Scaling Klaviyoâ€™s Event processing Pipeline with Stream Processing

Author: Seed Zeng

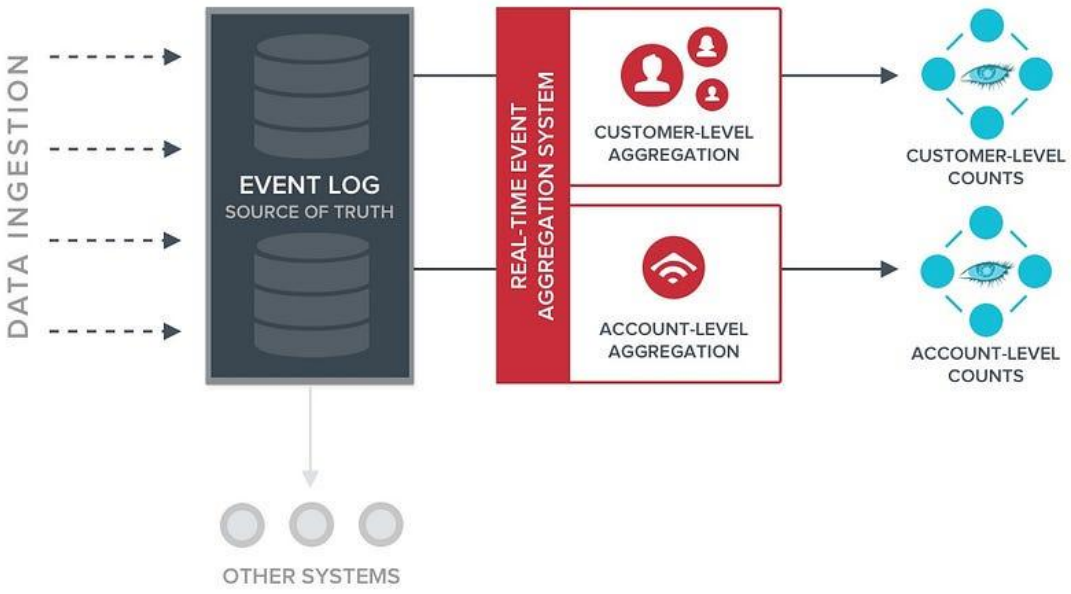
Claps: 437

Date: Jan 23, 2019

Klaviyo has been growing at an incredible rate for the last few years â€” doubling our customer base and ingesting an order of magnitude more data every year. To handle the increasing event volume, we developed a system named Abacus to replace the initial version of Klaviyoâ€™s real-time event aggregation system. Utilizing Apache Flink as its underlying framework and Kafka as its message broker, Abacus is a highly performant stateful stream processing application. In this article, I will cover the challenges of the initial version of Klaviyoâ€™s event aggregation system, the rationale behind choosing Flink as the streaming framework, and how we built and shipped Abacus.

Challenges of Klaviyoâ€™s real-time event aggregation System

The initial version of Klaviyoâ€™s event processing pipeline was a complex series of Celery tasks with RabbitMQ acting as a message bus. Historically, Klaviyo has heavily utilized Celery task processing framework with RabbitMQ, which is well proven for Python data processing workloads. Our overall event processing system consisted of the following components:



Initial Version Event Processing Pipeline

Amongst these components, Klaviyoâ€™s real-time event aggregation system had the most trouble handling the ever-increasing traffic. Although this system served Klaviyo well for many

years, it had some limitations, which weâ€™ll cover below. But first, to fully grasp the challenges that the aggregation system possessed, we need first to understand what Klaviyo aggregates or counts to empower our customers.

Klaviyo Counting 101

This blog post detailed the motivation and application of counting at Klaviyo. Essentially, Klaviyoâ€™s counting is an aggregation of events in real time. Here, I will illustrate this aggregation process using the example of ingesting an email event. Suppose we have an event payload coming in like this :

```
{
  "email": john@example.com",
  "message_id": "MSG123",
  "timestamp": 1544153562,
  "ip": "127.0.0.1",
  "browser": "Safari 12.0.1",
  "type": "open"
}
```

After validation and hydration, the event payload looks something like this:

```
{
  "email": john@example.com",
  "customer_id": "CUS123",
  "message_id": "MSG123",
  "campaign_name": "holiday sales"
  "timestamp": 1544153562,
  "ip": "127.0.0.1",
  "browser": "Safari 12.0.1",
  "statistic_id": "STA123",
  "company_id": "COM123"
}
```

Each key/value pair is a facet of what our customers want to track, so we count their occurrences at both an account (a Klaviyo customer) and an individual customer (our customersâ€™ customers) level. For example, at an account level**,** we increment the count by 1 for company COM123â€™s statistic STA123(email open) for December 7, 2018. At a customer level, we increment the count by 1 for customer CUS123â€™s statistic STA123(email open) for December 7, 2018.

