Introduction to Big Data

STREAM PROCESSING

Le Ngoc Thanh – Nguyen Ngoc Thao
{lnthanh, nnthao}@fit.hcmus.edu.vn

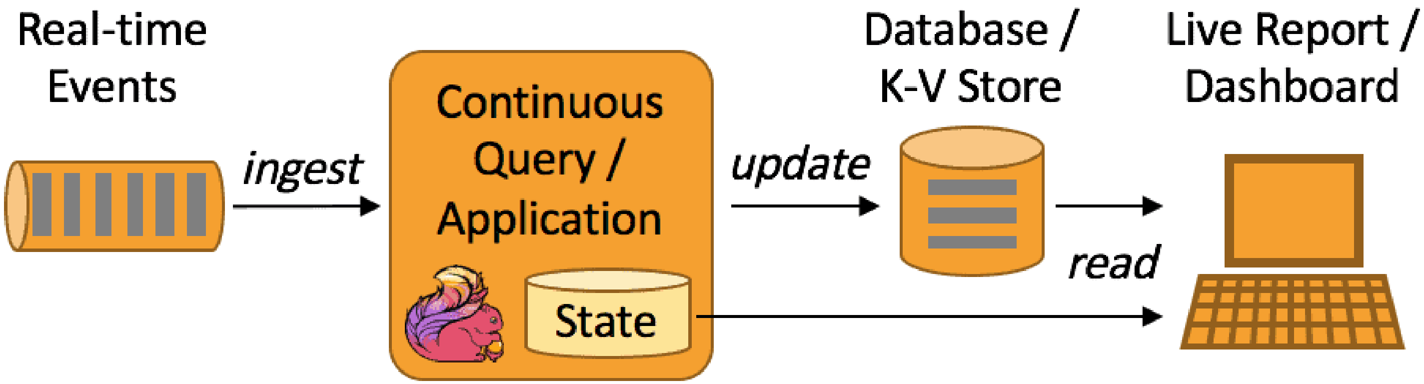
Outline

- Stream processing fundamentals
- Streaming API in Spark

Stream processing

Stream processing

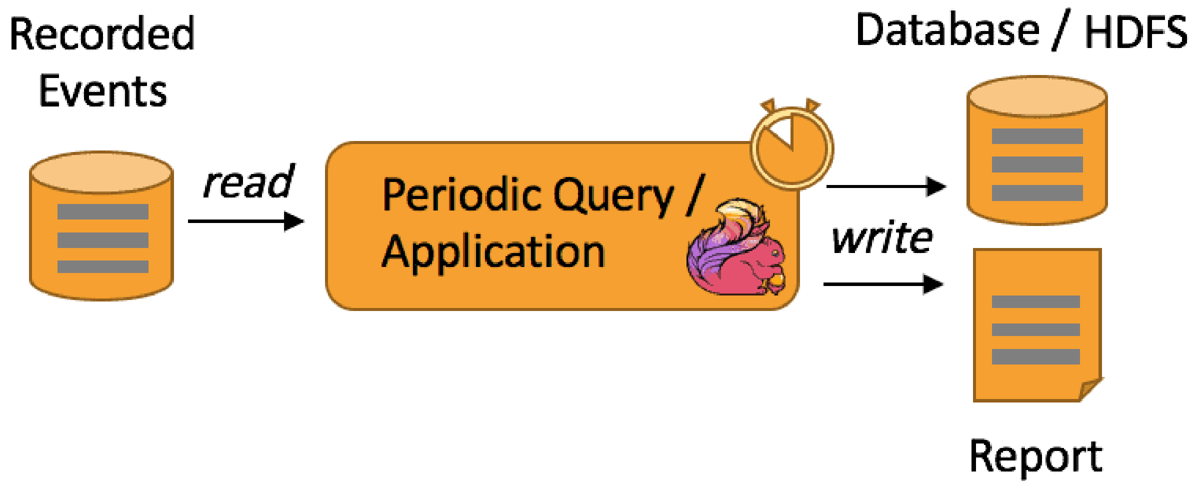
- Stream processing is the act of continuously incorporating new data to compute a result.



- The input data has no predetermined beginning or end.
  - It simply forms a series of events (e.g., credit card transactions, clicks on a website, or sensor readings from IoT devices).
- The output data is kept up-to-date in an external “sink” sytem.

Batch processing

- Batch processing runs the computation on a fixed dataset.
  - The data is often large-scale and stored in a data warehouse
  - It contains all the historical events from an application, e.g., all website visits or sensor readings for the past month



- It also takes a query to compute, like stream processing, but only computes the result once.

