Easy, Scalable, Fault-tolerant Stream Processing with **Structured Streaming**

Tathagata "*TD*" Das
@tathadas

Spark Summit Europe 2017
25th October, Dublin

databricks

About Me

Started Spark Streaming project in AMPLab, UC Berkeley

Currently focused on building Structured Streaming

Member of the Apache Spark PMC

Software Engineer at Databricks

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

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

you

should write simple queries

&

Spark

should continuously update the answer

Streaming word count

Anatomy of a Streaming Word Count

```
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.
- Can include multiple sources of different types using `union()`

Anatomy of a Streaming Query

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

Transformation

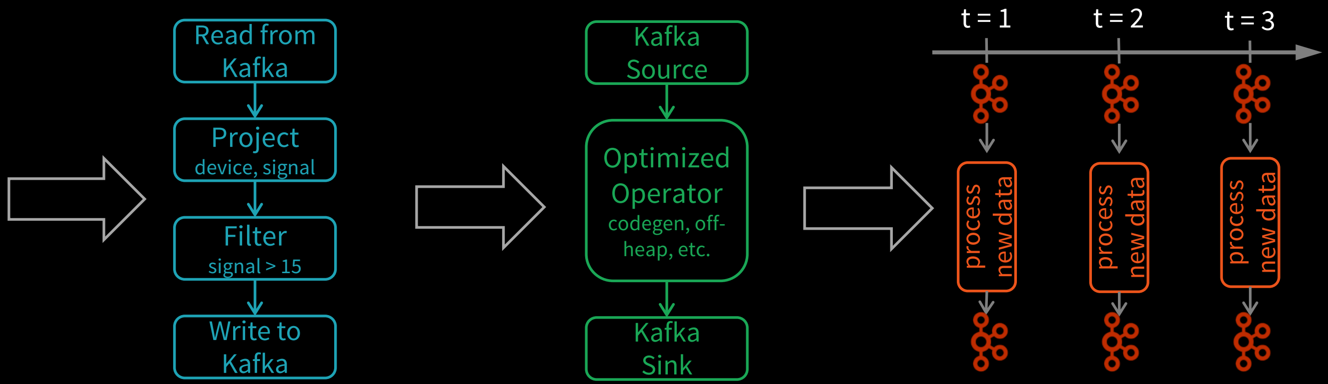
- Using DataFrames, Datasets and/or SQL.
- Catalyst figures out how to execute the transformation incrementally.
- Internal processing always exactly-once.

Spark automatically streamifies!

```
input = spark.readStream
   .format("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

databricks

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 supported sinks are transactional and exactly once (Files).
- 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()
```

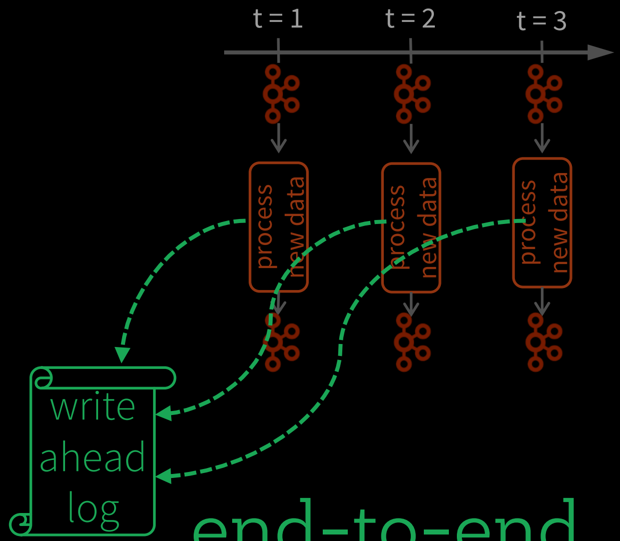
Checkpoint

- 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 – 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

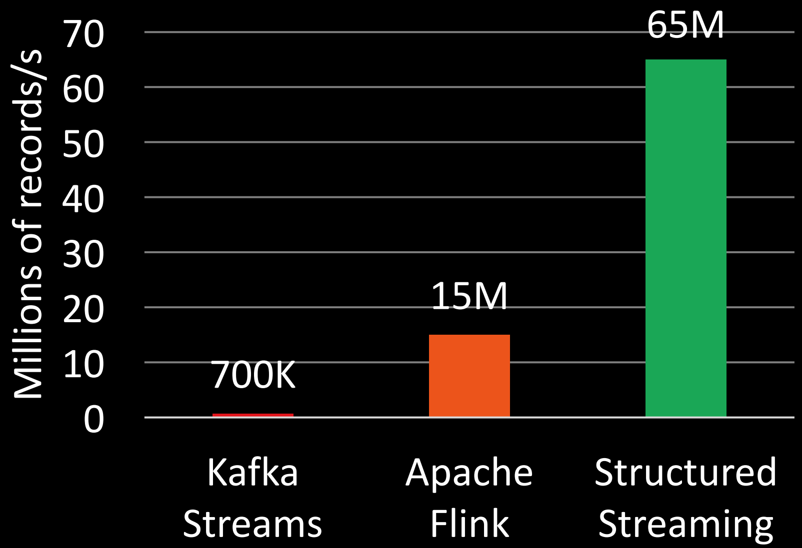
Performance: YAHOO! Benchmark

Structured Streaming reuses the **Spark SQL Optimizer** and **Tungsten Engine.**

4x

lower cost

System Throughput



Read more details in our blog post

databricks

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

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*

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.*")
```

