

Table of Contents

Introduction	1.1
Spark Structured Streaming and Streaming Queries	1.2
Batch Processing Time	1.2.1
Internals of Streaming Queries	1.3

Arbitrary Stateful Streaming Aggregation

Arbitrary Stateful Streaming Aggregation	2.1
GroupState	2.2
GroupStateImpl	2.2.1
GroupStateTimeout	2.3
StateManager	2.4
StateManagerImplV2	2.4.1
StateManagerImplBase	2.4.2
StateManagerImplV1	2.4.3
FlatMapGroupsWithStateExecHelper Helper Class	2.5
InputProcessor Helper Class of FlatMapGroupsWithStateExec Physical Operator	2.6

Demos

Demos	3.1
Internals of FlatMapGroupsWithStateExec Physical Operator	3.2
Arbitrary Stateful Streaming Aggregation with KeyValueGroupedDataset.flatMapGroupsWithState Operator	3.3
Exploring Checkpointed State	3.4
Streaming Watermark with Aggregation in Append Output Mode	3.5
Streaming Query for Running Counts (Socket Source and Complete Output Mode)	3.6
Streaming Join of Streaming Queries and StreamingSymmetricHashJoinExec Physical Operator	3.7
Streaming Aggregation with Kafka Data Source	3.8
groupByKey Streaming Aggregation in Update Mode	3.9

StateStoreSaveExec with Complete Output Mode	3.10
StateStoreSaveExec with Update Output Mode	3.11
Developing Custom Streaming Sink (and Monitoring SQL Queries in web UI)	3.12
current_timestamp Function For Processing Time in Streaming Queries	3.13
Using StreamingQueryManager for Query Termination Management	3.14

Streaming Join

Streaming Join	4.1
StateStoreAwareZipPartitionsRDD	4.2
SymmetricHashJoinStateManager	4.3
StateStoreHandler	4.3.1
KeyToNumValuesStore	4.3.2
KeyWithIndexToValueStore	4.3.3
OneSideHashJoiner	4.4
StateStoreAwareZipPartitionsHelper	4.5

Streaming Aggregation

Streaming Aggregation	5.1
StateStoreRDD	5.2
StateStoreOps	5.2.1
StreamingAggregationStateManager	5.3
StreamingAggregationStateManagerBaseImpl	5.3.1
StreamingAggregationStateManagerImplV1	5.3.2
StreamingAggregationStateManagerImplV2	5.3.3

Stateful Stream Processing

Stateful Stream Processing	6.1
Streaming Watermark	6.2
Streaming Deduplication	6.3
Streaming Limit	6.4

StateStore	6.5
StateStoreId	6.5.1
HDFSBackedStateStore	6.5.2
StateStoreProvider	6.6
StateStoreProviderId	6.6.1
HDFSBackedStateStoreProvider	6.6.2
StateStoreCoordinator	6.7
StateStoreCoordinatorRef	6.7.1
WatermarkSupport	6.8
StatefulOperator	6.9
StateStoreReader	6.9.1
StateStoreWriter	6.9.2
StatefulOperatorStateInfo	6.10
StateStoreMetrics	6.11
StateStoreCustomMetric	6.12
StateStoreUpdater	6.13
EventTimeStatsAccum	6.14
StateStoreConf	6.15

Developing Streaming Applications

DataStreamReader	7.1
DataStreamWriter	7.2
OutputMode	7.2.1
Trigger	7.2.2
StreamingQuery	7.3
Streaming Operators	7.4
dropDuplicates Operator	7.4.1
explain Operator	7.4.2
groupBy Operator	7.4.3
groupByKey Operator	7.4.4
withWatermark Operator	7.4.5
window Function	7.5
KeyValueGroupedDataset	7.6

mapGroupsWithState Operator	7.6.1
flatMapGroupsWithState Operator	7.6.2
StreamingQueryManager	7.7
SQLConf	7.8
Configuration Properties	7.9

Monitoring

StreamingQueryListener — Intercepting Streaming Events	8.1
 StreamingQueryProgress	8.1.1
MetricsReporter	8.2
ProgressReporter Contract	8.3
 ExecutionStats	8.3.1
 StreamingQueryStatus	8.3.2
 SourceProgress	8.3.3
 SinkProgress	8.3.4
Web UI	8.4
Logging	8.5

Extending Structured Streaming

DataSource	9.1
BaseStreamingSource	9.2
BaseStreamingSink	9.3
Streaming Source	9.4
 StreamSourceProvider	9.4.1
Streaming Sink	9.5
 StreamSinkProvider	9.5.1
StreamWriterSupport	9.6
 StreamWriter	9.6.1

File-Based Data Source

FileStreamSource	10.1
----------------------------------	------

FileStreamSink	10.2
FileStreamSinkLog	10.3
SinkFileStatus	10.4
ManifestFileCommitProtocol	10.5
MetadataLogFileIndex	10.6

Kafka Data Source

Kafka Data Source — Streaming Data Source for Apache Kafka	11.1
KafkaSourceProvider — Data Source Provider for Apache Kafka	11.2
KafkaSource	11.3
KafkaRelation	11.4
KafkaSourceRDD	11.5
CachedKafkaConsumer	11.6
KafkaOffsetReader	11.7
ConsumerStrategy Contract for KafkaConsumer Providers	11.8
KafkaSourceOffset	11.9
KafkaSink	11.10
KafkaOffsetRangeLimit — Desired Offset Range Limits	11.11
KafkaContinuousReader — ContinuousReader for Kafka Data Source in Continuous Stream Processing	11.12
KafkaMicroBatchReader	11.13
KafkaOffsetRangeCalculator	11.13.1
KafkaContinuousInputPartition	11.14
KafkaSourceInitialOffsetWriter	11.15

Text Socket Data Source

TextSocketSourceProvider	12.1
TextSocketSource	12.2

Rate Data Source

RateSourceProvider	13.1
------------------------------------	------

RateStreamSource	13.2
RateStreamMicroBatchReader	13.3

Console Data Sink

ConsoleSinkProvider	14.1
ConsoleWriter	14.2

Foreach Data Sink

ForeachWriterProvider	15.1
ForeachWriter	15.2
ForeachSink	15.3

ForeachBatch Data Sink

ForeachBatchSink	16.1
----------------------------------	------

Memory Data Sink

MemorySinkV2	17.1
MemorySink	17.2
MemoryStream	17.3

Micro-Batch Stream Processing (Structured Streaming V1)

Micro-Batch Stream Processing	18.1
MicroBatchExecution	18.2
MicroBatchWriter	18.2.1
MicroBatchReadSupport Contract	18.3
MicroBatchReader Contract	18.3.1
WatermarkTracker	18.4

Offsets and Metadata Checkpointing (Fault-Tolerance and Reliability)

Offsets and Metadata Checkpointing	19.1
MetadataLog	19.2
HDFSMetadataLog	19.3
CommitLog	19.4
CommitMetadata	19.4.1
OffsetSeqLog	19.5
OffsetSeq	19.5.1
CompactibleFileStreamLog	19.6
FileStreamSourceLog	19.6.1
OffsetSeqMetadata	19.7
CheckpointFileManager	19.8
FileContextBasedCheckpointFileManager	19.8.1
FileSystemBasedCheckpointFileManager	19.8.2
Offset	19.9
StreamProgress	19.10

Continuous Stream Processing (Structured Streaming V2)

Continuous Stream Processing	20.1
ContinuousExecution	20.2
ContinuousReadSupport Contract	20.3
ContinuousReader Contract	20.4
ContinuousMemoryStream	20.5
RateStreamContinuousReader	20.6
EpochCoordinator RPC Endpoint	20.7
EpochCoordinatorRef	20.7.1
EpochTracker	20.7.2
ContinuousQueuedDataReader	20.8
DataReaderThread	20.8.1

EpochMarkerGenerator	20.8.2
PartitionOffset	20.9
ContinuousExecutionRelation Leaf Logical Operator	20.10
WriteToContinuousDataSource Unary Logical Operator	20.11
WriteToContinuousDataSourceExec Unary Physical Operator	20.12
ContinuousWriteRDD	20.12.1
ContinuousDataSourceRDD	20.13

Query Planning and Execution

StreamExecution — Base of Streaming Query Execution Engines	21.1
StreamingQueryWrapper — Serializable StreamExecution	21.1.1
TriggerExecutor	21.2
IncrementalExecution	21.3
StreamingQueryListenerBus — Notification Bus for Streaming Events	21.4
StreamMetadata	21.5

Logical Operators

EventTimeWatermark Unary Logical Operator	22.1
FlatMapGroupsWithState Unary Logical Operator	22.2
Deduplicate Unary Logical Operator	22.3
MemoryPlan Logical Query Plan	22.4
StreamingRelation Leaf Logical Operator for Streaming Source	22.5
StreamingRelationV2 Leaf Logical Operator	22.6
StreamingExecutionRelation Leaf Logical Operator for Streaming Source At Execution	22.7

Physical Operators

EventTimeWatermarkExec	23.1
FlatMapGroupsWithStateExec	23.2
StateStoreRestoreExec	23.3
StateStoreSaveExec	23.4
StreamingDeduplicateExec	23.5

StreamingGlobalLimitExec	23.6
StreamingRelationExec	23.7
StreamingSymmetricHashJoinExec	23.8

Execution Planning Strategies

FlatMapGroupsWithStateStrategy	24.1
StatefulAggregationStrategy	24.2
StreamingDeduplicationStrategy	24.3
StreamingGlobalLimitStrategy	24.4
StreamingJoinStrategy	24.5
StreamingRelationStrategy	24.6

Varia

UnsupportedOperationChecker	25.1
-----------------------------	------

The Internals of Spark Structured Streaming (Apache Spark 2.4.3)

Welcome to **The Internals of Spark Structured Streaming** gitbook! I'm very excited to have you here and hope you will enjoy exploring the internals of Spark Structured Streaming as much as I have.

I write to discover what I know.

— Flannery O'Connor

I'm [Jacek Laskowski](#), a freelance IT consultant, software engineer and technical instructor specializing in [Apache Spark](#), [Apache Kafka](#) and [Kafka Streams](#) (with [Scala](#) and [sbt](#)).

I offer software development and consultancy services with hands-on in-depth workshops and mentoring. Reach out to me at jacek@japila.pl or [@jaceklaskowski](https://twitter.com/jaceklaskowski) to discuss opportunities.

Consider joining me at [Warsaw Scala Enthusiasts](#) and [Warsaw Spark](#) meetups in Warsaw, Poland.

Tip

I'm also writing other books in the "The Internals of" series about [Apache Spark](#), [Spark SQL](#), [Apache Kafka](#), and [Kafka Streams](#).

Expect text and code snippets from a variety of public sources. Attribution follows.

Now, let me introduce you to [Spark Structured Streaming and Streaming Queries](#).

Spark Structured Streaming and Streaming Queries

Spark Structured Streaming (aka *Spark Streams*) is the module of Apache Spark for stream processing using **streaming queries**.

Streaming queries can be expressed using a [high-level declarative streaming API](#) (*Dataset API*) or good ol' SQL (*SQL over stream / streaming SQL*). The declarative streaming Dataset API and SQL are executed on the underlying highly-optimized Spark SQL engine.

The semantics of the Structured Streaming model is as follows (see the article [Structured Streaming In Apache Spark](#)):

At any time, the output of a continuous application is equivalent to executing a batch job on a prefix of the data.

Note	As of Spark 2.2.0, Structured Streaming has been marked stable and ready for production use. With that the other older streaming module Spark Streaming is considered obsolete and not used for developing new streaming applications with Apache Spark.
------	---

Spark Structured Streaming comes with two [stream execution engines](#) for executing streaming queries:

- [MicroBatchExecution](#) for [Micro-Batch Stream Processing](#)
- [ContinuousExecution](#) for [Continuous Stream Processing](#)

The goal of Spark Structured Streaming is to unify streaming, interactive, and batch queries over structured datasets for developing end-to-end stream processing applications dubbed **continuous applications** using Spark SQL's Datasets API with additional support for the following features:

- [Streaming Aggregation](#)
- [Streaming Join](#)
- [Streaming Watermark](#)
- [Arbitrary Stateful Streaming Aggregation](#)
- [Stateful Stream Processing](#)

In Structured Streaming, Spark developers describe custom streaming computations in the same way as with Spark SQL. Internally, Structured Streaming applies the user-defined structured query to the continuously and indefinitely arriving data to analyze real-time streaming data.

Structured Streaming introduces the concept of **streaming datasets** that are *infinite datasets* with primitives like input **streaming data sources** and output **streaming data sinks**.

A `dataset` is **streaming** when its logical plan is streaming.

```
val batchQuery = spark.  
  read. // <-- batch non-streaming query  
  csv("sales")  
  
assert(batchQuery.isStreaming == false)  
  
val streamingQuery = spark.  
  readStream. // <-- streaming query  
  format("rate").  
  load  
  
assert(streamingQuery.isStreaming)
```

Tip

Read up on Spark SQL, Datasets and logical plans in [The Internals of Spark SQL book](#).

Structured Streaming models a stream of data as an infinite (and hence continuous) table that could be changed every streaming batch.

You can specify **output mode** of a streaming dataset which is what gets written to a streaming sink (i.e. the infinite result table) when there is a new data available.

Streaming Datasets use **streaming query plans** (as opposed to regular batch Datasets that are based on batch query plans).

From this perspective, batch queries can be considered streaming Datasets executed once only (and is why some batch queries, e.g. [KafkaSource](#), can easily work in batch mode).

Note

```
val batchQuery = spark.read.format("rate").load  
  
assert(batchQuery.isStreaming == false)  
  
val streamingQuery = spark.readStream.format("rate").load  
  
assert(streamingQuery.isStreaming)
```

With Structured Streaming, Spark 2 aims at simplifying **streaming analytics** with little to no need to reason about effective data streaming (trying to hide the unnecessary complexity in your streaming analytics architectures).

Structured streaming is defined by the following data abstractions in `org.apache.spark.sql.streaming` package:

- [StreamingQuery](#)
- [Streaming Source](#)
- [Streaming Sink](#)
- [StreamingQueryManager](#)

Structured Streaming follows micro-batch model and periodically fetches data from the data source (and uses the `DataFrame` data abstraction to represent the fetched data for a certain batch).

With Datasets as Spark SQL's view of structured data, structured streaming checks input sources for new data every [trigger](#) (time) and executes the (continuous) queries.

Note	The feature has also been called Streaming Spark SQL Query , Streaming DataFrames , Continuous DataFrame or Continuous Query . There have been lots of names before the Spark project settled on Structured Streaming.
------	--

Further Reading Or Watching

- [SPARK-8360 Structured Streaming \(aka Streaming DataFrames\)](#)
- The official [Structured Streaming Programming Guide](#)
- (article) [Structured Streaming In Apache Spark](#)
- (video) [The Future of Real Time in Spark](#) from Spark Summit East 2016 in which Reynold Xin presents the concept of **Streaming DataFrames**
- (video) [Structuring Spark: DataFrames, Datasets, and Streaming](#)
- (article) [What Spark's Structured Streaming really means](#)
- (video) [A Deep Dive Into Structured Streaming](#) by Tathagata "TD" Das from Spark Summit 2016
- (video) [Arbitrary Stateful Aggregations in Structured Streaming in Apache Spark](#) by Burak Yavuz

Batch Processing Time

Batch Processing Time (aka **Batch Timeout Threshold**) is the processing time (*processing timestamp*) of the current streaming batch.

The following standard functions (and their Catalyst expressions) allow accessing the batch processing time in [Micro-Batch Stream Processing](#):

- `now`, `current_timestamp`, and `unix_timestamp` functions (`CurrentTimestamp`)
- `current_date` function (`CurrentDate`)

Note

`CurrentTimestamp` or `CurrentDate` expressions are not supported in [Continuous Stream Processing](#).

Internals

`GroupStateImpl` is given the batch processing time when [created](#) for a streaming query (that is actually the [batch processing time](#) of the `FlatMapGroupsWithStateExec` physical operator).

When created, `FlatMapGroupsWithStateExec` physical operator has the processing time undefined and set to the current timestamp in the [state preparation rule](#) every streaming batch.

The current timestamp (and other batch-specific configurations) is given as the `OffsetSeqMetadata` (as part of the query planning phase) when a [stream execution engine](#) does the following:

- `MicroBatchExecution` is requested to [construct a next streaming micro-batch](#) in [Micro-Batch Stream Processing](#)
- In [Continuous Stream Processing](#) the base `StreamExecution` is requested to [run stream processing](#) and initializes `offsetSeqMetadata` to `0 s`.

Internals of Streaming Queries

Note	The page is to keep notes about how to guide readers through the codebase and may disappear if merged with the other pages or become an intro page.
------	---

1. [DataStreamReader and Streaming Data Source](#)
2. [Data Source Resolution, Streaming Dataset and Logical Query Plan](#)
3. [Dataset API — High-Level DSL to Build Logical Query Plan](#)
4. [DataStreamWriter and Streaming Data Sink](#)
5. [StreamingQuery](#)
6. [StreamingQueryManager](#)

DataStreamReader and Streaming Data Source

It all starts with `SparkSession.readStream` method which lets you define a [streaming source](#) in a **stream processing pipeline** (aka *streaming processing graph* or *dataflow graph*).

```
import org.apache.spark.sql.SparkSession
assert(spark instanceof[SparkSession])

val reader = spark.readStream

import org.apache.spark.sql.streaming.DataStreamReader
assert(reader instanceof[DataStreamReader])
```

`SparkSession.readStream` method creates a [DataStreamReader](#).

The fluent API of `DataStreamReader` allows you to describe the input data source (e.g. [DataStreamReader.format](#) and [DataStreamReader.options](#)) using method chaining (with the goal of making the readability of the source code close to that of ordinary written prose, essentially creating a domain-specific language within the interface. See [Fluent interface](#) article in Wikipedia).

```
reader
  .format("csv")
  .option("delimiter", "|")
```

There are a couple of built-in data source formats. Their names are the names of the corresponding `DataStreamReader` methods and so act like shortcuts of `DataStreamReader.format` (where you have to specify the format by name), i.e. `csv`, `json`, `orc`, `parquet` and `text`, followed by `DataStreamReader.load`.

You may also want to use `DataStreamReader.schema` method to specify the schema of the streaming data source.

```
reader.schema("a INT, b STRING")
```

In the end, you use `DataStreamReader.load` method that simply creates a streaming Dataset (the good ol' Dataset that you may have already used in Spark SQL).

```
val input = reader
  .format("csv")
  .option("delimiter", "\t")
  .schema("word STRING, num INT")
  .load("data/streaming")

import org.apache.spark.sql.DataFrame
assert(input.isInstanceOf[DataFrame])
```

The Dataset has the `isStreaming` property enabled that is basically the only way you could distinguish streaming Datasets from regular, batch Datasets.

```
assert(input.isStreaming)
```

In other words, Spark Structured Streaming is designed to extend the features of Spark SQL and let your structured queries be streaming queries.

Data Source Resolution, Streaming Dataset and Logical Query Plan

Being curious about the internals of streaming Datasets is where you start...seeing numbers not humans (sorry, couldn't resist drawing the comparison between `Matrix the movie` and the internals of Spark Structured Streaming).

Whenever you create a Dataset (be it batch in Spark SQL or streaming in Spark Structured Streaming) is when you create a **logical query plan** using the **high-level Dataset DSL**.

A logical query plan is made up of logical operators.

Spark Structured Streaming gives you two logical operators to represent streaming sources, i.e. [StreamingRelationV2](#) and [StreamingRelation](#).

When `DataStreamReader.load` method is executed, `load` first looks up the requested data source (that you specified using `DataStreamReader.format`) and creates an instance of it (*instantiation*). That'd be **data source resolution** step (that I described in...FIXME).

`DataStreamReader.load` is where you can find the intersection of the former [Micro-Batch Stream Processing V1 API](#) with the new [Continuous Stream Processing V2 API](#).

For [MicroBatchReadSupport](#) or [ContinuousReadSupport](#) data sources,

`DataStreamReader.load` creates a logical query plan with a [StreamingRelationV2](#) leaf logical operator. That is the new **V2 code path**.

StreamingRelationV2 Logical Operator for Data Source V2

```
// rate data source is V2
val rates = spark.readStream.format("rate").load
val plan = rates.queryExecution.logical
scala> println(plan.numberedTreeString)
00 StreamingRelationV2 org.apache.spark.sql.execution.streaming.sources.RateStreamProvider@2ed03b1a, rate, [timestamp#12, value#13L]
```

For all other types of streaming data sources, `DataStreamReader.load` creates a logical query plan with a [StreamingRelation](#) leaf logical operator. That is the former **V1 code path**.

StreamingRelation Logical Operator for Data Source V1

```
// text data source is V1
val texts = spark.readStream.format("text").load("data/streaming")
val plan = texts.queryExecution.logical
scala> println(plan.numberedTreeString)
00 StreamingRelation DataSource(org.apache.spark.sql.SparkSession@35edd886, text, List(), None, List(), None, Map(path -> data/streaming), None), FileSource[data/streaming], [value#18]
```

Dataset API—High-Level DSL to Build Logical Query Plan

With a streaming Dataset created, you can now use all the methods of `Dataset` API, including but not limited to the following operators:

- [Dataset.dropDuplicates](#) for streaming deduplication
- [Dataset.groupBy](#) and [Dataset.groupByKey](#) for streaming aggregation
- [Dataset.withWatermark](#) for event time watermark

Please note that a streaming Dataset is a regular Dataset (*with some streaming-related limitations*).

```
val rates = spark
  .readStream
  .format("rate")
  .load
val countByTime = rates
  .withWatermark("timestamp", "10 seconds")
  .groupBy($"timestamp")
  .agg(count("*") as "count")

import org.apache.spark.sql.Dataset
assert(countByTime.isInstanceOf[Dataset[_]])
```

The point is to understand that the Dataset API is a domain-specific language (DSL) to build a more sophisticated stream processing pipeline that you could also build using the low-level logical operators directly.

Use [Dataset.explain](#) to learn the underlying logical and physical query plans.

```

assert(countByTime.isStreaming)

scala> countByTime.explain(extended = true)
== Parsed Logical Plan ==
Aggregate ['timestamp], [unresolvedalias('timestamp, None), count(1) AS count#131L]
+- EventTimeWatermark timestamp#88: timestamp, interval 10 seconds
  +- StreamingRelationV2 org.apache.spark.sql.execution.streaming.sources.RateStreamP
rovider@2fcb3082, rate, [timestamp#88, value#89L]

== Analyzed Logical Plan ==
timestamp: timestamp, count: bigint
Aggregate [timestamp#88-T10000ms], [timestamp#88-T10000ms, count(1) AS count#131L]
+- EventTimeWatermark timestamp#88: timestamp, interval 10 seconds
  +- StreamingRelationV2 org.apache.spark.sql.execution.streaming.sources.RateStreamP
rovider@2fcb3082, rate, [timestamp#88, value#89L]

== Optimized Logical Plan ==
Aggregate [timestamp#88-T10000ms], [timestamp#88-T10000ms, count(1) AS count#131L]
+- EventTimeWatermark timestamp#88: timestamp, interval 10 seconds
  +- Project [timestamp#88]
    +- StreamingRelationV2 org.apache.spark.sql.execution.streaming.sources.RateStre
amProvider@2fcb3082, rate, [timestamp#88, value#89L]

== Physical Plan ==
*(5) HashAggregate(keys=[timestamp#88-T10000ms], functions=[count(1)], output=[timesta
mp#88-T10000ms, count#131L])
+- StateStoreSave [timestamp#88-T10000ms], state info [ checkpoint = <unknown>, runId
= 28606ba5-9c7f-4f1f-ae41-e28d75c4d948, opId = 0, ver = 0, numPartitions = 200], Append
, 0, 2
  +- *(4) HashAggregate(keys=[timestamp#88-T10000ms], functions=[merge_count(1)], out
put=[timestamp#88-T10000ms, count#136L])
    +- StateStoreRestore [timestamp#88-T10000ms], state info [ checkpoint = <unknown
>, runId = 28606ba5-9c7f-4f1f-ae41-e28d75c4d948, opId = 0, ver = 0, numPartitions = 200
], 2
      +- *(3) HashAggregate(keys=[timestamp#88-T10000ms], functions=[merge_count(1)
], output=[timestamp#88-T10000ms, count#136L])
        +- Exchange hashpartitioning(timestamp#88-T10000ms, 200)
          +- *(2) HashAggregate(keys=[timestamp#88-T10000ms], functions=[partial_
count(1)], output=[timestamp#88-T10000ms, count#136L])
            +- EventTimeWatermark timestamp#88: timestamp, interval 10 seconds
              +- *(1) Project [timestamp#88]
                +- StreamingRelation rate, [timestamp#88, value#89L]

```

Or go pro and talk to `QueryExecution` directly.

```

val plan = countByTime.queryExecution.logical
scala> println(plan.numberedTreeString)
00 'Aggregate ['timestamp], [unresolvedalias('timestamp, None), count(1) AS count#131L
]
01 +- EventTimeWatermark timestamp#88: timestamp, interval 10 seconds
02   +- StreamingRelationV2 org.apache.spark.sql.execution.streaming.sources.RateStreamProvider@2fcb3082, rate, [timestamp#88, value#89L]

```

Please note that most of the stream processing operators you may also have used in batch structured queries in Spark SQL. Again, the distinction between Spark SQL and Spark Structured Streaming is very thin from a developer's point of view.

DataStreamWriter and Streaming Data Sink

Once you're satisfied with building a stream processing pipeline (using the APIs of [DataStreamReader](#), [Dataset](#), [RelationalGroupedDataset](#) and [KeyValueGroupedDataset](#)), you should define how and when the result of the streaming query is persisted in (*sent out to*) an external data system using a [streaming sink](#).

Tip

Read up on the APIs of [Dataset](#), [RelationalGroupedDataset](#) and [KeyValueGroupedDataset](#) in [The Internals of Spark SQL](#) book.

You should use [Dataset.writeStream](#) method that simply creates a [DataStreamWriter](#).

```

// Not only is this a Dataset, but it is also streaming
assert(countByTime.isStreaming)

val writer = countByTime.writeStream

import org.apache.spark.sql.streaming.DataStreamWriter
assert(writer.asInstanceOf[DataStreamWriter[_]])

```

The fluent API of [DataStreamWriter](#) allows you to describe the output data sink (e.g. [DataStreamWriter.format](#) and [DataStreamWriter.options](#)) using method chaining (with the goal of making the readability of the source code close to that of ordinary written prose, essentially creating a domain-specific language within the interface. See [Fluent interface](#) article in Wikipedia).

```

writer
  .format("csv")
  .option("delimiter", "\t")

```

Like in [DataStreamReader](#) data source formats, there are a couple of built-in data sink formats. Unlike data source formats, their names do not have corresponding `DataStreamWriter` methods. The reason is that you will use [DataStreamWriter.start](#) to create and immediately start a [StreamingQuery](#).

There are however two special output formats that do have corresponding `DataStreamWriter` methods, i.e. [DataStreamWriter.foreach](#) and [DataStreamWriter.foreachBatch](#), that allow for persisting query results to external data systems that do not have streaming sinks available. They give you a trade-off between developing a full-blown streaming sink and simply using the methods (that lay the basis of what a custom sink would have to do anyway).

`DataStreamWriter` API defines two new concepts (that are not available in the "base" Spark SQL):

- [OutputMode](#) that you specify using [DataStreamWriter.outputMode](#) method
- [Trigger](#) that you specify using [DataStreamWriter.trigger](#) method

You may also want to give a streaming query a name using [DataStreamWriter.queryName](#) method.

In the end, you use [DataStreamWriter.start](#) method to create and immediately start a [StreamingQuery](#).

```
import org.apache.spark.sql.streaming.OutputMode
import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._
val sq = writer
  .format("console")
  .option("truncate", false)
  .option("checkpointLocation", "/tmp/csv-to-csv-checkpoint")
  .outputMode(OutputMode.Append)
  .trigger(Trigger.ProcessingTime(30.seconds))
  .queryName("csv-to-csv")
  .start("/tmp")

import org.apache.spark.sql.streaming.StreamingQuery
assert(sq.isInstanceOf[StreamingQuery])
```

When `DataStreamWriter` is requested to [start a streaming query](#), it allows for the following data source formats:

- **memory** with [MemorySinkV2](#) (with [ContinuousTrigger](#)) or [MemorySink](#)
- **foreach** with [ForeachWriterProvider](#) sink

- **foreachBatch** with [ForeachBatchSink](#) sink (that does not support [ContinuousTrigger](#))
- Any `DataSourceRegister` data source
- Custom data sources specified by their fully-qualified class names or `[name].DefaultSource`
- **avro, kafka and some others** (see `DataSource.lookupDataSource` object method)
- [StreamWriter](#)
- `DataSource` is requested to [create a streaming sink](#) that accepts [StreamSinkProvider](#) or `FileFormat` data sources only

With a streaming sink, `DataStreamWriter` requests the [StreamingQueryManager](#) to [start a streaming query](#).

StreamingQuery

When a stream processing pipeline is started (using `DataStreamWriter.start` method), `DataStreamWriter` creates a [StreamingQuery](#) and requests the [StreamingQueryManager](#) to [start a streaming query](#).

StreamingQueryManager

[StreamingQueryManager](#) is used to manage streaming queries.

Arbitrary Stateful Streaming Aggregation

Arbitrary Stateful Streaming Aggregation is a [streaming aggregation query](#) that uses the following [KeyValueGroupedDataset](#) operators:

- [mapGroupsWithState](#) for implicit state logic
- [flatMapGroupsWithState](#) for explicit state logic

`KeyValueGroupedDataset` represents a grouped dataset as a result of [Dataset.groupByKey](#) operator.

`mapGroupsWithState` and `flatMapGroupsWithState` operators use [GroupState](#) as **group streaming aggregation state** that is created separately for every **aggregation key** with an **aggregation state value** (of a user-defined type).

`mapGroupsWithState` and `flatMapGroupsWithState` operators use [GroupStateTimeout](#) as an **aggregation state timeout** that defines when a [GroupState](#) can be considered **timed-out (expired)**.

Demos

Use the following demos and complete applications to learn more:

- [Demo: Internals of FlatMapGroupsWithStateExec Physical Operator](#)
- [Demo: Arbitrary Stateful Streaming Aggregation with KeyValueGroupedDataset.flatMapGroupsWithState Operator](#)
- [groupByKey Streaming Aggregation in Update Mode](#)
- [FlatMapGroupsWithStateApp](#)

Performance Metrics

Arbitrary Stateful Streaming Aggregation uses **performance metrics** (of the [StateStoreWriter](#) through [FlatMapGroupsWithStateExec](#) physical operator).

Internals

One of the most important internal execution components of Arbitrary Stateful Streaming Aggregation is [FlatMapGroupsWithStateExec](#) physical operator.

When requested to [execute and generate a recipe for a distributed computation](#) (as an [RDD\[InternalRow\]](#)), [FlatMapGroupsWithStateExec](#) first validates a selected [GroupStateTimeout](#):

- For [ProcessingTimeTimeout](#), batch timeout threshold has to be defined
- For [EventTimeTimeout](#), event-time watermark has to be defined and the input schema has the [watermark](#) attribute

Note	FIXME When are the above requirements met?
------	--

[FlatMapGroupsWithStateExec](#) physical operator then [mapPartitionsWithStateStore](#) with a custom [storeUpdateFunction](#) of the following signature:

```
(StateStore, Iterator[T]) => Iterator[U]
```

While generating the recipe, [FlatMapGroupsWithStateExec](#) uses [StateStoreOps](#) extension method object to register a listener that is executed on a task completion. The listener makes sure that a given [StateStore](#) has all state changes either [committed](#) or [aborted](#).

In the end, [FlatMapGroupsWithStateExec](#) creates a new [StateStoreRDD](#) and adds it to the RDD lineage.

[StateStoreRDD](#) is used to properly distribute tasks across executors (per [preferred locations](#)) with help of [StateStoreCoordinator](#) (that runs on the driver).

[StateStoreRDD](#) uses [StateStore](#) helper to look up a [StateStore](#) by [StateStoreProviderId](#) and store version.

[FlatMapGroupsWithStateExec](#) physical operator uses [state managers](#) that are different than [state managers](#) for [Streaming Aggregation](#). [StateStore](#) abstraction is the same as in [Streaming Aggregation](#).

One of the important execution steps is when [InputProcessor](#) (of [FlatMapGroupsWithStateExec](#) physical operator) is requested to [callFunctionAndUpdateState](#). That executes the **user-defined state function** on a per-group state key object, value objects, and a [GroupStateImpl](#).

GroupState — Group State in Arbitrary Stateful Streaming Aggregation

`GroupState` is an abstraction of `group state` (of type `s`) in `Arbitrary Stateful Streaming Aggregation`.

`GroupState` is used with the following `KeyValueGroupedDataset` operations:

- `mapGroupsWithState`
- `flatMapGroupsWithState`

`GroupState` is created separately for every **aggregation key** to hold a state as an **aggregation state value**.

Table 1. GroupState Contract

Method	Description
<code>exists</code>	<pre>exists: Boolean</pre> <p>Checks whether the state value exists or not If not exists, <code>get</code> throws a <code>NoSuchElementException</code>. Use <code>getOption</code> instead.</p>
<code>get</code>	<pre>get: S</pre> <p>Gets the state value if it <code>exists</code> or throws a <code>NoSuchElementException</code></p>
<code>getCurrentProcessingTimeMs</code>	<pre>getCurrentProcessingTimeMs(): Long</pre> <p>Gets the current processing time (as milliseconds in epoch time)</p>
<code>getCurrentWatermarkMs</code>	<pre>getCurrentWatermarkMs(): Long</pre> <p>Gets the current event time watermark (as milliseconds in epoch time)</p>
	<pre>getOption: Option[S]</pre>

	<p>Gets the state value as a Scala <code>option</code> (regardless whether it exists or not)</p> <p><code>getOption</code></p> <ul style="list-style-type: none"> • <code>InputProcessor</code> is requested to <code>callFunctionAndUpdateState</code> (when the row iterator is consumed and a state value has been updated, removed or timeout changed) • <code>GroupStateImpl</code> is requested for the textual representation
<code>hasTimedOut</code>	<pre>hasTimedOut: Boolean</pre> <p>Whether the state (for a given key) has timed out or not.</p> <p>Can only be <code>true</code> when timeouts are enabled using setTimeoutDuration</p>
<code>remove</code>	<pre>remove(): Unit</pre> <p>Removes the state</p>
<code>setTimeoutDuration</code>	<pre>setTimeoutDuration(durationMs: Long): Unit setTimeoutDuration(duration: String): Unit</pre> <p>Specifies the timeout duration for the state key (in millis or as a string, e.g. "10 seconds", "1 hour") for GroupStateTimeout.ProcessingTimeTimeout</p>
<code>setTimeoutTimestamp</code>	<pre>setTimeoutTimestamp(timestamp: java.sql.Date): Unit setTimeoutTimestamp(timestamp: java.sql.Date, additionalDuration: String): Unit setTimeoutTimestamp(timestampMs: Long): Unit setTimeoutTimestamp(timestampMs: Long, additionalDuration: String): Unit</pre> <p>Specifies the timeout timestamp for the state key for GroupStateTimeout.EventTimeTimeout</p>
<code>update</code>	<pre>update(newState: S): Unit</pre> <p>Updates the state (sets the state to a new value)</p>

Note	<p>GroupStateImpl is the default and only known implementation of the GroupState Contract in Spark Structured Streaming.</p>
------	--

GroupStateImpl

`GroupStateImpl` is the default and only known `GroupState` in Spark Structured Streaming.

`GroupStateImpl` holds per-group `state value` of type `s` per group key.

`GroupStateImpl` is `created` when `GroupStateImpl` helper object is requested for the following:

- `createForStreaming`
- `createForBatch`

Creating GroupStateImpl Instance

`GroupStateImpl` takes the following to be created:

- State value (of type `s`)
- Batch processing time
- `eventTimeWatermarkMs`
- `GroupStateTimeout`
- `hasTimedOut` flag
- `watermarkPresent` flag

`GroupStateImpl` initializes the internal properties.

Creating GroupStateImpl for Streaming Query — `createForStreaming` Factory Method

```
createForStreaming[S](  
    optionalValue: Option[S],  
    batchProcessingTimeMs: Long,  
    eventTimeWatermarkMs: Long,  
    timeoutConf: GroupStateTimeout,  
    hasTimedOut: Boolean,  
    watermarkPresent: Boolean): GroupStateImpl[S]
```

`createForStreaming` simply creates a new `GroupStateImpl` with the given input arguments.

Note

`createForStreaming` is used exclusively when `InputProcessor` is requested to `callFunctionAndUpdateState` (when `InputProcessor` is requested to `processNewData` and `processTimedOutState`).

Creating GroupStateImpl for Batch Query**— `createForBatch` Factory Method**

```
createForBatch(  
    timeoutConf: GroupStateTimeout,  
    watermarkPresent: Boolean): GroupStateImpl[Any]
```

`createForBatch` ...FIXME

Note

`createForBatch` is used when...FIXME

Textual Representation — `toString` Method

```
toString: String
```

Note

`toString` is part of the `java.lang.Object` contract for the string representation of the object.

`toString` ...FIXME

Specifying Timeout Duration for ProcessingTimeTimeout**— `setTimeoutDuration` Method**

```
setTimeoutDuration(durationMs: Long): Unit
```

Note

`setTimeoutDuration` is part of the `GroupState Contract` to specify timeout duration for the state key (in millis or as a string).

`setTimeoutDuration` ...FIXME

Specifying Timeout Timestamp for EventTimeTimeout**— `setTimeoutTimestamp` Method**

```
setTimeoutTimestamp(durationMs: Long): Unit
```

Note

`setTimeoutTimestamp` is part of the [GroupState Contract](#) to specify timeout timestamp for the state key.

`setTimeoutTimestamp ...FIXME`

Getting Processing Time

— `getCurrentProcessingTimeMs` Method

`getCurrentProcessingTimeMs(): Long`

Note

`getCurrentProcessingTimeMs` is part of the [GroupState Contract](#) to get the current processing time (as milliseconds in epoch time).

`getCurrentProcessingTimeMs` simply returns the [batchProcessingTimeMs](#).

Updating State — `update` Method

`update(newValue: S): Unit`

Note

`update` is part of the [GroupState Contract](#) to update the state.

`update ...FIXME`

Removing State — `remove` Method

`remove(): Unit`

Note

`remove` is part of the [GroupState Contract](#) to remove the state.

`remove ...FIXME`

Internal Properties

Name	Description
value	FIXME Used when...FIXME
defined	FIXME Used when...FIXME
updated	Updated flag that says whether the state has been updated or not Default: <code>false</code> Disabled (<code>false</code>) when <code>GroupStateImpl</code> is requested to remove the state Enabled (<code>true</code>) when <code>GroupStateImpl</code> is requested to update the state
removed	Removed flag that says whether the state is marked removed or not Default: <code>false</code> Disabled (<code>false</code>) when <code>GroupStateImpl</code> is requested to update the state Enabled (<code>true</code>) when <code>GroupStateImpl</code> is requested to remove the state
timeoutTimestamp	Current timeout timestamp (in millis) for GroupStateTimeout.EventTimeTimeout or GroupStateTimeout.ProcessingTimeTimeout Default: <code>-1</code> Defined using setTimeoutTimestamp (for <code>EventTimeTimeout</code>) and setTimeoutDuration (for <code>ProcessingTimeTimeout</code>)

GroupStateTimeout — Group State Timeout in Arbitrary Stateful Streaming Aggregation

`GroupStateTimeout` represents an **aggregation state timeout** that defines when a `GroupState` can be considered **timed-out (expired)** in `Arbitrary Stateful Streaming Aggregation`.

`GroupStateTimeout` is used with the following `KeyValueGroupedDataset` operations:

- `mapGroupsWithState`
- `flatMapGroupsWithState`

Table 1. GroupStateTimeouts

GroupStateTimeout	Description
<code>EventTimeTimeout</code>	Timeout based on event time Used when...FIXME
<code>NoTimeout</code>	No timeout Used when...FIXME
<code>ProcessingTimeTimeout</code>	Timeout based on processing time <code>FlatMapGroupsWithStateExec</code> physical operator requires that <code>batchTimestampMs</code> is specified when <code>ProcessingTimeTimeout</code> is used. <code>batchTimestampMs</code> is defined when <code>IncrementalExecution</code> is created (with the <code>state</code>). <code>IncrementalExecution</code> is given <code>OffsetSeqMetadata</code> when <code>StreamExecution</code> is requested to run a streaming batch.

StateManager Contract — State Managers for Arbitrary Stateful Streaming Aggregation

`StateManager` is the abstraction of state managers that act as *middlemen* between state stores and the `FlatMapGroupsWithStateExec` physical operator used in Arbitrary Stateful Streaming Aggregation.

Table 1. StateManager Contract

Method	Description
<code>getAllState</code>	<pre>getAllState(store: StateStore): Iterator[StateData]</pre> <p>Retrieves all state data (for all keys) from the <code>StateStore</code> Used exclusively when <code>InputProcessor</code> is requested to <code>processTimedOutState</code></p>
<code>getState</code>	<pre>getState(store: StateStore, keyRow: UnsafeRow): StateData</pre> <p>Gets the state data for the key from the <code>StateStore</code> Used exclusively when <code>InputProcessor</code> is requested to <code>processNewData</code></p>
<code>putState</code>	<pre>putState(store: StateStore, keyRow: UnsafeRow, state: Any, timeoutTimestamp: Long): Unit</pre> <p>Persists (<i>puts</i>) the state value for the key in the <code>StateStore</code> Used exclusively when <code>InputProcessor</code> is requested to <code>callFunctionAndUpdateState</code> (right after all rows have been processed)</p>
<code>removeState</code>	<pre>removeState(store: StateStore, keyRow: UnsafeRow): Unit</pre> <p>Removes the state for the key from the <code>StateStore</code></p>

	<p>Used exclusively when <code>InputProcessor</code> is requested to <code>callFunctionAndUpdateState</code> (right after all rows have been processed)</p>		
<code>stateSchema</code>	<p><code>stateSchema: StructType</code></p> <h3>State schema</h3> <table border="1"> <tr> <td>Note</td><td> <p>It looks like (in <code>StateManager</code> of the <code>FlatMapGroupsWithStateExec</code> physical operator) <code>stateSchema</code> is used for the schema of state value objects (not state keys as they are described by the grouping attributes instead).</p> </td></tr> </table> <p>Used when:</p> <ul style="list-style-type: none"> • <code>FlatMapGroupsWithStateExec</code> physical operator is requested to <code>execute and generate a recipe for a distributed computation (as an RDD[InternalRow])</code> • <code>StateManagerImplBase</code> is requested for the <code>stateDeserializerFunc</code> 	Note	<p>It looks like (in <code>StateManager</code> of the <code>FlatMapGroupsWithStateExec</code> physical operator) <code>stateSchema</code> is used for the schema of state value objects (not state keys as they are described by the grouping attributes instead).</p>
Note	<p>It looks like (in <code>StateManager</code> of the <code>FlatMapGroupsWithStateExec</code> physical operator) <code>stateSchema</code> is used for the schema of state value objects (not state keys as they are described by the grouping attributes instead).</p>		
Note	<code>StateManagerImplBase</code> is the one and only known direct implementation of the <code>StateManager Contract</code> in Spark Structured Streaming.		
Note	<code>StateManager</code> is a Scala sealed trait which means that all the <code>implementations</code> are in the same compilation unit (a single file).		

StateManagerImplV2 — Default StateManager of FlatMapGroupsWithStateExec Physical Operator

`StateManagerImplV2` is a concrete `StateManager` (as a `StateManagerImplBase`) that is used by default in `FlatMapGroupsWithStateExec` physical operator (per `spark.sql.streaming.flatMapGroupsWithState.stateFormatVersion` internal configuration property).

`StateManagerImplV2` is `created` exclusively when `FlatMapGroupsWithStateExecHelper` utility is requested for a `StateManager` (when the `stateFormatVersion` is `2`).

Creating StateManagerImplV2 Instance

`StateManagerImplV2` takes the following to be created:

- State encoder (`ExpressionEncoder[Any]`)
- `shouldStoreTimestamp` flag

`StateManagerImplV2` initializes the [internal properties](#).

State Schema — stateSchema Value

```
stateSchema: StructType
```

Note	<code>stateSchema</code> is part of the StateManager Contract for the schema of the state.
------	--

`stateSchema` ...FIXME

State Serializer — stateSerializerExprs Value

```
stateSerializerExprs: Seq[Expression]
```

Note	<code>stateSerializerExprs</code> is part of the StateManager Contract for the state serializer, i.e. Catalyst expressions to serialize a state object to a row (<code>UnsafeRow</code>).
------	---

`stateSerializerExprs` ...FIXME

State Deserializer — `stateDeserializerExpr` Value

`stateDeserializerExpr`: Expression

Note	<code>stateDeserializerExpr</code> is part of the StateManager Contract for the state deserializer, i.e. a Catalyst expression to deserialize a state object from a row (<code>UnsafeRow</code>).
------	---

`stateDeserializerExpr` ...FIXME

Internal Properties

Name	Description
<code>nestedStateOrdinal</code>	Position of the state in a state row (<code>0</code>) Used when...FIXME
<code>timeoutTimestampOrdinalInRow</code>	Position of the timeout timestamp in a state row (<code>1</code>) Used when...FIXME

StateManagerImplBase

`StateManagerImplBase` is the extension of the [StateManager contract](#) for [state managers](#) of [FlatMapGroupsWithStateExec](#) physical operator with the following features:

- Use Catalyst expressions for [state serialization](#) and [deserialization](#)
- Use `timeoutTimestampOrdinalInRow` when `shouldStoreTimestamp` with the `shouldStoreTimestamp` flag on

Table 1. StateManagerImplBase Contract (Abstract Methods Only)

Method	Description
<code>stateDeserializerExpr</code>	<code>stateDeserializerExpr: Expression</code> State deserializer , i.e. a Catalyst expression to deserialize a state object from a row (<code>UnsafeRow</code>) Used exclusively for the stateDeserializerFunc
<code>stateSerializerExprs</code>	<code>stateSerializerExprs: Seq[Expression]</code> State serializer , i.e. Catalyst expressions to serialize a state object to a row (<code>unsafeRow</code>) Used exclusively for the stateSerializerFunc
<code>timeoutTimestampOrdinalInRow</code>	<code>timeoutTimestampOrdinalInRow: Int</code> Position of the timeout timestamp in a state row Used when <code>StateManagerImplBase</code> is requested to get and set timeout timestamp

Table 2. StateManagerImplBases

StateManagerImplBase	Description
StateManagerImplV1	Legacy StateManager
StateManagerImplV2	Default StateManager

Creating StateManagerImplBase Instance

`StateManagerImplBase` takes a single `shouldStoreTimestamp` flag to be created (that is set when the [concrete StateManagerImplBases](#) are created).

Note

`StateManagerImplBase` is a Scala abstract class and cannot be [created](#) directly. It is created indirectly for the [concrete StateManagerImplBases](#).

`StateManagerImplBase` initializes the [internal properties](#).

Getting State Data for Key from StateStore — `getState` Method

```
getState(  
    store: StateStore,  
    keyRow: UnsafeRow): StateData
```

Note

`getState` is part of the [StateManager Contract](#) to get the state data for the key from the [StateStore](#).

`getState` ...FIXME

Persisting State Value for Key in StateStore — `putState` Method

```
putState(  
    store: StateStore,  
    key: UnsafeRow,  
    state: Any,  
    timestamp: Long): Unit
```

Note

`putState` is part of the [StateManager Contract](#) to persist (*put*) the state value for the key in the [StateStore](#).

`putState` ...FIXME

Removing State for Key from StateStore — `removeState` Method

```
removeState(  
    store: StateStore,  
    keyRow: UnsafeRow): Unit
```

Note

`removeState` is part of the [StateManager Contract](#) to remove the state for the key from the [StateStore](#).

`removeState` ...FIXME

Getting All State Data (for All Keys) from StateStore

— `getAllState` Method

```
getAllState(store: StateStore): Iterator[StateData]
```

Note

`getAllState` is part of the [StateManager Contract](#) to retrieve all state data (for all keys) from the [StateStore](#).

`getAllState` ...FIXME

getStateObject Internal Method

```
getStateObject(row: UnsafeRow): Any
```

`getStateObject` ...FIXME

Note

`getStateObject` is used when...FIXME

getStateRow Internal Method

```
getStateRow(obj: Any): UnsafeRow
```

`getStateRow` ...FIXME

Note

`getStateRow` is used when...FIXME

Getting Timeout Timestamp (from State Row)

— `getTimestamp` Internal Method

```
getTimestamp(stateRow: UnsafeRow): Long
```

`getTimestamp` ...FIXME

Note

`getTimestamp` is used when...FIXME

Setting Timeout Timestamp (to State Row)

— `setTimestamp` Internal Method

```
setTimestamp(
    stateRow: UnsafeRow,
    timeoutTimestamps: Long): Unit
```

`setTimestamp` ...FIXME

Note

`setTimestamp` is used when...FIXME

Internal Properties

Name	Description
<code>stateSerializerFunc</code>	<p>State object serializer (of type <code>Any ⇒ UnsafeRow</code>) to serialize a state object (for a per-group state key) to a row (<code>UnsafeRow</code>)</p> <ul style="list-style-type: none"> The serialization expression (incl. the type) is specified as the <code>stateSerializerExprs</code> <p>Used exclusively in <code>getStateRow</code></p>
<code>stateDeserializerFunc</code>	<p>State object deserializer (of type <code>InternalRow ⇒ Any</code>) to deserialize a row (for a per-group state value) to a Scala value</p> <ul style="list-style-type: none"> The deserialization expression (incl. the type) is specified as the <code>stateDeserializerExpr</code> <p>Used exclusively in <code>getStateObject</code></p>
<code>stateDataForGets</code>	Empty <code>StateData</code> to share (<i>reuse</i>) between <code>getState</code> calls (to avoid high use of memory with many <code>StateData</code> objects)

StateManagerImplV1

StateManagerImplV1 is...FIXME

FlatMapGroupsWithStateExecHelper

`FlatMapGroupsWithStateExecHelper` is a utility with the main purpose of creating a `StateManager` for `FlatMapGroupsWithStateExec` physical operator.

Creating StateManager — `createStateManager` Method

```
createStateManager(  
    stateEncoder: ExpressionEncoder[Any],  
    shouldStoreTimestamp: Boolean,  
    stateFormatVersion: Int): StateManager
```

`createStateManager` simply creates a `StateManager` (with the `stateEncoder` and `shouldStoreTimestamp` flag) based on `stateFormatVersion`:

- `StateManagerImplV1` for 1
- `StateManagerImplV2` for 2

`createStateManager` throws an `IllegalArgumentException` for `stateFormatVersion` not 1 or 2:

```
Version [stateFormatVersion] is invalid
```

Note

`createStateManager` is used exclusively for the `StateManager` for `FlatMapGroupsWithStateExec` physical operator.

InputProcessor Helper Class of FlatMapGroupsWithStateExec Physical Operator

`InputProcessor` is a helper class to manage state in the `state store` for every partition of a `FlatMapGroupsWithStateExec` physical operator.

`InputProcessor` is `created` exclusively when `FlatMapGroupsWithStateExec` physical operator is requested to `execute and generate a recipe for a distributed computation (as an RDD[InternalRow])` (and uses `InputProcessor` in the `storeUpdateFunction` while processing rows per partition with a corresponding per-partition state store).

`InputProcessor` takes a single `StateStore` to be created. The `statestore` manages the per-group state (and is used when processing `new data` and `timed-out state data`, and in the "all rows processed" callback).

Processing New Data (Creating Iterator of New Data Processed) — `processNewData` Method

```
processNewData(dataIter: Iterator[InternalRow]): Iterator[InternalRow]
```

`processNewData` creates a grouped iterator of (of pairs of) per-group state keys and the row values from the given data iterator (`dataIter`) with the `grouping attributes` and the output schema of the `child operator` (of the parent `FlatMapGroupsWithStateExec` physical operator).

For every per-group state key (in the grouped iterator), `processNewData` requests the `StateManager` (of the parent `FlatMapGroupsWithStateExec` physical operator) to `get the state` (from the `StateStore`) and `callFunctionAndUpdateState` (with the `hasTimedOut` flag off, i.e. `false`).

Note

`processNewData` is used exclusively when `FlatMapGroupsWithStateExec` physical operator is requested to `execute and generate a recipe for a distributed computation (as an RDD[InternalRow])`.

Processing Timed-Out State Data (Creating Iterator of Timed-Out State Data) — `processTimedOutState` Method

```
processTimedOutState(): Iterator[InternalRow]
```

`processTimedOutState` does nothing and simply returns an empty iterator for `GroupStateTimeout.NoTimeout`.

With `timeout` enabled, `processTimedOutState` gets the current timeout threshold per `GroupStateTimeout`:

- `batchTimestampMs` for `ProcessingTimeTimeout`
- `eventTimeWatermark` for `EventTimeTimeout`

`processTimedOutState` creates an iterator of timed-out state data by requesting the `StateManager` for all the available state data (in the `StateStore`) and takes only the state data with timeout defined and below the current timeout threshold.

In the end, for every timed-out state data, `processTimedOutState` `callFunctionAndUpdateState` (with the `hasTimedOut` flag enabled).

Note	<code>processTimedOutState</code> is used exclusively when <code>FlatMapGroupsWithStateExec</code> physical operator is requested to execute and generate a recipe for a distributed computation (as an <code>RDD[InternalRow]</code>).
------	--

callFunctionAndUpdateState Internal Method

```
callFunctionAndUpdateState(
    stateData: StateData,
    valueRowIter: Iterator[InternalRow],
    hasTimedOut: Boolean): Iterator[InternalRow]
```

Note	<code>callFunctionAndUpdateState</code> is used when <code>InputProcessor</code> is requested to process new data and timed-out state data. When processing new data, <code>hasTimedOut</code> flag is off (<code>false</code>). When processing timed-out state data, <code>hasTimedOut</code> flag is on (<code>true</code>).
------	---

`callFunctionAndUpdateState` creates a key object by requesting the given `StateData` for the `UnsafeRow` of the key (`keyRow`) and converts it to an object (using the internal `state key converter`).

`callFunctionAndUpdateState` creates value objects by taking every value row (from the given `valueRowIter` iterator) and converts them to objects (using the internal `state value converter`).

`callFunctionAndUpdateState` creates a new `GroupStateImpl` with the following:

- The current state value (of the given `StateData`) that could possibly be `null`
- The `batchTimestampMs` of the parent `FlatMapGroupsWithStateExec` operator (that could possibly be `-1`)
- The `event-time watermark` of the parent `FlatMapGroupsWithStateExec` operator (that could possibly be `-1`)
- The `GroupStateTimeout` of the parent `FlatMapGroupsWithStateExec` operator
- The `watermarkPresent` flag of the parent `FlatMapGroupsWithStateExec` operator
- The given `hasTimedOut` flag

`callFunctionAndUpdateState` then executes the `user-defined state function` (of the parent `FlatMapGroupsWithStateExec` operator) on the key object, value objects, and the newly-created `GroupStateImpl`.

For every output value from the user-defined state function, `callFunctionAndUpdateState` updates `numOutputRows` performance metric and wraps the values to an internal row (using the internal `output value converter`).

In the end, `callFunctionAndUpdateState` returns a `Iterator[InternalRow]` which calls the `completion function` right after rows have been processed (so the iterator is considered fully consumed).

"All Rows Processed" Callback — `onIteratorCompletion` Internal Method

```
onIteratorCompletion: Unit
```

`onIteratorCompletion` branches off per whether the `GroupStateImpl` has been marked `removed` and no `timeout timestamp` is specified or not.

