Making Structured Streaming Ready For Production

Tathagata "*TD*" Das
@tathadas

Spark Summit East
8th February 2017

databricks™

About Me

Spark PMC Member

Built Spark Streaming in UC Berkeley

Currently focused on Structured Streaming

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

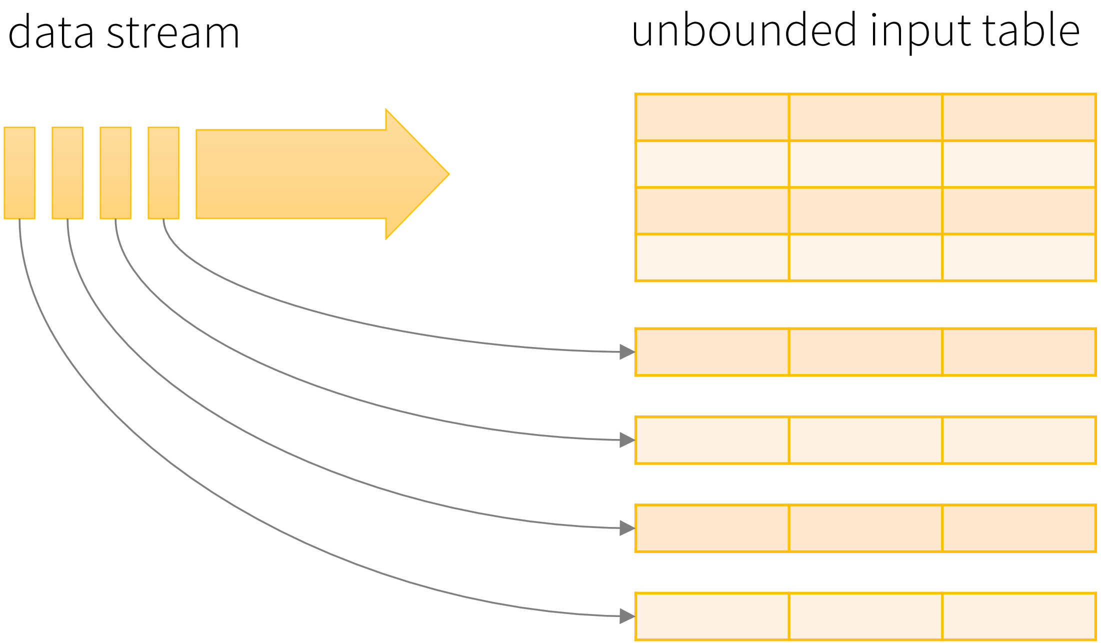
rich, unified, high level APIs
deal with *complex data* and *complex workloads*

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

Philosophy

the simplest way to perform stream processing is *not having to reason* about streaming at all

Treat Streams as Unbounded Tables



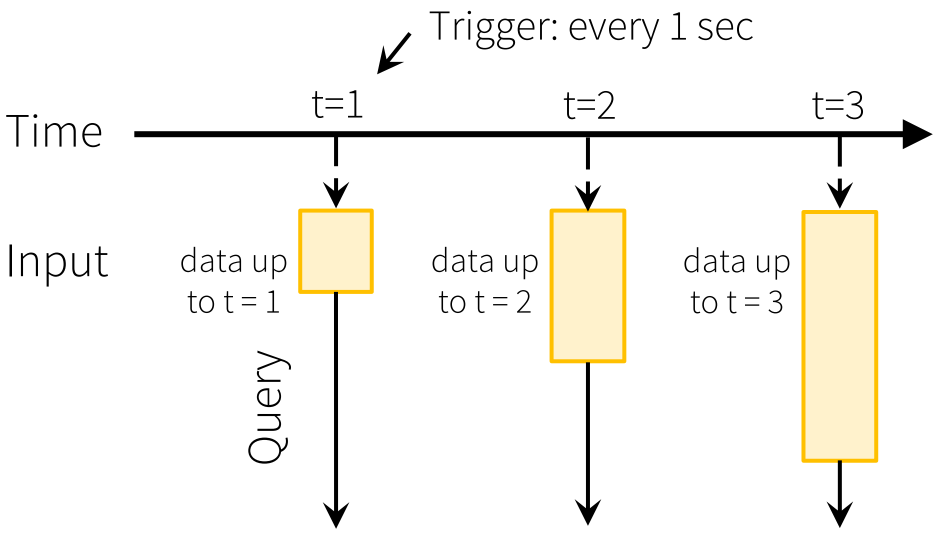
new data in the data stream
=
new rows appended to a unbounded table

