04: Time & Windows

CONFLUENT
Global Education

Module Overview

This module contains five lessons:

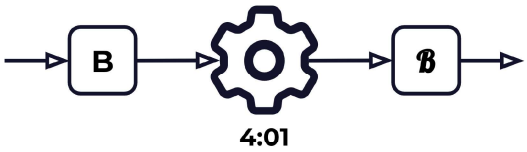
- Event Time vs. Processing Time
- Time Attributes vs. Timestamps
- Watermarks
- Windows
- Time Functions & Data Types

04a: Event Time vs. Processing Time

Description

Differences between Event Time and Processing Time.

Event Time vs. Processing Time

Event Time	Processing Time
Timestamp embedded within the records before they enter Flink	System time of the machine (wall-clock) that's executing the operation when the record happens to be processed
	

• Generally preferred
• Consistent (deterministic results)
  ◦ CreateTime
  ◦ LogAppendTime
  ◦ Any Timestamp Column in the payload
	• Simpler to work with
	• Non-deterministic
	• Currently, disabled in Confluent Cloud

A critical aspect in stream processing is the notion of time, and how it is modeled and integrated. When working with stream processing it is important to understand the concept of time, for example, some operations like windowing are defined based on time boundaries. It often helps to agree on specifying time information in UTC or in Unix time (such as seconds since the epoch). You should also not mix topics with different time semantics.

Event Time For example, if the event is a geolocation change reported by a GPS sensor in a car, then the associated event-time would be the time when the GPS sensor captured the location change.

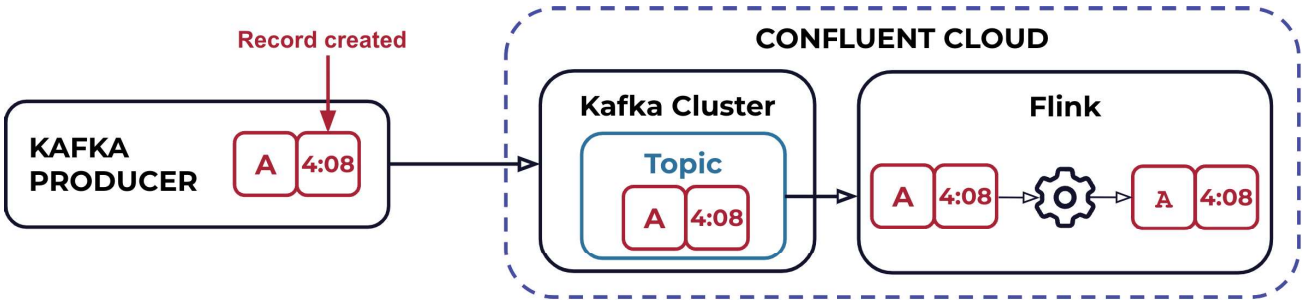
Ingestion Time Ingestion-time is similar to event-time, as a timestamp gets embedded in the data record itself. The difference is that the timestamp is generated when the record is appended to the target topic by the Kafka broker, not when the record is created at the source. This may be a reasonable alternative for use cases where event-time semantics is not possible, perhaps because the data producers don't embed timestamps (such as with older versions of Kafka's Java producer client) or the producer cannot assign timestamps directly (for example, when the producer does not have access to a local clock).

Processing Time The processing-time may be milliseconds, hours, days, etc. later than the original event-time.

Event Time - CreateTime

CreateTime

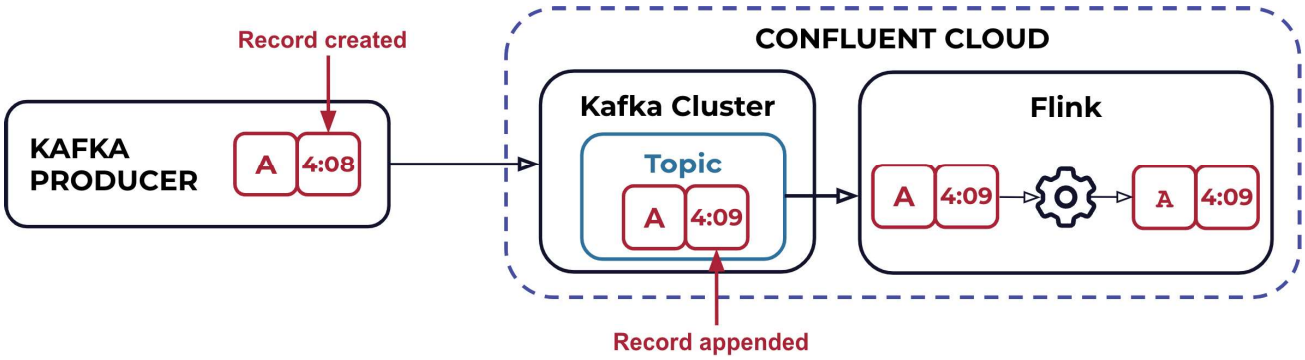
- Timestamp when a record was created
- Default configuration



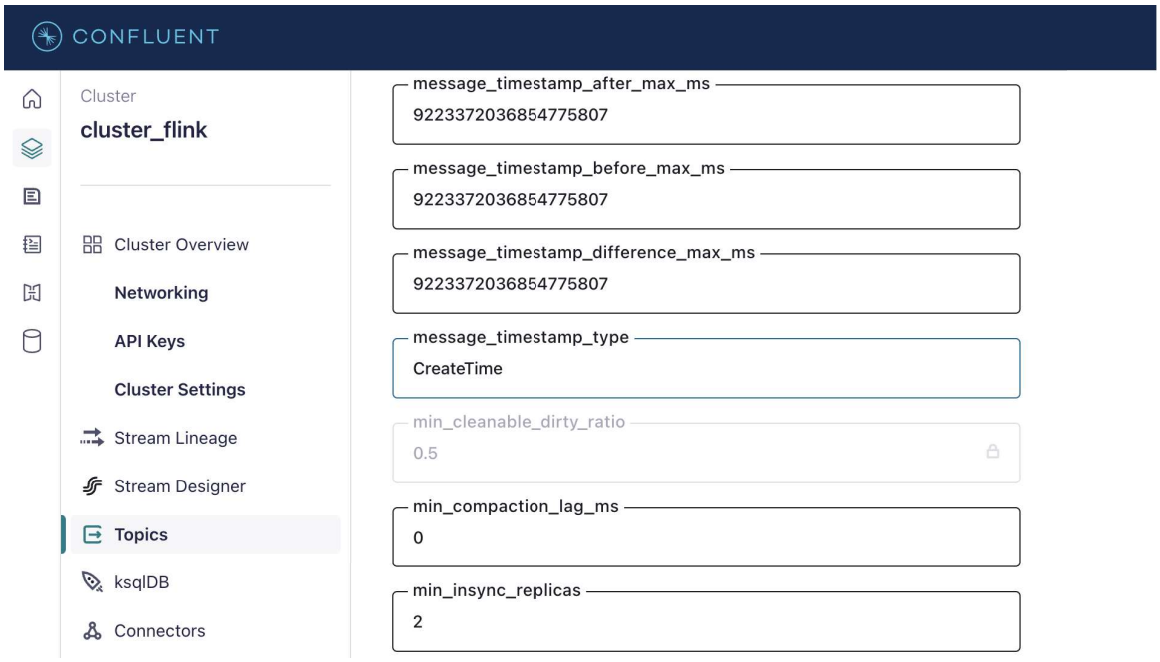
Event Time - LogAppendTime

LogAppendTime

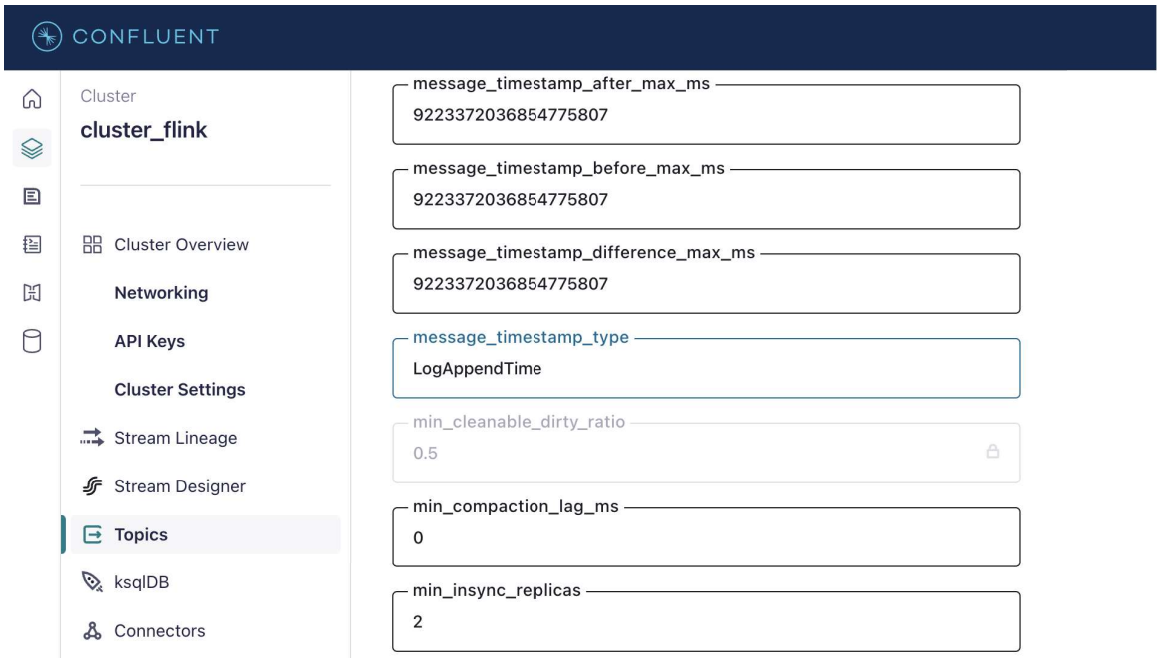
- Timestamp when a record is stored in a topic partition by a Kafka broker
- Configurable per topic in Confluent Cloud in "expert mode"



Config CreateTime in Confluent Cloud

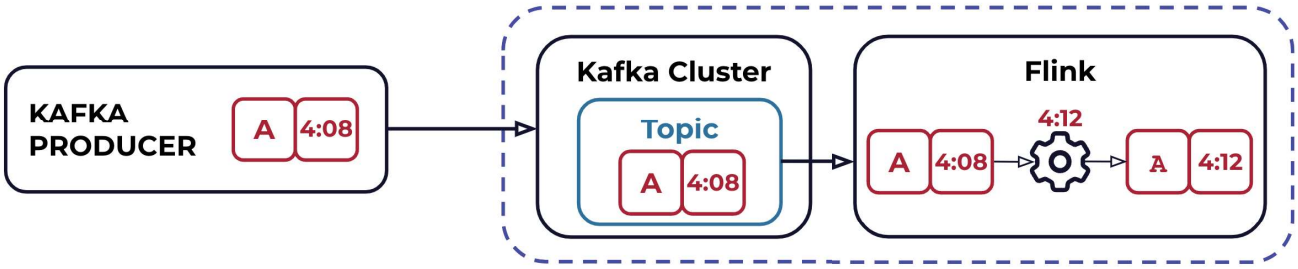


Config LogAppendTime in Confluent Cloud



Processing Time

- System time of the machine (wall-clock) that's executing the operation when the record happens to be processed
- It gives non-deterministic results
- Currently disabled in Confluent Cloud



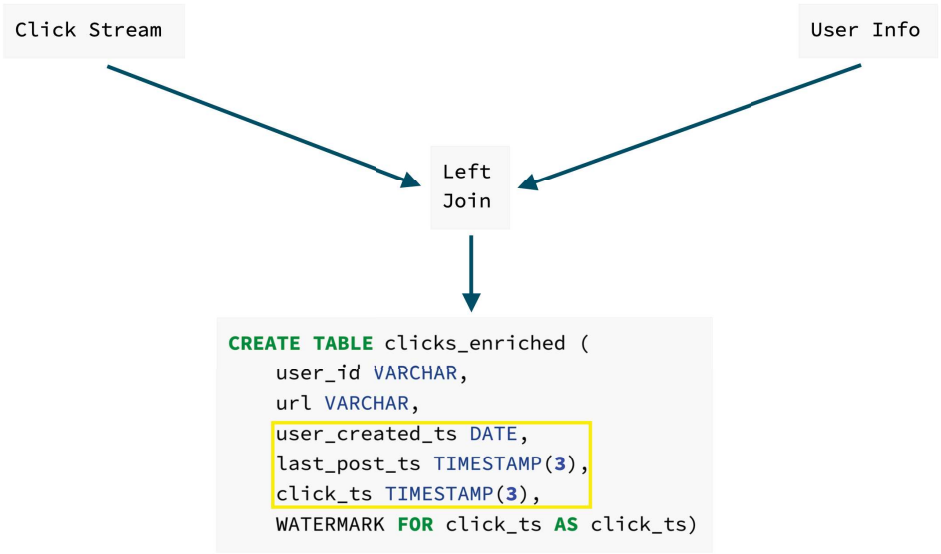
Lesson 04b

04b: Time Attributes vs. Timestamps

Description

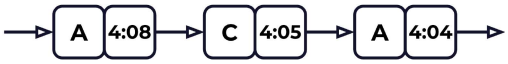
Differences between Time Attributes and Timestamps.

