Data 603 – Big Data Platforms

Lecture 8
Structured Streaming (Part 2)

Streaming Data Sources and Sinks

- DataFrames from streaming sources can be created using SparkSession.readStream() method
- DataFrames output using DataFrame.writeStream()
- Source type is specified using format() method.

Data Sources - Files

- Structured Streaming supports reading and writing data to and from files
- When reading from files:
  - All the files must be of the same format and are expected to have the same schema.
  - Each file must appear in the directory listing automatically
    - Whole file must be available at once.
    - Any changes to the file will not be processed
  - Within micro-batch, there is no predefined order of reading of files. All of them are read in parallel
  - Only some of the new files will be picked up during the next micro-batch

Data Sources - Files

- Writing to files
  - Only append mode is supported.
  - End-to-end exactly-once guarantees when writing to files by maintaining a log of the data files that have been written to the directory.
    - The log is maintained in *_spark_metadata* sub-directory.
    - Spark query on the directory will automatically use the log to read the correct set of data files to maintain exactly-one guarantee
  - Any schema change after the restart will create data in multiple schemas
    - These schemas will need to be reconciled when querying the directory

Data Sources – Apache Kafka

- Popular sub/pub system
  - Used for storage of data streams
  - Structured Streaming has built-in support for reading from and writing to Apache Kafka

```
inputDF = (spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
    .option("subscribe", "events")
    .load())
```

Table 8-1. Schema of the DataFrame generated by the Kafka source

Column name	Column type	Description
key	binary	Key data of the record as bytes.
value	binary	Value data of the record as bytes.
topic	string	Kafka topic the record was in. This is useful when subscribed to multiple topics.
partition	int	Partition of the Kafka topic the record was in.
offset	long	Offset value of the record.
timestamp	long	Timestamp associated with the record.
timestampType	int	Enumeration for the type of the timestamp associated with the record.

Data Sources – Apache Kafka

- Writing to Kafka
  - Structured Streaming expects the result DataFrame to have a few columns of specified names and types
  - Can write to Kafka in all three modes.
    - Complete mode is not recommended

Table 8-2. Schema of DataFrame that can be written to the Kafka sink

Column name	Column type	Description
key (optional)	`string` or `binary`	If present, the bytes will be written as the Kafka record key; otherwise, the key will be empty.
`value` (required)	`string` or `binary`	The bytes will be written as the Kafka record value.
`topic` (required only if `"topic"` is not specified as option)	`string`	If `"topic"` is not specified as an option, this determines the topic to write the key/value to. This is useful for fanning out the writes to multiple topics. If the `"topic"` option has been specified, this value is ignored.

Data Sources – Apache Kafka

- Writing to Kafka Examples

```
counts = ... # DataFrame[word: string, count: long]
streamingQuery = (counts
    .selectExpr(
    "cast(word as string) as key",
    "cast(count as string) as value")
    .writeStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
    .option("topic", "wordCounts")
    .outputMode("update")
    .option("checkpointLocation", checkpointDir)
    .start())
```

Data Transformations

- Only the DataFrame operations that can be executed incrementally are supported in Structured Streaming
- Catalyst optimizer in Spark SQL converts all the DataFrame operations to an optimized logical plan
  - SQL Planner generates a continuous sequence of execution plans (vs. converting the logical plan to a one-time physical execution plan)
  - The plan processes only a chunk of new data from the input streams and possible some intermediate, partial result computed by the previous execution plan.
- Each execution is considered as a micro-batch
- The partial intermediate result that is communicated between the executions is called the streaming “state”.

Stateless Transformations

- An operation that process each input record individually without needing any information from previous rows
- All project operations (e.g. select(), explode(), map(), flatMap()) and selection operations (e.g. filter(), where())
- Streaming query with only stateless operations support the append and update output modes (no complete mode).
  - Any processed output row of such a query cannot be modified by future data.
- These queries do not combine information across input records.

Stateful Transformations

- Simplest form: DataFrame.groupBy().count()
  - Generating a running count of the number of records received since the beginning of the query.
- State:
  - In micro-batch, the incremental plan adds the count of new records to the previous count generated by the previous micro-batch
  - This partial count communicated between plans is the state.
  - The state is maintained in the memory of the Spark executors.
  - The state is checkpointed to the configured location in order to tolerate failures.