When the `GroupStateImpl` has been marked `removed` and no `timeout timestamp` is specified, `onIteratorCompletion` does the following:

1. Requests the `StateManager` (of the parent `FlatMapGroupsWithStateExec` operator) to remove the state (from the `StateStore` for the key row of the given `StateData`)
2. Increments the `numUpdatedStateRows` performance metric

Otherwise, when the `GroupStateImpl` has not been marked `removed` or the `timeout timestamp` is specified, `onIteratorCompletion` checks whether the timeout timestamp has changed by comparing the timeout timestamps of the `GroupStateImpl` and the given `StateData`.

(only when the `GroupStateImpl` has been `updated`, `removed` or the timeout timestamp changed) `onIteratorCompletion` does the following:

1. Requests the `StateManager` (of the parent `FlatMapGroupsWithStateExec` operator) to `persist the state` (in the `StateStore` with the key row, updated state object, and the timeout timestamp of the given `StateData`)
2. Increments the `numUpdatedStateRows` performance metrics

Note	<code>onIteratorCompletion</code> is used exclusively when <code>InputProcessor</code> is requested to <code>callFunctionAndUpdateState</code> (right after rows have been processed)
------	---

Internal Properties

Name	Description
getKeyObj	<p>A state key converter (of type <code>InternalRow = Any</code>) to deserialize a given row (for a per-group state key) to the current state value</p> <ul style="list-style-type: none"> The deserialization expression for keys is specified as the key deserializer expression when the parent <code>FlatMapGroupsWithStateExec</code> operator is created The data type of state keys is specified as the grouping attributes when the parent <code>FlatMapGroupsWithStateExec</code> operator is created <p>Used exclusively when <code>InputProcessor</code> is requested to callFunctionAndUpdateState.</p>
getOutputRow	<p>A output value converter (of type <code>Any = InternalRow</code>) to wrap a given output value (from the user-defined state function) to a row</p> <ul style="list-style-type: none"> The data type of the row is specified as the data type of the output object attribute when the parent <code>FlatMapGroupsWithStateExec</code> operator is created <p>Used exclusively when <code>InputProcessor</code> is requested to callFunctionAndUpdateState.</p>
getValueObj	<p>A state value converter (of type <code>InternalRow = Any</code>) to deserialize a given row (for a per-group state value) to a Scala value</p> <ul style="list-style-type: none"> The deserialization expression for values is specified as the value deserializer expression when the parent <code>FlatMapGroupsWithStateExec</code> operator is created The data type of state values is specified as the data attributes when the parent <code>FlatMapGroupsWithStateExec</code> operator is created <p>Used exclusively when <code>InputProcessor</code> is requested to callFunctionAndUpdateState.</p>
numOutputRows	<code>numOutputRows</code> performance metric

Demos

1. Demo: Internals of FlatMapGroupsWithStateExec Physical Operator
2. Demo: Exploring Checkpointed State
3. Demo: Streaming Watermark with Aggregation in Append Output Mode
4. Demo: Streaming Query for Running Counts (Socket Source and Complete Output Mode)
5. Demo: Streaming Join of Streaming Queries and StreamingSymmetricHashJoinExec Physical Operator
6. Demo: Streaming Aggregation with Kafka Data Source
7. Demo: groupByKey Streaming Aggregation in Update Mode
8. Demo: StateStoreSaveExec with Complete Output Mode
9. Demo: StateStoreSaveExec with Update Output Mode
10. Developing Custom Streaming Sink (and Monitoring SQL Queries in web UI)
11. current_timestamp Function For Processing Time in Streaming Queries
12. Using StreamingQueryManager for Query Termination Management

Demo: Internals of FlatMapGroupsWithStateExec Physical Operator

The following demo shows the internals of [FlatMapGroupsWithStateExec](#) physical operator in a [Arbitrary Stateful Streaming Aggregation](#).

```
// Reduce the number of partitions and hence the state stores
// That is supposed to make debugging state checkpointing easier
val numShufflePartitions = 1
import org.apache.spark.sql.internal.SQLConf.SHUFFLE_PARTITIONS
spark.sessionState.conf.setConf(SHUFFLE_PARTITIONS, numShufflePartitions)

assert(spark.sessionState.conf.numShufflePartitions == numShufflePartitions)

// Define event "format"
// Use :paste mode in spark-shell
import java.sql.Timestamp
case class Event(time: Timestamp, value: Long)
import scala.concurrent.duration._
object Event {
  def apply(secs: Long, value: Long): Event = {
    Event(new Timestamp(secs.seconds.toMillis), value)
  }
}

// Using memory data source for full control of the input
import org.apache.spark.sql.execution.streaming.MemoryStream
implicit val sqlCtx = spark.sqlContext
val events = MemoryStream[Event]
val values = events.toDS
assert(values.isStreaming, "values must be a streaming Dataset")

values.printSchema
/*
root
 |-- time: timestamp (nullable = true)
 |-- value: long (nullable = false)
 */

import scala.concurrent.duration._
val delayThreshold = 10.seconds
val valuesWatermarked = values
  .withWatermark(eventTime = "time", delayThreshold.toString) // required for EventTim
eTimeout

// Could use Long directly, but...
```

```

// Let's use case class to make the demo a bit more advanced
case class Count(value: Long)

import java.sql.Timestamp
import org.apache.spark.sql.streaming.GroupState
val keyCounts = (key: Long, values: Iterator[(Timestamp, Long)], state: GroupState[Count]) => {
    println(s""">>>> keyCounts(key = $key, state = ${state.getOption.getOrElse("<empty>")})
})"""
    println(s">>> >>> currentProcessingTimeMs: ${state.getCurrentProcessingTimeMs}")
    println(s">>> >>> currentWatermarkMs: ${state.getCurrentWatermarkMs}")
    println(s">>> >>> hasTimedOut: ${state.hasTimedOut}")
    val count = Count(values.length)
    Iterator((key, count))
}

import org.apache.spark.sql.streaming.{GroupStateTimeout, OutputMode}
val valuesCounted = valueswatermarked
    .as[(Timestamp, Long)] // convert DataFrame to Dataset to make groupByKey easier to
write
    .groupByKey { case (time, value) => value }
    .flatMapGroupsWithState(
        OutputMode.Update,
        timeoutConf = GroupStateTimeout.EventTimeTimeout(func = keyCounts)
    .toDF("value", "count")

valuesCounted.explain
/**
== Physical Plan ==
*(2) Project [_1#928L AS value#931L, _2#929 AS count#932]
+- *(2) SerializeFromObject [assertnonnull(input[0, scala.Tuple2, true])._1 AS _1#928L
, if (isnull(assertnonnull(input[0, scala.Tuple2, true])._2)) null else named_struct(v
alue, assertnonnull(assertnonnull(input[0, scala.Tuple2, true])._2).value) AS _2#929]
    +- FlatMapGroupsWithState $line140.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$Lambda$4117/181063008@d2cdc82, value#923: bigint, newInstance(class s
cala.Tuple2), [value#923L], [time#915-T10000ms, value#916L], obj#927: scala.Tuple2, st
ate info [ checkpoint = <unknown>, runId = 9af3d00c-fe1f-46a0-8630-4e0d0af88042, opId
= 0, ver = 0, numPartitions = 1], class[value[0]: bigint], 2, Update, EventTimeTimeout
, 0, 0
        +- *(1) Sort [value#923L ASC NULLS FIRST], false, 0
            +- Exchange hashpartitioning(value#923L, 1)
                +- AppendColumns $line140.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$Lambda$4118/2131767153@3e606b4c, newInstance(class scala.Tuple2), [in
put[0, bigint, false] AS value#923L]
                    +- EventTimeWatermark time#915: timestamp, interval 10 seconds
                        +- StreamingRelation MemoryStream[time#915,value#916L], [time#915, v
alue#916L]
*/
val queryName = "FlatMapGroupsWithStateExec_demo"
val checkpointLocation = s"/tmp/checkpoint-$queryName"

```

```

// Delete the checkpoint location from previous executions
import java.nio.file.{Files, FileSystems}
import java.util.Comparator
import scala.collection.JavaConverters._
val path = FileSystems.getDefault.getPath(checkpointLocation)
if (Files.exists(path)) {
    Files.walk(path)
        .sorted(Comparator.reverseOrder())
        .iterator
        .asScala
        .foreach(p => p.toFile.delete)
}

import org.apache.spark.sql.streaming.OutputMode.Update
val streamingQuery = valuesCounted
    .writeStream
    .format("memory")
    .queryName(queryName)
    .option("checkpointLocation", checkpointLocation)
    .outputMode(update)
    .start

assert(streamingQuery.status.message == "Waiting for data to arrive")

// Use web UI to monitor the metrics of the streaming query
// Go to http://localhost:4040/SQL/ and click one of the Completed Queries with Job IDs

// You may also want to check out checkpointed state
// in /tmp/checkpoint-FlatMapGroupsWithStateExec_demo/state/0/0

val batch = Seq(
    Event(secs = 1, value = 1),
    Event(secs = 15, value = 2))
events.addData(batch)
streamingQuery.processAllAvailable()

/**
>>> keyCounts(key = 1, state = <empty>)
>>> >>> currentProcessingTimeMs: 1561881557237
>>> >>> currentWatermarkMs: 0
>>> >>> hasTimedOut: false
>>> keyCounts(key = 2, state = <empty>)
>>> >>> currentProcessingTimeMs: 1561881557237
>>> >>> currentWatermarkMs: 0
>>> >>> hasTimedOut: false
*/
spark.table(queryName).show(truncate = false)
/**+
+-----+-----+
|value|count|

```

```

+-----+
|1    | [1]  |
|2    | [1]  |
+-----+
*/
// With at least one execution we can review the execution plan
streamingQuery.explain
/** 
== Physical Plan ==
*(2) Project [_1#928L AS value#931L, _2#929 AS count#932]
+- *(2) SerializeFromObject [assertnotnull(input[0, scala.Tuple2, true])._1 AS _1#928L
, if (isnull(assertnotnull(input[0, scala.Tuple2, true])._2)) null else named_struct(v
alue, assertnotnull(assertnotnull(input[0, scala.Tuple2, true])._2).value) AS _2#929]
   +- FlatMapGroupsWithState $line140.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$Lambda$4117/181063008@d2cdc82, value#923: bigint, newInstance(class s
cala.Tuple2), [value#923L], [time#915-T10000ms, value#916L], obj#927: scala.Tuple2, st
ate info [ checkpoint = file:/tmp/checkpoint-FlatMapGroupsWithStateExec_demo/state, ru
nId = 95c3917c-2fd7-45b2-86f6-6c001f0115e1d, opId = 0, ver = 1, numPartitions = 1], cla
ss[value[0]: bigint], 2, Update, EventTimeTimeout, 1561881557499, 5000
   +- *(1) Sort [value#923L ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(value#923L, 1)
         +- AppendColumns $line140.$read$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$iw$$Lambda$4118/2131767153@3e606b4c, newInstance(class scala.Tuple2), [in
put[0, bigint, false] AS value#923L]
            +- EventTimeWatermark time#915: timestamp, interval 10 seconds
               +- LocalTableScan <empty>, [time#915, value#916L]
*/
type Millis = Long
def toMillis(datetime: String): Millis = {
  import java.time.format.DateTimeFormatter
  import java.time.LocalDateTime
  import java.time.ZoneOffset
  LocalDateTime
    .parse(datetime, DateTimeFormatter.ISO_DATE_TIME)
    .toInstant(ZoneOffset.UTC)
    .toEpochMilli
}
val currentWatermark = streamingQuery.lastProgress.eventTime.get("watermark")
val currentWatermarkSecs = toMillis(currentWatermark).millis.toSeconds.seconds

val expectedWatermarkSecs = 5.seconds
assert(currentWatermarkSecs == expectedWatermarkSecs, s"Current event-time watermark i
s $currentWatermarkSecs, but should be $expectedWatermarkSecs (maximum event time - de
layThreshold ${delayThreshold.toMillis})")

// Let's access the FlatMapGroupsWithStateExec physical operator
import org.apache.spark.sql.execution.streaming.StreamingQueryWrapper
import org.apache.spark.sql.execution.streaming.StreamExecution

```

```

val engine: StreamExecution = streamingQuery
  .asInstanceOf[StreamingQueryWrapper]
  .streamingQuery

import org.apache.spark.sql.execution.streaming.IncrementalExecution
val lastMicroBatch: IncrementalExecution = engine.lastExecution

// Access executedPlan that is the optimized physical query plan ready for execution
// All streaming optimizations have been applied at this point
val plan = lastMicroBatch.executedPlan

// Find the FlatMapGroupsWithStateExec physical operator
import org.apache.spark.sql.execution.streaming.FlatMapGroupsWithStateExec
val flatMapOp = plan.collect { case op: FlatMapGroupsWithStateExec => op }.head

// Display metrics
import org.apache.spark.sql.execution.metric.SQLMetric
def formatMetrics(name: String, metric: SQLMetric) = {
  val desc = metric.name.getOrElse("")
  val value = metric.value
  f"$name%-30s | $desc%-69s | $value%-10s"
}
flatMapOp.metrics.map { case (name, metric) => formatMetrics(name, metric) }.foreach(println)
/***
| numTotalStateRows           | number of total state rows
| 0
| stateMemory                | memory used by state total (min, med, max)
| 390
| loadedMapCacheHitCount     | count of cache hit on states cache in provider
| 1
| numOutputRows               | number of output rows
| 0
| stateOnCurrentVersionSizeBytes | estimated size of state only on current version total (min, med, max) | 102
| loadedMapCacheMissCount     | count of cache miss on states cache in provider
| 0
| commitTimeMs                 | time to commit changes total (min, med, max)
| -2
| allRemovalsTimeMs            | total time to remove rows total (min, med, max)
| -2
| numUpdatedStateRows          | number of updated state rows
| 0
| allUpdatesTimeMs             | total time to update rows total (min, med, max)
| -2
*/
val batch = Seq(
  Event(secs = 1, value = 1), // under the watermark (5000 ms) so it's disregarded
  Event(secs = 6, value = 3)) // above the watermark so it should be counted
events.addData(batch)
streamingQuery.processAllAvailable()

```

```

/**
>>> keyCounts(key = 3, state = <empty>
>>> >>> currentProcessingTimeMs: 1561881643568
>>> >>> currentWatermarkMs: 5000
>>> >>> hasTimedOut: false
*/

```

```

spark.table(queryName).show(truncate = false)
/***
+----+----+
|value|count|
+----+----+
|1    |[1]   |
|2    |[1]   |
|3    |[1]   |
+----+----+
*/

```

```

val batch = Seq(
  Event(secs = 17, value = 3)) // advances the watermark
events.addData(batch)
streamingQuery.processAllAvailable()

```

```

/***
>>> keyCounts(key = 3, state = <empty>
>>> >>> currentProcessingTimeMs: 1561881672887
>>> >>> currentWatermarkMs: 5000
>>> >>> hasTimedOut: false
*/

```

```

val currentWatermark = streamingQuery.lastProgress.eventTime.get("watermark")
val currentWatermarkSecs = toMillis(currentWatermark).millis.toSeconds.seconds

```

```

val expectedWatermarkSecs = 7.seconds
assert(currentWatermarkSecs == expectedWatermarkSecs, s"Current event-time watermark is $currentWatermarkSecs, but should be $expectedWatermarkSecs (maximum event time - delayThreshold ${delayThreshold.toMillis})")

```

```

spark.table(queryName).show(truncate = false)
/***
+----+----+
|value|count|
+----+----+
|1    |[1]   |
|2    |[1]   |
|3    |[1]   |
|3    |[1]   |
+----+----+
*/

```

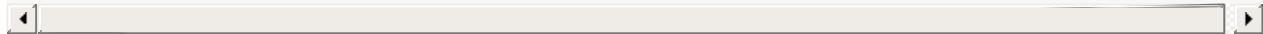
```

val batch = Seq(
  Event(secs = 18, value = 3)) // advances the watermark
events.addData(batch)

```

```
streamingQuery.processAllAvailable()

/*
>>> keyCounts(key = 3, state = <empty>)
>>> >>> currentProcessingTimeMs: 1561881778165
>>> >>> currentWatermarkMs: 7000
>>> >>> hasTimedOut: false
*/
// Eventually...
streamingQuery.stop()
```



Demo: Arbitrary Stateful Streaming Aggregation with KeyValueGroupedDataset.flatMapGroupsWithState Operator

The following demo shows an example of [Arbitrary Stateful Streaming Aggregation](#) with `KeyValueGroupedDataset.flatMapGroupsWithState` operator.

```
import java.sql.Timestamp
type DeviceId = Long
case class Signal(timestamp: Timestamp, deviceId: DeviceId, value: Long)

// input stream
import org.apache.spark.sql.functions._
val signals = spark
  .readStream
  .format("rate")
  .option("rowsPerSecond", 1)
  .load
  .withColumn("deviceId", rint(rand() * 10) cast "int") // 10 devices randomly assigned to values
  .withColumn("value", $"value" % 10) // randomize the values (just for fun)
  .as[Signal] // convert to our type (from "unpleasant" Row)

import org.apache.spark.sql.streaming.GroupState
type Key = Int
type Count = Long
type State = Map[Key, Count]
case class EventsCounted(deviceId: DeviceId, count: Long)
def countValuesPerDevice(
  deviceId: Int,
  signals: Iterator[Signal],
  state: GroupState[State]): Iterator[EventsCounted] = {
  val values = signals.toSeq
  println(s"Device: $deviceId")
  println(s"Signals (${values.size}):")
  values.zipWithIndex.foreach { case (v, idx) => println(s"$idx. $v") }
  println(s"State: $state")

  // update the state with the count of elements for the key
  val initialState: State = Map(deviceId -> 0)
  val oldState = state.getOption.getOrElse(initialState)
  // the name to highlight that the state is for the key only
  val newValue = oldState(deviceId) + values.size
  val newState = Map(deviceId -> newValue)
  state.update(newState)
}
```

Arbitrary Stateful Streaming Aggregation with KeyValueGroupedDataset.flatMapGroupsWithState Operator

```
// you must not return as it's already consumed
// that leads to a very subtle error where no elements are in an iterator
// iterators are one-pass data structures
Iterator(EventsCounted(deviceId, newValue))
}

// stream processing using flatMapGroupsWithState operator
val deviceId: Signal => DeviceId = { case Signal(_, deviceId, _) => deviceId }
val signalsByDevice = signals.groupByKey(deviceId)

import org.apache.spark.sql.streaming.{GroupStateTimeout, OutputMode}
val signalCounter = signalsByDevice.flatMapGroupsWithState(
    outputMode = OutputMode.Append,
    timeoutConf = GroupStateTimeout.NoTimeout)(countValuesPerDevice)

import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val sq = signalCounter.
    writeStream.
    format("console").
    option("truncate", false).
    trigger(Trigger.ProcessingTime(10.seconds)).
    outputMode(OutputMode.Append).
    start
```

Demo: Exploring Checkpointed State

The following demo shows the internals of the checkpointed state of a [stateful streaming query](#).

The demo uses the state checkpoint directory that was used in [Demo: Streaming Watermark with Aggregation in Append Output Mode](#).

```
// Change the path to match your configuration
val checkpointRootLocation = "/tmp/checkpoint-watermark_demo/state"
val version = 1L

import org.apache.spark.sql.execution.streaming.state.StateStoreId
val storeId = StateStoreId(
    checkpointRootLocation,
    operatorId = 0,
    partitionId = 0)

// The key and value schemas should match the watermark demo
// .groupBy(window($"time", windowDuration.toString) as "sliding_window")
import org.apache.spark.sql.types.{TimestampType, StructField, StructType}
val keySchema = StructType(
    StructField("sliding_window",
        StructType(
            StructField("start", TimestampType, nullable = true) :::
            StructField("end", TimestampType, nullable = true) :: Nil),
        nullable = false) :: Nil)
scala> keySchema.printTreeString
root
|-- sliding_window: struct (nullable = false)
|   |-- start: timestamp (nullable = true)
|   |-- end: timestamp (nullable = true)

// .agg(collect_list("batch") as "batches", collect_list("value") as "values")
import org.apache.spark.sql.types.{ArrayType, LongType}
val valueSchema = StructType(
    StructField("batches", ArrayType(LongType, true), true) :::
    StructField("values", ArrayType(LongType, true), true) :: Nil)
scala> valueSchema.printTreeString
root
|-- batches: array (nullable = true)
|   |-- element: long (containsNull = true)
|-- values: array (nullable = true)
|   |-- element: long (containsNull = true)

val indexOrdinal = None
import org.apache.spark.sql.execution.streaming.state.StateStoreConf
val storeConf = StateStoreConf(spark.sessionState.conf)
val hadoopConf = spark.sessionState.newHadoopConf()
```

```

import org.apache.spark.sql.execution.streaming.state.StateStoreProvider
val provider = StateStoreProvider.createAndInit(
  storeId, keySchema, valueSchema, indexOrdinal, storeConf, hadoopConf)

// You may want to use the following higher-level code instead
import java.util.UUID
val queryRunId = UUID.randomUUID
import org.apache.spark.sql.execution.streaming.state.StateStoreProviderId
val storeProviderId = StateStoreProviderId(storeId, queryRunId)
import org.apache.spark.sql.execution.streaming.state.StateStore
val store = StateStore.get(
  storeProviderId,
  keySchema,
  valueSchema,
  indexOrdinal,
  version,
  storeConf,
  hadoopConf)

import org.apache.spark.sql.execution.streaming.state.UnsafeRowPair
def formatRowPair(rowPair: UnsafeRowPair) = {
  s"${rowPair.key.getLong(0)}, ${rowPair.value.getLong(0)}"
}
store.iterator.map(formatRowPair).foreach(println)

// WIP: Missing value (per window)
def formatRowPair(rowPair: UnsafeRowPair) = {
  val window = rowPair.key.getStruct(0, 2)
  import scala.concurrent.duration._
  val begin = window.getLong(0).millis.toSeconds
  val end = window.getLong(1).millis.toSeconds

  val value = rowPair.value.getStruct(0, 4)
  // input is (time, value, batch) all longs
  val t = value.getLong(1).millis.toSeconds
  val v = value.getLong(2)
  val b = value.getLong(3)
  s"(key: [$begin, $end], ($t, $v, $b))"
}
store.iterator.map(formatRowPair).foreach(println)

```

Demo: Streaming Watermark with Aggregation in Append Output Mode

The following demo shows the internals of [streaming watermark](#) with a [streaming aggregation](#) in [append](#) output mode.

```
// Reduce the number of partitions and hence the state stores
// That is supposed to make debugging state checkpointing easier
val numShufflePartitions = 1
import org.apache.spark.sql.internal.SQLConf.SHUFFLE_PARTITIONS
spark.sessionState.conf.setConf(SHUFFLE_PARTITIONS, numShufflePartitions)

assert(spark.sessionState.conf.numShufflePartitions == numShufflePartitions)

// Define event "format"
// Use :paste mode in spark-shell
import java.sql.Timestamp
case class Event(time: Timestamp, value: Long, batch: Long)
import scala.concurrent.duration._
object Event {
  def apply(secs: Long, value: Long, batch: Long): Event = {
    Event(new Timestamp(secs.seconds.toMillis), value, batch)
  }
}

// Using memory data source for full control of the input
import org.apache.spark.sql.execution.streaming.MemoryStream
implicit val sqlCtx = spark.sqlContext
val events = MemoryStream[Event]
val values = events.toDS
assert(values.isStreaming, "values must be a streaming Dataset")

values.printSchema
/**
root
 |-- time: timestamp (nullable = true)
 |-- value: long (nullable = false)
 |-- batch: long (nullable = false)
 */

import scala.concurrent.duration._
val delayThreshold = 10.seconds
val valueswatermarked = values
  .withWatermark(eventTime = "time", delayThreshold.toString) // defines watermark (before groupBy!)

// EventTimeWatermark logical operator is planned as EventTimeWatermarkExec physical operator
```

```

// Note that as a physical operator EventTimeWatermarkExec shows itself without the Exec suffix
valuesWatermarked.explain
/**
== Physical Plan ==
EventTimeWatermark time#3: timestamp, interval 10 seconds
+- StreamingRelation MemoryStream[time#3,value#4L,batch#5L], [time#3, value#4L, batch#5L]
 */

val windowDuration = 5.seconds
import org.apache.spark.sql.functions.window
val countsPer5secWindow = valuesWatermarked
  .groupBy(window($"time", windowDuration.toString) as "sliding_window")
  .agg(collect_list("batch") as "batches", collect_list("value") as "values")

countsPer5secWindow.printSchema
/**
root
 |-- sliding_window: struct (nullable = false)
 |   |-- start: timestamp (nullable = true)
 |   |-- end: timestamp (nullable = true)
 |-- batches: array (nullable = true)
 |   |-- element: long (containsNull = true)
 |-- values: array (nullable = true)
 |   |-- element: long (containsNull = true)
 */

```

```

val queryName = "watermark_demo"
val checkpointLocation = s"/tmp/checkpoint-$queryName"

// Delete the checkpoint location from previous executions
import java.nio.file.{Files, FileSystems}
import java.util.Comparator
import scala.collection.JavaConverters._
val path = FileSystems.getDefault.getPath(checkpointLocation)
if (Files.exists(path)) {
  Files.walk(path)
    .sorted(Comparator.reverseOrder())
    .iterator
    .asScala
    .foreach(p => p.toFile.delete)
}

// FIXME Use foreachBatch for batchId and the output Dataset
import org.apache.spark.sql.streaming.OutputMode.Append
val streamingQuery = countsPer5secWindow
  .writeStream
  .format("memory")
  .queryName(queryName)
  .option("checkpointLocation", checkpointLocation)
  .outputMode(Append)
  .start

```

```

assert(streamingQuery.status.message == "Waiting for data to arrive")

type Millis = Long
def toMillis(datetime: String): Millis = {
  import java.time.format.DateTimeFormatter
  import java.time.LocalDateTime
  import java.time.ZoneOffset
  LocalDateTime
    .parse(datetime, DateTimeFormatter.ISO_DATE_TIME)
    .toInstant(ZoneOffset.UTC)
    .toEpochMilli
}

// Use web UI to monitor the state of state (no pun intended)
// StateStoreSave and StateStoreRestore operators all have state metrics
// Go to http://localhost:4040/SQL/ and click one of the Completed Queries with Job IDs

// You may also want to check out checkpointed state
// in /tmp/checkpoint-watermark_demo/state/0/0

// The demo is aimed to show the following:
// 1. The current watermark
// 2. Check out the stats:
// - expired state (below the current watermark, goes to output and purged later)
// - late state (dropped as if never received and processed)
// - saved state rows (above the current watermark)

val batch = Seq(
  Event(1, 1, batch = 1),
  Event(15, 2, batch = 1))
events.addData(batch)
streamingQuery.processAllAvailable()

val currentWatermark = streamingQuery.lastProgress.eventTime.get("watermark")
val currentWatermarkMs = toMillis(currentWatermark)

val maxTime = batch.maxBy(_.time.toInstant.toEpochMilli).time.toInstant.toEpochMilli.millis.toSeconds
val expectedMaxTime = 15
assert(maxTime == expectedMaxTime, s"Maximum time across events per batch is $maxTime, but should be $expectedMaxTime")

val expectedWatermarkMs = 5.seconds.toMillis
assert(currentWatermarkMs == expectedWatermarkMs, s"Current event-time watermark is $currentWatermarkMs, but should be $expectedWatermarkMs (maximum event time ${maxTime.seconds.toMillis} minus delayThreshold ${delayThreshold.toMillis})")

// FIXME Saved State Rows
// Use the metrics of the StateStoreSave operator
// Or simply streamingQuery.lastProgress.stateOperators.head
spark.table(queryName).orderBy("sliding_window").show(truncate = false)

```

```

/**
+-----+-----+
|sliding_window|batches|values|
+-----+-----+
|[1970-01-01 01:00:00, 1970-01-01 01:00:05]| [1] | [1] |
+-----+-----+
*/

// With at least one execution we can review the execution plan
streamingQuery.explain
/***
== Physical Plan ==
ObjectHashAggregate(keys=[window#21-T10000ms], functions=[collect_list(batch#5L, 0, 0)
, collect_list(value#4L, 0, 0)])
+- StateStoreSave [window#21-T10000ms], state info [ checkpoint = file:/tmp/checkpoint
-watermark_demo/state, runId = f1b3f7a6-95a9-4a15-af06-13325784b5b4, opId = 0, ver = 1
, numPartitions = 1], Append, 5000, 2
    +- ObjectHashAggregate(keys=[window#21-T10000ms], functions=[merge_collect_list(bat
ch#5L, 0, 0), merge_collect_list(value#4L, 0, 0)])
        +- StateStoreRestore [window#21-T10000ms], state info [ checkpoint = file:/tmp/c
heckpoint-watermark_demo/state, runId = f1b3f7a6-95a9-4a15-af06-13325784b5b4, opId = 0
, ver = 1, numPartitions = 1], 2
            +- ObjectHashAggregate(keys=[window#21-T10000ms], functions=[merge_collect_li
st(batch#5L, 0, 0), merge_collect_list(value#4L, 0, 0)])
                +- Exchange hashpartitioning(window#21-T10000ms, 1)
                    +- ObjectHashAggregate(keys=[window#21-T10000ms], functions=[partial_co
llect_list(batch#5L, 0, 0), partial_collect_list(value#4L, 0, 0)])
                        +- *(1) Project [named_struct(start, precisetimestampconversion((((
CASE WHEN (cast(CEIL((cast((precisestampconversion(time#3-T10000ms, TimestampType,
LongType) - 0) as double) / 5000000.0)) as double) = (cast((precisestampconversio
n(time#3-T10000ms, TimestampType, LongType) - 0) as double) / 5000000.0)) THEN (CEIL((
cast((precisestampconversion(time#3-T10000ms, TimestampType, LongType) - 0) as dou
ble) / 5000000.0)) + 1) ELSE CEIL((cast((precisestampconversion(time#3-T10000ms, T
imestampType, LongType) - 0) as double) / 5000000.0)) END + 0) - 1) * 5000000) + 0), L
ongType, TimestampType), end, precisestampconversion(((CASE WHEN (cast(CEIL((cas
t((precisestampconversion(time#3-T10000ms, TimestampType, LongType) - 0) as double
) / 5000000.0)) as double) = (cast((precisestampconversion(time#3-T10000ms, Timest
ampType, LongType) - 0) as double) / 5000000.0)) THEN (CEIL((cast((precisestampcon
version(time#3-T10000ms, TimestampType, LongType) - 0) as double) / 5000000.0)) + 1) E
lse CEIL((cast((precisestampconversion(time#3-T10000ms, TimestampType, LongType) -
0) as double) / 5000000.0)) END + 0) - 1) * 5000000) + 5000000), LongType, TimestampT
ype)) AS window#21-T10000ms, value#4L, batch#5L]
                        +- *(1) Filter isnotnull(time#3-T10000ms)
                            +- EventTimeWatermark time#3: timestamp, interval 10 seconds
                                +- LocalTableScan <empty>, [time#3, value#4L, batch#5L]
*/
import org.apache.spark.sql.execution.streaming.StreamingQueryWrapper
import org.apache.spark.sql.execution.streaming.StreamExecution
val engine: StreamExecution = streamingQuery
.asInstanceOf[StreamingQueryWrapper]
.streamingQuery

```

```

import org.apache.spark.sql.execution.streaming.IncrementalExecution
val lastMicroBatch: IncrementalExecution = engine.lastExecution

// Access executedPlan that is the optimized physical query plan ready for execution
// All streaming optimizations have been applied at this point
// We just need the EventTimeWatermarkExec physical operator
val plan = lastMicroBatch.executedPlan

// Let's find the EventTimeWatermarkExec physical operator in the plan
// There should be one only
import org.apache.spark.sql.execution.streaming.EventTimeWatermarkExec
val watermarkOp = plan.collect { case op: EventTimeWatermarkExec => op }.head

// Let's check out the event-time watermark stats
// They correspond to the concrete EventTimeWatermarkExec operator for a micro-batch
import org.apache.spark.sql.execution.streaming.EventTimeStats
val stats: EventTimeStats = watermarkOp.eventTimeStats.value
scala> println(stats)
EventTimeStats(-9223372036854775808, 9223372036854775807, 0.0, 0)

val batch = Seq(
  Event(1, 1, batch = 2),
  Event(15, 2, batch = 2),
  Event(35, 3, batch = 2))
events.addData(batch)
streamingQuery.processAllAvailable()

val currentWatermark = streamingQuery.lastProgress.eventTime.get("watermark")
val currentWatermarkMs = toMillis(currentWatermark)

val maxTime = batch.maxBy(_.time.toInstant.toEpochMilli).time.toInstant.toEpochMilli.millis.toSeconds
val expectedMaxTime = 35
assert(maxTime == expectedMaxTime, s"Maximum time across events per batch is $maxTime, but should be $expectedMaxTime")

val expectedWatermarkMs = 25.seconds.toMillis
assert(currentWatermarkMs == expectedWatermarkMs, s"Current event-time watermark is ${currentWatermarkMs}, but should be $expectedWatermarkMs (maximum event time ${maxTime.seconds.toMillis} minus delayThreshold ${delayThreshold.toMillis})")

// FIXME Expired State
// FIXME Late Events
// FIXME Saved State Rows
spark.table(queryName).orderBy("sliding_window").show(truncate = false)
/**+
+-----+-----+
|sliding_window          |batches|values|
+-----+-----+
|[1970-01-01 01:00:00, 1970-01-01 01:00:05]| [1] | [1] |
|[1970-01-01 01:00:15, 1970-01-01 01:00:20]| [1, 2] | [2, 2] |
+-----+-----+
*/

```

```

// Check out the stats
val plan = engine.lastExecution.executedPlan
import org.apache.spark.sql.execution.streaming.EventTimeWatermarkExec
val watermarkOp = plan.collect { case op: EventTimeWatermarkExec => op }.head
import org.apache.spark.sql.execution.streaming.EventTimeStats
val stats: EventTimeStats = watermarkOp.eventTimeStats.value
scala> println(stats)
EventTimeStats(-9223372036854775808, 9223372036854775807, 0.0, 0)

val batch = Seq(
  Event(15,1, batch = 3),
  Event(15,2, batch = 3),
  Event(20,3, batch = 3),
  Event(26,4, batch = 3))
events.addData(batch)
streamingQuery.processAllAvailable()

val currentWatermark = streamingQuery.lastProgress.eventTime.get("watermark")
val currentWatermarkMs = toMillis(currentWatermark)

val maxTime = batch.maxBy(_.time.toInstant.toEpochMilli).time.toInstant.toEpochMilli.millis.toSeconds
val expectedMaxTime = 26
assert(maxTime == expectedMaxTime, s"Maximum time across events per batch is $maxTime,
but should be $expectedMaxTime")

// Current event-time watermark should be the same as previously
// val expectedWatermarkMs = 25.seconds.toMillis
// The current max time is merely 26 so subtracting delayThreshold gives merely 16
assert(currentWatermarkMs == expectedWatermarkMs, s"Current event-time watermark is ${currentWatermarkMs}, but should be $expectedWatermarkMs (maximum event time ${maxTime.seconds.toMillis} minus delayThreshold ${delayThreshold.toMillis})")

// FIXME Expired State
// FIXME Late Events
// FIXME Saved State Rows
spark.table(queryName).orderBy("sliding_window").show(truncate = false)
/*
+-----+-----+
|sliding_window          |batches|values|
+-----+-----+
|[1970-01-01 01:00:00, 1970-01-01 01:00:05]| [1] | [1] |
|[1970-01-01 01:00:15, 1970-01-01 01:00:20]| [1, 2] | [2, 2] |
+-----+-----+
*/
// Check out the stats
val plan = engine.lastExecution.executedPlan
import org.apache.spark.sql.execution.streaming.EventTimeWatermarkExec
val watermarkOp = plan.collect { case op: EventTimeWatermarkExec => op }.head
import org.apache.spark.sql.execution.streaming.EventTimeStats
val stats: EventTimeStats = watermarkOp.eventTimeStats.value

```

```

scala> println(stats)
EventTimeStats(26000,15000,19000.0,4)

val batch = Seq(
  Event(36, 1, batch = 4))
events.addData(batch)
streamingQuery.processAllAvailable()

val currentWatermark = streamingQuery.lastProgress.eventTime.get("watermark")
val currentWatermarkMs = toMillis(currentWatermark)

val maxTime = batch.maxBy(_.time.toInstant.toEpochMilli).time.toInstant.toEpochMilli.millis.toSeconds
val expectedMaxTime = 36
assert(maxTime == expectedMaxTime, s"Maximum time across events per batch is $maxTime,
but should be $expectedMaxTime")

val expectedWatermarkMs = 26.seconds.toMillis
assert(currentWatermarkMs == expectedWatermarkMs, s"Current event-time watermark is ${currentWatermarkMs}, but should be $expectedWatermarkMs (maximum event time ${maxTime.seconds.toMillis} minus delayThreshold ${delayThreshold.toMillis})")

// FIXME Expired State
// FIXME Late Events
// FIXME Saved State Rows
spark.table(queryName).orderBy("sliding_window").show(truncate = false)
/*
+-----+-----+
|sliding_window          |batches|values|
+-----+-----+
|[1970-01-01 01:00:00, 1970-01-01 01:00:05]| [1] | [1] |
|[1970-01-01 01:00:15, 1970-01-01 01:00:20]| [1, 2] | [2, 2] |
+-----+-----+
*/
// Check out the stats
val plan = engine.lastExecution.executedPlan
import org.apache.spark.sql.execution.streaming.EventTimeWatermarkExec
val watermarkOp = plan.collect { case op: EventTimeWatermarkExec => op }.head
import org.apache.spark.sql.execution.streaming.EventTimeStats
val stats: EventTimeStats = watermarkOp.eventTimeStats.value
scala> println(stats)
EventTimeStats(-9223372036854775808, 9223372036854775807, 0.0, 0)

val batch = Seq(
  Event(50, 1, batch = 5)
)
events.addData(batch)
streamingQuery.processAllAvailable()

val currentWatermark = streamingQuery.lastProgress.eventTime.get("watermark")
val currentWatermarkMs = toMillis(currentWatermark)

```

```

val maxTime = batch.maxBy(_.time.toInstant.toEpochMilli).time.toInstant.toEpochMilli.millis.toSeconds
val expectedMaxTime = 50
assert(maxTime == expectedMaxTime, s"Maximum time across events per batch is $maxTime, but should be $expectedMaxTime")

val expectedWatermarkMs = 40.seconds.toMillis
assert(currentWatermarkMs == expectedWatermarkMs, s"Current event-time watermark is ${currentWatermarkMs}, but should be $expectedWatermarkMs (maximum event time ${maxTime.seconds.toMillis} minus delayThreshold ${delayThreshold.toMillis})")

// FIXME Expired State
// FIXME Late Events
// FIXME Saved State Rows
spark.table(queryName).orderBy("sliding_window").show(truncate = false)
/*
+-----+-----+
|sliding_window|batches|values|
+-----+-----+
|[1970-01-01 01:00:00, 1970-01-01 01:00:05]|[1] |[1]   |
|[1970-01-01 01:00:15, 1970-01-01 01:00:20]|[1, 2] |[2, 2]|
|[1970-01-01 01:00:25, 1970-01-01 01:00:30]|[3] |[4]   |
|[1970-01-01 01:00:35, 1970-01-01 01:00:40]|[2, 4] |[3, 1]|
+-----+-----+
*/
// Check out the stats
val plan = engine.lastExecution.executedPlan
import org.apache.spark.sql.execution.streaming.EventTimeWatermarkExec
val watermarkOp = plan.collect { case op: EventTimeWatermarkExec => op }.head
import org.apache.spark.sql.execution.streaming.EventTimeStats
val stats: EventTimeStats = watermarkOp.eventTimeStats.value
scala> println(stats)
EventTimeStats(-9223372036854775808, 9223372036854775807, 0.0, 0)

// Eventually...
streamingQuery.stop()

```

Demo: Streaming Query for Running Counts (Socket Source and Complete Output Mode)

The following code shows a [streaming aggregation](#) (with `Dataset.groupBy` operator) in [complete](#) output mode that reads text lines from a socket (using socket data source) and outputs running counts of the words.

Note

The example is "borrowed" from [the official documentation of Spark](#). Changes and errors are only mine.

Important

Run `nc -lk 9999` first before running the demo.

```
// START: Only for easier debugging
// Reduce the number of partitions
// The state is then only for one partition
// which should make monitoring easier
val numShufflePartitions = 1
import org.apache.spark.sql.internal.SQLConf.SHUFFLE_PARTITIONS
spark.sessionState.conf.setConf(SHUFFLE_PARTITIONS, numShufflePartitions)

assert(spark.sessionState.conf.numShufflePartitions == numShufflePartitions)
// END: Only for easier debugging

val lines = spark
.readStream
.format("socket")
.option("host", "localhost")
.option("port", 9999)
.load

scala> lines.printSchema
root
 |-- value: string (nullable = true)

import org.apache.spark.sql.functions.explode
val words = lines
.select(explode(split($"value", """\W+""")) as "word")

val counts = words.groupBy("word").count

scala> counts.printSchema
root
 |-- word: string (nullable = true)
 |-- count: long (nullable = false)

// nc -lk 9999 is supposed to be up at this point
```

```

val queryName = "running_counts"
val checkpointLocation = s"/tmp/checkpoint-$queryName"

// Delete the checkpoint location from previous executions
import java.nio.file.{Files, FileSystems}
import java.util.Comparator
import scala.collection.JavaConverters._
val path = FileSystems.getDefault.getPath(checkpointLocation)
if (Files.exists(path)) {
  Files.walk(path)
    .sorted(Comparator.reverseOrder())
    .iterator
    .asScala
    .foreach(p => p.toFile.delete)
}

import org.apache.spark.sql.streaming.OutputMode.Complete
val runningCounts = counts
  .writeStream
  .format("console")
  .option("checkpointLocation", checkpointLocation)
  .outputMode(Complete)
  .start

scala> runningCounts.explain
== Physical Plan ==
WriteToDataSourceV2 org.apache.spark.sql.execution.streaming.sources.MicroBatchWriter@
205f195c
+- *(5) HashAggregate(keys=[word#72], functions=[count(1)])
  +- StateStoreSave [word#72], state info [ checkpoint = file:/tmp/checkpoint-running
  _counts/state, runId = f3b2e642-1790-4a17-ab61-3d894110b063, opId = 0, ver = 0, numPar
  titions = 1], Complete, 0, 2
    +- *(4) HashAggregate(keys=[word#72], functions=[merge_count(1)])
      +- StateStoreRestore [word#72], state info [ checkpoint = file:/tmp/checkpoin
t-running_counts/state, runId = f3b2e642-1790-4a17-ab61-3d894110b063, opId = 0, ver = 0
, numPartitions = 1], 2
        +- *(3) HashAggregate(keys=[word#72], functions=[merge_count(1)])
          +- Exchange hashpartitioning(word#72, 1)
            +- *(2) HashAggregate(keys=[word#72], functions=[partial_count(1)])
              +- Generate explode(split(value#83, \W+)), false, [word#72]
                +- *(1) Project [value#83]
                  +- *(1) ScanV2 socket[value#83] (Options: [host=localhost,p
ort=9999])

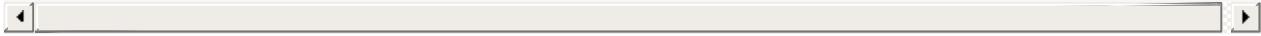
// Type lines (words) in the terminal with nc
// Observe the counts in spark-shell

// Use web UI to monitor the state of state (no pun intended)
// StateStoreSave and StateStoreRestore operators all have state metrics
// Go to http://localhost:4040/SQL/ and click one of the Completed Queries with Job IDs

// You may also want to check out checkpointed state

```

```
// in /tmp/checkpoint-running_counts/state/0/0  
  
// Eventually...  
runningCounts.stop()
```



Demo: Streaming Join of Streaming Queries and StreamingSymmetricHashJoinExec Physical Operator

The following code shows a [streaming join](#) of two streaming queries (and [StreamingSymmetricHashJoinExec](#) physical operator).

```
// START: Only for easier debugging
// Reduce the number of partitions
// The state is then only for one partition
// which should make monitoring easier
val numShufflePartitions = 1
import org.apache.spark.sql.internal.SQLConf.SHUFFLE_PARTITIONS
spark.sessionState.conf.setConf(SHUFFLE_PARTITIONS, numShufflePartitions)

assert(spark.sessionState.conf.numShufflePartitions == numShufflePartitions)
// END: Only for easier debugging

import java.sql.Timestamp
case class Event(time: Timestamp, value: Long)
import scala.concurrent.duration.-
object Event {
  def apply(n: Long, value: Long): Event = {
    Event(new Timestamp(n.seconds.toMillis), value)
  }
}

import org.apache.spark.sql.execution.streaming.MemoryStream
implicit val sqlCtx = spark.sqlContext

val leftEvents = MemoryStream[Event]
leftEvents.addData(
  Event(1,1),
  Event(1,2),
  Event(3,3),
  Event(4,3),
  Event(5,2),
  Event(6,2),
  Event(10,9))
val left = leftEvents.toDS

val rightEvents = MemoryStream[Event]
rightEvents.addData(
  Event(1,1),
  Event(1,2),
  Event(3,3),
  Event(4,3),
```

Streaming Join of Streaming Queries and StreamingSymmetricHashJoinExec Physical Operator

```
Event(5,2),
Event(6,2),
Event(10,9))
val right = rightEvents.tods

// Streaming inner join
val equiStreamingJoin = left.join(right, Seq("value", "time"), "inner")
== Physical Plan ==
*(3) Project [value#3L, time#2]
+- StreamingSymmetricHashJoin [value#3L, time#2], [value#8L, time#7], Inner, condition
= [ leftOnly = null, rightOnly = null, both = null, full = null ], state info [ check
point = <unknown>, runId = c079027b-b68d-4289-a96f-b3c860e76e28, opId = 0, ver = 0, nu
mPartitions = 1], 0, state cleanup [ left = null, right = null ]
  :- Exchange hashpartitioning(value#3L, time#2, 1)
    : +- *(1) Filter isnotnull(time#2)
      :     +- StreamingRelation MemoryStream[time#2,value#3L], [time#2, value#3L]
  +- Exchange hashpartitioning(value#8L, time#7, 1)
    +- *(2) Filter isnotnull(time#7)
      +- StreamingRelation MemoryStream[time#7,value#8L], [time#7, value#8L]

val queryName = "stream_stream_inner_join"
val checkpointLocation = s"/tmp/checkpoint-$queryName"

// Delete the checkpoint location from previous executions
import java.nio.file.{Files, FileSystems}
import java.util.Comparator
import scala.collection.JavaConverters.-
val path = FileSystems.getDefault.getPath(checkpointLocation)
if (Files.exists(path)) {
  Files.walk(path)
    .sorted(Comparator.reverseOrder())
    .iterator
    .asScala
    .foreach(p => p.toFile.delete)
}

import org.apache.spark.sql.streaming.OutputMode.Append
import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration.-
val batchInterval = 30.seconds
val streamingQuery = equiStreamingJoin
  .writeStream
  .format("memory")
  .queryName(queryName)
  .option("checkpointLocation", checkpointLocation)
  .outputMode(Append)
  .trigger(Trigger.ProcessingTime(batchInterval))
  .start

streamingQuery.processAllAvailable()

val output = spark.table(queryName).orderBy("value", "time")
output.show(truncate = false)
```

Streaming Join of Streaming Queries and StreamingSymmetricHashJoinExec Physical Operator

```
/**+
+-----+-----+
|value|time      |
+-----+-----+
|1    |1970-01-01 01:00:01|
|2    |1970-01-01 01:00:01|
|2    |1970-01-01 01:00:05|
|2    |1970-01-01 01:00:06|
|3    |1970-01-01 01:00:03|
|3    |1970-01-01 01:00:04|
|9    |1970-01-01 01:00:10|
+-----+-----+
*/
assert(output.collect.size == 7)

scala> streamingQuery.explain
== Physical Plan ==
*(3) Project [value#68L, time#67]
+- StreamingSymmetricHashJoin [value#68L, time#67], [value#66L, time#65], Inner, condition = [ leftOnly = null, rightOnly = null, both = null, full = null ], state info [ checkpoint = file:/tmp/checkpoint-stream_stream_inner_join/state, runId = 7adaf1c4-c0a5-471d-aa8e-11c08c039de4, opId = 0, ver = 0, numPartitions = 1], 0, state cleanup [ left = null, right = null ]
   :- *(1) Filter isnotnull(time#67)
     :  +- *(1) Project [time#67, value#68L]
       :    +- *(1) ScanV2 MemoryStreamDataSource$time#67, value#68L]
   +- *(2) Filter isnotnull(time#65)
     +- *(2) Project [time#65, value#66L]
       +- *(2) ScanV2 MemoryStreamDataSource$time#65, value#66L]

// Eventually...
streamingQuery.stop()
```

Demo: Streaming Aggregation with Kafka Data Source

The following example code shows a streaming aggregation (with `Dataset.groupBy` operator) that reads records from Kafka (with [Kafka Data Source](#)).

Important

Start up Kafka cluster and spark-shell with `spark-sql-kafka-0-10` package before running the demo.

Tip

You may want to consider copying the following code to `append.txt` and using `:load append.txt` command in spark-shell to load it (rather than copying and pasting it).

```
// START: Only for easier debugging
// The state is then only for one partition
// which should make monitoring easier
val numShufflePartitions = 1
import org.apache.spark.sql.internal.SQLConf.SHUFFLE_PARTITIONS
spark.sessionState.conf.setConf(SHUFFLE_PARTITIONS, numShufflePartitions)

assert(spark.sessionState.conf.numShufflePartitions == numShufflePartitions)
// END: Only for easier debugging

val records = spark
  .readStream
  .format("kafka")
  .option("subscribePattern", """topic-\d{2}""") // topics with two digits at the end
  .option("kafka.bootstrap.servers", ":9092")
  .load
scala> records.printSchema
root
|-- key: binary (nullable = true)
|-- value: binary (nullable = true)
|-- topic: string (nullable = true)
|-- partition: integer (nullable = true)
|-- offset: long (nullable = true)
|-- timestamp: timestamp (nullable = true)
|-- timestampType: integer (nullable = true)

// Since the streaming query uses Append output mode
// it has to define a streaming event-time watermark (using Dataset.withWatermark operator)
// UnsupportedOperationChecker makes sure that the requirement holds
val ids = records
  .withColumn("tokens", split($"value", ","))
  .withColumn("seconds", 'tokens(0) cast "long")
  .withColumn("event_time", to_timestamp(from_unixtime('seconds))) // <-- Event time h
```

```

as to be a timestamp
    .withColumn("id", 'tokens(1))
    .withColumn("batch", 'tokens(2) cast "int")
    .withWatermark(eventTime = "event_time", delayThreshold = "10 seconds") // <-- define watermark (before groupBy!)
    .groupBy($"event_time") // <-- use event_time for grouping
    .agg(collect_list("batch") as "batches", collect_list("id") as "ids")
    .withColumn("event_time", to_timestamp($"event_time")) // <-- convert to human-readable date
scala> ids.printSchema
root
|-- event_time: timestamp (nullable = true)
|-- batches: array (nullable = true)
|   |-- element: integer (containsNull = true)
|-- ids: array (nullable = true)
|   |-- element: string (containsNull = true)

assert(ids.isStreaming, "ids is a streaming query")

// ids knows nothing about the output mode or the current streaming watermark yet
// - Output mode is defined on writing side
// - streaming watermark is read from rows at runtime
// That's why StatefulOperatorStateInfo is generic (and uses the default Append for output mode)
// and no batch-specific values are printed out
// They will be available right after the first streaming batch
// Use explain on a streaming query to know the trigger-specific values
scala> ids.explain
== Physical Plan ==
ObjectHashAggregate(keys=[event_time#118-T10000ms], functions=[collect_list(batch#141,
0, 0), collect_list(id#129, 0, 0)])
+- StateStoreSave [event_time#118-T10000ms], state info [ checkpoint = <unknown>, runId = a870e6e2-b925-4104-9886-b211c0be1b73, opId = 0, ver = 0, numPartitions = 1], Append
, 0, 2
    +- ObjectHashAggregate(keys=[event_time#118-T10000ms], functions=[merge_collect_list(batch#141, 0, 0), merge_collect_list(id#129, 0, 0)])
        +- StateStoreRestore [event_time#118-T10000ms], state info [ checkpoint = <unknown>, runId = a870e6e2-b925-4104-9886-b211c0be1b73, opId = 0, ver = 0, numPartitions = 1 ], 2
            +- ObjectHashAggregate(keys=[event_time#118-T10000ms], functions=[merge_collect_list(batch#141, 0, 0), merge_collect_list(id#129, 0, 0)])
                +- Exchange hashpartitioning(event_time#118-T10000ms, 1)
                    +- ObjectHashAggregate(keys=[event_time#118-T10000ms], functions=[partial_collect_list(batch#141, 0, 0), partial_collect_list(id#129, 0, 0)])
                        +- EventTimeWatermark event_time#118: timestamp, interval 10 seconds
                            +- *(1) Project [cast(from_unixtime(cast(split(cast(value#8 as string), ,)[0] as bigint), yyyy-MM-dd HH:mm:ss, Some(Europe/Warsaw)) as timestamp) AS event_time#118, split(cast(value#8 as string), ,)[1] AS id#129, cast(split(cast(value#8 as string), ,)[2] as int) AS batch#141]
                                +- StreamingRelation kafka, [key#7, value#8, topic#9, partition#10, offset#11L, timestamp#12, timestampType#13]

val queryName = "ids-kafka"

```

```

val checkpointLocation = s"/tmp/checkpoint-$queryName"

// Delete the checkpoint location from previous executions
import java.nio.file.{Files, FileSystems}
import java.util.Comparator
import scala.collection.JavaConverters._
val path = FileSystems.getDefault.getPath(checkpointLocation)
if (Files.exists(path)) {
  Files.walk(path)
    .sorted(Comparator.reverseOrder())
    .iterator
    .asScala
    .foreach(p => p.toFile.delete)
}

// The following make for an easier demo
// Kafka cluster is supposed to be up at this point
// Make sure that a Kafka topic is available, e.g. topic-00
// Use ./bin/kafka-console-producer.sh --broker-list :9092 --topic topic-00
// And send a record, e.g. 1,1,1

// Define the output mode
// and start the query
import scala.concurrent.duration._
import org.apache.spark.sql.streaming.OutputMode.Append
import org.apache.spark.sql.streaming.Trigger
val streamingQuery = ids
  .writeStream
  .format("console")
  .option("truncate", false)
  .option("checkpointLocation", checkpointLocation)
  .queryName(queryName)
  .outputMode(Append)
  .start

val lastProgress = streamingQuery.lastProgress
scala> :type lastProgress
org.apache.spark.sql.streaming.StreamingQueryProgress

assert(lastProgress.stateOperators.length == 1, "There should be one stateful operator"
)

scala> println(lastProgress.stateOperators.head.prettyJson)
{
  "numRowsTotal" : 1,
  "numRowsUpdated" : 0,
  "memoryUsedBytes" : 742,
  "customMetrics" : {
    "loadedMapCacheHitCount" : 1,
    "loadedMapCacheMissCount" : 1,
    "stateOnCurrentVersionSizeBytes" : 374
  }
}

```

```
assert(lastProgress.sources.length == 1, "There should be one streaming source only")
scala> println(lastProgress.sources.head.prettyJson)
{
  "description" : "KafkaV2[SubscribePattern[topic-\\d{2}]]",
  "startOffset" : {
    "topic-00" : {
      "0" : 1
    }
  },
  "endOffset" : {
    "topic-00" : {
      "0" : 1
    }
  },
  "numInputRows" : 0,
  "inputRowsPerSecond" : 0.0,
  "processedRowsPerSecond" : 0.0
}

// Eventually...
streamingQuery.stop()
```

Demo: groupByKey Streaming Aggregation in Update Mode

The example shows `Dataset.groupByKey` streaming operator to count rows in `Update` output mode.

