Engineering

Real-Time Exactly-Once Ad Event Processing with Apache Flink, Kafka, and Pinot

Jacob Tsafatinos, Yuriy Bondaruk, Yupeng Fu, and James Kwon September 23, 2021

Uber recently launched a new capability: Ads on UberEats. With this new ability came new challenges that needed to be solved at Uber, such as systems for ad auctions, bidding, attribution, reporting, and more. This article focuses on how we leveraged open source technology to build Uber's first "near real-time" exactly-once events processing system. We'll dive into the details of how we achieved exactly-once processing as well as the inner workings of our event processing jobs.

Problem Statement:

With every ad served, there are corresponding events per user (impressions, clicks). The responsibility of the ad events processing system is to manage the flow of events, cleanse them, aggregate clicks and impressions, attribute them to orders, and provide this data in an accessible format for reporting and analytics as well as dependent clients (e.g., other ads systems).

This necessitates a system that is optimized for:

b. Customers will get to see their performance metrics with the least amount of delay.

2. Reliability:
   a. The system must be reliable in terms of data integrity. Ad events represent actual money paid to Uber. If events are lost, then Uber loses on potential revenue.
   b. We must be able to accurately represent the performance of ads to our customers. Data loss would lead to underreporting the success of ads, resulting in a poor customer experience.
3. Accuracy:
   a. We can't afford to overcount events. Double counting clicks, results in overcharging advertisers and overreporting the success of ads. Both being poor customer experiences, this requires processing events **exactly-once**.
   b. Uber is the marketplace in which ads are being served, therefore our ad attribution must be 100% accurate.

Architecture

In order to address these requirements, we designed an architecture that heavily relies on 4 key open source technologies: Apache Flink®, Apache Kafka®, Apache Pinot™ and Apache Hive™. Below is the reasoning behind choosing each technology.

Stream Processing with Apache Flink

The core building block of the system uses Apache Flink, a stream processing framework for processing unbounded data in near real-time. It has a rich feature set that's well suited for ads, such as exactly-once guarantees, connectors to Kafka (Uber's choice messaging queue), windowing functions for aggregation, and is well integrated and supported at Uber.

Message Queues with Apache Kafka

Kafka is a cornerstone of Uber's technology stack: we have one of the largest deployments of Kafka in the world and plenty of interesting work has gone into making sure it's performant and reliable. Kafka can also provide exactly-once guarantees, and scales well with the ads use case.

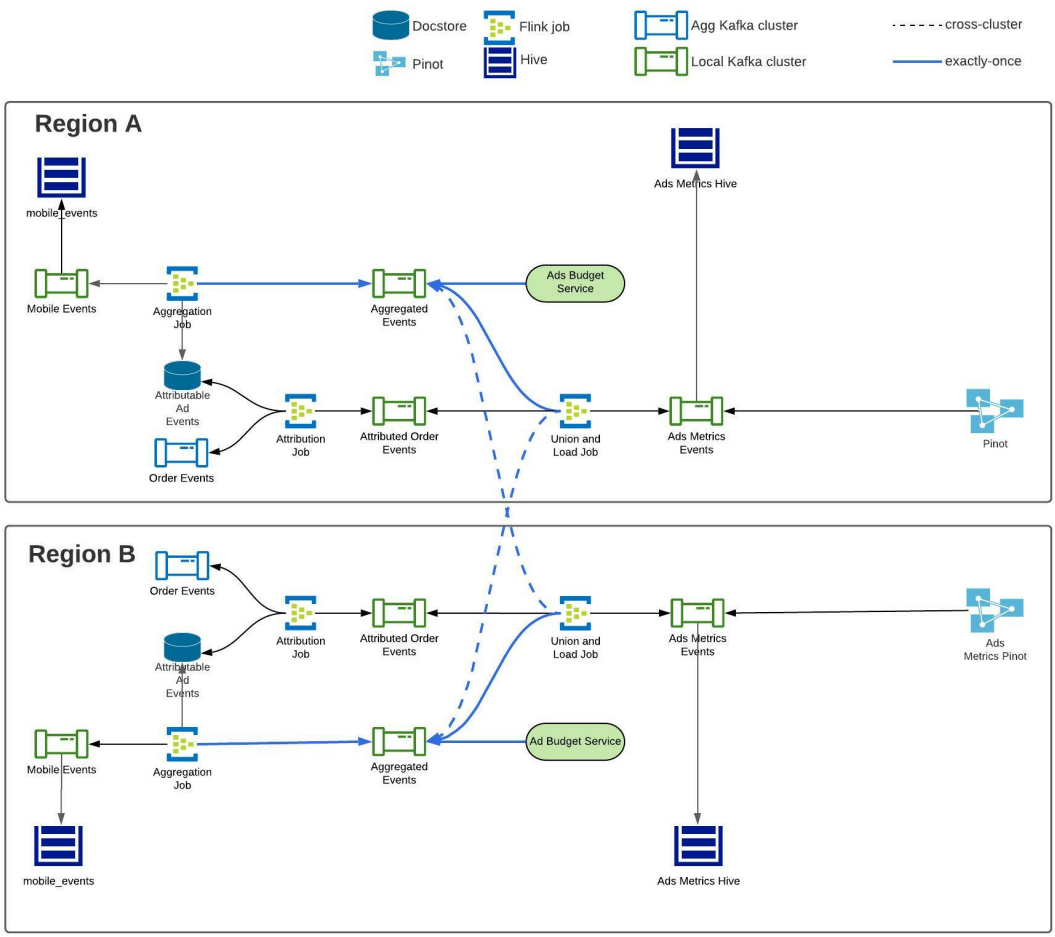
our customers, quickly: in comes Apache Pinot. Pinot is a distributed, scalable, OnLine Analytical Processing (OLAP) datastore. It's designed for low-latency delivery of analytical queries and supports near-real-time data ingestion through Kafka.