Fault-tolerant State Management

- Spark's scheduler (running in the driver) breaks down high-level operations into smaller tasks.
  - Each task is put into a queue.
  - When resources are available, the executors pull the tasks from the queue to execute them.
  - Each micro-batch in a streaming query performs such set of tasks:
    - Read data from streaming sources and write updated output to streaming sinks
    - Intermediate state data are generated by the micro-batch of tasks
    - The intermediate state data are consumed by the next micro-batch
  - State data generation is partitioned and distributed.
    - It is cached in the executor memory for efficient consumption

Types of Stateful Operations

Two types of stateful operations

- Managed stateful operations
  - Automatically identify and clean up old state based on an operation specific definition of "old" (can be defined).
    - Streaming aggregations
    - Stream-stream joins
    - Streaming deduplication
- Unmanaged stateful operations
  - Users define their own custom state clean up logic
    - MapGroupsWithState
    - FlatMapGroupsWithState

Stateful Streaming Aggregations

Aggregations not based on time

- Global aggregations
  - Aggregations across all the data in the stream. E.g. calculating the running count of the total number of readings received from a sensor.
  - *runningCount = sensorReadings.groupBy().count()*
  - NOTE: For streaming DataFrames, you always need to use DataFrame.groupBy() or Dataset.groupByKey() for aggregations. Direct aggregation operations like DataFrame.count() and Dataset.reduce() cannot be used.
- Grouped aggregations
  - Aggregation within each group or key present in the data stream. E.g. Aggregating data coming from multiple sensors – calculating the running average reading of each sensor.
  - *baselineValues = sensorReadings.groupBy("sensorId").mean("value")*

Streaming DataFrame supported aggregations

- All built-in aggregation functions
  - sum(), mean(), stddev(), countDistinct(), collect_set(), approx_count_distinct(), etc.

- Multiple aggregations computed together

```
from pyspark.sql.functions import *
multipleAggs =
(sensorReadings.groupBy("sensorId").agg(count("*"),mean("value").alias("baselineValue"),
collect_set("errorCode").alias("allErrorCodes")))
```

- User-defined Aggregation Functions
  - All user-defined aggregation functions are supported.

Stateful Streaming Aggregations

Aggregations based on time – aggregation over data bucketed by time windows.

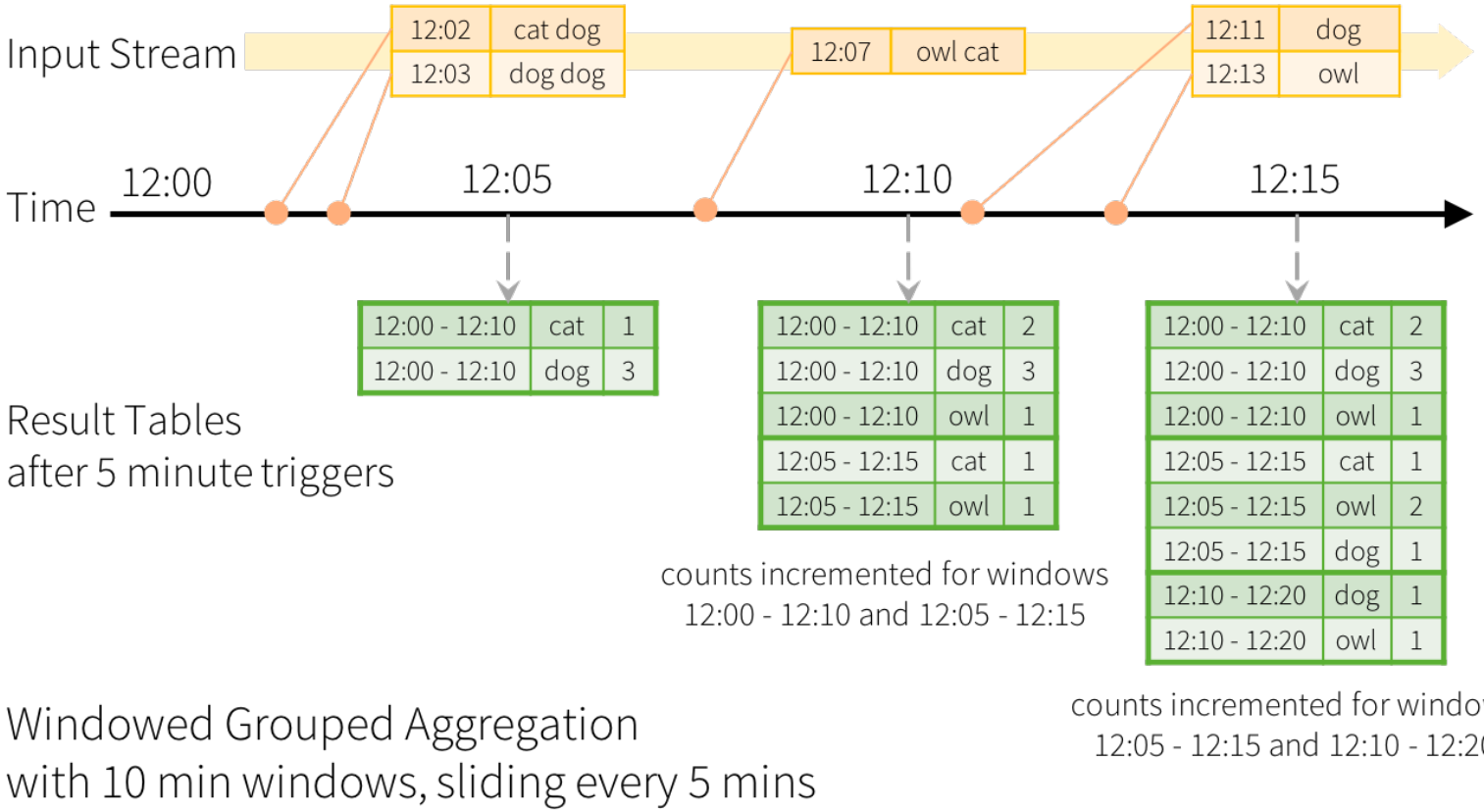
- In a grouped aggregation, aggregate values (e.g. counts) are maintained for each unique value in the user-specified grouping column. In case of window-based aggregations, aggregate values are maintained for each window the event-time of a row falls into.
- Use the event time – the timestamp in the record representing when the record was generated.

```
from pyspark.sql.functions import *
(sensorReadings.groupBy("sensorId", window("eventTime", "5 minute")).count())
```

