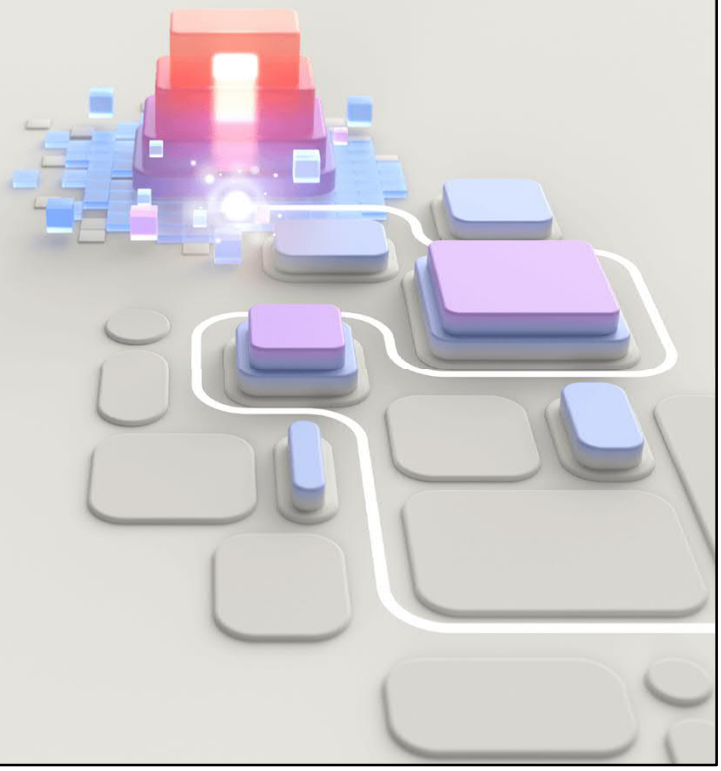
Microsoft

Streaming Analytics



Agenda

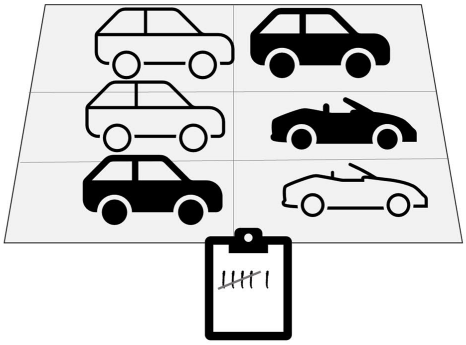
- Streaming and real-time analytics

Learning objectives

After completing this module, you will be able to:

1. Describe common features of real-time analytics solutions

Batch vs stream processing

Batch processing	Stream processing
	
Data is collected and processed at regular intervals	Data is processed in (near) real-time as it arrives

Data processing is simply the conversion of raw data to meaningful information through a process. There are two general ways to process data:

- *Batch processing*, in which multiple data records are collected and stored before being processed together in a single operation.
- *Stream processing*, in which a source of data is constantly monitored and processed in real time as new data events occur.

In batch processing, newly arriving data elements are collected into a group. The whole group is then processed at a future time as a batch. Exactly when each group is processed can be determined in a number of ways. For example, you can process data based on a scheduled time interval (for example, every hour), or it could be triggered when a certain amount of data has arrived, or as the result of some other event.

For example, suppose you want to analyze road traffic by counting the number of cars on a stretch of road. A batch processing approach to this would require that you collect the cars in a parking lot, and then count them in a single operation while they're at rest.

In stream processing, each new piece of data is processed when it arrives. Unlike batch processing, there's no waiting until the next batch processing interval - data is processed as individual units in real-time rather than being processed a batch at a time. Streaming data processing is beneficial in most scenarios where new, dynamic data is generated on a continual basis.

For example, a better approach to our hypothetical car counting problem might be to apply a *streaming* approach, by counting the cars in real-time as they pass:

In this approach, you don't need to wait until all of the cars have parked to start processing them, and you can aggregate the data over time intervals; for example, by counting the number of cars that pass each minute.

