A Deep Dive into Stateful Stream Processing in Structured Streaming

Tathagata "*TD*" Das
@tathadas

Spark + AI Summit Europe 2018
4th October, London

databricks

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

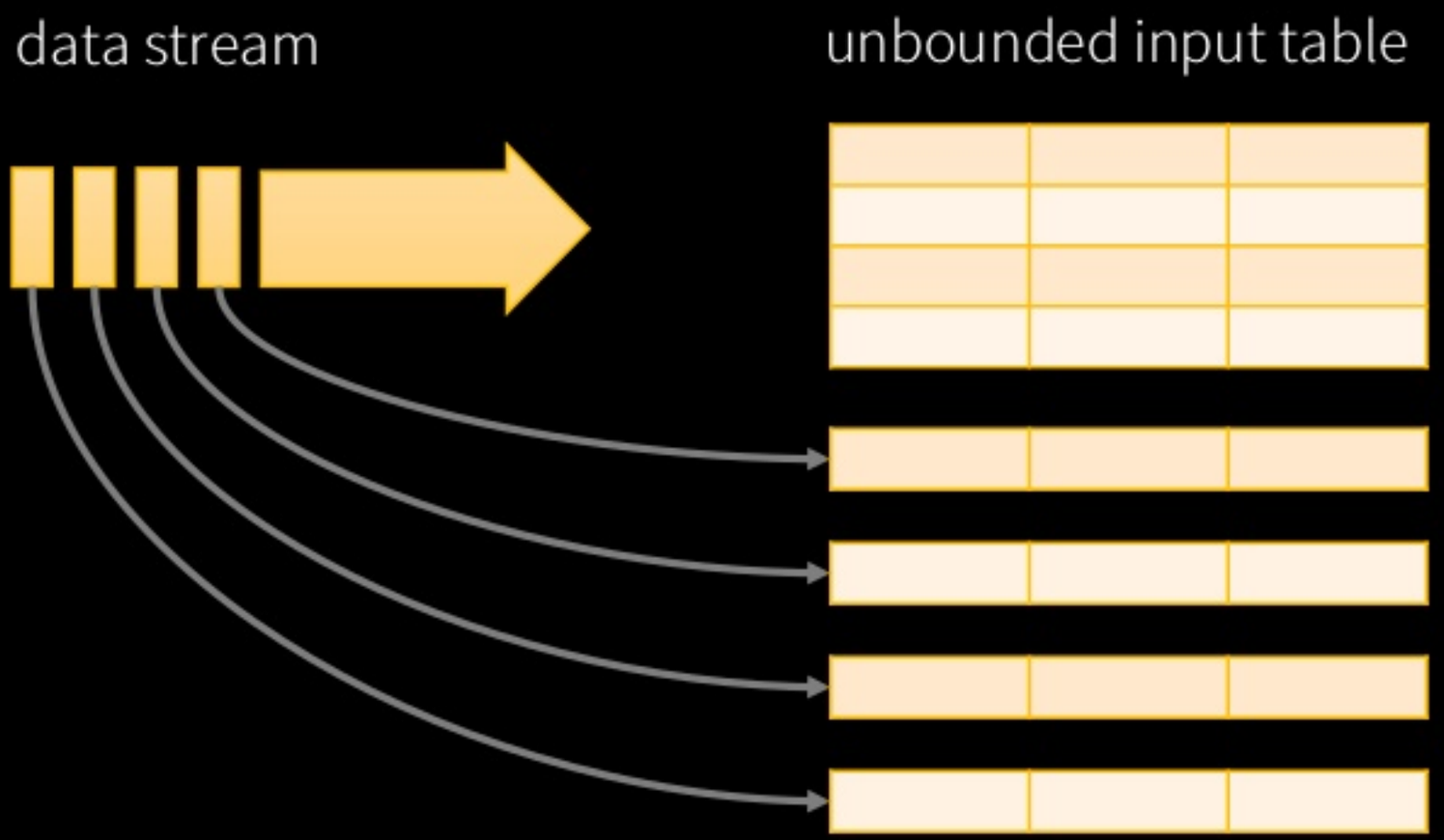
should write simple queries

&

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 Spark DataFrame
(common API for batch & streaming data)

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

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

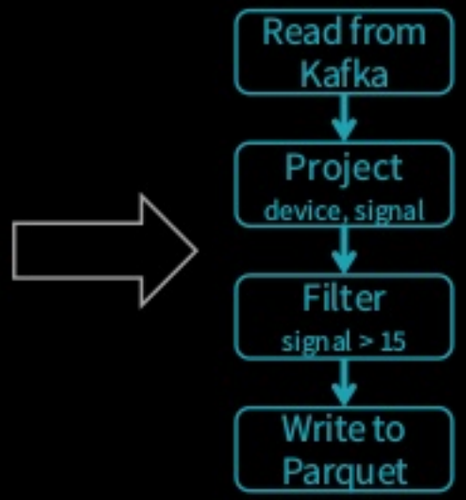
Trigger: when to process data

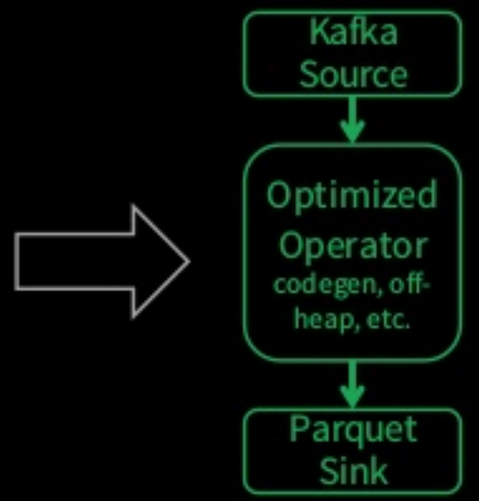
- Fixed interval micro-batches
- As fast as possible micro-batches
- Continuously (new in Spark 2.3)

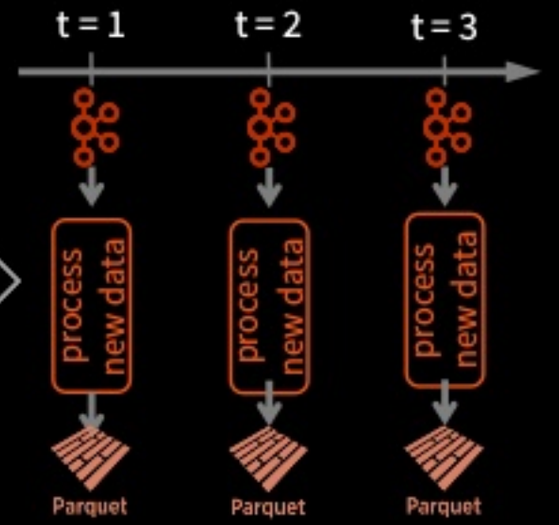
Checkpoint location: for tracking the progress of the query

