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

Tathagata "*TD*" Das
@tathadas

Big Data Streaming Meetup
Beijing, 2018

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

Event time processing

Combining streaming with interactive queries, machine learning

Complex Systems

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

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

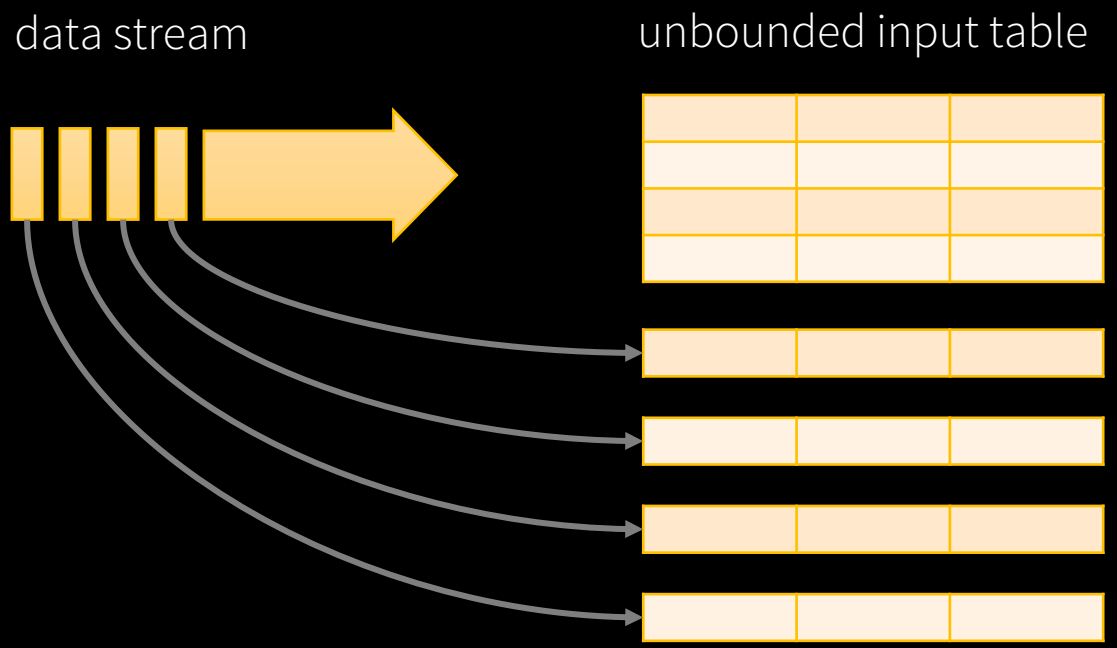
should write simple queries

&

Spark

should continuously update the answer

Treat Streams as Unbounded Tables



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

Anatomy of a Streaming Query

Example

Read JSON data from Kafka

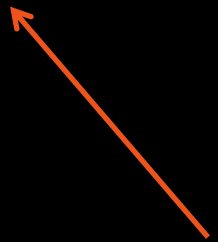
Parse nested JSON

Store in structured Parquet table

Get end-to-end failure guarantees

Anatomy of a Streaming Query