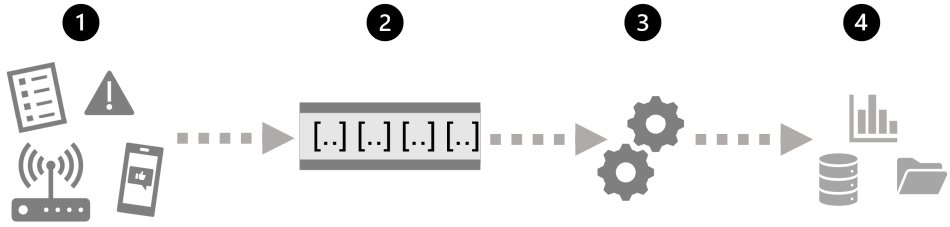
Many large-scale analytics solutions include a mix of batch and stream processing, enabling both historical and real-time data analysis. It's common for stream processing solutions to capture real-time data, process it by filtering or aggregating it, and present it through real-time dashboards and visualizations (for example, showing the running total of cars that have passed along a road within the current hour), while persisting the processed results in a data store for historical analysis alongside batch processed data (for example, to enable analysis of

traffic volumes over the past year).

A toll station is a common phenomenon – we encounter them in many expressways, bridges, and tunnels across the world. Each toll station has multiple toll booths, which may be manual – meaning that you stop to pay the toll to an attendant, or automated – where a sensor placed on top of the booth scans an RFID card affixed to the windshield of your vehicle as you pass the toll booth. It is easy to visualize the passage of vehicles through these toll stations as an event stream over which interesting operations can be performed.

Common elements of stream processing

1. An event generates some data.
2. The generated data is captured in a streaming source for processing.
3. The event data is processed.
4. The results of the stream processing operation are written to an output (or sink).



Microsoft Azure supports multiple technologies that you can use to implement real-time analytics of streaming data, including:

•Azure Stream Analytics: A platform-as-a-service (PaaS) solution that you can use to define *streaming jobs* that ingest data from a streaming source, apply a perpetual query, and write the results to an output.

•Spark Structured Streaming: An open-source library that enables you to develop complex streaming solutions on Apache Spark based services, including Microsoft Fabric and Azure Databricks.

•Microsoft Fabric: A high-performance database and analytics platform that includes Data Engineering, Data Factory, Data Science, Real-Time Analytics, Data Warehouse, and Databases.

Sources for stream processing

The following services are commonly used to ingest data for stream processing on Azure:

•Azure Event Hubs: A data ingestion service that you can use to manage queues of event data, ensuring that each event is processed in order, exactly once.

•Azure IoT Hub: A data ingestion service that is similar to Azure Event Hubs, but which is optimized for managing event data from *Internet-of-things* (IoT) devices.

•Azure Data Lake Store Gen 2: A highly scalable storage service that is often used in *batch processing* scenarios, but which can also be used as a source of streaming data.

•Apache Kafka: An open-source data ingestion solution that is commonly used together with Apache Spark.

Sinks for stream processing

The output from stream processing is often sent to the following services:

•Azure Event Hubs: Used to queue the processed data for further downstream processing.

•Azure Data Lake Store Gen 2, or Azure blob storage: Used to persist the processed results as a file.

•Azure SQL Database or Azure Databricks: Used to persist the processed results in a database table for querying

and analysis.

- Microsoft Power BI: Used to generate real time data visualizations in reports and dashboards.

If your stream process requirements are complex or resource-intensive, you can create a Stream Analysis *cluster*, which uses the same underlying processing engine as a Stream Analytics job, but in a dedicated tenant (so your processing is not affected by other customers) and with configurable scalability that enables you to define the right balance of throughput and cost for your specific scenario.

Azure Stream Analytics Query Language

- Azure Stream Analytics offers a SQL query language for performing transformations and computations over streams of events.
- Stream Analytics query language is a subset of standard T-SQL syntax for doing Streaming computations.
- Handles continuous data processing with time-based windows:
  - Tumbling, Hopping, Sliding, and Session windows for aggregations.
- Provides temporal constructs like:
  - TIMESTAMP BY for custom event time.
  - System.Timestamp() for system time.
- Supports late arrival and out-of-order policies for real-time accuracy.

Late Arrival Policy

Definition: Specifies how long after the window closes an event can still be accepted.

Configuration:

- In the **Stream Analytics job settings**, go to **Event Ordering**.
- Set **Late Arrival Tolerance** (e.g., 00:00:15 for 15 seconds).

Effect:

- If an event arrives later than the tolerance, it's dropped.

