Backfilling Flink Pipelines using Iceberg Source

Flink Forward Global 2021

Sundaram Ananthanarayanan (Real Time Data Infrastructure)
Xinran Waibel (Personalization Data Engineering)

Agenda

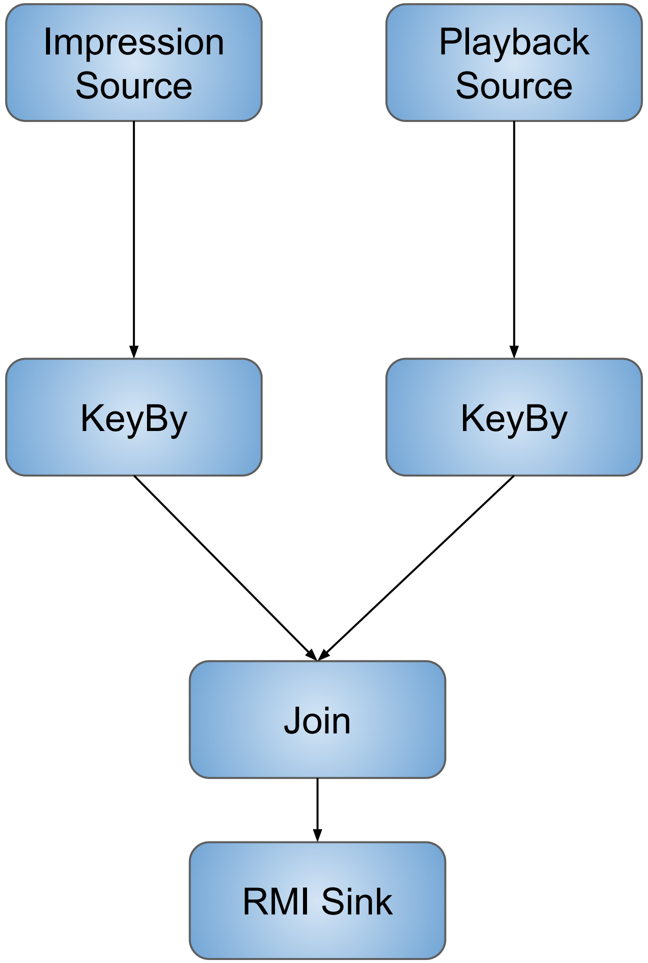
- ➤ Needs for backfilling Flink Applications
- ➤ Existing approaches
- ➤ Iceberg Source
- ➤ Event ordering challenges
- ➤ Enabling Iceberg backfill

Flink Use Cases 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: Take Rate, Evidence E/E[1], etc.
- One of the largest stateful Flink apps at Netflix



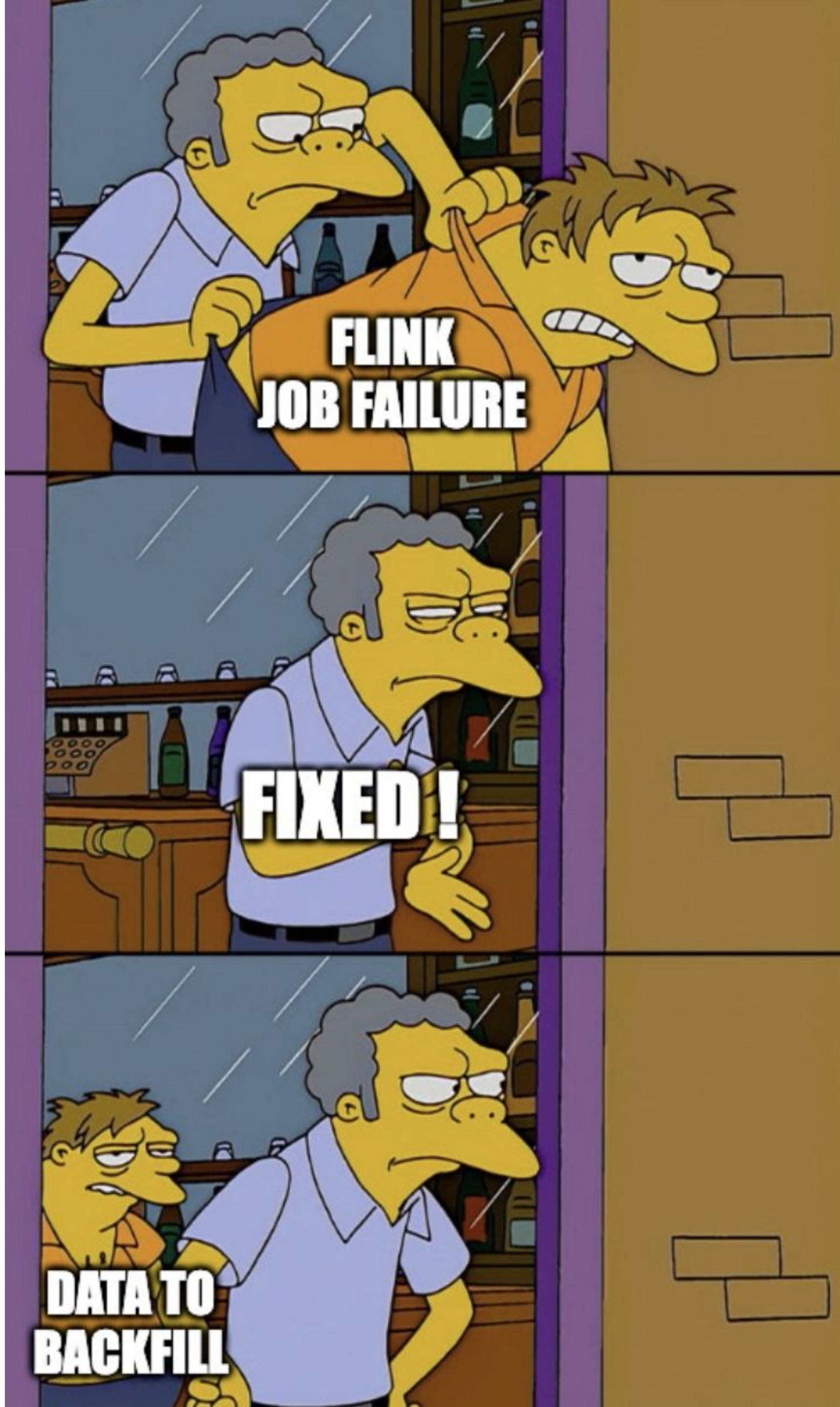
[1] https://netflixtechblog.com/artwork-personalization-c589f074ad76

Challenges with Flink Ops

Flink 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.



Challenges with Flink Ops

Possible types of backfilling needs:

- Correcting wrong data
- Backfilling missing data
- Bootstrapping state

Backfill Option #1: Replaying the Kafka Source

Methodology

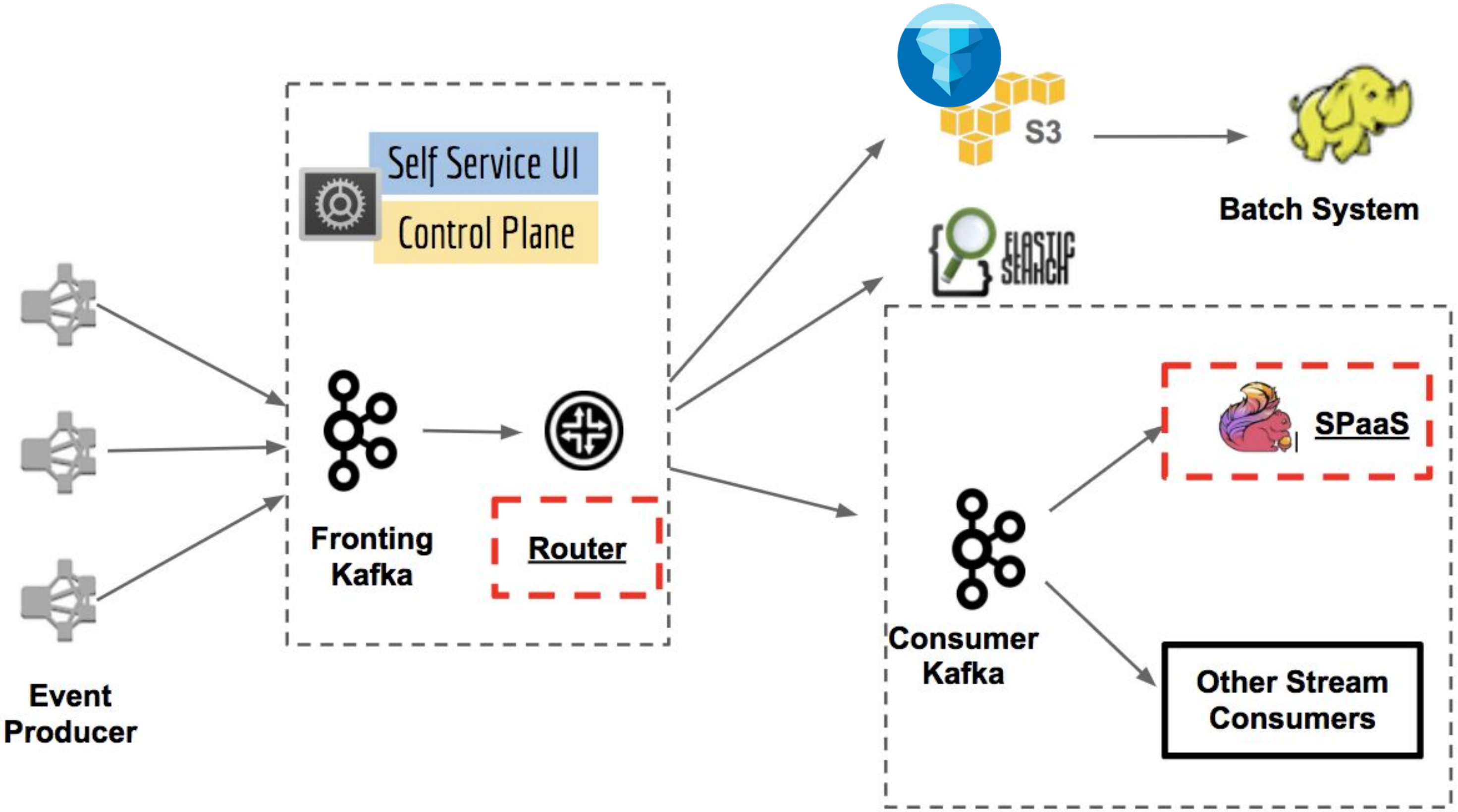
The easiest way to backfill is by re-running the Flink job to reprocess source events from the problematic period.

Challenges

- 👎 Kafka topics have limited retention.
- 👎 Troubleshooting failures can take hours or days.
- 👎 Increasing Kafka retention is very expensive. ($93M/year to retain 30 days of data generated by all apps, but some apps need > 30 days).

Can we store source Kafka events somewhere else?

Netflix's Keystone[1] platform provides a routing service that makes Kafka events available in other storage systems, e.g. Iceberg (on top of S3).



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

What is ICEBERG?

Apache Iceberg[1] is a table format for huge analytic datasets.

Features

- Schema evolution: supporting column updates.
- File pruning: based on partitions & column-level statistics.
- Time traveling: for reproducing results, plus version rollback.
- Cost effective[2]: 12x better compression rate and 98% less storage cost compared to Kafka storage.

[1] https://iceberg.apache.org/ [2] Comparing Iceberg tables on top of Parquet files in S3 with Kafka storage in gp2 EBS.

Kafka Events in 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 Iceberg tables?

Backfill Option #2: Batch Pipelines reading from Iceberg

Methodology

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

Challenges

- 👎 Initial development of such Spark job can take days or weeks, incl. data validation between two parallel applications.
- 👎 Continuous engineering efforts to keep the Spark app up to date.

Batch-driven Backfill

- Methodology: Maintain a separate batch app equivalent to the Flink app.
- Pros: Low data retention cost.
- Cons: Have to maintain two applications in parallel. 😰

Real-time Backfill

- Methodology: Rerun Flink app before Kafka sources expire.
- Pros: Backfill using the same app.
- Cons: Increasing Kafka retention is expensive. 💸

Can we combine the best things from both worlds?

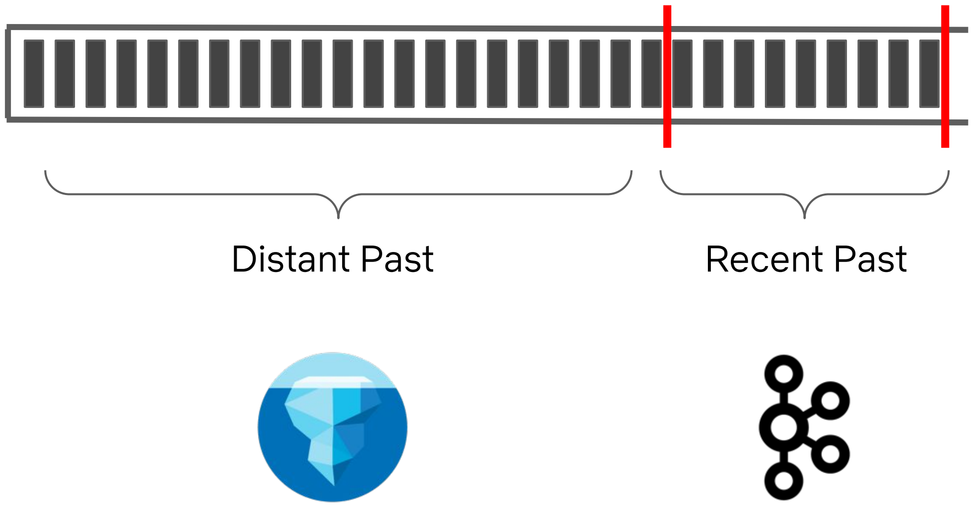

Introducing
Iceberg Source!