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



returns a DataFrame

Source

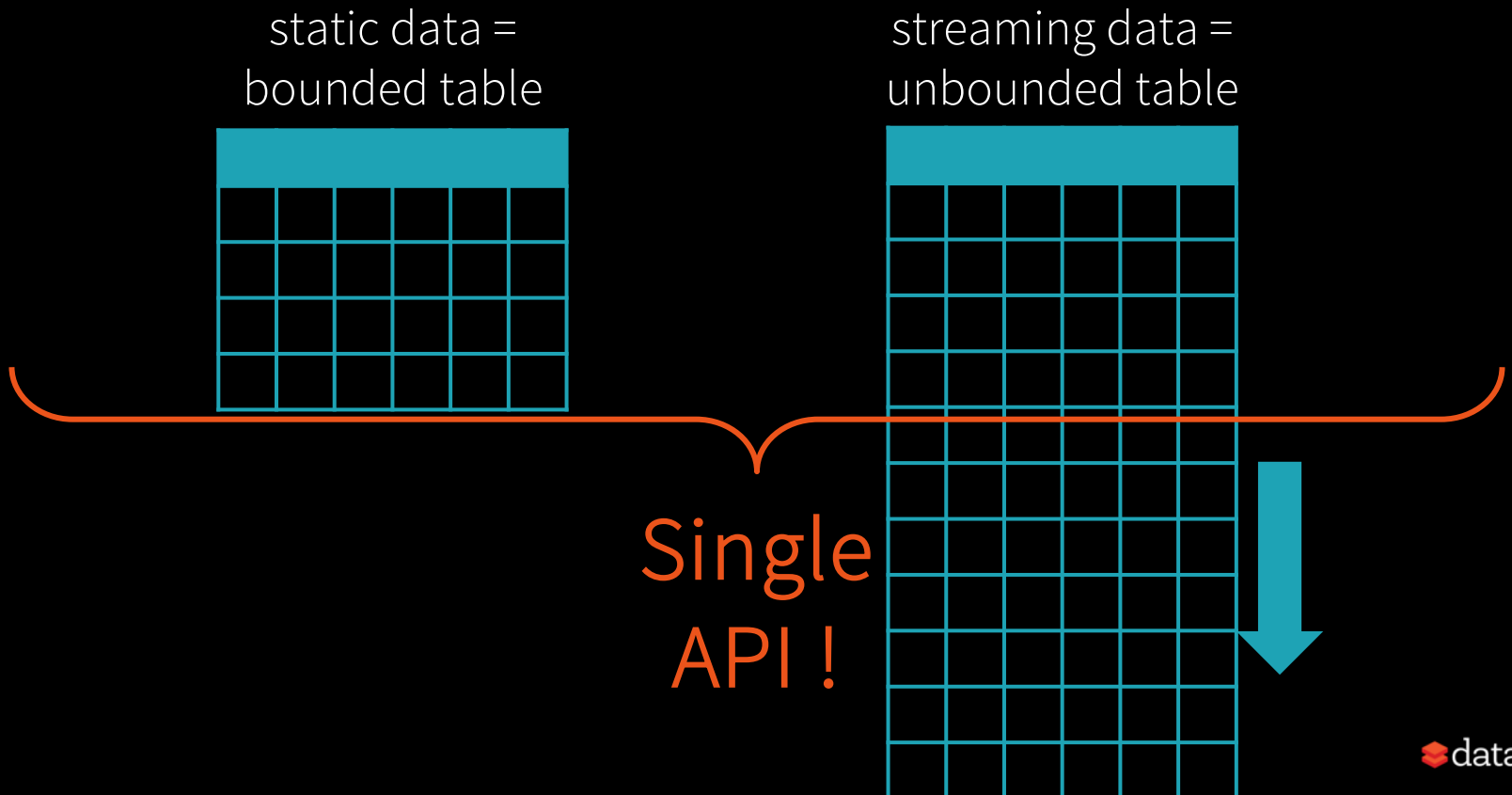
Specify where to read data from

Built-in support for Files / Kafka / Kinesis*

Can include multiple sources of different types using `join()` / `union()`

*Available only on Databricks Runtime

DataFrame ⇔ Table



DataFrame/Dataset

SQL

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

Most familiar to BI Analysts

Supports SQL-2003, HiveQL

DataFrame

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

Great for Data Scientists familiar with Pandas, R Dataframes

Dataset

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

Great for Data Engineers who want compile-time type safety

Same semantics, same performance

Choose your hammer for whatever nail you have!

