DATA+AI SUMMIT 2022

Backfill Streaming Data Pipelines in Kappa Architecture

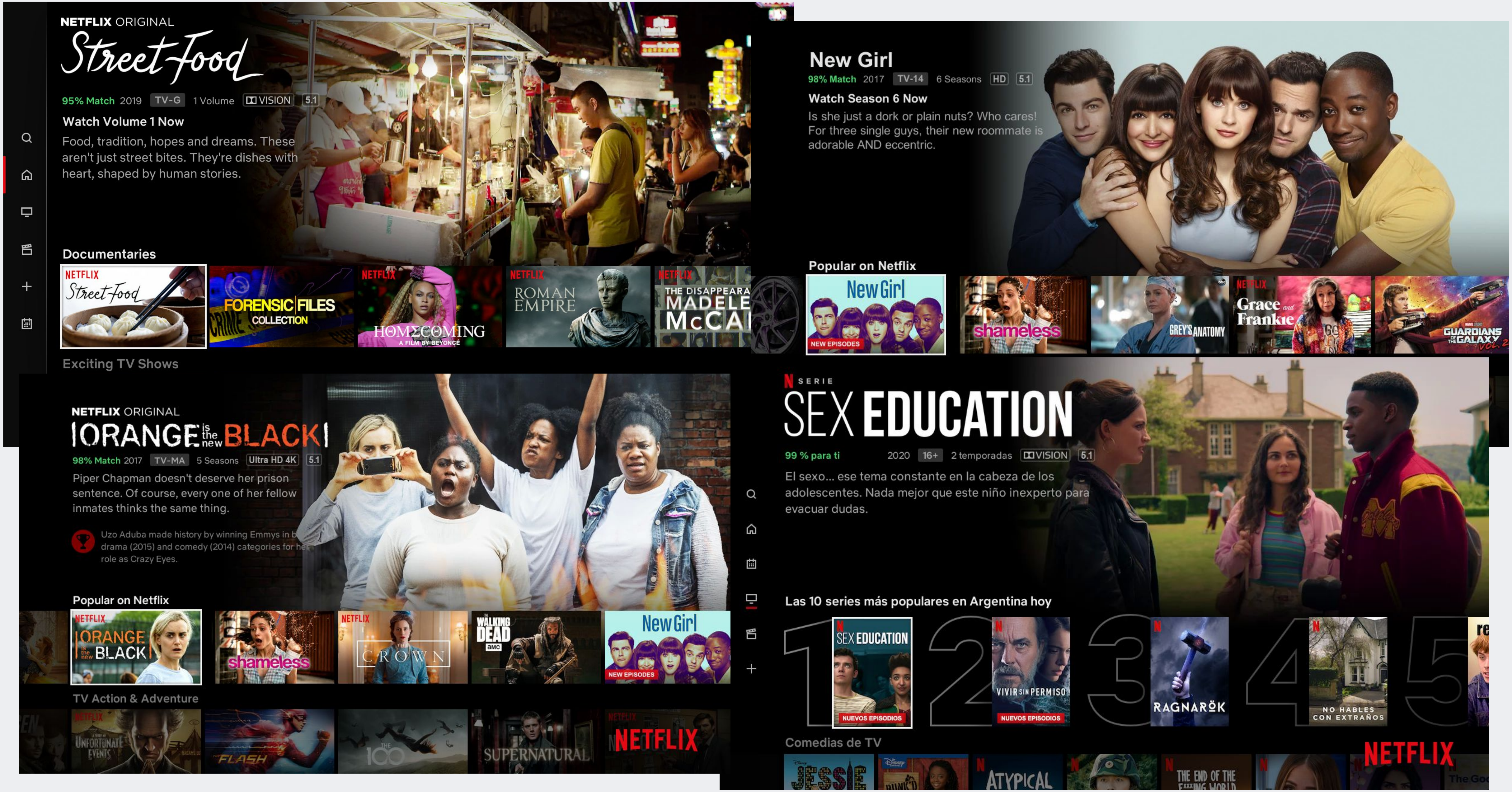
Xinran Waibel
Senior Data Engineer, Netflix

Sundaram Ananthanarayanan
Senior Software Engineer, Netflix

ORGANIZED BY databricks

Agenda

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



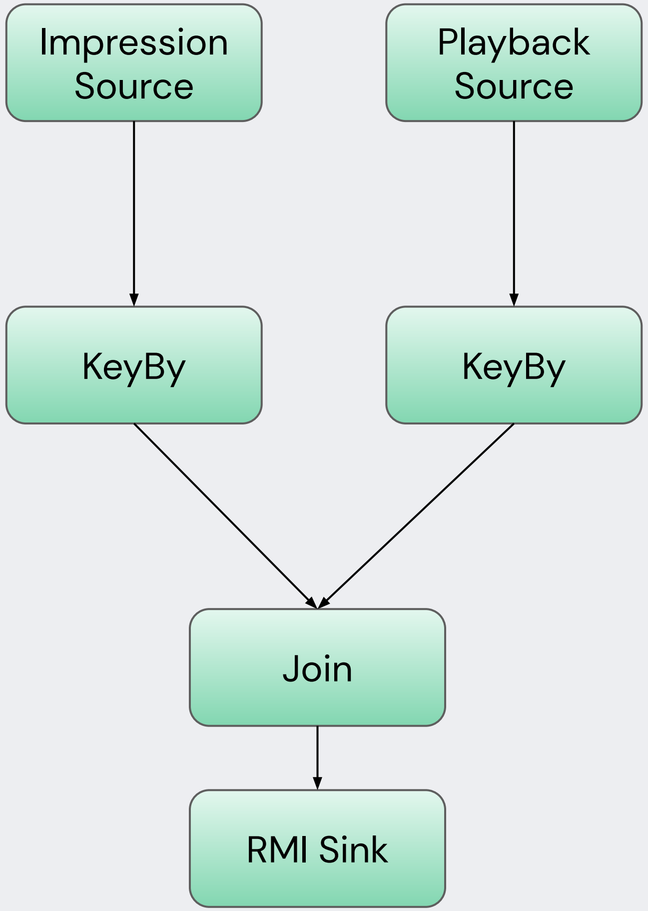
NETFLIX ORIGINAL
Street Food
95% Match 2019 TV-G 1 Volume VISION 5.1
Watch Volume 1 Now
Food, tradition, hopes and dreams. These aren't just street bites. They're dishes with heart, shaped by human stories.
Documentaries
FORENSIC FILES COLLECTION
HOMECOMING
A FILM BY BEYONCÉ
ROMAN EMPIRE
Exciting TV Shows
New Girl
98% Match 2017 TV-14 6 Seasons HD 5.1
Watch Season 6 Now
Is she just a dork or plain nuts? Who cares! For three single guys, their new roommate is adorable AND eccentric.
Popular on Netflix
NEW EPISODES
shameless
GREY'S ANATOMY
Grace and Frankie
GUARDIANS OF THE GALAXY
NETFLIX ORIGINAL
ORANGE is the new BLACK
98% Match 2017 TV-MA 5 Seasons Ultra HD 4K 5.1
Piper Chapman doesn't deserve her prison sentence. Of course, every one of her fellow inmates thinks the same thing.
Popular on Netflix
THE CROWN
THE WALKING DEAD
TV Action & Adventure
FLASH
THE 100
SUPERNATURAL
NETFLIX
SERIE
SEX EDUCATION
99 % para ti 2020 16+ 2 temporadas VISION 5.1
El sexo... ese tema constante en la cabeza de los adolescentes. Nada mejor que este niño inexperto para evacuar dudas.
Las 10 series más populares en Argentina hoy
NUEVOS EPISODIOS
VIVIR SIN PERMISO
RAGNARÖK
NO HABLES CON EXTRAÑOS
Comedias de TV
JESSIE
ATYPICAL
NETFLIX

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.

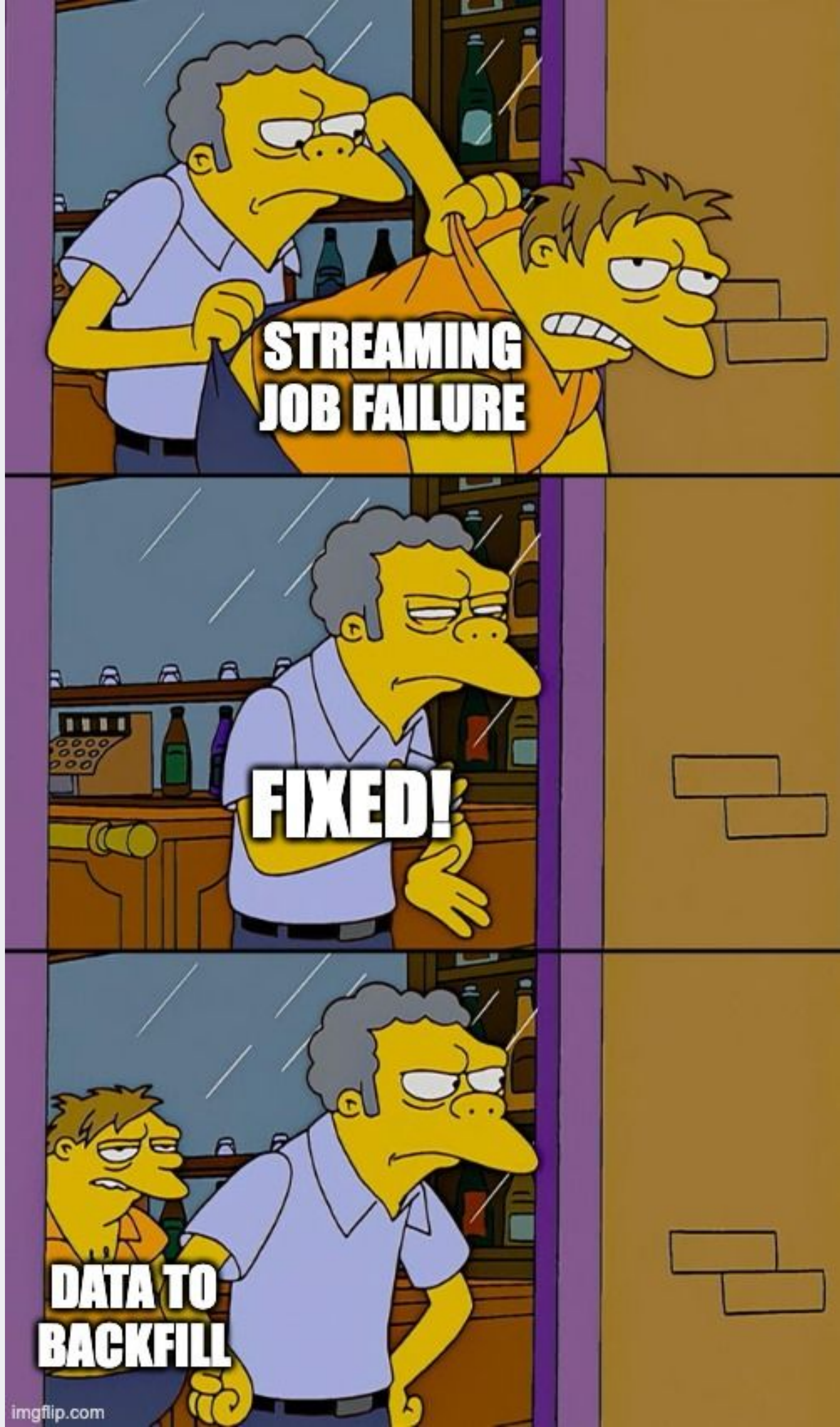


Event streaming operations

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

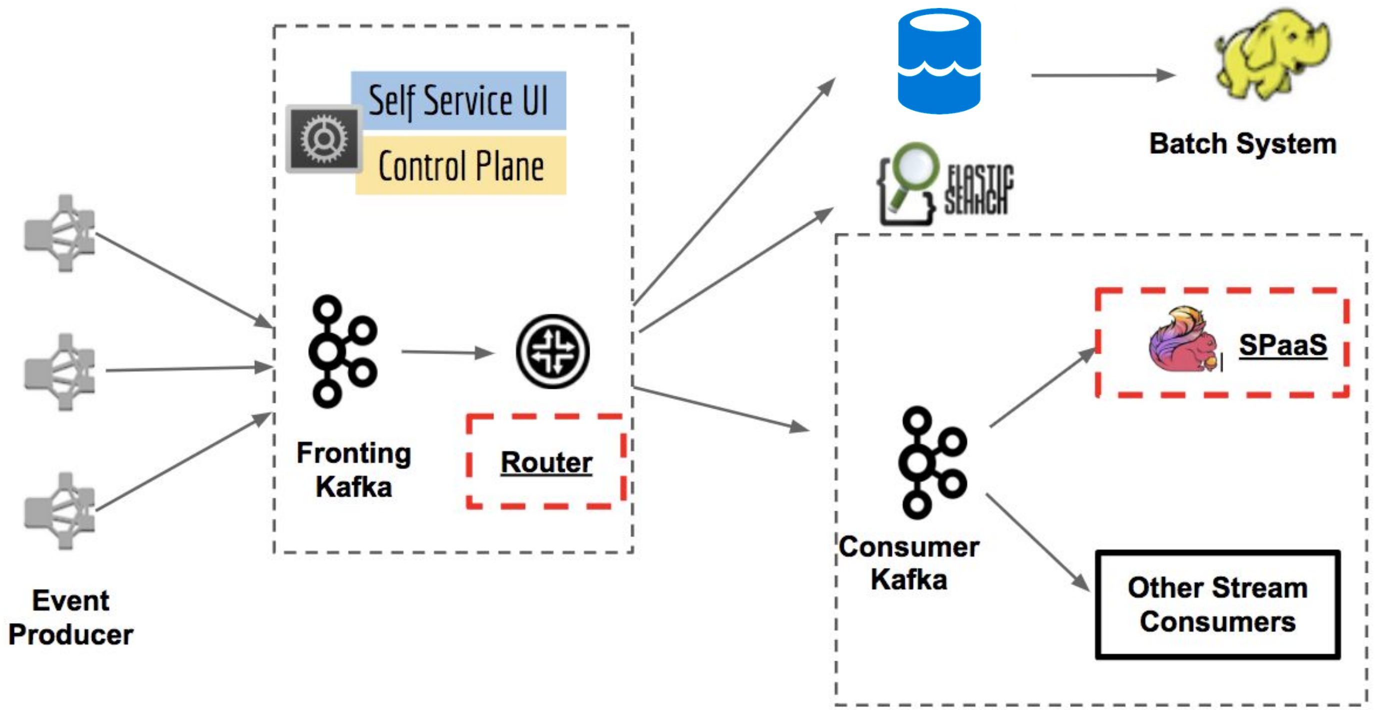
😭 Troubleshooting can take hours or days and source data can expire.

