Kafka at Uber 1

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

- Uber launched Ads on UberEats.
- there are few challanges:
  - ad auctions
  - bidding
  - attribution
  - reporting
- to solve this, we built "near real time" exactly-once events processing system.

Problem Statement:

- with every ad, 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 for reporting and analytics as well as dependent clients (eg. other ads systems)
- This necessitates a system that is optimized for:
- 1. speed:
  - Downstream ad systems (pacing of ads, budget updates) require real-time context of user-generated ad events in order to perform their responsibilities.
  - Customers will get to see their performance metrics with least amount of delay.
- 2. Reliability:
  - 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.
  - performance of ads shown to customer should be accurate. data loss would lead to underreporting the success of ads, resulting in a poor customer experiece.
- 3. Accuracy:
  - We can't afford to over count events.
  - double counting clicks, results in overcharging advertisers and overreporting success of ads.
  - both being poor customer experience, this requires processing events **exactly-once**.

Architecture:

- Apache Flink
- Apache Kafka,
- Apache Pinot
- Apache Hive
- reasoning behind choosing each technology:

Stream Processing with Apache Flink:

- used 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, windowing functions for aggregation, and is well integrated and supported at Uber.

Message Queues with Apache Kafka:

- kafka can also provide exactly-once guarantees, and scales well with ads use case.

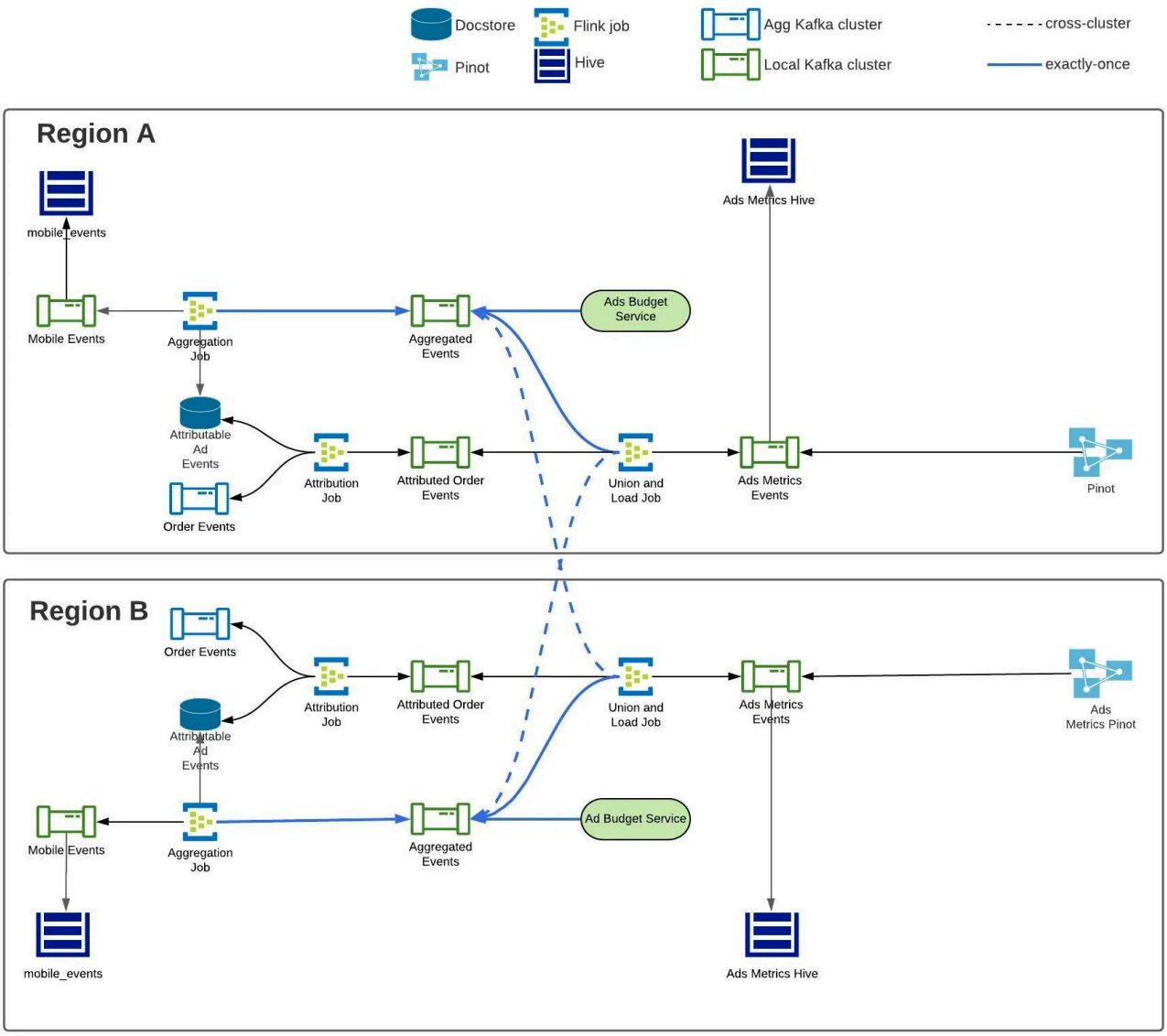
Real-Time Analytics with Apache Pinot:

