Stream Processing

Lecture 3

2022/2023

Table of Contents

- Structured Streaming Programming
  - Fundamentals
  - Programming Model

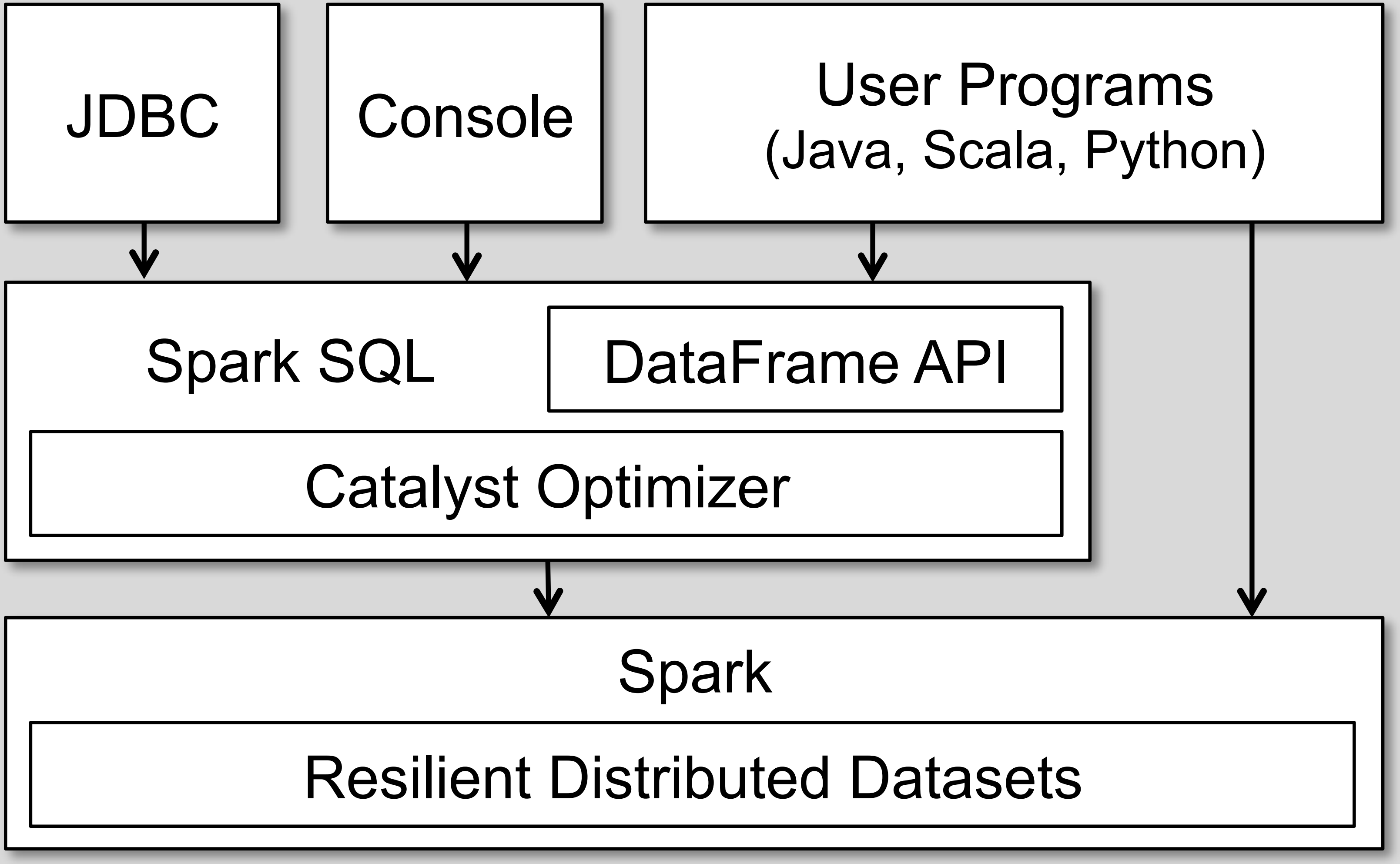
(Unstructured) Spark Streaming

- Interface
  - Discretized stream, with each mini-batch composed of RDDs
    - RDD: distributed collections.
    - RDDs manipulated through transformation operators (e.g., map, filter, reduce, etc.).
- Execution
  - Mini-batch of RDDs evaluated periodically.

Goals for Spark SQL

1. Support relational processing both within Spark programs and on external data sources using a programmer-friendly API.
2. Provide high performance using established DBMS optimization techniques.
3. Easily support new data sources, including semi-structured data and external databases amenable to query federation.
4. Enable extension with advanced analytics algorithms such as graph processing and machine learning.

SparkSQL Architecture



Spark DataFrames

- DataFrames are distributed collections of data that is grouped into named columns.
- DataFrames can be seen as RDDs **with a schema** that names the fields of the underlying tuples.
- How to create a DataFrame:
  - Import data from a file: JSON, CSV, parquet, etc.;
  - Import data from other systems: SQL DBs, Hive;
  - Convert a RDD into a DataFrame by supplying a suitable schema.

