Structured Streaming in Apache Spark: Easy, Fault Tolerant and Scalable Stream Processing

Juliusz Sompolski

10th Extremely Large Databases Conference (XLDB)
October 11th 2017, Clermont-Ferrand, France

About Databricks

TEAM

Started Spark project (now Apache Spark) at UC Berkeley in 2009

MISSION

Making Big Data Simple

PRODUCT

Unified Analytics Platform

About Me

Software Engineer working in the new Databricks engineering office in Amsterdam

Opened in January 2017

So far expanded to 11 people and growing!

building robust stream processing apps is hard

Complexities in stream processing

Complex Data

Diverse data formats
(json, avro, binary, ...)

Data can be dirty, late, out-of-order

Complex Workloads

Event time processing

Combining streaming with interactive queries, machine learning

Complex Systems

Diverse storage systems and formats
(SQL, NoSQL, parquet, ...)

System failures

you
should not have to
reason about streaming

you

should write simple queries

&

Spark

should continuously update the answer

Structured Streaming

stream processing on Spark SQL engine

fast, scalable, fault-tolerant

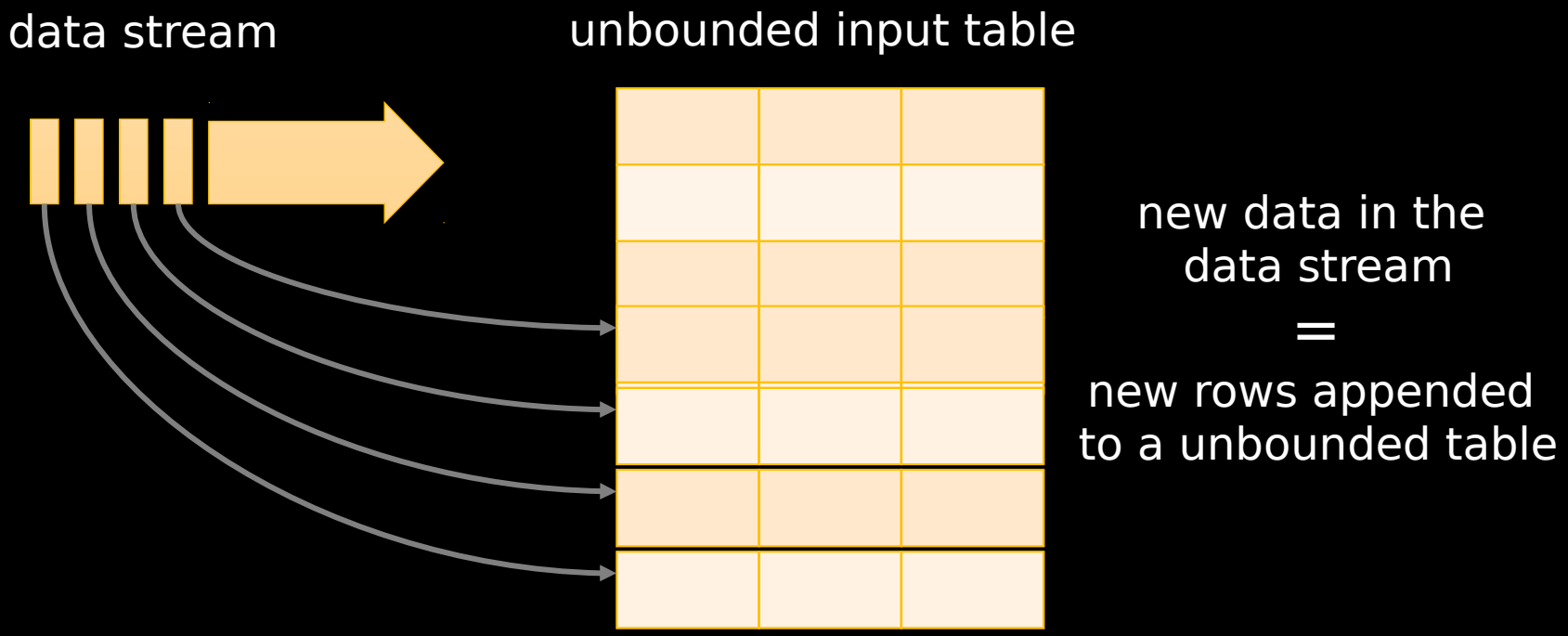
rich, unified, high level APIs

deal with *complex data* and *complex workloads*

rich ecosystem of data sources

integrate with many *storage systems*

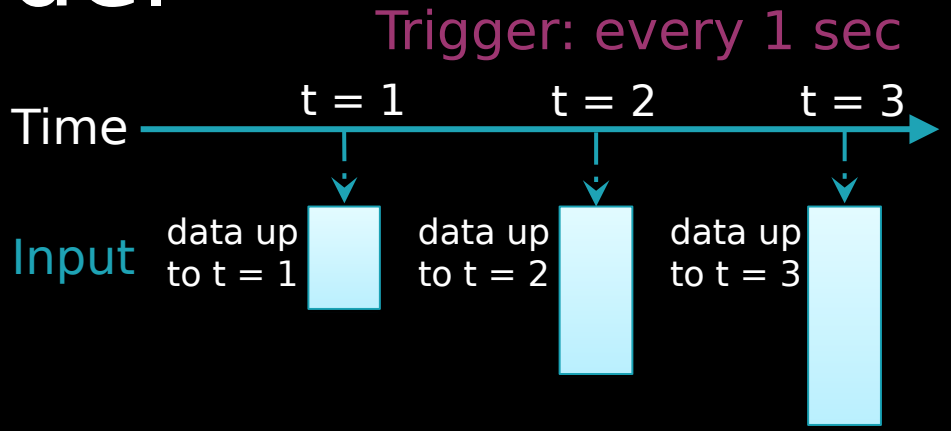
Treat Streams as Unbounded Tables



Conceptual Model

Treat input stream as an input table

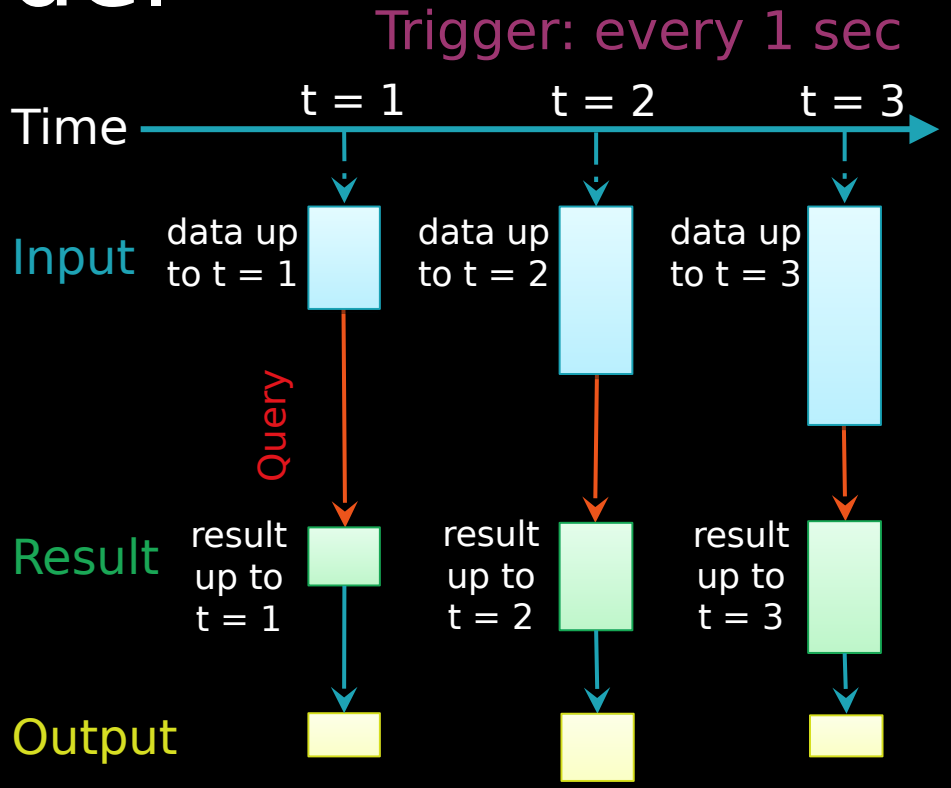
Every trigger interval, input table is effectively growing



Conceptual Model

If you apply a query on the input table, the result table changes with the input

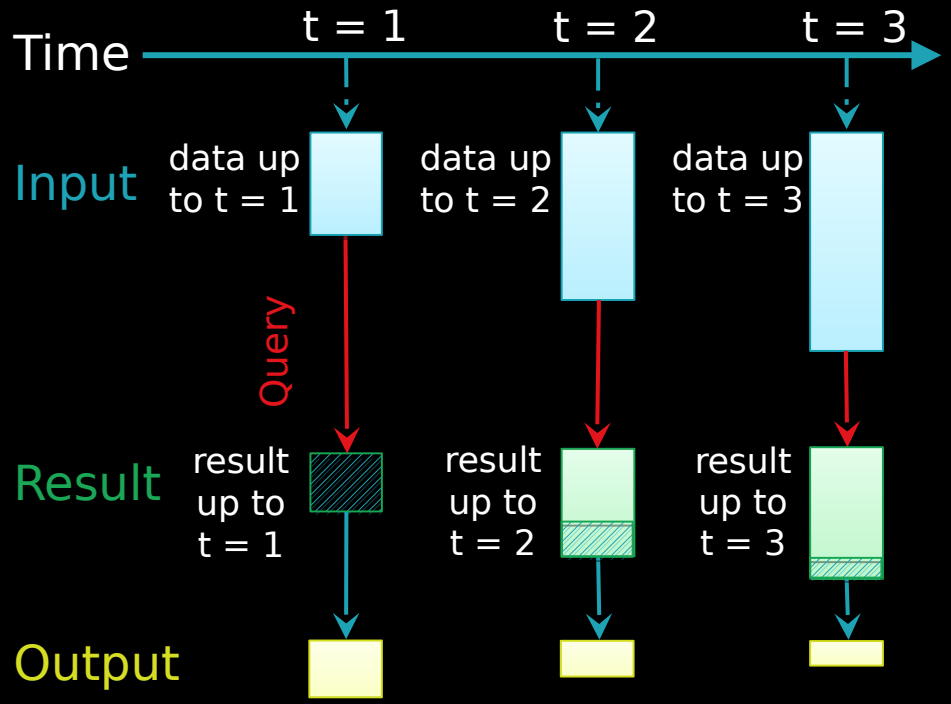
Every trigger interval, we can output the changes in the result