Click Stream Enrichment - Use Case



Time Attributes vs. Timestamps

- Used in temporal operations (Windows, Joins, Pattern Matching)
- Connected to the forward progress of time:
  - **Processing Time** → *system wall-clock time*
  - **Event Time** → *time connected to the watermarks*
- Must be a column type:
  - TIMESTAMP(3)
  - TIMESTAMP_LTZ(3)




```
CREATE TABLE clicks_enriched (
    user_id VARCHAR,
    url VARCHAR,
    user_created_ts DATE,
    last_post_ts TIMESTAMP(3),
    click_ts TIMESTAMP(3),
    WATERMARK FOR click_ts AS click_ts)
```

A time attribute must be of type `TIMESTAMP(p)` or `TIMESTAMP_LTZ(p)`, with `0 <= p <= 3`

The time attribute must be connected to a column that represent the progress of time in the timeline. It must uniquely indicate where the message stays on the timeline.

Define Time Attributes

Connected to the forward progress of time:

- **Processing Time** → *system wall-clock time*
- **Event Time** → *time connected to the watermarks*

Processing Time	Event Time
`CREATE TABLE clicks_enriched ( user_id VARCHAR, url VARCHAR, user_created_ts DATE, last_post_ts TIMESTAMP(3), click_ts AS PROCTIME() )`	`CREATE TABLE clicks_enriched ( user_id VARCHAR, url VARCHAR, user_created_ts DATE, last_post_ts TIMESTAMP(3), click_ts TIMESTAMP(3) METADATA FROM 'timestamp', WATERMARK FOR click_ts AS click_ts)`

Time Attributes in Confluent Cloud

Connected to the forward progress of time:

- **Processing Time** → *system wall-clock time*
- **Event Time** → *time connected to the watermarks*

Processing Time

```
CREATE TABLE clicks_enriched (
    user_id VARCHAR,
    url VARCHAR,
    user_created_ts DATE,
    last_post_ts TIMESTAMP(3),
    click_ts AS PROCTIME() )
```

Not supported yet

Event Time

```
CREATE TABLE clicks_enriched (
    user_id VARCHAR,
    url VARCHAR,
    user_created_ts DATE,
    last_post_ts TIMESTAMP(3),
    click_ts TIMESTAMP(3) METADATA FROM `timestamp` ,
    WATERMARK FOR click_ts AS click_ts)
```

Not needed

Time Attributes in Confluent Cloud

Connected to the forward progress of time:

- **Processing Time** → *system wall-clock time*
- **Event Time** → *time connected to the watermarks*

Processing Time	Event Time
~~`CREATE TABLE clicks_enriched ( user_id VARCHAR, url VARCHAR, user_created_ts DATE, last_post_ts TIMESTAMP(3), click_ts AS PROCTIME() )`~~ **Not supported yet**	`CREATE TABLE clicks_enriched ( user_id VARCHAR, url VARCHAR, user_created_ts DATE, last_post_ts TIMESTAMP(3) )` Confluent Cloud automatically creates a System column called `$rowtime` with the internal timestamp of the record

Time Attributes vs. **Timestamps**

- **NOT** used for temporal operations (Windows, Joins, Pattern Matching)
- No guarantee that the timestamp column is ordered by time
- Can be any time column type :
  - TIMESTAMP
  - TIMESTAMP_LTZ
  - TIME
  - DATE
  - ...

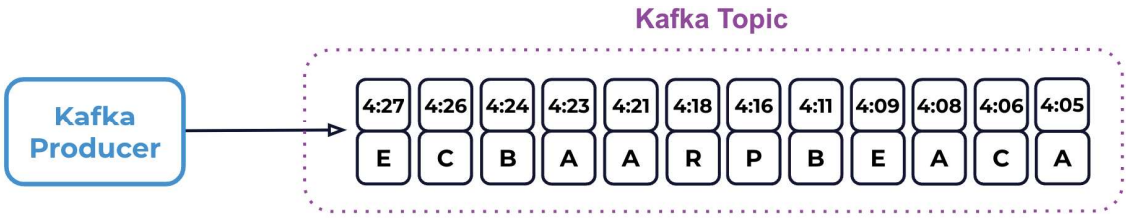

```
CREATE TABLE clicks_enriched (
    user_id VARCHAR,
    url VARCHAR,
    user_created_ts DATE,
    last_post_ts TIMESTAMP(3),
    click_ts TIMESTAMP(3),
    WATERMARK FOR click_ts AS click_ts)
```

04c: Watermarks

Description

Understand through examples how watermarks work in Flink.

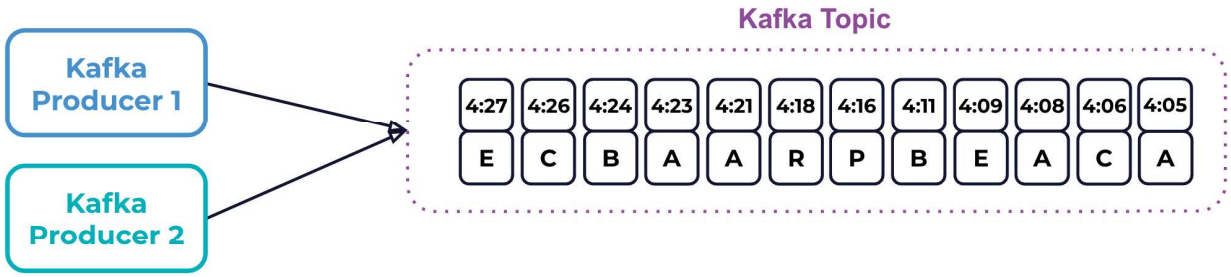
Why are Watermarks important to Event Time operations?



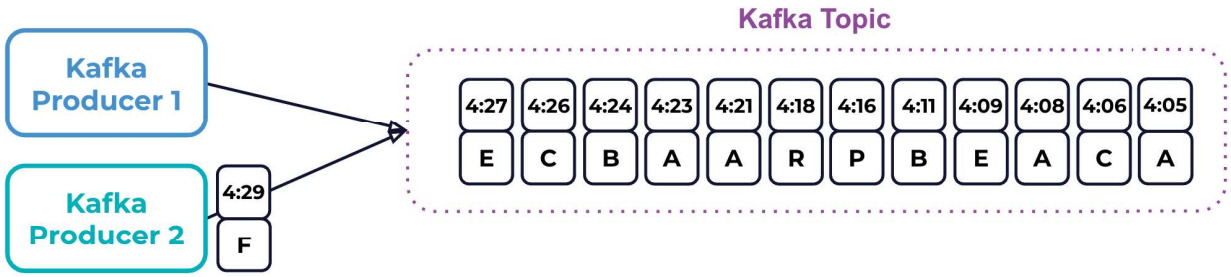
- Here, messages in the topic are perfectly ordered by time
- There are NOT out-of-order messages

When working in event time, it is possible for event streams to be perfectly in order by time.

Why are Watermarks important to Event Time operations?

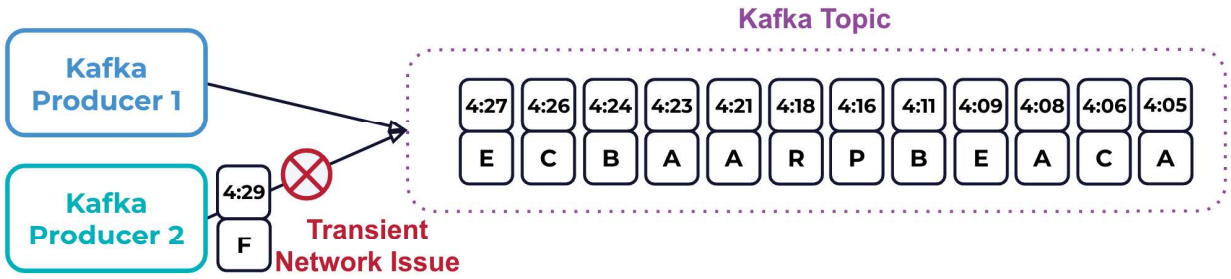


Why are Watermarks important to Event Time operations?



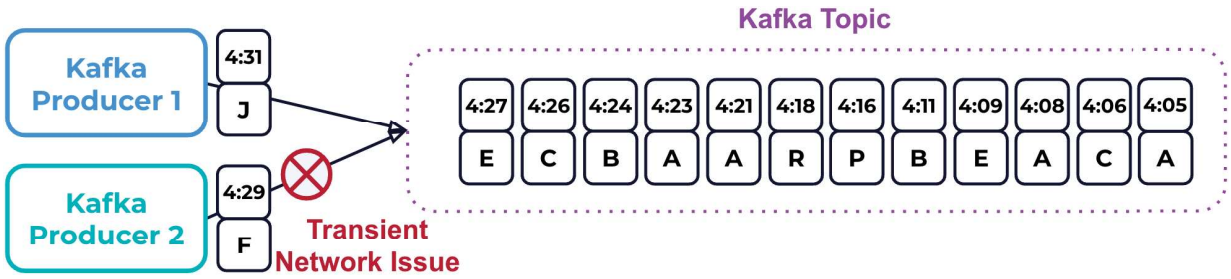
Producer 2 creates a new record

Why are Watermarks important to Event Time operations?

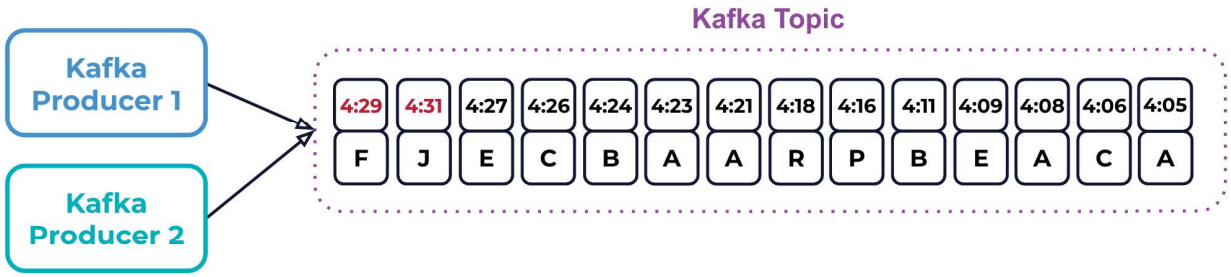


Why are Watermarks important to Event Time operations?

Producer 1 creates a new record



Why are Watermarks important to Event Time operations?