Out-of-Order Policy

Definition: Handles events that arrive out of sequence based on application time.

Configuration:

- In **Event Ordering**, set **Out-of-Order Tolerance** (e.g., 00:00:05 for 5 seconds).

Effect:

- If an event's timestamp is older than the tolerance compared to previously processed events, it's dropped.

Example:

- If tolerance is 5 seconds, and an event arrives with a timestamp 10 seconds older than the last processed event, it's discarded.

Arrival Time vs Application Time

- Every event that flows through the system comes with a timestamp
- This timestamp can either be an:
  - Arrival Time: Timestamp assigned when the event reaches the input source (e.g., Event Hub, IoT Hub, Blob Storage).
  - Application time: This is the timestamp embedded in the event payload, representing when the event was actually generated by the source system or device.
  - To use application time, you must explicitly specify **`TIMESTAMP BY <column>`** in your query.

Arrival Time

This is the timestamp assigned **when the event reaches the input source** (e.g., Event Hub, IoT Hub, Blob Storage).

Examples:

- For Event Hubs: EventEnqueuedUtcTime
- For IoT Hub: IoTHub.EnqueuedTime
- For Blob: BlobProperties.LastModified

Default behavior: If you don't specify anything, Stream Analytics uses arrival time for processing.

Best suited for: Archival or simple ingestion scenarios where temporal logic (like windowing or joins) is not critical. [Understand...soft Learn | Learn.Microsoft.com]

Application Time (Event Time)

This is the timestamp **embedded in the event payload**, representing when the event was actually generated by the source system or device.

To use it, you must explicitly specify TIMESTAMP BY <column> in your query.

Best suited for: Scenarios requiring accurate temporal computations (e.g., windowing, joins, detecting patterns) because it accounts for network delays or buffering

If you rely on **arrival time**, events delayed in transit will appear "late," potentially skewing analytics. Using **application time** ensures correctness for time-sensitive logic but introduces complexity: You need to handle **late arrivals** and **out-of-order events** using policies like Late Arrival Tolerance and Out-of-Order Policy.

Example: If an event arrives 40 seconds late but your late arrival policy is 15 seconds, its timestamp may be adjusted or the event dropped.

Best Practices

Prefer Application Time for real-time analytics, anomaly detection, and IoT scenarios. Use TIMESTAMP BY on a

datetime column in the payload.

Arrival Time is fine for logging or archiving where event order doesn't matter.

Configure **Late Arrival and Out-of-Order policies** when using application time to maintain data integrity.

Start with small tolerance windows (e.g., 3–5 seconds) and tune based on latency and correctness requirements.

Event Stream - Sample Query

```
SELECT
    DeviceId,
    AVG(Temperature) AS AvgTemp,
    System.Timestamp AS WindowEnd
FROM
    IoTHubInput TIMESTAMP BY eventTime
GROUP BY
    TumblingWindow(minute, 5)
```

