Writing Continuous Applications with Structured Streaming in PySpark

Jules S. Damji

Spark + AI Summit , SF
April 24, 2019

databricks

I have used Apache Spark 2.x Before…

Apache Spark Community & Developer Advocate @ Databricks

Developer Advocate @ Hortonworks

Software engineering @ Sun Microsystems, Netscape, @Home, VeriSign, Scalix, Centrify, LoudCloud/Opsware, ProQuest

Program Chair Spark + AI Summit

https://www.linkedin.com/in/dmatrix
@2twitme

VISION

Accelerate innovation by unifying data science, engineering and business

---

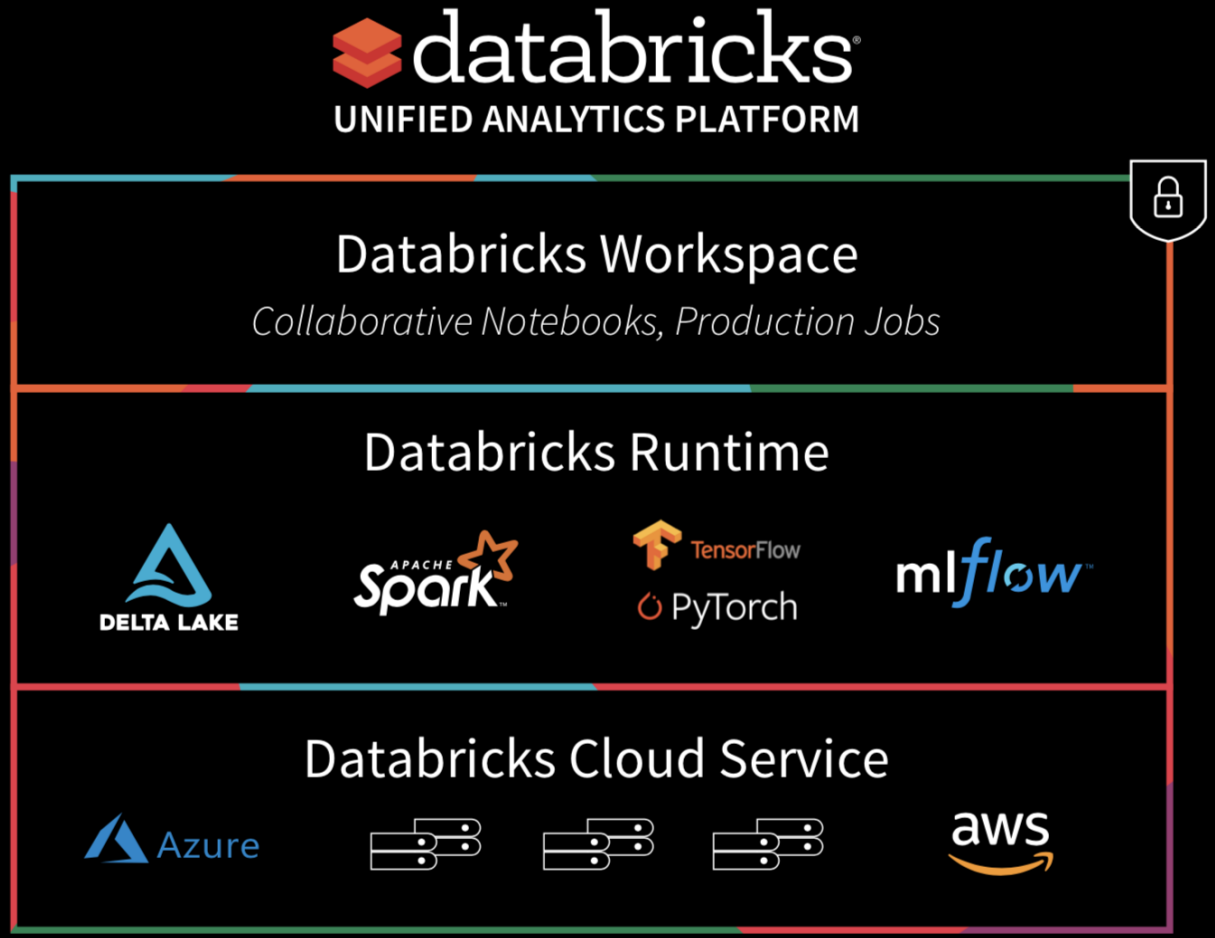
SOLUTION

Unified Analytics Platform

WHO WE ARE

- Original creators of 