- window() function – allows for the expression of regular time interval windows as a dynamically computed grouping column.

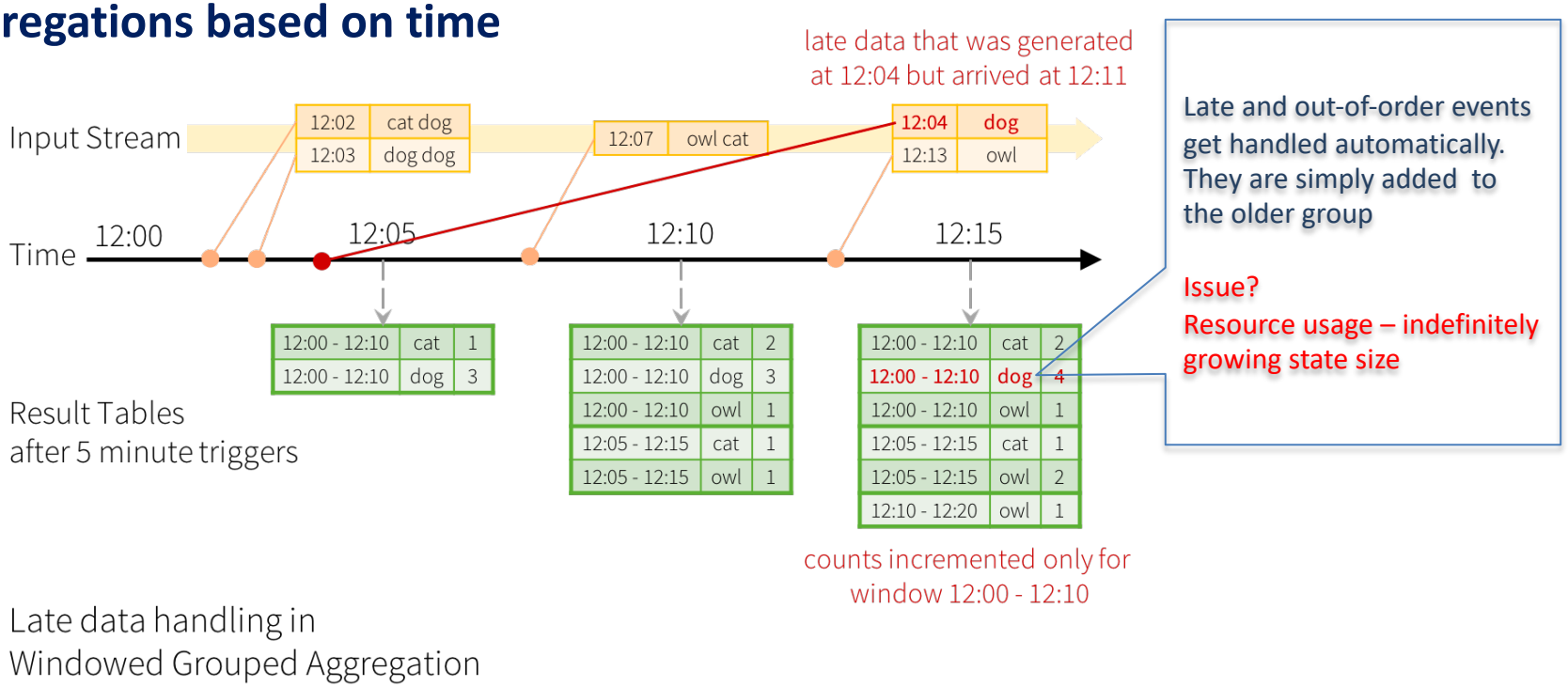
Stateful Streaming Aggregations

Aggregations based on time



Stateful Streaming Aggregations

Aggregations based on time



Late data handling in Windowed Grouped Aggregation

```
(sensorReadings.groupBy("sensorId", window("eventTime", "10 minute", "5 minute")).count())
```

