

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

Burak Yavuz

DataEngConf NYC
October 31st 2017



Who am I



- Software Engineer – Databricks
 - “We make your streams come true”
- Apache Spark Committer
- MS in Management Science & Engineering - Stanford University
- BS in Mechanical Engineering - Bogazici University, Istanbul

About Databricks

TEAM

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

MISSION

Making Big Data Simple

PRODUCT

Unified Analytics Platform

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

Anatomy of a Streaming Query

Streaming word count

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.
- 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.

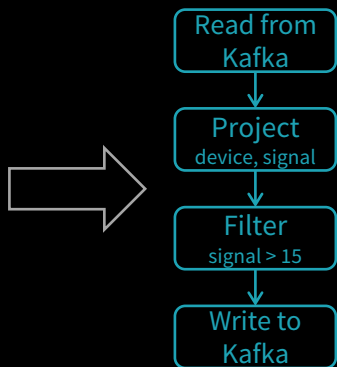
Spark automatically streamifies!

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

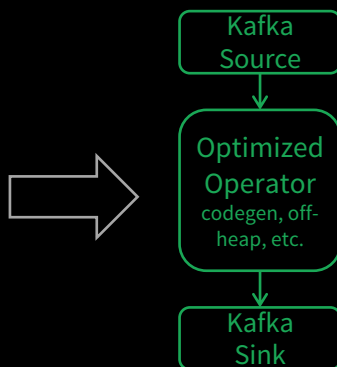
result = input
    .select("device", "signal")
    .where("signal > 15")

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

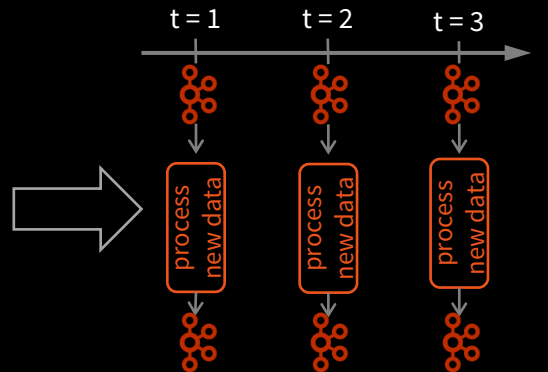
DataFrames,
Datasets, SQL



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

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

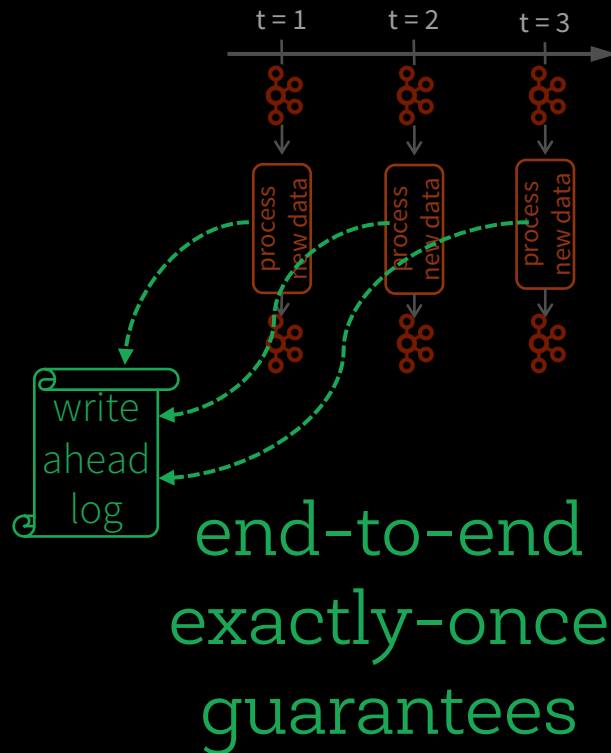
- Tracks the progress of a query in persistent storage
- Can be used to restart the query if there is a failure.

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





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

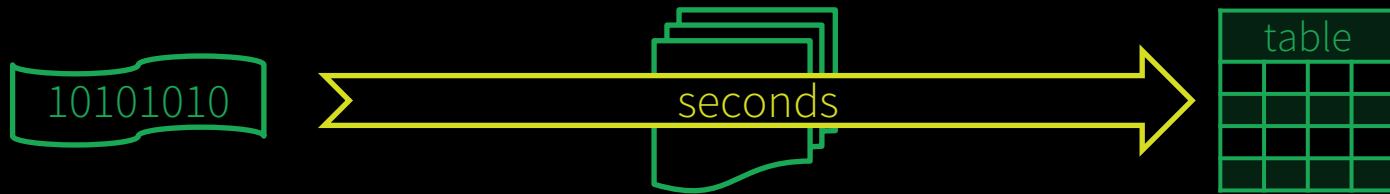
Traditional ETL



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.bootstrap.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

How?

```
kafka.bootstrap.servers => broker1,broker2
```

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

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

rawData dataframe has
the following columns

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



key	value	topic	partition	offset	timestamp
<i>[binary]</i>	<i>[binary]</i>	"topicA"	0	345	1486087873
<i>[binary]</i>	<i>[binary]</i>	"topicB"	3	2890	1486086721

Transforming Data

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

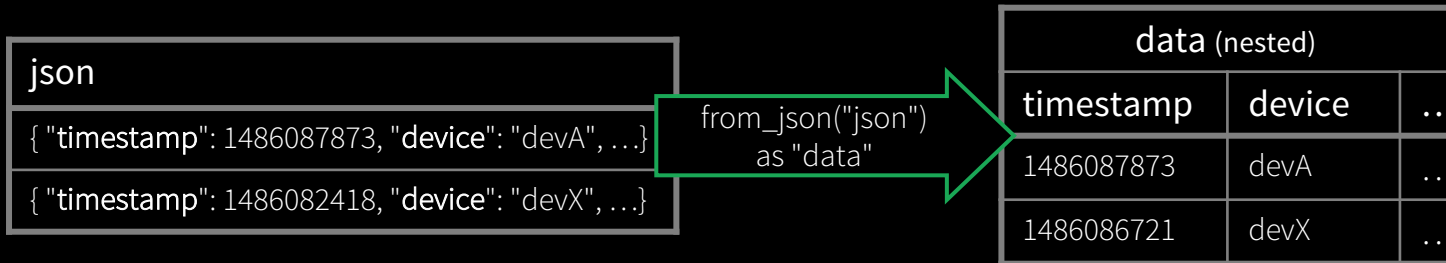
```
val parsedData = rawData
    .selectExpr("cast (value as string) as json")
    .select(from_json("json", schema).as("data"))
    .select("data.*")
```