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

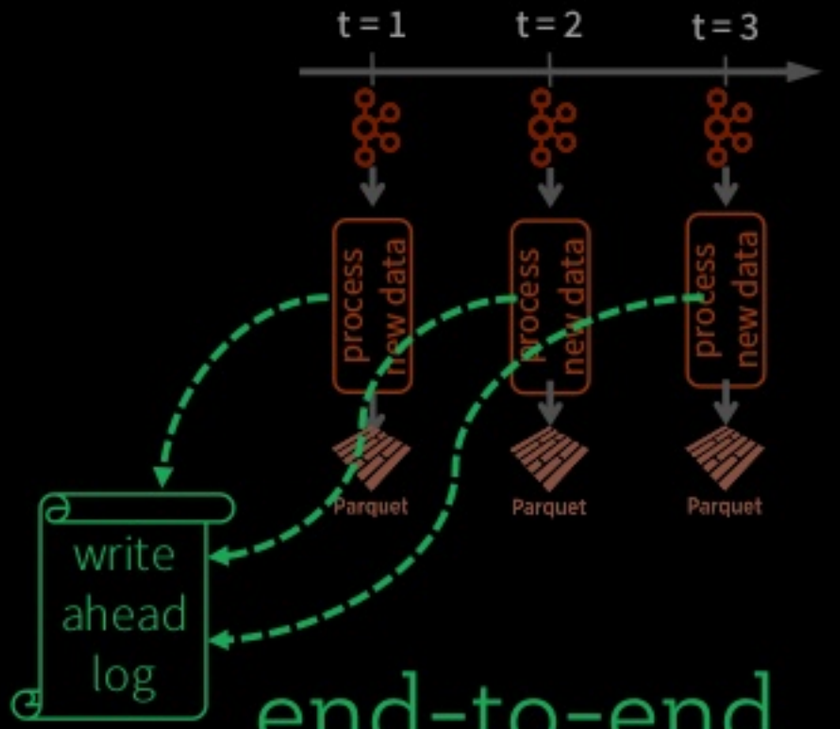
Spark SQL converts batch-like query to a series of incremental execution plans operating on new micro-batches of data

Fault-tolerance with Checkpointing

Checkpointing

Saves processed offset info to stable storage
- 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

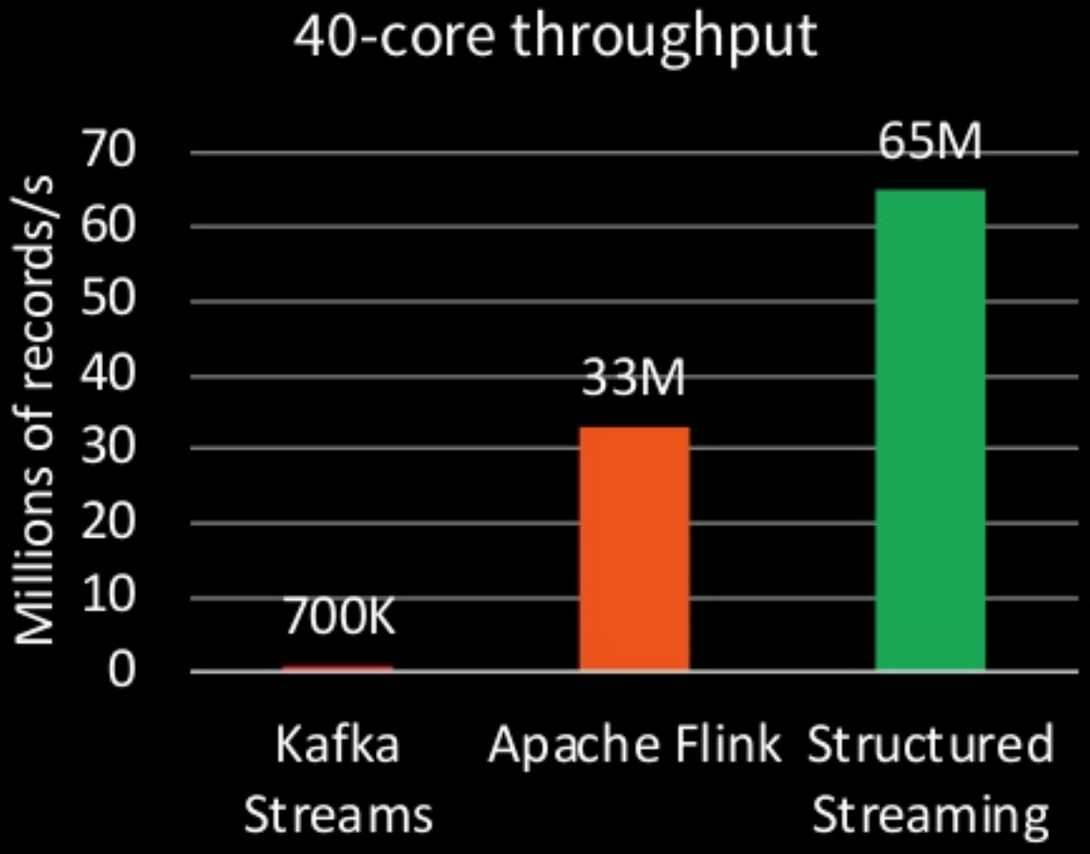
```
spark.readStream.format("kafka")
  .option("kafka.boostrap.servers",...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .select(from_json("json", schema).as("data"))
  .writeStream
  .format("parquet")
  .option("path", "/parquetTable/")
  .trigger("1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

2x cheaper



More details in our blog post

Stateful

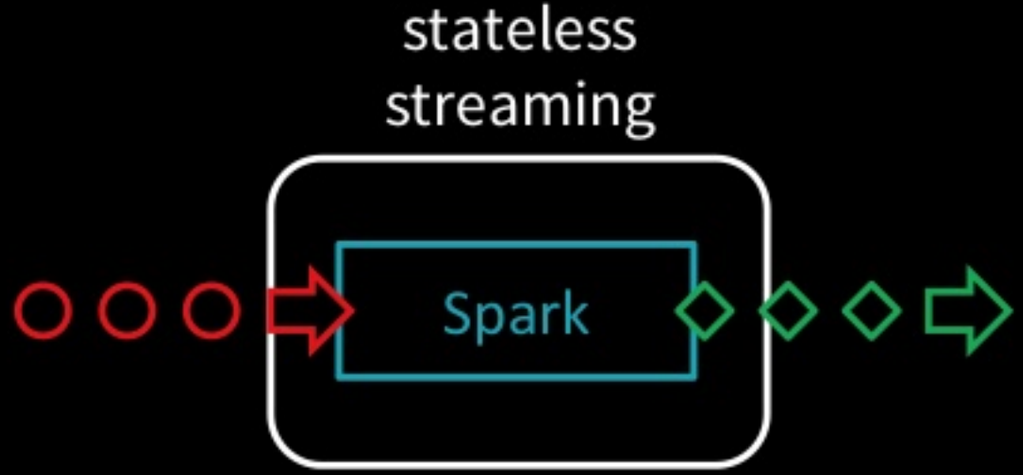
Stream Processing

What is State*less* Stream Processing?

Stateless streaming queries (e.g. ETL) process each record independent of other records