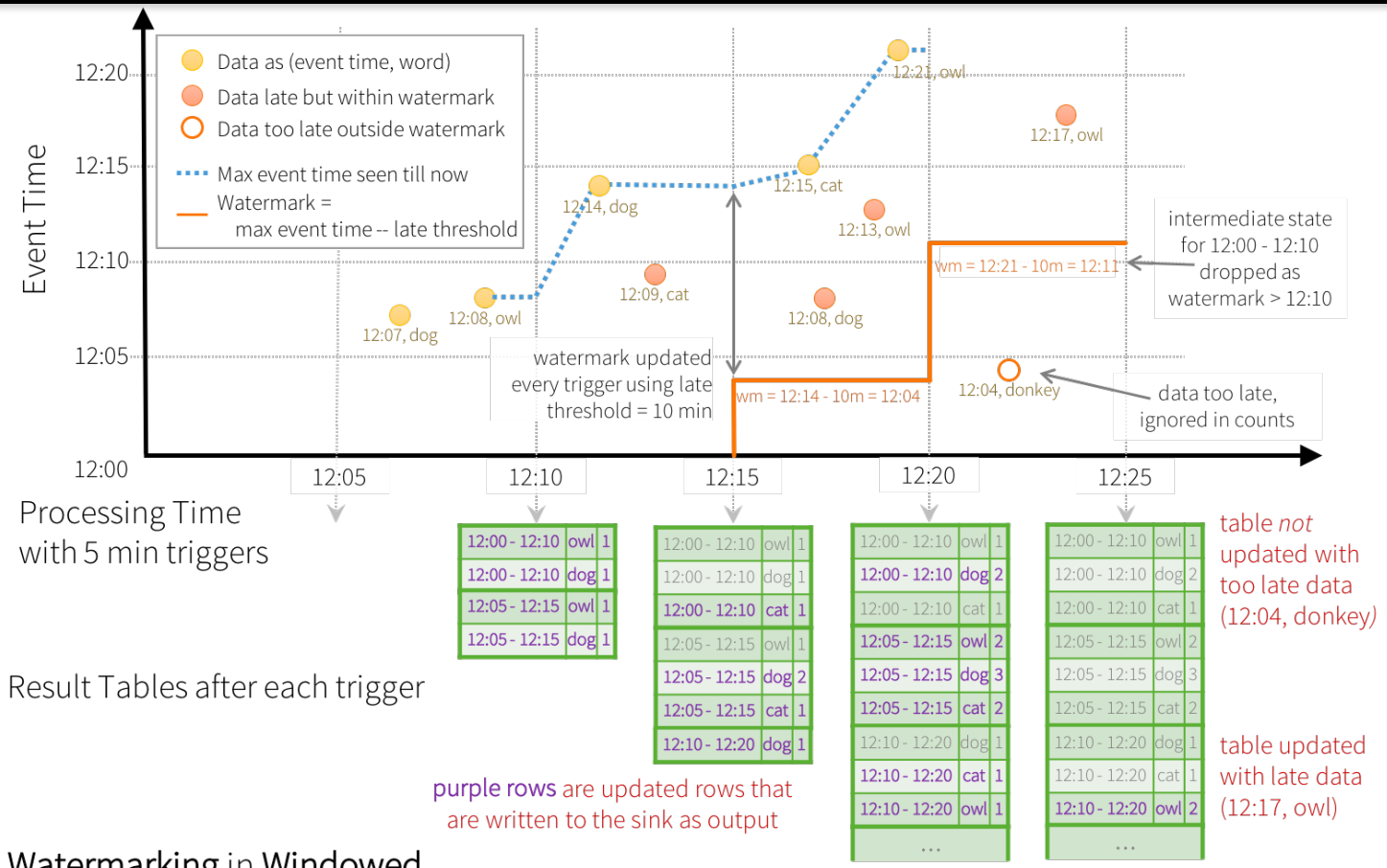
Handling Late Data

- *Watermark* – moving threshold in event time that trails behind the maximum event time seen by the query in the processed data.
- *Watermark delay* – the trailing gap. Defines how long the engine will wait for late data to arrive.
- Limiting the total amount of state that the engine has to maintain to compute the results of the query

If the sensor data will not be late by more than 10 minutes:

```
(sensorReadings.withWatermark("eventTime", "10 minutes")
.groupBy("sensorId", window("eventTime", "10 minutes", "5 minutes"))
.mean("value"))
```

Note: withWaterMark() will need to be called before the groupBy() and on the same timestamp column as used to define windows.



Watermarking in Windowed Grouped Aggregation with Update Mode

The maximum observed value of the eventTime column is tracked, and the watermark is updated accordingly. “Late” data are filtered and the old state is cleared.

Semantic Guarantees with Watermarks

A watermark of 10 minutes …

- Guarantees that the **engine will never drop any data** that is delayed by less than 10 minutes **compared to the latest event time** seen in the input data.
- The guarantee is strict only in one direction
- Data delayed by more than 10 minutes is **not** guaranteed to be dropped and may get aggregated.
  - Depends on the exact timing of when the record was received an when the micro-batch processing it was triggered.

Supported Output Modes for Streaming Aggregation

Update mode

- Every micro-batch will output only the rows where the aggregate got updated.
- Can be used with all types of aggregations.
  - With time window aggregations, watermarking will ensure that the state will get cleaned up regularly.