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*

```
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

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")
```

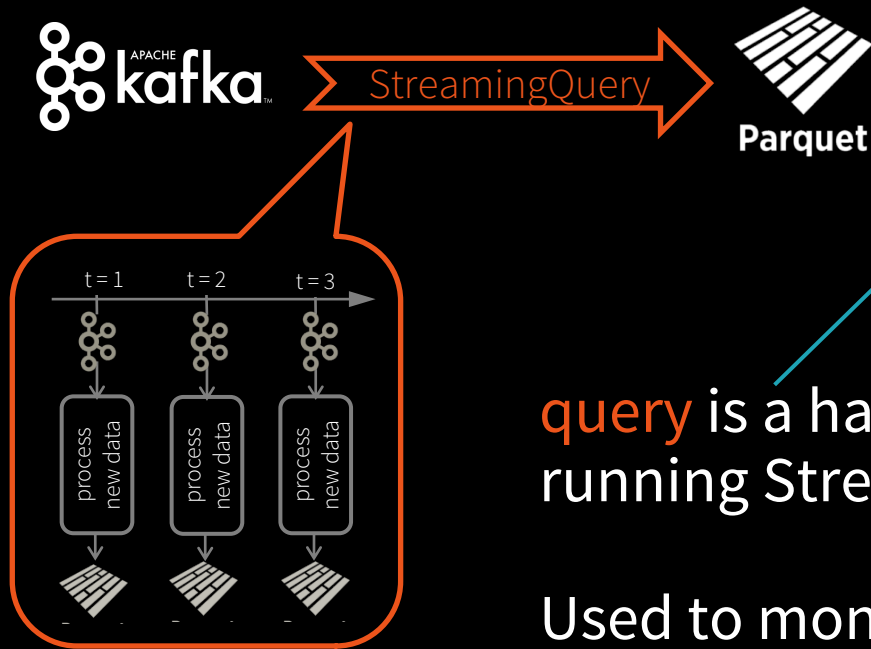
Checkpointing

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 [Spark 2.2]

Write out to Kafka

Dataframe must have binary fields
named key and value

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

Direct, interactive and batch queries on Kafka

Makes Kafka even more powerful
as a storage platform!

```
val df = spark  
  .read           // not readStream  
  .format("kafka")  
  .option("subscribe", "topic")  
  .load()  
  
df.registerTempTable("topicData")  
spark.sql("select value from topicData")
```


Amazon Kinesis [Databricks Runtime 3.0]

Configure with options (similar to Kafka)

How?

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

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

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 operations

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 every 10 mins

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

Support UDAFs!

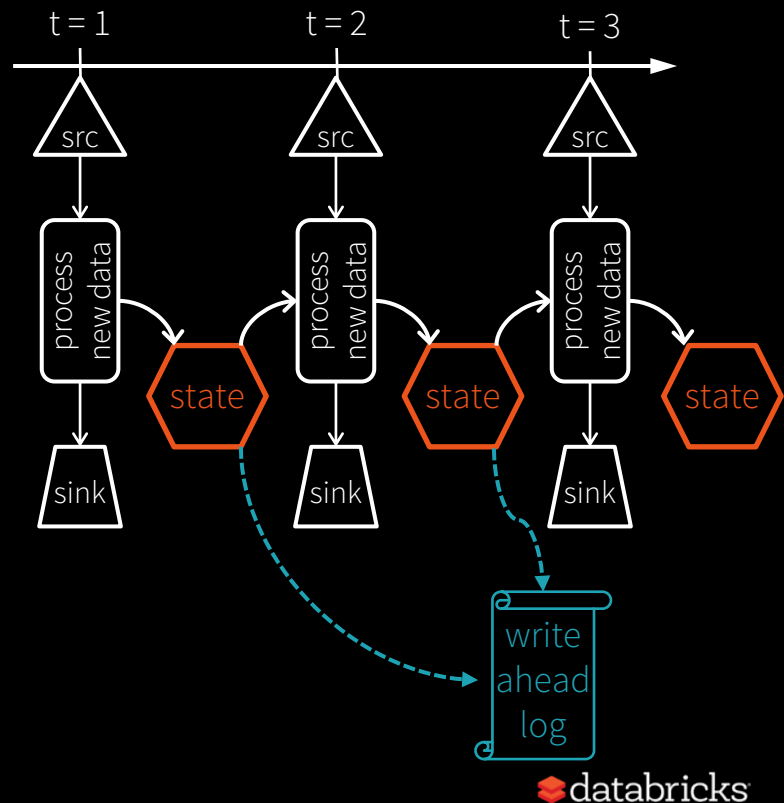
Stateful Processing for Aggregations

Aggregates has to be saved as **distributed state** between triggers

Each trigger reads previous state and writes updated state

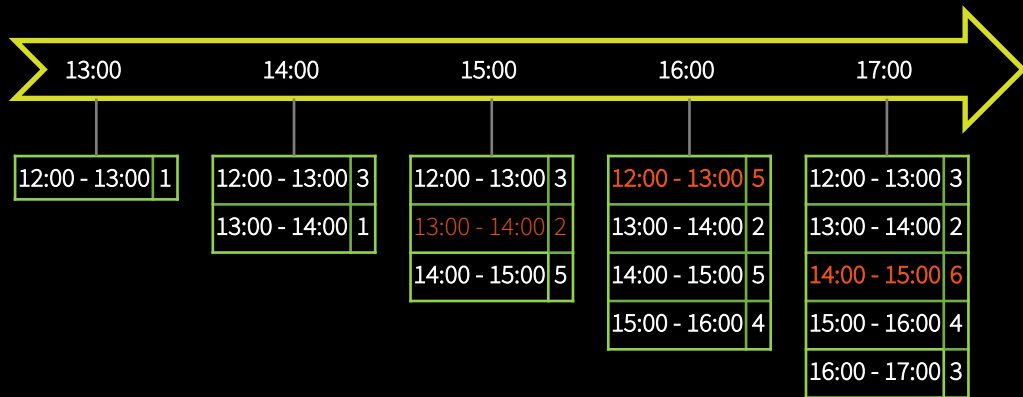
State stored in memory, backed by *write ahead log* in HDFS/S3