Stream and Batch processing

- These processing modes are different, yet they often need to work together in practice.

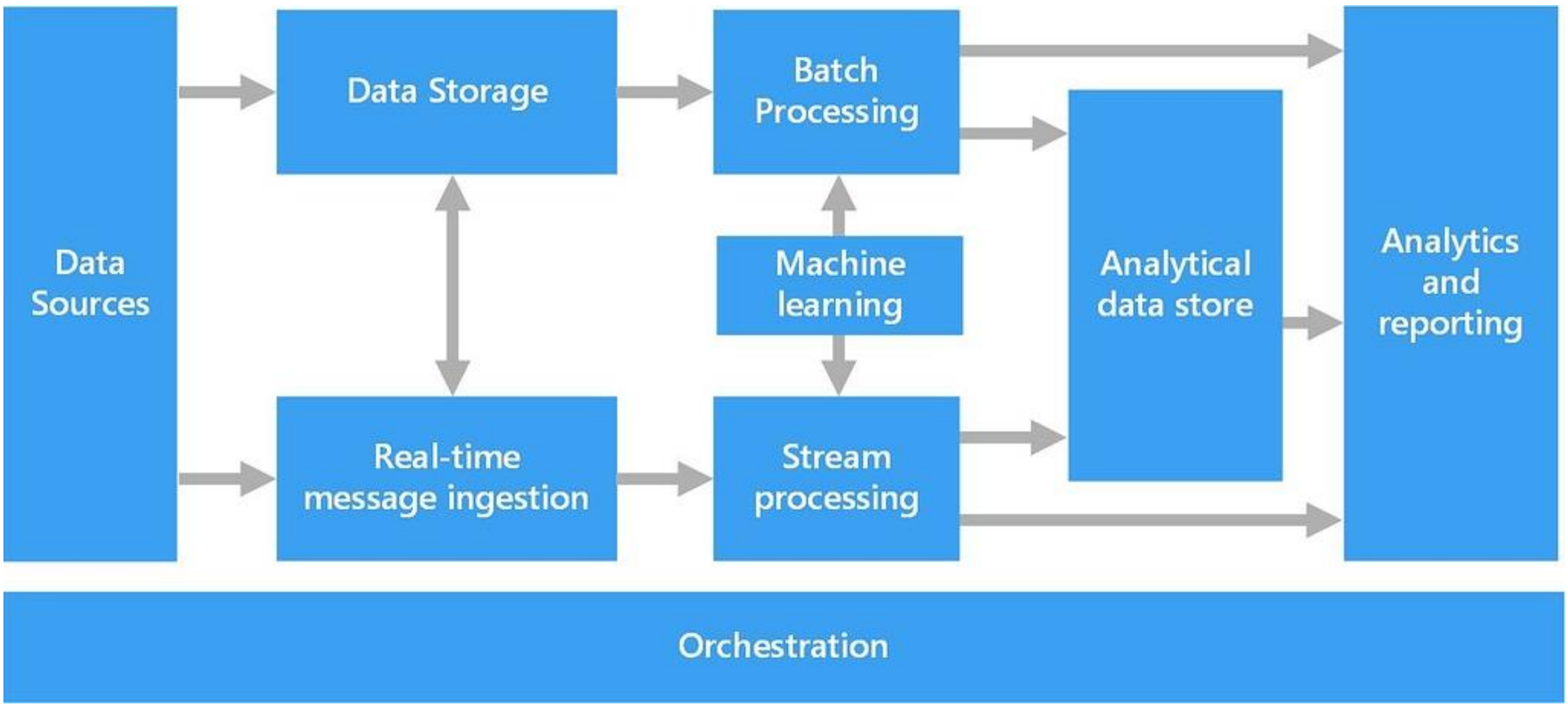


Image credit: [Mage Store]

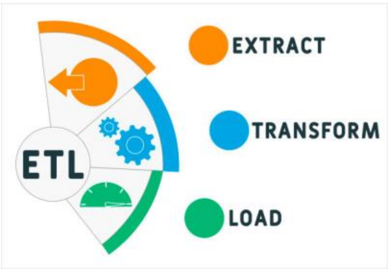
Stream processing: Use cases

Notifications and alerting

- A notification or alert should be triggered if some sort of event or series of events occurs
- E.g., credit card fraud detection, elderly home monitoring

Real-time reporting

- Dashboards are common in several organizations to announce live updates to audiences.
- E.g., live reports for stock market center, city traffic, etc.