  - Most useful and efficient mode to run queries with streaming aggregations.
  - Cannot use this mode to write aggregates to append-only streaming sinks (file-based formats including Parquet, ORC, etc.)

Supported Output Modes for Streaming Aggregation

Complete Mode

- Every micro-batch will output all the updated aggregates, irrespective of their age or whether they contain changes.
- For time window aggregations, state will not be cleaned up even if a watermark is specified.
- Can lead to an indefinite increase in state size and memory usage

Supported Output Modes for Streaming Aggregation

Append Mode

- Does not allow previously output results to change.
- Can be used only with aggregations on event-time windows and with watermarking enabled.
  - For aggregation without watermarks, every aggregate may be updated with any future data
- Allows writing of aggregates to append-only streaming sinks (e.g. files)
- The output will be delayed by the watermark duration.

Streaming Joins

Structured Streaming supports joining a streaming Dataset with another static or streaming Dataset.

Streaming Joins

Stream-Static Joins

```
impressionStatic = spark.read … # batch
clickstream = spark.readStream  # streaming
matched = clickstream.join(impressionStatic, “adId”) # inner join
```

- Every micro-batch of clicks is inner-joined against the static impression table to generate the output stream of matched events.
- Outer joins:
  - Supports left outer join when the left side is streaming DataFrame
  - Supports right outer join when the right side is streaming DataFrame
  - Full outer and left outer with a Streaming DataFrame on the right are not supported => Q: Can you think of why?

```
matched = clicksStream.join(impressionsStatic, "adId", "leftOuter")
```