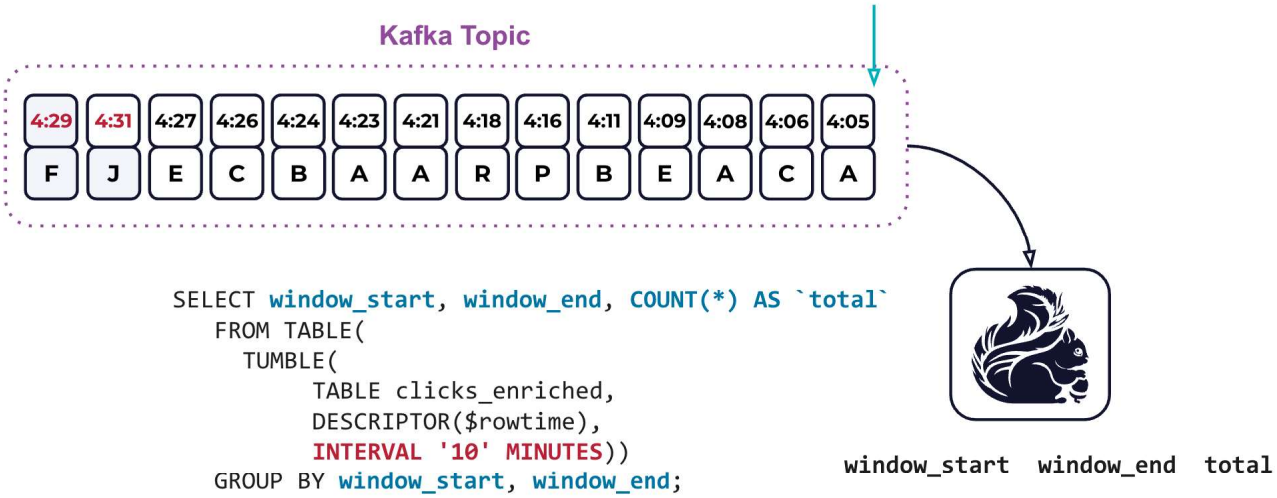
- There are Out-of-order Messages
- Streams are (roughly) ordered by time

But very often event streams are only approximately ordered by their timestamps.

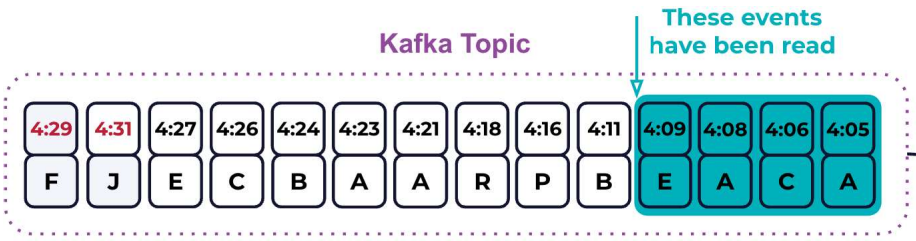
This can happen whenever events can take different paths, and experience different delays.

This is especially common in situations involving mobile devices where an event that occurred a few seconds later ends up being processed first because it experiences less delay than an earlier event.

Why are Watermarks important to Event Time operations?



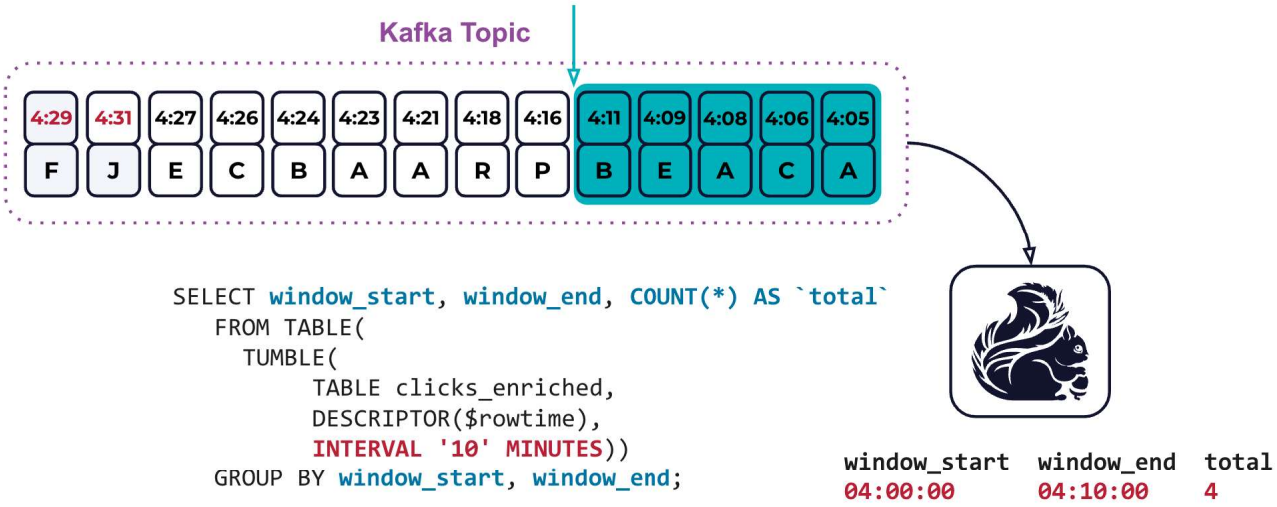
Why are Watermarks important to Event Time operations?



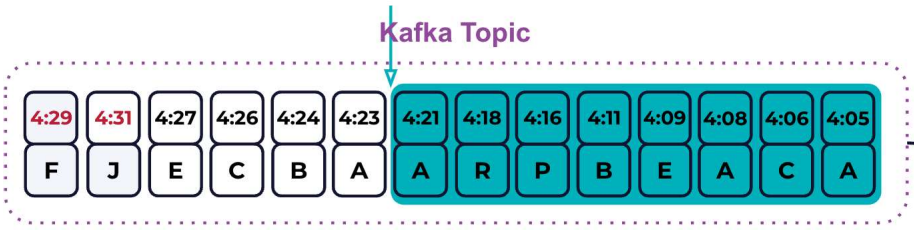

```
SELECT window_start, window_end, COUNT(*) AS `total`
   FROM TABLE(
     TUMBLE(
          TABLE clicks_enriched,
          DESCRIPTOR($rowtime),
          INTERVAL '10' MINUTES))
   GROUP BY window_start, window_end;
```

```
window_start   window_end   total
```

Why are Watermarks important to Event Time operations?



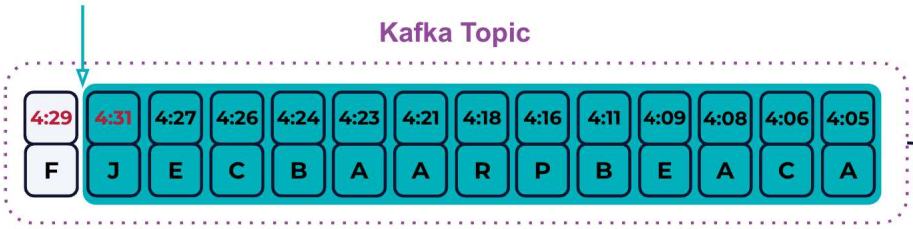
Why are Watermarks important to Event Time operations?




```
SELECT window_start, window_end, COUNT(*) AS `total`
   FROM TABLE(
     TUMBLE(
          TABLE clicks_enriched,
          DESCRIPTOR($rowtime),
          INTERVAL '10' MINUTES))
   GROUP BY window_start, window_end;
```

window_start	window_end	total
04:00:00	04:10:00	4
04:10:00	04:20:00	3

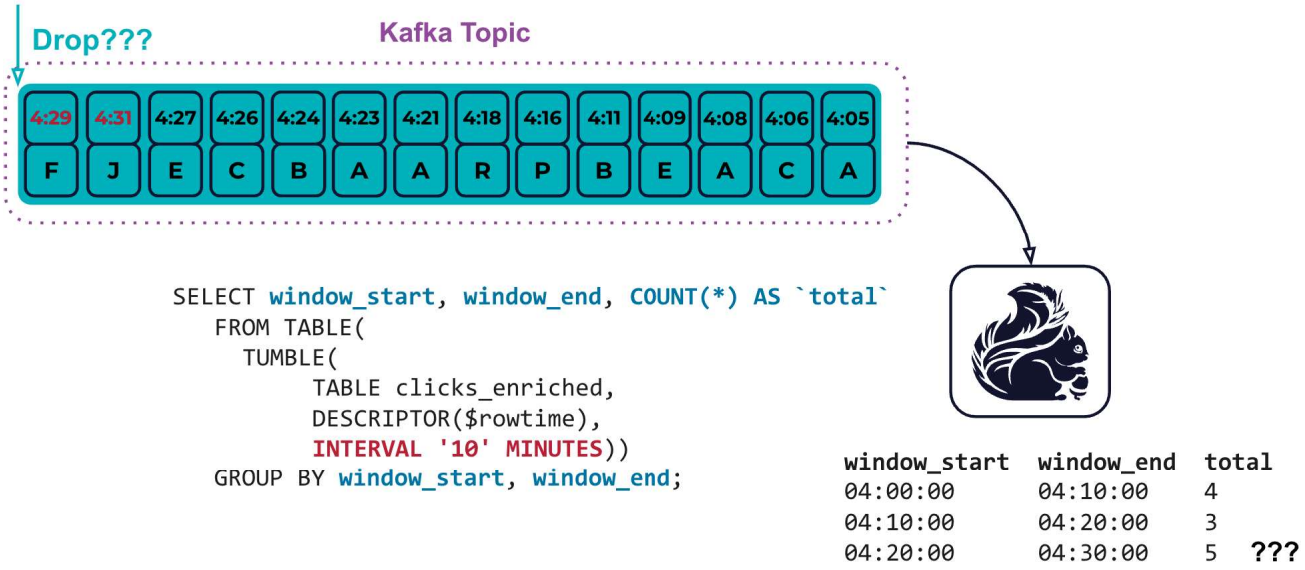
Why are Watermarks important to Event Time operations?




```
SELECT window_start, window_end, COUNT(*) AS `total`
   FROM TABLE(
     TUMBLE(
          TABLE clicks_enriched,
          DESCRIPTOR($rowtime),
          INTERVAL '10' MINUTES))
   GROUP BY window_start, window_end;
```

window_start	window_end	total
04:00:00	04:10:00	4
04:10:00	04:20:00	3
04:20:00	04:30:00	5

Why are Watermarks important to Event Time operations?



Watermarks define when it's safe to produce the final count.

- How should this window decide that it has seen every event that belongs to those 10 minutes?
- It's not like the window can peek ahead in the stream and know for sure that it's safe to produce the final count for the 10-minute window ending at 4:30
- So we need to implement some heuristic that the window can use to determine when the window is probably complete, and that no further events from before 4:30 are likely to exist

Watermarks

Watermarks establish when it's safe to produce the final count

```
WATERMARK = max_timestamp - out_of_orderness
```

`max_timestamp`: largest timestamp seen so far
`out_of_orderness`: maximum arrival delay

These Watermark timestamps are computed by subtracting the out-of-orderness estimate from the largest timestamp seen so far.

By default, every 200 milliseconds the Kafka source is going to generate a watermark, and send it downstream. Watermarks are actual internal messages sent through the pipeline.