json
{ "timestamp": 1486087873, "device": "devA", …}
{ "timestamp": 1486082418, "device": "devX", …}

from_json("json")
as "data"

data (nested)		
timestamp	device	…
1486087873	devA	…
1486086721	devX	…

databricks

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.*")
```

data (nested)		
timestamp	device	…
1486087873	devA	…
1486086721	devX	…

select("data.*")

(not nested)

timestamp	device	…
1486087873	devA	…
1486086721	devX	…

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

Flatten the nested columns

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
      .option("checkpointLocation", ...)
      .partitionBy("date")
      .format("parquet")
      .start("/parquetTable")
```

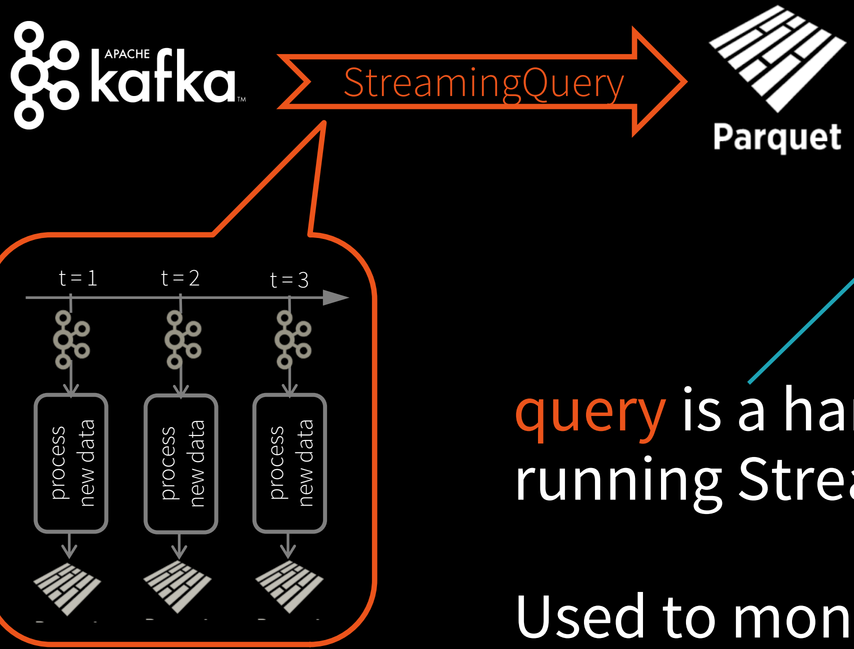
Checkpointing

Enable checkpointing by setting the checkpoint location to save offset logs

`start` actually starts a continuous running StreamingQuery in the Spark cluster

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

Streaming Query




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

More Kafka Support

Write out to Kafka

- Dataframe must have binary fields named key and value

Direct, interactive and batch queries on Kafka

- Makes Kafka even more powerful as a storage platform!