```
df.select(from_json("json", schema).as("data"))
  .where("data.type = 'typeA')
```

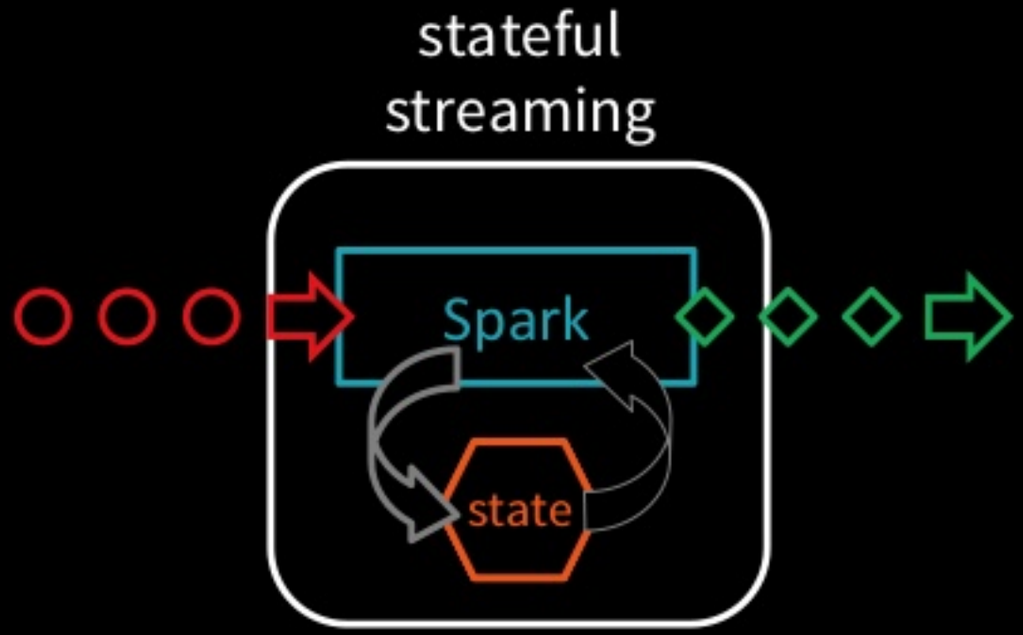
Every record is parsed into a structured form and then selected (or not) by the filter



What is State*ful* Stream Processing?

Stateful streaming queries combine information from multiple records together

State is the information that is maintained for future use




```
df.select(from_json("json", schema).as("data"))
  .where("data.type = 'typeA')
  .count()
```

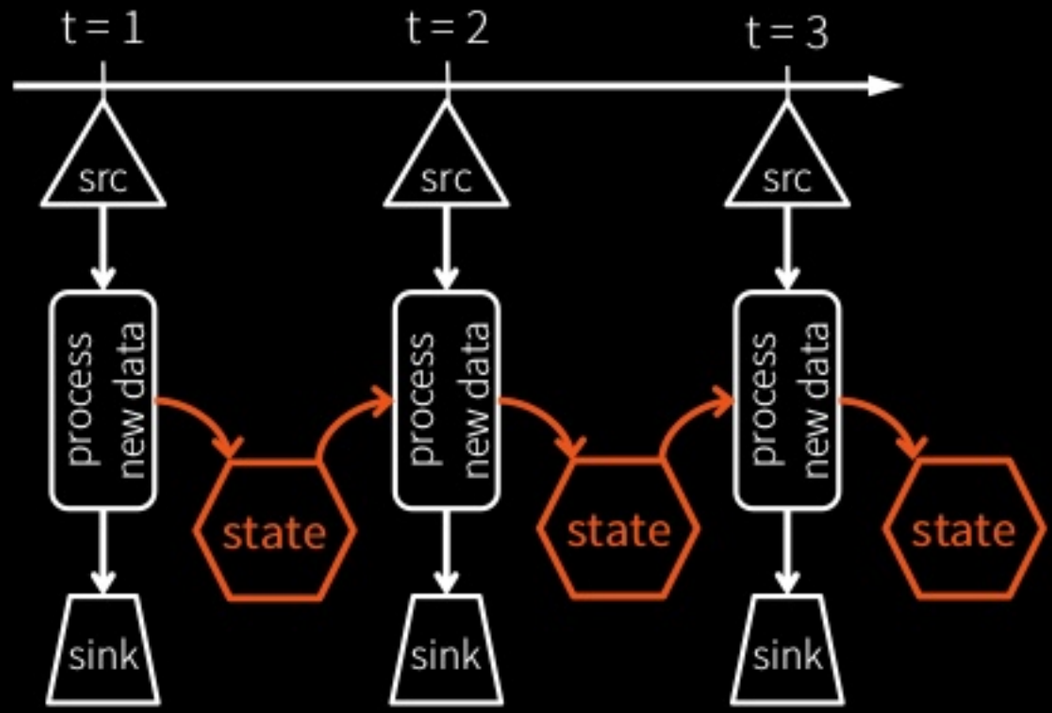
Count is the streaming state and every selected record increments the count

Stateful Micro-Batch Processing

State is versioned between micro-batches while streaming query is running

Each micro-batch reads previous version state and updates it to new version

Versions used for fault recovery

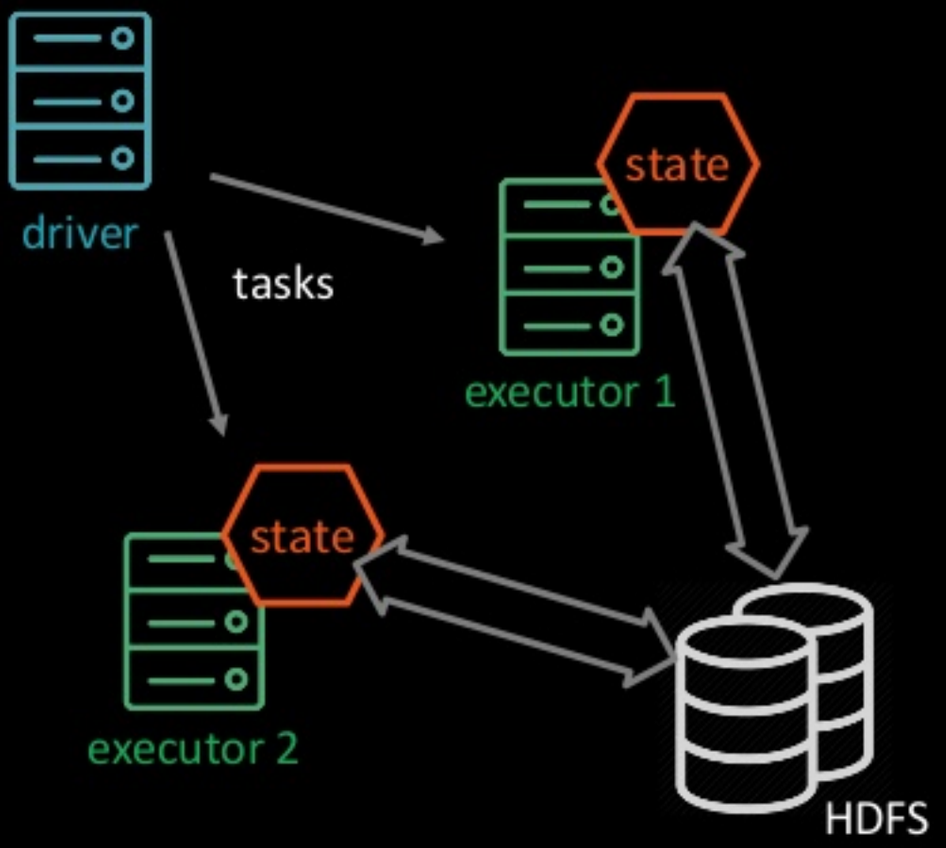


micro-batch incremental execution

Distributed, Fault-tolerant State

State data is distributed across executors

- State stored in the executor memory
- Micro-batch tasks update the state
- Changes are checkpointed with version to given checkpoint location (e.g. HDFS)
- Recovery from failure is automatic



Exactly-once fault-tolerance guarantees!

Philosophy of Stateful Operations

Two types of Stateful Operations

Automatic State Cleanup

User-defined State Cleanup

Automatic State Cleanup

For SQL operations with well known semantics

State cleanup is automatic with watermarking because we precisely know when state data is not needed any more

User-defined State Cleanup

For user-defined, arbitrary stateful operations

No automatic state cleanup

User has to explicitly manage state

Automatic State Cleanup

aggregations

deduplications

joins

User-defined State Cleanup

`mapGroupsWithState`

`flatMapGroupsWithState`

Rest of this talk

Explore built-in stateful operations

How to use watermarks to control state size

How to build arbitrary stateful operations

How to monitor and debug stateful queries

Streaming *Aggregation*

Aggregation by key and/or time windows

Aggregation by key only