In other words, it is an example of using `Dataset.groupByKey` with `count` aggregation function to count customer orders (`t`) per zip code (`k`).

Complete Spark Structured Streaming Application

```

package pl.japila.spark.examples

import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.streaming.{OutputMode, Trigger}

object GroupByKeyStreamingApp extends App {

    val inputTopic = "GroupByKeyApp-input"
    val appName = this.getClass.getSimpleName.replace("$", "")

    val spark = SparkSession.builder
        .master("local[*]")
        .appName(appName)
        .getOrCreate
    import spark.implicits._

    case class Order(id: Long, zipCode: String)

    // Input (source node)
    val orders = spark
        .readStream
        .format("kafka")
        .option("startingOffsets", "latest")
        .option("subscribe", inputTopic)
        .option("kafka.bootstrap.servers", ":9092")
        .load
        .select($"offset" as "id", $"value" as "zipCode") // FIXME Use csv, json, avro
        .as[Order]

    // Processing logic
    // groupByKey + count
    val byZipCode = (o: Order) => o.zipCode
    val ordersByZipCode = orders.groupByKey(byZipCode)

    import org.apache.spark.sql.functions.count
    val typedCountCol = (count("zipCode") as "count").as[String]
    val counts = ordersByZipCode
        .agg(typedCountCol)
        .select($"value" as "zip_code", $"count")

    // Output (sink node)
    import scala.concurrent.duration._
    counts
        .writeStream
        .format("console")
        .outputMode(OutputMode.Update) // FIXME Use Complete
        .queryName(appName)
        .trigger(Trigger.ProcessingTime(5.seconds))
        .start
        .awaitTermination()
}

```

Credits

- The example with customer orders and postal codes is borrowed from Apache Beam's [Using GroupByKey Programming Guide](#).

Demo: StateStoreSaveExec with Complete Output Mode

The following example code shows the behaviour of [StateStoreSaveExec](#) in Complete output mode.

```
// START: Only for easier debugging
// The state is then only for one partition
// which should make monitoring it easier
import org.apache.spark.sql.internal.SQLConf.SHUFFLE_PARTITIONS
spark.sessionState.conf.setConf(SHUFFLE_PARTITIONS, 1)
scala> spark.sessionState.conf.numShufflePartitions
res1: Int = 1
// END: Only for easier debugging

// Read datasets from a Kafka topic
// ./bin/spark-shell --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0-SNAPS
HOT
// Streaming aggregation using groupBy operator is required to have StateStoreSaveExec
operator
val valuesPerGroup = spark.
  readStream.
  format("kafka").
  option("subscribe", "topic1").
  option("kafka.bootstrap.servers", "localhost:9092").
  load.
  withColumn("tokens", split('value, ",")).
  withColumn("group", 'tokens(0)).
  withColumn("value", 'tokens(1) cast "int").
  select("group", "value").
  groupBy($"group").
  agg(collect_list("value") as "values").
  orderBy($"group".asc)

// valuesPerGroup is a streaming Dataset with just one source
// so it knows nothing about output mode or watermark yet
// That's why StatefulOperatorStateInfo is generic
// and no batch-specific values are printed out
// That will be available after the first streaming batch
// Use sq.explain to know the runtime-specific values
scala> valuesPerGroup.explain
== Physical Plan ==
*Sort [group#25 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(group#25 ASC NULLS FIRST, 1)
   +- ObjectHashAggregate(keys=[group#25], functions=[collect_list(value#36, 0, 0)])
      +- Exchange hashpartitioning(group#25, 1)
         +- StateStoreSave [group#25], StatefulOperatorStateInfo(<unknown>, 899f0fd1-b2
02-45cd-9ebd-09101ca90fa8, 0, 0), Append, 0
```

```

        +- ObjectHashAggregate(keys=[group#25], functions=[merge_collect_list(value#36, 0, 0)])
        +- Exchange hashpartitioning(group#25, 1)
        +- StateStoreRestore [group#25], StatefulOperatorStateInfo(<unknown>, 899f0fd1-b202-45cd-9ebd-09101ca90fa8,0,0)
        +- ObjectHashAggregate(keys=[group#25], functions=[merge_collect_list(value#36, 0, 0)])
        +- Exchange hashpartitioning(group#25, 1)
        +- ObjectHashAggregate(keys=[group#25], functions=[partial_collect_list(value#36, 0, 0)])
        +- *Project [split(cast(value#1 as string), ,)[0] AS group#25, cast(split(cast(value#1 as string), ,)[1] as int) AS value#36]
        +- StreamingRelation kafka, [key#0, value#1, topic#2, partition#3, offset#4L, timestamp#5, timestampType#6]

// Start the query and hence StateStoreSaveExec
// Use Complete output mode
import scala.concurrent.duration._
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
val sq = valuesPerGroup.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Complete).
  start

-----
Batch: 0
-----
+----+----+
|group|values|
+----+----+
+----+----+

// there's only 1 stateful operator and hence 0 for the index in stateOperators
scala> println(sq.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 0,
  "numRowsUpdated" : 0,
  "memoryUsedBytes" : 60
}

// publish 1 new key-value pair in a single streaming batch
// 0,1

-----
Batch: 1
-----
+----+----+
|group|values|
+----+----+
|0    |[1]   |

```

```
+-----+
// it's Complete output mode so numRowsTotal is the number of keys in the state store
// no keys were available earlier (it's just started!) and so numRowsUpdated is 0
scala> println(sq.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 1,
  "numRowsUpdated" : 0,
  "memoryUsedBytes" : 324
}

// publish new key and old key in a single streaming batch
// new keys
// 1,1
// updates to already-stored keys
// 0,2

-----
Batch: 2
-----
+-----+
|group|values|
+-----+
|0    |[2, 1]|
|1    |[1]     |
+-----+

// it's Complete output mode so numRowsTotal is the number of keys in the state store
// no keys were available earlier and so numRowsUpdated is...0?!
// Think it's a BUG as it should've been 1 (for the row 0,2)
// 8/30 Sent out a question to the Spark user mailing list
scala> println(sq.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 2,
  "numRowsUpdated" : 0,
  "memoryUsedBytes" : 572
}

// In the end...
sq.stop
```

Demo: StateStoreSaveExec with Update Output Mode

Caution	FIXME Example of Update with StateStoreSaveExec (and optional watermark)
---------	--

Demo: Developing Custom Streaming Sink (and Monitoring SQL Queries in web UI)

The demo shows the steps to develop a custom [streaming sink](#) and use it to monitor whether and what SQL queries are executed at runtime (using web UI's SQL tab).

Note	<p>The main motivation was to answer the question Why does a single structured query run multiple SQL queries per batch? that happened to have turned out fairly surprising.</p> <p>You're very welcome to upvote the question and answers at your earliest convenience. Thanks!</p>
------	--

The steps are as follows:

1. [Creating Custom Sink — DemoSink](#)
2. [Creating StreamSinkProvider — DemoSinkProvider](#)
3. [Optional Sink Registration using META-INF/services](#)
4. [build.sbt Definition](#)
5. [Packaging DemoSink](#)
6. [Using DemoSink in Streaming Query](#)
7. [Monitoring SQL Queries using web UI's SQL Tab](#)

Findings (aka *surprises*):

1. Custom sinks require that you define a checkpoint location using `checkpointLocation` option (or `spark.sql.streaming.checkpointLocation` Spark property). Remove the checkpoint directory (or use a different one every start of a streaming query) to have consistent results.

Creating Custom Sink — DemoSink

A streaming sink follows the [Sink contract](#) and a sample implementation could look as follows.

```

package pl.japila.spark.sql.streaming

case class DemoSink(
    sqlContext: SQLContext,
    parameters: Map[String, String],
    partitionColumns: Seq[String],
    outputMode: OutputMode) extends Sink {

    override def addBatch(batchId: Long, data: DataFrame): Unit = {
        println(s"addBatch($batchId)")
        data.explain()
        // Why so many lines just to show the input DataFrame?
        data.sparkSession.createDataFrame(
            data.sparkSession.sparkContext.parallelize(data.collect()), data.schema)
            .show(10)
    }
}

```

Save the file under `src/main/scala` in your project.

Creating StreamSinkProvider — DemoSinkProvider

```

package pl.japila.spark.sql.streaming

class DemoSinkProvider extends StreamSinkProvider
    with DataSourceRegister {

    override def createSink(
        sqlContext: SQLContext,
        parameters: Map[String, String],
        partitionColumns: Seq[String],
        outputMode: OutputMode): Sink = {
        DemoSink(sqlContext, parameters, partitionColumns, outputMode)
    }

    override def shortName(): String = "demo"
}

```

Save the file under `src/main/scala` in your project.

Optional Sink Registration using META-INF/services

The step is optional, but greatly improve the experience when using the custom sink so you can use it by its name (rather than a fully-qualified class name or using a special class name for the sink provider).

Create `org.apache.spark.sql.sources.DataSourceRegister` in `META-INF/services` directory with the following content.

```
pl.japila.spark.sql.streaming.DemoSinkProvider
```

Save the file under `src/main/resources` in your project.

build.sbt Definition

If you use my beloved build tool `sbt` to manage the project, use the following `build.sbt`.

```
organization := "pl.japila.spark"
name := "spark-structured-streaming-demo-sink"
version := "0.1"

scalaVersion := "2.11.11"

libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.2.0"
```

Packaging DemoSink

The step depends on what build tool you use to manage the project. Use whatever command you use to create a jar file with the above classes compiled and bundled together.

```
$ sbt package
[info] Loading settings from plugins.sbt ...
[info] Loading project definition from /Users/jacek/dev/sandbox/spark-structured-streaming-demo-sink/project
[info] Loading settings from build.sbt ...
[info] Set current project to spark-structured-streaming-demo-sink (in build file:/Users/jacek/dev/sandbox/spark-structured-streaming-demo-sink/)
[info] Compiling 1 Scala source to /Users/jacek/dev/sandbox/spark-structured-streaming-demo-sink/target/scala-2.11/classes ...
[info] Done compiling.
[info] Packaging /Users/jacek/dev/sandbox/spark-structured-streaming-demo-sink/target/scala-2.11/spark-structured-streaming-demo-sink_2.11-0.1.jar ...
[info] Done packaging.
[success] Total time: 5 s, completed Sep 12, 2017 9:34:19 AM
```

The jar with the sink is `/Users/jacek/dev/sandbox/spark-structured-streaming-demo-sink/target/scala-2.11/spark-structured-streaming-demo-sink_2.11-0.1.jar`.

Using DemoSink in Streaming Query

The following code reads data from the `rate` source and simply outputs the result to our custom `DemoSink`.

```
// Make sure the DemoSink jar is available
$ ls /Users/jacek/dev/sandbox/spark-structured-streaming-demo-sink/target/scala-2.11/spark-structured-streaming-demo-sink_2.11-0.1.jar
/Users/jacek/dev/sandbox/spark-structured-streaming-demo-sink/target/scala-2.11/spark-structured-streaming-demo-sink_2.11-0.1.jar

// "Install" the DemoSink using --jars command-line option
$ ./bin/spark-shell --jars /Users/jacek/dev/sandbox/spark-structured-streaming-custom-sink/target/scala-2.11/spark-structured-streaming-custom-sink_2.11-0.1.jar

scala> spark.version
res0: String = 2.3.0-SNAPSHOT

import org.apache.spark.sql.streaming._
import scala.concurrent.duration._
val sq = spark.
  readStream.
  format("rate").
  load.
  writeStream.
  format("demo").
  option("checkpointLocation", "/tmp/demo-checkpoint").
  trigger(Trigger.ProcessingTime(10.seconds)).
  start

// In the end...
scala> sq.stop
17/09/12 09:59:28 INFO StreamExecution: Query [id = 03cd78e3-94e2-439c-9c12-cfed0c996812, runId = 6938af91-9806-4404-965a-5ae7525d5d3f] was stopped
```

Monitoring SQL Queries using web UI's SQL Tab

Open <http://localhost:4040/SQL/>.

You should find that every trigger (aka *batch*) results in 3 SQL queries. Why?

ID	Description	Submitted	Duration	Job IDs
80	start at <console>:43	+details 2017/09/12 09:55:30	20 ms	103 104 105
79	start at <console>:43	+details 2017/09/12 09:55:30	9 ms	102
78	start at <console>:43	+details 2017/09/12 09:55:30	36 ms	
77	start at <console>:43	+details 2017/09/12 09:55:20	32 ms	99 100 101
76	start at <console>:43	+details 2017/09/12 09:55:20	9 ms	98
75	start at <console>:43	+details 2017/09/12 09:55:20	49 ms	
74	start at <console>:43	+details 2017/09/12 09:55:10	21 ms	95 96 97
73	start at <console>:43	+details 2017/09/12 09:55:10	8 ms	94
72	start at <console>:43	+details 2017/09/12 09:55:10	39 ms	
71	start at <console>:43	+details 2017/09/12 09:55:00	19 ms	91 92 93
70	start at <console>:43	+details 2017/09/12 09:55:00	9 ms	90
69	start at <console>:43	+details 2017/09/12 09:55:00	37 ms	

Figure 1. web UI's SQL Tab and Completed Queries (3 Queries per Batch)

The answer lies in what sources and sink a streaming query uses (and differs per streaming query).

In our case, `DemoSink` collects the rows from the input `DataFrame` and shows it afterwards. That gives 2 SQL queries (as you can see after executing the following batch queries).

```
// batch non-streaming query
val data = (0 to 3).toDF("id")

// That gives one SQL query
data.collect

// That gives one SQL query, too
data.show
```

The remaining query (which is the first among the queries) is executed when you load the data.

That can be observed easily when you change `DemoSink` to not "touch" the input `data` (in `addBatch`) in any way.

```
override def addBatch(batchId: Long, data: DataFrame): Unit = {
    println(s"addBatch($batchId)")
}
```

Re-run the streaming query (using the new `DemoSink`) and use web UI's SQL tab to see the queries. You should have just one query per batch (and no Spark jobs given nothing is really done in the sink's `addBatch`).

The screenshot shows the Spark web UI interface. At the top, there is a navigation bar with tabs: Jobs, Stages, Storage, Environment, Executors, and SQL. The SQL tab is currently selected, indicated by a grey background. To the right of the navigation bar, it says "Spark shell application UI". Below the navigation bar, the title "SQL" is displayed in bold. Underneath "SQL", the heading "Completed Queries" is shown. A table follows, listing two completed queries. The columns in the table are ID, Description, Submitted, Duration, and Job IDs. The first query (ID 1) was submitted at 2017/09/12 10:26:40 and took 0 ms. The second query (ID 0) was submitted at 2017/09/12 10:26:38 and also took 0 ms. Both queries are described as "start at <console>:37" and have a "+details" link next to them.

ID	Description	Submitted	Duration	Job IDs
1	start at <console>:37	+details 2017/09/12 10:26:40	0 ms	
0	start at <console>:37	+details 2017/09/12 10:26:38	0 ms	

Figure 2. web UI's SQL Tab and Completed Queries (1 Query per Batch)

Demo: current_timestamp Function For Processing Time in Streaming Queries

The demo shows what happens when you use `current_timestamp` function in your structured queries.

Note

The main motivation was to answer the question [How to achieve ingestion time?](#) in Spark Structured Streaming.

You're very welcome to upvote the question and answers at your earliest convenience. Thanks!

Quoting the [Apache Flink documentation](#):

Event time is the time that each individual event occurred on its producing device. This time is typically embedded within the records before they enter Flink and that event timestamp can be extracted from the record.

That is exactly how event time is considered in `withWatermark` operator which you use to describe what column to use for event time. The column could be part of the input dataset or...generated.

And that is the moment where my confusion starts.

In order to generate the event time column for `withWatermark` operator you could use `current_timestamp` or `current_date` standard functions.

```
// rate format gives event time
// but let's generate a brand new column with ours
// for demo purposes
val values = spark.
  readStream.
  format("rate").
  load.
  withColumn("current_timestamp", current_timestamp)
scala> values.printSchema
root
|-- timestamp: timestamp (nullable = true)
|-- value: long (nullable = true)
|-- current_timestamp: timestamp (nullable = false)
```

Both are special for Spark Structured Streaming as `streamExecution` replaces their underlying Catalyst expressions, `CurrentTimestamp` and `CurrentDate` respectively, with `CurrentBatchTimestamp` expression and the time of the current batch.

```

import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._
val sq = values.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  start

// note the value of current_timestamp
// that corresponds to the batch time

-----
Batch: 1
-----
+-----+-----+
|timestamp          |value|current_timestamp |
+-----+-----+
|2017-09-18 10:53:31.523|0    |2017-09-18 10:53:40|
|2017-09-18 10:53:32.523|1    |2017-09-18 10:53:40|
|2017-09-18 10:53:33.523|2    |2017-09-18 10:53:40|
|2017-09-18 10:53:34.523|3    |2017-09-18 10:53:40|
|2017-09-18 10:53:35.523|4    |2017-09-18 10:53:40|
|2017-09-18 10:53:36.523|5    |2017-09-18 10:53:40|
|2017-09-18 10:53:37.523|6    |2017-09-18 10:53:40|
|2017-09-18 10:53:38.523|7    |2017-09-18 10:53:40|
+-----+-----+

// Use web UI's SQL tab for the batch (Submitted column)
// or sq.recentProgress
scala> println(sq.recentProgress(1).timestamp)
2017-09-18T08:53:40.000Z

// Note current_batch_timestamp

scala> sq.explain(extended = true)
== Parsed Logical Plan ==
'Project [timestamp#2137, value#2138L, current_batch_timestamp(1505725650005, TimestampType, None) AS current_timestamp#50]
+- LogicalRDD [timestamp#2137, value#2138L], true

== Analyzed Logical Plan ==
timestamp: timestamp, value: bigint, current_timestamp: timestamp
Project [timestamp#2137, value#2138L, current_batch_timestamp(1505725650005, TimestampType, Some(Europe/Berlin)) AS current_timestamp#50]
+- LogicalRDD [timestamp#2137, value#2138L], true

== Optimized Logical Plan ==
Project [timestamp#2137, value#2138L, 1505725650005000 AS current_timestamp#50]
+- LogicalRDD [timestamp#2137, value#2138L], true

== Physical Plan ==

```

```
*Project [timestamp#2137, value#2138L, 1505725650005000 AS current_timestamp#50]
+- Scan ExistingRDD[timestamp#2137,value#2138L]
```

That *seems* to be closer to processing time than ingestion time given the definition from the [Apache Flink documentation](#):

Processing time refers to the system time of the machine that is executing the respective operation.

Ingestion time is the time that events enter Flink.

What do you think?

Demo: Using StreamingQueryManager for Query Termination Management

The demo shows how to use [StreamingQueryManager](#) (and specifically [awaitAnyTermination](#) and [resetTerminated](#)) for query termination management.

`demo-StreamingQueryManager.scala`

```
// Save the code as demo-StreamingQueryManager.scala
// Start it using spark-shell
// $ ./bin/spark-shell -i demo-StreamingQueryManager.scala

// Register a StreamingQueryListener to receive notifications about state changes of streaming queries
import org.apache.spark.sql.streaming.StreamingQueryListener
val myQueryListener = new StreamingQueryListener {
    import org.apache.spark.sql.streaming.StreamingQueryListener._
    def onQueryTerminated(event: QueryTerminatedEvent): Unit = {
        println(s"Query ${event.id} terminated")
    }

    def onQueryStarted(event: QueryStartedEvent): Unit = {}
    def onQueryProgress(event: QueryProgressEvent): Unit = {}
}
spark.streams.addListener(myQueryListener)

import org.apache.spark.sql.streaming._
import scala.concurrent.duration._

// Start streaming queries

// Start the first query
val q4s = spark.readStream.
    format("rate").
    load.
    writeStream.
    format("console").
    trigger(Trigger.ProcessingTime(4.seconds)).
    option("truncate", false).
    start

// Start another query that is slightly slower
val q10s = spark.readStream.
    format("rate").
    load.
    writeStream.
    format("console").
    trigger(Trigger.ProcessingTime(10.seconds)).
    option("truncate", false).
```

```

start

// Both queries run concurrently
// You should see different outputs in the console
// q4s prints out 4 rows every batch and twice as often as q10s
// q10s prints out 10 rows every batch

/*
-----
Batch: 7
-----
+-----+-----+
|timestamp|value|
+-----+-----+
|2017-10-27 13:44:07.462|21 |
|2017-10-27 13:44:08.462|22 |
|2017-10-27 13:44:09.462|23 |
|2017-10-27 13:44:10.462|24 |
+-----+-----+

-----
Batch: 8
-----
+-----+-----+
|timestamp|value|
+-----+-----+
|2017-10-27 13:44:11.462|25 |
|2017-10-27 13:44:12.462|26 |
|2017-10-27 13:44:13.462|27 |
|2017-10-27 13:44:14.462|28 |
+-----+-----+

-----
Batch: 2
-----
+-----+-----+
|timestamp|value|
+-----+-----+
|2017-10-27 13:44:09.847|6 |
|2017-10-27 13:44:10.847|7 |
|2017-10-27 13:44:11.847|8 |
|2017-10-27 13:44:12.847|9 |
|2017-10-27 13:44:13.847|10 |
|2017-10-27 13:44:14.847|11 |
|2017-10-27 13:44:15.847|12 |
|2017-10-27 13:44:16.847|13 |
|2017-10-27 13:44:17.847|14 |
|2017-10-27 13:44:18.847|15 |
+-----+-----+
*/
// Stop q4s on a separate thread
// as we're about to block the current thread awaiting query termination

```

```
import java.util.concurrent.Executors
import java.util.concurrent.TimeUnit.SECONDS
def queryTerminator(query: StreamingQuery) = new Runnable {
  def run = {
    println(s"Stopping streaming query: ${query.id}")
    query.stop
  }
}
import java.util.concurrent.TimeUnit.SECONDS
// Stop the first query after 10 seconds
Executors.newSingleThreadScheduledExecutor.
  scheduleWithFixedDelay(queryTerminator(q4s), 10, 60 * 5, SECONDS)
// Stop the other query after 20 seconds
Executors.newSingleThreadScheduledExecutor.
  scheduleWithFixedDelay(queryTerminator(q10s), 20, 60 * 5, SECONDS)

// Use StreamingQueryManager to wait for any query termination (either q1 or q2)
// the current thread will block indefinitely until either streaming query has finished

spark.streams.awaitAnyTermination

// You are here only after either streaming query has finished
// Executing spark.streams.awaitAnyTermination again would return immediately

// You should have received the QueryTerminatedEvent for the query termination

// reset the last terminated streaming query
spark.streams.resetTerminated

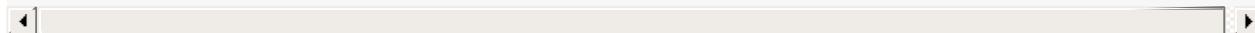
// You know at least one query has terminated

// Wait for the other query to terminate
spark.streams.awaitAnyTermination

assert(spark.streams.active.isEmpty)

println("The demo went all fine. Exiting...")

// leave spark-shell
System.exit(0)
```



Streaming Join

In Spark Structured Streaming, a streaming query is an example of **streaming join** when it uses `Dataset.join` or SQL's `JOIN` high-level query operators.

Under the covers, the high-level operators create a logical query plan with one or more `Join` logical operators.

Tip	Read up on Join Logical Operator in The Internals of Spark SQL book.
-----	--

Streaming joins can be **stateless** or **stateful**. Joins of a streaming query and a batch query (aka *stream-static joins*) are stateless and no state management is necessary. Joins of two streaming queries (aka *stream-stream joins*) are stateful and require streaming state (with a [streaming watermark](#)).

Demos

Use the following demos to learn more:

- [Demo: Streaming Join of Streaming Queries and StreamingSymmetricHashJoinExec Physical Operator](#)

IncrementalExecution — QueryExecution of Streaming Queries

In Spark Structured Streaming it is [IncrementalExecution](#) that plans streaming queries for execution.

While [planning a streaming query for execution](#) (aka *query planning*), `IncrementalExecution` uses the [StreamingJoinStrategy](#) execution planning strategy for planning streaming joins (`Join` logical operators) as [StreamingSymmetricHashJoinExec](#) physical operators.

StateStoreAwareZipPartitionsRDD

`StateStoreAwareZipPartitionsRDD` is a `ZippedPartitionsRDD2` with the `left` and `right` parent RDDs.

`StateStoreAwareZipPartitionsRDD` is created exclusively when `StreamingSymmetricHashJoinExec` physical operator is requested to execute and generate a recipe for a distributed computation (as an `RDD[InternalRow]`) (and requests `StateStoreAwareZipPartitionsHelper` for one).

Creating StateStoreAwareZipPartitionsRDD Instance

`StateStoreAwareZipPartitionsRDD` takes the following to be created:

- `SparkContext`
- `Function (Iterator[A], Iterator[B]) => Iterator[V]`
- **Left RDD** - the RDD of the left side of a join (`RDD[A]`)
- **Right RDD** - the RDD of the right side of a join (`RDD[B]`)
- `StatefulOperatorStateInfo`
- Names of the `state stores`
- `StateStoreCoordinatorRef`

Placement Preferences of Partition (Preferred Locations) — `getPreferredLocations` Method

```
getPreferredLocations(partition: Partition): Seq[String]
```

Note	<code>getPreferredLocations</code> is a part of the RDD Contract to specify placement preferences (aka <i>preferred task locations</i>), i.e. where tasks should be executed to be as close to the data as possible.
------	---

`getPreferredLocations` ...FIXME

SymmetricHashJoinStateManager

`SymmetricHashJoinStateManager` is created exclusively for [OneSideHashJoiner](#) (for [StreamingSymmetricHashJoinExec](#) physical operator to execute).

Creating SymmetricHashJoinStateManager Instance

`SymmetricHashJoinStateManager` takes the following to be created:

- `JoinSide`
- Input value attributes
- Join keys (`Seq[Expression]`)
- [StatefulOperatorStateInfo](#)
- [StateStoreConf](#)
- Hadoop [Configuration](#)

`SymmetricHashJoinStateManager` initializes the [internal properties](#).

Performance Metrics — `metrics` Method

```
metrics: StateStoreMetrics
```

`metrics` returns the combined [StateStoreMetrics](#) of the [KeyToNumValuesStore](#) and the [KeyWithIndexToValueStore](#).

Note

`metrics` is used exclusively when `OneSideHashJoiner` is requested to [commitStateAndGetMetrics](#).

allStateStoreNames Object Method

```
allStateStoreNames(joinSides: JoinSide*): Seq[String]
```

`allStateStoreNames` ...FIXME

Note

`allStateStoreNames` is used when...FIXME

Internal Properties

Name	Description
keyAttributes	Key attributes Used when...FIXME
keySchema	Key schema Used when...FIXME
keyToNumValues	KeyToNumValuesStore Used when...FIXME
keyWithIndexToValue	KeyWithIndexToValueStore Used when...FIXME

StateStoreHandler Internal Contract

`StateStoreHandler` is the internal base of state store handlers that manage a `StateStore` (i.e. `commit`, `abortIfNeeded` and `metrics`).

`StateStoreHandler` takes a single `stateStoreType` to be created:

- `KeyToNumValuesType` for `KeyToNumValuesStore`
- `KeyWithIndexToValueType` for `KeyWithIndexToValueStore`

Note

`StateStoreHandler` is a Scala private abstract class and cannot be created directly. It is created indirectly for the concrete StateStoreHandlers.

Table 1. StateStoreHandler Contract

Method	Description
<code>stateStore</code>	<code>stateStore: StateStore</code> <code>StateStore</code>

Table 2. StateStoreHandlers

StateStoreHandler	Description
<code>KeyToNumValuesStore</code>	
<code>KeyWithIndexToValueStore</code>	

Tip

Enable `ALL` logging levels for
`org.apache.spark.sql.execution.streaming.state.SymmetricHashJoinStateManager.State`
happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.execution.streaming.state.SymmetricHashJoinSta
```

Refer to [Logging](#).

Performance Metrics — `metrics` Method

`metrics: StateStoreMetrics`

`metrics` simply requests the [StateStore](#) for the [StateStoreMetrics](#).

Note	<code>metrics</code> is used exclusively when SymmetricHashJoinStateManager is requested for the <code>metrics</code> .
------	---

Committing (Changes to) State Store — `commit` Method

`commit(): Unit`

`commit ...FIXME`

Note	<code>commit</code> is used when...FIXME
------	--

abortIfNeeded Method

`abortIfNeeded(): Unit`

`abortIfNeeded ...FIXME`

Note	<code>abortIfNeeded</code> is used when...FIXME
------	---

Looking Up StateStore — `getStateStore` Method

`getStateStore(keySchema: StructType, valueSchema: StructType): StateStore`

`getStateStore ...FIXME`

Note	<code>getStateStore</code> is used when KeyToNumValuesStore and KeyWithIndexToValueStore are created.
------	---

StateStoreType Contract (Sealed Trait)

`StateStoreType` is required to create a [StateStoreHandler](#).

Table 3. StateStoreTypes

StateStoreType	toString	Description
<code>KeyToNumValuesType</code>	<code>keyToNumValues</code>	
<code>KeyWithIndexToValueType</code>	<code>keyWithIndexToValue</code>	

Note

`stateStoreType` is a Scala private **sealed trait** which means that all the [implementations](#) are in the same compilation unit (a single file).

KeyToNumValuesStore

KeyToNumValuesStore is...FIXME

KeyWithIndexToValueStore

KeyWithIndexToValueStore is...FIXME

OneSideHashJoiner

`OneSideHashJoiner` is created exclusively when `StreamingSymmetricHashJoinExec` physical operator is requested to execute and generate a recipe for a distributed computation (as an `RDD[InternalRow]`) (through process partitions).

Note

`OneSideHashJoiner` is a Scala private internal class of `StreamingSymmetricHashJoinExec`.

storeAndJoinWithOtherSide Method

```
storeAndJoinWithOtherSide(  
    otherSideJoiner: OneSideHashJoiner)(  
    generateJoinedRow: (InternalRow, InternalRow) => JoinedRow): Iterator[InternalRow]
```

`storeAndJoinWithOtherSide` ...FIXME

Note

`storeAndJoinWithOtherSide` is used when `StreamingSymmetricHashJoinExec` physical operator is requested to process partitions (when requested to execute and generate a recipe for a distributed computation (as an `RDD[InternalRow]`)).

Creating OneSideHashJoiner Instance

`OneSideHashJoiner` takes the following to be created:

- `JoinSide`
- Input attributes (`Seq[Attribute]`)
- Join keys (`Seq[Expression]`)
- Input internal rows (`Iterator[InternalRow]`)
- Optional pre-join filter Catalyst expression
- Post-join filter (`(InternalRow) => Boolean`)
- Optional `JoinStateWatermarkPredicate`

`OneSideHashJoiner` initializes the internal registries and counters.

removeOldState Method

```
removeOldState(): Iterator[UnsafeRowPair]
```

`removeOldState` ...FIXME

Note

`removeOldState` is used when...FIXME

Getting Values For Key — get Method

```
get(key: UnsafeRow): Iterator[UnsafeRow]
```

`get` ...FIXME

Note

`get` is used when...FIXME

commitStateAndGetMetrics Method

```
commitStateAndGetMetrics(): StateStoreMetrics
```

`commitStateAndGetMetrics` simply requests the [SymmetricHashJoinStateManager](#) to commit and then for the [metrics](#).

Note

`commitStateAndGetMetrics` is used exclusively when `StreamingSymmetricHashJoinExec` physical operator is requested to `processPartitions` (when requested to `execute` and `generate a recipe for a distributed computation (as an RDD[InternalRow])`).

Internal Properties

Name	Description
joinStateManager	SymmetricHashJoinStateManager Used when...FIXME
keyGenerator	<code>UnsafeProjection</code> to generate join keys Used when...FIXME
preJoinFilter	<code>InternalRow ⇒ Boolean</code> Used when...FIXME
stateKeyWatermarkPredicateFunc	<code>InternalRow ⇒ Boolean</code> Used when...FIXME
stateValueWatermarkPredicateFunc	<code>InternalRow ⇒ Boolean</code> Used when...FIXME
updatedStateRowsCount	Counter Used exclusively when requested to storeAndJoinWithOtherSide

StateStoreAwareZipPartitionsHelper — Extension Methods for Creating StateStoreAwareZipPartitionsRDD

`StateStoreAwareZipPartitionsHelper` is a **Scala implicit class** of a data RDD (of type `RDD[T]`) to [create a `StateStoreAwareZipPartitionsRDD`](#) for [StreamingSymmetricHashJoinExec](#) physical operator.

Note

[Implicit Classes](#) are a language feature in Scala for **implicit conversions** with **extension methods** for existing types.

Creating StateStoreAwareZipPartitionsRDD — `stateStoreAwareZipPartitions` Method

```
stateStoreAwareZipPartitions[U: ClassTag, V: ClassTag](
  dataRDD2: RDD[U],
  stateInfo: StatefulOperatorStateInfo,
  storeNames: Seq[String],
  storeCoordinator: StateStoreCoordinatorRef
)(f: (Iterator[T], Iterator[U]) => Iterator[V]): RDD[V]
```

`stateStoreAwareZipPartitions` simply creates a new [StateStoreAwareZipPartitionsRDD](#).

Note

`stateStoreAwareZipPartitions` is used exclusively when [StreamingSymmetricHashJoinExec](#) physical operator is requested to [execute](#) and [generate a recipe for a distributed computation \(as an `RDD\[InternalRow\]`\)](#).

Streaming Aggregation

In Spark Structured Streaming, a streaming query is an example of **streaming aggregation** when it uses the following high-level operators:

- `Dataset.groupBy`, `Dataset.rollup`, `Dataset.cube` (that simply create a `RelationalGroupedDataset`)
- `Dataset.groupByKey` (that simply creates a `KeyValueGroupedDataset`)
- SQL's `GROUP BY` clause (including `WITH CUBE` and `WITH ROLLUP`)

Under the covers, the high-level operators create a logical query plan with one or more `Aggregate` logical operators.

Tip	Read up on Aggregate logical operator in The Internals of Spark SQL book.
-----	---

Demos

Use the following demos to learn more:

- [Demo: Streaming Watermark with Aggregation in Append Output Mode](#)
- [Demo: Streaming Query for Running Counts \(Socket Source and Complete Output Mode\)](#)
- [Demo: Streaming Aggregation with Kafka Data Source](#)
- [Demo: groupByKey Streaming Aggregation in Update Mode](#)

IncrementalExecution — QueryExecution of Streaming Queries

In Spark Structured Streaming it is [IncrementalExecution](#) that plans streaming queries for execution.

While [planning a streaming query for execution](#) (aka *query planning*), `IncrementalExecution` uses the [StatefulAggregationStrategy](#) execution planning strategy for planning streaming aggregations (`Aggregate` unary logical operators) as pairs of [StateStoreRestoreExec](#) and [StateStoreSaveExec](#) physical operators.

StateStoreRDD — RDD for Updating State (in StateStores Across Spark Cluster)

`StateStoreRDD` is an `RDD` for executing `storeUpdateFunction` with `StateStore` (and data from partitions of the `data RDD`).

`StateStoreRDD` is created for the following stateful physical operators (using `StateStoreOps.mapPartitionsWithStateStore`):

- `FlatMapGroupsWithStateExec`
- `StateStoreRestoreExec`
- `StateStoreSaveExec`
- `StreamingDeduplicateExec`
- `StreamingGlobalLimitExec`

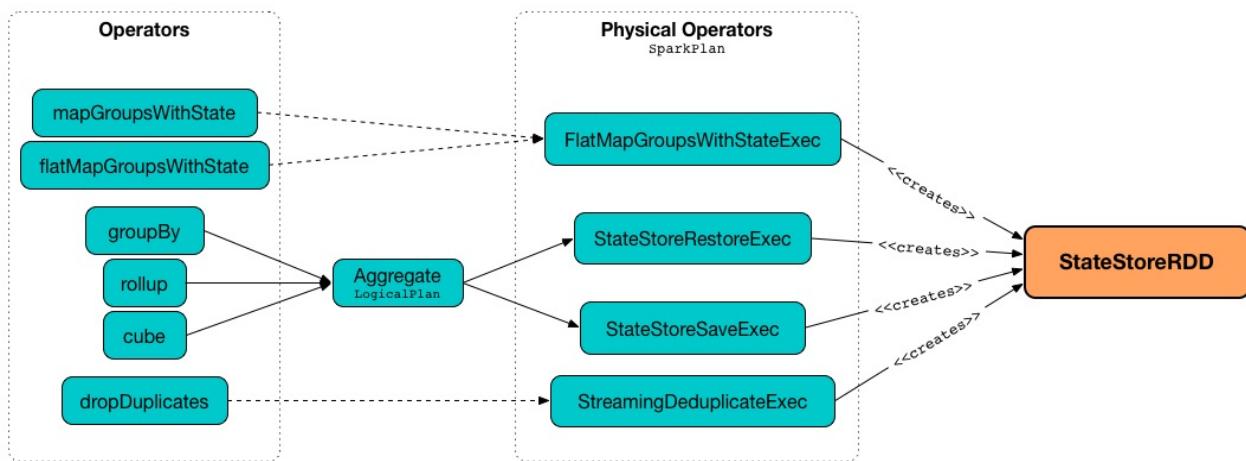


Figure 1. StateStoreRDD, Physical and Logical Plans, and operators

`StateStoreRDD` uses `StateStoreCoordinator` for the preferred locations of a partition for job scheduling.

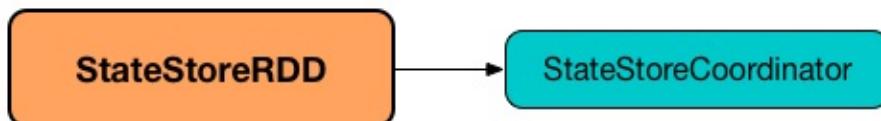


Figure 2. StateStoreRDD and StateStoreCoordinator

`getPartitions` is exactly the partitions of the `data RDD`.

Computing Partition — `compute` Method

```
compute(
    partition: Partition,
    ctxt: TaskContext): Iterator[U]
```

Note `compute` is part of the RDD Contract to compute a given partition.

`compute` computes `dataRDD` passing the result on to `storeUpdateFunction` (with a configured `StateStore`).

Internally, (and similarly to `getPreferredLocations`) `compute` creates a `StateStoreProviderId` with `statestoreId` (using `checkpointLocation`, `operatorId` and the index of the input `partition`) and `queryRunId`.

`compute` then requests `statestore` for the store for the `StateStoreProviderId`.

In the end, `compute` computes `dataRDD` (using the input `partition` and `ctxt`) followed by executing `storeUpdateFunction` (with the store and the result).

Placement Preferences of Partition (Preferred Locations)

— `getPreferredLocations` Method

```
getPreferredLocations(partition: Partition): Seq[String]
```

Note `getPreferredLocations` is a part of the RDD Contract to specify placement preferences (aka *preferred task locations*), i.e. where tasks should be executed to be as close to the data as possible.

`getPreferredLocations` creates a `StateStoreProviderId` with `statestoreId` (using `checkpointLocation`, `operatorId` and the index of the input `partition`) and `queryRunId`.

Note `checkpointLocation` and `operatorId` are shared across different partitions and so the only difference in `StateStoreProviderIds` is the partition index.

In the end, `getPreferredLocations` requests `StateStoreCoordinatorRef` for the location of the `state store` for the `StateStoreProviderId`.

Note `StateStoreCoordinator` coordinates instances of `statestores` across Spark executors in the cluster, and tracks their locations for job scheduling.

Creating StateStoreRDD Instance

`StateStoreRDD` takes the following to be created:

- Data RDD (`RDD[T]` to update the aggregates in a state store)

- Store update function (`(stateStore, Iterator[T]) → Iterator[U]` where `T` is the type of rows in the [data RDD](#))
- Checkpoint directory
- Run ID of the streaming query
- Operator ID
- Version of the store
- **Key schema** - schema of the keys
- **Value schema** - schema of the values
- Index
- `SessionState`
- Optional [StateStoreCoordinatorRef](#)

`StateStoreRDD` initializes the [internal properties](#).

Internal Properties

Name	Description
<code>hadoopConfBroadcast</code>	
<code>storeConf</code>	Configuration parameters (as <code>stateStoreConf</code>) using the current <code>SQLConf</code> (from <code>SessionState</code>)

StateStoreOps — Extension Methods for Creating StateStoreRDD

`StateStoreOps` is a **Scala implicit class** of a data RDD (of type `RDD[T]`) to [create a `StateStoreRDD`](#) for the following physical operators:

- [FlatMapGroupsWithStateExec](#)
- [StateStoreRestoreExec](#)
- [StateStoreSaveExec](#)
- [StreamingDeduplicateExec](#)

Note

[Implicit Classes](#) are a language feature in Scala for **implicit conversions** with **extension methods** for existing types.

Creating StateStoreRDD (with `storeUpdateFunction` Aborting StateStore When Task Fails)

— `mapPartitionsWithStateStore` Method

```
mapPartitionsWithStateStore[U](
    sqlContext: SQLContext,
    stateInfo: StatefulOperatorStateInfo,
    keySchema: StructType,
    valueSchema: StructType,
    indexOrdinal: Option[Int])(
    storeUpdateFunction: (StateStore, Iterator[T]) => Iterator[U]): StateStoreRDD[T, U] (
  1)
mapPartitionsWithStateStore[U](
    stateInfo: StatefulOperatorStateInfo,
    keySchema: StructType,
    valueSchema: StructType,
    indexOrdinal: Option[Int],
    sessionState: SessionState,
    storeCoordinator: Option[StateStoreCoordinatorRef])(
    storeUpdateFunction: (StateStore, Iterator[T]) => Iterator[U]): StateStoreRDD[T, U]
```

1. Uses `sqlContext.streams.stateStoreCoordinator` to access `StateStoreCoordinator`

Internally, `mapPartitionsWithStateStore` requests `SparkContext` to clean `storeUpdateFunction` function.

Note	<code>mapPartitionsWithStateStore</code> uses the enclosing RDD to access the current <code>SparkContext</code> .
Note	Function Cleaning is to clean a closure from unreferenced variables before it is serialized and sent to tasks. <code>SparkContext</code> reports a <code>SparkException</code> when the closure is not serializable.
<code>mapPartitionsWithStateStore</code> then creates a (wrapper) function to <code>abort</code> the <code>stateStore</code> if <code>state updates had not been committed</code> before a task finished (which is to make sure that the <code>StateStore</code> has been <code>committed</code> or <code>aborted</code> in the end to follow the contract of <code>StateStore</code>).	
Note	<code>mapPartitionsWithStateStore</code> uses <code>TaskCompletionListener</code> to be notified when a task has finished.
In the end, <code>mapPartitionsWithStateStore</code> creates a <code>StateStoreRDD</code> (with the wrapper function, <code>SessionState</code> and <code>StateStoreCoordinatorRef</code>).	
Note	<p><code>mapPartitionsWithStateStore</code> is used when the following physical operators are executed:</p> <ul style="list-style-type: none"> • <code>FlatMapGroupsWithStateExec</code> • <code>StateStoreRestoreExec</code> • <code>StateStoreSaveExec</code> • <code>StreamingDeduplicateExec</code> • <code>StreamingGlobalLimitExec</code>

StreamingAggregationStateManager Contract — State Managers for Streaming Aggregation

`StreamingAggregationStateManager` is the [abstraction](#) of [state managers](#) that act as [middlemen](#) between [state stores](#) and the physical operators used in [Streaming Aggregation](#) (e.g. [StateStoreSaveExec](#) and [StateStoreRestoreExec](#)).

Table 1. StreamingAggregationStateManager Contract

Method	Description
<code>commit</code>	<pre>commit(store: StateStore): Long</pre> <p>Commits all the updates to the state store and returns the new version Used exclusively when StateStoreSaveExec physical operator is executed.</p>
<code>get</code>	<pre>get(store: StateStore, key: UnsafeRow): UnsafeRow</pre> <p>Looks up the value of the key from the state store (the key is non-null) Used exclusively when StateStoreRestoreExec physical operator is executed.</p>
<code>getKey</code>	<pre>getKey(row: UnsafeRow): UnsafeRow</pre> <p>Extracts the columns for the key from the input row Used when: <ul style="list-style-type: none"> • StateStoreRestoreExec physical operator is executed • <code>StreamingAggregationStateManagerImplV1</code> legacy state manager is requested to put a row to a state store </p>
<code>getStateValueSchema</code>	<pre>getStateValueSchema: StructType</pre> <p>Gets the schema of the values in a state store Used when StateStoreRestoreExec and StateStoreSaveExec physical operators are executed</p>

	<code>iterator(store: StateStore): Iterator[UnsafeRowPair]</code>
<code>iterator</code>	<p>Returns all the <code>UnsafeRow</code> key-value pairs in the <code>state store</code></p> <p>Used exclusively when <code>StateStoreSaveExec</code> physical operator is executed.</p>
<code>keys</code>	<code>keys(store: StateStore): Iterator[UnsafeRow]</code> <p>Returns all the keys in the <code>state store</code></p> <p>Used exclusively when physical operators with <code>WatermarkSupport</code> are requested to <code>removeKeysOlderThanWatermark</code> (i.e. exclusively when <code>StateStoreSaveExec</code> physical operator is executed).</p>
<code>put</code>	<code>put(store: StateStore, row: UnsafeRow): Unit</code> <p>Stores the row in the <code>state store</code></p> <p>Used exclusively when <code>StateStoreSaveExec</code> physical operator is executed.</p>
<code>remove</code>	<code>remove(store: StateStore, key: UnsafeRow): Unit</code> <p>Removes the key from the <code>state store</code></p> <p>Used exclusively when <code>StateStoreSaveExec</code> physical operator is executed (directly or indirectly as a <code>WatermarkSupport</code>)</p>
<code>values</code>	<code>values(store: StateStore): Iterator[UnsafeRow]</code> <p>Returns all the values in the <code>state store</code></p> <p>Used exclusively when <code>StateStoreSaveExec</code> physical operator is executed.</p>

`StreamingAggregationStateManager` supports two versions of state managers for streaming aggregations (per the `spark.sql.streaming.aggregation.stateFormatVersion` internal configuration property):

- 1 (for the legacy `StreamingAggregationStateManagerImplV1`)

- 2 (for the default `StreamingAggregationStateManagerImplV2`)

Note `StreamingAggregationStateManagerBaseImpl` is the one and only known direct implementation of the `StreamingAggregationStateManager Contract` in Spark Structured Streaming.

Note `StreamingAggregationStateManager` is a Scala **sealed trait** which means that all the `implementations` are in the same compilation unit (a single file).

Creating `StreamingAggregationStateManager` Instance — `createStateManager` Factory Method

```
createStateManager(  
    keyExpressions: Seq[Attribute],  
    inputRowAttributes: Seq[Attribute],  
    stateFormatVersion: Int): StreamingAggregationStateManager
```

`createStateManager` creates a new `StreamingAggregationStateManager` for a given `stateFormatVersion`:

- `StreamingAggregationStateManagerImplV1` for `stateFormatVersion` being 1
- `StreamingAggregationStateManagerImplV2` for `stateFormatVersion` being 2

`createStateManager` throws a `IllegalArgumentException` for any other `stateFormatVersion`:

Version [stateFormatVersion] is invalid

Note `createStateManager` is used when `StateStoreRestoreExec` and `StateStoreSaveExec` physical operators are created.

StreamingAggregationStateManagerBaseImpl — Base State Manager for Streaming Aggregation

`StreamingAggregationStateManagerBaseImpl` is the base implementation of the `StreamingAggregationStateManager` contract for state managers for streaming aggregations that use `UnsafeProjection` to `getKey`.

`StreamingAggregationStateManagerBaseImpl` uses `UnsafeProjection` to `getKey`.

Table 1. `StreamingAggregationStateManagerBaseImpls`

<code>StreamingAggregationStateManagerBaseImpl</code>	Description
<code>StreamingAggregationStateManagerImplV1</code>	Legacy <code>StreamingAggregationStateManager</code> when <code>spark.sql.streaming.aggregation.state</code> configuration property is <code>1</code>)
<code>StreamingAggregationStateManagerImplV2</code>	Default <code>StreamingAggregationStateManager</code> when <code>spark.sql.streaming.aggregation.state</code> configuration property is <code>2</code>)

`StreamingAggregationStateManagerBaseImpl` takes the following to be created:

- Catalyst expressions for the keys (`Seq[Attribute]`)
- Catalyst expressions for the input rows (`Seq[Attribute]`)

Note

`StreamingAggregationStateManagerBaseImpl` is a Scala abstract class and cannot be `created` directly. It is created indirectly for the `concrete` `StreamingAggregationStateManagerBaseImpls`.

Committing (Changes to) State Store — `commit` Method

```
commit(store: StateStore): Long
```

Note

`commit` is part of the `StreamingAggregationStateManager` Contract to commit changes to a `state store`.

`commit` simply requests the `state store` to `commit` state changes.

Removing Key From State Store — `remove` Method

```
remove(store: StateStore, key: UnsafeRow): Unit
```

Note

`remove` is part of the [StreamingAggregationStateManager Contract](#) to remove a key from a state store.

`remove` ...FIXME

getKey Method

```
getKey(row: UnsafeRow): UnsafeRow
```

Note

`getKey` is part of the [StreamingAggregationStateManager Contract](#) to...FIXME

`getKey` ...FIXME

Getting All Keys in State Store — `keys` Method

```
keys(store: StateStore): Iterator[UnsafeRow]
```

Note

`keys` is part of the [StreamingAggregationStateManager Contract](#) to get all keys in a state store (as an iterator).

`keys` ...FIXME

StreamingAggregationStateManagerImplV1 — Legacy State Manager for Streaming Aggregation

`StreamingAggregationStateManagerImplV1` is the legacy state manager for streaming aggregations.

Note	The version of a state manager is controlled using <code>spark.sql.streaming.aggregation.stateFormatVersion</code> internal configuration property.
------	---

`StreamingAggregationStateManagerImplV1` is created exclusively when `StreamingAggregationStateManager` is requested for a new `StreamingAggregationStateManager`.

Storing Row in State Store — `put` Method

```
put(store: StateStore, row: UnsafeRow): Unit
```

Note	<code>put</code> is part of the <code>StreamingAggregationStateManager Contract</code> to store a row in a state store.
------	---

`put` ...FIXME

Creating `StreamingAggregationStateManagerImplV1` Instance

`StreamingAggregationStateManagerImplV1` takes the following when created:

- Attribute expressions for keys (`Seq[Attribute]`)
- Attribute expressions of input rows (`Seq[Attribute]`)

StreamingAggregationStateManagerImplV2 — Default State Manager for Streaming Aggregation

`StreamingAggregationStateManagerImplV2` is the default state manager for streaming aggregations.

Note	The version of a state manager is controlled using <code>spark.sql.streaming.aggregation.stateFormatVersion</code> internal configuration property.
------	---

`StreamingAggregationStateManagerImplV2` is created exclusively when `StreamingAggregationStateManager` is requested for a new `StreamingAggregationStateManager`.

`StreamingAggregationStateManagerImplV2` (like the parent `StreamingAggregationStateManagerBaseImpl`) takes the following to be created:

- Catalyst expressions for the keys (`Seq[Attribute]`)
- Catalyst expressions for the input rows (`Seq[Attribute]`)

Storing Row in State Store — put Method

```
put(store: StateStore, row: UnsafeRow): Unit
```

Note	<code>put</code> is part of the <code>StreamingAggregationStateManager Contract</code> to store a row in a state store.
------	---

`put` ...FIXME

Getting Saved State for Non-Null Key from State Store — get Method

```
get(store: StateStore, key: UnsafeRow): UnsafeRow
```

Note	<code>get</code> is part of the <code>StreamingAggregationStateManager Contract</code> to get the saved state for a given non-null key from a given state store.
------	--

`get` requests the given `StateStore` for the current state value for the given key.

`get` returns `null` if the key could not be found in the state store. Otherwise, `get restoreOriginalRow` (for the key and the saved state).

restoreOriginalRow Internal Method

```
restoreOriginalRow(key: UnsafeRow, value: UnsafeRow): UnsafeRow
restoreOriginalRow(rowPair: UnsafeRowPair): UnsafeRow
```

`restoreOriginalRow ...FIXME`

Note	<code>restoreOriginalRow</code> is used when <code>StreamingAggregationStateManagerImplV2</code> is requested to get the saved state for a given non-null key from a state store , iterator and values .
------	--

getStateValueSchema Method

```
getStateValueSchema: StructType
```

Note	<code>getStateValueSchema</code> is part of the StreamingAggregationStateManager Contract to... FIXME.
------	---

`getStateValueSchema` simply requests the [valueExpressions](#) for the schema.

iterator Method

```
iterator: iterator(store: StateStore): Iterator[UnsafeRowPair]
```

Note	<code>iterator</code> is part of the StreamingAggregationStateManager Contract to... FIXME.
------	--

`iterator` simply requests the input [state store](#) for the [iterator](#) that is mapped to an iterator of [UnsafeRowPairs](#) with the key (of the input [UnsafeRowPair](#)) and the value as a [restored original row](#).

Note	<code>scala.collection.Iterator</code> is a data structure that allows to iterate over a sequence of elements that are usually fetched lazily (i.e. no elements are fetched from the underlying store until processed).
------	---

values Method

```
values(store: StateStore): Iterator[UnsafeRow]
```

Note	values is part of the StreamingAggregationStateManager Contract to...FIXME.
------	---

values ...FIXME

Internal Properties

Name	Description
joiner	
keyValueJoinedExpressions	
needToProjectToRestoreValue	
restoreValueProjector	
valueExpressions	
valueProjector	

Stateful Stream Processing

Stateful Stream Processing is a stream processing with some state (implicit or explicit).

In Spark Structured Streaming, a streaming query is stateful when it is one of the following:

- Streaming Aggregation
- Arbitrary Stateful Streaming Aggregation
- Streaming Join
- Streaming Deduplication
- Streaming Limit

IncrementalExecution — QueryExecution of Streaming Queries

Regardless of the query language ([Dataset API](#) or SQL), any structured query (incl. streaming queries) becomes a logical query plan.

In Spark Structured Streaming it is [IncrementalExecution](#) that plans streaming queries for execution.

While [planning a streaming query for execution](#) (aka *query planning*), [IncrementalExecution](#) uses the [state preparation rule](#). The rule fills out the following physical operators with the execution-specific configuration (with [StatefulOperatorStateInfo](#) being the most important for stateful stream processing):

- [StateStoreSaveExec](#) (used for [streaming aggregation](#))
- [StateStoreRestoreExec](#)
- [StreamingDeduplicateExec](#)
- [FlatMapGroupsWithStateExec](#)
- [StreamingSymmetricHashJoinExec](#)
- [StreamingGlobalLimitExec](#)

Micro-Batch Stream Processing and Extra Non-Data Batch for StateStoreWriter Stateful Operators

In [Micro-Batch Stream Processing](#) (with [MicroBatchExecution engine](#)), `IncrementalExecution` uses `shouldRunAnotherBatch` flag that allows `StateStoreWriters` stateful physical operators to indicate whether the last batch execution requires another non-data batch.

The following table shows the `StateStoreWriters` that redefine `shouldRunAnotherBatch` flag.

Table 1. StateStoreWriters and `shouldRunAnotherBatch` Flag

StateStoreWriter	shouldRunAnotherBatch Flag
<code>FlatMapGroupsWithStateExec</code>	Based on <code>GroupStateTimeout</code>
<code>StateStoreSaveExec</code>	Based on <code>OutputMode</code> and event-time watermark
<code>StreamingDeduplicateExec</code>	Based on event-time watermark
<code>StreamingSymmetricHashJoinExec</code>	Based on event-time watermark

StateStoreRDD

Right after [query planning](#), a stateful streaming query (a single micro-batch actually) becomes an RDD with one or more `StateStoreRDDs`.

You can find the `StateStoreRDDs` of a streaming query in the RDD lineage.

