Spark Streaming

What is Streaming for you?

Data generated from a source and
transmitted to one or multiple receivers

Latency and Throughput

- These are the two metrics mostly used in streaming applications
- **LATENCY:** time required to perform some action or to produce some result (the lower the better)
- **THROUGHPUT:** number of such actions executed or results produced per unit of time (the higher the better)

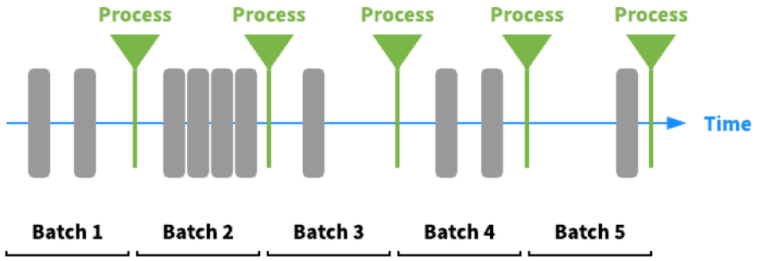
An assembly line is manufacturing cars. It takes 8 hours to manufacture a car and that the factory produces one hundred and twenty cars per day

LATENCY: 8 hours

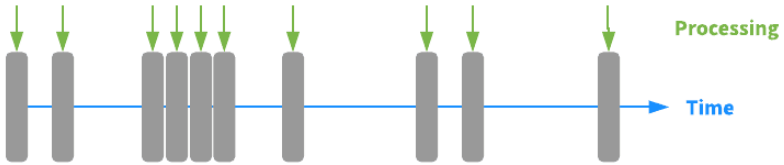
THROUGHPUT: 120 cars/day or 5 cars/hour

Batch, Micro-Batch, and Stream Processing

- **BATCH or MICRO-BATCH:** newly arriving data elements are collected into a group. The whole group is then processed at a future time

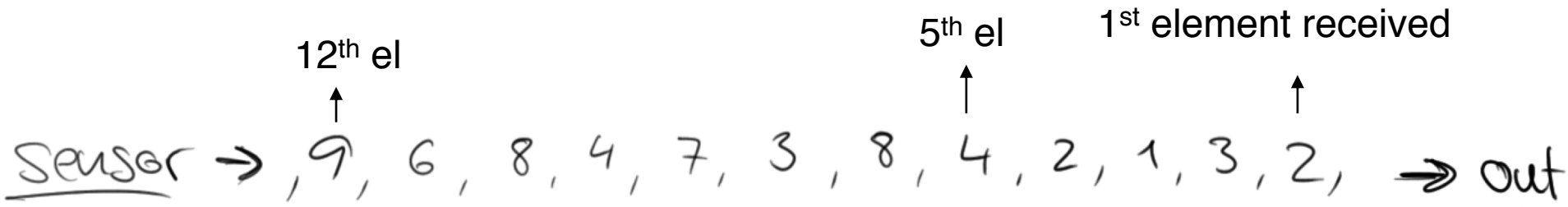


- **STREAM:** each new piece of data is processed when it arrives. Unlike batch processing, there is no waiting until the next batch processing interval and data is processed as individual pieces rather than being processed a batch at a time.



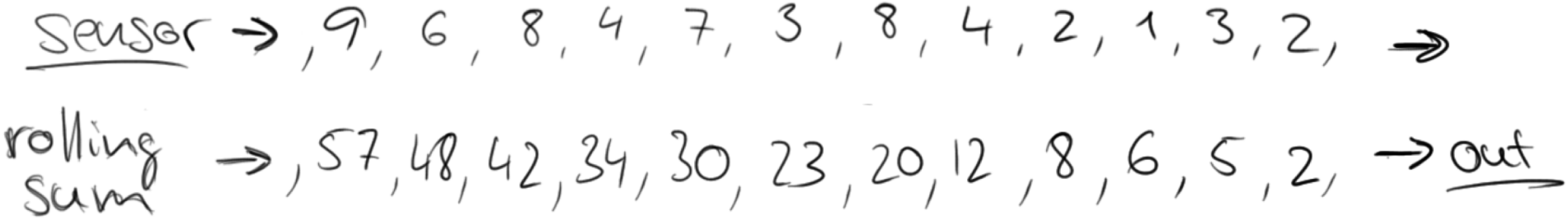
dbTrento

Windows

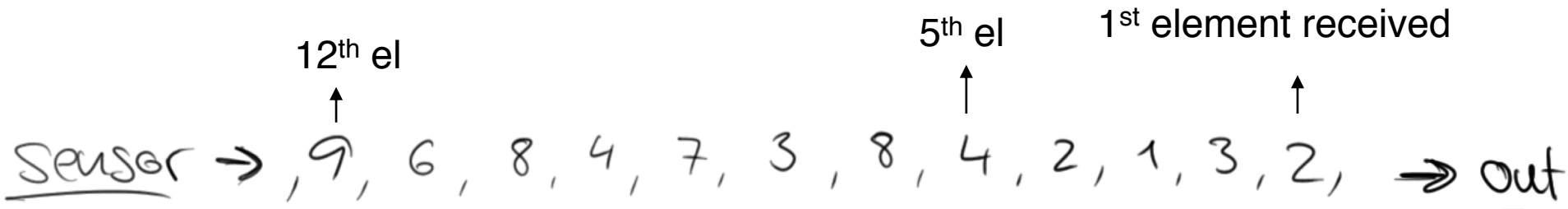


If we think as in normal (non streaming) datasets:

We sum the cars as they are coming

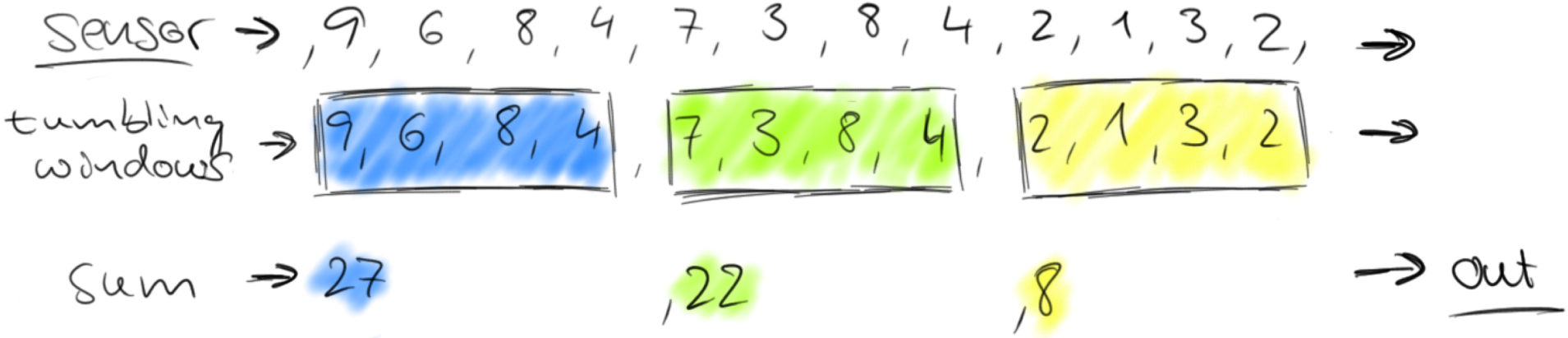


Windows



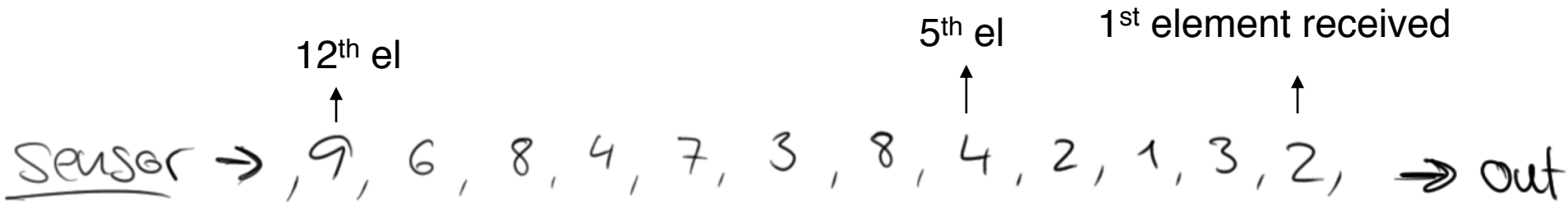
Tumbling windows

- e.g., count cars passed every 1 minute
- Non-overlapping windows



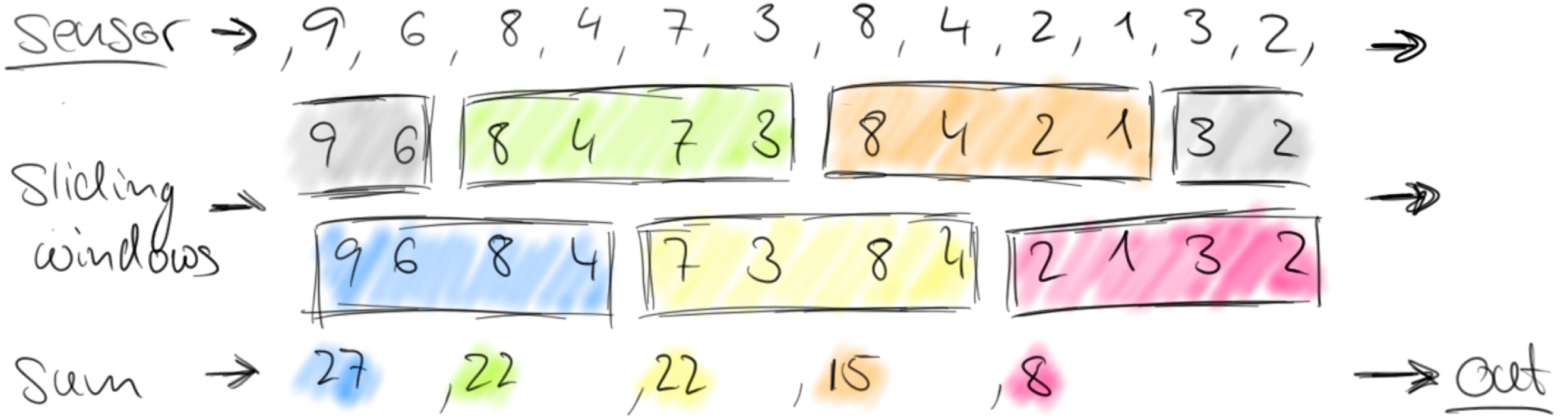
dbTrento

Windows



Sliding windows:

- e.g., count every 30 seconds the car passed in 1 minute
- Overlapping windows



dbTrento

Windows

- a window defines a finite set of elements on an unbounded stream

Can be based on

- time (as in the previous example)
- element counts
- a combination of counts and time
- or some custom logic to assign elements to windows

Window operations

Transformation	Meaning
window(*windowLength*, *slideInterval*)	Return a new DStream which is computed based on windowed batches of the source DStream.
countByWindow(*windowLength*, *slideInterval*)	Return a sliding window count of elements in the stream.
reduceByWindow(*func*, *windowLength*, *slideInterval*)	Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using *func*. The function should be associative and commutative so that it can be computed correctly in parallel.
reduceByKeyAndWindow(*func*, *windowLength*, *slideInterval*, [*numTasks*])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function *func* over batches in a sliding window. **Note:** By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property `spark.default.parallelism`) to do the grouping. You can pass an optional `numTasks` argument to set a different number of tasks.
reduceByKeyAndWindow(*func*, *invFunc*, *windowLength*, *slideInterval*, [*numTasks*])	A more efficient version of the above `reduceByKeyAndWindow()` where the reduce value of each window is calculated incrementally using the reduce values of the previous window. This is done by reducing the new data that enters the sliding window, and "inverse reducing" the old data that leaves the window. An example would be that of "adding" and "subtracting" counts of keys as the window slides. However, it is applicable only to "invertible reduce functions", that is, those reduce functions which have a corresponding "inverse reduce" function (taken as parameter *invFunc*). Like in `reduceByKeyAndWindow`, the number of reduce tasks is configurable through an optional argument. Note that checkpointing must be enabled for using this operation.
countByValueAndWindow(*windowLength*, *slideInterval*, [*numTasks*])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window. Like in `reduceByKeyAndWindow`, the number of reduce tasks is configurable through an optional argument.

Semantic

- **At-most-once delivery:** a message is delivered zero or one times; messages may be lost

  cheapest, highest performance, least implementation overhead, no state needed

- **At-least-once delivery:** potentially multiple attempts are made at delivering it, such that at least one succeeds; messages may be duplicated but not lost

  retries to counter transport losses, keeping state at the sending end and acknowledgement at the receiving end

- **Exactly-once delivery:** exactly one delivery is made to the recipient; the message can neither be lost nor duplicated

  most expensive and consequently worst performance. In addition to the second, it requires state to be kept at the receiving end in order to filter out duplicate deliveries