```
events
  .groupBy("key")
  .count()
```

Aggregation by event time windows

```
events
  .groupBy(window("timestamp","10 mins"))
  .avg("value")
```

Aggregation by both

```
events
  .groupBy(
    col(key),
    window("timestamp","10 mins"))
  .agg(avg("value"), corr("value"))
```

Supports multiple aggregations, user-defined functions (UDAFs)

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 state

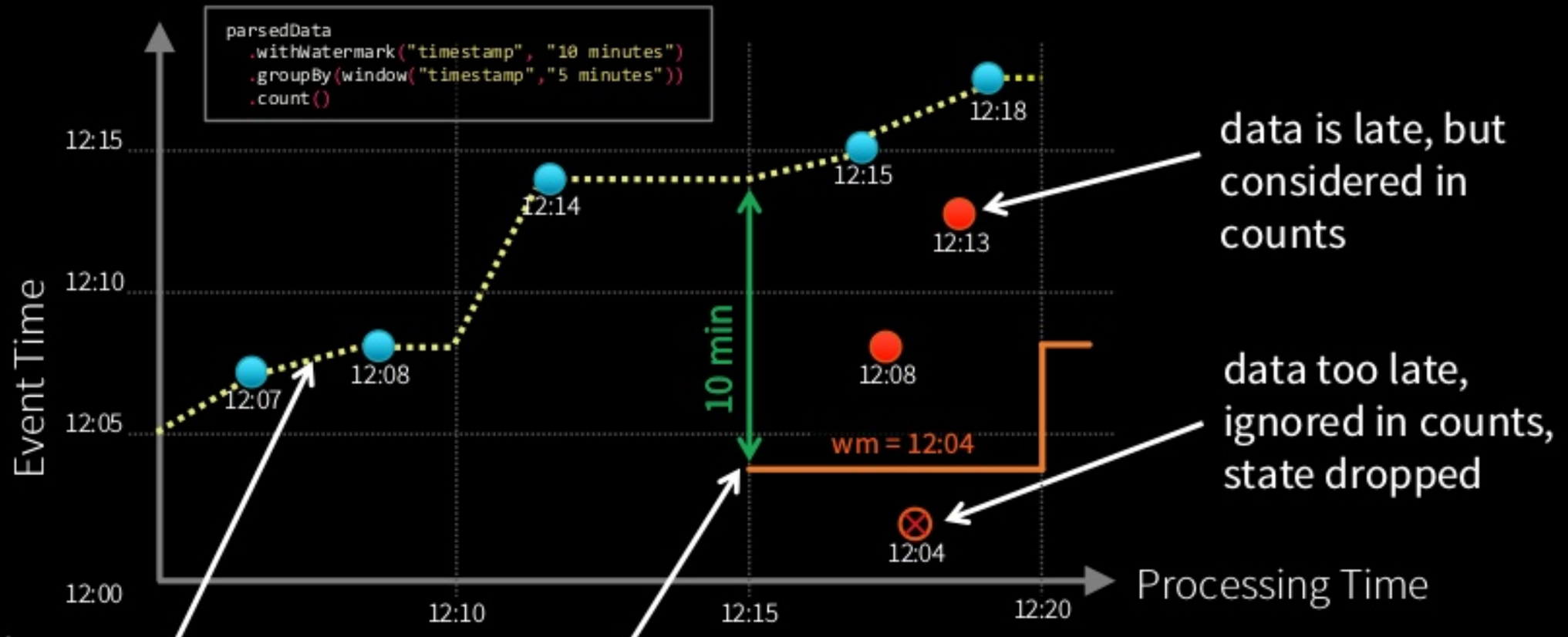
Watermarking

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

Used 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

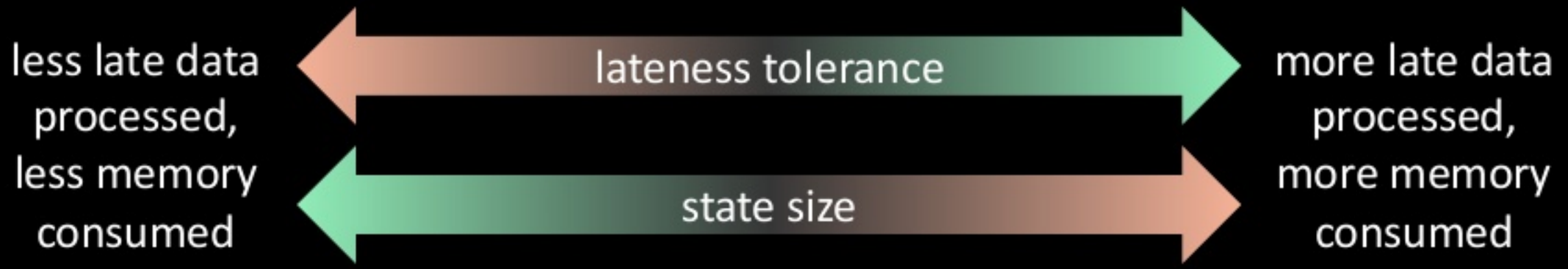
watermark updated to 12:14 - 10m = 12:04 for next trigger, state < 12:04 deleted

More details in my blog post

databricks

Watermarking

Trade off between lateness tolerance and state size



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?

Streaming *Deduplication*

Streaming Deduplication

Drop duplicate records in a stream

Specify columns which uniquely identify a record

```
userActions
  .dropDuplicates("uniqueRecordId")
```

Spark SQL will store past unique column values as state and drop any record that matches the state

Streaming Deduplication with Watermark

Timestamp as a unique column along with watermark allows old values in state to dropped

- Records older than watermark delay is not going to get any further duplicates

Timestamp must be same for duplicated records