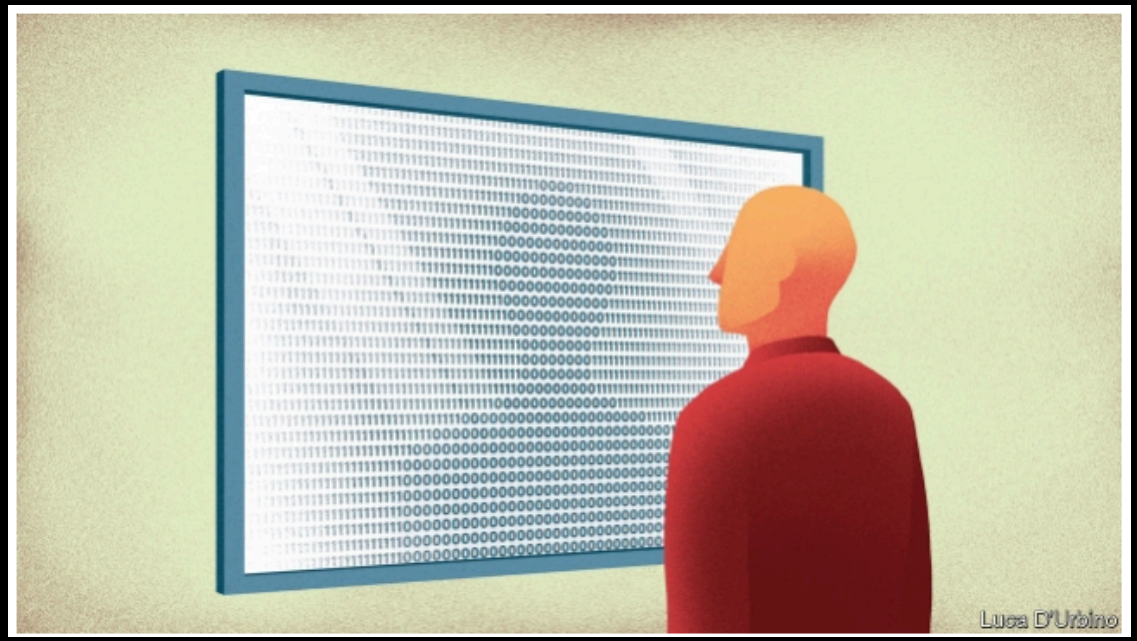
- 2000+ global companies use our platform across big data & machine learning lifecycle


databricks
UNIFIED ANALYTICS PLATFORM
Databricks Workspace
Collaborative Notebooks, Production Jobs
Databricks Runtime
DELTA LAKE
APACHE Spark
TensorFlow
PyTorch
mlflow
Databricks Cloud Service
Azure
aws

Agenda for Today's Talk

- Why Apache Spark
- Why Streaming Applications are Difficult
- What's Structured Streaming
  - Anatomy of a Continunous Application
  - Tutorials
- Q & A

How to think about data in 2019 - 2020



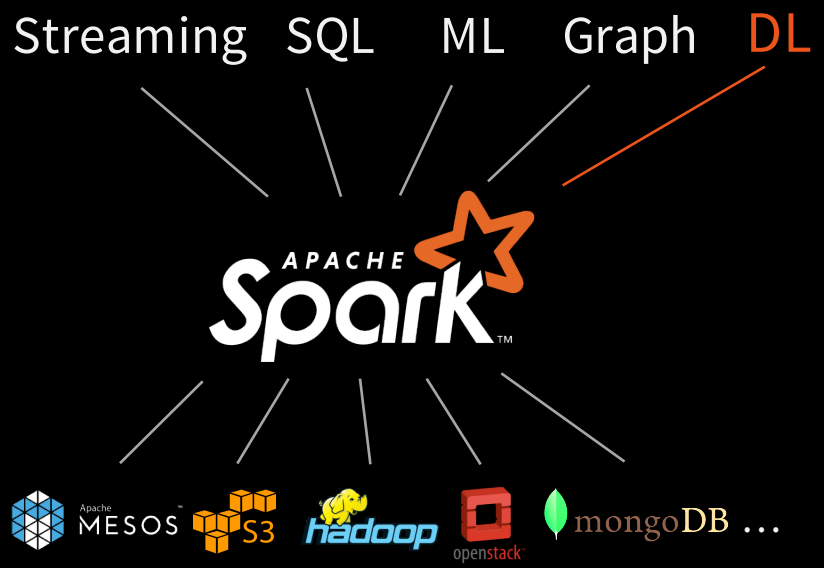
“Data is the new currency"

Why Apache Spark?

databricks

What is Apache Spark?

