MONASH University

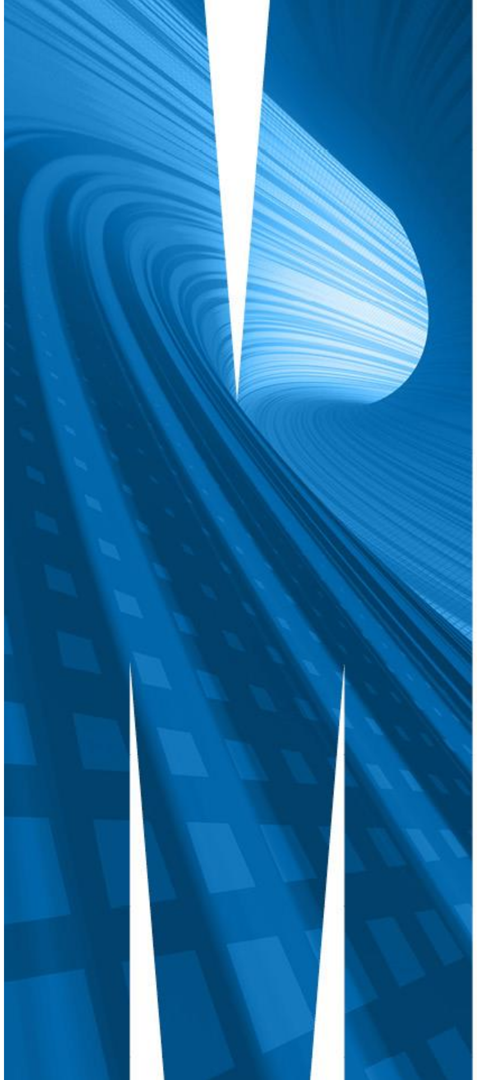
Week 10

FIT5202 Big Data Processing

Revised by Chee-Ming Ting
(13 May 2025)

Data Streaming using Apache Kafka and Spark

Spark Structured Streaming

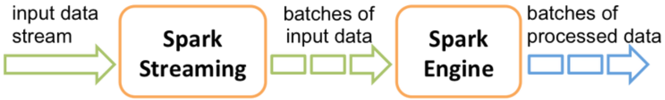


Week 10 Agenda

- Week 10 Review
  - Apache Kafka
    - Kafka Producer and Kafka Consumer
- Spark Structured Streaming
  - Introduction
  - Typical Use Case with Kafka
  - DEMO :
    - Word Count (Reading from Socket)
    - Click Stream Analysis (Producer and Consumer)
  - **Use case : Log Analysis**
- Other Topics on Structured Streaming
  - Output Modes
  - Output Sink
  - Triggers

Spark Streaming

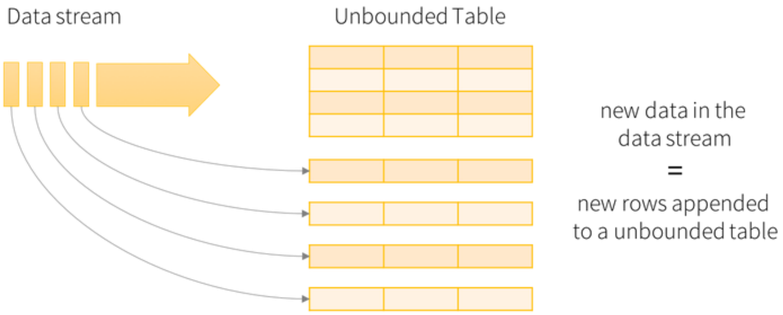
Spark Streaming : used the concept of microbatches, incoming record was a dstream, and RDD based API



Dstream (Discretized stream) - provide us data divided into chunks as RDDs

Resource : [Comparison between Spark Streaming and Structured Streaming]()

Structured Streaming : no concept of batch, data is received in a trigger, appended continuously to the unbounded result table.

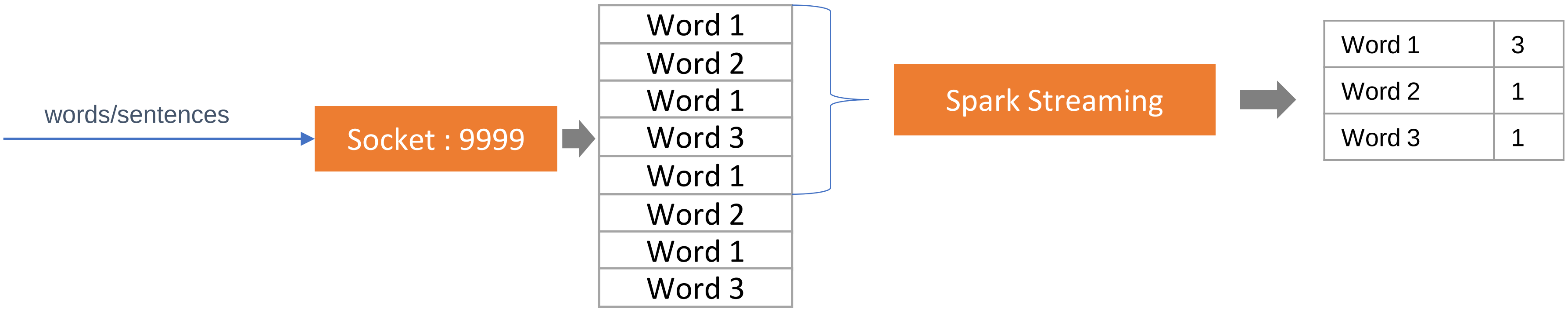

Data stream
Unbounded Table
new data in the data stream
=
new rows appended to a unbounded table