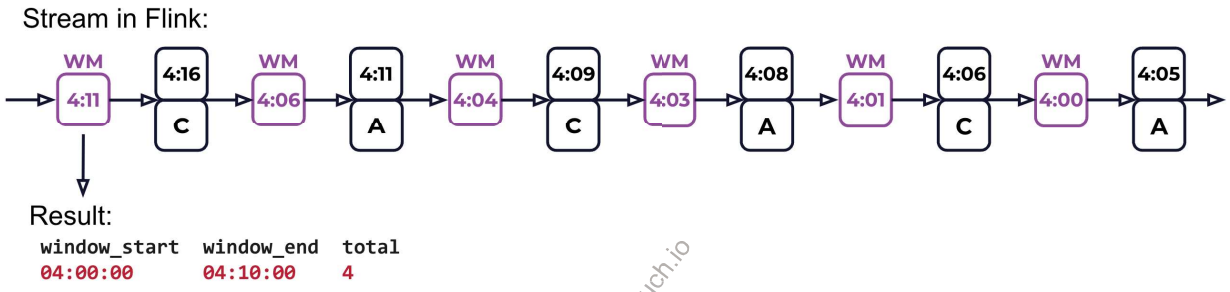
The built-in function, `CURRENT_WATERMARK`, enables printing the current watermark for the executing operator.

Watermarks - Example

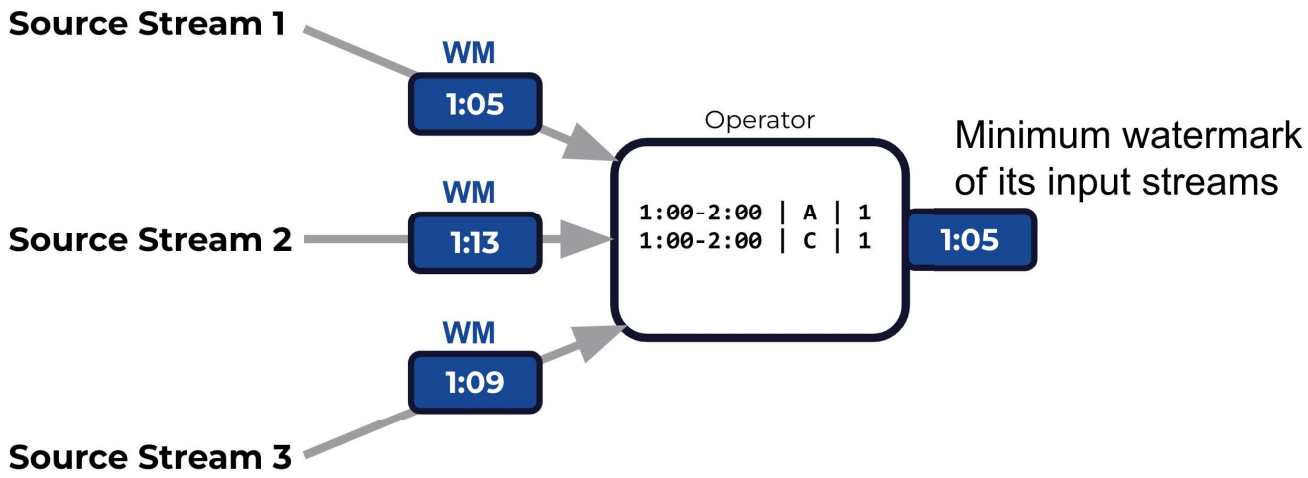
Input Stream:

Expected out-of-orderness: `5 minutes`

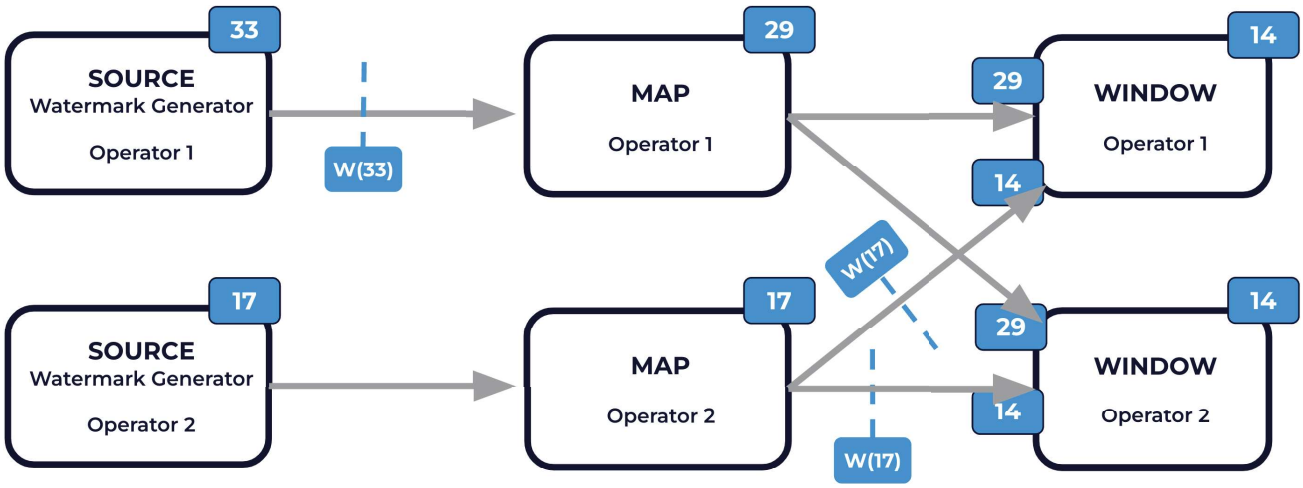
```
WATERMARK = max_timestamp - 5 minutes
```



Watermarks in Multi-Input Operators

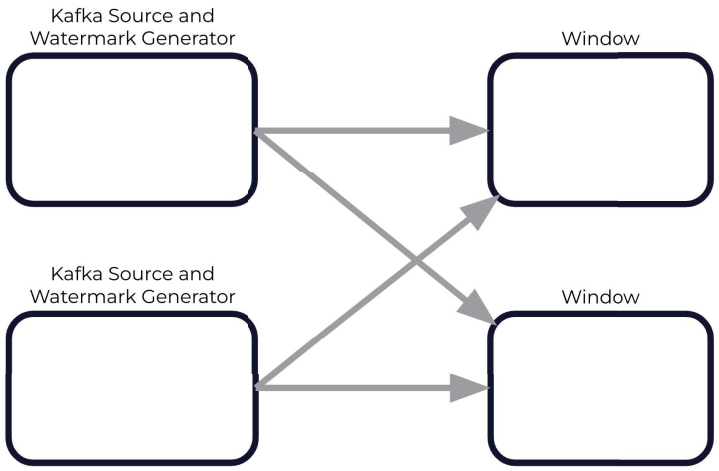


Watermarks in Parallel Streams



* watermarks are travelling along the topology

Watermarks from a Topic with Multiple Partitions

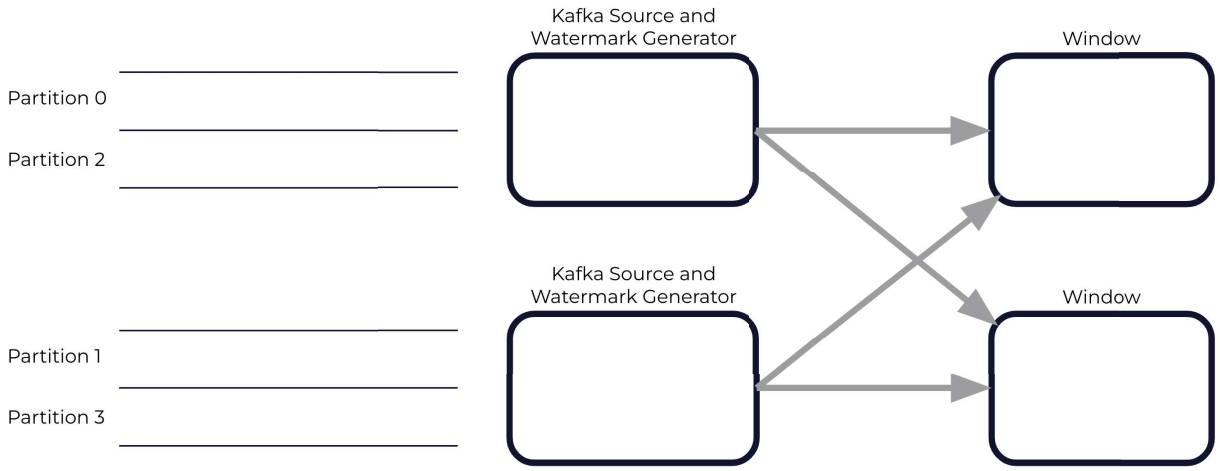


Flink's window operator relies on Watermarks to know when to produce results. These watermarks come from a watermark generator that runs inside of Flink's Kafka consumer. This example will get a bit complex, but these details are important because if the watermarks aren't working correctly, the application may fail to produce any results.

This example is a kafka source, followed by a window. Both the source and the window have a parallelism of two. And the window is counting events by LETTER, which is why the sources are connected to the windows by a network shuffle that implements this re-partitioning.

For clarification: The message value consists of single letters, so Flink needs to repartition events to properly group them by letter.

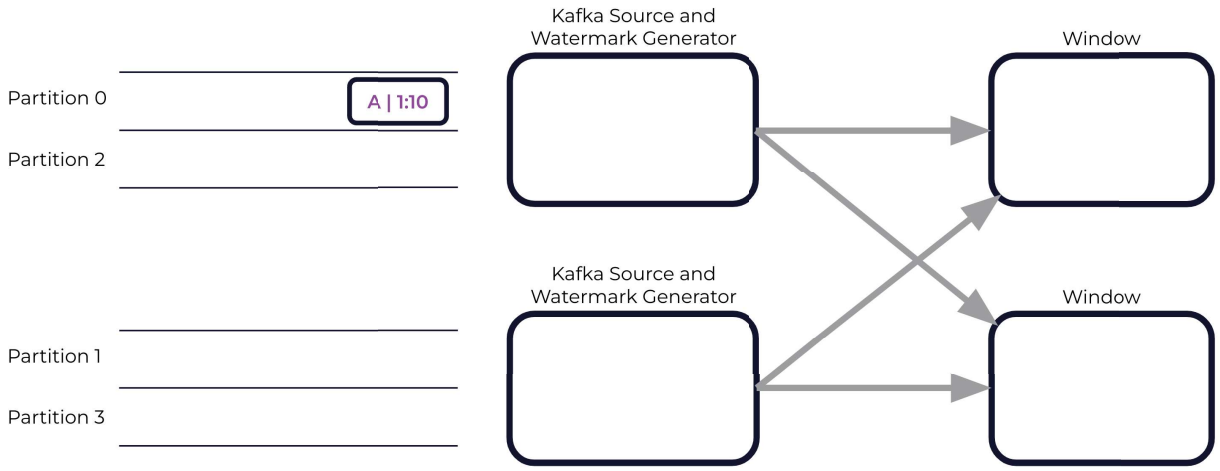
Watermarks from a Topic with Multiple Partitions



In this example, each instance of the Kafka source operator is reading from two kafka partitions

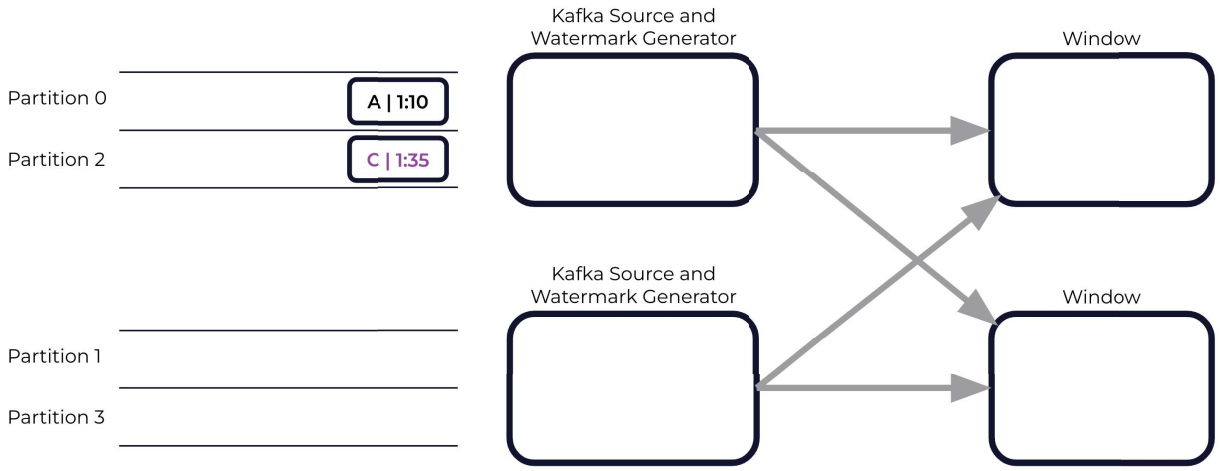
We need this much parallelism in our example to fully explore what happens when Flink generates, and propagates, watermarks in real applications

