Big Data
Lecture 4 – Apache Spark's Structured APIs and Structured Streaming

Gianluca Quercini

gianluca.quercini@centralesupelec.fr

Centrale DigitalLab, 2021

Spark RDDs: recap

Resilient Distributed Dataset (RDD)

An RDD consists of three components:

- **Dependencies.** Tell Spark how to construct an RDD from its input (DAG).
- **Partitions.** Used to parallelize the computation across multiple nodes.
- **Compute function.** For each partition, it creates an iterator for the data stored in the partition.

☞ Partitions often contain *locality information* (e.g., for data stored in HDFS) to bring computation to the data.

Spark RDDs: drawbacks

1. **Low-level**. Developers tell Spark *how* to do a computation.
   - Risk of writing inefficient code.
2. The **computation** is **opaque** to Spark.
   - Spark only sees a "lambda" function that cannot inspect to understand the developer's intentions.
   - No way to optimize the computation.
3. The **type** of the RDD elements is **opaque**.
   - Each element is seen as a generic Python object.
   - Objects are serialized as streams of bytes, with no compression techniques.

RDDs: when to use them?

- Full control on *how* Spark implements a computation.
- Use of third-party code that relies on RDDs.
- Computations on unstructured data.

DataFrames

- Spark 1.3 introduced a new **DataFrames API** that provides operators to work with structured data.
- Like an RDD, a **DataFrame** is an *immutable*, *distributed* collection of data.
- Unlike an RDD, a **DataFrame** has a **schema**, like a relational table.
- Once created, a DataFrame can be manipulated with functions that commonly apply to structured tables (e.g., aggregations, operations on columns..).
  - These functions are normally referred to as **domain-specific-language (DSL)**.

☞ Expressing a computation on structured data through DSL functions is much easier than through RDD transformations.

Datasets

- Spark 1.6 introduced a new **Dataset API** that extends the DataFrame API to provide a **compile-time type-safe** interface.

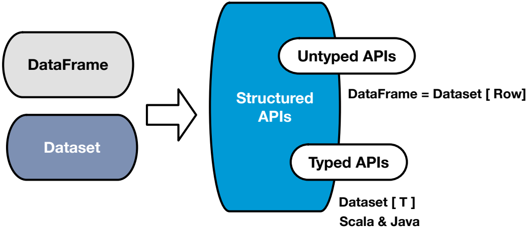
- Like a DataFrame, a Dataset can be viewed as a relational table **but**

- In a **DataFrame**, each row maps to a generic (**untyped**) object *Row*.
- In a **Dataset**, each row maps to a specific (**typed**) JVM object.

- In a **DataFrame**, any type error or access to a non-existing column is **caught at runtime**.
- In a **Dataset**, any type error or access to a non-existing column is **caught at compile-time**.

Spark Structured APIs

- Before Spark 2.x, DataFrames and Dataset APIs were **distinct.**
- Spark 2.x **unified** the two APIs (same functions for both Datasets and DataFrames).



- DataFrames are supported in Python, Scala, Java and R.
- Datasets are only supported in Java and Scala (compile-time type-safe languages).

Using DataFrames

- Possibility of loading data from **different data sources**.
  - Different file formats (csv, json, parquet...) and databases.
- We can use **DSL functions** on a DataFrame.
- Alternatively, we can use **SQL** to manipulate the data.

☞ We'll now see some practical examples.

Spark Structured APIs: benefits

- **Expressivity.** Developers can instruct Spark **what to do** rather than **how to do** it.
- **Composability.** Computations are expressed by composing high-level **DSL** (Domain Specific Language) **operators** (e.g., filtering, aggregating...).
  - Spark can inspect these computations and apply optimizations.
- **Simplicity.** The code is easier to write and read.
- **Uniformity.** The code for any given computation looks similar, whether it is written in Python, Scala or Java.

☞ The computations are implemented by the **SparkSQL engine**.

Spark Structured APIs: benefits

- **Goal:** compute the average temperature for each year.
- **Input:** file where each line is in the format *year,month,temperature*.

Example: average computation with RDDs

```
from pyspark.sql import SparkSession
spark = (SparkSession\
          .builder\
          .appName("Avg temperature")\
          .getOrCreate())

temp_rdd = spark.sparkContext.textFile("./data/temperature.csv")\
      .map(lambda x: x.split(","))\
      .map(lambda x: (x[0], (float(x[2]), 1)))\
      .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))\
      .mapValues(lambda x: x[0]/x[1])
```

☞ Spark **does not understand** what we're doing!