Incremental ETL

- We can incorporate new data within seconds, enabling users to query it faster downstream.
- The data must be in a fault-tolerance manner

Batch processing: Advantages

- Batch processing is simpler in the majority of use cases.


understanding


troubleshoot


application building

- It allows for vastly higher data processing throughput than many streaming systems.

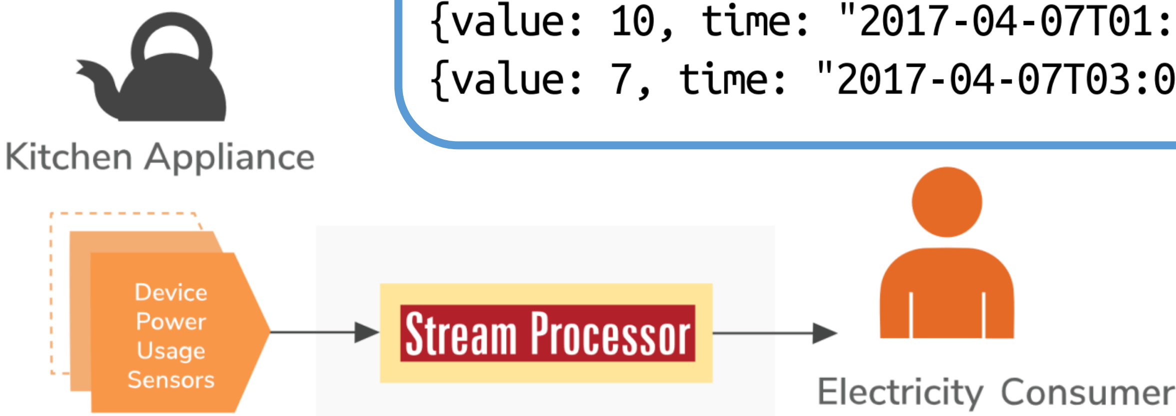
Stream processing: Advantages

- Lower latency: the application needs to respond quickly (on a timescale of minutes, seconds, or milliseconds)
  - A streaming system keeps state in memory to get good performance.
  - Many decision making and alerting use cases fall into this camp.
- More efficient in updating a result than repeated batch jobs: the computation is automatically incrementalized
  - Case study: compute web traffic statistics over the past 24 hours
  - A streaming system recalls the state from the previous computation and only count the new data, instead of scanning all the data.

Stream processing: Challenges

- Consider an application that receives input messages from a sensor, which reports values at different times.

```
{value: 1, time: "2017-04-07T00:00:00"}
{value: 2, time: "2017-04-07T01:00:00"}
{value: 5, time: "2017-04-07T02:00:00"}
{value: 10, time: "2017-04-07T01:30:00"}
{value: 7, time: "2017-04-07T03:00:00"}
```



- We want to search the stream for a certain value or pattern of values.
- Unfortunately, the input records might arrive in an out-of-order fashion due to delays and retransmissions.

Stream processing: Challenges

- **Case study:** trigger some action based on a specific sequence of values received, say, 2 then 10 then 5.
- **Batch processing:** not notably difficult, it can simply sort all the events
- **Stream processing:** more challenging, it needs to track some state across events to realize the actual order of events

