POLITECNICO
MILANO 1863

Apache Spark

Alessandro Margara
alessandro.margara@polimi.it
https://margara.faculty.polimi.it

Spark Streaming

Stream processing

- Batch processing focuses on static datasets
  - Do not change over time

- Stream processing focuses on dynamic datasets
  - Continuously change over time
  - Model: new data is continuously appended to the stream
    - Example: stream of readings from a sensor

Stream processing

- Goal: continuously update the results of a computation as new data becomes available

- Requirements
  - Handle high rates of data generation
  - Produce new results with low latency
    - To enable timely reactions

Spark Streaming

- Extension of the original batch API to process streaming data

- Adopts a "micro-batch" approach
  - It splits the input streams into small batches
  - Batches are processed (almost) independently by the Spark engine
  - It produces a new result for each batch

Spark Streaming

- Other approaches use continuous processing
  - Operators are statically deployed rather than scheduled on demand
  - Data flows through the operators
  - It is processed as soon it becomes available

- Pro of micro-batching: dynamic adaptation is easier
  - Scheduling decisions can change over time
  - Elasticity: increase or decrease the number of resources
- Cons of micro-batching: higher processing delay

Spark Streaming API

- Spark Streaming's main abstraction is the *discretized stream* (*DStream*)

- Internally, a DStream is represented as a sequence of RDDs

- DStreams provide operations to transform the RDDs in the sequence
  - Also provides stateful operations that preserve internal state across invocations

Spark Streaming example

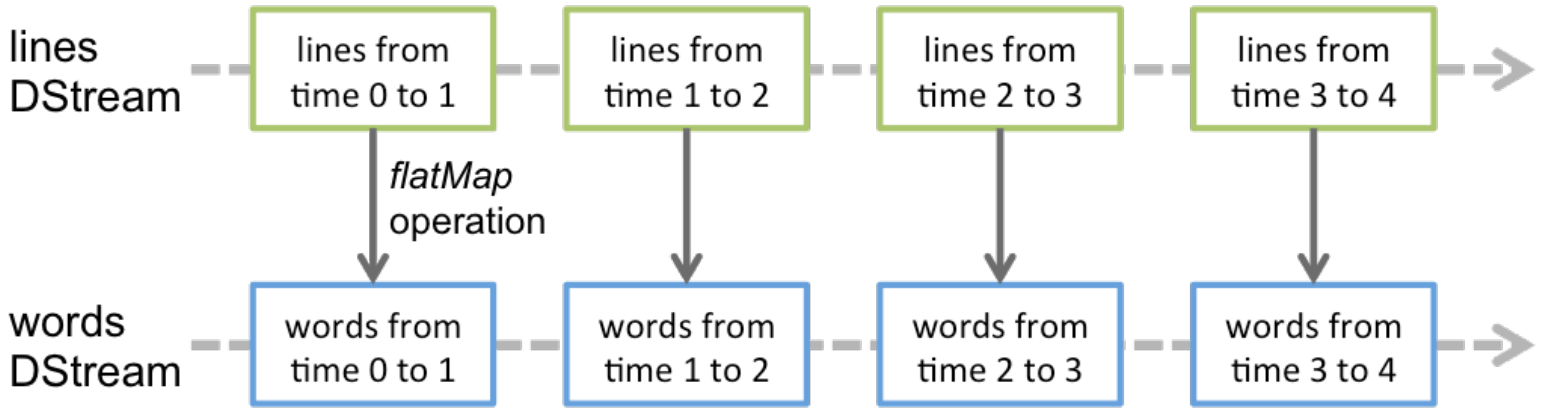
- As an example, consider again the word count application

- Assume we read streaming data from some source
  - E.g., TCP channel, Apache Kafka topic, …

- When applied in a streaming context, the count is performed separately on each RDD

Spark Streaming API

- Any operation applied on a DStream translates to operations on the underlying RDDs



DStreams transformations

- DStreams support most of the transformations available on normal RDDs
  - map, flatMap, filter, union, reduce, …
- These transformations are applied separately on each RDD of the DStream
  - See the streaming word count example

DStreams transformations

- Another class of interesting transformations are *stateful* operations
  - When processing an element, they preserve some state that can be subsequently accessed while processing further elements