Iceberg Source Connector for Backfilling Flink Applications

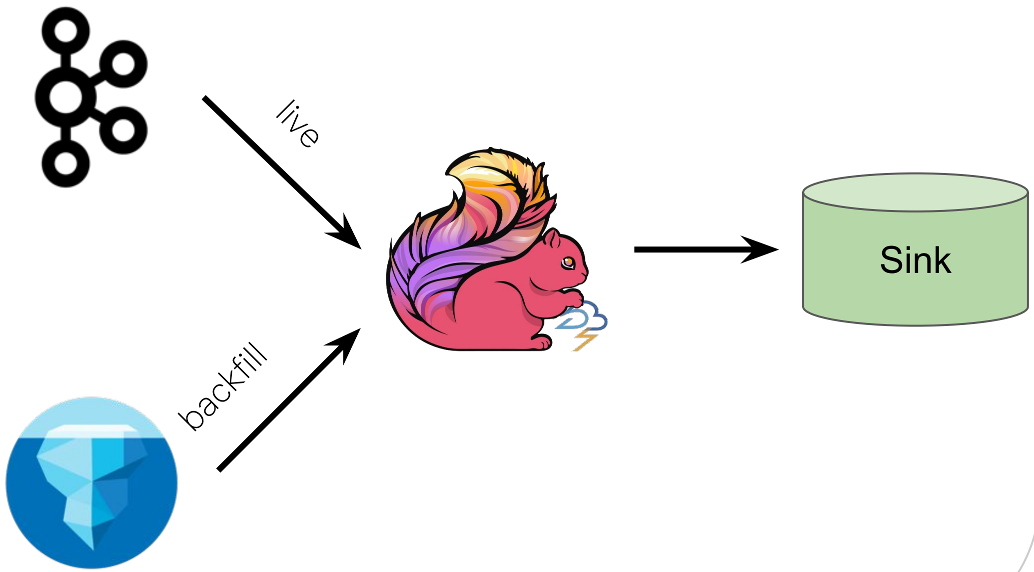
- Provides a generic solution for backfilling
- Minimal code changes to add support
- Scales horizontally to backfill quickly
- Evaluated Iceberg Source Connector in production deployment

Mechanics of backfilling using the Iceberg Source

Semantics of the DataStream



Use data from Iceberg during backfill

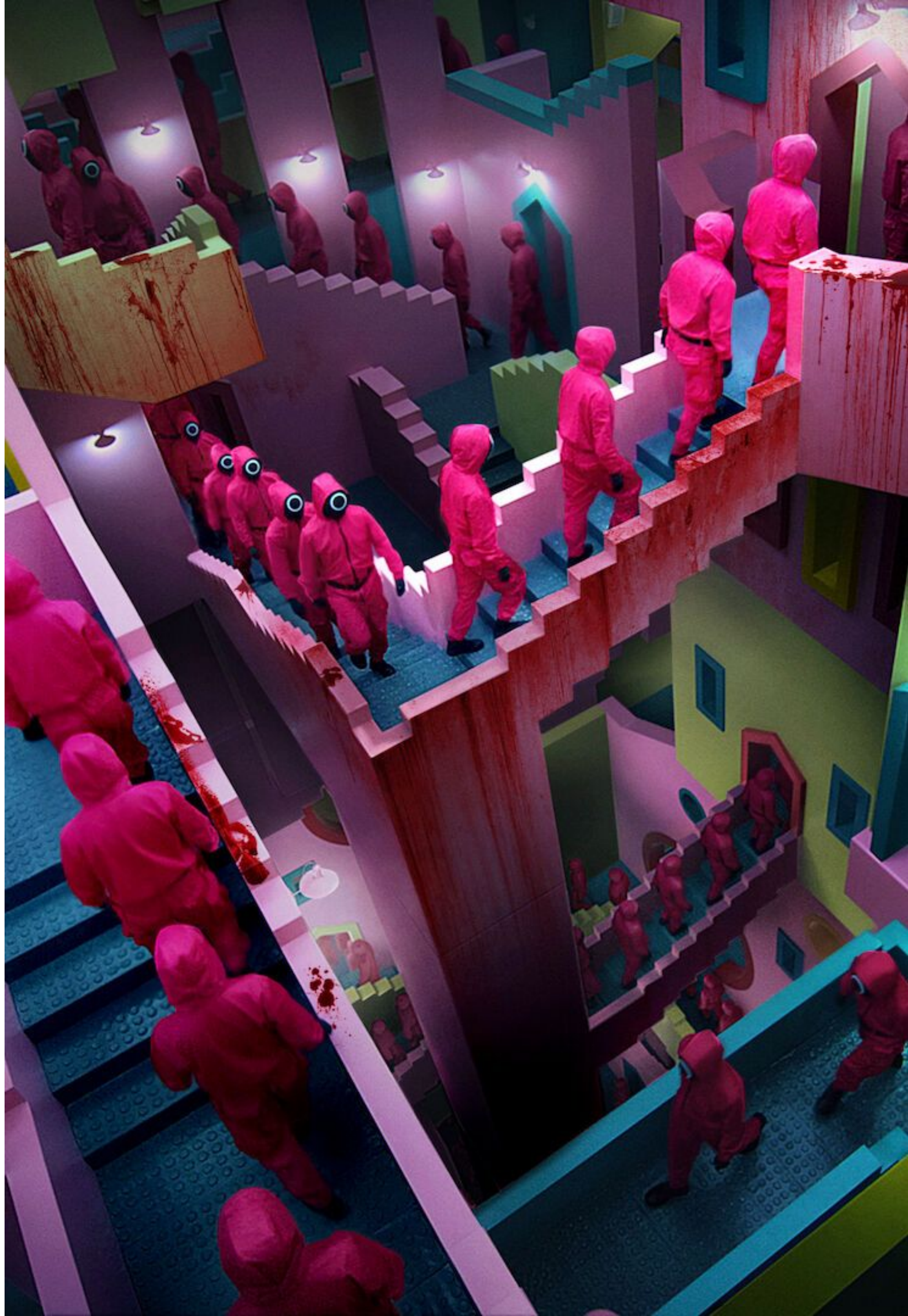


Why not use the existing OSS Iceberg Source?

- ✔ Supports reading from Iceberg Tables
- ✔ Works for both bounded and continuously streaming use-cases
- ❌ Does not support Flink use-cases where <u>ordering</u> can affect results
- ❌ Was written using Flink’s old source interfaces.

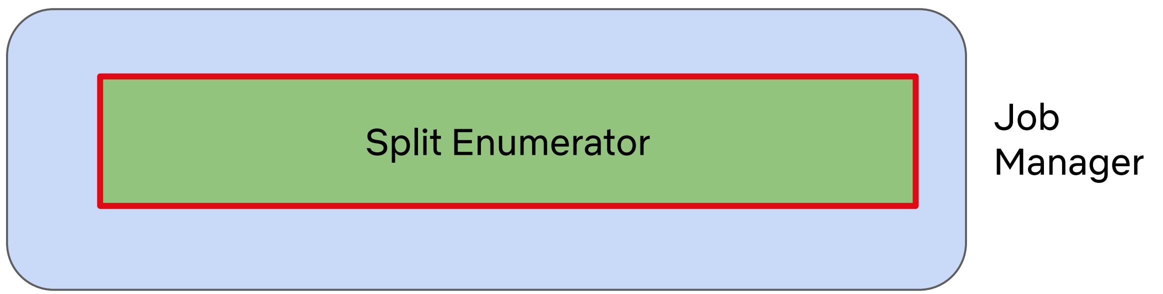
Let's build the Iceberg Source Connector based on the Source API introduced in FLIP-27[1].

How hard can it be?

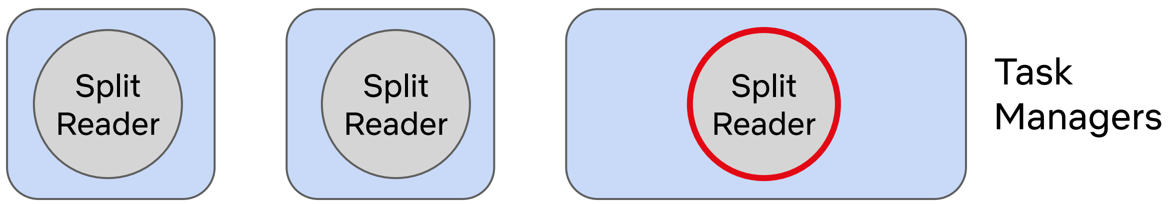


[1] https://cwiki.apache.org/confluence/display/FLINK/FLIP-27%3A+Refactor+Source+Interface

Building a FLIP-27 Source

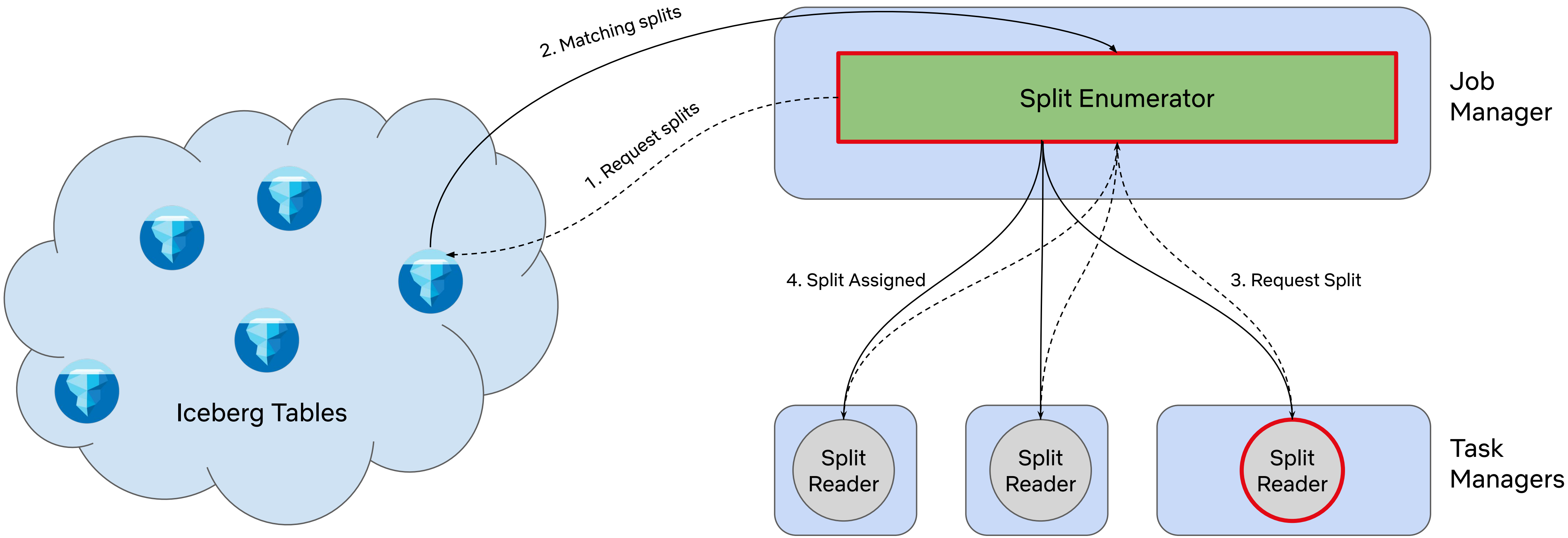


Responsible for (a). discovering splits and (b). assigning them to readers.



Responsible for emitting records by reading splits assigned to them.

Building the Iceberg Source Connector



Talk is cheap.

Show me the code.

Building the Iceberg Source Connector

```
class IcebergSplitEnumerator extends SplitEnumerator {

  def start(): Unit = ???

  def handleSplitRequest(subtaskId: Int, requesterHostname: String): Unit = ???

  def addSplitsBack(splits: util.List[IcebergSplit], subtaskId: Int): Unit = ???

  def addReader(subtaskId: Int): Unit = ???

  def snapshotState(): List[IcebergSplit] = ???

  def close(): Unit = ???


}
```

Building the Iceberg Source Connector

```
class IcebergSplitEnumerator extends SplitEnumerator {

  def start(): Unit = ???

  def handleSplitRequest(subtaskId: Int, requesterHostname: String): Unit = ???

  def addSplitsBack(splits: util.List[IcebergSplit], subtaskId: Int): Unit = ???

  def addReader(subtaskId: Int): Unit = ???

  def snapshotState(): List[IcebergSplit] = ???

  def close(): Unit = ???


}
```

Building the Iceberg Source Connector

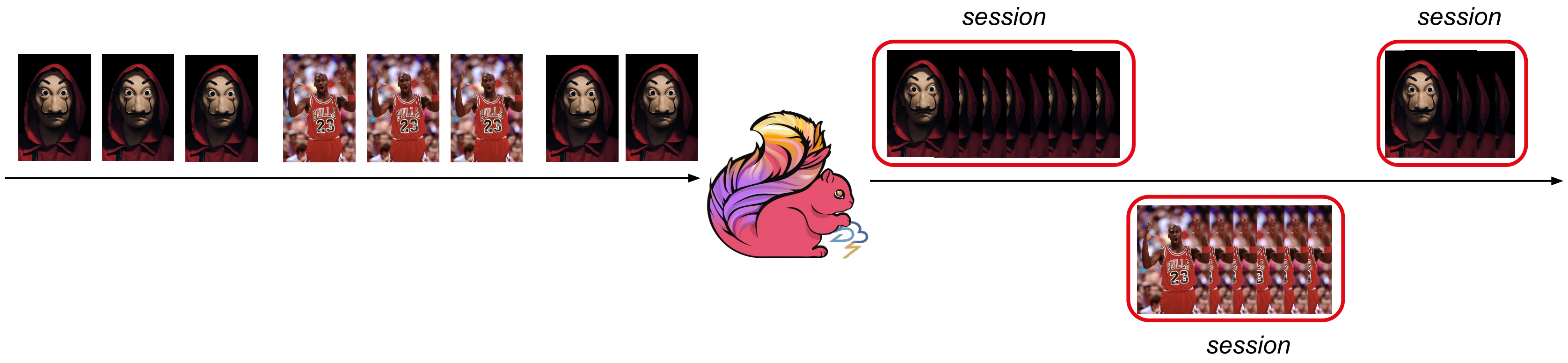
```
class IcebergSplitEnumerator extends SplitEnumerator {
  var pendingSplits: mutable.ListBuffer[IcebergSplit] = _

  def start(): Unit =
     pendingSplits =
      table
        .newScan()
        .filter(filterExpr) // filter only the table that falls in backfill period
        .planTasks()
        .iterator().asScala
        .map(toSplit)
        .to[ListBuffer]

  def handleSplitRequest(subtaskId: Int, requesterHostname: String): Unit = ???


}
```

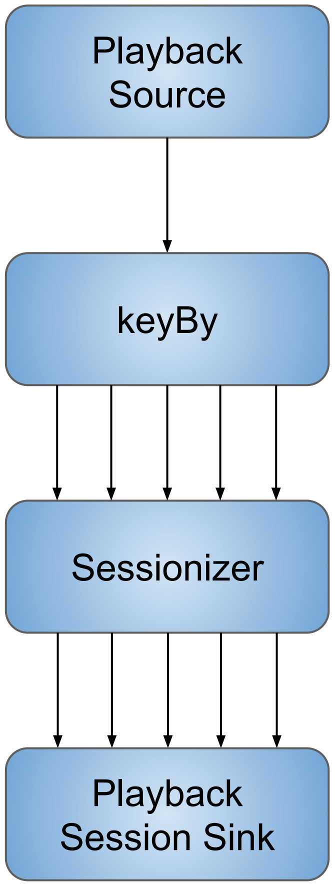
Building the Iceberg Source Connector

```
class IcebergSplitEnumerator extends SplitEnumerator {
  var pendingSplits: mutable.ListBuffer[IcebergSplit] = _

  def start(): Unit = {...}

  def handleSplitRequest(subtaskId: Int, requesterHostname: String): Unit =
    if (pendingSplits.nonEmpty) {
      context.assignSplit(pendingSplits.head, subtaskId)
      pendingSplits = pendingSplits.tail
    } else {
      context.signalNoMoreSplits(subtaskId)
    }

}
```


There’s no free lunch!

Challenge 1: Applications assume ordering!