Spark Structured APIs: benefits

- **Goal:** compute the average temperature for each year.
- **Input:** file where each line is in the format *year,month,temperature*.

Example: average computation with DataFrames

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg
spark = (SparkSession\
          .builder\
          .appName("Avg temperature")\
          .getOrCreate())

temp_df = spark.read.csv("./data/temperature.csv",
                          "year STRING, month INT, temp FLOAT")\
              .groupBy("year").agg(avg("temp"))
```

☞ Spark **understands** what we're doing!

Spark Structured APIs: benefits

- **Goal:** compute the average temperature for each year.
- **Input:** file where each line is in the format *year,month,temperature*.

Example: average computation with DataFrames in Scala

```
import org.apache.spark.sql.functions.avg
import org.apache.spark.sql.SparkSession
val spark = (SparkSession
          .builder
          .appName("Avg temperature")
          .getOrCreate())

val schema = "year STRING, month INT, temp FLOAT"
val temp_df = spark.read.format("csv")
    .schema(schema)
    .load("./data/temperature.csv")
    .groupBy("year")
    .agg(avg("temp"))
```

Spark SQL

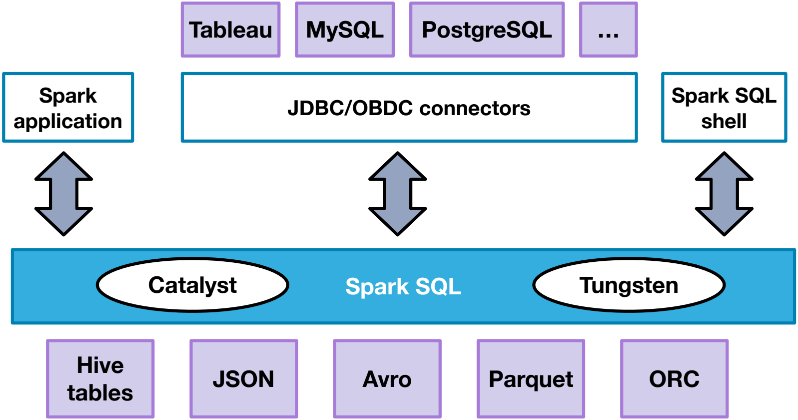
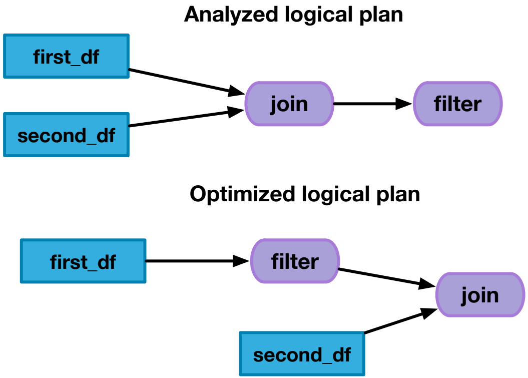


Image source

Catalyst Optimizer

Example