New Model

Input: data from source as an append-only table

Trigger: how frequently to check input for new data

Query: operations on input
usual map/filter/reduce
new window, session ops

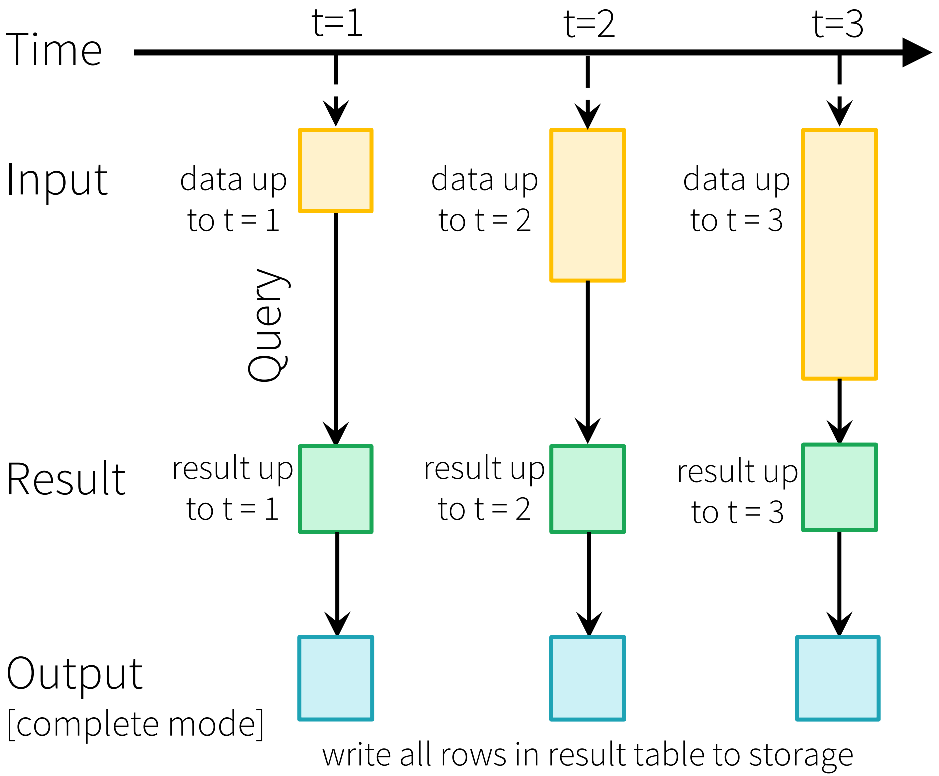


New Model

Result: final operated table
updated after every trigger

Output: what part of result to write
to storage after every trigger

Complete output: write full result table every time



New Model

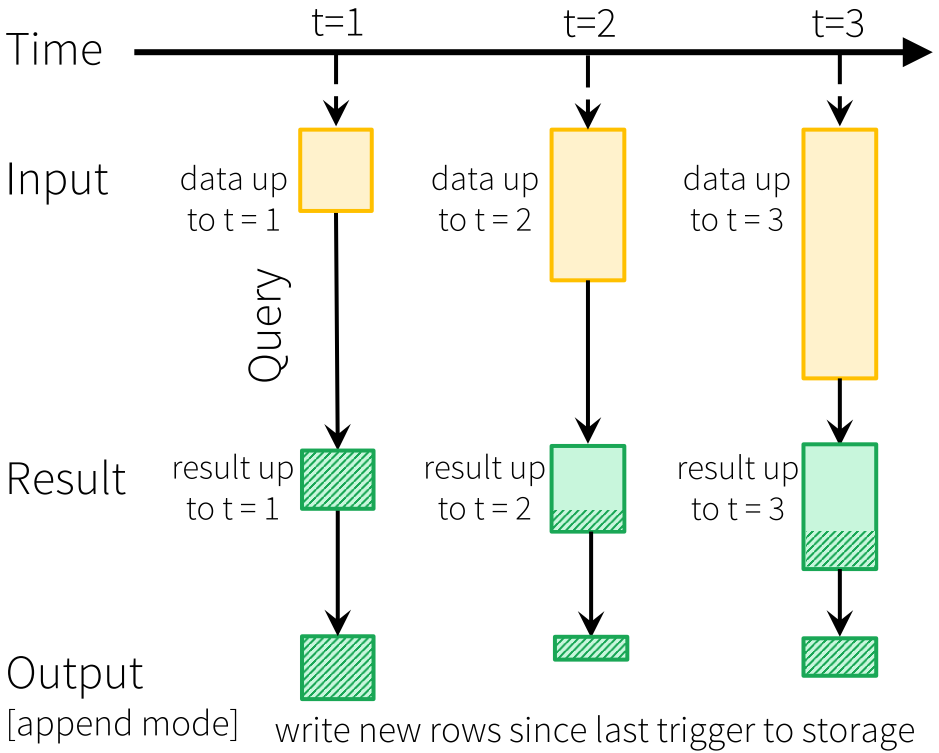
Result: final operated table updated after every trigger

Output: what part of result to write to storage after every trigger

Complete output: write full result table every time

Append output: write only new rows that got added to result table since previous batch