```
scala> :type streamingQuery
org.apache.spark.sql.streaming.StreamingQuery

scala> streamingQuery.explain
== Physical Plan ==
*(4) HashAggregate(keys=[window#13-T0ms, value#3L], functions=[count(1)])
+- StateStoreSave [window#13-T0ms, value#3L], state info [ checkpoint = file:/tmp/checkpoint-counts/state, runId = 1dec2d81-f2d0-45b9-8f16-39ede66e13e7, opId = 0, ver = 1, numPartitions = 1], Append, 10000, 2
    +- *(3) HashAggregate(keys=[window#13-T0ms, value#3L], functions=[merge_count(1)])
        +- StateStoreRestore [window#13-T0ms, value#3L], state info [ checkpoint = file:/tmp/checkpoint-counts/state, runId = 1dec2d81-f2d0-45b9-8f16-39ede66e13e7, opId = 0, ver = 1, numPartitions = 1], 2
            +- *(2) HashAggregate(keys=[window#13-T0ms, value#3L], functions=[merge_count(1)])
                +- Exchange hashpartitioning(window#13-T0ms, value#3L, 1)
                    +- *(1) HashAggregate(keys=[window#13-T0ms, value#3L], functions=[partial_count(1)])
                        +- *(1) Project [named_struct(start, precisetimestampconversion(((CASE WHEN (cast(CEIL((cast((precisetimestampconversion(time#2-T0ms, TimestampType, LongType) - 0) as double) / 5000000.0)) as double) = (cast((precisetimestampconversion(time#2-T0ms, TimestampType, LongType) - 0) as double) / 5000000.0)) THEN (CEIL((cast((pr
```

```

ecisetimestampconversion(time#2-T0ms, TimestampType, LongType) - 0) as double) / 500000.0)) + 1) ELSE CEIL((cast((precisetimestampconversion(time#2-T0ms, TimestampType, LongType) - 0) as double) / 5000000.0)) END + 0) - 1) * 5000000) + 0), LongType, TimestampType), end, precisetimestampconversion((((CASE WHEN (cast(CEIL((cast((precisetimestampconversion(time#2-T0ms, TimestampType, LongType) - 0) as double) / 5000000.0)) as double) = (cast((precisetimestampconversion(time#2-T0ms, TimestampType, LongType) - 0) as double) / 5000000.0)) THEN (CEIL((cast((precisetimestampconversion(time#2-T0ms, TimestampType, LongType) - 0) as double) / 5000000.0)) + 1) ELSE CEIL((cast((precisetimestampconversion(time#2-T0ms, TimestampType, LongType) - 0) as double) / 5000000.0)) END + 0) - 1) * 5000000) + 5000000), LongType, TimestampType)) AS window#13-T0ms, value#3L]
+- *(1) Filter isnotnull(time#2-T0ms)
+- EventTimeWatermark time#2: timestamp, interval
+- LocalTableScan <empty>, [time#2, value#3L]

import org.apache.spark.sql.execution.streaming.{StreamExecution, StreamingQueryWrapper}
}
val se = streamingQuery.asInstanceOf[StreamingQueryWrapper].streamingQuery

scala> :type se
org.apache.spark.sql.execution.streaming.StreamExecution

scala> :type se.lastExecution
org.apache.spark.sql.execution.streaming.IncrementalExecution

val rdd = se.lastExecution.toRdd
scala> rdd.toDebugString
res3: String =
(1) MapPartitionsRDD[39] at toRdd at <console>:40 []
| StateStoreRDD[38] at toRdd at <console>:40 [] // <-- here
| MapPartitionsRDD[37] at toRdd at <console>:40 []
| StateStoreRDD[36] at toRdd at <console>:40 [] // <-- here
| MapPartitionsRDD[35] at toRdd at <console>:40 []
| ShuffledRowRDD[17] at start at <pastie>:67 []
+-(1) MapPartitionsRDD[16] at start at <pastie>:67 []
| MapPartitionsRDD[15] at start at <pastie>:67 []
| MapPartitionsRDD[14] at start at <pastie>:67 []
| MapPartitionsRDD[13] at start at <pastie>:67 []
| ParallelCollectionRDD[12] at start at <pastie>:67 []

```

When planned for execution, the `StateStoreRDD` is first asked for the [preferred locations of a partition](#) (which happens on the driver) and to [compute it](#) (later on an executor).

`StateStoreRDD` uses `StateStoreId` to uniquely identify the `state store` to use for ([associate with](#)) a stateful operator and a partition.

State Management

The state in a stateful streaming query can be implicit or explicit.

Streaming Watermark

Streaming Watermark (Event-Time Watermark) of a [stateful streaming query](#) is a **time threshold** to wait for late and possibly out-of-order events until a streaming state can be considered final.

Streaming watermark is used to mark events that are older than the threshold as "too late", and not "interesting" to update partial non-final aggregates.

With streaming watermark, memory usage of a streaming state can be controlled as late events can easily be dropped, and old aggregates that are never going to be updated removed. That avoids unbounded streaming state that would inevitably use up all the available memory of long-running streaming queries and end up in out of memory errors.

In Spark Structured Streaming, you use [Dataset.withWatermark](#) high-level operator to define a streaming watermark.

A streaming watermark has to be defined on one or many grouping expressions of a streaming aggregation (directly or using [window](#) standard function).

Note	Dataset.withWatermark operator has to be used before an aggregation operator (for the watermark to have an effect).
------	---

In [Append](#) output mode the current event-time streaming watermark is used for the following:

- Output saved state rows that became expired (**Expired events** in the demo)
- Dropping late events, i.e. don't save them to a state store or include in aggregation (**Late events** in the demo)

Streaming watermark is [required](#) for a [streaming aggregation](#) in [append](#) output mode.

Watermark delay says how late and possibly out-of-order events are still acceptable and contribute to the final result of a stateful streaming query.

Demos

Use the following demos to learn more:

- [Demo: Streaming Watermark with Aggregation in Append Output Mode](#)

Internals

Under the covers, [Dataset.withWatermark](#) high-level operator creates a logical query plan with [EventTimeWatermark](#) logical operator.

`EventTimeWatermark` logical operator is planned to [EventTimeWatermarkExec](#) physical operator that extracts the event times (from the data being processed) and adds them to an accumulator.

Since the execution (data processing) happens on Spark executors, using the accumulator is the only *Spark-approved way* for communication between the tasks (on the executors) and the driver. Using accumulator updates the driver with the current event-time watermark.

During the query planning phase (in [MicroBatchExecution](#) and [ContinuousExecution](#)) that also happens on the driver, `IncrementalExecution` is given the current [OffsetSeqMetadata](#) with the current event-time watermark.

Further Reading Or Watching

- [SPARK-18124 Observed delay based event time watermarks](#)

Streaming Deduplication

Streaming Deduplication is...FIXME

Streaming Limit

Streaming Limit is...FIXME

StateStore Contract — Kay-Value Store for Streaming State Data

`StateStore` is the [abstraction](#) of a [versioned](#) and possibly fault-tolerant [key-value stores](#) for streaming state data (e.g. for persisting running aggregates in [Streaming Aggregation](#)).

`StateStore` supports [incremental checkpointing](#) in which only the key-value "Row" pairs that changed can be [committed](#) or [aborted](#) (without touching other key-value pairs).

`StateStore` is identified with the [aggregating operator id](#) and the [partition id](#) (among other properties for identification).

Note

[HDFSBackedStateStore](#) is the default and only known implementation of the [StateStore Contract](#) in Spark Structured Streaming.

Table 1. StateStore Contract

Method	Description
abort	<p><code>abort(): Unit</code></p> <p>Aborts (<i>discards</i>) changes to the state store</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>StateStoreOps</code> implicit class is requested to mapPartitionsWithStateStore (when the state store has not been committed for a task that finishes, possibly with an error) • <code>StateStoreHandler</code> (of SymmetricHashJoinStateManager) is requested to abortIfNeeded (when the state store has not been committed for a task that finishes, possibly with an error)
commit	<p><code>commit(): Long</code></p> <p>Commits the changes to the state store (and returns the current version)</p> <p>Used when:</p> <ul style="list-style-type: none"> • FlatMapGroupsWithStateExec, StreamingDeduplicateExec and StreamingGlobalLimitExec physical operators are executed (right after all rows in a partition have been processed)

	<ul style="list-style-type: none"> • <code>StreamingAggregationStateManagerBaseImpl</code> is requested to commit (changes to) a state store (exclusively when <code>StateStoreSaveExec</code> physical operator is executed) • <code>StateStoreHandler</code> (of <code>SymmetricHashJoinStateManager</code>) is requested to commit changes to a state store 		
	<pre>get(key: UnsafeRow): UnsafeRow</pre> <p>Gets the value of the (non-null) key</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>StreamingDeduplicateExec</code> and <code>StreamingGlobalLimitExec</code> physical operators are executed 		
get	<ul style="list-style-type: none"> • <code>StateManagerImplBase</code> (of <code>FlatMapGroupsWithStateExecHelper</code>) is requested to <code>getState</code> • <code>StreamingAggregationStateManagerImplV1</code> and <code>StreamingAggregationStateManagerImplV2</code> are requested to get the value of a non-null key • <code>KeyToNumValuesStore</code> is requested to get • <code>KeyWithIndexToValueStore</code> is requested to get and getAll 		
getRange	<pre>getRange(start: Option[UnsafeRow], end: Option[UnsafeRow]): Iterator[UnsafeRowPair]</pre> <p>Gets the key-value pairs for the specified range (with optional approximate <code>start</code> and <code>end</code> extents)</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>WatermarkSupport</code> is requested to removeKeysOlderThanWatermark • <code>StateManagerImplBase</code> is requested to <code>getAllState</code> • <code>StreamingAggregationStateManagerBaseImpl</code> is requested to <code>keys</code> • <code>KeyToNumValuesStore</code> and <code>KeyWithIndexToValueStore</code> are requested to <code>iterator</code> <table border="1"> <tr> <td style="padding: 5px;">Note</td> <td style="padding: 5px;">All the uses above assume the <code>start</code> and <code>end</code> as <code>None</code> that basically is <code>iterator</code>.</td> </tr> </table>	Note	All the uses above assume the <code>start</code> and <code>end</code> as <code>None</code> that basically is <code>iterator</code> .
Note	All the uses above assume the <code>start</code> and <code>end</code> as <code>None</code> that basically is <code>iterator</code> .		

	<pre>hasCommitted: Boolean</pre> <p>Returns whether all changes have been committed (<code>true</code>) or not (<code>false</code>)</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>stateStoreOps</code> implicit class is requested to mapPartitionsWithStateStore (when a task finishes, possibly with an error, to check whether to abort state updates) • <code>StateStoreHandler</code> (of SymmetricHashJoinStateManager) is requested to abortIfNeeded (when a task finishes, possibly with an error, to check whether to abort state updates)
<code>id</code>	<pre>id: StateStoreId</pre> <p>The ID of the state store</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>HDFSBackedStateStore</code> state store is requested for the textual representation • <code>StateStoreHandler</code> (of SymmetricHashJoinStateManager) is requested to abortIfNeeded and getStateStore
<code>iterator</code>	<pre>iterator(): Iterator[UnsafeRowPair]</pre> <p>Returns an iterator with all the key-value pairs in the state store</p> <p>Used when:</p> <ul style="list-style-type: none"> • StateStoreRestoreExec physical operator is requested to execute • <code>HDFSBackedStateStore</code> state store in particular and any <code>StateStore</code> in general are requested to getRange • <code>StreamingAggregationStateManagerImplV1</code> state manager is requested for the iterator and values • <code>StreamingAggregationStateManagerImplV2</code> state manager is requested to iterator and values
	<pre>metrics: StateStoreMetrics</pre>

metrics	<p>StateStoreMetrics of the state store</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>StateStoreWriter</code> stateful physical operator is requested to setStoreMetrics • <code>StateStoreHandler</code> (of SymmetricHashJoinStateManager) is requested to commit and for the metrics
put	<pre>put(key: UnsafeRow, value: UnsafeRow): Unit</pre> <p>Stores (<i>puts</i>) the value for the (non-null) key</p> <p>Used when:</p> <ul style="list-style-type: none"> • StreamingDeduplicateExec and StreamingGlobalLimitExec physical operators are executed • <code>StateManagerImplBase</code> is requested to putState • StreamingAggregationStateManagerImplV1 and StreamingAggregationStateManagerImplV2 are requested to store a row in a state store • KeyToNumValuesStore and KeyWithIndexToValueStore are requested to store a new value for a given key
remove	<pre>remove(key: UnsafeRow): Unit</pre> <p>Removes the (non-null) key from the state store</p> <p>Used when:</p> <ul style="list-style-type: none"> • Physical operators with <code>WatermarkSupport</code> are requested to removeKeysOlderThanWatermark • <code>StateManagerImplBase</code> is requested to removeState • StreamingAggregationStateManagerBaseImpl is requested to remove a key from a state store • <code>KeyToNumValuesStore</code> is requested to remove a key • <code>KeyWithIndexToValueStore</code> is requested to remove a key and removeAllValues
version	<pre>version: Long</pre> <p>Version of the state store</p>

Used exclusively when `HDFSBackedStateStore` state store is requested for a [new version](#) (that simply the current version incremented)

Note

`StateStore` was introduced in [\[SPARK-13809\]](#)[\[SQL\]](#) State store for streaming aggregations.

Read the motivation and design in [State Store for Streaming Aggregations](#).

Tip

Enable `ALL` logging level for `org.apache.spark.sql.execution.streaming.state.StateStore$` logger to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.execution.streaming.state.StateStore$=ALL
```

Refer to [Logging](#).

Creating (and Caching) RPC Endpoint Reference to StateStoreCoordinator for Executors — `coordinatorRef` Internal Object Method

```
coordinatorRef: Option[StateStoreCoordinatorRef]
```

`coordinatorRef` requests the `SparkEnv` helper object for the current `SparkEnv`.

If the `SparkEnv` is available and the `_coordRef` is not assigned yet, `coordinatorRef` prints out the following DEBUG message to the logs followed by requesting the `StateStoreCoordinatorRef` for the [StateStoreCoordinator endpoint](#).

```
Getting StateStoreCoordinatorRef
```

If the `SparkEnv` is available, `coordinatorRef` prints out the following INFO message to the logs:

```
Retrieved reference to StateStoreCoordinator: [_coordRef]
```

Note

`coordinatorRef` is used when `StateStore` helper object is requested to [reportActiveStoreInstance](#) (when `StateStore` object helper is requested to [find the StateStore by StateStoreProviderId](#)) and [verifyIfStoreInstanceActive](#) (when `StateStore` object helper is requested to [doMaintenance](#)).

Unloading State Store Provider — `unload` Method

```
unload(storeProviderId: StateStoreProviderId): Unit
```

`unload` ...FIXME

Note

`unload` is used when `StateStore` helper object is requested to [stop](#) and [doMaintenance](#).

`stop` Object Method

```
stop(): Unit
```

`stop` ...FIXME

Note

`stop` seems only be used in tests.

Announcing New StateStoreProvider — `reportActiveStoreInstance` Internal Object Method

```
reportActiveStoreInstance(storeProviderId: StateStoreProviderId): Unit
```

`reportActiveStoreInstance` takes the current host and `executorId` (from the `BlockManager` on the Spark executor) and requests the [StateStoreCoordinatorRef](#) to [reportActiveInstance](#).

Note

`reportActiveStoreInstance` uses `SparkEnv` to access the `BlockManager`.

In the end, `reportActiveStoreInstance` prints out the following INFO message to the logs:

```
Reported that the loaded instance [storeProviderId] is active
```

Note

`reportActiveStoreInstance` is used exclusively when `StateStore` helper object is requested to [find the StateStore by StateStoreProviderId](#).

MaintenanceTask Daemon Thread

`MaintenanceTask` is a daemon thread that [triggers maintenance work of every registered StateStoreProvider](#).

When an error occurs, `MaintenanceTask` clears `loadedProviders` registry.

`MaintenanceTask` is scheduled on **state-store-maintenance-task** thread pool that runs periodically every `spark.sql.streaming.stateStore.maintenanceInterval` configuration property (default: `60s`).

Looking Up StateStore by ID and Version — `get` Factory Method

```
get(
  storeProviderId: StateStoreProviderId,
  keySchema: StructType,
  valueSchema: StructType,
  indexOrdinal: Option[Int],
  version: Long,
  storeConf: StateStoreConf,
  hadoopConf: Configuration): StateStore
```

`get` finds `StateStore` for the given `StateStoreProviderId`.

Internally, `get` looks up the `StateStoreProvider` (for `storeProviderId`) in `loadedProviders` registry. If unavailable, `get` creates and initializes one.

`get` will also `start the periodic maintenance task` (unless already started) and announce the new `StateStoreProvider`.

In the end, `get` gets the `StateStore` (for the `version`).

Note	<p><code>get</code> is used when:</p> <ul style="list-style-type: none"> • <code>StateStoreRDD</code> is requested to <code>compute</code> • <code>StateStoreHandler</code> (of <code>SymmetricHashJoinStateManager</code>) is requested to <code>getStateStore</code>
-------------	--

Starting Periodic Maintenance Task (Unless Already Started) — `startMaintenanceIfNeeded` Internal Object Method

```
startMaintenanceIfNeeded(): Unit
```

`startMaintenanceIfNeeded` schedules `MaintenanceTask` to start after and every `spark.sql.streaming.stateStore.maintenanceInterval` (defaults to `60s`).

Note

`startMaintenanceIfNeeded` does nothing when the maintenance task has already been started and is still running.

Note

`startMaintenanceIfNeeded` is used exclusively when `statestore` is requested to find the StateStore by `StateStoreProviderId`.

Doing Maintenance of Registered State Store Providers

— `doMaintenance` Internal Object Method

```
doMaintenance(): Unit
```

Internally, `doMaintenance` prints the following DEBUG message to the logs:

```
Doing maintenance
```

`doMaintenance` then requests every `StateStoreProvider` (registered in `loadedProviders`) to do its own internal maintenance (only when a `StateStoreProvider` is still active).

When a `StateStoreProvider` is inactive, `doMaintenance` removes it from the provider registry and prints the following INFO message to the logs:

```
Unloaded [provider]
```

Note

`doMaintenance` is used exclusively in `MaintenanceTask` daemon thread.

`verifyIfStoreInstanceActive` Internal Object Method

```
verifyIfStoreInstanceActive(storeProviderId: StateStoreProviderId): Boolean
```

```
verifyIfStoreInstanceActive ...FIXME
```

Note

`verifyIfStoreInstanceActive` is used exclusively when `StateStore` helper object is requested to `doMaintenance` (from a running `MaintenanceTask` daemon thread).

Internal Properties

Name	Description
loadedProviders	Loaded providers , i.e. <code>StateStoreProviders</code> per <code>StateStoreProviderId</code> Used in...FIXME
_coordRef	<code>StateStoreCoordinator</code> RPC endpoint (a <code>RpcEndpointRef</code> to <code>StateStoreCoordinator</code>) Used in...FIXME

StateStoreId — Unique Identifier of State Store

`StateStoreId` is a unique identifier of a [state store](#) with the following attributes:

- **Checkpoint Root Location** - the root directory for state checkpointing
- **Operator ID** - a unique ID of the stateful operator
- **Partition ID** - the index of the partition
- **Store Name** - the name of the [state store](#) (default: `default`)

`StateStoreId` is [created](#) when:

- `StateStoreRDD` is requested for the [preferred locations of a partition](#) (executed on the driver) and to [compute it](#) (later on an executor)
- `StateStoreProviderId` helper object is requested to create a `StateStoreProviderId` (with a `StateStoreId` and the run ID of a streaming query) that is then used for the [preferred locations of a partition](#) of a `StateStoreAwareZipPartitionsRDD` (executed on the driver) and to...FIXME

The name of the **default state store** (for reading state store data that was generated before store names were used, i.e. in Spark 2.2 and earlier) is `default`.

storeCheckpointLocation Method

```
storeCheckpointLocation(): Path
```

`storeCheckpointLocation` gives the Hadoop HDFS's [Path](#) of the checkpoint location (for the stateful operator by [operator ID](#), the partition by the [partition ID](#) in the [checkpoint root location](#)).

If the [default store name](#) is used (for Spark 2.2 and earlier), the `storeName` is not included in the path.

Note

`storeCheckpointLocation` is used exclusively when `HDFSBackedStateStoreProvider` is requested for the [state checkpoint base directory](#).

HDFSBackedStateStore — State Store on HDFS-Compatible File System

`HDFSBackedStateStore` is a concrete [StateStore](#) that uses a HDFS-compatible file system for versioned state persistence.

`HDFSBackedStateStore` is [created](#) exclusively when `HDFSBackedStateStoreProvider` is requested for a [state store for a given version](#) (when `StateStore` helper object is requested to [retrieve the StateStore for a given ID and version](#)).

`HDFSBackedStateStore` can be in the following [states](#):

- `UPDATING`
- `COMMITTED`
- `ABORTED`

`HDFSBackedStateStore` uses the [StateStoreId](#) of the owning [HDFSBackedStateStoreProvider](#).

When requested for the textual representation, `HDFSBackedStateStore` gives `HDFSStateStore[id=(op=[operatorId],part=[partitionId]),dir=[baseDir]]`.

`HDFSBackedStateStore` takes the following to be created:

- Current version
- Key-value registry of `UnsafeRows` (as [java.util.concurrent.ConcurrentHashMap](#))

Table 1. HDFSBackedStateStore's Internal Registries

Name	Description
compressedStream	compressedStream: DataOutputStream The compressed java.io.DataOutputStream for the deltaFileStream
deltaFileStream	deltaFileStream: CheckpointFileManager.CancellableFSDataOutputStream
finalDeltaFile	finalDeltaFile: Path The Hadoop Path of the deltaFile for the version
newVersion	newVersion: Long Used exclusively when HDFSBackedStateStore is requested for the finalDeltaFile , to commit and abort
state	state: STATE
Tip	HDFSBackedStateStore is an internal class of HDFSBackedStateStoreProvider and uses its logger .

writeUpdateToDeltaFile Internal Method

```
writeUpdateToDeltaFile(
    output: DataOutputStream,
    key: UnsafeRow,
    value: UnsafeRow): Unit
```

Caution

FIXME

put Method

```
put(key: UnsafeRow, value: UnsafeRow): Unit
```

Note

put is a part of [StateStore Contract](#) to...FIXME

`put` stores the copies of the key and value in `mapToUpdate` internal registry followed by writing them to a delta file (using `tempDeltaFileStream`).

Note

`put` can only be used when `HDFSBackedStateStore` is in `UPDATING` state and reports a `IllegalStateException` otherwise.

Cannot put after already committed or aborted

Committing State Changes — `commit` Method

`commit(): Long`

Note

`commit` is part of the [StateStore Contract](#) to commit state changes.

`commit` firstly `commitUpdates` (with the `newVersion`, the `mapToUpdate` and the compressed stream).

`commit` sets the state to `COMMITTED`.

`commit` prints out the following INFO message to the logs:

Committed version [newVersion] for [this] to file [finalDeltaFile]

`commit` returns a `newVersion`.

`commit` throws a `IllegalStateException` when executed in any state but `UPDATING` state:

Cannot commit after already committed or aborted

`commit` throws a `IllegalStateException` for any `NonFatal` exception:

Error committing version [newVersion] into [this]

Aborting State Changes — `abort` Method

`abort(): Unit`

Note

`abort` is part of the [StateStore Contract](#) to abort the state changes.

`abort` ...FIXME

commitUpdates Internal Method

```
commitUpdates(newVersion: Long, map: MapType, output: DataOutputStream): Unit
```

commitUpdates ...FIXME

Note

commitUpdates is used exclusively when HDFSBackedStateStore is requested to commit state changes.

metrics Method

```
metrics: StateStoreMetrics
```

Note

metrics is part of the [StateStore Contract](#) to get the [StateStoreMetrics](#).

metrics ...FIXME

StateStoreProvider Contract

`StateStoreProvider` is the abstraction of state store providers that manage state data in stateful streaming queries.

Note	<code>StateStoreProvider</code> helper object uses <code>spark.sql.streaming.stateStore.providerClass</code> internal configuration property for the name of the class of the <code>StateStoreProvider</code> implementation.
Note	<code>HDFSBackedStateStoreProvider</code> is the only available implementation of the <code>StateStoreProvider Contract</code> in Spark Structured Streaming.

Table 1. StateStoreProvider Contract

Method	Description
<code>close</code>	<pre>close(): Unit</pre> <p>Closes the state store provider Used exclusively when <code>statestore</code> helper object is requested to unload a state store provider</p>
<code>doMaintenance</code>	<pre>doMaintenance(): Unit = {}</pre> <p>Does maintenance if needed Used exclusively when <code>statestore</code> helper object is requested to perform maintenance of registered state store providers</p>
<code>getStore</code>	<pre>getStore(version: Long): StateStore</pre> <p>Returns the <code>StateStore</code> for the specified version Used exclusively when <code>statestore</code> helper object is requested to get the StateStore for the given ID and version</p>
<code>init</code>	<pre>init(statestoreId: StatestoreId, keySchema: StructType, valueSchema: StructType, keyIndexOrdinal: Option[Int], storeConfs: StateStoreConf, hadoopConf: Configuration): Unit</pre> <p>Initializes the state store provider</p>

	Used exclusively when <code>stateStoreProvider</code> helper object is requested to create and initialize the StateStoreProvider for a given <code>StateStoreId</code> (when <code>StateStore</code> helper object is requested to retrieve a StateStore by ID and version)
<code>stateStoreId</code>	<p><code>stateStoreId: StateStoreId</code></p> <p><code>StateStoreId</code> associated with the provider (at initialization)</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>HDFSBackedStateStore</code> is requested for the unique id • <code>HDFSBackedStateStoreProvider</code> is created and requested for the textual representation
<code>supportedCustomMetrics</code>	<p><code>supportedCustomMetrics: Seq[StateStoreCustomMetric]</code></p> <p><code>StateStoreCustomMetrics</code> of the state store provider</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>StateStoreWriter</code> stateful physical operators are requested for the stateStoreCustomMetrics (when requested for the metrics and getProgress) • <code>HDFSBackedStateStore</code> is requested for the metrics

Lifecycle of StateStoreProvider

The lifecycle of a `StateStoreProvider` starts when `StateStore` helper object (on a Spark executor) is requested for the [StateStore by given provider ID and version](#).

Note	<code>HDFSBackedStateStoreProvider</code> is the only available implementation of the StateStoreProvider Contract in Spark Structured Streaming.
Note	Since <code>StateStore</code> and <code>StateStoreProvider</code> helper objects are Scala objects that gives that there can only be one instance of <code>StateStore</code> and <code>StateStoreProvider</code> on a JVM. That in turn means that there will be only one instance of each per JVM which is exactly the JVM of a Spark executor.

`StateStore` helper object requests `StateStoreProvider` helper object to [createAndInit](#) that creates the `StateStoreProvider` implementation (given `spark.sql.streaming.stateStore.providerClass` internal configuration property) and requests it to [initialize](#).

The initialized `stateStoreProvider` is cached in `loadedProviders` internal lookup table (for a `StateStoreId`) for later lookups.

`StateStoreProvider` helper object then requests the `stateStoreProvider` to `getStore` for the version.

An instance of `StateStoreProvider` is requested to `do its own maintenance` or `close` (when a corresponding `StateStore` is inactive) in `MaintenanceTask` daemon thread that runs periodically every `spark.sql.streaming.stateStore.maintenanceInterval` configuration property (default: `60s`).

Creating and Initializing StateStoreProvider — `createAndInit` Factory Method

```
createAndInit(  
    stateStoreId: StateStoreId,  
    keySchema: StructType,  
    valueSchema: StructType,  
    indexOrdinal: Option[Int],  
    storeConf: StateStoreConf,  
    hadoopConf: Configuration): StateStoreProvider
```

`createAndInit` creates a new `StateStoreProvider` (per `spark.sql.streaming.stateStore.providerClass` internal configuration property).

`createAndInit` requests the `StateStoreProvider` to `initialize`.

Note

`createAndInit` is used exclusively when `stateStore` helper object is requested for the `StateStore` by given provider ID and version.

StateStoreProviderId — Unique Identifier of State Store Provider

`StateStoreProviderId` is a unique identifier of a [state store provider](#) with the following properties:

- [StateStoreId](#)
- Run ID of a streaming query ([java.util.UUID](#))

In other words, `StateStoreProviderId` is a [StateStoreId](#) with the [run ID](#) that is different every restart.

`StateStoreProviderId` is used by the following execution components:

- `StateStoreCoordinator` to track the [executors of state store providers](#) (on the driver)
- `StateStore` object to manage [state store providers](#) (on executors)

`StateStoreProviderId` is [created](#) (directly or using [apply](#) factory method) when:

- `StateStoreRDD` is requested for the [placement preferences of a partition](#) and to [compute a partition](#)
- `StateStoreAwareZipPartitionsRDD` is requested for the [preferred locations of a partition](#)
- `StateStoreHandler` is requested to [look up a state store](#)

Creating StateStoreProviderId — `apply` Factory Method

```
apply(
  stateInfo: StatefulOperatorStateInfo,
  partitionIndex: Int,
  storeName: String): StateStoreProviderId
```

`apply` simply creates a [new StateStoreProviderId](#) for the [StatefulOperatorStateInfo](#), the partition and the store name.

Internally, `apply` requests the `StatefulOperatorStateInfo` for the [checkpoint directory](#) (aka `checkpointLocation`) and the [stateful operator ID](#) and creates a new [StateStoreId](#) (with the `partitionIndex` and `storeName`).

In the end, `apply` requests the `StatefulOperatorStateInfo` for the [run ID of a streaming query](#) and creates a [new StateStoreProviderId](#) (together with the run ID).

	<p><code>apply</code> is used when:</p> <ul style="list-style-type: none">• <code>StateStoreAwareZipPartitionsRDD</code> is requested for the preferred locations of a partition• <code>StateStoreHandler</code> is requested to look up a state store
Note	

HDFSBackedStateStoreProvider — Default StateStoreProvider

`HDFSBackedStateStoreProvider` is the default `StateStoreProvider` (as specified by the `spark.sql.streaming.stateStore.providerClass` internal configuration property).

`HDFSBackedStateStoreProvider` is `created` and immediately requested to `initialize` when `StateStoreProvider` helper object is requested to `create and initialize a StateStoreProvider`.

`HDFSBackedStateStoreProvider` takes no arguments to be created.

`HDFSBackedStateStoreProvider` uses the **state checkpoint base directory** (that is the `storeCheckpointLocation` of the `StateStoreId`) for `delta` and `snapshot` state files. The checkpoint directory is created when `HDFSBackedStateStoreProvider` is requested to `initialize`.

Tip Enable ALL logging level for `org.apache.spark.sql.execution.streaming.state.HDFSBackedStateStoreProvider` logger happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.execution.streaming.state.HDFSBackedStateStoreProvider=ALL
```

Refer to [Logging](#).

StateStoreId — Unique Identifier of State Store

As a `StateStoreProvider`, `HDFSBackedStateStoreProvider` is associated with a `StateStoreId` (which is a unique identifier of the `state store` for an operator and a partition).

`HDFSBackedStateStoreProvider` is given the `StateStoreId` at `initialization` (as requested by the `StateStoreProvider` contract).

The `StateStoreId` is then used for the following:

- `HDFSBackedStateStore` is requested for the `id`
- `HDFSBackedStateStoreProvider` is requested for the `textual representation` and the `state checkpoint base directory`

Textual Representation — `toString` Method

```
toString(): String
```

Note

`toString` is part of the [java.lang.Object](#) contract for the string representation of the object.

`HDFSBackedStateStoreProvider` uses the [StateStoreId](#) and the state checkpoint base [directory](#) for the textual representation:

```
HDFSStateStoreProvider[id = (op=[operatorId],part=[partitionId]),dir = [baseDir]]
```

Retrieving State Store by Version — `getStore` Method

```
getStore(version: Long): StateStore
```

Note

`getStore` is part of the [StateStoreProvider Contract](#) to get the [StateStore](#) for a given version.

`getStore` ...FIXME

`deltaFile` Internal Method

```
deltaFile(version: Long): Path
```

`deltaFile` simply returns the Hadoop [Path](#) of the `[version].delta` file in the state checkpoint base [directory](#).

Note

`deltaFile` is used when:

- `HDFSBackedStateStore` is created (and creates the [final delta file](#))
- `HDFSBackedStateStoreProvider` is requested to [updateFromDeltaFile](#)

Fetching All Delta And Snapshot Files — `fetchFiles` Internal Method

```
fetchFiles(): Seq[StoreFile]
```

`fetchFiles` ...FIXME

Note

`fetchFiles` is used when `HDFSBackedStateStoreProvider` is requested to `latestIterator`, `doSnapshot` and `cleanup`.

Initializing StateStoreProvider — `init` Method

```
init(  
    statestoreId: StatestoreId,  
    keySchema: StructType,  
    valueSchema: StructType,  
    indexOrdinal: Option[Int],  
    storeConf: StatestoreConf,  
    hadoopConf: Configuration): Unit
```

Note

`init` is part of the [StateStoreProvider Contract](#) to initialize itself.

`init` assigns the values of the input arguments to `statestoreId`, `keySchema`, `valueSchema`, `storeConf`, and `hadoopConf`.

`init` uses the `StatestoreConf` to requests for the `spark.sql.streaming.maxBatchesToRetainInMemory` configuration property (that is then the `numberOfVersionsToRetainInMemory`).

In the end, `init` requests the `CheckpointFileManager` to [create](#) the `baseDir` directory (with parent directories).

latestIterator Internal Method

```
latestIterator(): Iterator[UnsafeRowPair]
```

`latestIterator` ...FIXME

Note

`latestIterator` seems to be used exclusively in tests.

doSnapshot Internal Method

```
doSnapshot(): Unit
```

`doSnapshot` ...FIXME

Note

`doSnapshot` is used when...FIXME

Cleaning Up — `cleanup` Internal Method

```
cleanup(): Unit
```

`cleanup` ...FIXME

Note

`cleanup` is used exclusively when [doMaintenance](#).

Doing Maintenance — `doMaintenance` Method

```
doMaintenance(): Unit
```

Note

`doMaintenance` is part of the [StateStoreProvider Contract](#) to do maintenance if needed.

`doMaintenance` ...FIXME

Closing State Store Provider — `close` Method

```
close(): Unit
```

Note

`close` is part of the [StateStoreProvider Contract](#) to close the state store provider.

`close` ...FIXME

`putStateIntoStateCacheMap` Internal Method

```
putStateIntoStateCacheMap(  
    newVersion: Long,  
    map: ConcurrentHashMap[UnsafeRow, UnsafeRow]): Unit
```

`putStateIntoStateCacheMap` ...FIXME

Note

`putStateIntoStateCacheMap` is used when `HDFSBackedStateStoreProvider` is requested to [commitUpdates](#) and [loadMap](#).

`commitUpdates` Internal Method

```
commitUpdates(
  newVersion: Long,
  map: ConcurrentHashMap[UnsafeRow, UnsafeRow],
  output: DataOutputStream): Unit
```

`commitUpdates ...FIXME`

Note

`commitUpdates` is used exclusively when `HDFSBackedStateStore` is requested to [commit state changes](#).

loadMap Internal Method

```
loadMap(version: Long): ConcurrentHashMap[UnsafeRow, UnsafeRow]
```

`loadMap ...FIXME`

Note

`loadMap` is used when `HDFSBackedStateStoreProvider` is requested to [retrieve the state store for a specified version](#) and [latestIterator](#).

writeSnapshotFile Internal Method

```
writeSnapshotFile(
  version: Long,
  map: MapType): Unit
```

`writeSnapshotFile ...FIXME`

Note

`writeSnapshotFile` is used when...FIXME

updateFromDeltaFile Internal Method

```
updateFromDeltaFile(
  version: Long,
  map: MapType): Unit
```

`updateFromDeltaFile ...FIXME`

Note

`updateFromDeltaFile` is used exclusively when `HDFSBackedStateStoreProvider` is requested to [loadMap](#).

readSnapshotFile Internal Method

```
readSnapshotFile(  
    version: Long): Option[MapType]
```

readSnapshotFile ...FIXME

Note

readSnapshotFile is used...FIXME

Internal Properties

Name	Description
fm	<p><code>CheckpointFileManager</code></p> <p><code>loadedMaps</code>: <code>TreeMap[Long, ConcurrentHashMap[UnsafeObject, MapType]]</code></p> <p><code>java.util.TreeMap</code> of FIXME sorted according to the natural ordering of the keys</p> <p>The current size estimation of <code>loadedMaps</code> is the metric in the metrics.</p> <p>A new entry (a version and the associated map) is added when <code>HDFSBackedStateStoreProvider</code> is requested to putStateIntoStateCacheMap.</p> <p>Used when...FIXME</p>
numberOfVersionsToRetainInMemory	<p><code>numberOfVersionsToRetainInMemory</code>: <code>Int</code></p> <p><code>spark.sql.streaming.maxBatchesToRetainInMemory</code> is the property that sets the upper limit on the number of entries in <code>loadedMaps</code> internal registry.</p> <p><code>numberOfVersionsToRetainInMemory</code> is used when <code>HDFSBackedStateStoreProvider</code> is requested to putStateIntoStateCacheMap.</p>

StateStoreCoordinator — Tracking Locations of StateStores for StateStoreRDD

`StateStoreCoordinator` keeps track of `StateStores` loaded in Spark executors (across the nodes in a Spark cluster).

The main purpose of `StateStoreCoordinator` is for `StateStoreRDD` to [get the location preferences for partitions](#) (based on the location of the stores).

`StateStoreCoordinator` uses instances internal registry of `StateStoreProviders` by their ids and `ExecutorCacheTaskLocations`.

`StateStoreCoordinator` is a `ThreadSafeRpcEndpoint` RPC endpoint that manipulates instances registry through [RPC messages](#).

Table 1. StateStoreCoordinator RPC Endpoint's Messages and Message Handler

Message	Message Handler
DeactivateInstances	<p>Removes <code>StateStoreProviderIds</code> (from <code>instances</code>) with <code>queryRunners</code>.</p> <p>You should see the following DEBUG message in the logs:</p> <pre>Deactivating instances related to checkpoint location [runId]</pre>
GetLocation	<p>Gives the location of <code>StateStoreProviderId</code> (from <code>instances</code>) with <code>host</code>.</p> <p>You should see the following DEBUG message in the logs:</p> <pre>Got location of the state store [id]: [executorId]</pre>
ReportActiveInstance	<p>Registers <code>StateStoreProviderId</code> that is active on an executor (given by <code>host</code>).</p> <p>You should see the following DEBUG message in the logs:</p> <pre>Reported state store [id] is active at [executorId]</pre>
StopCoordinator	<p>Stops StateStoreCoordinator RPC Endpoint.</p> <p>You should see the following DEBUG message in the logs:</p> <pre>StateStoreCoordinator stopped</pre>
VerifyIfInstanceActive	<p>Verifies if a given <code>StateStoreProviderId</code> is registered (in <code>instances</code>).</p> <p>You should see the following DEBUG message in the logs:</p> <pre>Verified that state store [id] is active: [response]</pre>
Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging level for <code>org.apache.spark.sql.execution.streaming.state.StateStoreCoordinator</code> to see what inside.</p> <p>Add the following line to <code>conf/log4j.properties</code>:</p> <pre>log4j.logger.org.apache.spark.sql.execution.streaming.state.StateStoreCoordinator=INFO</pre> <p>Refer to Logging.</p>

StateStoreCoordinatorRef — RPC Endpoint Reference to StateStoreCoordinator

`StateStoreCoordinatorRef` is used to (let the tasks on Spark executors to) send [messages](#) to the [StateStoreCoordinator](#) (that lives on the driver).

`StateStoreCoordinatorRef` is given the `RpcEndpointRef` to the [StateStoreCoordinator](#) RPC endpoint when created.

`StateStoreCoordinatorRef` is created through `StateStoreCoordinatorRef` helper object when requested to create one for the [driver](#) (when `StreamingQueryManager` is created) or an [executor](#) (when `Statestore` helper object is requested for the [RPC endpoint reference to StateStoreCoordinator for Executors](#)).

Table 1. StateStoreCoordinatorRef's Methods and Underlying RPC Messages

Method	Description
<code>deactivateInstances</code>	<pre>deactivateInstances(runId: UUID): Unit</pre> <p>Requests the RpcEndpointRef to send a DeactivateInstances synchronous message with the given <code>runId</code> and waits for a <code>true</code> / <code>false</code> response</p> <p>Used exclusively when <code>StreamingQueryManager</code> is requested to handle termination of a streaming query (when <code>StreamExecution</code> is requested to run a streaming query and the query has finished (running streaming batches)).</p>
<code>getLocation</code>	<pre>getLocation(stateStoreProviderId: StateStoreProviderId): Option[String]</pre> <p>Requests the RpcEndpointRef to send a GetLocation synchronous message with the given <code>StateStoreProviderId</code> and waits for the location</p> <p>Used when:</p> <ul style="list-style-type: none"> <code>StateStoreAwareZipPartitionsRDD</code> is requested for the preferred locations of a partition (when <code>StreamingSymmetricHashJoinExec</code> physical operator is requested to execute and generate a recipe for a distributed computation (as an RDD[InternalRow])) <code>StateStoreRDD</code> is requested for preferred locations for a task for a partition

	<pre>reportActiveInstance(stateStoreProviderId: StateStoreProviderId, host: String, executorId: String): Unit</pre>
reportActiveInstance	<p>Requests the RpcEndpointRef to send a ReportActiveInstance one-way asynchronous (fire-and-forget) message with the given StateStoreProviderId, host and executorId</p> <p>Used exclusively when statestore helper object is requested for reportActiveStoreInstance (when StateStore helper object is requested to find the StateStore by StateStoreProviderId)</p>
stop	<pre>stop(): Unit</pre> <p>Requests the RpcEndpointRef to send a StopCoordinator synchronous message</p> <p>Used exclusively for unit testing</p>
verifyIfInstanceActive	<pre>verifyIfInstanceActive(stateStoreProviderId: StateStoreProviderId, executorId: String): Boolean</pre> <p>Requests the RpcEndpointRef to send a VerifyIfInstanceActive synchronous message with the given StateStoreProviderId and executorId, and waits for a true / false response</p> <p>Used exclusively when statestore helper object is requested for verifyIfStoreInstanceActive (when requested to doMaintenance from a running MaintenanceTask daemon thread)</p>

Creating StateStoreCoordinatorRef to StateStoreCoordinator RPC Endpoint for Driver — `forDriver` Factory Method

```
forDriver(env: SparkEnv): StateStoreCoordinatorRef
```

`forDriver` ...FIXME

Note	<code>forDriver</code> is used exclusively when StreamingQueryManager is created .
------	--

Creating StateStoreCoordinatorRef to StateStoreCoordinator RPC Endpoint for Executor — forExecutor Factory Method

```
forExecutor(env: SparkEnv): StateStoreCoordinatorRef
```

```
forExecutor ...FIXME
```

Note

`forExecutor` is used exclusively when `StateStore` helper object is requested for the [RPC endpoint reference to StateStoreCoordinator for Executors](#).

WatermarkSupport Contract — Unary Physical Operators with Streaming Watermark Support

`WatermarkSupport` is the [abstraction](#) of unary physical operators (`UnaryExecNode`) with support for streaming event-time watermark.

Note

Watermark (aka "allowed lateness") is a moving threshold of **event time** and specifies what data to consider for aggregations, i.e. the threshold of late data so the engine can automatically drop incoming late data given event time and clean up old state accordingly.

Read the official documentation of Spark in [Handling Late Data and Watermarking](#).

Table 1. WatermarkSupport's (Lazily-Initialized) Properties

Property	Description					
	<p>Optional Catalyst expression that matches rows older than the event time watermark.</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="padding: 2px;">Note</td> <td style="padding: 2px;">Use withWatermark operator to specify streaming watermark.</td> </tr> </table>		Note	Use withWatermark operator to specify streaming watermark.		
Note	Use withWatermark operator to specify streaming watermark.					
<code>watermarkExpression</code>	<p>When initialized, <code>watermarkExpression</code> finds <code>spark.watermark</code> watermark attribute in the child output's metadata.</p> <p>If found, <code>watermarkExpression</code> creates <code>evictionExpression</code> attribute that is less than or equal <code>eventTimeWatermark</code>.</p> <p>The <code>watermark</code> attribute may be of type <code>StructType</code>. If it is, <code>watermarkExpression</code> uses the first field as the watermark.</p> <p><code>watermarkExpression</code> prints out the following INFO message if <code>spark.watermarkDelayMs</code> watermark attribute is found.</p> <pre style="background-color: #f0f0f0; padding: 5px;">INFO [physicaloperator]Exec: Filtering state store on: [</pre> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="padding: 2px;">Note</td> <td style="padding: 2px;"><code>physicaloperator</code> can be <code>FlatMapGroupsWithStateSaveExec</code>, <code>StateStoreSaveExec</code> or <code>StreamingDeduplicateExec</code>.</td> </tr> </table> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="padding: 2px;">Tip</td> <td style="padding: 2px;">Enable INFO logging level for one of the stateful physical operators to see the INFO message in the logs.</td> </tr> </table>		Note	<code>physicaloperator</code> can be <code>FlatMapGroupsWithStateSaveExec</code> , <code>StateStoreSaveExec</code> or <code>StreamingDeduplicateExec</code> .	Tip	Enable INFO logging level for one of the stateful physical operators to see the INFO message in the logs.
Note	<code>physicaloperator</code> can be <code>FlatMapGroupsWithStateSaveExec</code> , <code>StateStoreSaveExec</code> or <code>StreamingDeduplicateExec</code> .					
Tip	Enable INFO logging level for one of the stateful physical operators to see the INFO message in the logs.					
<code>watermarkPredicateForData</code>	Optional <code>Predicate</code> that uses <code>watermarkExpression</code> and the <code>keyExpressions</code> to match rows older than the event-time watermark.					
<code>watermarkPredicateForKeys</code>	Optional <code>Predicate</code> that uses <code>keyExpressions</code> to match rows older than the event time watermark.					

WatermarkSupport Contract

```
package org.apache.spark.sql.execution.streaming

trait WatermarkSupport extends UnaryExecNode {
    // only required methods that have no implementation
    def eventTimeWatermark: Option[Long]
    def keyExpressions: Seq[Attribute]
}
```

Table 2. WatermarkSupport Contract

Method	Description
eventTimeWatermark	Used mainly in watermarkExpression to create a <code>LessThanOrEqual</code> Catalyst binary expression that matches rows older than the watermark.
keyExpressions	Grouping keys (in FlatMapGroupsWithStateExec), duplicate keys (in StreamingDeduplicateExec) or key attributes (in StateStoreSaveExec) with at most one that may have <code>spark.watermarkDelayMs</code> watermark attribute in metadata
	Used in watermarkPredicateForKeys to create a <code>Predicate</code> to match rows older than the event time watermark. Used also when StateStoreSaveExec and StreamingDeduplicateExec physical operators are executed.

Removing Keys From StateStore Older Than Watermark

— `removeKeysOlderThanWatermark` Method

```
removeKeysOlderThanWatermark(store: StateStore): Unit
```

`removeKeysOlderThanWatermark` requests the input `store` for all rows.

`removeKeysOlderThanWatermark` then uses [watermarkPredicateForKeys](#) to remove matching rows from the store.

Note

`removeKeysOlderThanWatermark` is used exclusively when `StreamingDeduplicateExec` physical operator is requested to execute and generate a recipe for a distributed computation (as an `RDD[InternalRow]`).

removeKeysOlderThanWatermark Method

```
removeKeysOlderThanWatermark(
  storeManager: StreamingAggregationStateManager,
  store: StateStore): Unit
```

`removeKeysOlderThanWatermark` ...FIXME

Note

`removeKeysOlderThanWatermark` is used exclusively when `StateStoreSaveExec` physical operator is requested to execute and generate a recipe for a distributed computation (as an `RDD[InternalRow]`).

StatefulOperator Contract — Physical Operators That Read or Write to StateStore

`StatefulOperator` is the base of physical operators that read or write state (described by `stateInfo`).

Table 1. StatefulOperator Contract

Method	Description
<code>stateInfo</code>	<code>stateInfo: Option[StatefulOperatorStateInfo]</code> The <code>StatefulOperatorStateInfo</code> of the physical operator

Table 2. StatefulOperators (Direct Implementations)

StatefulOperator	Description
<code>StateStoreReader</code>	
<code>StateStoreWriter</code>	Physical operator that writes to a state store and collects the write metrics for execution progress reporting

StateStoreReader

StateStoreReader is...FIXME

StateStoreWriter Contract — Stateful Physical Operators That Write to StateStore

`StateStoreWriter` is the extension of the [StatefulOperator Contract](#) for physical operators that write to a state store and collect the [write metrics](#) for [execution progress reporting](#).

Table 1. StateStoreWriter's Performance Metrics

Key	Name (in web UI)	Description
<code>numOutputRows</code>	number of output rows	
<code>numTotalStateRows</code>	number of total state rows	
<code>numUpdatedStateRows</code>	number of updated state rows	
<code>allUpdatesTimeMs</code>	total time to update rows	
<code>allRemovalsTimeMs</code>	total time to remove rows	
<code>commitTimeMs</code>	time to commit changes	
<code>stateMemory</code>	memory used by state	

Table 2. StateStoreWriters

StateStoreWriter	Description
FlatMapGroupsWithStateExec	
StateStoreSaveExec	
StreamingDeduplicateExec	
StreamingGlobalLimitExec	
StreamingSymmetricHashJoinExec	

Setting StateStore-Specific Metrics for Stateful Physical Operator — `setStoreMetrics` Method

```
setStoreMetrics(store: StateStore): Unit
```

`setStoreMetrics` requests the specified `StateStore` for the `metrics` and records the following metrics of a physical operator:

- `numTotalStateRows` as the `number of keys`
- `stateMemory` as the `memory used (in bytes)`

`setStoreMetrics` records the `custom metrics`.

Note

`setStoreMetrics` is used when the following physical operators are executed:

- `FlatMapGroupsWithStateExec`
- `StateStoreSaveExec`
- `StreamingDeduplicateExec`
- `StreamingGlobalLimitExec`

getProgress Method

`getProgress(): StateOperatorProgress`

`getProgress` ...FIXME

Note

`getProgress` is used exclusively when `ProgressReporter` is requested to `extractStateOperatorMetrics` (when `MicroBatchExecution` is requested to `run` the activated streaming query).

Checking Out Whether Last Batch Execution Requires Another Non-Data Batch or Not — `shouldRunAnotherBatch` Method

`shouldRunAnotherBatch(newMetadata: OffsetSeqMetadata): Boolean`

`shouldRunAnotherBatch` is negative (`false`) by default (to indicate that another non-data batch is not required given the `OffsetSeqMetadata` with the event-time watermark and the batch timestamp).

Note

`shouldRunAnotherBatch` is used exclusively when `IncrementalExecution` is requested to `check out whether the last batch execution requires another batch` (when `MicroBatchExecution` is requested to `run the activated streaming query`).

stateStoreCustomMetrics Internal Method

```
stateStoreCustomMetrics: Map[String, SQLMetric]
```

```
stateStoreCustomMetrics ...FIXME
```

Note	<code>stateStoreCustomMetrics</code> is used when <code>StateStoreWriter</code> is requested for the <code>metrics</code> and <code>getProgress</code> .
------	--

timeTakenMs Method

```
timeTakenMs(body: => Unit): Long
```

```
timeTakenMs ...FIXME
```

Note	<code>timeTakenMs</code> is used when...FIXME
------	---

StatefulOperatorStateInfo

`StatefulOperatorStateInfo` identifies the state store for a given operator:

- Checkpoint directory (aka *checkpointLocation*)
- Run ID of a streaming query (`queryRunId`)
- Stateful operator ID (`operatorId`)
- `storeVersion`
- Number of partitions

`StatefulOperatorStateInfo` is [created](#) exclusively when `IncrementalExecution` is requested for [nextStatefulOperationStateInfo](#).

When requested for a textual representation, `StatefulOperatorStateInfo` returns the following:

```
state info [ checkpoint = [checkpointLocation], runId = [queryRunId], opId = [operatorId], ver = [storeVersion], numPartitions = [numPartitions] ]
```

StateStoreMetrics

`StateStoreMetrics` holds the metrics of a [state store](#):

- Total number of keys
- Memory used (in bytes)
- [StateStoreCustomMetrics](#) with their current values (`Map[StateStoreCustomMetric, Long]`)

`StateStoreMetrics` is used (and [created](#)) when:

- `StateStore` is requested for the [metrics](#)
- `StateStoreHandler` is requested for the [metrics](#)
- `SymmetricHashJoinStateManager` is requested for the [metrics](#)

StateStoreCustomMetric Contract

`StateStoreCustomMetric` is the [abstraction of metrics](#) that a state store may wish to expose (as [StateStoreMetrics](#) or [supportedCustomMetrics](#)).

`StateStoreCustomMetric` is used when:

- `StateStoreProvider` is requested for the [custom metrics](#)
- `StateStoreMetrics` is [created](#)

Table 1. StateStoreCustomMetric Contract

Method	Description
<code>desc</code>	<p><code>desc: String</code></p> <p>Description of the custom metrics</p>
<code>name</code>	<p><code>name: String</code></p> <p>Name of the custom metrics</p>

Table 2. StateStoreCustomMetrics

StateStoreCustomMetric	Description
<code>StateStoreCustomSizeMetric</code>	
<code>StateStoreCustomSumMetric</code>	
<code>StateStoreCustomTimingMetric</code>	

StateStoreUpdater

StateStoreUpdater is...FIXME

updateStateForKeysWithData Method

Caution	FIXME
---------	-------

updateStateForTimedOutKeys Method

Caution	FIXME
---------	-------

EventTimeStatsAccum Accumulator — Event-Time Column Statistics for EventTimeWatermarkExec Physical Operator

`EventTimeStatsAccum` is a Spark accumulator that is used for the [statistics of the event-time column](#) (that `EventTimeWatermarkExec` physical operator uses for event-time watermark):

- Maximum value
- Minimum value
- Average value
- Number of updates (count)

`EventTimeStatsAccum` is [created](#) and registered exclusively for `EventTimeWatermarkExec` physical operator.

Note

When `EventTimeWatermarkExec` physical operator is requested to [execute and generate a recipe for a distributed computation \(as a `RDD\[InternalRow\]`\)](#), every task simply [adds](#) the values of the event-time watermark column to the `EventTimeStatsAccum` accumulator.

As per design of Spark accumulators in Apache Spark, accumulator updates are automatically sent out (*propagated*) from tasks to the driver every heartbeat and then they are accumulated together.

Tip

Read up on [Accumulators](#) in [The Internals of Apache Spark](#) book.

`EventTimeStatsAccum` takes a single `EventTimeStats` to be created (default: `zero`).

Accumulating Value — `add` Method

```
add(v: Long): Unit
```

Note

`add` is part of the `AccumulatorV2` Contract to add (*accumulate*) a given value.

`add` simply requests the `EventTimeStats` to [add](#) the given `v` value.

Note

`add` is used exclusively when `EventTimeWatermarkExec` physical operator is requested to [execute and generate a recipe for a distributed computation \(as a `RDD\[InternalRow\]`\)](#).

EventTimeStats

`EventTimeStats` is a Scala case class for the event-time column statistics.

`EventTimeStats` defines a special value `zero` with the following values:

- `Long.MinValue` for the `max`
- `Long.MaxValue` for the `min`
- `0.0` for the `avg`
- `0L` for the `count`

EventTimeStats.add Method

```
add(eventTime: Long): Unit
```

`add` simply updates the event-time column statistics per given `eventTime`.

Note	<code>add</code> is used exclusively when <code>EventTimeStatsAccum</code> is requested to accumulate the value of an event-time column.
------	--

EventTimeStats.merge Method

```
merge(that: EventTimeStats): Unit
```

`merge` ...FIXME

Note	<code>merge</code> is used when...FIXME
------	---

StateStoreConf

`StateStoreConf` is...FIXME

Table 1. StateStoreConf's Properties

Name	Configuration Property
<code>maxVersionsToRetainInMemory</code>	<code>spark.sql.streaming.maxBatchesToRetainInMemory</code>
<code>minVersionsToRetain</code>	<code>spark.sql.streaming.minBatchesToRetain</code>
<code>providerClass</code>	<code>spark.sql.streaming.stateStore.providerClass</code> Used exclusively when <code>StateStoreProvider</code> helper object is requested to create and initialize the StateStoreProvider .

DataStreamReader — Loading Data from Streaming Source

`DataStreamReader` is the [interface](#) to describe how data is [loaded](#) to a streaming `Dataset` from a [streaming source](#).

Table 1. DataStreamReader's Methods

Method	Description
<code>csv</code>	<pre>csv(path: String): DataFrame</pre> <p>Sets <code>csv</code> as the format of the data source</p>
<code>format</code>	<pre>format(source: String): DataStreamReader</pre> <p>Specifies the format of the data source The format is used internally as the name (<i>alias</i>) of the streaming source to use to load the data</p>
<code>json</code>	<pre>json(path: String): DataFrame</pre> <p>Sets <code>json</code> as the format of the data source</p>
<code>load</code>	<pre>load(): DataFrame load(path: String): DataFrame (1)</pre> <p>1. Explicit <code>path</code> (that could also be specified as an option) "Loads" data as a streaming <code>DataFrame</code> (that is internally a logical plan with a StreamingRelationV2 or StreamingRelation leaf logical operators)</p>
<code>option</code>	<pre>option(key: String, value: Boolean): DataStreamReader option(key: String, value: Double): DataStreamReader option(key: String, value: Long): DataStreamReader option(key: String, value: String): DataStreamReader</pre> <p>Sets a loading option</p>
	<pre>options(options: Map[String, String]): DataStreamReader</pre>

	options	Specifies the configuration options of a data source		
		<table border="1"> <tr> <td>Note</td><td>You could use <code>option</code> method if you prefer specifying the options one by one or there is only one in use.</td></tr> </table>	Note	You could use <code>option</code> method if you prefer specifying the options one by one or there is only one in use.
Note	You could use <code>option</code> method if you prefer specifying the options one by one or there is only one in use.			
	orc	<pre>orc(path: String): DataFrame</pre> <p>Sets <code>orc</code> as the <code>format</code> of the data source</p>		
	parquet	<pre>parquet(path: String): DataFrame</pre> <p>Sets <code>parquet</code> as the <code>format</code> of the data source</p>		
	schema	<pre>schema(schema: StructType): DataStreamReader schema(schemaString: String): DataStreamReader (1)</pre> <p>1. Uses a DDL-formatted table schema</p> <p>Specifies the <code>user-defined schema</code> of the streaming data source (as a <code>StructType</code> or DDL-formatted table schema, e.g. <code>a INT, b STRING</code>)</p>		
	text	<pre>text(path: String): DataFrame</pre> <p>Sets <code>text</code> as the <code>format</code> of the data source</p>		
	textFile	<pre>textFile(path: String): Dataset[String]</pre>		



Figure 1. DataStreamReader and The Others

`DataStreamReader` is used for a Spark developer to describe how Spark Structured Streaming loads datasets from a streaming source (that `in the end` creates a logical plan for a streaming query).

Note	<code>DataStreamReader</code> is the Spark developer-friendly API to create a <code>StreamingRelation</code> logical operator (that represents a <code>streaming source</code> in a logical plan).
-------------	--

You can access `DataStreamReader` using `SparkSession.readStream` method.