Data Warehousing with Apache Hive

Apache Hive is a data warehouse that facilitates reading, writing, and managing large datasets with rich tooling that allows the data to be queried via SQL. Uber has automated data ingestion flows through Kafka, and internal tooling that makes Hive a great solution to store data to be leveraged by data scientists for reporting and data analysis.

High-Level Architecture

Now that we know the building blocks of the system, let's dive into the high-level architecture.



exactly-once semantics enabled to guarantee that only messages that are transactionally committed will be read. Before diving into each Flink job, let's take a pitstop to talk about how this system achieves exactly-once semantics.

Exactly-Once

As mentioned above, a primary constraint that we're working with is requiring exactly-once semantics across the system. It is one of the hardest problems in distributed systems, but we were able to solve it through a combination of efforts.

First, we rely on the exactly-once configuration in Flink and Kafka to ensure that any messages processed through Flink and sunk to Kafka are done so transactionally. Flink uses a KafkaConsumer with "read_committed" mode enabled, where it will only read transactional messages. This feature was enabled at Uber as a direct result of the work discussed in this blog. Secondly, we generate unique identifiers for every record produced by the Aggregation job which will be detailed below. The identifiers are used for idempotency and deduplication purposes in the downstream consumers.

The first Flink job, Aggregation, consumes raw events from Kafka and aggregates them into buckets by minute. This is done by truncating a timestamp field of the message to a minute and using it as a part of the composite key along with the ad identifier. At this step, we also generate a random unique identifier (record UUID) for every aggregated result.

Every minute the tumbling window triggers sending aggregated results to a Kafka sink in an "uncommitted" state until the next Flink checkpoint triggers. When the next checkpointing triggers (every 2 minutes), the messages are converted to the "committed" state using the two-phase commit protocol. This ensures that Kafka read-offsets stored in the checkpoint are always in line with the committed messages.

Consumers of the Kafka topic (e.g., Ad Budget Service and Union & Load Job) are configured to read committed events only. This means that all uncommitted events that could be caused by Flink failures are ignored. So when Flink recovers, it re-processes them again, generates new aggregation results, commits them to Kafka, and then they become available to the consumers for processing.

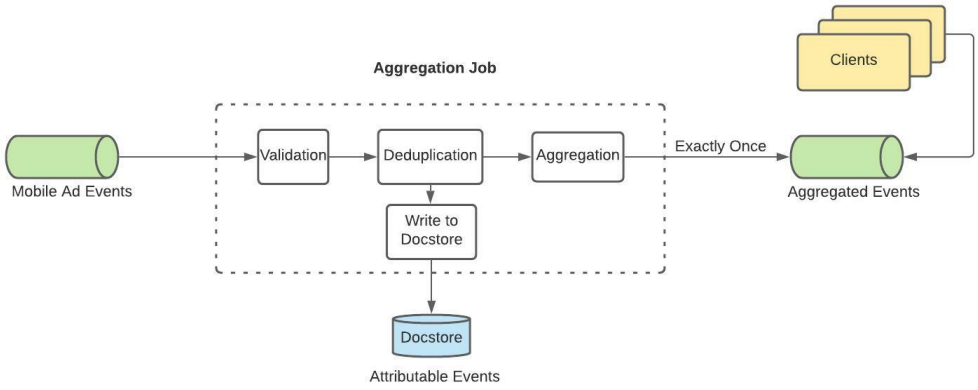
A record UUID is used as an idempotency key in ad-budget service. For Hive it is used as an identifier for deduplication purposes. In Pinot, we leverage the upsert feature to ensure that we never duplicate records with the same identifier.