*Not all output modes are feasible with all queries

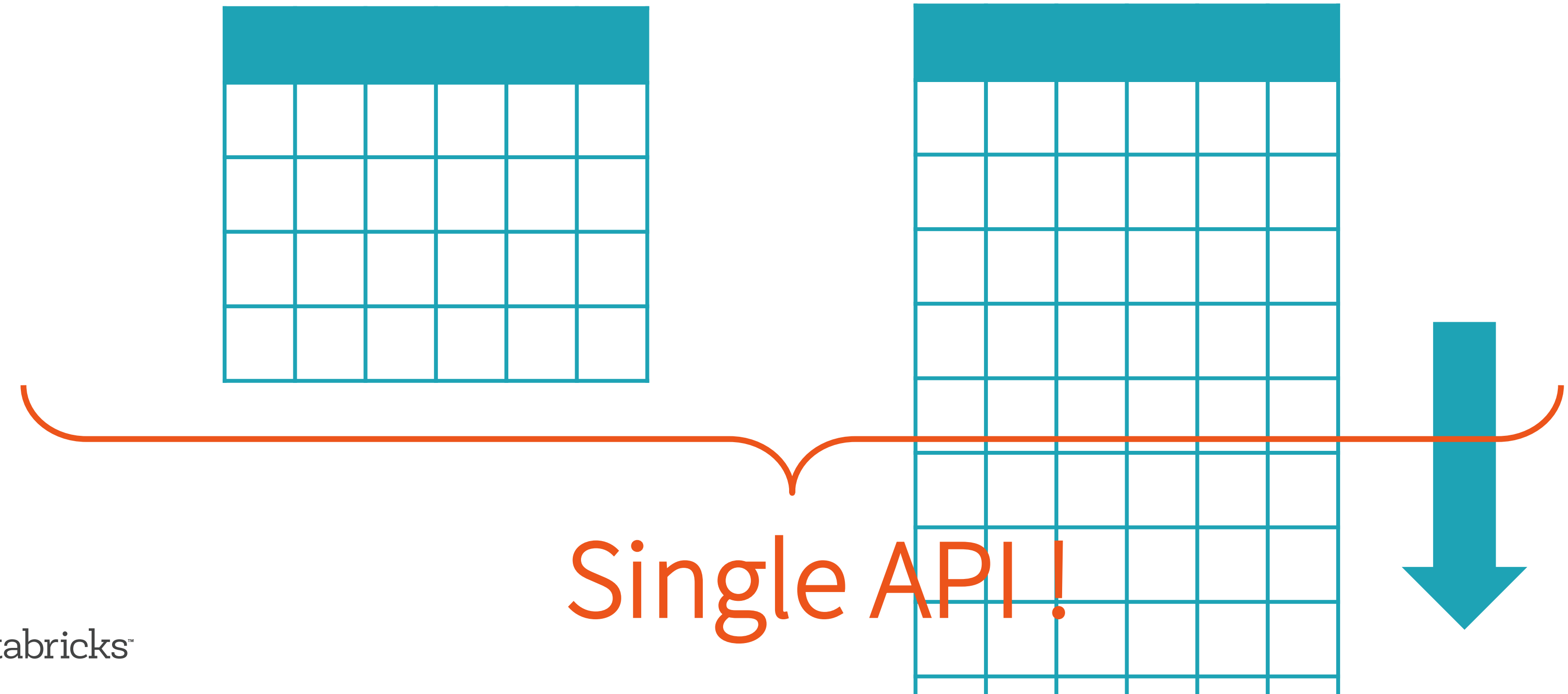


databricks

API - *Dataset/DataFrame*

static data =
bounded table

streaming data =
unbounded table



Single API !

Batch Queries with DataFrames

```
input = spark.read
      .format("json")
      .load("source-path")
```

Read from Json file

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

Select some devices

```
result.write
      .format("parquet")
      .save("dest-path")
```

Write to parquet file

Streaming Queries with DataFrames

```
input = spark.readStream
     .format("json")
     .load("source-path")
```

Read from Json file stream

Replace read with readStream

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

Select some devices

Code does not change

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

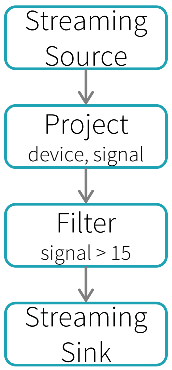
Write to Parquet file stream

Replace save() with start()

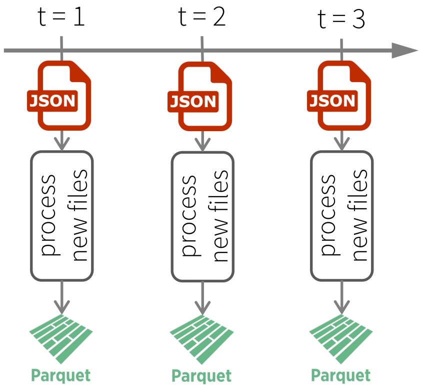
Streaming Query Execution

```
input = spark.readStream
  .format("json")
  .load("source-path")

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

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