Watermarks from a Topic with Multiple Partitions



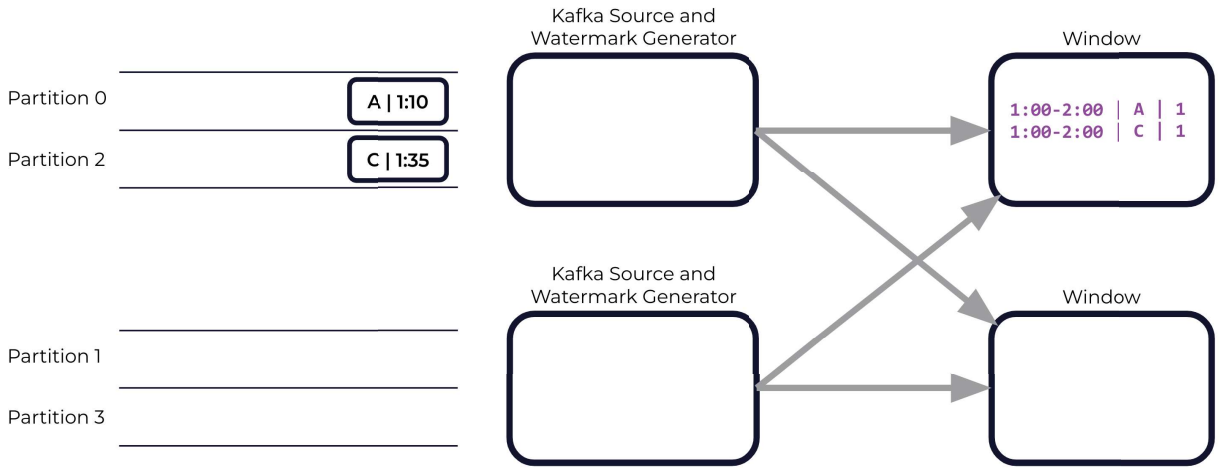
Here is a sample event, with a key of A and a timestamp at 10 minutes after 1 o'clock

Watermarks from a Topic with Multiple Partitions



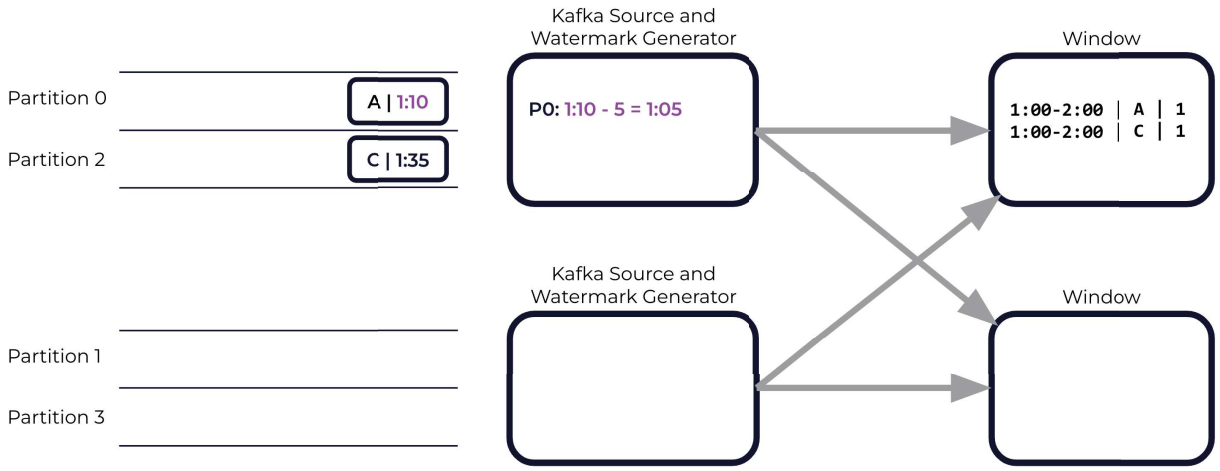
And here is another event on the other partition connected to this same source

Watermarks from a Topic with Multiple Partitions



After processing these two events, our window has recorded 1 event for A during the window from 1 to 2 o'clock, and one event for C

Watermarks from a Topic with Multiple Partitions

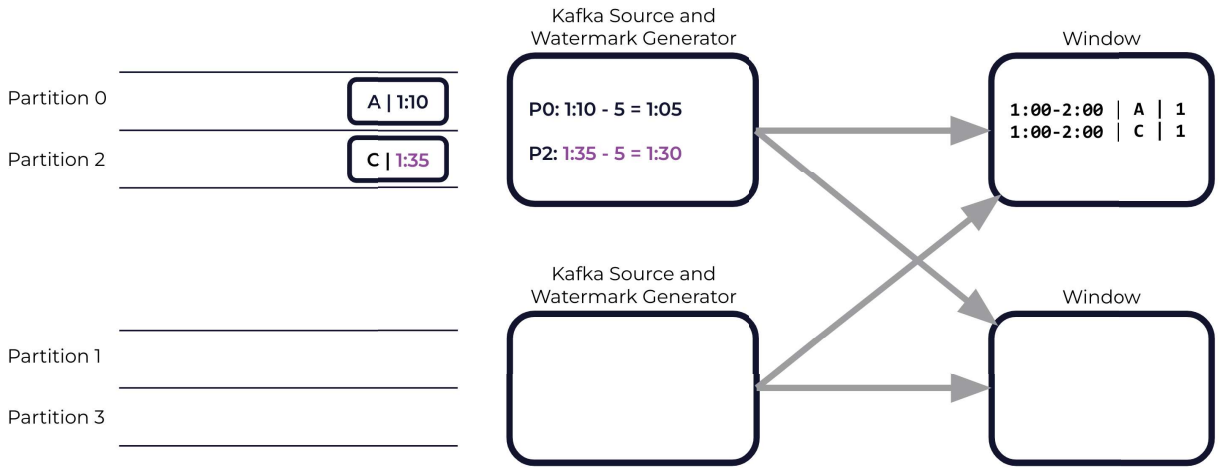


By default, **every 200 milliseconds** the Kafka source is going to generate a watermark, and send it downstream

It does this by first computing the watermark for each partition, independently

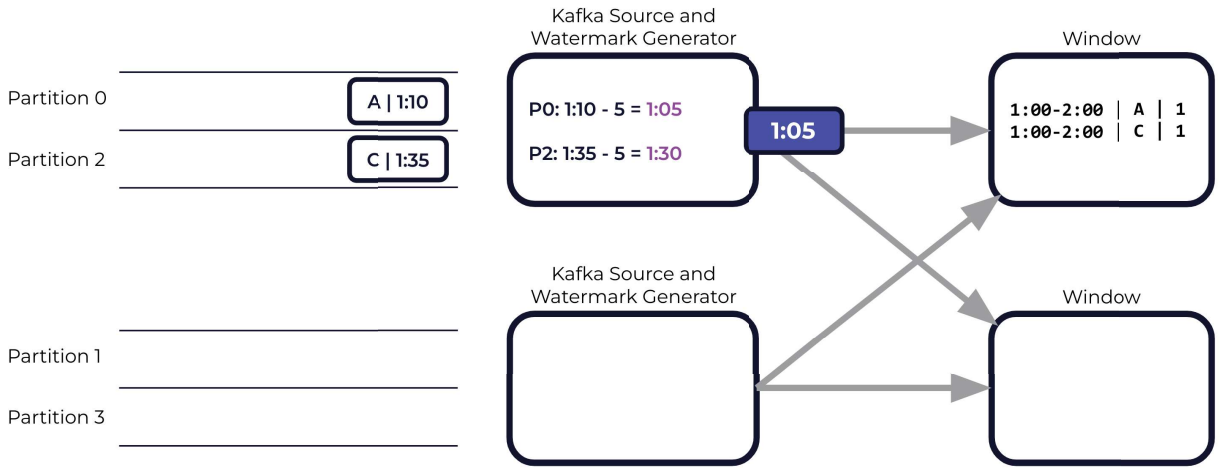
Using the same 5-minutes-out-of-order estimate we used before, this produces a watermark at 1:05 for partition zero, since the largest timestamp so far on partition zero is 1:10, and 1:10 minus 5 minutes is 1:05.

Watermarks from a Topic with Multiple Partitions



Using the same logic, the watermark for partition 2 is 1:30

Watermarks from a Topic with Multiple Partitions

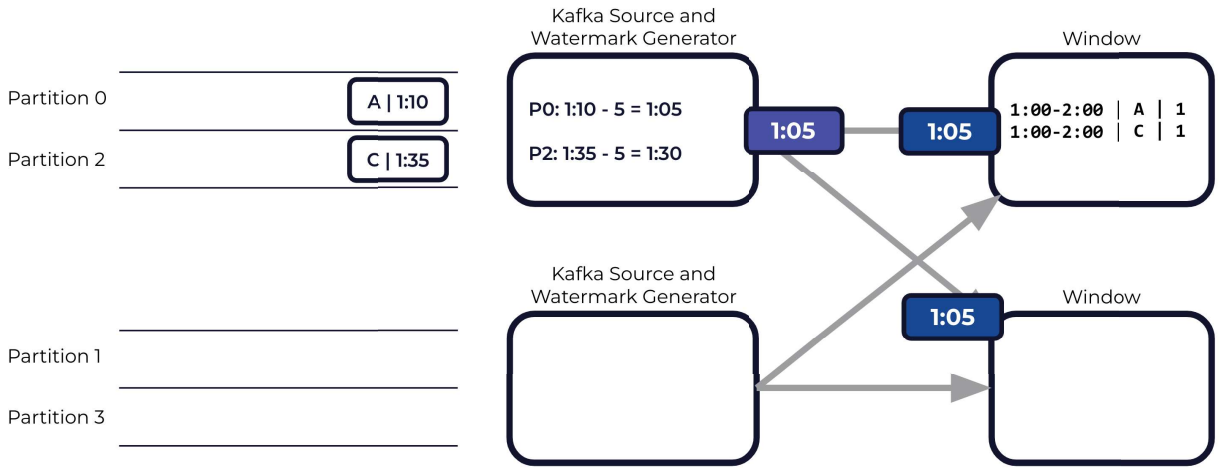


The watermark that the Kafka source produces is actually the minimum of these per-partition watermarks

Now why does this make sense?