😭 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.



[1] https://netflixtechblog.com/keystone-real-time-stream-processing-platform-a3ee651812a

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.

[1] https://databricks.com/discover/data-lakes/introduction

Kafka events stored in an Iceberg table

Playback Kafka Events

```
{
        "account_id":98524989,
        "show_id":4236781,
        "view_duration_sec": 123,
        …
},
{
        "account_id":87934298,
        "show_title_id":8754782,
        "view_duration_sec": 45,
        …
},
{
        "account_id":79403754,
        "show_id":3648295,
        "view_duration_sec": 81,
        …
},
…
```

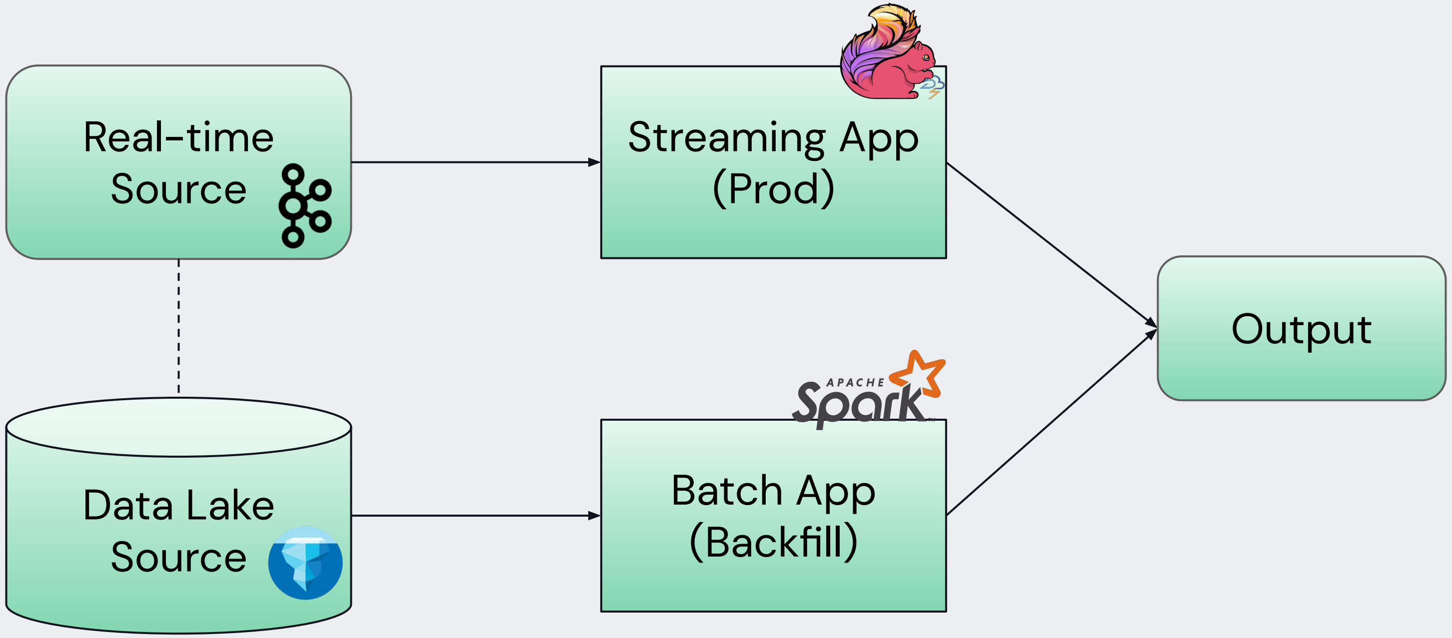
Playback Iceberg Table

account_id	show_id	view_duration	__metadata__
98524989	4236781	123	{kafka_ingestion_ts: …}
87934298	8754782	45	{kafka_ingestion_ts: …}
79403754	3648295	81	{kafka_ingestion_ts: …}
…	…	…	…

🤔 *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/ [2] https://beam.apache.org/documentation/runners/capability-matrix/

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. 💔💔💔

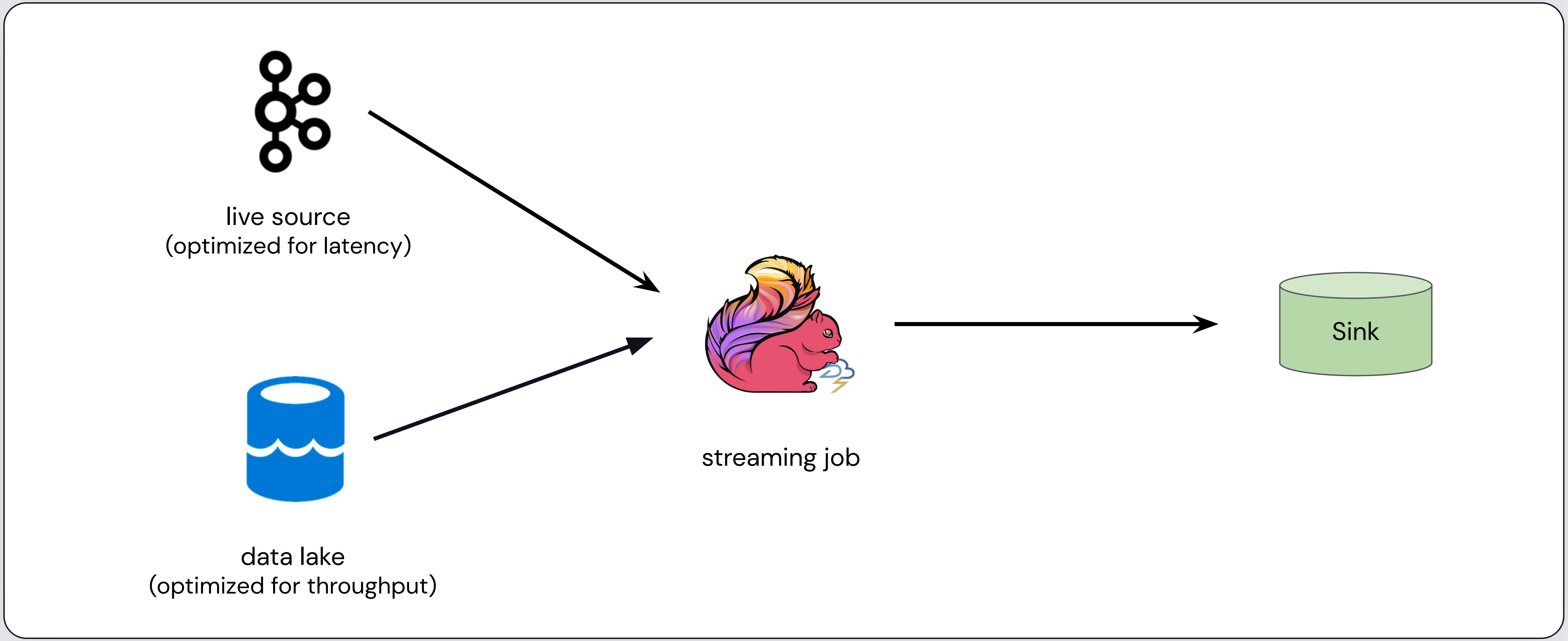
Can we combine the best things from all three worlds?

Backfilling
In Kappa Architecture
(feat. Data Lake)

Backfilling using Data Lake: Goals

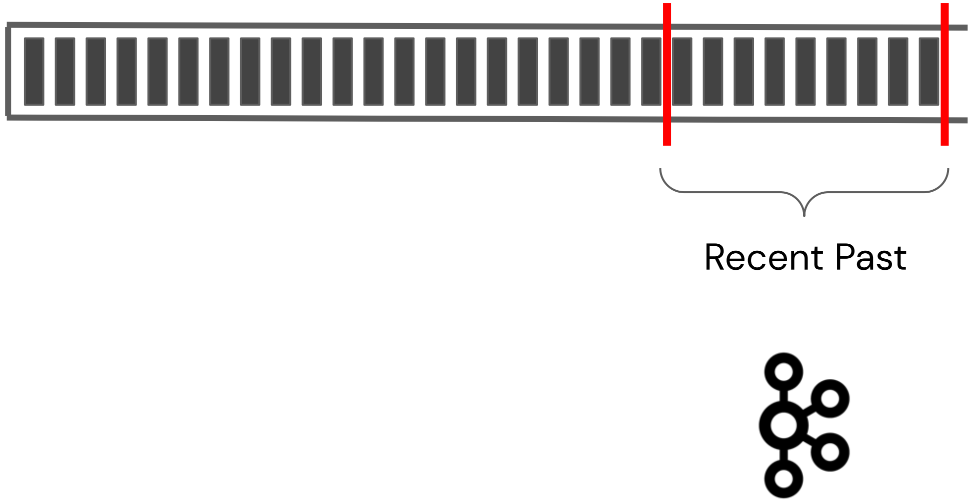
- Provide a generic solution that works for all classes of applications
- Minimal code changes to add support
- Scales horizontally to backfill quickly