- One of the main goals of ads events processing system is to provide performance analytics to our customers, quickly.
- Pinot is distributed, scalable, OnLine Analytical Processing (OLAP) datastore.
- designed for low-latency delivery of analytical queries and supports near-real-time data ingestion through kafka.

Data Warehousing with Apache Hive:

- 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.

Architecture:



Exactly-Once

- First, We rely on exactly-once configuration in Flink and Kafka to ensure that any msgs processed through Flink and sink to Kafka are done so transactionally.

```
Kafka supports a special **transactional mode**
where producers (senders) can **batch multiple messages into a single
transaction**.
This helps ensure that all messages in that batch are either **committed
(saved)**
or **discarded** together, avoiding partial or incomplete messages.
```

- Flink uses KafkaConsumer with "read_committed" mode enabled, where it will only read transactional msgs.

```
means that **Flink** will only read messages that have been **successfully committed** (finalized) in Kafka
and ignore any messages that are **in-progress or aborted**.
This ensures that Flink processes only **valid data**.
```

- Secondly, we generate unique identifiers for every record produced by aggregation job.
- The identifiers are used for idempotency and deduplication purposes in downstream consumers.

```
- Idempotency:
        - means processing an event multiple times will have the same result as processing it once.
        - If the event is processed more than once, system will recognize it and ignore it, making sure there are no mistakes like overcounting.
- Deduplication:
        - means even if an event is sent or processed multiple times (perhaps due to a system crash or retry), it will be ignored if the system has already seen it before, bcz unique identifier will match.
```

- First Flink Job, Aggregation, consumes raw events from Kafka and aggregates them into buckets by minute.

```
Let's say you're processing ad impressions. At **10:05:00**,
three impressions are recorded for **Ad-123**:

-   Event 1: `Ad-123`, `user-001`, timestamp 10:05:23
-   Event 2: `Ad-123`, `user-002`, timestamp 10:05:42
-   Event 3: `Ad-123`, `user-003`, timestamp 10:05:57

Flink aggregates these three events into a **minute bucket (10:05)** and generates a **unique identifier** (let's say `UUID-abc123`) for this aggregation.

-   Resulting aggregation: `Ad-123`, bucket `10:05`, `UUID-abc123`, count = 3 impressions.

The goal is to combine events into **minute-level aggregates** and use the **unique identifier** (`UUID-abc123`) to ensure that
```

```
this aggregation is tracked and can be **deduplicated** in case of a
failure.
```

- This is done by truncating a timestamp field of msg to a minute and using it as a part of composite key along with ad identifier.
- At this step, we also generate random unique identifier (record UUID) for every aggregated result.
- Every minute tumbling window triggers sending aggregated results to a Kafka sink in an "uncommitted" state until the next Flink checkpoint triggers.

```
- Let's say that Flink aggregates the impressions for **Ad-123** at 10:05
into `UUID-abc123`
and sends it to Kafka. But at this point, this result is **not committed**
yet.

- Kafka will hold the aggregation in an **uncommitted** state,
meaning it's still **pending**. If something goes wrong (like Flink
crashes),
the system can retry the operation without worrying about **duplicate or
incomplete data**.
```