Added to Spark 2.2

```
result.writeStream
  .format("kafka")
  .option("topic", "output")
  .start()
```

```
val df = spark
  .read        // not readStream
  .format("kafka")
  .option("subscribe", "topic")
  .load()

 df.registerTempTable("topicData")
 spark.sql("select value from topicData")
```

databricks

Amazon Kinesis

Configure with options (similar to Kafka)
Available with Databricks Runtime

```
spark.readStream
  .format("kinesis")
  .option("streamName", "myStream")
  .option("region", "us-west-2")
  .option("awsAccessKey", ...)
  .option("awsSecretKey", ...)
  .load()
```

How?

```
region => us-west-2 / us-east-1 / ...
awsAccessKey (optional) => AKIA...
awsSecretKey (optional) => ...
```

What?

streamName => *name-of-the-stream*

Where?

initialPosition => latest (default) / earliest / trim_horizon

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 stuff

Event time Aggregations

Windowing is just another type of grouping in Struct. Streaming

number of records every hour

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

avg signal strength of each device every 10 mins

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

Support UDAFs!

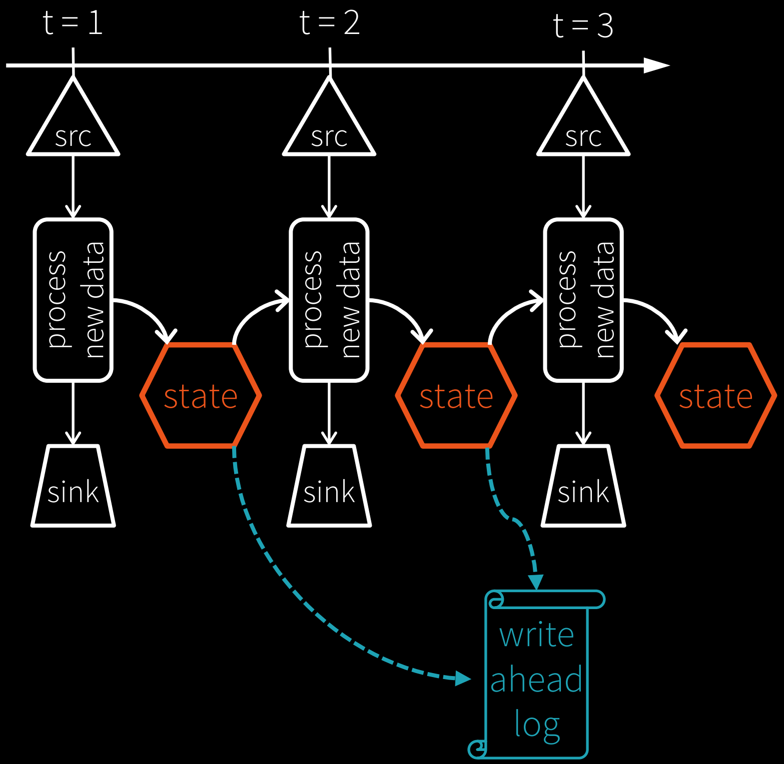
Stateful Processing for Aggregations

Aggregates has to be saved as distributed state between triggers

Each trigger reads previous state and writes updated state

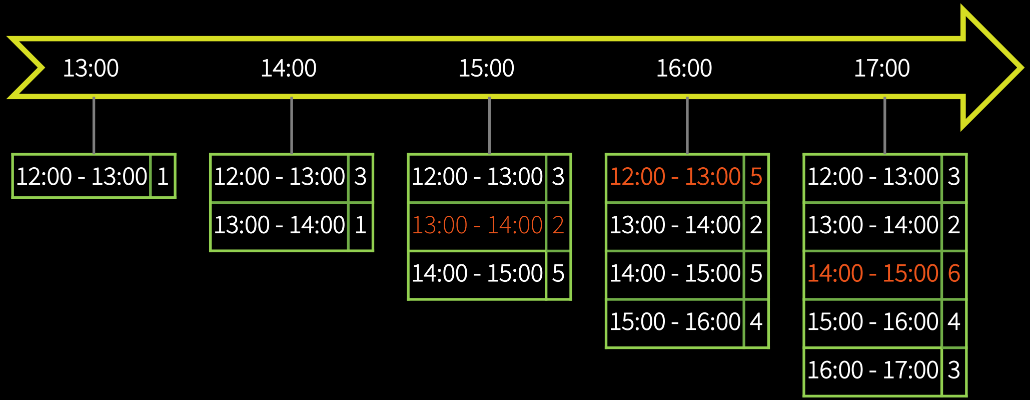
State stored in memory, backed by *write ahead log* in HDFS/S3