Anatomy of a Streaming Query

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

Kafka DataFrame

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

Anatomy of a Streaming Query

```
spark.readStream.format("kafka")
  .option("kafka.boostrap.servers",...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .select(from_json("json", schema).as("data"))
```

Transformations

Cast bytes from Kafka records to a string, parse it as a json, and generate nested columns

100s of built-in, optimized SQL functions like `from_json`

user-defined functions, lambdas, function literals with map, flatMap…

Anatomy of a Streaming Query

```
spark.readStream.format("kafka")
  .option("kafka.boostrap.servers",...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .select(from_json("json", schema).as("data"))
  .writeStream
  .format("parquet")
  .option("path", "/parquetTable/")
```

Sink

Write transformed output to external storage systems

Built-in support for Files / Kafka

Use `foreach` to execute arbitrary code with the output data

Some sinks are transactional and exactly once (e.g. files)

Anatomy of a Streaming Query

```
spark.readStream.format("kafka")
  .option("kafka.boostrap.servers",...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .select(from_json("json", schema).as("data"))
  .writeStream
  .format("parquet")
  .option("path", "/parquetTable/")
  .trigger("1 minute")
  .option("checkpointLocation", "…")
  .start()
```

Processing Details

Trigger: when to process data

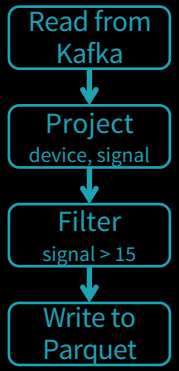
- Fixed interval micro-batches
- As fast as possible micro-batches
- Continuously (new in Spark 2.3)

Checkpoint location: for tracking the progress of the query

Spark automatically streamifies!

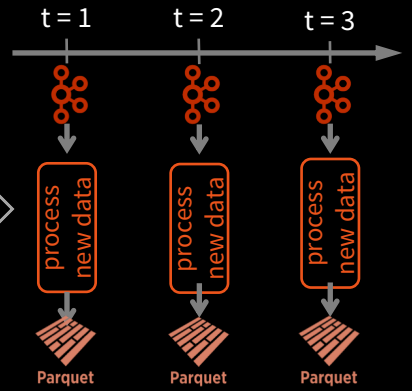
```
spark.readStream.format("kafka")
  .option("kafka.boostrap.servers",...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .select(from_json("json", schema).as("data"))
  .writeStream
  .format("parquet")
  .option("path", "/parquetTable/")
  .trigger("1 minute")
  .option("checkpointLocation", "…")
  .start()
```

DataFrames, Datasets, SQL



Logical Plan

Optimized 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

Fault-tolerance with Checkpointing

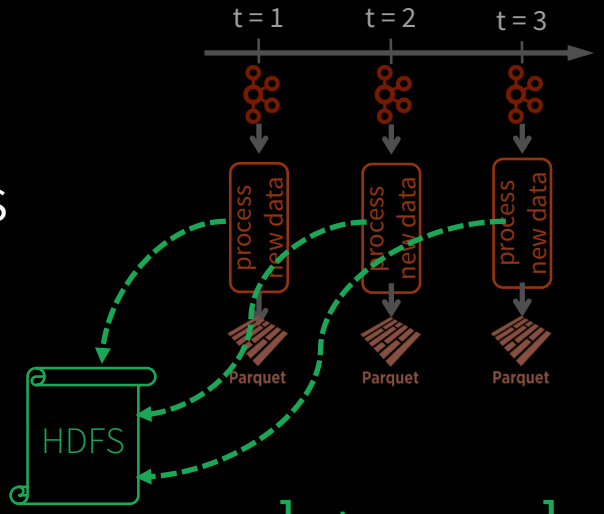
Checkpointing

Saves processed offset info to stable storage like HDFS

Saved as JSON for forward-compatibility

Allows recovery from any failure

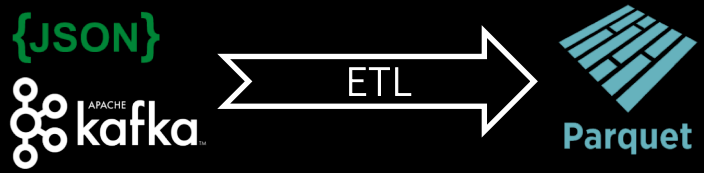
Can resume after limited changes to your streaming transformations (e.g. adding new filters to drop corrupted data, etc.)