Engineering

One common challenge Uber had was to update existing data in Pinot with the changelog in Kafka, and deliver an accurate view in the real-time analytical results. For example, the financial dashboard needed to report gross bookings with corrected ride fares, and restaurant owners required analyzing UberEats orders with their latest delivery status. In the ads event processing case, duplicate copies needed to be detected and eliminated. To address this challenge, Uber's Data Infrastructure team made contributions to Pinot, and provided native support of upsert during the real-time ingestion process. This greatly enhanced the data manipulation capability in Pinot, and enabled a large category of use cases. To learn more about this, please check out the user guide on Pinot docs.

Now we'll dive deeper into each Flink job to explore the nuances:

Aggregation Job



The Aggregation Job does a lot of the heavy lifting, so we'll detail it as 4 components: Data cleansing, persistence for order attribution, aggregation, and record UUID generation.

Data Cleansing

We start by ingesting a single stream of ad events representing both ad clicks and impressions from the Mobile Ad Events Kafka topic. Using a Filter operator we validate events on certain parameters like the existence of certain fields or age of the event. Using the Keyby operator we partition the data into logical groupings and we use the Map operator to deduplicate events from the input stream. We leverage Flink's keyed state in the deduplication mapper function to keep

Engineering

Persistence for Order Attribution

Once we have “clean” ad events we store them into a Docstore table (built on top of Schemaless) to be used by the Order Attribution job. We chose Docstore because it’s fast, reliable, and allows us to set a time to live on the rows. This in turn enables the system to only store the events for as long as our attribution window exists.

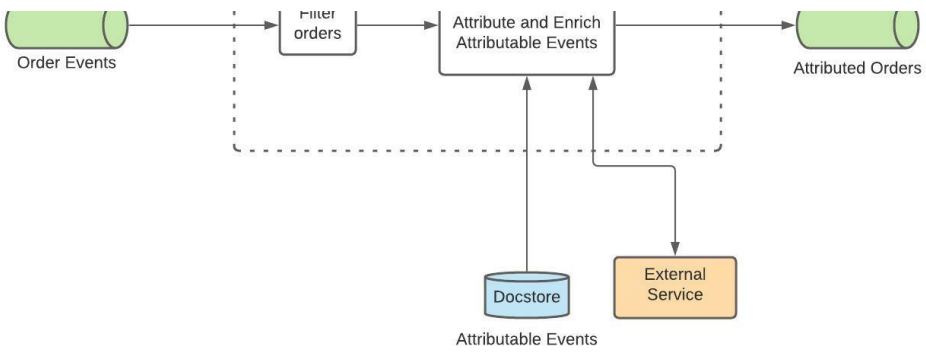
Aggregation

For aggregation, we first key the events based on a combination of an ad identifier and the minute time bucket to which they should correspond—we do this to make sure we’re always aggregating events into the correct time range, regardless of late arrival. Next, we put the events into a Tumbling window of one minute in length. We chose one minute for a few reasons: it’s small enough to be acceptable for dependent clients who rely on the aggregated data; it’s a small enough granularity for analysis (one minute can be rolled up to one hour, 6 hours, and so on); simultaneously it’s a large enough window to not overload our databases with writes. Finally, we use an Aggregation function to count all the clicks and impressions within the tumbling window. From a performance perspective, aggregation functions are great because they only hold one event in memory at a time, allowing us to scale with increased ads traffic.

Record UUID Generation

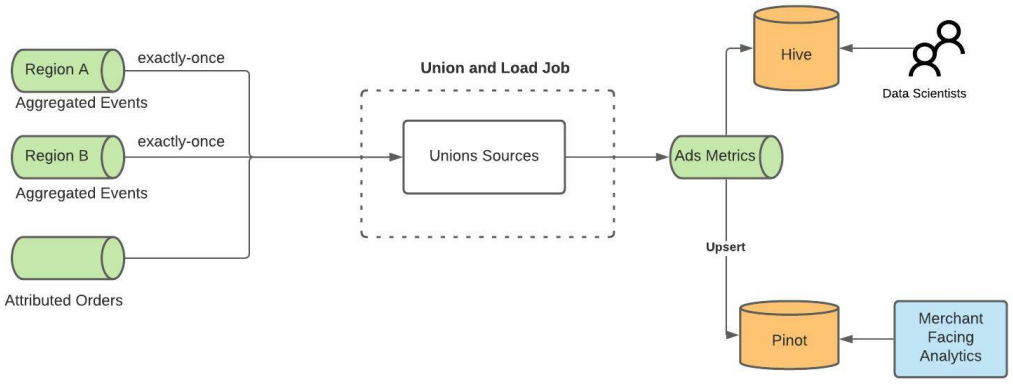
The last piece is to transform the aggregation result into a form that only contains the information we need. As part of this transformation, we generate a record UUID which is used as an idempotency and deduplication key. Given that exactly-once semantics are enabled on Flink/Kafka, we can be confident that once a message is inserted to the destination Kafka topic and committed, then the record UUID can be used for deduplication in Hive and upsert in Pinot.