Conceptual Model

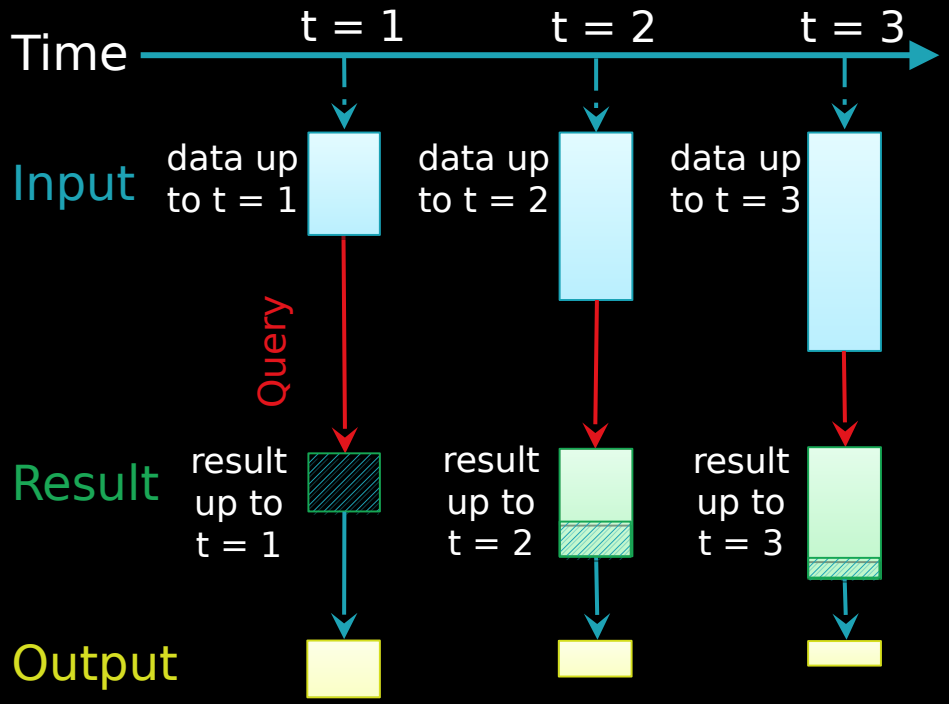
Full input does not need to be processed every trigger

Spark does not materialize the full input table



Conceptual Model

Spark converts query to an incremental query that operates only on new data to generate output



Anatomy of a Streaming Query

```
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.
  - Additional connectors, e.g. Amazon Kinesis available on Databricks platform
- Can union() multiple sources.

Anatomy of a Streaming Query

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

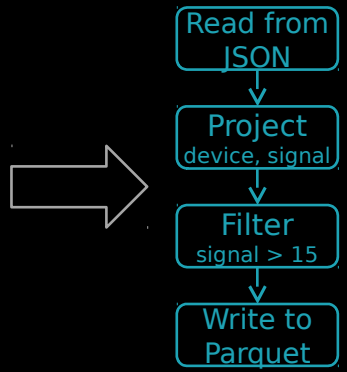
Transformation

- Using DataFrames, Datasets and/or SQL.
- Catalyst figures out how to execute the transformation incrementally.
- Internal processing always exactly-once.

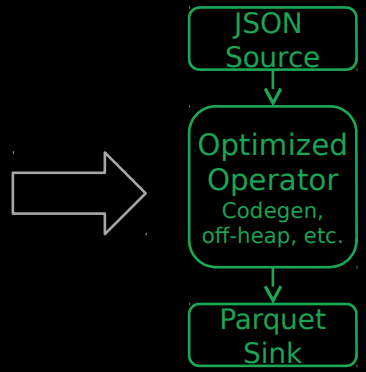
Spark automatically streamifies!

```
input = spark.readStream
  .format("json")
  .load("subscribe")

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

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

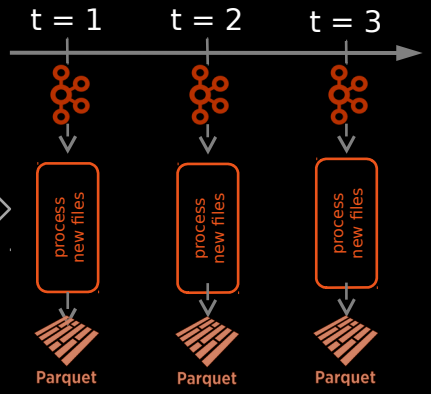
DataFrames, Datasets, SQL



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

Anatomy of a Streaming Query

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

Sink

- Accepts the output of each batch.
- When sinks are transactional, exactly once semantics.
- Use foreach to execute arbitrary code.

Anatomy of a Streaming Query

```
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")
```

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

Anatomy of a Streaming Query

```
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", "...")
  .start()
```

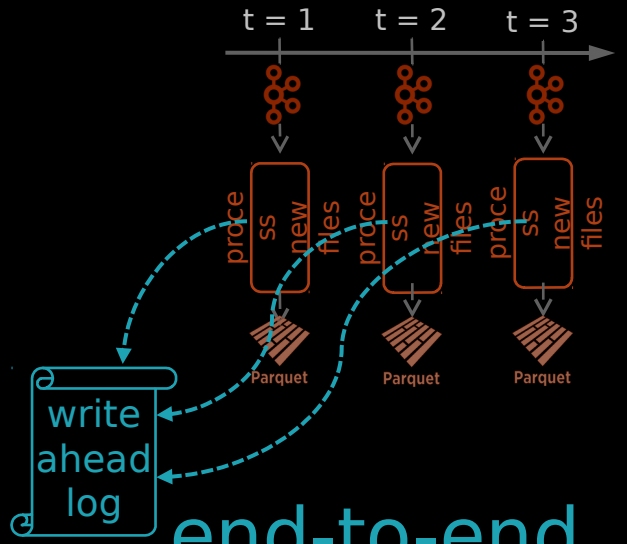
Checkpoint

- Tracks the progress of a query in persistent storage
- Can be used to restart the query if there is a failure.