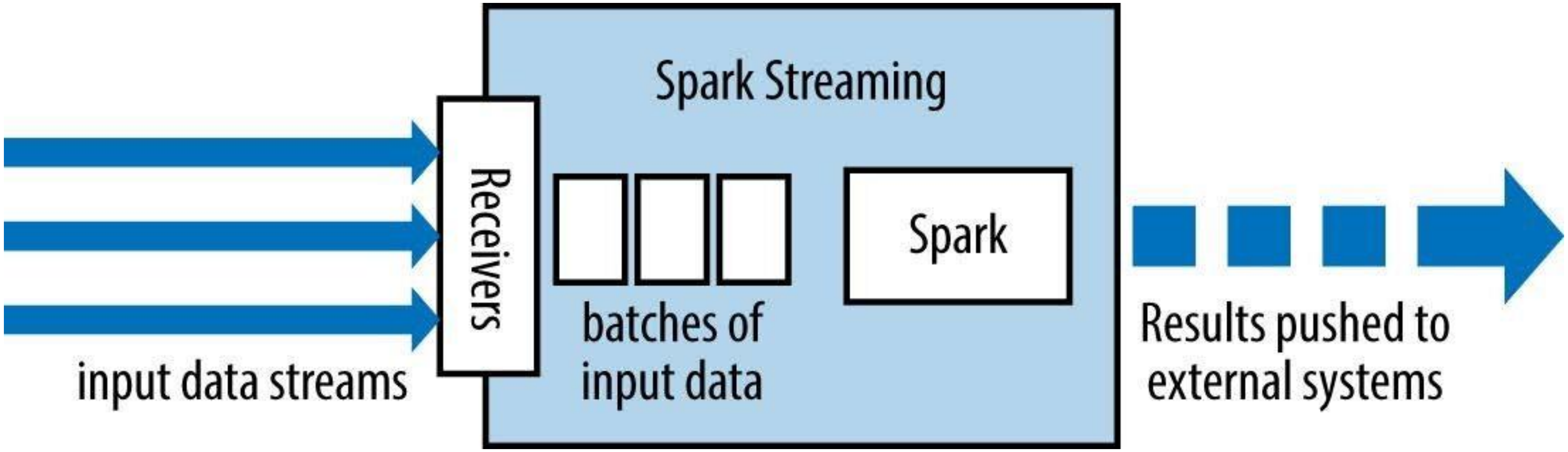
Streaming API in Spark

The history of Spark Streaming

- Spark has a long history of high-level support for streaming.
- Spark Streaming (2012) uses DStreams API, which is based on relatively low-level operations on Java/Python objects.
  - It is one of the first APIs to enable stream processing using high-level functional operators like `map` and `reduce`.
  - However, the opportunities for higher-level optimization is limited.
  - It is purely micro-batch oriented and based on processing time.
- Structured Streaming (2016, stable since v.2.2) is built on DataFrames, attaining rich optimizations and truly simpler integration with other code.
  - It offers a superset of the majority of the functionality of DStreams.
  - Better performance due to code generation and Catalyst optimizer
  - Higher-level optimizations, event time, and support continuous processing

Spark Streaming

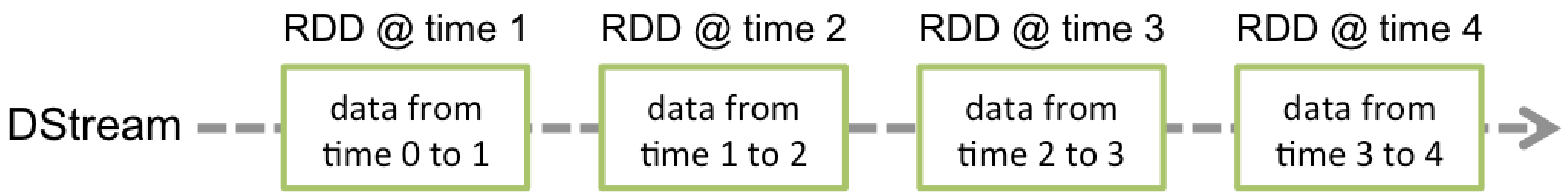
- Spark Streaming has a micro-batch architecture
  - The tranmission is a series of data batches, which are created at regular time intervals.
  - The batch interval is usually between 500ms and several seconds.



Credit

DStream API in Spark Streaming

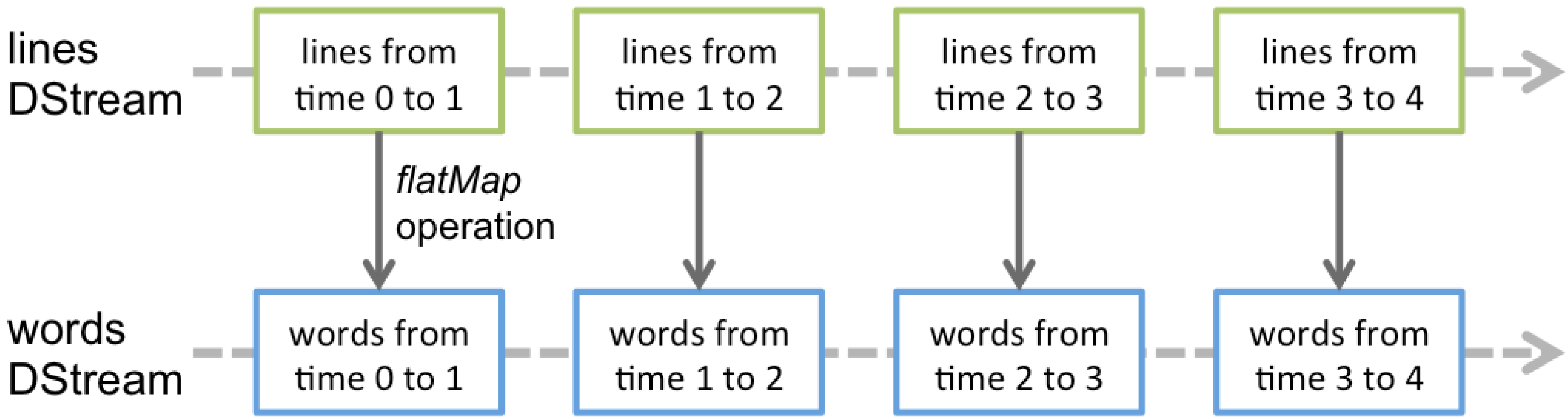
- DStream is the basic abstraction in Spark Streaming, which represents a continuous flow of data.
  - The data flow is either the input stream received from a source or the processed data stream generated by transforming the input stream.



