Stream Processing

Lecture 8

2018/2019

Exercises

- Combine two streams
- Combine a stream with an RDD
- Use the storedata

Table of Contents

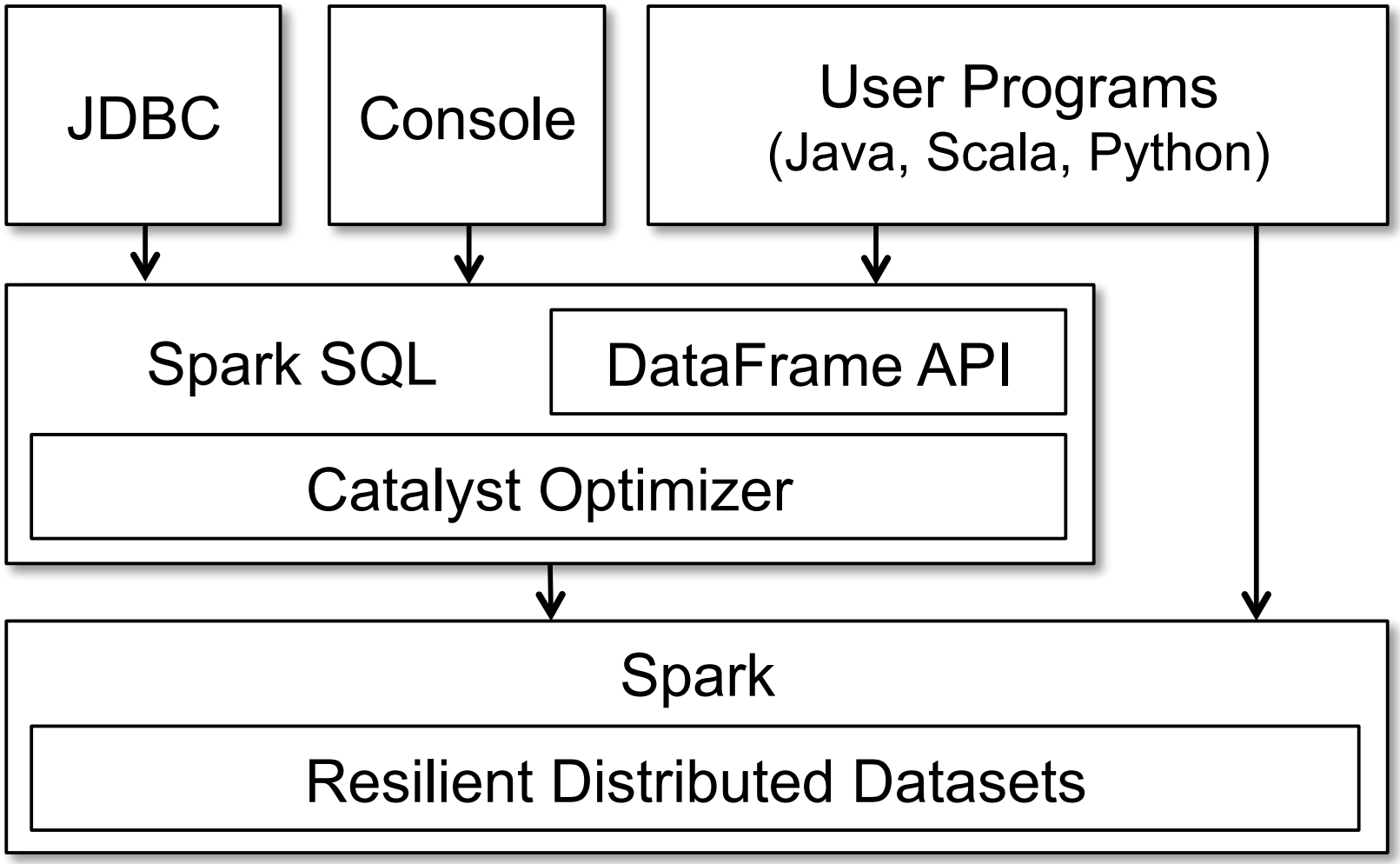
Spark Streaming

- 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,)