DataFrames,
Datasets, SQL



Logical Plan



Series of Incremental
Execution Plans

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

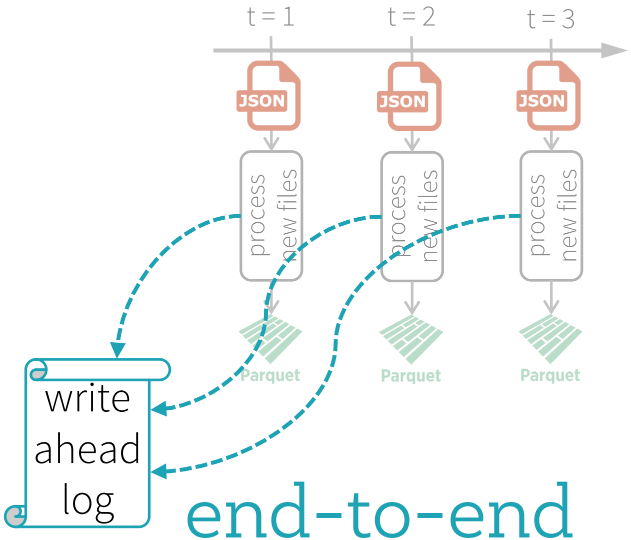
Fault-tolerance with Checkpointing

Checkpointing - metadata (e.g. offsets) of current batch stored in a *write ahead log* in HDFS/S3

Query can be restarted from the log

Streaming sources can replay the exact data range in case of failure

Streaming sinks can dedup reprocessed data when writing, idempotent by design



end-to-end exactly-once guarantees

databricks

Complex Streaming ETL

Traditional ETL

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

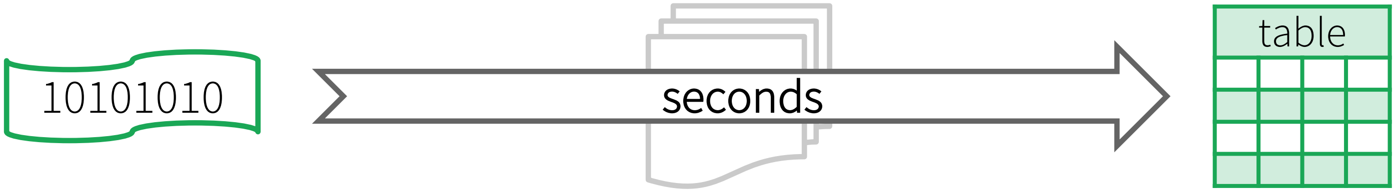
Periodic jobs run every few hours to convert raw data to structured data ready for further analytics

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("subscribe", "topic")
  .option("kafka.boostrap.servers",...)
  .load()

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

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

Reading from Kafka [Spark 2.1]

Support Kafka 0.10.0.1
Specify options to configure

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

How?