Backfilling using Data Lake: Overview



Backfilling using Data Lake: Overview

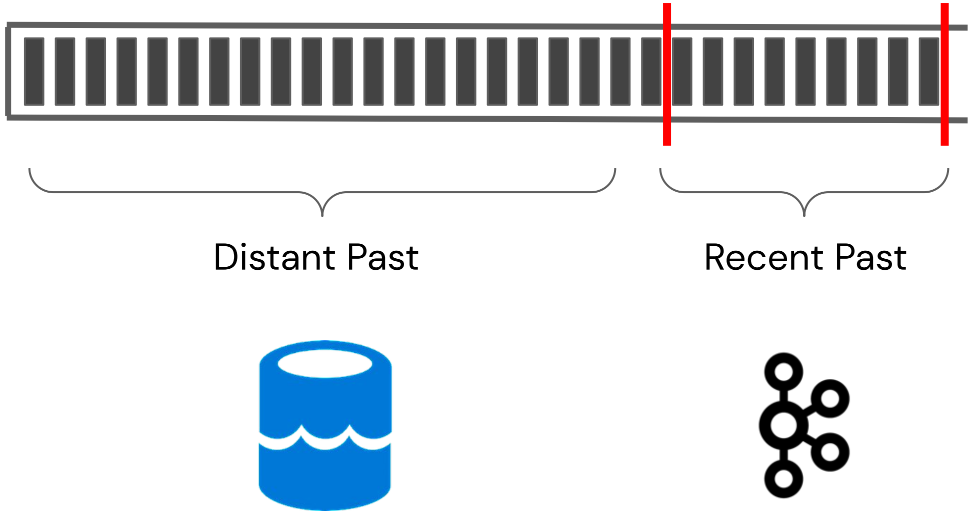
Semantics



Handling real-time data

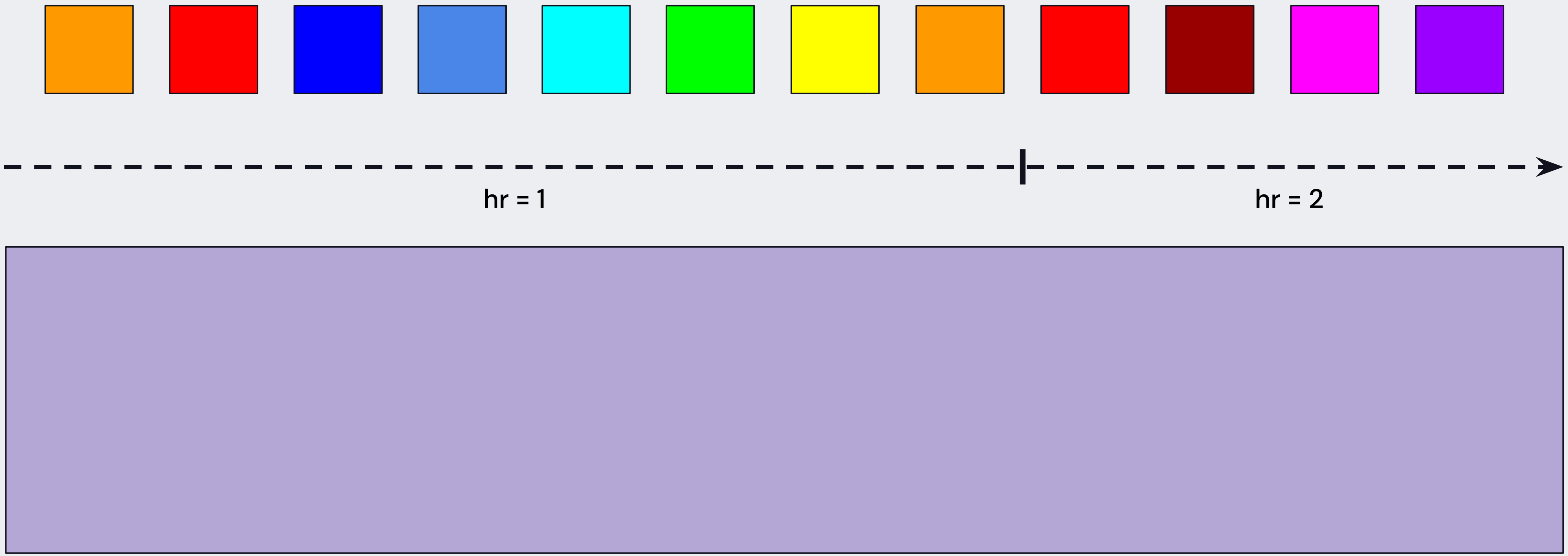
Backfilling using Data Lake: Overview

Semantics

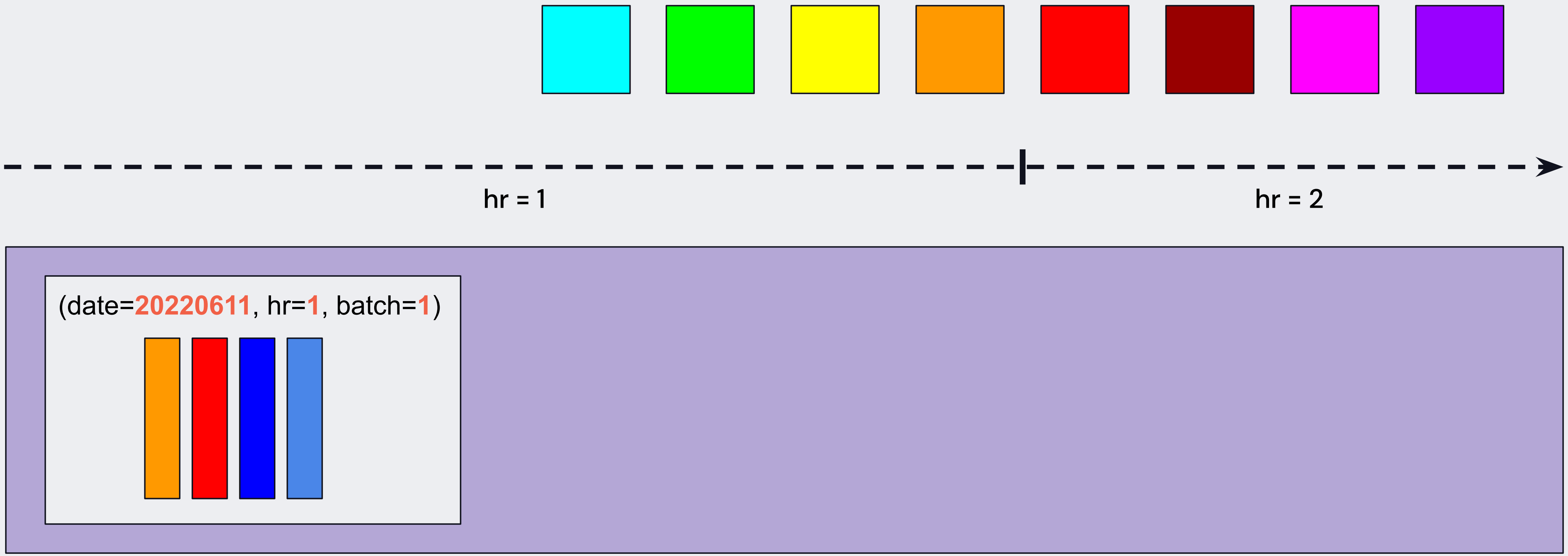


Backfilling

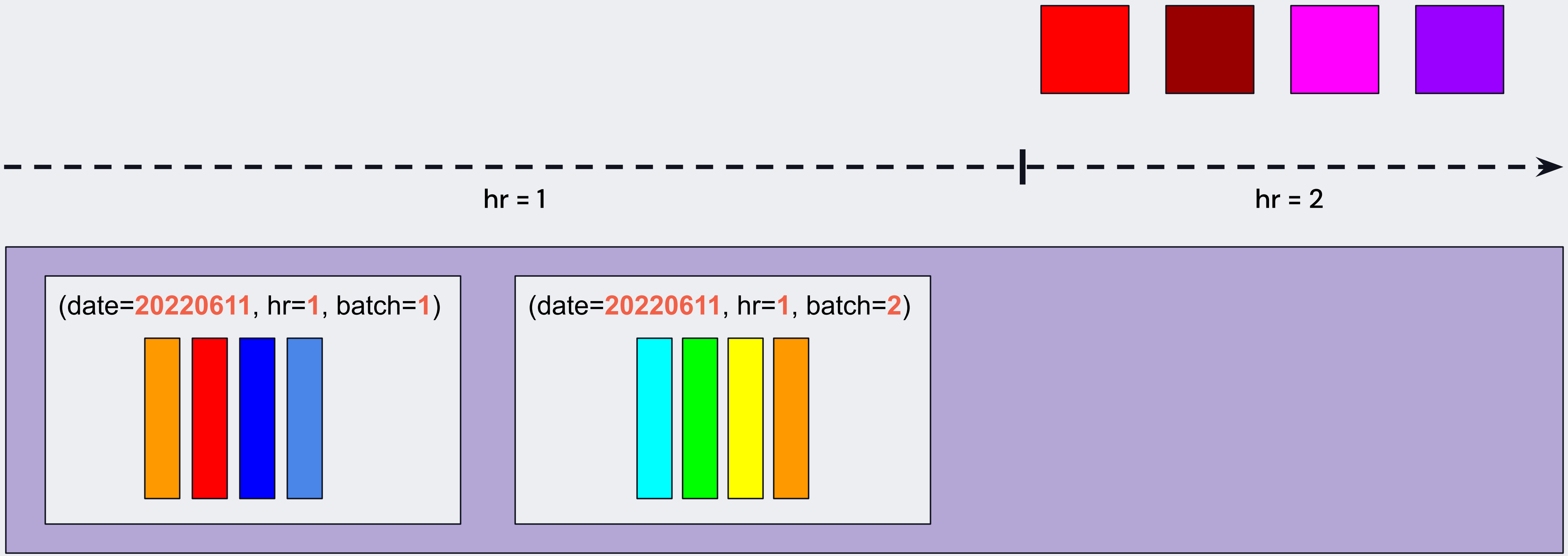
Ingesting streaming data into data lake



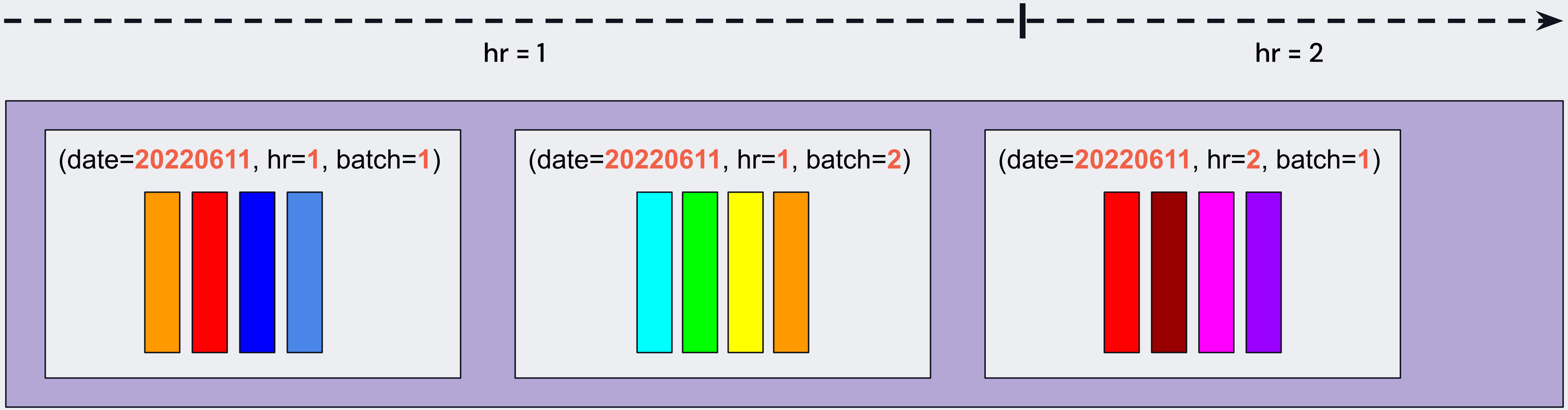
Ingesting streaming data into data lake



Ingesting streaming data into data lake



Ingesting streaming data into data lake

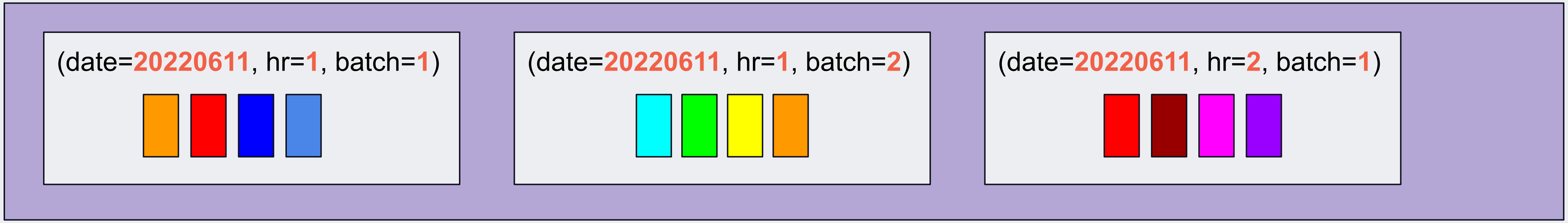


✔ Batching events results in good compression ratios.
✔ Avoids small file problem.

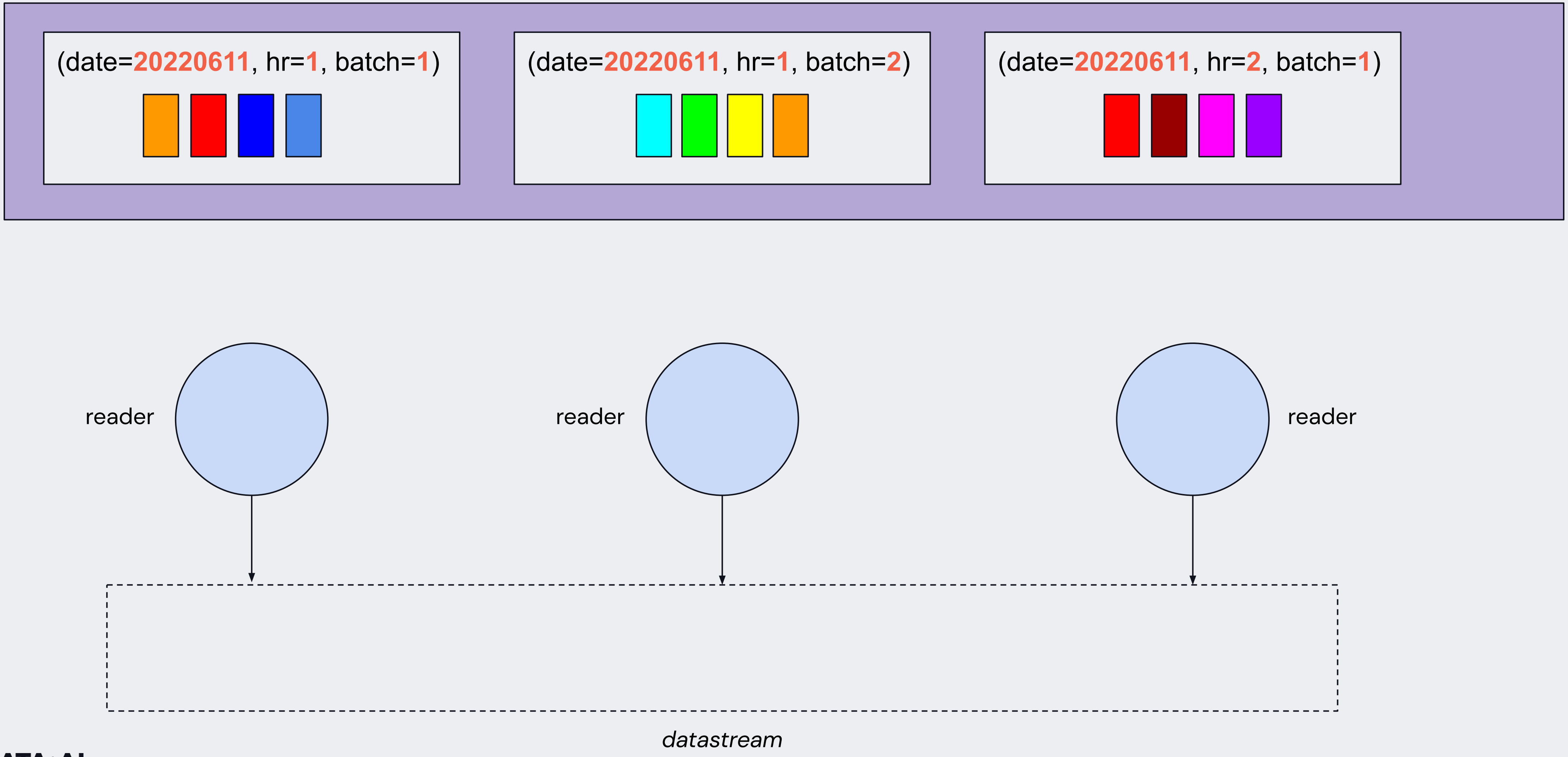
How to backfill?

- **Strawman 1:** Read events from files filtered by backfill dates

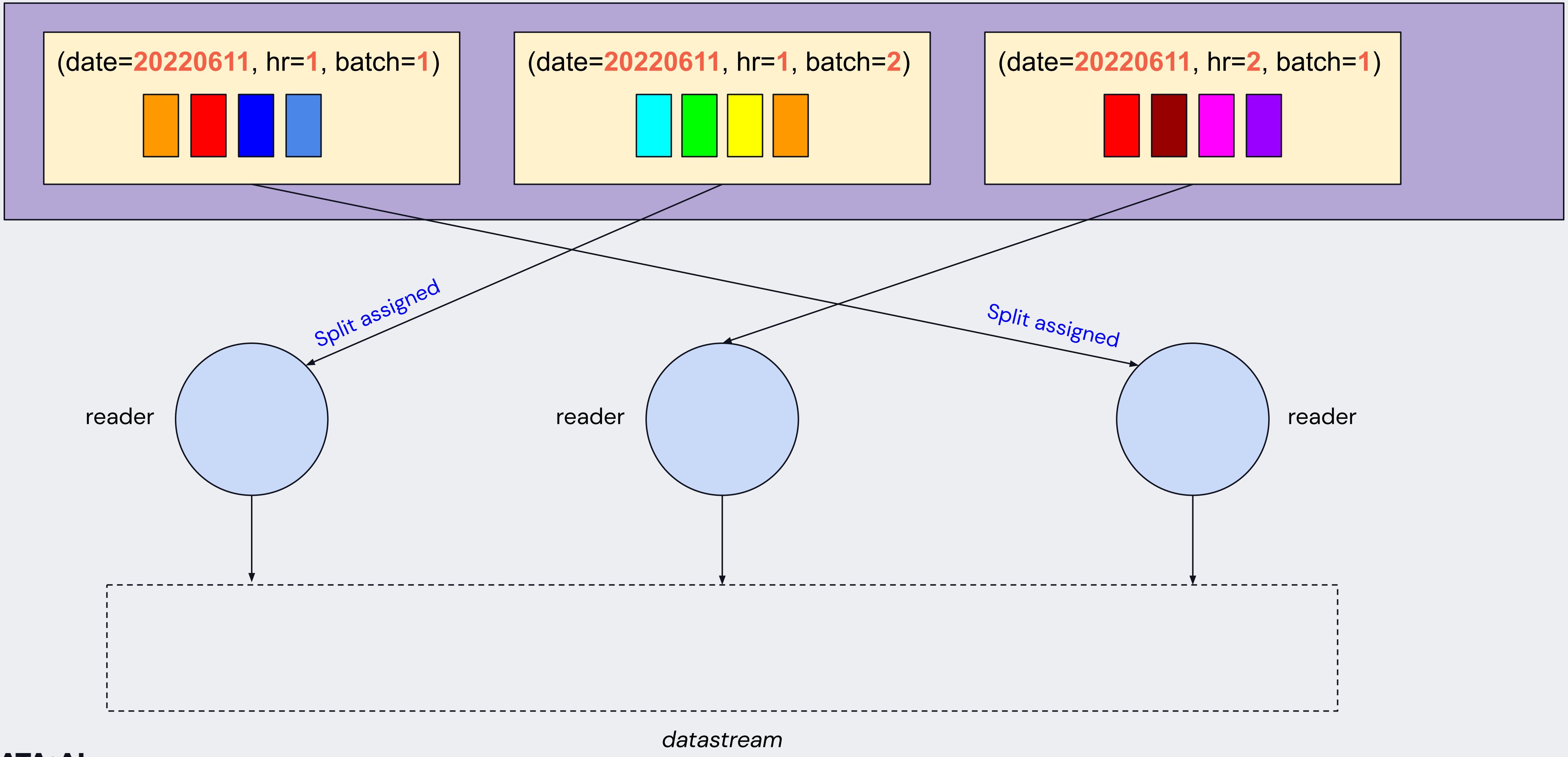
Strawman 1: Read events from selected files