Fault-tolerance with Checkpointing

Checkpointing - metadata (e.g. offsets) of current batch stored in a *write ahead log*

- Huge improvement over Spark Streaming checkpoints
- Offsets saved as JSON, no binary saved
- Can restart after app code change



end-to-end exactly-once guarantees

Dataset/DataFrame

SQL

```
spark.sql("
  SELECT type, sum(signal)
  FROM devices
  GROUP BY type
")
```

Most familiar to BI Analysts

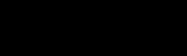
Supports SQL-2003, HiveQL

DataFrames

```
val df: DataFrame =
  spark.table("devices")
    .groupBy("type")
    .sum("signal"))
```

Great for Data Scientists familiar with Pandas, R Dataframes

Dataset



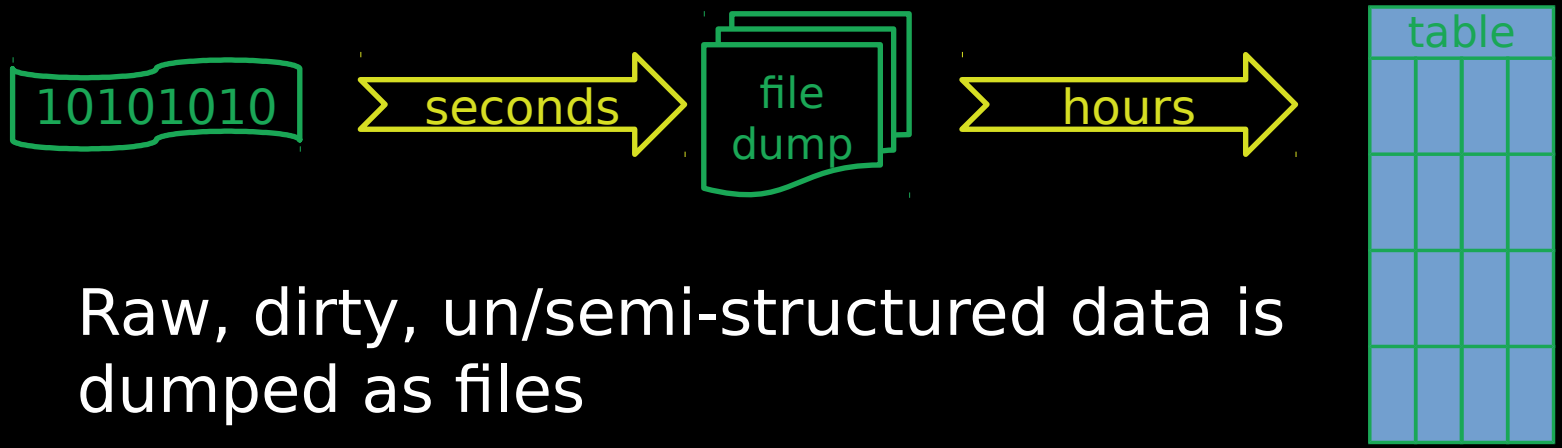
```
val ds: Dataset[(String, Double)] =
  spark.table("devices")
    .as[DeviceData]
    .groupByKey(_.type)
    .mapValues(_.signal)
    .reduceGroups(_ + _)
```

Great for Data Engineers who want compile-time type safety

You choose your hammer for whatever nail you have!

Complex Streaming ETL

Traditional ETL



Raw, dirty, un/semi-structured data is dumped as files

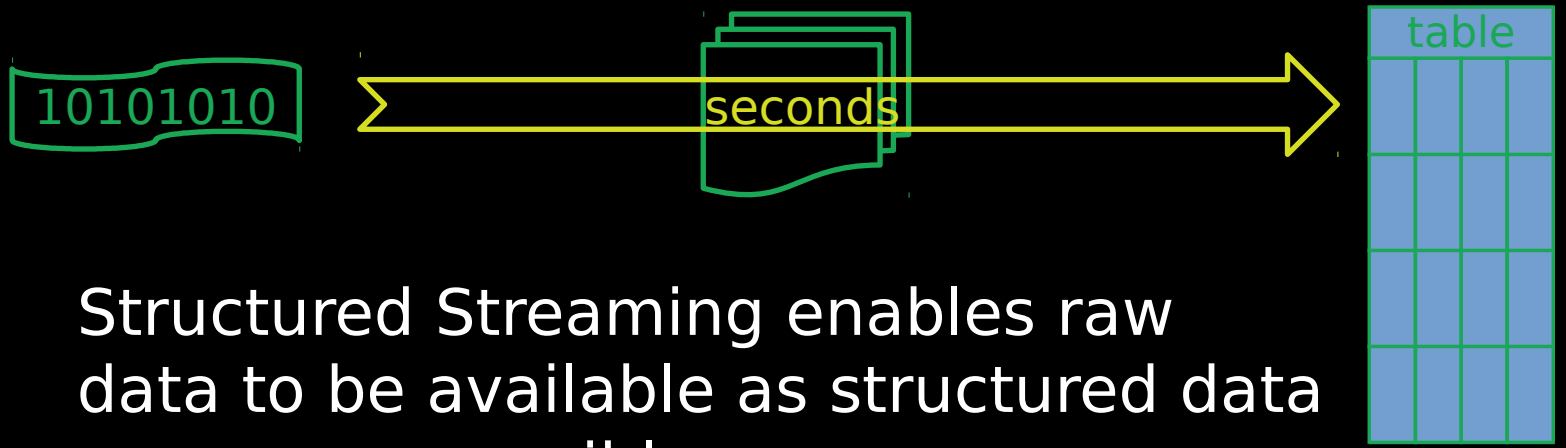
Periodic jobs run every few hours to convert raw data to structured data ready for further analytics

Traditional ETL

Hours of delay before taking decisions on latest data

Unacceptable when time is of essence
[intrusion detection, anomaly detection, etc.]