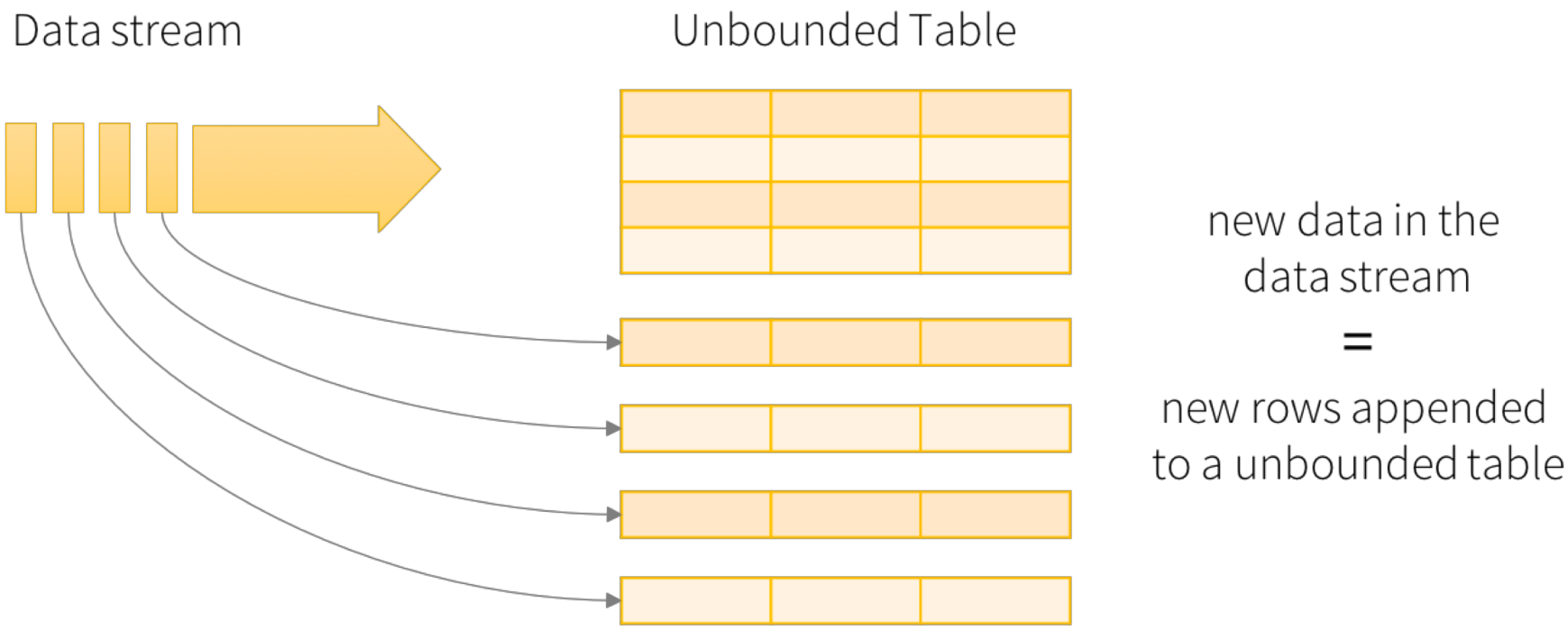
Spark : DataFrame advantages

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

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 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