end-to-end exactly-once guarantees

Anatomy of a Streaming Query

```
spark.readStream.format("kafka")
  .option("kafka.boostrap.servers",...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .select(from_json("json", schema).as("data"))
  .writeStream
  .format("parquet")
  .option("path", "/parquetTable/")
  .trigger("1 minute")
  .option("checkpointLocation", "…")
  .start()
```

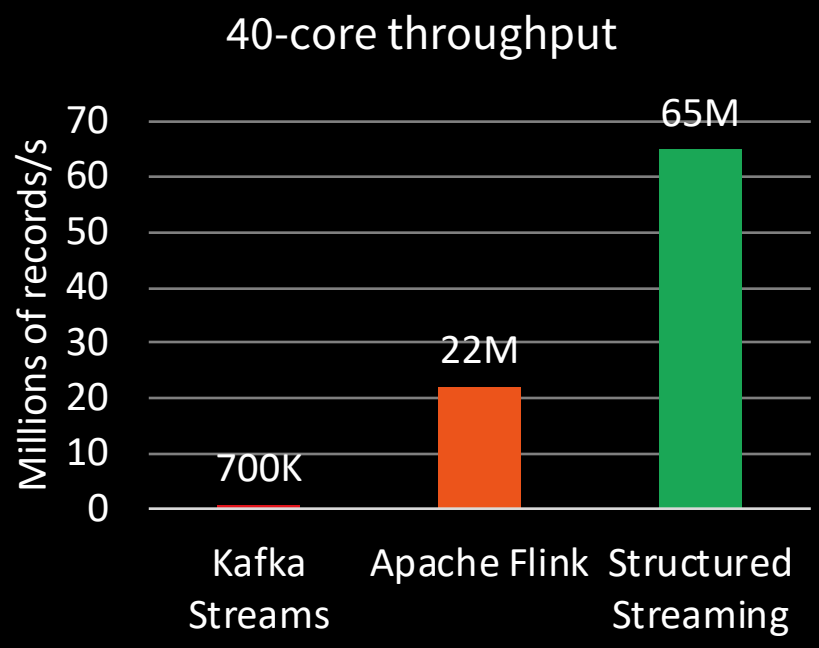


Raw data from Kafka available as structured data in seconds, ready for querying

Performance: YAHOO! Benchmark

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

3x cheaper



More details in our blog post

Business Logic independent of Execution Mode

```
spark.readStream.format("kafka")
  .option("kafka.boostrap.servers",...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .select(from_json("json", schema).as("data"))
  .writeStream
  .format("parquet")
  .option("path", "/parquetTable/")
  .trigger("1 minute")
  .option("checkpointLocation", "…")
  .start()
```

Business logic

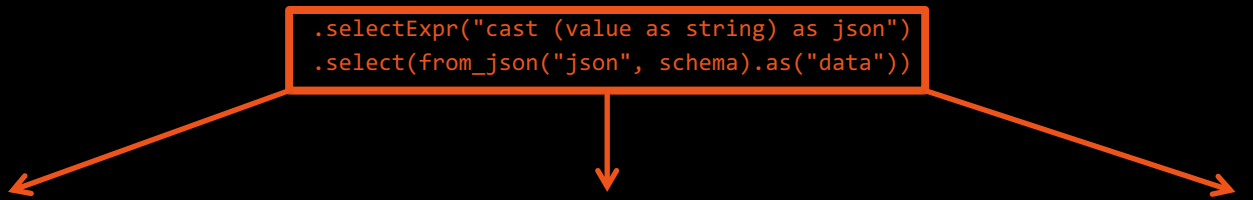
Business Logic independent of Execution Mode

```
spark.read.format("kafka")
  .option("kafka.boostrap.servers",...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .select(from_json("json", schema).as("data"))
  .write
  .format("parquet")
  .option("path", "/parquetTable/")
  .load()
```

Business logic remains unchanged

Peripheral code decides whether it's a batch or a streaming query

Business Logic independent of Execution Mode



Batch

high latency
(hours/minutes)

execute on-demand

high throughput

Micro-batch

Streaming

low latency
(seconds)

efficient resource allocation

high throughput

Continuous**

Streaming

ultra-low latency
(milliseconds)

static resource allocation

**experimental release in Spark 2.3, read our blog

databricks

Working With Time

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!