Streaming ETL w/ Structured Streaming



Structured Streaming enables raw data to be available as structured data as soon as possible

Streaming ETL w/ Structured Streaming

Example

Json data being received in Kafka

Parse nested json and flatten it

Store in structured Parquet table

Get end-to-end failure guarantees

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

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

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

Reading from Kafka

Specify options to configure

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

How?

kafka.boostrap.servers => broker1,broker2

What?

```
subscribe        =>  topic1,topic2,topic3   // fixed list of topics
subscribePattern =>  topic*                 // dynamic list of topics
assign           =>  {"topicA":[0,1] }      // specific partitions
```

Where?

startingOffsets => latest(default) / earliest / {"topicA":{"0":23,"1":345} }

Reading from Kafka

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

rawData dataframe has the following columns

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*

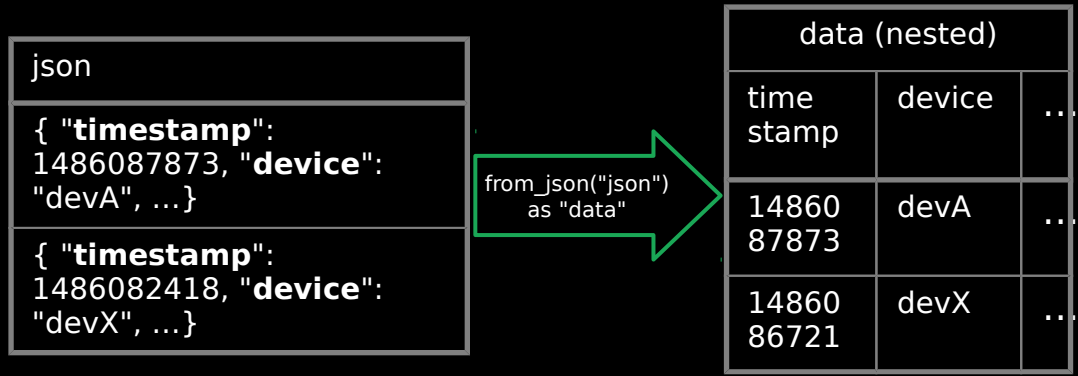
```
val parsedData = rawData
        .selectExpr("cast (value as string) as json")
        .select(from_json("json", schema).as("data"))
        .select("data.*")
```

Transforming Data

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

```
val 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*

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

Flatten the nested columns

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

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

powerful built-in APIs to perform complex data transformations

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

(see our blog post)

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

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

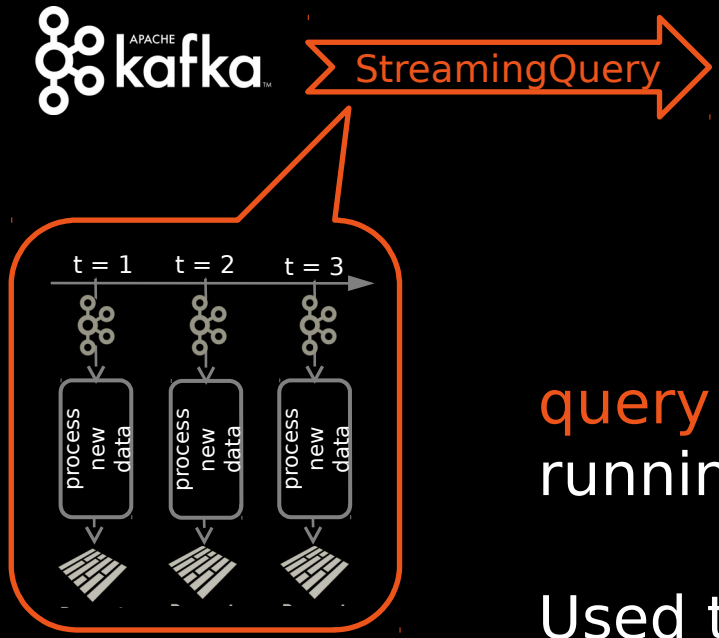
Fault tolerance

Enable checkpointing by setting the checkpoint location for fault tolerance

start() actually starts a continuous running StreamingQuery in the Spark cluster

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

Streaming Query




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

query is a handle to the continuously running StreamingQuery

Used to monitor and manage the execution

Data Consistency on Ad-hoc Queries

Data available for complex, ad-hoc analytics within seconds

Parquet table is updated atomically, ensures *prefix integrity*

Even if distributed, ad-hoc queries will see either all updates from streaming query or none, read more in our blog

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

Working With Time

Event Time

Many use cases require aggregate statistics by event time

- E.g. what's the #errors in each system in the 1 hour windows?

Many challenges

- Extracting event time from data, handling late, out-of-order data

DStream APIs were insufficient for event-time processing

Event time Aggregations

Windowing is just another type of grouping in Structured Streaming

number of records every hour