Most Famous Streaming Platforms

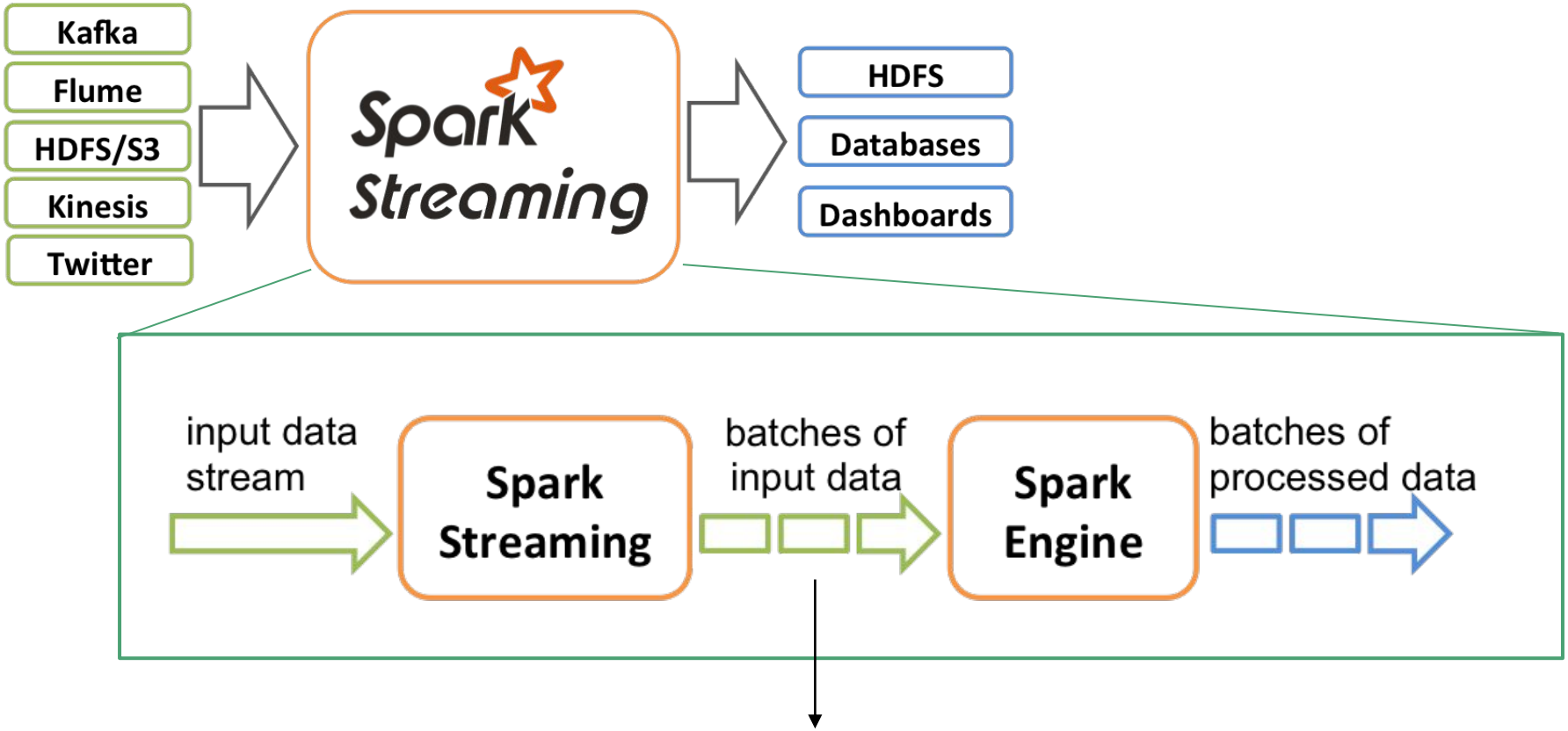
- **Apache Spark:** exactly-one semantic, micro-batch processing

  high throughput, high latency

- **Apache Flink:** exactly-one semantic, stream processing

  high throughput, low latency

Streaming in the Spark Environment

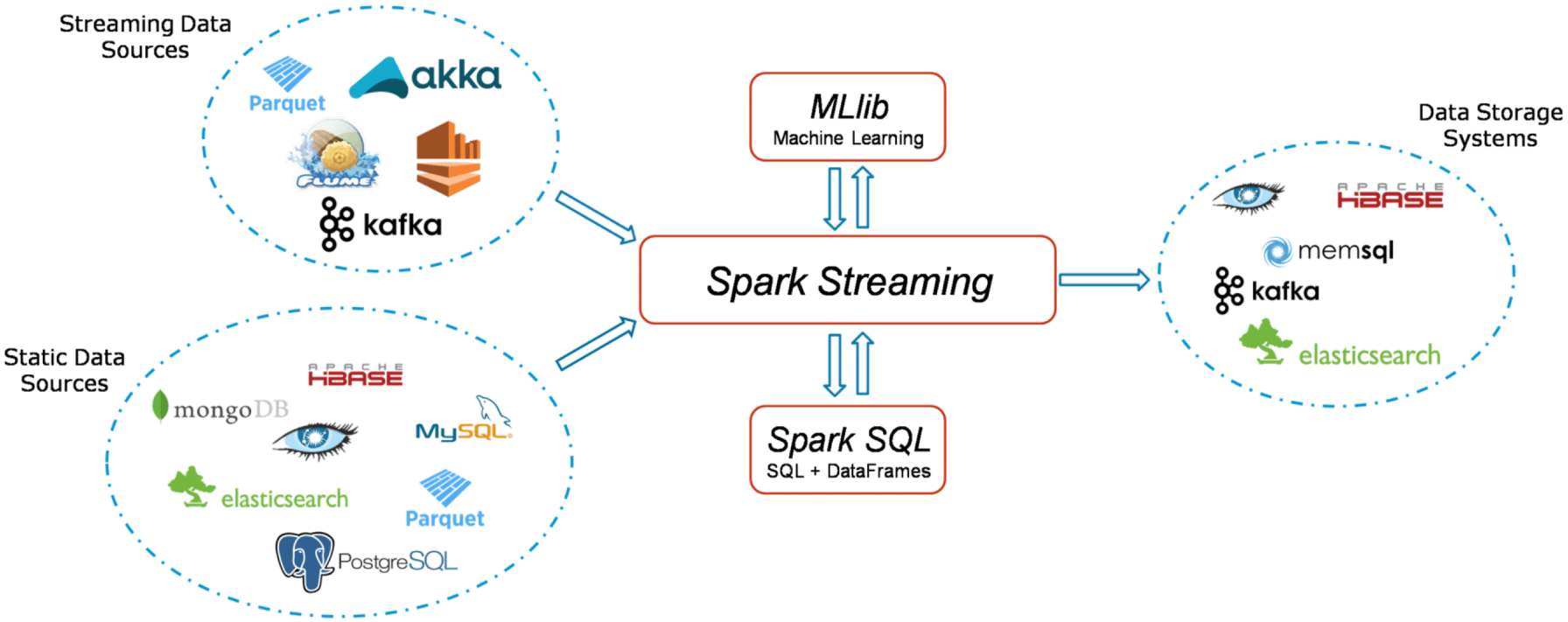


DStream

Discretized Stream = Continuous stream of data

A DStream is represented as a sequence of RDDs

Streaming in the Spark Environment



dbTrento

Streaming Context

- Re-use or create a SparkContext
- Manage the streaming application
  - Start
  - Stop
  - Error management and checkpoints

