

Temporal Analysis with Stream Windowing Functions in DuckDB

Petrica Leuca, Gabor Szarnyas

2025-05-02 · 7 min

TL;DR: DuckDB can perform time-based analytics using windows with different semantics (e.g., tumbling, hopping and sliding windows). In this post, we demonstrate these by detecting trends and anomalies in the railway service at Amsterdam Centraal Station.

Introduction

In data platforms, we usually categorize the data into dimension and fact data. While dimensions contain information about entities (name, address, serial number, etc.), facts contain events related to such entities (clicks, sales, bank transactions, readings from IoT devices, etc.). In general, fact data includes a timestamp attribute, denoting the moment when the event happened (or was observed).

When the timestamped data is processed on a streaming platform, it is often processed with *stream windowing functions* in order to organize the data into time windows. In this post, we will show how to apply stream windows on static timestamped fact data in DuckDB, as part of a data analysis task to compute train service summaries, trends and interruptions at Amsterdam Centraal Station.

In a future post, we'll cover streaming design patterns with DuckDB.

For the current implementation, we will use the DuckDB database created in the dbt project detailed in the article [“Fully Local Data Transformation with dbt and DuckDB”](#), based on the open data from the [Rijden de Treinen \(*Are the trains running?*\) application](#). We start by attaching (in any DuckDB session) the database from our storage location.

```
ATTACH 'http://blobs.duckdb.org/data/dutch_railway_network' AS  
USE dutch_railway_network.main_main;
```

Warning

The database is rather big (approx. 1.2 GB), therefore make sure to have a stable internet connection. Instead of attaching the database, you can also [download the database file](#) and connect to it from the command line:

```
$ duckdb dutch_railway_network.duckdb -cmd 'USE main_main;
```

Tumbling Windows

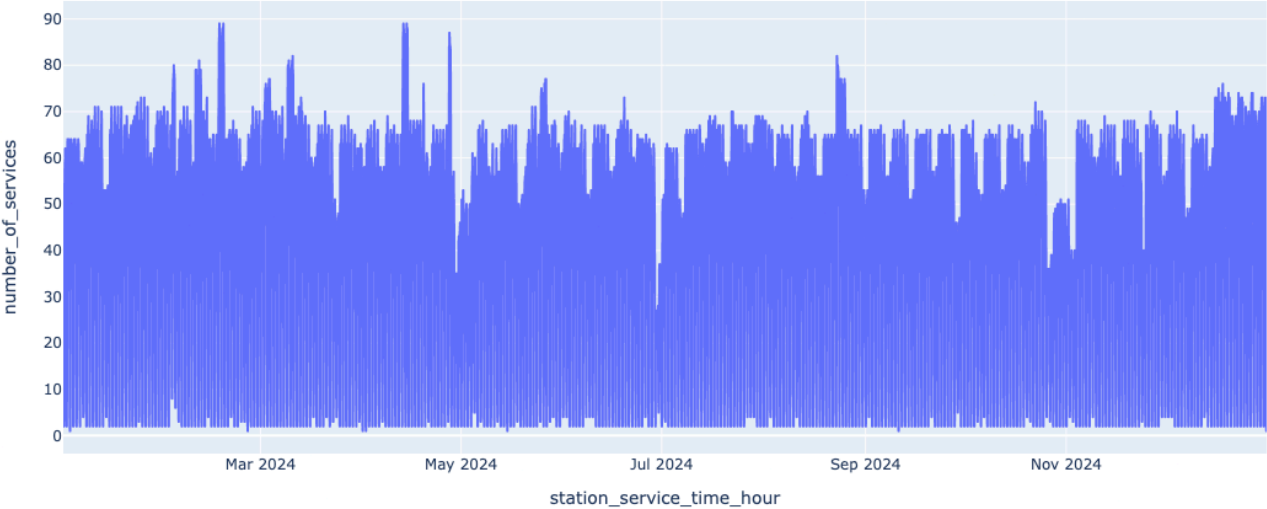
Tumbling windows are fixed-size [left-closed, right-open) time intervals, used to calculate summaries at a certain time unit level (year, day, hour, etc.). Tumbling windows are also used to transform (irregular) fact data into time series data, by aggregating it at a regular time interval.

