What is Streaming?

Data Streaming is a technique for transferring data so that it can be processed as a steady and continuous stream. Streaming technologies are becoming increasingly important with the growth of the Internet.

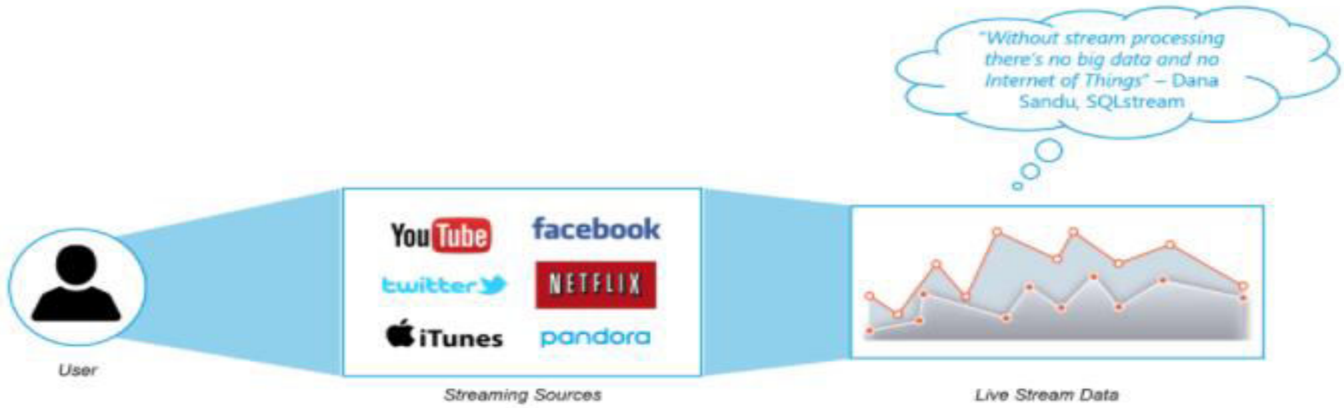
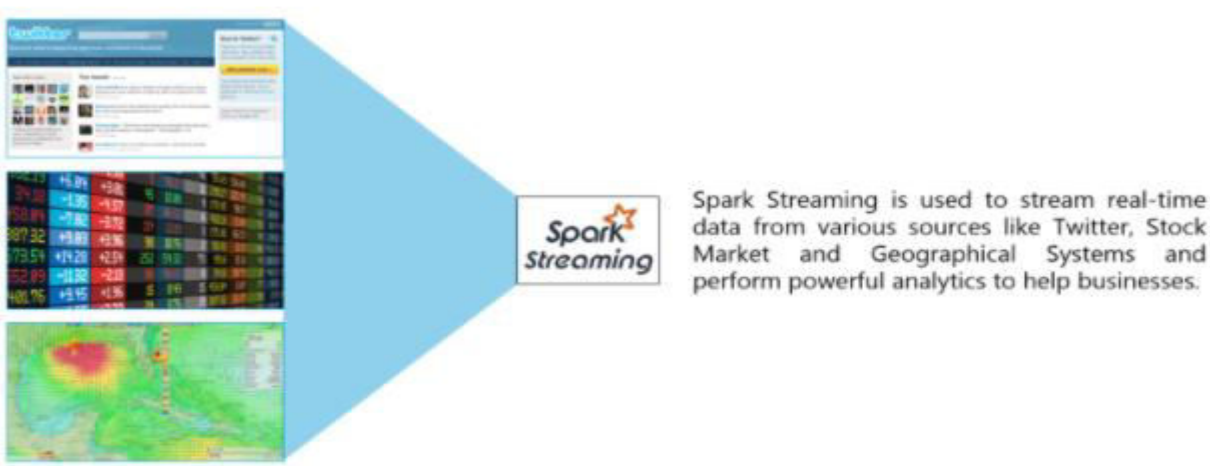


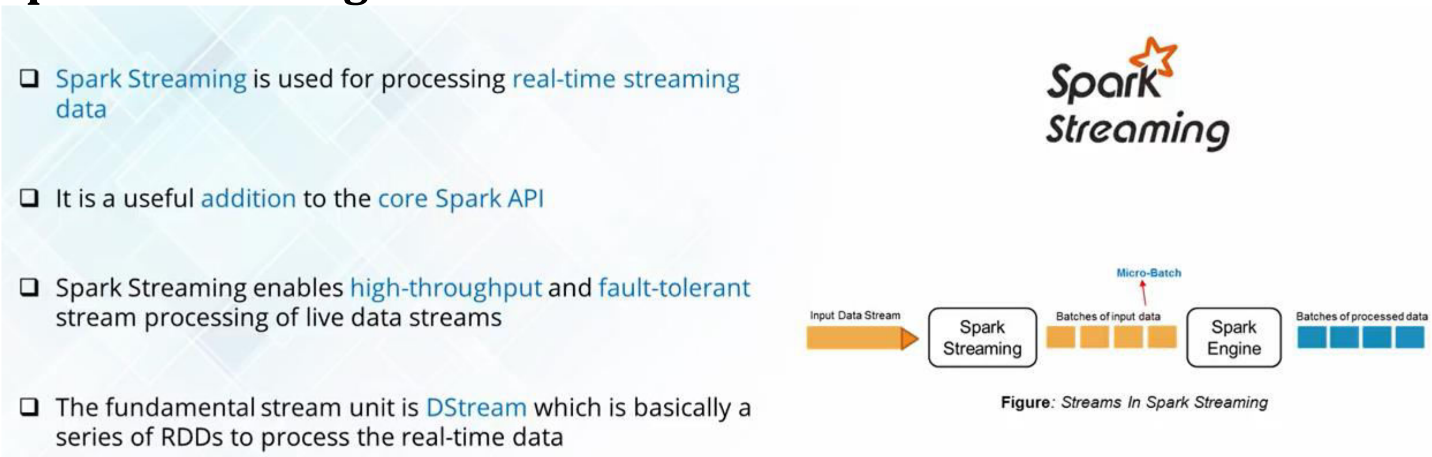
Figure: *What is Streaming?*

Spark Streaming:

- Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
- Spark Streaming can be used to stream live data and processing can happen in real time.
- Data can be ingested from many sources like Twitter, Stock Market, Geographical Systems, Kafka, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
- Finally, processed data can be pushed out to file systems, databases, and live dashboards.
- Spark Streaming's ever-growing user base consists of household names like Uber, Netflix and Pinterest.



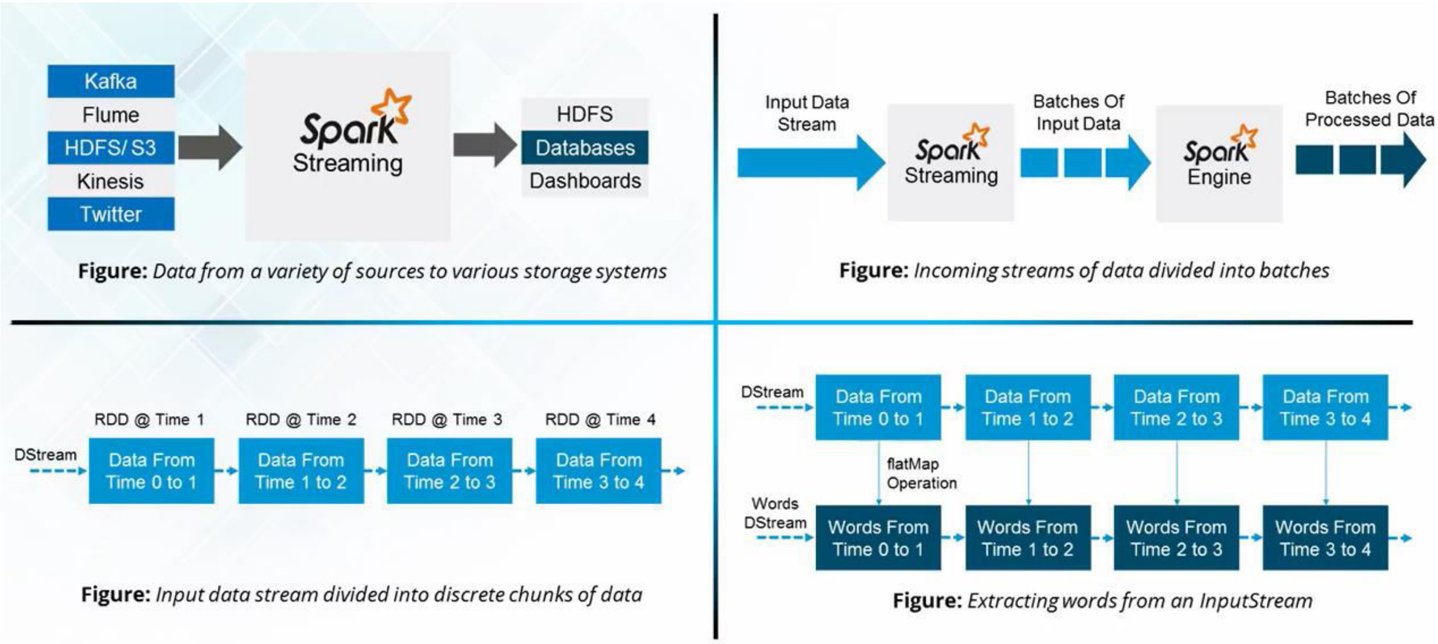
Spark Streaming Overview:



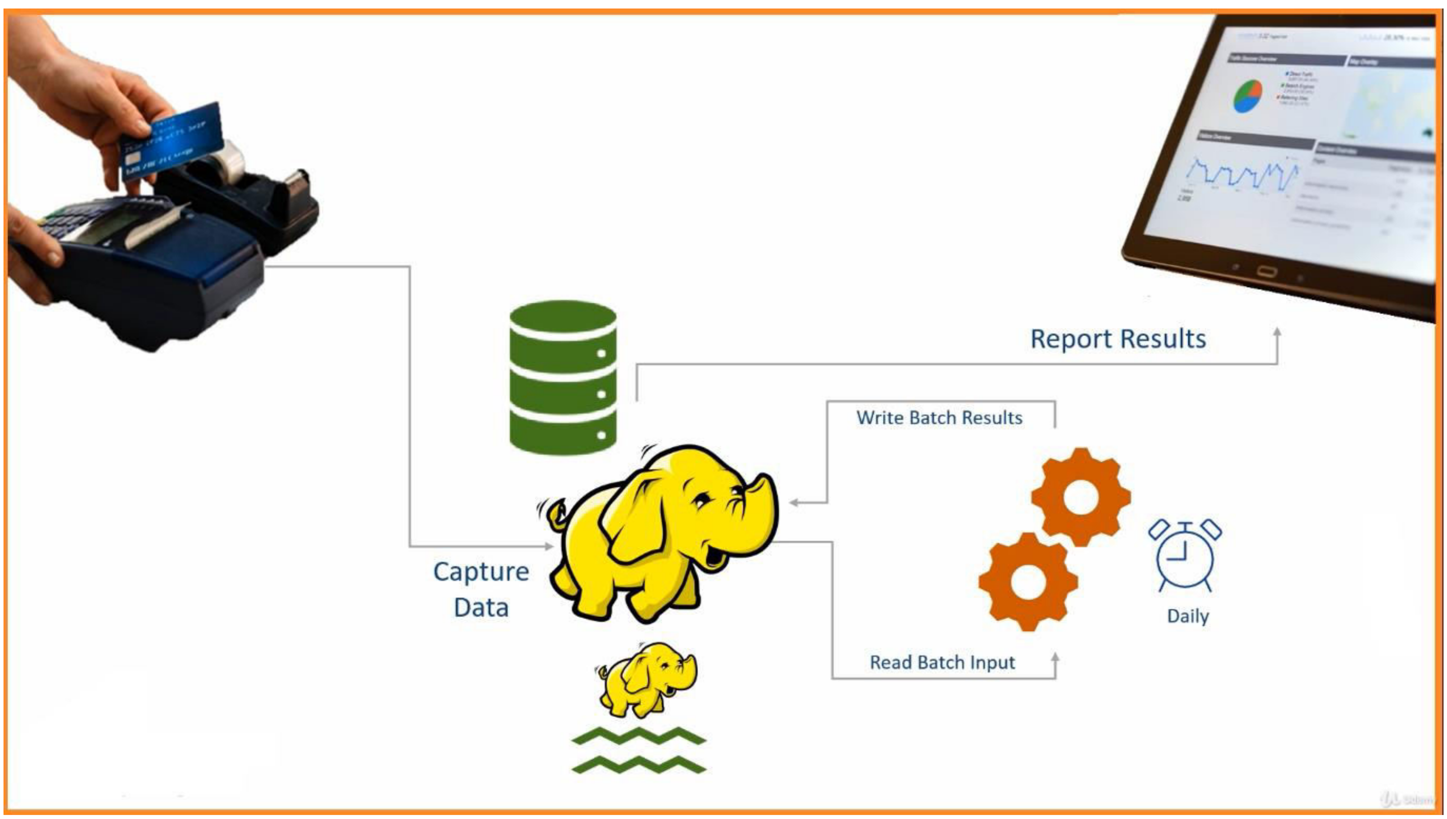
Internally, it works as follows. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

Spark Streaming provides a high-level abstraction called ***discretized stream*** or ***DStream***, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka and Kinesis or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

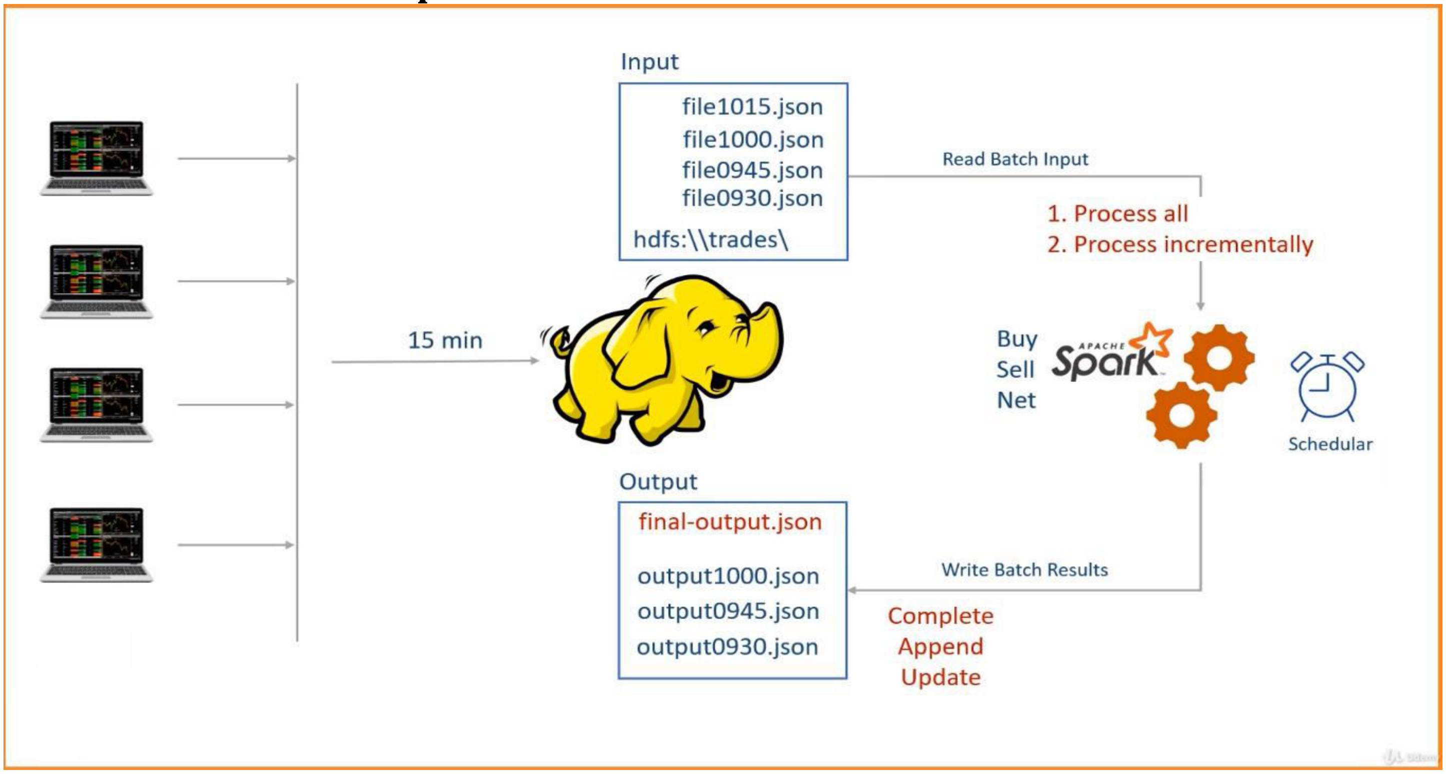
Spark Streaming Workflow:



Data Lake is a process where data is brought & then processed using a batch.



Stock Information Example:



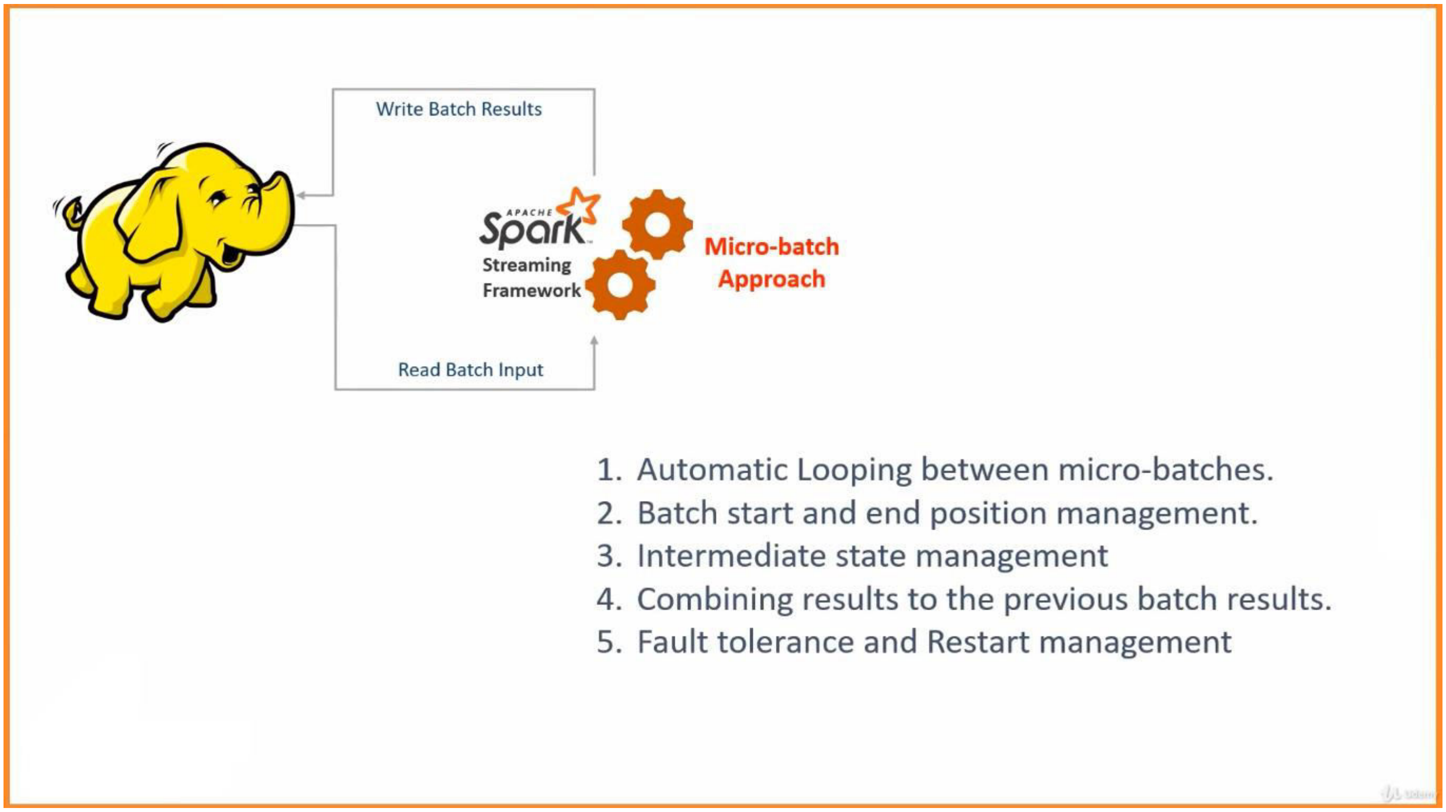
Questions:

1) How many files do I have in directory?
2) Which one was processed last & which one is new?
3) What if the new file is not arrived? Should I wait or terminate the process for next 15 min?
4) What if the current batch fails for some reason? How would I know where to start again?
5) What if the previous batch is still running so should I start or wait for the last batch to finish?
6) Transaction created at 9:29 reach to the Data Lake by 9:44, However the Data Lake is creating its first file at 9:30 so it won't make first file but only make in second file? The sums calculated in first file are incorrect they do not have records between 9:15 to 9:30. Some records are arriving so late. So we need to consider late arriving records.

Key Takeaways:

1) Batch processing is a subset of Stream processing. Hence Stream Processing is a superset of Batch processing.
2) Many problems of batch processing are addressed by stream processing.
3) Stream processing job can take care of scheduling requirements of your batch jobs. It can handle job failures & can start from the point where the job was failed.
4) Streaming Jobs can maintain intermediate results & combined results of previous batches to current batch. Stream Processing is an extension of Dataframe API.

Same approach of Batch Processing taken by Streaming processing **i.e.** processing data in smaller chunks with reduced time frame.



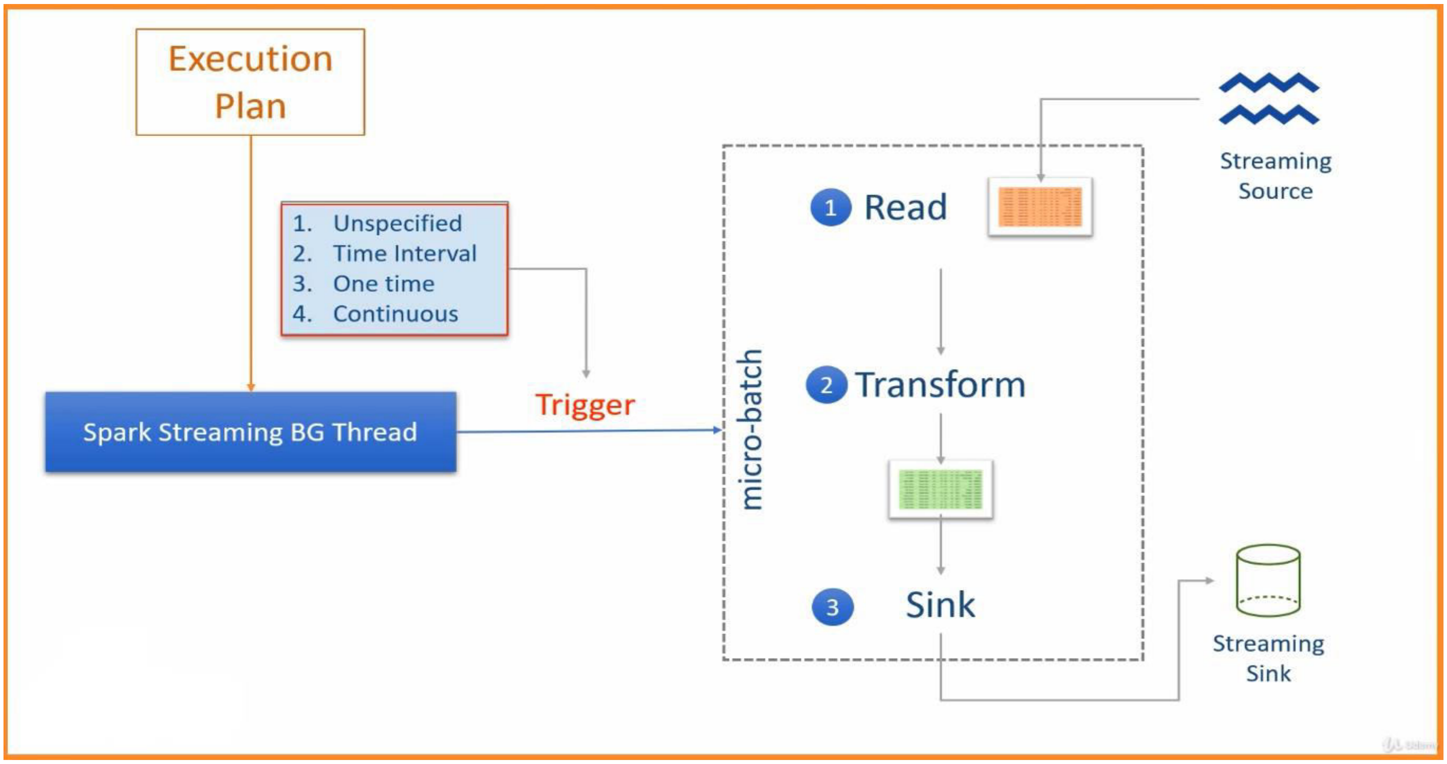
DStreams VS Dataframe API:

Spark Streaming (DStream) API	Structured Streaming API
1. RDD based Streaming API 2. Lacks Spark SQL Engine Optimization 3. No Support for Even Time Semantics 4. No future upgrades and enhancements expected	1. Dataframe based Streaming API 2. SQL Engine Optimization 3. Supports Event Time Semantics 4. Expected further enhancements and new features

1) Streaming Word Count:

2) Socket Streaming with Schema & Flatten Data:

Stream Processing Model in Spark:

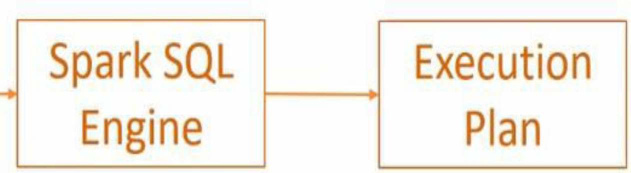


Spark SQL Engine: Analyze the code, **optimize** the code & **compiles** it to generate Execution Plan.

```
# Read
lines_df = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", "9999") \
    .load()

# Transform
words_df = lines_df.select(expr("explode(split(value,' ')) as word"))
counts_df = words_df.groupBy("word").count()

# Sink
word_count_query = counts_df.writeStream \
    .format("console") \
    .option("checkpointLocation", "chk-point-dir") \
    .outputMode("complete") \
    .start()
```



Spark Web UI for Word Count Messages:

1) Empty
2) Hello Spark Streaming
3) Hello Spark

▾ Completed Jobs (3)

Page: 1   1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id (Job Group) ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2 (658335c4-7c84-4188-b9c6-c11958bae8ad)	id = 64281b1e-280b-4b1a-86f8-f9a1624a7a82 runId = 658335c4-7c84-4188-b9c6-c11958bae8ad batch = 2 start at NativeMethodAccessorImpl.java:0	2020/09/04 20:11:17	0.4 s	2/2	6/6
1 (658335c4-7c84-4188-b9c6-c11958bae8ad)	id = 64281b1e-280b-4b1a-86f8-f9a1624a7a82 runId = 658335c4-7c84-4188-b9c6-c11958bae8ad batch = 1 start at NativeMethodAccessorImpl.java:0	2020/09/04 20:10:09	0.6 s	2/2	6/6
0 (658335c4-7c84-4188-b9c6-c11958bae8ad)	id = 64281b1e-280b-4b1a-86f8-f9a1624a7a82 runId = 658335c4-7c84-4188-b9c6-c11958bae8ad batch = 0 start at NativeMethodAccessorImpl.java:0	2020/09/04 20:07:40	1 s	2/2	6/6

Page: 1   1 Pages. Jump to 1 . Show 100 items in a page. Go

Spark Streaming Sources:

1) Socket Source
2) File Source
3) Kafka Source

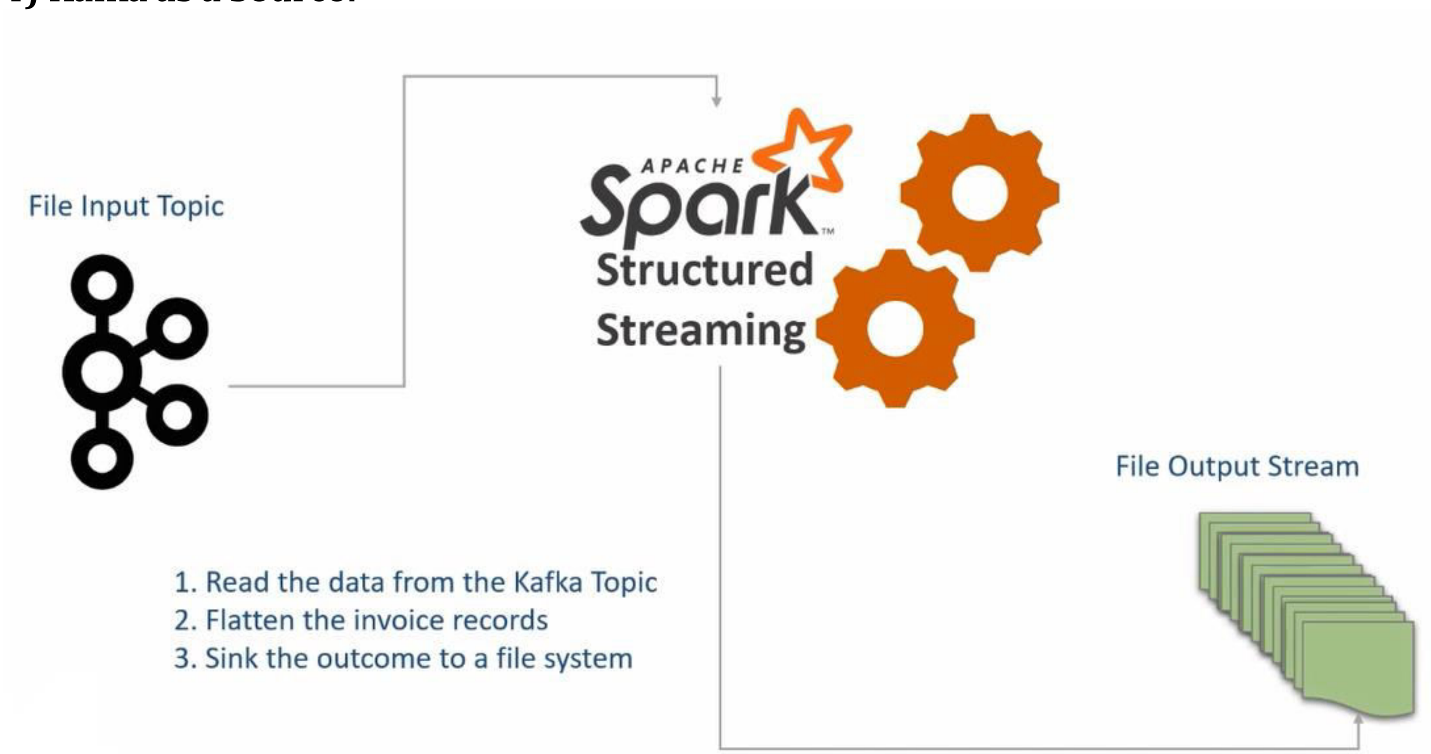
3) File Stream as Source: Invoice Json

```
{
    "InvoiceNumber":"91372973",
    "CreatedTime":1595688901219,
    "StoreID":"STR8513",
    "PosID":"POS163",
    "CustomerType":"PRIME",
    "PaymentMethod":"CARD",
    "DeliveryType":"HOME-DELIVERY",
    "City":"Shivapuri",
    "State":"Madhya Pradesh",
    "PinCode":"561012",
    "ItemCode":"413",
    "ItemDescription":"Slipcover",
    "ItemPrice":1896.0,
    "ItemQty":1,
    "TotalValue":1896.0
}
```


Output Modes:

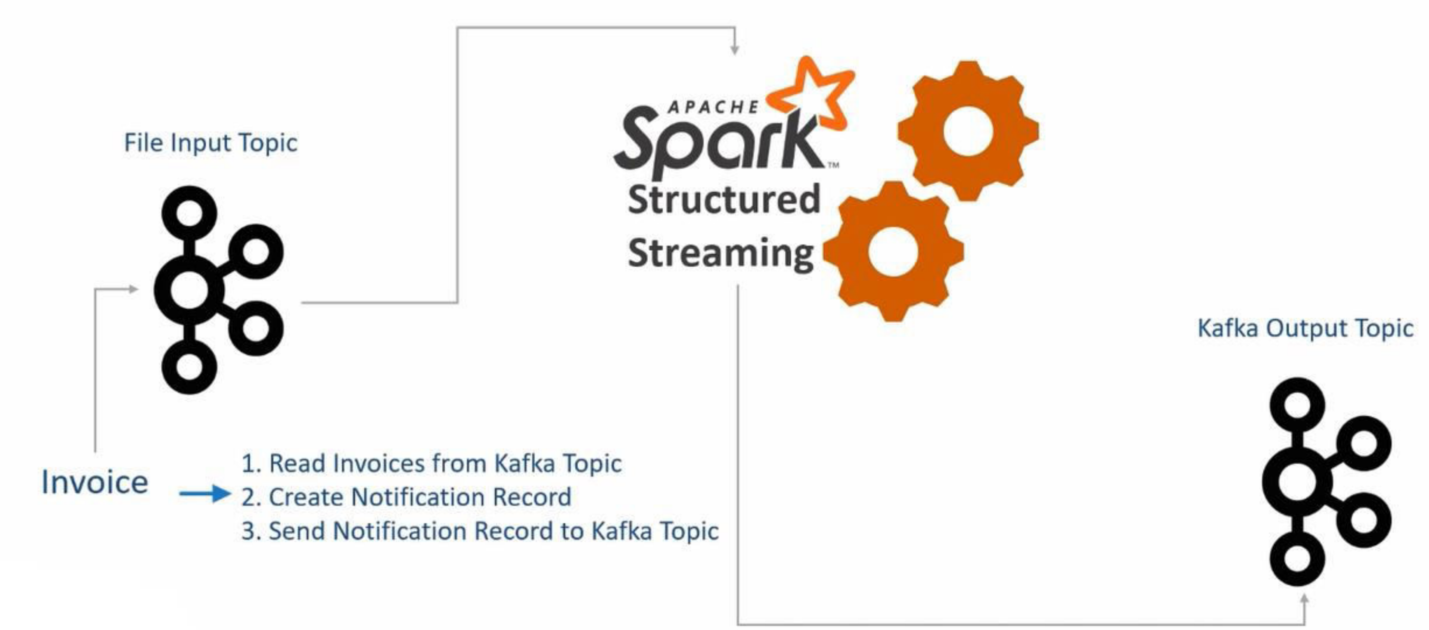
1) Append → Insert
2) Update → Upsert
3) Complete → Overwrite

4) Kafka as a Source:

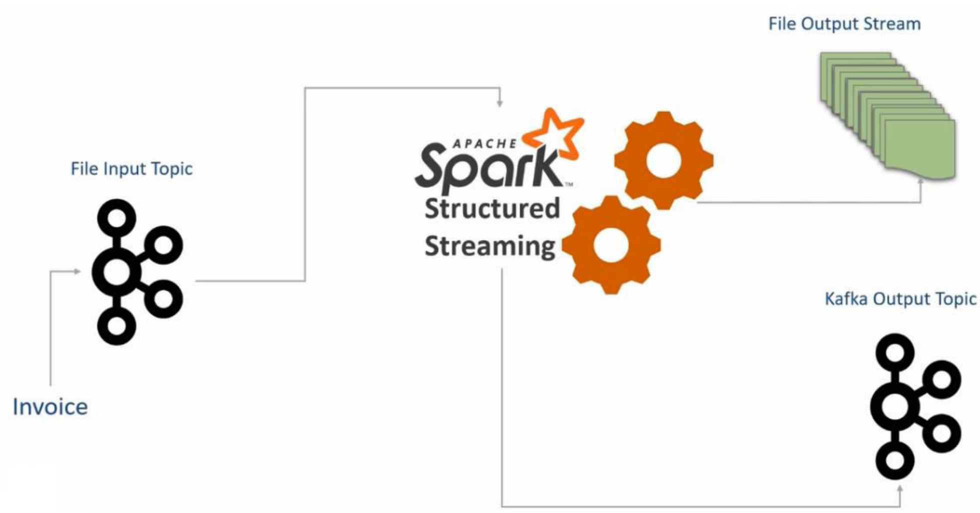


5) Read Kafka & Send Notifications to Kafka Topic:

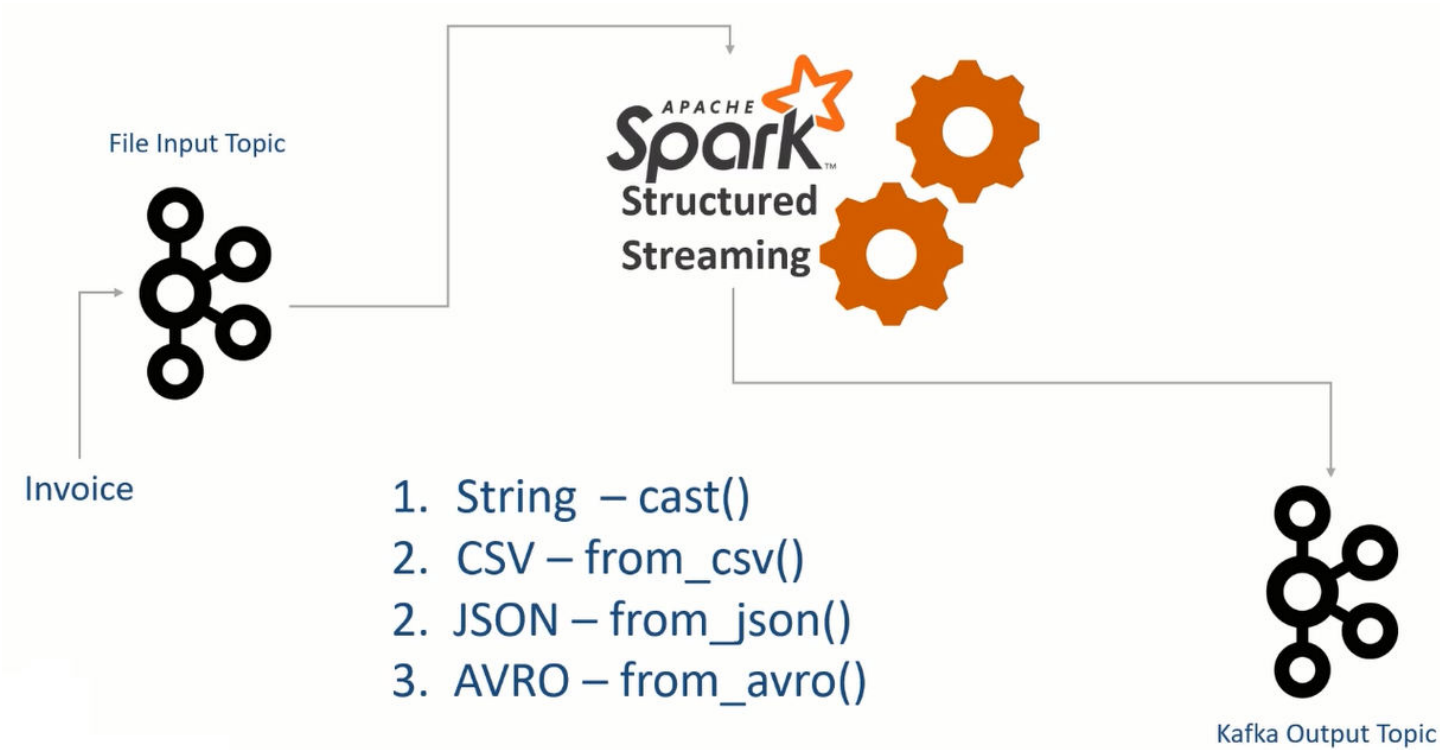
{"CustomerCardNo":"4629185211","TotalAmount":11114.0,"EarnedLoyaltyPoints":2222.8}



Multi-Query Implementation: **Give it as Assignment to Student**



Different Formats Reads in Kafka:



Formats	Read	Write
csv	from_csv	to_csv
json	from_json	to_json
avro	from_avro	to_avro

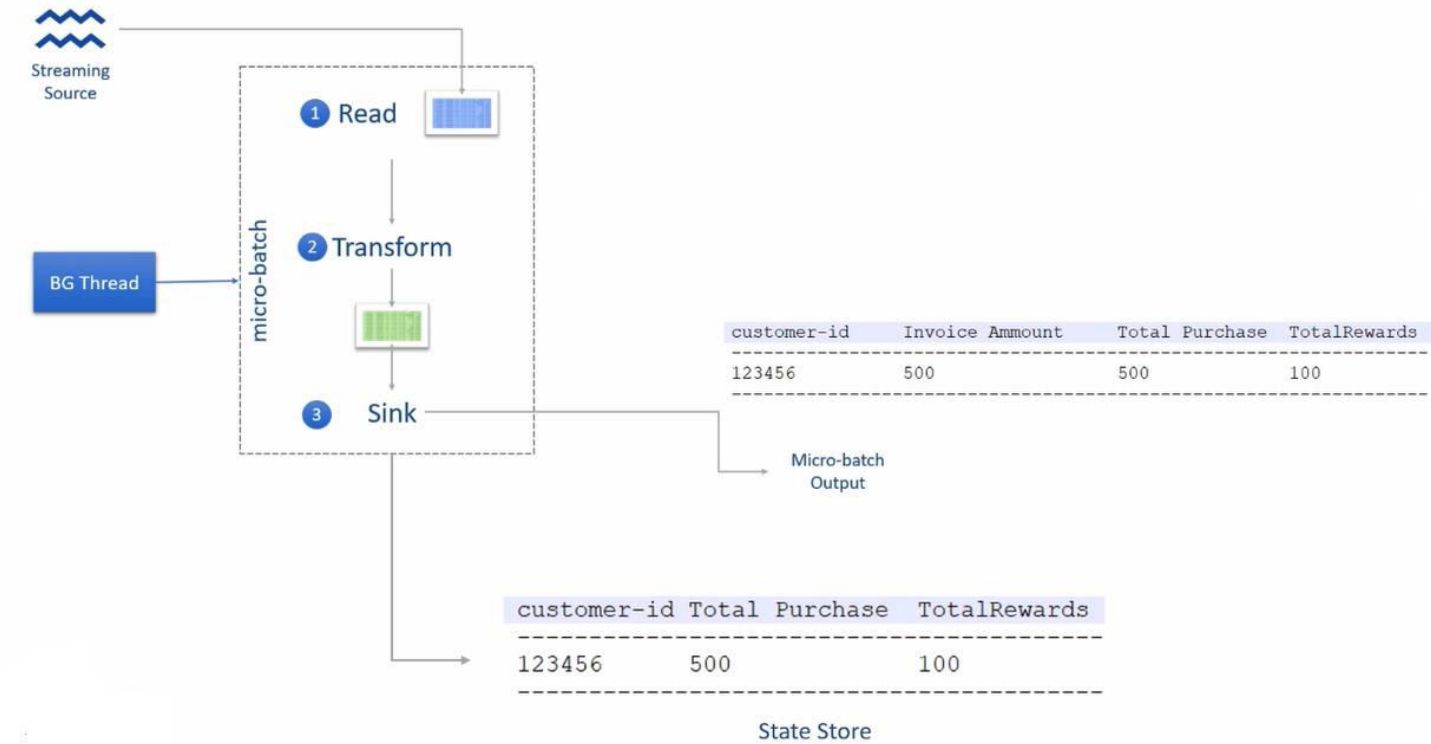
6) Read Kafka Topic Json data & Write Avro format:

7) Read Avro Topic Data & Write to Kafka Topic in Json format:

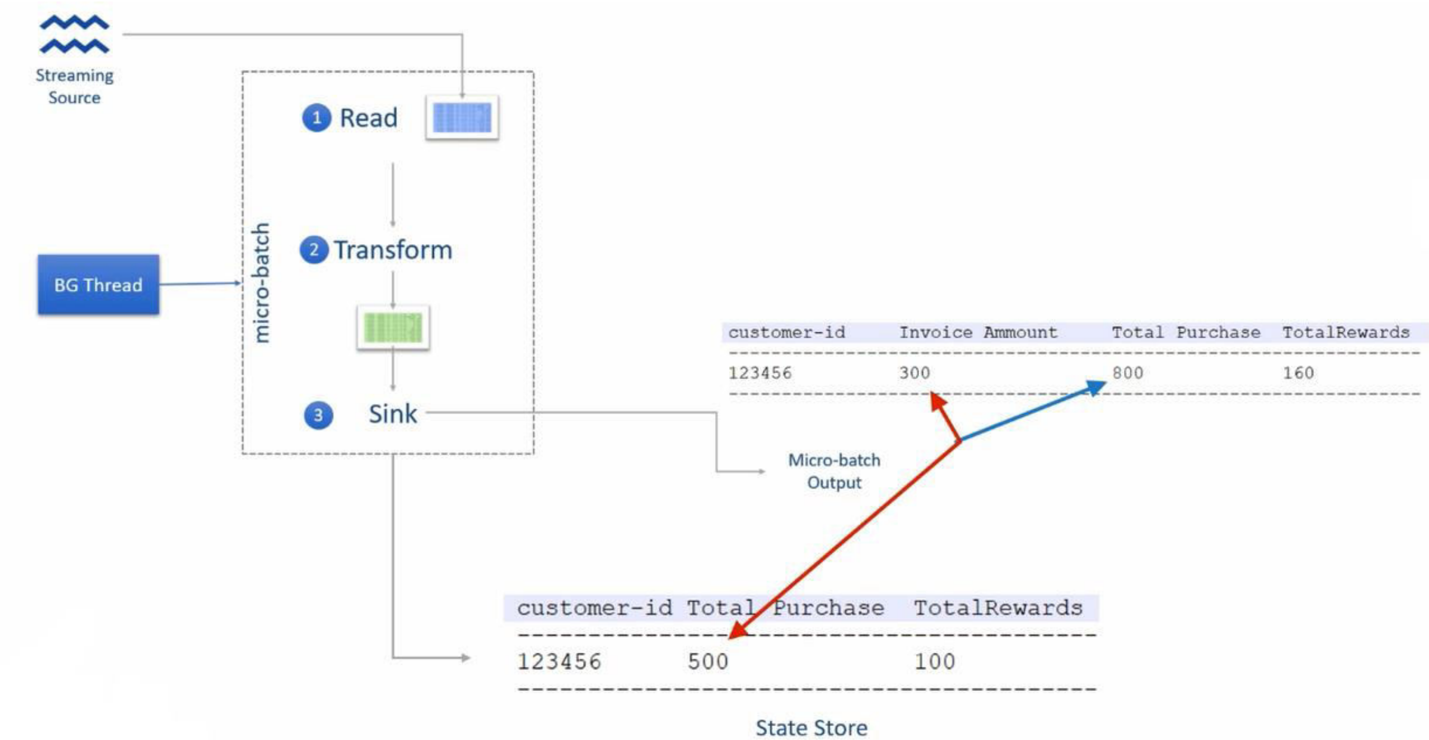
Stateless Vs Stateful Transformations:

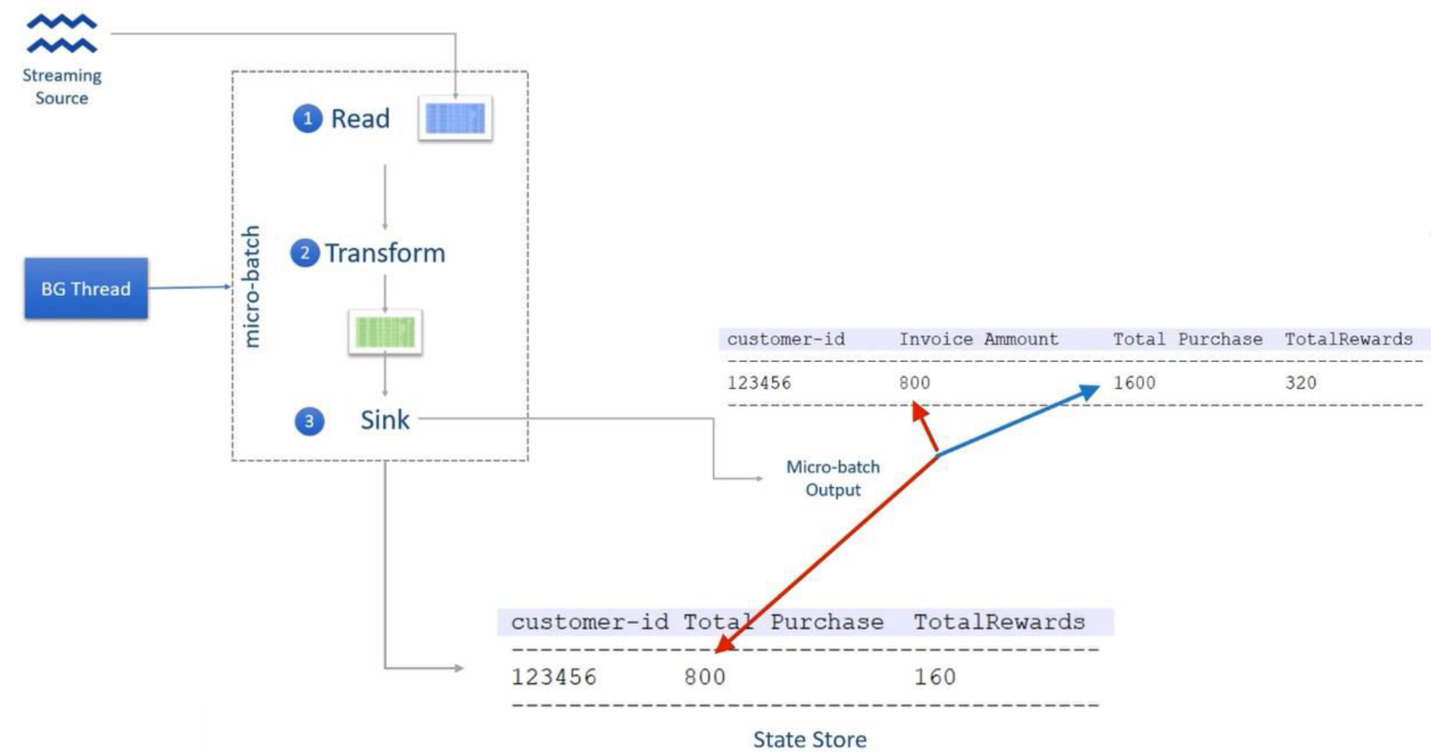
What is the state?

1st Micro batch:



2nd Micro batch:





1) Stateless means that the logic of handling the new data is independent of the previous data.
E.g. select ( ), filter ( ), map ( ), flatMap ( ), explode ( ) etc.
Drawback: Stateless transformation will not support **COMPLETE** output mode.

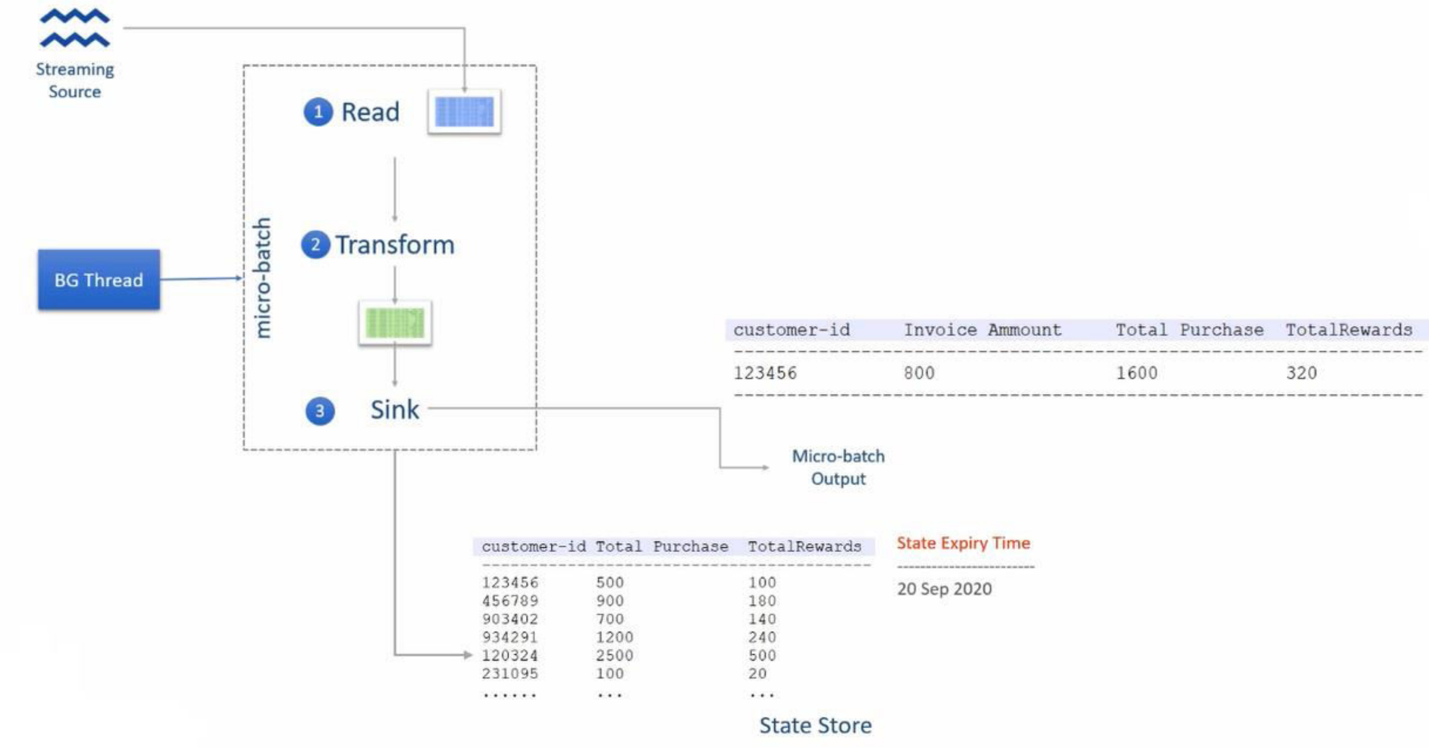
2) Stateful in contrast to stateless, it means that you need somehow combine the data with old records or previous batches.
E.g. Grouping, Aggregations, Windowing & Joins.
Drawbacks: Excessive state causes out of memory

- **a)** Managed Stateful Operations
- **b)** Unmanaged Stateful Operations → Not Supported in PySpark

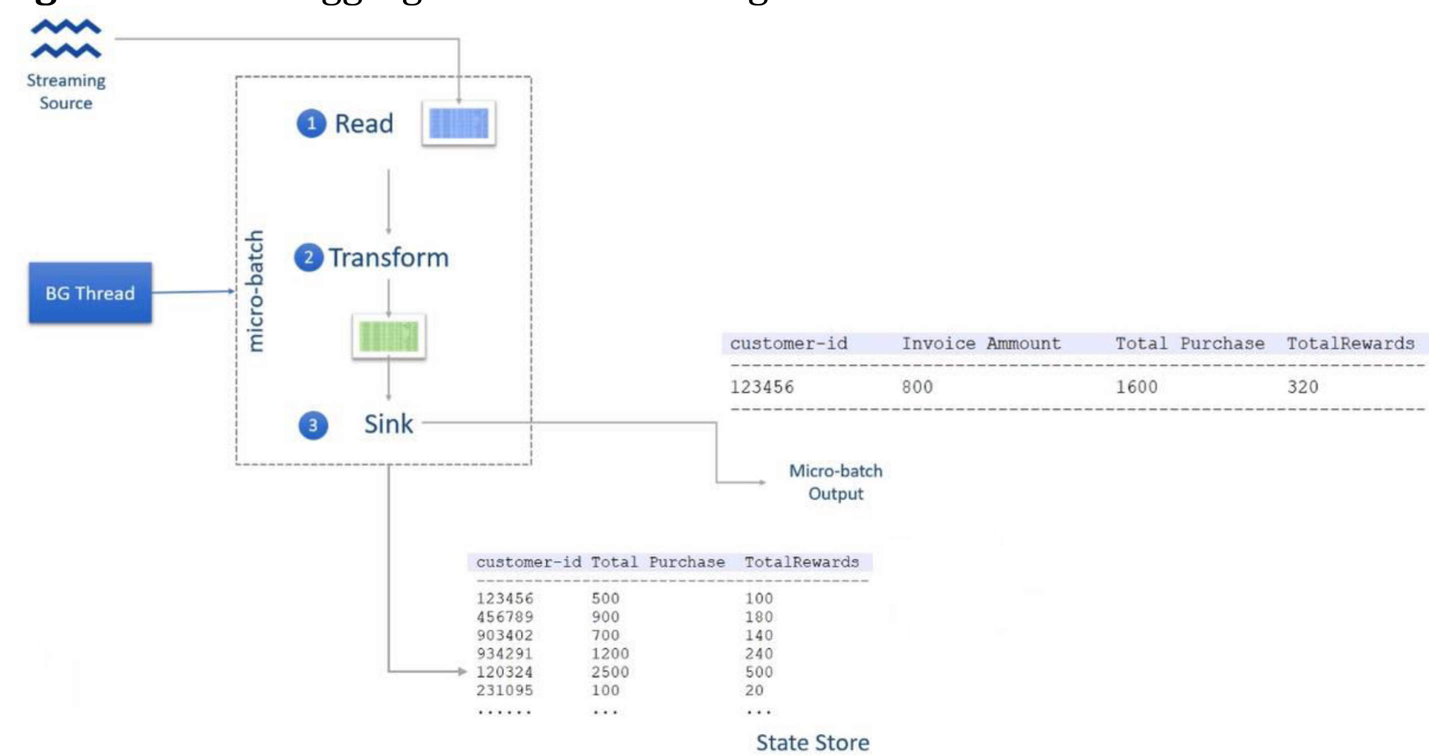
Q) So, when to use Managed & Unmanaged Stateful Operations?
A) Based on Aggregations:

1) Continuous
2) Time-Bound

E.g. Time-Bound Aggregations → Managed Stateful

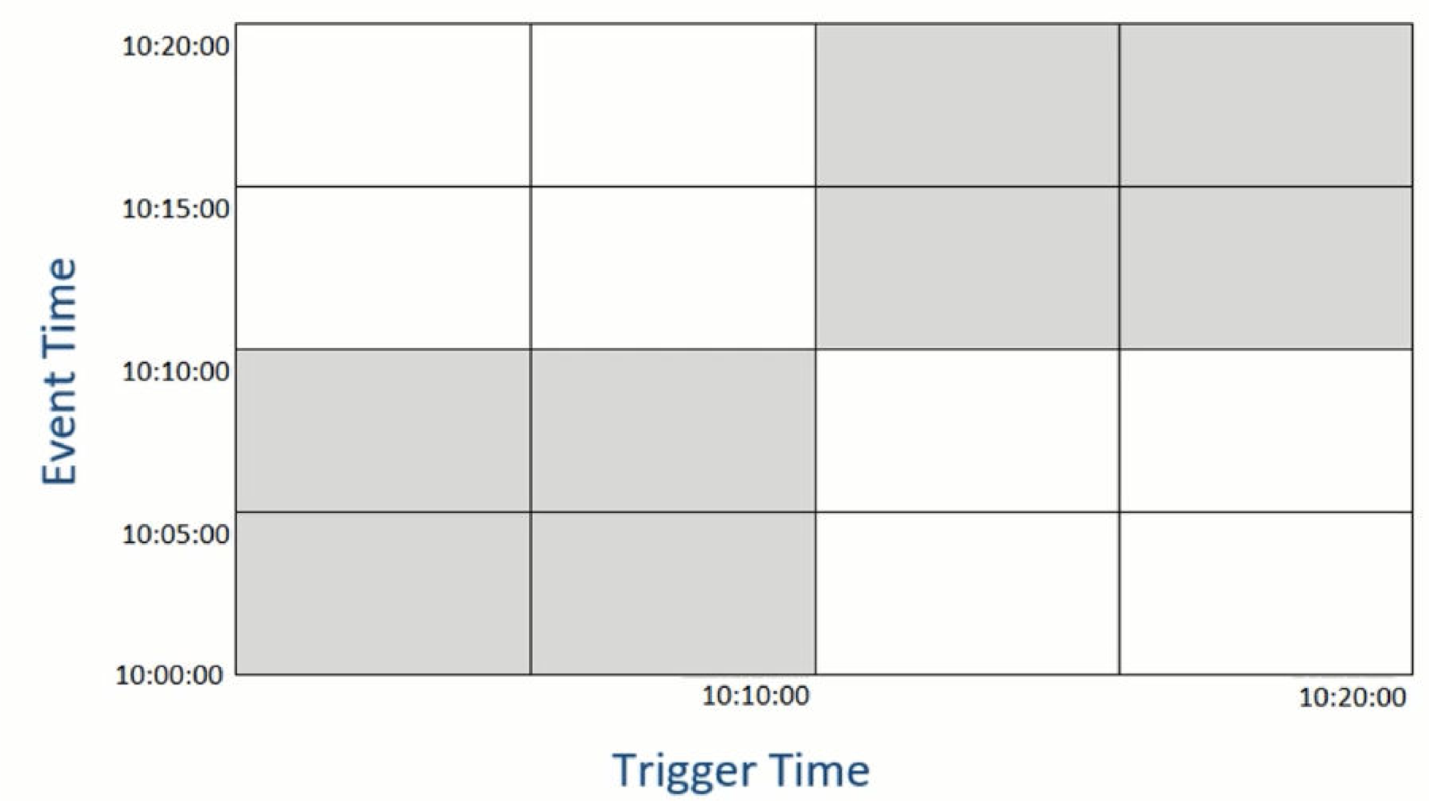


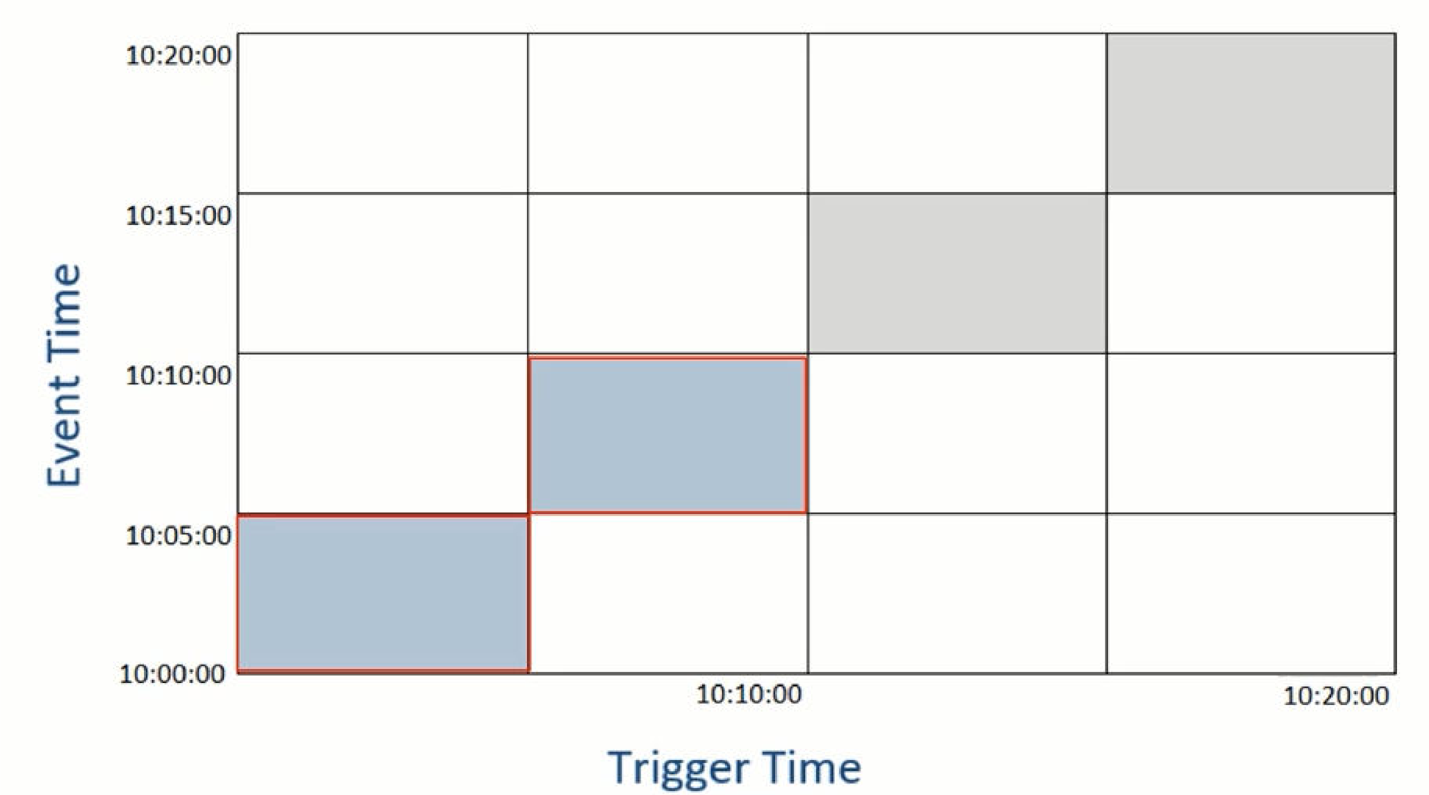
E.g. Continuous Aggregations → Unmanaged Stateful

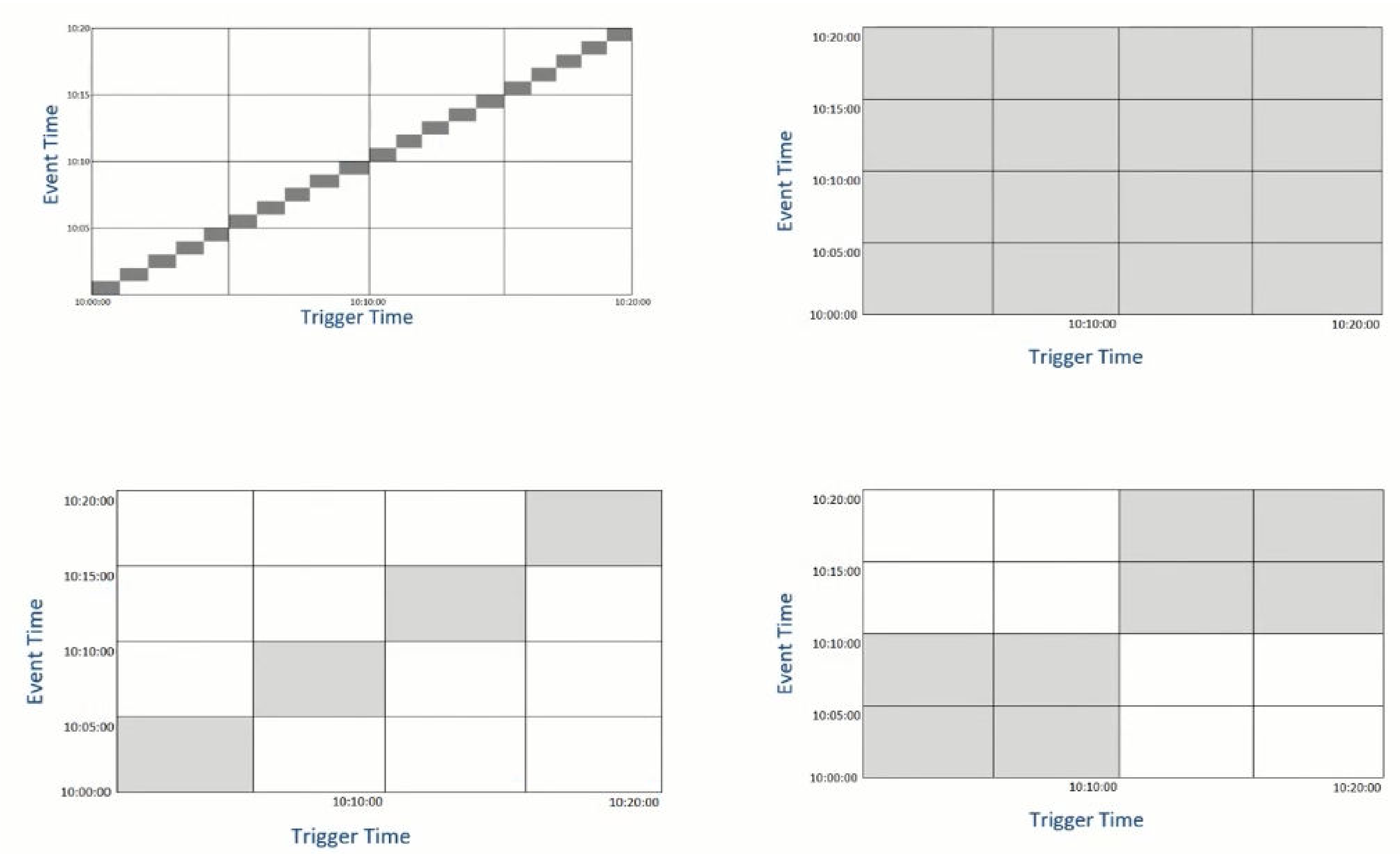


Window Aggregates:

1) Tumbling Time Window
2) Sliding Time Window







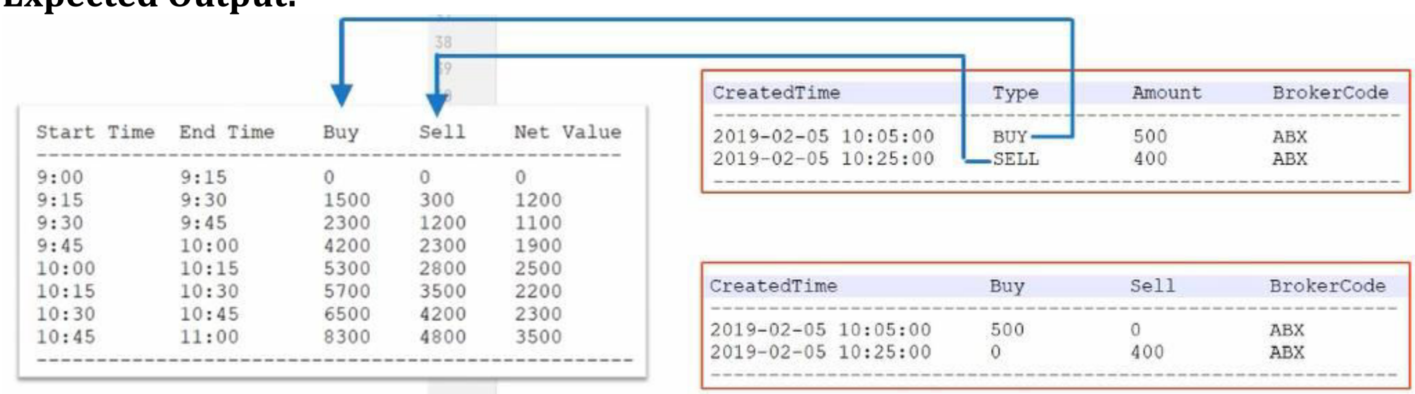

```
{"TransactionTime": "2019-02-05 10:01:12",
 "TransactionType": "BUY",
 "TransactionAmount": 500,
 "BrokerCode": "ABX"}

{"TransactionTime": "2019-02-05 10:05:30",
 "TransactionType": "SELL",
 "TransactionAmount": 300, ,
 "BrokerCode": "ABX"}
```


Start Time	End Time	Buy	Sell	Net Value
9:00	9:15	0	0	0
9:15	9:30	1500	300	1200
9:30	9:45	2300	1200	1100
9:45	10:00	4200	2300	1900
10:00	10:15	5300	2800	2500
10:15	10:30	5700	3500	2200
10:30	10:45	6500	4200	2300
10:45	11:00	8300	4800	3500

8) Tumbling Window:

Expected Output:



Start Time	End Time	Buy	Sell	Net Value
9:00	9:15	0	0	0
9:15	9:30	1500	300	1200
9:30	9:45	2300	1200	1100
9:45	10:00	4200	2300	1900
10:00	10:15	5300	2800	2500
10:15	10:30	5700	3500	2200
10:30	10:45	6500	4200	2300
10:45	11:00	8300	4800	3500

CreatedTime	Type	Amount	BrokerCode
2019-02-05 10:05:00	BUY	500	ABX
2019-02-05 10:25:00	SELL	400	ABX

CreatedTime	Buy	Sell	BrokerCode
2019-02-05 10:05:00	500	0	ABX
2019-02-05 10:25:00	0	400	ABX

start	end	TotalBuy	TotalSell
2019-02-05 10:00:00	2019-02-05 10:15:00	800	0
2019-02-05 10:15:00	2019-02-05 10:30:00	800	400
2019-02-05 10:30:00	2019-02-05 10:45:00	900	0
2019-02-05 10:45:00	2019-02-05 11:00:00	0	600

Time	Buy	Sell	Net Value
9:15	0	0	0
9:30	1500	300	1200
9:45	2300	1200	1100
10:00	4200	2300	1900
10:15	5300	2800	2500
10:30	5700	3500	2200
10:45	6500	4200	2300
11:00	8300	4800	3500
11:15	9000	6200	2800
11:30	10200	7300	2900
11:45	10700	8900	1800
12:00	12300	10700	1600
12:15	12900	11200	1700
12:30	13400	11800	1600
12:45	14200	12300	1900
13:00	15300	12500	2800
13:15	15800	12800	3000
13:30	17200	13200	4000
13:45	17800	13300	4500
14:00	19050	13800	5250
14:15	19400	14200	5200
14:30	19930	15700	4230
14:45	21800	16300	5500
15:00	22520	17200	5320
15:15	22900	17900	5000
15:30	23000	20300	2700

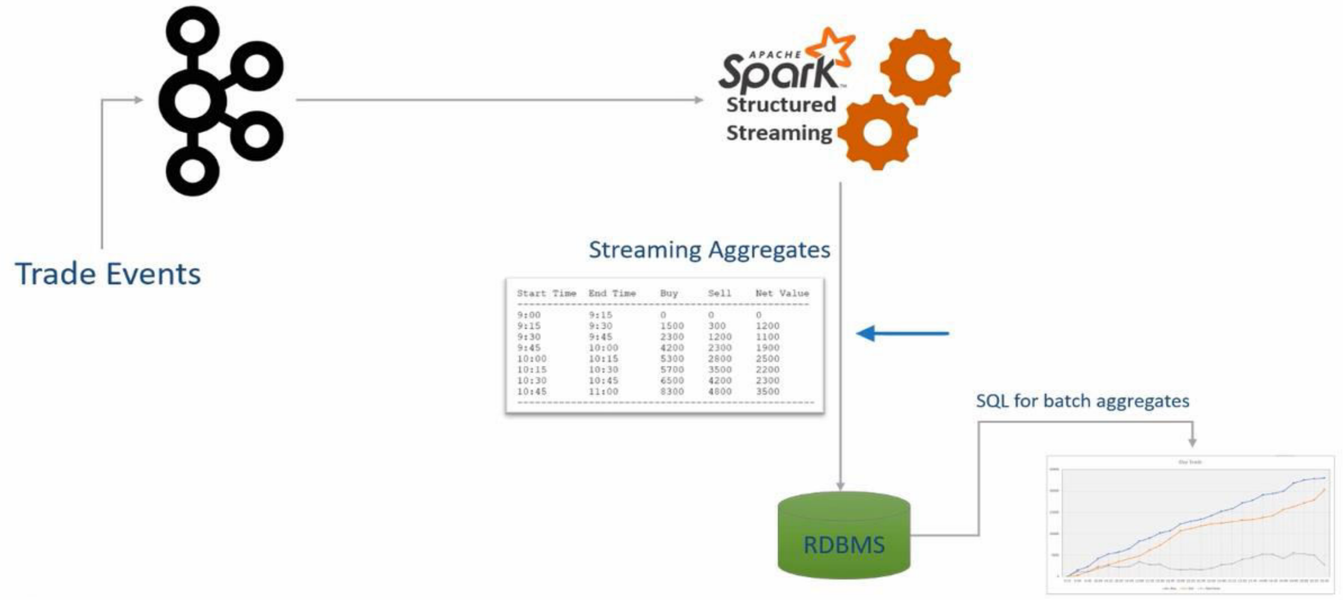

```
+-------------------+-------------------+--------+---------+
|              start|                end|TotalBuy|TotalSell|
+-------------------+-------------------+--------+---------+
|2019-02-05 10:00:00|2019-02-05 10:15:00|     800|        0|
|2019-02-05 10:15:00|2019-02-05 10:30:00|     800|      400|
|2019-02-05 10:30:00|2019-02-05 10:45:00|     900|        0|
|2019-02-05 10:45:00|2019-02-05 11:00:00|       0|      600|
+-------------------+-------------------+--------+---------+
```

```
+---------+----------+--------+
|RTotalBuy|RTotalSell|NetValue|
+---------+----------+--------+
|800      |0         |800     |
|1600     |400       |1200    |
|2500     |400       |2100    |
|2500     |1000      |1500    |
+---------+----------+--------+
```

```
+-------------------+-------------------+--------+---------+---------+----------+--------+
|start              |end                |TotalBuy|TotalSell|RTotalBuy|RTotalSell|NetValue|
+-------------------+-------------------+--------+---------+---------+----------+--------+
|2019-02-05 10:00:00|2019-02-05 10:15:00|800     |0        |800      |0         |800     |
|2019-02-05 10:15:00|2019-02-05 10:30:00|800     |400      |1600     |400       |1200    |
|2019-02-05 10:30:00|2019-02-05 10:45:00|900     |0        |2500     |400       |2100    |
|2019-02-05 10:45:00|2019-02-05 11:00:00|0       |600      |2500     |1000      |1500    |
+-------------------+-------------------+--------+---------+---------+----------+--------+
```

Note: Spark Streaming does not support analytical functions & windowing aggregates inside a streaming query.

Below is the **solution/workaround** for now:

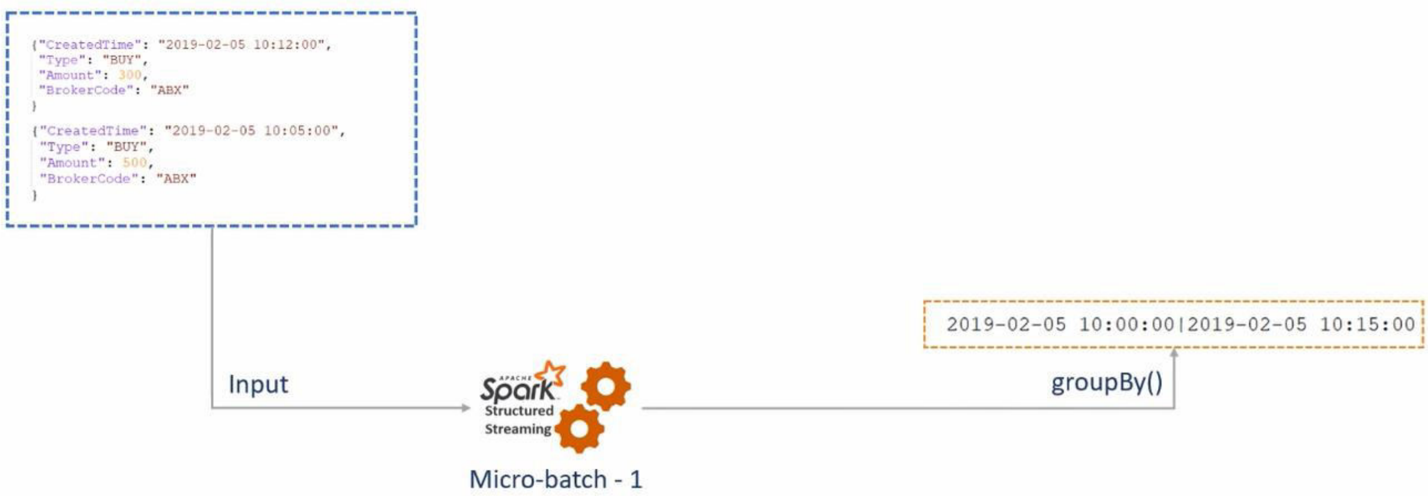


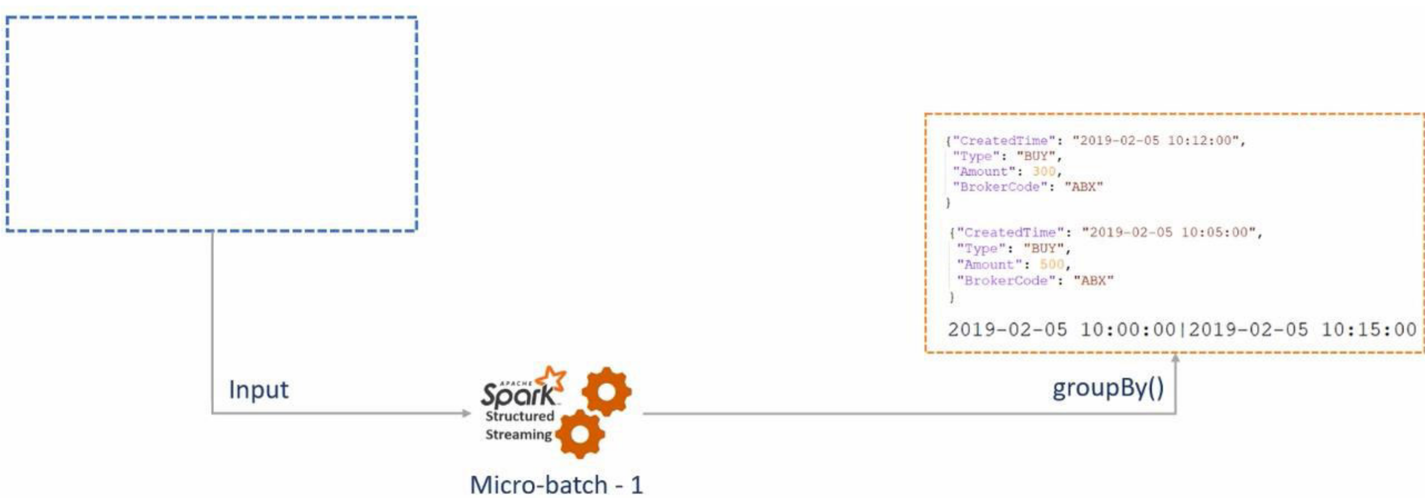
Watermarking:

start	end	TotalBuy	TotalSell
2019-02-05 10:00:00	2019-02-05 10:15:00	800	0
2019-02-05 10:15:00	2019-02-05 10:30:00	800	400
2019-02-05 10:30:00	2019-02-05 10:45:00	900	0
2019-02-05 10:45:00	2019-02-05 11:00:00	0	600

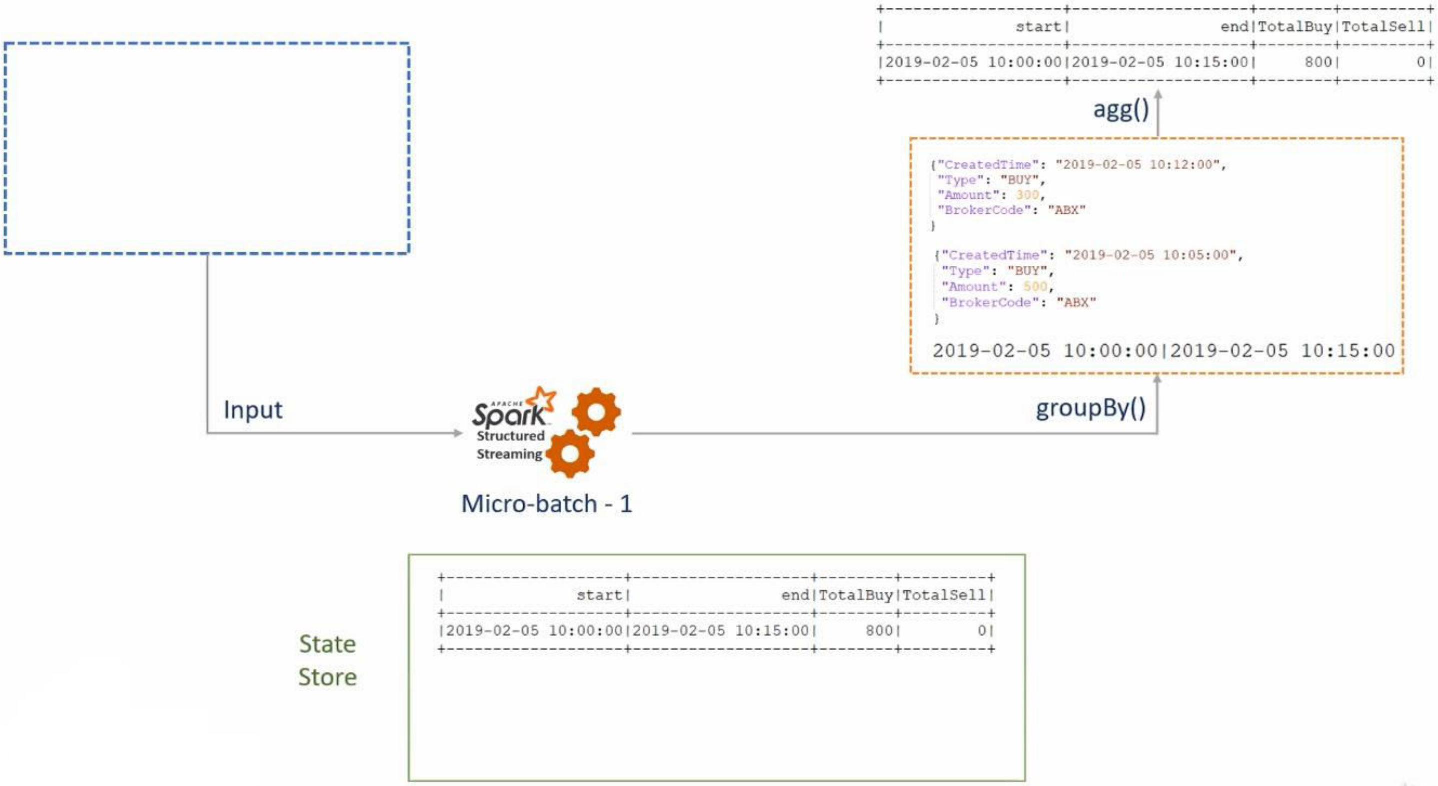
How exactly it happens let's see:

1st Micro Batch:

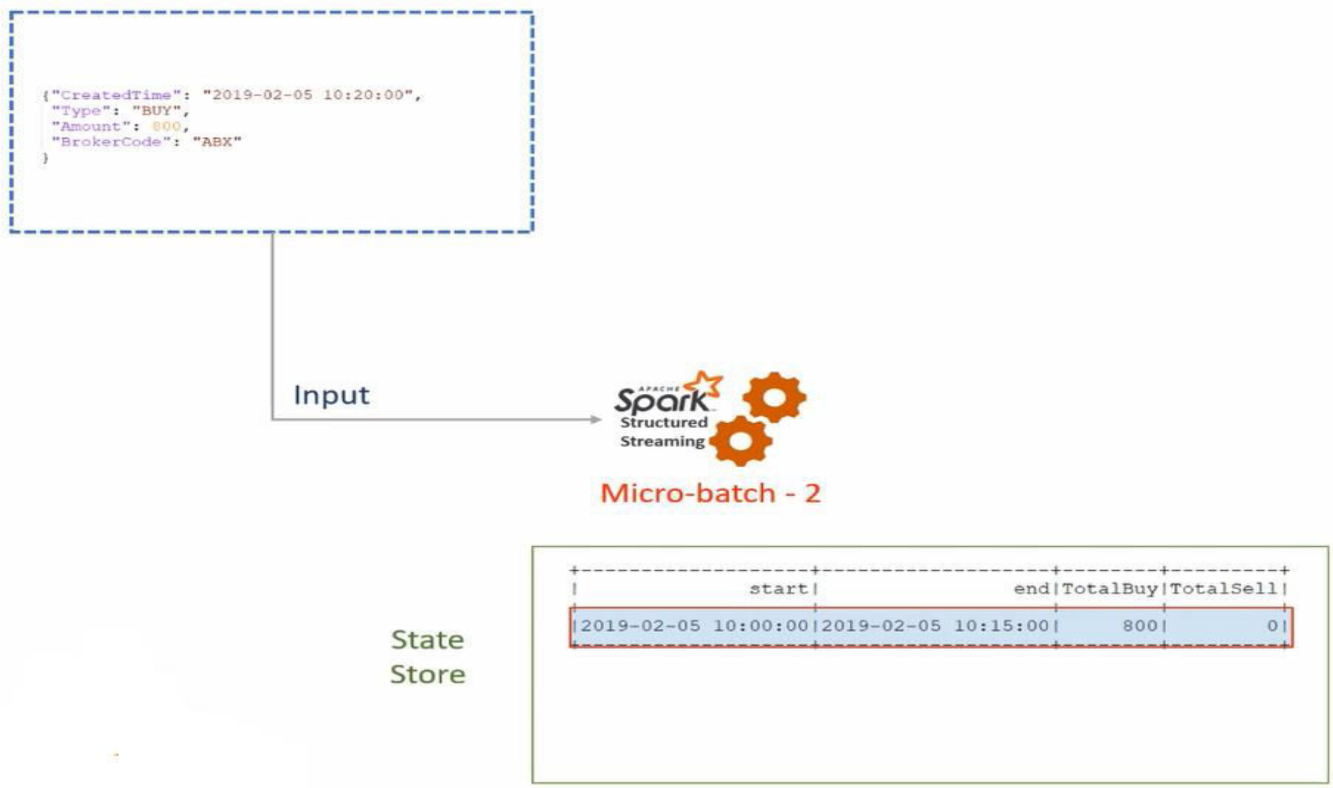




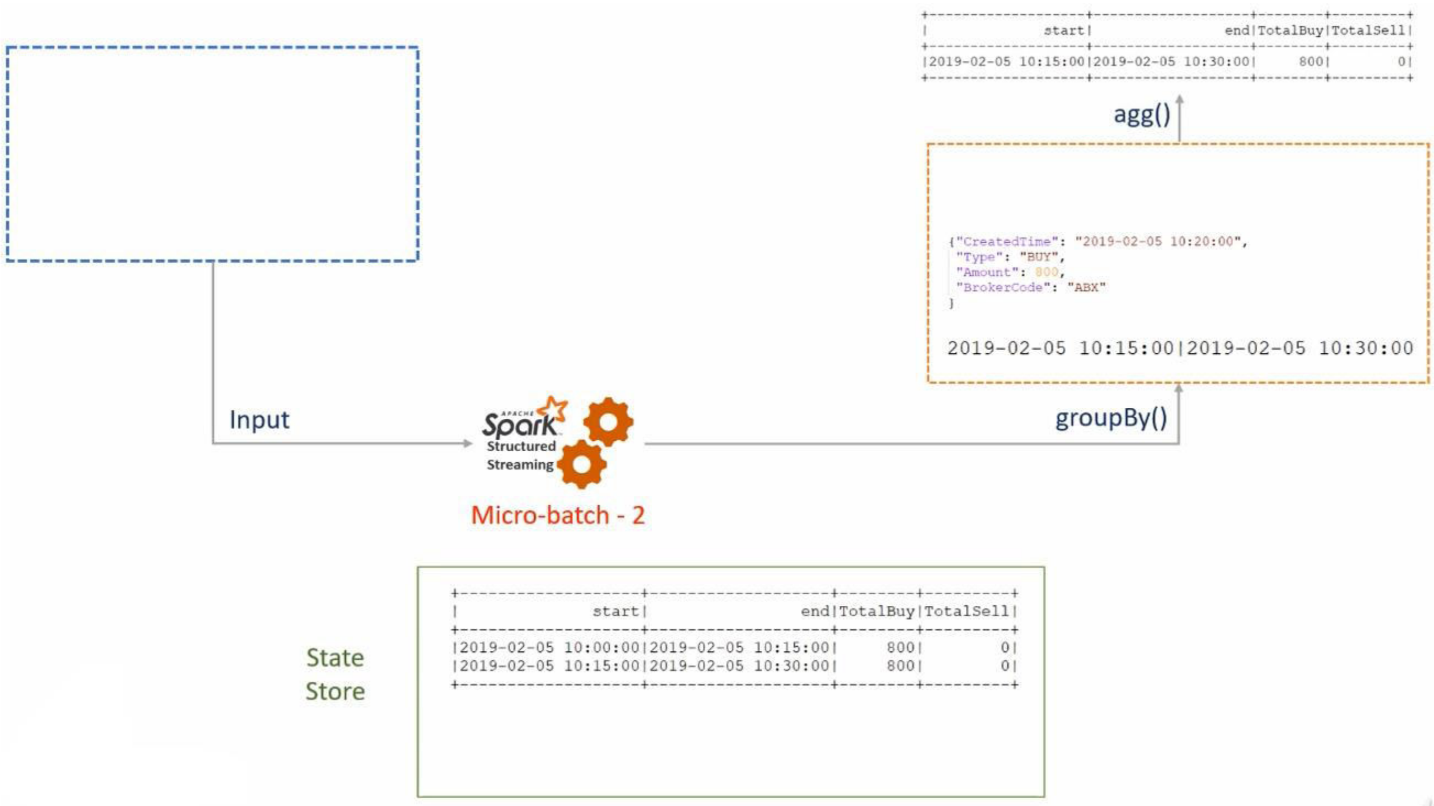
Now it's time to compute aggregates for the records received:



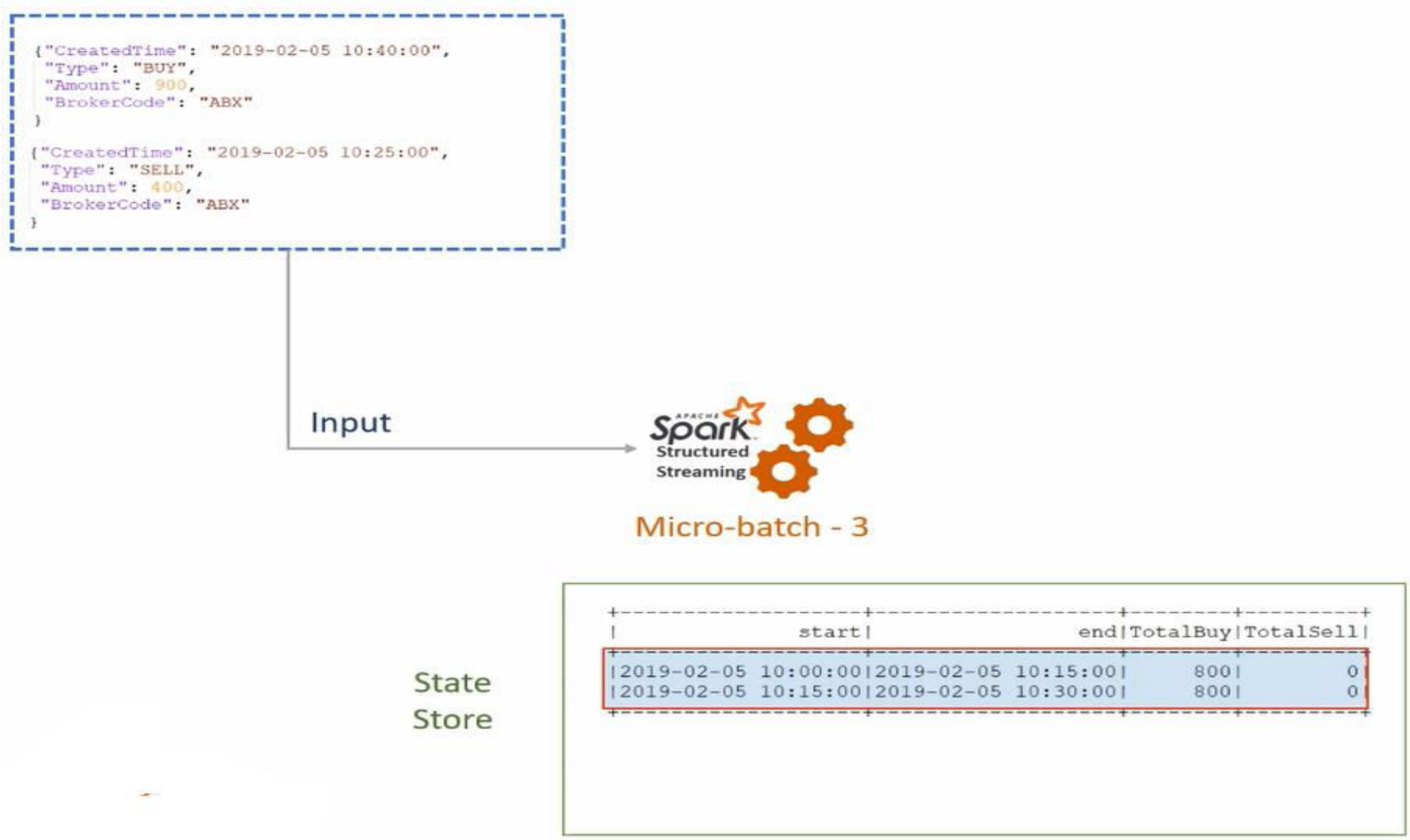
2nd Micro Batch:



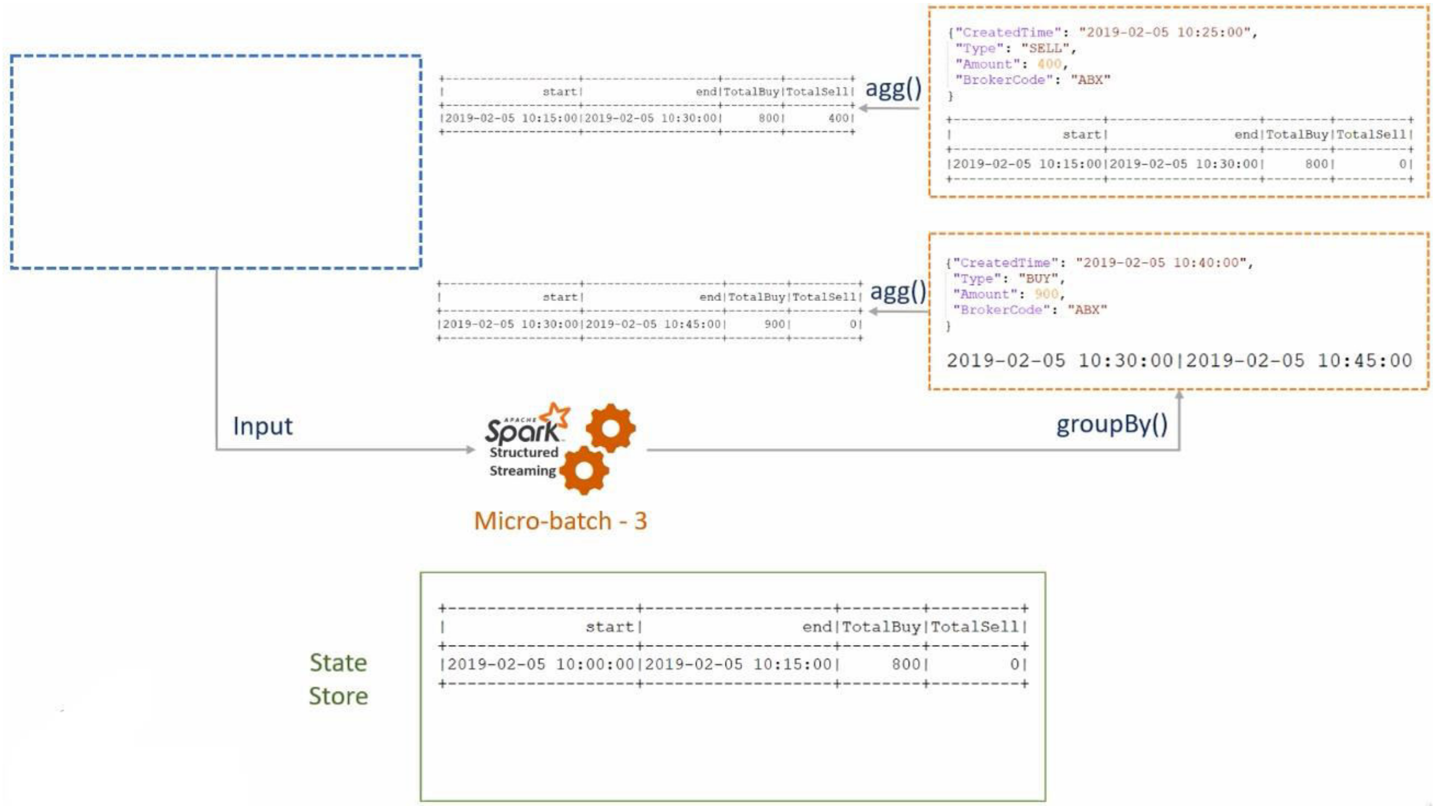
2nd Micro Batch does not fit in 1st Window hence it creates second state of 15mins & computes the aggregates.



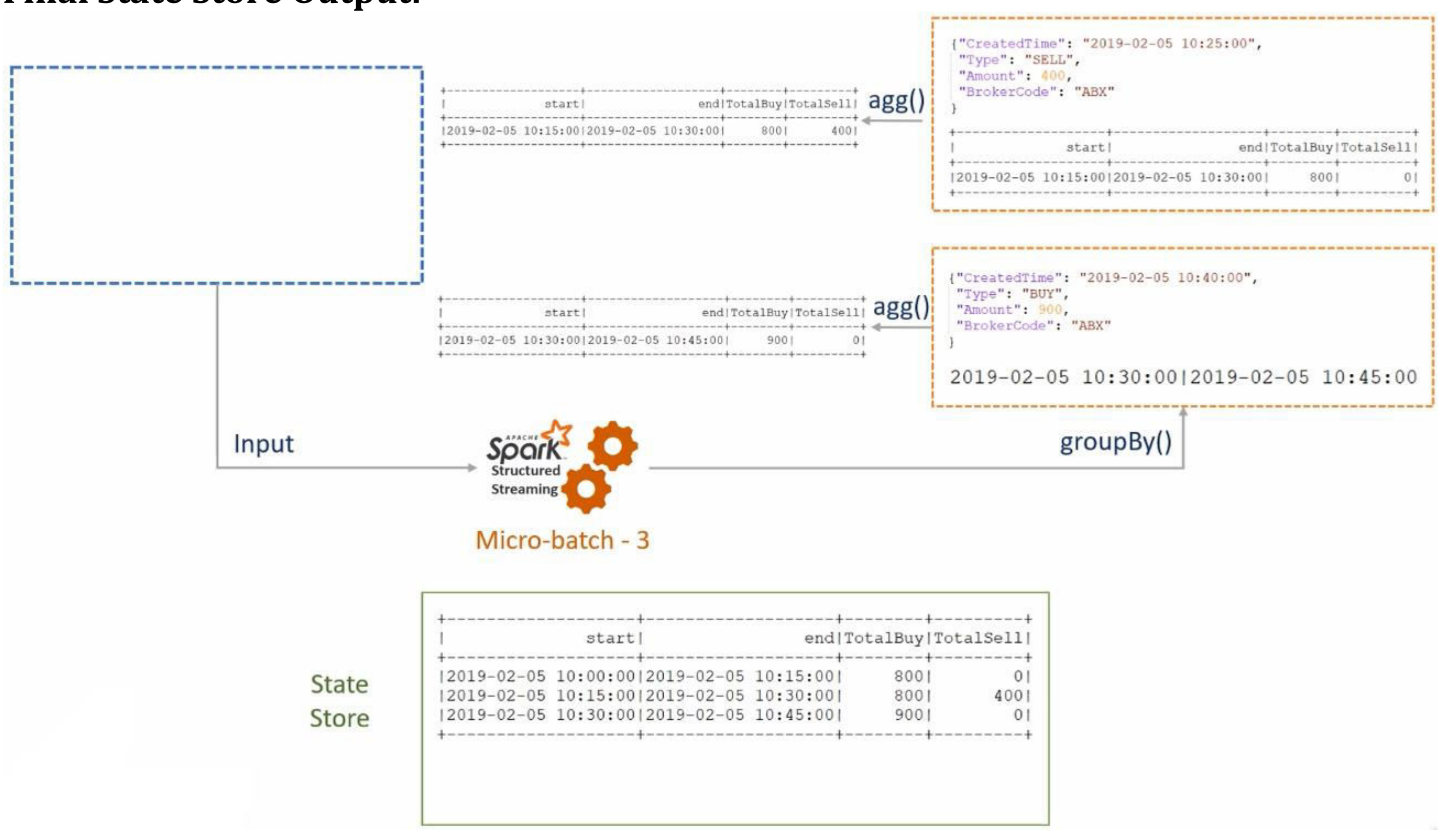
3rd Micro Batch:



3rd Micro Batch has one record for new window & one late arriving record so its state will be referenced from the state store & it will compute the aggregates.



Final State Store Output:



So, How to set an Expiry date?

Ans) Watermark

1) What is the maximum possible delay?
2) When late records are not relevant?

Watermark = 30 minutes

TumblingWindowDemo.py | samples.txt

```
{"CreatedTime": "2019-02-05 10:05:00", "Type": "BUY", "Amount": 500, "BrokerCode": "ABX"}
{"CreatedTime": "2019-02-05 10:40:00", "Type": "BUY", "Amount": 900, "BrokerCode": "ABX"}
{"CreatedTime": "2019-02-05 10:20:00", "Type": "BUY", "Amount": 800, "BrokerCode": "ABX"}
{"CreatedTime": "2019-02-05 10:48:00", "Type": "SELL", "Amount": 600, "BrokerCode": "ABX"}

{"CreatedTime": "2019-02-05 10:14:00", "Type": "SELL", "Amount": 300, "BrokerCode": "ABX"}
{"CreatedTime": "2019-02-05 10:16:00", "Type": "SELL", "Amount": 300, "BrokerCode": "ABX"}
```

Formula to calculate Watermark:

```
Max(Event Time) - Watermark = Watermark Boundary
```

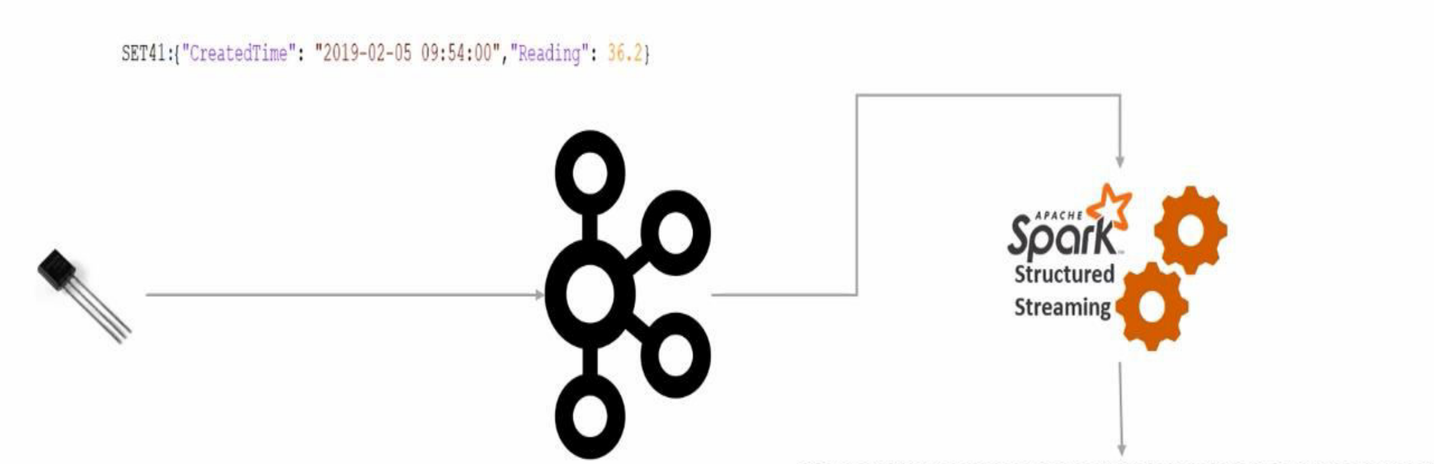
1. Watermark is the key for state store cleanup
2. Events within the watermark is taken
3. Event outside the watermark may or may not be taken

Note:

1) You should set watermark before the groupBy ( )
2) Event time column should be same on what you are doing windowing

9) Watermarking:

10) Sliding Window also called Hopping Window i.e. overlapping:



SensorID	start	end	MaxReading
SET41	2019-02-05 09:40:00	2019-02-05 09:55:00	36.2
SET41	2019-02-05 09:45:00	2019-02-05 10:00:00	36.5
SET41	2019-02-05 09:50:00	2019-02-05 10:05:00	36.8
SET41	2019-02-05 09:55:00	2019-02-05 10:10:00	36.8
SET41	2019-02-05 10:00:00	2019-02-05 10:15:00	36.8
SET41	2019-02-05 10:05:00	2019-02-05 10:20:00	36.5
SET41	2019-02-05 10:10:00	2019-02-05 10:25:00	37.7
SET41	2019-02-05 10:15:00	2019-02-05 10:30:00	37.7
SET41	2019-02-05 10:20:00	2019-02-05 10:35:00	37.7
SET41	2019-02-05 10:25:00	2019-02-05 10:40:00	37.3
SET41	2019-02-05 10:30:00	2019-02-05 10:45:00	37.3

SensorID	start	end	MaxReading
SET41	2019-02-05 09:40:00	2019-02-05 09:55:00	36.2
SET41	2019-02-05 09:45:00	2019-02-05 10:00:00	36.5
SET41	2019-02-05 09:50:00	2019-02-05 10:05:00	36.8
SET41	2019-02-05 09:55:00	2019-02-05 10:10:00	36.8
SET41	2019-02-05 10:00:00	2019-02-05 10:15:00	36.8
SET41	2019-02-05 10:05:00	2019-02-05 10:20:00	36.5
SET41	2019-02-05 10:10:00	2019-02-05 10:25:00	37.7
SET41	2019-02-05 10:15:00	2019-02-05 10:30:00	37.7
SET41	2019-02-05 10:20:00	2019-02-05 10:35:00	37.7
SET41	2019-02-05 10:25:00	2019-02-05 10:40:00	37.3
SET41	2019-02-05 10:30:00	2019-02-05 10:45:00	37.3