[Credit]

DStream API in Spark Streaming

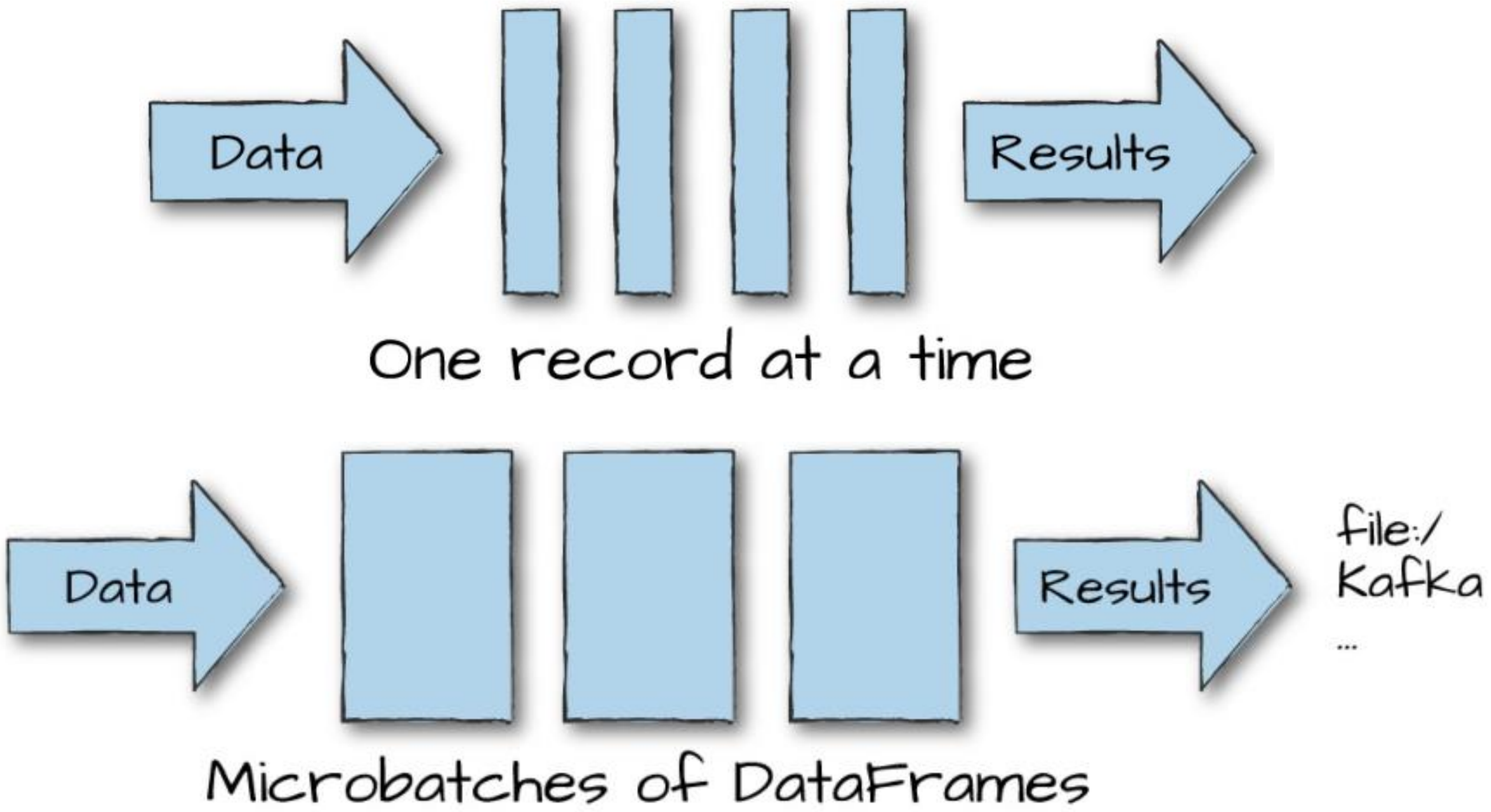
- Any operation taken in a DStream is translated into those in the underlying RDDs.