One way of implementing tumbling windows is to use the `date_trunc` function, which will truncate the timestamp to the specified precision. For example, in the following, we retrieve the number of services for each hour and each day in 2024:

```
SELECT
    date_trunc('hour', station_service_time) AS window_start,
    window_start + INTERVAL 1 HOUR AS window_end,
    count(*) AS number_of_services
FROM ams_traffic_v
WHERE year(station_service_time) = 2024
GROUP BY ALL
ORDER BY 1;
```

window_start timestamp	window_end timestamp	number_of_servic int64
2024-01-01 01:00:00	2024-01-01 02:00:00	
2024-01-01 02:00:00	2024-01-01 03:00:00	
2024-01-01 03:00:00	2024-01-01 04:00:00	
.	.	
.	.	
.	.	
2024-12-31 20:00:00	2024-12-31 21:00:00	
2024-12-31 21:00:00	2024-12-31 22:00:00	
2024-12-31 23:00:00	2025-01-01 00:00:00	
8781 rows (6 shown)		3 column

Hourly Train Services, 2024



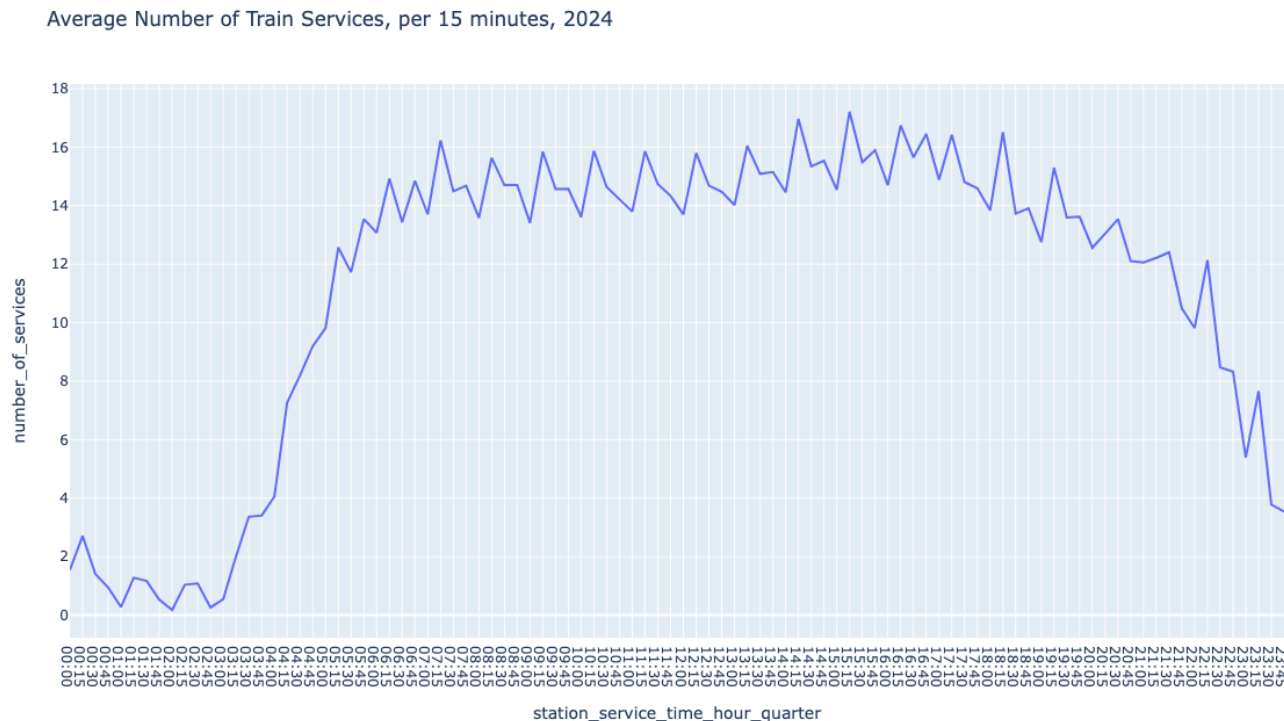
Another approach is to use the `time_bucket` function, which will truncate the timestamp to the bucket width provided, starting from the specified offset. For example, we calculate the number of services each quarter of an hour, starting with `00`:

```
SELECT
  time_bucket(
    INTERVAL 15 MINUTE, -- bucket width
    station_service_time,
    INTERVAL 0 MINUTE -- offset
  ) AS window_start,
  window_start + INTERVAL 15 MINUTE as window_end,
  count(*) AS number_of_services
FROM ams_traffic_v
WHERE year(station_service_time) = 2024
GROUP BY ALL
ORDER BY 1;
```