```
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...

val streamReader = spark.readStream
```

`DataStreamReader` supports many [source formats](#) natively and offers the [interface to define custom formats](#):

- [json](#)
- [csv](#)
- [parquet](#)
- [text](#)

Note	<code>DataStreamReader</code> assumes parquet file format by default that you can change using <code>spark.sql.sources.default</code> property.
------	---

Note	<code>hive</code> source format is not supported.
------	---

After you have described the **streaming pipeline** to read datasets from an external streaming data source, you eventually trigger the loading using format-agnostic [load](#) or format-specific (e.g. [json](#), [csv](#)) operators.

Table 2. DataStreamReader's Internal Properties (in alphabetical order)

Name	Initial Value	Description
<code>source</code>	<code>spark.sql.sources.default</code> property	Source format of datasets in a streaming data source
<code>userSpecifiedSchema</code>	(empty)	Optional user-defined schema
<code>extraOptions</code>	(empty)	Collection of key-value configuration options

Specifying Loading Options — `option` Method

```
option(key: String, value: String): DataStreamReader
option(key: String, value: Boolean): DataStreamReader
option(key: String, value: Long): DataStreamReader
option(key: String, value: Double): DataStreamReader
```

`option` family of methods specifies additional options to a streaming data source.

There is support for values of `String`, `Boolean`, `Long`, and `Double` types for user convenience, and internally are converted to `String` type.

Internally, `option` sets [extraOptions](#) internal property.

Note

You can also set options in bulk using [options](#) method. You have to do the type conversion yourself, though.

Creating Streaming Dataset (to Represent Loading Data From Streaming Source) — `load` Method

```
load(): DataFrame
load(path: String): DataFrame (1)
```

1. Specifies `path` option before passing the call to parameterless `load()`

```
load ...FIXME
```

Built-in Formats

```
json(path: String): DataFrame
csv(path: String): DataFrame
parquet(path: String): DataFrame
text(path: String): DataFrame
textFile(path: String): Dataset[String] (1)
```

1. Returns `Dataset[String]` not `DataFrame`

`DataStreamReader` can load streaming datasets from data sources of the following [formats](#):

- `json`
- `csv`
- `parquet`
- `text`

The methods simply pass calls to [format](#) followed by `load(path)`.

DataStreamWriter — Writing Datasets To Streaming Sink

`DataStreamWriter` is the [interface](#) to describe when and what rows of a streaming query are sent out to the [streaming sink](#).

Table 1. DataStreamWriter's Methods

Method	Description
<code>foreach</code>	<pre>foreach(writer: ForeachWriter[T]): DataStreamWriter[T]</pre> <p>Sets ForeachWriter in the full control of streaming writes</p>
<code>foreachBatch</code>	<pre>foreachBatch(function: (Dataset[T], Long) => Unit): DataStreamWriter[</pre> <p>(New in 2.4.0) Sets the source to <code>foreachBatch</code> and the <code>foreachBatchWriter</code> to the given function.</p> <p>As per SPARK-24565 Add API for in Structured Streaming for exposing output rows of each microbatch as a DataFrame, the purpose of the method is to expose the micro-batch output as a dataframe for the following:</p> <ul style="list-style-type: none"> • Pass the output rows of each batch to a library that is designed for the batch jobs only • Reuse batch data sources for output whose streaming version does not exist • Multi-writes where the output rows are written to multiple outputs by writing twice for every batch
<code>format</code>	<pre>format(source: String): DataStreamWriter[T]</pre> <p>Specifies the format of the data sink (aka <i>output format</i>)</p> <p>The format is used internally as the name (<i>alias</i>) of the streaming sink to write the data to</p>
<code>option</code>	<pre>option(key: String, value: Boolean): DataStreamWriter[T] option(key: String, value: Double): DataStreamWriter[T] option(key: String, value: Long): DataStreamWriter[T] option(key: String, value: String): DataStreamWriter[T]</pre>
	<pre>options(options: Map[String, String]): DataStreamWriter[T]</pre>

options	Specifies the configuration options of a data sink
	<p>Note You could use option method if you prefer specifying the option one by one or there is only one in use.</p>
outputMode	<pre>outputMode(outputMode: OutputMode): DataStreamWriter[T] outputMode(outputMode: String): DataStreamWriter[T]</pre> <p>Specifies the output mode</p>
partitionBy	<pre>partitionBy(colNames: String*): DataStreamWriter[T]</pre>
queryName	<pre>queryName(queryName: String): DataStreamWriter[T]</pre> <p>Assigns the name of a query</p>
start	<pre>start(): StreamingQuery start(path: String): StreamingQuery (1)</pre> <ol style="list-style-type: none"> Explicit <code>path</code> (that could also be specified as an option) <p>Creates and immediately starts a StreamingQuery</p>
trigger	<pre>trigger(trigger: Trigger): DataStreamWriter[T]</pre> <p>Sets the Trigger for how often a streaming query should be executed and the result saved.</p>

	A streaming query is a Dataset with a streaming logical plan .
Note	<pre>import org.apache.spark.sql.streaming.Trigger import scala.concurrent.duration._ import org.apache.spark.sql.DataFrame val rates: DataFrame = spark. readStream. format("rate"). load scala> rates.isStreaming res1: Boolean = true scala> rates.queryExecution.logical.isStreaming res2: Boolean = true</pre>

`DataStreamWriter` is available using `writeStream` method of a streaming `Dataset`.

```

import org.apache.spark.sql.streaming.DataStreamWriter
import org.apache.spark.sql.Row

val streamingQuery: Dataset[Long] = ...

scala> streamingQuery.isStreaming
res0: Boolean = true

val writer: DataStreamWriter[Row] = streamingQuery.writeStream

```

Like the batch `DataStreamWriter`, `DataStreamWriter` has a direct support for many [file formats](#) and an extension point to plug in new formats.

```

// see above for writer definition

// Save dataset in JSON format
writer.format("json")

```

In the end, you start the actual continuous writing of the result of executing a `dataset` to a sink using `start` operator.

```
writer.save
```

Beside the above operators, there are the following to work with a `dataset` as a whole.

Note

hive [is not supported](#) for streaming writing (and leads to a `AnalysisException`).

Note

`DataStreamWriter` is responsible for writing in a batch fashion.

Table 2. DataStreamWriter's Internal Properties (e.g. Registries, Counters and Flags)

Name	Initial Value	Description
extraOptions		
foreachBatchWriter	null	foreachBatchWriter: (Dataset[T], Long) The function that is used as the batch writer for ForeachBatchSink for foreachBatch
foreachWriter		
partitioningColumns		
source		
outputMode	OutputMode.Append	OutputMode of the streaming sink Set using outputMode method.
trigger		

Specifying Write Option — `option` Method

```
option(key: String, value: String): DataStreamWriter[T]
option(key: String, value: Boolean): DataStreamWriter[T]
option(key: String, value: Long): DataStreamWriter[T]
option(key: String, value: Double): DataStreamWriter[T]
```

Internally, `option` adds the `key` and `value` to [extraOptions](#) internal option registry.

Specifying Output Mode — `outputMode` Method

```
outputMode(outputMode: String): DataStreamWriter[T]
outputMode(outputMode: OutputMode): DataStreamWriter[T]
```

`outputMode` specifies the [output mode](#) of a streaming query, i.e. what data is sent out to a [streaming sink](#) when there is new data available in [streaming data sources](#).

Note	When not defined explicitly, <code>outputMode</code> defaults to Append output mode.
------	--

`outputMode` can be specified by name or one of the [OutputMode](#) values.

Setting Query Name — `queryName` method

```
queryName(queryName: String): DataStreamWriter[T]
```

`queryName` sets the name of a [streaming query](#).

Internally, it is just an additional [option](#) with the key `queryName`.

Setting How Often to Execute Streaming Query — `trigger` method

```
trigger(trigger: Trigger): DataStreamWriter[T]
```

`trigger` method sets the time interval of the `trigger` (that executes a batch runner) for a streaming query.

Note	<code>Trigger</code> specifies how often results should be produced by a StreamingQuery . See Trigger .
------	--

The default trigger is [ProcessingTime\(0L\)](#) that runs a streaming query as often as possible.

Tip	Consult Trigger to learn about <code>Trigger</code> and <code>ProcessingTime</code> types.
-----	--

Creating and Starting Execution of Streaming Query — `start` Method

```
start(): StreamingQuery
start(path: String): StreamingQuery (1)
```

1. Sets `path` option to `path` and passes the call on to `start()`

`start` starts a streaming query.

`start` gives a [StreamingQuery](#) to control the execution of the continuous query.

Note	Whether or not you have to specify <code>path</code> option depends on the streaming sink in use.
------	---

Internally, `start` branches off per `source`.

- `memory`
- `foreach`

- other formats

...FIXME

Table 3. start's Options

Option	Description
queryName	Name of active streaming query
checkpointLocation	Directory for checkpointing (and to store query metadata like offsets before and after being processed, the query id , etc.)

`start` reports a `AnalysisException` when `source` is `hive`.

```
val q = spark.  
  readStream.  
  text("server-logs/*").  
  writeStream.  
  format("hive") <-- hive format used as a streaming sink  
scala> q.start  
org.apache.spark.sql.AnalysisException: Hive data source can only be used with tables,  
you can not write files of Hive data source directly.;  
  at org.apache.spark.sql.streaming.DataStreamWriter.start(DataStreamWriter.scala:234)  
    ... 48 elided
```

Note	Define options using option or options methods.
------	---

Making ForeachWriter in Charge of Streaming Writes

— `foreach` method

```
foreach(writer: ForeachWriter[T]): DataStreamWriter[T]
```

`foreach` sets the input [ForeachWriter](#) to be in control of streaming writes.

Internally, `foreach` sets the streaming output [format](#) as `foreach` and `foreachwriter` as the input `writer`.

Note	<code>foreach</code> uses <code>SparkSession</code> to access <code>SparkContext</code> to clean the <code>ForeachWriter</code> .
------	---

Note	<code>foreach</code> reports an <code>IllegalArgumentException</code> when <code>writer</code> is <code>null</code> .
------	---

	<code>foreach writer</code> cannot be <code>null</code>
--	---

OutputMode

Output mode (`OutputMode`) of a streaming query describes what data is written to a [streaming sink](#).

There are three available output modes:

- [Append](#)
- [Complete](#)
- [Update](#)

The output mode is specified on the *writing side* of a streaming query using [DataStreamWriter.outputMode](#) method (by alias or a value of `org.apache.spark.sql.streaming.OutputMode` object).

```
import org.apache.spark.sql.streaming.OutputMode.Update
val inputStream = spark
  .readStream
  .format("rate")
  .load
  .writeStream
  .format("console")
  .outputMode(Update) // <-- update output mode
  .start
```

Append Output Mode

Append (alias: `append`) is the [default output mode](#) that writes "new" rows only.

In [streaming aggregations](#), a "new" row is when the intermediate state becomes final, i.e. when new events for the grouping key can only be considered late which is when watermark moves past the event time of the key.

`Append` output mode requires that a streaming query defines event-time watermark (using [withWatermark](#) operator) on the event time column that is used in aggregation (directly or using [window](#) function).

Required for datasets with `FileFormat` format (to create [FileStreamSink](#))

`Append` is [mandatory](#) when multiple `flatMapGroupsWithState` operators are used in a structured query.

Complete Output Mode

Complete (alias: **complete**) writes all the rows of a Result Table (and corresponds to a traditional batch structured query).

Complete mode does not drop old aggregation state and preserves all data in the Result Table.

Supported only for [streaming aggregations](#) (as asserted by [UnsupportedOperationChecker](#)).

Update Output Mode

Update (alias: **update**) writes only the rows that were updated (every time there are updates).

For queries that are not [streaming aggregations](#), `Update` is equivalent to the [Append](#) output mode.

Trigger — How Frequently to Check Sources For New Data

`Trigger` defines how often a `streaming query` should be executed (*triggered*) and emit a new data (which `StreamExecution` uses to `resolve a TriggerExecutor`).

Table 1. Trigger's Factory Methods

Trigger	Creating Instance
<code>ContinuousTrigger</code>	<code>Trigger Continuous(long intervalMs)</code> <code>Trigger Continuous(long interval, TimeUnit timeUnit)</code> <code>Trigger Continuous(Duration interval)</code> <code>Trigger Continuous(String interval)</code>
<code>OneTimeTrigger</code>	<code>Trigger Once()</code>
<code>ProcessingTime</code>	<code>Trigger ProcessingTime(Duration interval)</code> <code>Trigger ProcessingTime(long intervalMs)</code> <code>Trigger ProcessingTime(long interval, TimeUnit timeUnit)</code> <code>Trigger ProcessingTime(String interval)</code>
	Examples of ProcessingTime
Note	You specify the trigger for a streaming query using <code>DataStreamWriter</code> 's <code>trigger</code> method.

```

import org.apache.spark.sql.streaming.Trigger
val query = spark.
  readStream.
  format("rate").
  load.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.Once). // <-- execute once and stop
  queryName("rate-once").
  start

assert(query.isActive == false)

scala> println(query.lastProgress)
{
  "id" : "2ae4b0a4-434f-4ca7-a523-4e859c07175b",
  "runId" : "24039ce5-906c-4f90-b6e7-bbb3ec38a1f5",
  "name" : "rate-once",
  "timestamp" : "2017-07-04T18:39:35.998Z",
  "numInputRows" : 0,
  "processedRowsPerSecond" : 0.0,
  "durationMs" : {
    "addBatch" : 1365,
    "getBatch" : 29,
    "getOffset" : 0,
    "queryPlanning" : 285,
    "triggerExecution" : 1742,
    "walCommit" : 40
  },
  "stateOperators" : [ ],
  "sources" : [ {
    "description" : "RateSource[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8]"
  ,
    "startOffset" : null,
    "endOffset" : 0,
    "numInputRows" : 0,
    "processedRowsPerSecond" : 0.0
  } ],
  "sink" : {
    "description" : "org.apache.spark.sql.execution.streaming.ConsoleSink@7dbf277"
  }
}

```

Note

Although `Trigger` allows for custom implementations, `StreamExecution` refuses such attempts and reports an `IllegalStateException`.

```

import org.apache.spark.sql.streaming.Trigger
case object MyTrigger extends Trigger
scala> val sq = spark
    .readStream
    .format("rate")
    .load
    .writeStream
    .format("console")
    .trigger(MyTrigger) // <-- use custom trigger
    .queryName("rate-custom-trigger")
    .start
java.lang.IllegalStateException: Unknown type of trigger: MyTrigger
  at org.apache.spark.sql.execution.streaming.MicroBatchExecution.<init>(MicroBatchExecution.scala:60)
  at org.apache.spark.sql.streaming.StreamingQueryManager.createQuery(StreamingQueryManager.scala:275)
  at org.apache.spark.sql.streaming.StreamingQueryManager.startQuery(StreamingQueryManager.scala:316)
  at org.apache.spark.sql.streaming.DataStreamWriter.start(DataStreamWriter.scala:325)
... 57 elided

```

Note

`Trigger` was introduced in [the commit for \[SPARK-14176\]\[SQL\] Add DataFrameWriter.trigger to set the stream batch period.](#)

Examples of ProcessingTime

`ProcessingTime` is a `Trigger` that assumes that milliseconds is the minimum time unit.

You can create an instance of `ProcessingTime` using the following constructors:

- `ProcessingTime(Long)` that accepts non-negative values that represent milliseconds.

```
ProcessingTime(10)
```

- `ProcessingTime(interval: String)` or `ProcessingTime.create(interval: String)` that accept `CalendarInterval` instances with or without leading `interval` string.

```
ProcessingTime("10 milliseconds")
ProcessingTime("interval 10 milliseconds")
```

- `ProcessingTime(Duration)` that accepts `scala.concurrent.duration.Duration` instances.

```
ProcessingTime(10.seconds)
```

- `ProcessingTime.create(interval: Long, unit: TimeUnit)` for `Long` and `java.util.concurrent.TimeUnit` instances.

```
ProcessingTime.create(10, TimeUnit.SECONDS)
```

StreamingQuery Contract

`StreamingQuery` is the [contract](#) for a streaming query that is executed continuously and concurrently (i.e. on a separate thread).

Note `StreamingQuery` is called **continuous query** or **streaming query**.

Note `StreamingQuery` is a Scala trait with the only implementation being [StreamExecution](#) (and less importantly `StreamingQueryWrapper` for serializing a non-serializable `streamExecution`).

`StreamingQuery` can be in two states:

- active (started)
- inactive (stopped)

If inactive, `StreamingQuery` may have transitioned into the state due to an `StreamingQueryException` (that is available under `exception`).

`StreamingQuery` tracks current state of all the sources, i.e. `sourceStatus`, as `sourceStatuses`.

There could only be a single [Sink](#) for a `StreamingQuery` with many [Sources](#).

`StreamingQuery` can be stopped by `stop` or an exception.

Table 1. StreamingQuery Contract

Method	Description
<code>awaitTermination</code>	<pre>awaitTermination(): Unit awaitTermination(timeoutMs: Long): Boolean</pre> <p>Used when...FIXME</p>
<code>exception</code>	<pre>exception: Option[StreamingQueryException]</pre> <p><code>StreamingQueryException</code> if the query has finished due to an exception</p> <p>Used when...FIXME</p>
<code>explain</code>	<pre>explain(): Unit explain(extended: Boolean): Unit</pre>

	Used when...FIXME
<code>id</code>	<pre>id: UUID</pre> <p>The unique identifier of the streaming query (that does not change across restarts unlike <code>runId</code>)</p>
	Used when...FIXME
<code>isActive</code>	<pre>isActive: Boolean</pre> <p>Indicates whether the streaming query is active (<code>true</code>) or not (<code>false</code>)</p>
	Used when...FIXME
<code>lastProgress</code>	<pre>lastProgress: StreamingQueryProgress</pre> <p>The last <code>StreamingQueryProgress</code> of the streaming query</p>
	Used when...FIXME
<code>name</code>	<pre>name: String</pre> <p>The name of the query that is unique across all active queries</p>
	Used when...FIXME
<code>processAllAvailable</code>	<pre>processAllAvailable(): Unit</pre> <p>Waits the streaming query until there are no data available in sources or the query has been terminated.</p>
	Used when...FIXME
<code>recentProgress</code>	<pre>recentProgress: Array[StreamingQueryProgress]</pre> <p>Collection of the recent <code>StreamingQueryProgress</code> updates.</p>
	Used when...FIXME
	<pre>runId: UUID</pre>

runId	The unique identifier of the current execution of the streaming query (that is different every restart unlike id) Used when...FIXME
sparkSession	sparkSession: SparkSession Used when...FIXME
status	status: StreamingQueryStatus StreamingQueryStatus of the streaming query (as StreamExecution has accumulated being a ProgressReporter while running the streaming query) Used when...FIXME
stop	stop(): Unit Stops the streaming query

Streaming Operators — High-Level Declarative Streaming Dataset API

Dataset API comes with a set of [operators](#) that are of particular use in Spark Structured Streaming that together constitute so-called **High-Level Declarative Streaming Dataset API**.

Table 1. Streaming Operators

Operator	Description
<code>dropDuplicates</code>	<pre>dropDuplicates(): Dataset[T] dropDuplicates(colNames: Seq[String]): Dataset[T] dropDuplicates(col1: String, cols: String*): Dataset[T]</pre> <p>Drops duplicate records (given a subset of columns)</p>
<code>explain</code>	<pre>explain(): Unit explain(extended: Boolean): Unit</pre> <p>Explains query plans</p>
<code>groupBy</code>	<pre>groupBy(cols: Column*): RelationalGroupedDataset groupBy(col1: String, cols: String*): RelationalGroupedDataset</pre> <p>Aggregates rows by zero, one or more columns</p>
<code>groupByKey</code>	<pre>groupByKey(func: T => K): KeyValueGroupedDataset[K, T]</pre> <p>Aggregates rows by a typed grouping function (and gives a KeyValueGroupedDataset)</p>
<code>withWatermark</code>	<pre>withWatermark(eventTime: String, delayThreshold: String): Dataset[T]</pre> <p>Defines a streaming (event-time) watermark for late events (on the given <code>eventTime</code> column with a delay threshold)</p>
<code>writeStream</code>	<pre>writeStream: DataStreamWriter[T]</pre> <p>Creates a DataStreamWriter for persisting the result of a streaming query to an external data system</p>

```
val rates = spark
    .readStream
    .format("rate")
    .option("rowsPerSecond", 1)
    .load

// stream processing
// replace [operator] with the operator of your choice
rates.[operator]

// output stream
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val sq = rates
    .writeStream
    .format("console")
    .option("truncate", false)
    .trigger(Trigger.ProcessingTime(10.seconds))
    .outputMode(OutputMode.Complete)
    .queryName("rate-console")
    .start

// eventually...
sq.stop
```

dropDuplicates Operator — Streaming Deduplication

```
dropDuplicates(): Dataset[T]
dropDuplicates(colNames: Seq[String]): Dataset[T]
dropDuplicates(col1: String, cols: String*): Dataset[T]
```

`dropDuplicates` operator...FIXME

Note

For a streaming Dataset, `dropDuplicates` will keep all data across triggers as intermediate state to drop duplicates rows. You can use [withWatermark](#) operator to limit how late the duplicate data can be and system will accordingly limit the state. In addition, too late data older than watermark will be dropped to avoid any possibility of duplicates.

```
scala> spark.version
res0: String = 2.3.0-SNAPSHOT

// Start a streaming query
// Using old-fashioned MemoryStream (with the deprecated SQLContext)
import org.apache.spark.sql.execution.streaming.MemoryStream
import org.apache.spark.sql.SQLContext
implicit val sqlContext: SQLContext = spark.sqlContext
val source = MemoryStream[(Int, Int)]
val ids = source.toDS.toDF("time", "id").
  withColumn("time", $"time" cast "timestamp"). // <-- convert time column from Int to
  Timestamp
  dropDuplicates("id").
  withColumn("time", $"time" cast "long") // <-- convert time column back from Timest
amp to Int

// Conversions are only for display purposes
// Internally we need timestamps for watermark to work
// Displaying timestamps could be too much for such a simple task

scala> println(ids.queryExecution.analyzed.numberedTreeString)
00 Project [cast(time#10 as bigint) AS time#15L, id#6]
01 +- Deduplicate [id#6], true
02   +- Project [cast(timestamp#5 as timestamp) AS time#10, id#6]
03     +- Project [_1#2 AS time#5, _2#3 AS id#6]
04       +- StreamingExecutionRelation MemoryStream[_1#2,_2#3], [_1#2, _2#3]

import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration.-
val q = ids.
  writeStream.
    format("memory").
```

```
queryName("dups").
  outputMode(OutputMode.Append).
  trigger(Trigger.ProcessingTime(30.seconds)).
  option("checkpointLocation", "checkpoint-dir"). // <-- use checkpointing to save state between restarts
  start

// Publish duplicate records
source.addData(1 -> 1)
source.addData(2 -> 1)
source.addData(3 -> 1)

q.processAllAvailable()

// Check out how dropDuplicates removes duplicates
// --> per single streaming batch (easy)
scala> spark.table("dups").show
+---+---+
|time| id|
+---+---+
|  1|  1|
+---+---+

source.addData(4 -> 1)
source.addData(5 -> 2)

// --> across streaming batches (harder)
scala> spark.table("dups").show
+---+---+
|time| id|
+---+---+
|  1|  1|
|  5|  2|
+---+---+

// Check out the internal state
scala> println(q.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 2,
  "numRowsUpdated" : 1,
  "memoryUsedBytes" : 17751
}

// You could use web UI's SQL tab instead
// Use Details for Query

source.addData(6 -> 2)

scala> spark.table("dups").show
+---+---+
|time| id|
+---+---+
|  1|  1|
```

```

|   5|  2|
+----+---+
// Check out the internal state
scala> println(q.lastProgress.stateOperators(0).prettyJson)
{
  "numRowsTotal" : 2,
  "numRowsUpdated" : 0,
  "memoryUsedBytes" : 17751
}

// Restart the streaming query
q.stop

val q = ids.
  writeStream.
  format("memory").
  queryName("dups").
  outputMode(OutputMode.Complete). // <-- memory sink supports checkpointing for Complete output mode only
  trigger(Trigger.ProcessingTime(30.seconds)).
  option("checkpointLocation", "checkpoint-dir"). // <-- use checkpointing to save state between restarts
  start

// Doh! MemorySink is fine, but Complete is only available with a streaming aggregation

// Answer it if you know why --> https://stackoverflow.com/q/45756997/1305344

// It's a high time to work on https://issues.apache.org/jira/browse/SPARK-21667
// to understand the low-level details (and the reason, it seems)

// Disabling operation checks and starting over
// ./bin/spark-shell -c spark.sql.streaming.unsupportedOperationCheck=false
// it works now --> no exception!

scala> spark.table("dups").show
+----+---+
|time| id|
+----+---+
+----+---+

source.addData(0 -> 1)
// wait till the batch is triggered
scala> spark.table("dups").show
+----+---+
|time| id|
+----+---+
|   0|  1|
+----+---+

source.addData(1 -> 1)
source.addData(2 -> 1)

```

dropDuplicates Operator

```
// wait till the batch is triggered
scala> spark.table("dups").show
+---+---+
|time| id|
+---+---+
+---+---+

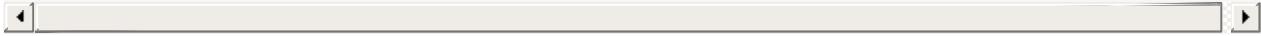
// What?! No rows?! It doesn't look as if it worked fine :(

// Use groupBy to pass the requirement of having streaming aggregation for Complete ou
tput mode
val counts = ids.groupBy("id").agg(first($"time") as "first_time")
scala> counts.explain
== Physical Plan ==
*HashAggregate(keys=[id#246], functions=[first(time#255L, false)])
+- StateStoreSave [id#246], StatefulOperatorStateInfo(<unknown>, 3585583b-42d7-4547-8d62
-255581c48275, 0, 0), Append, 0
    +- *HashAggregate(keys=[id#246], functions=[merge_first(time#255L, false)])
        +- StateStoreRestore [id#246], StatefulOperatorStateInfo(<unknown>, 3585583b-42d7
-4547-8d62-255581c48275, 0, 0)
            +- *HashAggregate(keys=[id#246], functions=[merge_first(time#255L, false)])
                +- *HashAggregate(keys=[id#246], functions=[partial_first(time#255L, false
)])
                    +- *Project [cast(time#250 as bigint) AS time#255L, id#246]
                        +- StreamingDeduplicate [id#246], StatefulOperatorStateInfo(<unknow
n>, 3585583b-42d7-4547-8d62-255581c48275, 1, 0), 0
                            +- Exchange hashpartitioning(id#246, 200)
                                +- *Project [cast(_1#242 as timestamp) AS time#250, _2#243 AS
id#246]
                                    +- StreamingRelation MemoryStream[_1#242,_2#243], [_1#242,
_2#243]
val q = counts.
  writeStream.
format("memory").
queryName("dups").
outputMode(OutputMode.Complete). // <-- memory sink supports checkpointing for Comp
lete output mode only
  trigger(Trigger.ProcessingTime(30.seconds)).
  option("checkpointLocation", "checkpoint-dir"). // <-- use checkpointing to save sta
te between restarts
  start

source.addData(0 -> 1)
source.addData(1 -> 1)
// wait till the batch is triggered
scala> spark.table("dups").show
+---+-----+
| id|first_time|
+---+-----+
|  1|         0|
+---+-----+
// Publish duplicates
```

dropDuplicates Operator

```
// Check out how dropDuplicates removes duplicates  
  
// Stop the streaming query  
// Specify event time watermark to remove old duplicates
```



Dataset.explain High-Level Operator — Explaining Streaming Query Plans

```
explain(): Unit (1)
explain(extended: Boolean): Unit
```

1. Calls `explain` with `extended` flag disabled

`Dataset.explain` is a high-level operator that prints the [logical](#) and (with `extended` flag enabled) [physical](#) plans to the console.

```
val records = spark.
  readStream.
  format("rate").
  load
scala> records.explain
== Physical Plan ==
StreamingRelation rate, [timestamp#0, value#1L]

scala> records.explain(extended = true)
== Parsed Logical Plan ==
StreamingRelation DataSource(org.apache.spark.sql.SparkSession@4071aa13, rate, List(), None, List(), None, Map(), None), rate, [timestamp#0, value#1L]

== Analyzed Logical Plan ==
timestamp: timestamp, value: bigint
StreamingRelation DataSource(org.apache.spark.sql.SparkSession@4071aa13, rate, List(), None, List(), None, Map(), None), rate, [timestamp#0, value#1L]

== Optimized Logical Plan ==
StreamingRelation DataSource(org.apache.spark.sql.SparkSession@4071aa13, rate, List(), None, List(), None, Map(), None), rate, [timestamp#0, value#1L]

== Physical Plan ==
StreamingRelation rate, [timestamp#0, value#1L]
```

Internally, `explain` creates a `ExplainCommand` runnable command with the logical plan and `extended` flag.

`explain` then executes the plan with `ExplainCommand` runnable command and collects the results that are printed out to the standard output.

explain uses `SparkSession` to access the current `SessionState` to execute the plan.

Note

```
import org.apache.spark.sql.execution.command.ExplainCommand
val explain = ExplainCommand(...)
spark.sessionState.executePlan(explain)
```

For streaming Datasets, `ExplainCommand` command simply creates a [IncrementalExecution](#) for the `SparkSession` and the logical plan.

Note

For the purpose of `explain`, `IncrementalExecution` is created with the output mode `Append`, checkpoint location `<unknown>`, run id a random number, current batch id `0` and offset metadata empty. They do not really matter when explaining the load-part of a streaming query.

groupBy Operator — Untyped Streaming Aggregation (with Implicit State Logic)

```
groupBy(cols: Column*): RelationalGroupedDataset
groupBy(col1: String, cols: String*): RelationalGroupedDataset
```

groupBy operator...FIXME

```
scala> spark.version
res0: String = 2.3.0-SNAPSHOT

// Since I'm with SNAPSHOT
// Remember to remove ~/.ivy2/cache/org.apache.spark
// Make sure that ~/.ivy2/jars/org.apache.spark_spark-sql-kafka-0-10_2.11-2.3.0-SNAPSHOT.jar is the latest
// Start spark-shell as follows
/***
./bin/spark-shell --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0-SNAPSHOT
***/

val fromTopic1 = spark.
  readStream.
  format("kafka").
  option("subscribe", "topic1").
  option("kafka.bootstrap.servers", "localhost:9092").
  load

// extract event time et al
// time,key,value
/*
2017-08-23T00:00:00.002Z,1,now
2017-08-23T00:05:00.002Z,1,5 mins later
2017-08-23T00:09:00.002Z,1,9 mins later
2017-08-23T00:11:00.002Z,1,11 mins later
2017-08-23T01:00:00.002Z,1,1 hour later
// late event = watermark should be (1 hour - 10 minutes) already
2017-08-23T00:49:59.002Z,1,==> SHOULD NOT BE INCLUDED in aggregation as too late <=

CAUTION: FIXME SHOULD NOT BE INCLUDED is included contrary to my understanding?!
*/
val timedValues = fromTopic1.
  select('value cast "string").
  withColumn("tokens", split('value, ",")).
  withColumn("time", to_timestamp('tokens(0))).
  withColumn("key", 'tokens(1) cast "int").
  withColumn("value", 'tokens(2)).
  select("time", "key", "value")
```

```

// aggregation with watermark
val counts = timedValues.
  withWatermark("time", "10 minutes").
  groupBy("key").
  agg(collect_list('value) as "values", collect_list('time) as "times")

// Note that StatefulOperatorStateInfo is mostly generic
// since no batch-specific values are currently available
// only after the first streaming batch
scala> counts.explain
== Physical Plan ==
ObjectHashAggregate(keys=[key#27], functions=[collect_list(value#33, 0, 0), collect_li
st(time#22-T600000ms, 0, 0)])
+- Exchange hashpartitioning(key#27, 200)
  +- StateStoreSave [key#27], StatefulOperatorStateInfo(<unknown>, 25149816-1f14-4901-
af13-896286a26d42,0,0), Append, 0
    +- ObjectHashAggregate(keys=[key#27], functions=[merge_collect_list(value#33, 0,
0), merge_collect_list(time#22-T600000ms, 0, 0)])
      +- Exchange hashpartitioning(key#27, 200)
        +- StateStoreRestore [key#27], StatefulOperatorStateInfo(<unknown>, 25149816-
1f14-4901-af13-896286a26d42,0,0)
          +- ObjectHashAggregate(keys=[key#27], functions=[merge_collect_list(val
ue#33, 0, 0), merge_collect_list(time#22-T600000ms, 0, 0)])
            +- Exchange hashpartitioning(key#27, 200)
              +- ObjectHashAggregate(keys=[key#27], functions=[partial_collect_
list(value#33, 0, 0), partial_collect_list(time#22-T600000ms, 0, 0)])
                +- EventTimeWatermark time#22: timestamp, interval 10 minutes
                  +- *Project [cast(split(cast(value#1 as string), ,)[0] as t
imestamp) AS time#22, cast(split(cast(value#1 as string), ,)[1] as int) AS key#27, spl
it(cast(value#1 as string), ,)[2] AS value#33]
                    +- StreamingRelation kafka, [key#0, value#1, topic#2, pa
rtition#3, offset#4L, timestamp#5, timestampType#6]

import org.apache.spark.sql.streaming.-
import scala.concurrent.duration.-
val sq = counts.writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(30.seconds)).
  outputMode(OutputMode.Update). // <-- only Update or Complete acceptable because of
groupBy aggregation
  start

// After StreamingQuery was started,
// the physical plan is complete (with batch-specific values)
scala> sq.explain
== Physical Plan ==
ObjectHashAggregate(keys=[key#27], functions=[collect_list(value#33, 0, 0), collect_li
st(time#22-T600000ms, 0, 0)])
+- Exchange hashpartitioning(key#27, 200)
  +- StateStoreSave [key#27], StatefulOperatorStateInfo(file:/private/var/folders/0w/
kb0d3rqn4zb9fcc91pxhgn8w0000gn/T/temporary-635d6519-b6ca-4686-9b6b-5db0e83cf51/state,

```

```
855cec1c-25dc-4a86-ae54-c6cdd4ed02ec, 0, 0), Update, 0
    +- ObjectHashAggregate(keys=[key#27], functions=[merge_collect_list(value#33, 0,
0), merge_collect_list(time#22-T600000ms, 0, 0)])
        +- Exchange hashpartitioning(key#27, 200)
            +- StateStoreRestore [key#27], StatefulOperatorStateInfo(file:/private/var
/folders/0w/kb0d3rqn4zb9fcc91pxhgn8w0000gn/T/temporary-635d6519-b6ca-4686-9b6b-5db0e83
cf51/state,855cec1c-25dc-4a86-ae54-c6cdd4ed02ec, 0, 0)
                +- ObjectHashAggregate(keys=[key#27], functions=[merge_collect_list(val
ue#33, 0, 0), merge_collect_list(time#22-T600000ms, 0, 0)])
                    +- Exchange hashpartitioning(key#27, 200)
                        +- ObjectHashAggregate(keys=[key#27], functions=[partial_collect_
list(value#33, 0, 0), partial_collect_list(time#22-T600000ms, 0, 0)])
                            +- EventTimeWatermark time#22: timestamp, interval 10 minutes
                                +- *Project [cast(split(cast(value#76 as string), ,)[0] as
timestamp) AS time#22, cast(split(cast(value#76 as string), ,)[1] as int) AS key#27,
split(cast(value#76 as string), ,)[2] AS value#33]
                                    +- Scan ExistingRDD[key#75,value#76,topic#77,partition#78
,offset#79L,timestamp#80,timestampType#81]
```

groupByKey Operator — Streaming Aggregation

- [Introduction](#)
- [Example: Aggregating Orders Per Zip Code](#)
- [Example: Aggregating Metrics Per Device](#)

Introduction

```
groupByKey[K: Encoder](func: T => K): KeyValueGroupedDataset[K, T]
```

`groupByKey` operator creates a [KeyValueGroupedDataset](#) (with keys of type `K` and rows of type `T`) to apply aggregation functions over groups of rows (of type `T`) by key (of type `K`) per the given `func` key-generating function.

Note	The type of the input argument of <code>func</code> is the type of rows in the Dataset (i.e. <code>Dataset[T]</code>).
------	---

`groupByKey` simply applies the `func` function to every row (of type `T`) and associates it with a logical group per key (of type `K`).

```
func: T => K
```

Internally, `groupByKey` creates a structured query with the `AppendColumns` unary logical operator (with the given `func` and the analyzed logical plan of the target `Dataset` that `groupByKey` was executed on) and creates a new `QueryExecution`.

In the end, `groupByKey` creates a [KeyValueGroupedDataset](#) with the following:

- Encoders for `K` keys and `T` rows
- The new `QueryExecution` (with the `AppendColumns` unary logical operator)
- The output schema of the analyzed logical plan
- The new columns of the `AppendColumns` logical operator (i.e. the attributes of the key)

```

scala> :type sq
org.apache.spark.sql.Dataset[Long]

val baseCode = 'A'.toInt
val byUpperChar = (n: java.lang.Long) => (n % 3 + baseCode).toString
val kvs = sq.groupByKey(byUpperChar)

scala> :type kvs
org.apache.spark.sql.KeyValueGroupedDataset[String, Long]

// Peeking under the surface of KeyValueGroupedDataset
import org.apache.spark.sql.catalyst.plans.logical.AppendColumns
val appendColumnsOp = kvs.queryExecution.analyzed.collect { case ac: AppendColumns =>
  ac }.head
scala> println(appendColumnsOp.newColumns)
List(value#7)

```

Example: Aggregating Orders Per Zip Code

Go to [Demo: groupByKey Streaming Aggregation in Update Mode](#).

Example: Aggregating Metrics Per Device

The following example code shows how to apply `groupByKey` operator to a structured stream of timestamped values of different devices.

```

// input stream
import java.sql.Timestamp
val signals = spark.
  readStream.
  format("rate").
  option("rowsPerSecond", 1).
  load.
  withColumn("value", $"value" % 10) // <-- randomize the values (just for fun)
  withColumn("deviceId", lit(util.Random.nextInt(10))). // <-- 10 devices randomly assigned to values
  as[(Timestamp, Long, Int)] // <-- convert to a "better" type (from "unpleasant" Row)

// stream processing using groupByKey operator
// groupByKey(func: ((Timestamp, Long, Int)) => K): KeyValueGroupedDataset[K, (Timestamp, Long, Int)]
// K becomes Int which is a device id
val deviceId: ((Timestamp, Long, Int)) => Int = { case (_, _, deviceId) => deviceId }
scala> val signalsByDevice = signals.groupByKey(deviceId)
signalsByDevice: org.apache.spark.sql.KeyValueGroupedDataset[Int,(java.sql.Timestamp, Long, Int)] = org.apache.spark.sql.KeyValueGroupedDataset@19d40bc6

```


withWatermark Operator — Event-Time Watermark

```
withWatermark(eventTime: String, delayThreshold: String): Dataset[T]
```

`withWatermark` specifies the `eventTime` column for **event time watermark** and `delayThreshold` for **event lateness**.

`eventTime` specifies the column to use for watermark and can be either part of `Dataset` from the source or custom-generated using `current_time` or `current_timestamp` functions.

Note	<p>Watermark tracks a point in time before which it is assumed no more late events are supposed to arrive (and if they have, the late events are considered really late and simply dropped).</p>
------	---

Note	<p>Spark Structured Streaming uses watermark for the following:</p> <ul style="list-style-type: none">• To know when a given time window aggregation (using groupBy operator with window function) can be finalized and thus emitted when using output modes that do not allow updates, like Append output mode.• To minimize the amount of state that we need to keep for ongoing aggregations, e.g. mapGroupsWithState (for implicit state management), flatMapGroupsWithState (for user-defined state management) and dropDuplicates operators.
------	---

The **current watermark** is computed by looking at the maximum `eventTime` seen across all of the partitions in a query minus a user-specified `delayThreshold`. Due to the cost of coordinating this value across partitions, the actual watermark used is only guaranteed to be at least `delayThreshold` behind the actual event time.

Note	<p>In some cases Spark may still process records that arrive more than <code>delayThreshold</code> late.</p>
------	--

window Function — Stream Time Windows

`window` is a standard function that generates **tumbling**, **sliding** or **delayed** stream time window ranges (on a timestamp column).

```
window(
    timeColumn: Column,
    windowDuration: String): Column  (1)
window(
    timeColumn: Column,
    windowDuration: String,
    slideDuration: String): Column   (2)
window(
    timeColumn: Column,
    windowDuration: String,
    slideDuration: String,
    startTime: String): Column       (3)
```

1. Creates a tumbling time window with `slideDuration` as `windowDuration` and `0` second for `startTime`
2. Creates a sliding time window with `0` second for `startTime`
3. Creates a delayed time window

	From Tumbling Window (Azure Stream Analytics) :
Note	<ul style="list-style-type: none"> ■ Tumbling windows are a series of fixed-sized, non-overlapping and contiguous time intervals.

	From Introducing Stream Windows in Apache Flink :
Note	<ul style="list-style-type: none"> ■ Tumbling windows group elements of a stream into finite sets where each set corresponds to an interval. ■ Tumbling windows discretize a stream into non-overlapping windows.

```
scala> val timeColumn = window($"time", "5 seconds")
timeColumn: org.apache.spark.sql.Column = timewindow(time, 5000000, 5000000, 0) AS `window`
```

`timeColumn` should be of `TimestampType`, i.e. with [java.sql.Timestamp](#) values.

Tip	Use java.sql.Timestamp.from or java.sql.Timestamp.valueOf factory methods to create <code>Timestamp</code> instances.
-----	---

```
// https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html
import java.time.LocalDateTime
// https://docs.oracle.com/javase/8/docs/api/java/sql/Timestamp.html
import java.sql.Timestamp
val levels = Seq(
    // (year, month, dayOfMonth, hour, minute, second)
    ((2012, 12, 12, 12, 12, 12), 5),
    ((2012, 12, 12, 12, 12, 14), 9),
    ((2012, 12, 12, 13, 13, 14), 4),
    ((2016, 8, 13, 0, 0, 0), 10),
    ((2017, 5, 27, 0, 0, 0), 15)).
    map { case ((yy, mm, dd, h, m, s), a) => (LocalDateTime.of(yy, mm, dd, h, m, s), a)}
).
    map { case (ts, a) => (Timestamp.valueOf(ts), a) }.
    toDF("time", "level")
scala> levels.show
+-----+-----+
|          time|level|
+-----+-----+
|2012-12-12 12:12:12|    5|
|2012-12-12 12:12:14|    9|
|2012-12-12 13:13:14|    4|
|2016-08-13 00:00:00|   10|
|2017-05-27 00:00:00|   15|
+-----+-----+

val q = levels.select(window($"time", "5 seconds"), $"level")
scala> q.show(truncate = false)
+-----+-----+
|window                               |level|
+-----+-----+
|[2012-12-12 12:12:10.0,2012-12-12 12:12:15.0]|5    |
|[2012-12-12 12:12:10.0,2012-12-12 12:12:15.0]|9    |
|[2012-12-12 13:13:10.0,2012-12-12 13:13:15.0]|4    |
|[2016-08-13 00:00:00.0,2016-08-13 00:00:05.0]|10   |
|[2017-05-27 00:00:00.0,2017-05-27 00:00:05.0]|15   |
+-----+-----+

scala> q.printSchema
root
|-- window: struct (nullable = true)
|   |-- start: timestamp (nullable = true)
|   |-- end: timestamp (nullable = true)
|-- level: integer (nullable = false)

// calculating the sum of levels every 5 seconds
val sums = levels.
    groupBy(window($"time", "5 seconds")).
    agg(sum("level") as "level_sum").
    select("window.start", "window.end", "level_sum")
scala> sums.show
+-----+-----+-----+
```

	start	end	level_sum
	2012-12-12 13:13:10	2012-12-12 13:13:15	4
	2012-12-12 12:12:10	2012-12-12 12:12:15	14
	2016-08-13 00:00:00	2016-08-13 00:00:05	10
	2017-05-27 00:00:00	2017-05-27 00:00:05	15

`windowDuration` and `slideDuration` are strings specifying the width of the window for duration and sliding identifiers, respectively.

Tip

Use `CalendarInterval` for valid window identifiers.

There are a couple of rules governing the durations:

1. The window duration must be greater than 0
2. The slide duration must be greater than 0.
3. The start time must be greater than or equal to 0.
4. The slide duration must be less than or equal to the window duration.
5. The start time must be less than the slide duration.

Note

Only one `window` expression is supported in a query.

Note

`null` values are filtered out in `window` expression.

Internally, `window` creates a [Column](#) with `TimeWindow` Catalyst expression under `window` alias.

```
scala> val timeColumn = window($"time", "5 seconds")
timeColumn: org.apache.spark.sql.Column = timewindow(time, 5000000, 5000000, 0) AS `window`  
  

val windowExpr = timeColumn.expr
scala> println(windowExpr.numberedTreeString)
00 timewindow('time, 5000000, 5000000, 0) AS window#23
01 +- timewindow('time, 5000000, 5000000, 0)
02     +- 'time
```

Internally, `TimeWindow` Catalyst expression is simply a struct type with two fields, i.e. `start` and `end`, both of `TimestampType` type.

```

scala> println(windowExpr.dataType)
StructType(StructField(start,.TimestampType,true), StructField(end,.TimestampType,true))

scala> println(windowExpr.dataType.prettyJson)
{
  "type" : "struct",
  "fields" : [ {
    "name" : "start",
    "type" : "timestamp",
    "nullable" : true,
    "metadata" : { }
  }, {
    "name" : "end",
    "type" : "timestamp",
    "nullable" : true,
    "metadata" : { }
  } ]
}

```

Note

`TimeWindow` time window Catalyst expression is planned (i.e. *converted*) in `TimeWindowing` logical optimization rule (i.e. `Rule[LogicalPlan]`) of the Spark SQL logical query plan analyzer.

Find more about the Spark SQL logical query plan analyzer in [Mastering Apache Spark 2](#) gitbook.

Example — Traffic Sensor

Note

The example is borrowed from [Introducing Stream Windows in Apache Flink](#).

The example shows how to use `window` function to model a traffic sensor that counts every 15 seconds the number of vehicles passing a certain location.

KeyValueGroupedDataset — Streaming Aggregation

`KeyValueGroupedDataset` represents a **grouped dataset** as a result of `Dataset.groupByKey` operator (that aggregates records by a grouping function).

```
// Dataset[T]
groupByKey(func: T => K): KeyValueGroupedDataset[K, T]
```

```
import java.sql.Timestamp
val numGroups = spark.
  readStream.
  format("rate").
  load.
  as[(Timestamp, Long)].
  groupByKey { case (time, value) => value % 2 }

scala> :type numGroups
org.apache.spark.sql.KeyValueGroupedDataset[Long, (java.sql.Timestamp, Long)]
```

`KeyValueGroupedDataset` is also created for `KeyValueGroupedDataset.keyAs` and `KeyValueGroupedDataset.mapValues` operators.

```
scala> :type numGroups
org.apache.spark.sql.KeyValueGroupedDataset[Long, (java.sql.Timestamp, Long)]

scala> :type numGroups.keyAs[String]
org.apache.spark.sql.KeyValueGroupedDataset[String, (java.sql.Timestamp, Long)]
```

```
scala> :type numGroups
org.apache.spark.sql.KeyValueGroupedDataset[Long, (java.sql.Timestamp, Long)]

val mapped = numGroups.mapValues { case (ts, n) => s"($ts, $n)" }
scala> :type mapped
org.apache.spark.sql.KeyValueGroupedDataset[Long, String]
```

`KeyValueGroupedDataset` works for batch and streaming aggregations, but shines the most when used for [Streaming Aggregation](#).

```

scala> :type numGroups
org.apache.spark.sql.KeyValueGroupedDataset[Long, (java.sql.Timestamp, Long)]


import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._

numGroups.
  mapGroups { case(group, values) => values.size }.
  writeStream.
  format("console").
  trigger(Trigger.ProcessingTime(10.seconds)).
  start

-----
Batch: 0
-----
+---+
| value|
+---+
+---+


-----
Batch: 1
-----
+---+
| value|
+---+
|   3|
|   2|
+---+


-----
Batch: 2
-----
+---+
| value|
+---+
|   5|
|   5|
+---+


// Eventually...
spark.streams.active.foreach(_.stop)

```

The most prestigious use case of `KeyValueGroupedDataset` however is [Arbitrary Stateful Streaming Aggregation](#) that allows for accumulating **streaming state** (by means of `GroupState`) using `mapGroupsWithState` and the more advanced `flatMapGroupsWithState` operators.

Table 1. KeyValueGroupedDataset's Operators

Operator	Description

agg	<pre> agg[U1](col1: TypedColumn[V, U1]): Dataset[(K, U1)] agg[U1, U2](col1: TypedColumn[V, U1], col2: TypedColumn[V, U2]): Dataset[(K, U1, U2)] agg[U1, U2, U3](col1: TypedColumn[V, U1], col2: TypedColumn[V, U2], col3: TypedColumn[V, U3]): Dataset[(K, U1, U2, U3)] agg[U1, U2, U3, U4](col1: TypedColumn[V, U1], col2: TypedColumn[V, U2], col3: TypedColumn[V, U3], col4: TypedColumn[V, U4]): Dataset[(K, U1, U2, U3, U4)] </pre>		
cogroup	<pre>cogroup[U, R : Encoder](other: KeyValueGroupedDataset[K, U])(f: (K, Iterator[V], Iterator[U]) => TraversableOnce[R]): Dataset[(K, R)]</pre>		
count	<pre>count(): Dataset[(K, Long)]</pre>		
flatMapGroups	<pre>flatMapGroups[U : Encoder](f: (K, Iterator[V]) => TraversableOnce[U]): Dataset[(K, U)]</pre>		
flatMapGroupsWithState	<pre>flatMapGroupsWithState[S: Encoder, U: Encoder](outputMode: OutputMode, timeoutConf: GroupStateTimeout)(func: (K, Iterator[V], GroupState[S]) => Iterator[U]): Dataset[(K, U)]</pre> <p>Arbitrary Stateful Streaming Aggregation - streaming aggregation and state timeout</p> <table border="1"> <tr> <td>Note</td><td>The difference between this <code>flatMapGroupsWithState</code> <code>mapGroupsWithState</code> operators is the state function or more elements (that are in turn the rows in the resulting Dataset).</td></tr> </table>	Note	The difference between this <code>flatMapGroupsWithState</code> <code>mapGroupsWithState</code> operators is the state function or more elements (that are in turn the rows in the resulting Dataset).
Note	The difference between this <code>flatMapGroupsWithState</code> <code>mapGroupsWithState</code> operators is the state function or more elements (that are in turn the rows in the resulting Dataset).		
keyAs	<pre>keys: Dataset[K] keyAs[L : Encoder]: KeyValueGroupedDataset[L, V]</pre>		
mapGroups	<pre>mapGroups[U : Encoder](f: (K, Iterator[V]) => U): Dataset[U]</pre>		
	<pre>mapGroupsWithState[S: Encoder, U: Encoder](func: (K, Iterator[V], GroupState[S]) => U): Dataset[U] mapGroupsWithState[S: Encoder, U: Encoder](timeoutConf: GroupStateTimeout)(func: (K, Iterator[V], GroupState[S]) => U): Dataset[U]</pre>		

<code>mapGroupsWithState</code>	Creates a new Dataset with FlatMapGroupsWithState logical <table border="1"> <tr> <td>Note</td><td>The difference between <code>mapGroupsWithState</code> and <code>flatMapGroupsWithState</code> is the state function that generates exactly one element per row in the result dataset.</td></tr> </table>	Note	The difference between <code>mapGroupsWithState</code> and <code>flatMapGroupsWithState</code> is the state function that generates exactly one element per row in the result dataset.
Note	The difference between <code>mapGroupsWithState</code> and <code>flatMapGroupsWithState</code> is the state function that generates exactly one element per row in the result dataset.		
<code>mapValues</code>	<code>mapValues[W : Encoder](func: V => W): KeyValueGroupedDataset[W]</code>		
<code>reduceGroups</code>	<code>reduceGroups(f: (V, V) => V): Dataset[(K, V)]</code>		

Creating KeyValueGroupedDataset Instance

`KeyValueGroupedDataset` takes the following when created:

- `Encoder` for keys
- `Encoder` for values
- `QueryExecution`
- Data attributes
- Grouping attributes

mapGroupsWithState Operator — Stateful Streaming Aggregation (with Explicit State Logic)

```
mapGroupsWithState[S: Encoder, U: Encoder](
  func: (K, Iterator[V], GroupState[S]) => U): Dataset[U] (1)
mapGroupsWithState[S: Encoder, U: Encoder](
  timeoutConf: GroupStateTimeout)(
  func: (K, Iterator[V], GroupState[S]) => U): Dataset[U]
```

1. Uses `GroupStateTimeout.NoTimeout` for `timeoutConf`

`mapGroupsWithState` operator...FIXME

Note

`mapGroupsWithState` is a special case of [flatMapGroupsWithState](#) operator with the following:

- `func` being transformed to return a single-element `Iterator`
- [Update](#) output mode

`mapGroupsWithState` also creates a `FlatMapGroupsWithState` with [isMapGroupsWithState](#) internal flag enabled.