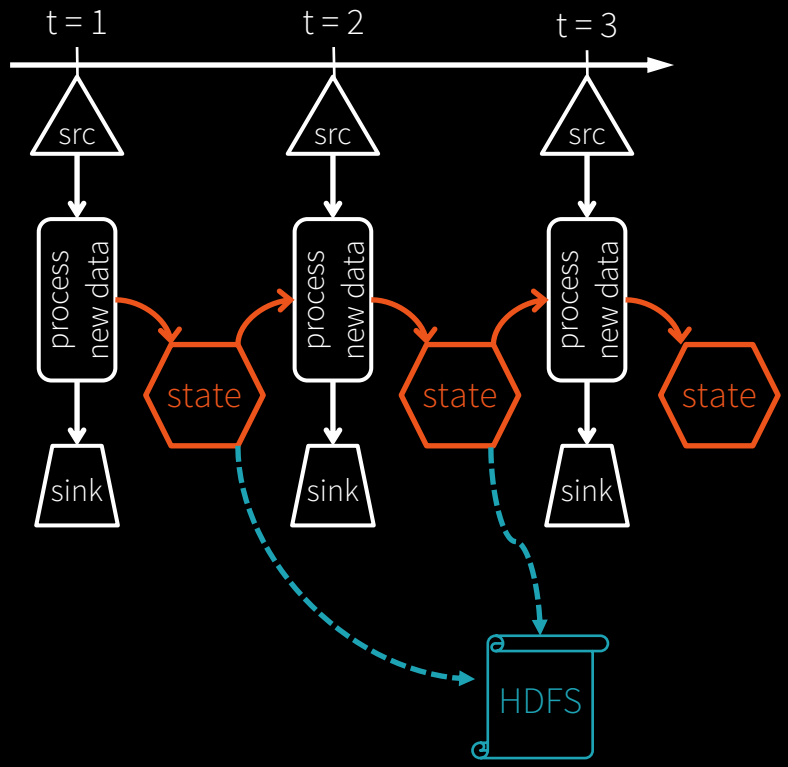
Stateful Processing for Aggregations

Aggregates has to be saved as distributed state between triggers

Each trigger reads previous state and writes updated state

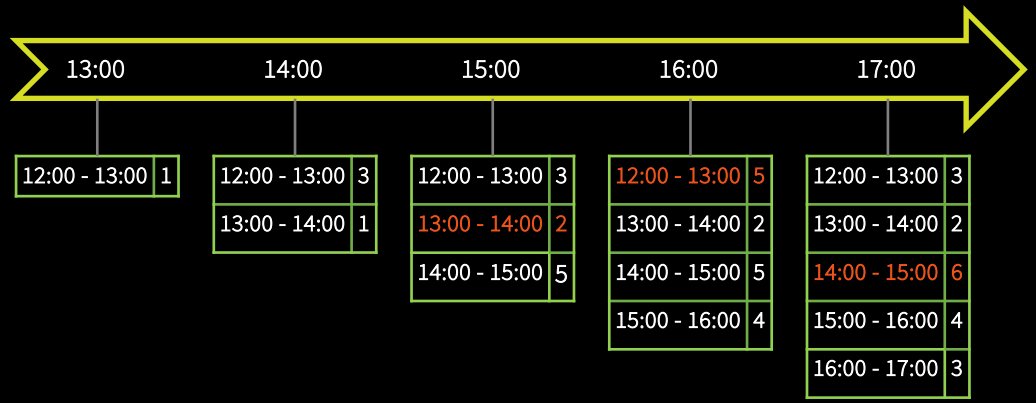
State stored in memory,
backed by *write ahead log* in HDFS

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 event time seen by the engine