Credit

Structured Streaming

- Continuous processing vs. Micro-batch execution



Structured Streaming: Continuous mode

- Continuous processing: one record at a time
  - Each node in the system is continually listening to messages from other nodes and outputting new updates to its child nodes.
- For example, an application implements a map-reduce computation over several input streams.
  - Each of the nodes implementing map would read records one by one from an input source, compute its function on them, and send them to the appropriate reducer.
  - The reducer then updates its state whenever it gets a new record.
- Lowest possible latency, lower maximum throughput

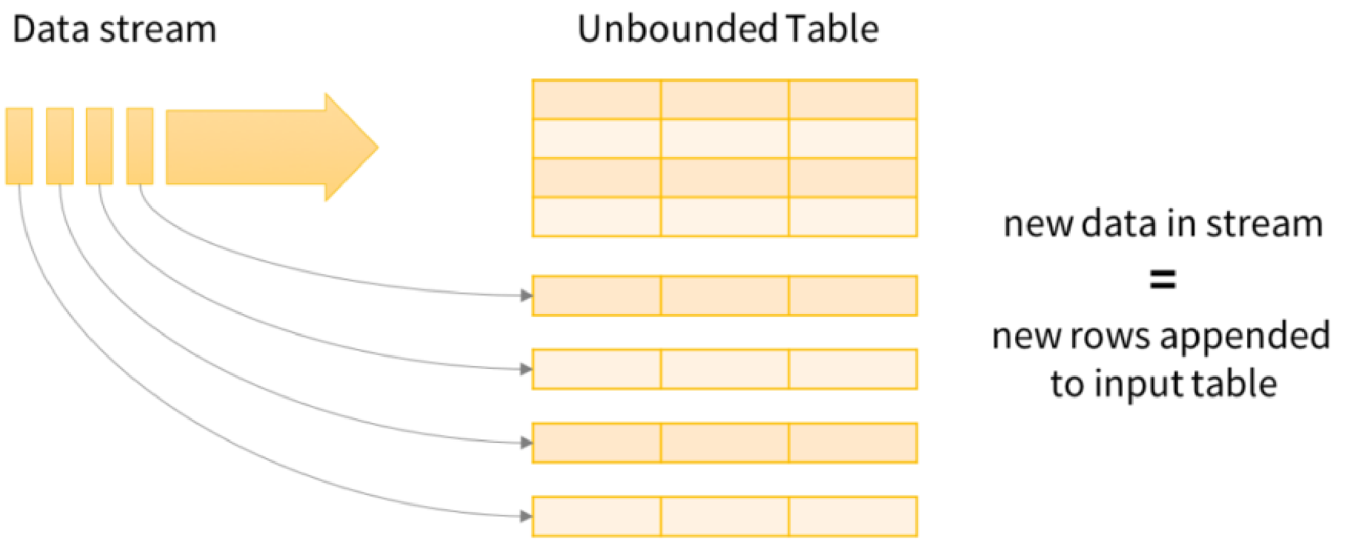
Structured Streaming: Micro-batch

- Micro-batch processing: the system accumulates small data batches (e.g., in 500 ms) then process each batch.
  - Each batch is processed similarly to the execution of a batch job.
- High throughput per node: leverage same optimizations as batch systems (e.g., vectorized processing), and no any extra per-record overhead incurred.
- High latency due to waiting to accumulate a micro-batch
- In practice, a fairly large streaming application needs to distribute its computation to prioritize throughput.

An example of Structure Streaming

Streaming processing: Input data

- All data arriving are treated as an unbounded Input table.
- Every data item is like a new row appended to the table.