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

databricks

Watermarking

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

Trails behind max event time seen by the engine

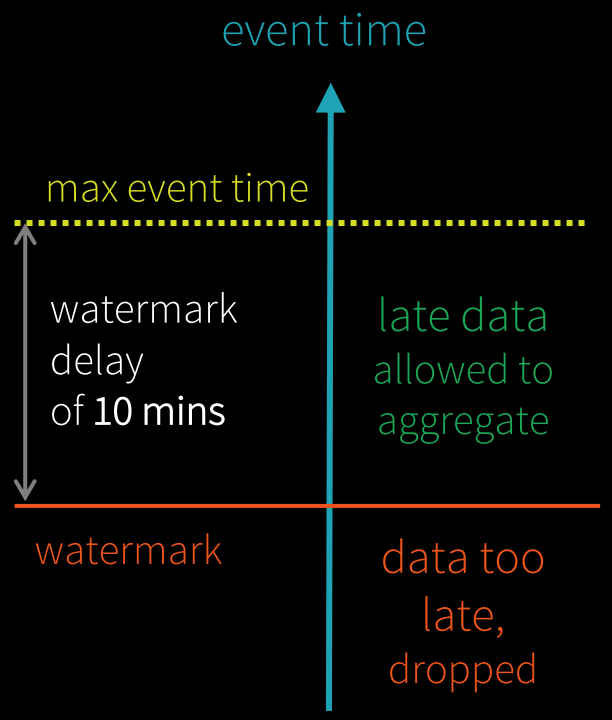
Watermark delay = trailing gap

Watermarking

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



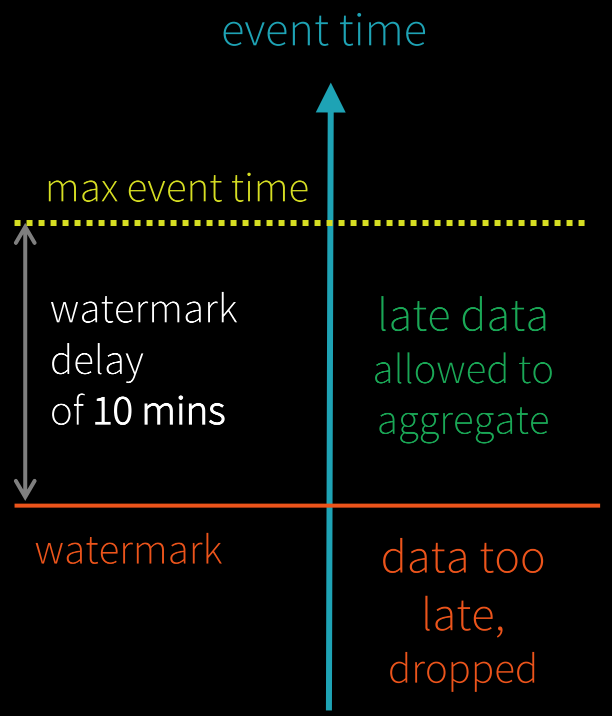
databricks

Watermarking

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

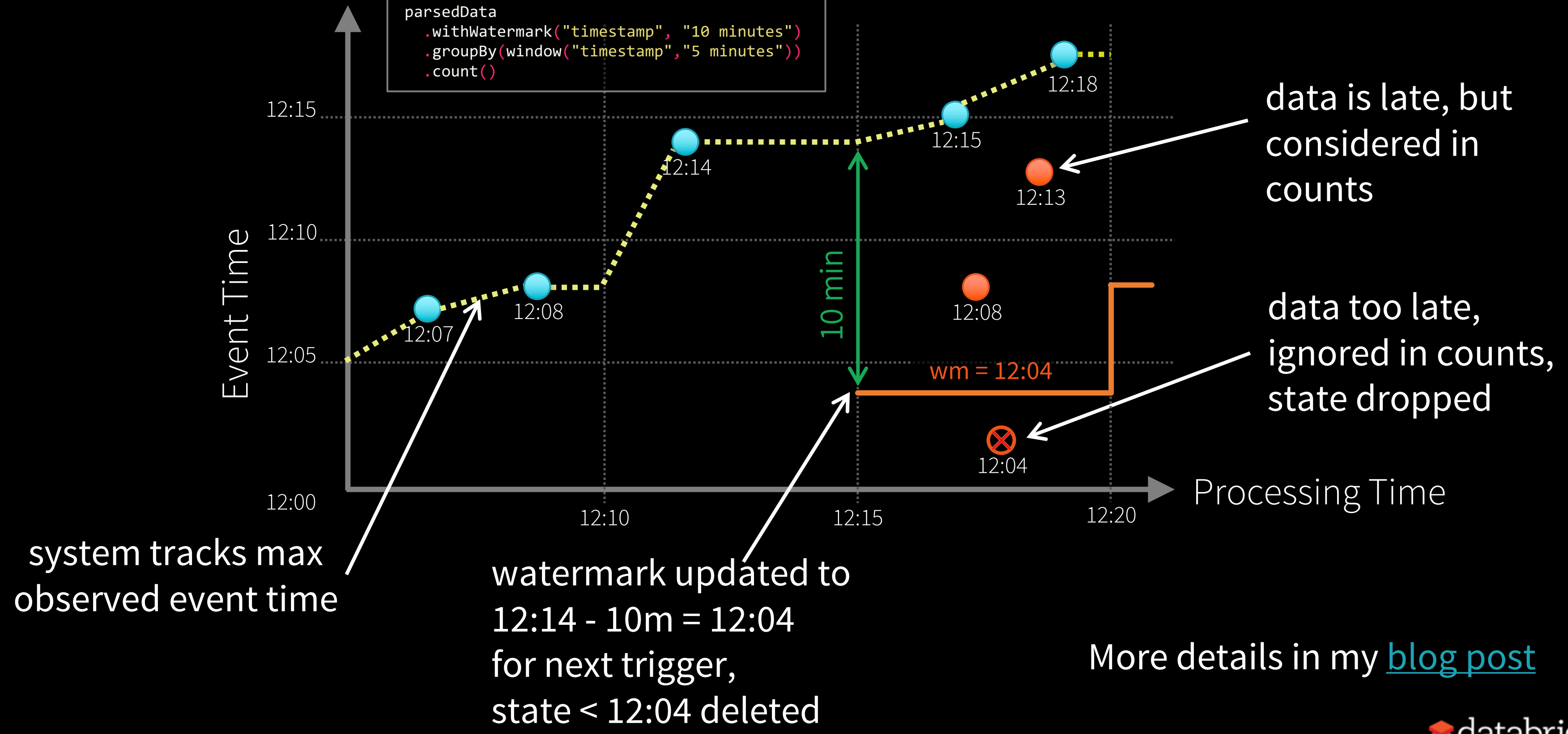
Useful only in stateful operations

Ignored in non-stateful streaming queries and batch queries



Watermarking

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



More details in my [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()
```

databricks

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)

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

Processing Details

How late can data be?

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()
```

databricks

Other Interesting Operations

Streaming Deduplication

Watermarks to limit state

```
parsedData.dropDuplicates("eventId")
```

Joins

Stream-batch joins supported,
stream-stream joins coming in 2.3

```
parsedData.join(batchData, "device")
```

Arbitrary Stateful Processing

[map|flatMap]GroupsWithState

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

[See my other talk at 4:20 PM, today for a deep dive into stateful ops]

Monitoring Streaming Queries

Get last progress of the streaming query

- Current input and processing rates
- Current processed offsets
- Current state metrics

```
streamingQuery.lastProgress()

  { ...
    "inputRowsPerSecond" : 10024.225210926405,
    "processedRowsPerSecond" : 10063.737001006373,
    "durationMs" : { ... },
    "sources" : [ ... ],
    "sink" : { ... }
    ...
  }
```

Get progress asynchronously through by registering your own StreamingQueryListener

```
new StreamingQueryListener {
  def onQueryStart(...)
  def onQueryProgress(...)
  def onQueryTermination(...)
}
```

databricks

Dropwizard Metrics

Metrics into Ganglia, Graphite, etc.