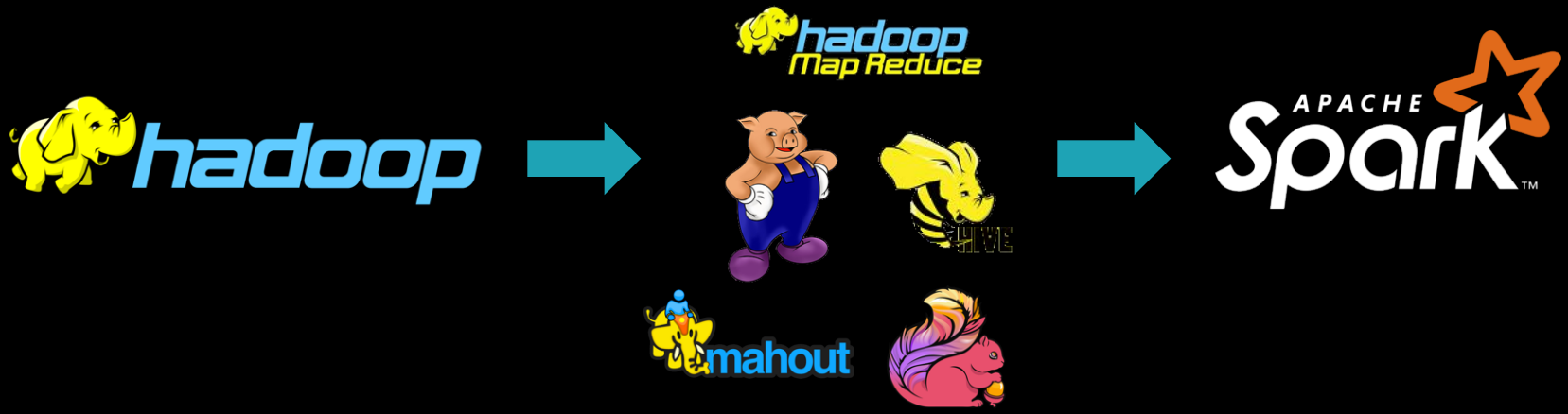
- General cluster computing engine that extends MapReduce
- Rich set of APIs and libraries
- Unified Engine
- Large community: 1000+ orgs, clusters up to 8000 nodes
- Supports DL Frameworks



Unique Thing about Spark

- Unification: same engine and same API for diverse use cases
  - Streaming, batch, or interactive
  - ETL, SQL, machine learning, or graph
  - Deep Learning Frameworks w/Horovod
    - TensorFlow
    - Keras
    - PyTorch

Faster, Easier to Use, Unified



First Distributed Processing Engine

Specialized Data Processing Engines

Unified Data Processing Engine

Benefits of Unification

1. Simpler to use and operate

2. Code reuse: e.g. only write monitoring, FT, etc once

3. New apps that span processing types: e.g. interactive queries on a stream, online machine learning

An Analogy

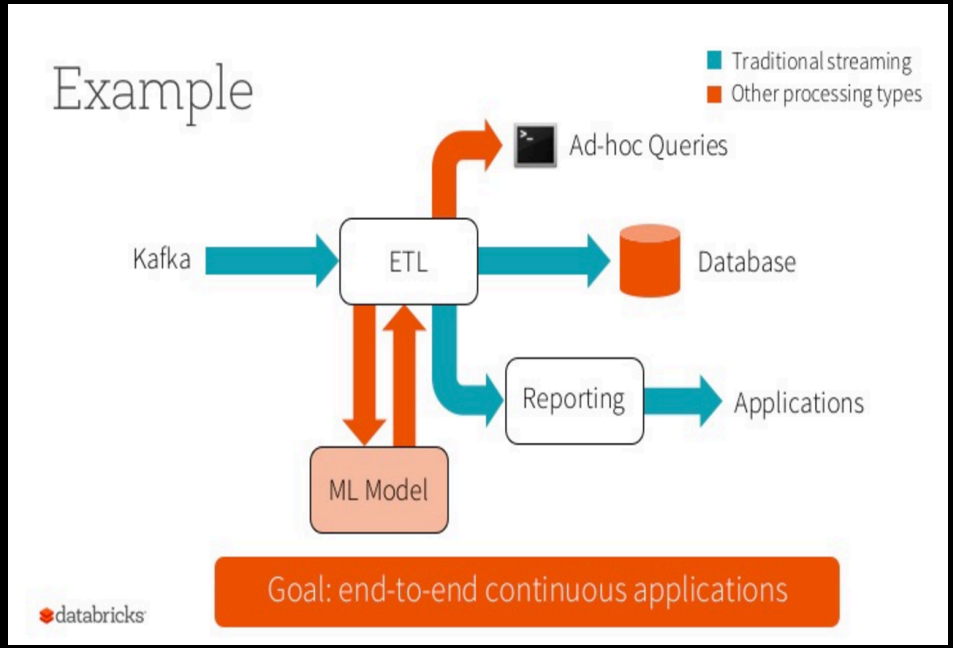
New applications

Specialized devices

Unified device

Why Streaming Applications are Inherently Difficult?

building robust stream processing apps is hard



Complexities in stream processing

COMPLEX DATA

Diverse data formats
(json, avro, txt, csv, binary, …)

Data can be dirty,
And tardy (out-of-order)

COMPLEX WORKLOADS

Combining streaming with interactive queries

Machine learning

COMPLEX SYSTEMS

Diverse storage systems
(Kafka, S3, Kinesis, RDBMS, …)