- One active for each application
- N Streams per Streaming Context

Streaming Context

Figure: *Spark Streaming Context*

Figure: *Default Implementation Sources*

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

# Create a local StreamingContext with two working thread and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
```

Input DStreams and Receivers

Two categories of built-in streaming sources:

***Basic sources*:**

- Sources directly available in the StreamingContext API.
- Examples: file systems, and socket connections.

Advanced sources:

- Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes.
- These require linking against extra dependencies

- Management of multiple Streams
- Multiple receiver
- Configuration is tricky

Points to remember

Do not use “local” or local[1] as master url

- Only one thread for your application
- Receivers (e.g. sockets, Kafka, Flume, etc.) needs one thread itself.
- No thread for processing the data

Running with ”local[n]” as master url

- use **n** greater than the number of receivers

- **Apply the same logic on a cluster**

Sources

Advanced sources:

- **Kafka**
- **Flume**
- **Kinesis**

Custom Sources

- **User define Receiver**
- **Two different type of receivers**

Receiver Reliability

Sources (like Kafka and Flume) allow the transferred data to be acknowledged.

Receive data from a reliable sources: No data has to be lost

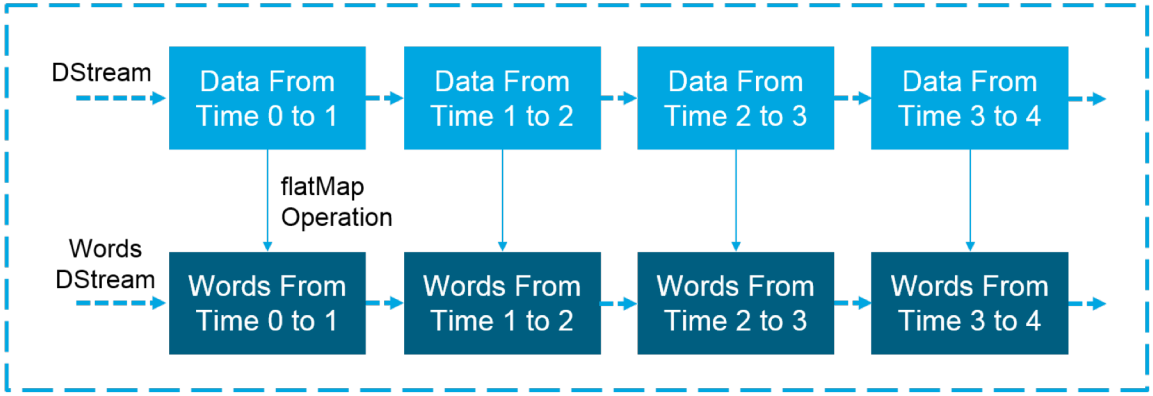
Two kind of receivers:

- *Reliable Receiver: send* acknowledgments to the source
  acknowledgment to the source acknowledgment to the source
  - **Complex and secure: data stored with replication**

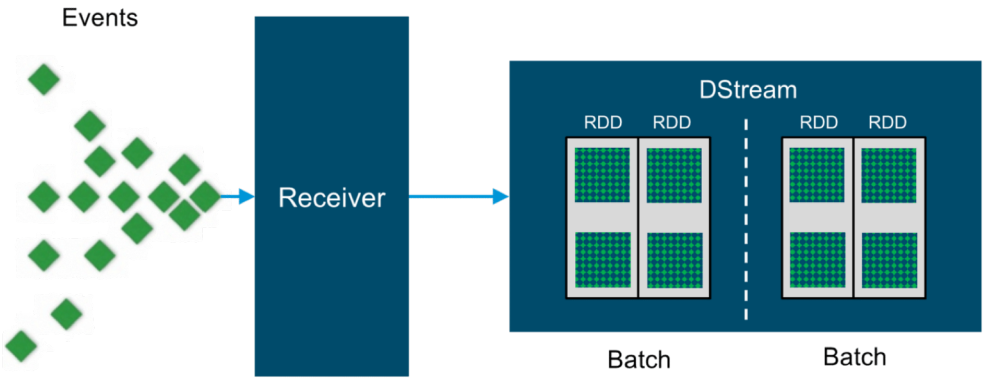
- *Unreliable Receiver: do NOT send* acknowledgments to the source

  - **Simple** but NOT reliable

DStream




```
# Create a DStream that will connect to hostname:port, like localhost:9999
lines = ssc.socketTextStream("localhost", 9999)

# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))
```



dbTrento

Transformations

Transformation	Meaning
map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items.
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.
repartition(*numPartitions*)	Changes the level of parallelism in this DStream by creating more or fewer partitions.
union(*otherStream*)	Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
reduce(*func*)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative and commutative so that it can be computed in parallel.
countByValue()	When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
reduceByKey(*func*, [*numTasks*])	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function. **Note:** By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property `spark.default.parallelism`) to do the grouping. You can pass an optional `numTasks` argument to set a different number of tasks.

Transformations (2)

join(*otherStream*, [*numTasks*])	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.
cogroup(*otherStream*, [*numTasks*])	When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.
transform(*func*)	Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.
updateStateByKey(*func*)	Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key.

UpdateStateByKey:

Maintain an arbitrity state while continuosly updating it with new information

1. Define the state - The state can be an arbitrary data type.
2. Define the state update function: function to updates the state using the previous state and the new values from an input stream

DStream

```
# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)

# Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.pprint()
```

```
ssc.start()             # Start the computation
ssc.awaitTermination()  # Wait for the computation to terminate
```

Into a new terminal, run the following for generating the input data:

```
$ nc -lk 9999
```

DStream

```
# TERMINAL 1:
# Running Netc
at

$ nc -lk 9999

hello world



...
```

```
# TERMINAL 2: RUNNING network_wordcount.py

$ ./bin/spark-submit examples/src/main/python/streaming/network_wordcount.py loca
lhost 9999
...
-------------------------------------------
Time: 2014-10-14 15:25:21
-------------------------------------------
(hello,1)
(world,1)
...
```

Structured Streaming

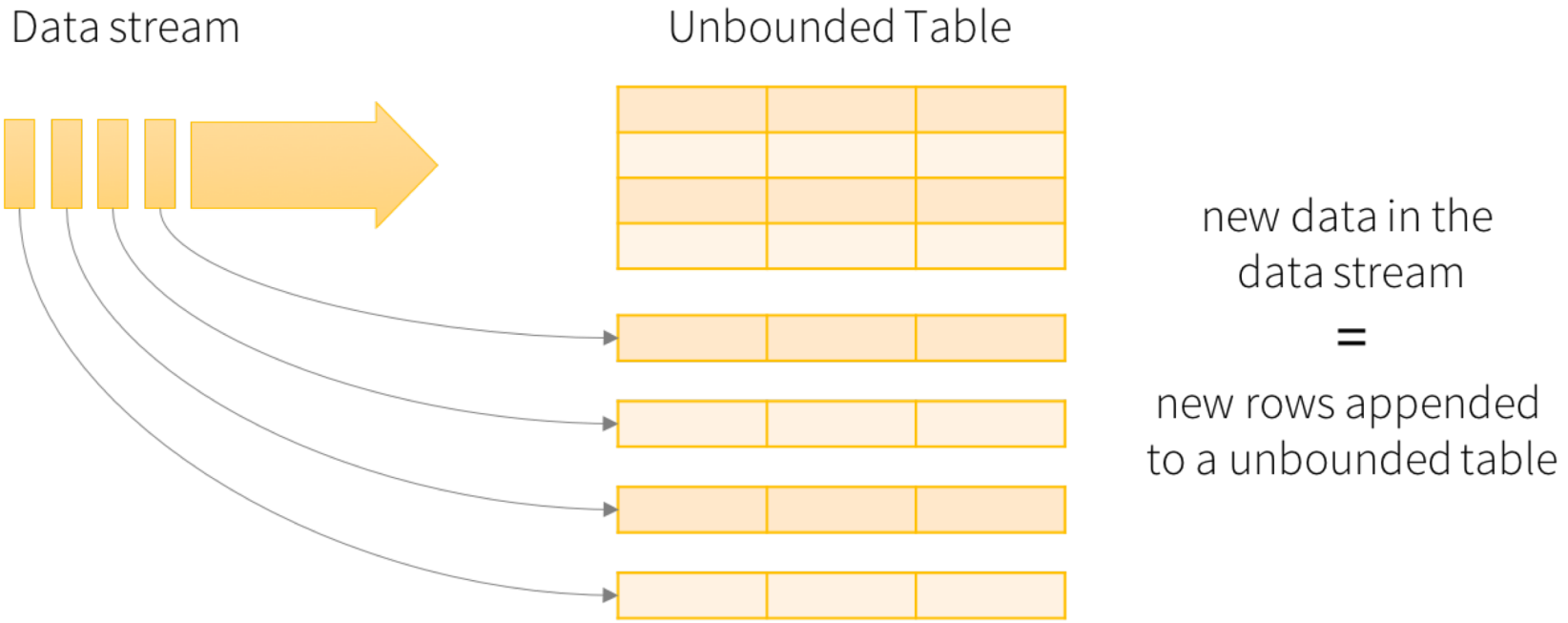
```
# Create DataFrame representing the stream of input lines from connection to localhost:9999
lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

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

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

Exactly the same as DataFrames: it reads the stream and it keeps generating from it the dataframe