window_start timestamp	window_end timestamp	number_of_services int64
2024-01-01 01:30:00	2024-01-01 01:45:00	
2024-01-01 01:45:00	2024-01-01 02:00:00	
2024-01-01 02:15:00	2024-01-01 02:30:00	
.	.	
.	.	
.	.	
2024-12-31 20:45:00	2024-12-31 21:00:00	
2024-12-31 21:00:00	2024-12-31 21:15:00	
2024-12-31 23:45:00	2025-01-01 00:00:00	
32932 rows (6 shown)		3 columns

The time bucket function is generating the buckets from the timestamp column itself, there could be gaps in the time series data. As seen in the above result, the first record is `2024-01-01 01:30:00`, because there are no records before that timestamp.

Given that tumbling windows are non-overlapping intervals, we can calculate summaries, such as the average number of services during a 15 minute interval. It is interesting to observe that the number of train services is quite stable during the day but it's much lower during the night – even in Amsterdam.



```
WITH time_range AS (  
  SELECT  
    range AS window_start,  
    window_start + INTERVAL 15 MINUTE AS window_end  
  FROM range(  
    '2024-01-01 00:00:00'::TIMESTAMP,  
    '2025-01-01 00:00:00'::TIMESTAMP,  
    INTERVAL 5 MINUTE -- hopping size  
  )  
)
```

window_start timestamp	window_end timestamp
2024-01-01 00:00:00	2024-01-01 00:15:00
2024-01-01 00:05:00	2024-01-01 00:20:00
2024-01-01 00:10:00	2024-01-01 00:25:00
.	.
.	.
.	.
2024-12-31 23:45:00	2025-01-01 00:00:00
2024-12-31 23:50:00	2025-01-01 00:05:00
2024-12-31 23:55:00	2025-01-01 00:10:00
105408 rows (6 shown) 2 columns	

We then join the above intervals with the train service data in order to calculate the number of services for each [left-closed, right-open) interval:

```
SELECT  
  window_start,  
  window_end,  
  count(service_sk) AS number_of_services  
FROM ams_traffic_v  
INNER JOIN time_range AS ts  
  ON station_service_time >= ts.window_start  
  AND station_service_time < ts.window_end  
GROUP BY ALL
```

```
ORDER BY 3 DESC, 1 ASC
LIMIT 5;
```

resulting in:

window_start timestamp	window_end timestamp	number_of_servic int64
2024-02-17 10:25:00	2024-02-17 10:40:00	
2024-02-17 11:25:00	2024-02-17 11:40:00	
2024-02-17 16:25:00	2024-02-17 16:40:00	
2024-02-17 09:25:00	2024-02-17 09:40:00	
2024-02-17 12:25:00	2024-02-17 12:40:00	

Can you imagine how it must have been like in the control room when within 15 minutes, 28 trains were arriving or departing in a station with 15 tracks?

By applying a `RIGHT OUTER JOIN` in the above query, gaps are filled with 0 number of services.