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

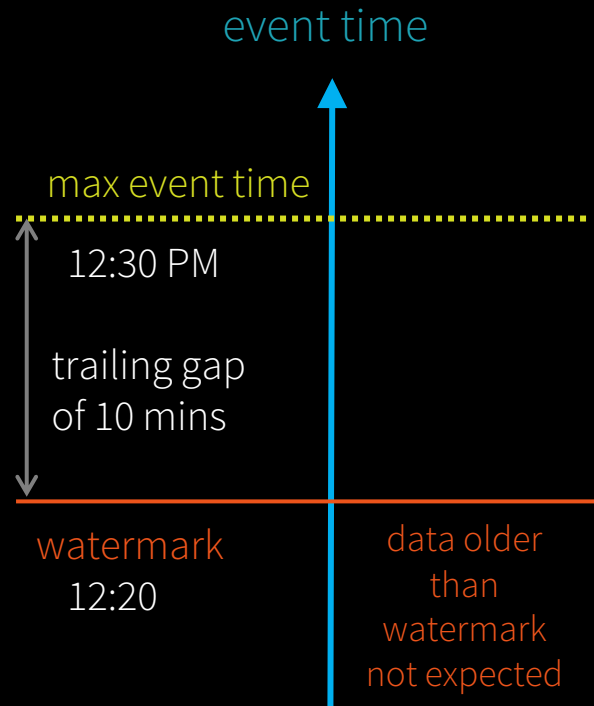
red = state updated with late data

Watermarking

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

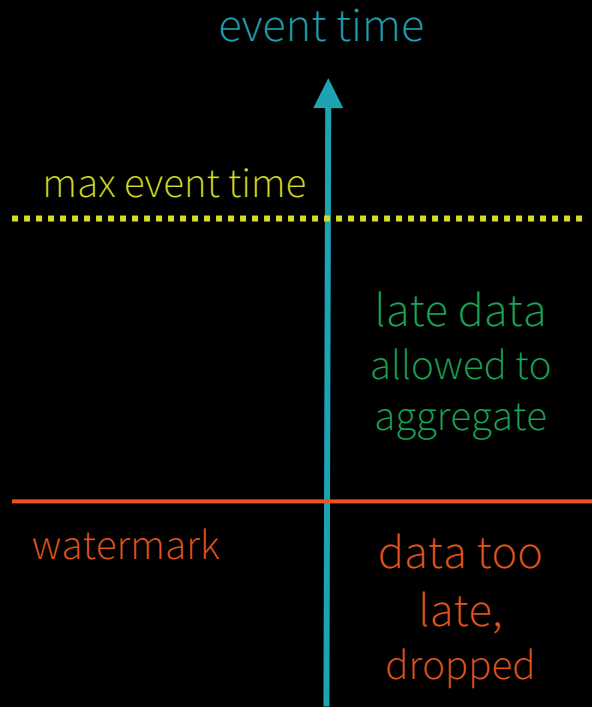


Watermarking

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

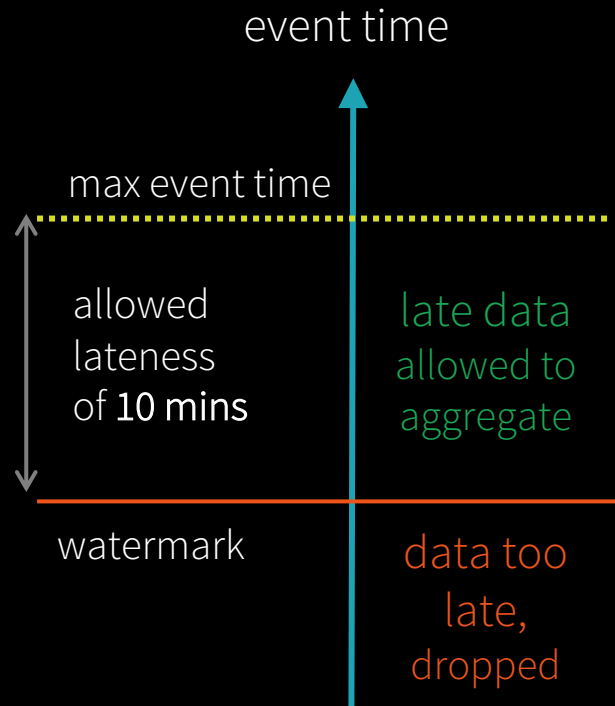
Useful only in stateful operations

(streaming aggs, dropDuplicates, mapGroupsWithState, ...)