Structured Streaming

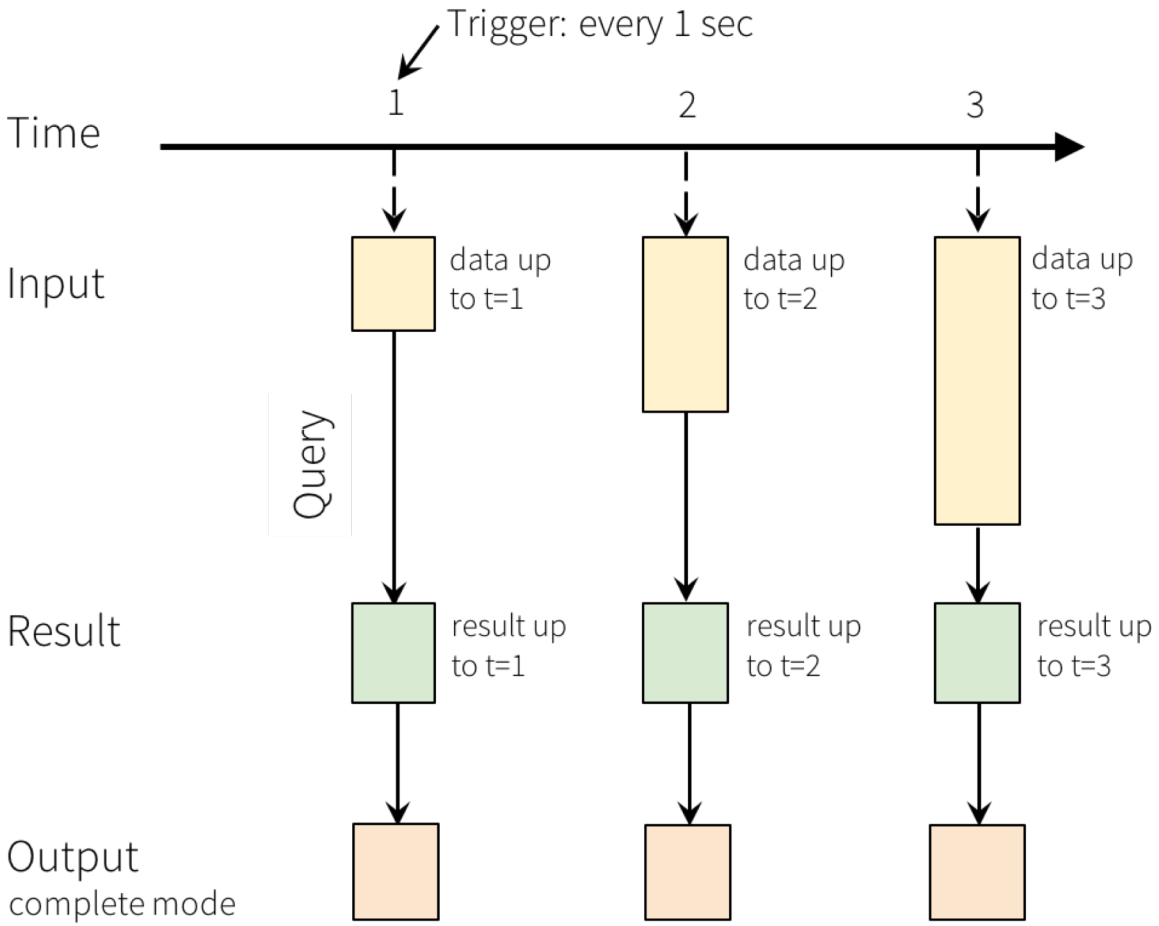
```
 # Start running the query that prints the running counts to the console
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

Structured Streaming



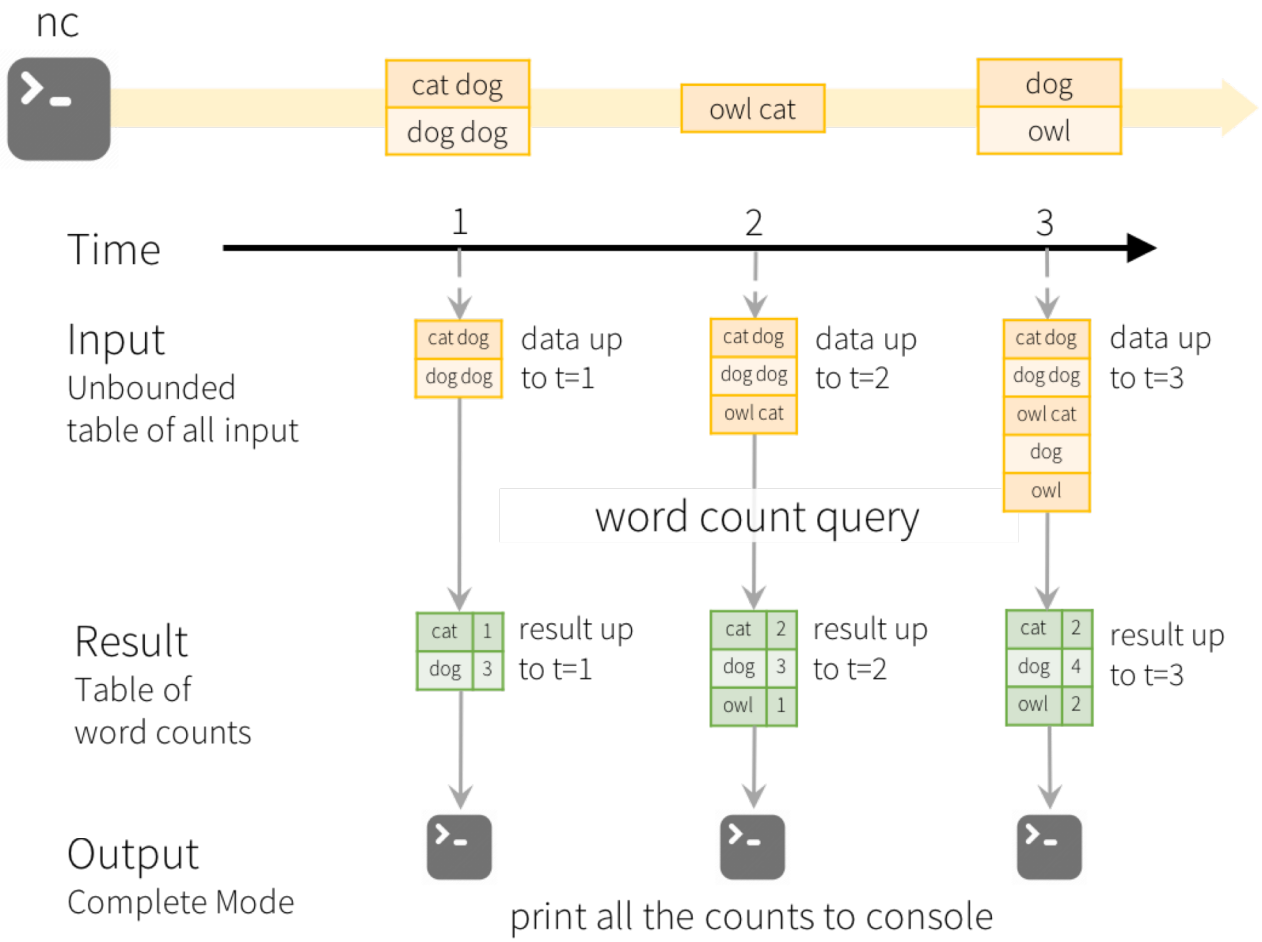
Data stream as an unbounded table

Structured Streaming



Programming Model for Structured Streaming

Structured Streaming



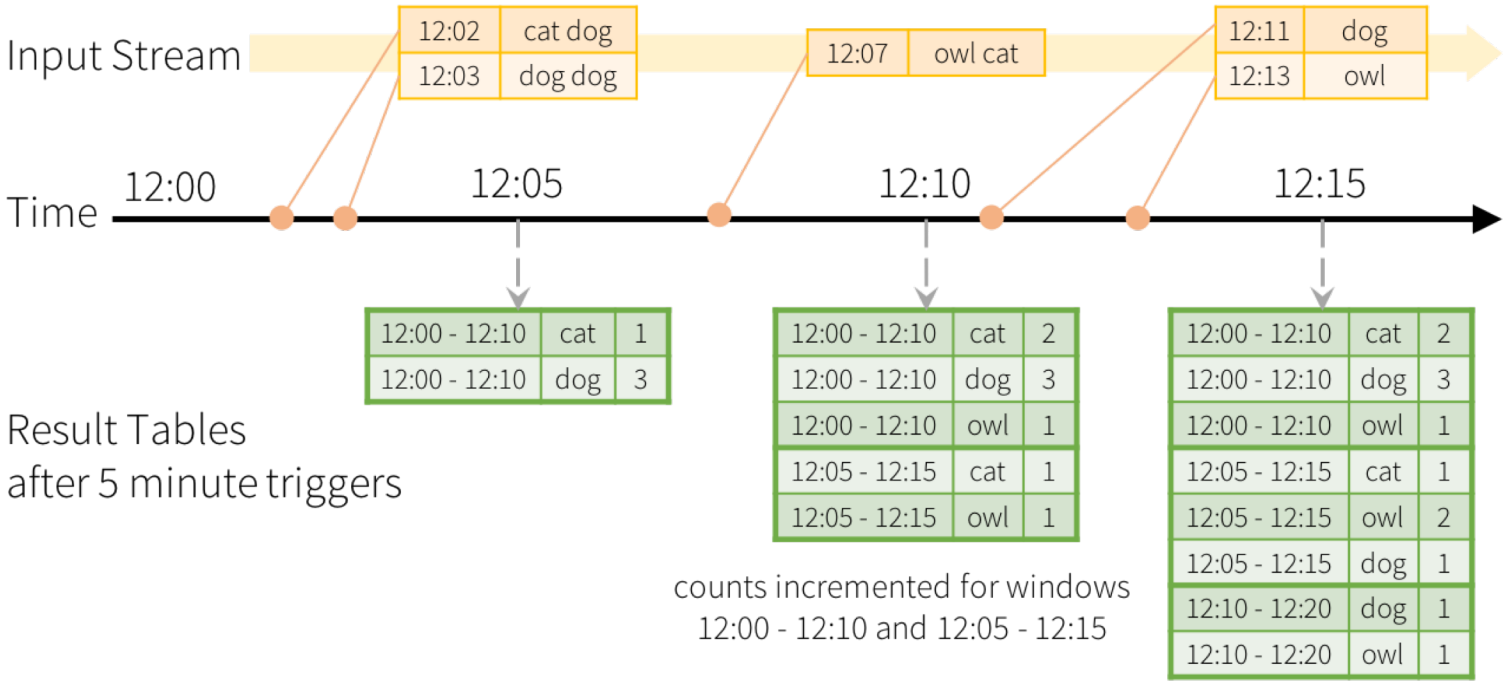
dbTrento

Output mode

The “Output” is defined as what gets written out to the external storage.

- ***Complete Mode***
  - The entire updated Result Table will be written to the external storage.
  - It is up to the storage connector to decide how to handle writing of the entire table.
- ***Append Mode***