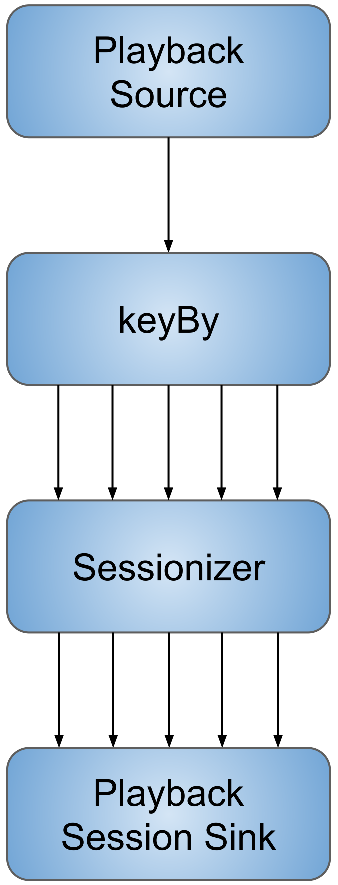




Example Flink Application that converts playback events into playback sessions

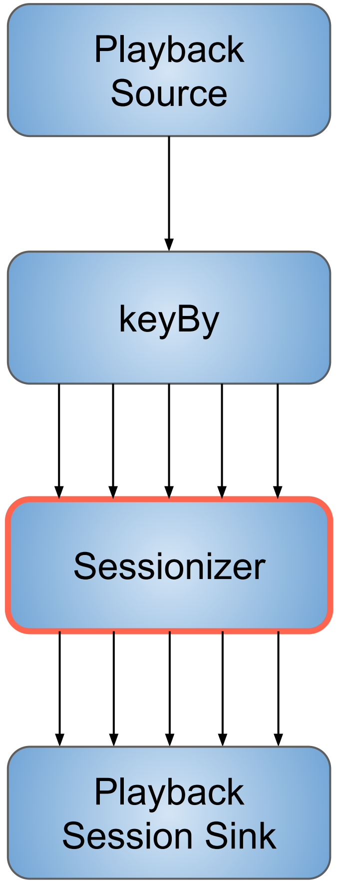
Challenge 1: Applications assume ordering!

```
@SpringBootApplication
class PlaybackSessionJob {
  @Bean
  def flinkJob(@Source("playback") playbackSrcBuilder:
SourceBuilder[PlaybackEvent]): FlinkJob =
    env => {
        val playbackSrc = playbackSrcBuilder.build(env);

        playbackSrc
            .keyBy(_.userId)
            .process(new Sessionizer)
            .addSink(new PlaybackSessionSink)
      }
  }
}
```



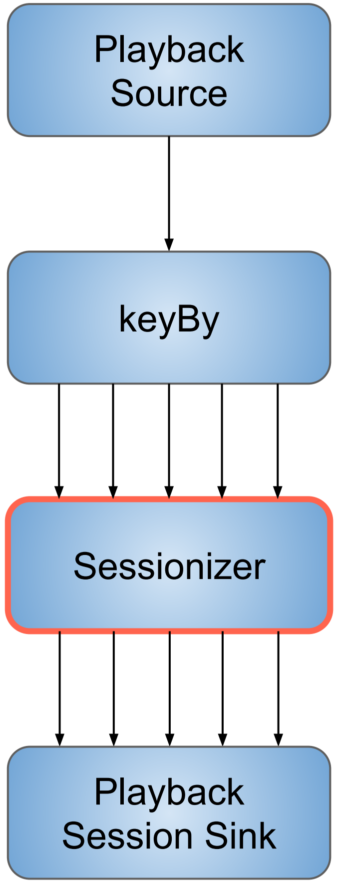
Challenge 1: Applications assume ordering!

```
class Sessionizer extends KeyedProcessFunction {
  private var start: ValueState[Long] = _
  private var end: ValueState[Long] = _

  override def processElement(evt: PlaybackEvent, ...) {...}

  override def onTimer(timestamp: long, ...) {...}
}
```



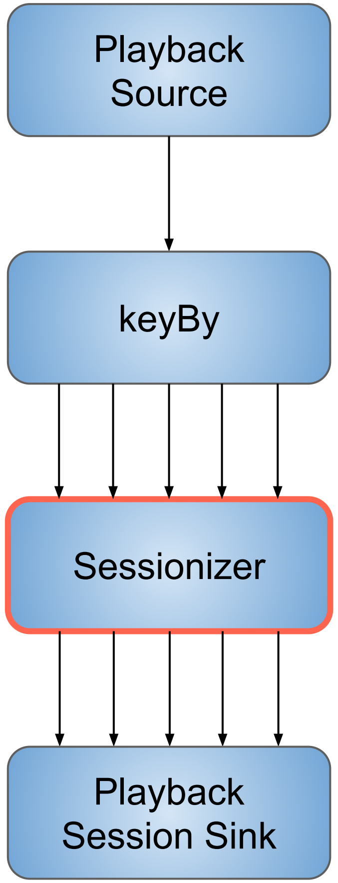
Challenge 1: Applications assume ordering!

```
class Sessionizer extends KeyedProcessFunction {
  private var start: ValueState[Long] = _
  private var end: ValueState[Long] = _
  override def processElement(evt: PlaybackEvent, ...) {
    // does this represent a new session?
    if (!start.value) {
      start.update(evt.timestamp)
    }

    // is this the latest event for the session?
    if (!end.value || evt.timestamp > end.value) {
        end.update(evt.timestamp)
    }

    // setup a probe to check for session completion in a minute
    ctx.timerService().registerEventTimeTimer(evt.timestamp + 60*1000);
  }

  override def onTimer(timestamp: long, ...) {...}
}
```



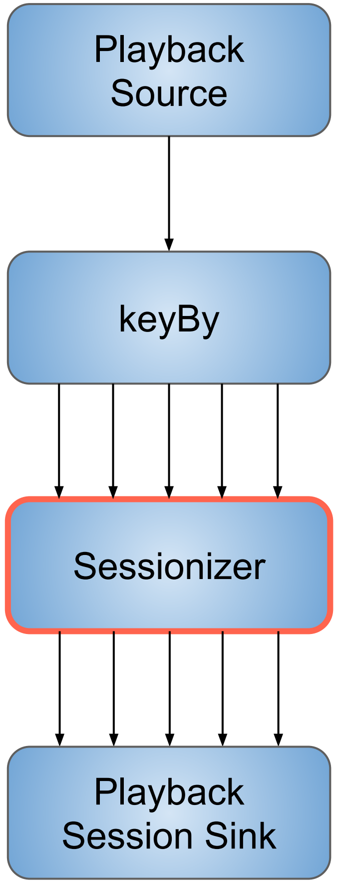
Challenge 1: Applications assume ordering!

```
class Sessionizer extends KeyedProcessFunction {
  ...

  override def processElement(evt: PlaybackEvent, ...) {...}

  override def onTimer(timestamp: long, ...) {
    // emit session if no new events
    if (end.value && timestamp - end.value >= THRESHOLD) {
      output.collect(PlaybackSession(start.value, end.value))
      start.clear
      end.clear
    }
  }
}
```



Challenge 1: Applications assume ordering!

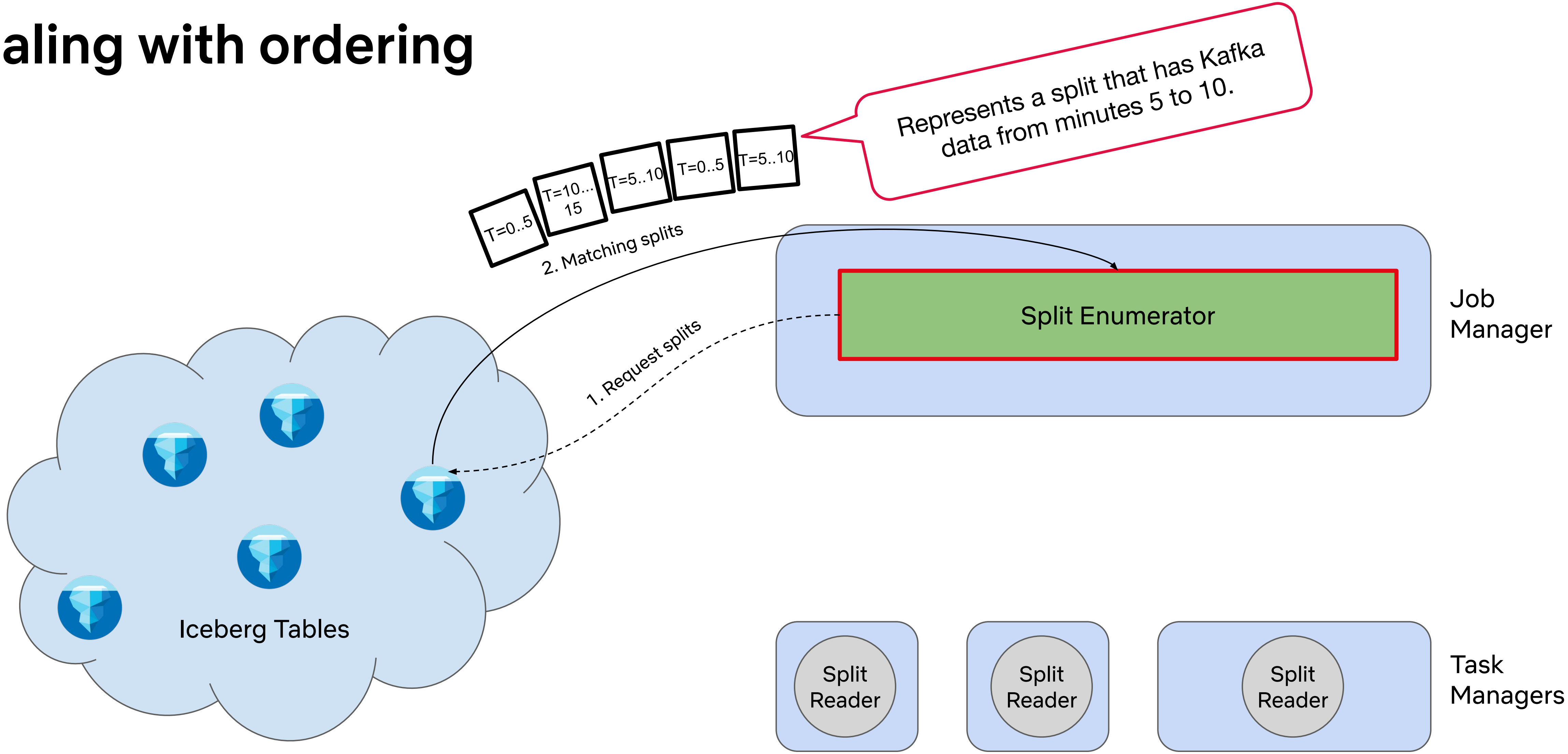
```
class Sessionizer extends KeyedProcessFunction {
  private var start: ValueState[Long] = _
  private var end: ValueState[Long] = _

  override def processElement(evt: PlaybackEvent, ...) {
    // does this represent a new session?
    if (!start.value) {
      start.update(evt.timestamp)
    }

    // is this the latest event for the session?
    if (!end.value || evt.timestamp > end.value) {
        end.update(evt.timestamp)
    }

    // setup a probe to check for session completion in a minute
    ctx.timerService().registerEventTimeTimer(evt.timestamp + 60*1000);
  }

  override def onTimer(timestamp: long, ...) {
    // emit session if no new events
    if (end.value && timestamp - end.value >= THRESHOLD) {
      output.collect(PlaybackSession(start.value, end.value))
      start.clear
      end.clear
    }
  }
}
```



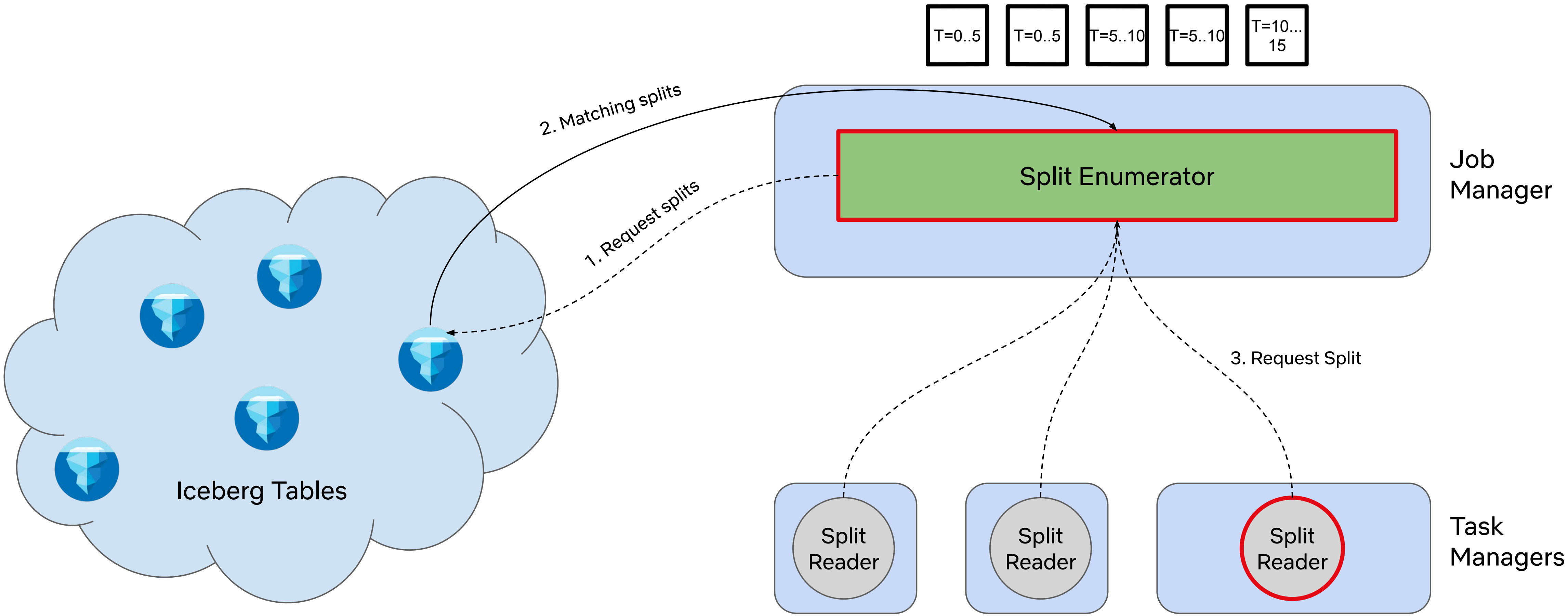
😞 If events were to be emitted in a different order during backfill, the results will not match.

Can we order the splits based on their ingestion timestamps and assign them in the exact same order?