```
kafka.boostrap.servers => broker1
```

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} }
```

databricks

Reading from Kafka

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

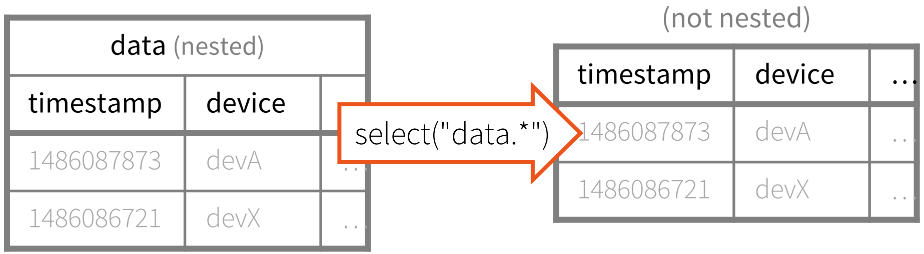
Transforming Data

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

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

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

Flatten the nested columns



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").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

Writing to Parquet table

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")
```

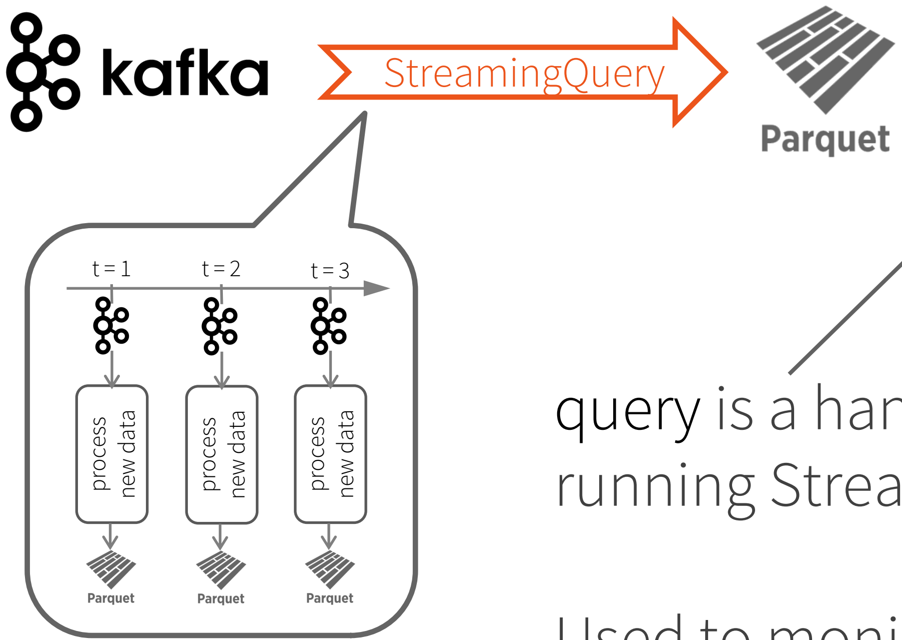
Checkpointing

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

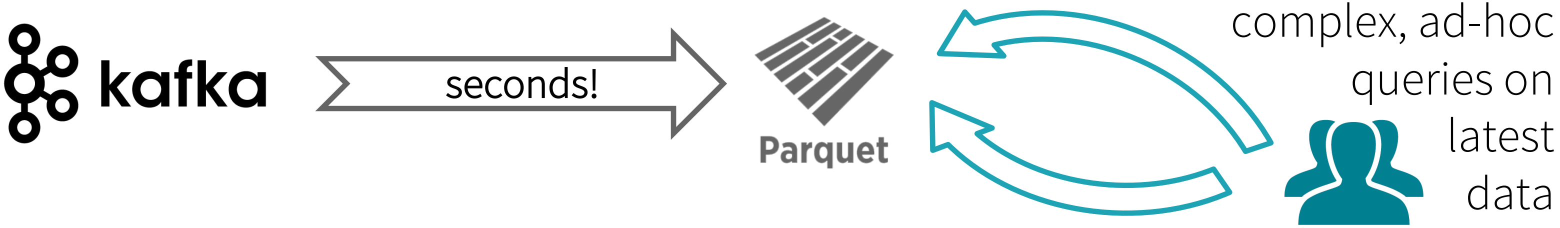