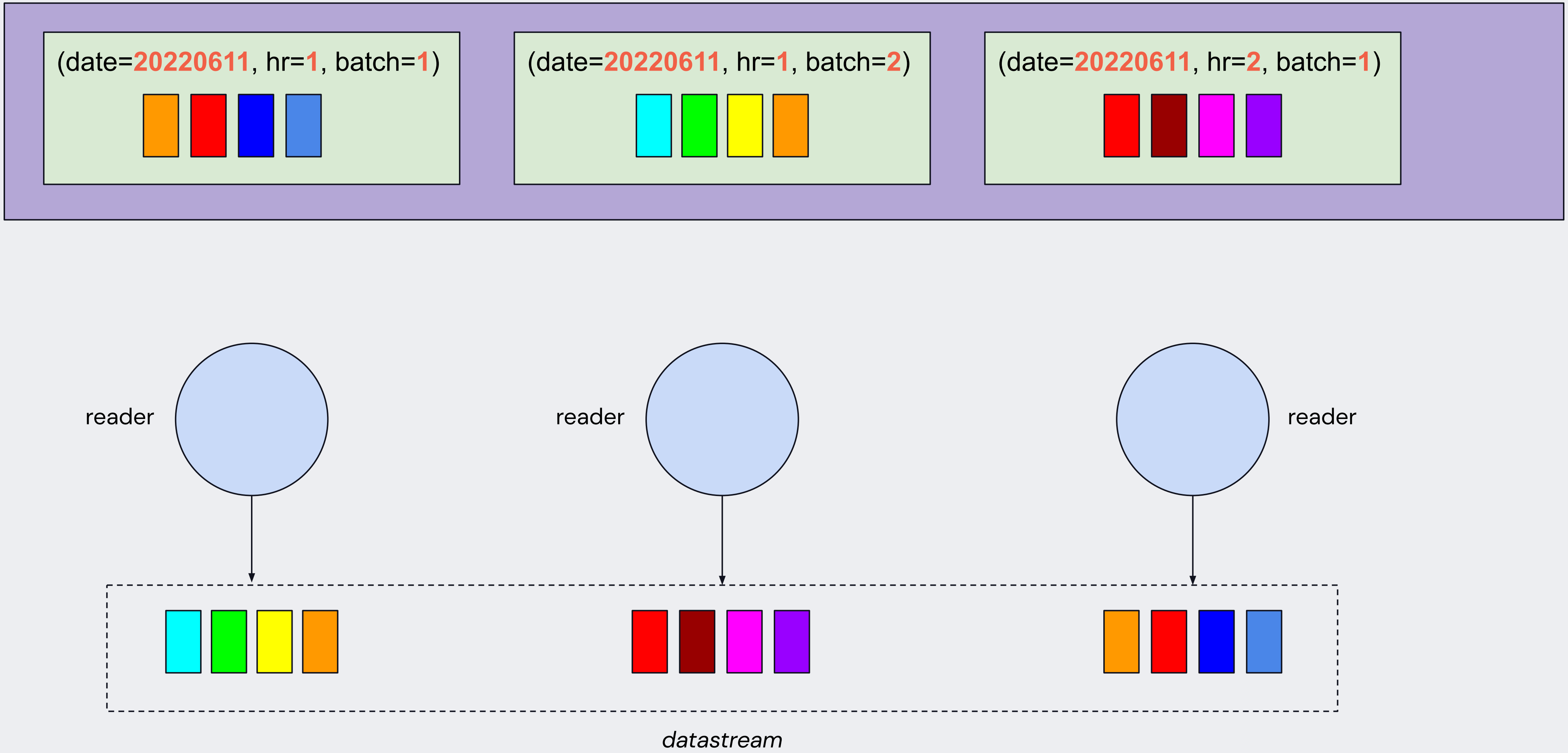
Strawman 1: Read events from selected files



Strawman 1: Read events from selected files



Strawman 1: Read events from selected files

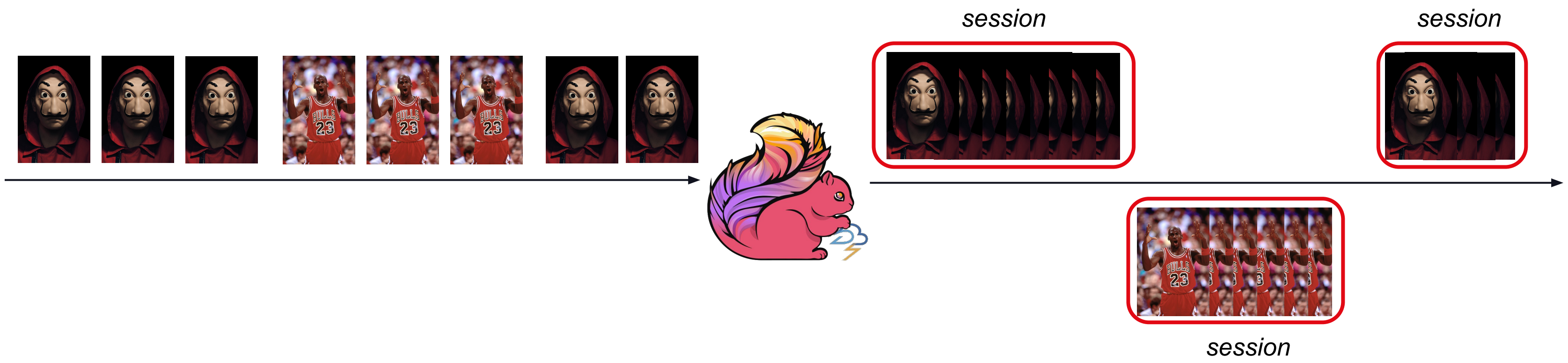


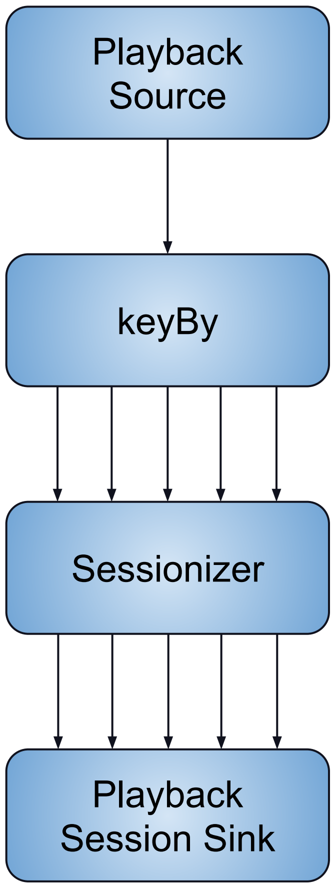
How to backfill?

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

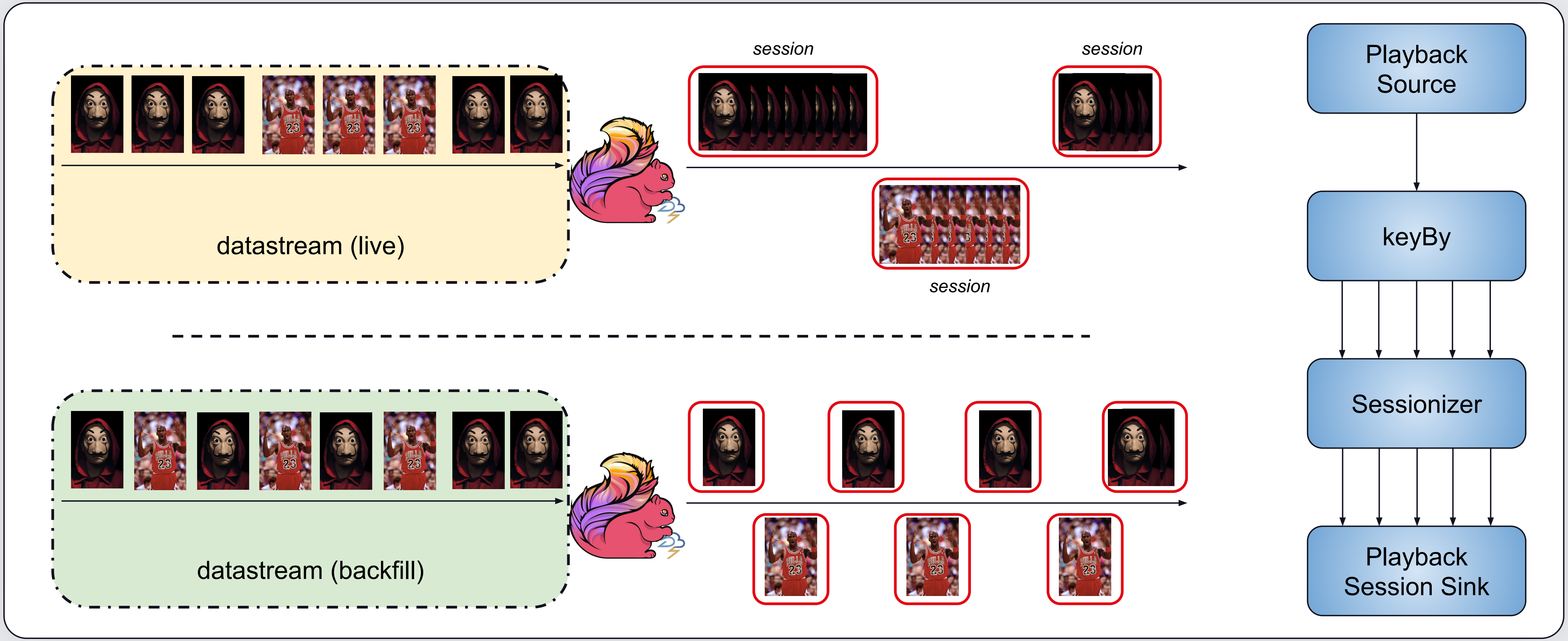
Challenge #1: Applications assume ordering