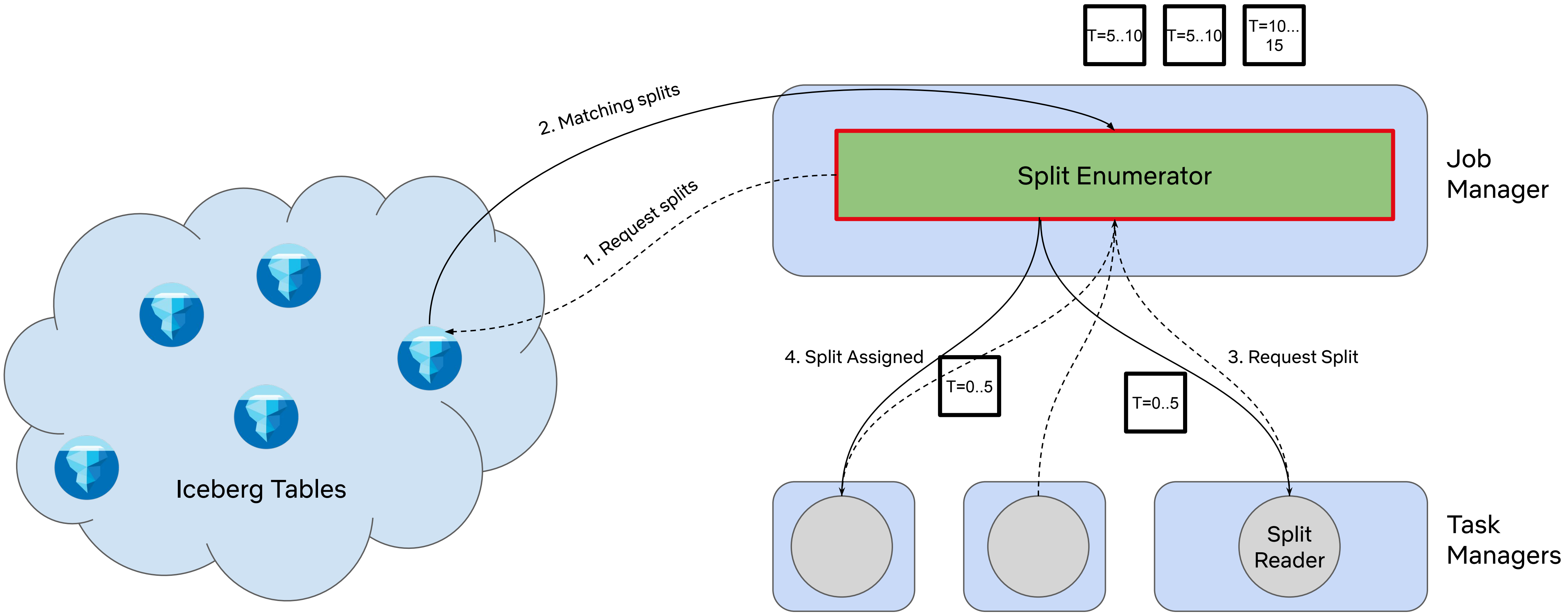
Dealing with ordering



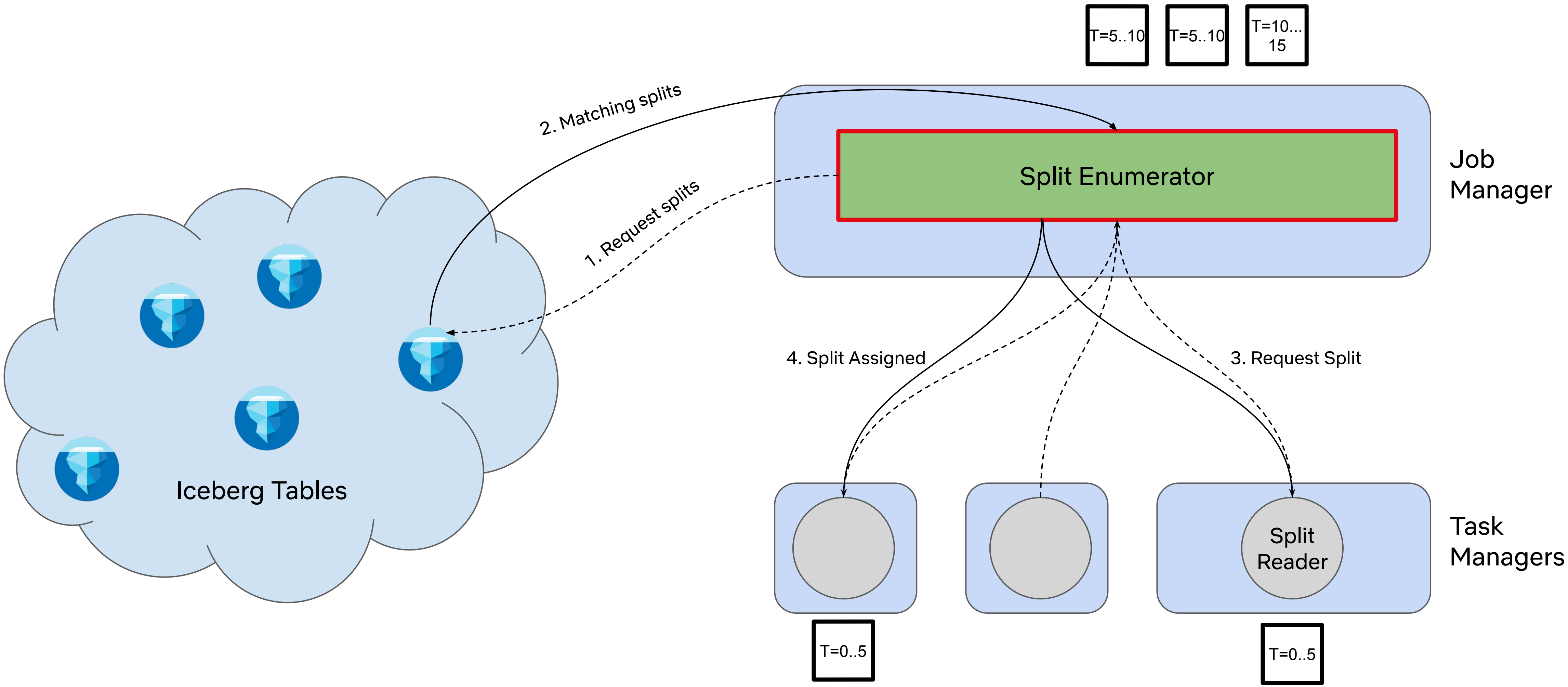
Dealing with ordering



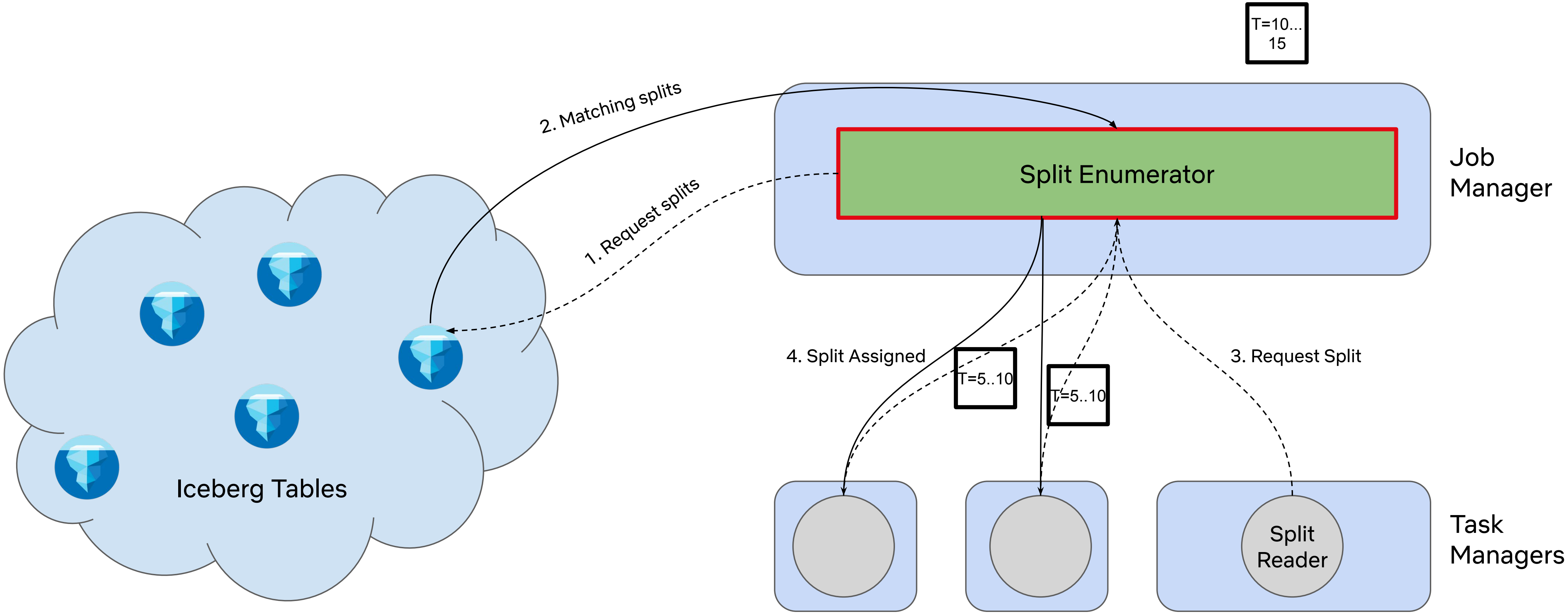
Dealing with ordering



Dealing with ordering



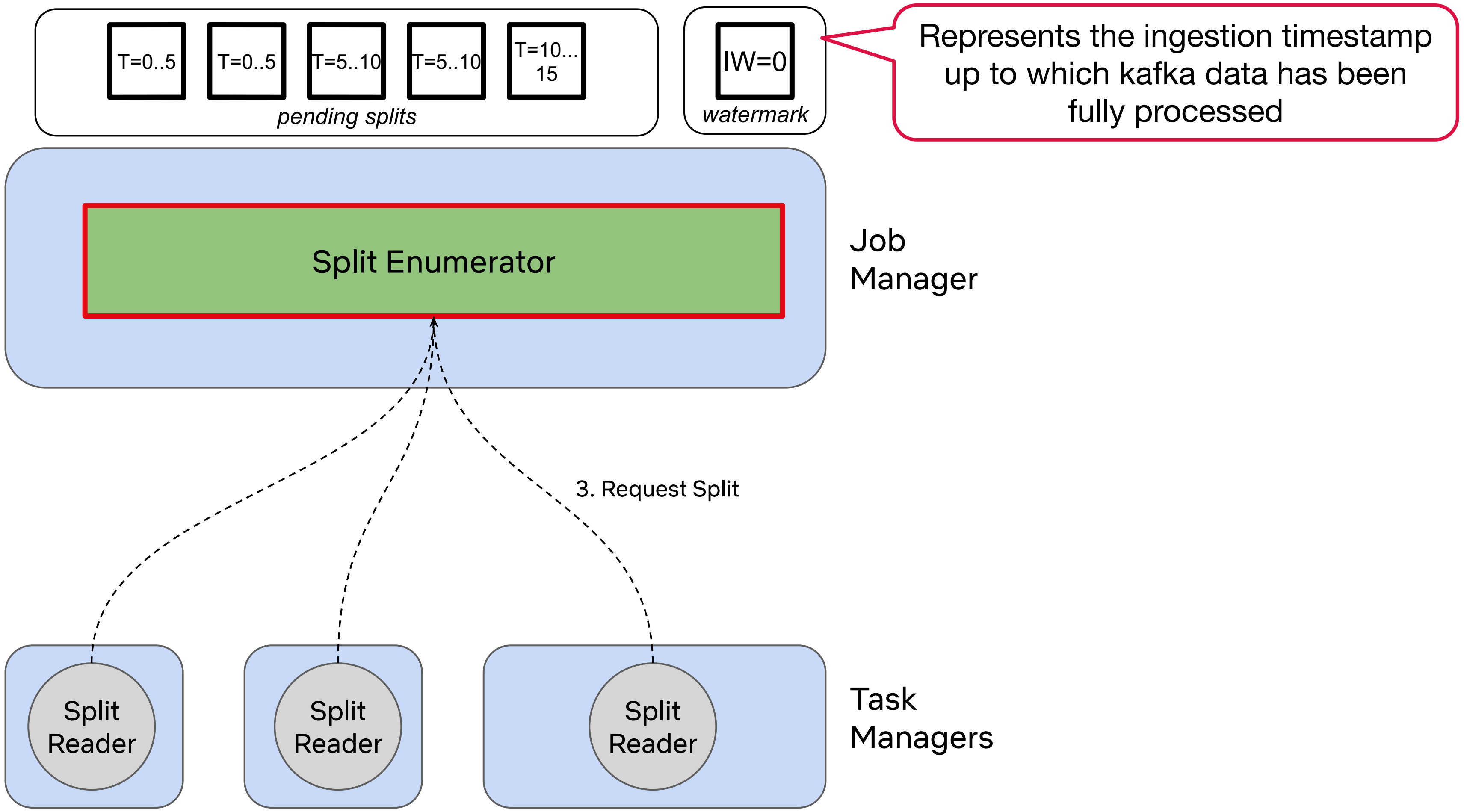
Dealing with ordering



Observation: Throughput is low because of the need to strictly order the splits.

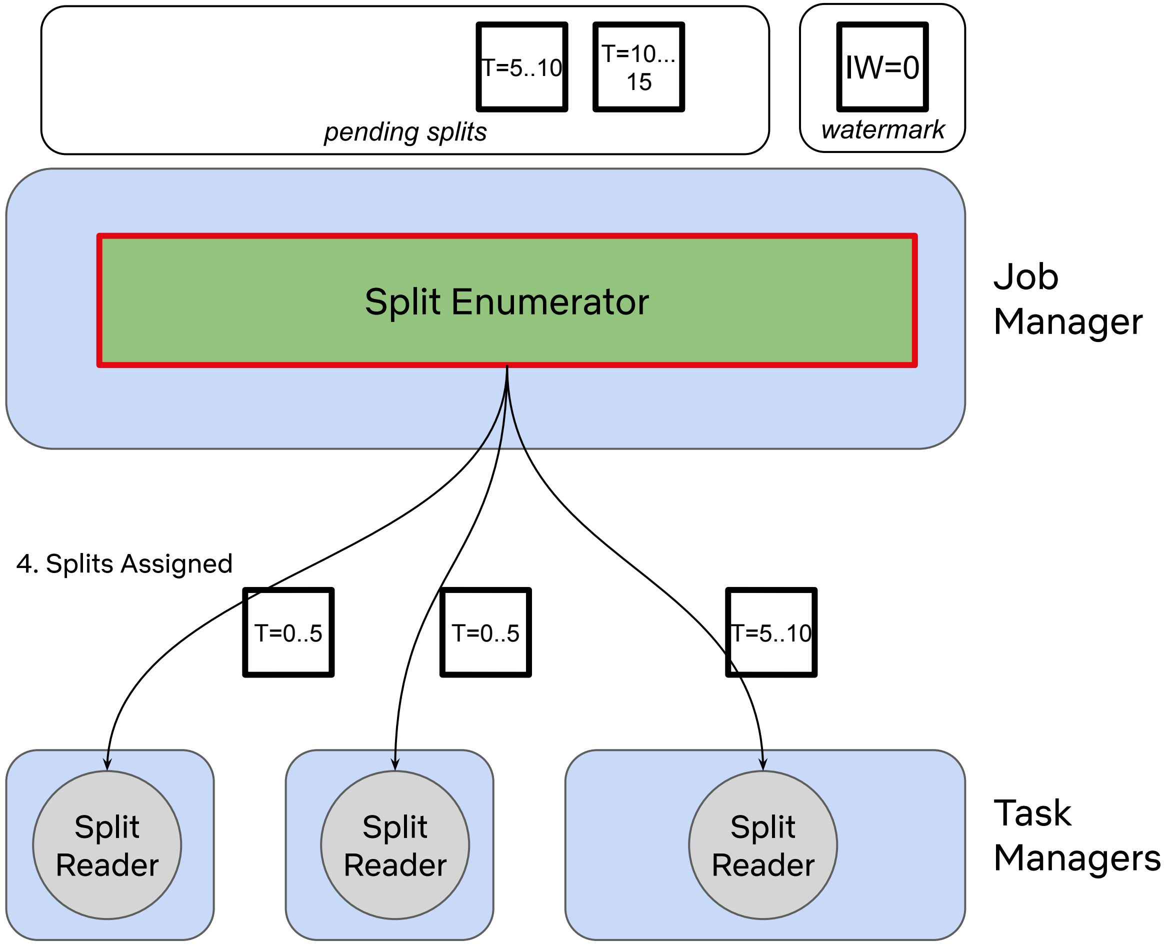
Not all Flink Applications rely on such strong ordering guarantees. They can generally tolerate some lateness.

