Backfill Streaming Data Pipelines in Kappa Architecture

Xinran Waibel
Senior Data Engineer, Netflix

Sundaram Ananthanarayanan
Senior Software Engineer, Netflix
Agenda

❖ Why backfill streaming pipelines
❖ Existing approaches
❖ Backfill in Kappa Style using Data Lake
❖ Event ordering challenges
❖ Adopting Kappa backfill
Event streaming at Netflix

Personalization DE built various data systems that power data analytics and ML algorithms.

Real-time Merched Impression (RMI) Flink App:
- Join Impression events with Playback events in real-time to attribute plays to impressions.
- Use Cases: Algo training, AB test analysis, etc.
- One of the largest stateful Flink apps at Netflix.
Event streaming operations

Streaming apps can fail due to various reasons:

- Source / sink failures
- Dependent service failures
- Upstream data changes

After failures, we need to backfill to mitigate downstream impact.
Event streaming operations

Possible types of backfilling needs:

- Correcting wrong data
- Backfilling missing data
- Bootstrapping state
How should we backfill?
Option #1: Replaying source events

The easiest way to backfill is by re-running the streaming job to reprocess source events from the problematic period.

Challenges

😭 Troubleshooting can take hours or days and source data can expire.
😭 Increasing message queue retention is very expensive.

- Row-based formats (e.g. Avro) have lower compression rate (v.s. Parquet/ORC).
- Low-latency storage solutions (e.g. EBS gp2) are more costly (v.s. S3).
- It would cost Netflix $93M/year to retain 30 days of data generated by all apps.
Can we store events somewhere else?

Netflix’s Keystone\(^1\) platform provides a routing service that makes Kafka events available in other storage systems, e.g. a **data lake** for batch processing.

Why Data Lake?
What is a data lake? 🏞️

A data lake\(^1\) is a central location that stores a large amount of data in its native raw format, using a flat architecture and object storage.

- Frameworks: Delta Lake, Apache Iceberg (Netflix’s choice)

**Why data lake?**

💖 **Cost effective**: data are stored in compressed formats e.g. Parquet.
💖 **Other features**: file pruning, schema evolution, engine-agnostic, etc.

---

Kafka events stored in an Iceberg table

<table>
<thead>
<tr>
<th>account_id</th>
<th>show_id</th>
<th>view_duration</th>
<th><strong>metadata</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>98524989</td>
<td>4236781</td>
<td>123</td>
<td>{kafka_ingestion_ts: ...}</td>
</tr>
<tr>
<td>87934298</td>
<td>8754782</td>
<td>45</td>
<td>{kafka_ingestion_ts: ...}</td>
</tr>
<tr>
<td>79403754</td>
<td>3648295</td>
<td>81</td>
<td>{kafka_ingestion_ts: ...}</td>
</tr>
</tbody>
</table>

Can we backfill from the data lake?
Option #2: Lambda Architecture

Build and maintain a batch-based application (e.g. Spark job) that is equivalent to the streaming application but reads from Iceberg tables.
Option #2: Lambda Architecture

Build and maintain a batch-based application (e.g. Spark job) that is equivalent to the streaming application but reads from Iceberg tables.

Challenges

😃 Initial development of such batch job can take days or weeks, incl. data validation between two different applications.
😃 Continuous engineering efforts to keep the batch app up to date.
Option #3: Unified batch and streaming
Taking two birds with one stone?

Frameworks

- Apache Flink: offers both batch and streaming modes.
- Apache Beam\(^1\): a unified programming model for batch and streaming data processing pipelines.

Limitations

😭 Flink requires significant code changes to run batch mode.
😭 Beam only has partial support on state, timers, and watermark\(^2\).

[1] https://beam.apache.org/about/
Backfill Option Comparison

Pros & cons in summary

**Rerunning Streaming Job**
- Method: Rerun the streaming app before source data expire.
- Pros: Backfill using the same app.
- Cons: Increasing message queue retention is expensive.

**Separate Batch Job**
- Methodology: Maintain an equivalent batch app reading from a data lake.
- Pros: Low data retention cost in data lake.
- Cons: Engineers have to maintain two applications in parallel.

**Unified Batch & Streaming**
- Prerequisite: Use a framework with both batch & streaming modes.
- Pros: Backfill using the batch mode.
- Cons: Might still require significant code changes.
Can we combine the best things from all three worlds?
Backfilling In Kappa Architecture (feat. Data Lake)
Backfilling using Data Lake: Goals

- Provide a generic solution that works for all classes of applications
- Minimal code changes to add support
- Scales horizontally to backfill quickly
Backfilling using Data Lake: Overview

- **Live source** (optimized for latency)
- **Data lake** (optimized for throughput)
- **Streaming job**
- **Sink**
Backfilling using Data Lake: Overview

Semantics

Recent Past

Handling real-time data

Sink
Backfilling using Data Lake: Overview

Semantics

- Distant Past
- Recent Past

Backfilling

Sink
Ingesting streaming data into data lake
Ingesting streaming data into data lake

(date=20220611, hr=1, batch=1)
Ingesting streaming data into data lake

(hr = 1) (date=20220611, hr=1, batch=1)  (date=20220611, hr=1, batch=2)

(hr = 2)
Ingesting streaming data into data lake

✔ Batching events results in good compression ratios.
✔ Avoids small file problem.
How to backfill?