- When the next checkpointing triggers (every 2 minutes), msgs are converted to "committed" state using 2-Phase commit protocol.

```
The 2-Phase Commit Protocol:

-   When Flink is ready to commit the data, it uses a **2-phase commit
protocol**
to ensure that the data is written correctly to Kafka:

  1.  **Phase 1: Prepare** - Flink signals Kafka to prepare for committing
the data.
  At this point, Kafka holds the data in a "prepare" state but doesn't yet
finalize it.
  2.  **Phase 2: Commit** - After the checkpoint is confirmed,
  Flink tells Kafka to finalize and commit the data.
  At this point, Kafka knows the data is fully processed and **safe to
read**.
```

```
-   This ensures that Flink's **Kafka offsets** (i.e., where it last read
from Kafka) and
the **committed data** are always in sync.
That way, **no events are lost**, and **no duplicates are written**.
```

```
Example:
At **10:07:00**, Flink successfully processes all events for the 10:05
window.
A checkpoint occurs, and Flink sends a "prepare" signal to Kafka to mark
the 10:05 aggregation as ready to be committed.

After the checkpoint, Kafka finally commits the aggregation (`UUID-
abc123`),
and now the data is **committed** and ready for downstream use.

Resulting committed message: `Ad-123`, bucket `10:05`, `UUID-abc123`,
count = 3 impressions.
```

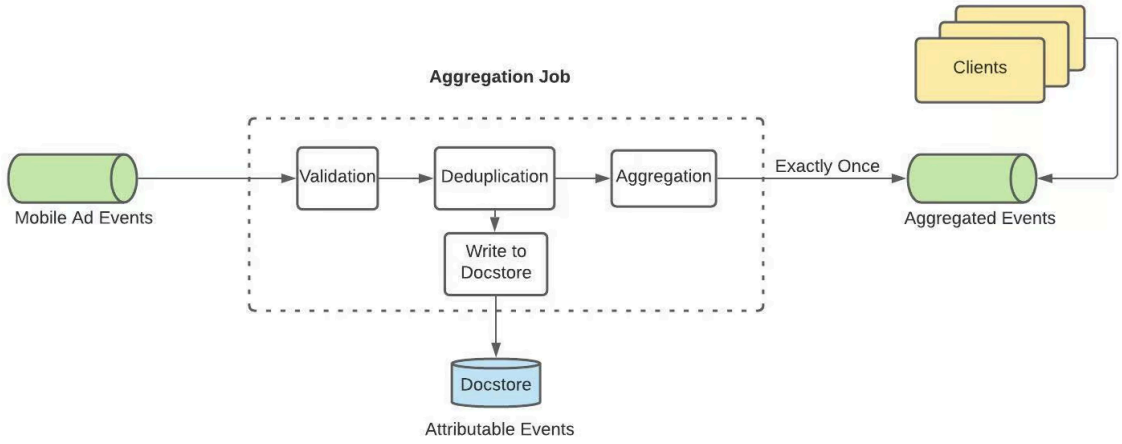
- this ensures that kafka read-offsets stored in checkpoint are always in line with committe msgs.
- consumers of the kafka topic (Ad budget service and Union&Load Job) are configured to read committed events only.
- means that all uncommitted events that could be caused by Flink failures are ignored.
- 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 idempotency key in ad-budget service.
- For Hive it is used as an identifier for deduplication purposes.
- In Pinot, we leverage upsert feature to ensure that we never duplicate records with same identifier.

Upsert In Pinot:

- ### 1. Gross Bookings Dashboard (Financial Data)
  - Imagine that Uber is tracking the **gross bookings** for rides, but sometimes ride fares get **updated** after the ride is completed. For example, a customer may receive a refund, or there could be a fare adjustment due to an error.
  - Without upsert:
    - You would have to delete the old booking record and insert a new one.