Enabled using SQL configuration

```
spark.conf.set("spark.sql.streaming.metricsEnabled", "true")
```

Building Complex Continuous Apps

databricks

Metric Processing @ databricks

Events generated by user actions (logins, clicks, spark job updates)

ETL	Clean, normalize and store historical data
Dashboards	Analyze trends in usage as they occur
Alerts	Notify engineers of critical issues
Ad-hoc Analysis	Diagnose issues when they occur

Metric Processing @ databricks

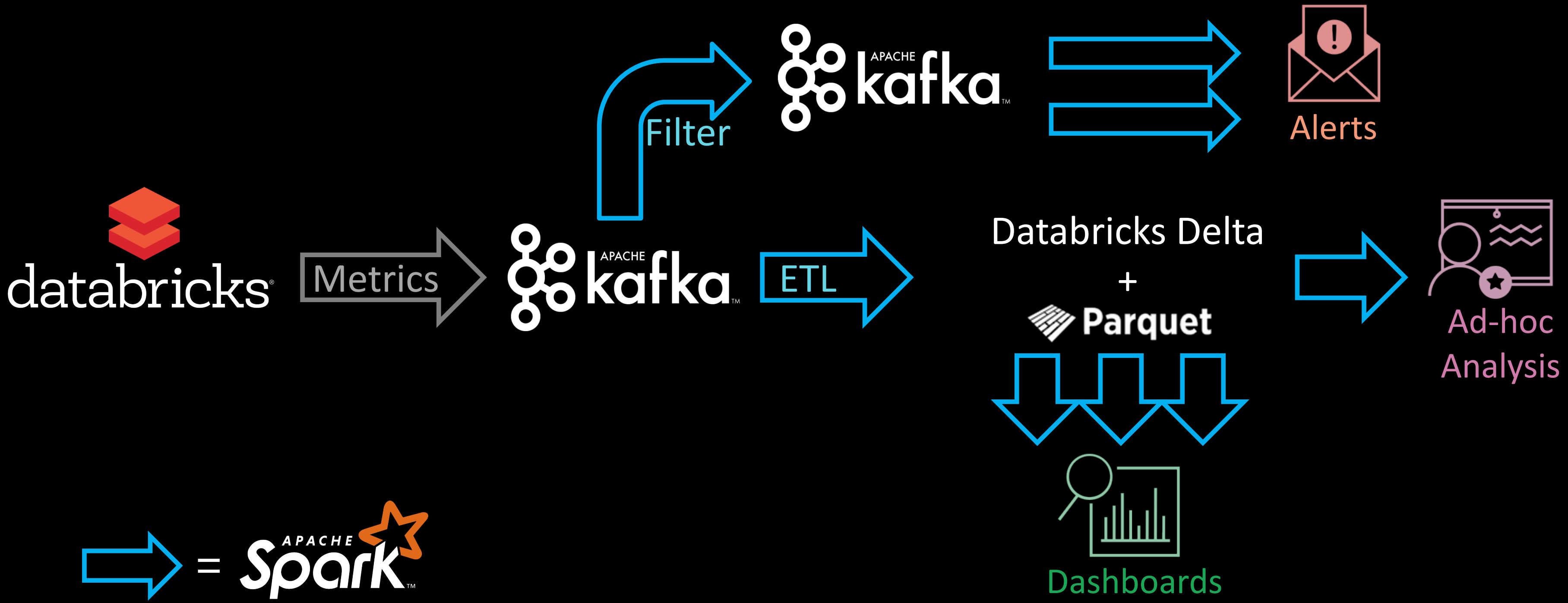
Difficult with only streaming frameworks

- ETL
- Dashboards
- Alerts
- Ad-hoc Analysis

kafka

- Limited retention in streaming storage
- Inefficient for ad-hoc queries
- Hard for novice users (limited SQL support)

Metric Processing @ databricks



Read from APACHE kafka


```
rawLogs = spark.readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", ...)
    .option("subscribe", "rawLogs")
    .load()

augmentedLogs = rawLogs
    .withColumn("msg",
       from_json($"value".cast("string"),
       schema))
    .select("timestamp", "msg.*")
    .join(table("customers"), ["customer_id"])
```