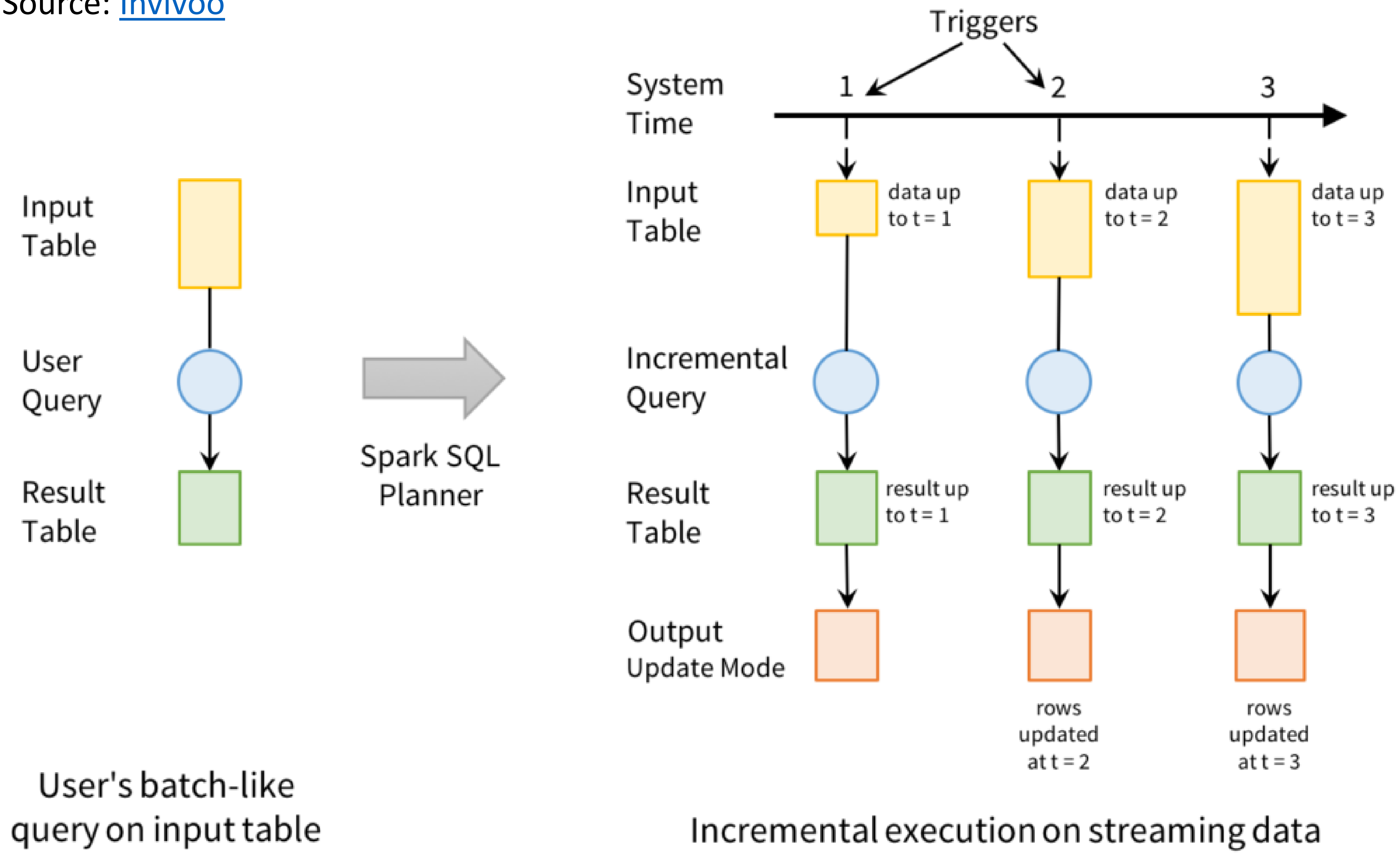
Data stream as an unbounded Input Table

Source: [Invivoo]

Streaming processing: Query

- A query on the input will generate the Result table.
- Each time a trigger fires, Spark checks for new data (new row in the Input table), and incrementally updates the result.

Source: Invivoo



Structured Streaming Processing Model

Users express queries using a batch API; Spark incrementalizes them to run on streams