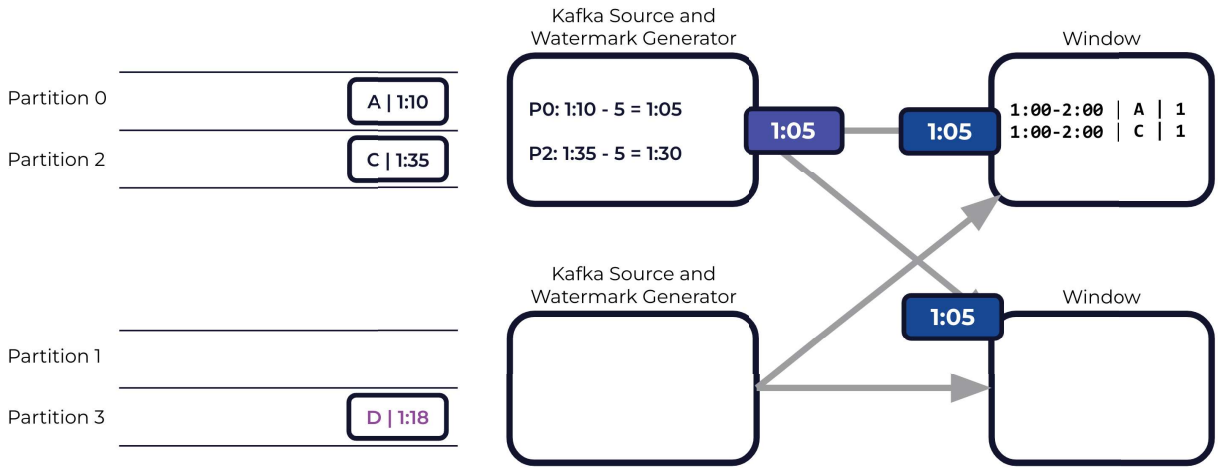
- The watermark this Kafka source produces should carry a timestamp that reflects how complete the stream is that it is producing
- This stream from uppermost kafka source includes events from both partition 0 and partition 2, so it can be no more complete than the furthest behind of these two partitions – which is partition 0
- And although partition 0 has seen an event with a timestamp of 1:10, it is reporting its watermark as being 1:05 because it is allowing for its events to be up to 5 minutes out-of-order

Watermarks from a Topic with Multiple Partitions



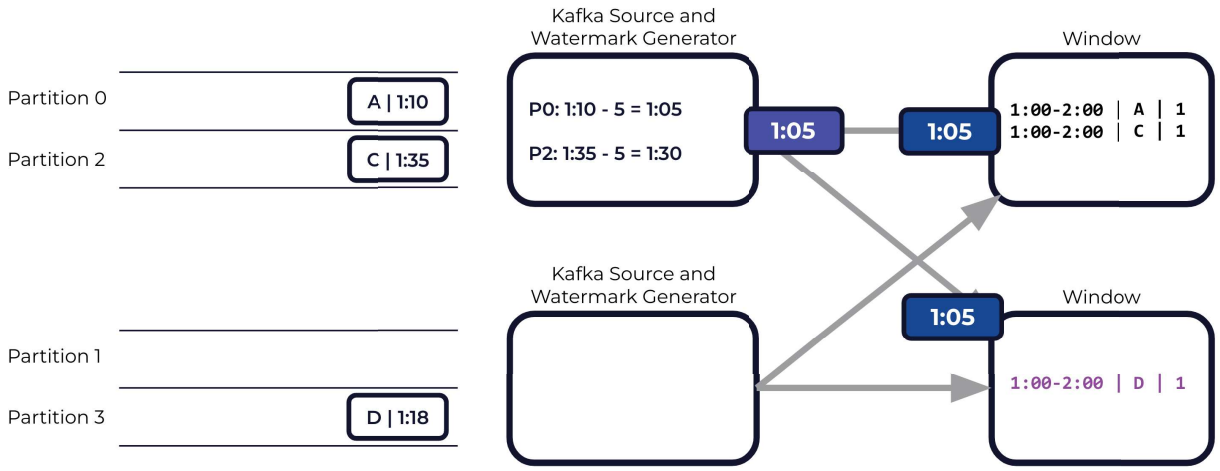
After generating this watermark, the source sends it downstream to both window operators

Watermarks from a Topic with Multiple Partitions

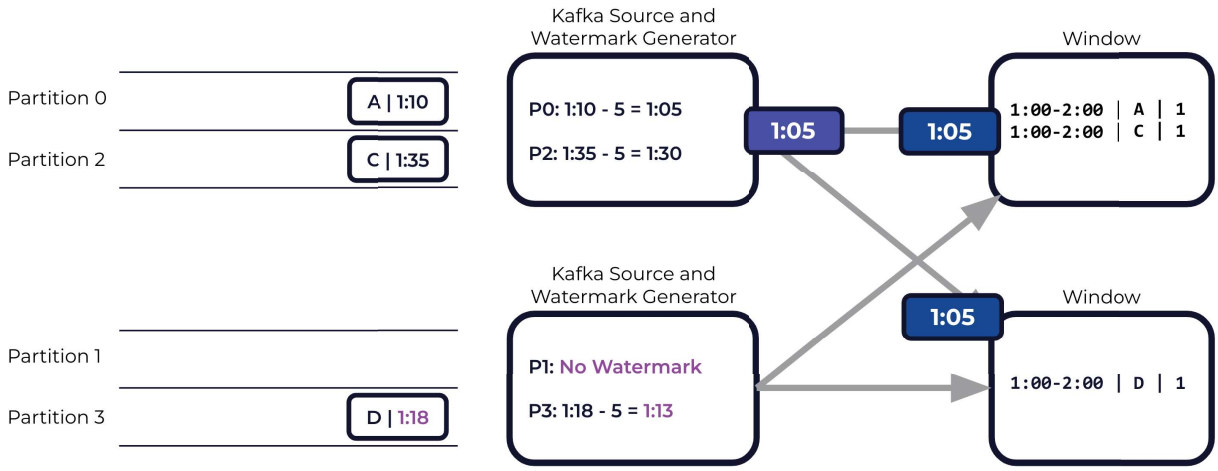


- We are building up toward understanding how Flink computes the watermark at each instance of the window operator.
- This will need to somehow reflect the watermarks coming into the windows from each of the sources.
- So far, all this source has consumes is an event at 1:18 from partition 3

Watermarks from a Topic with Multiple Partitions



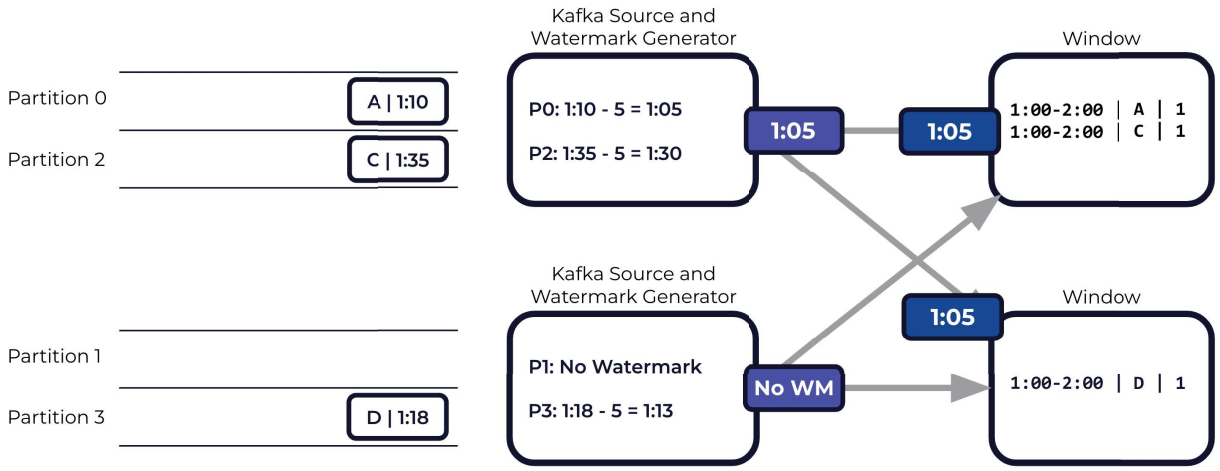
Watermarks from a Topic with Multiple Partitions



Based on this event, it has generated a watermark at 1:13 for partition 3

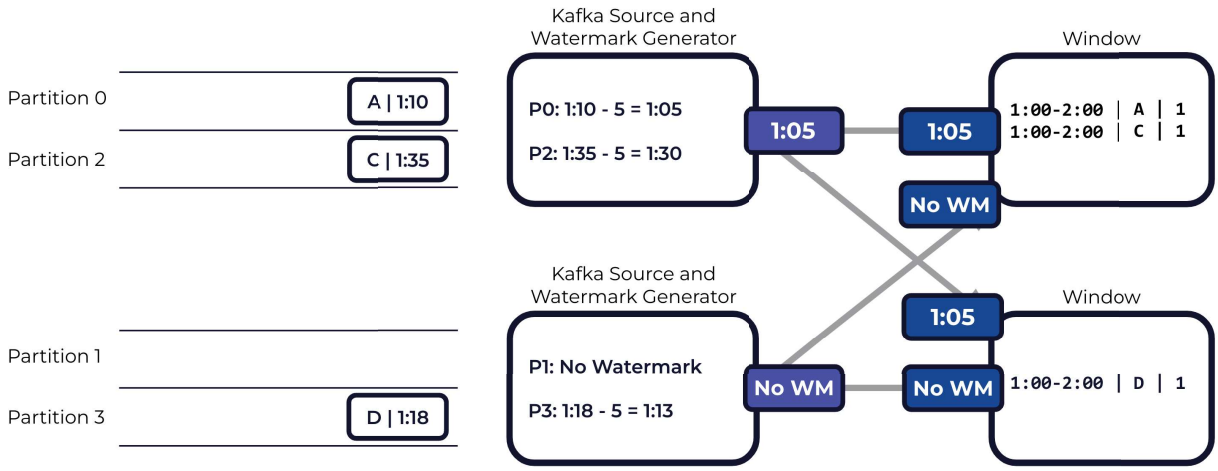
But lacking any events for partition 1, that partition doesn't have a valid watermark

Watermarks from a Topic with Multiple Partitions



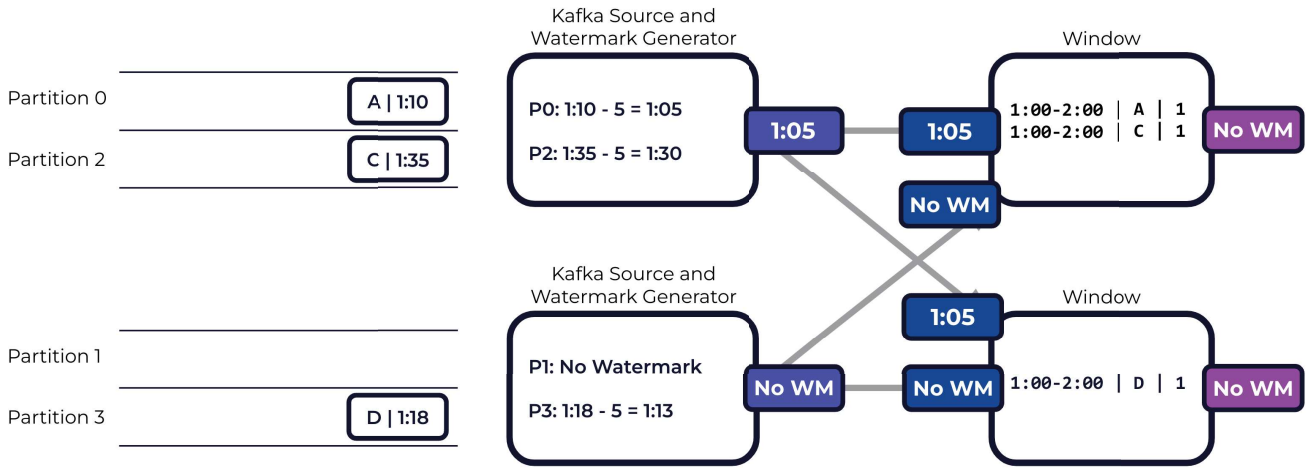
As a result, when asked for a watermark, the source will send this information downstream

Watermarks from a Topic with Multiple Partitions



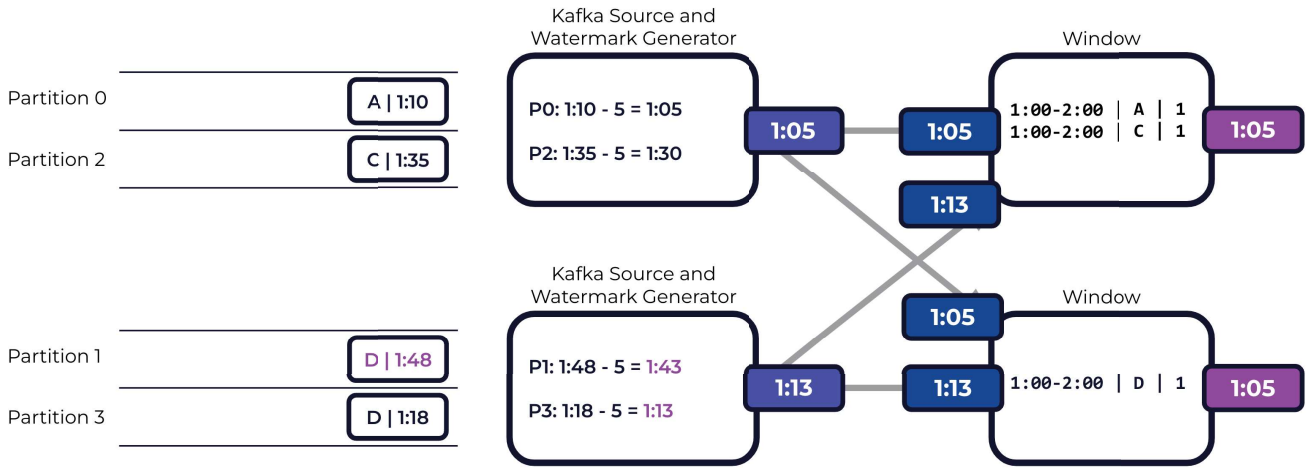
Now we are in a position to answer the question: What is the current watermark at the window?