```
parsedData
    .groupBy(window("timestamp","1 hour"))
    .count()
```

avg signal strength of each device in 10 min windows, sliding every 5 minutes

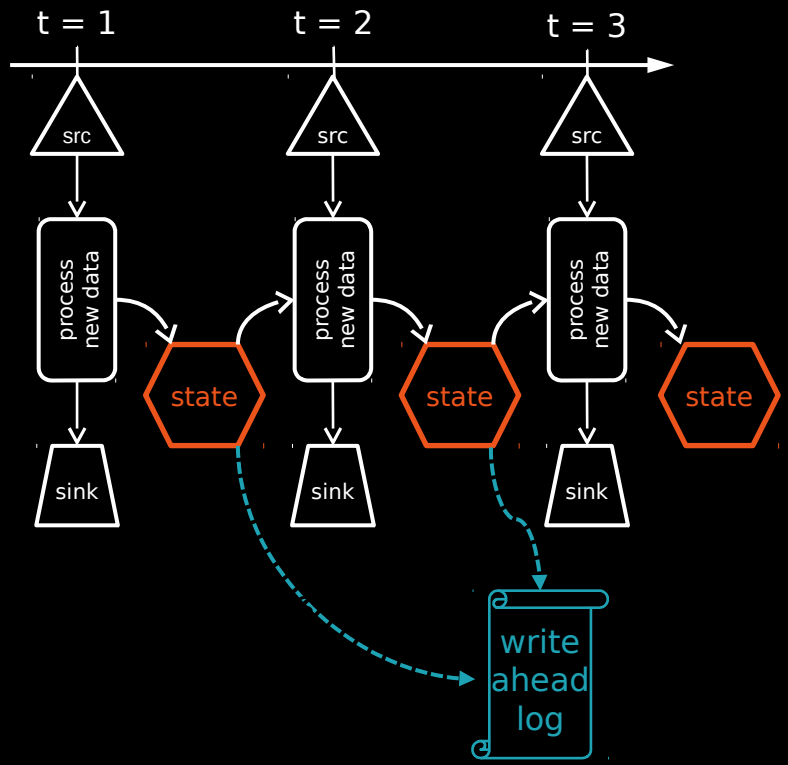
```
parsedData
    .groupBy(
    "device",
    window("timestamp","10 mins", “5 mins”))
    .avg("signal")
```

Support UDAFs!

Stateful Processing for Aggregations

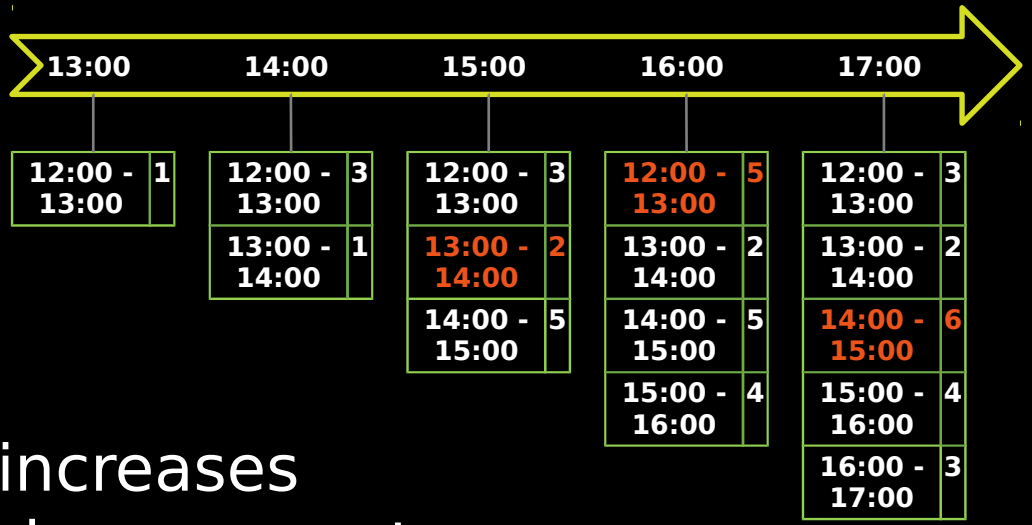
Aggregates has to be saved as distributed state between triggers

- Each trigger reads previous state and writes updated state
- State stored in memory, backed by *write ahead log* in HDFS/S3
- Fault-tolerant, exactly-once guarantee!



Automatically handles Late Data

Keeping state allows late data to update counts of old windows



But size of the state increases indefinitely if old windows are not dropped

red = state updated with late data

Watermarking to limit State

Watermark - moving threshold of how late data is expected to be and when to drop old state

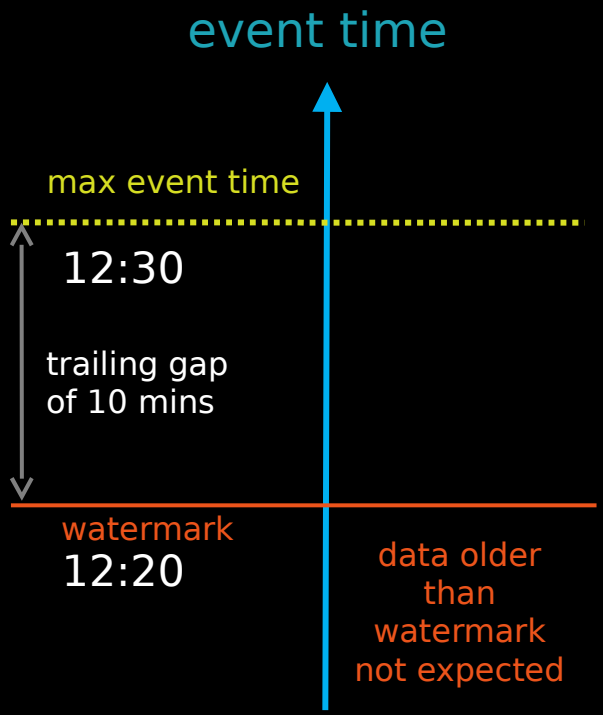