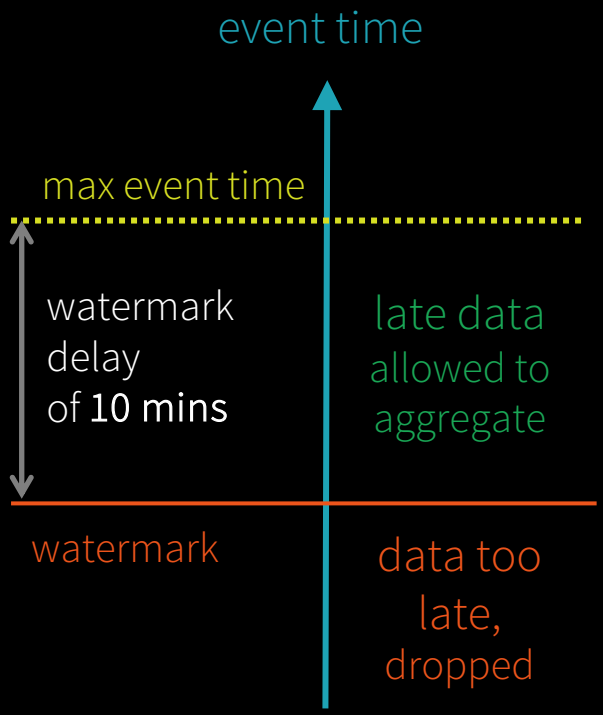
Watermark delay = trailing gap

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

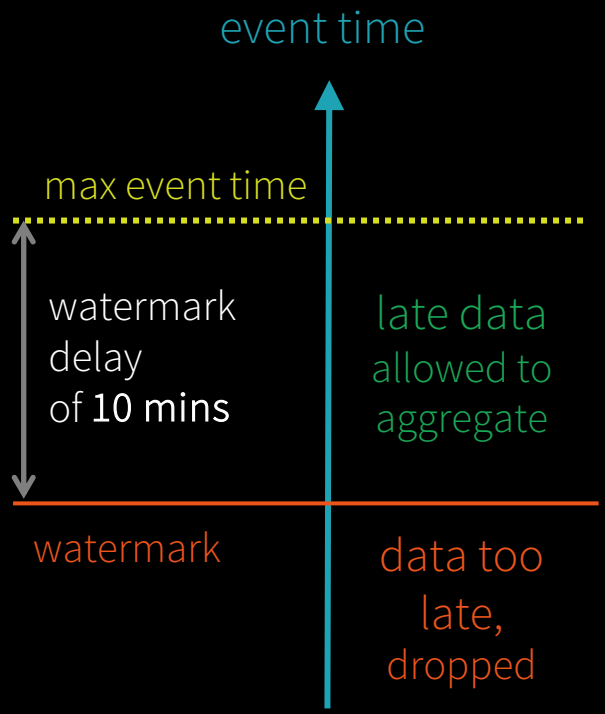


Watermarking

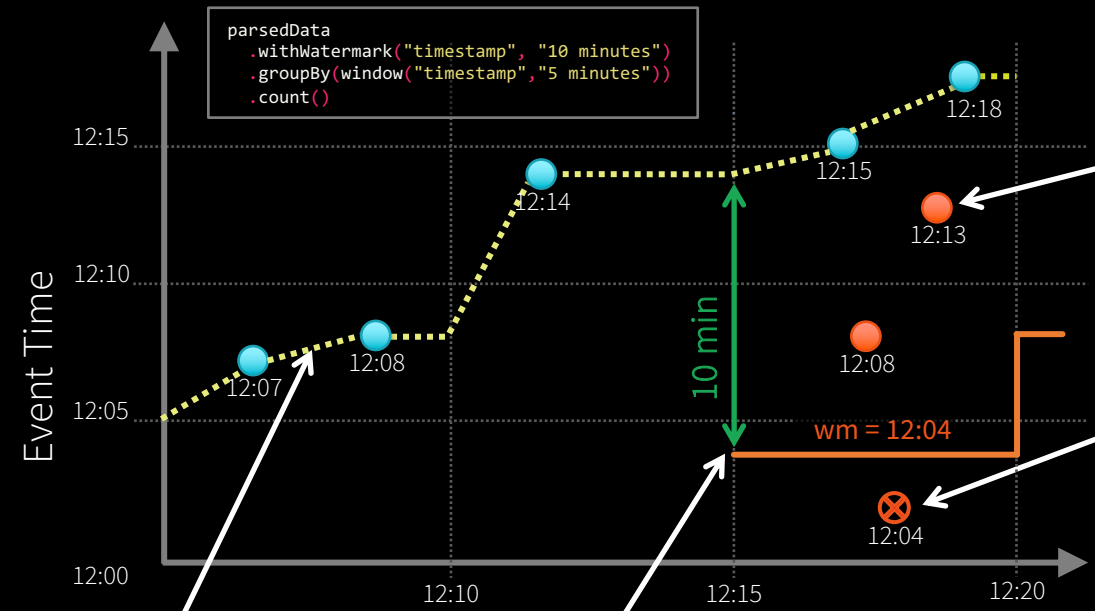
```
parsedData
  .withWatermark("timestamp", "10 minutes")
  .groupBy(window("timestamp","5 minutes"))
  .count()
```