DataFrames can be reused for multiple streams

Can build libraries of useful DataFrames and share code between applications

Write to Parquet

APACHE kafka → ETL → Databricks Delta + Parquet

Store augmented stream as efficient columnar data for later processing

Latency: ~1 minute

```
augmentedLogs
  .repartition(1)
  .writeStream
  .format("delta")
  .option("path", "/data/metrics")
  .trigger("1 minute")
  .start()
```

Buffer data and write one large file every minute for efficient reads

Dashboards

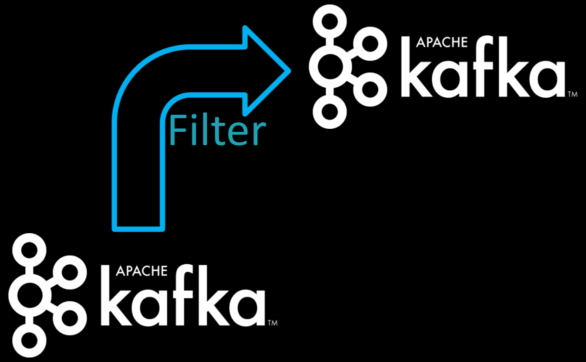
Always up-to-date visualizations of important business trends

Latency: ~1 minute to hours (configurable)

```
logins = spark.readStream.parquet("/data/metrics")
  .where("metric = 'login'")
  .groupBy(window("timestamp", "1 minute"))
  .count()

display(logins)      // visualize in Databricks notebooks
```

Filter and write to APACHE kafka



Forward filtered and augmented events back to Kafka

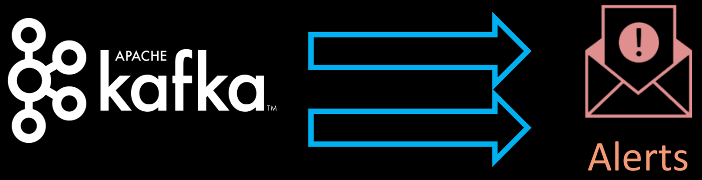
Latency: ~100 ms average

```
filteredLogs = augmentedLogs
  .where("eventType = 'clusterHeartbeat'")
  .selectExpr("to_json(struct("*")) as value")

filteredLogs.writeStream
  .format("kafka")
  .option("kafka.bootstrap.servers", ...)
  .option("topic", "clusterHeartbeats")
  .start()
```

`to_json()` to convert columns back into json string, and then save as different Kafka topic

Simple Alerts



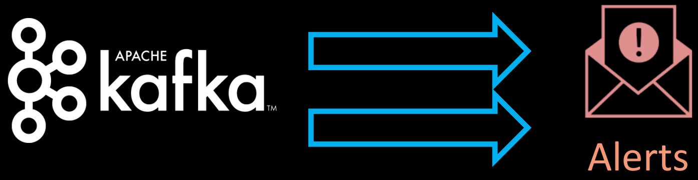
E.g. Alert when Spark cluster load > threshold

Latency: ~100 ms

```
sparkErrors
  .as[ClusterHeartBeat]
  .filter(_.load > 99)
  .writeStream
  .foreach(new PagerdutySink(credentials))
```

notify PagerDuty

Complex Alerts



E.g. Monitor health of Spark clusters using custom stateful logic

Latency: ~10 seconds

react if no heartbeat from cluster for 1 min

```
sparkErrors
  .as[ClusterHeartBeat]
  .groupBy(_.id)
  .flatMapGroupsWithState(Update, EventTimeTimeout) {
    (id: Int, events: Iterator[ClusterHeartBeat], state: GroupState[ClusterState]) =>
    ... // check if cluster non-responsive for a while
  }
```

