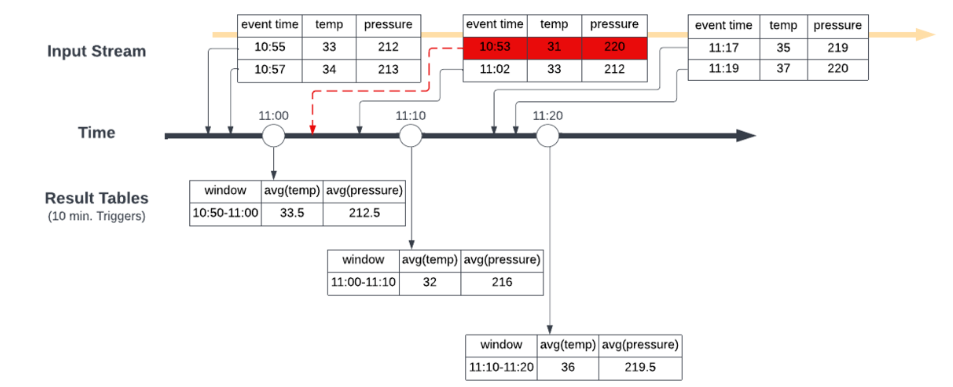
**Generally speaking, when working with real-time streaming data there will be delays between event time and processing time due to how data is ingested and whether the overall application experiences issues like downtime. Due to these potential variable delays, the engine that you use to process this data needs to have some mechanism to decide when to close the aggregate windows and produce the aggregate result.**

While the natural inclination to remedy these issues might be to use a fixed delay based on the wall clock time, we will show in this upcoming example why this is not the best solution.

To explain this visually let’s take a scenario where we are receiving data at various times from around 10:50 AM → 11:20 AM. We are creating 10-minute tumbling windows that calculate the average of the temperature and pressure readings that came in during the windowed period.



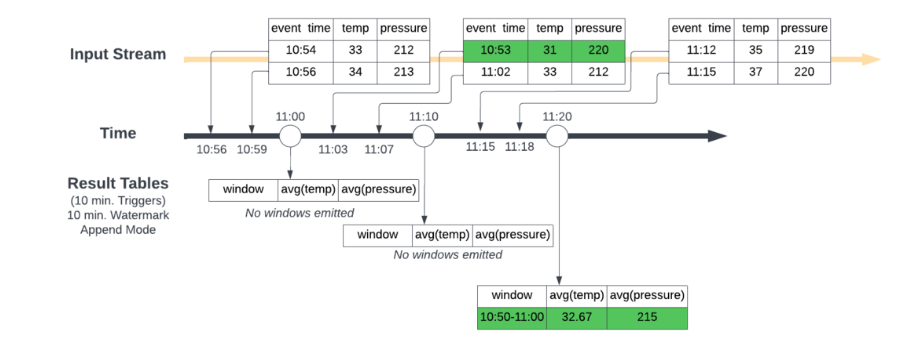
In this first picture, we have the tumbling windows trigger at 11:00 AM, 11:10 AM and 11:20 AM leading to the result tables shown at the respective times. When the second batch of data comes around 11:10 AM with data that has an event time of 10:53 AM this gets incorporated into the temperature and pressure averages calculated for the 11:00 AM → 11:10 AM window that closes at 11:10 AM, which does not give the correct result.

To ensure we get the correct results for the aggregates we want to produce, we need to define a watermark that will allow Spark to understand when to close the aggregate window and produce the correct aggregate result.

In Structured Streaming applications, we can ensure that all relevant data for the aggregations we want to calculate, is collected by using a feature called watermarking.

In the most basic sense, by defining a watermark Spark Structured Streaming then knows when it has ingested all data up to some time, T, (based on a set lateness expectation) so that it can close and produce windowed aggregates up to timestamp T.

This is the second scenario. Watermark is 10 minutes and output mode is append.



Unlike the first scenario where Spark will emit the windowed aggregation for the previous ten minutes every ten minutes (i.e. emit the 11:00 AM →11:10 AM window at 11:10 AM), Spark now waits to close and output the windowed aggregation once the max event time seen minus the specified watermark is greater than the upper bound of the window.

In other words, Spark needs to wait until it sees data points where the latest event time seen minus 10 minutes was greater than 11:00 AM to emit the 10:50 AM → 11:00 AM aggregate window. At 11:00 AM, it does not see this (10:56 – 10 mins = 10:46 < 11:00) so it only initializes the aggregate calculation in Spark’s internal state store. At 11:10 AM (11:02 – 10 mins = 10:52 < 11:00), this condition is still not met, but we have a new data point for 10:53 AM so the internal state gets updated, just not emitted. Then finally by 11:20 AM Spark has seen a data point with an event time of 11:15 AM and since 11:15 AM minus 10 minutes is 11:05 AM which is later than 11:00 AM the 10:50 AM → 11:00 AM window can be emitted to the result table.

This produces the correct result by properly incorporating the data based on the expected lateness defined by the watermark. Once the results are emitted the corresponding state is removed from the state store.

To incorporate watermarking we first needed to identify two items:

The column that represents the event time of the sensor reading

The estimated expected time skew of the data

*Watermarks in Different Output Modes:*

Before we dive deeper, it is important to understand how your choice of output mode affects the behaviour of the watermarks you set.

**Watermarks can only be used when you are running your streaming application in *append* or *update* output modes.** There is a third output mode, *complete* mode, in which the entire result table is written to storage. This mode cannot be used because it requires all aggregate data to be preserved, and *hence cannot use watermarking to drop intermediate state*.

The implication of these output modes in the context of window aggregation and watermarks is that in ‘append’ mode an aggregate can be produced only once and cannot be updated. Therefore, once the aggregate is produced, the engine can delete the aggregate’s state and thus keep the overall aggregation state bounded. Late records – the ones for which the approximate watermark heuristic did not apply (they were older than the watermark delay period), therefore have to be dropped by necessity – the aggregate has been produced and the aggregate state deleted.

Inversely, for ‘update’ mode, the aggregate can be produced repeatedly starting from the first record and on each received record, thus a watermark is optional. The watermark is only useful for trimming the state once heuristically the engine knows that no more records for that aggregate can be received. Once the state is deleted, again any late records have to be dropped as the aggregate value has been lost and can’t be updated.

It is important to understand how state, late-arriving records, and the different output modes could lead to different behaviours of your application running on Spark. **The main takeaway here is that in both append and update modes, once the watermark indicates that all data is received for an aggregate time window, the engine can trim the window state. In append mode the aggregate is produced only later than the closing time window, while in update mode it is produced on every update to the window.**

**Lastly, by increasing your watermark delay window you will cause the pipeline to wait longer for data and potentially drop less data – higher precision, but also higher latency to produce the aggregates. On the flip side, smaller watermark delay leads to lower precision but also lower latency to produce the aggregates.**