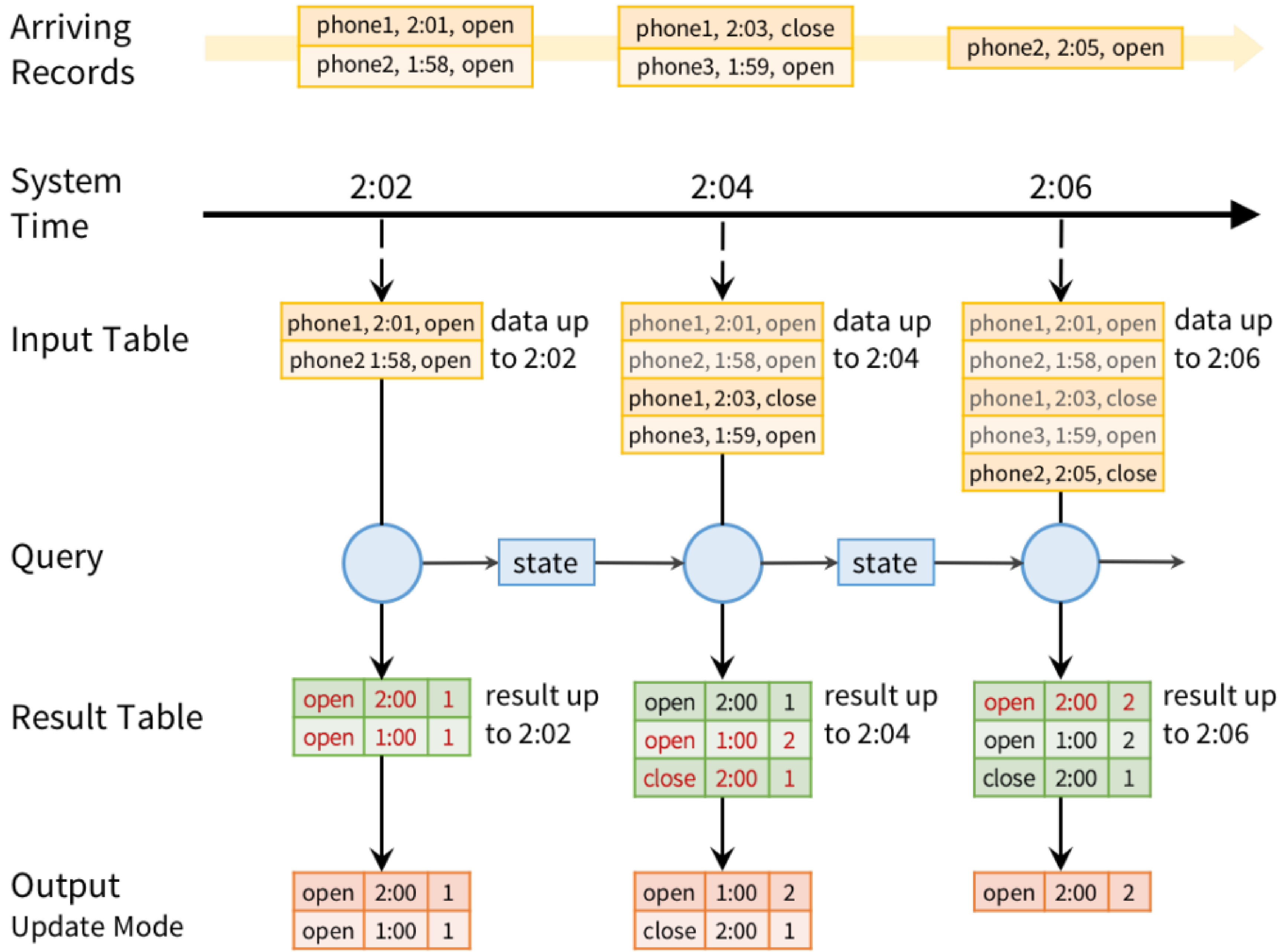
Stream processing: Output data

- We usually want to write the output incrementally for each time the Result table is updated.
- Structured Streaming provides three output modes.

Append | **Complete** | **Update**

Stream processing: Output modes

- Append: Only the new rows appended to the result table since the last trigger will be written to the external storage.
  - This is applicable only on queries where existing rows in the result table cannot change (e.g. a map on an input stream).
- Complete: The entire updated result table will be written to external storage.
- Update: Only the rows that were updated in the result table since the last trigger will be changed in the external storage.
  - This mode works for output sinks that can be updated in place, such as a MySQL table.



Source: Invivoo

Streaming word count on Ncat

```python
# Creating a SparkSession in Python
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local")\
          .appName("Spark Streaming Demonstration")\
          .config("spark.some.config.option", "some-value")\
          .getOrCreate()
# keep the size of shuffles small
spark.conf.set("spark.sql.shuffle.partitions", "2")
```

Streaming word count on Ncat

```
from pyspark.sql.functions import explode
from pyspark.sql.functions import split
# Create DataFrame representing the stream of input lines
# from connection to localhost:50050
lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 50050) \
    .load()
# Split the lines into words
words = lines.select(
   explode(
       split(lines.value, " ")
   ).alias("word")
)
# Generate running word count
wordCounts = words.groupBy("word").count()
```

Streaming word count on Ncat

```
# Initialize ncat first
# ncat server: ncat -l -p 50050
# ncat client: ncat localhost 50050 (not the same with spark)
# Start running the query that prints the running counts to the console
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()
query.awaitTermination(60)
```

```
query.stop()
```


...the end.