- We will see three examples of stateful operations
  - updateStateByKey
  - mapWithState
  - windows

DStreams transformations

- updateStateByKey creates a *state* DStream
  - This is used to maintain a key-value store
  - The value is updated by applying a given function on the previous state of the key and the new state of the key

DStreams transformations

- Spark also enables state to be updated and used as part of a transformation
  - Example: mapWithState

- We can use this to change the semantics of the streaming word count application
  - The count is preserved and updated across RDDs

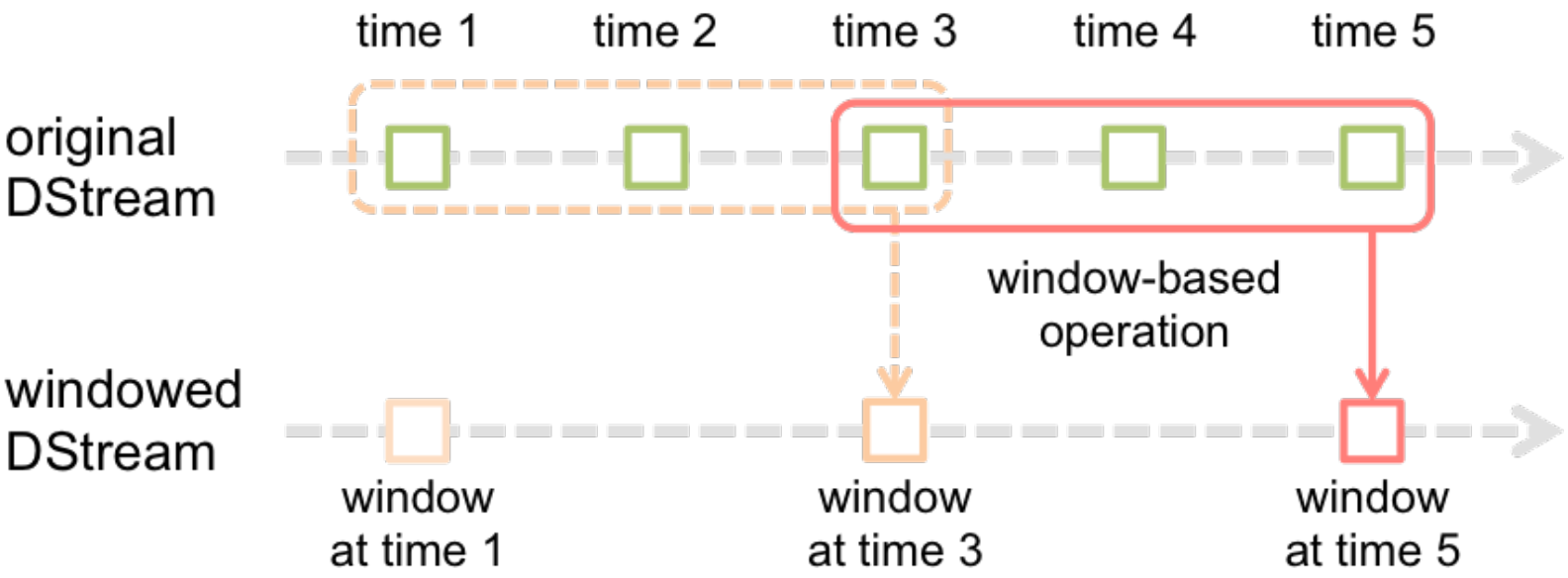
Windows

- Spark Streaming provides windowed computations, to apply transformations over a sliding window of data

- A window is defined in terms of two parameters
  - Window length: the duration of the window
  - Sliding interval: the interval (rate) at which the window operation is performed

- Note: these two parameters must be multiples of the batch interval of the source DStream

Windows


time 1
time 2
time 3
time 4
time 5
original DStream
window-based operation
windowed DStream
window at time 1
window at time 3
window at time 5

Windows

- Spark Streaming offers several operations to define windows and perform computations over windows
  - countByWindow
  - reduceByWindow
  - countByKeyAndWindow
  - reduceByKeyAndWindow
  - …