Streaming Joins

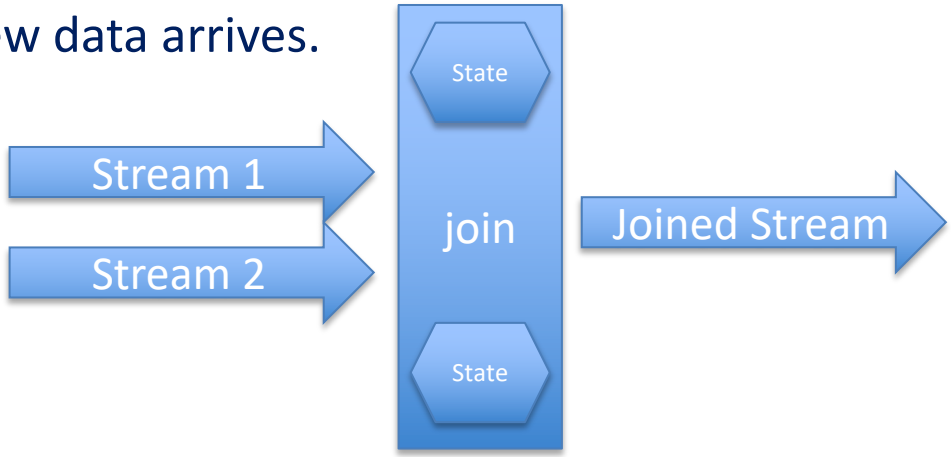
Stream-Static Joins

- Stream-static joins are stateless operations – do not require watermarking
- Static DataFrame is read repeatedly while joining with the streaming data of every micro-batch. Use cache to speed up the reads.
- If the underlying data in the data source on which the static DataFrame was defined changes, whether those changes are seen by the streaming query depends on the specific behavior of the data source.
  - The change may not be reflected until the streaming query is restarted.

Streaming Joins

Stream-Stream Joins

- Challenge of joins between two data streams:
  - At any point in time, the view of either Dataset is incomplete.
  - The matching events (records) from the two streams may arrive in any order and may be arbitrarily delayed.
- Structured streaming accounts for the delays by buffering the input data from both sides as the streaming state, and continuously checks for matches as new data arrives.



Streaming Joins

Stream-Stream Joins – Inner joins with optional watermarking

```
impressions = spark.readStream …
clicks  = spark.readStream …
matched = impressions.join(clicks, “adId”)
```

The engine buffers each stream as a state, and generates joined records as the buffered streams match.

Limiting the streaming state maintained by stream-stream joins

- The maximum time range between the generation of the two events at their respective sources
- The maximum duration an event can be delayed in transit between the source and the processing engine
- Use watermarks and time range conditions.

Streaming Joins

Additional steps in the join to ensure state cleanup:

1. Define watermark delays on both inputs
2. Define a constraint on event time across the two inputs
   - Helps the engine to figure out when old rows of one input are not going to be required.
   - Time range join conditions (e.g., join condition = "leftTime BETWEEN rightTime AND rightTime + INTERVAL 1 HOUR")
   - Join on event-time windows (e.g., join condition = "leftTimeWindow = rightTimeWindow")

```
impressionsWithWatermark = (impression.selectExpr("adId AS impressionAdId", "impressionTime")
.withWatermark("impressionTime", "2 hours"))

clicksWithWatermark = (clicks.selectExpr("adId AS clickAdId", "clickTime").withWatermark("clickTime", "3 hours"))

(impressionsWithWatermark.join(clicksWithWatermark,
expr("""clickAdId = impressionAdId AND clickTime BETWEEN impressionTime AND impressionTime + interval 1 hour""")))
```

Streaming Joins

Stream-Stream Joins – Outer joins with watermarking

```
# Left outer join with time range conditions
(impressionsWithWatermark.join(clicksWithWatermark,
expr("""
clickAdId = impressionAdId AND
clickTime BETWEEN impressionTime AND impressionTime + interval 1 hour"""),
"leftOuter")) # only change: set the outer join type
```

- Unlike with inner joins, the watermark delay and event-time constraints are not optional for outer joins.
- The outer NULL results will be generated with a delay
  - The engine has to wait for a while to ensure that there neither were nor would be any matches.
  - This delay is the maximum buffering time (with respect to event time) calculated by the engine for each event

Streaming Joins Support Matrix

Left Input	Right Input	Join Type	
Static	Static	All types	Supported, since its not on streaming data even though it can be present in a streaming query
Stream	Static	Inner	Supported, not stateful
		Left Outer	Supported, not stateful
		Right Outer	Not supported
		Full Outer	Not supported
Static	Stream	Inner	Supported, not stateful
		Left Outer	Not supported
		Right Outer	Supported, not stateful
		Full Outer	Not supported
Stream	Stream	Inner	Supported, optionally specify watermark on both sides + time constraints for state cleanup
		Left Outer	Conditionally supported, must specify watermark on right + time constraints for correct results, optionally specify watermark on left for all state cleanup
		Right Outer	Conditionally supported, must specify watermark on left + time constraints for correct results, optionally specify watermark on right for all state cleanup
		Full Outer	Not supported

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#support-matrix-for-joins-in-streaming-queries

Questions