```
userActions
  .withWatermark("timestamp")
  .dropDuplicates(
    "uniqueRecordId",
    "timestamp")
```

Streaming *Joins*

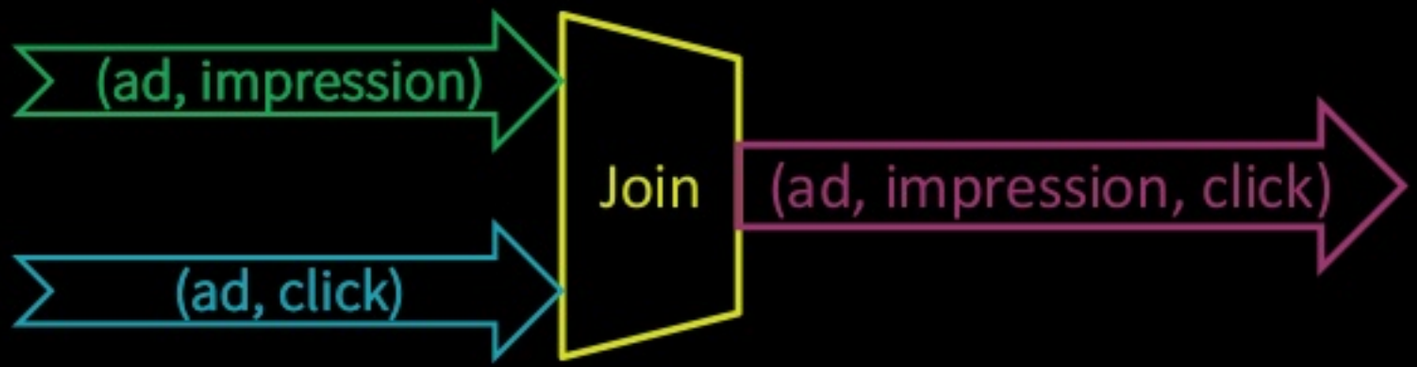
Streaming Joins

Spark 2.0+ supports joins between streams and static datasets

Spark 2.3+ supports joins between multiple streams

Example: Ad Monetization

Join stream of ad impressions with another stream of their corresponding user clicks

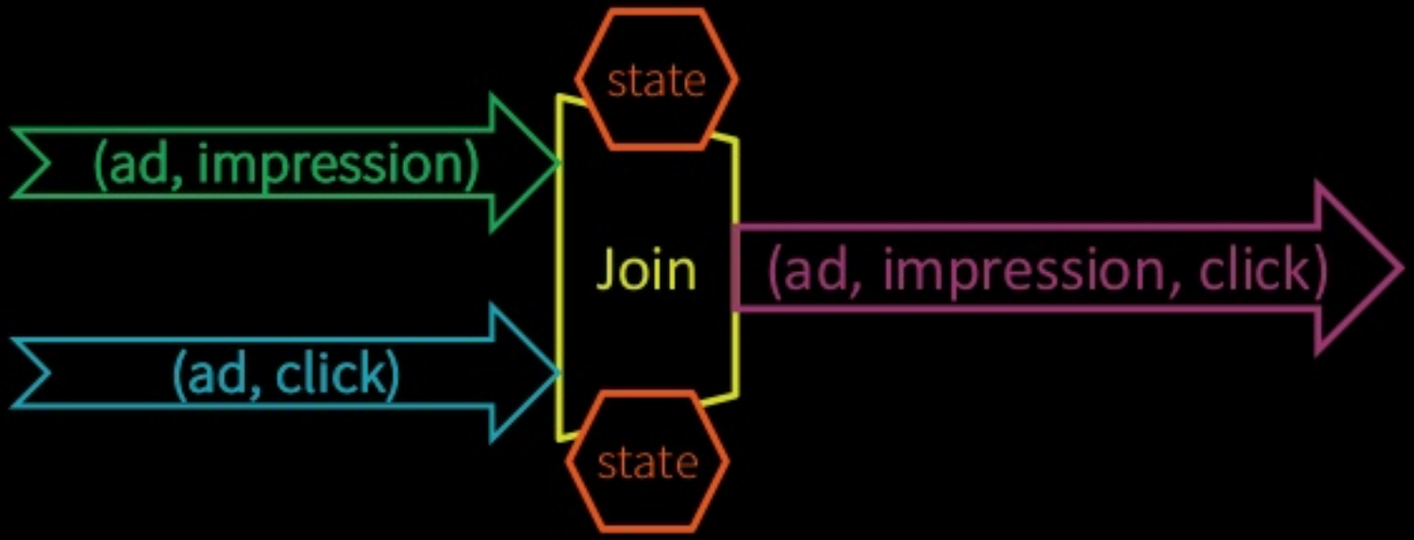


Streaming Joins

Most of the time click events arrive after their impressions

Sometimes, due to delays, impressions can arrive after clicks

Each stream in a join needs to buffer past events as *state* for matching with future events of the other stream



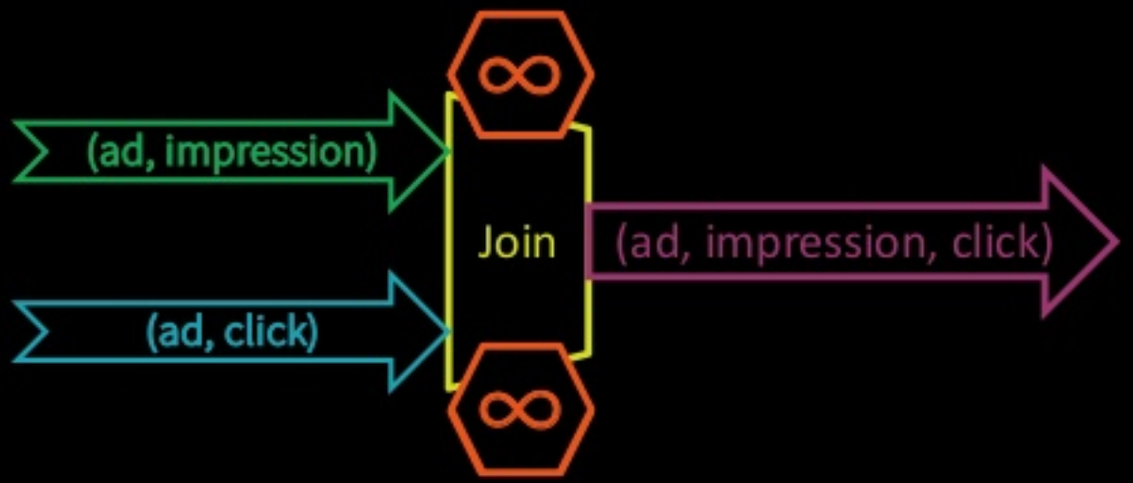
Simple Inner Join

Inner join by ad ID column

```
impressions.join(
  clicks,
  expr("clickAdId = impressionAdId")
)
```

Need to buffer all past events as state, a match can come on the other stream any time in the future

To allow buffered events to be dropped, query needs to provide more time constraints



Inner Join + Time constraints + Watermarks

Time constraints

Let's assume

- Impressions can be 2 hours late

- Clicks can be 3 hours late

- A click can occur within 1 hour after the corresponding impression

