AWS S3** and **Athena.**

1. Intro to Athena

Athena is an Amazon Web Service that essentially allows developers to run standard SQL queries on top of S3 files. It is serverless and charges are based on the amount of data your query needs to scan. Since we do not want to develop/manage another streaming pipeline for simple data auditing and events replaying, it suits our needs nicely in this case.

Athena supports structured and unstructured data in S3, such as CSV, JSON, or columnar data formats such as Apache Parquet and Apache ORC. In our use case, we will stick to good old JSON format for our data.

2. Configure S3 Bucket

In order to run Athena queries we need to create an S3 bucket to store our log data. Here are a few things to keep in mind when configuring an S3 Bucket for an Athena query:

Faster & cheaper with partitioning

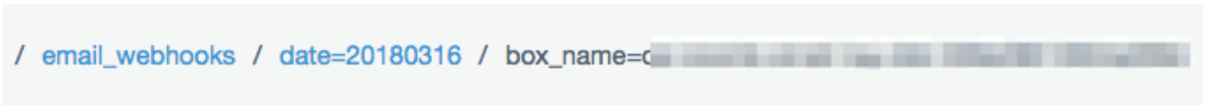
In comparison to Apache Hive that executes MapReduce job against HDFS, Athena runs MapReduce job on top of S3 when tasked with a SQL query. To optimize the MapReduce job by only scanning relevant data, you should leverage the concept called `partitioning`.

You can define `partitions` at table creation time which divides your table into parts and keeps the related data together based on column values such as date, destination, etc. You can then **restrict the amount of data scanned in a query** through a ***WHERE*** clause at run time.

To take advantage of partitioning, you need to structure your S3 path like this:

```
s3://yourBucket/pathToTable/<PARTITION_COLUMN_NAME>=<VALUE>/<PARTITION_COL
```

The structure is evidently dependent upon your workload. For our pipeline, since the payloads are best scoped by date time and worker name (server where the task is processed), we have structured the path like this:

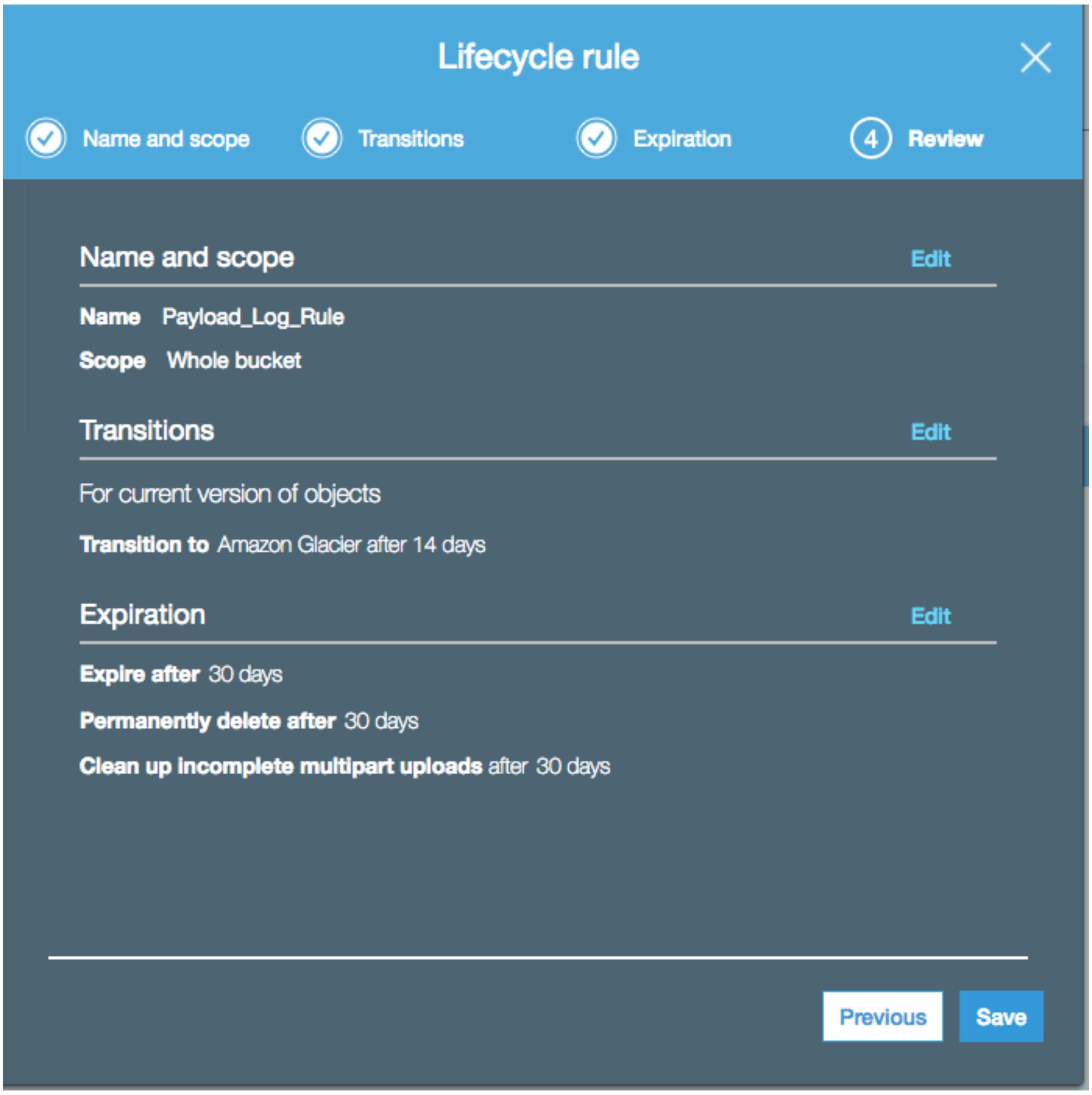


/ email_webhooks / date=20180316 / box_name=c[redacted]

Know your limit

It is often a good idea to somehow bound your S3 bucket size. Luckily, S3 rotation policy makes this rather straightforward. Here is a detailed guide from AWS on how to set lifecycle policy for S3 Bucket.

For us, we have set the lifecycle policy to transition objects to Glacier after 14 days (the time period we are supporting **fast daily audits**) and permanently delete after 30 days.



Name and scope
Edit

Name Payload_Log_Rule

Scope Whole bucket

Transitions
Edit

For current version of objects

Transition to Amazon Glacier after 14 days

Expiration
Edit

Expire after 30 days

Permanently delete after 30 days

Clean up incomplete multipart uploads after 30 days

There is no file like the right file

According to AWS, â€œCompressing your data can speed up your queries significantly, as long as the files are either of an optimal size or splittableâ€�. In general, AWS suggests using Apache Parquet or Apache ORC for compressing files, which compress data by default and are splittable. We chose to take an easier approach where each line is a separate record written in JSON format which can be processed independently by Athena, and the entire file is gzipped to a target size. Empirically, we have found 150MB files to be the proper size to aim for an individual log file to be analyzed by Athena.

Without going into too much detail about how we build our target sized files, we suggest constraining the file size by maintaining a count of lines of the file so that the size of the file can be tied to t**he processing speed of your workload**.

Do not stop here

To read more about how to optimize Athena performance, here is a good place to start.

3. Configure Athena

After data is properly stored in an S3 Bucket, itâ€™s time to create an Athena database and table. Very much like a traditional relational database, you can create databases and tables by running simple SQL DDL statements. You can easily carry this out via Athena UI, but we are going to demonstrate this in Python using our own wrapper (we will go into detail about this wrapper in the following section).

S3 Sample Data

sample data

Create database and partitioned table

First you need to install/configure your boto3 client

```
pip install boto3
```

Now we can use Python to interact with Athena. Again, we will go into detail about `AthenaQuery` wrapper in a bit. Right now, it shall be deemed as magic to run any given SQL statement.

table creation

Notes

- As you can see, we have partitioned the table on date leveraging the S3 file structure illustrated in previous section. When querying this table, we can then filter on this column to scan targeted amount of data.
- For a partitioned table in Athena, you will need to run a repair when new directory (for a partition) is introduced into underlying S3 path.

Simply run

```
MSCK REPAIR TABLE table_name
```

The above command recovers partitions and data associated with partitions. Use this command when you add partitions to the catalog. It is possible it will take some time to add all partitions. If this operation times out, it will be in an incomplete state where only a few partitions are added to the catalog. You should run the statement on the same table until all partitions are added.

For our example table, youâ€™d run

```
MSCK REPAIR TABLE example_audit
```

4. Python Wrapper for audit

Admittedly, one can use the Athena UI to do most query related tasks and boto3 already provides a rather low-level API. One might ponder, why another Python wrapper? Well, we want to achieve three things building a rather lightweight wrapper:

- easy abstraction where we do not have to think about waiting for query results. Athena does not provide a waiter in boto3 (Athena April 6, 2018, boto3 1.7.4)
- easy interface for us to leverage parallel execution ability of async query
- update partition as an option instead of a separate query

Custom Exceptions

```
class AthenaQueryException(Exception): pass
class AthenaReplayServiceException(Exception): pass
```

AthenaQueryResult

A lightweight wrapper around Athena query result.

Input

- execution id
- [optionally] iterms_per_page
- [optionally] max_items
- [optionally] wait_for_result, default to be True

Usage

- get query result for that id (via get_result)

```
1. returns a generator2. invokes pagination with default setting3. page_si
```

- [optionally] wait for the query result to finish since a waiter is not exposed in boto3

AthenaAueryResult

AthenaQuery

A lightweight wrapper around the execution of Athena Query given a SQL statement.