```
val query = parsedData.writeStream
    .option("checkpointLocation", ...)
    .format("parquet")
    .partitionBy("date")
    .start("/parquetTable/")
```

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

Advanced Streaming Analytics

Event time Aggregations

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!

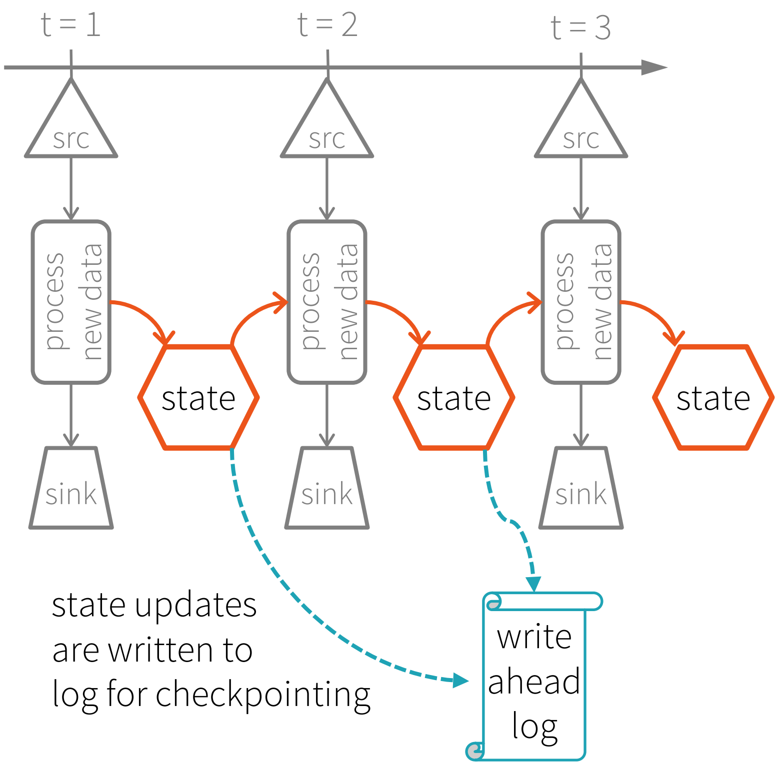
Stateful Processing for Aggregations

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

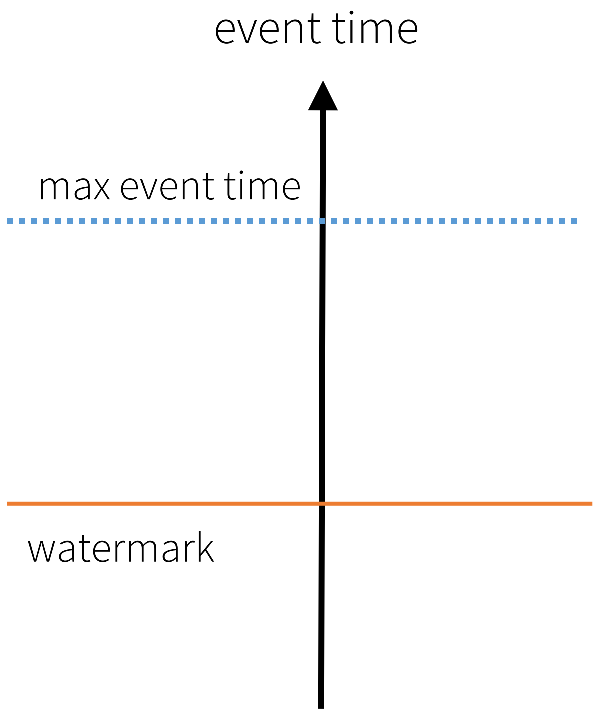
Fault-tolerant, exactly-once guarantee!



Watermarking and Late Data

Watermark [Spark 2.1]
boundary in event time trailing
behind max observed event time

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

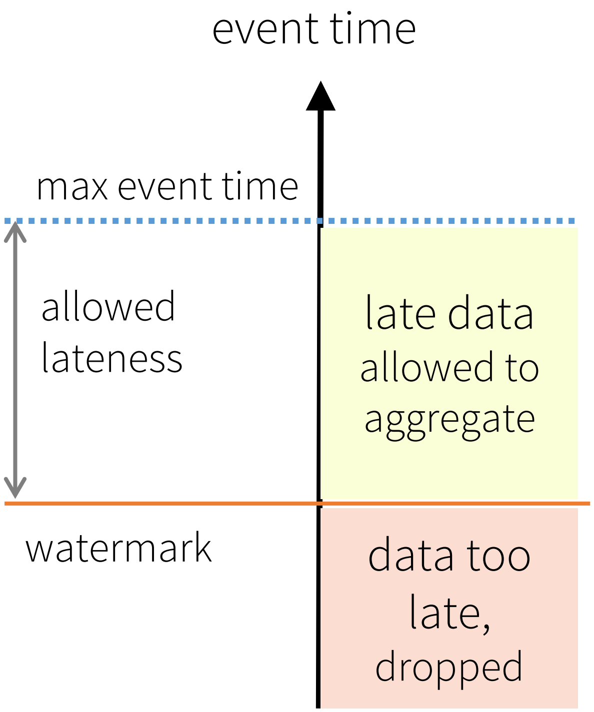


Watermarking and Late Data