Sliding Windows

Sliding windows are overlapping intervals, but, compared to hopping windows, they are dynamically generated from the time column analyzed, therefore changing when new records are inserted. Sliding windows can be implemented by using the `RANGE` window framing:

```
SELECT
  station_service_time - INTERVAL 15 MINUTE AS window_start
  station_service_time AS window_end,
  count(service_sk) OVER (
    ORDER BY station_service_time
    RANGE
```

```

        BETWEEN INTERVAL 15 MINUTE PRECEDING -- window
        AND CURRENT ROW
    ) AS number_of_services
FROM ams_traffic_v
ORDER BY 3 DESC, 1
LIMIT 5;
```

window_start timestamp	window_end timestamp	number_of_servic int64
2024-02-17 11:25:00	2024-02-17 11:40:00	
2024-02-17 10:24:00	2024-02-17 10:39:00	
2024-02-17 11:18:00	2024-02-17 11:33:00	
2024-02-17 11:18:00	2024-02-17 11:33:00	
2024-02-17 11:23:00	2024-02-17 11:38:00	

Because the current row is included in the calculation, the sliding windows are [left-closed, right-closed].

Session Windows

A *session window* groups events that happen close together in time, separated by inactivity gaps. A new session starts when the time between two events exceeds a defined timeout. The most common use case of session windows is to detect gaps in the timestamped data.

We continue the data analysis by identifying the days in which there were periods of time larger than 10 minutes in which no train was arriving/departing in/from the Amsterdam Centraal Station. In this context, a session window is the period of time in which train services run without a service inactivity gap longer than 10 minutes.

We start by calculating, for each record, the previous service time, by using the `Lag` window function. We observed above that there is almost no traffic

during the night, therefore we include only services between 6 AM and 11 PM:

```
SELECT
    service_sk,
    station_service_time,
    lag(station_service_time) OVER (
        PARTITION BY station_service_time::DATE
        ORDER BY station_service_time
    ) AS previous_service_time,
    date_diff('minute', previous_service_time, station_service_time) AS gap_minutes
FROM ams_traffic_v
WHERE hour(station_service_time) BETWEEN 6 AND 23
```

In the above query we also calculate the gap, in minutes, between the current service and the previous service, with `date_diff`. If there is no previous service, the column will be `NULL`, depicting the first service session in the day:

station_service_time timestamp	previous_service_time timestamp	gap_minutes int64
2024-01-09 06:00:00	NULL	NULL
2024-01-16 06:00:00	NULL	NULL
2024-01-22 06:00:00	NULL	NULL
.	.	.
.	.	.
.	.	.
2024-11-28 06:01:00	NULL	NULL
2024-12-05 06:01:00	NULL	NULL
2024-12-23 06:00:00	NULL	NULL
366 rows (6 shown)		3 columns

Tip

Because `gap_minutes` is computed based on a window function, we can filter on it with `QUALIFY`, e.g.: `QUALIFY gap_minutes IS NULL`

We then mark if the current record is in the same session as the previous one, by comparing the minutes elapsed to a timeout, in our case 10 minutes:

```
IF(gap_minutes >= 10 OR gap_minutes IS NULL, 1, 0) AS new_session
```

By applying a moving sum, at day level, over the `new_session` attribute, we assign an identifier to the session:

```
sum(new_session) OVER (
  PARTITION BY station_service_date
  ORDER BY station_service_time ROWS UNBOUNDED PRECEDING
) AS session_id_in_day
```

Bringing it all together, we can now retrieve the dates which had at least one inactivity gap of 10 minutes during the 18 hours day service time (the number of hours between 6 AM and 11 PM):

```
WITH ams_daily_traffic AS (
  SELECT
    service_sk,
    station_service_time,
    lag(station_service_time) OVER (
      PARTITION BY station_service_time::DATE
      ORDER BY station_service_time
    ) AS previous_service_time,
    date_diff('minute', previous_service_time, station_service_time) AS gap_minutes
  FROM ams_traffic_v
  WHERE hour(station_service_time) BETWEEN 6 AND 23
), window_calculation AS (
  SELECT
    service_sk,
    station_service_time,
    station_service_time::DATE AS station_service_date,
    gap_minutes,
    IF(gap_minutes >= 10 OR gap_minutes IS NULL, 1, 0) AS new_session,
    sum(new_session) OVER (
      PARTITION BY station_service_date
      ORDER BY station_service_time ROWS UNBOUNDED PRECEDING
    ) AS session_id_in_day
```

FROM ams_daily_traffic

), session_window AS (

SELECT

station_service_date,

session_id_in_day,

max(gap_minutes) AS gap_minutes,

min(station_service_time) AS window_start,

max(station_service_time) AS window_end,

count(service_sk) AS number_of_services

FROM window_calculation

GROUP BY ALL

)