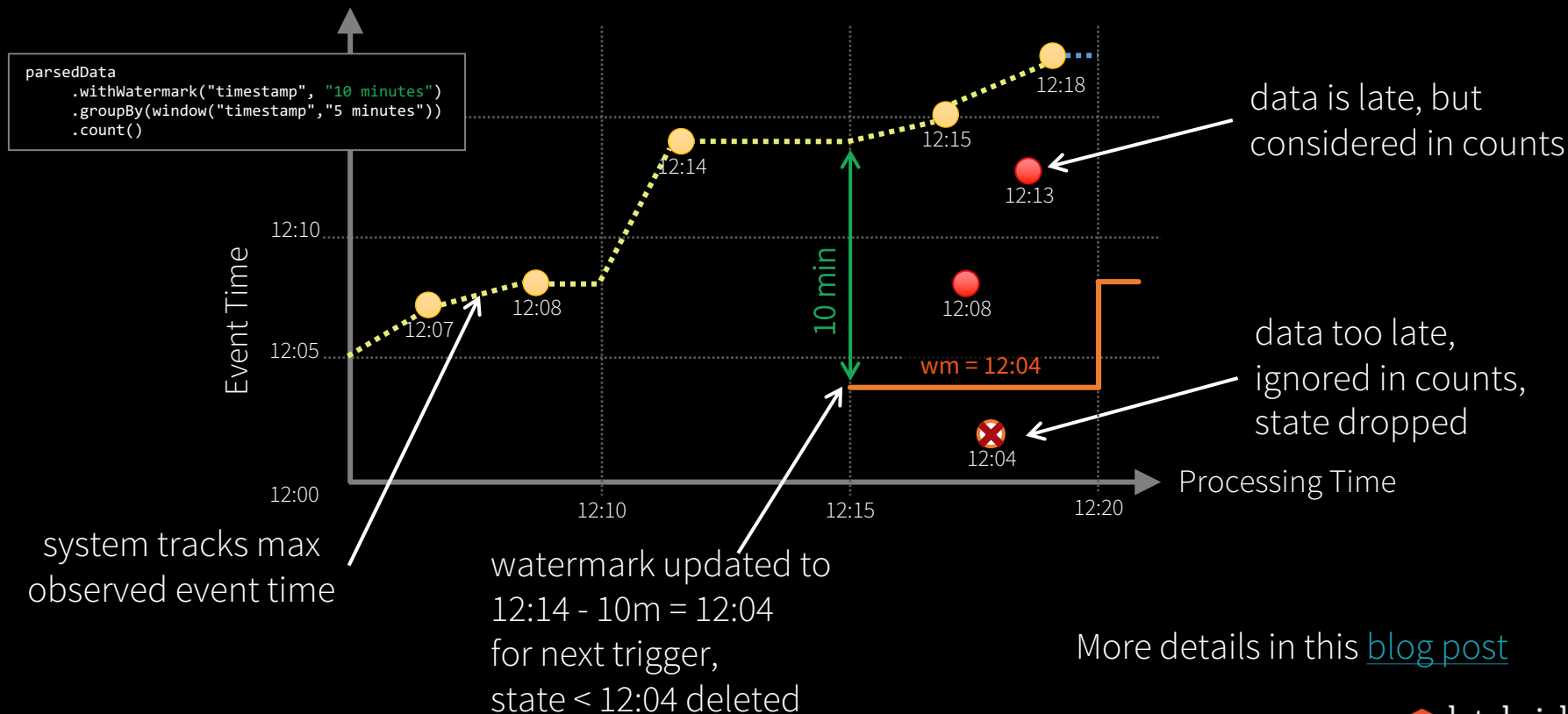
Ignored in non-stateful streaming queries and batch queries

`parsedData`

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



Watermarking



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)

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

Processing Details

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)

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

Processing Details

How late can data be?

How often to emit updates?

Arbitrary Stateful Operations [Spark 2.2]

mapGroupsWithState
applies 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
}
```

Other interesting operations

Streaming Deduplication

Watermarks to limit state

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

Stream-batch Joins