DataFrame Operations

- DataFrames provide a DSL for executing relational operations, as available in frameworks like Python Pandas.
- Some operations:
  - select(cols)
  - filter(condition)
  - join(RDD, on, how)
  - groupBy(cols)
  - sort(cols,)

Spark : DataFrame advantages

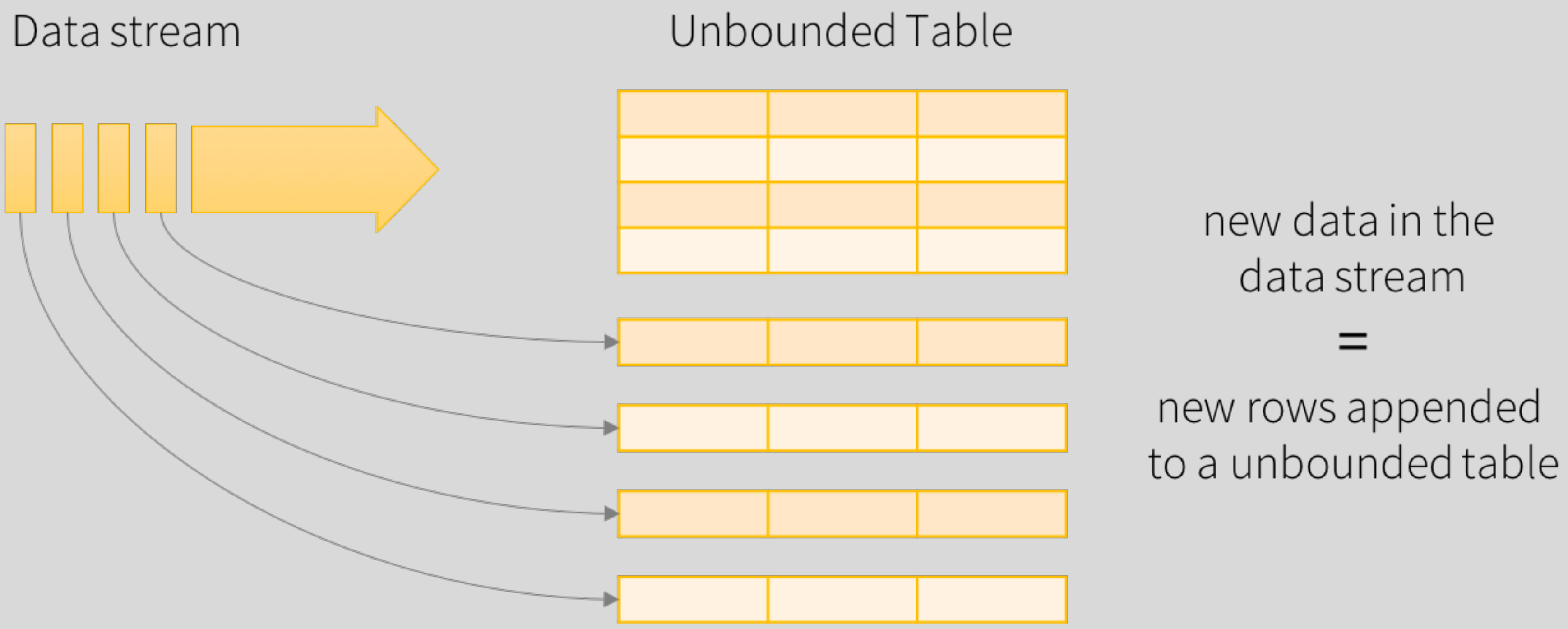
- Spark programs based on DataFrames are more readable due to its higher-level API.
- API close to relational operators of SQL.
- Some common programming patterns are exposed as high-level operations on DataFrames, also leading to shorter programs.

Pause

Structured Streaming

- Key idea is to treat a live data stream as a **table** that is being **continuously appended**.
  - Similar to the batch processing model.
- Express streaming computation as a standard batch-like query as on a static table, and Spark runs it as an *incremental* query on the *unbounded* input table.

Data stream model



Data stream as an unbounded table

Start Spark Structured Streaming

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

spark = SparkSession \
    .builder \
    .appName("StructuredWebLogExample") \
    .getOrCreate()


query = ... \     # some query definition
    .start()

query.awaitTermination(20)
query.stop()
```

Start Spark Structured Streaming

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

spark = SparkSession \
    .builder \
    .appName("StructuredWebLogExample") \
    .getOrCreate()


query = ... \     # some query definition
    .start()

query.awaitTermination(20)
query.stop()
```

Create a representation of a Spark session.

Start Spark Structured Streaming

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

spark = SparkSession \
    .builder \
    .appName("StructuredWebLogExample") \
    .getOrCreate()


query = ... \     # some query definition
    .start()

query.awaitTermination(20)
query.stop()
```

After defining a computation (see later), run start for start stream processing

Start Spark Structured Streaming

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

spark = SparkSession \
    .builder \
    .appName("StructuredWebLogExample") \
    .getOrCreate()


query = ... \     # some query definition
    .start()

query.awaitTermination(20)
query.stop()
```

Wait for the end of stream for 20 seconds and then stop.

First example

- Get data from the stream and print the data frames produced.

Input sources

- **File source** - Reads files written in a directory as a stream of data.
- **Kafka source** - Reads data from Kafka.
- **Socket source (for testing)** - Reads UTF8 text data from a socket connection. No fault-tolerance guarantees.

Connect to a stream

- **readStream**
- Read a stream.
- For a socket, specify **host** and **port**.

```
# Create DataFrame representing the stream of input
# lines from connection to logsender 7776
lines = spark.readStream.format("socket") \
    .option("host", "logsender") \
    .option("port", 7776) \
    .load()
```