```
val impressionsWithWatermark = impressions
     .withWatermark("impressionTime", "2 hours")

val clicksWithWatermark = clicks
     .withWatermark("clickTime", "3 hours")
```

```
impressionsWithWatermark.join(
  clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime >= impressionTime AND
    clickTime <= impressionTime + interval 1 hour
    """
  ))
```

Range Join

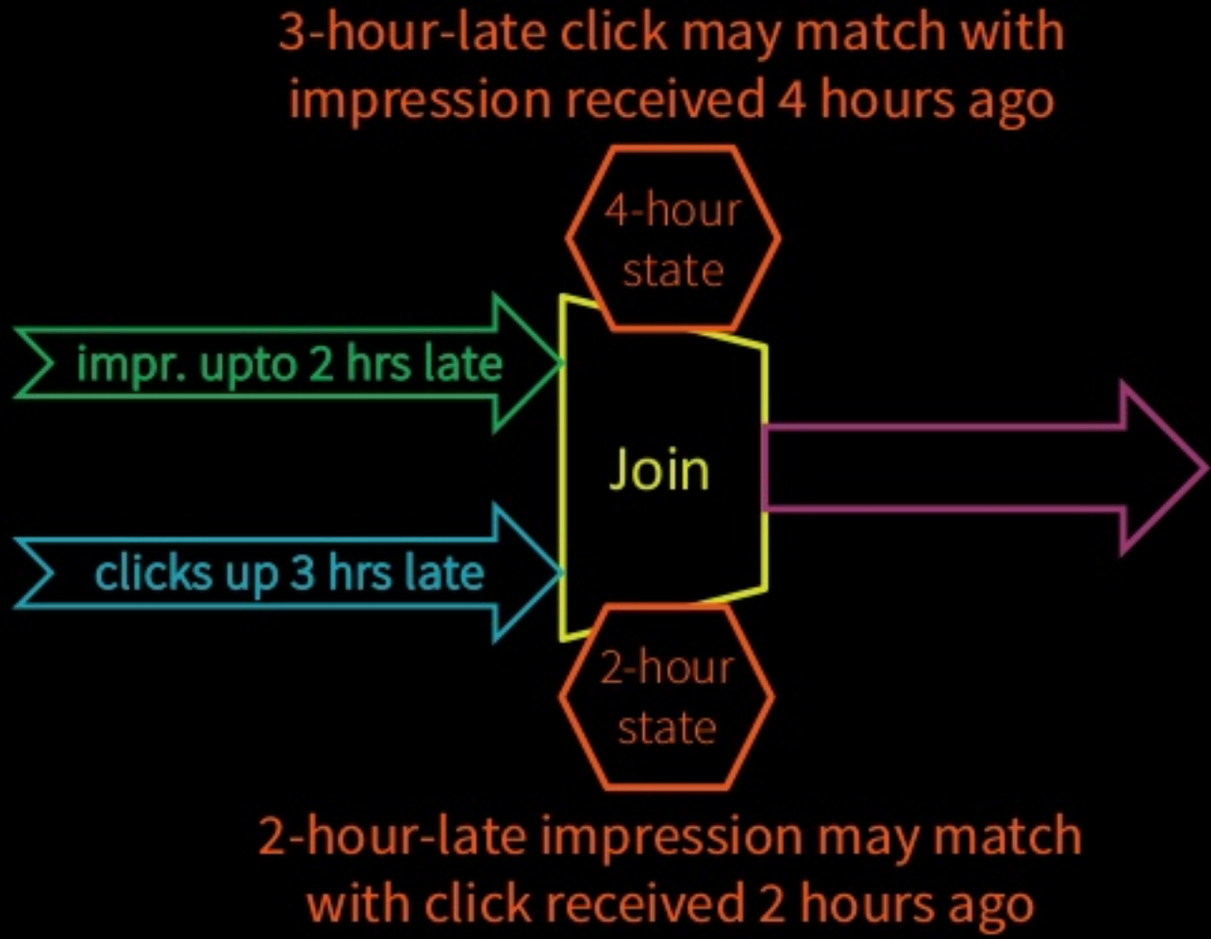
databricks

Inner Join + Time constraints + Watermarks

Spark calculates

- impressions need to be buffered for 4 hours
- clicks need to be buffered for 2 hours

Spark drops events older than these thresholds



Outer Join + Time constraints + Watermarks

Left and right outer joins are allowed only with time constraints and watermarks

Needed for correctness, Spark must output nulls when an event cannot get any future match

Note: null outputs are delayed as Spark has to wait for sometime to be sure that there cannot be any match

```
impressionsWithWatermark.join(
  clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime >= impressionTime AND
    clickTime <= impressionTime + interval 1 hour
    """
  ),
  joinType = "leftOuter"
)
```

Can be "inner" (default) / "leftOuter" / "rightOuter"

Arbitrary Stateful Operations

Arbitrary Stateful Operations

Many use cases require more complicated logic than SQL ops

Example: Tracking user activity on your product

- Input: User actions (login, clicks, logout, …)
- Output: Latest user status (online, active, inactive, …)

Solution: MapGroupsWithState / FlatMapGroupsWithState

- General API for per-key user-defined stateful processing
- Since Spark 2.2, for Scala and Java only

MapGroupsWithState / FlatMapGroupsWithState

No automatic state clean up and dropping of late data

- Adding watermark does not automatically manage late and state data
- Explicit state clean up by the user

More powerful + efficient than DStream's `mapWithState` and `updateStateByKey`

MapGroupsWithState - How to use?

1. Define the data structures
   - *Input event:* UserAction
   - *State data:* UserStatus
   - *Output event:* UserStatus
     (can be different from state)

```
case class UserAction(
    userId: String, action: String)

case class UserStatus(
    userId: String, active: Boolean)
```

MapGroupsWithState - How to use?

2. Define function to update state of each grouping key using the new data
   - *Input*
     - *Grouping key:* userId
     - *New data:* new user actions
     - *Previous state:* previous status of this user

```
case class UserAction(
    userId: String, action: String)

case class UserStatus(
    userId: String, active: Boolean)


def updateState(
  userId: String,
  actions: Iterator[UserAction],
  state: GroupState[UserStatus]):UserStatus = {


}
```

MapGroupsWithState - How to use?

2. Define function to update state of each grouping key using the new data
   - *Body*
     - Get previous user status
     - Update user status with actions
     - Update state with latest user status
     - Return the status

```
def updateState(
  userId: String,
  actions: Iterator[UserAction],
  state: GroupState[UserStatus]):UserStatus = {

  val prevStatus = state.getOption.getOrElse {
    new UserStatus()
  }

  actions.foreah { action =>
    prevStatus.updateWith(action)
  }

  state.update(prevStatus)

  return prevStatus

}
```

MapGroupsWithState - How to use?

3. Use the user-defined function on a grouped Dataset

```
def updateState(
  userId: String,
  actions: Iterator[UserAction],
  state: GroupState[UserStatus]):UserStatus = {

  // process actions, update and return status

}
```

Works with both batch and streaming queries

- In batch query, the function is called only once per group with no prior state

```
userActions
  .groupByKey(_.userId)
  .mapGroupsWithState(updateState)
```

Timeouts

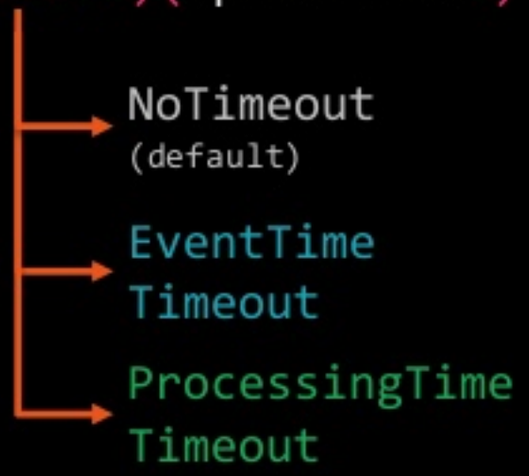
Example: Mark a user as inactive when there is no actions in 1 hour

Timeouts: When a group does not get any event for a while, then the function is called for that group with an empty iterator

Must specify a global timeout type, and set per-group timeout timestamp/duration

Ignored in a batch queries