```
val batchData = spark.read  
    .format("parquet")  
    .load("/additional-data")  
parsedData.join(batchData, "device")
```

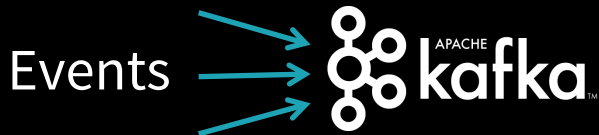
Stream-stream Joins

Can use mapGroupsWithState

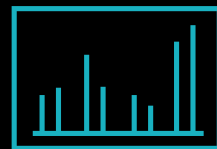
Direct support coming with
Spark 2.3!

ETL @  databricks®

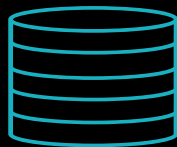
Evolution of a Cutting-Edge Data Pipeline



?



Streaming
Analytics

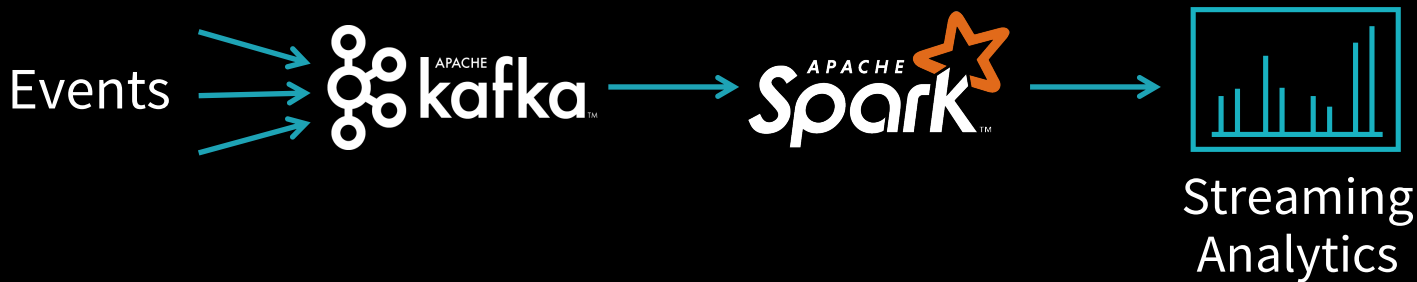


Data Lake



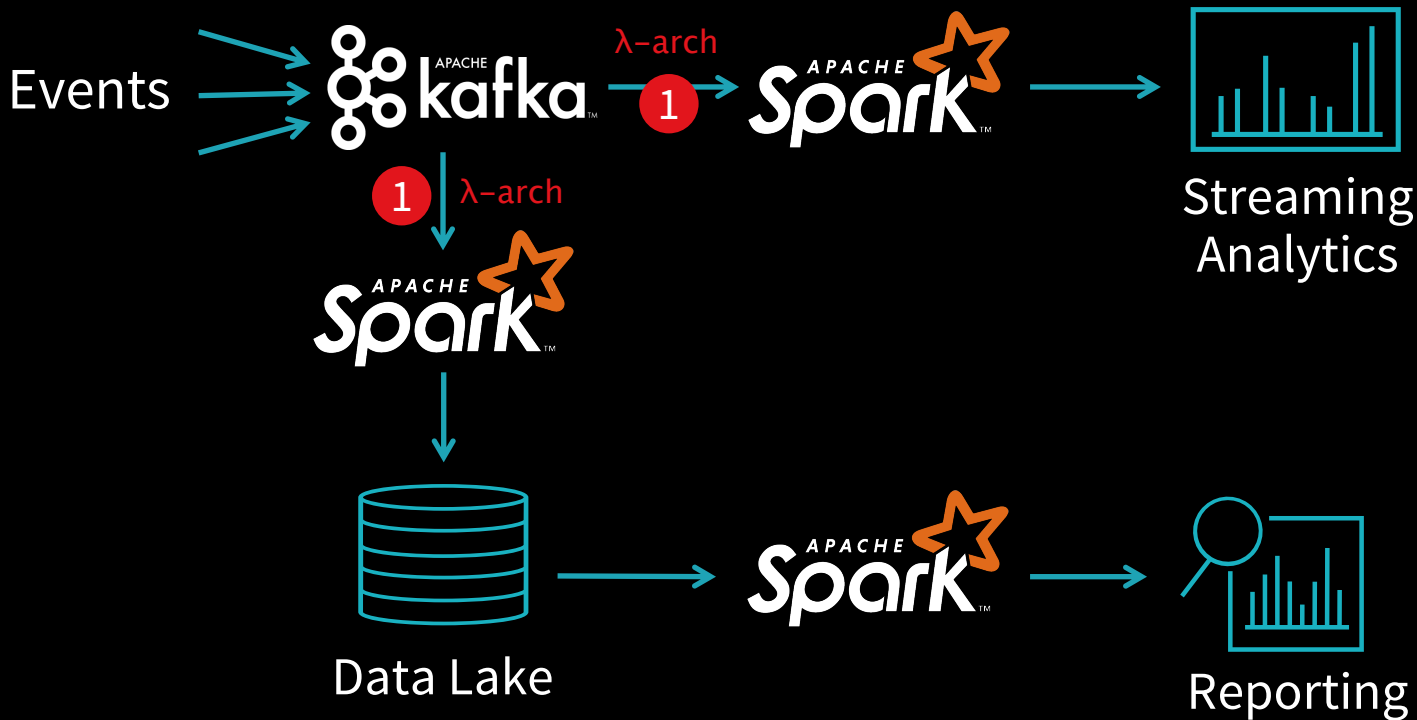
Reporting

Evolution of a Cutting-Edge Data Pipeline

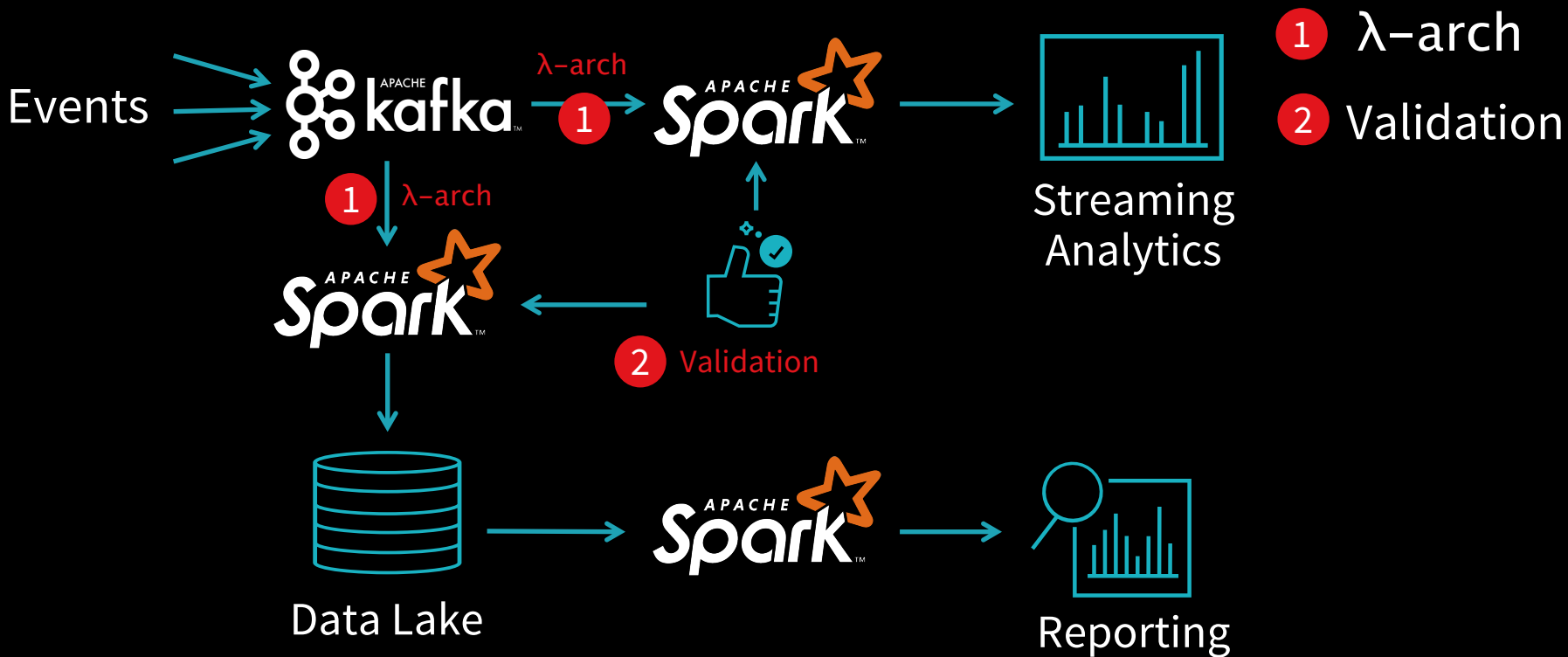


Challenge #1: Historical Queries?