Output sinks

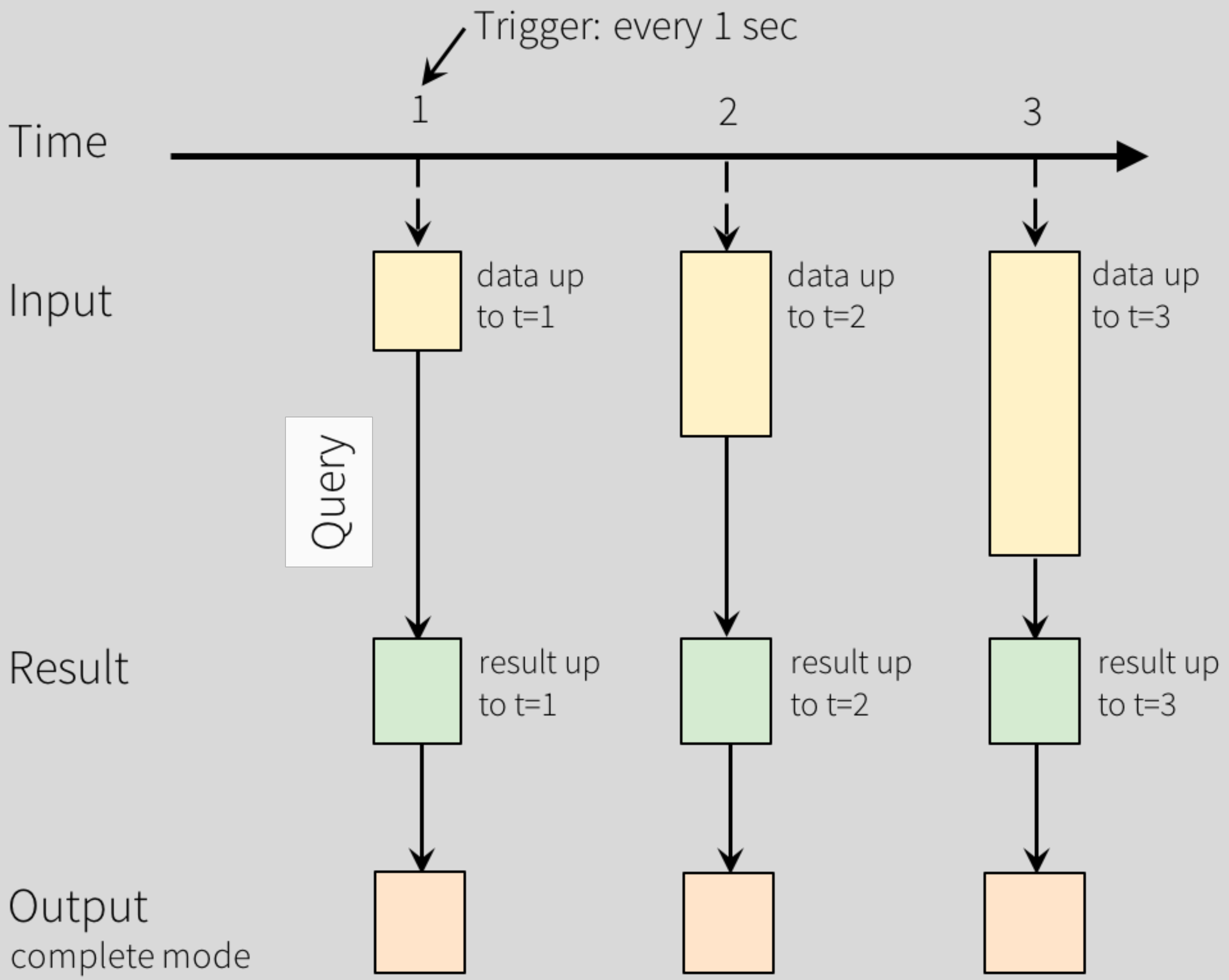
- **File sink** - Stores the output to a directory.
- **Kafka sink** - Stores the output to one or more topics in Kafka.
- **Console sink (for debugging)** - Prints the output to the console/stdout every time there is a trigger.

  Does not integrate well with Jupyter.
- **Foreach sink** - Runs arbitrary computation on the records of the output..

Output modes

- ***Complete Mode*** - The entire updated Result Table will be written to the external storage.
- ***Append Mode*** - Only the new rows appended in the Result Table since the last trigger will be written to the external storage.
- ***Update Mode*** - Only the rows that were updated in the Result Table since the last trigger will be written to the external storage.

Execution model (cont.)



Programming Model for Structured Streaming

Output with foreach

- **writeStream**
- Write a stream to an output sink, with a given output mode.

```
def dumpBatchDF(df, epoch_id):
    df.show(20, False)

query = lines \
    .writeStream \
    .outputMode("append") \
    .foreachBatch(dumpBatchDF) \
    .start()
```

Output with foreach

- **foreach(function) / foreachBatch(function)**
- Specify the function to run for each data frame created

```
def dumpBatchDF(df, epoch_id):
    df.show(20, False)

query = lines \
    .writeStream \
    .outputMode("append") \
    .foreachBatch(dumpBatchDF) \
    .start()
```

Overall execution model

- Source provides rows that are appended to the Input Table every trigger interval.
- A query on the input will generate the “Result Table”.
- Whenever the result table gets updated, the changes can be sent to an external sink.

First example

- Get data from the stream and print the data frames produced.

- Get da
  frames

```
+-----+
|value|
+-----+
+-----+

+--------------------------------------------------------------------+
|value                                                               |
+--------------------------------------------------------------------+
|2020-03-15T10:00:00.000+0000 37.139.9.11 200 GET /date/10h00m00s 0.1|
+--------------------------------------------------------------------+

+---------------------------------------------------------------------+
|value                                                                |
+---------------------------------------------------------------------+
|2020-03-15T10:00:05.000+0000 37.139.9.12 200 GET /date/10h00m05s 0.12|
+---------------------------------------------------------------------+

+--------------------------------------------------------------------+
|value                                                               |
+--------------------------------------------------------------------+
|2020-03-15T10:00:10.000+0000 37.139.9.13 200 GET /date/10h00m10s 0.2|
+--------------------------------------------------------------------+

+--------------------------------------------------------------------+
|value                                                               |
+--------------------------------------------------------------------+
|2020-03-15T10:00:15.000+0000 37.139.9.14 200 GET /date/10h00m15s 0.1|
+--------------------------------------------------------------------+

+--------------------------------------------------------------------+
|value                                                               |
+--------------------------------------------------------------------+
|2020-03-15T10:00:17.000+0000 37.139.9.24 200 GET /date/10h00m17s 0.1|
+--------------------------------------------------------------------+
```