Dealing with ordering

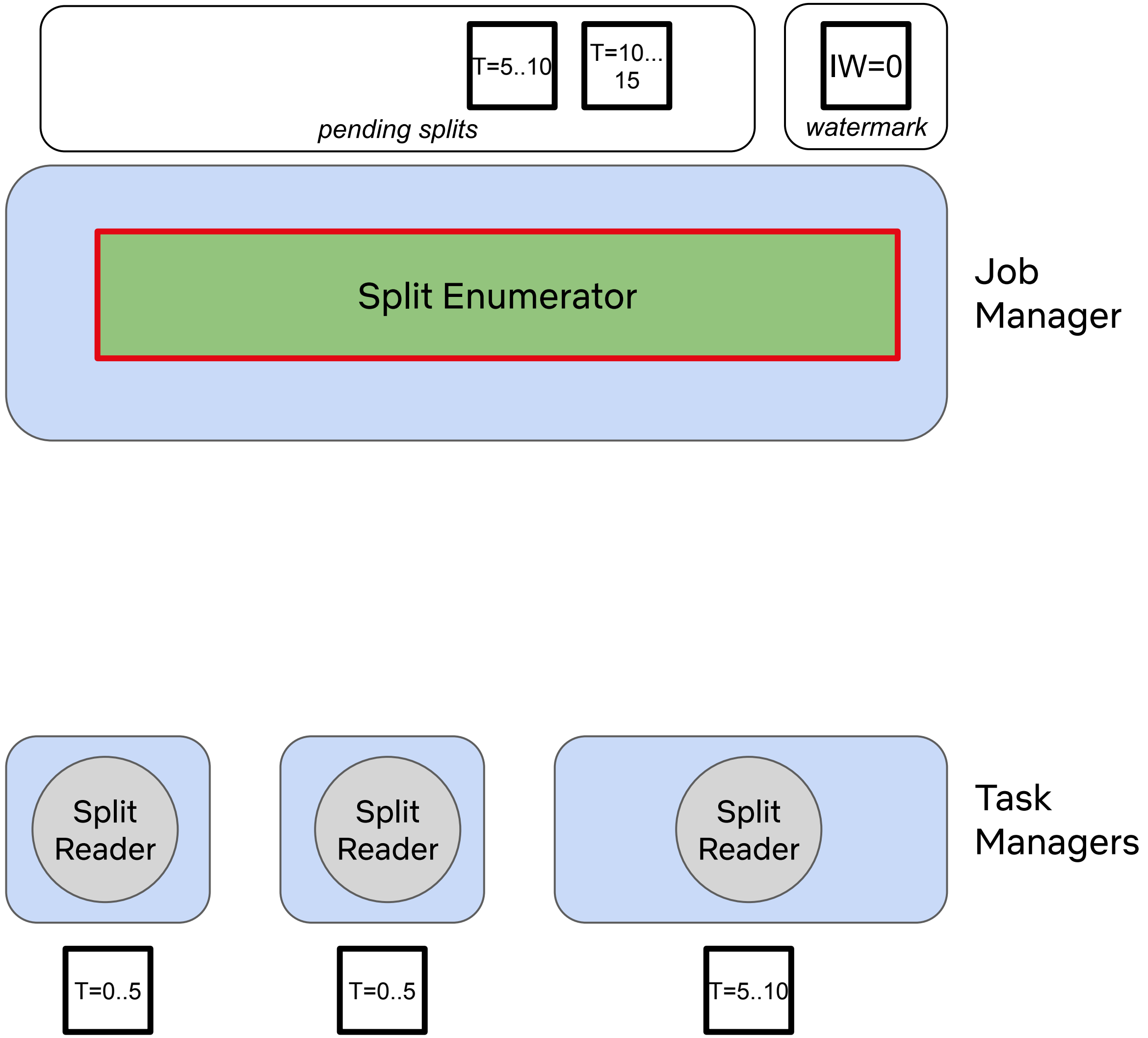


Assuming lateness of “10” minutes is okay.

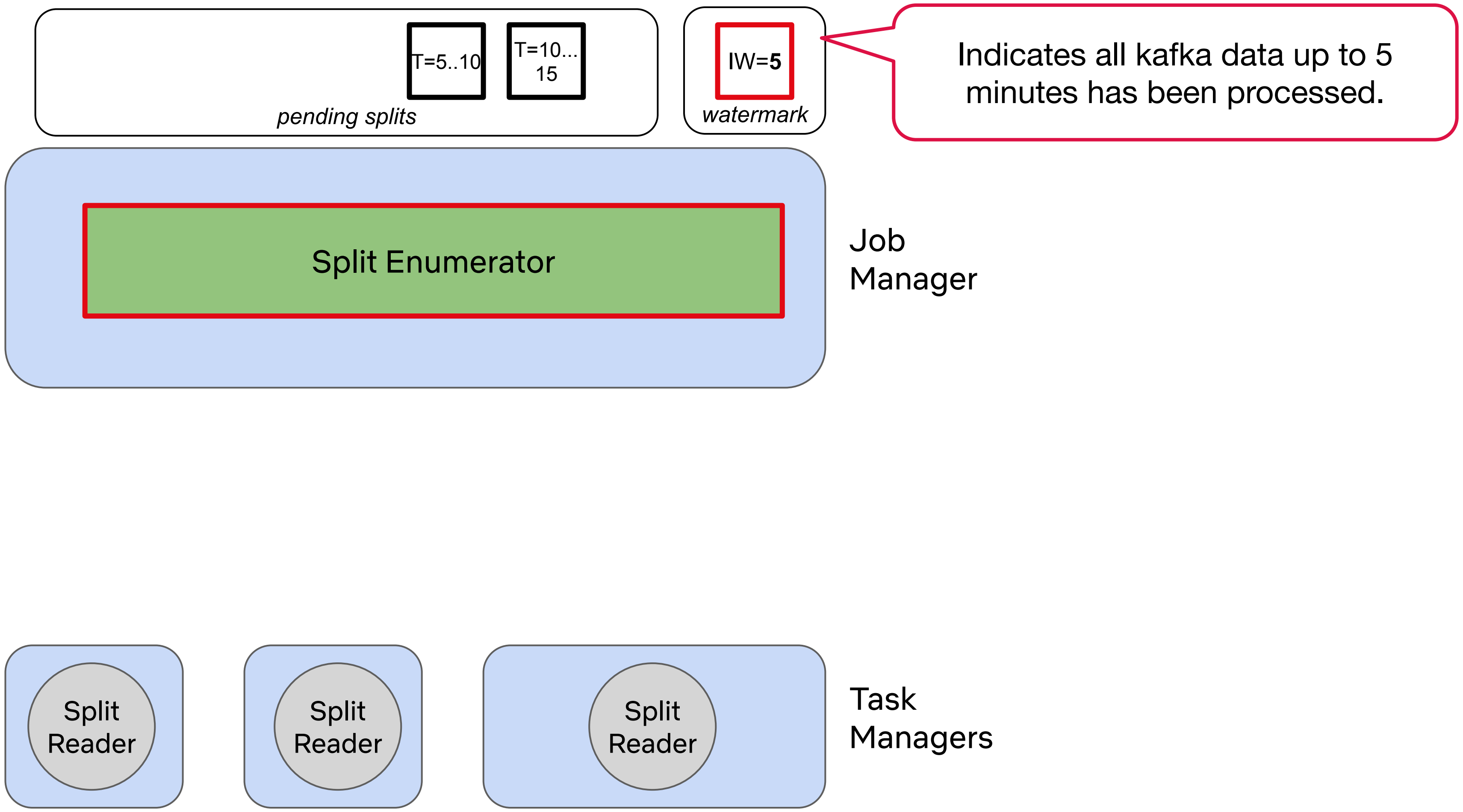
Dealing with ordering



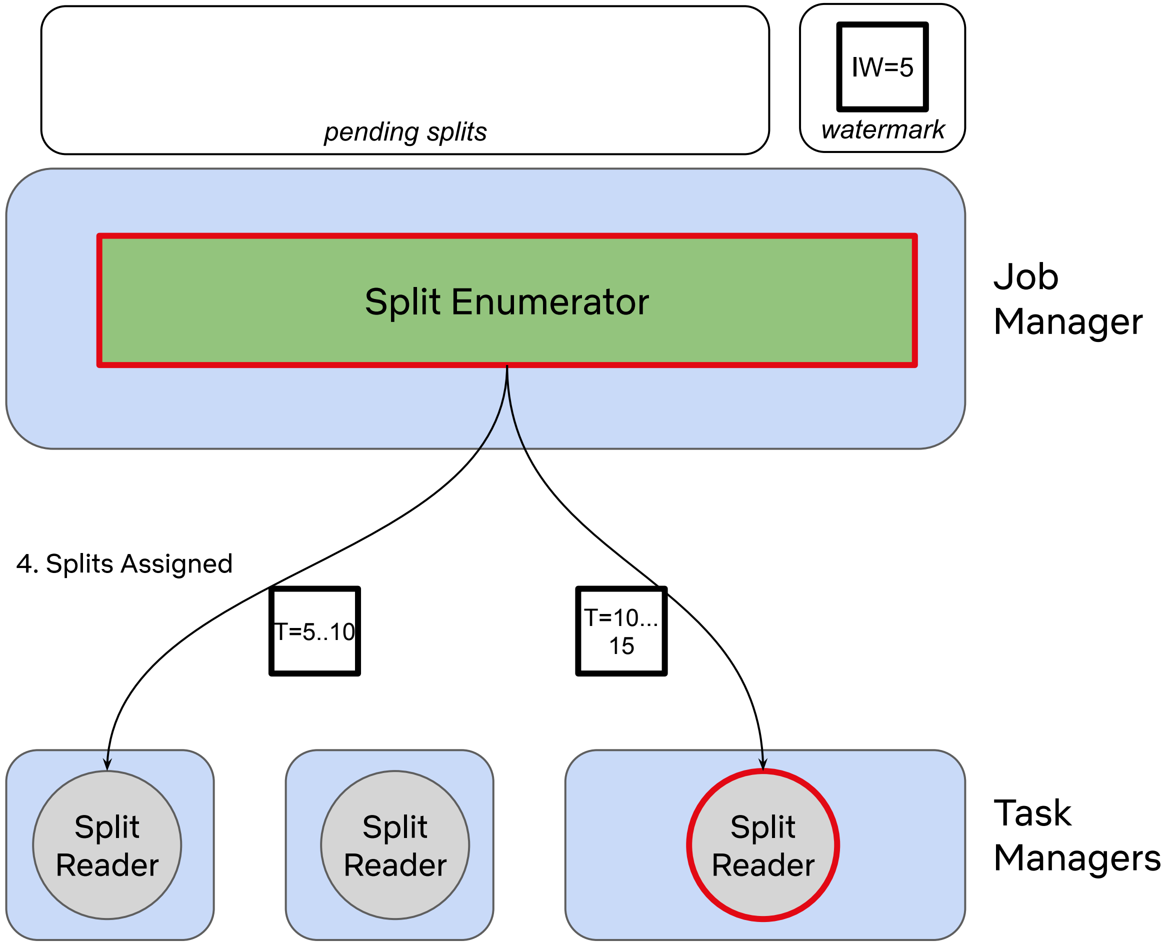
Dealing with ordering



Dealing with ordering



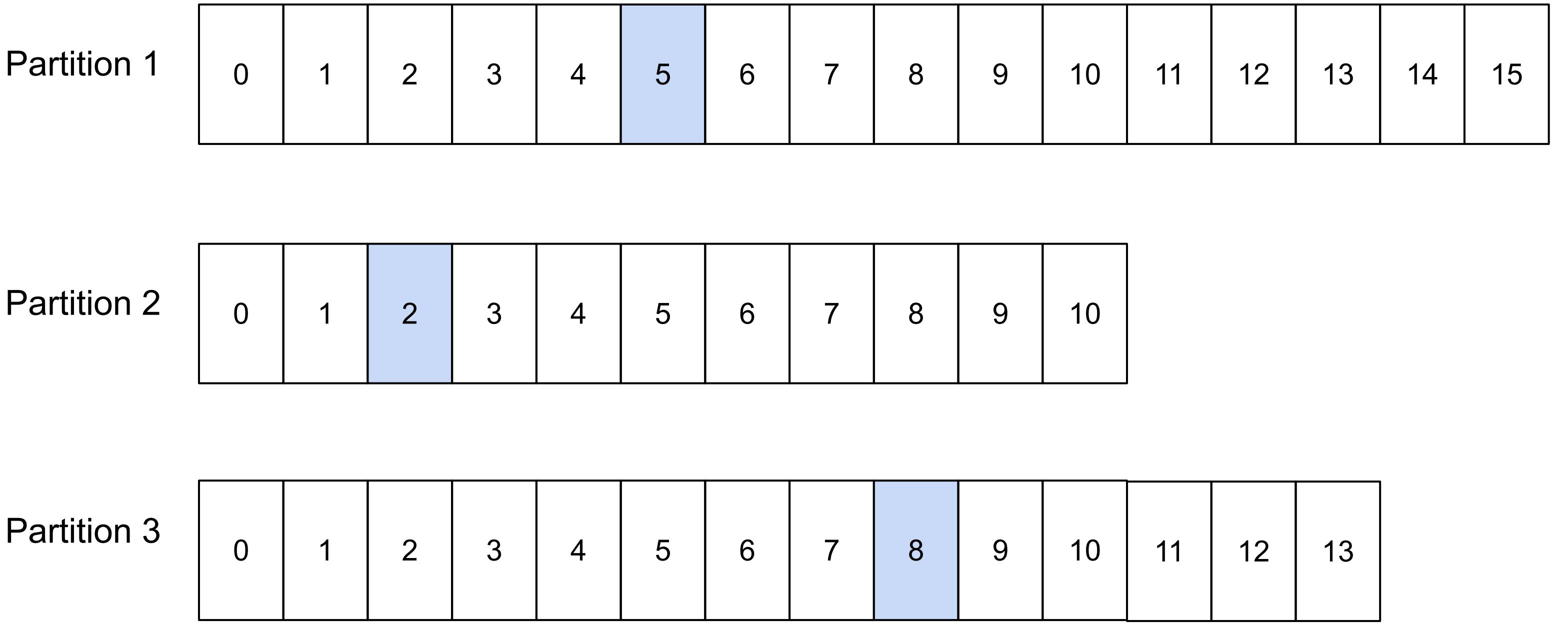
Dealing with ordering



✔ Improves the throughput for most Flink applications that can tolerate some lateness