Gap is a configurable allowed lateness

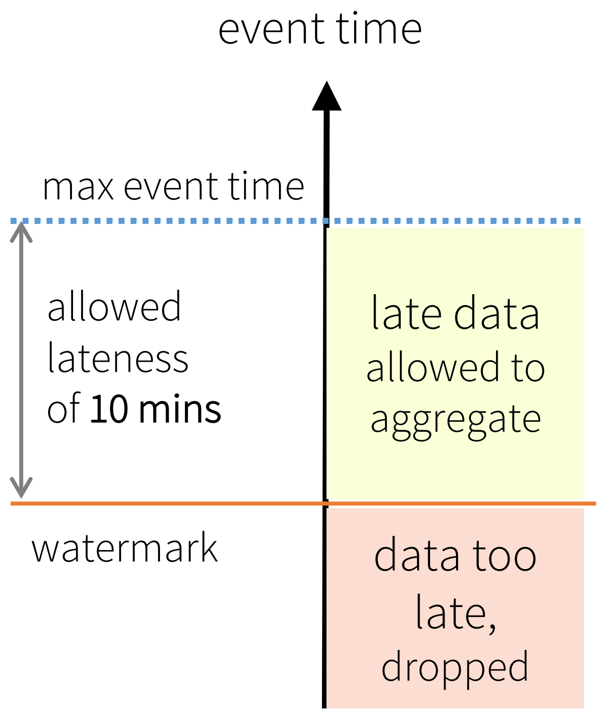
Data newer than watermark may be late, but allowed to aggregate

Data older than watermark is "too late" and dropped, state removed



Watermarking and Late Data

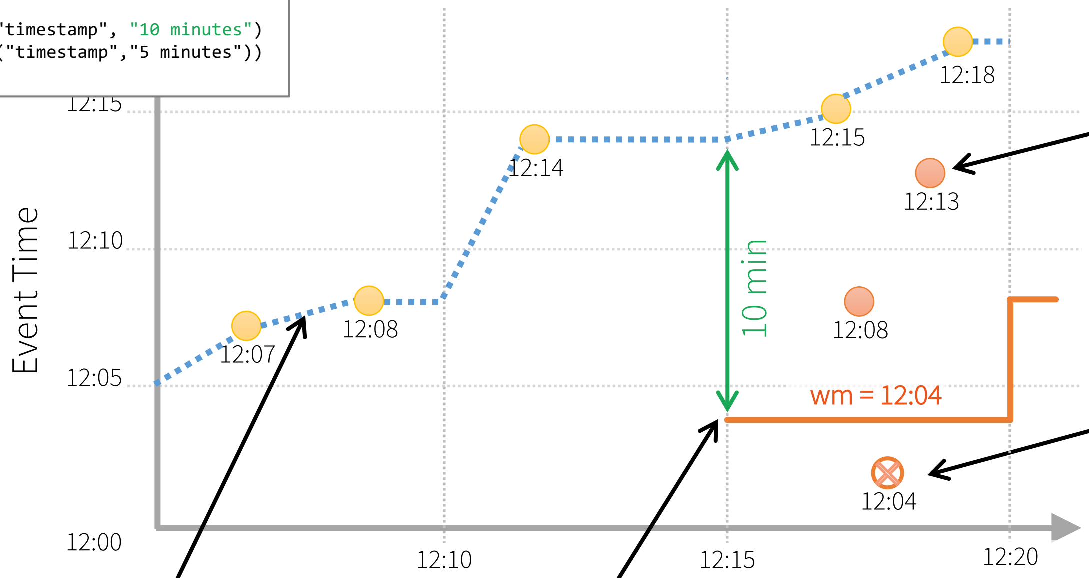
```
parsedData
      .withWatermark("timestamp", "10 minutes")
      .groupBy(window("timestamp","5 minutes"))
      .count()
```



databricks

Watermarking to Limit State [Spark 2.1]

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



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 online [programming guide]

databricks

Arbitrary Stateful Operations [Spark 2.2]

mapGroupsWithState
allows any user-defined stateful ops to a user-defined state

fault-tolerant, exactly-once

supports type-safe langs
Scala and Java