  - Only the new rows appended in the Result Table since the last trigger will be written to the external storage.
  - This is applicable only on the queries where existing rows in the Result Table are not expected to change.
- ***Update Mode***
  - Only the rows that were updated in the Result Table since the last trigger will be written to the external storage (available since Spark 2.1.1).
  - Note that this is different from the Complete Mode in that this mode only outputs the rows that have changed since the last trigger.
  - If the query doesn’t contain aggregations, it will be equivalent to Append mode.

Structured Streaming



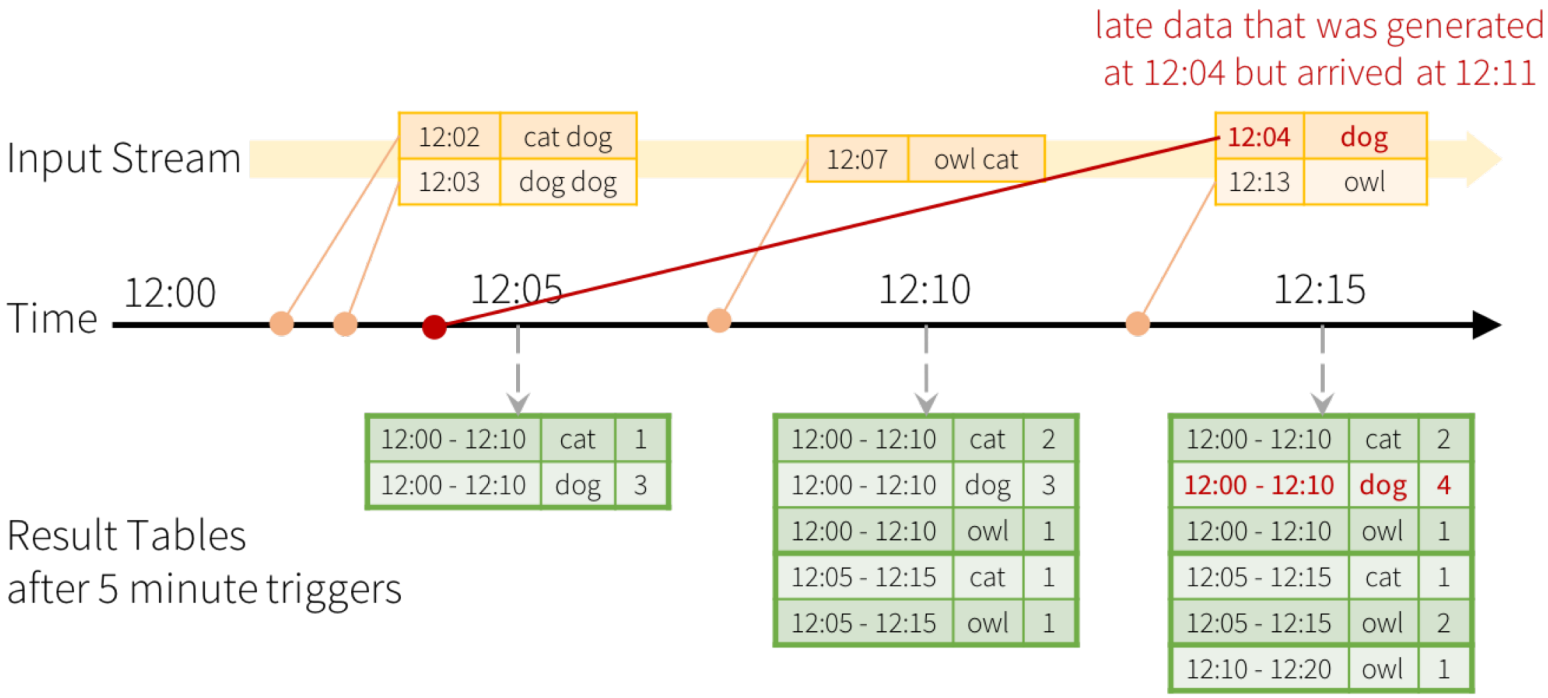
Windowed Grouped Aggregation
with 10 min windows, sliding every 5 mins

Code