SELECT

station_service_date,

max(ceil(date_diff('minute', window_start, window_end) / 60) AS number_of_sessions,

count(*) AS number_of_sessions,

sum(number_of_services) as number_of_services,

FROM session_window

GROUP BY ALL

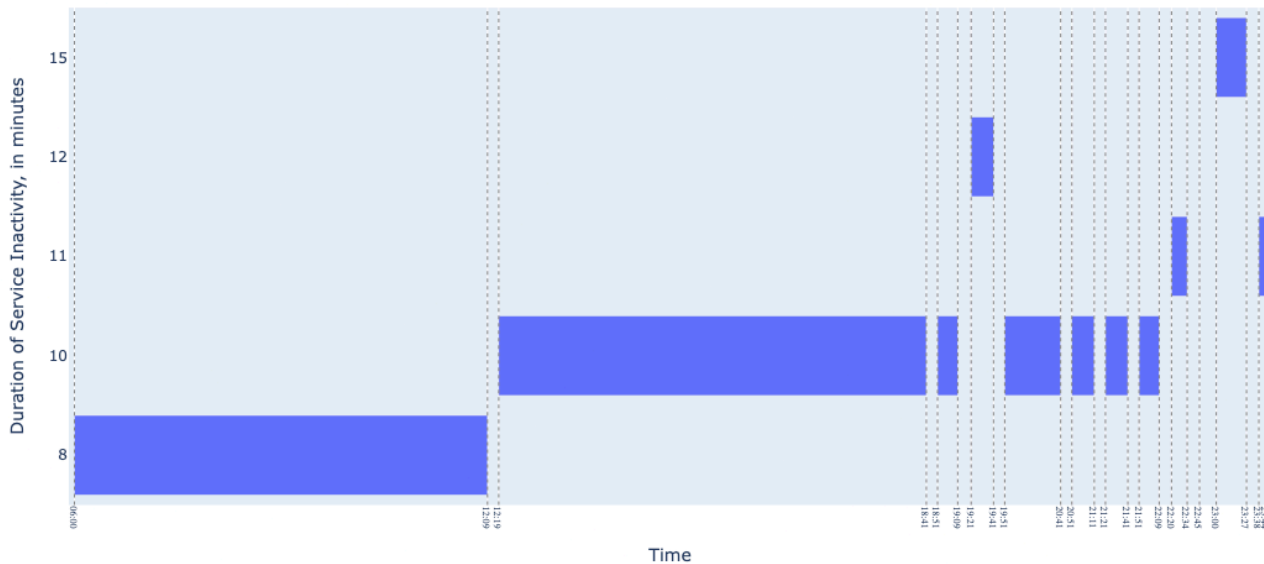
HAVING number_of_hours_without_gap < 18

ORDER BY 2, 1;

station_service_date date	number_of_hours_without_gap double	number_of_sessions integer
2024-04-29	7.0	1
2024-12-31	14.0	1
2024-01-01	16.0	1
2024-04-30	16.0	1
2024-04-14	17.0	1
2024-05-01	17.0	1
2024-05-02	17.0	1
2024-05-03	17.0	1
2024-05-04	17.0	1
2024-05-19	17.0	1
2024-10-28	17.0	1
2024-10-29	17.0	1
2024-10-30	17.0	1
2024-11-02	17.0	1
14 rows		

Something must have happened on 29 April 2024! We observe that, during the 18 hours of service, there were 12 session windows, which means that, for at least 10 times, no train arrived or departed during a period of 10 minutes. A reason for this could be that a regular train service was not running on that day. And indeed maintenance work started between Amsterdam and Utrecht.

Session Windows on 2024-04-29



Tip

Time windows are visualized with Plotly timeline charts, a type of Gantt charts.

Conclusion

In this post we have demonstrated how stream windowing functions can be implemented on historical timestamped data in DuckDB, offering a starting point in time (series) data analysis. We also recommend “Catching up with Windowing”, a post about DuckDB's windowing features, which can be adopted in the functions presented herein.