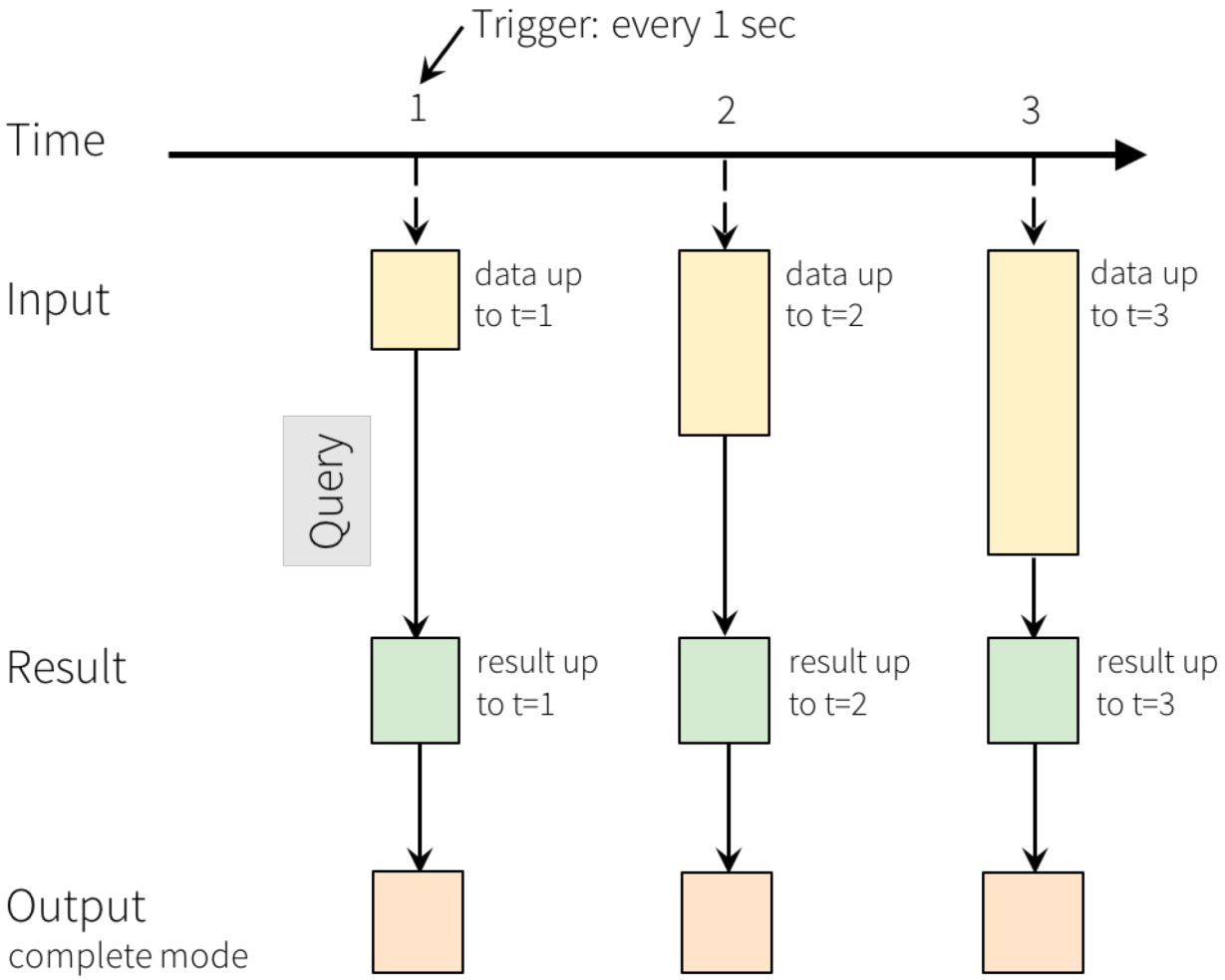


Data stream as an unbounded table

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.

Execution model (cont.)



Programming Model for Structured Streaming

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.