Example: Application that converts playback events into playback sessions

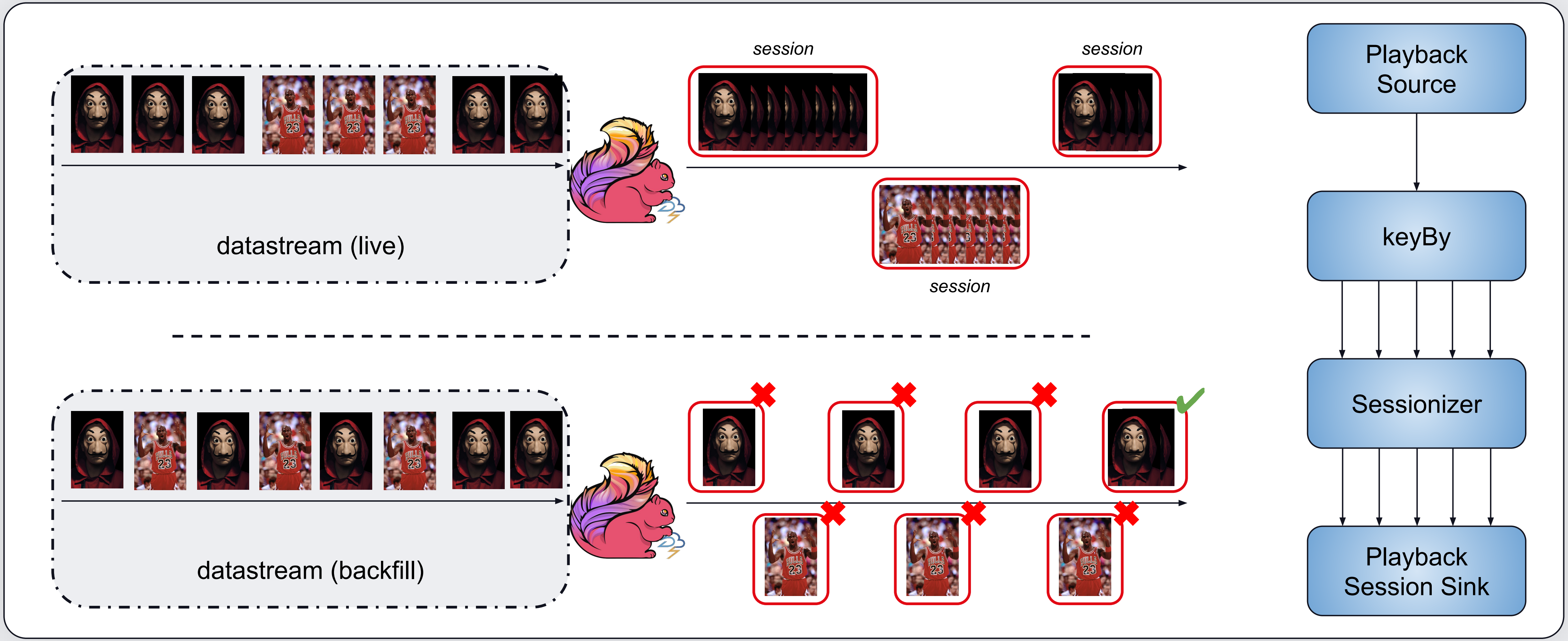




Challenge #1: Applications assume ordering

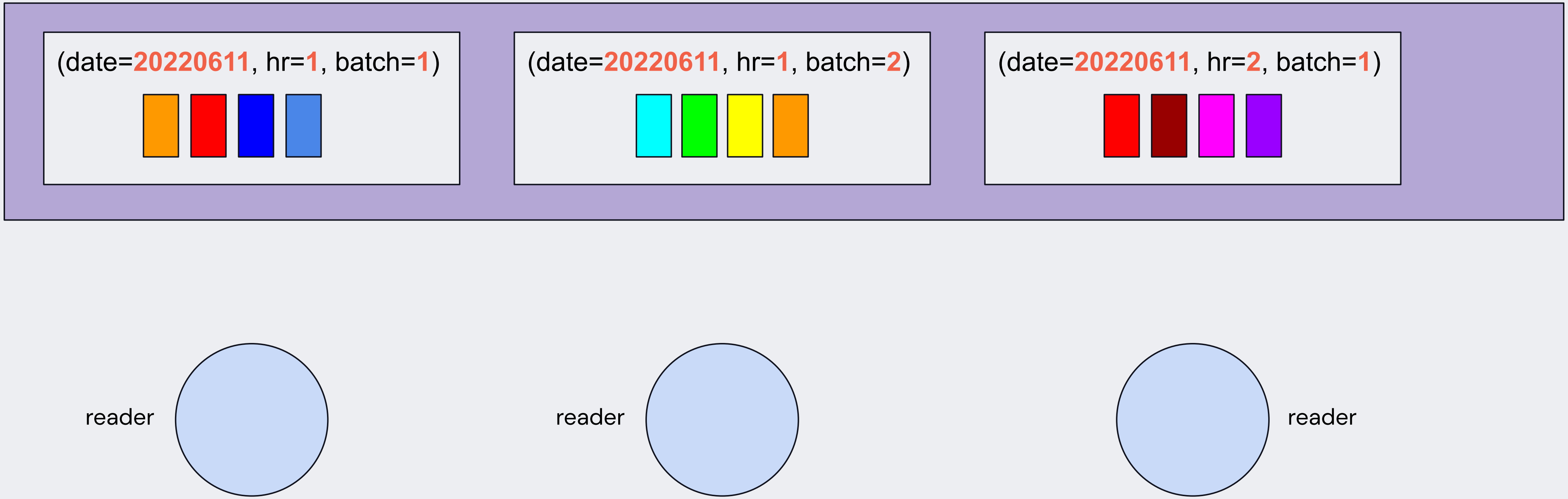


Challenge #1: Applications assume ordering

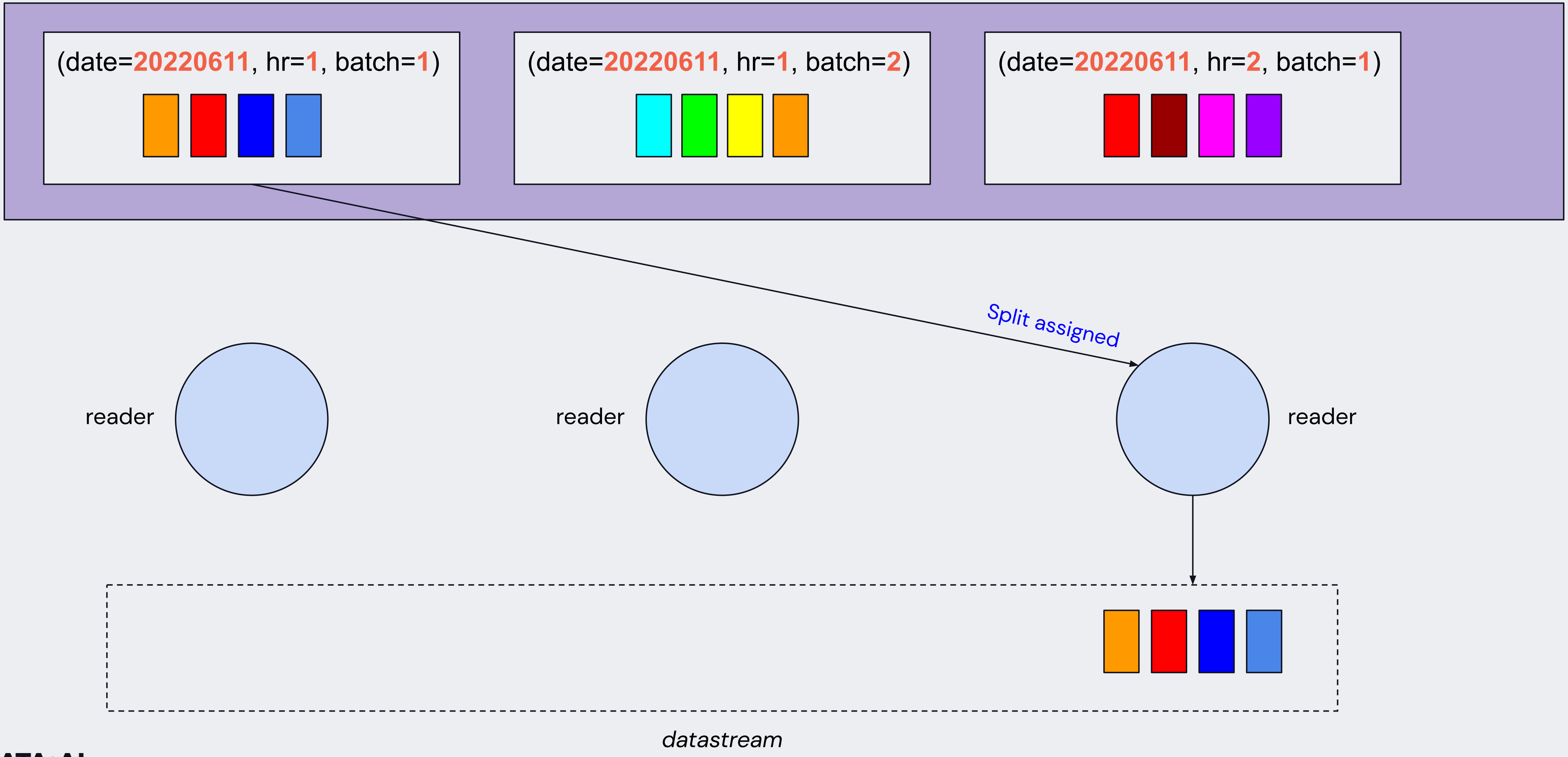


😭Different ordering leads to different results

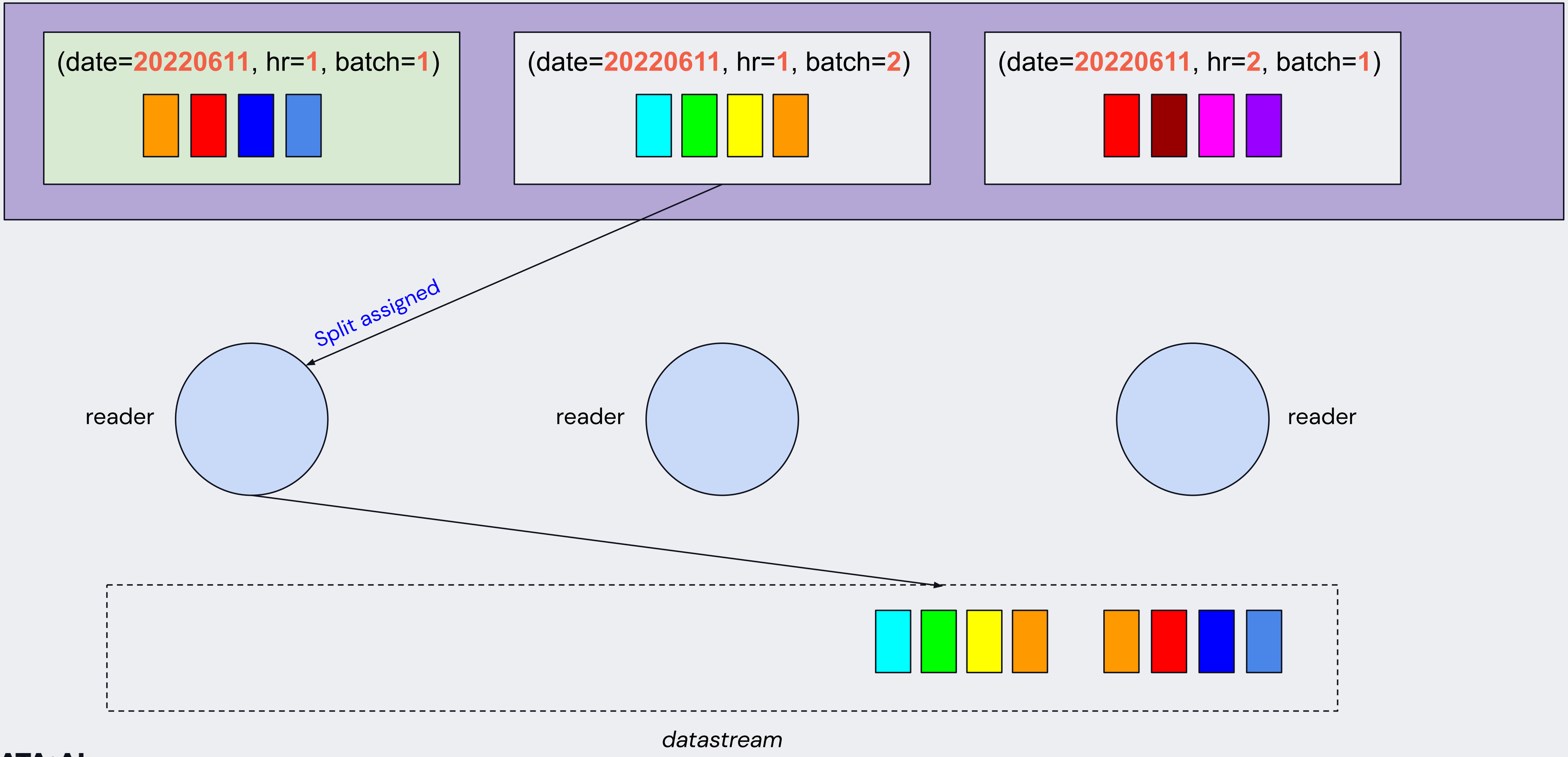
Strawman 2: Order all files and read in order



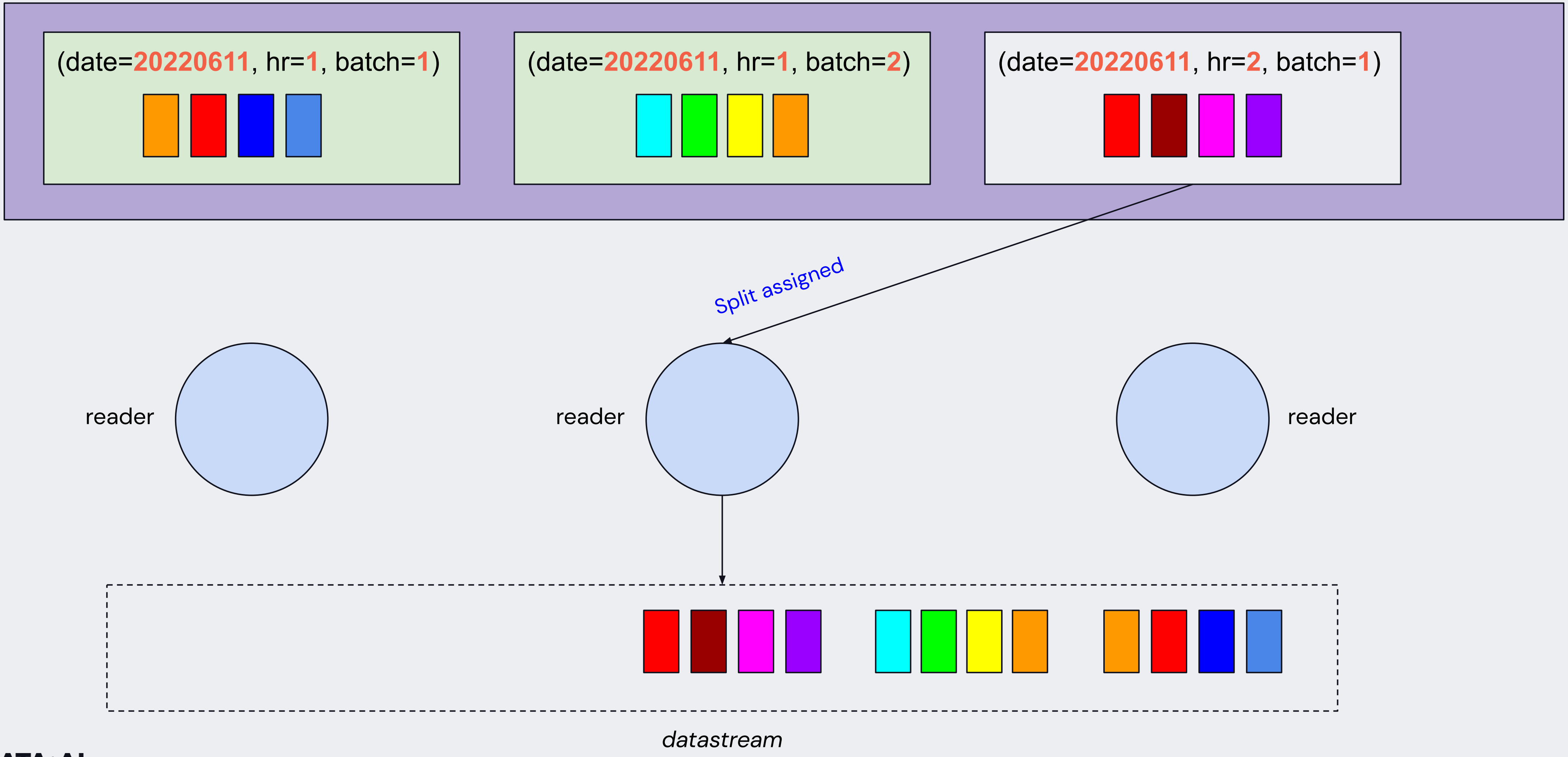
Strawman 2: Order all files and read in order



Strawman 2: Order all files and read in order



Strawman 2: Order all files and read in order



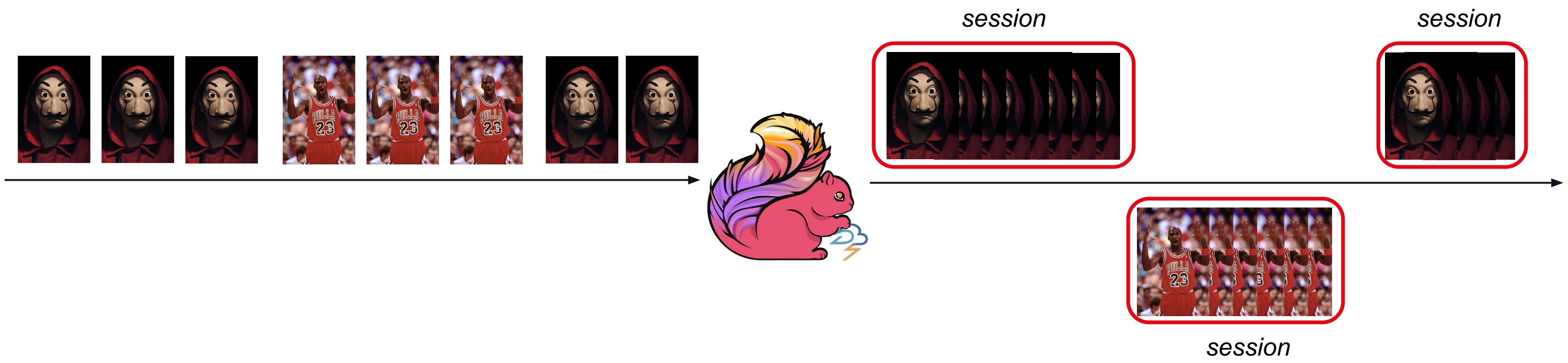
How to backfill?