Watermarks from a Topic with Multiple Partitions



- The answer is that the current watermark for any operator with multiple inputs is the minimum of the incoming watermarks.
- So in this case, the window operators don't yet have a valid watermark.
- This is because no events have been processed for partition 1.

Watermarks from a Topic with Multiple Partitions



- As soon as partition 1 has an event, the second source can produce a proper watermark, which will flow downstream.
- That watermark is 1:13, because that is the smaller of the two per-partition watermarks in this source.
- Now the watermarks for both instances of the window operator have advanced, but until these watermarks reaches 2:00, the results that are being accumulated inside the windows won't be emitted.

A few last details

- 5 minutes is a very long out-of-orderness interval
  - E.g. the window for 1:00-2:00 won't produce results until **2:05**
- The actual formula used for computing watermarks is:
  - `max_timestamp - out_of_orderness - 1ms`
  - ~~**1:05**~~ → **1:04:59.999**
- Technically speaking, there's always a watermark
  - If you look in the metrics, you may see -9223372036854775808
  - Rather than **No Watermark**, the implementation uses **-2^{63}**

Finally, there's always a watermark. Internally the watermarks are initialized to a very large negative value, which is what you'll see if you inspect the watermarks. So whenever you see a crazy watermark like this, understand that this means that the watermark generator hasn't seen any events yet, so the maximum timestamp is still this very very small value.

Lesson 04d

04d: Windows

Description

Explaining the different time windows available in Flink SQL

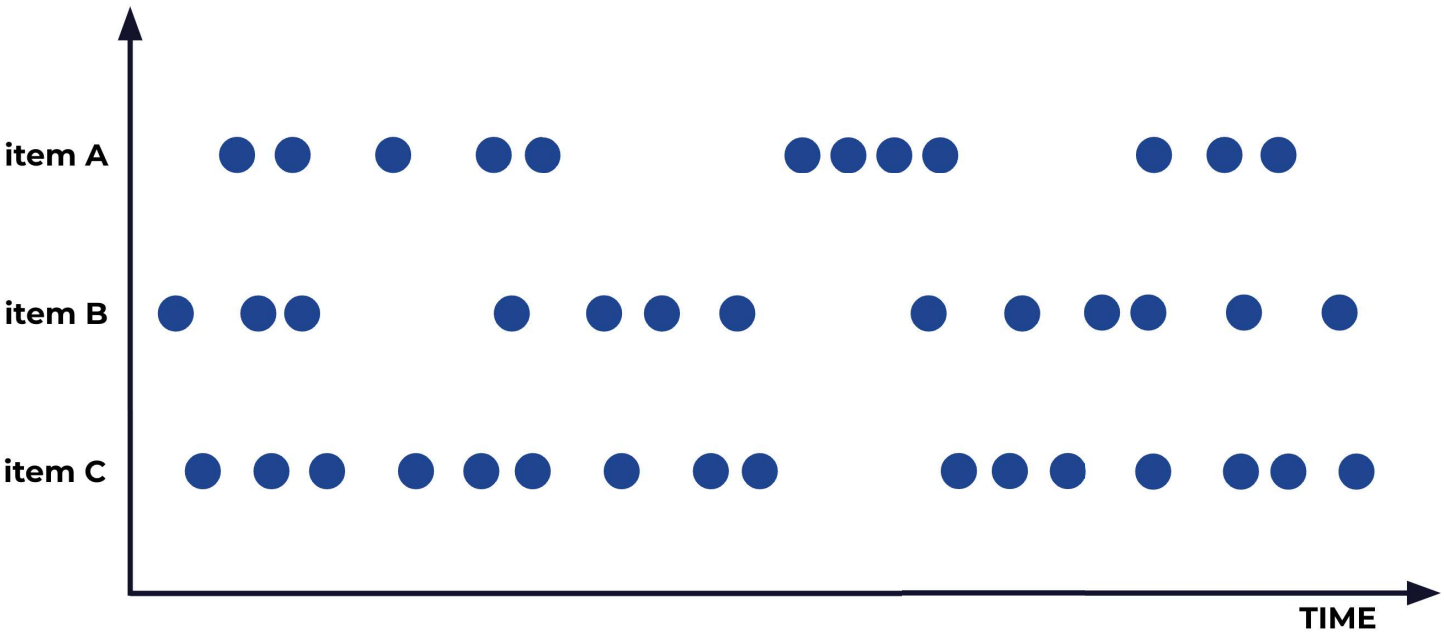
Windows in Flink SQL

Flink provides 4 built-in windowing Table-Valued Functions (TVFs) :

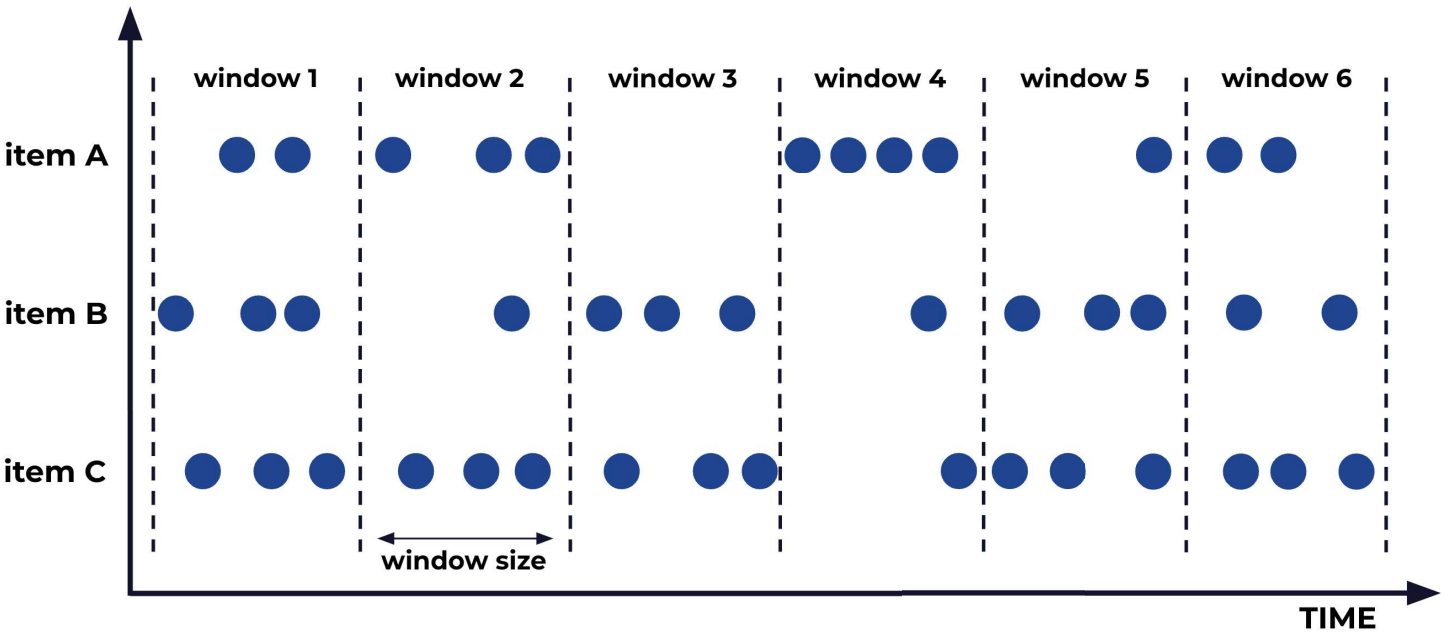
- Tumble Windows
- Hop Windows
- Cumulate Windows
- Session Windows

Windows are at the heart of processing infinite streams. Windows split the stream into "buckets" of finite size, over which we can apply computations.

Window Types - Example



Tumble Window



Tumble Window Code

Syntax:

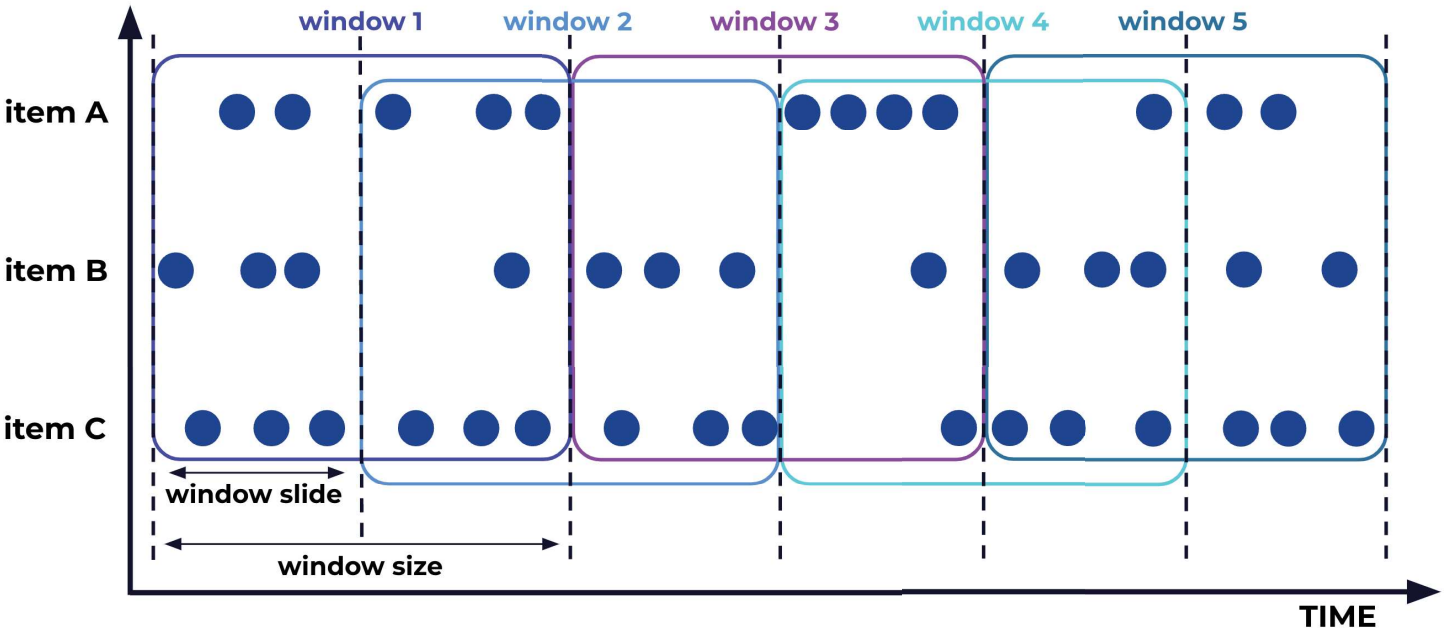
```
TUMBLE(TABLE data, DESCRIPTOR(timecol), size)
```