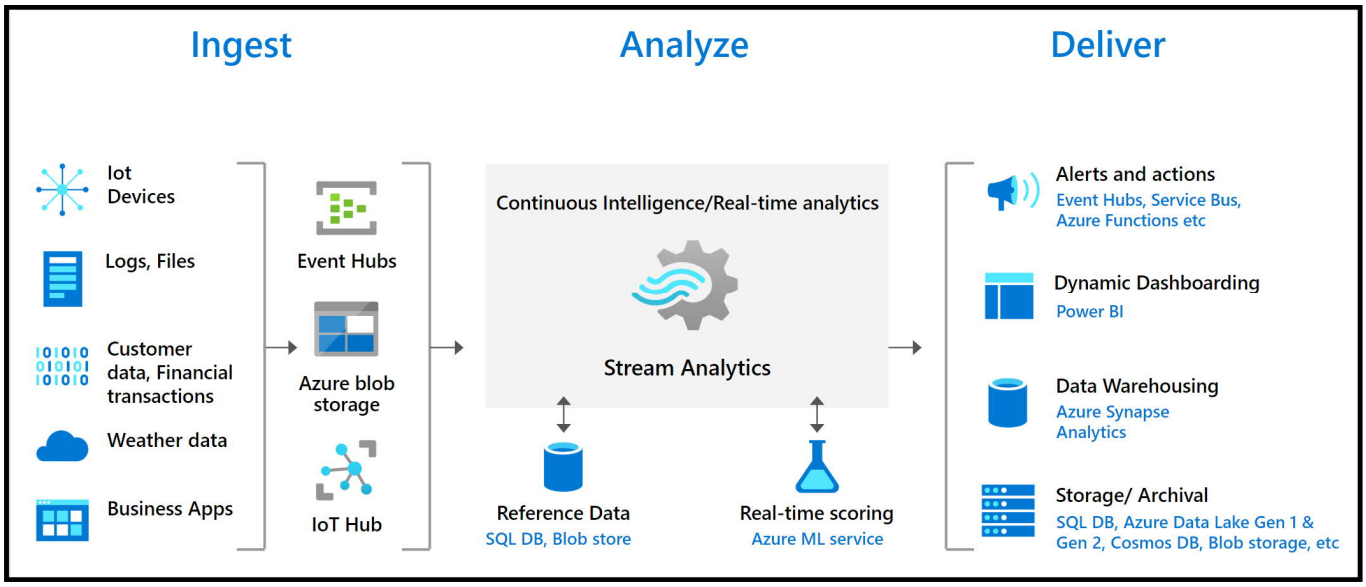

TIMESTAMP BY eventTime
Tells Stream Analytics to use the eventTime column from the payload as the event's timestamp instead of arrival time.

System.Timestamp
Returns the end time of the window based on application time.

TumblingWindow(minute, 5)
Groups events into 5-minute windows according to eventTime.

If you omit TIMESTAMP BY, Stream Analytics will default to **arrival time**, which can cause inaccurate results if events arrive late or out of order.

Azure Stream Analytics



Windowing Functions

- In time-streaming scenarios, performing operations on the data contained in temporal (latin root, meaning *time*) windows is a common pattern.

- There are five kinds of temporal windows to choose from:
  - Tumbling
  - Hopping
  - Sliding
  - Session
  - Snapshot

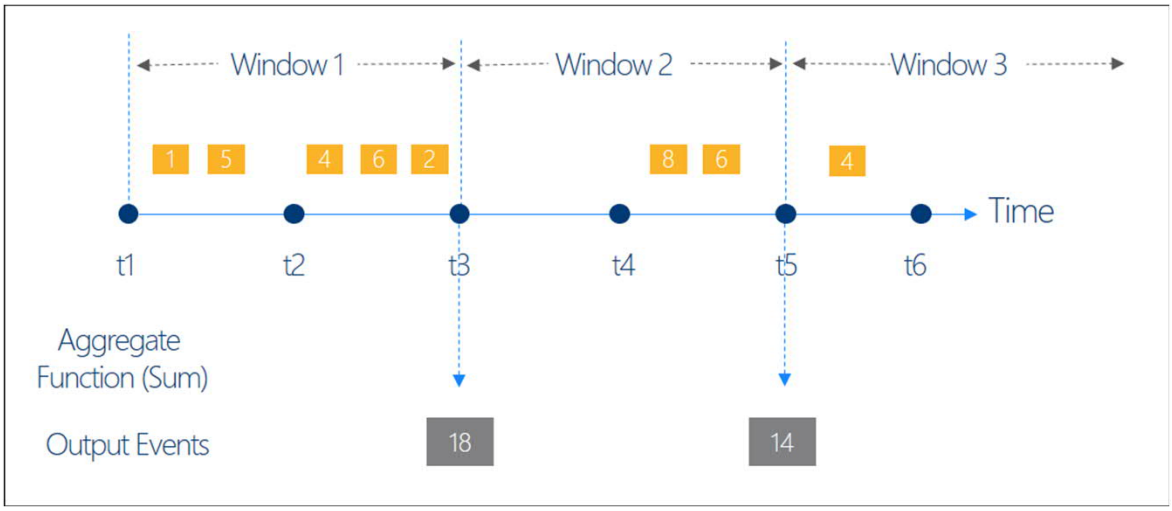
In time-streaming scenarios, performing operations on the data contained in temporal windows is a common pattern. Stream Analytics has native support for windowing functions, enabling developers to author complex stream processing jobs with minimal effort.

There are five kinds of temporal windows to choose from:

- Tumbling
- Hopping
- Sliding
- Session
- Snapshot

Windowing Operations

All the windowing operations output results at the end of the window.