```
userActions
  .groupByKey(_.userId)
  .mapGroupsWithState
    (timeoutConf)(updateState)
```



Event-time Timeout - How to use?

1. Enable EventTimeTimeout in mapGroupsWithState

2. Enable watermarking

3. Update the mapping function
   - Every time function is called, set the timeout timestamp using the max seen event timestamp + timeout duration
   - Update state when timeout occurs

```
userActions
  .withWatermark("timestamp")
  .groupByKey(_.userId)
  .mapGroupsWithState
    (EventTimeTimeout)(updateState)

 def updateState(...): UserStatus = {
   if (!state.hasTimedOut) {
     // track maxActionTimestamp while
     // processing actions and updating state
     state.setTimeoutTimestamp(
       maxActionTimestamp, "1 hour")
   } else {  // handle timeout
     userStatus.handleTimeout()
     state.remove()
   }
   // return user status
 }
```

Event-time Timeout - When?

Watermark is calculated with max event time across all groups

For a specific group, if there is no event till watermark exceeds the timeout timestamp,

Then

Function is called with an empty iterator, and hasTimedOut = true

Else

Function is called with new data, and timeout is disabled

Needs to explicitly set timeout timestamp every time

Processing-time Timeout

Instead of setting timeout timestamp, function sets timeout duration (in terms of wall-clock-time) to wait before timing out

- Independent of watermarks
- Note, query downtimes will cause lots of timeouts after recovery

```
userActions
  .groupByKey(_.userId)
  .mapGroupsWithState
    (ProcessingTimeTimeout)(updateState)


def updateState(...): UserStatus = {
  if (!state.hasTimedOut) {
    // handle new data
    state.setTimeoutDuration("1 hour")
  } else {
    // handle timeout
  }
  return userStatus
}
```

FlatMapGroupsWithState

More general version where the function can return any number of events, possibly none at all

Example: instead of returning user status, want to return specific actions that are significant based on the history

```
userActions
  .groupByKey(_.userId)
  .flatMapGroupsWithState
    (outputMode, timeoutConf)
    (updateState)
```

```
def updateState(
    userId: String,
    actions: Iterator[UserAction],
    state: GroupState[UserStatus]):
  Iterator[SpecialUserAction] = {

}
```

Function Output Mode

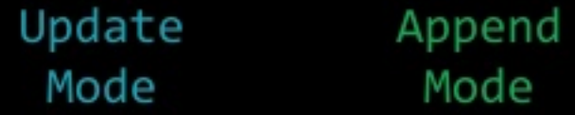
Function output mode* gives Spark insights into the output from this opaque function

Update Mode - Output events are key-value pairs, each output is updating the value of a key in the result table

Append Mode - Output events are independent rows that being appended to the result table

Allows Spark SQL planner to correctly compose flatMapGroupsWithState with other operations

```
userActions
  .groupByKey(_.userId)
  .flatMapGroupsWithState
    (outputMode, timeoutConf)
    (updateState)
```



*Not to be confused with output mode of the query

Managing Stateful Streaming Queries

Optimizing Query State

Set # shuffle partitions to 1-3 times number of cores

- Too low = not all cores will be used → lower throughput
- Too high = cost of writing state to HDFS will increases → higher latency

Total size of state per worker

- Larger state leads to higher overheads of snapshotting, JVM GC pauses, etc.

Monitoring the *state* of Query State

Get current state metrics using the last progress of the query

- Total number of rows in state
- Total memory consumed (approx.)

```
val progress = query.lastProgress
print(progress.json)
```

```
{
 ...
 "stateOperators" : [ {
   "numRowsTotal" : 660000,
   "memoryUsedBytes" : 120571087
   ...
 } ],
}
```

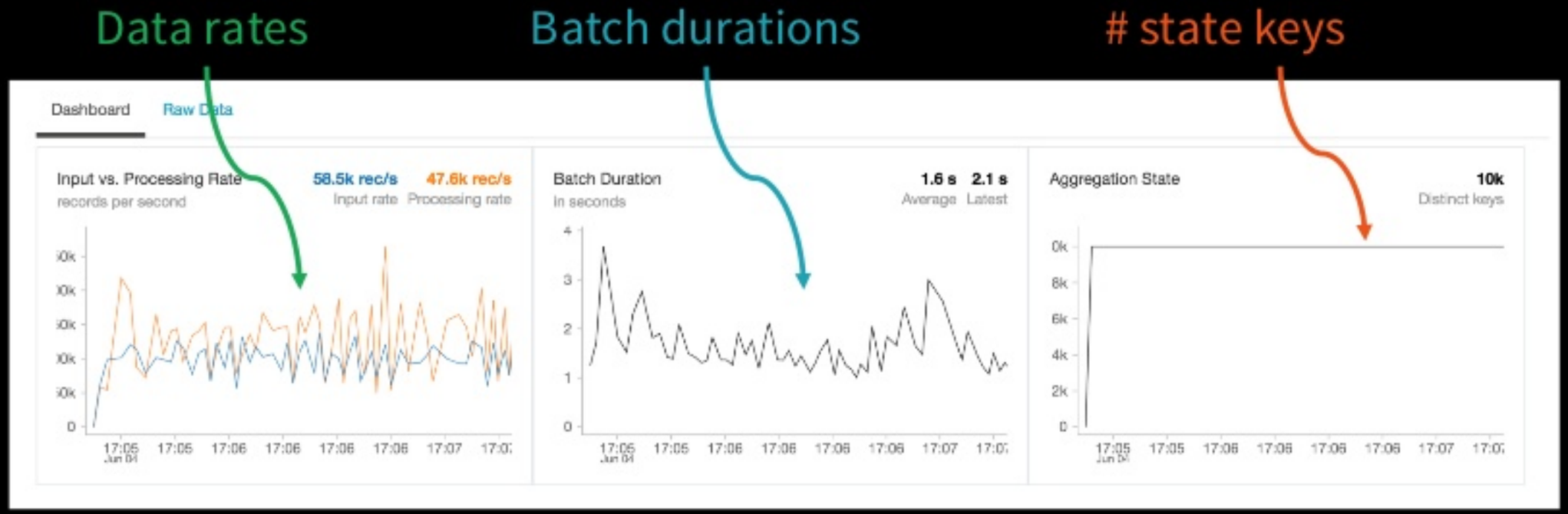
Get it asynchronously through StreamingQueryListener API