```
joined_df = first_df\
    .join(second_df, first_df.Id == second_df.Id)\
    .where(first_df.Date < "01/11/2020")
```



Stream processing

Definition (Stream processing)

Stream processing is the continuous processing of endless streams of data.

- **Single-node** stream data processing: feasible with **little data**.
- **Distributed** stream data processing: necessary with **big data**.
- Most distributed stream data processing use a **continuous operator model**.

Continuous operator model

- Set of **worker nodes**, each runs one or more **continuous operators**.
- The operator processes the input stream **one record at a time**.
- The output is forwarded to the other operators in the pipeline.

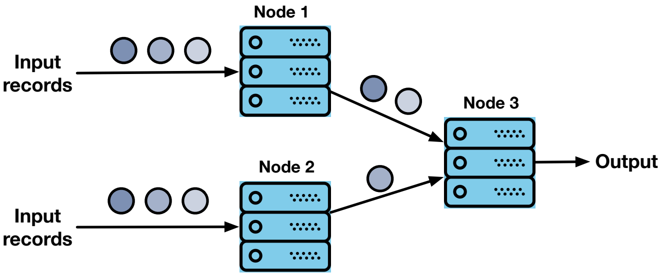


Image source

Continuous operator model

Advantages

- **Simple** and natural model for streaming processing.
- **Low latency**. The output is available within milliseconds.

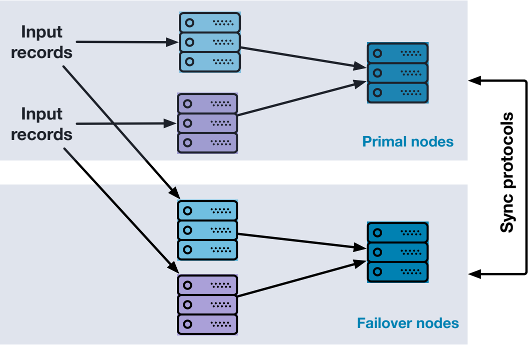
Disadvantages

- Inefficient failure recovery.
- Inefficient handling of stragglers (i.e., slow nodes).

☞ Two techniques to handle failure recovery: **node replication** and **upstream backup**.

Continuous operator model – Node replication

- For each operator, two nodes (**primal** and **failover**) process the same data stream.
- On **failure**, the system switches to the **failover nodes**.
- Each node and the corresponding failover twin must be **synchronized**.

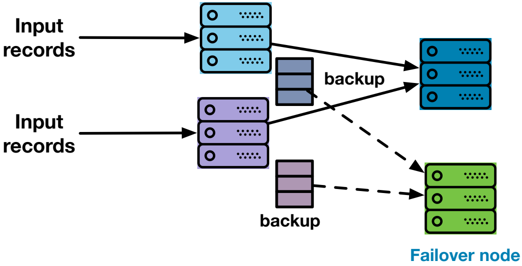


Fast recovery but **high cost** (2x hardware)

Image source

Continuous operator model – Upstream backup

- Backup copy of the forwarded record at each node (with checkpoints).
- On failure, upstream nodes forward (**serially**) the records to the failover node.



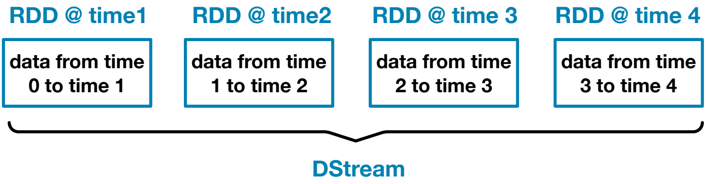
Low hardware cost but **slow recovery**.

Image source

☞ **Stragglers** are not well handled with neither technique.

Spark Streaming – Discretized streams (DStreams)

- Spark streaming introduced **micro-batch stream processing**.
- Streaming computation: continuous series of **small**, **deterministic batch** jobs on small chunks of the input stream.
- A stream of data is modeled as a series of **discretized streams (DStreams)**.
- Internally, each DStream is a RDD (with partitions).

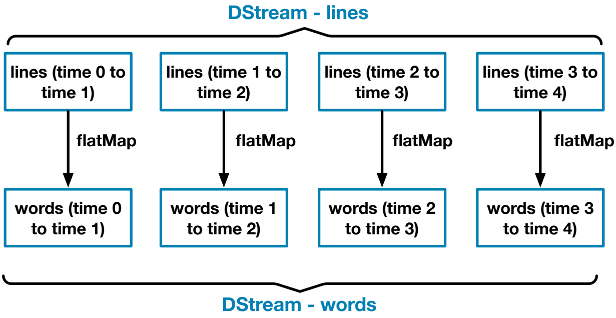


▸ Image source

Spark Streaming – Discretized streams (DStreams)

Example (Spark Streaming API)