Attribution Job



The Attribution job is more straightforward. It ingests order data from a Kafka topic that houses all order data for UberEats and filters out invalid order events. It then reaches out to the Docstore table, populated by the Aggregation job, and queries for matching ad events, if there's a match then we have an attribution to make! We then call an external service that provides more details about the order which is used to enrich the events. Then a unique and deterministic record UUID is generated for idempotency for Pinot and Hive. Lastly, we sink the data to the Attributed Orders Kafka topic to be consumed by the Union and Load job.

Union and Load Job



This last job is responsible for unioning events from the output of the Aggregation Job in both Region A and Region B and then sinking the data to an output topic, which disperses data into both Pinot and Hive for end users to query. We have to union events between regions because our deployment of Pinot is Active-Active (it does not replicate between the 2 regions), and so we use this job to ensure that both regions have the same data. In order to maintain exactly-once semantics we leverage the upsert feature in Pinot.

Now that we've gone over the high-level architecture, how we achieve exactly-once and the details of each individual Flink job, we can see how we met all of our requirements:

"processed." The default checkpoint interval is 10 minutes, ours is 2 minutes, so it's not exactly "real-time" but it's good enough to keep our internal systems up to date as well as to report performance of ads in a reasonable time period for our customers.

2. Reliability:
   a. We get reliability from a few different processes. Cross-region replication allows us to failover in case of issues specific to a data center which would otherwise result in data loss. Flink's checkpointing allows us to pick up from where we left off if something goes wrong in processing. We have a 3 day retention period on our Kafka topics in case we need to do some disaster recovery as well.
   b. The biggest pain point for reliability comes from the aggregation job itself. If it goes down there will be a delay in processing events, it can cause various issues in other services, mainly pacing. If we don't process events quick enough, then pacing states won't be updated and could result in overspend. This is an area that still requires some solutioning.
3. Accuracy:
   a. Through a mix of exactly-once semantics and idempotency on Kafka/Flink and upsert on Pinot, we're able to provide the accuracy guarantees we need to have confidence that we'll only ever process a message once.

Parting Thoughts

In this blog we showed how we leveraged open source technology (Flink, Kafka, Pinot, and Hive) to build an event processing system that meets the requirements of a fast, reliable, and accurate system. We discussed some of the challenges with ensuring exactly-once semantics across a distributed system, and showed how we achieved this by generating idempotency keys and relying on one of the most recent features of Pinot: Upsert. As of writing this blog post, this system is processing hundreds of millions of ad events per week, growing more every day.

If you're interested in joining the Ads team, tackling hard problems like the one described above and more, then please apply to join our team!

Photo Credit: "Waterwheel, Pukekura Park, New Plymouth" by russellstreet

Engineering

Jacob Tsafatinos

Jacob Tsafatinos is a former Senior Software Engineer at Uber who led the efforts of the Ad Events Processing system. In his spare time he can be found playing lead guitar in his band Good Kid.

Yuriy Bondaruk

Yuriy Bondaruk is a Senior Software Engineer II at Uber leading the ads platform development, and primarily focuses on core components such as auctioning, bidding, and budget pacing.

Yupeng Fu

Yupeng Fu is a Staff Software Engineer at Uber on the Data Analytics team. He leads several streaming teams building scalable, reliable, and performant streaming solutions. Yupeng is an Apache Pinot committer.

James Kwon

James Kwon is a Software Engineer II at Uber leading the efforts around billing for Ads. He also worked on the Ad Events Processing system with Jacob.

Engineering

Engineering

UberEngineering

UberEngineering

Uber Engineering Blog Categories

AI

Architecture

Culture

General Engineering

Mobile

Open Source

Uber Data

Uber Links

Uber Open Source

Uber Research

Uber.com

Uber Eats

Uber for Business

Help

Newsroom

Careers

Privacy Policy

Terms and Conditions