Apache Flink Performance Optimization

Author: Ning Shi

Claps: 369

Date: Mar 26, 2019

This is the third part of our â€œCountingâ€¦ blog series. In the previous posts, we described what problem we are trying to solve and the technical challenges we face. In this post, we will go deep into the lessons learned from using Apache Flink, especially around performance optimization. This blog post requires knowledge of basic stream processing concepts and Flink usages.

Klaviyoâ€™s workload

Klaviyo has a unique stream processing workload that has high fan-out with a high cardinality dataset. Each second, up to 100,000 events are ingested into our event processing system. Each event can fan-out to hundreds of different dimensions that need to be tracked and counted for different timeframes. That results in hundreds of thousands of writes per second to our Cassandra databases.

Due to the large aggregation windows that may span up to a month, the event processing system has to store a large amount of state. In its compressed form, we have over 1.5TB of state across the event processing cluster. These are hot data kept in the cluster for normal operation. We have close to 1PB of data for warm and cold storage and the number is doubling year over year.

With over 1 billion user profiles in Klaviyo, each dimension we track has high cardinality. The state is large mostly because of the scale of distinct keys. Each entry is actually fairly small, ranging from 8 bytes to 100 bytes.

To solve these technical challenges, we decided to use stream processing technology and picked Apache Flink as the system. A detailed description of the choice and how we implemented the job is in the previous post.

As with any new technology, Flink worked well enough for evaluation, but fell a little short of performance expectations for our specific workload. After understanding how it works, we made a series of code changes and tuning, then the performance increased significantly. In the rest of this post, we describe in detail the changes we have made to get the best performance out of Flink.

The optimizations are divided into two categories, code logic changes and configuration tuning. Some of them are more generic, so they may be more applicable to other Flink use cases. The rest are specific to the high cardinality, high fan-out use case we have at Klaviyo.

Code changes

Processing time vs. event time

Time is at the core of any stream processing system. It is not only used to measure progress, but also used in windowing functions.

The question everyone faces when they start to use stream processing is whether to use event time or processing time. Event time is generated at event creation. It is intrinsic to the event and does not change afterwards. Processing time is the time the event is processed. It has nothing to do with the event itself, but when and where the event is processed. Flink further differentiates processing time generated at each machine in a cluster and processing time generated at the entry point machine in a cluster. The latter is also known as ingestion time because it is generated at the time an event is ingested into Flink.

Most people recommend using event time. It is a good starting point for various reasons. Since event time remains constant to the events, any operation based on it is stable. In other words, operations can generate deterministic results regardless of the throughput or the topology.

On the other hand, processing time is the wall clock time observed by the machine that processes the event. It changes the next time you process the same event. Operations based on processing time is not deterministic. For these reasons, we did not think there was a compelling reason to use processing time.

We used event time when we started evaluating Flink. It worked fairly well with the synthetic workload. We wrote a single custom data source to generate events with monotonically increasing timestamps. Throughput was good and the job worked as expected.

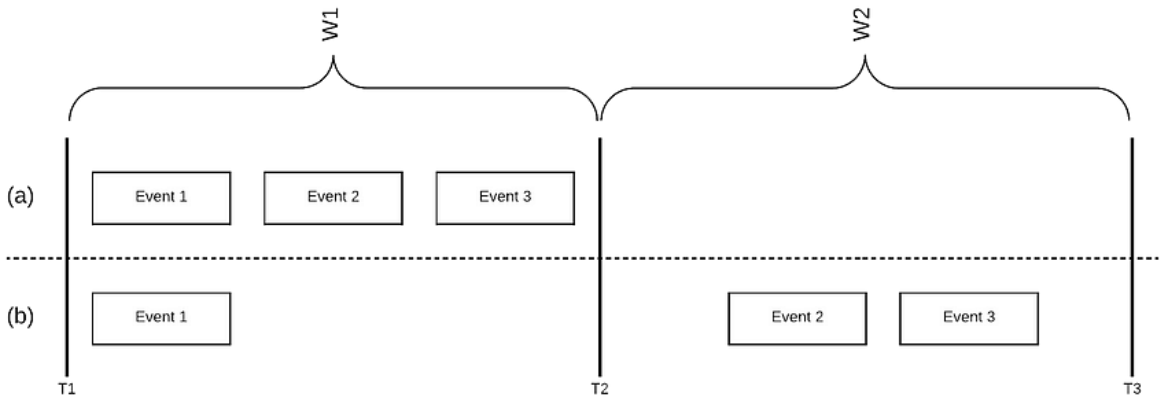
In reality, there are hundreds to thousands of external sources that generate events. Some are third-party ecommerce platforms like Shopify or Magento; however, the rest are custom events using our public API endpoints.

Even though events are supposed to be real-time, the clocks at these event sources are nowhere near synchronized â€” in fact they can be off by hours. This completely breaks the assumption that event time is mostly monotonically increasing. Increasing allowed lateness solves the dropped events problem but large lateness value comes with significant performance impact which I will discuss in a section below. Without setting allowed lateness on the windows, we saw events being dropped due to stale timestamps.

Besides out of order timestamps from real-time events, the ingestion process is also complicated by historical event synchronization running in parallel, ensuring an out of order processing. As we onboard new customers, which we do all the time, we allow them to synchronize all of their historical events since the beginning of their business to us so that they have data continuity. These events can date back years.

With both of these issues in mind, we decided to use processing time and handle both real-time and historical events in the same job.

The catch for moving to processing time is to make sure the business logic can handle events in the same event time window being split into multiple smaller processing time windows. For example, a batch of events can arrive closely together in a 10 minute window in one case and be spread out into three consecutive 10 minute windows over half an hour. They should generate the same results in both cases.



Events can arrive at difference paces. In (a), all three events arrive closely together so they fall into the same window W1. In (b), Event 1 arrives in W1 and Event 2 and 3 arrive later in W2. When using processing time, both cases should produce the same outcome at the end of time T3. Otherwise, the Flink job is not deterministic.

Reduce internal events

As mentioned in the overview, Klaviyo has a very high cardinality dataset due to large amount of user profiles and event metric dimensions. We have hundreds of millions of distinct keys in hot state at any given point of time.

Take the following event for example, there are two approaches to handle the event in the Flink job. One approach is to flat map each dimension into individual internal events and process them separately. The other approach is to keep the complex event intact and process dimensions in a loop when the event is processed.