```
lines = ssc.socketTextStream("localhost", 9999)
words = lines.flatMap(lambda line: line.split(" "))
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)
```



Spark Streaming – DStream Processing

- Each RDD (data at time x) is processed with **batch operators**.
- **Fault tolerance**: same techniques used on RDDs.
- **Exactly-one processing**: output does not depend on how many times tasks are re-executed.

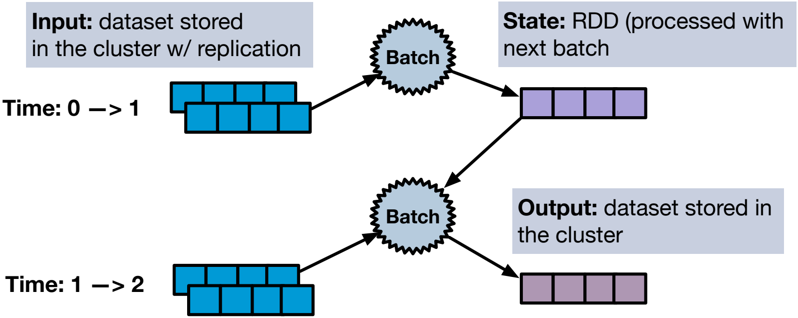
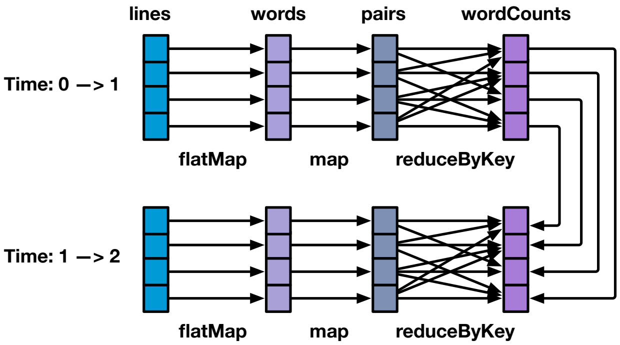


Image source

Spark Streaming – DStream Processing

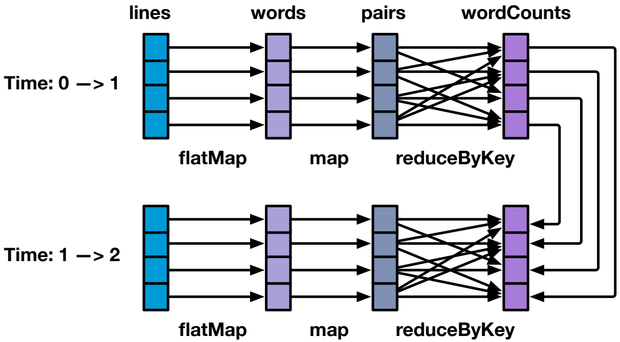
Example (Spark Streaming API)