        - This might result in **inconsistent reports** because the system would momentarily have **no data** for that booking during the deletion.
    - With **upsert**:
        - When the updated booking fare is received (e.g., through a Kafka changelog), the **new fare** replaces the old one in **Pinot**, and the updated booking amount is reflected immediately in your **real-time analytics**.
        - No duplicates are created, and the dashboard always shows the **correct** data.
- **2. UberEats Delivery Status**
    - Another example is **UberEats** delivery orders. Let's say a customer places an order, and at first, it's marked as **"pending"**. Later, it gets updated to **"delivered"** once the order is completed.
    - Without upsert:
        - If you have two records (one for the initial "pending" status and another for the updated "delivered" status), it might be difficult to track the **latest** state in reports, leading to confusion.
    - With **upsert**:
        - The **"pending"** order record is replaced with the **"delivered"** status when the update comes in.
        - This means that analysts looking at the order status in real-time will always see the **most up-to-date information**, with no duplicates.

Aggregation Job:



- Data cleansing
- persistence for order attribution
- aggregation
- record UUID generation

1. Data Cleansing:
    - we start by ingesting a single stream of ad events representing both ad clicks and impressions from Mobile ad events Kafka topic.
    - Using a Filter operator we validate events on certain params like existance of certain fields or age of the events.
    - Using Keyby operator, we partition data into logical groupings and
        - Eg. If we're processing events for different ads, we might use the **ad ID** to group all events related to that ad. All events with **ad ID = "123"** would go into one group, and events with **ad ID = "124"** would go into another group.
    - we use Map Operator to deduplicate events from input stream.
        - Eg. Suppose a user clicks on the same ad twice in a short time. The system will only count that click once by using the **Map operator** to remove duplicate events.
    - leverage Flink's keyed state in deduplication mapper function to keep track of previously seen events.
    - this ensures that we remove duplicates and protects against certain kinds of fraud.
2. Persistence for Order Attribution

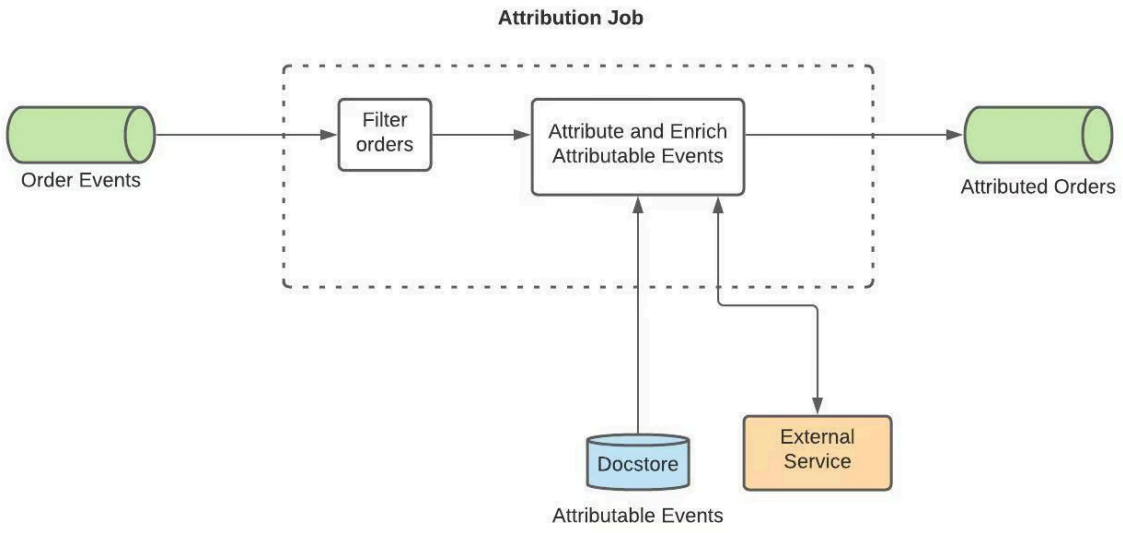
```
- we are temporarily storing the events in docstore instead
of directly consuming from kafka, clean and aggregate and send
-  bcz order attribution involves tracking relationship between
ads (like clicks/impressions) and
user orders (like purchase/order confirmation)
- To do this, we need to maintain state info across multiple ad events
- Example:
        - If a user clicks on an ad at **10:00 AM** and then places an
order at **10:05 AM**,
        you need to track that the click was associated with the later
order.
        - By storing the raw ad events in a **Docstore**, you can **join**
them later to attribute the order correctly.
        If you tried to **directly consume from Kafka**, you wouldn't be
able to store these events
        and associate them later on.
        Storing them in a Docstore gives you time to process and combine
them for attribution.
```

    - After cleaning ad events, we store them into a Docstore table (built on top of schemaless) to be used by Order Attribution job.