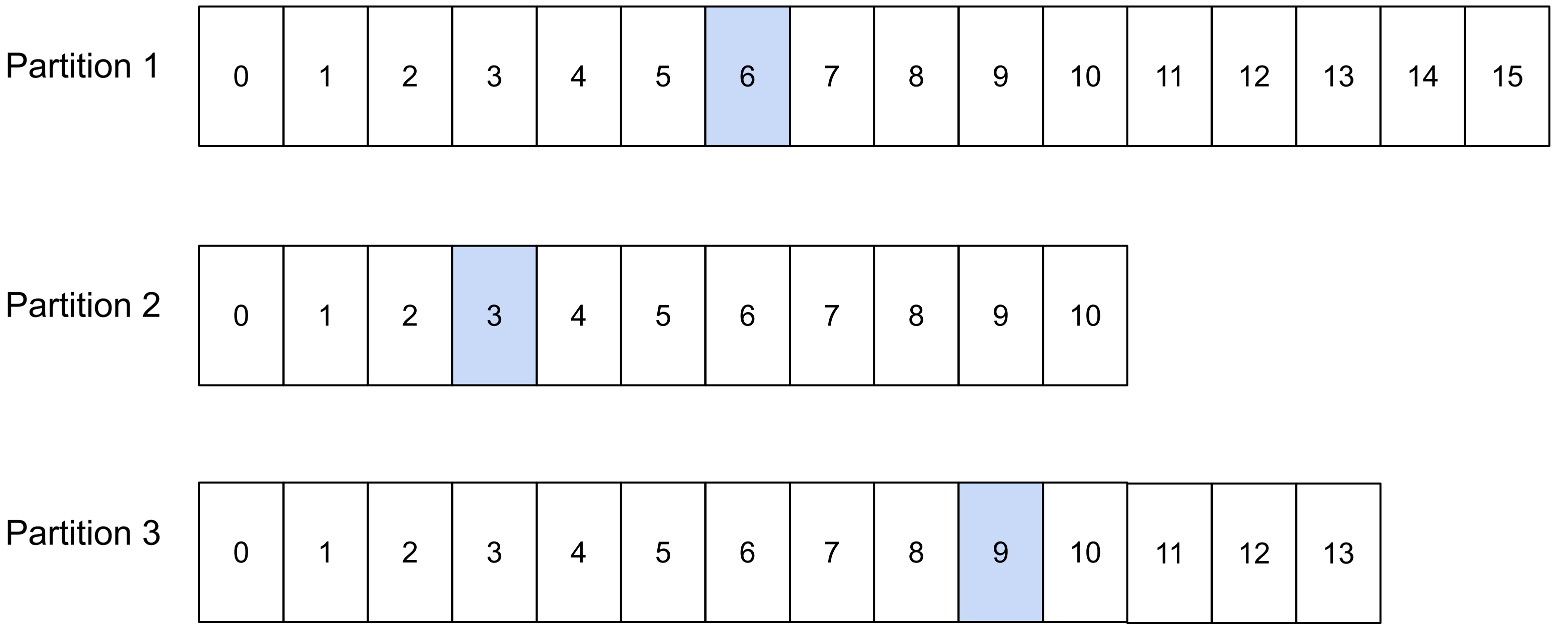
What about Kafka Ordering?

- Kafka guarantees strict ordering per partition.



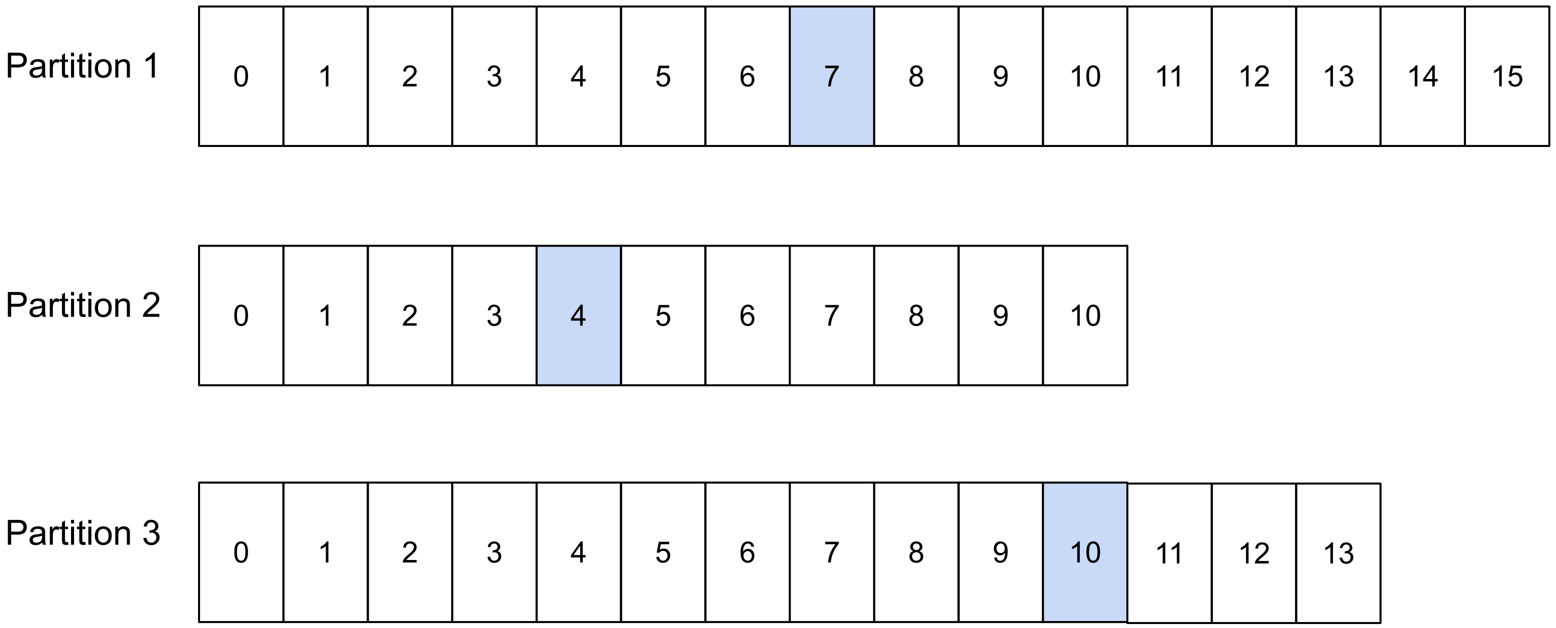
What about Kafka Ordering?

- Kafka guarantees strict ordering per partition.



What about Kafka Ordering?

- Kafka guarantees strict ordering per partition.



What about Kafka Ordering?

- Kafka guarantees strict ordering per partition

- Most analytical use-cases (streaming-joins, sessionization) use event-time semantics and are written with lateness in mind.

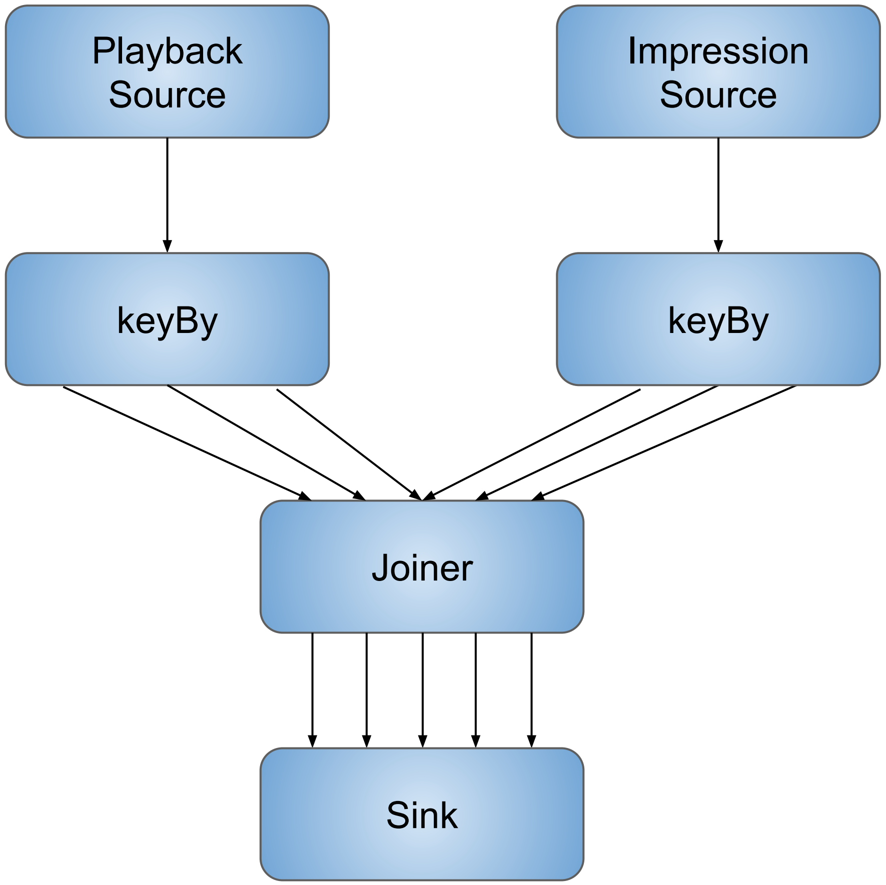
If we need to guarantee Kafka ordering

❌ On the write path, data will have to be partitioned along kafka partitioning schema producing too many small files.

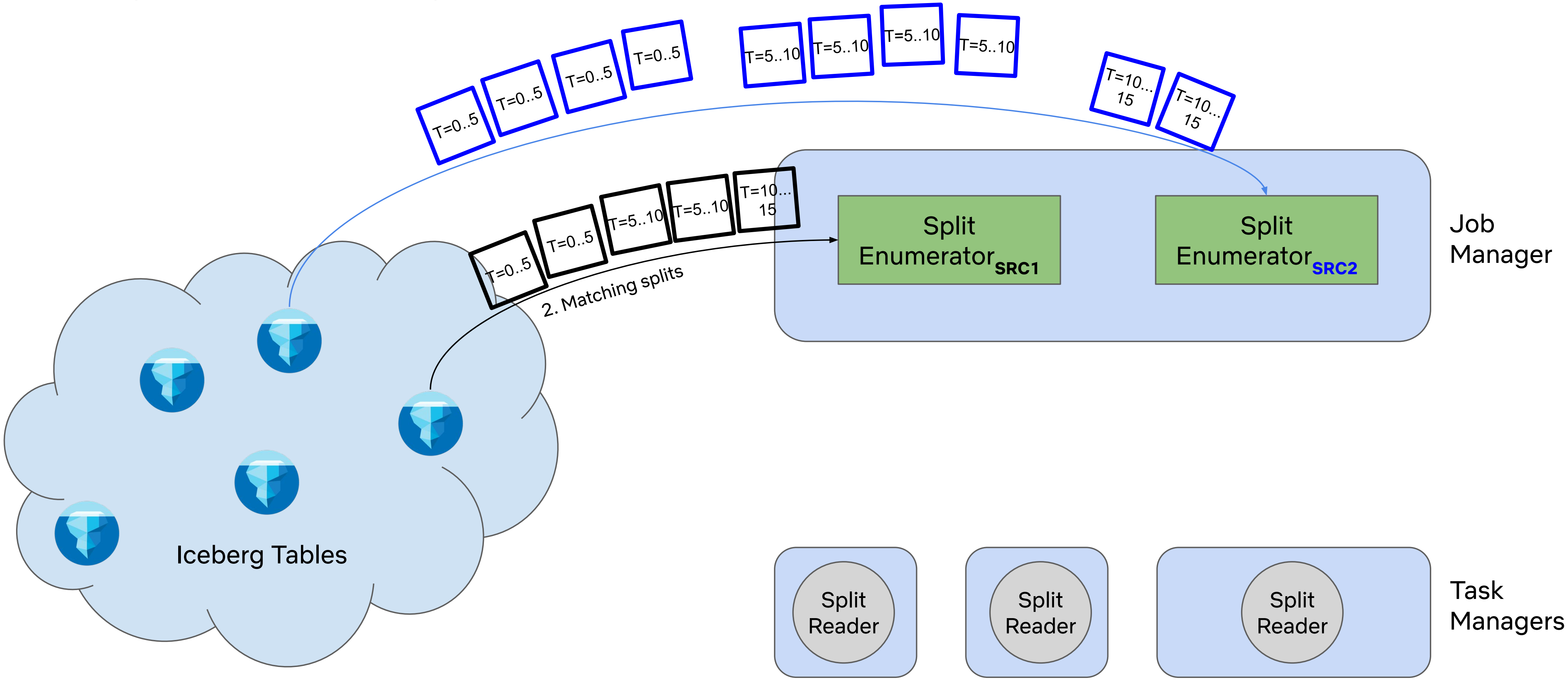
❌ Will hurt backfilling performance.