```
// numGroups defined at the beginning
scala> :type numGroups
org.apache.spark.sql.KeyValueGroupedDataset[Long, (java.sql.Timestamp, Long)]

import org.apache.spark.sql.streaming.GroupState
def mappingFunc(key: Long, values: Iterator[(java.sql.Timestamp, Long)], state: GroupState[Long]): Long = {
  println(s">>> key: $key => state: $state")
  val newState = state.getOption.map(_ + values.size).getOrElse(0L)
  state.update(newState)
  key
}

import org.apache.spark.sql.streaming.GroupStateTimeout
val longs = numGroups.mapGroupsWithState(
  timeoutConf = GroupStateTimeout.ProcessingTimeTimeout(
    func = mappingFunc)

import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration._
val q = longs.
  writeStream.
```

mapGroupsWithState Operator

```
format("console").  
trigger(Trigger.ProcessingTime(10.seconds)).  
outputMode(OutputMode.Update). // <-- required for mapGroupsWithState  
start  
  
// Note GroupState  
  
-----  
Batch: 1  
-----  
>>> key: 0 => state: GroupState(<undefined>)  
>>> key: 1 => state: GroupState(<undefined>)  
+---+  
|value|  
+---+  
| 0 |  
| 1 |  
+---+  
  
-----  
Batch: 2  
-----  
>>> key: 0 => state: GroupState(0)  
>>> key: 1 => state: GroupState(0)  
+---+  
|value|  
+---+  
| 0 |  
| 1 |  
+---+  
  
-----  
Batch: 3  
-----  
>>> key: 0 => state: GroupState(4)  
>>> key: 1 => state: GroupState(4)  
+---+  
|value|  
+---+  
| 0 |  
| 1 |  
+---+  
  
// in the end  
spark.streams.active.foreach(_.stop)
```

flatMapGroupsWithState Operator — Arbitrary Stateful Streaming Aggregation (with Explicit State Logic)

```
KeyValueGroupedDataset[K, V].flatMapGroupsWithState[S: Encoder, U: Encoder](
  outputMode: OutputMode,
  timeoutConf: GroupStateTimeout)(
  func: (K, Iterator[V], GroupState[S]) => Iterator[U]): Dataset[U]
```

`flatMapGroupsWithState` operator is used for [Arbitrary Stateful Streaming Aggregation \(with Explicit State Logic\)](#).

`flatMapGroupsWithState` requires that the given `OutputMode` is either `Append` or `Update` (and reports an `IllegalArgumentException` at runtime).

Note

An `OutputMode` is a required argument, but does not seem to be used at all. Check out the question [What's the purpose of OutputMode in flatMapGroupsWithState? How/where is it used?](#) on StackOverflow.

Every time the state function `func` is executed for a key, the state (as `GroupState[S]`) is for this key only.

Note

- `k` is the type of the keys in `KeyValueGroupedDataset`
- `v` is the type of the values (per key) in `KeyValueGroupedDataset`
- `s` is the user-defined type of the state as maintained for each group
- `u` is the type of rows in the result `Dataset`

Internally, `flatMapGroupsWithState` creates a new `Dataset` with [FlatMapGroupsWithState](#) unary logical operator.

StreamingQueryManager — Streaming Query Management

`StreamingQueryManager` is the management interface for streaming queries in a single `SparkSession`.

Table 1. `StreamingQueryManager` API

Method	Description
<code>active</code>	<pre>active: Array[StreamingQuery]</pre> <p>Returns active structured queries</p>
<code>addListener</code>	<pre>addListener(listener: StreamingQueryListener): Unit</pre> <p>Registers a <code>StreamingQueryListener</code></p>
<code>awaitAnyTermination</code>	<pre>awaitAnyTermination(): Unit awaitAnyTermination(timeoutMs: Long): Boolean</pre> <p>Waits for any streaming query to be terminated</p>
<code>get</code>	<pre>get(id: String): StreamingQuery get(id: UUID): StreamingQuery</pre> <p>Gets the <code>StreamingQuery</code> by id</p>
<code>removeListener</code>	<pre>removeListener(listener: StreamingQueryListener): Unit</pre> <p>De-registers the <code>StreamingQueryListener</code></p>
<code>resetTerminated</code>	<pre>resetTerminated(): Unit</pre> <p>Resets the internal registry of the terminated streaming queries (that lets <code>awaitAnyTermination</code> to be used again)</p>

`StreamingQueryManager` is available using `SparkSession.streams` property.

```

scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.streams
org.apache.spark.sql.streaming.StreamingQueryManager

```

`StreamingQueryManager` is created when `SessionState` is created.

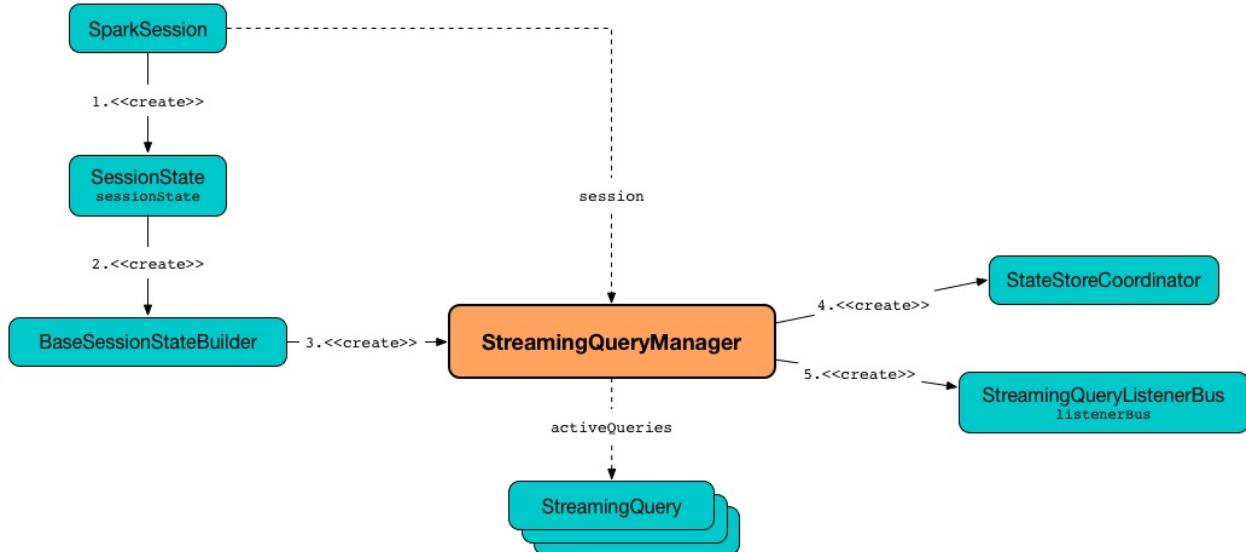


Figure 1. `StreamingQueryManager`

Tip	Refer to the Mastering Apache Spark 2 gitbook to learn about <code>SessionState</code> .
-----	--

`StreamingQueryManager` is used (internally) to create a `StreamingQuery` (with its `StreamExecution`).

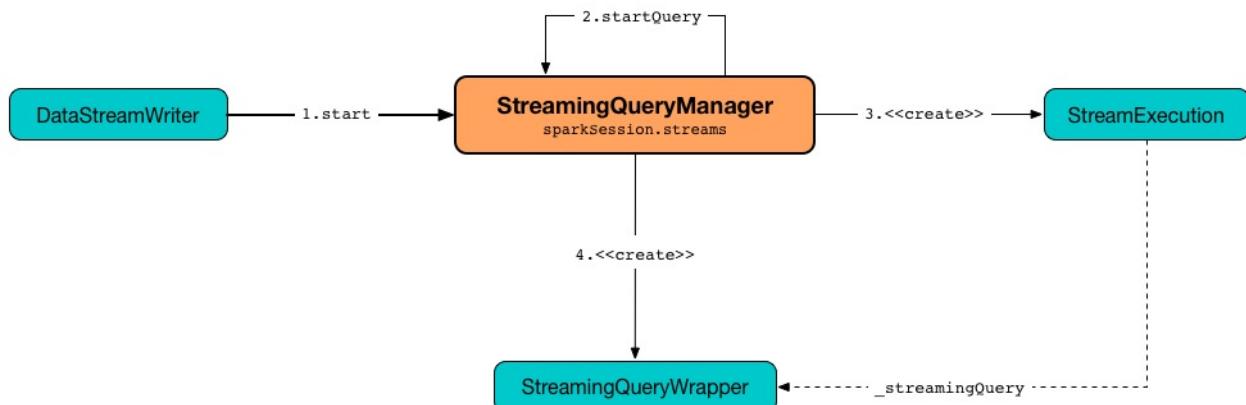


Figure 2. `StreamingQueryManager` Creates `StreamingQuery` (and `StreamExecution`)
`StreamingQueryManager` is notified about state changes of a structured query and passes them along (to query listeners).

`StreamingQueryManager` takes a single `SparkSession` when created.

Table 2. StreamingQueryManager's Internal Registries and Counters (in alphabetical order)

Name	Description
activeQueries	<p>Registry of <code>StreamingQueries</code> per <code>UUID</code></p> <p>Used when <code>StreamingQueryManager</code> is requested for <code>active streaming queries</code>, <code>get a streaming query by id</code>, starts a <code>streaming query</code> and is notified that a <code>streaming query</code> has terminated.</p>
activeQueriesLock	
awaitTerminationLock	
lastTerminatedQuery	<p><code>StreamingQuery</code> that has recently been terminated, i.e. stopped or due to an exception.</p> <p><code>null</code> when no <code>streaming query</code> has terminated yet or <code>resetTerminated</code>.</p> <ul style="list-style-type: none"> Used in <code>awaitAnyTermination</code> to know when a <code>streaming query</code> has terminated Set when <code>StreamingQueryManager</code> is notified that a <code>streaming query</code> has terminated
listenerBus	<p><code>StreamingQueryListenerBus</code> (for the current <code>SparkSession</code>)</p> <p>Used to:</p> <ul style="list-style-type: none"> register or deregister a <code>StreamingQueryListener</code> Post a streaming event (and notify <code>StreamingQueryListener</code> listeners about streaming events)
stateStoreCoordinator	<p><code>StateStoreCoordinatorRef</code> to the <code>StateStoreCoordinator</code> RPC Endpoint</p> <ul style="list-style-type: none"> Created when <code>StreamingQueryManager</code> is created <p>Used when:</p> <ul style="list-style-type: none"> <code>StreamingQueryManager</code> is notified that a <code>streaming query</code> has terminated Stateful operators are executed, i.e. <code>FlatMapGroupsWithStateExec</code>, <code>StateStoreRestoreExec</code>, <code>StateStoreSaveExec</code>, <code>StreamingDeduplicateExec</code> and <code>StreamingSymmetricHashJoinExec</code> Creating <code>StateStoreRDD</code> (with <code>storeUpdateFunction</code> aborting <code>StateStore</code> when a task fails)

Getting All Active Streaming Queries — `active` Method

```
active: Array[StreamingQuery]
```

`active` gets all active streaming queries.

Getting Active Continuous Query By Name — `get` Method

```
get(name: String): StreamingQuery
```

`get` method returns a `StreamingQuery` by `name`.

It may throw an `IllegalArgumentException` when no `StreamingQuery` exists for the `name`.

```
java.lang.IllegalArgumentException: There is no active query with name hello
  at org.apache.spark.sql.StreamingQueryManager$$anonfun$get$1.apply(StreamingQueryManager.scala:59)
  at org.apache.spark.sql.StreamingQueryManager$$anonfun$get$1.apply(StreamingQueryManager.scala:59)
  at scala.collection.MapLike$class.getOrElse(MapLike.scala:128)
  at scala.collection.AbstractMap.getOrElse(Map.scala:59)
  at org.apache.spark.sql.StreamingQueryManager.get(StreamingQueryManager.scala:58)
  ... 49 elided
```

Registering StreamingQueryListener — `addListener` Method

```
addListener(listener: StreamingQueryListener): Unit
```

`addListener` requests the `StreamingQueryListenerBus` to add the input `listener`.

De-Registering StreamingQueryListener — `removeListener` Method

```
removeListener(listener: StreamingQueryListener): Unit
```

`removeListener` requests `StreamingQueryListenerBus` to remove the input `listener`.

Waiting for Any Streaming Query Termination — `awaitAnyTermination` Method

```
awaitAnyTermination(): Unit
awaitAnyTermination(timeoutMs: Long): Boolean
```

`awaitAnyTermination` acquires a lock on `awaitTerminationLock` and waits until any streaming query has finished (i.e. `lastTerminatedQuery` is available) or `timeoutMs` has expired.

`awaitAnyTermination` re-throws the `StreamingQueryException` from `lastTerminatedQuery` if it reported one.

`resetTerminated` Method

```
resetTerminated(): Unit
```

`resetTerminated` forgets about the past-terminated query (so that `awaitAnyTermination` can be used again to wait for a new streaming query termination).

Internally, `resetTerminated` acquires a lock on `awaitTerminationLock` and simply resets `lastTerminatedQuery` (i.e. sets it to `null`).

Creating Streaming Query — `createQuery` Internal Method

```
createQuery(
  userSpecifiedName: Option[String],
  userSpecifiedCheckpointLocation: Option[String],
  df: DataFrame,
  extraOptions: Map[String, String],
  sink: BaseStreamingSink,
  outputMode: OutputMode,
  useTempCheckpointLocation: Boolean,
  recoverFromCheckpointLocation: Boolean,
  trigger: Trigger,
  triggerClock: Clock): StreamingQueryWrapper
```

`createQuery` creates a `StreamingQueryWrapper` (for a `StreamExecution` per the input user-defined properties).

Internally, `createQuery` first finds the name of the checkpoint directory of a query (aka **checkpoint location**) in the following order:

1. Exactly the input `userSpecifiedCheckpointLocation` if defined
2. `spark.sql.streaming.checkpointLocation` Spark property if defined for the parent directory with a subdirectory per the optional `userSpecifiedName` (or a randomly-generated UUID)
3. (only when `useTempCheckpointLocation` is enabled) A temporary directory (as specified by `java.io.tmpdir` JVM property) with a subdirectory with `temporary` prefix.

Note

`userSpecifiedCheckpointLocation` can be any path that is acceptable by Hadoop's [Path](#).

If the directory name for the checkpoint location could not be found, `createQuery` reports a `AnalysisException`.

```
checkpointLocation must be specified either through option("checkpointLocation", ...)
or SparkSession.conf.set("spark.sql.streaming.checkpointLocation", ...)
```

`createQuery` reports a `AnalysisException` when the input `recoverFromCheckpointLocation` flag is turned off but there is **offsets** directory in the checkpoint location.

`createQuery` makes sure that the logical plan of the structured query is analyzed (i.e. no logical errors have been found).

Unless `spark.sql.streaming.unsupportedOperationCheck` Spark property is turned on, `createQuery` checks the logical plan of the streaming query for unsupported operations.

(only when `spark.sql.adaptive.enabled` Spark property is turned on) `createQuery` prints out a WARN message to the logs:

```
WARN spark.sql.adaptive.enabled is not supported in streaming DataFrames/Datasets and
will be disabled.
```

In the end, `createQuery` creates a [StreamingQueryWrapper](#) with a new [MicroBatchExecution](#).

`recoverFromCheckpointLocation` flag corresponds to `recoverFromCheckpointLocation` flag that `streamingQueryManager` uses to start a streaming query and which is enabled by default (and is in fact the only place where `createQuery` is used).

Note

- `memory` sink has the flag enabled for Complete output mode only
- `foreach` sink has the flag always enabled
- `console` sink has the flag always disabled
- all other sinks have the flag always enabled

Note

`userSpecifiedName` corresponds to `queryName` option (that can be defined using `DataStreamWriter`'s `queryName` method) while `userSpecifiedCheckpointLocation` is `checkpointLocation` option.

Note

`createQuery` is used exclusively when `StreamingQueryManager` is requested to start a streaming query (when `DataStreamWriter` is requested to start an execution of a streaming query).

Starting Streaming Query Execution — `startQuery` Internal Method

```
startQuery(
    userSpecifiedName: Option[String],
    userSpecifiedCheckpointLocation: Option[String],
    df: DataFrame,
    extraOptions: Map[String, String],
    sink: BaseStreamingSink,
    outputMode: OutputMode,
    useTempCheckpointLocation: Boolean = false,
    recoverFromCheckpointLocation: Boolean = true,
    trigger: Trigger = ProcessingTime(0),
    triggerClock: Clock = new SystemClock()): StreamingQuery
```

`startQuery` starts a streaming query and returns a handle to it.

Note

`trigger` defaults to 0 milliseconds (as `ProcessingTime(0)`).

Internally, `startQuery` first creates a `StreamingQueryWrapper`, registers it in `activeQueries` internal registry (by the `id`), requests it for the underlying `StreamExecution` and starts it.

In the end, `startQuery` returns the `StreamingQueryWrapper` (as part of the fluent API so you can chain operators) or throws the exception that was reported when attempting to start the query.

`startQuery` throws an `IllegalArgumentException` when there is another query registered under `name`. `startQuery` looks it up in the `activeQueries` internal registry.

Cannot start query with name [name] as a query with that name is already active

`startQuery` throws an `IllegalStateException` when a query is started again from checkpoint. `startQuery` looks it up in `activeQueries` internal registry.

Cannot start query with id [id] as another query with same id is already active. Perhaps you are attempting to restart a query from checkpoint that is already active.

Note	<code>startQuery</code> is used exclusively when <code>DataStreamWriter</code> is requested to start an execution of the streaming query .
------	--

Posting StreamingQueryListener Event to StreamingQueryListenerBus — `postListenerEvent` Internal Method

```
postListenerEvent(event: StreamingQueryListener.Event): Unit
```

`postListenerEvent` simply posts the input `event` to [StreamingQueryListenerBus](#).

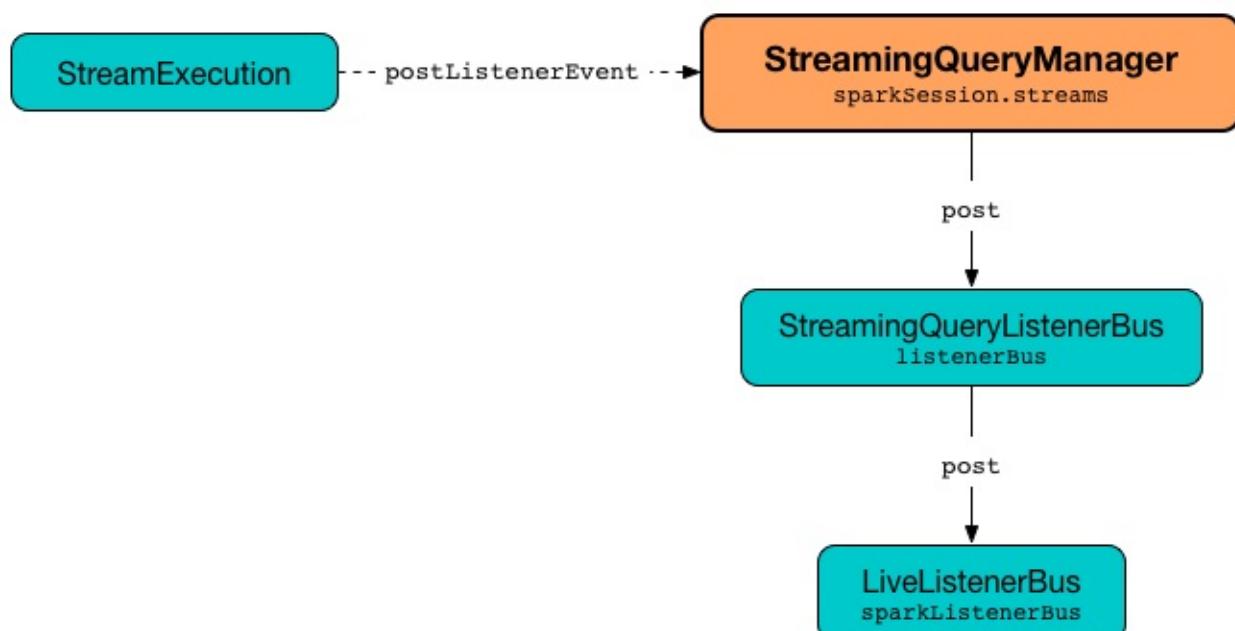


Figure 3. StreamingQueryManager Propagates StreamingQueryListener Events

Note	<code>postListenerEvent</code> is used exclusively when <code>StreamExecution</code> posts a streaming event .
------	--

Handling Termination of Streaming Query (and Deactivating Query in StateStoreCoordinator)

— `notifyQueryTermination` Internal Method

```
notifyQueryTermination(terminatedQuery: StreamingQuery): Unit
```

`notifyQueryTermination` removes the `terminatedQuery` from `activeQueries` internal registry (by the `query id`).

`notifyQueryTermination` records the `terminatedQuery` in `lastTerminatedQuery` internal registry (when no earlier streaming query was recorded or the `terminatedQuery` terminated due to an exception).

`notifyQueryTermination` notifies others that are blocked on `awaitTerminationLock`.

In the end, `notifyQueryTermination` requests `StateStoreCoordinator` to deactivate all active runs of the streaming query.

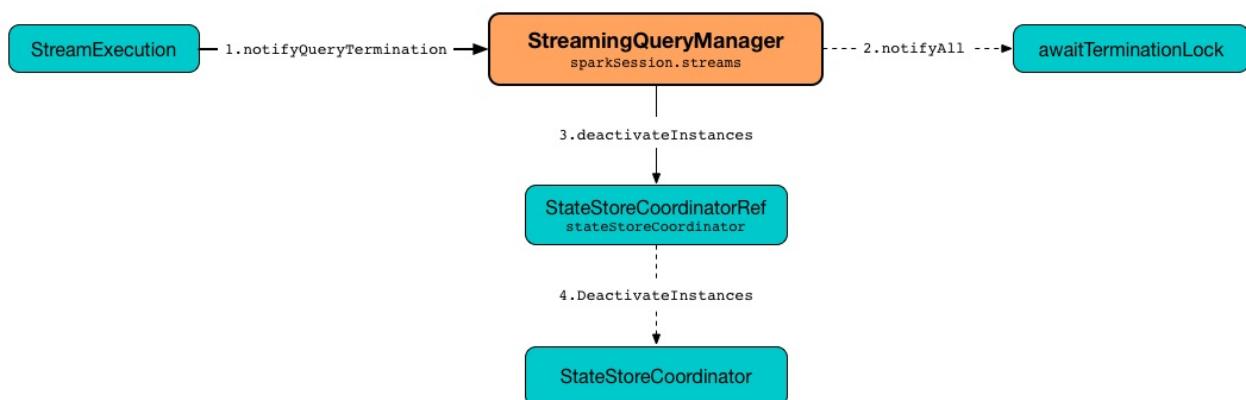


Figure 4. StreamingQueryManager's Marking Streaming Query as Terminated

Note	<code>notifyQueryTermination</code> is used exclusively when <code>StreamExecution</code> is requested to run a streaming query and the query has finished (running streaming batches) (with or without an exception).
------	--

SQLConf—Internal Configuration Store

`SQLConf` is an **internal key-value configuration store** for parameters and hints used to configure a Spark Structured Streaming application (and Spark SQL applications in general).

The parameters and hints are accessible as [property accessor methods](#).

`SQLConf` is available as the `conf` property of the `SessionState` of a `SparkSession`.

```
scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.sessionState.conf
org.apache.spark.sql.internal.SQLConf
```

Table 1. SQLConf's Property Accessor Methods

Method Name / Property	Description
<code>continuousStreamingExecutorQueueSize</code> spark.sql.streaming.continuous.executorQueueSize	Used when: <ul style="list-style-type: none"> <code>DataSourceV2Scan</code> physical operator the input RDDs (ContinuousDataSource) <code>ContinuousCoales</code> physical operator <code>execute</code>
<code>continuousStreamingExecutorPollIntervalMs</code> spark.sql.streaming.continuous.executorPollIntervalMs	Used exclusively when <code>DataSourceV2ScanExec</code> operator is requested for RDDs (and creates a ContinuousDataSource)
<code>disabledV2StreamingMicroBatchReaders</code> spark.sql.streaming.disabledV2MicroBatchReaders	Used exclusively when <code>MicroBatchExecution</code> the analyzed logical plan of a streaming query)
<code>fileSourceLogDeletion</code> spark.sql.streaming.fileSource.log.deletion	Used exclusively when <code>FileStreamSourceLog</code> the <code>isDeletingExpiredFile</code>
<code>fileSourceLogCleanupDelay</code> spark.sql.streaming.fileSource.log.cleanupDelay	Used exclusively when <code>FileStreamSourceLog</code> the <code>fileCleanupDelay</code>

<pre>fileSourceLogCompactInterval</pre> <p>spark.sql.streaming.fileSource.log.compactInterval</p>	<p>Used exclusively when FileStreamSourceLog is used as the default compaction.</p>
<pre>FLATMAPGROUPSWITHSTATE_STATE_FORMAT_VERSION</pre> <p>spark.sql.streaming.flatMapGroupsWithState.stateFormatVersion</p>	<p>Used when:</p> <ul style="list-style-type: none"> • FlatMapGroupsWithState execution plan is requested to plan a query (and creates the FlatMapGroupsWithState physical operator) • Among the check properties
<pre>minBatchesToRetain</pre> <p>spark.sql.streaming.minBatchesToRetain</p>	<p>Used when:</p> <ul style="list-style-type: none"> • CompactibleFileSets are created • StreamExecution • StateStoreConf is used
<pre>SHUFFLE_PARTITIONS</pre> <p>spark.sql.shuffle.partitions</p>	<p>See spark.sql.shuffleInternals. Internals of Spark SQL</p>
<pre>stateStoreProviderClass</pre> <p>spark.sql.streaming.stateStore.providerClass</p>	<p>Used when:</p> <ul style="list-style-type: none"> • StateStoreWriter, StateStoreCustomWriter or StateStoreWriter for the metrics are used • StateStoreConf is used
<pre>STREAMING_AGGREGATION_STATE_FORMAT_VERSION</pre> <p>spark.sql.streaming.aggregation.stateFormatVersion</p>	<p>Used when:</p> <ul style="list-style-type: none"> • StatefulAggregations execution plan is requested to be executed • OffsetSeqMetadata for the relevantSQLConf is used
<pre>STREAMING_CHECKPOINT_FILE_MANAGER_CLASS</pre> <p>spark.sql.streaming.checkpointFileManagerClass</p>	<p>Used exclusively when CheckpointFileManager is requested to create CheckpointFileManager.</p>

<code>streamingMetricsEnabled</code>	<code>spark.sql.streaming.metricsEnabled</code>	Used exclusively when <code>StreamExecution</code> is registered (to control register a metrics reporter for streaming query)
<code>STREAMING_MULTIPLE_WATERMARK_POLICY</code>	<code>spark.sql.streaming.multipleWatermarkPolicy</code>	
<code>streamingNoDataMicroBatchesEnabled</code>	<code>spark.sql.streaming.noDataMicroBatches.enabled</code>	Used exclusively when <code>MicroBatchExecution</code> execution engine is running an activated streaming query
<code>streamingNoDataProgressEventInterval</code>	<code>spark.sql.streaming.noDataProgressEventInterval</code>	Used exclusively for ProgressReporter
<code>streamingPollingDelay</code>	<code>spark.sql.streaming.pollingDelay</code>	Used exclusively when <code>StreamExecution</code> is created

Configuration Properties

Configuration properties are used to fine-tune Spark Structured Streaming applications.

You can set them for a `SparkSession` when it is created using `config` method.

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession
  .builder
  .config("spark.sql.streaming.metricsEnabled", true)
  .getOrCreate
```

Tip

Read up on [SparkSession](#) in [The Internals of Spark SQL](#) book.

Table 1. Structured Streaming's Properties

Name	Description
<code>spark.sql.streaming.aggregation.stateFormatVersion</code>	<p>(internal) Version of the state format.</p> <p>Default: 2</p> <p>Supported values:</p> <ul style="list-style-type: none"> 1 (for the legacy StreamingAggregation) 2 (for the default StreamingAggregation) <p>Used when StatefulAggregation planning strategy is executed on a streaming query with a <code>stateFormat</code> that boils down to creating a <code>StreamingAggregation</code> with the proper <i>implementation</i> of StreamingAggregation.</p> <p>Among the <code>checkpointingStrategy</code> values, the one supposed to be overridden by this setting has once been started from a checkpoint after a failure.</p>
<code>spark.sql.streaming.checkpointFileManagerClass</code>	<p>(internal) CheckpointFileManager class used to manage checkpoint files atomicity.</p> <p>Default: FileContextBasedCheckpointFileManager (with FileSystemBasedCheckpointFileManager as a fall-back case of unsupported file systems for metadata files).</p>

<code>spark.sql.streaming.checkpointLocation</code>	Default checkpoint directory Default: (empty)
<code>spark.sql.streaming.continuous.executorQueueSize</code>	(internal) The size (max number of partitions) of the queue used in continuous execution to store the results of a Continuous Query. Default: 1024
<code>spark.sql.streaming.continuous.executorPollIntervalMs</code>	(internal) The interval (in ms) between consecutive continuous execution runs. It is triggered when whether the epoch has finished or not. Default: 100 (ms)
<code>spark.sql.streaming.disabledV2MicroBatchReaders</code>	(internal) A comma-separated list of class names of data sources that do not support MicroBatchReadSupport . If these sources will fall back to BatchReadSupport . Default: (empty) Use SQLConf.disabledV2StreamingReaders to get the current value.
<code>spark.sql.streaming.fileSource.log.cleanupDelay</code>	(internal) How long (in minutes) a file needs to be visible for all readers to see it. Default: 10 (minutes) Use SQLConf.fileSourceLogCleanupDelay to get the current value.
<code>spark.sql.streaming.fileSource.log.compactInterval</code>	(internal) Number of log files to keep before previous files are compacted. Default: 10 Must be a positive value. Use SQLConf.fileSourceLogCompactInterval to get the current value.
<code>spark.sql.streaming.fileSource.log.deletion</code>	(internal) Whether to clean up log files for a file stream source. Default: true Use SQLConf.fileSourceLogDeletion to get the current value.

	<p>(internal) State format StateManager for FlatN physical operator</p> <p>Default: 2</p> <p>Supported values:</p> <ul style="list-style-type: none"> • 1 • 2 <p>Among the checkpoints supposed to be overridden has once been started from a checkpoint after</p>
	<p>(internal) The maximum number of batches will be retained in memory files.</p> <p>Default: 2</p> <p>Maximum count of versions implementation should</p> <p>The value adjusts a trade-off between usage vs cache miss:</p> <ul style="list-style-type: none"> • 2 covers both success cases • 1 covers only success cases • 0 or negative values maximize memory <p>Used exclusively when HDFSBackedStateStorePlugin.initialize.</p>
spark.sql.streaming.metricsEnabled	<p>Flag whether Dropwizard metrics are reported for active streams.</p> <p>Default: false</p> <p>Use SQLConf.streamMetricsEnabled current value</p>
spark.sql.streaming.minBatchesToRetain	<p>(internal) The minimum number of batches for failure recovery.</p> <p>Default: 100</p> <p>Use SQLConf.minBatchesToRetain current value</p>

	<p>Global watermark policy calculate the global watermark across multiple watermark query</p> <p>Default: <code>min</code></p> <p>Supported values:</p> <ul style="list-style-type: none"> • <code>min</code> - chooses the minimum watermark reported across multiple operators • <code>max</code> - chooses the maximum watermark reported across multiple operators <p>Cannot be changed before the same checkpoint location.</p>
<code>spark.sql.streaming.multipleWatermarkPolicy</code>	<p>Flag to control whether <code>engine</code> should execute the process for eager state streaming queries (true or false)</p> <p>Default: <code>true</code></p> <p>Use <code>SQLConf.streamingNoDataMicroBatchesEnabled</code> to get the current value</p>
<code>spark.sql.streaming.noDataProgressEventInterval</code>	<p>(internal) How long to wait for events when there is no data. <code>ProgressReporter</code> is responsible for reporting progress.</p> <p>Default: <code>10000L</code></p> <p>Use <code>SQLConf.streamingNoDataProgressEventInterval</code> to get the current value</p>
<code>spark.sql.streaming.numRecentProgressUpdates</code>	<p>Number of <code>progress update</code> to consider for streaming query</p> <p>Default: <code>100</code></p>
<code>spark.sql.streaming.pollingDelay</code>	<p>(internal) How long (in milliseconds) to wait for StreamExecution before <code>no data was available</code> in the stream.</p> <p>Default: <code>10</code> (milliseconds)</p>
<code>spark.sql.streaming.stateStore.maintenanceInterval</code>	<p>The initial delay and how often StateStore's <code>maintenance</code> task runs.</p> <p>Default: <code>60s</code></p>

spark.sql.streaming.stateStore.providerClass	(internal) The fully-qualified class name of the StateStoreProvider implementation to use for state data in stateful streaming operations. It must have a zero-arg constructor. Default: HDFSBackedStateStore Use SQLConf.stateStoreProviderClass to get the current value.
spark.sql.streaming.unsupportedOperationCheck	(internal) When enabled, this configuration will cause the StreamingQueryManager to check the plan of a streaming query for unsupported operations only. Default: true

StreamingQueryListener — Intercepting Streaming Events

`StreamingQueryListener` is the [contract](#) for listeners that want to be notified about the [life cycle events](#) of streaming queries, i.e. start, progress and termination of a query.

```
package org.apache.spark.sql.streaming

abstract class StreamingQueryListener {
  def onQueryStarted(event: QueryStartedEvent): Unit
  def onQueryProgress(event: QueryProgressEvent): Unit
  def onQueryTerminated(event: QueryTerminatedEvent): Unit
}
```

Table 1. StreamingQueryListener’s Life Cycle Events and Callbacks

Event	Callback	When Posted
<code>QueryStartedEvent</code> <ul style="list-style-type: none"> • <code>id</code> • <code>runId</code> • <code>name</code> 	<code>onQueryStarted</code>	Right after <code>StreamExecution</code> has started running streaming batches .
<code>QueryProgressEvent</code> <ul style="list-style-type: none"> • StreamingQueryProgress 	<code>onQueryProgress</code>	ProgressReporter reports query progress (which is when <code>StreamExecution</code> runs batches and a trigger has finished).
<code>QueryTerminatedEvent</code> <ul style="list-style-type: none"> • <code>id</code> • <code>runId</code> • <code>Optional exception</code> if terminated due to an error 	<code>onQueryTerminated</code>	Right before <code>StreamExecution</code> finishes running streaming batches (due to a stop or an exception).

You can register a `StreamingQueryListener` using `StreamingQueryManager.addListener` method.

```
val queryListener: StreamingQueryListener = ...
spark.streams.addListener(queryListener)
```

You can remove a `StreamingQueryListener` using `StreamingQueryManager.removeListener` method.

```
val queryListener: StreamingQueryListener = ...
spark.streams.removeListener(queryListener)
```

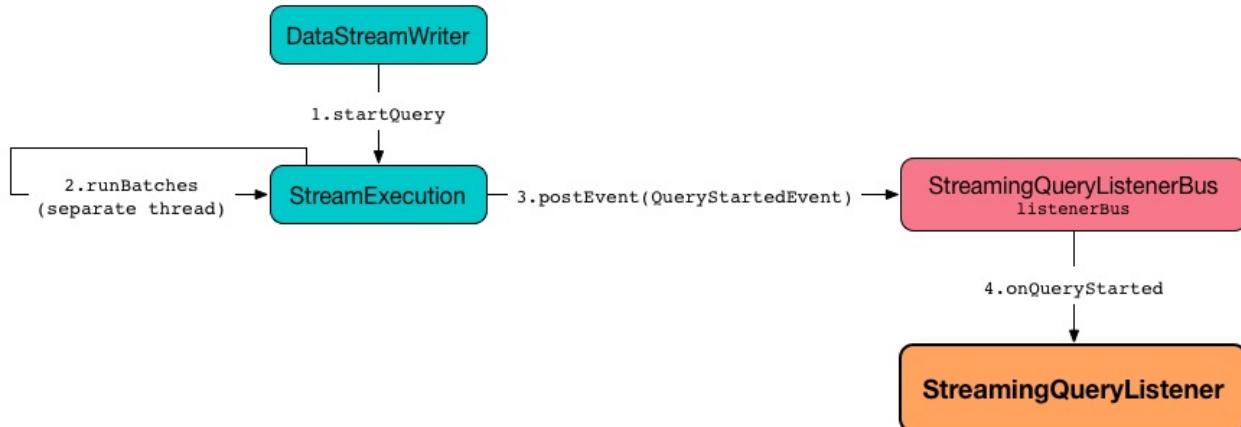


Figure 1. `StreamingQueryListener` Notified about Query's Start (`onQueryStarted`)

Note

`onQueryStarted` is used internally to unblock the `starting thread` of `StreamExecution`.

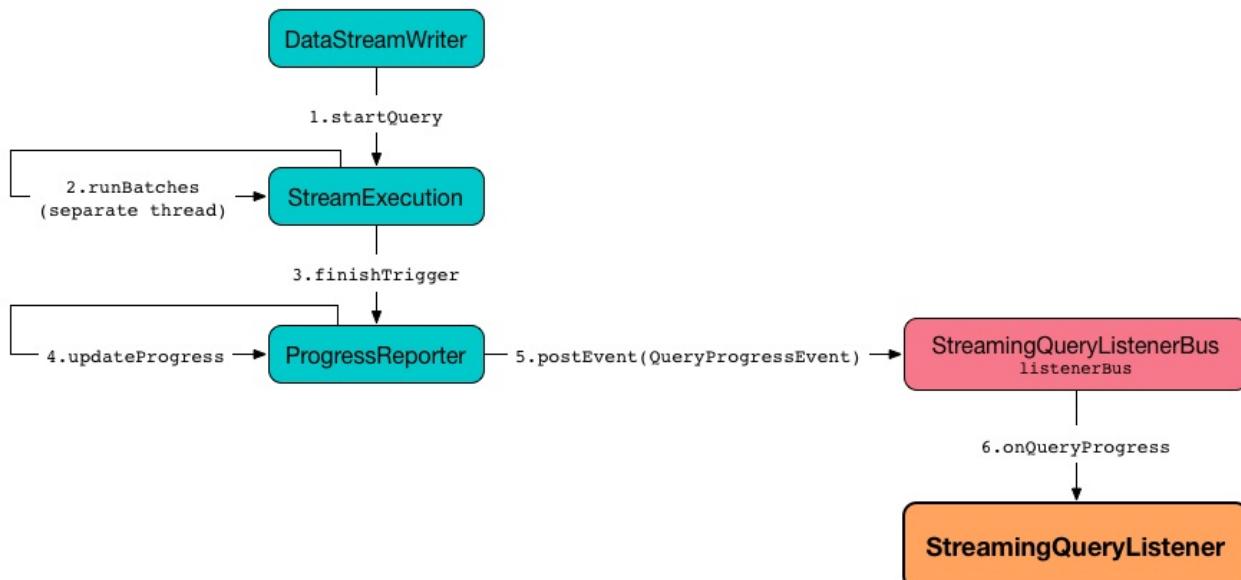


Figure 2. `StreamingQueryListener` Notified about Query's Progress (`onQueryProgress`)



Figure 3. `StreamingQueryListener` Notified about Query's Termination (`onQueryTerminated`)

Note	You can also register a streaming event listener using the general <code>SparkListener</code> interface. Details on <code>SparkListener</code> interface can be found in the Mastering Apache Spark 2 gitbook.
------	--

StreamingQueryProgress

`StreamingQueryProgress` holds information about the progress of a streaming query.

`StreamingQueryProgress` is created exclusively when `StreamExecution` finishes a trigger.

Note

Use `lastProgress` property of a `StreamingQuery` to access the most recent `StreamingQueryProgress` update.

```
val sq: StreamingQuery = ...
sq.lastProgress
```

Note

Use `recentProgress` property of a `StreamingQuery` to access the most recent `StreamingQueryProgress` updates.

```
val sq: StreamingQuery = ...
sq.recentProgress
```

Note

Use `StreamingQueryListener` to get notified about `StreamingQueryProgress` updates while a streaming query is executed.

Table 1. StreamingQueryProgress's Properties

Name	Description
id	Unique identifier of a streaming query
runId	Unique identifier of the current execution of a streaming query
name	Optional query name
timestamp	Time when the trigger has started (in ISO8601 format).
batchId	Unique id of the current batch
durationMs	Durations of the internal phases (in milliseconds)
eventTime	Statistics of event time seen in this batch
stateOperators	Information about stateful operators in the query that store state.
sources	Statistics about the data read from every streaming source in a streaming query
sink	Information about progress made for a sink

MetricsReporter

MetricsReporter is...FIXME

ProgressReporter Contract

`ProgressReporter` is the [contract](#) of stream execution progress reporters that report the statistics of a streaming query.

Table 1. ProgressReporter Contract

Method	Description
<code>currentBatchId</code>	<pre>currentBatchId: Long</pre> <p><code>Id of the current streaming micro-batch</code></p>
<code>id</code>	<pre>id: UUID</pre> <p>Universally unique identifier (UUID) of the streaming query (that stays unchanged between restarts)</p>
<code>lastExecution</code>	<pre>lastExecution: QueryExecution</pre> <p><code>QueryExecution</code> of the streaming query</p>
<code>logicalPlan</code>	<pre>logicalPlan: LogicalPlan</pre> <p>The logical query plan of the streaming query Used when <code>ProgressReporter</code> is requested for the following:</p> <ul style="list-style-type: none"> • extract statistics from the most recent query execution (to add <code>watermark</code> metric when a streaming watermark is used) • extractSourceToNumInputRows
<code>name</code>	<pre>name: String</pre> <p>Name of the streaming query</p>
<code>newData</code>	<pre>newData: Map[BaseStreamingSource, LogicalPlan]</pre> <p>Streaming sources with the new data (as a <code>LogicalPlan</code>) Used when:</p>

	<ul style="list-style-type: none"> • ProgressReporter extracts statistics from the most recent query execution (to calculate the so-called <code>inputRows</code>)
<code>offsetSeqMetadata</code>	<pre>offsetSeqMetadata: OffsetSeqMetadata</pre> <p>OffsetSeqMetadata (with the current micro-batch event-time watermark and timestamp)</p>
<code>postEvent</code>	<pre>postEvent(event: StreamingQueryListener.Event): Unit</pre> <p>Posts StreamingQueryListener.Event</p>
<code>runId</code>	<pre>runId: UUID</pre> <p>Universally unique identifier (UUID) of the single run of the streaming query (that changes every restart)</p>
<code>sink</code>	<pre>sink: BaseStreamingSink</pre> <p>The one and only streaming sink of the streaming query</p>
<code>sources</code>	<pre>sources: Seq[BaseStreamingSource]</pre> <p>Streaming sources of the streaming query Used when finishing a trigger (and updating progress and marking current status as trigger inactive)</p>
<code>sparkSession</code>	<pre>sparkSession: SparkSession</pre> <p>SparkSession of the streaming query</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> Tip Read up on SparkSession in The Internals of Spark SQL book. </div>
<code>triggerClock</code>	<pre>triggerClock: Clock</pre> <p>Clock of the streaming query</p>

Note	<p><code>StreamExecution</code> is the one and only known direct extension of the ProgressReporter Contract in Spark Structured Streaming.</p>
------	--

`ProgressReporter` uses the `spark.sql.streaming.noDataProgressEventInterval` configuration property to control how long to wait between two progress events when there is no data (default: `10000L`) when [finishing trigger](#).

`ProgressReporter` uses **yyyy-MM-dd'T'HH:mm:ss.SSS'Z'** time format (with **UTC** timezone).

```

import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._
val sampleQuery = spark
  .readStream
  .format("rate")
  .load
  .writeStream
  .format("console")
  .option("truncate", false)
  .trigger(Trigger.ProcessingTime(10.seconds))
  .start

// Using public API
import org.apache.spark.sql.streaming.SourceProgress
scala> sampleQuery.
|   lastProgress.
|   sources.
|   map { case sp: SourceProgress =>
|     s"source = ${sp.description} => endOffset = ${sp.endOffset}" }.
|   foreach(println)
source = RateSource[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8] => endOffset
t = 663

scala> println(sampleQuery.lastProgress.sources(0))
res40: org.apache.spark.sql.streaming.SourceProgress =
{
  "description" : "RateSource[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8]",
  "startOffset" : 333,
  "endOffset" : 343,
  "numInputRows" : 10,
  "inputRowsPerSecond" : 0.9998000399920015,
  "processedRowsPerSecond" : 200.0
}

// With a hack
import org.apache.spark.sql.execution.streaming.StreamingQueryWrapper
val offsets = sampleQuery.
  asInstanceOf[StreamingQueryWrapper].
  streamingQuery.
  availableOffsets.
  map { case (source, offset) =>
    s"source = $source => offset = $offset" }
scala> offsets.foreach(println)
source = RateSource[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8] => offset =
293

```

status Method

```
status: StreamingQueryStatus
```

`status` gives the [current StreamingQueryStatus](#).

Note

`status` is used when `StreamingQueryWrapper` is requested for the current status of a streaming query (that is part of [StreamingQuery Contract](#)).

Updating Progress — `updateProgress` Internal Method

```
updateProgress(newProgress: StreamingQueryProgress): Unit
```

`updateProgress` records the input `newProgress` and posts a [QueryProgressEvent](#) event.

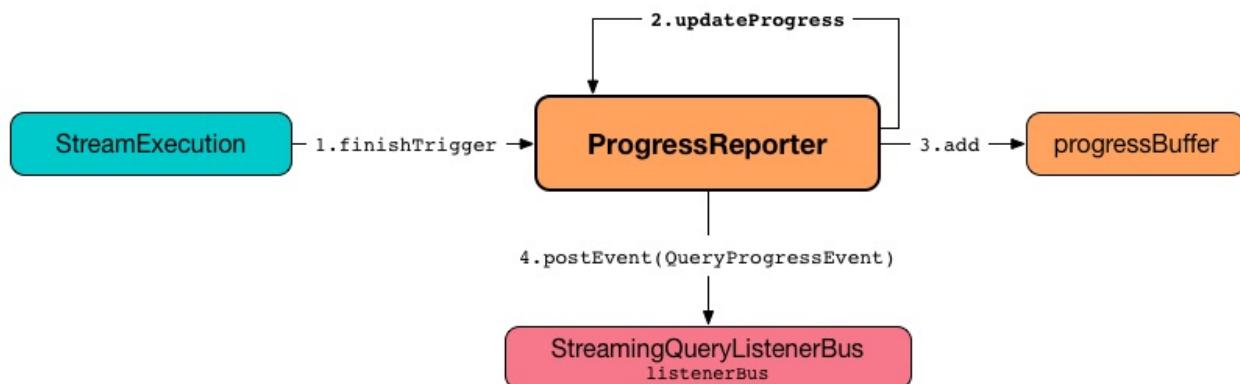


Figure 1. ProgressReporter's Reporting Query Progress

`updateProgress` adds the input `newProgress` to [progressBuffer](#).

`updateProgress` removes elements from `progressBuffer` if their number is or exceeds the value of `spark.sql.streaming.numRecentProgressUpdates` property.

`updateProgress` posts a [QueryProgressEvent](#) (with the input `newProgress`).

`updateProgress` prints out the following INFO message to the logs:

```
Streaming query made progress: [newProgress]
```

Note

`updateProgress` synchronizes concurrent access to the `progressBuffer` internal registry.

Note

`updateProgress` is used exclusively when `ProgressReporter` finishes a trigger.

Initializing Query Progress for New Trigger — `startTrigger` Method

```
startTrigger(): Unit
```

`startTrigger` prints out the following DEBUG message to the logs:

```
Starting Trigger Calculation
```

Table 2. `startTrigger`'s Internal Registry Changes For New Trigger

Registry	New Value
<code>lastTriggerStartTimestamp</code>	<code>currentTriggerStartTimestamp</code>
<code>currentTriggerStartTimestamp</code>	Requests the <code>trigger clock</code> for the current timestamp (in millis)
<code>currentStatus</code>	Enables (<code>true</code>) the <code>isTriggerActive</code> flag of the <code>currentStatus</code>
<code>currentTriggerStartOffsets</code>	<code>null</code>
<code>currentTriggerEndOffsets</code>	<code>null</code>
<code>currentDurationsMs</code>	Clears the <code>currentDurationsMs</code>

Note

`startTrigger` is used exclusively when `StreamExecution` starts running batches (as part of `TriggerExecutor` executing a batch runner).

Finalizing Query Progress for Trigger— `finishTrigger` Method

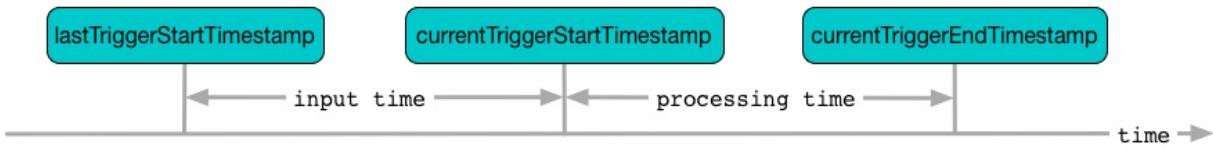
```
finishTrigger(hasNewData: Boolean): Unit
```

Internally, `finishTrigger` sets `currentTriggerEndTimestamp` to the current time (using `triggerClock`).

`finishTrigger` extractExecutionStats.

`finishTrigger` calculates the **processing time** (in seconds) as the difference between the end and `start` timestamps.

`finishTrigger` calculates the **input time** (in seconds) as the difference between the start time of the `current` and `last` triggers.

Figure 2. ProgressReporter's `finishTrigger` and Timestamps

`finishTrigger` prints out the following DEBUG message to the logs:

```
Execution stats: [executionStats]
```

`finishTrigger` creates a [SourceProgress](#) (aka source statistics) for [every source used](#).

`finishTrigger` creates a [SinkProgress](#) (aka sink statistics) for the [sink](#).

`finishTrigger` creates a [StreamingQueryProgress](#).

If there was any data (using the input `hasNewData` flag), `finishTrigger` resets `lastNoDataProgressEventTime` (i.e. becomes the minimum possible time) and updates query progress.

Otherwise, when no data was available (using the input `hasNewData` flag), `finishTrigger` updates query progress only when `lastNoDataProgressEventTime` passed.

In the end, `finishTrigger` disables `isTriggerActive` flag of [StreamingQueryStatus](#) (i.e. sets it to `false`).

Note	<code>finishTrigger</code> is used exclusively when <code>MicroBatchExecution</code> is requested to run the activated streaming query.
------	---

Reporting Execution Time — `reportTimeTaken` Method

```
reportTimeTaken[T](triggerDetailKey: String)(body: => T): T
```

`reportTimeTaken` measures the time to execute `body` and records it in the [currentDurationsMs](#) internal registry.

In the end, `reportTimeTaken` prints out the following DEBUG message to the logs and returns the result of executing `body`.

```
[triggerDetailKey] took [time] ms
```

	<p><code>reportTimeTaken</code> is used when <code>streamExecution</code> wants to record the time taken for (as <code>triggerDetailKey</code> in the DEBUG message above):</p> <ul style="list-style-type: none"> • <code>addBatch</code> • <code>getBatch</code> • <code>getOffset</code> • <code>queryPlanning</code> • <code>triggerExecution</code> • <code>walCommit</code> when writing offsets to log
Note	

Updating Status Message — `updateStatusMessage` Method

```
updateStatusMessage(message: String): Unit
```

`updateStatusMessage` simply updates the `message` in the `StreamingQueryStatus` internal registry.

	<p><code>updateStatusMessage</code> is used when:</p> <ul style="list-style-type: none"> • <code>StreamExecution</code> is requested to run stream processing • <code>MicroBatchExecution</code> is requested to run an activated streaming query, construct the next streaming micro-batch
Note	

`extractSourceToNumInputRows` Internal Method

```
extractSourceToNumInputRows(): Map[BaseStreamingSource, Long]
```

`extractSourceToNumInputRows` ...FIXME

	<p><code>extractSourceToNumInputRows</code> is used exclusively when <code>ProgressReporter</code> is requested to extractExecutionStats.</p>
Note	

Extracting Execution Statistics — `extractExecutionStats` Internal Method

```
extractExecutionStats(hasNewData: Boolean): ExecutionStats
```

```
extractExecutionStats ...FIXME
```

Note

`extractExecutionStats` is used exclusively when `ProgressReporter` is requested to [finishTrigger](#).

extractStateOperatorMetrics Internal Method

```
extractStateOperatorMetrics(hasNewData: Boolean): Seq[StateOperatorProgress]
```

```
extractStateOperatorMetrics ...FIXME
```

Note

`extractStateOperatorMetrics` is used exclusively when `ProgressReporter` is requested to [extractExecutionStats](#).

formatTimestamp Internal Method

```
formatTimestamp(millis: Long): String
```

```
formatTimestamp ...FIXME
```

Note

`formatTimestamp` is used when...FIXME

Recording Trigger Offsets (StreamProgress) — recordTriggerOffsets Method

```
recordTriggerOffsets(  
  from: StreamProgress,  
  to: StreamProgress): Unit
```

`recordTriggerOffsets` simply sets (*records*) the `currentTriggerStartOffsets` and `currentTriggerEndOffsets` internal registries to the `json` representations of the `from` and `to` `StreamProgresses`.