Fault tolerance semantics

- We already mentioned how Spark provides fault tolerance for RDDs
  - Spark Streaming also replicates the input streaming data

- In the case of failure, the system needs to recover
  1. Data received and stored (replicated): it survives the failure of a node as a copy exists in other nodes
  2. Data received but not yet replicated: the only way to recover this data is to get it again from the source, if possible
     - E.g., if the source is a Kafka topic

Structured Streaming

Structured Streaming

- Builds on the Spark SQL engine

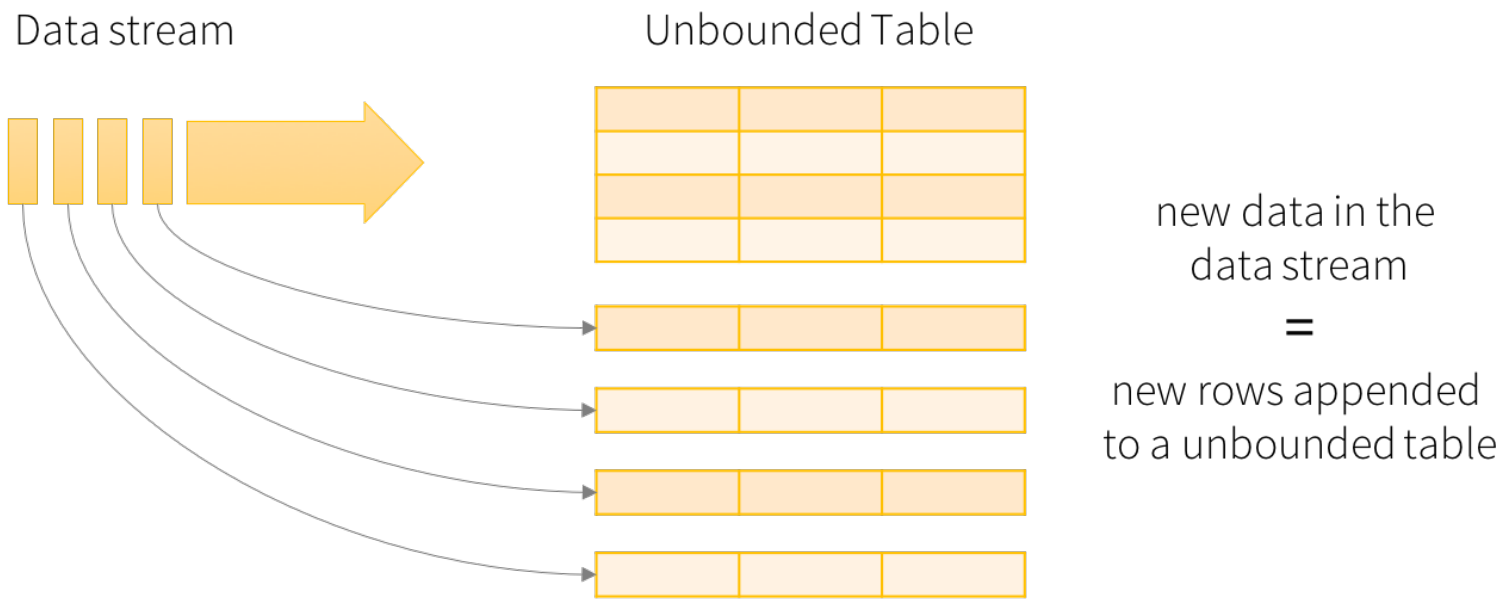
- Core ideas
  - Express streaming computations in the same way as batch computations on static data
  - The engine takes care of continuous and incremental execution to update the results as new data arrives

- Different programming abstraction …
- … same execution model
  - Internally it builds on the same micro-batch approach
  - But does not expose it to the end user

Programming model

- Key ideas
  - Consider a stream as a table that is being continuously appended
  - Express streaming computations as standard batch-like queries on static tables
  - Spark automatically updates the output tables

Programming model



Data stream as an unbounded table

Programming model

- A result / output table can be defined in different modes

1. Complete mode: returns the entire result table
2. Append mode: returns only the new rows appended to the result table since the last trigger
3. Update mode: returns only the rows that were updated in the result table since the last trigger