System failures

Structured Streaming

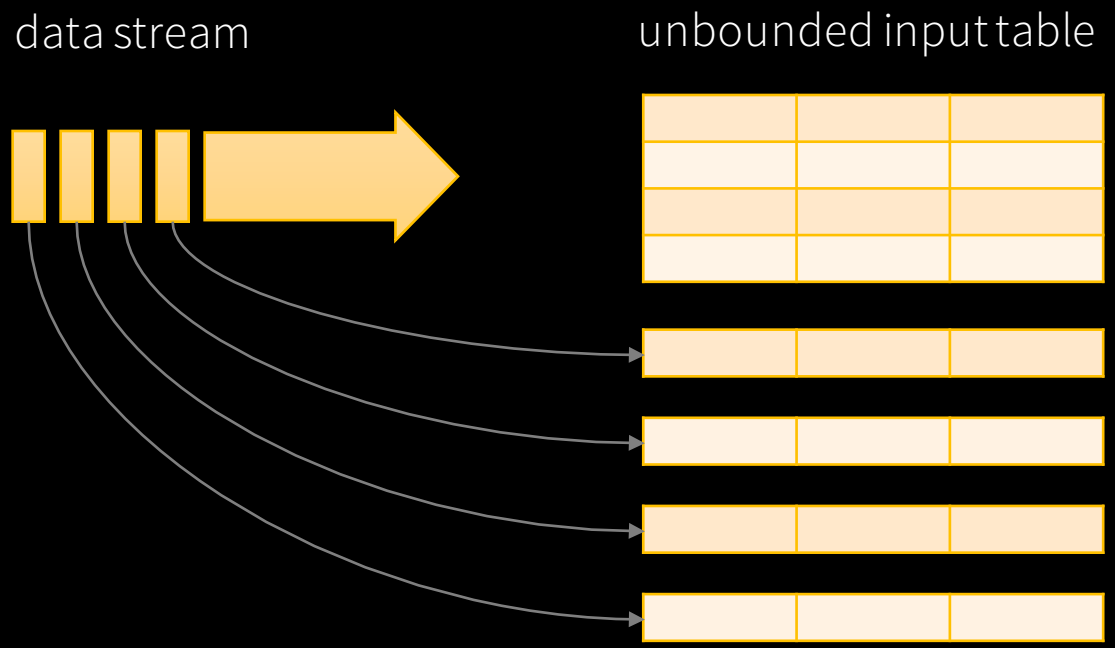
stream processing on Spark SQL engine
fast, scalable, fault-tolerant

rich, unified, high level APIs
deal with *complex data* and *complex workloads*

rich ecosystem of data sources
integrate with many *storage systems*

you
should not have to
reason about streaming

Treat Streams as Unbounded Tables



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

you

should write queries

&

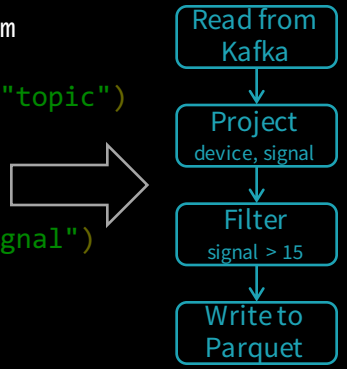
Apache Spark

should continuously update the answer

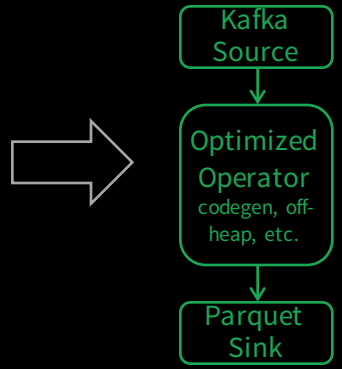
Apache Spark automatically *streamifies*!

```
input = spark.readStream
  .format("kafka")
  .option("subscribe", "topic")
  .load()

result = input
  .select("device", "signal")
  .where("signal > 15")

result.writeStream
  .format("parquet")
  .start("dest-path")
```