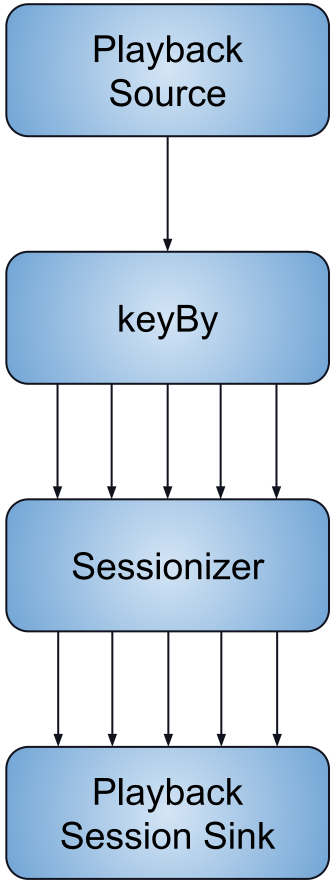
- **Strawman 1:** Read events from files filtered by backfill dates
  - ✔ Scales horizontally to backfill quickly
  - ✖ Does not work for all types of applications
- **Strawman 2:** Order all files and read them in order
  - ✔ Guarantees similar ordering semantics as the live traffic
  - ✖ Does not scale horizontally

But, not all streaming apps rely on strong ordering guarantees.

Event-Time Semantics

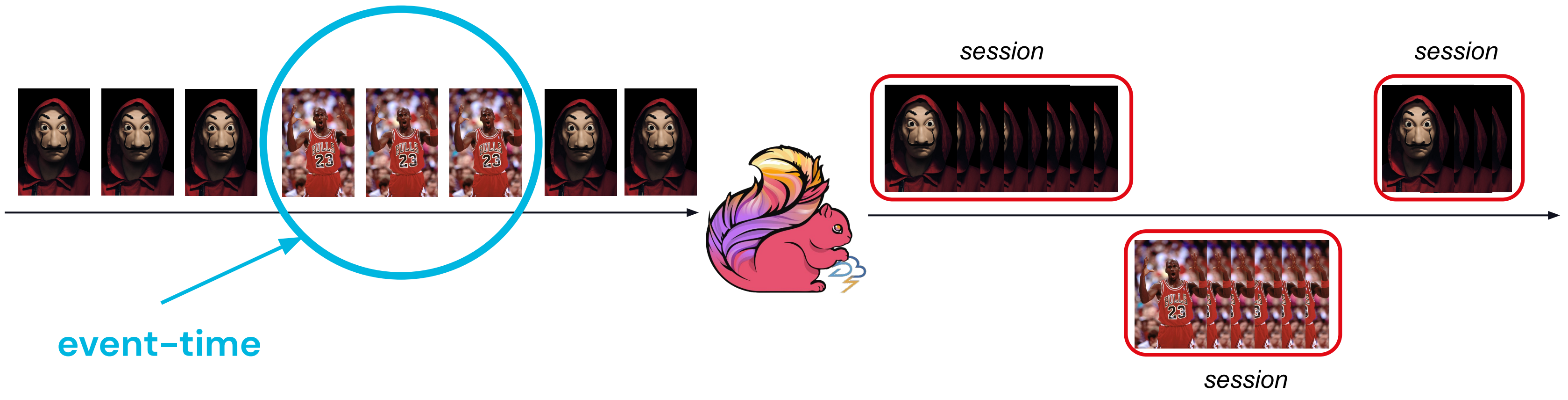
Example: Application that converts playback events into playback sessions

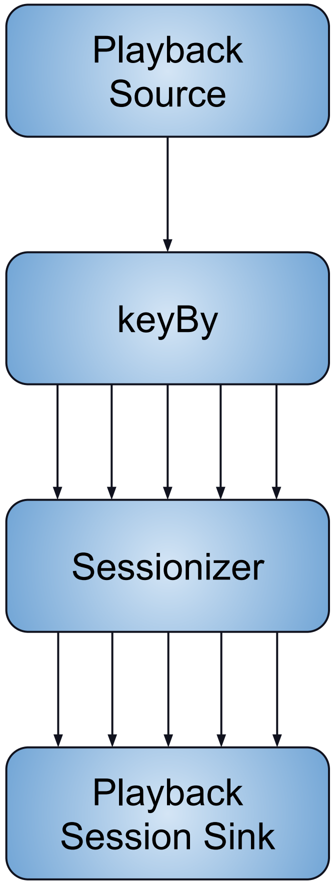




Event-Time Semantics

Example: Application that converts playback events into playback sessions

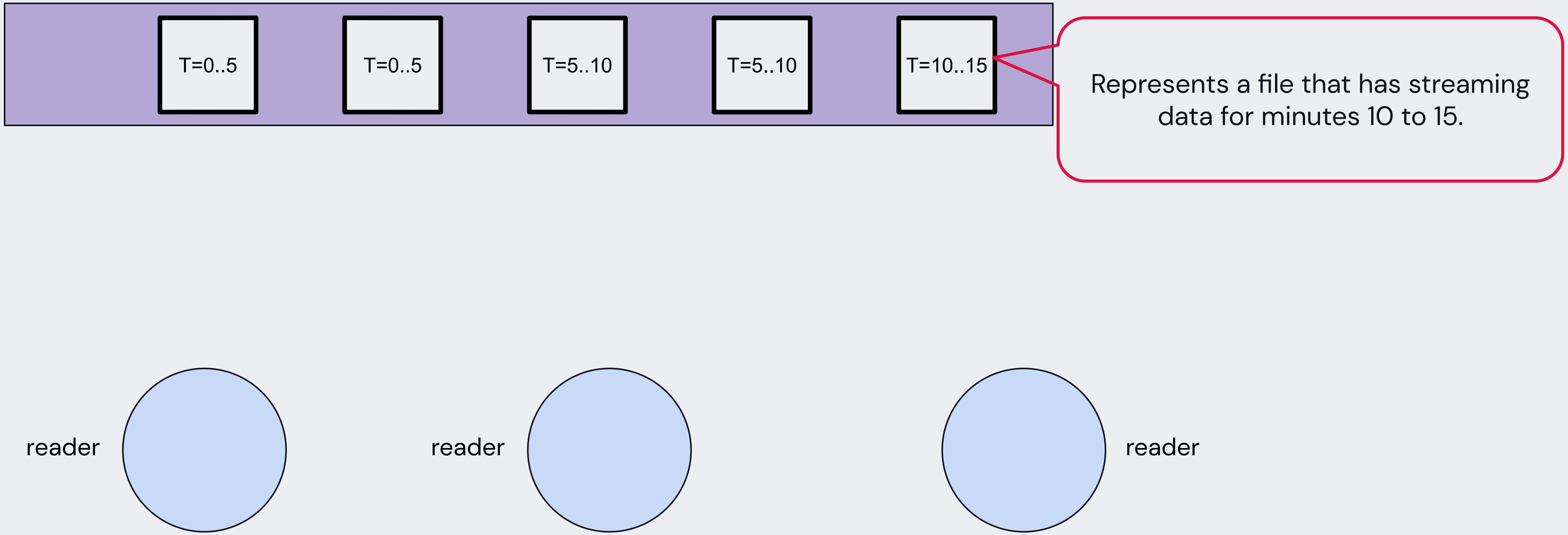




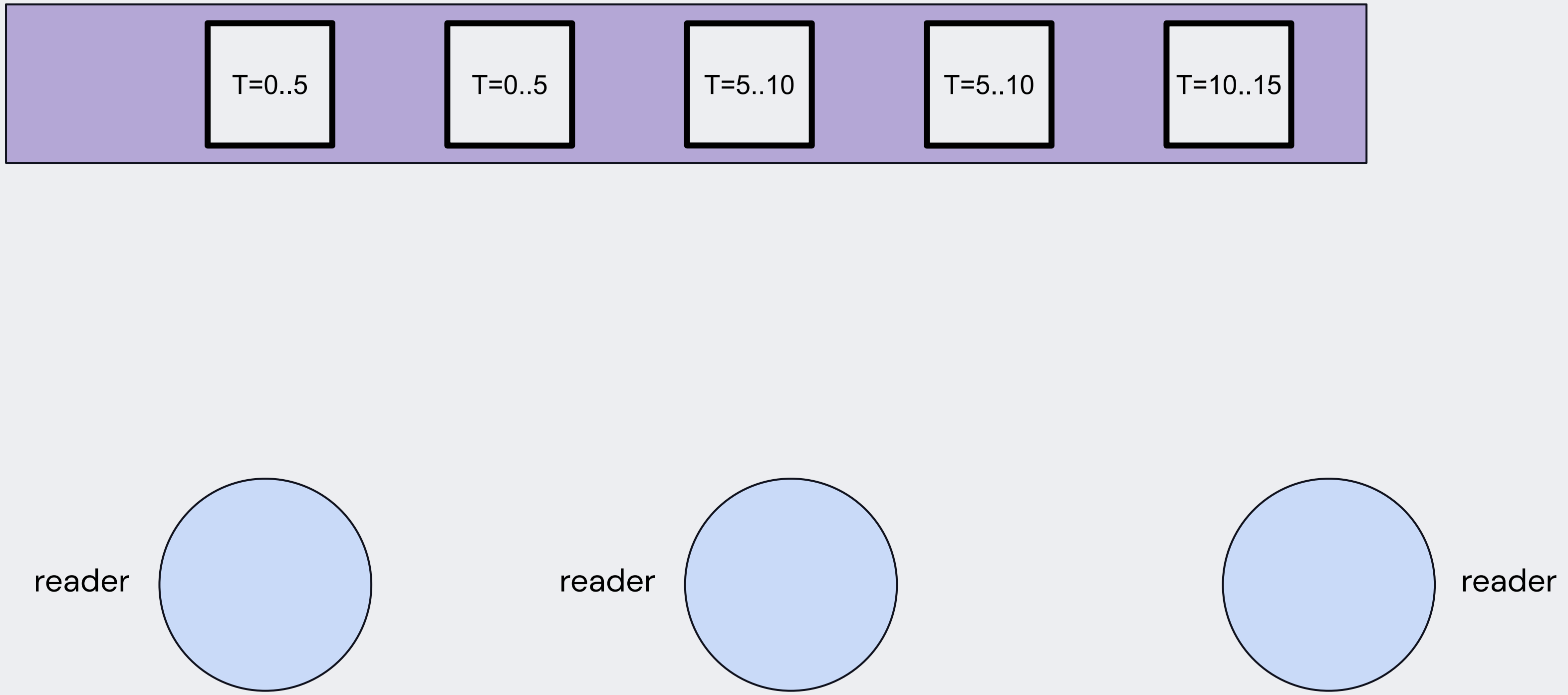
1. Sessions are derived from event timestamps - not ingestion times.
2. Because events can arrive late, applications tolerate lateness.