We record counts for the same action for different time resolutions; in addition to December 7, 2018, we record the same action for the hourly time interval when the action falls into(10 PM to 11 PM for timestamp 1544153562). Similarly, we store a record for the monthly interval.

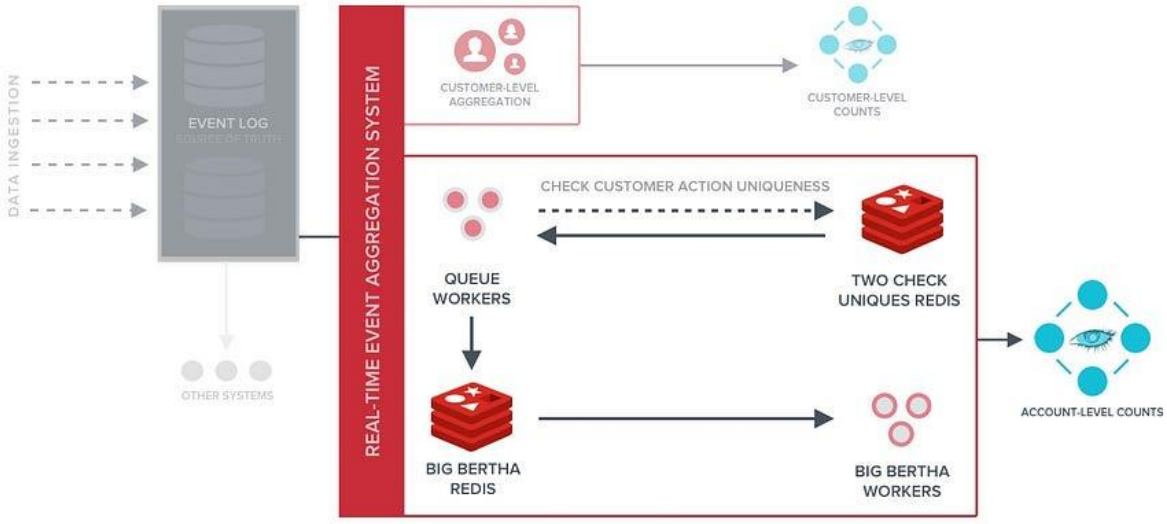
To make this even more challenging, we also record unique counts at an account level to answer questions like â€œ*How many unique individuals opened the campaign â€˜Holiday Salesâ€™.â€*ï¿½ To determine the uniqueness of a particular action, we have to track all actions any customer performed in a certain timeframe. Essentially, we keep a set of actions each customer has performed that looked something like this:

Uniqueness Set

As you can see, a single event can fan out to lots of increments for our system to keep track of, which results in many writes to our counter databases.

Problems of the Initial Version

The initial version of Klaviyoâ€™s real-time event aggregation system was prone to data accuracy issues because the ingestion was **not idempotent**. The aggregation system relied upon two Klaviyo proprietary subsystems â€” *Big Bertha* and *Check Uniques* â€” to process events and write to Cassandra. When the aggregation system ingested an event, the Check Uniques system queried Redis clusters to identify the uniqueness of the associated actions and utilized Big Bertha (a microservice built upon Redis and SupervisorD) to pre-aggregate account-level counts to reduce IO on our Cassandra clusters.



Account-level Aggregation Subsystems

As a result, the systemâ€™s dependency on Big Bertha and Check Uniques made the ingestion of an event no longer idempotent, which meant any failure in these subsystems and their storage tiers could cause data accuracy issues. To audit, debug and heal these ongoing data integrity issues**,** we built a number of automated processes and even built a standalone trace logging system called Athena Logging (discussed in my previous blog post).

Additionally, the system offered no isolation between account-level and customer-level aggregations. Although those workloads impacted different areas of Klaviyo â€” customer-level aggregates were used for customer-level analytics, flow triggering and segmentation, while the account-level aggregates were used by campaign sending and analytics reporting. **The lack of isolation** meant that either systemâ€™s operational incidents could negatively impact the other.

The aggregation system was also **slow.** While average execution time of the entire event processing pipeline hovered around 450 milliseconds, the real-time event aggregation system alone took up to 380 ms. To make things worse, since the system aggregated increments of counts (deltas instead of final counts), we used **Cassandraâ€™s counter data type** to perform the final aggregation. For the counter data type, Cassandra demanded a read before write blocking operation, which made the writing to counter data type less performant than writing to other Cassandra data types. You can find more details about the design and limitations of Cassandraâ€™s counter data type in this blog post.