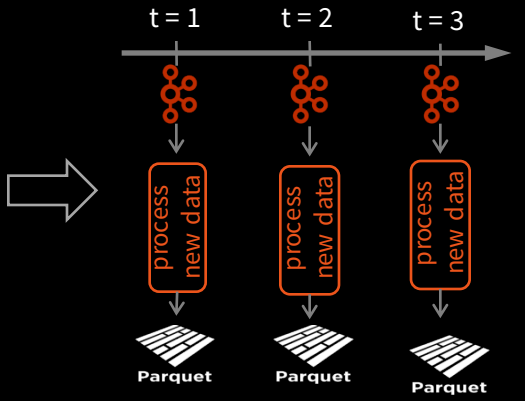
DataFrames, Datasets, SQL



Logical Plan



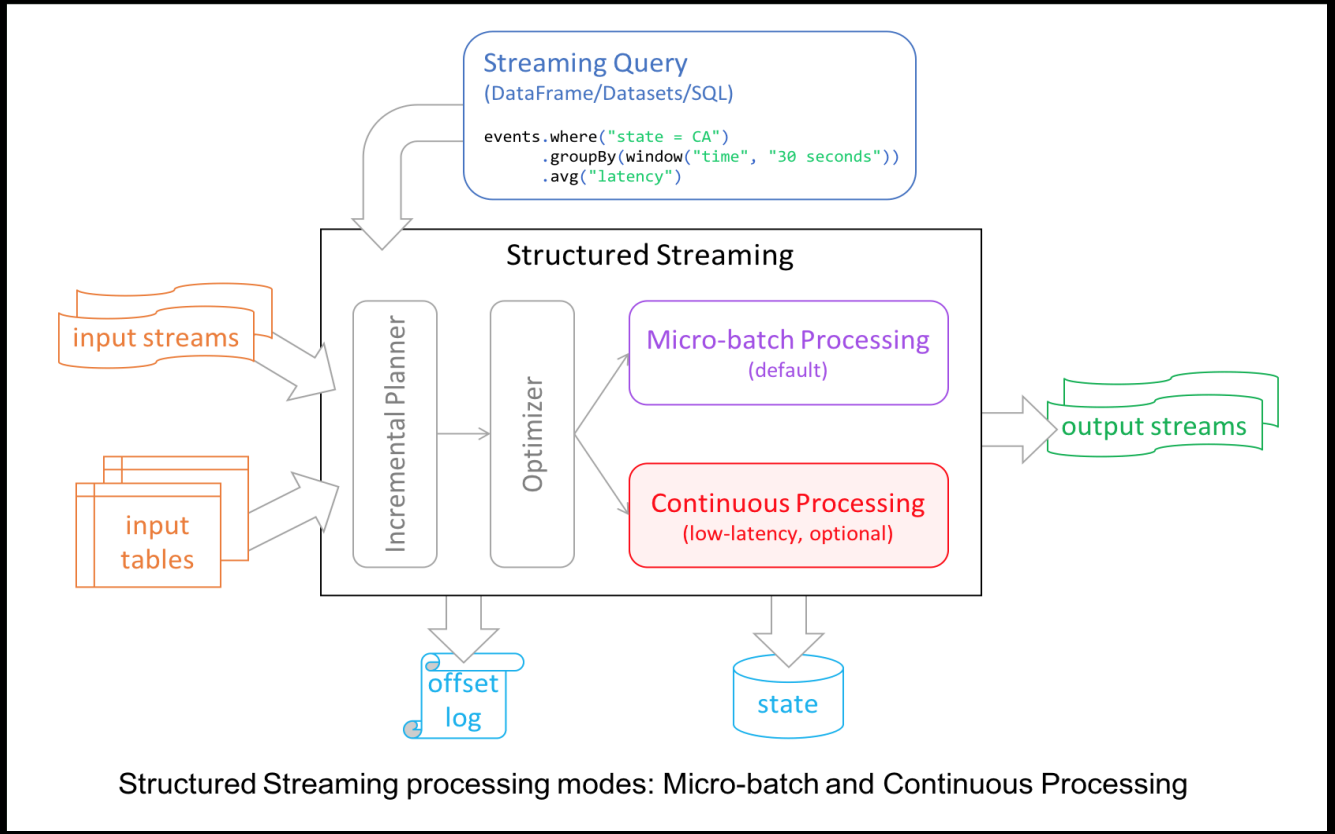
Optimized Physical Plan



Series of Incremental Execution Plans

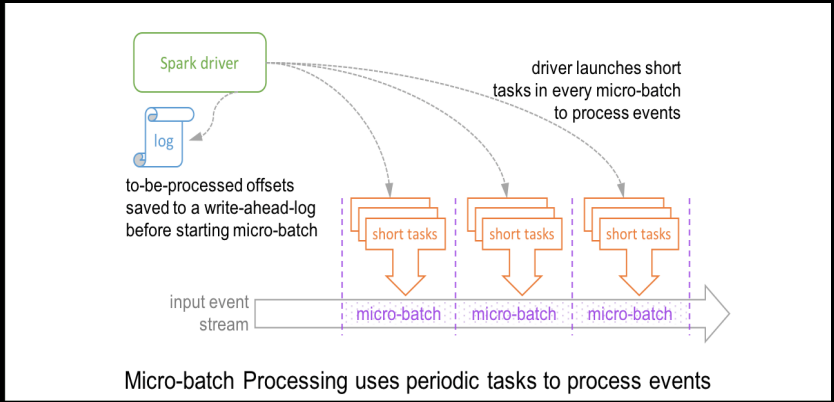
Spark SQL converts batch-like query to a series of incremental execution plans operating on new batches of data