Example SQL query:

```
SELECT * FROM TABLE(
      TUMBLE(TABLE Orders, DESCRIPTOR($rowtime), INTERVAL '10' MINUTES));
```

```
+------------------+-------+------+------------------+------------------
+--------------------------+
|         $rowtime | price | item |     window_start |       window_end |
window_time  |
+------------------+-------+------+------------------+------------------
+--------------------------+
| 2020-04-15 08:05 |  4.00 | C    | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15
08:09:59.999 |
| 2020-04-15 08:07 |  2.00 | A    | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15
08:09:59.999 |
| 2020-04-15 08:09 |  5.00 | D    | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15
08:09:59.999 |
| 2020-04-15 08:11 |  3.00 | B    | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15
08:19:59.999 |
| 2020-04-15 08:13 |  1.00 | E    | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15
08:19:59.999 |
| 2020-04-15 08:17 |  6.00 | F    | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15
08:19:59.999 |
+------------------+-------+------+------------------+------------------
+--------------------------+
```

Hop Window



Hop Window Code

Syntax:

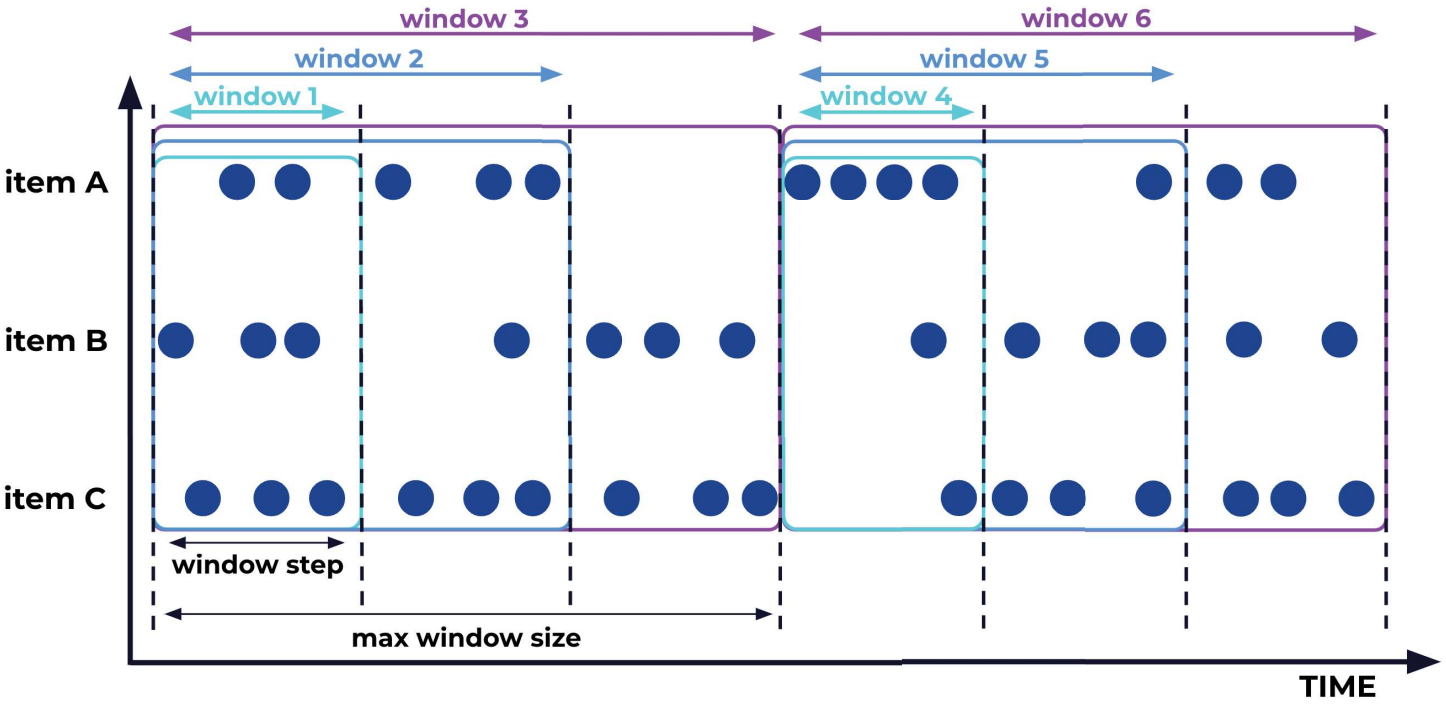
```
HOP(TABLE data, DESCRIPTOR(timecol), slide, size)
```

Example SQL query:

```
SELECT * FROM TABLE(
      HOP(TABLE Orders, DESCRIPTOR($rowtime), INTERVAL '5' MINUTES,
INTERVAL '10' MINUTES));
```

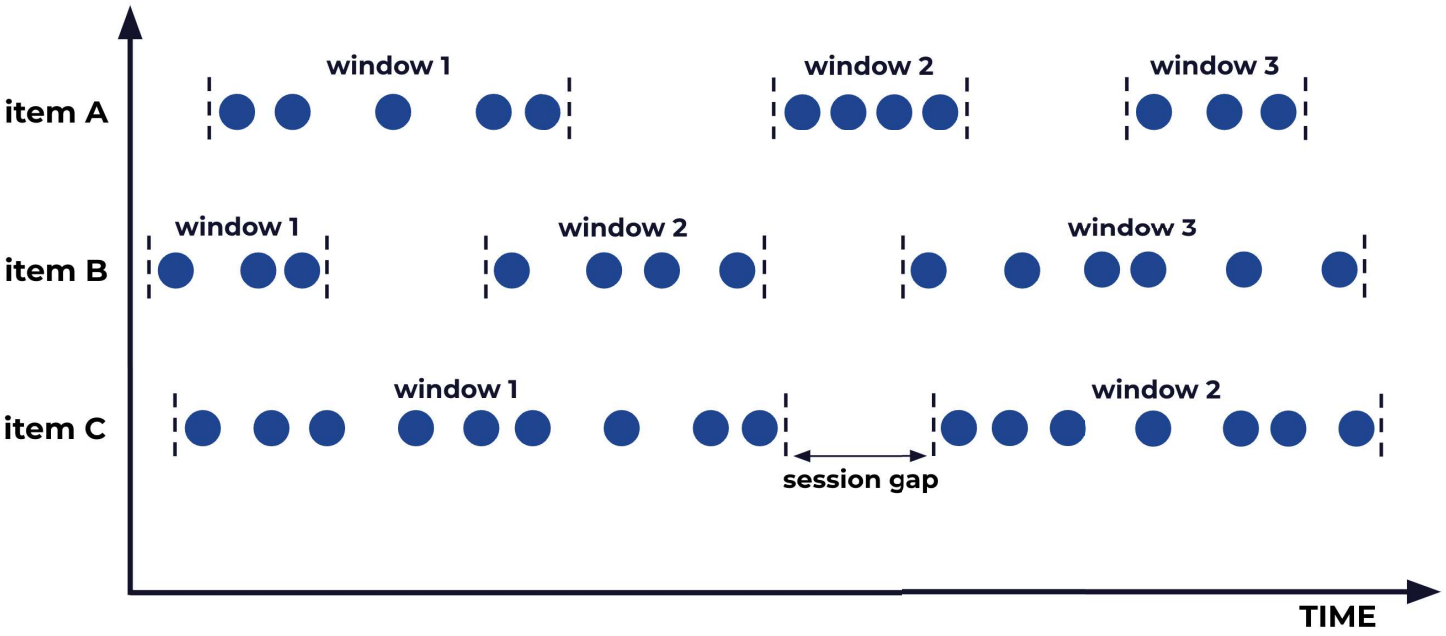
```
+------------------+-------+------+------------------+------------------
+-------------------------+
|         $rowtime | price | item |     window_start |       window_end |
window_time   |
+------------------+-------+------+------------------+------------------
+-------------------------+
| 2020-04-15 08:05 |  4.00 | C    | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15
08:09:59.999 |
| 2020-04-15 08:05 |  4.00 | C    | 2020-04-15 08:05 | 2020-04-15 08:15 | 2020-04-15
08:14:59.999 |
| 2020-04-15 08:09 |  5.00 | D    | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15
08:09:59.999 |
| 2020-04-15 08:09 |  5.00 | D    | 2020-04-15 08:05 | 2020-04-15 08:15 | 2020-04-15
08:14:59.999 |
| 2020-04-15 08:11 |  3.00 | B    | 2020-04-15 08:05 | 2020-04-15 08:15 | 2020-04-15
08:14:59.999 |
| 2020-04-15 08:11 |  3.00 | B    | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15
08:19:59.999 |
| 2020-04-15 08:17 |  6.00 | F    | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15
08:19:59.999 |
| 2020-04-15 08:17 |  6.00 | F    | 2020-04-15 08:15 | 2020-04-15 08:25 | 2020-04-15
08:24:59.999 |
+------------------+-------+------+------------------+------------------
+-------------------------+
```

Cumulate Window



A use case for this type of window would be analyzing exit polls during elections.

Session Window



04e: Time Functions & Data Types

Description

Brief description of the different Flink SQL functions and data types related to Time.

Time Data Types

Type	Example
`DATE`	2024-12-24
`INTERVAL DAY TO SECOND`	+2 07:33:20.000
`INTERVAL YEAR TO MONTH`	+2000-02
`TIME`	10:56:22.541
`TIMESTAMP`	2024-12-24 10:59:32.628
`TIMESTAMP_LTZ`	2024-12-24 11:06:47.224Z

DATE
Represents a calendar date without time.

- Format: YYYY-MM-DD
- Example: 2024-12-24
- Use Case: Useful for storing birth dates, event dates, and any scenario where only the date is relevant.

INTERVAL DAY TO SECOND
Represents a span of time, specifically the difference in days, hours, minutes, seconds, and fractional seconds.

- Format: +DD HH:MI:SS.FFF
- Example: +2 07:33:20.000 (2 days, 7 hours, 33 minutes, and 20 seconds)
- Use Case: Useful for measuring durations and intervals down to fractions of a second, such as tracking elapsed time between events.

INTERVAL YEAR TO MONTH
Represents a span of time in terms of years and months.

- Format: +YYYY-MM
- Example: +2000-02 (2000 years and 2 months)
- Use Case: Useful for representing long-term durations, such as the lifespan of products, contracts, or historical periods.

Time Functions

Date	Time	Timestamp	Utility
CURRENT_DATE	CONVERT_TZ	CURRENT_TIMESTAMP	CEIL
DATE_FORMAT	CURRENT_TIME	CURRENT_ROW_TIMESTAMP	CURRENT_WATERMARK
DATE	HOUR	LOCALTIMESTAMP	EXTRACT
DAYOFMONTH	LOCALTIME	TIMESTAMP	FLOOR
DAYOFWEEK	MINUTE	TO_TIMESTAMP	FROM_UNIXTIME
DAYOFYEAR	NOW	TO_TIMESTAMP_LTZ	INTERVAL
MONTH	SECOND	TIMESTAMPADD	OVERLAPS
QUARTER	TIME	TIMESTAMPDIFF	PROCTIME
TO_DATE		UNIX_TIMESTAMP	
WEEK		UNIX_TIMESTAMP	
YEAR			

Lab Module 04: Using Watermarks and Windows

Please work on **Lab Module 04: Using Watermarks and Windows**.

Refer to the Exercise Guide.