Ad-hoc Analysis

Databricks Delta
+
Parquet

Ad-hoc Analysis

Trouble shoot problems as they occur with latest information

Latency: ~1 minute

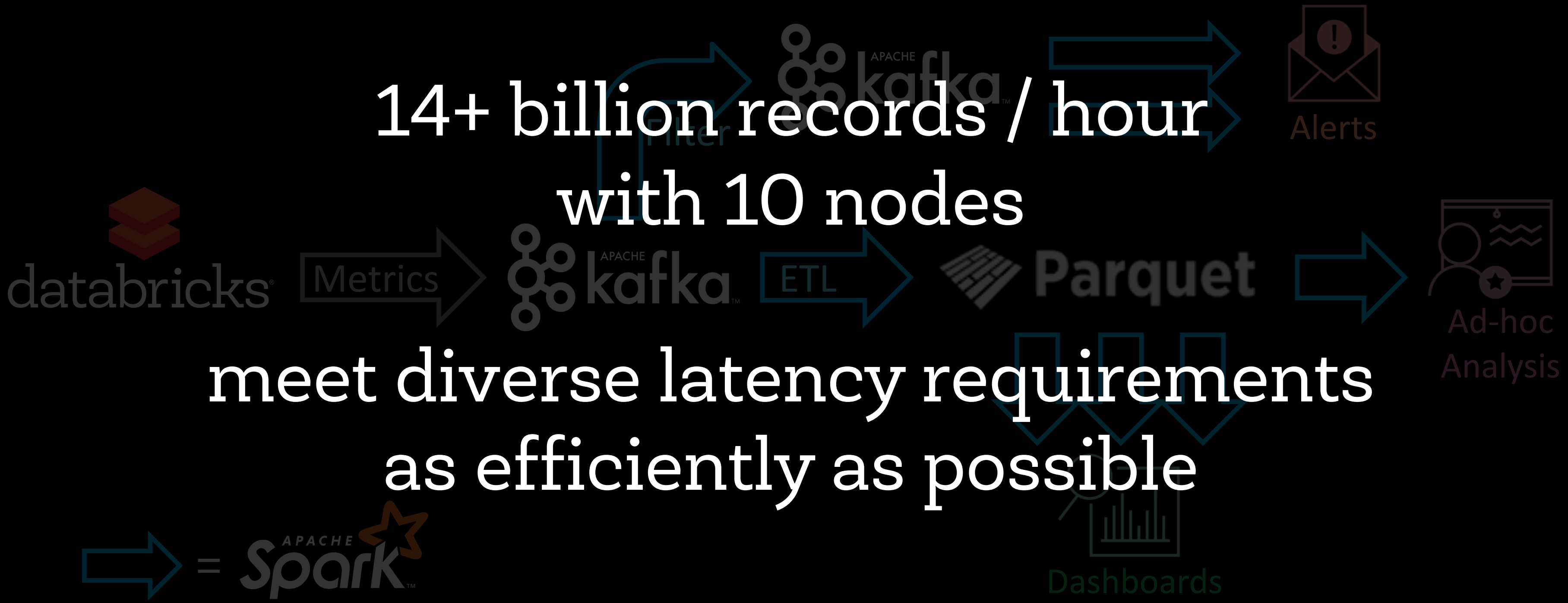
```
SELECT *
FROM delta.`/data/metrics`
WHERE level IN ('WARN', 'ERROR')
  AND customer = "…"
  AND timestamp < now() – INTERVAL 1 HOUR
```

will read latest data when query executed

Metric Processing @ databricks

14+ billion records / hour
with 10 nodes

meet diverse latency requirements
as efficiently as possible



Structured Streaming @ databricks

100s of customer streaming apps
in production on Databricks

Largest app process 10s of trillions
of records per month

Future Direction: Continuous Processing

Continuous processing mode to run without micro-batches

<1 ms latency (same as per-record streaming systems)

No changes to user code

Proposal in SPARK-20928, expected in Spark 2.3

More Info

Structured Streaming Programming Guide

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

Databricks blog posts for more focused discussions

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

https://databricks.com/blog/2017/01/19/real-time-streaming-etl-structured-streaming-apache-spark-2-1.html

https://databricks.com/blog/2017/02/23/working-complex-data-formats-structured-streaming-apache-spark-2-1.html

https://databricks.com/blog/2017/04/26/processing-data-in-apache-kafka-with-structured-streaming-in-apache-spark-2-2.html

https://databricks.com/blog/2017/05/08/event-time-aggregation-watermarking-apache-sparks-structured-streaming.html

https://databricks.com/blog/2017/10/11/benchmarking-structured-streaming-on-databricks-runtime-against-state-of-the-art-streaming-systems.html

and more to come, stay tuned!!

Try Apache Spark in Databricks!

UNIFIED ANALYTICS PLATFORM

- Collaborative cloud environment
- Free version (community edition)

DATABRICKS RUNTIME 3.0

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

Try for free today

databricks.com