Challenge 2: Dealing with multiple sources

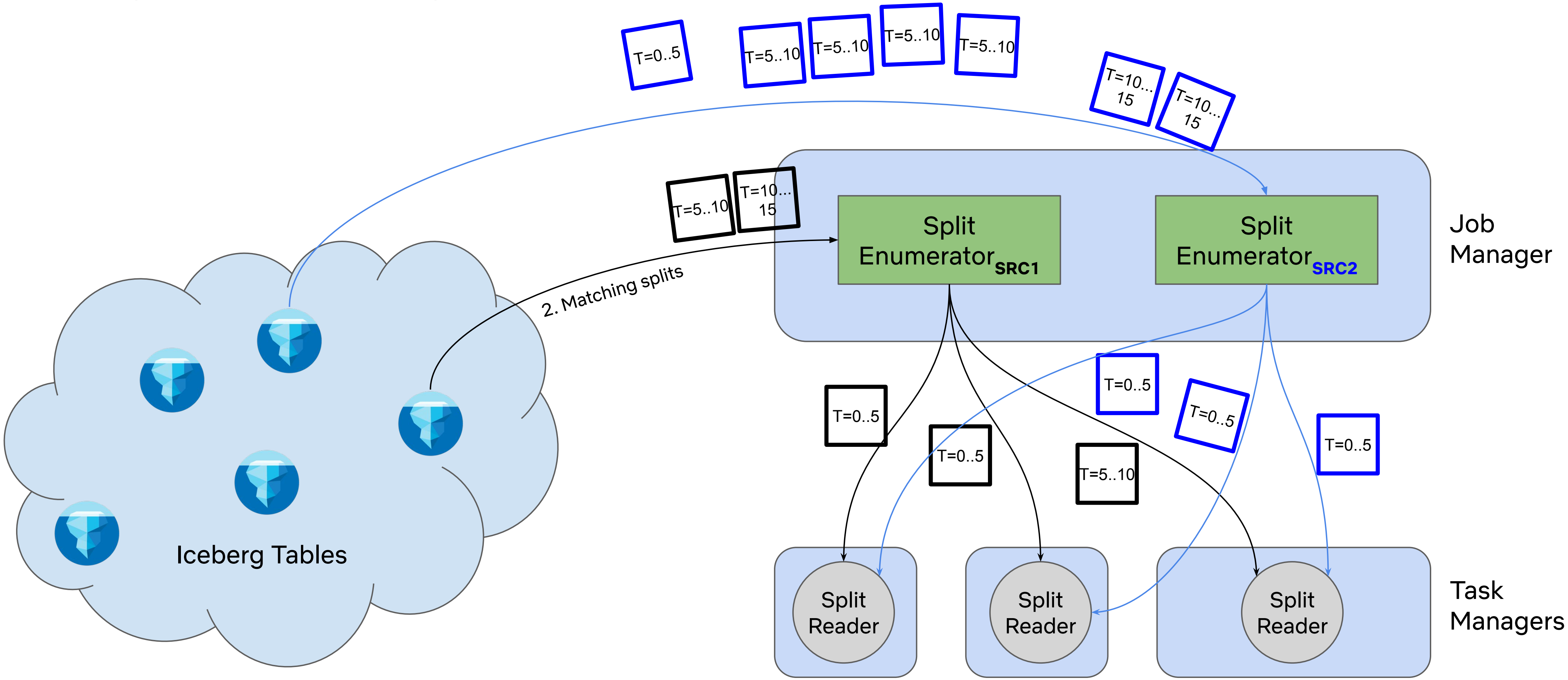
- One source can have significantly way more data than the other.
- During backfill, this could lead to a watermark skew resulting in state size explosion.
- This can eventually lead to slow checkpoints or checkpoint timeouts.



Challenge 2: Dealing with multiple sources



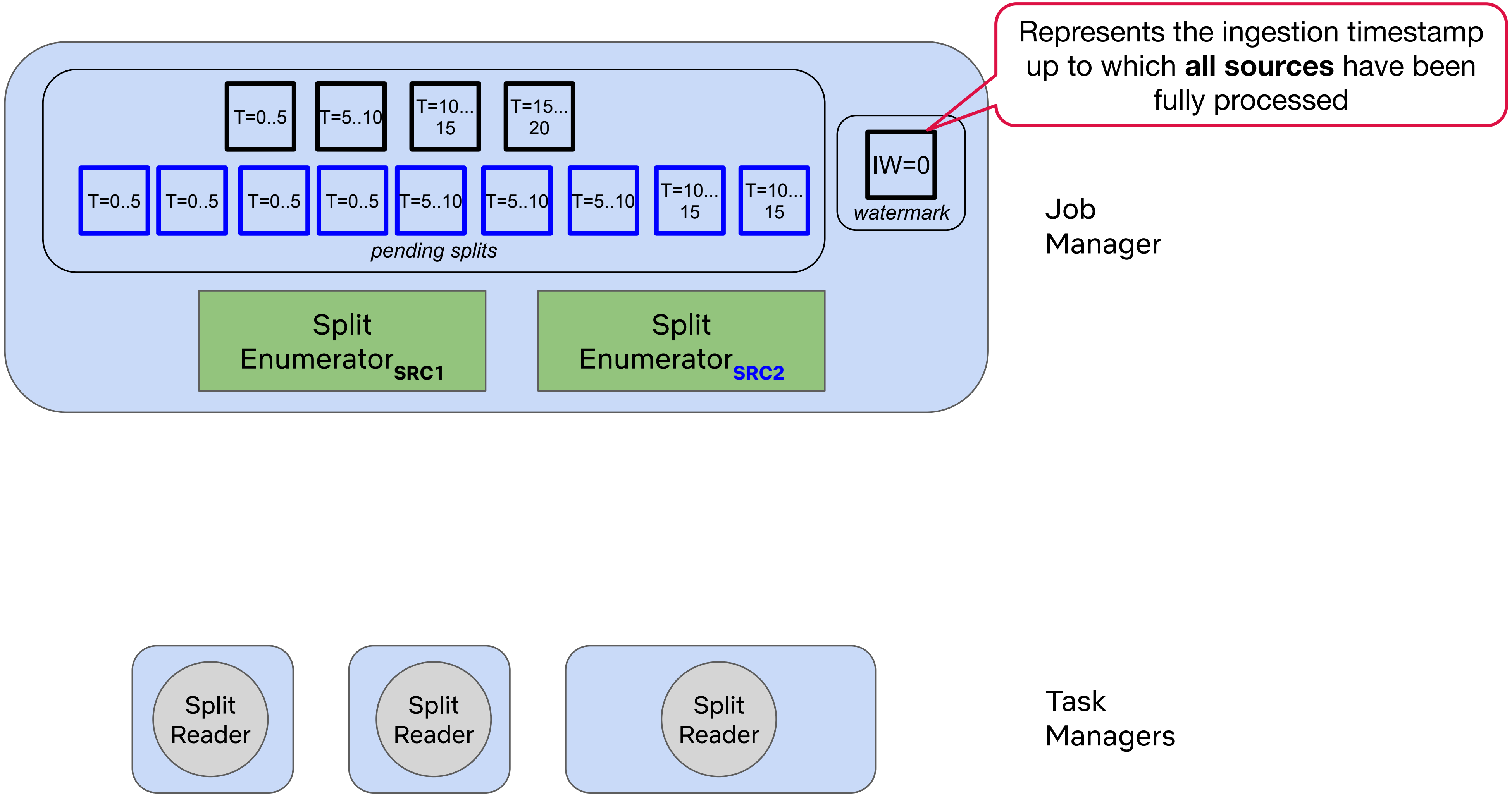
Challenge 2: Dealing with multiple sources



Source$_1$ is progressing at 2x the rate as Source$_2$.

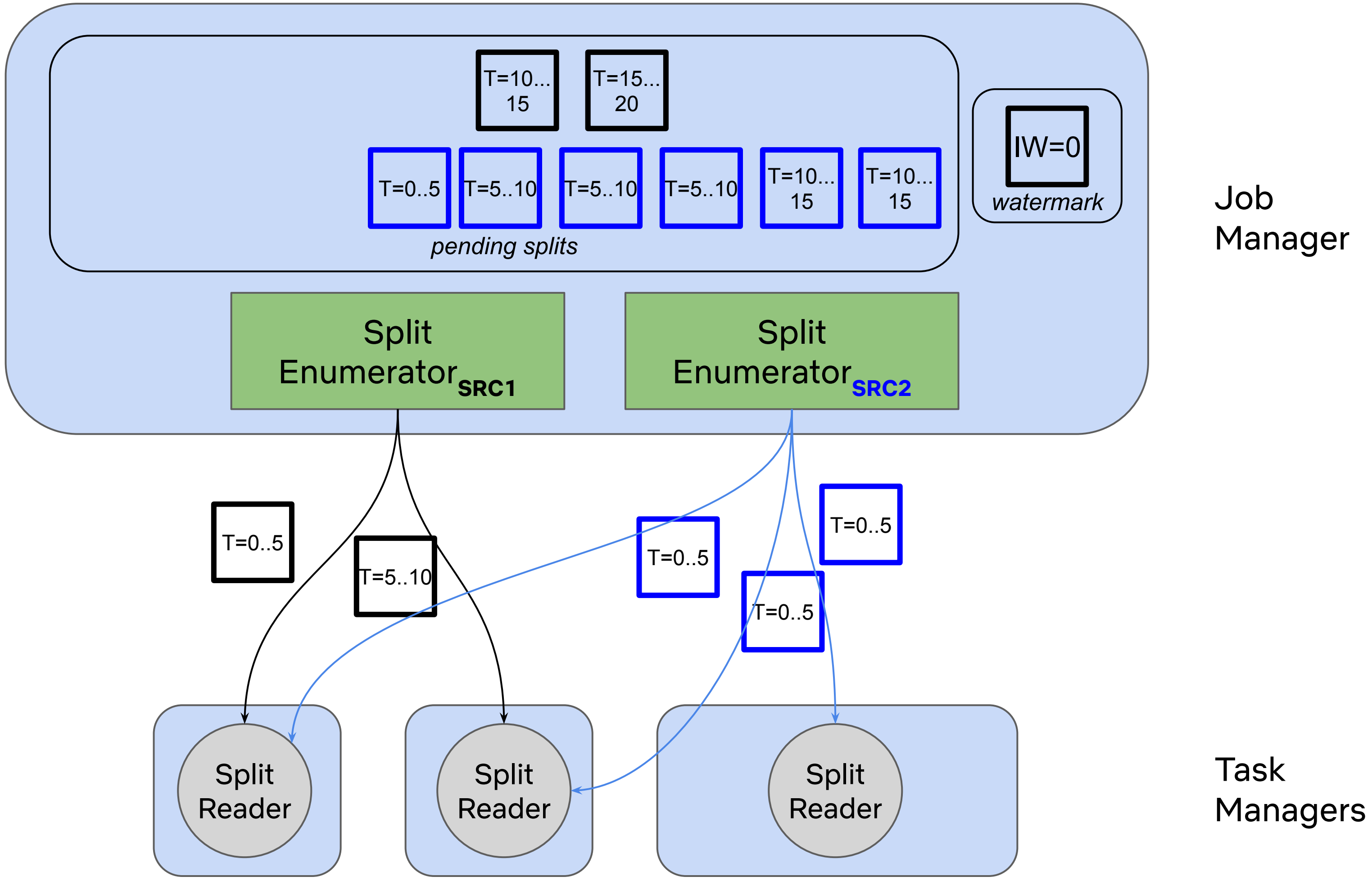
Can we coordinate the enumerators such that their ingestion watermarks advance similarly?

Dealing with multiple sources

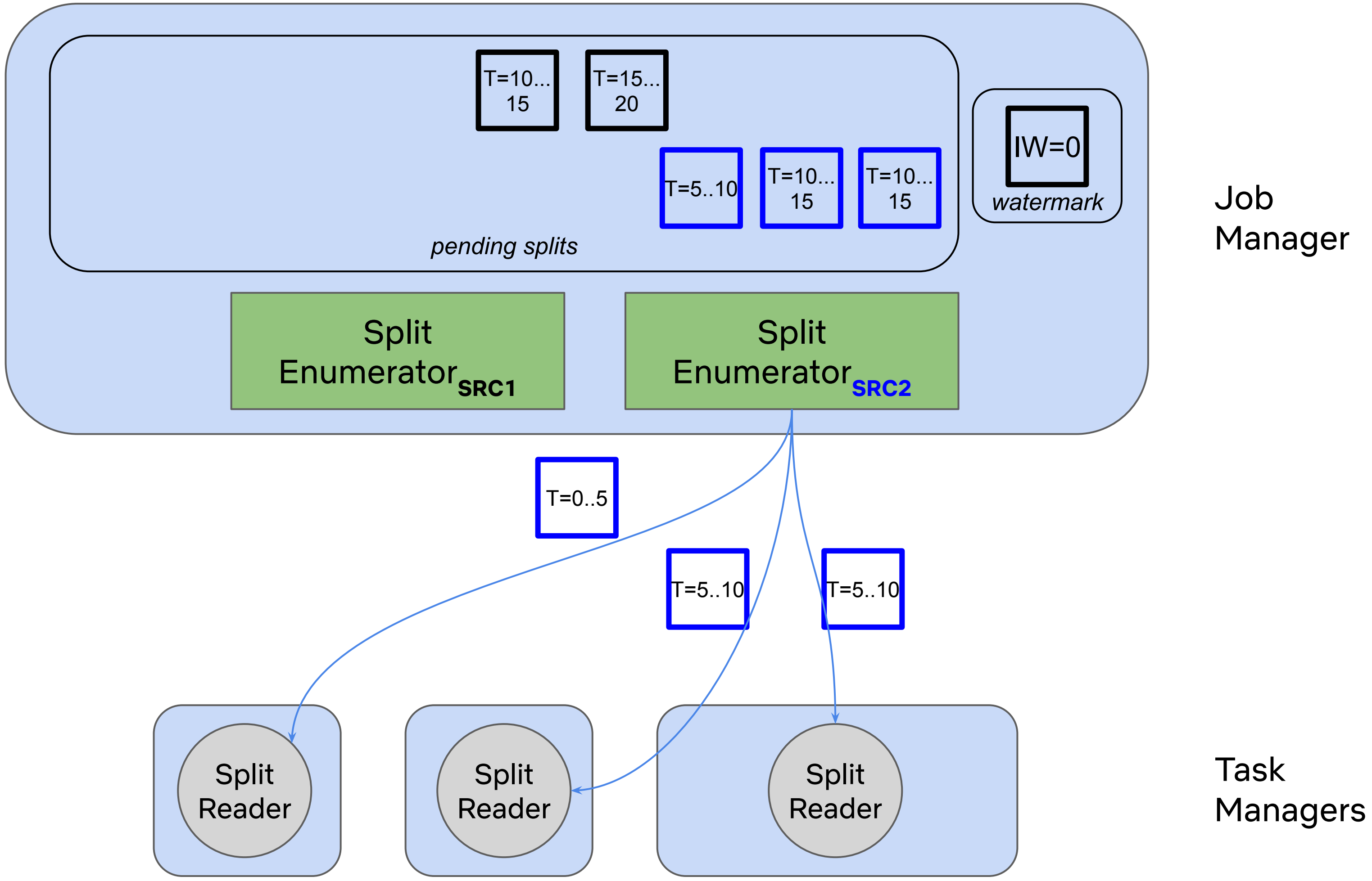


Assuming lateness of “10” minutes is okay.

