Databricks Structured Streaming: Watermark Best Practices & Learning Story

This document outlines best practices for using watermarking in Databricks Structured Streaming, based on real-time use cases and practical implementation.

Purpose of Watermark

Watermarking in Structured Streaming enables late data handling by setting a threshold that determines how long the engine should wait for late-arriving data before finalizing a windowed aggregation or stateful operation.

.withWatermark("event\_time", "10 minutes")

Spark **waits up to 10 minutes** (based on event time) before finalizing a time window.

**Watermarking** sets a **waiting time threshold** that tells Spark how long to **wait for late-arriving data**, based on the event's timestamp — **not the file arrival time**.

Key Benefits:

Controls state size by discarding old state

Handles late-arriving data predictably

* Supports time-windowed aggregations

Real-Time Use Case: Sensor Events Stream  
Scenario:

* IoT sensors continuously emit event logs with fields: sensor\_id, event\_time, temperature
* Files land in /mnt/data/sensors/incoming/
* Goal: Aggregate the average temperature over 10-minute event-time windows, with tolerance for 5 minutes of late data

Setup Configuration:

source\_path = "/mnt/data/sensors/incoming/"  
checkpoint\_path = "/mnt/data/sensors/checkpoints/"  
target\_path = "/mnt/data/sensors/output/"

Streaming Pipeline with Watermark:

from pyspark.sql.functions import window, col

sensor\_df = spark.readStream.format("cloudFiles")   
.option("cloudFiles.format", "json")   
.option("cloudFiles.schemaLocation", "/mnt/data/sensors/schema/")   
.load(source\_path)

sensor\_agg = sensor\_df   
.withWatermark("event\_time", "5 minutes")   
.groupBy(window(col("event\_time"), "10 minutes"), col("sensor\_id"))   
.agg({'temperature': 'avg'})

query = sensor\_agg.writeStream   
.format("delta")   
.option("checkpointLocation", checkpoint\_path)   
.outputMode("append")   
.start(target\_path)

How Watermark Works:

* Watermark defines the maximum delay allowed for late data
* Spark tracks the max event\_time seen and drops data older than (max event\_time - watermark)
* Window aggregation finalizes once watermark crosses the end of a window

Resetting Watermark Behavior:

* You can't reset watermark mid-stream; restart with a new checkpoint
* To reprocess old data, delete checkpoint and reconfigure

Best Practices Checklist:

|  |  |
| --- | --- |
| **Area** | **Best Practice** |
| Event Time Column | Always use a consistent and correct event\_time field |
| Watermark Threshold | Use a realistic value based on data delay tolerance |
| State Management | Tune watermark to reduce state memory overhead |
| Late Data Auditing | Optionally route dropped late events to a quarantine path |
| Aggregation Windows | Use time-based windows with watermark for accuracy |
| Checkpointing | Ensure checkpointLocation is always defined |

Example: Watermark and Window

.withWatermark("event\_time", "10 minutes")  
.groupBy(window(col("event\_time"), "30 minutes"))

Monitoring and Troubleshooting:

* Use Spark UI to monitor watermark progression and state size
* Log metrics like max event\_time and processed rows
* Alert if input event\_time is significantly behind processing time

Summary

Watermarking is essential for processing unbounded streams with late data. It improves correctness and efficiency in windowed aggregations and stateful operations by bounding state retention. Proper tuning of watermark values balances latency, accuracy, and resource usage.

Learn to control your stream, not be controlled by it.

**Appendix :**

How Watermark works:

## Step by step:

### 1. ****Spark looks at the data’s**** event\_time column

It tracks the **latest (maximum) event time** it has seen **from your incoming data**, not based on file arrival.

Example:  
If Spark has seen events up to 11:30 AM, it considers **11:30 AM** as the **current max event time**.

### 2. ****You define a watermark (e.g.,**** 10 minutes****)****

This means you’re telling Spark:

“Wait for late data that’s up to 10 minutes behind the most recent event seen.”

### 3. ****Spark calculates a cutoff time****

It subtracts the watermark value from the max event time:

java

CopyEdit

max event\_time = 11:30 AM

watermark = 10 minutes

cutoff = 11:30 - 10 min = 11:20 AM

### 4. ****Spark DROPS any data older than that cutoff****

Any record with event\_time < 11:20 AM is **too late** and will **not be processed** in windowed aggregations.

## 📊 Visual Example:

diff

CopyEdit

Max event\_time seen: 11:30 AM

Watermark: 10 minutes

Cutoff time (final): 11:20 AM

----------------------------------------------------

Data arriving now:

- event\_time = 11:25 AM ✅ accepted (within window)

- event\_time = 11:05 AM ❌ dropped (too late)

- event\_time = 11:10 AM ❌ dropped (too late)

- event\_time = 11:20 AM ✅ accepted (on edge of watermark)

## In simple words:

Spark is willing to **wait for late data** — but only **up to a limit** (the watermark). Once that limit is passed, any **older data is ignored**.

**Code Example:**

## Step-by-Step: Watermark Behavior in PySpark

### 1. Simulate Incoming Data with event\_time

python

CopyEdit

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, window

from datetime import datetime, timedelta

spark = SparkSession.builder.getOrCreate()

# Sample event data with event\_time

data = [

("sensor\_1", datetime(2024, 5, 25, 11, 5)), # old

("sensor\_2", datetime(2024, 5, 25, 11, 20)), # on the edge

("sensor\_3", datetime(2024, 5, 25, 11, 25)), # valid

("sensor\_4", datetime(2024, 5, 25, 11, 30)), # max seen

]

schema = ["sensor\_id", "event\_time"]

df = spark.createDataFrame(data, schema=schema)

### 2. Define Watermark (10 minutes)

python

CopyEdit

stream\_df = (

df

.withWatermark("event\_time", "10 minutes")

.groupBy(window(col("event\_time"), "30 minutes"), col("sensor\_id"))

.count()

)

This line tells Spark:

“Drop any data where event\_time is \*\*older than (max seen event\_time - 10 minutes)`.”

3. Calculate Cutoff (Explained)

Internally, Spark would compute something like:

python

CopyEdit

max\_event\_time = datetime(2024, 5, 25, 11, 30)

watermark\_delay = timedelta(minutes=10)

cutoff = max\_event\_time - watermark\_delay # => 11:20 AM

So, only rows with event\_time >= 11:20 AM are allowed into the aggregation.

### 4. Spark Drops Older Rows

You can filter manually to simulate this behavior in batch (for demonstration):

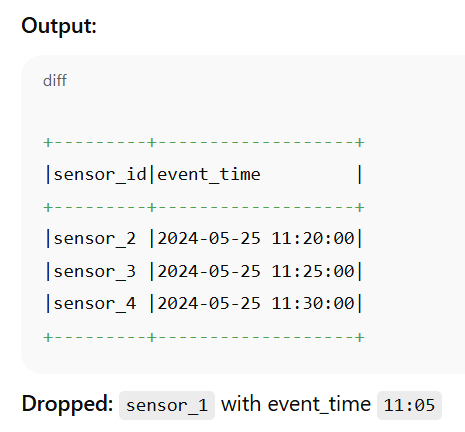
python

CopyEdit

cutoff\_time = datetime(2024, 5, 25, 11, 20)

df\_filtered = df.filter(col("event\_time") >= cutoff\_time)

df\_filtered.show()



## Summary

* .withWatermark("event\_time", "10 minutes") ensures late data is allowed **within a tolerance window**.
* Spark uses **max observed event\_time – watermark** as a **cutoff**.
* Any data **older than that cutoff** is excluded from aggregations.