Idea: Use lateness tolerated by app

Solution: Use lateness tolerated by app

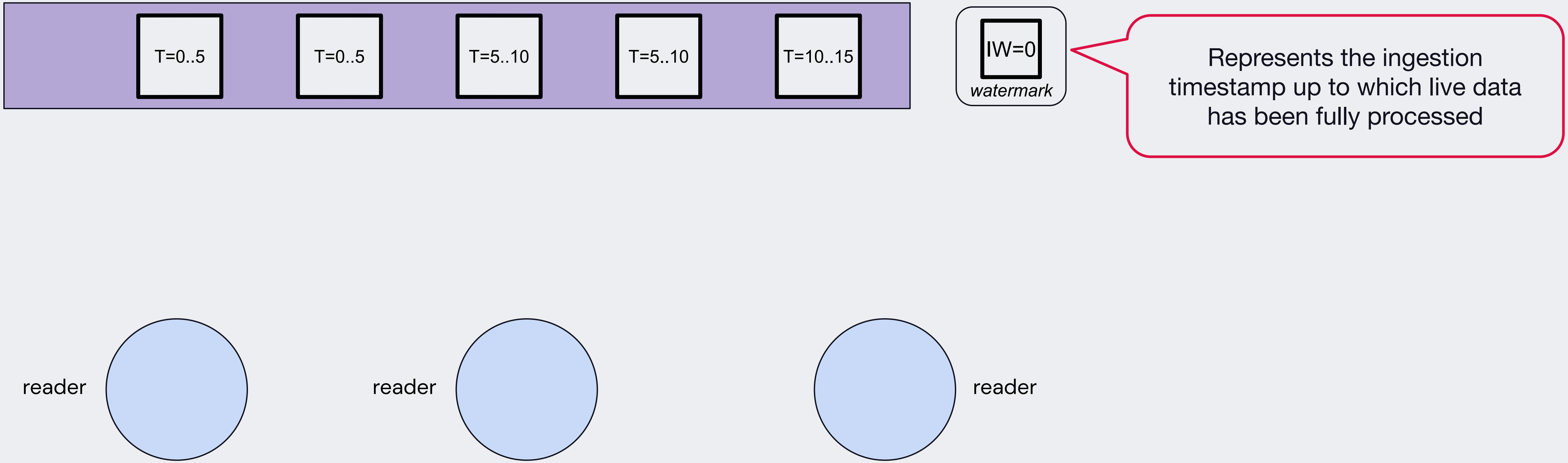


Solution: Use lateness tolerated by app

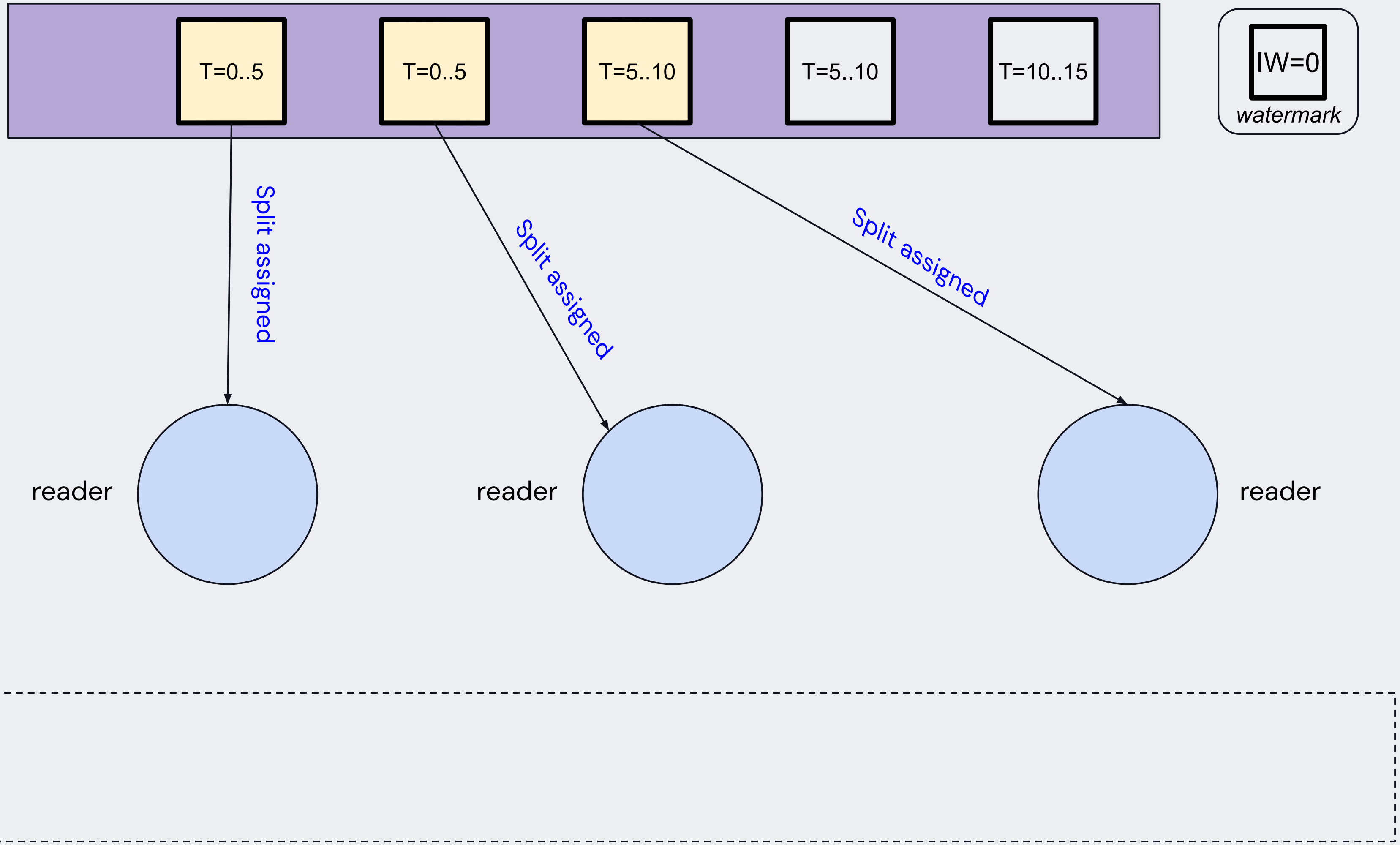


Assuming lateness of "10" minutes is okay.

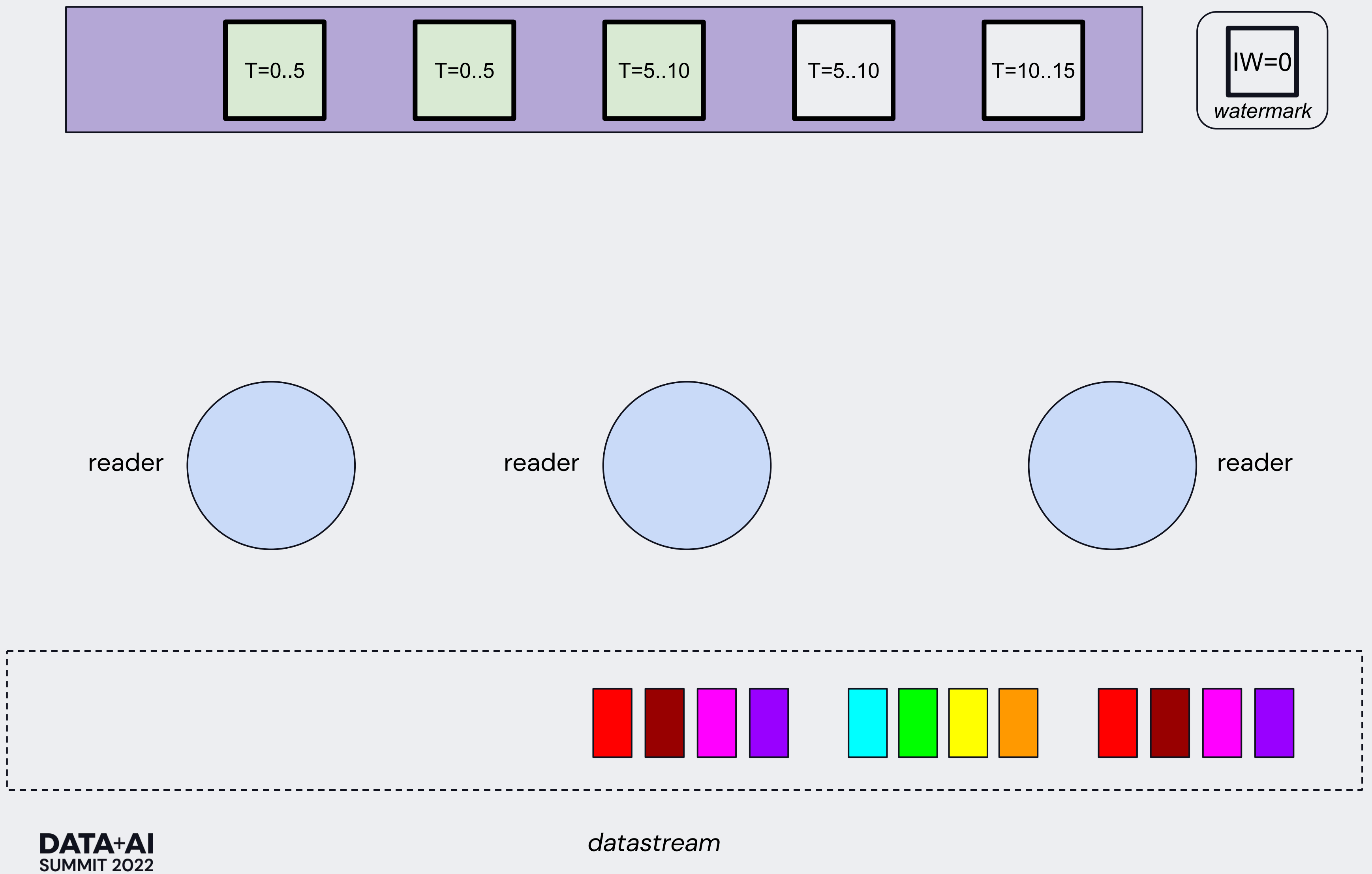
Solution: Use lateness tolerated by app



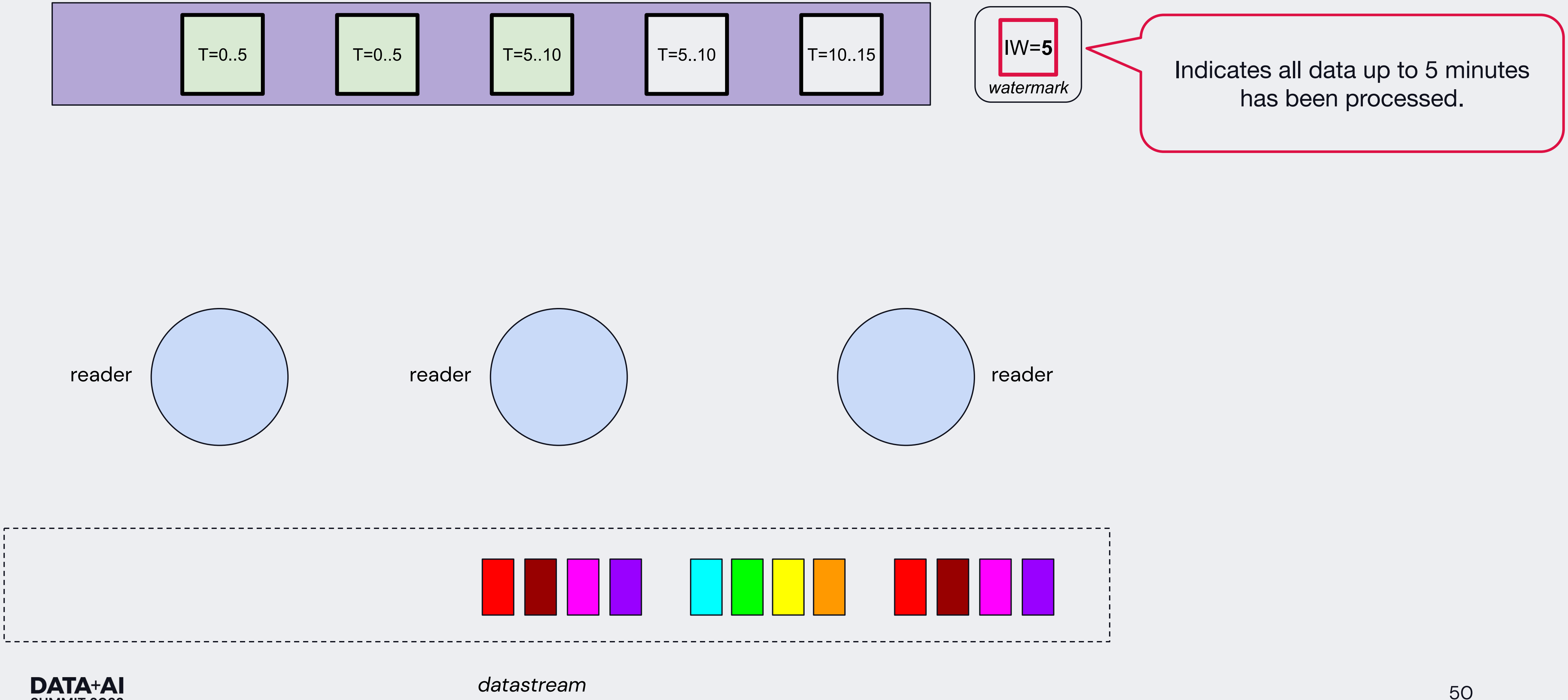
Solution: Use lateness tolerated by app



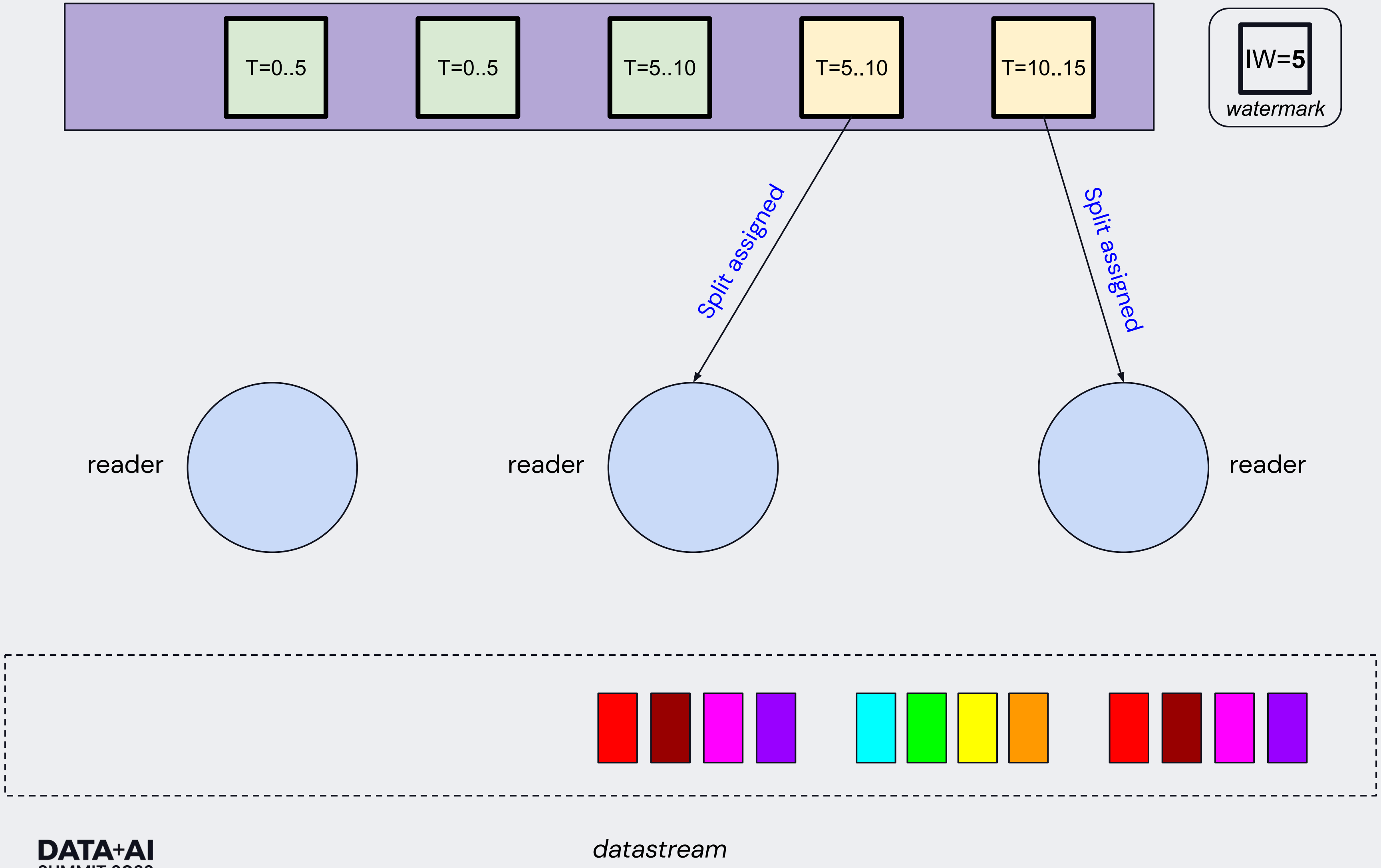
Solution: Use lateness tolerated by app



Solution: Use lateness tolerated by app



Solution: Use lateness tolerated by app



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

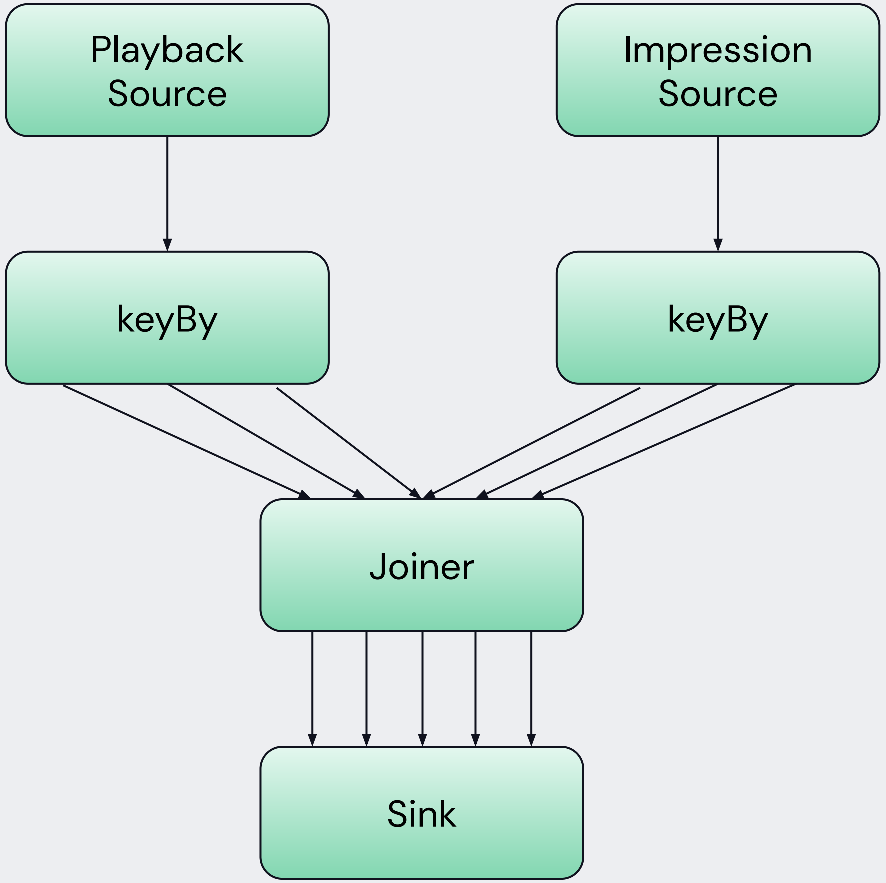
Messaging System's Ordering Guarantees

- Kafka provides strict ordering of events within a partition.