• Strawman 1: Read events from files filtered by backfill dates
Strawman 1: Read events from selected files

(date=20220611, hr=1, batch=1)

(date=20220611, hr=1, batch=2)

(date=20220611, hr=2, batch=1)
Strawman 1: Read events from selected files

Date: 20220611, Hour: 2, Batch: 1

Date: 20220611, Hour: 1, Batch: 2

Date: 20220611, Hour: 2, Batch: 1

Datastream
Strawman 1: Read events from selected files

- (date=20220611, hr=1, batch=1)
- (date=20220611, hr=1, batch=2)
- (date=20220611, hr=2, batch=1)
Strawman 1: Read events from selected files

(date=20220611, hr=1, batch=1)  
(date=20220611, hr=1, batch=2)  
(date=20220611, hr=2, batch=1)
How to backfill?

- **Strawman 1**: Read events from files filtered by backfill dates
  - ✔ Scales horizontally to backfill quickly
  - ✗ Does not work for all types of applications
Challenge #1: Applications assume ordering

**Example:** Application that converts playback events into playback sessions

![Diagram showing the conversion of playback events into playback sessions]
Challenge #1: Applications assume ordering
Challenge #1: Applications assume ordering.

Different ordering leads to different results.
Strawman 2: Order all files and read in order

(reader) (date=20220611, hr=1, batch=1)
(reader) (date=20220611, hr=1, batch=2)
(reader) (date=20220611, hr=2, batch=1)
Strawman 2: Order all files and read in order

(reader)

(datastream)

(reader)

(reader)

(reader)
Strawman 2: Order all files and read in order

(reader)

(datastream)

Split assigned

(reader)

(reader)
**Strawman 2:** Order all files and read in order

![Diagram showing file order and batch assignments](image)
How to backfill?

- **Strawman 1**: Read events from files filtered by backfill dates
  - ✓ Scales horizontally to backfill quickly
  - ✗ Does not work for all types of applications

- **Strawman 2**: Order all files and read them in order
  - ✓ Guarantees similar ordering semantics as the live traffic
  - ✗ Does not scale horizontally
But, not all streaming apps rely on strong ordering guarantees.
Event–Time Semantics

Example: Application that converts playback events into playback sessions
Event-Time Semantics

**Example:** Application that converts playback events into playback sessions

1. Sessions are derived from event timestamps - not ingestion times.
2. Because events can arrive late, applications tolerate lateness.
Idea: Use lateness tolerated by app
Solution: Use lateness tolerated by app

Represents a file that has streaming data for minutes 10 to 15.
Solution: Use lateness tolerated by app

Assuming lateness of “10” minutes is okay.
Solution: Use lateness tolerated by app

Represents the ingestion timestamp up to which live data has been fully processed.
Solution: Use lateness tolerated by app

Split assigned

reader

Split assigned

reader

Split assigned

reader

T=0..5  T=0..5  T=5..10  T=5..10  T=10..15
Solution: Use lateness tolerated by app

T=0..5  T=0..5  T=5..10  T=5..10  T=10..15

watermark

reader  reader  reader

datastream
Solution: Use lateness tolerated by app

Indicates all data up to 5 minutes has been processed.
Solution: Use lateness tolerated by app

\[ T = 0..5 \]
\[ T = 0..5 \]
\[ T = 5..10 \]
\[ T = 5..10 \]
\[ T = 10..15 \]

\[ \text{IW} = 5 \]

reader

reader

reader

\text{datastream}
How to backfill?

- **Strawman 1:** Read events from files filtered by backfill dates
  - ✔ Scales horizontally to backfill quickly
  - ✗ Does not work for all types of applications

- **Strawman 2:** Order all files and read them in order
  - ✔ Guarantees similar ordering semantics as the live traffic
  - ✗ Does not scale horizontally

- **Our Solution:** Read files while maintaining lateness constraints
  - ✔ Guarantees ordering that work for the application
  - ✔ Scales horizontally to finish backfill quickly
Messaging System’s Ordering Guarantees

- Kafka provides strict ordering of events within a partition.

- Most analytical use-cases (streaming-joins, sessionization) use event-time semantics and do not require such stronger guarantees.
Challenge 2: Reading Multiple Sources

- One source can have significantly way more data than the other.
- During backfill, this could lead to a watermark skew resulting in state size explosion.
- This can eventually lead to slow checkpoints or checkpoint timeouts.
Solution: Coordinate watermarks
Solution: Coordinate watermarks

Communicate watermark updates to the global tracker.
Solution: Coordinate watermarks

Global watermark should reflect the slowest source.
Solution: Coordinate watermarks

Use the *global watermark* to find if files can be dispatched without violating the ‘lateness’ constraint.
How to backfill?