Structured Streaming – Processing Modes

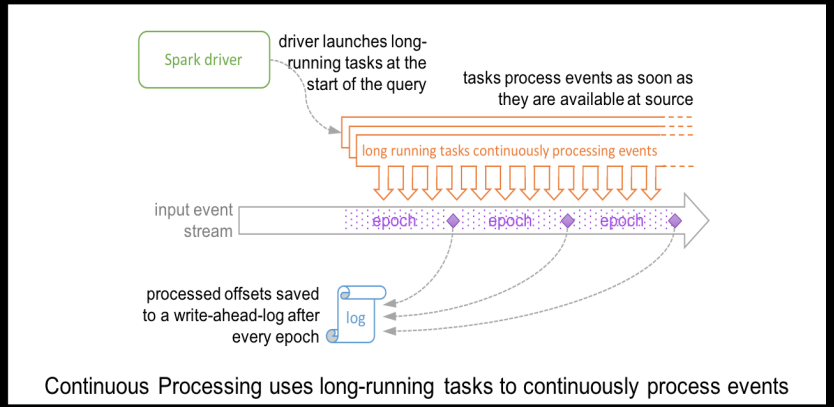


Structured Streaming processing modes: Micro-batch and Continuous Processing

Structured Streaming Processing Modes



Micro-batch Processing uses periodic tasks to process events



Continuous Processing uses long-running tasks to continuously process events

Anatomy of a Continunous Application

Anatomy of a Streaming Query

Streaming word count

Simple Streaming ETL

Anatomy of a Streaming Query: Step 1

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
```

Source

- Specify one or more locations to read data from
- Built in support for Files/Kafka/Socket, pluggable.

Anatomy of a Streaming Query: Step 2

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy(“value.cast("string") as key”)
  .agg(count("*") as “value”)
```

Transformation

- Using DataFrames, Datasets and/or SQL.
- Internal processing always exactly-once.

Anatomy of a Streaming Query: Step 3

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy(“value.cast("string") as key”)
  .agg(count("*") as “value”)
  .writeStream()
  .format("kafka")
  .option("topic", "output")

  .outputMode(OutputMode.Complete())
  .option("checkpointLocation", "…")
  .start()
```

Sink

- Accepts the output of each batch.
- When supported sinks are transactional and exactly once (Files).

Anatomy of a Streaming Query: Output Modes

```
from pyspark.sql import Trigger

spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy(“value.cast("string") as key”)
  .agg(count("*") as 'value’)
  .writeStream()
  .format("kafka")
  .option("topic", "output")
  .trigger("1 minute")
  .outputMode("update")
        ("checkpointLocation", "…")
  .start()
```

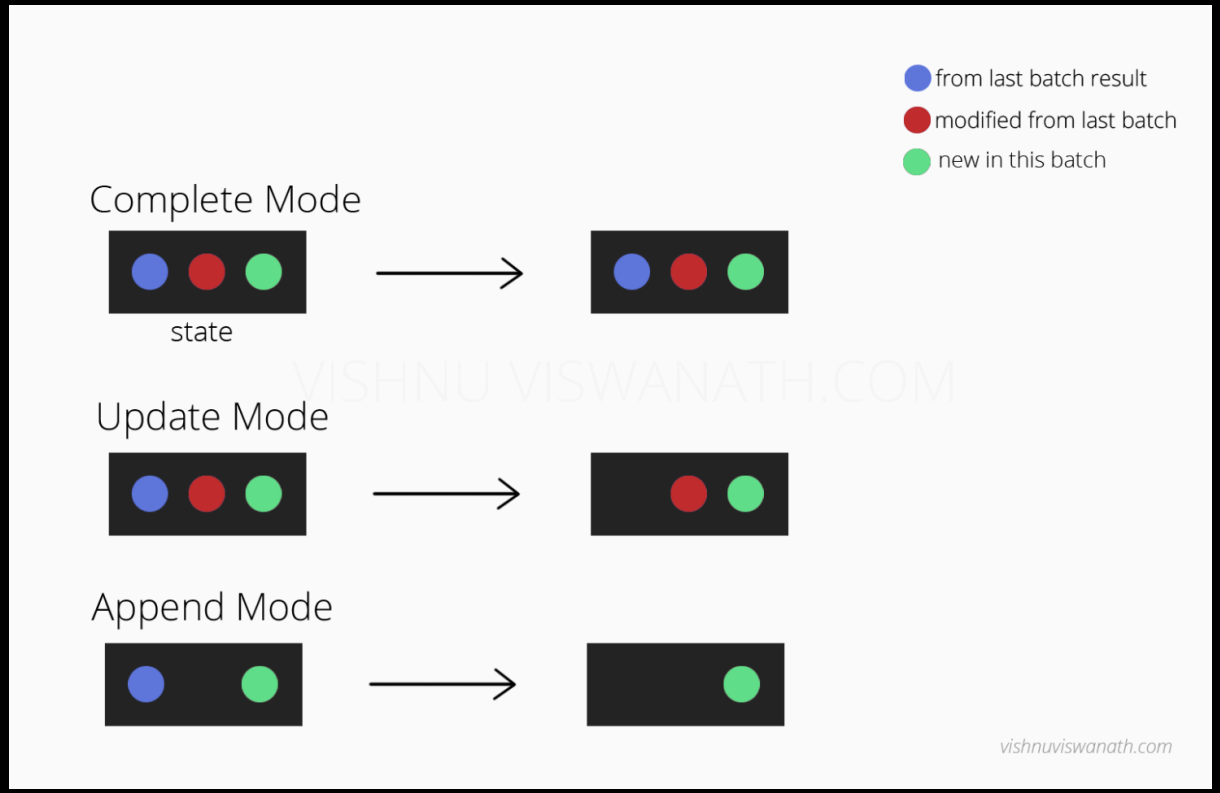
Output mode – What's output

- Complete – Output the whole answer every time
- Update – Output changed rows
- Append – Output new rows only

Trigger – When to output

- Specified as a time, eventually supports data size
- No trigger means as fast as possible

Streaming Query: Output Modes



Output mode – What's output

- Complete – Output the whole answer every time
- Update – Output changed rows
- Append – Output new rows only

Anatomy of a Streaming Query: Checkpoint