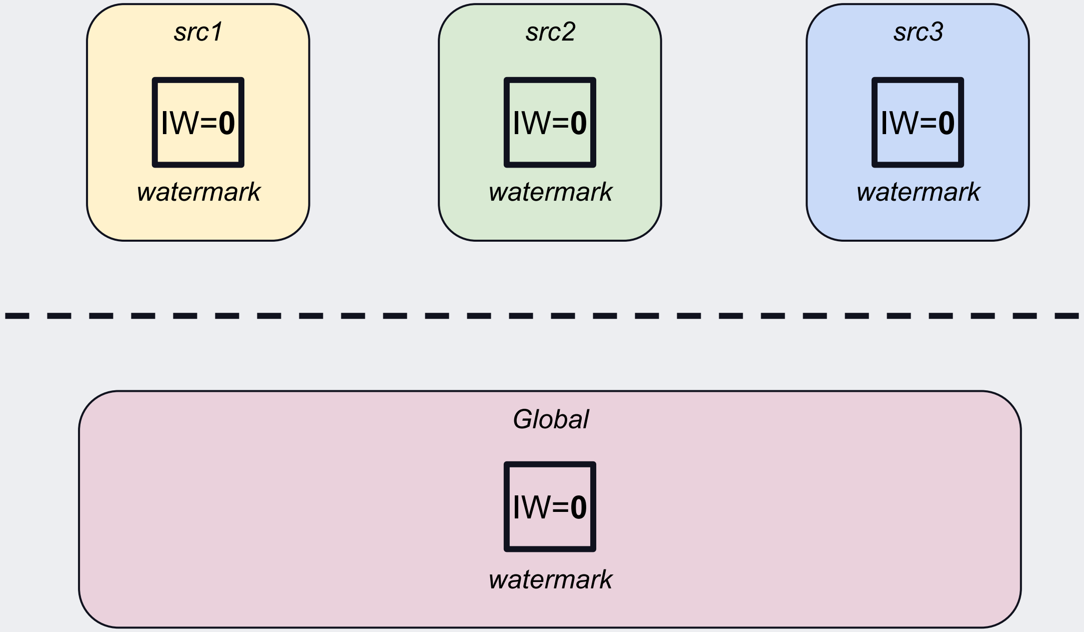
- Most analytical use-cases (streaming-joins, sessionization) use event-time semantics and do not require such stronger guarantees.

Challenge 2: Reading Multiple Sources

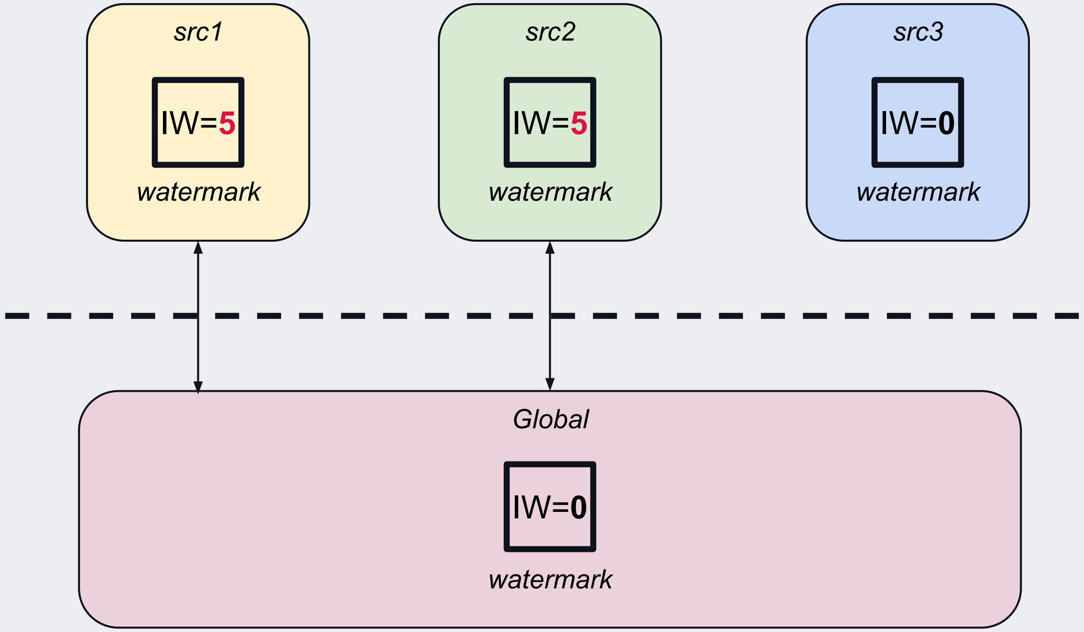
- One source can have significantly way more data than the other.
- During backfill, this could lead to a watermark skew resulting in state size explosion.
- This can eventually lead to slow checkpoints or checkpoint timeouts.



Solution: Coordinate watermarks

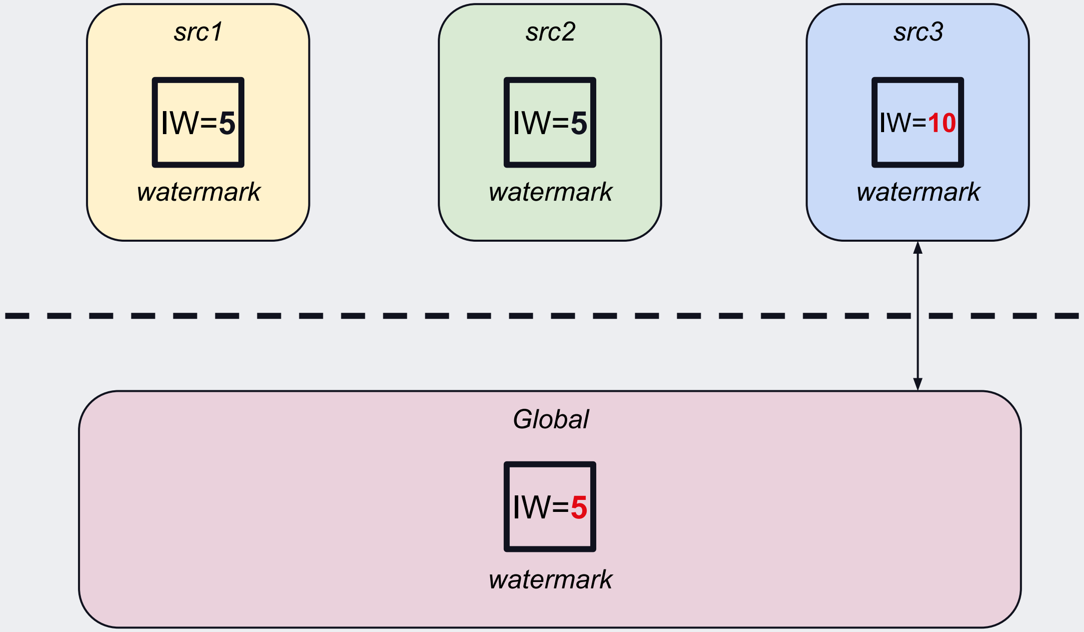


Solution: Coordinate watermarks



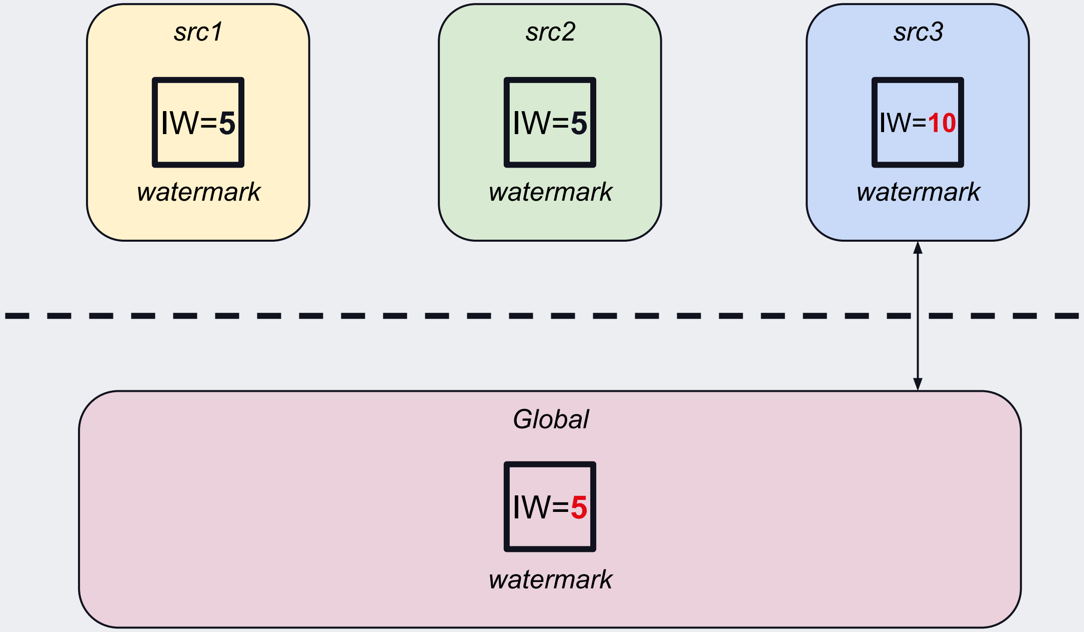
Communicate watermark updates to the global tracker.

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.

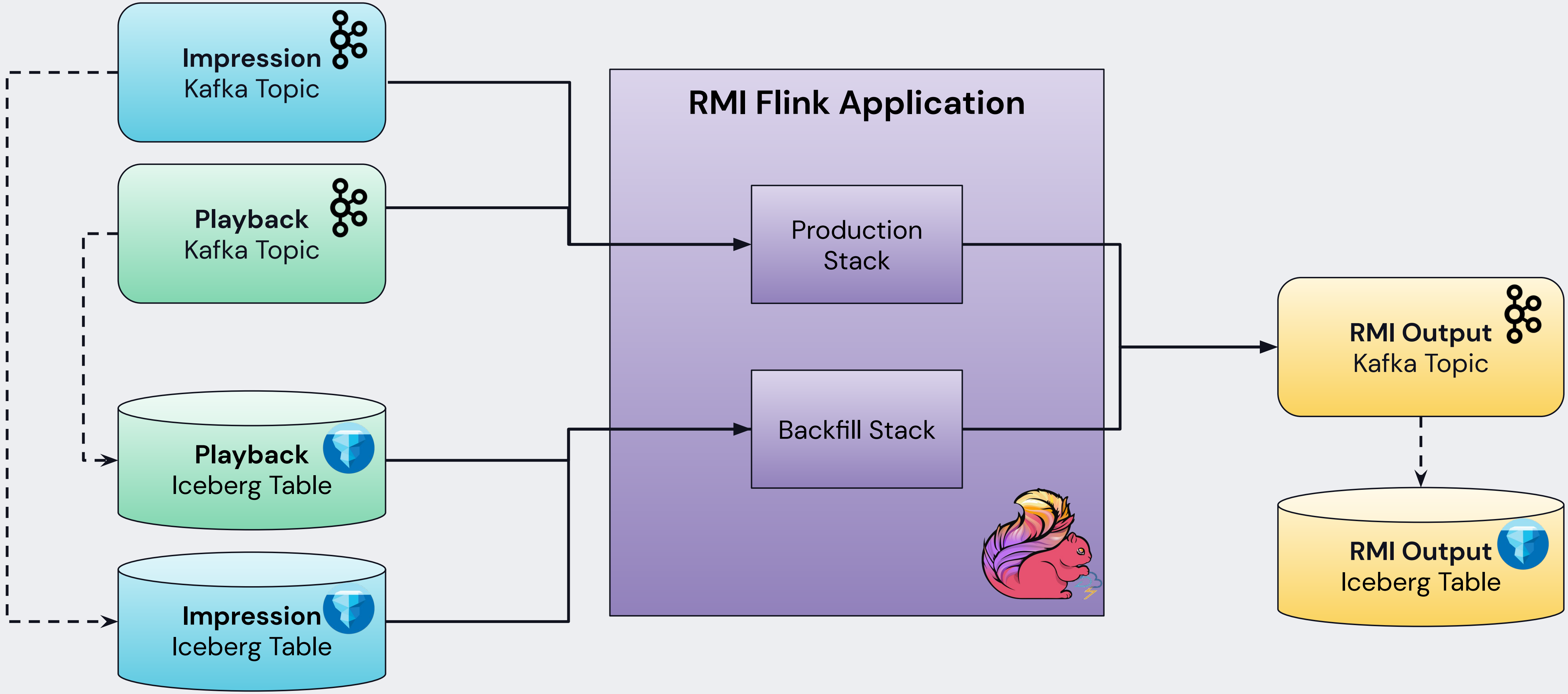
How to backfill?

- **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



Adopting Kappa Backfill

Minimal code changes

```
@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

```
@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])
}
```

Adopting Kappa Backfill

Minimal code changes

```
@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

```yaml
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)

DATA+AI SUMMIT 2022

Thank you

Sundaram Ananthanarayanan

Senior Software Engineer, Netflix

Xinran Waibel

Senior Data Engineer, Netflix