Example

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

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

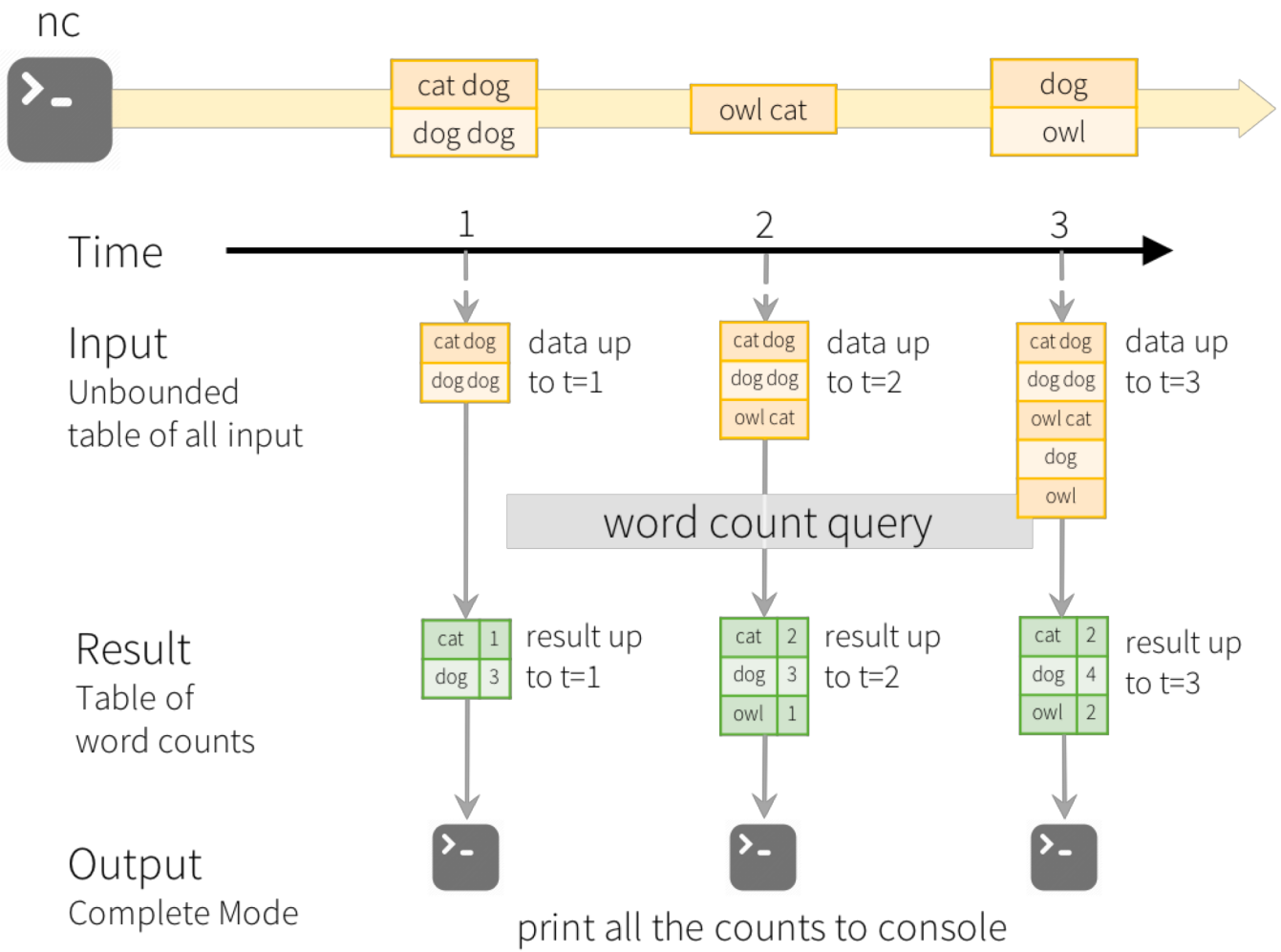
Example (cont)

```
# 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

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.

Exactly-once semantics

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

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.

Example

- In the example, we will consider a set of log files containing information for web accesses.

date IP_source return_value operation URL time

2016-12-06T08:58:35.318+0000 37.139.9.11 404 GET /codemove/TTCENCUFMH3C 0.026

Operation: select

`df.select(what).where(condition)`

- Select columns for rows matching a given condition

```
log = ...
# streaming DataFrame of schema
# { timestamp: Timestamp, IP: string, ... , time: float}

log.select("IP").where("time > 1.0")

log.groupBy("IP").count()
```

Operation: aggregation

`df.groupBy(what).`*`aggregation`*`()`

- Group rows by a given column and execute some aggregation function.