```
from pyspark.sql import Trigger

spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy(“value.cast("string") as key”)
  .agg(count("*") as 'value)
  .writeStream()
  .format("kafka")
  .option("topic", "output")
  .trigger("1 minute")
  .outputMode("update")
  .option("checkpointLocation", "…")
  .withWatermark(“timestamp” “2 minutes”)
  .start()
```

Checkpoint & Watermark

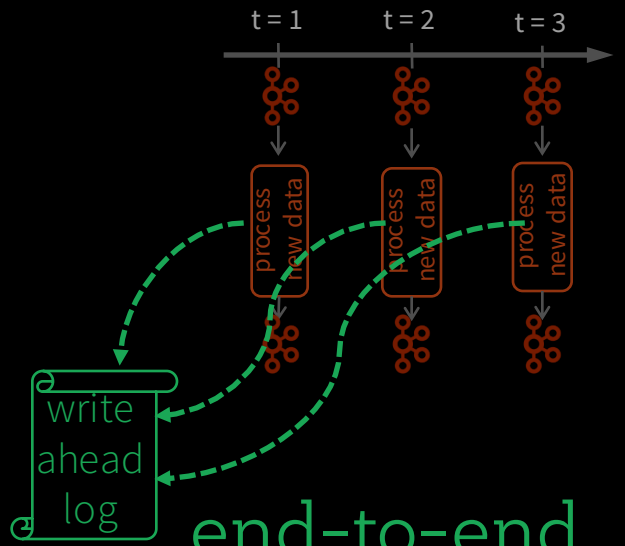
- Tracks the progress of a query in persistent storage
- Can be used to restart the query if there is a failure.
- *trigger(Trigger. Continunous(“ 1 second”))*

Set checkpoint location & watermark to drop very late events

Fault-tolerance with Checkpointing

Checkpointing – tracks progress (offsets) of consuming data from the source and intermediate state.

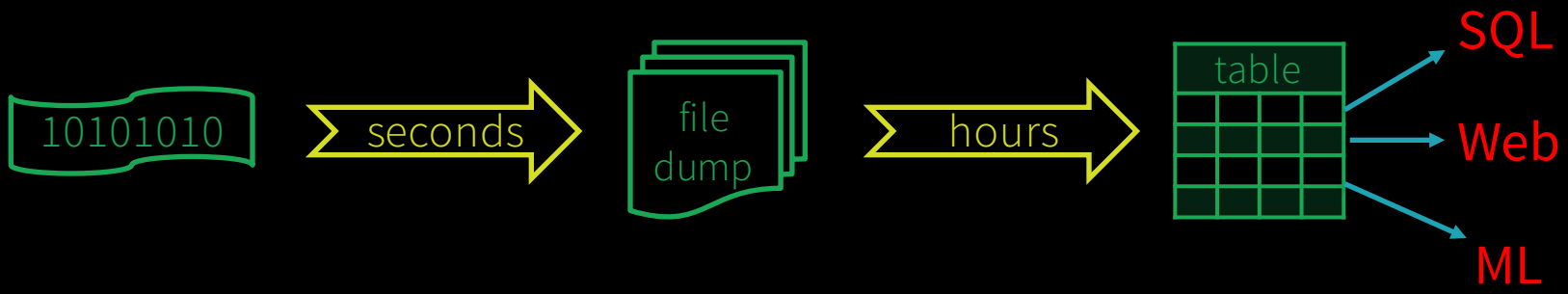
- Offsets and metadata saved as JSON
- Can resume after changing your streaming transformations



end-to-end exactly-once guarantees

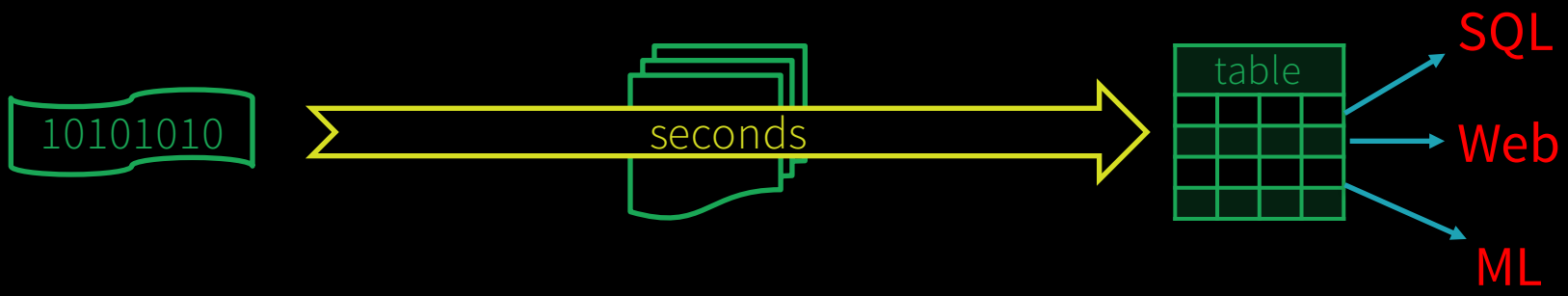
Complex Streaming ETL

Traditional ETL



- Raw, dirty, un/semi-structured is data dumped as files
- Periodic jobs run every few hours to convert raw data to structured data ready for further analytics
- Problem:
  - Hours of delay before taking decisions on latest data
  - Unacceptable when time is of essence
    - [intrusion , anomaly or fraud detection, monitoring IoT devices, etc.]

Streaming ETL w/ Structured Streaming



Structured Streaming Changes the Equation:

- eliminates latencies
- adds immediacy
- transforms data continuously

Streaming ETL w/ Structured Streaming

Example

1. Json data being received in Kafka
2. Parse nested json and flatten it
3. Store in structured Parquet table
4. Get end-to-end failure guarantees