- The most suitable mode depends on the consumer of the data
  - Update mode can be used to update the results stored in a database
  - Complete mode can be used to periodically write the entire result on a file

Programming model: example

- Let us consider again the classic word count example

- To interact with the Spark SQL engine, we first need a SparkSession

```
SparkSession spark = SparkSession
       .builder()
       .master(master)
       .appName("StructuredStreamingWordCount")
       .getOrCreate();
```

Programming model: example

```
Dataset<Row> lines = spark
    .readStream()
    .format("socket")
    .option("host", socketHost)
    .option("port", socketPort)
    .load();


Dataset<String> words = lines
    .as(Encoders.STRING())
    .flatMap( … );


DataSet<Row> wordCounts = words
    .groupBy("value")
    .count();
```

- Variable `lines` represents an unbounded table containing streaming text data
  - One "value" column
  - Each line becomes a row in the table
- We convert the DataFrame into a Dataset of String
- Variable `wordCounts` is again a DataFrame containing the count for each word

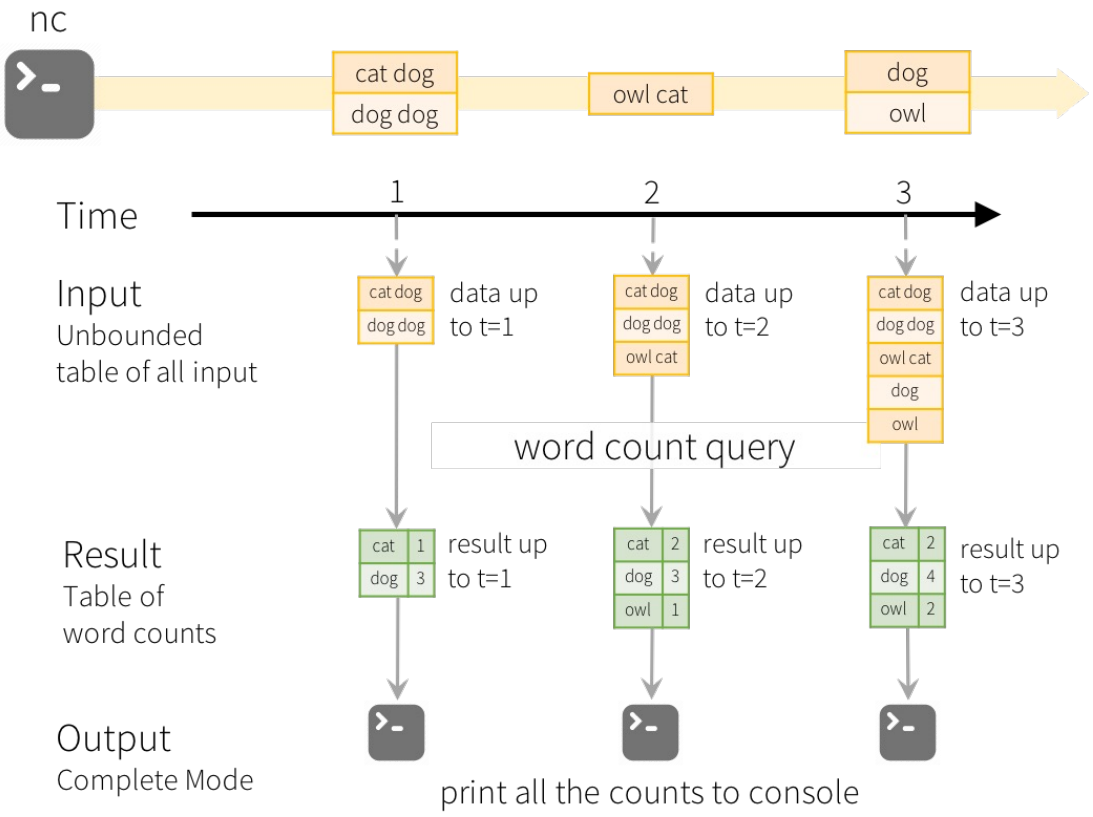
Programming model: example

- We can output the results using a query
  - We show the incremental computation model and the results with different modes

```
StreamingQuery query = wordCounts
        .writeStream()
        .outputMode("update")
        .format("console")
        .start();

query.awaitTermination()
```