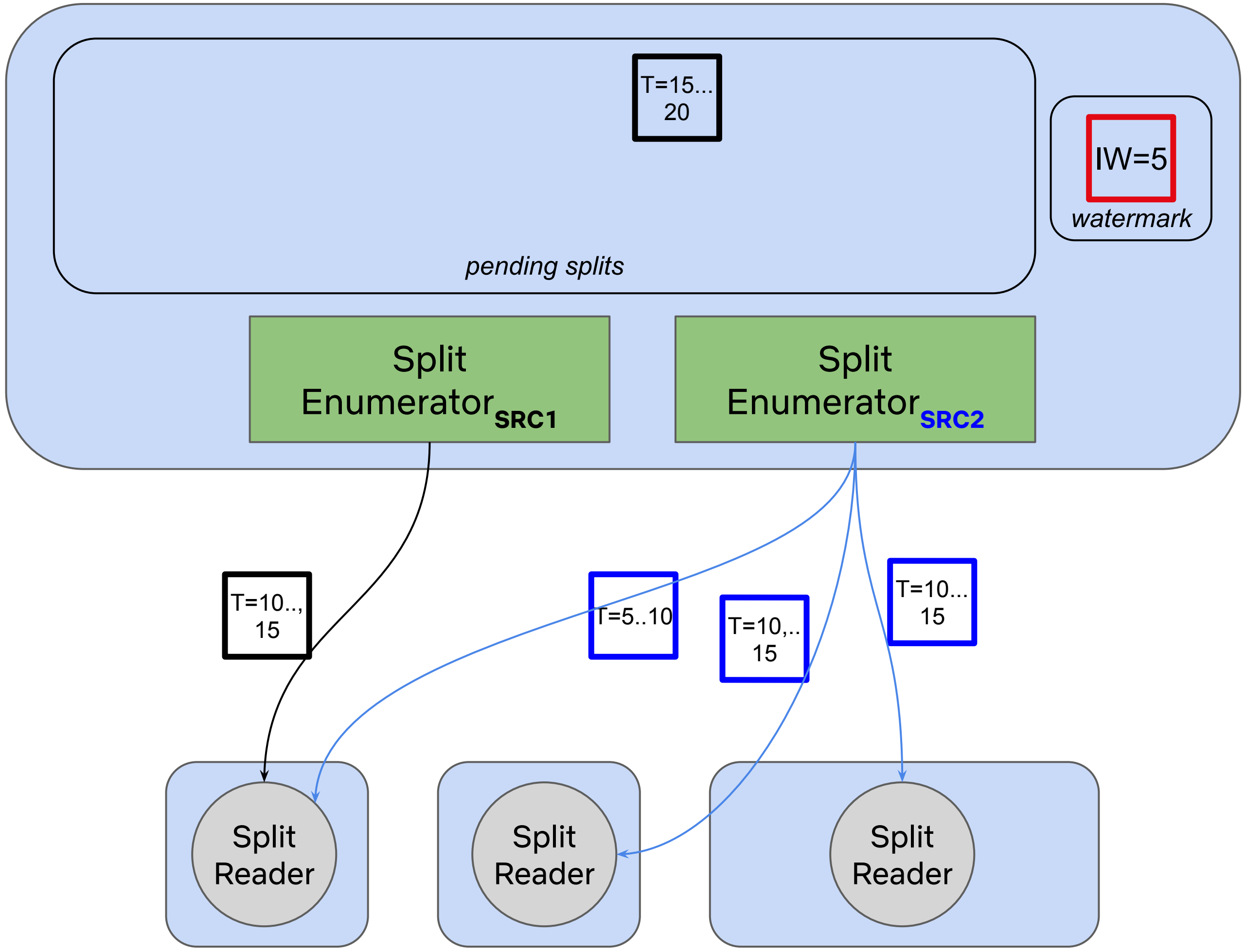
Dealing with multiple sources



Dealing with multiple sources



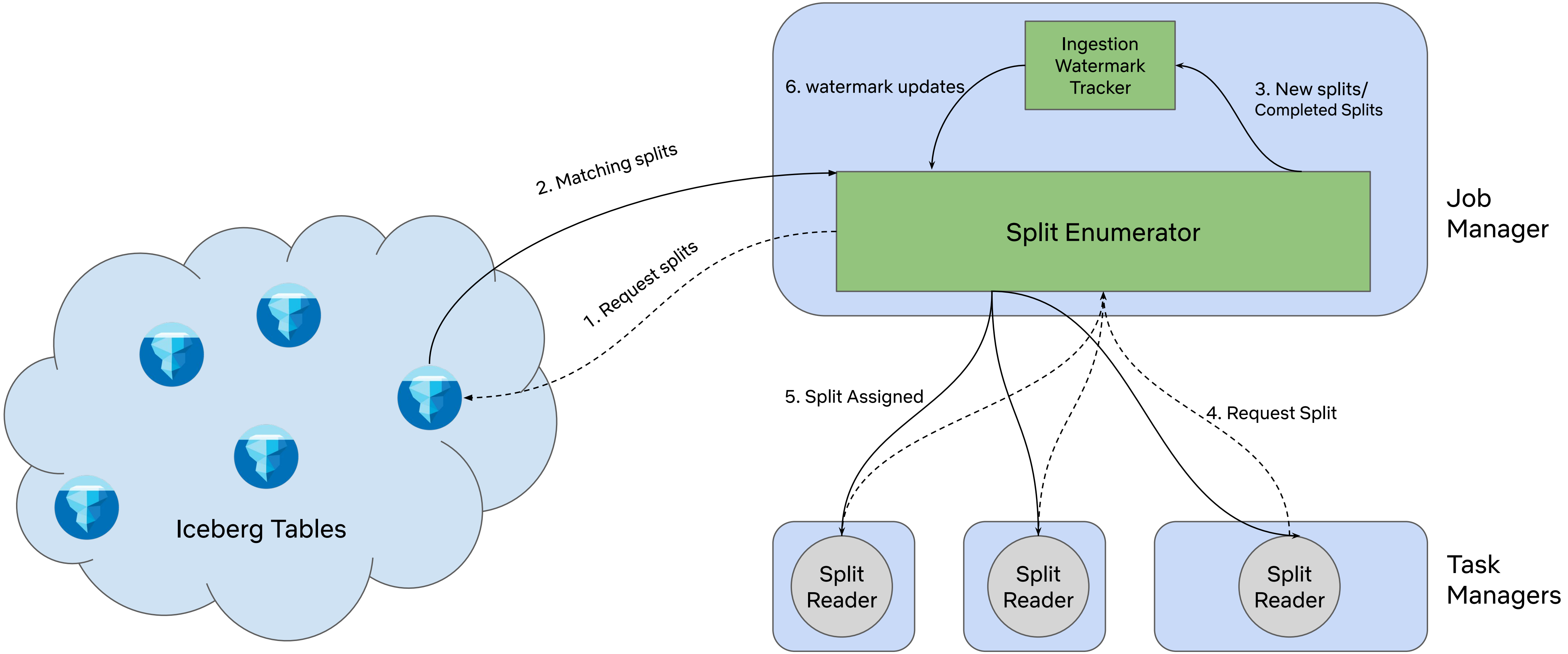
Dealing with multiple sources



Job
Manager

Task
Managers

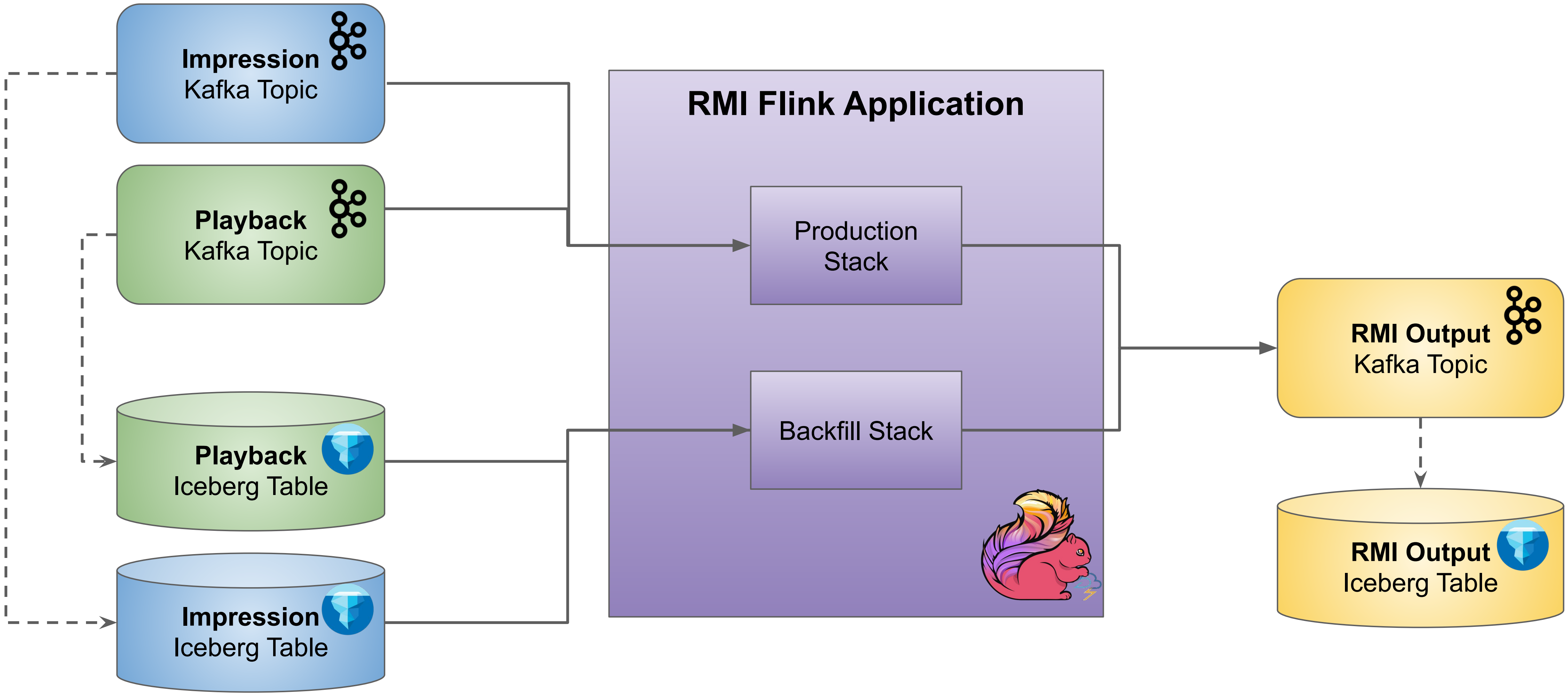
Iceberg Source Overview



Agenda

- ➤ Needs for backfilling Flink Applications
- ➤ Existing approaches
- ➤ Iceberg Source
- ➤ Event ordering challenges
- ➤ **Enabling Iceberg backfill**

Backfill RMI



Setting up a Flink Application for Backfilling: Example

```
@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])







}
```

Setting up a Flink Application for Backfilling: Example

```
@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])
}
```

Setting up a Flink Application for Backfilling: Example

```
@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.

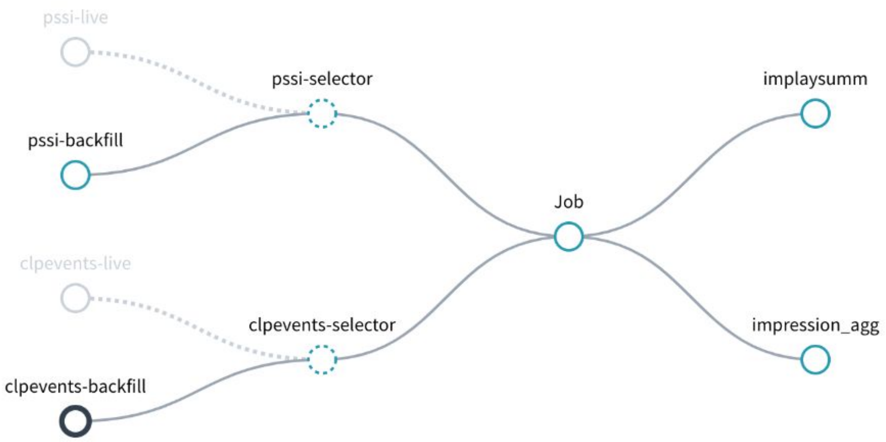
Setting up a Flink Application for Backfilling: Example

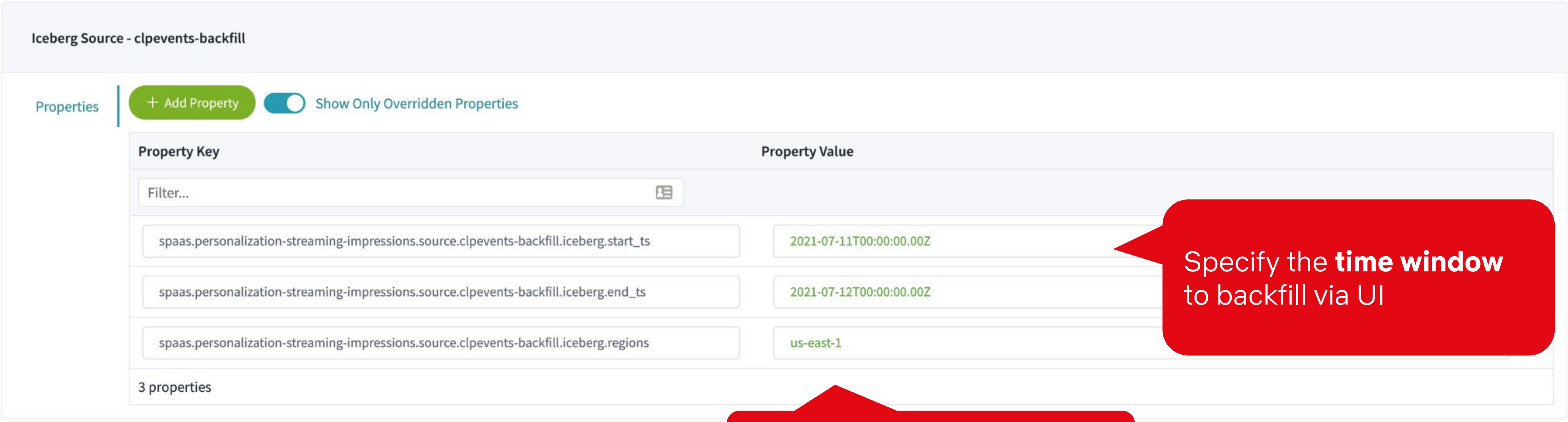
```yaml
nfflink:
  job.name: rmi-app
  connectors:
    sources:
      impression-source:
        type: dynamic
        selected: live-impression-source
        candidates:
        - live-impression-source
        - backfill-impression-source
      live-impression-source:
        type: kafka
        topics: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

Config changes to support backfilling.

Backfill RMI

Easily switch between **Kafka** and **Iceberg** sources via UI





Specify the **time window** to backfill via UI

Choose one or more **regions**

Backfill RMI

Results

- Processing 24 hours of data takes ~ 5 hours
- Backfill output matches 99.9% with Prod

Lessons Learned

- Backfilling window depends on Flink logic
- Set *max_misalignment_threshold* based on event ordering requirements
- Backfilling job configs need tuning (separately from prod job)

Benefits of Iceberg Source

- 👏 Use the same Flink app for backfilling
- 👏 Easy to set up
- 👏 Backfill large historical data quickly
- 👏 Cost Efficient ($2M/yr in Iceberg v.s $93M/yr in Kafka)

Future Work

- Provide support for continuously Streaming Iceberg Source for applications that do not require < second latency.

- Hybrid Streaming - Batch Source [FLIP-150] to bootstrap applications with historical data and continue with streaming.

- Strict Kafkaesque ordering for CDC apps

Thank You.

Contacts

Sundaram Ananthanarayanan (Linkedin)

Xinran Waibel (Linkedin)