```
log = ...
# streaming DataFrame of schema
# { timestamp: Timestamp, IP: string, ... , time: float}

log.groupBy("IP").count()
```

Using SQL

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

```
df.createOrReplaceTempView(name)
```

- Create a view from a data frame

```
spark.sql(sql_statement)
```

Event time vs. received time

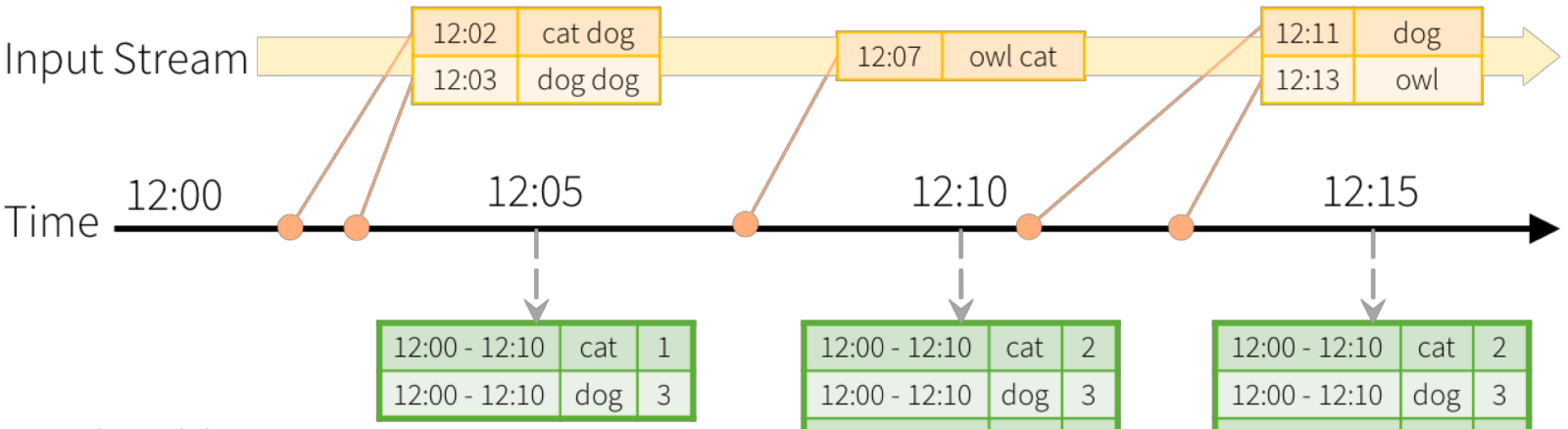
- Window operation allow to group elements based on event time