```
dataset
    .groupByKey(groupingFunc)
    .mapGroupsWithState(mappingFunc)


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

databricks

Many more updates!

StreamingQueryListener [Spark 2.1]
Receive of regular progress heartbeats for health and perf monitoring
Automatic in Databricks!!

Kafka Batch Queries [Spark 2.2]
Run batch queries on Kafka just like a file system

Kafka Sink [Spark 2.2]
Write to Kafka, can only give at-least-once guarantee as
Kafka doesn't support transactional updates

Update Output mode [Spark 2.2]
Only updated rows in result table to be written to sink

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/continuous-applications-evolving-streaming-in-apache-spark-2-0.html

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

and more, stay tuned!!

Comparison with Other Engines

Property	Structured Streaming	Spark Streaming	Apache Storm	Apache Flink	Kafka Streams	Google Dataflow
Streaming API	incrementalize batch queries	integrates with batch	separate from batch	separate from batch	separate from batch	integrates with batch
Prefix Integrity Guarantee	✔	✔	✘	✘	✘	✘
Internal Processing	exactly once	exactly once	at least once	exactly once	at least once	exactly once
Transactional Sources/Sinks	✔	some	some	some	✘	✘
Interactive Queries	✔	✔	✘	✘	✘	✘
Joins with Static Data	✔	✔	✘	✘	✘	✘

Read the blog to understand this table