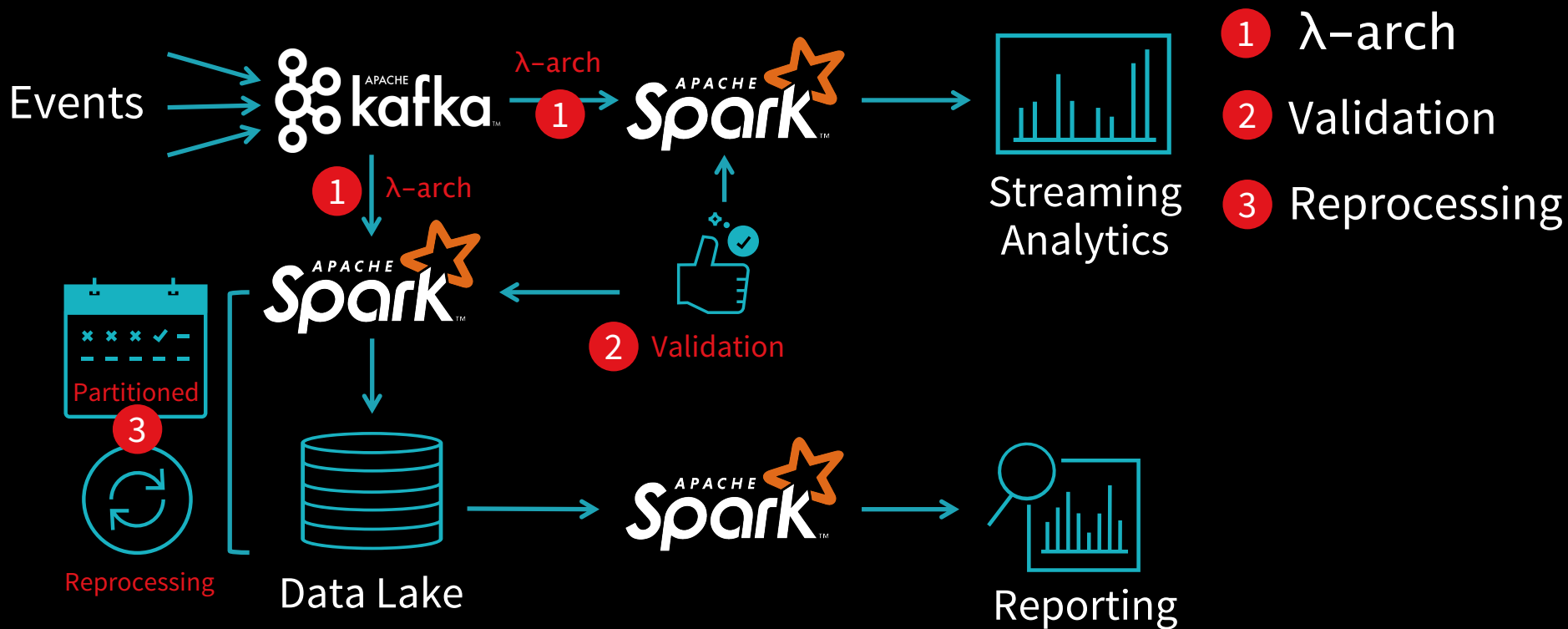
1 λ -arch



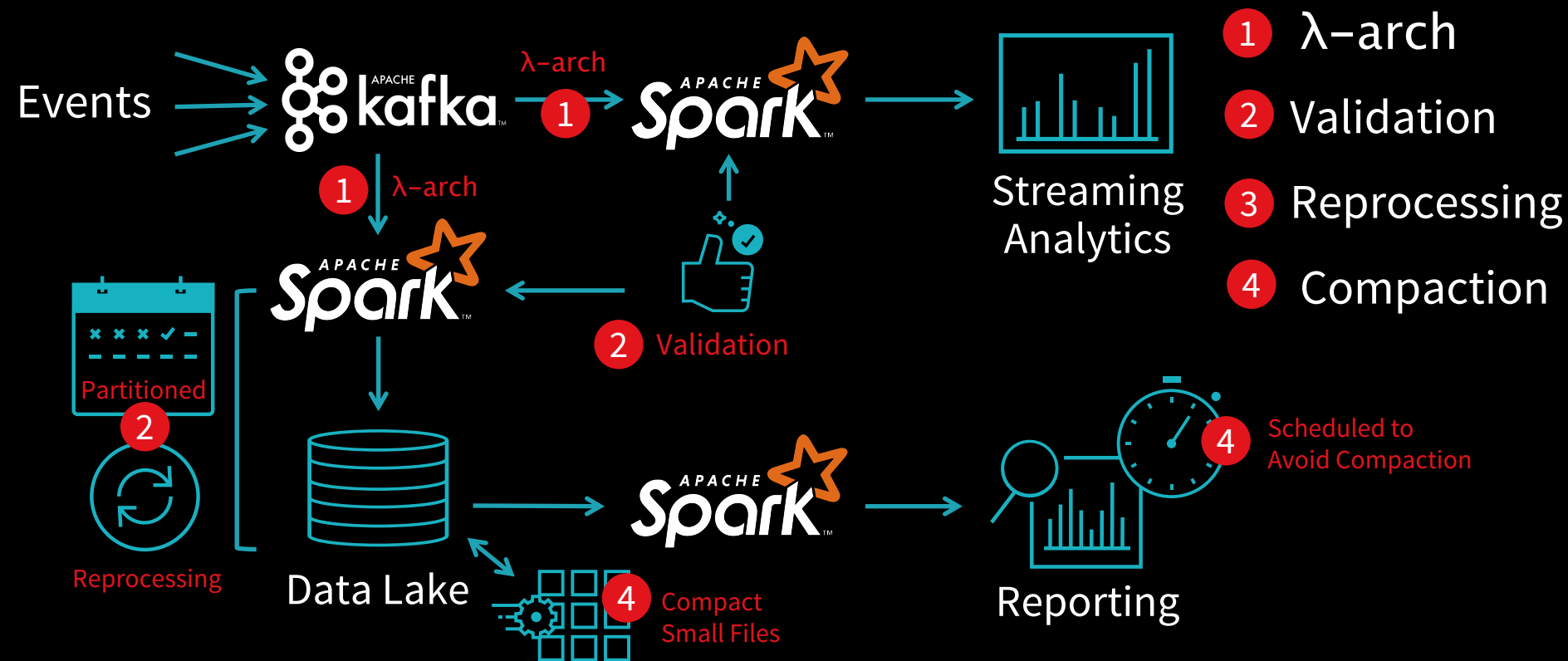
Challenge #2: Messy Data?



Challenge #3: Mistakes and Failures?



Challenge #4: Query Performance?



Databricks Delta

First **UNIFIED** data management system that delivers:

The
SCALE
of data lake

The
**RELIABILITY &
PERFORMANCE**
of data warehouse

The
LOW-LATENCY
of streaming

Databricks Delta

THE GOOD OF DATA LAKES

- Massive scale on Amazon S3
- Open Formats (Parquet, ORC)
- Predictions (ML) & Real Time Streaming

THE GOOD OF DATA WAREHOUSES

- Pristine Data
- Transactional Reliability
- Fast Queries (10-100x)

Enables Predictions, Real-time and Ad Hoc Analytics at Massive Scale