```
stream.groupBy( window(timestamp,
   windowDuration, slideDuration),what)
```

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.)



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

Result Tables
after 5 minute triggers

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

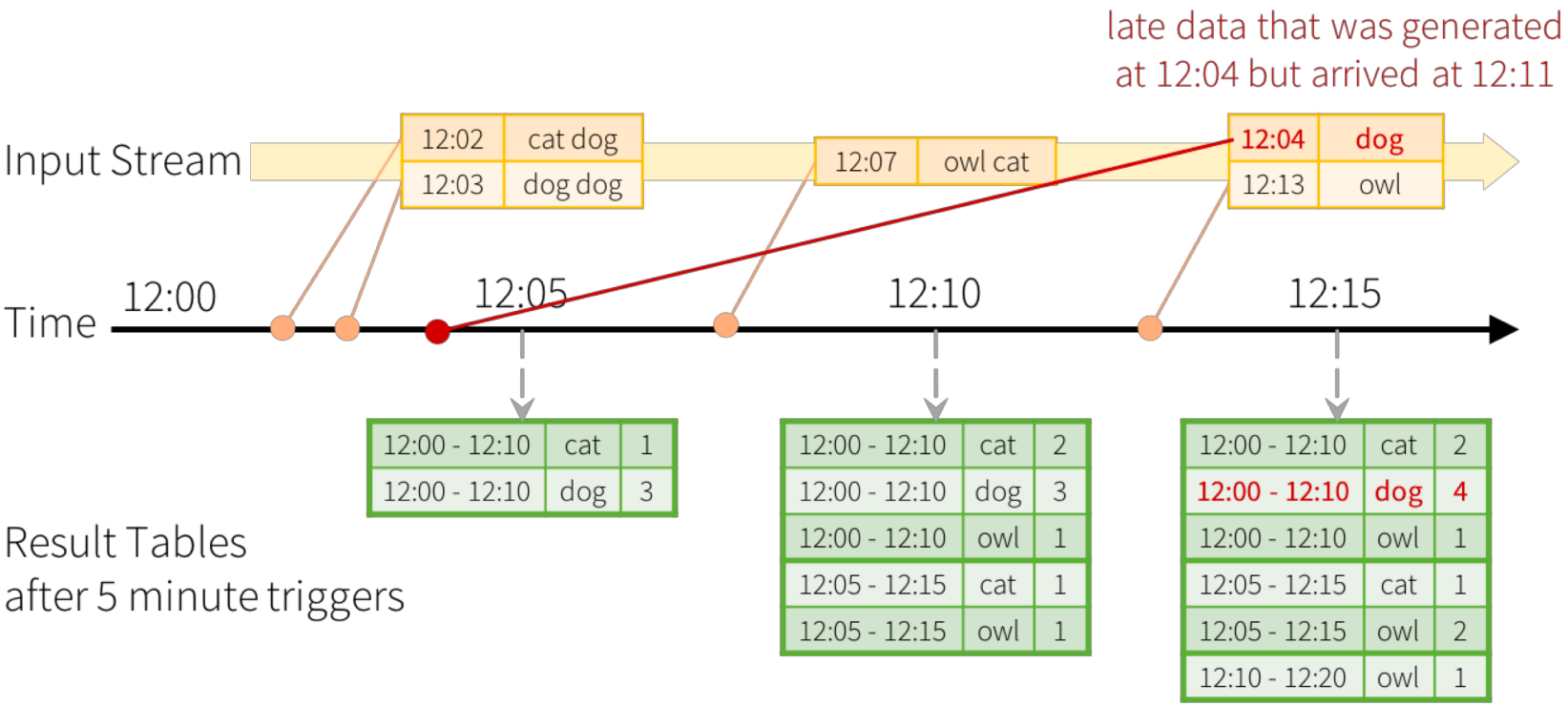
counts incremented for windows
12:00 - 12:10 and 12:05 - 12:15

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



Late data handling in
Windowed Grouped Aggregation