Data stream as an unbounded table

Treat a live data stream as a table that is being continuously appended.

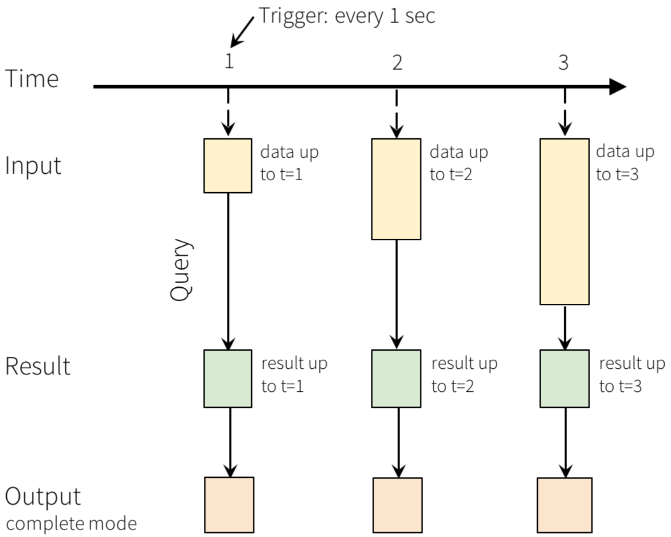
DEMO Spark Structured Streaming

Word Count Demo



Word 1	3
Word 2	1
Word 3	1

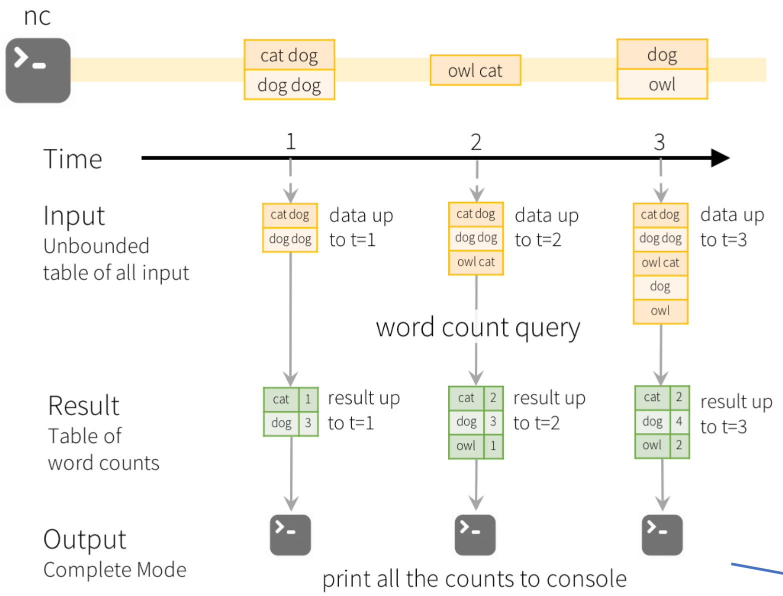
Spark Structured Streaming



Programming Model for Structured Streaming

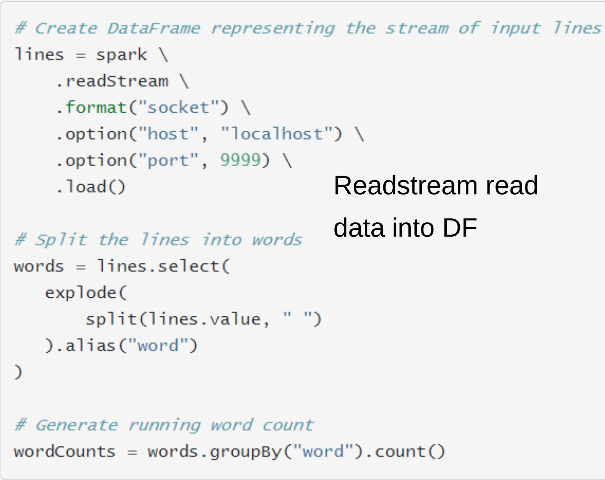
- A query on the input generates the “Result table”
- At every **trigger** interval (say, every 1 second), new rows are appended to the input table, which eventually updates the result table
- Whenever the result table is updated, the changed result rows are written to an external **sink**.
- The output is what is written to external storage, it has 3 modes
  1. Complete Mode
  2. Append Mode
  3. Update Mode

Structured Stream Example (Word Count)



Model of the Quick Example

lines streaming dataframe - an unbounded table containing streaming text data



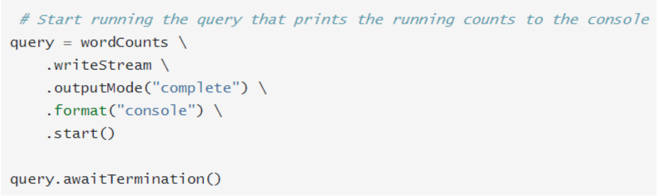
```
# Create DataFrame representing the stream of input lines
lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

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

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

Readstream read data into DF

wordCounts streaming DataFrame – is result table containing running word counts of the stream

When query is started

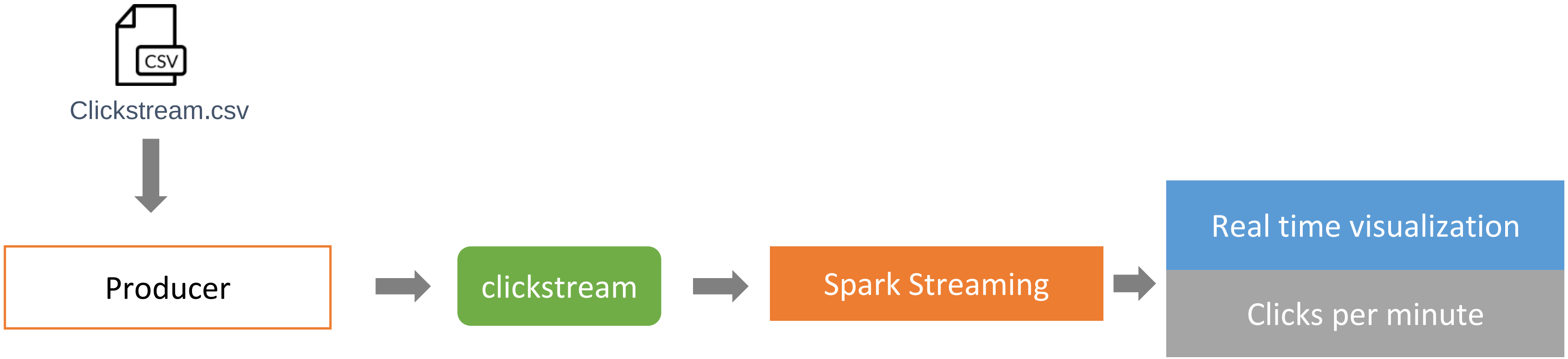
- Spark will continuously check for new data
- Spark will run incremental query to combine previous running counts with new data



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

query.awaitTermination()
```

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Demo for Spark Structured Streaming



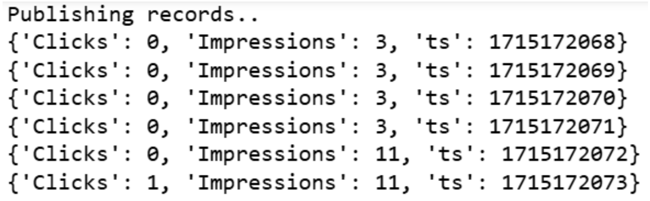
See Demo files (Week 10):

Clickstream-Producer DEMO (data sent row-by-row)

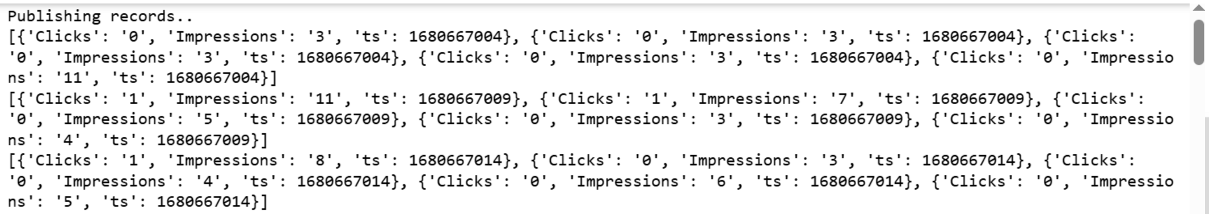
Spark Streaming - ClickStream-Analysis DEMO [V 1.1]

LT2-Producer (data sent batch-by-batch)

Clickstream Spark Streaming - Handling Json Array DEMO



```
Publishing records..
{'Clicks': 0, 'Impressions': 3, 'ts': 1715172068}
{'Clicks': 0, 'Impressions': 3, 'ts': 1715172069}
{'Clicks': 0, 'Impressions': 3, 'ts': 1715172070}
{'Clicks': 0, 'Impressions': 3, 'ts': 1715172071}
{'Clicks': 0, 'Impressions': 11, 'ts': 1715172072}
{'Clicks': 1, 'Impressions': 11, 'ts': 1715172073}
```

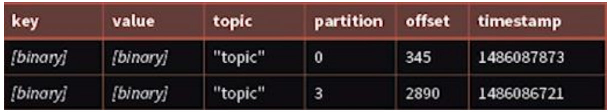


```
Publishing records..
[{'Clicks': '0', 'Impressions': '3', 'ts': 1680667004}, {'Clicks': '0', 'Impressions': '3', 'ts': 1680667004}, {'Clicks':
'0', 'Impressions': '3', 'ts': 1680667004}, {'Clicks': '0', 'Impressions': '3', 'ts': 1680667004}, {'Clicks': '0', 'Impressio
ns': '11', 'ts': 1680667004}]
[{'Clicks': '1', 'Impressions': '11', 'ts': 1680667009}, {'Clicks': '1', 'Impressions': '7', 'ts': 1680667009}, {'Clicks':
'0', 'Impressions': '5', 'ts': 1680667009}, {'Clicks': '0', 'Impressions': '3', 'ts': 1680667009}, {'Clicks': '0', 'Impressio
ns': '4', 'ts': 1680667009}]
[{'Clicks': '1', 'Impressions': '8', 'ts': 1680667014}, {'Clicks': '0', 'Impressions': '3', 'ts': 1680667014}, {'Clicks':
'0', 'Impressions': '4', 'ts': 1680667014}, {'Clicks': '0', 'Impressions': '6', 'ts': 1680667014}, {'Clicks': '0', 'Impressio
ns': '5', 'ts': 1680667014}]
```

Clickstream analysis

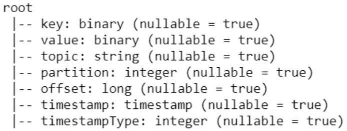
Kafka producer

```
{
  "Clicks":"0",
  "Impressions":"6",
  "ts":1602944650
}
```

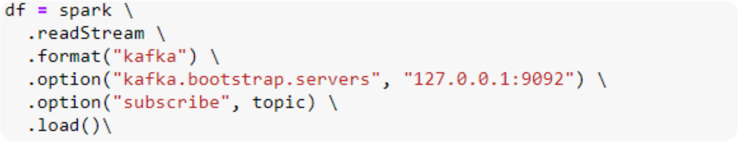


key	value	topic	partition	offset	timestamp
[binary]	*[binary]*	"topic"	0	345	1486087873
[binary]	*[binary]*	"topic"	3	2890	1486086721

Kafka format data
- Data in value are byte array

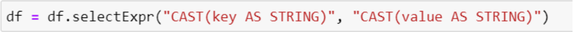


```
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```



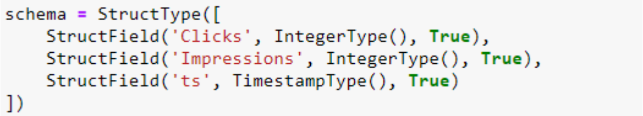
```
df = spark \
  .readStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "127.0.0.1:9092") \
  .option("subscribe", topic) \
  .load()\
```

Spark subscribe to topic & read data

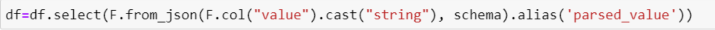


```
df = df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
```

Cast to String



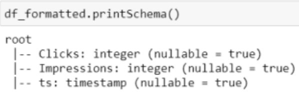
```
schema = StructType([
    StructField('Clicks', IntegerType(), True),
    StructField('Impressions', IntegerType(), True),
    StructField('ts', TimestampType(), True)
])
```



```
df=df.select(F.from_json(F.col("value").cast("string"), schema).alias('parsed_value'))
```

Using the schema, we convert the data to a Spark DataFrame

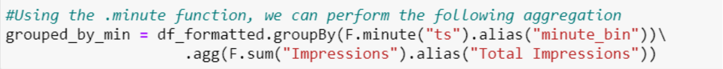
Parse the string data in json format according to the schema into StructType



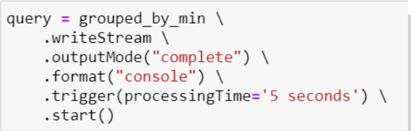
```
df_formatted.printSchema()

root
 |-- Clicks: integer (nullable = true)
 |-- Impressions: integer (nullable = true)
 |-- ts: timestamp (nullable = true)
```

Clicks	Impressions	ts
0	6	1602944650
0	10	1602944650
0	5	1602944650

Query on Streaming Dataframes



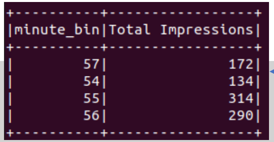
```
#Using the .minute function, we can perform the following aggregation
grouped_by_min = df_formatted.groupBy(F.minute("ts").alias("minute_bin"))\
                    .agg(F.sum("Impressions").alias("Total Impressions"))
```



```
query = grouped_by_min \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .trigger(processingTime='5 seconds') \
    .start()
```

For aggregation query, use ‘complete’ mode

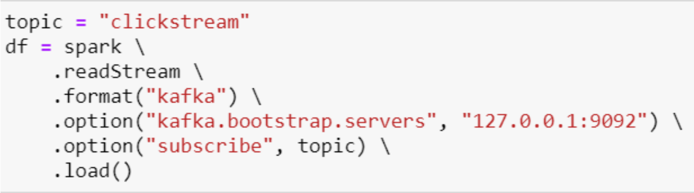
Output result table



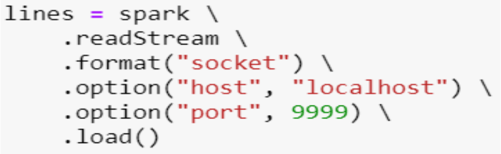
```
+----------+-----------------+
|minute_bin|Total Impressions|
+----------+-----------------+
|        57|              172|
|        54|              134|
|        55|              314|
|        56|              290|
+----------+-----------------+
```

Creating streaming DataFrames

Input Sources



```
topic = "clickstream"
df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "127.0.0.1:9092") \
    .option("subscribe", topic) \
    .load()
```



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

•**File source -** Reads files written in a directory as a stream of data. Files will be processed in the order of file modification time. If latestFirst is set, order will be reversed. Supported file formats are text, CSV, JSON, ORC, Parquet.

Kafka source - Reads data from Kafka. It's compatible with Kafka broker versions 0.10.0 or higher. See the Kafka Integration Guide for more details.

Socket source (for testing) - Reads UTF8 text data from a socket connection. The listening server socket is at the driver.

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#starting-streaming-queries

Output Modes

- ❑ The output mode specifies **the way the data in result table is written to output sink**

•**Append mode (default)** - Only the new rows added to the Result Table since the last trigger will be outputted to the sink. This is supported for only those queries where rows added to the Result Table is never going to change. Hence, this mode guarantees that each row will be output only once (assuming fault-tolerant sink). For example, queries with only select, where, map, flatMap, filter, join, etc. will support Append mode.

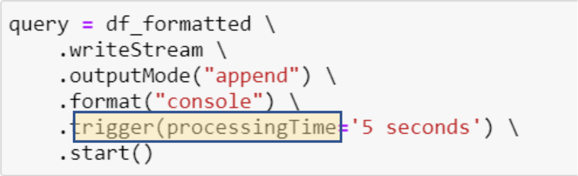
Just write new rows in result table to sink

•**Complete mode** - The whole Result Table will be outputted to the sink after every trigger. This is supported for aggregation queries (e.g., groupBy).

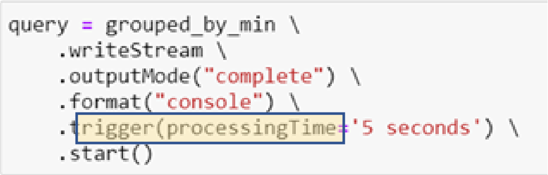
Write all the rows to sink

•**Update mode** - Only the rows in the Result Table that were updated since the last trigger will be outputted to the sink. When there are no aggregations it works exactly the same as “append” mode

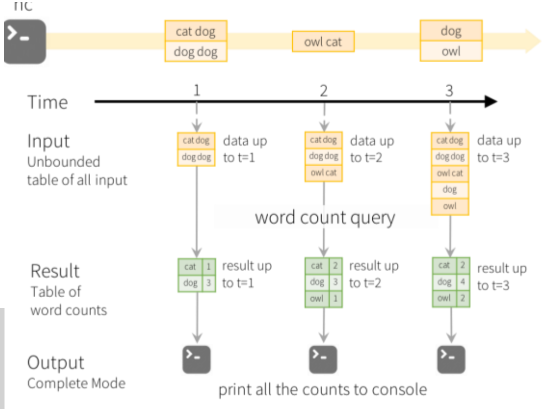
Write ‘only’ all the rows that are updated to sink



```
query = df_formatted \
    .writeStream \
    .outputMode("append") \
    .format("console") \
    .trigger(processingTime='5 seconds') \
    .start()
```

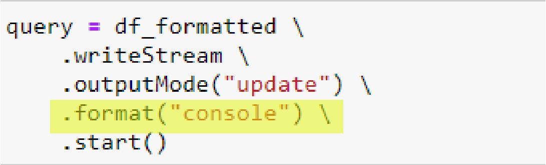


```
query = grouped_by_min \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .trigger(processingTime='5 seconds') \
    .start()
```

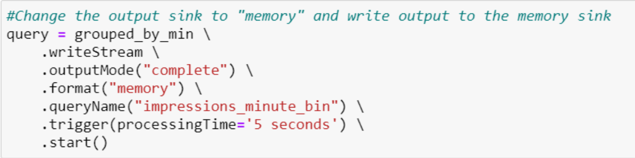


https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#starting-streaming-queries

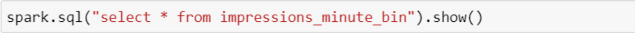
Output Sink



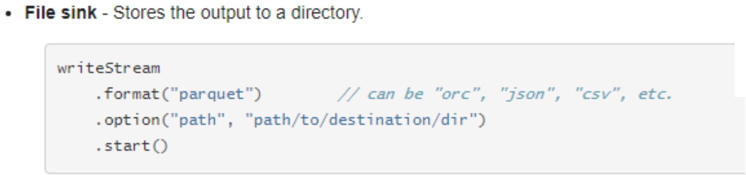
```
query = df_formatted \
    .writeStream \
    .outputMode("update") \
    .format("console") \
    .start()
```



```
#Change the output sink to "memory" and write output to the memory sink
query = grouped_by_min \
    .writeStream \
    .outputMode("complete") \
    .format("memory") \
    .queryName("impressions_minute_bin") \
    .trigger(processingTime='5 seconds') \
    .start()
```



```
spark.sql("select * from impressions_minute_bin").show()
```

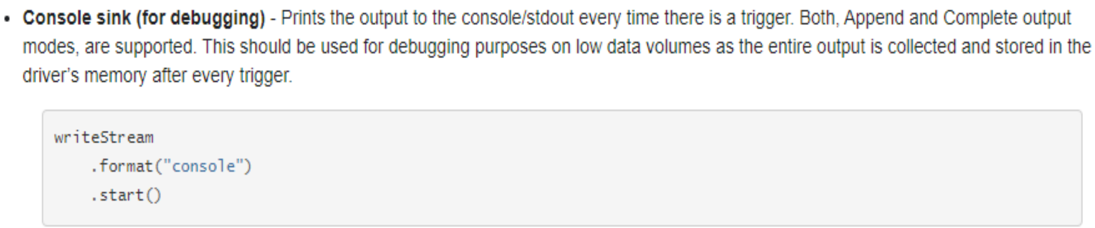


- **File sink** - Stores the output to a directory.

```
writeStream
    .format("parquet")        // can be "orc", "json", "csv", etc.
    .option("path", "path/to/destination/dir")
    .start()
```

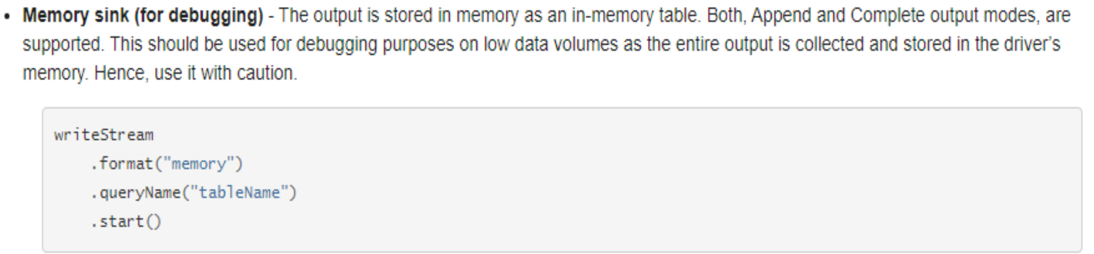
What is Parquet file format [Ref Link]?

Efficient as well as performant flat columnar storage format compared to row-based csv or tsv files



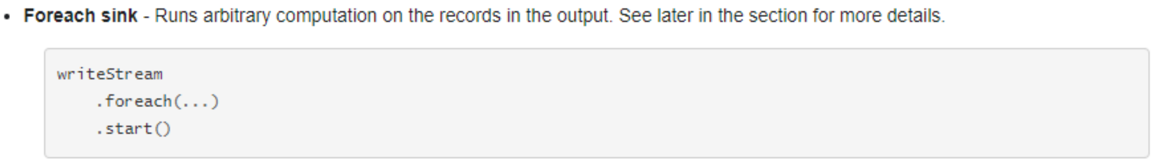
- **Console sink (for debugging)** - Prints the output to the console/stdout every time there is a trigger. Both, Append and Complete output modes, are supported. This should be used for debugging purposes on low data volumes as the entire output is collected and stored in the driver's memory after every trigger.

```
writeStream
    .format("console")
    .start()
```



- **Memory sink (for debugging)** - The output is stored in memory as an in-memory table. Both, Append and Complete output modes, are supported. This should be used for debugging purposes on low data volumes as the entire output is collected and stored in the driver's memory. Hence, use it with caution.

```
writeStream
    .format("memory")
    .queryName("tableName")
    .start()
```



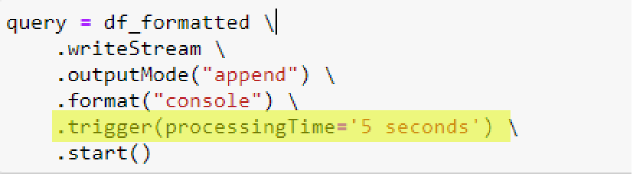
- **Foreach sink** - Runs arbitrary computation on the records in the output. See later in the section for more details.

```
writeStream
    .foreach(...)
    .start()
```

foreach operations allow you to apply arbitrary operations and writing logic on the output of a streaming query

Triggers — How Frequently to Check Sources For New Data

→ defines how often a streaming query should be executed (triggered) and emit the output



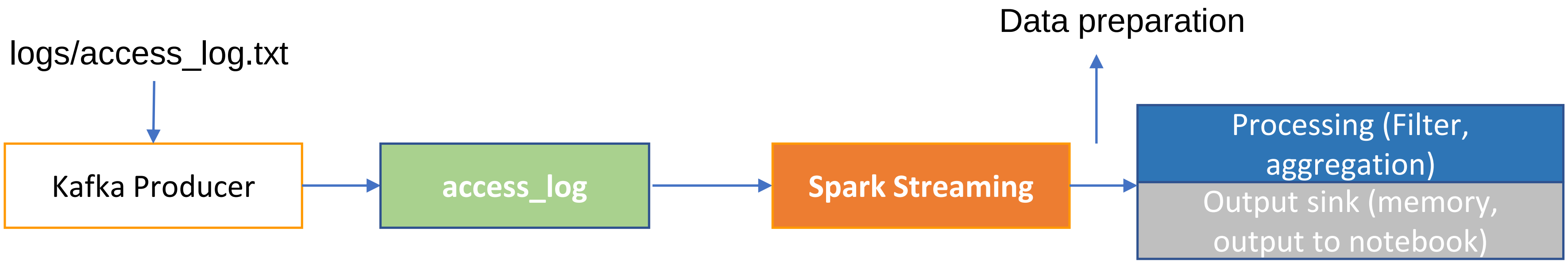
```
query = df_formatted \
    .writeStream \
    .outputMode("append") \
    .format("console") \
    .trigger(processingTime='5 seconds') \
    .start()
```

1. **Once** : only processes once and terminates the stream
2. **Processing time** : Most widely used and recommended
   - ❑ Processes datastreams as a series of small batch jobs
   - ❑ gives better control over how often micro batch jobs should get triggered (e.g. every 5 secs).
3. **Continuous** : Experimental Feature, allows to process records in milliseconds latency

https://dzone.com/articles/spark-trigger-options

Lab Task: Tracking Server Access Log

- ❑ Server is going to continuously send a records of a host who is trying to access some endpoint (url) from the web server
- ❑ Goal: Perform real time queries from this data stream and output the results.



logs/access_log.txt

Kafka Producer

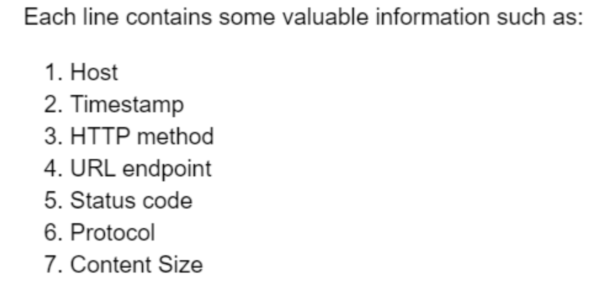
access_log

Spark Streaming

Data preparation

Processing (Filter, aggregation)

Output sink (memory, output to notebook)



Each line contains some valuable information such as:

1. Host
2. Timestamp
3. HTTP method
4. URL endpoint
5. Status code
6. Protocol
7. Content Size

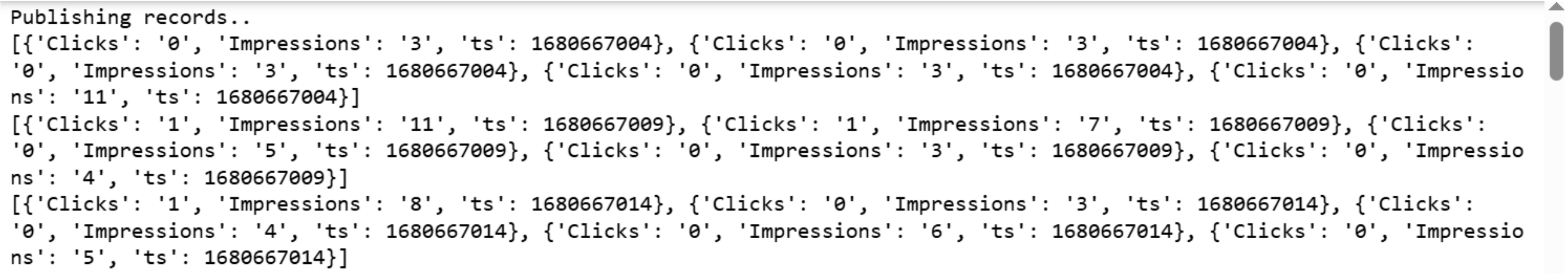
Task 2: filters those requests that were not successful (status != 200)
Task 3: Count the number of requests by access status code

Task 4: Output the unsucessful requests (unsucess_df) to memory sink

Handling Array Json

LT2-Producer.ipynb

Clickstream Spark Streaming - Handling Json Array DEMO



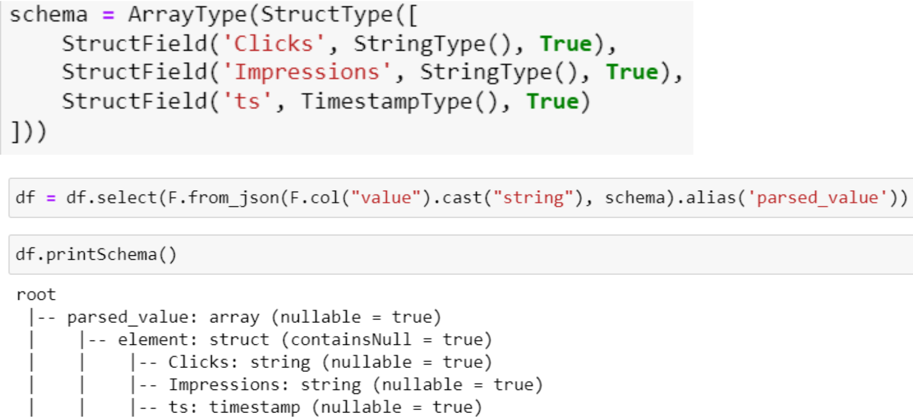
```
Publishing records..
[{'Clicks': '0', 'Impressions': '3', 'ts': 1680667004}, {'Clicks': '0', 'Impressions': '3', 'ts': 1680667004}, {'Clicks':
'0', 'Impressions': '3', 'ts': 1680667004}, {'Clicks': '0', 'Impressions': '3', 'ts': 1680667004}, {'Clicks': '0', 'Impressio
ns': '11', 'ts': 1680667004}]
[{'Clicks': '1', 'Impressions': '11', 'ts': 1680667009}, {'Clicks': '1', 'Impressions': '7', 'ts': 1680667009}, {'Clicks':
'0', 'Impressions': '5', 'ts': 1680667009}, {'Clicks': '0', 'Impressions': '3', 'ts': 1680667009}, {'Clicks': '0', 'Impressio
ns': '4', 'ts': 1680667009}]
[{'Clicks': '1', 'Impressions': '8', 'ts': 1680667014}, {'Clicks': '0', 'Impressions': '3', 'ts': 1680667014}, {'Clicks':
'0', 'Impressions': '4', 'ts': 1680667014}, {'Clicks': '0', 'Impressions': '6', 'ts': 1680667014}, {'Clicks': '0', 'Impressio
ns': '5', 'ts': 1680667014}]
```

Example batches of records sent by LT2-Producer – each data batch is a list of dictionary objects

Handling Array Json

1. LT2-Producer to generate and publish "Clicks" and "'Impressions" data in String type.

- ❑ from_json() function has a constraint when converting column value to a dataframe: Datatype defined in the schema should match with the value present in the json, if there is any column's mismatch value leads to null in all column values.
- ❑ The defined schema for "Clicks" & "Impressions" used in the from_jason should match the String type.
- ❑ These columns will be converted back to integer type later



```
schema = ArrayType(StructType([
    StructField('Clicks', StringType(), True),
    StructField('Impressions', StringType(), True),
    StructField('ts', TimestampType(), True)
]))
```

```
df = df.select(F.from_json(F.col("value").cast("string"), schema).alias('parsed_value'))
```

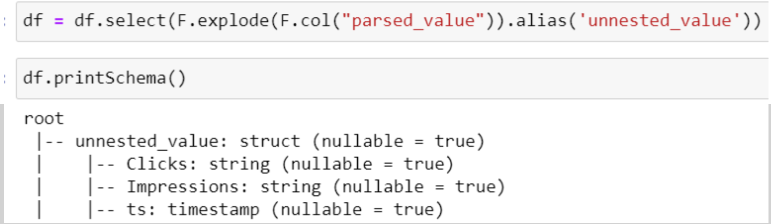
```
df.printSchema()
```

```
root
 |-- parsed_value: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- Clicks: string (nullable = true)
 |    |    |-- Impressions: string (nullable = true)
 |    |    |-- ts: timestamp (nullable = true)
```

2. LT2-Producer send data batch-by-batch in a list of dictionary objects

[{'Clicks': '0', 'Impressions': '3', 'ts': 1651720589}, {'Clicks': '0', 'Impressions': '3', 'ts': 1651720589}]

- ❑ The schema in from_jason uses ArrayType of StructType
- ❑ Use explode function to flatten the nested columns



```
df = df.select(F.explode(F.col("parsed_value")).alias('unnested_value'))
```

```
df.printSchema()
```

```
root
 |-- unnested_value: struct (nullable = true)
 |    |-- Clicks: string (nullable = true)
 |    |-- Impressions: string (nullable = true)
 |    |-- ts: timestamp (nullable = true)
```

References

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

https://docs.microsoft.com/en-us/azure/databricks/getting-started/spark/streaming

Thank You!

See you next week.