- Get da[...]
  frames

```
+-----+
|value|
+-----+
+-----+

+----------------------------------------------------------------+
|value                                                           |
+----------------------------------------------------------------+
|2020-03-15T10:00:00.000+0000 37.139.9.11 200 GET /date/10h00m00s 0.1|
+----------------------------------------------------------------+

+-----------------------------------------------------------------+
|value                                                            |
+-----------------------------------------------------------------+
|2020-03-15T10:00:05.000+0000 37.139.9.12 200 GET /date/10h00m05s 0.12|
+-----------------------------------------------------------------+

+----------------------------------------------------------------+
|value                                                           |
+----------------------------------------------------------------+
|2020-03-15T10:00:10.000+0000 37.139.9.13 200 GET /date/10h00m10s 0.2|
+----------------------------------------------------------------+

+----------------------------------------------------------------+
|value                                                           |
+----------------------------------------------------------------+
|2020-03-15T10:00:15.000+0000 37.139.9.14 200 GET /date/10h00m15s 0.1|
+----------------------------------------------------------------+

+----------------------------------------------------------------+
|value                                                           |
+----------------------------------------------------------------+
|2020-03-15T10:00:17.000+0000 37.139.9.24 200 GET /date/10h00m17s 0.1|
+----------------------------------------------------------------+
```

Each line leads to a data frame.

Second example

- Get data from the stream.
- List the top-3 IP sources with more accesses.

Create a data frame with a schema

- **split**
- Used to split a column in multiple value.

```
sl = split(lines['value'], ' ')
lines = lines \
   .withColumn('time',sl.getItem(0).cast("timestamp")) \
   .withColumn('IP', sl.getItem(1).cast("string")) \
   .withColumn('code', sl.getItem(2).cast("integer")) \
   .withColumn('op', sl.getItem(3).cast("string")) \
   .withColumn('URL', sl.getItem(4).cast("string")) \
   .withColumn('dur', sl.getItem(5).cast("float")) \
   .drop('value')
```

Create a data frame with a schema

- **withColumn(col,value)**
- Adds a columns to a data frame.

```
sl = split(lines['value'], ' ')
lines = lines \
   .withColumn('time',sl.getItem(0).cast("timestamp")) \
   .withColumn('IP', sl.getItem(1).cast("string")) \
   .withColumn('code', sl.getItem(2).cast("integer")) \
   .withColumn('op', sl.getItem(3).cast("string")) \
   .withColumn('URL', sl.getItem(4).cast("string")) \
   .withColumn('dur', sl.getItem(5).cast("float")) \
   .drop('value')
```

Create a data frame with a schema

- **drop(col)**
- Drops a column.

```
sl = split(lines['value'], ' ')
lines = lines \
   .withColumn('time',sl.getItem(0).cast("timestamp")) \
   .withColumn('IP', sl.getItem(1).cast("string")) \
   .withColumn('code', sl.getItem(2).cast("integer")) \
   .withColumn('op', sl.getItem(3).cast("string")) \
   .withColumn('URL', sl.getItem(4).cast("string")) \
   .withColumn('dur', sl.getItem(5).cast("float")) \
   .drop('value')
```

Create a data frame with a schema (alternative)

- **select(expr)**
- Creates a data frame from other data frame.

```
lines = lines.select( \
    split(lines.value, ' ')[0].alias('time').cast("timestamp"), \
    split(lines.value, ' ')[1].alias('IP').cast("string"),\
    split(lines.value, ' ')[2].alias('code').cast("integer"),\
    split(lines.value, ' ')[3].alias('op').cast("string"),\
    split(lines.value, ' ')[4].alias('URL').cast("string"),\
    split(lines.value, ' ')[5].alias('dur').cast("float"),\
    )
```

Operation: groupBy

- **groupBy(cols)**
- Groups the DataFrame using the specified columns, to run aggregation on them.

```
query = lines.groupBy('IP') \
    .count() \
    .orderBy('count',ascending=False) \
    .limit(3)
```

Operation: count

- **count()**
- Adds a column with the count (for each IP).

```
query = lines.groupBy('IP') \
    .count() \
    .orderBy('count',ascending=False) \
    .limit(3)
```

Operation: agg

- **agg()**
- Execute a general aggregation. E.g.: .agg({"∗": "count"})

```
query = lines.groupBy('IP') \
    .agg(count('*').alias('count')) \
    .orderBy('count',ascending=False) \
    .limit(3)
```

Operation: orderBy

- **orderBy(cols,ascending=True|False)**
- Orders the rows by the given column(s).

```
query = lines.groupBy('IP') \
    .count() \
    .orderBy('count',ascending=False) \
    .limit(3)
```

Operation: limit

- **limit(num)**
- Limits the result count to the number specified.

```
query = lines.groupBy('IP') \
    .count() \
    .orderBy('count',ascending=False) \
    .limit(3)
```

Incremental execution

- Spark Streaming processing:
  - reads the latest available data from the input;
  - process the data incrementally to update the result;
  - Discards the input data, keeping only minimal data to update the result.
- No need to maintain running aggregation or reason about fault-tolerance and data consistency.