Programming model: example



Model of the Quick Example

Incremental execution

- The engine does not materialize the entire table
- Instead, it *incrementally* updates the results upon receiving a new element from the input source
  - It only keeps the minimum intermediate state required to update the result

Time in stream processing

Time in stream processing

- Some operators rely on time (e.g., windows)

- But what is the meaning of time when running Spark in a distributed environment?
  - Different nodes in the cluster have different clocks
  - Sources and sinks have yet other internal clocks

Time in stream processing

- We can identify two "definitions" of time in stream processing
  1. Processing time: is the wall clock time of the processing node
  2. Event time: is the time attached to a data element by its source

Event time

- In most applications, event time is the most significant for the users
  - It is deterministic: in the case of replay, event time does not change and leads to the same results
  - It is set by the application
  - Does not depend on runtime concerns (e.g., the specific node where data is processed)

Event time

- However, event time is also the most complex to deal with

- In theory, we never know when we have all data up until some point in time
  - Data from multiple sources may arrive out of order
  - Data with an earlier event time may always arrive from some sources
  - We should wait forever for new data!

Event time and late data

- Structured streaming adopts an approach called retraction
  - It outputs data when new data is received
  - It changes the results in the case of late data
    - Retracts old results
  - It simply discards input data that is "too" old
    - To reduce the amount of old state to preserve