```
from pyspark.sql import Trigger

rawData = spark.readStream
  .format("kafka")
  .option("kafka.boostrap.servers",...)
  .option("subscribe", "topic")
  .load()

parsedData = rawData
  .selectExpr("cast (value as string) as json"))
  .select(from_json("json", schema).as("data"))
  .select("data.*") # do your ETL/Transformation

query = parsedData.writeStream
  .option("checkpointLocation", "/checkpoint")
  .partitionBy("date")
  .format("parquet")
  .trigger( Trigger. Continunous("5 second"))
  .start("/parquetTable")
```

Reading from Kafka

rawData dataframe has the following columns

```
raw_data_df = spark.readStream
  .format("kafka")
  .option("kafka.boostrap.servers",...)
  .option("subscribe", "topic")
  .load()
```

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

Transforming Data

Cast binary *value* to string
Name it column *json*

```
parsedData = rawData
      .selectExpr("cast (value as string) as json")
      .select(from_json("json", schema).as("data"))
      .select("data.*")
```

Parse *json* string and expand into nested columns, name it *data*

Transforming Data

Cast binary *value* to string
Name it column *json*

Parse *json* string and expand into nested columns, name it data

Flatten the nested columns

```
parsedData = rawData
    .selectExpr("cast (value as string) as json")
    .select(from_json("json", schema).as("data"))
    .select("data.*")
```

powerful built-in Python APIs to perform complex data transformations

from_json, to_json, explode, ...
100s of functions

(see our blog post & tutorial)

Writing to Parquet

Save parsed data as Parquet table in the given path

Partition files by date so that future queries on time slices of data is fast

e.g. query on last 48 hours of data

```
query = parsedData.writeStream
      .option("checkpointLocation", ...)
      .partitionBy("date")
      .format("parquet")
      .start("/parquetTable") #pathname
```

Tutorials

databricks

https://dbricks.co/sais_pyspark_sf

https://community.cloud.databricks.com/?o=8599738367597028#clusters/create

databricks

Home

Workspace

Recents

Data

Clusters

Jobs

Search

Create Cluster

New Cluster | Cancel | Create Cluster | **0 Workers:** 0.0 GB Memory, 0 Cores, 0 DBU **1 Driver:** 6.0 GB Memory, 0.88 Cores, 1 DBU

Cluster Name

my_pyspark_tutorial → Enter your cluster name

Databricks Runtime Version

Runtime: 5.2 (Scala 2.11, Spark 2.4.0) → Use DBR 5.3 and Apache Spark 2.4, Scala 2.11

Python Version

3 | New The default Python version for clusters was changed from major version 2 to 3.

Instance

Free 6GB Memory: As a Community Edition user, your cluster will automatically terminate after an idle period of two hours.
For more configuration options, please upgrade your Databricks subscription.

Instances | Spark

Availability Zone

us-west-2c

Summary

- Apache Spark best suited for unified analytics & processing at scale
- Structured Streaming APIs Enables Continunous Applications
- Demonstrated Continunous Application

Resources

- Getting Started Guide with Apache Spark on Databricks
- docs.databricks.com
- Spark Programming Guide
- Structured Streaming Programming Guide
- Anthology of Technical Assets for Structured Streaming
- Databricks Engineering Blogs
- https://databricks.com/training/instructor-led-training

Thank You ☺

jules@databricks.com
@2twitme
https://www.linkedin.com/in/dmatrix/

Appendix

databricks