Other example

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

spark = SparkSession \
    .builder \
    .appName("StructuredNetworkWordCount") \
    .getOrCreate()
```

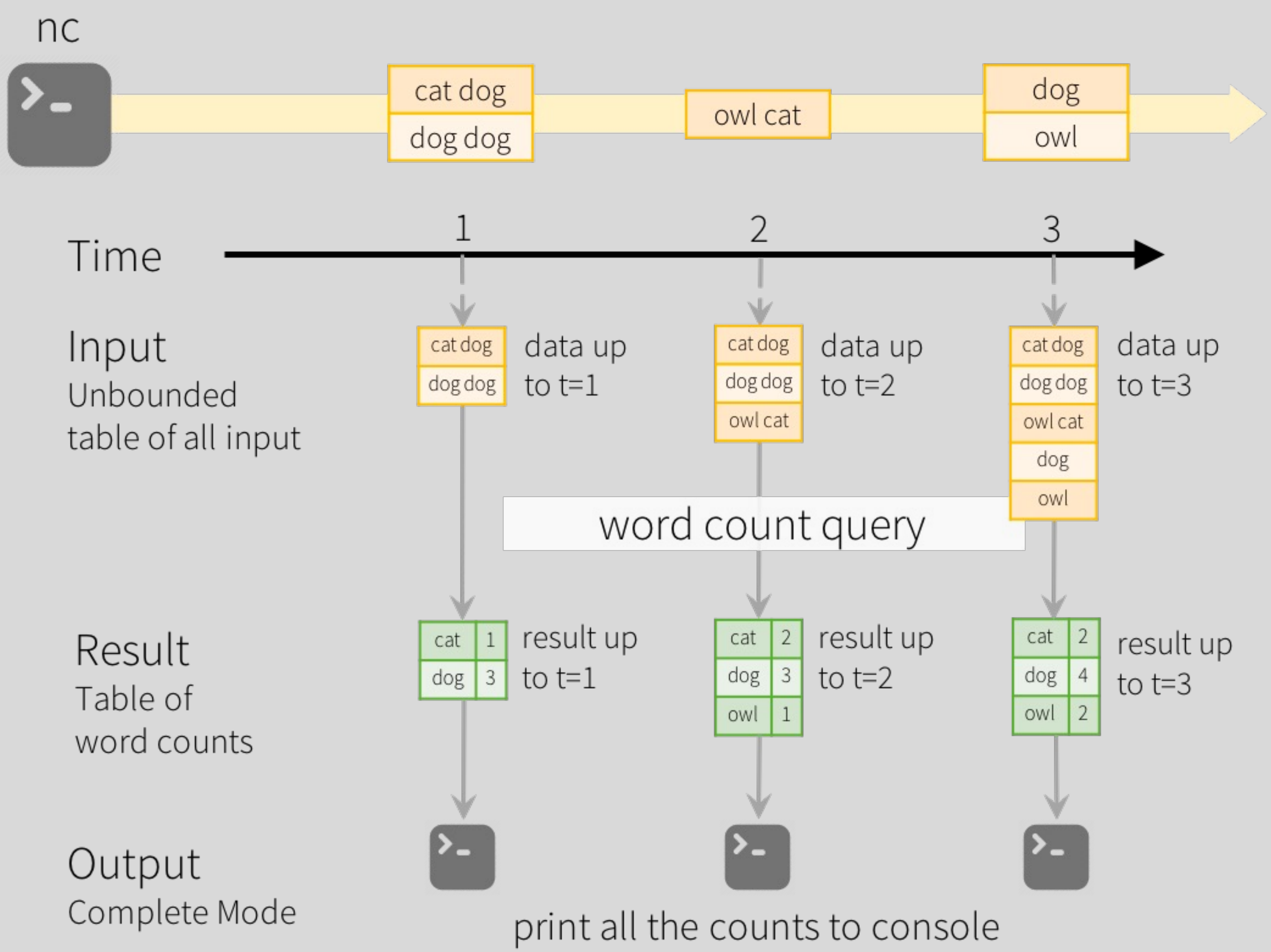
Example (cont)

```
# Create DataFrame representing the stream of input
# lines from connection to localhost:9999
lines = spark.readStream.format("socket") \
        .option("host", "localhost").option("port", 9999) \
        .load()

# Split the lines into words
words = lines.select(explode( split(lines.value, " ")) \
            .alias("word") )

# Generate running word count
wordCounts = words.groupBy("word").count()
```

Example (cont)

```
# Start running the query that prints the
# running counts to the console
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

Example (cont)



Model of the Quick Example

Pause

Exactly-once semantics

- Exactly-once semantics by combining:
  - Replayable sources
    - Spark uses checkpointing and write-ahead logs to record the offset range.
  - Idempotent sinks

- Why?
  - Failures impact on latency, but do not affect computation results, namely aggregations.
  - Deterministic results

Using SQL

- It is possible to use SQL by registering data frames as tables

Create a view from a data frame

- **createOrReplaceTempView(table)**
- Creates or replaces a local temporary view with this DataFrame.

```
lines.createOrReplaceTempView("weblog")
query = spark.sql("SELECT IP, count(*) as count FROM
weblog GROUP BY IP ORDER BY count DESC LIMIT 3")
```

Execute SQL statement

- **sql(stmt)**
- Executes a SQL statement.

```
lines.createOrReplaceTempView("weblog")
query = spark.sql("SELECT IP, count(*) as count FROM
weblog GROUP BY IP ORDER BY count DESC LIMIT 3")
```

Windows

- When executing aggregations, it is possible to execute computation over windows.

  – window aggregations based on event time are supported…

Define window

- **window(value,duration,slide)**
- Groups data in a window defined by the value, for duration time and slide time.

```
query = lines.groupBy( \
   window(lines.time,"30 seconds","10 seconds"),'IP') \
   .agg(count('*').alias('count')) \
   .orderBy('window','count',ascending=False) \
   .limit(3)
```

Define window

- **window(value,duration,slide)**
- Groups data in a window defined by the value, for duration time and slide time.