```
parsedData
    .withWatermark("timestamp", "10 minutes")
      .groupBy(window("timestamp","5 minutes"))
      .count()
```

Watermarking to limit State

Watermark - moving threshold of how late data is expected to be and when to drop old state

Trails behind max seen event time

Trailing gap is configurable

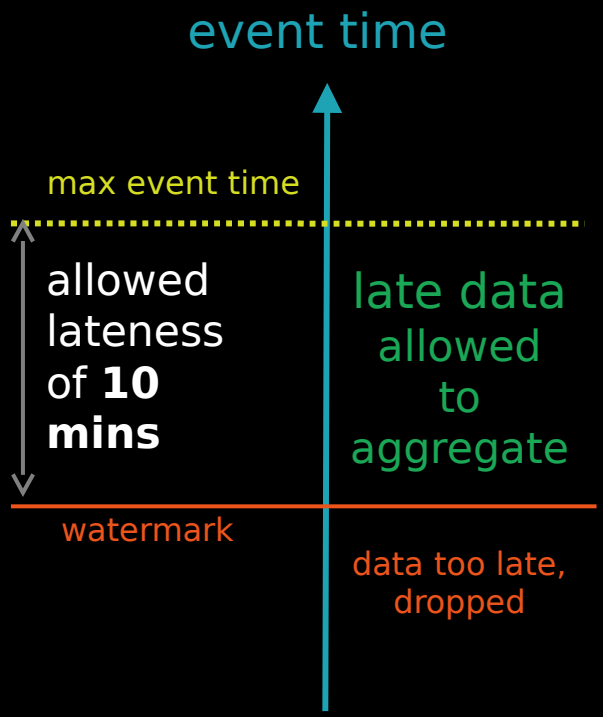


Watermarking to limit State

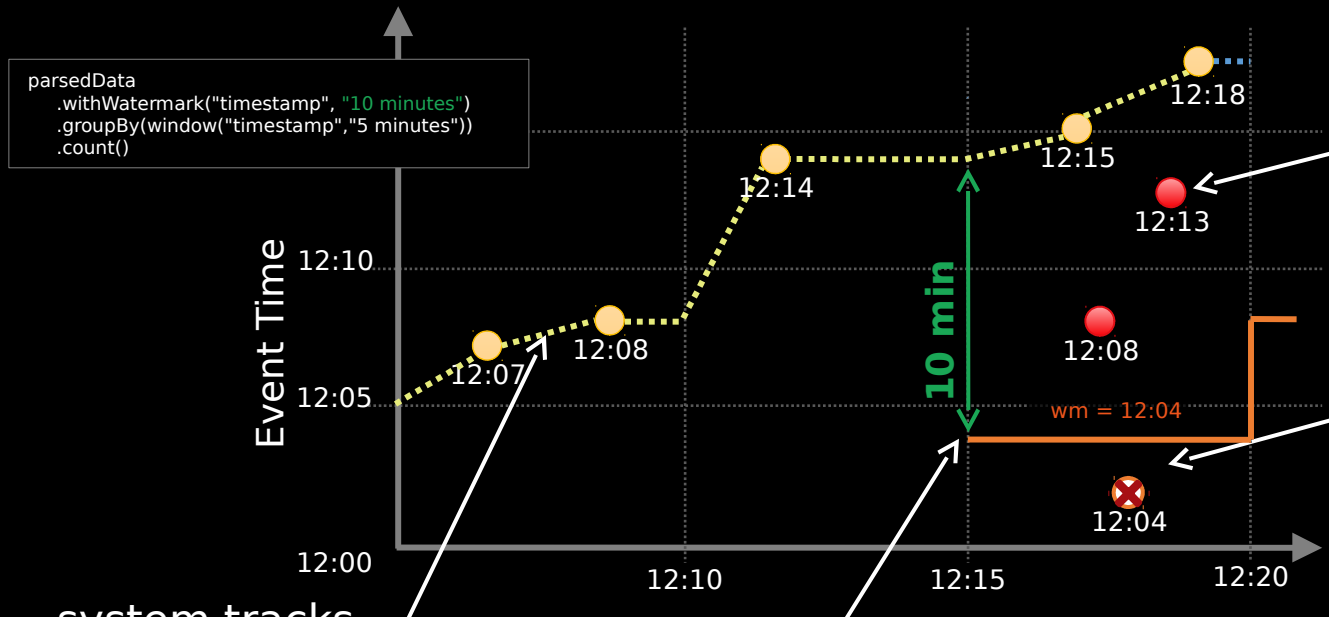
Data newer than watermark may be late, but allowed to aggregate

Data older than watermark is "too late" and dropped

Windows older than watermark automatically deleted to limit the amount of intermediate state



Watermarking to limit State



data is late, but considered in counts

data too late, ignored in counts, state dropped

system tracks max observed event time

watermark updated to 12:14 - 10m = 12:04 for next trigger, state < 12:04 deleted

More details in blog post

databricks

Clean separation of concerns

Query Semantics

separated from

Processing Details