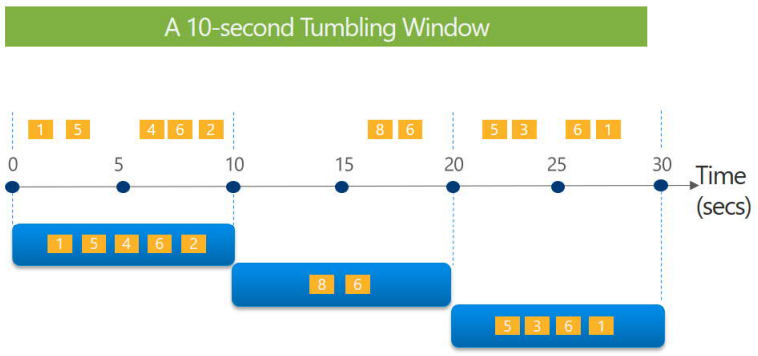
All the windowing operations output results at the **end** of the window. When you start a stream analytics job, you can specify the *Job output start time*, and the system automatically fetches previous events in the incoming streams to output the first window at the specified time; for example, when you start with the *Now* option, it starts to emit data immediately. The output of the window will be a single event based on the aggregate function used. The output event has the time stamp of the end of the window and all window functions are defined with a fixed length.

Every window operation outputs event at the end of the window. The windows of Azure Stream Analytics are opened at the window start time and closed at the window end time.

For example, if you have a 5 minute window from 12:00 AM to 12:05 AM all events with timestamp greater than 12:00 AM and up to timestamp 12:05 AM inclusive will be included within this window.

Tumbling window

Tell me the count of Tweets per time zone every 10 seconds




```
SELECT TimeZone, COUNT(*) AS Count
FROM TwitterStream TIMESTAMP BY CreatedAt
GROUP BY TimeZone, TumblingWindow(second,10)
```

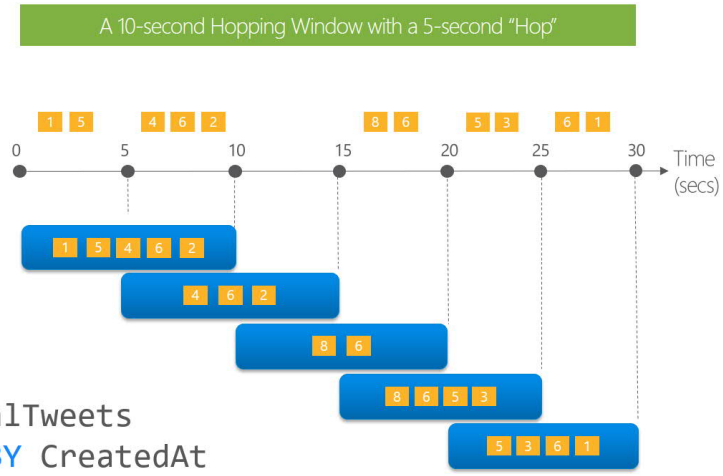
Use Tumbling window functions to segment a data stream into distinct time segments, and perform a function against them.

The key differentiators of a tumbling window are:

- They don't repeat.
- They don't overlap.
- An event can't belong to more than one tumbling window.

Hopping window

Every 5 seconds give me the count of Tweets over the last 10 seconds



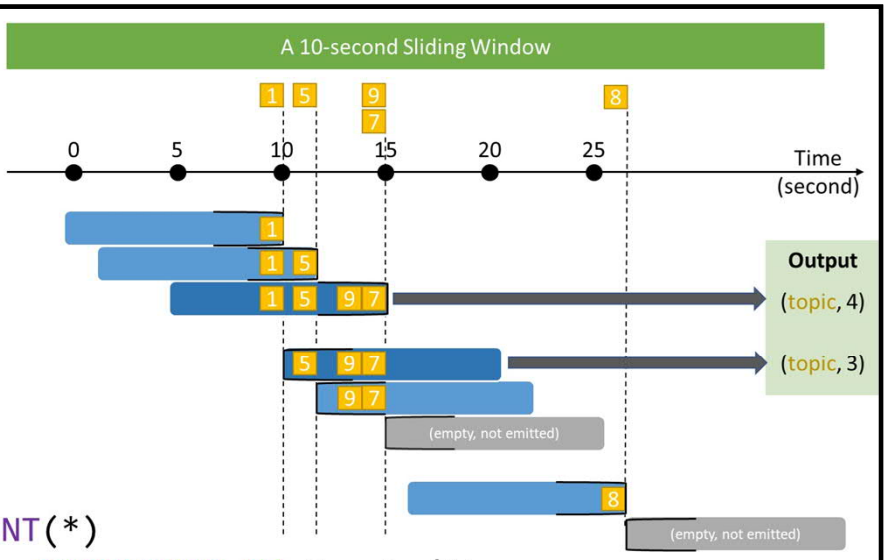

```
SELECT Topic, COUNT(*) AS TotalTweets
FROM TwitterStream TIMESTAMP BY CreatedAt
GROUP BY Topic, HoppingWindow(second, 10 , 5)
```