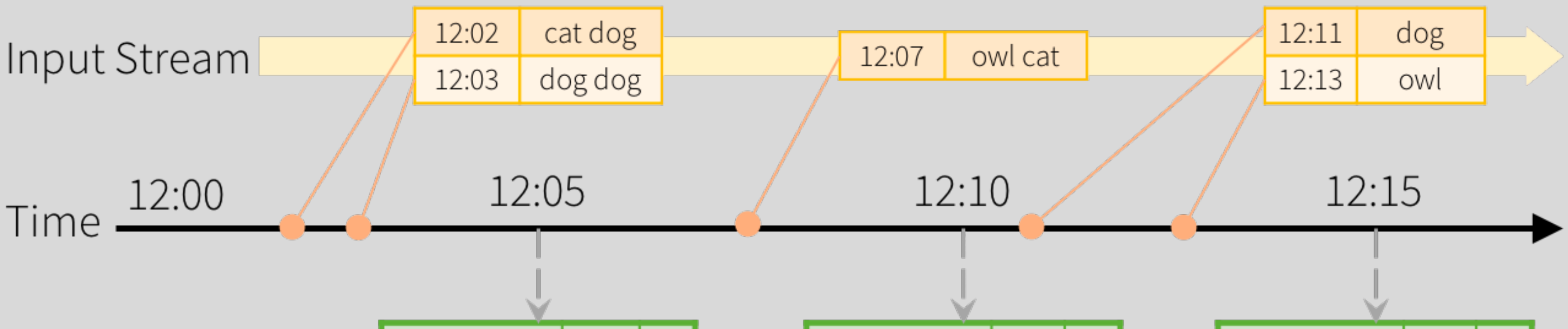
```
query = lines.groupBy( \
   window(lines.time,"30 seconds","10 seconds"),'IP') \
   .agg(count('*').alias('count')) \
   .orderBy('window','count',ascending=False) \
   .limit(3)
```

The results will have values for different windows.
Can use window to order the results.

Event time (example)

```
words = ...
# streaming DataFrame of schema
# { timestamp: Timestamp, word: String }

# Group the data by window and word
# and compute the count of each group
windowedCounts = words.groupBy( \
    window(words.timestamp, "10 minutes", "5 minutes"), \
    words.word
).count()
```

Event time (example, cont.)



Result Tables
after 5 minute triggers

12:00 - 12:10	cat	1
12:00 - 12:10	dog	3

12:00 - 12:10	cat	2
12:00 - 12:10	dog	3
12:00 - 12:10	owl	1
12:05 - 12:15	cat	1
12:05 - 12:15	owl	1

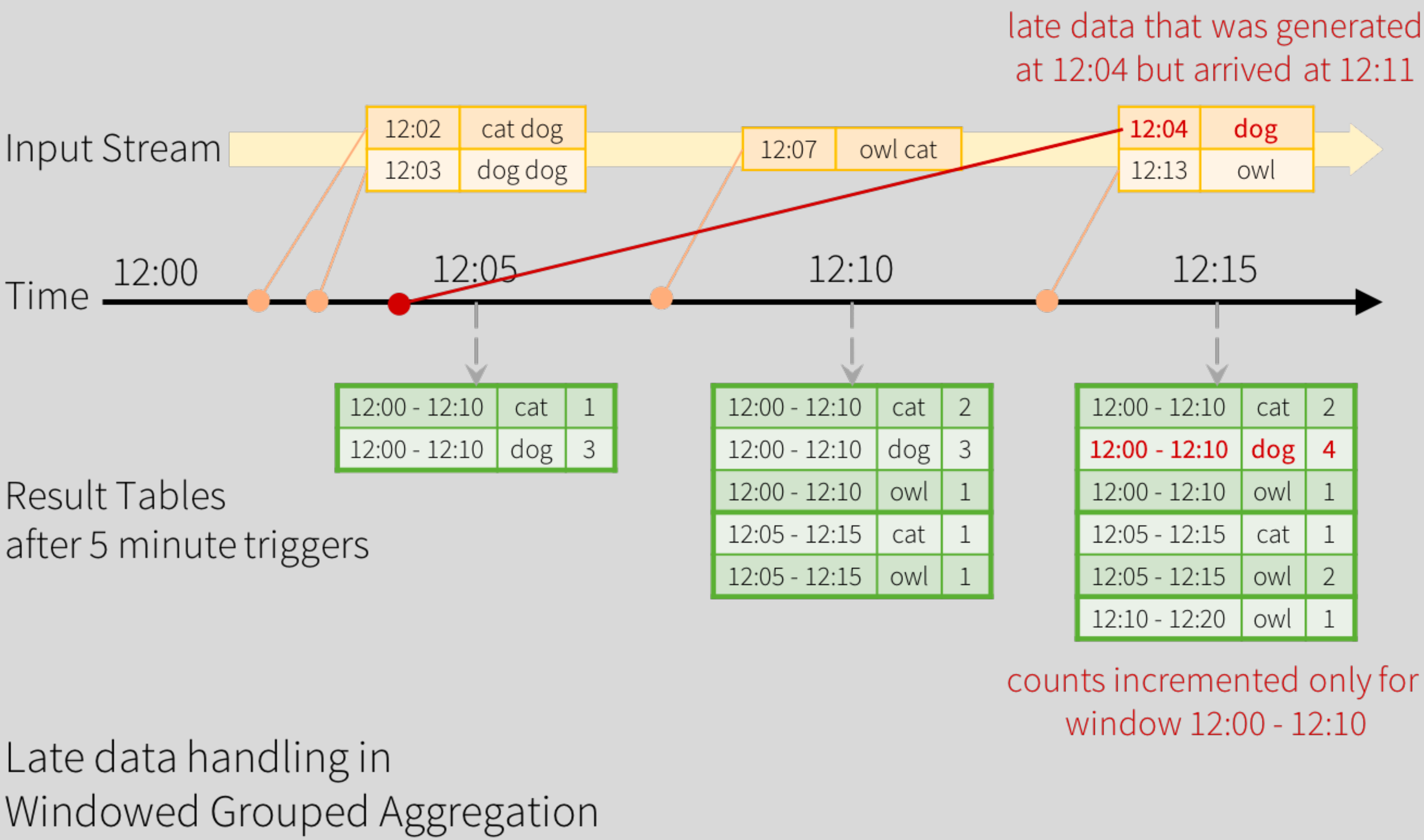
counts incremented for windows
12:00 - 12:10 and 12:05 - 12:15