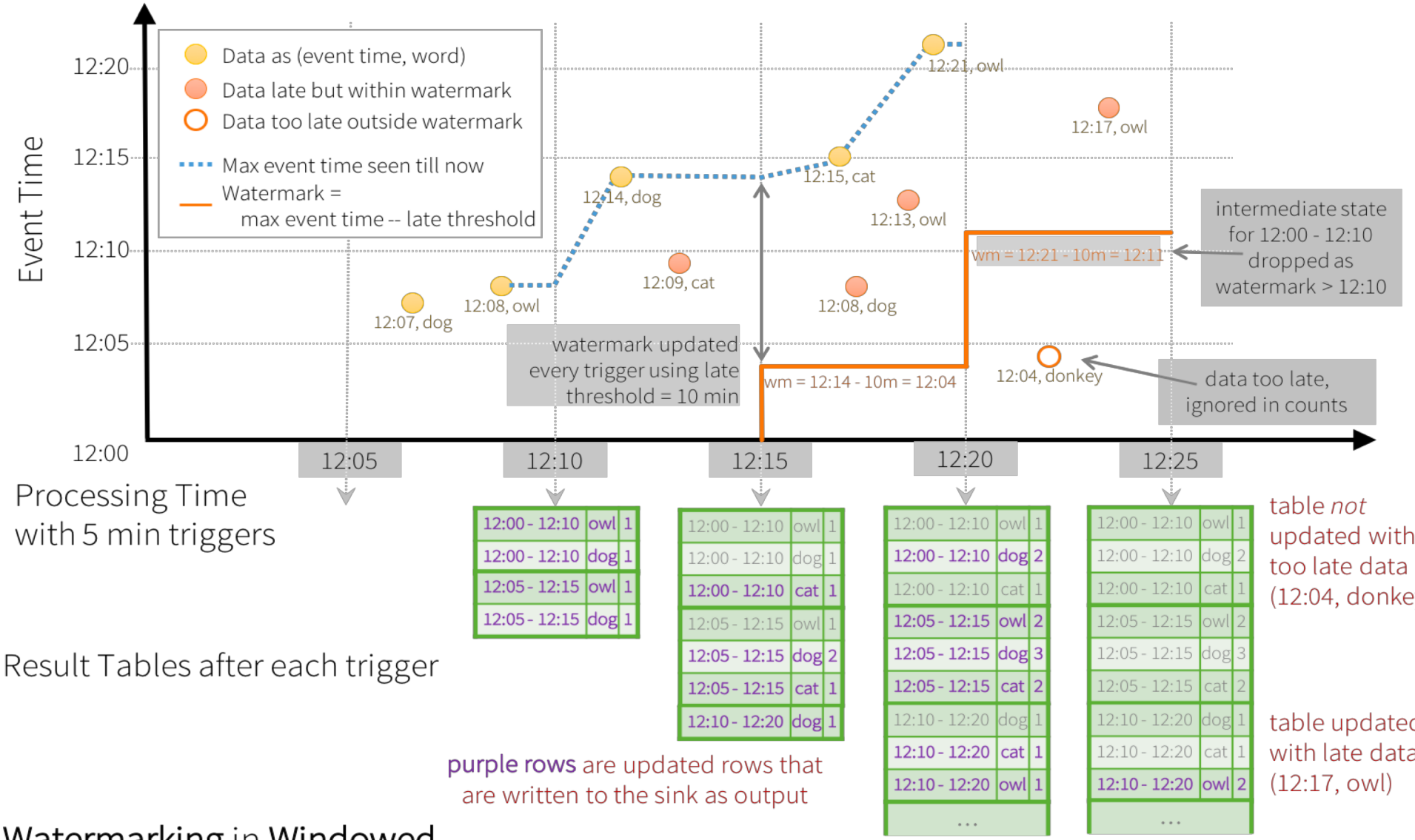
Handling late data (cont.)

- Problem?
  - Handling late data requires keeping data for
- 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)

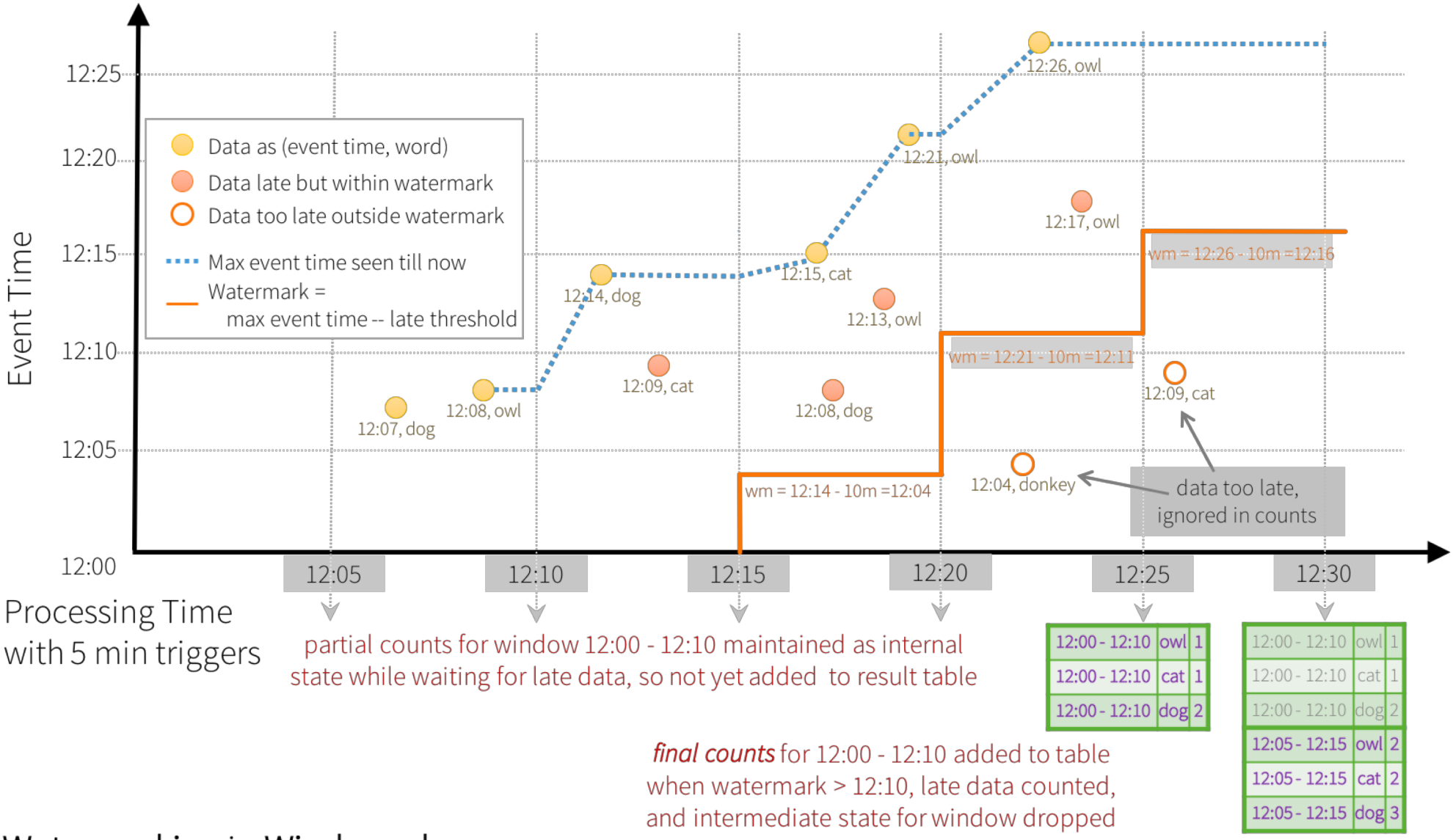


Result Tables after each trigger

purple rows are updated rows that are written to the sink as output

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

Watermarking (example)



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

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