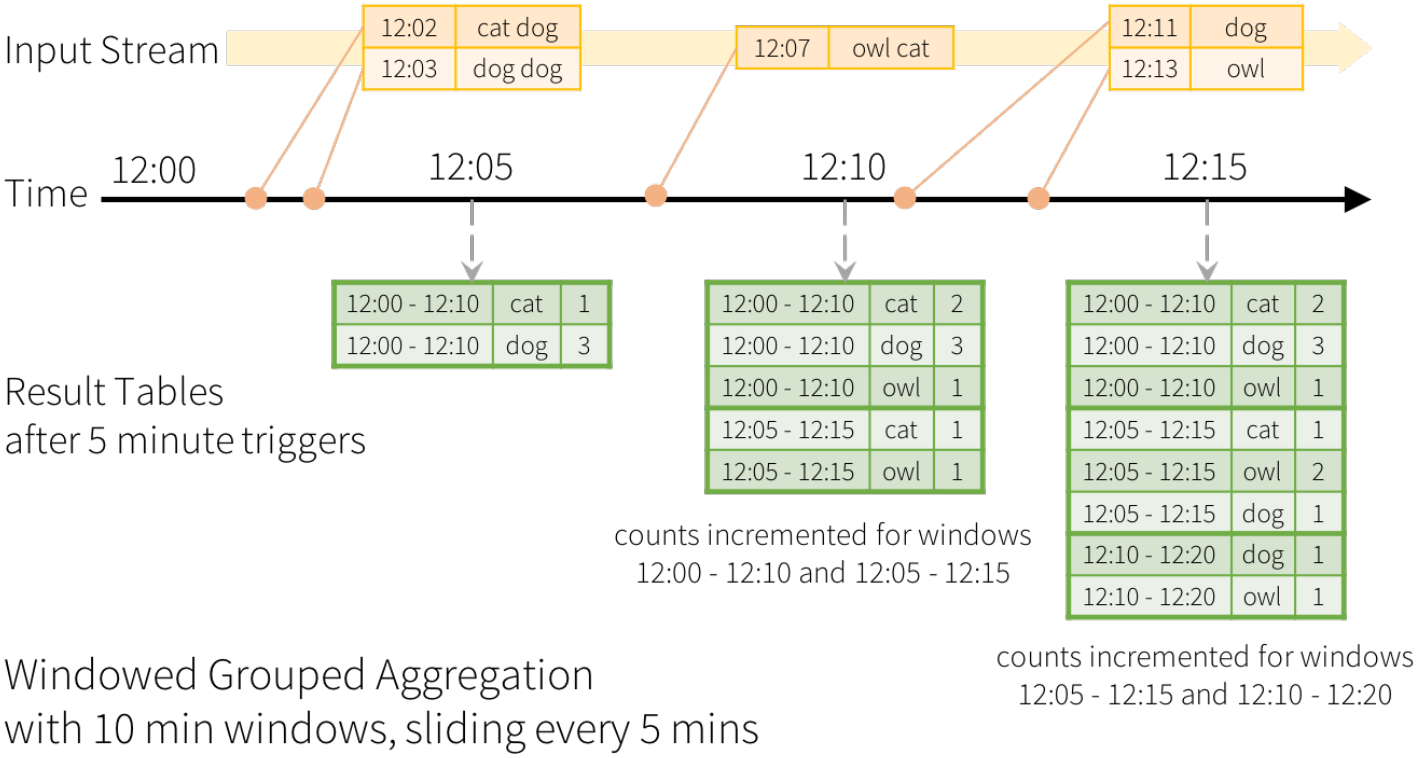
Event time and late data

- Window-based grouping is a good example to illustrate how Spark handles late data

- Spark maintains the intermediate state for partial aggregates for a long period of time, such that late data can update aggregates of old windows correctly

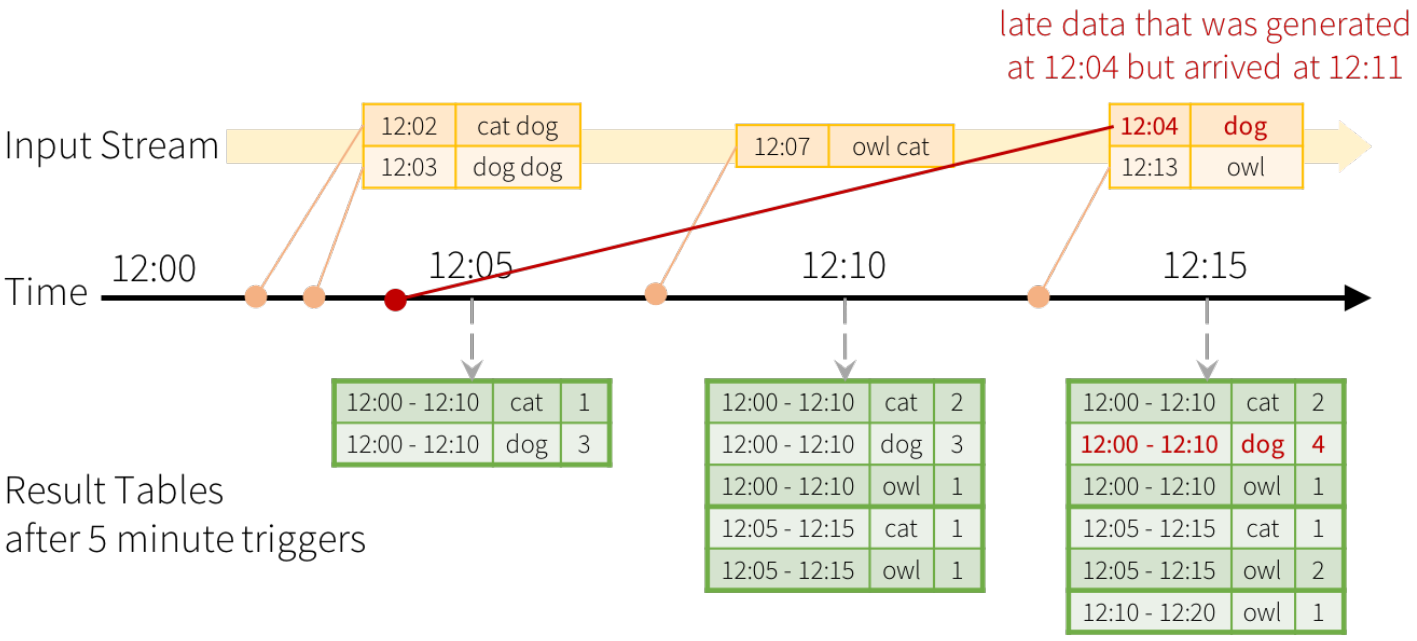
- After a threshold, late data is simply discarded
  - Garbage collection of old intermediate state

Windows and late data



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

Windows and late data



Late data handling in Windowed Grouped Aggregation