12:00 - 12:10	cat	2
12:00 - 12:10	dog	3
12:00 - 12:10	owl	1
12:05 - 12:15	cat	1
12:05 - 12:15	owl	2
12:05 - 12:15	dog	1
12:10 - 12:20	dog	1
12:10 - 12:20	owl	1

counts incremented for windows
12:05 - 12:15 and 12:10 - 12:20

Windowed Grouped Aggregation
with 10 min windows, sliding every 5 mins

Handling late data



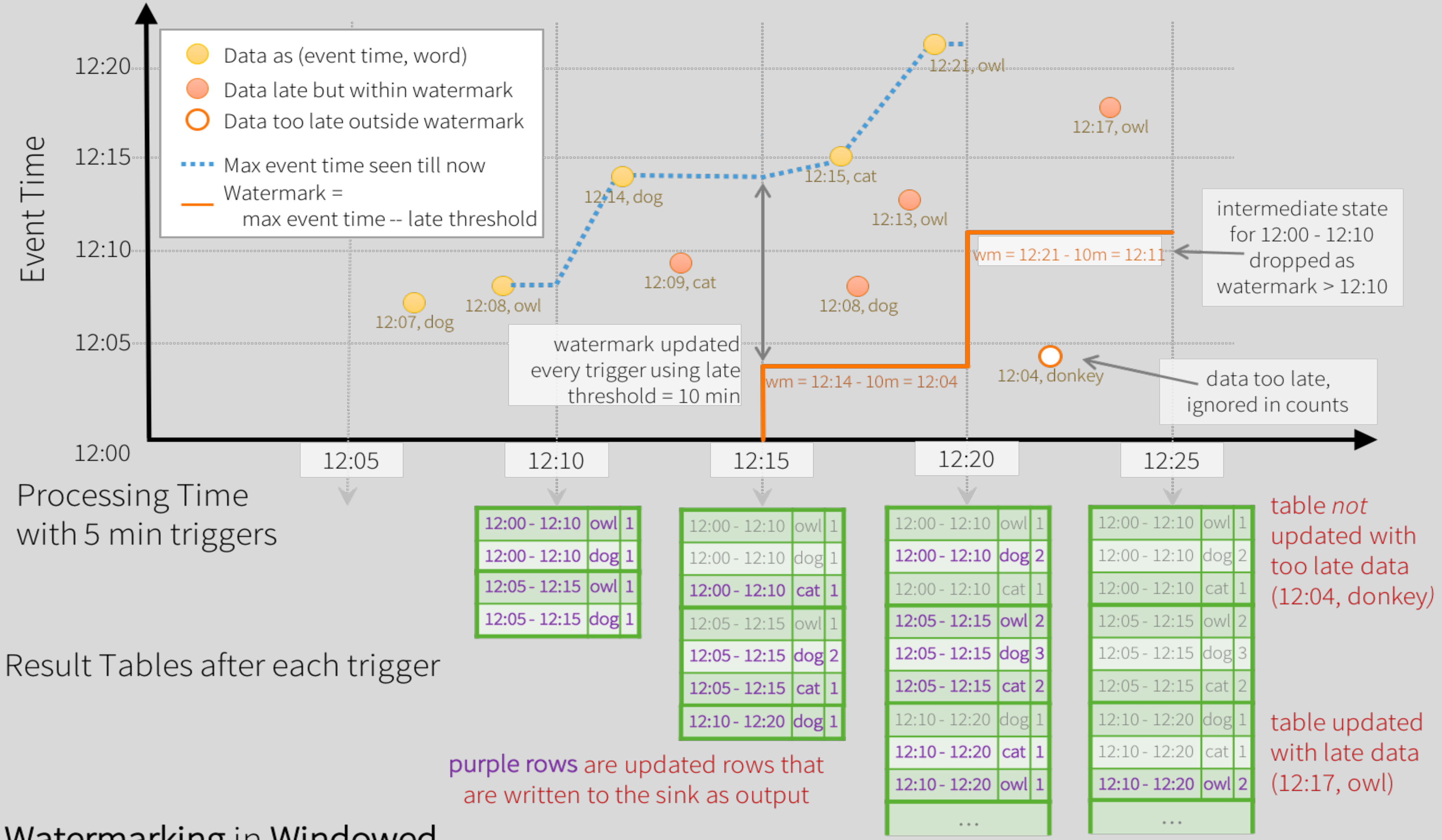
Handling late data (cont.)