Hopping window functions hop forward in time by a fixed period. It might be easy to think of them as tumbling windows that can overlap and be emitted more often than the window size. Events can belong to more than one Hopping window result set. To make a Hopping window the same as a Tumbling window, specify the hop size to be the same as the window size.

Sliding window

Alert me whenever a topic is mentioned more than 3 times in under 10 seconds

Note:
- all tweets on the diagram belong to the same topic

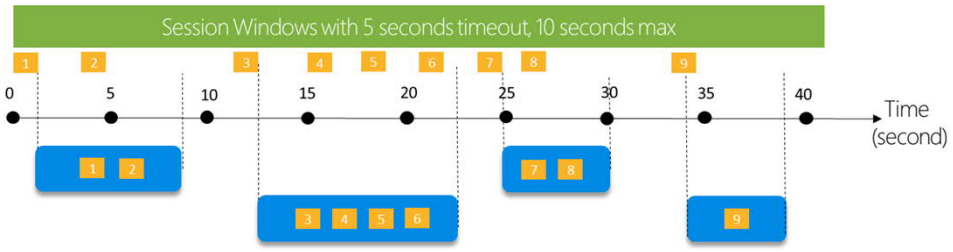



```
SELECT Topic, COUNT(*)
FROM TwitterStream TIMESTAMP BY CreatedAt
GROUP BY Topic, SlidingWindow(second, 10)
HAVING COUNT(*) >= 3
```

Sliding windows, unlike tumbling or hopping windows, output events only for points in time when the content of the window actually changes. In other words, when an event enters or exits the window. So, every window has at least one event. Similar to hopping windows, events can belong to more than one sliding window.

Session window

Tell me the count of Tweets that occur within 5 seconds of each other




```
SELECT System.Timestamp() as WindowEnd, Topic, COUNT(*)
FROM TwitterStream TIMESTAMP BY CreatedAt
GROUP BY Topic, SessionWindow(second, 5, 10)
```


Session window functions group events that arrive at similar times, filtering out periods of time where there's no data. It has three main parameters:

- Timeout
- Maximum duration
- Partitioning key (optional).

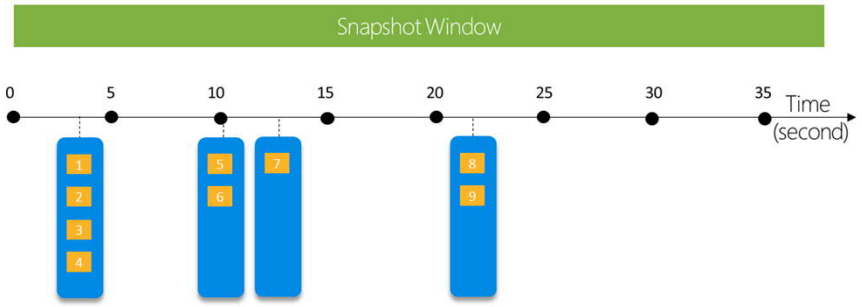
A session window begins when the first event occurs. If another event occurs within the specified timeout from the last ingested event, then the window extends to include the new event. Otherwise if no events occur within the timeout, then the window is closed at the timeout.

If events keep occurring within the specified timeout, the session window keeps extending until maximum duration is reached. The maximum duration checking intervals are set to be the same size as the specified max duration. For example, if the max duration is 10, then the checks on if the window exceeds maximum duration happen at t = 0, 10, 20, 30, etc.