- We chose Docstore because it’s fast, reliable, and allows us to set a time to live on the rows
- it enables system to only stores events for as long as attribution window exists.

3. Aggregation
   - we first key the events based on combination of ad identifier and minute time bucket to which they should correspond
   - making sure we always aggregate events into correct time range, regardless of late arrival.
   - Next, we put events into Tumbling window of 1 minute length.
   - we chose 1 minute bcz: it's small enough to be acceptable for dependent clients who rely on aggregated data; it's a small enough granularity for analysis.
   - also, it's a large enough window to not overload the DB with writes.
   - Finally, we use aggregation function to count all clicks and impressions within tumbling window.
4. Record UUID generation
   - as a part of the aggregation process, 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 msg is inserted to the destination kafka topic and committed, then record UUID can be used for deduplication in Hive and upsert in Pinot.

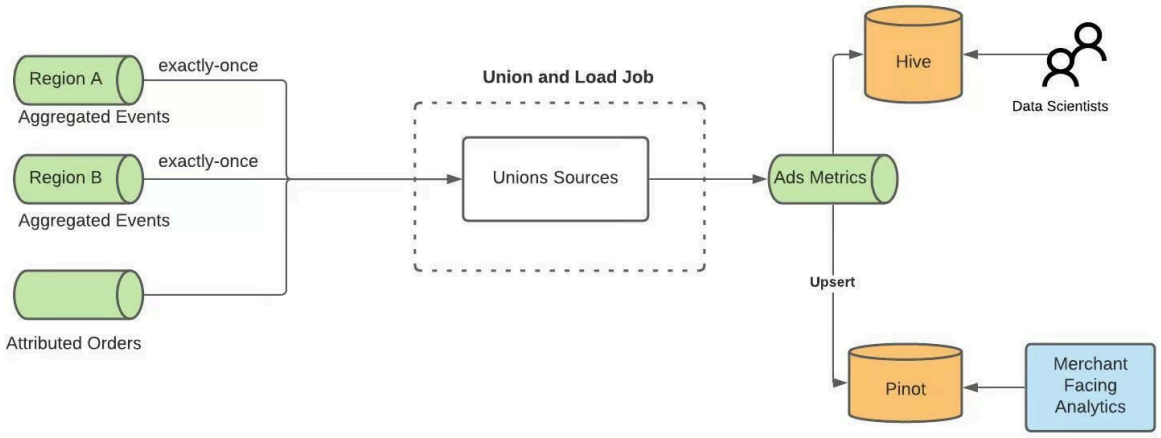
Attribution Job:



- straightforward

- ingests order data from kafka topic for ubdereats
- and filters out invalid order events.
- then reaches out to docstore table, populated by 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 order which is used to enrich the events.
- then a unique record UUID is generated for idempotency in Pinot and Hive.
- Lastly, we sink data to the attributed orders kafka topic to be consumed by union and Load Job.

Union and Load Job:



- this last job is responsible for unioning events from output of aggregation job in both region A and region B and then sinking data to an output topic, which disperses data into both Pinot and Hive for end user query.
- we have to union events between regions bcz our deployment of Pinot is Active-Active (it doesn't replicate between 2 regions) and so we use this job to ensure that both regions have same data.
- In order to maintain exactly-once semantics, we leverage upsert feature in Pinot.

Result:

1. Speed:
    - The main bottleneck for time is the Flink checkpoint interval.
    - Given our exactly-once constraints, we have to wait for the checkpoint to be committed before events can be called “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:
    - 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.
    - 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:
    - 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.