Note

`recordTriggerOffsets` is used when:

- `MicroBatchExecution` is requested to [run the activated streaming query](#)
- `ContinuousExecution` is requested to [commit an epoch](#)

lastProgress Method

```
lastProgress: StreamingQueryProgress
```

`lastProgress` ...FIXME

Note	<code>lastProgress</code> is used when...FIXME
------	--

recentProgress Method

```
recentProgress: Array[StreamingQueryProgress]
```

`recentProgress` ...FIXME

Note	<code>recentProgress</code> is used when...FIXME
------	--

Internal Properties

Name	Description
	<p><code>scala.collection.mutable.HashMap</code> of action names (aka <i>triggerDetailKey</i>) and their cumulative times (in milliseconds).</p> <p>The action names can be as follows:</p> <ul style="list-style-type: none"> • <code>addBatch</code> • <code>getBatch</code> (when <code>StreamExecution</code> runs a streaming batch) • <code>getOffset</code> • <code>queryPlanning</code> • <code>triggerExecution</code> • <code>walCommit</code> when writing offsets to log <p>Starts empty when <code>ProgressReporter</code> sets the state for a new batch with new entries added or updated when reporting execution time (of an action).</p> <p><code>currentDurationsMs</code></p>

	<p>You can see the current value of <code>currentDurationsMs</code> in progress reports under <code>durationMs</code>.</p> <p><code>scala> query.lastProgress.durationMs</code> <code>res3: java.util.Map[String,Long] =</code> <code>{triggerExecution=60,</code> <code>queryPlanning=1, getBatch=5,</code> <code>getOffset=0, addBatch=30,</code> <code>walCommit=23}</code></p>
<code>currentStatus</code>	<p>StreamingQueryStatus with the current status of the streaming query</p> <p>Available using <code>status</code> method</p> <ul style="list-style-type: none"> • message updated with <code>updateStatusMessage</code>
<code>currentTriggerEndOffsets</code>	
<code>currentTriggerEndTimestamp</code>	<p>Timestamp of when the current batch/trigger has ended</p> <p>Default: <code>-1L</code></p>
<code>currentTriggerStartOffsets</code>	
<code>currentTriggerStartTimestamp</code>	<p>Timestamp of when the current batch/trigger has started</p> <p>Default: <code>-1L</code></p>
<code>lastNoDataProgressEventTime</code>	<p>Default: <code>Long.MinValue</code></p>
<code>lastTriggerStartTimestamp</code>	<p>Timestamp of when the last batch/trigger started</p> <p>Default: <code>-1L</code></p>
<code>metricWarningLogged</code>	<p>Flag to...FIXME</p> <p>Default: <code>false</code></p>

	<p><code>scala.collection.mutable.Queue</code> of <code>StreamingQueryProgresses</code></p>
<code>progressBuffer</code>	<p>Elements are added and removed when <code>ProgressReporter</code> is requested to update progress.</p> <p>Used when <code>ProgressReporter</code> is requested for the lastProgress and recentProgress</p>

ExecutionStats

ExecutionStats is...FIXME

StreamingQueryStatus

StreamingQueryStatus is...FIXME

SourceProgress

SourceProgress is...FIXME

SinkProgress

SinkProgress is...FIXME

Web UI

Web UI...FIXME

Caution

FIXME What's visible on the plan diagram in the SQL tab of the UI

Logging

Caution	FIXME
---------	-------

DataSource — Pluggable Data Source

`DataSource` is...FIXME

`DataSource` is [created](#) when...FIXME

Tip	Read DataSource — Pluggable Data Sources (for Spark SQL's batch structured queries).
-----	--

Table 1. DataSource's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>providingClass</code>	<code>java.lang.Class</code> that corresponds to the <code>className</code> (that can be a fully-qualified class name or an alias of the data source)
<code>sourceInfo</code>	<p><code>SourceInfo</code> with the name, the schema, and optional partitioning columns of a source.</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>DataSource</code> creates a FileStreamSource (that requires the schema and the optional partitioning columns) • <code>StreamingRelation</code> is created (for a <code>DataSource</code>)

Describing Name and Schema of Streaming Source — `sourceSchema` Internal Method

`sourceSchema(): SourceInfo`

`sourceSchema` ...FIXME

Note	<code>sourceSchema</code> is used exclusively when <code>DataSource</code> is requested for the SourceInfo .
------	--

Creating DataSource Instance

`DataSource` takes the following when created:

- `SparkSession`
- `className`, i.e. the fully-qualified class name or an alias of the data source
- Paths (default: `Nil`, i.e. an empty collection)

- Optional user-defined schema (default: `None`)
- Names of the partition columns (default: empty)
- Optional `BucketSpec` (default: `None`)
- Configuration options (default: empty)
- Optional `CatalogTable` (default: `None`)

`DataSource` initializes the [internal registries and counters](#).

Creating Streaming Source — `createSource` Method

```
createSource(metadataPath: String): Source
```

`createSource` ...FIXME

Note	<code>createSource</code> is used exclusively when <code>MicroBatchExecution</code> is requested to initialize the analyzed logical plan.
------	---

Creating Streaming Sink — `createSink` Method

```
createSink(outputMode: OutputMode): Sink
```

`createSink` creates a [streaming sink](#) for [StreamSinkProvider](#) or [FileFormat](#) data sources.

Tip	Read up on FileFormat Data Source in The Internals of Spark SQL book.
-----	---

Internally, `createSink` creates a new instance of the `providerClass` and branches off per type:

- For a `StreamSinkProvider`, `createSink` simply delegates the call and requests it to [create a streaming sink](#)
- For a `FileFormat`, `createSink` creates a `FileStreamSink` when `path` option is specified and the output mode is [Append](#)

`createSink` throws a `IllegalArgumentException` when `path` option is not specified for a `FileFormat` data source:

```
'path' is not specified
```

`createSink` throws an `AnalysisException` when the given `OutputMode` is different from `Append` for a `FileFormat` data source:

```
Data source [className] does not support [outputMode] output mode
```

`createSink` throws an `UnsupportedOperationException` for unsupported data source formats:

```
Data source [className] does not support streamed writing
```

Note

`createSink` is used exclusively when `DataStreamWriter` is requested to `create` and start a streaming query.

BaseStreamingSource Contract — Base of Streaming Sources

`BaseStreamingSource` is the abstraction of streaming sources and readers that can be stopped.

The main purpose of `BaseStreamingSource` is to share a common abstraction between the former Data Source API V1 and the modern Data Source API V2 (before Spark Structured Streaming migrates to the Data Source API V2 fully).

Table 1. BaseStreamingSource Contract

Method	Description
<code>stop</code>	<pre>void stop()</pre> <p>Stops the streaming source or reader (and frees any resources it may have allocated)</p> <p>Used when:</p> <ul style="list-style-type: none"> <code>StreamExecution</code> is requested to stop streaming sources and readers <code>DataStreamReader</code> is requested to load data from a MicroBatchReadSupport data source (for read schema)

Table 2. BaseStreamingSources (Direct Implementations and Extensions Only)

BaseStreamingSource	Description
<code>ContinuousReader</code>	Data source readers in Continuous Stream Processing (based on Data Source API V2)
<code>MemoryStreamBase</code>	Base implementation of ContinuousMemoryStream (for Continuous Stream Processing) and MemoryStream (for Micro-Batch Stream Processing)
<code>MicroBatchReader</code>	Data source readers in Micro-Batch Stream Processing (based on Data Source API V2)
<code>Source</code>	Streaming data sources (based on Data Source API V1)

BaseStreamingSink Contract

BaseStreamingSink is...FIXME

Source Contract — Streaming Sources

Streaming Source is a "continuous" stream of data and is described using the [Source Contract](#).

`Source` can generate a streaming DataFrame (aka **batch**) given start and end offsets in a batch.

For fault tolerance, `Source` must be able to replay data given a start offset.

`Source` should be able to replay an arbitrary sequence of past data in a stream using a range of offsets. Streaming sources like Apache Kafka and Amazon Kinesis (with their per-record offsets) fit into this model nicely. This is the assumption so structured streaming can achieve end-to-end exactly-once guarantees.

Table 1. Sources

Format	Source
Any <code>FileFormat</code> <ul style="list-style-type: none"> • <code>csv</code> • <code>hive</code> • <code>json</code> • <code>libsvm</code> • <code>orc</code> • <code>parquet</code> • <code>text</code> 	FileStreamSource
<code>kafka</code>	KafkaSource
<code>memory</code>	MemoryStream
<code>rate</code>	RateStreamSource
<code>socket</code>	TextSocketSource

Source Contract

```

package org.apache.spark.sql.execution.streaming

trait Source {
  def commit(end: Offset) : Unit = {}
  def getBatch(start: Option[Offset], end: Offset): DataFrame
  def getOffset: Option[Offset]
  def schema: StructType
  def stop(): Unit
}

```

Table 2. Source Contract

Method	Description
getBatch	<p>Generates a <code>DataFrame</code> (with new rows) for a given batch (described using the optional start and end offsets).</p> <p>Used when <code>StreamExecution</code> runs a batch and <code>populateStartOffsets</code>.</p>
getOffset	<p><code>getOffset: Option[Offset]</code></p> <p>Latest (maximum available) <code>offset</code> (or <code>None</code> when never received any data)</p> <p>Used exclusively when <code>MicroBatchExecution</code> stream execution engine is requested to construct a next streaming micro-batch</p>
schema	<p>Schema of the data from this source</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>KafkaSource</code> generates a <code>DataFrame</code> with records from Kafka for a streaming batch • <code>FileStreamSource</code> generates a <code>DataFrame</code> for a streaming batch • <code>RateFileStreamSource</code> generates a <code>DataFrame</code> for a streaming batch • <code>StreamingExecutionRelation</code> is created (for <code>MemoryStream</code>)

StreamSourceProvider — Streaming Data Source Provider

`StreamSourceProvider` is the contract of data source providers that can [create a streaming data source](#) for a format (e.g. text file) or system (e.g. Apache Kafka).

Table 1. StreamSourceProvider Contract

Method	Description		
<code>createSource</code>	<pre>createSource(sqlContext: SQLContext, metadataPath: String, schema: Option[StructType], providerName: String, parameters: Map[String, String]): Source</pre> <p>Creates a streaming source for a format or system (to continually read data)</p> <table border="1"> <tr> <td>Note</td><td><code>metadataPath</code> is the value of the optional user-specified <code>checkpointLocation</code> option or resolved by StreamingQueryManager.</td></tr> </table> <p>Used exclusively when <code>DataSource</code> is requested to create a streaming source (when <code>MicroBatchExecution</code> is requested to initialize the analyzed logical plan)</p>	Note	<code>metadataPath</code> is the value of the optional user-specified <code>checkpointLocation</code> option or resolved by StreamingQueryManager .
Note	<code>metadataPath</code> is the value of the optional user-specified <code>checkpointLocation</code> option or resolved by StreamingQueryManager .		
<code>sourceSchema</code>	<pre>sourceSchema(sqlContext: SQLContext, schema: Option[StructType], providerName: String, parameters: Map[String, String]): (String, StructType)</pre> <p>Describes a streaming source with a name and the schema</p> <p>Used exclusively when <code>DataSource</code> is requested to describe the name and the schema of a streaming source (when <code>MicroBatchExecution</code> is requested to initialize the analyzed logical plan)</p>		
<p>Note <code>StreamSourceProvider</code> is an experimental contract.</p>			
<p>Note KafkaSourceProvider is the default implementation of the StreamSourceProvider Contract in Spark Structured Streaming.</p>			

Streaming Sink — Adding Batches of Data to Storage

`Sink` is the contract for **streaming writes**, i.e. adding batches to an output every trigger.

Note	<code>sink</code> is part of the so-called Structured Streaming V1 that is currently being rewritten to StreamWriteSupport in V2.
------	--

`Sink` is a single-method interface with `addBatch` method.

```
package org.apache.spark.sql.execution.streaming

trait Sink {
  def addBatch(batchId: Long, data: DataFrame): Unit
}
```

`addBatch` is used to "add" a batch of data to the sink (for `batchId` batch).

`addBatch` is used when `StreamExecution` runs a batch.

Table 1. Sinks

Format / Operator	Sink
<code>console</code>	
<code>Any FileFormat</code> <ul style="list-style-type: none"> • <code>csv</code> • <code>hive</code> • <code>json</code> • <code>libsvm</code> • <code>orc</code> • <code>parquet</code> • <code>text</code> 	<code>FileStreamSink</code>
<code>foreach</code> operator	<code>ForeachSink</code>
<code>kafka</code>	<code>KafkaSink</code>
<code>memory</code>	<code>MemorySink</code>

Tip

You can create your own streaming format implementing [StreamSinkProvider](#).

When creating a custom `Sink` it is recommended to accept the options (e.g. `Map[String, String]`) that the `DataStreamWriter` [was configured with](#). You can then use the options to fine-tune the write path.

```
class HighPerfSink(options: Map[String, String]) extends Sink {
  override def addBatch(batchId: Long, data: DataFrame): Unit = {
    val bucketName = options.get("bucket").orNull
    ...
  }
}
```

StreamSinkProvider

`StreamSinkProvider` is the [contract](#) for creating [streaming sinks](#) for a specific format or system.

`StreamSinkProvider` defines the one and only [createSink](#) method that creates a [streaming sink](#).

```
package org.apache.spark.sql.sources

trait StreamSinkProvider {
  def createSink(
    sqlContext: SQLContext,
    parameters: Map[String, String],
    partitionColumns: Seq[String],
    outputMode: OutputMode): Sink
}
```

StreamWriteSupport Contract — Writable Streaming Data Sources

`StreamWriteSupport` is the abstraction of `DataSourceV2` sinks that [create StreamWriters](#) for streaming write (when used in streaming queries in [MicroBatchExecution](#) and [ContinuousExecution](#)).

```
StreamWriter createStreamWriter(
    String queryId,
    StructType schema,
    OutputMode mode,
    DataSourceOptions options)
```

`createStreamWriter` creates a [StreamWriter](#) for streaming write and is used when the [stream execution thread for a streaming query](#) is [started](#) and requests the stream execution engines to start, i.e.

- `ContinuousExecution` is requested to [runContinuous](#)
- `MicroBatchExecution` is requested to [run a single streaming batch](#)

Table 1. StreamWriteSupports

StreamWriteSupport	Description
ConsoleSinkProvider	Streaming sink for <code>console</code> data source format
ForeachWriterProvider	
KafkaSourceProvider	
MemorySinkV2	

StreamWriter Contract

`StreamWriter` is the [extension](#) of the `DataSourceWriter` contract to support epochs, i.e. [streaming writers](#) that can [abort](#) and [commit](#) writing jobs for a specified epoch.

Tip

Read up on [DataSourceWriter](#) in [The Internals of Spark SQL](#) book.

Table 1. StreamWriter Contract

Method	Description
abort	<pre>void abort(long epochId, WriterCommitMessage[] messages)</pre> <p>Aborts the writing job for a specified <code>epochId</code> and <code>WriterCommitMessages</code></p> <p>Used exclusively when <code>MicroBatchWriter</code> is requested to abort</p>
commit	<pre>void commit(long epochId, WriterCommitMessage[] messages)</pre> <p>Commits the writing job for a specified <code>epochId</code> and <code>WriterCommitMessages</code></p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>EpochCoordinator</code> is requested to commitEpoch • <code>MicroBatchWriter</code> is requested to commit

Table 2. StreamWriters

StreamWriter	Description
ForeachWriterProvider	foreachWriter data source
ConsoleWriter	console data source
KafkaStreamWriter	kafka data source
MemoryStreamWriter	memory data source

FileStreamSource

`FileStreamSource` is a [Source](#) that reads text files from `path` directory as they appear. It uses `LongOffset` offsets.

Note

It is used by [DataSource.createSource](#) for `FileFormat`.

You can provide the `schema` of the data and `dataFrameBuilder` - the function to build a `DataFrame` in [getBatch](#) at instantiation time.

```
// NOTE The source directory must exist
// mkdir text-logs

val df = spark.readStream
  .format("text")
  .option("maxFilesPerTrigger", 1)
  .load("text-logs")

scala> df.printSchema
root
 |-- value: string (nullable = true)
```

Batches are indexed.

It lives in `org.apache.spark.sql.execution.streaming` package.

```
import org.apache.spark.sql.types._
val schema = StructType(
  StructField("id", LongType, nullable = false) ::
  StructField("name", StringType, nullable = false) ::
  StructField("score", DoubleType, nullable = false) :: Nil)

// You should have input-json directory available
val in = spark.readStream
  .format("json")
  .schema(schema)
  .load("input-json")

val input = in.transform { ds =>
  println("transform executed") // <-- it's going to be executed once only
  ds
}

scala> input.isStreaming
res9: Boolean = true
```

It tracks already-processed files in `seenFiles` hash map.

Tip Enable `DEBUG` or `TRACE` logging level for `org.apache.spark.sql.execution.streaming.FileStreamSource` to see what happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.execution.streaming.FileStreamSource=TRACE
```

Refer to [Logging](#).

Creating FileStreamSource Instance

Caution

FIXME

Options

maxFilesPerTrigger

`maxFilesPerTrigger` option specifies the maximum number of files per trigger (batch). It limits the file stream source to read the `maxFilesPerTrigger` number of files specified at a time and hence enables rate limiting.

It allows for a static set of files be used like a stream for testing as the file set is processed `maxFilesPerTrigger` number of files at a time.

schema

If the schema is specified at instantiation time (using optional `dataSchema` constructor parameter) it is returned.

Otherwise, `fetchAllFiles` internal method is called to list all the files in a directory.

When there is at least one file the schema is calculated using `dataFrameBuilder` constructor parameter function. Else, an `IllegalArgumentException("No schema specified")` is thrown unless it is for `text` provider (as `providerName` constructor parameter) where the default schema with a single `value` column of type `StringType` is assumed.

Note

text as the value of `providerName` constructor parameter denotes **text file stream provider**.

getOffset Method

```
getOffset: Option[Offset]
```

Note

`getOffset` is part of the [Source Contract](#) to find the latest `offset`.

`getOffset ...FIXME`

The maximum offset (`getOffset`) is calculated by fetching all the files in `path` excluding files that start with `_` (underscore).

When computing the maximum offset using `getOffset`, you should see the following DEBUG message in the logs:

```
Listed ${files.size} in ${(endTime.toDouble - startTime) / 1000000}ms
```

When computing the maximum offset using `getOffset`, it also filters out the files that were already seen (tracked in `seenFiles` internal registry).

You should see the following DEBUG message in the logs (depending on the status of a file):

```
new file: $file
// or
old file: $file
```

Generating DataFrame for Streaming Batch — `getBatch` Method

`FileStreamSource.getBatch` asks [metadataLog](#) for the batch.

You should see the following INFO and DEBUG messages in the logs:

```
INFO Processing ${files.length} files from ${startId + 1}:$endId
DEBUG Streaming ${files.mkString(", ")}
```

The method to create a result batch is given at instantiation time (as `dataFrameBuilder` constructor parameter).

metadataLog

`metadataLog` is a metadata storage using `metadataPath` path (which is a constructor parameter).

Note	It extends <code>HDFSMetadataLog[Seq[String]]</code> .
------	--

Caution	<code>FIXME Review</code> <code>HDFSMetadataLog</code>
---------	--

fetchMaxOffset Internal Method

```
fetchMaxOffset(): FileStreamSourceOffset
```

`fetchMaxOffset` ...`FIXME`

Note	<code>fetchMaxOffset</code> is used exclusively when <code>FileStreamSource</code> is requested to getOffset .
------	--

fetchAllFiles Internal Method

```
fetchAllFiles(): Seq[(String, Long)]
```

`fetchAllFiles` ...`FIXME`

Note	<code>fetchAllFiles</code> is used exclusively when <code>FileStreamSource</code> is requested to fetchMaxOffset .
------	--

allFilesUsingMetadataLogFileIndex Internal Method

```
allFilesUsingMetadataLogFileIndex(): Seq[FileStatus]
```

`allFilesUsingMetadataLogFileIndex` simply creates a new [MetadataLogFileIndex](#) and requests it to `allFiles` .

Note	<code>allFilesUsingMetadataLogFileIndex</code> is used exclusively when <code>FileStreamSource</code> is requested to fetchAllFiles (when requested for fetchMaxOffset when <code>FileStreamSource</code> is requested to getOffset).
------	--

FileStreamSink — Streaming Sink for File-Based Data Sources

`FileStreamSink` is a concrete [streaming sink](#) that writes out the results of a streaming query to files (of the specified [FileFormat](#)) in the [root path](#).

```
import scala.concurrent.duration._
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
val sq = in.
  writeStream.
  format("parquet").
  option("path", "parquet-output-dir").
  option("checkpointLocation", "checkpoint-dir").
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Append).
  start
```

`FileStreamSink` is [created](#) exclusively when `DataSource` is requested to [create a streaming sink](#) for a file-based data source (i.e. `FileFormat`).

Tip	Read up on FileFormat in The Internals of Spark SQL book.
-----	---

`FileStreamSink` supports [Append output mode](#) only.

`FileStreamSink` uses `spark.sql.streaming.fileSink.log.deletion` (as `isDeletingExpiredLog`)

The textual representation of `FileStreamSink` is **FileSink[path]**

`FileStreamSink` uses **_spark_metadata** directory for...FIXME

Tip	<p>Enable <code>ALL</code> logging level for <code>org.apache.spark.sql.execution.streaming.FileStreamSink</code> to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.execution.streaming.FileStreamSink=ALL</pre> <p>Refer to Logging.</p>
-----	--

Creating FileStreamSink Instance

`FileStreamSink` takes the following to be created:

- `SparkSession`
- Root directory
- `FileFormat`
- Names of the partition columns
- Configuration options

`FileStreamSink` initializes the [internal properties](#).

"Adding" Batch of Data to Sink — `addBatch` Method

```
addBatch(  
    batchId: Long,  
    data: DataFrame): Unit
```

Note	<code>addBatch</code> is a part of Sink Contract to "add" a batch of data to the sink.
------	--

`addBatch` ...FIXME

Creating BasicWriteJobStatsTracker — `basicWriteJobStatsTracker` Internal Method

```
basicWriteJobStatsTracker: BasicWriteJobStatsTracker
```

`basicWriteJobStatsTracker` simply creates a `BasicWriteJobStatsTracker` with the basic metrics:

- number of written files
- bytes of written output
- number of output rows
- number of dynamic partitions

Tip	Read up on BasicWriteJobStatsTracker in The Internals of Spark SQL book.
-----	--

Note	<code>basicWriteJobStatsTracker</code> is used exclusively when <code>FileStreamSink</code> is requested to addBatch .
------	--

`hasMetadata` Object Method

```
hasMetadata(  
    path: Seq[String],  
    hadoopConf: Configuration): Boolean
```

hasMetadata ...FIXME

Note	<p><code>hasMetadata</code> is used when:</p> <ul style="list-style-type: none"> • <code>DataSource</code> (Spark SQL) is requested to resolve a <code>FileFormat</code> relation (<code>resolveRelation</code>) and creates a <code>HadoopFsRelation</code> • <code>FileStreamSource</code> is requested to <code>fetchAllFiles</code>
------	---

Internal Properties

Name	Description
<code>basePath</code>	<p>Base path (Hadoop's Path for the given <code>path</code>)</p> <p>Used when...FIXME</p>
<code>logPath</code>	<p>Metadata log path (Hadoop's Path for the <code>base path</code> and the <code>_spark_metadata</code>)</p> <p>Used exclusively to create the FileStreamSinkLog</p>
<code>fileLog</code>	<p>FileStreamSinkLog (for the version 1 and the metadata log path)</p> <p>Used exclusively when <code>FileStreamSink</code> is requested to addBatch</p>
<code>hadoopConf</code>	<p>Hadoop's Configuration</p> <p>Used when...FIXME</p>

FileStreamSinkLog

`FileStreamSinkLog` is a concrete [CompactibleFileStreamLog](#) (of [SinkFileStatuses](#)) for [FileStreamSink](#) and [MetadataLogFileIndex](#).

`FileStreamSinkLog` uses **1** for the version.

`FileStreamSinkLog` uses **add** action to create new [metadata logs](#).

`FileStreamSinkLog` uses **delete** action to mark [metadata logs](#) that should be excluded from [compaction](#).

Creating FileStreamSinkLog Instance

`FileStreamSinkLog` (like the parent [CompactibleFileStreamLog](#)) takes the following to be created:

- Metadata version
- `SparkSession`
- Path of the metadata log directory

compactLogs Method

```
compactLogs(logs: Seq[SinkFileStatus]): Seq[SinkFileStatus]
```

Note	<code>compactLogs</code> is part of the CompactibleFileStreamLog Contract to...FIXME.
------	---

`compactLogs` ...FIXME

SinkFileStatus

`SinkFileStatus` represents the status of files of [FileStreamSink](#) (and the type of the metadata of [FileStreamSinkLog](#)):

- Path
- Size
- `isDir` flag
- Modification time
- Block replication
- Block size
- Action (either [add](#) or [delete](#))

toFileStatus Method

```
toFileStatus: FileStatus
```

`toFileStatus` simply creates a new Hadoop [FileStatus](#).

Note	<code>toFileStatus</code> is used exclusively when <code>MetadataLogFileIndex</code> is created .
------	---

Creating SinkFileStatus Instance — apply Object Method

```
apply(f: FileStatus): SinkFileStatus
```

`apply` simply creates a new [SinkFileStatus](#) (with [add](#) action).

Note	<code>apply</code> is used exclusively when <code>ManifestFileCommitProtocol</code> is requested to commitTask .
------	--

ManifestFileCommitProtocol

`ManifestFileCommitProtocol` is...FIXME

commitJob Method

```
commitJob(  
    jobContext: JobContext,  
    taskCommits: Seq[TaskCommitMessage]): Unit
```

Note `commitJob` is part of the `FileCommitProtocol` contract to...FIXME.

`commitJob` ...FIXME

commitTask Method

```
commitTask(  
    taskContext: TaskAttemptContext): TaskCommitMessage
```

Note `commitTask` is part of the `FileCommitProtocol` contract to...FIXME.

`commitTask` ...FIXME

MetadataLogFileIndex

`MetadataLogFileIndex` is a `PartitioningAwareFileIndex` of [metadata log files](#) (generated by [FileStreamSink](#)).

`MetadataLogFileIndex` is [created](#) when:

- `DataSource` (Spark SQL) is requested to resolve a `FileFormat` relation (`resolveRelation`) and creates a `HadoopFsRelation`
- `FileStreamSource` is requested to [allFilesUsingMetadataLogFileIndex](#)

Tip	<p>Enable <code>ALL</code> logging level for <code>org.apache.spark.sql.execution.streaming.MetadataLogFileIndex</code> to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.execution.streaming.MetadataLogFileIndex=ALL</pre> <p>Refer to Logging.</p>
--	---

Creating MetadataLogFileIndex Instance

`MetadataLogFileIndex` takes the following to be created:

- `SparkSession`
- Hadoop's [Path](#)
- User-defined schema (`Option[StructType]`)

`MetadataLogFileIndex` initializes the [internal properties](#).

While being created, `MetadataLogFileIndex` prints out the following INFO message to the logs:

```
Reading streaming file log from [metadataDirectory]
```

Internal Properties

Name	Description
metadataDirectory	Metadata directory (Hadoop's Path of the <code>_spark_metadata</code> directory under the path) Used when...FIXME
metadataLog	FileStreamSinkLog (with the <code>_spark_metadata</code> directory)
allFilesFromLog	Metadata log files

Kafka Data Source — Streaming Data Source for Apache Kafka

Kafka Data Source is the streaming data source for [Apache Kafka](#) in Spark Structured Streaming.

Kafka Data Source provides a [streaming source](#) and a [streaming sink](#) for [micro-batch](#) and [continuous](#) stream processing.

spark-sql-kafka-0-10 External Module

Kafka Data Source is part of the **spark-sql-kafka-0-10** external module that is distributed with the official distribution of Apache Spark, but it is not included in the CLASSPATH by default.

You should define `spark-sql-kafka-0-10` module as part of the build definition in your Spark project, e.g. as a `libraryDependency` in `build.sbt` for sbt:

```
libraryDependencies += "org.apache.spark" %% "spark-sql-kafka-0-10" % "2.4.3"
```

For Spark environments like `spark-submit` (and "derivatives" like `spark-shell`), you should use `--packages` command-line option:

```
./bin/spark-shell --packages org.apache.spark:spark-sql-kafka-0-10_2.12:2.4.3
```

Note	Replace the version of <code>spark-sql-kafka-0-10</code> module (e.g. <code>2.4.3</code> above) with one of the available versions found at The Central Repository's Search that matches your version of Apache Spark.
------	--

Streaming Source

With [spark-sql-kafka-0-10 module](#) you can use **kafka** data source format for reading records from one or more Kafka topics as a streaming Dataset.

```
val records = spark
  .readStream
  .format("kafka")
  .option("subscribePattern", """topic-\d{2}""") // topics with two digits at the end
  .option("kafka.bootstrap.servers", ":9092")
  .load
```

Internally, the **kafka** data source format for reading is available through [KafkaSourceProvider](#) that is a [MicroBatchReadSupport](#) and [ContinuousReadSupport](#) for micro-batch and [continuous](#) stream processing, respectively.

Predefined (Fixed) Schema

Kafka Data Source uses a predefined (fixed) schema.

Table 1. Kafka Data Source's Fixed Schema (in the positional order)

Name	Type
key	BinaryType
value	BinaryType
topic	StringType
partition	IntegerType
offset	LongType
timestamp	TimestampType
timestampType	IntegerType

```
scala> records.printSchema
root
|-- key: binary (nullable = true)
|-- value: binary (nullable = true)
|-- topic: string (nullable = true)
|-- partition: integer (nullable = true)
|-- offset: long (nullable = true)
|-- timestamp: timestamp (nullable = true)
|-- timestampType: integer (nullable = true)
```

Internally, the fixed schema is defined as part of the [DataSourceReader](#) contract through [MicroBatchReader](#) and [ContinuousReader](#) extension contracts for [micro-batch](#) and [continuous](#) stream processing, respectively.

Tip

Read up on [DataSourceReader](#) in [The Internals of Spark SQL](#) book.

Use `Column.cast` operator to cast `BinaryType` to a `StringType` (for key and value columns).

Tip

```
scala> :type records
org.apache.spark.sql.DataFrame

val values = records
  .select($"value" cast "string") // deserializing values
scala> values.printSchema
root
 |-- value: string (nullable = true)
```

Streaming Sink

With [spark-sql-kafka-0-10 module](#) you can use **kafka** data source format for writing the result of executing a streaming query (a streaming Dataset) to one or more Kafka topics.

```
val sq = records
  .writeStream
  .format("kafka")
  .option("kafka.bootstrap.servers", ":9092")
  .option("topic", "kafka2console-output")
  .option("checkpointLocation", "checkpointLocation-kafka2console")
  .start
```

Internally, the **kafka** data source format for writing is available through [KafkaSourceProvider](#) that is a [StreamWriterSupport](#).

Micro-Batch Stream Processing

Kafka Data Source supports [Micro-Batch Stream Processing](#) (i.e. `Trigger.Once` and `Trigger.ProcessingTime` triggers) via [KafkaMicroBatchReader](#).

```
import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._

val sq = spark
  .readStream
  .format("kafka")
  .option("subscribePattern", "kafka2console.*")
  .option("kafka.bootstrap.servers", ":9092")
  .load
  .withColumn("value", $"value" cast "string") // deserializing values
  .writeStream
  .format("console")
  .option("truncate", false) // format-specific option
  .option("checkpointLocation", "checkpointLocation-kafka2console") // generic query option
  .trigger(Trigger.ProcessingTime(30.seconds))
  .queryName("kafka2console-microbatch")
  .start

// In the end, stop the streaming query
sq.stop
```

Kafka Data Source can assign a single task per Kafka partition (using [KafkaOffsetRangeCalculator](#) in [Micro-Batch Stream Processing](#)).

Kafka Data Source can reuse a Kafka consumer (using [KafkaMicroBatchReader](#) in [Micro-Batch Stream Processing](#)).

Continuous Stream Processing

Kafka Data Source supports [Continuous Stream Processing](#) (i.e. `Trigger.Continuous` trigger) via [KafkaContinuousReader](#).

```

import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._

val sq = spark
  .readStream
  .format("kafka")
  .option("subscribePattern", "kafka2console.*")
  .option("kafka.bootstrap.servers", ":9092")
  .load
  .withColumn("value", $"value" cast "string") // convert bytes to string for display
purposes
  .writeStream
  .format("console")
  .option("truncate", false) // format-specific option
  .option("checkpointLocation", "checkpointLocation-kafka2console") // generic query o
ption
  .queryName("kafka2console-continuous")
  .trigger(Trigger.Continuous(10.seconds))
  .start

// In the end, stop the streaming query
sq.stop

```

Configuration Options

Note

Options with **kafka.** prefix (e.g. `kafka.bootstrap.servers`) are considered configuration properties for the Kafka consumers used on the `driver` and `executors`.

Table 2. Kafka Data Source's Options (Case-Insensitive)

Option	Description
<code>assign</code>	<p>Topic subscription strategy that accepts a JSON with topics and partitions, e.g.</p> <pre>{"topicA": [0,1], "topicB": [0,1]}</pre> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <p>Note Exactly one topic subscription strategy is allowed (that <code>KafkaSourceProvider</code> <code>validates</code> before creating <code>KafkaSource</code>).</p> </div>
<code>failOnDataLoss</code>	<p>Flag to control whether...FIXME</p> <p>Default: <code>true</code></p> <p>Used when <code>KafkaSourceProvider</code> is requested for <code>failOnDataLoss</code> configuration property</p>
<code>kafka.bootstrap.servers</code>	(required) <code>bootstrap.servers</code> configuration property of the Kafka consumers used on the driver and executors

	<p>Default: (empty)</p>
<code>kafkaConsumer.pollTimeoutMs</code>	<p>The time (in milliseconds) spent waiting in <code>consumer.poll</code> is not available in the buffer.</p> <p>Default: <code>spark.network.timeout</code> or 120s</p> <p>Used when...FIXME</p>
<code>maxOffsetsPerTrigger</code>	<p>Number of records to fetch per trigger (to limit the number of records to fetch).</p> <p>Default: (undefined)</p> <p>Unless defined, <code>KafkaSource</code> requests KafkaOffsetReader latest offsets.</p>
<code>minPartitions</code>	<p>Minimum number of partitions per executor (given Kafka partitions)</p> <p>Default: (undefined)</p> <p>Must be undefined (default) or greater than 0</p> <p>When undefined (default) or smaller than the number of <code>TopicPartitions</code> with records to consume from, <code>KafkaMicroBatchReader</code> uses KafkaOffsetRangeCalculator to determine the preferred executor for every <code>TopicPartition</code> (and the available executors).</p>
<code>startingOffsets</code>	<p>Starting offsets</p> <p>Default: <code>latest</code></p> <p>Possible values:</p> <ul style="list-style-type: none"> • <code>latest</code> • <code>earliest</code> • JSON with topics, partitions and their starting offsets, <pre>{"topicA":{"part":offset,"p1":-1}, "topicB":{"0":-2}}</pre> <p>Tip Use Scala's triple quotes for the JSON for topic partitions and offsets.</p> <pre>option("startingOffsets", """{"topic1":{"0":5, "4":-1}, "topic2":{"0":-2}""")</pre>

	<p>Topic subscription strategy that accepts topic names as a separated string, e.g.</p> <pre>topic1,topic2,topic3</pre>
subscribe	<p>Note Exactly one topic subscription strategy is allowed (that <code>kafkaSourceProvider validates</code> before creating <code>KafkaSource</code>).</p>
	<p>Topic subscription strategy that uses Java's <code>java.util.regex</code> for the topic subscription regex pattern of topics to subscribe, e.g.</p> <pre>topic\d</pre>
subscribepattern	<p>Tip Use Scala's triple quotes for the regular expression for topic subscription regex pattern.</p> <pre>option("subscribepattern", """topic\d""")</pre>
	<p>Note Exactly one topic subscription strategy is allowed (that <code>kafkaSourceProvider validates</code> before creating <code>KafkaSource</code>).</p>
topic	<p>Optional topic name to use for writing a streaming query Default: (empty)</p> <p>Unless defined, Kafka data source uses the topic names defined in the <code>topic</code> field in the incoming data.</p>

Logical Query Plan for Reading

When `DataStreamReader` is requested to load a dataset with **kafka** data source format, it creates a DataFrame with a [StreamingRelationV2](#) leaf logical operator.

```
scala> records.explain(extended = true)
== Parsed Logical Plan ==
StreamingRelationV2 org.apache.spark.sql.kafka010.KafkaSourceProvider@1a366d0, kafka,
Map(maxOffsetsPerTrigger -> 1, startingOffsets -> latest, subscribepattern -> topic\d,
kafka.bootstrap.servers -> :9092), [key#7, value#8, topic#9, partition#10, offset#11L
, timestamp#12, timestampType#13], StreamingRelation DataSource(org.apache.spark.sql.S
parkSession@39b3de87,kafka,List(),None,List(),None,Map(maxOffsetsPerTrigger -> 1, star
tingOffsets -> latest, subscribepattern -> topic\d, kafka.bootstrap.servers -> :9092),
None), kafka, [key#0, value#1, topic#2, partition#3, offset#4L, timestamp#5, timestamp
Type#6]
...
...
```

Logical Query Plan for Writing

When `DataStreamWriter` is requested to start a streaming query with **kafka** data source format for writing, it requests the `StreamingQueryManager` to [create a streaming query](#) that in turn creates (a `StreamingQueryWrapper` with) a [ContinuousExecution](#) or a [MicroBatchExecution](#) for [continuous](#) and [micro-batch](#) stream processing, respectively.

```
scala> sq.explain(extended = true)
== Parsed Logical Plan ==
WriteToDataSourceV2 org.apache.spark.sql.execution.streaming.sources.MicroBatchWriter@
bf98b73
+- Project [key#28 AS key#7, value#29 AS value#8, topic#30 AS topic#9, partition#31 AS
partition#10, offset#32L AS offset#11L, timestamp#33 AS timestamp#12, timestampType#34
AS timestampType#13]
  +- Streaming RelationV2 kafka[key#28, value#29, topic#30, partition#31, offset#32L,
timestamp#33, timestampType#34] (Options: [subscribePattern=kafka2console.*, kafka.boot
strap.servers=:9092])
```

Demo: Streaming Aggregation with Kafka Data Source

Check out [Demo: Streaming Aggregation with Kafka Data Source](#).

	Use the following to publish events to Kafka.
Tip	<pre>// 1st streaming batch \$ cat /tmp/1 1,1,1 15,2,1 \$ kafkacat -P -b localhost:9092 -t topic1 -l /tmp/1 // Alternatively (and slower due to JVM bootup) \$ cat /tmp/1 ./bin/kafka-console-producer.sh --topic topic1 --broker-list loca</pre>

KafkaSourceProvider — Data Source Provider for Apache Kafka

`KafkaSourceProvider` is a `DataSourceRegister` and registers a developer-friendly alias for **kafka** data source format in Spark Structured Streaming.

Tip Read up on [DataSourceRegister](#) in [The Internals of Spark SQL](#) book.

`KafkaSourceProvider` supports [micro-batch stream processing](#) (through [MicroBatchReadSupport](#) contract) and [creates a specialized KafkaMicroBatchReader](#).

`KafkaSourceProvider` requires the following options (that you can set using `option` method of [DataStreamReader](#) or [DataStreamWriter](#)):

- Exactly one of the following options: [subscribe](#), [subscribePattern](#) or [assign](#)
- [kafka.bootstrap.servers](#)

Tip Refer to [Kafka Data Source's Options](#) for the supported configuration options.

Internally, `KafkaSourceProvider` sets the [properties for Kafka Consumers on executors](#) (that are passed on to `InternalKafkaConsumer` when requested to create a Kafka consumer with a single `TopicPartition` manually assigned).

Table 1. KafkaSourceProvider's Properties for Kafka Consumers on Executors

ConsumerConfig's Key	Value	Description
KEY_DESERIALIZER_CLASS_CONFIG	ByteArrayDeserializer	FIXME
VALUE_DESERIALIZER_CLASS_CONFIG	ByteArrayDeserializer	FIXME
AUTO_OFFSET_RESET_CONFIG	none	FIXME
GROUP_ID_CONFIG	uniqueGroupId-executor	FIXME
ENABLE_AUTO_COMMIT_CONFIG	false	FIXME
RECEIVE_BUFFER_CONFIG	65536	Only when not set in the specifiedKafkaParams already

Tip Enable `INFO` or `DEBUG` logging levels for `org.apache.spark.sql.kafka010.KafkaSourceProvider` to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.kafka010.KafkaSourceProvider=DEBUG
```

Refer to [Logging](#).

Creating Streaming Source — `createSource` Method

```
createSource(  
    sqlContext: SQLContext,  
    metadataPath: String,  
    schema: Option[StructType],  
    providerName: String,  
    parameters: Map[String, String]): Source
```

Note

`createSource` is part of the [StreamSourceProvider Contract](#) to create a [streaming source](#) for a format or system (to continually read data).

`createSource` first [validates stream options](#).

`createSource` ...FIXME

Validating General Options For Batch And Streaming Queries — `validateGeneralOptions` Internal Method

```
validateGeneralOptions(parameters: Map[String, String]): Unit
```

Note

Parameters are case-insensitive, i.e. `optionN` and `option` are equal.

`validateGeneralOptions` makes sure that exactly one topic subscription strategy is used in `parameters` and can be:

- `subscribe`
- `subscribepattern`
- `assign`

`validateGeneralOptions` reports an `IllegalArgumentException` when there is no subscription strategy in use or there are more than one strategies used.

`validateGeneralOptions` makes sure that the value of subscription strategies meet the requirements:

- `assign` strategy starts with `{` (the opening curly brace)
- `subscribe` strategy has at least one topic (in a comma-separated list of topics)
- `subscribepattern` strategy has the pattern defined

`validateGeneralOptions` makes sure that `group.id` has not been specified and reports an `IllegalArgumentException` otherwise.

Kafka option 'group.id' is not supported as user-specified consumer groups are not used to track offsets.

`validateGeneralOptions` makes sure that `auto.offset.reset` has not been specified and reports an `IllegalArgumentException` otherwise.

Kafka option 'auto.offset.reset' is not supported.
Instead set the source option 'startingoffsets' to 'earliest' or 'latest' to specify where to start. Structured Streaming manages which offsets are consumed internally, rather than relying on the `kafkaConsumer` to do it. This will ensure that no data is missed when new topics/partitions are dynamically subscribed.
Note that 'startingoffsets' only applies when a new Streaming query is started, and
that resuming will always pick up from where the query left off.
See the docs for more details.

`validateGeneralOptions` makes sure that the following options have not been specified and reports an `IllegalArgumentException` otherwise:

- `kafka.key.deserializer`
- `kafka.value.deserializer`
- `kafka.enable.auto.commit`
- `kafka.interceptor.classes`

In the end, `validateGeneralOptions` makes sure that `kafka.bootstrap.servers` option was specified and reports an `IllegalArgumentException` otherwise.

```
Option 'kafka.bootstrap.servers' must be specified for configuring Kafka consumer
```

Note

`validateGeneralOptions` is used when `KafkaSourceProvider` validates options for [streaming](#) and [batch](#) queries.

Creating ConsumerStrategy — `strategy` Internal Method

```
strategy(caseInsensitiveParams: Map[String, String])
```

Internally, `strategy` finds the keys in the input `caseInsensitiveParams` that are one of the following and creates a corresponding [ConsumerStrategy](#).

Table 2. `KafkaSourceProvider.strategy`'s Key to ConsumerStrategy Conversion

Key	ConsumerStrategy
assign	<p>AssignStrategy with Kafka's <code>TopicPartitions</code>.</p> <hr/> <p><code>strategy</code> uses <code>JsonUtils.partitions</code> method to parse a JSON with topic names and partitions, e.g.</p> <pre>{"topicA": [0,1], "topicB": [0,1]}</pre> <p>The topic names and partitions are mapped directly to Kafka's <code>TopicPartition</code> objects.</p>
subscribe	<p>SubscribeStrategy with topic names</p> <hr/> <p><code>strategy</code> extracts topic names from a comma-separated string, e.g.</p> <pre>topic1,topic2,topic3</pre>
subscribepattern	<p>SubscribePatternStrategy with topic subscription regex pattern (that uses Java's <code>java.util.regex.Pattern</code> for the pattern), e.g.</p> <pre>topic\d</pre>

Note

- `strategy` is used when:
- `KafkaSourceProvider` creates a `KafkaOffsetReader` for `KafkaSource`.
 - `KafkaSourceProvider` creates a `KafkaRelation` (using `createRelation` method).

Describing Streaming Source with Name and Schema — `sourceSchema` Method

```
sourceSchema(  
    sqlContext: SQLContext,  
    schema: Option[StructType],  
    providerName: String,  
    parameters: Map[String, String]): (String, StructType)
```

Note

`sourceSchema` is part of the [StreamSourceProvider Contract](#) to describe a [streaming source](#) with a name and the schema.

`sourceSchema` gives the [short name](#) (i.e. `kafka`) and the [fixed schema](#).

Internally, `sourceSchema` [validates Kafka options](#) and makes sure that the optional input `schema` is indeed undefined.

When the input `schema` is defined, `sourceSchema` reports a `IllegalArgumentException`.

Kafka source has a fixed schema and cannot be set with a custom one

Validating Kafka Options for Streaming Queries — `validateStreamOptions` Internal Method

```
validateStreamOptions(caseInsensitiveParams: Map[String, String]): Unit
```

Firstly, `validateStreamOptions` makes sure that `endingoffsets` option is not used.

Otherwise, `validateStreamOptions` reports a `IllegalArgumentException`.

ending offset not valid in streaming queries

`validateStreamOptions` then [validates the general options](#).

Note

`validateStreamOptions` is used when `KafkaSourceProvider` is requested the [schema for Kafka source](#) and to [create a KafkaSource](#).

Creating ContinuousReader for Continuous Stream Processing — `createContinuousReader` Method

```
createContinuousReader(  
    schema: Optional[StructType],  
    metadataPath: String,  
    options: DataSourceOptions): KafkaContinuousReader
```

Note	<code>createContinuousReader</code> is part of the ContinuousReadSupport Contract to create a ContinuousReader .
------	--

`createContinuousReader` ...FIXME

Converting Configuration Options to KafkaOffsetRangeLimit — `getKafkaOffsetRangeLimit` Object Method

```
getKafkaOffsetRangeLimit(  
    params: Map[String, String],  
    offsetOptionKey: String,  
    defaultOffsets: KafkaOffsetRangeLimit): KafkaOffsetRangeLimit
```

`getKafkaOffsetRangeLimit` finds the given `offsetOptionKey` in the `params` and does the following conversion:

- **latest** becomes [LatestOffsetRangeLimit](#)
- **earliest** becomes [EarliestOffsetRangeLimit](#)
- A JSON-formatted text becomes [SpecificOffsetRangeLimit](#)
- When the given `offsetOptionKey` is not found, `getKafkaOffsetRangeLimit` returns the given `defaultOffsets`

Note	<code>getKafkaOffsetRangeLimit</code> is used when <code>kafkaSourceProvider</code> is requested to createSource , createMicroBatchReader , createContinuousReader , createRelation , and validateBatchOptions .
------	--

Creating MicroBatchReader for Micro-Batch Stream Processing — `createMicroBatchReader` Method

```
createMicroBatchReader(  
    schema: Optional[StructType],  
    metadataPath: String,  
    options: DataSourceOptions): KafkaMicroBatchReader
```

Note

`createMicroBatchReader` is part of the [MicroBatchReadSupport Contract](#) to create a [MicroBatchReader](#) in [Micro-Batch Stream Processing](#).

`createMicroBatchReader` [validateStreamOptions](#) (in the given `DataSourceOptions`).

`createMicroBatchReader` generates a unique group ID of the format **spark-kafka-source-[randomUUID]-[metadataPath_hashCode]** (to make sure that a new streaming query creates a new consumer group).

`createMicroBatchReader` finds all the parameters (in the given `DataSourceOptions`) that start with **kafka.** prefix, removes the prefix, and creates the current Kafka parameters.

`createMicroBatchReader` creates a [KafkaOffsetReader](#) with the following:

- `strategy` (in the given `DataSourceOptions`)
- [Properties for Kafka consumers on the driver](#) (given the current Kafka parameters, i.e. without **kafka.** prefix)
- The given `DataSourceOptions`
- **spark-kafka-source-[randomUUID]-[metadataPath_hashCode]-driver** for the `driverGroupIdPrefix`

In the end, `createMicroBatchReader` creates a [KafkaMicroBatchReader](#) with the following:

- the `KafkaOffsetReader`
- [Properties for Kafka consumers on executors](#) (given the current Kafka parameters, i.e. without **kafka.** prefix) and the unique group ID (`spark-kafka-source-[randomUUID]-[metadataPath_hashCode]-driver`)
- The given `DataSourceOptions` and the `metadataPath`
- [Starting stream offsets](#) (`startingOffsets` option with the default of `LatestOffsetRangeLimit offsets`)
- [failOnDataLoss configuration property](#)

Creating BaseRelation — `createRelation` Method

```
createRelation(  
    sqlContext: SQLContext,  
    parameters: Map[String, String]): BaseRelation
```

Note

`createRelation` is part of the [RelationProvider](#) contract to create a `BaseRelation`.

`createRelation` ...FIXME

Validating Configuration Options for Batch Processing — `validateBatchOptions` Internal Method

```
validateBatchOptions(caseInsensitiveParams: Map[String, String]): Unit
```

`validateBatchOptions` ...FIXME

Note

`validateBatchOptions` is used exclusively when `KafkaSourceProvider` is requested to [createSource](#).

kafkaParamsForDriver Method

```
kafkaParamsForDriver(specifiedKafkaParams: Map[String, String]): Map[String, Object]
```

`kafkaParamsForDriver` ...FIXME

Note

`kafkaParamsForDriver` is used when...FIXME

kafkaParamsForExecutors Method

```
kafkaParamsForExecutors(  
    specifiedKafkaParams: Map[String, String],  
    uniqueGroupId: String): Map[String, Object]
```

`kafkaParamsForExecutors` sets the [Kafka properties for executors](#).

While setting the properties, `kafkaParamsForExecutors` prints out the following DEBUG message to the logs:

```
executor: Set [key] to [value], earlier value: [value]
```

Note	<p><code>kafkaParamsForExecutors</code> is used when:</p> <ul style="list-style-type: none">• <code>KafkaSourceProvider</code> is requested to <code>createSource</code> (for a <code>KafkaSource</code>), <code>createMicroBatchReader</code> (for a <code>KafkaMicroBatchReader</code>), and <code>createContinuousReader</code> (for a <code>KafkaContinuousReader</code>)• <code>KafkaRelation</code> is requested to <code>buildScan</code> (for a <code>KafkaSourceRDD</code>)
------	--

Looking Up `failOnDataLoss` Configuration Property — `failOnDataLoss` Internal Method

```
failOnDataLoss(caseInsensitiveParams: Map[String, String]): Boolean
```

`failOnDataLoss` simply looks up the `failOnDataLoss` configuration property in the given `caseInsensitiveParams` (in case-insensitive manner) or defaults to `true`.

Note	<p><code>failOnDataLoss</code> is used when <code>KafkaSourceProvider</code> is requested to <code>createSource</code> (for a <code>KafkaSource</code>), <code>createMicroBatchReader</code> (for a <code>KafkaMicroBatchReader</code>), <code>createContinuousReader</code> (for a <code>KafkaContinuousReader</code>), and <code>createRelation</code> (for a <code>KafkaRelation</code>).</p>
------	--

KafkaSource

`KafkaSource` is a [streaming source](#) that generates `DataFrames` of records from one or more topics in Apache Kafka.

Note Kafka topics are checked for new records every [trigger](#) and so there is some noticeable delay between when the records have arrived to Kafka topics and when a Spark application processes them.

`KafkaSource` uses the [streaming metadata log directory](#) to persist offsets. The directory is the source ID under the `sources` directory in the `checkpointRoot` (of the `StreamExecution`).

Note The `checkpointRoot` directory is one of the following:

- `checkpointLocation` option
- `spark.sql.streaming.checkpointLocation` configuration property

`KafkaSource` is created for [kafka](#) format (that is registered by `KafkaSourceProvider`).

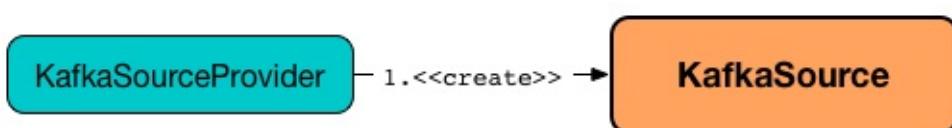


Figure 1. KafkaSource Is Created for kafka Format by KafkaSourceProvider

`KafkaSource` uses a [predefined \(fixed\) schema](#) (that cannot be changed).

`KafkaSource` also supports batch Datasets.

Table 1. KafkaSource's Internal Registries and Counters

Name	Description
<code>currentPartitionOffsets</code>	Current partition offsets (as <code>Map[TopicPartition, Long]</code>) Initially <code>NONE</code> and set when <code>KafkaSource</code> is requested to get the maximum available offsets or generate a <code>DataFrame</code> with records from Kafka for a batch.

Tip	Enable <code>INFO</code> or <code>DEBUG</code> logging levels for <code>org.apache.spark.sql.kafka010.KafkaSource</code> to see what happens inside. Add the following line to <code>conf/log4j.properties</code> : <code>log4j.logger.org.apache.spark.sql.kafka010.KafkaSource=DEBUG</code>
	Refer to Logging .

rateLimit Internal Method

```
rateLimit(  
    limit: Long,  
    from: Map[TopicPartition, Long],  
    until: Map[TopicPartition, Long]): Map[TopicPartition, Long]
```

`rateLimit` requests `KafkaOffsetReader` to `fetchEarliestOffsets`.

Caution	FIXME
Note	<code>rateLimit</code> is used exclusively when <code>KafkaSource</code> gets available offsets (when <code>maxOffsetsPerTrigger</code> option is specified).

getSortedExecutorList Method

Caution	FIXME
---------	-------

reportDataLoss Internal Method

Caution	FIXME
Note	<p><code>reportDataLoss</code> is used when <code>KafkaSource</code> does the following:</p> <ul style="list-style-type: none"> • fetches and verifies specific offsets • generates a DataFrame with records from Kafka for a batch

Generating DataFrame with Records From Kafka for Streaming Batch — getBatch Method

```
getBatch(start: Option[Offset], end: Offset): DataFrame
```

Note	<code>getBatch</code> is a part of Source Contract .
------	--

`getBatch` initializes [initial partition offsets](#) (unless initialized already).

You should see the following INFO message in the logs:

```
INFO KafkaSource: GetBatch called with start = [start], end = [end]
```

`getBatch` requests `KafkaSourceOffset` for `end partition offsets` for the input `end offset` (known as `untilPartitionOffsets`).

`getBatch` requests `KafkaSourceOffset` for `start partition offsets` for the input `start offset` (if defined) or uses `initial partition offsets` (known as `fromPartitionOffsets`).

`getBatch` finds the new partitions (as the difference between the topic partitions in `untilPartitionOffsets` and `fromPartitionOffsets`) and requests `KafkaOffsetReader` to fetch their earliest offsets.

`getBatch` reports a data loss if the new partitions don't match to what `KafkaOffsetReader` fetched.

```
Cannot find earliest offsets of [partitions]. Some data may have been missed
```

You should see the following INFO message in the logs:

```
INFO KafkaSource: Partitions added: [partitionOffsets]
```

`getBatch` reports a data loss if the new partitions don't have their offsets `0`.

```
Added partition [partition] starts from [offset] instead of 0. Some data may have been missed
```

`getBatch` reports a data loss if the `fromPartitionOffsets` partitions differ from `untilPartitionOffsets` partitions.

```
[partitions] are gone. Some data may have been missed
```

You should see the following DEBUG message in the logs:

```
DEBUG KafkaSource: TopicPartitions: [comma-separated topicPartitions]
```

`getBatch` gets the executors (sorted by `executorId` and `host` of the registered block managers).