When a partition key is provided, the events are grouped together by the key and session window is applied to each group independently. This partitioning is useful for cases where you need different session windows for different users or devices.

Snapshot window

Give me the count of tweets with the same topic type that occur at exactly the same time




```
SELECT System.Timestamp() as WindowEnd, Topic, COUNT(*)
FROM TwitterStream TIMESTAMP BY CreatedAt
GROUP BY Topic, System.Timestamp()
```


Snapshot windows group events that have the same timestamp. Unlike other windowing types, which require a specific window function (such as SessionWindow()), you can apply a snapshot window by adding System.Timestamp() to the GROUP BY clause.

Data analytics with Apache Spark

Apache Spark is a distributed processing framework for large scale data analytics. You can use Spark on Microsoft Azure in the following services:

- Microsoft Fabric
- Azure Databricks

Spark Structured Streaming

The Spark Structured Streaming library, which provides an application programming interface (API) for ingesting, processing, and outputting results from perpetual streams of data.

Delta Lake

Delta Lake can be used in Spark to define relational tables for both batch and stream processing.

Apache Spark is a distributed processing framework for large scale data analytics. You can use Spark on Microsoft Azure in the following services:

•Microsoft Fabric

•Azure Databricks

Spark can be used to run code (usually written in Python, Scala, or Java) in parallel across multiple cluster nodes, enabling it to process very large volumes of data efficiently. Spark can be used for both batch processing and stream processing.

Spark Structured Streaming

To process streaming data on Spark, you can use the *Spark Structured Streaming* library, which provides an application programming interface (API) for ingesting, processing, and outputting results from perpetual streams of data.

Spark Structured Streaming is built on a ubiquitous structure in Spark called a *dataframe*, which encapsulates a table of data. You use the Spark Structured Streaming API to read data from a real-time data source, such as a Kafka hub, a file store, or a network port, into a "boundless" dataframe that is continually populated with new data from the stream. You then define a query on the dataframe that selects, projects, or aggregates the data - often in temporal windows. The results of the query generate another dataframe, which can be persisted for analysis or further processing.

Delta Lake

Delta Lake is an open-source storage layer that adds support for transactional consistency, schema enforcement, and other common data warehousing features to data lake storage. It also unifies storage for streaming and batch data, and can be used in Spark to define relational tables for both batch and stream processing. When used for stream processing, a Delta Lake table can be used as a streaming source for queries against real-time data, or as a sink to which a stream of data is written.

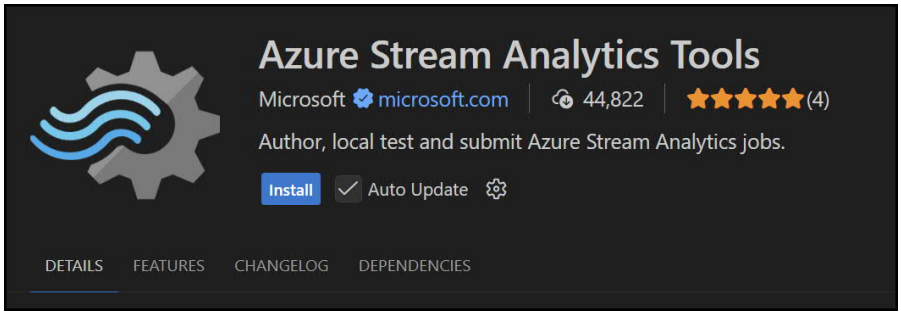
The Spark runtimes in Microsoft Fabric and Azure Databricks include support for Delta Lake.

Delta Lake combined with Spark Structured Streaming is a good solution when you need to abstract batch and stream processed data in a data lake behind a relational schema for SQL-based querying and analysis.

Hand-On Practice (Optional)

- Stream Analytics with VSCode and Data Generation using Azure Rasberry Pi Simulator

(Optional) Azure Stream Analytics Playground VSCode Stream Analytics



Allows you to test Stream Analytics queries locally with sample data using Visual Studio Code!

Optional

https://learn.microsoft.com/en-us/azure/stream-analytics/quick-create-visual-studio-code
https://learn.microsoft.com/en-us/azure/stream-analytics/visual-studio-code-local-run

Azure Raspberry Pi IoT Simulator