- **Strawman 1**: Read events from files filtered by backfill dates
  - ✔ Scales horizontally to backfill quickly
  - ✖ Does not work for all types of applications

- **Strawman 2**: Order all files and read them in order
  - ✔ Guarantees similar ordering semantics as the live traffic
  - ✖ Does not scale horizontally

- **Our Solution**: Read files while maintaining lateness constraints
  - ✔ Guarantees ordering that work for the application
  - ✔ Scales horizontally to finish backfill quickly
  - ✔ Alignment across sources to avoid state size explosion
Agenda

- Why backfill streaming pipelines
- Existing approaches
- Backfill in Kappa Style using Data Lake
- Event ordering challenges
- Adopting Kappa backfill
Adopting Kappa Backfill

RMI Flink Application

Production Stack

Backfill Stack

Impression Kafka Topic

Playback Kafka Topic

Playback Iceberg Table

Impression Iceberg Table

RMI Output Kafka Topic

RMI Output Iceberg Table
Adopting Kappa Backfill

Minimal code changes

```scala
@SpringBootApplication
class PersonlizationsStreamingApp {
    @Bean
def flinkJob(
        @Source("impression-source") impressionSource: SourceBuilder[Record[ImpressionEvent]],
        @Source("playback-source") playbackSource: SourceBuilder[Record[PlaybackEvent]],
        @Sink("summary-sink") summarySink: SinkBuilder[ImpressionPlaySummary]) {...}

    @Bean
def liveImpressionSourceConfigurer(): KafkaSourceConfigurer[Record[ImpressionEvent]] =
        new KafkaSourceConfigurer("live-impression-source", KafkaCirceDeserializer[ImpressionEvent])
}
```
Adopting Kappa Backfill

Minimal code changes

```java
@Component
class PersonalizationsStreamingApp {
    @Bean
def flinkJob(
        @Source("impression-source") impressionSource: SourceBuilder[Record[ImpressionEvent]],
        @Source("playback-source") playbackSource: SourceBuilder[Record[PlaybackEvent]],
        @Sink("summary-sink") summarySink: SinkBuilder[ImpressionPlaySummary]) {...}

    @Bean
def liveImpressionSourceConfigurer(): KafkaSourceConfigurer[Record[ImpressionEvent]] =
        new KafkaSourceConfigurer("live-impression-source", KafkaCirceDeserializer[ImpressionEvent])

    @Bean
def backfillImpressionSourceConfigurer(): IcebergSourceConfigurer[Record[ImpressionEvent]] =
        new IcebergSourceConfigurer("backfill-impression-source", Avro.deserializerFactory[ImpressionEvent])
}
Adopting Kappa Backfill

Minimal code changes

```java
@SpringBootApplication
class PersonlizationsStreamingApp {
    @Bean
def flinkJob(
        @Source("impression-source") impressionSource: SourceBuilder[Record[ImpressionEvent]],
        @Source("playback-source") playbackSource: SourceBuilder[Record[PlaybackEvent]],
        @Sink("summary-sink") summarySink: SinkBuilder[ImpressionPlaySummary]) {...}

    @Bean
def liveImpressionSourceConfigurer(): KafkaSourceConfigurer[Record[ImpressionEvent]] =
        new KafkaSourceConfigurer("live-impression-source", KafkaCirceDeserializer[ImpressionEvent])

    @Bean
def backfillImpressionSourceConfigurer(): IcebergSourceConfigurer[Record[ImpressionEvent]] =
        new IcebergSourceConfigurer("backfill-impression-source",
            Avro.deserializerFactory[ImpressionEvent])
}
```

Note: In-memory representation of the Iceberg source is consistent with the Kafka Source.
Adopting Kappa Backfill

Minimal code changes

nfflink:
  job.name: rmi-app
  connectors:
    sources:
      impression-source:
        type: dynamic
        selected: live-impression-source
        candidates:
        - live-impression-source
        - backfill-impression-source
      live-impression-source:
        type: kafka
        topics: impressions
        cluster: impressions_cluster
      backfill-impression-source:
        type: iceberg
        database: default
        table: impression_table_name
        max_misalignment_threshold: 15min

App config changes to support backfilling
Adopting Kappa Backfill

What we learned from backfilling in prod

Results

• High throughput: processing 24 hours of data takes ~ 5 hours.
• Consistent data quality: backfill output matches 99.9% with prod.

Lessons Learned

• Backfilling window and configs depend on application logic.
• Backfilling job needs tuning (separately from prod job).
Kappa Backfill benefits

👏 Use the same streaming application for production and backfilling
👏 Easy to set up
👏 Backfill large historical data quickly
👏 Cost Efficient ($2M/year in Iceberg v.s $93M/year in Kafka)
Thank you