Databricks Delta Under the Hood

MASSIVE SCALE

- Decouple Compute & Storage

RELIABILITY

- ACID Transactions & Data Validation

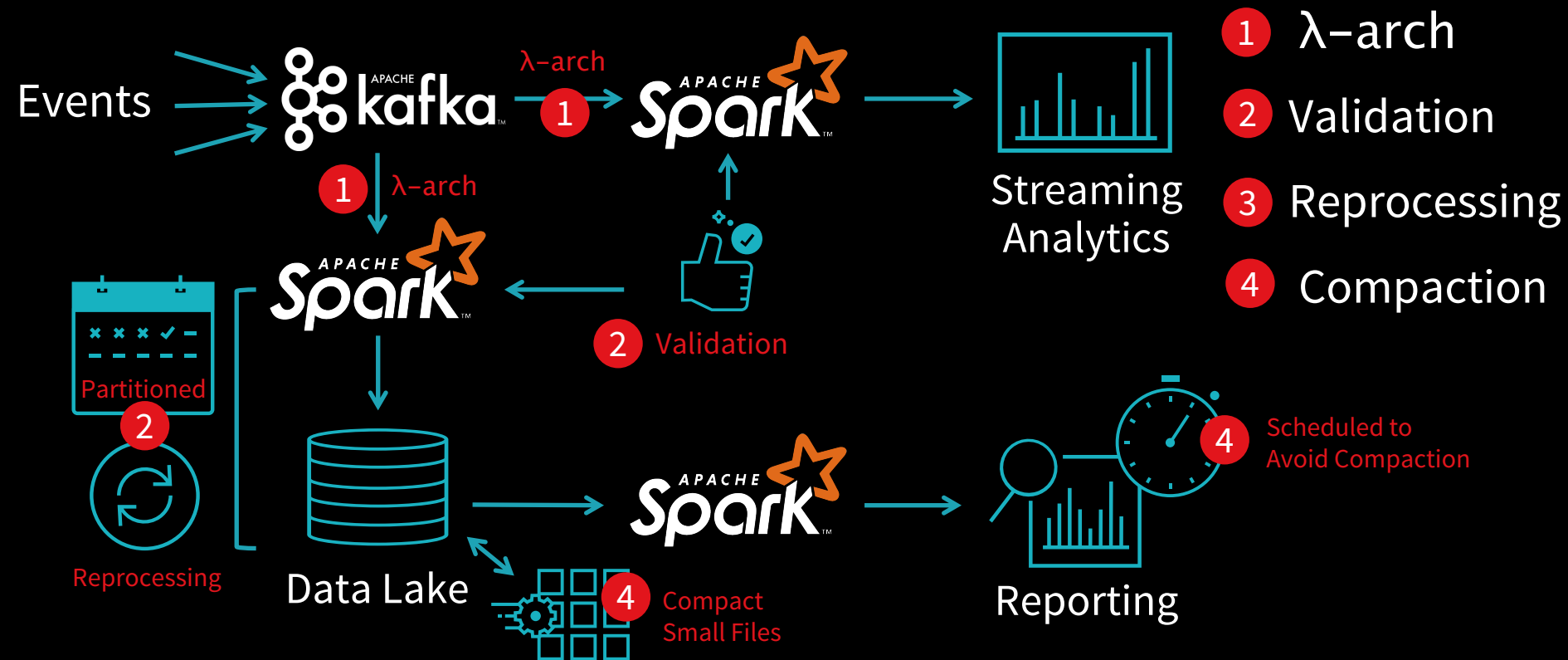
PERFORMANCE

- Data Indexing & Caching (10-100x)

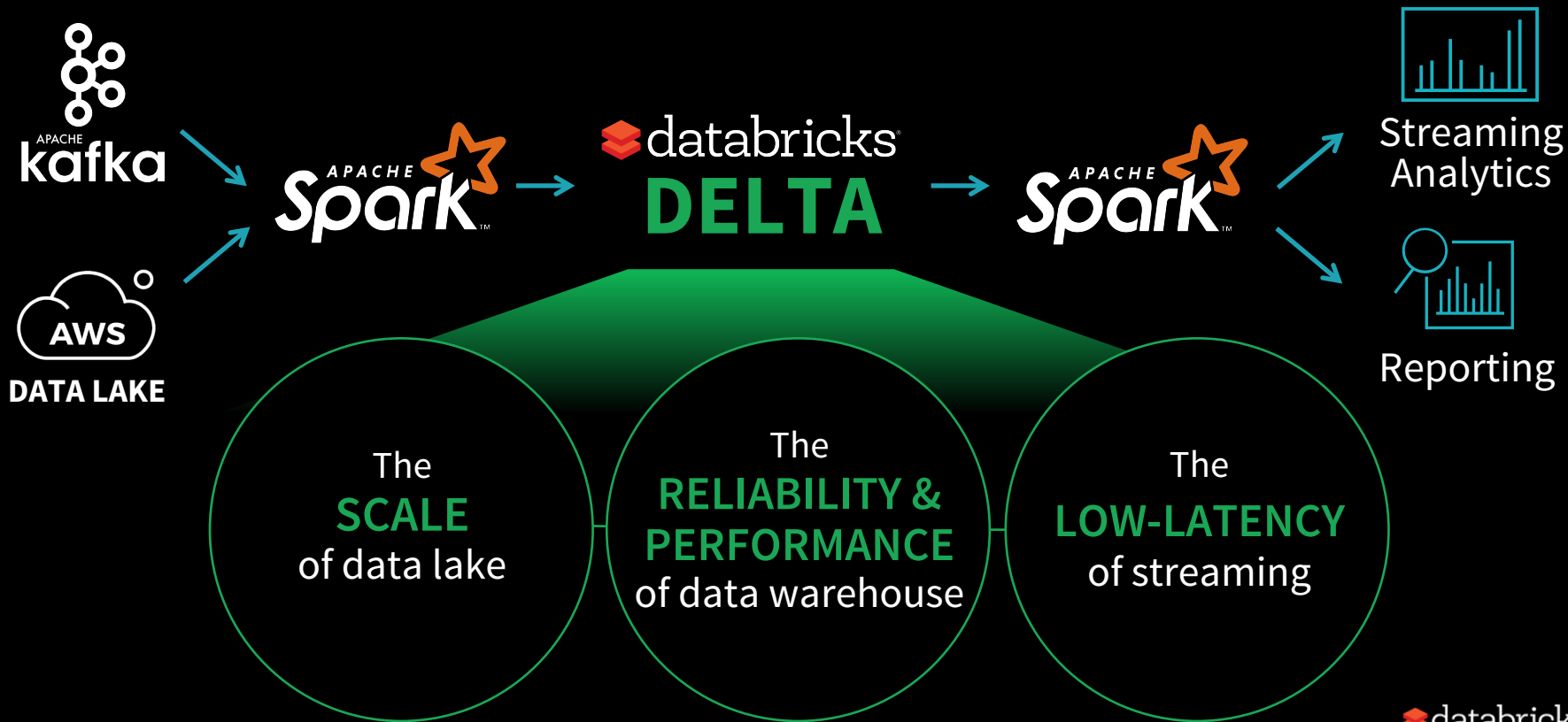
LOW-LATENCY

- Real-Time Streaming Ingest

The Canonical Data Pipeline



The Delta Architecture



Delta @  databricks®

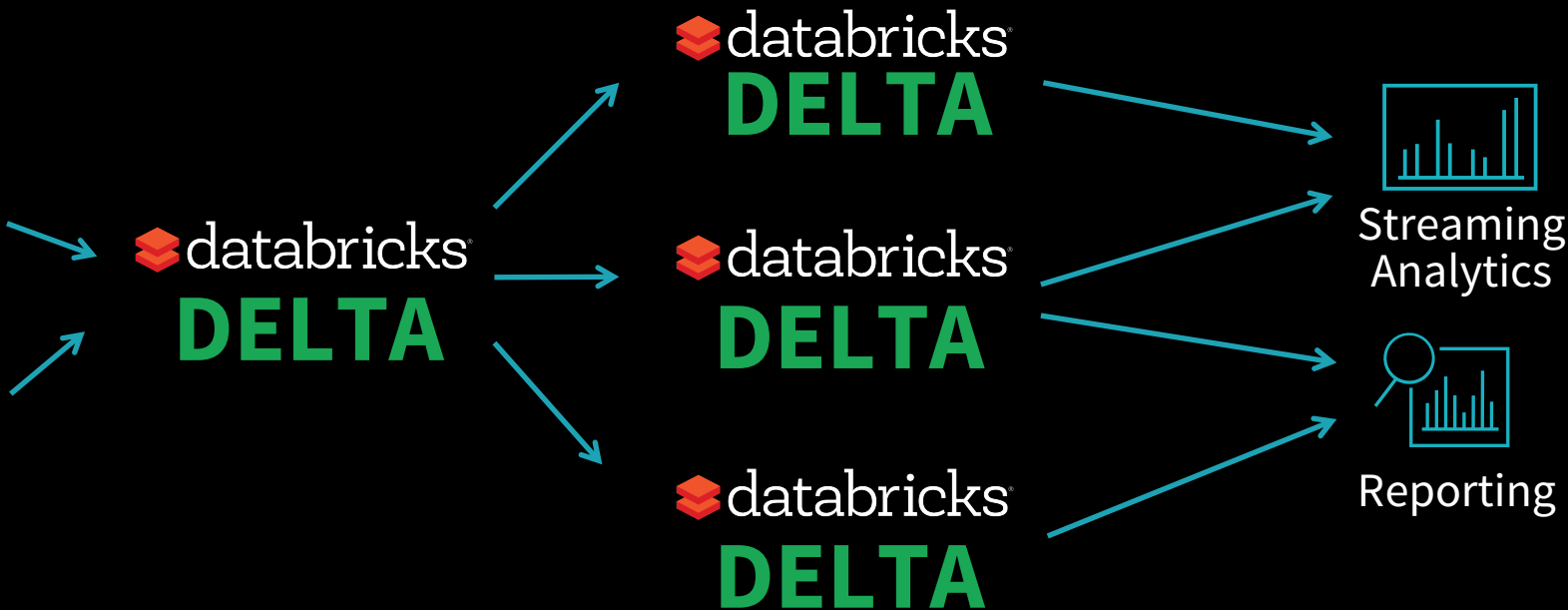
14+ billion records / hour
with 10 nodes

meet diverse latency requirements
as efficiently as possible

Delta @ databricks®

Raw Tables

Summary Tables



Larger Size

Longer Retention