Input

- standard SQL statement(Python string)

Usage

- [optionally] update partition repair
- [optionally] wait for query to finish
- gives back a AthenaQueryResult

AthenQuery

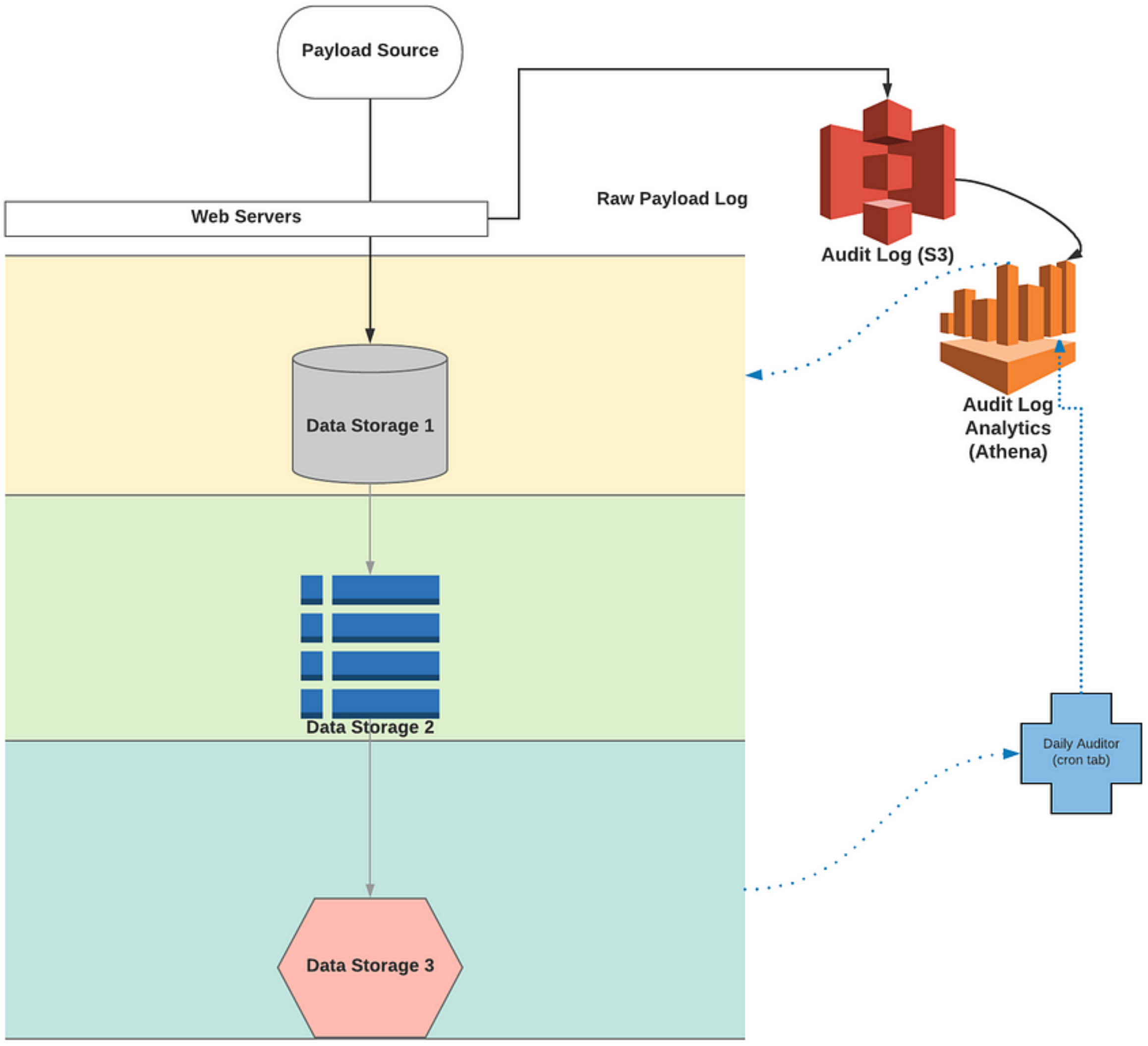
Auditing Workflow

Now with those two Python wrappers, auditing a certain workflow comes down to writing SQL statements specific to the workload. ***Do not*** forget to filter by the target column(date in this example) to leverage partitioning.

5. Beyond

As you can imagine, we did not stop here. While running an audit to figure out whatâ€™s missing and manually replay them can be fun at first, it is tantamount to barbarism if we do not automate the entire process. Klaviyo believes in automating away rote tasks to save developer hours. The end result, beyond the scope of this blog post, can be illustrated by the following graph (a drastically dumbed-down version of the Klaviyo ingestion pipeline)

- daily auditor (cron tab) will audit the data and find potentially missing events
- It then queries Athena (in parallel) to find the missing events
- It then chooses to either replay those events in place or dispatch async tasks back to the ingestion pipeline depending upon the size of the replay tasks
- An engineer identifies any patterns in the missing events that might identify a potential bug. Ideally we donâ€™t need to replay anything, but with Athena we at least have the option to.



6. Time to Build Your Own

Now we have shown you how we leveraged Athena to build a performant system to audit billions of records here at Klaviyo. It is time to start building **your own**. Keep the optimization suggestions I mentioned above in mind and directly leverage the wrapper I shared (if you like Python). With this you are only **days away** from a performant auditing/replaying subsystemðŸ˜Ž.