```
staticDf = spark.read. ...
streamingDf = spark.readStream. ...

streamingDf.join(staticDf, "type")
    # inner equi-join with a static DF
streamingDf.join(staticDf, "type", "right_join")
    # right outer join with a static DF
```

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
    """) \
)
```

Output sink

- **File sink** - Stores the output to a directory.
- **Kafka sink** - Stores the output to one or more topics in Kafka.
- **Foreach sink** - Runs arbitrary computation on the records in the output.
- **Console sink (for debugging)** - Prints the output to the console/stdout every time there is a trigger.
- **Memory sink (for debugging)** - The output is stored in memory as an in-memory table.

Table of Contents

Example

```
import socket
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

ssc = StreamingContext(sc, 5)
lines = ssc.socketTextStream("localhost", 7777)

lines = lines.window(15,5)
counts = lines.filter( lambda line : len(line) > 0 ) \
            .map(lambda line: (line.split(' ')[1],1)) \
            .reduceByKey(lambda a, b: a+b) \
            .transform( \
                lambda rdd: rdd.sortBy(lambda x: x[1],
                    ascending=False))
counts.pprint()

ssc.start()
ssc.awaitTermination(120)
ssc.stop()
```

Exercise 1

- Dump the number of request per 5 seconds if there is an IP address with more than 100 requests

Exercise 2

- Assume you have a CSV file with the mapping of IP to country
- Return the number of access per country

Bibliography

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