```
lines = ssc.socketTextStream("localhost", 9999)
words = lines.flatMap(lambda line: line.split(" "))
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)
```



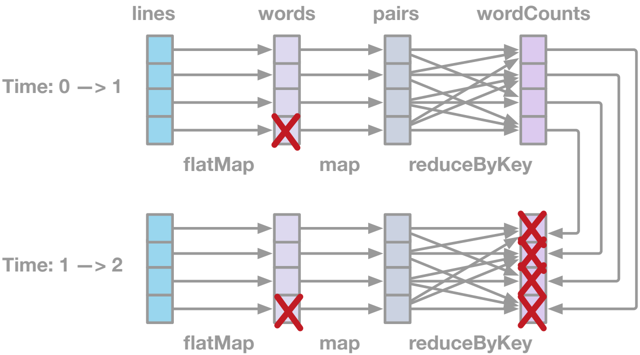
Spark Streaming – Fine-grained lineage

- Graph lineage of operations by RDD partition.
- No need to store the intermediate states.
    - Periodic checkpoints are performed though.
- Lost partitions are recalculated through the lineage.



Spark Streaming – Parallel recovery

- Partitions on **different time steps** can be recomputed **in parallel**.
- Partitions on the **same time step** can also be recomputed **in parallel**.



Spark Streaming – Discussion

Advantages

- Efficient fault tolerance.
- Streaming processing w/ seamless integration with the Spark Core API.

Disadvantages

- Low-level programming, developers need to optimize their code.
- Lack of support for event-time windows.

☞ The micro-batch model cannot achieve millisecond-level latencies.

Spark Structured Streaming

- Since Spark 2.0, integration of the **Structured Streaming API.**
- Built on top of the **Spark SQL engine**.
- Simple and **unified API** for both batch and streaming processing.
- Use of SQL or batch-like DataFrame queries on a stream.
- Suitable for applications that process stream **periodically** or **continuously**.

Structured Streaming programming model

- Each new record in the stream is a new row in an **unbounded table**.
- Structured streaming **does not materialize** the whole table.
- The output at time T is the same as a batch process on all the data up until time T.

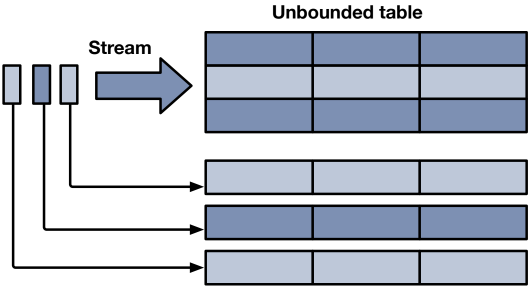


Image source

Structured Streaming processing model

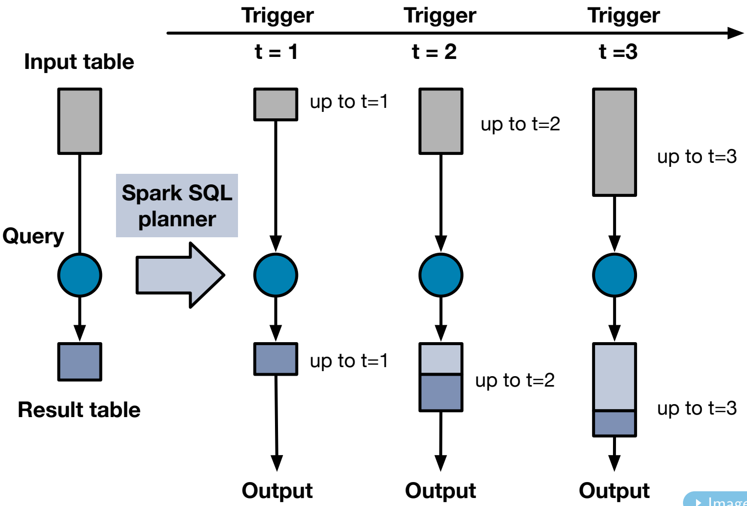


Image source

Structured Streaming processing model

- **Input:** conceptual unbounded table.
- **Query:** produces a **result** table.
- **Incrementalization:** the batch query is converted into a streaming execution plan.
- **Triggering policies**: determine when to read the next chunk of data.
- **Output:** the result at each time step is written to an external sink.
  - Filesystem (e.g., HDFS, Amazon S3), or database (e.g., PostgreSQL, Cassandra).

- Define an input DataFrame from a streaming data source.
- Use the **DataFrame API** to express streaming computations.

Structured Streaming processing model

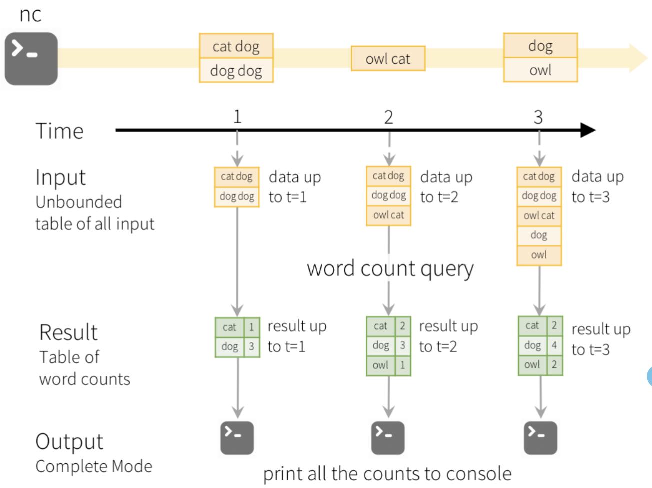


Image source

Define a streaming query

Step 1: define input sources

- Use a **DataStreamReader** to plug into a streaming data source.
- Several data sources: socket connections, JSON, Parquet, ...

Example

```
spark = SparkSession....
lines = spark\
          .readStream.format("socket")\
          .option("host", "localhost")\
          .option("port", 9999)\
          .load()
```

☞ load() **does not** start reading the data. The stream is read when the query is **explicitly started**.

Define a streaming query

Step 2: Transform data

- Data is transformed with usual **DataFrame operations**.
- **Stateless** transformations: do not require any information from previous rows to process the next row.
  - `select()`, `filter()`, `map()`...
- **Stateful** transformations: require information from previous rows to process the new row.
  - count(), groupBy(), ...
- Stateless transformations can be safely used in Structured Streaming.
- Some combinations of stateful transformations are not supported.

Define a streaming query

Step 3: Define output sink and mode

- Use a **DataStreamWriter** to specify where and how to write the output.
- **Where**: different formats, including "json", "console", "kafka"...
- **How**: different output modes.
  - **Append.** Only the new rows are added to the result at each time step.
  - **Complete.** All rows are added to the result at each time step.
  - **Update.** Only the rows that have been updated since the last time step are added to the result.

Example