```
new StreamingQueryListener {
  ...
  def onQueryProgress(
    event: QueryProgressEvent)
}
```

Monitoring the *state* of Query State

Databricks Notebooks integrated with Structured Streaming

Shows size of state along with other processing metrics

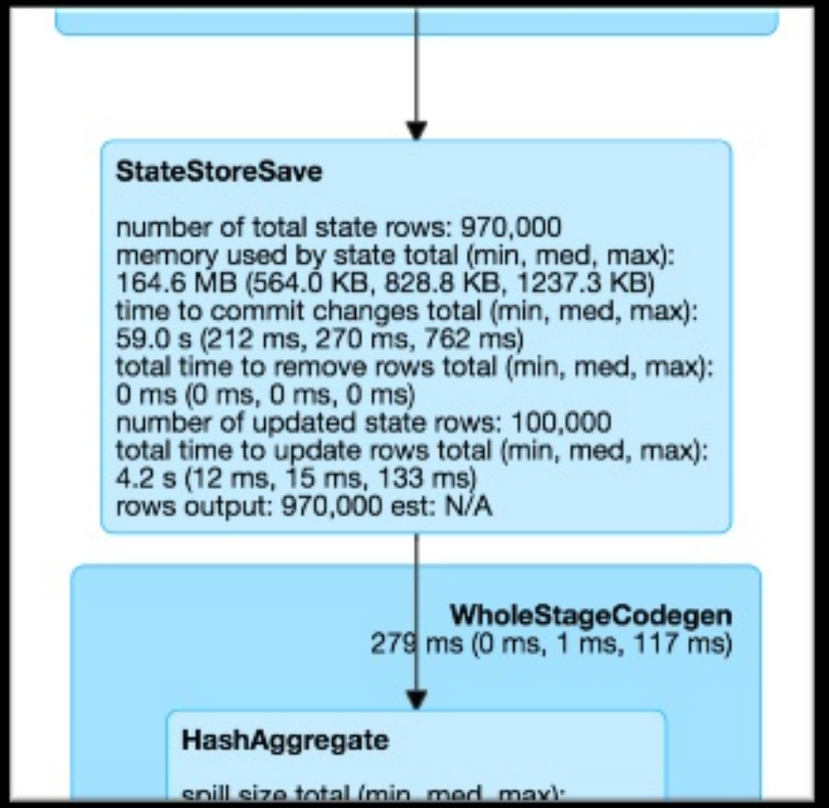


Debugging Query State

SQL metrics in the Spark UI (SQL tab, DAG view) expose more operator-specific stats

Answer questions like

- Is the memory usage skewed?
- Is removing rows slow?
- Is writing checkpoints slow?

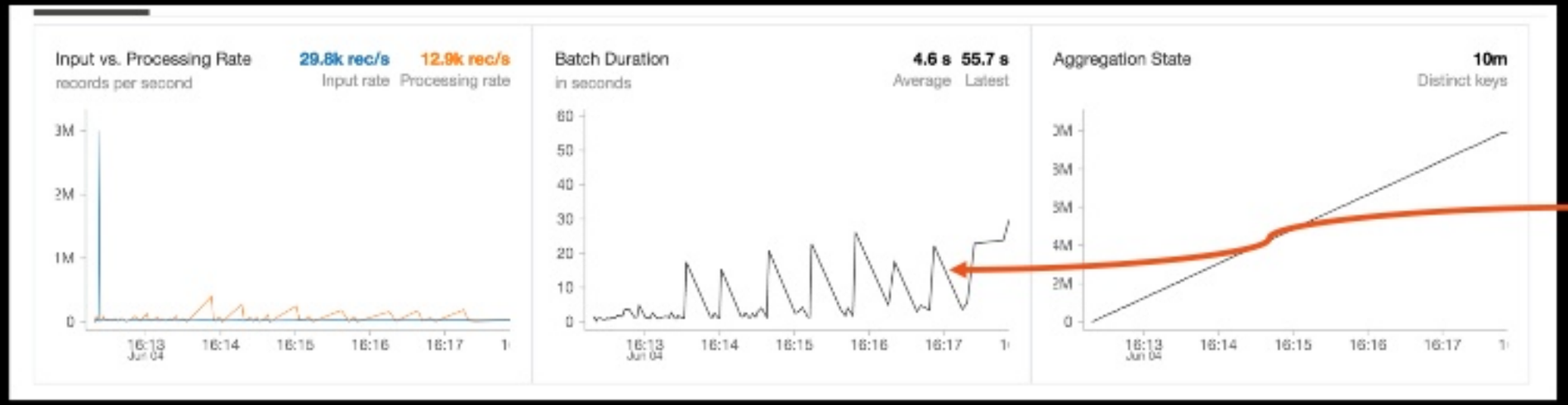


Managing Very Large State

State data kept on JVM heap

Can have GC issues with *millions* of state keys per worker

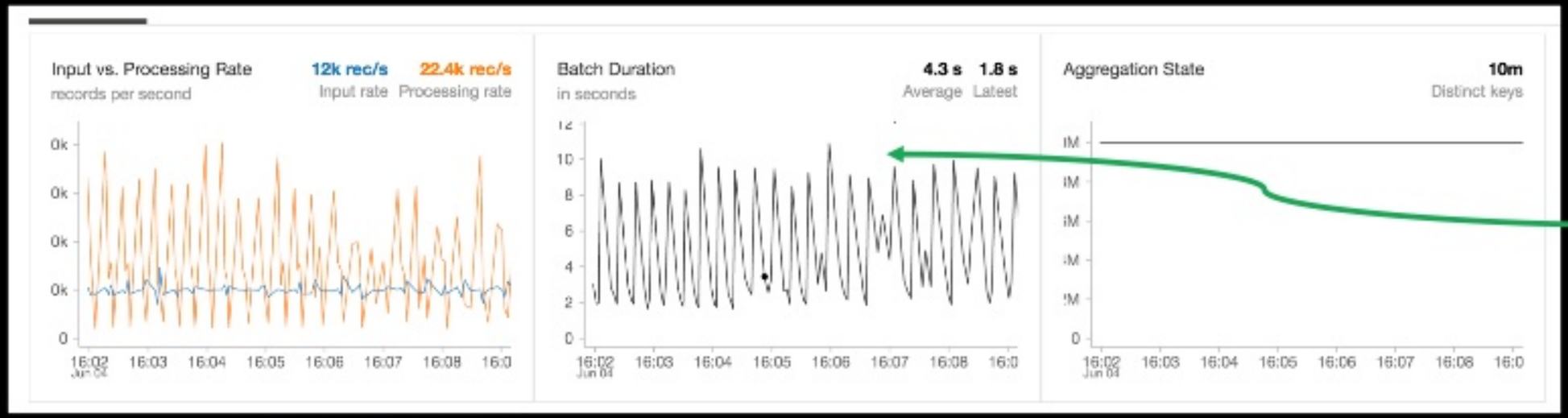
- Limits depend on the size and complexity of state data structures



Managing Very Large Statewith RocksDB

In Databricks Runtime, you can store state locally in RocksDB

- Avoids JVM heap, no GC issues with 100 millions state keys per worker
- Local RocksDB snapshot files automatically checkpointed to HDFS
- Same exactly-once fault-tolerant guarantees



[More info in Databricks Docs]

databricks

New in Apache Spark 2.4

- Lower state memory usage for streaming aggregations
- `foreach()` in Python
- `foreachBatch()` in Scala and Python
  - Reuse existing batch data sources for streaming output

    Example: Write to Cassandra using Cassandra batch data source
  - Write streaming output to multiple locations

```
streamingDF.writeStream.foreachBatch { (batchDF: DataFrame, batchId: Long) =>
    batchDF.write.format(...).save(...) // location 1
    batchDF.write.format(...).save(...) // location 2
}
```

More Info

Structured Streaming Docs

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

https://docs.databricks.com/spark/latest/structured-streaming/index.html

Databricks blog posts for more focused discussions

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

My previous talk on the basics of Structured Streaming

https://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming

Questions?