```
parsedData
  .withWatermark("timestamp", "10 minutes")
  .groupBy(window("timestamp","5 minutes"))
  .count()
  .writeStream
  .trigger("10 seconds")
  .start()
```

Clean separation of concerns

Query Semantics

How to group data by time?
(same for batch & streaming)

Processing Details

```
parsedData
  .withWatermark("timestamp", "10 minutes")
  .groupBy(window("timestamp","5 minutes"))
  .count()
  .writeStream
  .trigger("10 seconds")
  .start()
```

Clean separation of concerns

Query Semantics

How to group data by time?
(same for batch & streaming)

Processing Details

How late can data be?

```
parsedData
  .withWatermark("timestamp", "10 minutes")
  .groupBy(window("timestamp","5 minutes"))
  .count()
  .writeStream
  .trigger("10 seconds")
  .start()
```

Clean separation of concerns

Query Semantics

How to group data by time?
(same for batch & streaming)

Processing Details

How late can data be?
How often to emit updates?

```
parsedData
  .withWatermark("timestamp", "10 minutes")
  .groupBy(window("timestamp","5 minutes"))
  .count()
  .writeStream
  .trigger("10 seconds")
  .start()
```

Arbitrary Stateful Operations [Spark 2.2]

(flat)mapGroupsWithState allows any user-defined stateful function to a user-defined state

Direct support for per-key timeouts in event-time or processing-time

Supports Scala and Java

```
ds.groupByKey(_.id)
  .mapGroupsWithState
    (timeoutConf)
    (mappingWithStateFunc)


def mappingWithStateFunc(
    key: K,
    values: Iterator[V],
    state: GroupState[S]): U = {
      // update or remove state
      // set timeouts
      // return mapped value
}
```

Alerting

Monitor a stream using custom stateful logic with timeouts.

```
val alerts = stream
  .as[Event]
  .groupBy(_.id)
  .flatMapGroupsWithState(Append, GST.ProcessingTimeTimeout) {
    (id: Int, events: Iterator[Event], state: GroupState[...]) =>
    ...
  }
  .writeStream
  .queryName("alerts")
  .foreach(new PagerdutySink(credentials))
```

Sessionization

Analyze sessions of user/system behavior

```
val sessions = stream
  .as[Event]
  .groupBy(_.session_id)
  .mapGroupsWithState(GroupStateTimeout.EventTimeTimeout) {
    (id: Int, events: Iterator[Event], state: GroupState[...]) =>
    ...
  }
  .writeStream
  .parquet("/user/sessions")
```

Sneak-peek into the future

Stream-stream joins [Spark 2.3]

- Can join two streams together
- State of such operation would grow indefinitely...

```
val clickStream = spark.readStream
        ...
        .select(‘clickImpressionId,
                ‘timestamp as “clickTS”, ...)


val impressionsStream = spark.readStream
        ...
        .select(‘impressionId,
                ‘timestamp as “impressionTS”, ...)


impressionsStream.join(clickStream,
        expr(“clickImpressionId = impressionId”))
```

Stream-stream joins [Spark 2.3]

- Can join two streams together
- Watermarking limits how late the data can come come
- Join condition limits how late we expect a click to happen after an impression

```
val clickStream = spark.readStream
        ...
        .select(‘clickImpressionId,
                ‘timestamp as “clickTS”, ...)
        .withWatermark(‘clickTS, “10 minutes”)

val impressionsStream = spark.readStream
        ...
        .select(‘impressionId,
                ‘timestamp as “impressionTS”, ...)
        .withWatermark(‘impressionTS, “10 minutes”)

impressionsStream.join(clickStream,
        expr(“clickImpressionId = impressionId AND” +
                “clickTS BETWEEN impressionTS AND” +
                  “impressionTS + interval 10 minutes”))
```

Stream-stream joins [Spark 2.3]

- Can join two streams together
- With watermarking and join condition limiting when a match could come, outer joins are possible

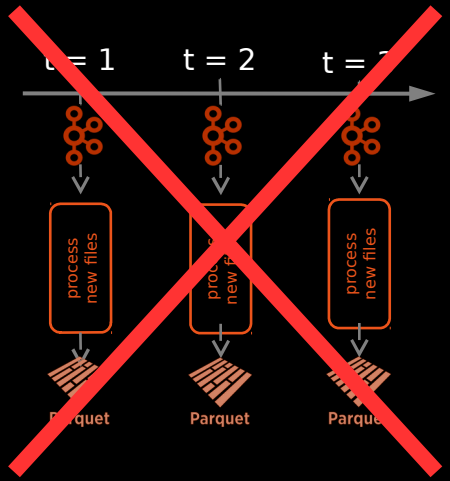
```
val clickStream = spark.readStream
        ...
        .select(‘clickImpressionId,
                ‘timestamp as “clickTS”, ...)
        .withWatermark(‘clickTS, “10 minutes”)

val impressionsStream = spark.readStream
        ...
        .select(‘impressionId,
                ‘timestamp as “impressionTS”, ...)
        .withWatermark(‘impressionTS, “10 minutes”)

impressionsStream.join(clickStream,
        expr(“clickImpressionId = impressionId AND” +
                “clickTS BETWEEN impressionTS AND” +
                  “impressionTS + interval 10 minutes”),
        “leftouter”)
```

Continuous processing [Spark 2.3]

A new execution mode that allows fully pipelined execution

- Streaming execution ***without microbatches***
- Supports async checkpointing and ~1ms latency
- No changes required to user code

Tracked in SPARK-20928



More Info

Structured Streaming Programming Guide

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

Anthology of Databricks blog posts and talks about structured streaming:

https://databricks.com/blog/2017/08/24/anthology-of-technical-assets-on-apache-sparks-structured-streaming.html

Try Apache Spark in Databricks!

UNIFIED ANALYTICS PLATFORM

- Collaborative cloud environment
- Free version (community edition)

DATABRICKS RUNTIME

- Apache Spark - optimized for the cloud
- Caching and optimization layer - DBIO
- Enterprise security - DBES

Try for free today

databricks.com

databricks

https://spark-summit.org/eu-2017/

Discount code:
Databricks

Pre-Spark Summit, Dublin