```
writer = counts.writeStream.format("console").outputMode("complete")
```

Define a streaming query

Step 4: Specify processing details

- Use a **DataStreamWriter** to specify triggering options and checkpoint location.
- **Triggering options**: determine when reading newly available data.
  - **Default**. Query processes data in micro-batches.
  - **ProcessingTime.** Micro-batches are triggered at fixed intervals.
  - **Once.** Executes only one micro-batch.
    - Useful when triggers are controlled by an external scheduler.
    - Saves costs in case of irregular streaming jobs. ▸ Click here
  - **Continuous**. Process data continuously instead of micro-batches (experimental as of Spark 3.0).
- **Checkpoint location.** Directory in HDFS where a streaming query saves its progress.
  - Progress = what data has been processed.
  - Checkpoints used on failure to restart the query where it left off.

Define a streaming query

Step 5: Start the query

- Use a **DataStreamWriter** to start the query.
- Upon start, the query reads the input stream.
- The function `start()` is **non-blocking**.
- In order to block the main thread, use `awaitTermination()`.

Reading from files

- Files written into a directory can be treated as a data stream.
- All the files must have the same format (e.g., `json`).
- All files must have the same **schema**.
  - The schema can be specified when creating the data stream.
- The whole file must be **available at once** for reading.
- Once available, any modification to the file will be **ignored**.
- The files with the **earliest timestamp** are added to the next micro-batch.
- Within the micro-batch, files are read **in parallel** (no predefined order in reading).

Writing to files

- It can write files in the **same formats** as reads.
- Only the **append mode** is supported.
  - Easy to **append** a file to a directory.
  - Difficult to modify existing files.
- Maintains a log of the data files that have been written.
- The log is used to **avoid** the output of **duplicate data** or **losing data** upon failure.
- **Changing the schema** of the result DataFrame between restarts **is possible**.
- However, the output directory will have files with different schemas.

Other input sources

- **Kafka.** Publish/subscribe system for storing of data streams.
  - Structured Streaming can read and write to Kafka.
- **Socket.** Reads UTF8 text data from a socket connection.
  - The listening server socket is at the **driver**.
  - Does not provide end-to-end fault-tolerance guarantees.
- **Rate source.** Generates data at the specified number of rows per second.
  - Each row contains a timestamp and a value (row counter – from 0).
  - Intended for **testing** and **benchmarking**.
- **Custom sources.** Still experimental.

Stateful transformations

Definition (Stateful transformations)

A **stateful transformation** is one whose output depends on the current micro-batch input and the results (i.e., the **state**) of the computations on previous mini-batches.

- The **state** is maintained **in the memory** of the Spark executors.
- The **state** is saved to the **checkpoint location** (in case of failure).

State management

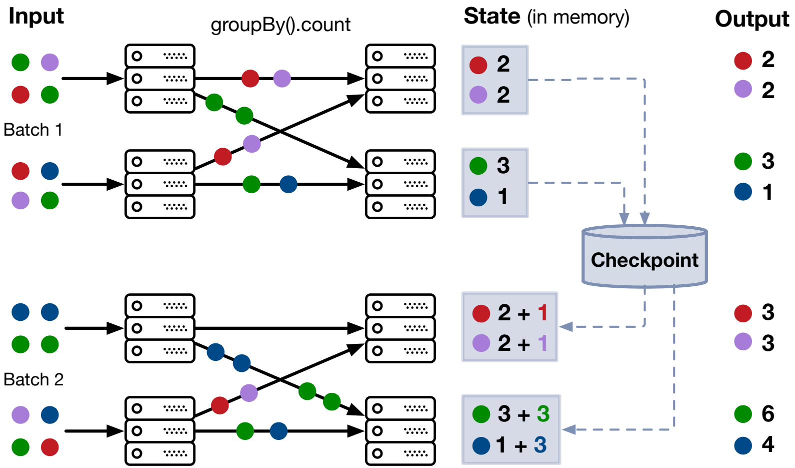


Image source

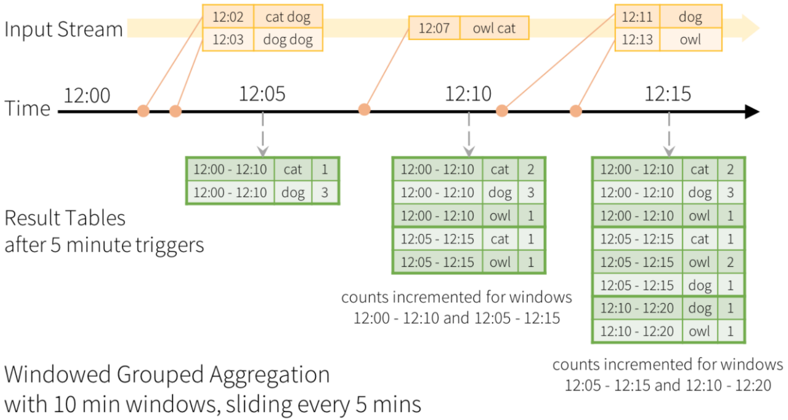
Event-time stateful streaming aggregations

- The aggregation is not run on the whole stream, but on a specified **time window**.
- Use of **sliding windows**.
- Use of the **event time** (when the record has been generated), instead of the **processing time** (when the record is read from the stream).

Example. Event-time word count

- **Input.** Text file, where each line is associated with a timestamp – the **event time**.
- **Goal.** Count words within 10-minute windows, updated every 5 minutes.

Event-time stateful streaming aggregations



▸ Image source

Event-time stateful streaming aggregations

Example. Event-time word count