```
Dataset<Row> words = ... // streaming DataFrame of schema { timestamp: Timestamp, word: String }

// Group the data by window and word and compute the count of each group
Dataset<Row> windowedCounts = words.groupBy(
  functions.window(words.col("timestamp"), "10 minutes", "5 minutes"),
  words.col("word")
).count();
```

```
Dataset<Row> words = ... // streaming DataFrame of schema { timestamp: Timestamp, word: String }

// Group the data by window and word and compute the count of each group
Dataset<Row> windowedCounts = words
    .withWatermark("timestamp", "10 minutes")
    .groupBy(
        functions.window(words.col("timestamp"), "10 minutes", "5 minutes"),
        words.col("word"))
    .count();
```

Late Events and Watermark

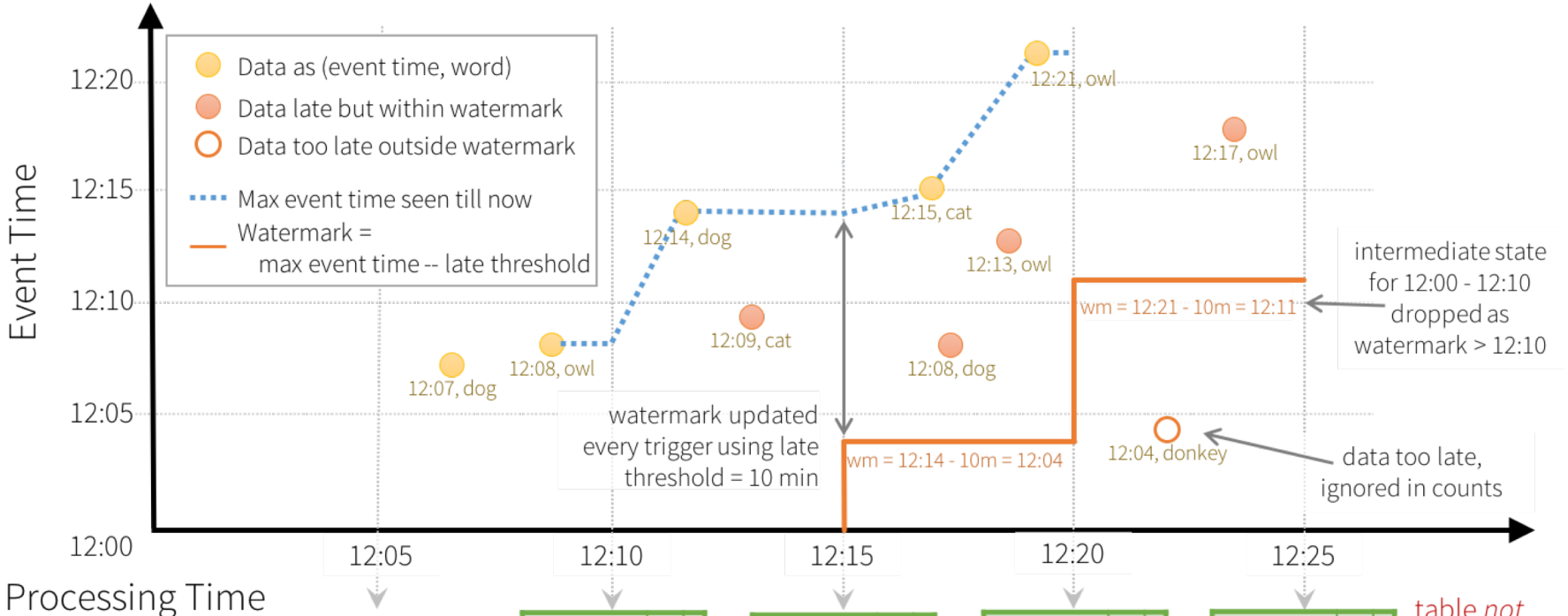


Late data handling in
Windowed Grouped Aggregation

Watermark of 10 minutes

Late Events and Watermark - Update Mode

Watermark of 10 minutes



Processing Time with 5 min triggers

Result Tables after each trigger

12:00 - 12:10	owl	1
12:00 - 12:10	dog	1
12:05 - 12:15	owl	1
12:05 - 12:15	dog	1

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

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

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

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

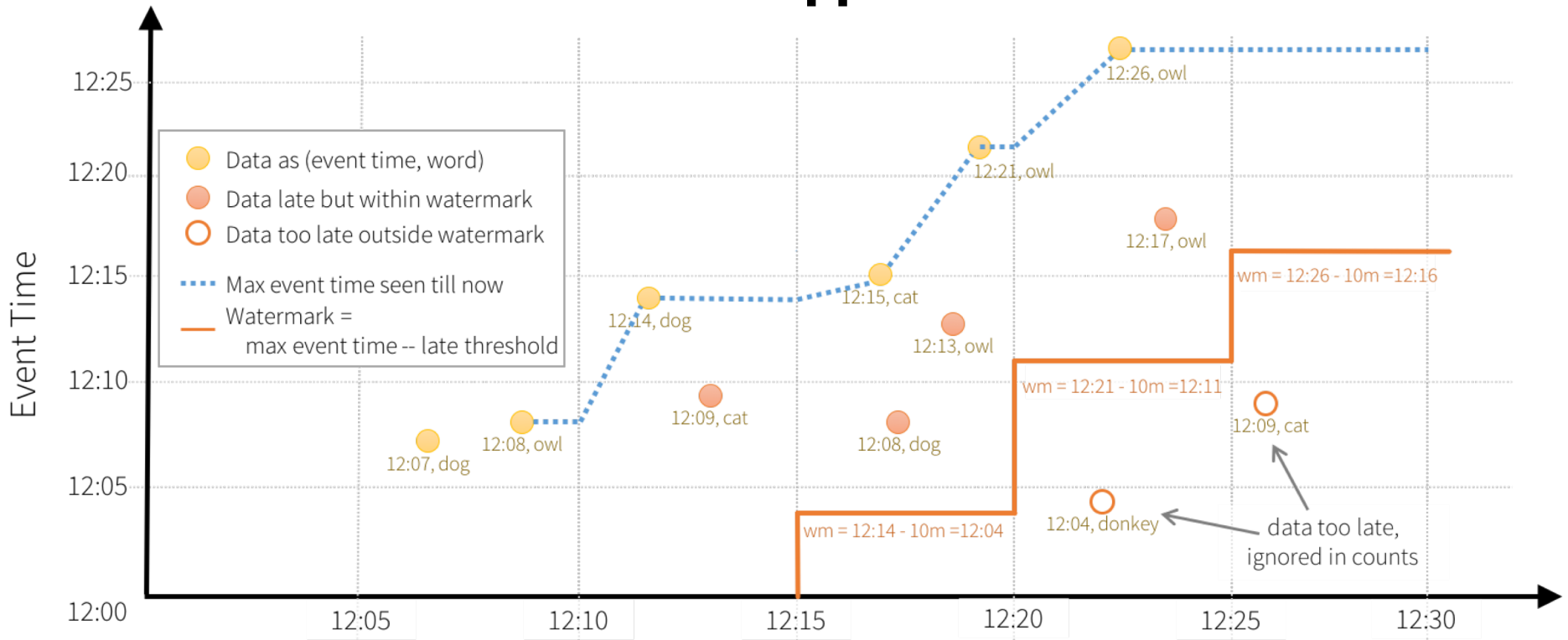
table *not* updated with too late data (12:04, donkey)

table updated with late data (12:17, owl)

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

Late Events and Watermark - Append Mode

Watermark of 10 minutes



partial counts for window 12:00 - 12:10 maintained as internal state while waiting for late data, so not yet added to result table

12:00 - 12:10	owl	1
12:00 - 12:10	cat	1
12:00 - 12:10	dog	2

final counts for 12:00 - 12:10 added to table when watermark > 12:10, late data counted, and intermediate state for window dropped

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

Result Tables after each trigger

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

dbTrento

Basic Steps

1. Create *streamingContext = StreamingContext(sc, duration)*
2. Define the input sources by creating input **DStreams**.
3. Define the streaming computations by applying transformation and output operations to **DStreams**.
4. Start receiving data and processing it using *streamingContext.start()*.
5. Wait for the processing to be stopped (manually or due to any error) using *streamingContext.awaitTermination().*
6. The processing can be manually stopped using *streamingContext.stop()*

Points to remember

- Once a context has been started, no new streaming computations can be set up or added to it.
- Once a context has been stopped, it cannot be restarted.
- Only one StreamingContext can be active at the same time.
- stop() on StreamingContext also stops the SparkContext. To stop only the StreamingContext, set the optional parameter of stop() called stopSparkContext to false.
- A SparkContext can be re-used to create multiple StreamingContexts, as long as the previous StreamingContext is stopped (without stopping the SparkContext) before the next StreamingContext is created.

Join operations

Join StreamRDD

```
stream1 = ...
stream2 = ...
joinedStream = stream1.join(stream2)
```

Join StreamRDD by Window

```
windowedStream1 = stream1.window(20)
windowedStream2 = stream2.window(60)
joinedStream = windowedStream1.join(windowedStream2)
```

Join StreamRDD and RDD

```
dataset = ... # some RDD
windowedStream = stream.window(20)
joinedStream = windowedStream.transform(lambda rdd: rdd.join(dataset))
```

The same applies to Structured Streaming!

Checkpointing

- Streaming application must operate 24/7
- Must be resilient to failures unrelated to the application logic
  - (e.g., system failures, JVM crashes, etc.)

Solution: **Checkpointing**

Checkpoint enough information to recover from a failure

Two different possibilities

Metadata checkpointing

- *Configuration* - The configuration that was used to create the streaming application.
- *DStream operations* - The set of DStream operations that define the streaming application.
- *Incomplete batches* - Batches whose jobs are queued but have not completed yet

Complete checkpointing

- Saving of the generated RDDs to reliable storage (e.g. HDFS)

Contacts

For any problem, send a mail to

daniele.foroni@unitn.it