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/2017/08/24/anthology-of-technical-assets-on-apache-sparks-structured-streaming.html>

<https://databricks.com/blog/2017/10/25/databricks-delta-a-unified-management-system-for-real-time-big-data.html>

and more to come, stay tuned!!

Try Apache Spark in Databricks!

UNIFIED ANALYTICS PLATFORM

- Collaborative cloud environment
- Free version (community edition)

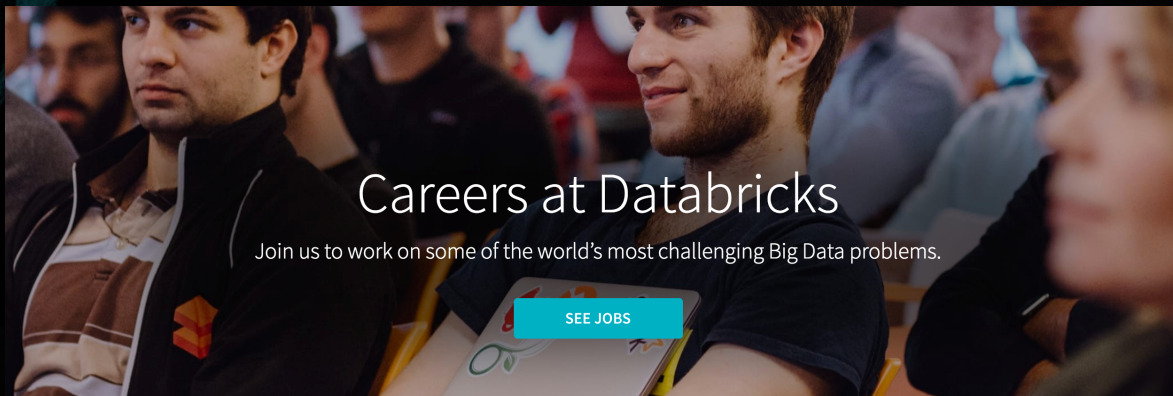
DATABRICKS RUNTIME 3.4

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

Try for free today
databricks.com



We are hiring!



<https://databricks.com/company/careers>





Thank you

“Does anyone have any questions for my answers?”
- Henry Kissinger

burak@databricks.com