```
windowedCounts = words.groupBy(
    window(words.timestamp, "10 minutes", "5 minutes"),
    words.word
).count()
```

Source

Handling late data

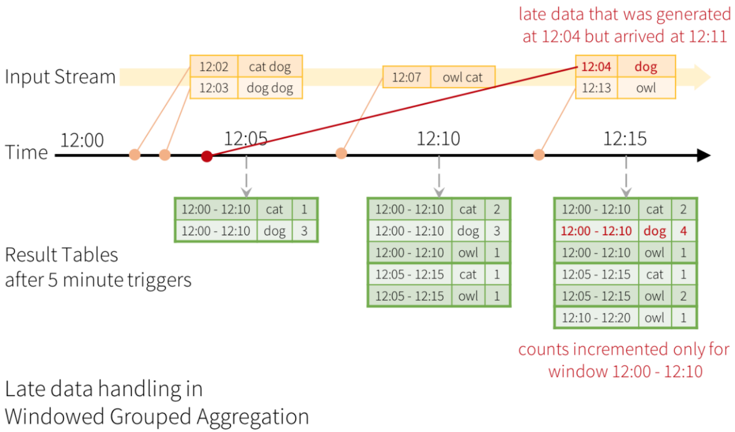


Image source

Watermarking

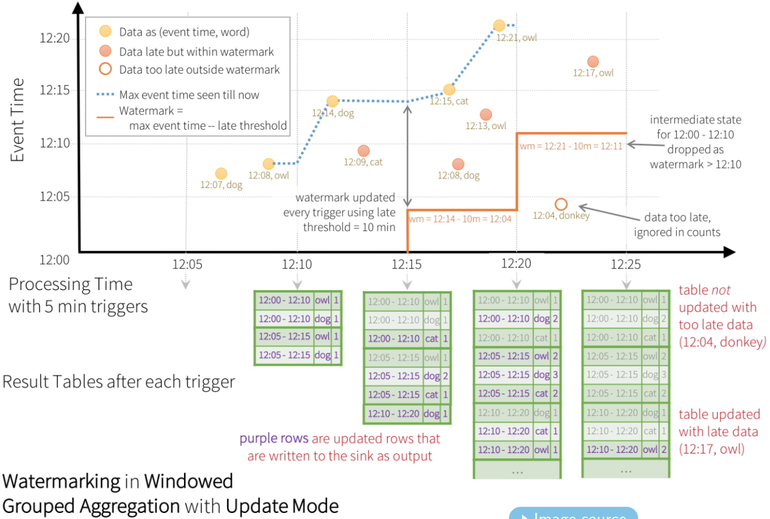


Image source

References

- Jules Damji et al. *Learning Spark: Lightning-Fast Data Analytics*. "O'Reilly Media, Inc.", 2020. Click here
- Zaharia, Matei, et al. *Discretized streams: Fault-tolerant streaming computation at scale.* Proceedings of the twenty-fourth ACM symposium on operating systems principles. 2013 Click here