Important	That is when <code>getBatch</code> goes very low-level to allow for cached <code>KafkaConsumers</code> in the executors to be re-used to read the same partition in every batch (aka <i>location preference</i>).
-----------	--

You should see the following DEBUG message in the logs:

```
DEBUG KafkaSource: Sorted executors: [comma-separated sortedExecutors]
```

`getBatch` creates a `KafkaSourceRDDOffsetRange` per `TopicPartition`.

`getBatch` filters out `KafkaSourceRDDOffsetRanges` for which until offsets are smaller than from offsets. `getBatch` reports a data loss if they are found.

Partition [topicPartition]'s offset was changed from [fromOffset] to [untilOffset], so me data may have been missed

`getBatch` creates a `KafkaSourceRDD` (with `executorKafkaParams`, `pollTimeoutMs` and `reuseKafkaConsumer` flag enabled) and maps it to an RDD of `InternalRow`.

Important	<code>getBatch</code> creates a <code>KafkaSourceRDD</code> with <code>reuseKafkaConsumer</code> flag enabled.
-----------	--

You should see the following INFO message in the logs:

```
INFO KafkaSource: GetBatch generating RDD of offset range: [comma-separated offsetRanges sorted by topicPartition]
```

`getBatch` sets `currentPartitionOffsets` if it was empty (which is when...FIXME)

In the end, `getBatch` creates a `DataFrame` from the RDD of `InternalRow` and `schema`.

Fetching Offsets (From Metadata Log or Kafka Directly) — `getOffset` Method

```
getOffset: Option[Offset]
```

Note	<code>getOffset</code> is a part of the Source Contract .
------	---

Internally, `getOffset` fetches the [initial partition offsets](#) (from the metadata log or Kafka directly).

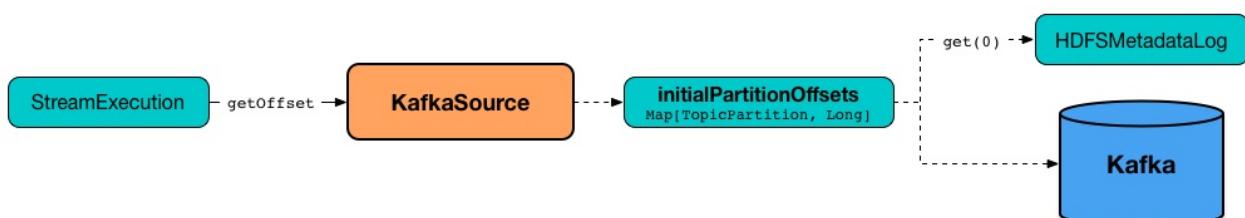


Figure 2. KafkaSource Initializing `initialPartitionOffsets` While Fetching Initial Offsets

Note `initialPartitionOffsets` is a lazy value and is initialized the very first time `getOffset` is called (which is when `StreamExecution` constructs a streaming micro-batch).

```

scala> spark.version
res0: String = 2.3.0-SNAPSHOT

// Case 1: Checkpoint directory undefined
// initialPartitionOffsets read from Kafka directly
val records = spark.
  readStream.
  format("kafka").
  option("subscribe", "topic1").
  option("kafka.bootstrap.servers", "localhost:9092").
  load

// Start the streaming query
// dump records to the console every 10 seconds
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration.-
val q = records.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Update).
  start

// Note the temporary checkpoint directory
17/08/07 11:09:29 INFO StreamExecution: Starting [id = 75dd261d-6b62-40fc-a368-9d95d3c
b6f5f, runId = f18a5eb5-ccab-4d9d-8a81-befed41a72bd] with file:///private/var/folders/
0w/kb0d3rqn4zb9fcc91pxhgn8w0000gn/T/temporary-d0055630-24e4-4d9a-8f36-7a12a0f11bc0 to
store the query checkpoint.
...
INFO KafkaSource: Initial offsets: {"topic1":{"0":1}}

// Stop the streaming query
q.stop

// Case 2: Checkpoint directory defined
// initialPartitionOffsets read from Kafka directly
// since the checkpoint directory is not available yet
// it will be the next time the query is started
val records = spark.
  readStream.
  format("kafka").
  option("subscribe", "topic1").
  option("kafka.bootstrap.servers", "localhost:9092").
  load.
  select($"value" cast "string", $"topic", $"partition", $"offset")
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration.-
val q = records.

```

```

writeStream.
format("console").
option("truncate", false).
option("checkpointLocation", "/tmp/checkpoint"). // <-- checkpoint directory
trigger(Trigger.ProcessingTime(10.seconds)).
outputMode(OutputMode.Update).
start
// Note the checkpoint directory in use
17/08/07 11:21:25 INFO StreamExecution: Starting [id = b8f59854-61c1-4c2f-931d-62bbaf9
0ee3b, runId = 70d06a3b-f2b1-4fa8-a518-15df4cf59130] with file:///tmp/checkpoint to st
ore the query checkpoint.
...
INFO KafkaSource: Initial offsets: {"topic1":{"0":1}}
...
INFO StreamExecution: Stored offsets for batch 0. Metadata OffsetSeqMetadata(0,1502098
526848,Map(spark.sql.shuffle.partitions -> 200, spark.sql.streaming.stateStore.provide
rClass -> org.apache.spark.sql.execution.streaming.state.HDFSBackedStateStoreProvider)
)

// Review the checkpoint location
// $ ls -ltr /tmp/checkpoint/offsets
// total 8
// -rw-r--r-- 1 jacek wheel 248 7 sie 11:21 0
// $ tail -2 /tmp/checkpoint/offsets/0 | jq

// Produce messages to Kafka so the latest offset changes
// And more importantly the offset gets stored to checkpoint location
-----
Batch: 1
-----
+-----+-----+-----+
|value          |topic |partition|offset|
+-----+-----+-----+
|testing checkpoint location|topic1|0      |2      |
+-----+-----+-----+

// and one more
// Note the offset
-----
Batch: 2
-----
+-----+-----+-----+
|value          |topic |partition|offset|
+-----+-----+-----+
|another test|topic1|0      |3      |
+-----+-----+-----+

// See what was checkpointed
// $ ls -ltr /tmp/checkpoint/offsets
// total 24
// -rw-r--r-- 1 jacek wheel 248 7 sie 11:35 0
// -rw-r--r-- 1 jacek wheel 248 7 sie 11:37 1
// -rw-r--r-- 1 jacek wheel 248 7 sie 11:38 2

```

```
// $ tail -2 /tmp/checkpoint/offsets/2 | jq

// Stop the streaming query
q.stop

// And start over to see what offset the query starts from
// Checkpoint location should have the offsets
val q = records.
  writeStream.
  format("console").
  option("truncate", false).
  option("checkpointLocation", "/tmp/checkpoint"). // <-- checkpoint directory
  trigger(Trigger.ProcessingTime(10.seconds)).
  outputMode(OutputMode.Update).
  start

// Whoops...console format does not support recovery (!)
// Reported as https://issues.apache.org/jira/browse/SPARK-21667
org.apache.spark.sql.AnalysisException: This query does not support recovering from ch
eckpoint location. Delete /tmp/checkpoint/offsets to start over.;
  at org.apache.spark.sql.streaming.StreamingQueryManager.createQuery(StreamingQueryMa
nager.scala:222)
  at org.apache.spark.sql.streaming.StreamingQueryManager.startQuery(StreamingQueryMan
ager.scala:278)
  at org.apache.spark.sql.streaming.DataStreamWriter.start(DataStreamWriter.scala:284)
  ... 61 elided

// Change the sink (= output format) to JSON
val q = records.
  writeStream.
  format("json").
  option("path", "/tmp/json-sink").
  option("checkpointLocation", "/tmp/checkpoint"). // <-- checkpoint directory
  trigger(Trigger.ProcessingTime(10.seconds)).
  start

// Note the checkpoint directory in use
17/08/07 12:09:02 INFO StreamExecution: Starting [id = 02e00924-5f0d-4501-bcb8-80be8a8
be385, runId = 5eba2576-dad6-4f95-9031-e72514475edc] with file:///tmp/checkpoint to st
ore the query checkpoint.
...
17/08/07 12:09:02 INFO KafkaSource: GetBatch called with start = Some({"topic1":{"0":3
}}), end = {"topic1":{"0":4}}
17/08/07 12:09:02 INFO KafkaSource: Partitions added: Map()
17/08/07 12:09:02 DEBUG KafkaSource: TopicPartitions: topic1-0
17/08/07 12:09:02 DEBUG KafkaSource: Sorted executors:
17/08/07 12:09:02 INFO KafkaSource: GetBatch generating RDD of offset range: KafkaSour
cerRDDOffsetRange(topic1-0,3,4,None)
17/08/07 12:09:03 DEBUG KafkaOffsetReader: Partitions assigned to consumer: [topic1-0]
. Seeking to the end.
17/08/07 12:09:03 DEBUG KafkaOffsetReader: Got latest offsets for partition : Map(topi
c1-0 -> 4)
17/08/07 12:09:03 DEBUG KafkaSource: GetOffset: ArrayBuffer((topic1-0,4))
17/08/07 12:09:03 DEBUG StreamExecution: getOffset took 122 ms
17/08/07 12:09:03 DEBUG StreamExecution: Resuming at batch 3 with committed offsets {K
```

```
afkaSource[Subscribe[topic1]]: {"topic1":{"0":4}} and available offsets {KafkaSource[Subscribe[topic1]]: {"topic1":{"0":4}}}
17/08/07 12:09:03 DEBUG StreamExecution: Stream running from {KafkaSource[Subscribe[topic1]]: {"topic1":{"0":4}}} to {KafkaSource[Subscribe[topic1]]: {"topic1":{"0":4}}}
```

`getOffset` requests `KafkaOffsetReader` to `fetchLatestOffsets` (known later as `latest`).

Note	(Possible performance degradation?) It is possible that <code>getOffset</code> will request the latest offsets from Kafka twice, i.e. while initializing <code>initialPartitionOffsets</code> (when no metadata log is available and KafkaSource's <code>KafkaOffsetRangeLimit</code> is <code>LatestOffsetRangeLimit</code>) and always as part of <code>getOffset</code> itself.
------	---

`getOffset` then calculates `currentPartitionOffsets` based on the `maxOffsetsPerTrigger` option.

Table 2. `getOffset`'s Offset Calculation per `maxOffsetsPerTrigger`

maxOffsetsPerTrigger	Offsets
Unspecified (i.e. <code>None</code>)	<code>latest</code>
Defined (but <code>currentPartitionOffsets</code> is empty)	<code>rateLimit</code> with <code>limit limit</code> , <code>initialPartitionOffsets as from</code> , <code>until as latest</code>
Defined (and <code>currentPartitionOffsets</code> contains partitions and offsets)	<code>rateLimit</code> with <code>limit limit</code> , <code>currentPartitionOffsets as from</code> , <code>until as latest</code>

You should see the following DEBUG message in the logs:

```
DEBUG KafkaSource: GetOffset: [offsets]
```

In the end, `getOffset` creates a `KafkaSourceOffset` with `offsets` (as `Map[TopicPartition, Long]`).

Creating KafkaSource Instance

`KafkaSource` takes the following when created:

- `SQLContext`
- `KafkaOffsetReader`
- Parameters of executors (reading from Kafka)
- Collection of key-value options

- **Streaming metadata log directory**, i.e. the directory for streaming metadata log (where `KafkaSource` persists `KafkaSourceOffset` offsets in JSON format)
- **Starting offsets** (as defined using `startingOffsets` option)
- Flag used to `create KafkaSourceRDDs` every trigger and when checking to `report a IllegalStateException` on data loss.

`KafkaSource` initializes the `internal registries and counters`.

Fetching and Verifying Specific Offsets

— `fetchAndVerify` Internal Method

```
fetchAndVerify(specificOffsets: Map[TopicPartition, Long]): KafkaSourceOffset
```

`fetchAndVerify` requests `KafkaOffsetReader` to `fetchSpecificOffsets` for the given `specificOffsets`.

`fetchAndVerify` makes sure that the starting offsets in `specificOffsets` are the same as in Kafka and `reports a data loss` otherwise.

```
startingOffsets for [tp] was [off] but consumer reset to [result(tp)]
```

In the end, `fetchAndVerify` creates a `KafkaSourceOffset` (with the result of `KafkaOffsetReader`).

Note	<code>fetchAndVerify</code> is used exclusively when <code>KafkaSource</code> initializes <code>initial partition offsets</code> .
------	--

Initial Partition Offsets (of 0th Batch)

— `initialPartitionOffsets` Internal Lazy Property

```
initialPartitionOffsets: Map[TopicPartition, Long]
```

`initialPartitionOffsets` is the **initial partition offsets** for the batch `0` that were already persisted in the `streaming metadata log directory` or persisted on demand.

As the very first step, `initialPartitionOffsets` creates a custom `HDFSMetadataLog` (of `KafkaSourceOffsets` metadata) in the `streaming metadata log directory`.

`initialPartitionOffsets` requests the `HDFSMetadataLog` for the `metadata` of the `0` th batch (as `KafkaSourceOffset`).

If the metadata is available, `initialPartitionOffsets` requests the metadata for the collection of TopicPartitions and their offsets.

If the metadata could not be found, `initialPartitionOffsets` creates a new `KafkaSourceOffset` per [KafkaOffsetRangeLimit](#):

- For `EarliestOffsetRangeLimit`, `initialPartitionOffsets` requests the `KafkaOffsetReader` to [fetchEarliestOffsets](#)
- For `LatestOffsetRangeLimit`, `initialPartitionOffsets` requests the `KafkaOffsetReader` to [fetchLatestOffsets](#)
- For `SpecificOffsetRangeLimit`, `initialPartitionOffsets` requests the `KafkaOffsetReader` to [fetchSpecificOffsets](#) (and report a data loss per the `failOnDataLoss` flag)

`initialPartitionOffsets` requests the custom `HDFSMetadataLog` to [add the offsets to the metadata log](#) (as the metadata of the `0` th batch).

`initialPartitionOffsets` prints out the following INFO message to the logs:

```
Initial offsets: [offsets]
```

`initialPartitionOffsets` is used when `KafkaSource` is requested for the following:

Note

- Fetch offsets (from metadata log or Kafka directly)
- Generate a DataFrame with records from Kafka for a streaming batch (when the start offsets are not defined, i.e. before `StreamExecution` [commits the first streaming batch](#) and so nothing is in `committedOffsets` registry for a `KafkaSource` data source yet)

HDFSMetadataLog.serialize

```
serialize(metadata: KafkaSourceOffset, out: OutputStream): Unit
```

Note

`serialize` is part of the [HDFSMetadataLog Contract](#) to...FIXME.

`serialize` requests the `OutputStream` to write a zero byte (to support Spark 2.1.0 as per SPARK-19517).

`serialize` creates a `BufferedWriter` over a `OutputStreamWriter` over the `OutputStream` (with `UTF_8` charset encoding).

`serialize` requests the `BufferedWriter` to write the **v1** version indicator followed by a new line.

`serialize` then requests the `KafkaSourceOffset` for a JSON-serialized representation and the `BufferedWriter` to write it out.

In the end, `serialize` requests the `BufferedWriter` to flush (the underlying stream).

KafkaRelation

`KafkaRelation` represents a **collection of rows** with a [predefined schema](#) (`BaseRelation`) that supports [column pruning](#) (`TableScan`).

Tip	Read up on BaseRelation and TableScan in The Internals of Spark SQL book .
-----	--

`KafkaRelation` is [created](#) exclusively when `KafkaSourceProvider` is requested to [create a BaseRelation](#).

Table 1. KafkaRelation's Options

Name	Description
<code>kafkaConsumer.pollTimeoutMs</code>	Default: <code>spark.network.timeout</code> configuration if set or 120s

Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging levels for <code>org.apache.spark.sql.kafka010.KafkaRelation</code> to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.kafka010.KafkaRelation=DEBUG</pre> <p>Refer to Logging.</p>

Creating KafkaRelation Instance

`KafkaRelation` takes the following when created:

- `SQLContext`
- [ConsumerStrategy](#)
- `Source options (Map[String, String])`
- User-defined Kafka parameters (`Map[String, String]`)
- `failOnDataLoss` flag
- [Starting offsets](#)
- [Ending offsets](#)

getPartitionOffsets Internal Method

```
getPartitionOffsets(
    kafkaReader: KafkaOffsetReader,
    kafkaOffsets: KafkaOffsetRangeLimit): Map[TopicPartition, Long]
```

Caution

FIXME

Note

`getPartitionOffsets` is used exclusively when `KafkaRelation` builds RDD of rows (from the tuples).

Building Distributed Data Scan with Column Pruning — `buildScan` Method

```
buildScan(): RDD[Row]
```

Note

`buildScan` is part of the `TableScan` contract to build a distributed data scan with column pruning.

`buildScan` generates a unique group ID of the format **spark-kafka-relation-[randomUUID]** (to make sure that a streaming query creates a new consumer group).

`buildScan` creates a `KafkaOffsetReader` with the following:

- The given `ConsumerStrategy` and the `source options`
- `Kafka parameters for the driver` based on the given `specifiedKafkaParams`
- **spark-kafka-relation-[randomUUID]-driver** for the `driverGroupIdPrefix`

`buildScan` uses the `KafkaOffsetReader` to `getPartitionOffsets` for the starting and ending offsets (based on the given `KafkaOffsetRangeLimit` and the `KafkaOffsetRangeLimit`, respectively). `buildScan` requests the `KafkaOffsetReader` to `close` afterwards.

`buildScan` creates offset ranges (that are a collection of `KafkaSourceRDDOffsetRanges` with a `Kafka TopicPartition`, beginning and ending offsets and undefined preferred location).

`buildScan` prints out the following INFO message to the logs:

```
Generating RDD of offset ranges: [offsetRanges]
```

`buildScan` creates a `KafkaSourceRDD` with the following:

- `Kafka parameters for executors` based on the given `specifiedKafkaParams` and the unique group ID (`spark-kafka-relation-[randomUUID]`)

- The offset ranges created
- `pollTimeoutMs` configuration
- The given `failOnDataLoss` flag
- `reuseKafkaConsumer` flag off (`false`)

`buildScan` requests the `KafkaSourceRDD` to map `Kafka ConsumerRecords` to `InternalRows`.

In the end, `buildScan` requests the `SQLContext` to create a `DataFrame` (with the name `kafka` and the predefined `schema`) that is immediately converted to a `RDD[InternalRow]`.

`buildScan` throws a `IllegalStateException` when...FIXME

different topic partitions for starting offsets `topics[[fromTopics]]` and ending offset
s `topics[[untilTopics]]`

`buildScan` throws a `IllegalStateException` when...FIXME

[tp] doesn't have a from offset

KafkaSourceRDD

`KafkaSourceRDD` is an `RDD` of Kafka's `ConsumerRecords` (with keys and values being collections of bytes, i.e. `Array[Byte]`).

`KafkaSourceRDD` is created when:

- `KafkaRelation buildScan`
- `KafkaSource getBatch`

Placement Preferences of Partition (Preferred Locations)

— `getPreferredLocations` Method

Caution	FIXME
---------	-------

compute Method

Caution	FIXME
---------	-------

getPartitions Method

Caution	FIXME
---------	-------

persist Method

Caution	FIXME
---------	-------

Creating KafkaSourceRDD Instance

`KafkaSourceRDD` takes the following when created:

- `SparkContext`
- Collection of key-value settings for executors reading records from Kafka topics
- Collection of `KafkaSourceRDDOffsetRange` offsets
- Timeout (in milliseconds) to poll data from Kafka

Used when `KafkaSourceRDD` is requested for records (for given offsets) and in turn requests `CachedKafkaConsumer` to poll for Kafka's `ConsumerRecords`.

- Flag to...FIXME
- Flag to...FIXME

`KafkaSourceRDD` initializes the internal registries and counters.

CachedKafkaConsumer

Caution	FIXME
---------	-------

poll Internal Method

Caution	FIXME
---------	-------

fetchData Internal Method

Caution	FIXME
---------	-------

KafkaOffsetReader

`KafkaOffsetReader` relies on the [ConsumerStrategy](#) to create a Kafka Consumer.

`KafkaOffsetReader` creates a Kafka Consumer with **group.id**

(`ConsumerConfig.GROUP_ID_CONFIG`) configuration explicitly set to [nextGroupId](#) (i.e. the given [driverGroupIdPrefix](#) followed by [nextId](#)).

`KafkaOffsetReader` is created when:

- `KafkaRelation` is requested to build a distributed data scan with column pruning
- `KafkaSourceProvider` is requested to create a [KafkaSource](#), [createMicroBatchReader](#), and [createContinuousReader](#)

Table 1. KafkaOffsetReader's Options

Name	Description
<code>fetchOffset.numRetries</code>	Default: 3
<code>fetchOffset.retryIntervalMs</code>	How long to wait before retries Default: 1000

`KafkaOffsetReader` defines the [predefined fixed schema](#).

Table 2. KafkaOffsetReader's Internal Registries and Counters

Name	Description
_consumer	<p>Kafka's Consumer (<code>Consumer[Array[Byte], Array[Byte]]</code>)</p> <p>Initialized when <code>KafkaOffsetReader</code> is created.</p> <p>Used when <code>KafkaOffsetReader</code> :</p> <ul style="list-style-type: none"> • <code>fetchTopicPartitions</code> • fetches offsets for selected TopicPartitions • <code>fetchEarliestOffsets</code> • <code>fetchLatestOffsets</code> • <code>resetConsumer</code> • is closed
execContext	<code>scala.concurrent.ExecutionContextExecutorService</code>
groupId	
kafkaReaderThread	<code>java.util.concurrent.ExecutorService</code>
maxOffsetFetchAttempts	
nextId	Initially 0
offsetFetchAttemptIntervalMs	
Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging levels for <code>org.apache.spark.sql.kafka010.KafkaOffsetReader</code> to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.kafka010.KafkaOffsetReader=DEBUG</pre> <p>Refer to Logging.</p>

nextGroupId Internal Method

```
nextGroupId(): String
```

`nextGroupId` sets the `groupId` to be the `driverGroupIdPrefix`, - followed by the `nextId` (i.e. `[driverGroupIdPrefix]-[nextId]`).

In the end, `nextGroupId` increments the `nextId` and returns the `groupId`.

Note

`nextGroupId` is used exclusively when `KafkaOffsetReader` is requested for a [Kafka Consumer](#).

resetConsumer Internal Method

```
resetConsumer(): Unit
```

`resetConsumer` ...FIXME

Note

`resetConsumer` is used when...FIXME

fetchTopicPartitions Method

```
fetchTopicPartitions(): Set[TopicPartition]
```

Caution

FIXME

Note

`fetchTopicPartitions` is used when `KafkaRelation` [getPartitionOffsets](#).

Fetching Earliest Offsets — `fetchEarliestOffsets` Method

```
fetchEarliestOffsets(): Map[TopicPartition, Long]
```

```
fetchEarliestOffsets(newPartitions: Seq[TopicPartition]): Map[TopicPartition, Long]
```

Caution

FIXME

Note

`fetchEarliestOffsets` is used when `KafkaSource` [rateLimit](#) and [generates a DataFrame for a batch](#) (when new partitions have been assigned).

Fetching Latest Offsets — `fetchLatestOffsets` Method

```
fetchLatestOffsets(): Map[TopicPartition, Long]
```

Caution	FIXME
Note	<code>fetchLatestOffsets</code> is used when <code>KafkaSource</code> gets offsets or <code>initialPartitionOffsets</code> is initialized.

withRetriesWithoutInterrupt Internal Method

```
withRetriesWithoutInterrupt(  
    body: => Map[TopicPartition, Long]): Map[TopicPartition, Long]
```

`withRetriesWithoutInterrupt` ...FIXME

Note	<code>withRetriesWithoutInterrupt</code> is used when...FIXME
------	---

Creating KafkaOffsetReader Instance

`KafkaOffsetReader` takes the following when created:

- [ConsumerStrategy](#)
- Kafka parameters (as name-value pairs that are used exclusively to [create a Kafka consumer](#))
- Options (as name-value pairs)
- Prefix of the group ID

`KafkaOffsetReader` initializes the [internal registries and counters](#).

Fetching Offsets for Selected TopicPartitions — `fetchSpecificOffsets` Method

```
fetchSpecificOffsets(  
    partitionOffsets: Map[TopicPartition, Long],  
    reportDataLoss: String => Unit): KafkaSourceOffset
```

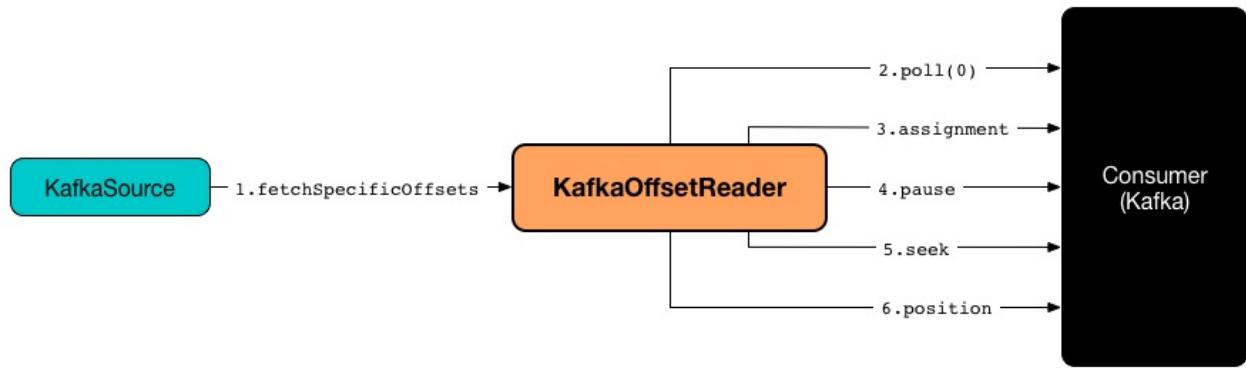


Figure 1. KafkaOffsetReader's `fetchSpecificOffsets`

`fetchSpecificOffsets` requests the [Kafka Consumer](#) to `poll(0)`.

`fetchSpecificOffsets` requests the [Kafka Consumer](#) for assigned partitions (using `Consumer.assignment()`).

`fetchSpecificOffsets` requests the [Kafka Consumer](#) to `pause(partitions)`.

You should see the following DEBUG message in the logs:

```
DEBUG KafkaOffsetReader: Partitions assigned to consumer: [partitions]. Seeking to [partitionOffsets]
```

For every partition offset in the input `partitionOffsets`, `fetchSpecificOffsets` requests the [Kafka Consumer](#) to:

- `seekToEnd` for the latest (aka `-1`)
- `seekToBeginning` for the earliest (aka `-2`)
- `seek` for other offsets

In the end, `fetchSpecificOffsets` creates a collection of Kafka's `TopicPartition` and `position` (using the [Kafka Consumer](#)).

Note	<code>fetchSpecificOffsets</code> is used when KafkaSource fetches and verifies initial partition offsets.
------	--

Creating Kafka Consumer— `createConsumer` Internal Method

```
createConsumer(): Consumer[Array[Byte], Array[Byte]]
```

`createConsumer` requests [ConsumerStrategy](#) to create a Kafka Consumer with `driverKafkaParams` and new generated `group.id` Kafka property.

Note

`createConsumer` is used when `KafkaOffsetReader` is created (and initializes consumer) and `resetConsumer`

Creating Kafka Consumer (Unless Already Available)

— `consumer` Method

```
consumer: Consumer[Array[Byte], Array[Byte]]
```

`consumer` gives the cached [Kafka Consumer](#) or creates one itself.

Note

Since `consumer` method is used (to access the internal [Kafka Consumer](#)) in the `fetch` methods that gives the property of creating a new Kafka Consumer whenever the internal [Kafka Consumer](#) reference become `null`, i.e. as in `resetConsumer`.

`consumer` ...FIXME

Note

`consumer` is used when `KafkaOffsetReader` is requested to `fetchTopicPartitions`, `fetchSpecificOffsets`, `fetchEarliestOffsets`, and `fetchLatestOffsets`.

Closing — `close` Method

```
close(): Unit
```

`close` [stop the Kafka Consumer](#) (if the [Kafka Consumer](#) is available).

`close` requests the `ExecutorService` to shut down.

Note

`close` is used when:

- `KafkaContinuousReader`, `KafkaMicroBatchReader`, and `KafkaSource` are requested to stop a streaming reader or source
- `KafkaRelation` is requested to [build a distributed data scan with column pruning](#)

runUninterruptibly Internal Method

```
runUninterruptibly[T](body: => T): T
```

`runUninterruptibly` ...FIXME

Note	runUninterruptibly is used when...FIXME
------	---

stopConsumer Internal Method

```
stopConsumer(): Unit
```

stopConsumer ...FIXME

Note	stopConsumer is used when...FIXME
------	-----------------------------------

toString Method

```
toString(): String
```

Note	toString is part of the java.lang.Object Contract for a string representation of the object.
------	--

toString ...FIXME

ConsumerStrategy Contract for KafkaConsumer Providers

`ConsumerStrategy` is the [contract](#) for components that can [create a KafkaConsumer](#) using the given Kafka parameters.

```
createConsumer(kafkaParams: java.util.Map[String, Object]): Consumer[Array[Byte], Array[Byte]]
```



Table 1. Available ConsumerStrategies

ConsumerStrategy	createConsumer		
AssignStrategy	Uses KafkaConsumer.assign(Collection<TopicPartition> partitions)		
SubscribeStrategy	Uses KafkaConsumer.subscribe(Collection<String> topics)		
SubscribePatternStrategy	Uses KafkaConsumer.subscribe(Pattern pattern, ConsumerRebalanceListener listener) with <code>NoOpConsumerRebalanceListener</code> . <table border="1"> <tr> <td style="padding: 5px;">Tip</td> <td style="padding: 5px;">Refer to java.util.regex.Pattern for the format of supported topic subscription regex patterns.</td> </tr> </table>	Tip	Refer to java.util.regex.Pattern for the format of supported topic subscription regex patterns.
Tip	Refer to java.util.regex.Pattern for the format of supported topic subscription regex patterns.		

KafkaSourceOffset

`KafkaSourceOffset` is an [Offset](#) that...FIXME

`KafkaSourceOffset` takes a collection of Kafka `TopicPartitions` and their offsets when created.

Creating KafkaSourceOffset Instance

Caution	FIXME
---------	-------

Getting Partition Offsets — `getPartitionOffsets` Method

```
getPartitionOffsets(offset: Offset): Map[TopicPartition, Long]
```

`getPartitionOffsets` takes [KafkaSourceOffset.partitionToOffsets](#) from `offset`.

If `offset` is `KafkaSourceOffset`, `getPartitionOffsets` takes the partitions and offsets straight from it.

If however `offset` is `SerializedOffset`, `getPartitionOffsets` deserializes the offsets from JSON.

`getPartitionOffsets` reports an `IllegalArgumentException` when `offset` is neither `KafkaSourceOffset` or `SerializedOffset`.

```
Invalid conversion from offset of [class] to KafkaSourceOffset
```

Note	<code>getPartitionOffsets</code> is used exclusively when <code>KafkaSource</code> generates a DataFrame with records from Kafka for a batch .
------	--

KafkaSink

`KafkaSink` is a [streaming sink](#) that [KafkaSourceProvider](#) registers as the `kafka` format.

```
// start spark-shell or a Spark application with spark-sql-kafka-0-10 module
// spark-shell --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0-SNAPSHOT
import org.apache.spark.sql.SparkSession
val spark: SparkSession = ...
spark.
  readStream.
  format("text").
  load("server-logs/*.out").
  as[String].
  writeStream.
  queryName("server-logs processor").
  format("kafka"). // <-- uses KafkaSink
  option("topic", "topic1").
  option("checkpointLocation", "/tmp/kafka-sink-checkpoint"). // <-- mandatory
  start

// in another terminal
$ echo hello > server-logs/hello.out

// in the terminal with Spark
FIXME
```

Creating KafkaSink Instance

`KafkaSink` takes the following when created:

- `SQLContext`
- Kafka parameters (used on executor) as a map of `(String, Object)` pairs
- Optional topic name

addBatch Method

```
addBatch(batchId: Long, data: DataFrame): Unit
```

Internally, `addBatch` requests `KafkaWriter` to write the input `data` to the [topic](#) (if defined) or a topic in [executorKafkaParams](#).

Note	<code>addBatch</code> is a part of Sink Contract to "add" a batch of data to the sink.
------	--

KafkaOffsetRangeLimit — Desired Offset Range Limits

`KafkaOffsetRangeLimit` represents the desired offset range limits for starting, ending, and specific offsets in [Kafka Data Source](#).

Table 1. KafkaOffsetRangeLimits

KafkaOffsetRangeLimit	Description
<code>EarliestOffsetRangeLimit</code>	Intent to bind to the earliest offset
<code>LatestOffsetRangeLimit</code>	Intent to bind to the latest offset
<code>SpecificOffsetRangeLimit</code>	Intent to bind to specific offsets with the following special offset "magic" numbers: <ul style="list-style-type: none"> • <code>-1</code> or <code>KafkaOffsetRangeLimit.LATEST</code> - the latest offset • <code>-2</code> or <code>KafkaOffsetRangeLimit.EARLIEST</code> - the earliest offset

Note

`KafkaOffsetRangeLimit` is a **sealed trait** in Scala which means that [implementations](#) are all in the same compilation unit (a single file).

`KafkaOffsetRangeLimit` is often used in a text-based representation and is converted to from **latest**, **earliest** or a **JSON-formatted text** using [KafkaSourceProvider.getKafkaOffsetRangeLimit](#) object method.

Note

A JSON-formatted text is of the following format `{"topicName": {"partition":offset},...}` , e.g. `{"topicA":{"0":23,"1":-1}, "topicB":{"0":-2}}` .

`KafkaOffsetRangeLimit` is used when:

- [KafkaContinuousReader](#) is created (with the [initial offsets](#))
- [KafkaMicroBatchReader](#) is created (with the [starting offsets](#))
- [KafkaRelation](#) is created (with the [starting](#) and [ending](#) offsets)
- [KafkaSource](#) is created (with the [starting offsets](#))
- `KafkaSourceProvider` is requested to [convert configuration options](#) to [KafkaOffsetRangeLimits](#)

KafkaContinuousReader—ContinuousReader for Kafka Data Source in Continuous Stream Processing

`KafkaContinuousReader` is a [ContinuousReader](#) for [Kafka Data Source](#) in [Continuous Stream Processing](#).

`KafkaContinuousReader` is [created](#) exclusively when `KafkaSourceProvider` is requested to [create a ContinuousReader](#).

`KafkaContinuousReader` uses **kafkaConsumer.pollTimeoutMs** configuration parameter (default: `512`) for [KafkaContinuousInputPartitions](#) when requested to [planInputPartitions](#).

<p>Tip</p>	<p>Enable <code>INFO</code> or <code>WARN</code> logging levels for <code>org.apache.spark.sql.kafka010.KafkaContinuousReader</code> to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.kafka010.KafkaContinuousReader=INFO</pre> <p>Refer to Logging.</p>
-------------------	--

Creating KafkaContinuousReader Instance

`KafkaContinuousReader` takes the following to be created:

- [KafkaOffsetReader](#)
- Kafka parameters (as `java.util.Map[String, Object]`)
- Source options (as `Map[String, String]`)
- Metadata path
- [Initial offsets](#)
- `failOnDataLoss` flag

planInputPartitions Method

```
planInputPartitions(): java.util.List[InputPartition[InternalRow]]
```

Note	planInputPartitions is part of the DataSourceReader contract to...FIXME.
------	--

planInputPartitions ...FIXME

KafkaMicroBatchReader

`KafkaMicroBatchReader` is a [MicroBatchReader](#) for `kafka` data source in [Micro-Batch Stream Processing](#).

`KafkaMicroBatchReader` is [created](#) exclusively when `KafkaSourceProvider` is requested to [create a MicroBatchReader](#).

`KafkaMicroBatchReader` uses the [DataSourceOptions](#) to access the `kafkaConsumer.pollTimeoutMs` option (default: `spark.network.timeout` or `120s`).

`KafkaMicroBatchReader` uses the [DataSourceOptions](#) to access the `maxOffsetsPerTrigger` option (default: `(undefined)`).

`KafkaMicroBatchReader` uses the [Kafka properties for executors](#) to create `KafkaMicroBatchInputPartitions` when requested to [planInputPartitions](#).

Table 1. KafkaMicroBatchReader's Internal Properties (e.g. Registries, Counters and Flags)

Name	Description
<code>endPartitionOffsets</code>	Ending offsets for the assigned partitions (<code>Map[TopicPartition, Long]</code>) Used when...FIXME
<code>rangeCalculator</code>	KafkaOffsetRangeCalculator (for the given DataSourceOptions) Used exclusively when <code>KafkaMicroBatchReader</code> is requested to planInputPartitions (to calculate offset ranges)
<code>startPartitionOffsets</code>	Starting offsets for the assigned partitions (<code>Map[TopicPartition, Long]</code>) Used when...FIXME

Tip	Enable <code>WARN</code> , <code>INFO</code> or <code>DEBUG</code> logging levels for <code>org.apache.spark.sql.kafka010.KafkaMicroBatchReader</code> to see what happens inside. Add the following line to <code>conf/log4j.properties</code> : <code>log4j.logger.org.apache.spark.sql.kafka010.KafkaMicroBatchReader=DEBUG</code>
	Refer to Logging .

Creating KafkaMicroBatchReader Instance

`KafkaMicroBatchReader` takes the following to be created:

- `KafkaOffsetReader`
- Kafka properties for executors (`Map[String, Object]`)
- `DataSourceOptions`
- Metadata Path
- Desired starting `KafkaOffsetRangeLimit`
- `failOnDataLoss` option

`KafkaMicroBatchReader` initializes the internal registries and counters.

readSchema Method

```
readSchema(): StructType
```

Note	<code>readSchema</code> is part of the <code>DataSourceReader</code> contract to...FIXME.
------	---

`readSchema` simply returns the predefined fixed schema.

Stopping Streaming Reader— stop Method

```
stop(): Unit
```

Note	<code>stop</code> is part of the <code>BaseStreamingSource Contract</code> to stop a streaming reader.
------	--

`stop` simply requests the `KafkaOffsetReader` to close.

planInputPartitions Method

```
planInputPartitions(): java.util.List[InputPartition[InternalRow]]
```

Note	<code>planInputPartitions</code> is part of the <code>DataSourceReader</code> contract to...FIXME.
------	--

`planInputPartitions` first finds the new partitions (`TopicPartitions` that are in the `endPartitionOffsets` but not in the `startPartitionOffsets`) and requests the `KafkaOffsetReader` to fetch their earliest offsets.

`planInputPartitions` prints out the following INFO message to the logs:

```
Partitions added: [newPartitionInitialOffsets]
```

`planInputPartitions` then prints out the following DEBUG message to the logs:

```
TopicPartitions: [comma-separated list of TopicPartitions]
```

`planInputPartitions` requests the [KafkaOffsetRangeCalculator](#) for [offset ranges](#) (given the `startPartitionOffsets` and the newly-calculated `newPartitionInitialOffsets` as the `fromOffsets`, the `endPartitionOffsets` as the `untilOffsets`, and the [available executors](#) (sorted in descending order)).

In the end, `planInputPartitions` creates a `KafkaMicroBatchInputPartition` for every offset range (with the [Kafka properties for executors](#), the `pollTimeoutMs`, the `failOnDataLoss` flag and whether to reuse a Kafka consumer among Spark tasks).

Note

A `KafkaMicroBatchInputPartition` uses a shared Kafka consumer only when all the offset ranges have distinct `TopicPartitions`, so concurrent tasks (of a stage in a Spark job) will not interfere and read the same `TopicPartitions`.

`planInputPartitions` [reports data loss](#) when...FIXME

Available Executors in Spark Cluster (Sorted By Host and Executor ID in Descending Order)

— `getSortedExecutorList` Internal Method

```
getSortedExecutorList(): Array[String]
```

`getSortedExecutorList` requests the `BlockManager` to request the `BlockManagerMaster` to get the peers (the other nodes in a Spark cluster), creates a `ExecutorCacheTaskLocation` for every pair of host and executor ID, and in the end, sort it in descending order.

Note

`getSortedExecutorList` is used exclusively when `KafkaMicroBatchReader` is requested to `planInputPartitions` (and calculates offset ranges).

`getOrCreateInitialPartitionOffsets` Internal Method

```
getOrCreateInitialPartitionOffsets(): PartitionOffsetMap
```

`getOrCreateInitialPartitionOffsets ...FIXME`

Note

`getOrCreateInitialPartitionOffsets` is used when...FIXME

KafkaOffsetRangeCalculator

`KafkaOffsetRangeCalculator` is created when `KafkaMicroBatchReader` is created with the only purpose of calculating offset ranges (when `kafkaMicroBatchReader` is requested to `planInputPartitions`).

`KafkaOffsetRangeCalculator` takes an optional **minimum number of partitions per executor** (`minPartitions`) to be created (that can either be undefined or greater than `0`).

When created with a `DataSourceOptions`, `KafkaOffsetRangeCalculator` uses `minPartitions` option for the **minimum number of partitions per executor**.

Calculating Offset Ranges — `getRanges` Method

```
getRanges(
  fromOffsets: PartitionOffsetMap,
  untilOffsets: PartitionOffsetMap,
  executorLocations: Seq[String] = Seq.empty): Seq[KafkaOffsetRange]
```

`getRanges` finds the common `TopicPartitions` (i.e. the `TopicPartitions` included in the given `untilOffsets` and `fromOffsets`).

For every `TopicPartition`, `getRanges` creates a `KafkaOffsetRange` (with the `preferredLoc` undefined). `getRanges` filters out the `TopicPartitions` that have no records to consume.

At this point, `getRanges` knows the `TopicPartitions` with records to consume.

For the `minPartitions` undefined or smaller than the number of `KafkaOffsetRanges` (i.e. `TopicPartitions` to consume records from), `getRanges` finds the preferred executor for a `TopicPartition` (and the given `executorLocations`).

Otherwise (with the `minPartitions` defined and greater than the number of `KafkaOffsetRanges`), `getRanges` ...FIXME

Note

`getRanges` is used exclusively when `KafkaMicroBatchReader` is requested to `planInputPartitions`.

KafkaOffsetRange Class

`KafkaOffsetRange` is a case class with the following attributes:

- `TopicPartition`

- `fromOffset offset`
- `untilOffset offset`
- Optional preferred location

`KafkaOffsetRange` knows the size, i.e. the number of records between the `untilOffset` and `fromOffset` offsets.

Selecting Preferred Executor for TopicPartition

— `getLocation` Internal Method

```
getLocation(  
    tp: TopicPartition,  
    executorLocations: Seq[String]): Option[String]
```

`getLocation` ...FIXME

Note

`getLocation` is used exclusively when `KafkaOffsetRangeCalculator` is requested to calculate offset ranges.

KafkaContinuousInputPartition

KafkaContinuousInputPartition is...FIXME

KafkaSourceInitialOffsetWriter

`KafkaSourceInitialOffsetWriter` is...FIXME

TextSocketSourceProvider

`TextSocketSourceProvider` is a [StreamSourceProvider](#) for [TextSocketSource](#) that read records from `host` and `port`.

`TextSocketSourceProvider` is a [DataSourceRegister](#), too.

The short name of the data source is `socket`.

It requires two mandatory options (that you can set using `option` method):

1. `host` which is the host name.
2. `port` which is the port number. It must be an integer.

`TextSocketSourceProvider` also supports [includeTimestamp](#) option that is a boolean flag that you can use to include timestamps in the schema.

includeTimestamp Option

Caution	FIXME
---------	-------

createSource

`createSource` grabs the two mandatory options — `host` and `port` — and returns an [TextSocketSource](#).

sourceSchema

`sourceSchema` returns `textSocket` as the name of the source and the schema that can be one of the two available schemas:

1. `SCHEMA_REGULAR` (default) which is a schema with a single `value` field of String type.
2. `SCHEMA_TIMESTAMP` when `includeTimestamp` flag option is set. It is not, i.e. `false`, by default. The schema are `value` field of `StringType` type and `timestamp` field of `TimestampType` type of format `yyyy-MM-dd HH:mm:ss`.

Tip	Read about schema .
-----	-------------------------------------

Internally, it starts by printing out the following WARN message to the logs:

```
WARN TextSocketSourceProvider: The socket source should not be used for production app  
lications! It does not support recovery and stores state indefinitely.
```

It then checks whether `host` and `port` parameters are defined and if not it throws a `AnalysisException`:

```
Set a host to read from with option("host", ...).
```

TextSocketSource

`TextSocketSource` is a [streaming source](#) that reads lines from a socket at the `host` and `port` (defined by parameters).

It uses `lines` internal in-memory buffer to keep all of the lines that were read from a socket forever.

Caution	This source is not for production use due to design constraints, e.g. infinite in-memory collection of lines read and no fault recovery. It is designed only for tutorials and debugging.
---------	---

```

import org.apache.spark.sql.SparkSession
val spark: SparkSession = SparkSession.builder.getOrCreate()

// Connect to localhost:9999
// You can use "nc -lk 9999" for demos
val textSocket = spark.
  readStream.
  format("socket").
  option("host", "localhost").
  option("port", 9999).
  load

import org.apache.spark.sql.Dataset
val lines: Dataset[String] = textSocket.as[String].map(_.toUpperCase)

val query = lines.writeStream.format("console").start

// Start typing the lines in nc session
// They will appear UPPERCASE in the terminal

-----
Batch: 0
-----
+---+
|   value|
+---+
|UPPERCASE|
+---+

scala> query.explain
== Physical Plan ==
*SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0, java.lang.String, true], true) AS value#21]
+- *MapElements <function1>, obj#20: java.lang.String
  +- *DeserializeToObject value#43.toString, obj#19: java.lang.String
    +- LocalTableScan [value#43]

scala> query.stop

```

lines Internal Buffer

```
lines: ArrayBuffer[(String, Timestamp)]
```

`lines` is the internal buffer of all the lines `TextSocketSource` read from the socket.

Maximum Available Offset (getOffset method)

Note

`getOffset` is a part of the [Streaming Source Contract](#).

`TextSocketSource`'s offset can either be none or `Longoffset` of the number of lines in the internal `lines` buffer.

Schema (schema method)

`TextSocketSource` supports two [schemas](#):

1. A single `value` field of String type.
2. `value` field of `StringType` type and `timestamp` field of `TimestampType` type of format `yyyy-MM-dd HH:mm:ss`.

Tip

Refer to [sourceSchema](#) for `TextSocketSourceProvider`.

Creating TextSocketSource Instance

```
TextSocketSource(  
    host: String,  
    port: Int,  
    includeTimestamp: Boolean,  
    sqlContext: SQLContext)
```

When `TextSocketSource` is created (see [TextSocketSourceProvider](#)), it gets 4 parameters passed in:

1. `host`
2. `port`
3. [includeTimestamp](#) flag
4. [SQLContext](#)

Caution

It appears that the source did not get "renewed" to use [SparkSession](#) instead.

It opens a socket at given `host` and `port` parameters and reads a buffering character-input stream using the default charset and the default-sized input buffer (of `8192` bytes) line by line.

Caution

FIXME Review Java's `Charset.defaultCharset()`

It starts a `readThread` daemon thread (called `TextSocketSource(host, port)`) to read lines from the socket. The lines are added to the internal `lines` buffer.

Stopping TextSocketSource (stop method)

When stopped, `TextSocketSource` closes the socket connection.

RateSourceProvider

`RateSourceProvider` is a [StreamSourceProvider](#) for [RateStreamSource](#) (that acts as the source for **rate** format).

Note	<code>RateSourceProvider</code> is also a <code>DataSourceRegister</code> .
------	---

The short name of the data source is **rate**.

RateStreamSource

`RateStreamSource` is a [streaming source](#) that generates [consecutive numbers](#) with [timestamp](#) that can be useful for testing and PoCs.

`RateStreamSource` is created for **rate** format (that is registered by [RateSourceProvider](#)).

```
val rates = spark
  .readStream
  .format("rate") // <-- use RateStreamSource
  .option("rowsPerSecond", 1)
  .load
```

Table 1. RateStreamSource's Options

Name	Default Value	Description
<code>numPartitions</code>	(default parallelism)	Number of partitions to use
<code>rampUpTime</code>	0 (seconds)	
<code>rowsPerSecond</code>	1	Number of rows to generate per second (has to be greater than 0)

`RateStreamSource` uses a predefined schema that cannot be changed.

```
val schema = rates.schema
scala> println(schema.treeString)
root
| -- timestamp: timestamp (nullable = true)
| -- value: long (nullable = true)
```

Table 2. RateStreamSource's Dataset Schema (in the positional order)

Name	Type
<code>timestamp</code>	<code>TimestampType</code>
<code>value</code>	<code>LongType</code>

Table 3. RateStreamSource's Internal Registries and Counters

Name	Description
clock	
lastTimeMs	
maxSeconds	
startTimeMs	

Tip	<p>Enable <code>INFO</code> or <code>DEBUG</code> logging levels for <code>org.apache.spark.sql.execution.streaming.RateStreamSource</code> to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.execution.streaming.RateStreamSource=DEBUG</pre> <p>Refer to Logging.</p>

Getting Maximum Available Offsets — `getOffset` Method

```
getOffset: Option[Offset]
```

Note	<code>getOffset</code> is a part of the Source Contract .
Caution	FIXME

Generating DataFrame for Streaming Batch — `getBatch` Method

```
getBatch(start: Option[Offset], end: Offset): DataFrame
```

Note	<code>getBatch</code> is a part of Source Contract .
------	--

Internally, `getBatch` calculates the seconds to start from and end at (from the input `start` and `end` offsets) or assumes `0`.

`getBatch` then calculates the values to generate for the start and end seconds.

You should see the following DEBUG message in the logs:

```
DEBUG RateStreamSource: startSeconds: [startSeconds], endSeconds: [endSeconds], rangeStart: [rangeStart], rangeEnd: [rangeEnd]
```

If the start and end ranges are equal, `getBatch` creates an empty `DataFrame` (with the [schema](#)) and returns.

Otherwise, when the ranges are different, `getBatch` creates a `DataFrame` using `SparkContext.range` operator (for the start and end ranges and [numPartitions](#) partitions).

Creating RateStreamSource Instance

`RateStreamSource` takes the following when created:

- `SQLContext`
- Path to the metadata
- Rows per second
- RampUp time in seconds
- Number of partitions
- Flag to whether to use `ManualClock` (`true`) or `SystemClock` (`false`)

`RateStreamSource` initializes the [internal registries and counters](#).

RateStreamMicroBatchReader

RateStreamMicroBatchReader is...FIXME

ConsoleSinkProvider

`ConsoleSinkProvider` is a `DataSourceV2` with `StreamWriterSupport` for **console** data source format

Tip	Read up on DataSourceV2 Contract in The Internals of Spark SQL book.
-----	--

`ConsoleSinkProvider` is a `DataSourceRegister` and registers itself as the **console** data source format.

```
import org.apache.spark.sql.streaming.Trigger
val q = spark
  .readStream
  .format("rate")
  .load
  .writeStream
  .format("console") // <-- requests ConsoleSinkProvider for a sink
  .trigger(Trigger.Once)
  .start
scala> println(q.lastProgress.sink)
{
  "description" : "org.apache.spark.sql.execution.streaming.ConsoleSinkProvider@2392cf
b1"
}
```

When requested for a `StreamWriter`, `ConsoleSinkProvider` simply creates a `ConsoleWriter` (with the given schema and options).

`ConsoleSinkProvider` is a [CreatableRelationProvider](#).

Tip	Read up on CreatableRelationProvider in The Internals of Spark SQL book.
-----	--

createRelation Method

```
createRelation(
  sqlContext: SQLContext,
  mode: SaveMode,
  parameters: Map[String, String],
  data: DataFrame): BaseRelation
```

Note	<code>createRelation</code> is part of the <code>CreatableRelationProvider</code> Contract to support writing a structured query (a <code>DataFrame</code>) per save mode.
------	---

`createRelation ...FIXME`

ConsoleWriter

ConsoleWriter is a [StreamWriter](#) for **console** data source format.

ForeachWriterProvider

ForeachWriterProvider is...FIXME

ForeachWriter

`ForeachWriter` is the [contract](#) for a **foreach writer** that is a [streaming format](#) that controls streaming writes.

Note	<code>ForeachWriter</code> is set using foreach operator.
------	---

```
val foreachWriter = new ForeachWriter[String] { ... }
streamingQuery.
writeStream.
foreach(foreachWriter).
start
```

ForeachWriter Contract

```
package org.apache.spark.sql

abstract class ForeachWriter[T] {
  def open(partitionId: Long, version: Long): Boolean
  def process(value: T): Unit
  def close(errorOrNull: Throwable): Unit
}
```

Table 1. ForeachWriter Contract

Method	Description
<code>open</code>	Used when...
<code>process</code>	Used when...
<code>close</code>	Used when...

ForeachSink

`ForeachSink` is a typed [streaming sink](#) that passes rows (of the type `T`) to [ForeachWriter](#) (one record at a time per partition).

Note	<code>ForeachSink</code> is assigned a <code>ForeachWriter</code> when <code>DataStreamWriter</code> is started .
------	---

`ForeachSink` is used exclusively in [foreach](#) operator.

```
val records = spark.  
  readStream  
  format("text").  
  load("server-logs/*.out").  
  as[String]  
  
import org.apache.spark.sql.ForeachWriter  
val writer = new ForeachWriter[String] {  
  override def open(partitionId: Long, version: Long) = true  
  override def process(value: String) = println(value)  
  override def close(errorOrNull: Throwable) = {}  
}  
  
records.writeStream  
  .queryName("server-logs processor")  
  .foreach(writer)  
  .start
```

Internally, `addBatch` (the only method from the [Sink Contract](#)) takes records from the input [DataFrame](#) (as `data`), transforms them to expected type `T` (of this `ForeachSink`) and (now as a [Dataset](#)) processes each partition.

addBatch(batchId: Long, data: DataFrame): Unit
--

`addBatch` then opens the constructor's [ForeachWriter](#) (for the [current partition](#) and the input batch) and passes the records to process (one at a time per partition).

Caution	FIXME Why does Spark track whether the writer failed or not? Why couldn't it <code>finally</code> and do <code>close</code> ?
---------	---

Caution	FIXME Can we have a constant for "foreach" for <code>source</code> in <code>DataStreamWriter</code> ?
---------	---

ForeachBatchSink

`ForeachBatchSink` is a [streaming sink](#) that is used for the `foreachBatch` source.

`ForeachBatchSink` is created exclusively when `DataStreamWriter` is requested to [start execution of the streaming query](#) (with the `foreachBatch` source).

`ForeachBatchSink` uses **ForeachBatchSink** name.

```
import org.apache.spark.sql.Dataset
val q = spark.readStream
  .format("rate")
  .load
  .writeStream
  .foreachBatch { (output: Dataset[_], batchId: Long) => // <- creates a ForeachBatch
Sink
  println(s"Batch ID: $batchId")
  output.show
}
.start
// q.stop

scala> println(q.lastProgress.sink.description)
ForeachBatchSink
```

Note

`ForeachBatchSink` was added in Spark 2.4.0 as part of [SPARK-24565 Add API for in Structured Streaming for exposing output rows of each microbatch as a DataFrame](#).

Creating ForeachBatchSink Instance

`ForeachBatchSink` takes the following when created:

- Batch writer (`(Dataset[T], Long) ