Structured Streaming and Continuous Processing in Apache Spark

Ramin Orujov 19.05.2018

Big Data Day Baku 2018 #BDDB2018

About me

Software Developer @ FHN 2008-2009

Azercell Telecom 2009-2016

- Software developer 2009-2012
- Software dev team lead 2012-2014
- Datawarehouse unit head 2014 – 2016

Big Data Engineer @ Luxoft Poland 2017 -

Agenda

Stream Processing Challenges

Structured Streaming in Apache Spark

Programming Model

Output Modes
Handling Late Data

Fault Tolerance

Agenda

Stream Deduplication

Operations on streaming

Triggers

Continuous Processing

Stream Processing Challenges

Different data formats (json, xml, avro, parquet, binary)

Data can be dirty, late and out of order

Programming complexity

Complex Use Cases - combining streaming with interactive queries, machine learning, etc

Different storage systems (HDFS, Kafka, NoSQL, RDBMS, S3, Kinesis, ...)

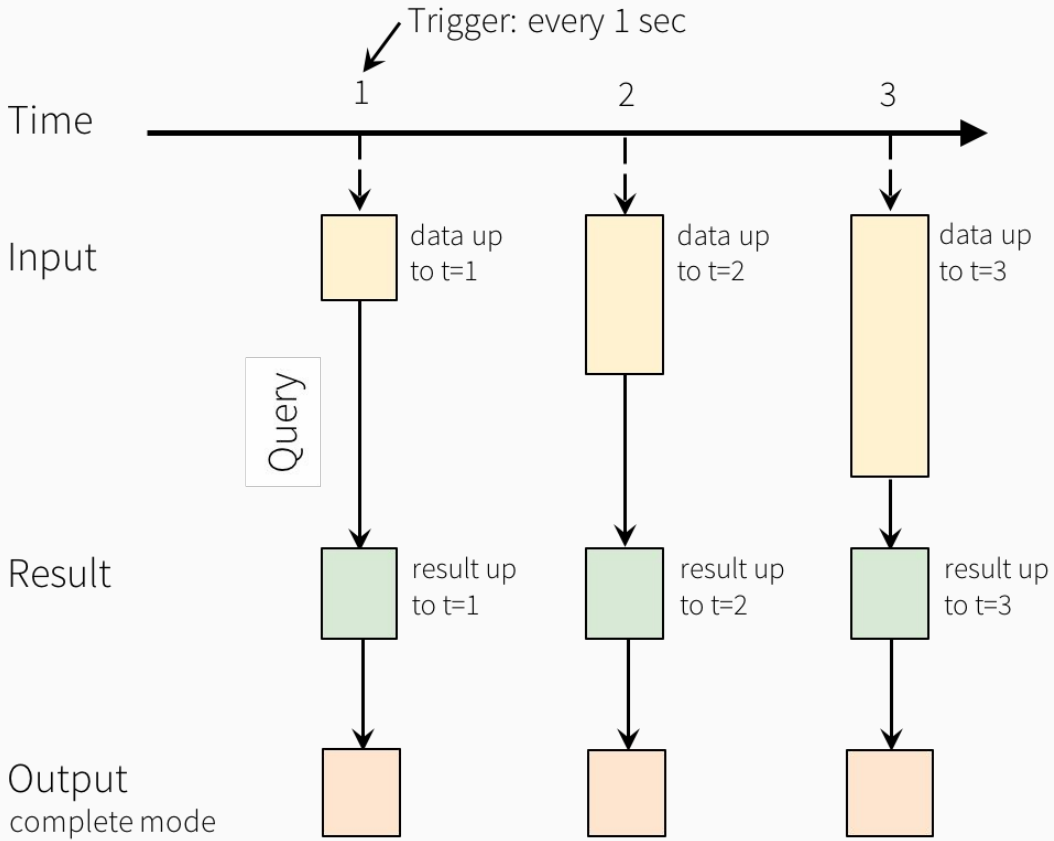
System failures and restarts

Structured Streaming

Stream processing on Spark SQL engine

Rich, unified and High level APIs

Rich Ecosystem of data sources



Programming Model for Structured Streaming

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

spark = SparkSession \
    .builder \
    .appName("StructuredNetworkWordCount") \
    .getOrCreate()
```

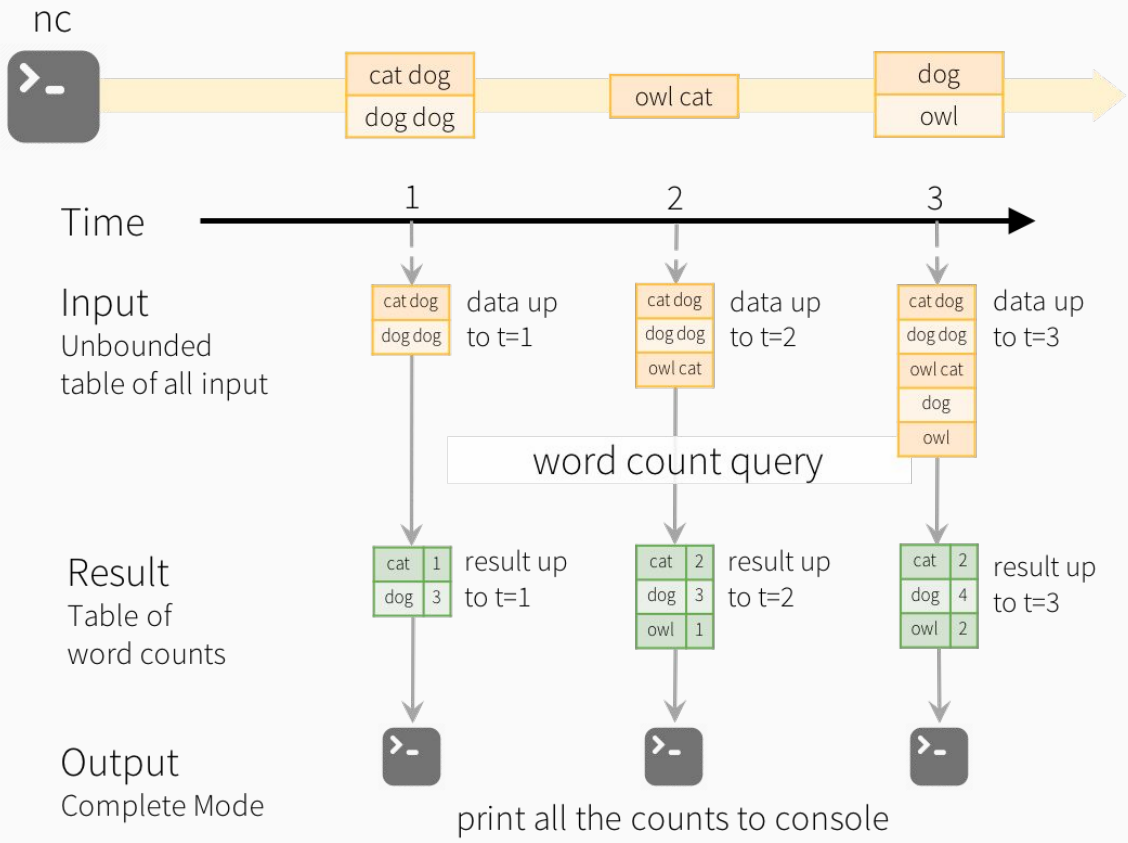
```
# Create DataFrame representing the stream of input lines from connection to localhost:9999
lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

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

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


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

query.awaitTermination()
```



Model of the Quick Example

Output Modes

Append mode (default) - only the new rows added to the Result Table since the last trigger will be outputted to the sink.

Complete mode - The whole Result Table will be outputted to the sink after every trigger. This is supported for aggregation queries.

Update mode - (Spark 2.1.1) Only the rows in the Result Table that were updated since the last trigger will be outputted to the sink.

Fault Tolerance

Checkpointing

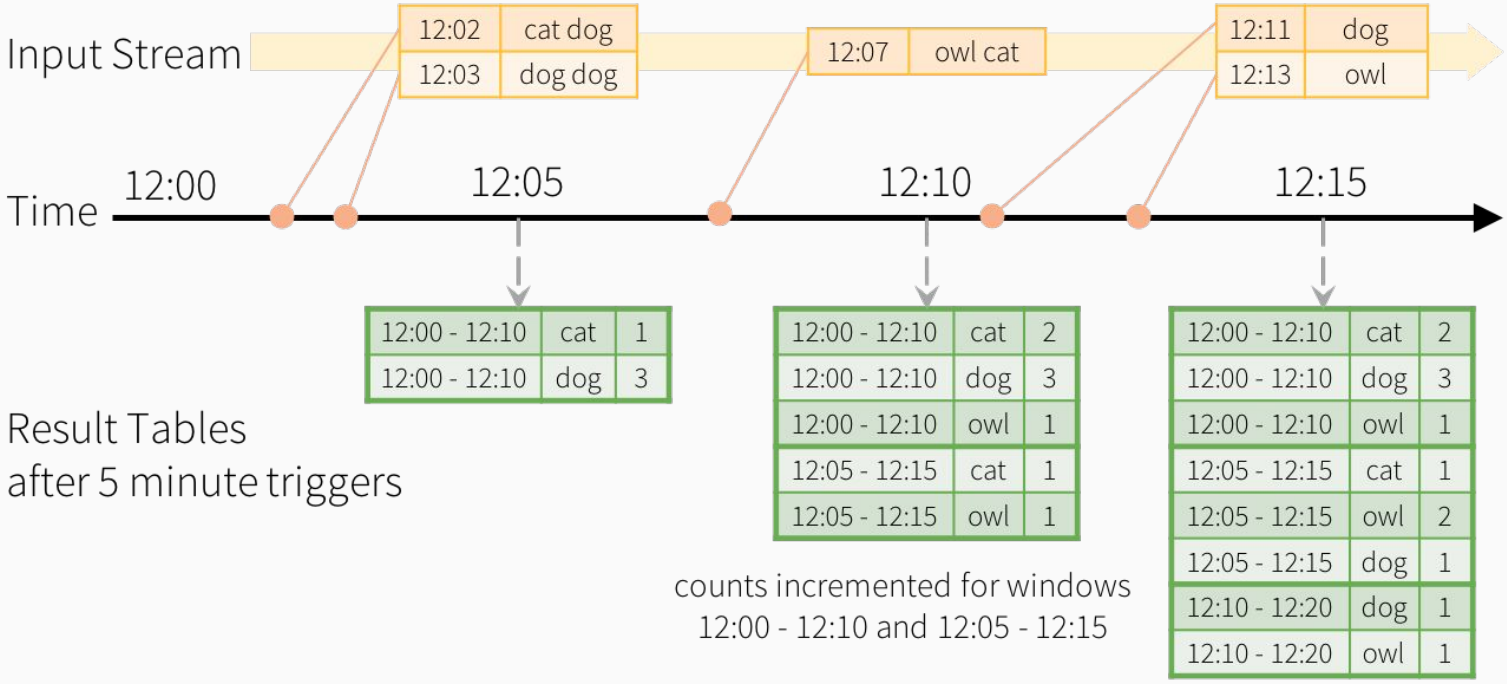
Write ahead logs - WAL

```
aggDF \
    .writeStream \
    .outputMode("complete") \
    .option("checkpointLocation", "path/to/HDFS/dir") \
    .format("memory") \
    .start()
```

Aggregations in Windows

```
words = ...  # streaming DataFrame of schema { timestamp: Timestamp, word: String }


# Group the data by window and word and compute the count of each group
windowedCounts = words.groupBy(
    window(words.timestamp, "10 minutes", "5 minutes"),
    words.word
).count()
```



Result Tables
after 5 minute triggers

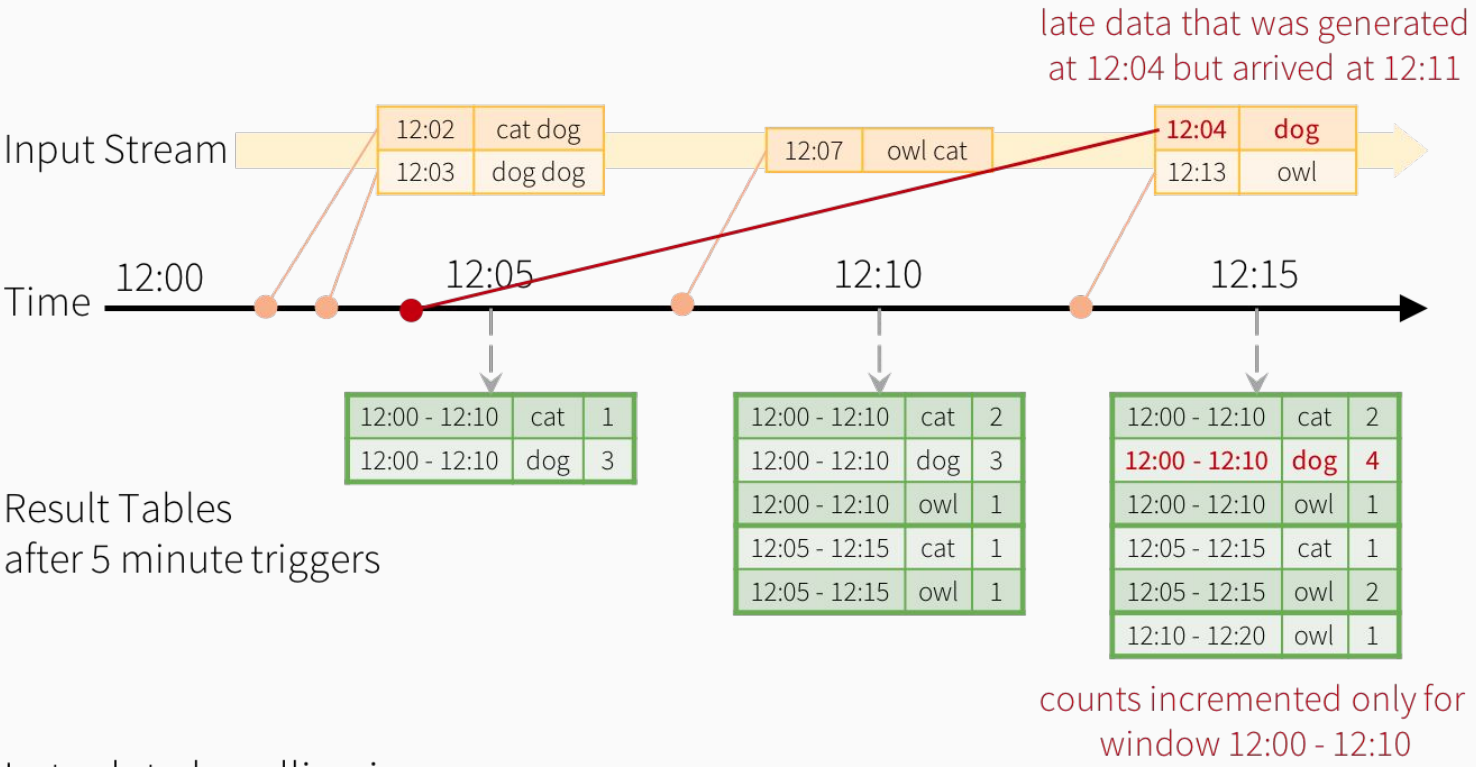
counts incremented for windows
12:00 - 12:10 and 12:05 - 12:15

counts incremented for windows
12:05 - 12:15 and 12:10 - 12:20

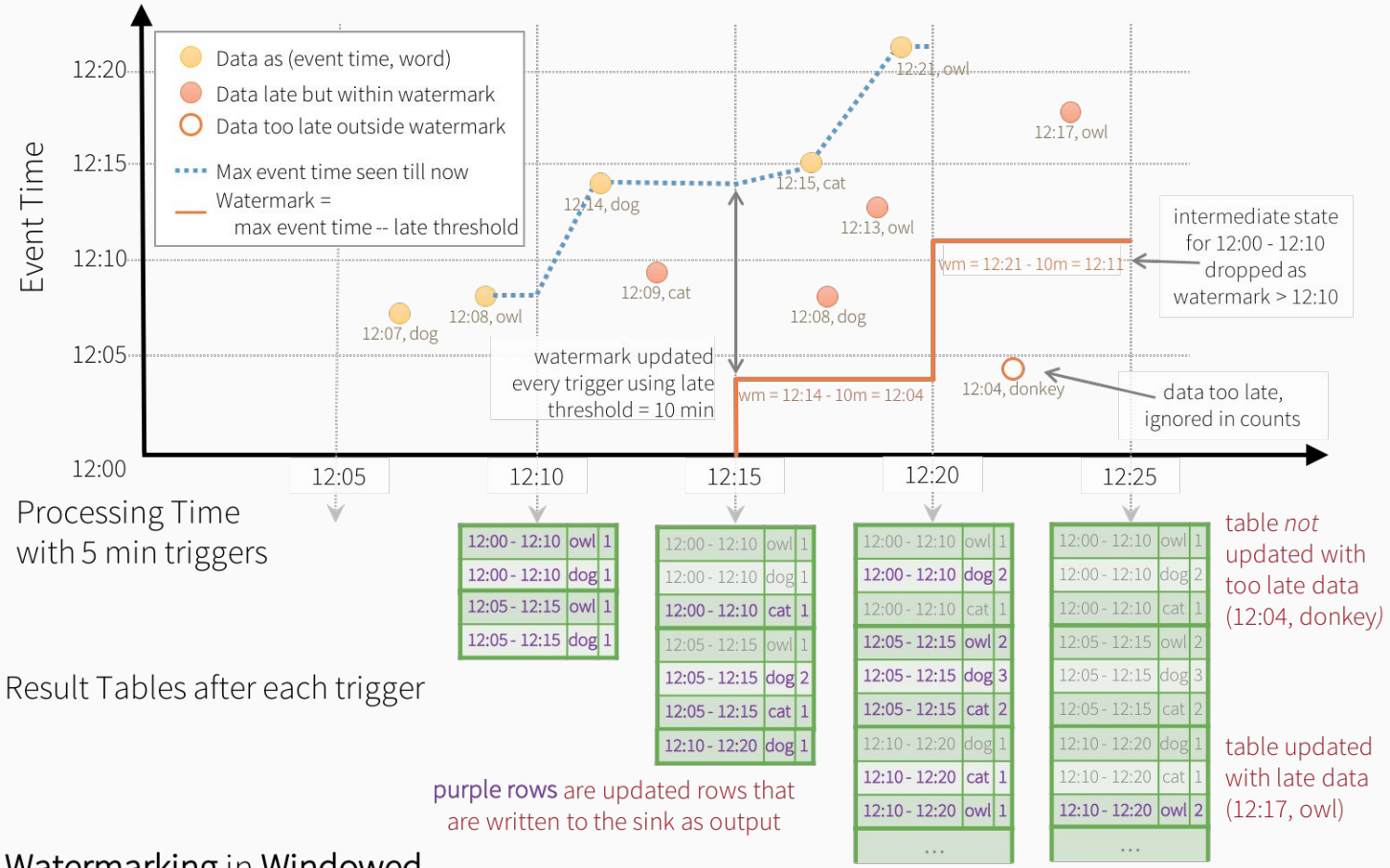
Windowed Grouped Aggregation
with 10 min windows, sliding every 5 mins

Handling Late Data and Watermarking

```
# Group the data by window and word and compute the count of each group
windowedCounts = words \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(
        window(words.timestamp, "10 minutes", "5 minutes"),
        words.word) \
    .count()
```



Late data handling in Windowed Grouped Aggregation



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

Stream Deduplication

```
streamingDf = spark.readStream. ...

# Without watermark using guid column
streamingDf.dropDuplicates("guid")

# With watermark using guid and eventTime columns
streamingDf \
  .withWatermark("eventTime", "10 seconds") \
  .dropDuplicates("guid", "eventTime")
```

Stream-static joins

With Spark 2.0, Structured Streaming has supported joins (inner join and some type of outer joins) between a streaming and a static DataFrame/Dataset.

```
staticDf = spark.read. ...
streamingDf = spark.readStream. ...
streamingDf.join(staticDf, "type")  # inner equi-join with a static DF
streamingDf.join(staticDf, "type", "right_join")  # right outer join with a static DF
```

Stream-stream joins

Spark 2.3 added support for stream-stream joins, that is, you can join two streaming Datasets/DataFrames. The challenge of generating join results between two data streams is that, at any point of time, the view of the dataset is incomplete for both sides of the join making it much harder to find matches between inputs. Any row received from one input stream can match with any future, yet-to-be-received row from the other input stream.

Stream-stream joins

```
impressions = spark.readStream. ...
clicks = spark.readStream. ...

# Apply watermarks on event-time columns
impressionsWithWatermark = impressions.withWatermark("impressionTime", "2 hours")
clicksWithWatermark = clicks.withWatermark("clickTime", "3 hours")
```

Stream-stream joins

```
# Join with event-time constraints
impressionsWithWatermark.join(
  clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime >= impressionTime AND
    clickTime <= impressionTime + interval 1 hour
    """)
)
```

Advanced Stateful Operations

Many usecases require more advanced stateful operations than aggregations. For example, you have to track sessions from data streams of events. For doing such sessionization, you will have to save arbitrary types of data as state, and perform arbitrary operations on the state using the data stream events in every trigger. Since Spark 2.2, this can be done using the operation mapGroupsWithState and the more powerful operation flatMapGroupsWithState. Both operations allow to apply user-defined code on grouped Datasets to update user-defined state.

Triggers

unspecified (default micro-batch mode)

Fixed interval micro-batches

One-time micro-batch

Continuous with fixed checkpoint interval

Triggers

```
# Default trigger (runs micro-batch as soon as it can)
df.writeStream \
  .format("console") \
  .start()

# ProcessingTime trigger with two-seconds micro-batch interval
df.writeStream \
  .format("console") \
  .trigger(processingTime='2 seconds') \
  .start()
```

Triggers

```
# One-time trigger
df.writeStream \
  .format("console") \
  .trigger(once=True) \
  .start()

# Continuous trigger with one-second checkpointing interval
df.writeStream
  .format("console")
  .trigger(continuous='1 second')
  .start()
```

Continuous processing

Default micro-batch processing engine which can achieve exactly-once guarantees but achieve latencies of ~100ms at best.

Continuous processing is a new, experimental streaming execution mode introduced in Spark 2.3 that enables low (~1 ms) end-to-end latency with at-least-once fault-tolerance guarantees.

Continuous processing

```
spark \
  .readStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2") \
  .option("subscribe", "topic1") \
  .load() \
  .selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)") \
  .writeStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2") \
  .option("topic", "topic1") \
  .trigger(continuous="1 second") \     # only change in query
  .start()
```

Continuous processing limitations

Only map-like Dataset/DataFrame operations are supported

All SQL functions are supported except aggregation functions

Non deterministic functions not supported - current_timestamp() and current_date()

Continuous processing limitations

Continuous processing engine launches multiple long-running tasks that continuously read data from sources, process it and continuously write to sinks. The number of tasks required by the query depends on how many partitions the query can read from the sources in parallel. Therefore, before starting a continuous processing query, you must ensure there are enough cores in the cluster to all the tasks in parallel.

There are currently no automatic retries of failed tasks. Any failure will lead to the query being stopped and it needs to be manually restarted from the checkpoint.

References

1.Structured Streaming Programming Guide

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

2.Structured Streaming In Apache Spark

https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html

3.Apache Spark 2.0: A Deep Dive Into Structured Streaming

https://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming

References

4.Easy, scalable, fault tolerant stream processing with structured streaming

https://www.slideshare.net/databricks/easy-scalable-fault-tolerant-stream-processing-with-structured-streaming-with-tathagata-das

5.Deep dive into stateful stream processing in structured streaming

https://www.slideshare.net/databricks/deep-dive-into-stateful-stream-processing-in-structured-streaming-by-tathagata-das

6.Continuous Applications: Evolving Streaming in Apache Spark 2.0

https://databricks.com/blog/2016/07/28/continuous-applications-evolving-streaming-in-

Streaming is the future of big data

Thanks!

Contact info

https://www.linkedin.com/in/raminorujov

https://www.facebook.com/ramin.orucov

raminorujov@gmail.com

+48 730 063 160

+994 50 231 01 09