- Problem?
  - Handling late data requires keeping data for as long we expect to receive late data…
- **Watermarking**: define the threshold on how late the data is expected to be in terms of event time
  - Late data within the threshold will be aggregated, but data later than the threshold will start getting dropped

Watermarking (example)

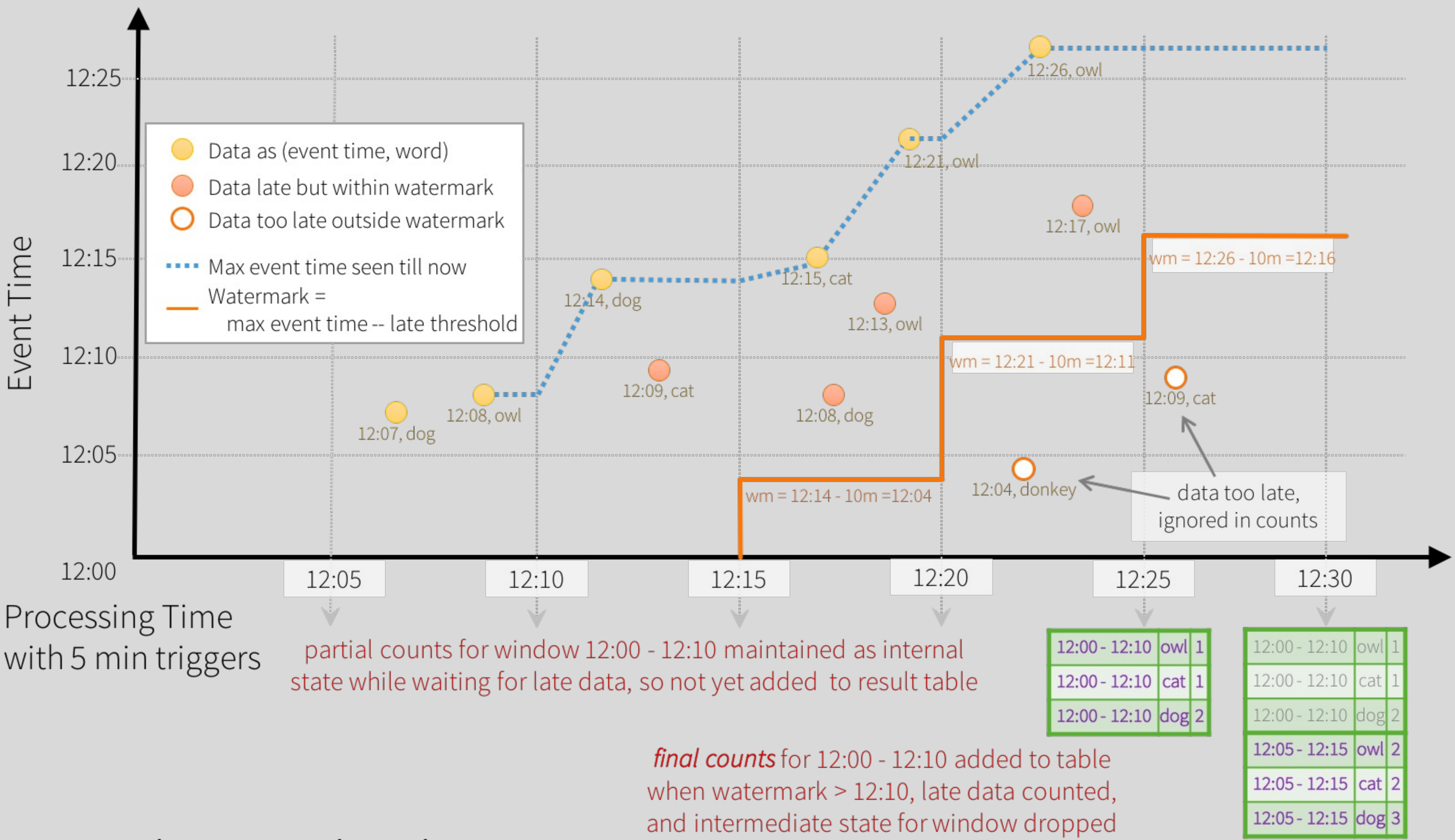
```
words = ...
# streaming DataFrame of schema
# { timestamp: Timestamp, word: String }

# Group the data by window and word
# and compute the count of each group
windowedCounts = words \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy( \
        window(words.timestamp, "10 minutes", "5 minutes"),\
        words.word) \
    .count()
```

Watermarking (example)



Watermarking in **Windowed Grouped Aggregation** with **Update Mode**

Watermarking (example)



Watermarking in **Windowed Grouped Aggregation** with **Append Mode**

Result Tables after each trigger

Join operations

- Supports joining:
  - a streaming Dataset/DataFrame with a static Dataset/DataFrame
  - a streaming Dataset/DataFrame with another streaming Dataset/DataFrame
- The result of the streaming join is generated incrementally.

Stream-static joins

- Create a static data frame from a file.

```
# Read the countries file
userSchema = StructType().add("IP", "string") \
   .add("country", "string")

countries = spark.read.schema(userSchema) \
   .csv("countries.csv")
```

Stream-static joins

- **join(df,condition,type)**
- Type join with df on condition.

```
query = query.join(countries,query.IP == countries.IP, \
   "inner")
```

Stream-stream join

- Challenge: at any given point, the view of the data is incomplete. Need to buffer past input for matching with new input.
- Solution: use watermarking to guarantee that data is not buffered forever.

Stream-stream joins

```
from pyspark.sql.functions import expr

impressions = spark.readStream. ...
clicks = spark.readStream. ...

# Apply watermarks on event-time columns
impressionsWithWatermark = impressions.withWatermark("impressionTime",
"2 hours")
clicksWithWatermark = clicks.withWatermark("clickTime", "3 hours")

# Join with event-time constraints
impressionsWithWatermark.join( \
  clicksWithWatermark, \
  expr("""
    clickAdId = impressionAdId AND
    clickTime >= impressionTime AND
    clickTime <= impressionTime + interval 1 hour
    """) \
)
```

Bibliography

- https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html