Useful only in stateful operations

Ignored in non-stateful streaming queries and batch queries



Watermarking



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 my blog post

databricks

Arbitrary Stateful Operations

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
}
```

Other interesting operations

Streaming Deduplication

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

Stream-batch Joins

```
eventStream.join(deviceStaticInfo, "deviceId")
```

Stream-stream Joins

```
eventStream.join(userActionStream, "deviceId")
```

More details in my Spark Summit talk
https://databricks.com/session/a-deep-dive-into-stateful-stream-processing-in-structured-streaming

Building Complex Streaming Apps

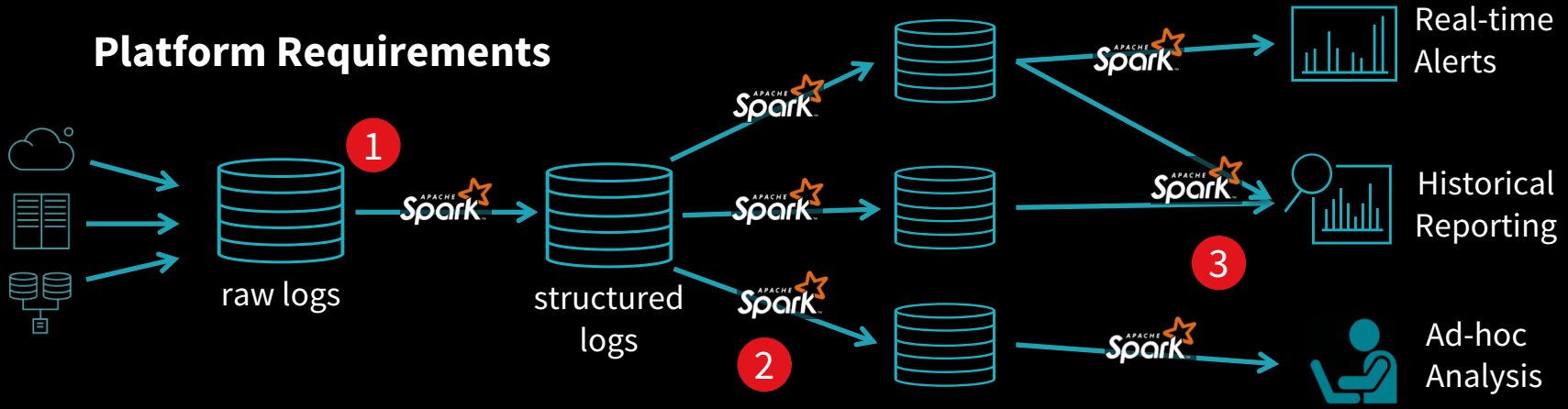
Information Security Platform @ Apple

Infrastructure generates
millions of log events per sec

detect threats!

Information Security Platform @ Apple

Platform Requirements



1. Streaming ETL: raw logs to structured logs
2. Refinement and enrichment: enrich with other information
3. Mixed workloads: real-time alerting, historical reporting, ad-hoc analysis, ML

Solving Ops Challenges @ Apple

1. Streaming ETL: raw logs to structured logs

   Fast failure recovery with adaptive batch sizing

   Large batches to catch up fast, small batches when caught up

2. Refinement and enrichment: enrich with other information

3. Mixed workloads: real-time alerting, historical reporting, ad-hoc analysis, ML

Solving Ops Challenges @ Apple

1. Streaming ETL: raw logs to structured logs
2. Refinement and enrichment: enrich with other information

   Arbitrary stateful operations allow tracking DHCP sessions, etc.

   Stream-stream and stream-batch joins allow joining between various fast and slow data with clear semantics

3. Mixed workloads: real-time alerting, historical reporting, ad-hoc analysis, ML

Solving Ops Challenges @ Apple

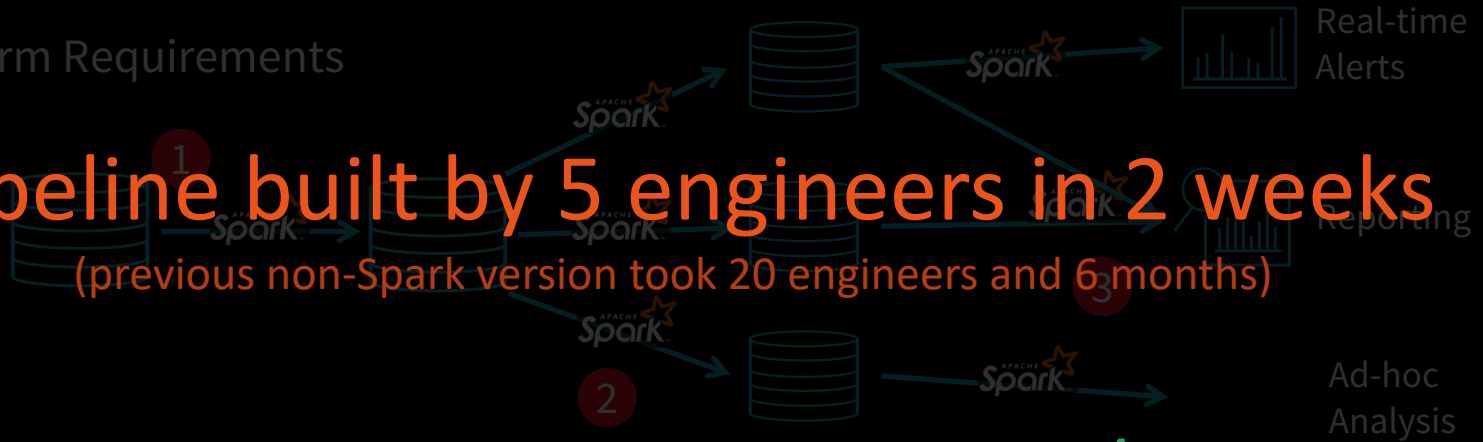
1. Streaming ETL: raw logs to structured logs
2. Refinement and enrichment: enrich with other information
3. Mixed workloads: real-time, historical reports, ad-hoc, ML

Same APIs allows shared codebase for all analysis, faster deployment

E.g. New threat patterns found in interactive analysis can be immediately fined tunes on historical data and applied on real-time alerting application

Information Security Platform @ Apple

Platform Requirements



Pipeline built by 5 engineers in 2 weeks

(previous non-Spark version took 20 engineers and 6 months)

Processing millions events/sec,
few TBs/day

1. Streaming ETL: raw logs to structured logs
2. Refinement and enrichment: enrich with other information
3. Mixed workloads: real-time alerting, historical reporting, ad-hoc analysis

More Info

Structured Streaming Programming Guide

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

Databricks blog posts for more focused discussions on streaming

https://databricks.com/blog/category/engineering/streaming

Databricks Delta

https://databricks.com/product/databricks-delta

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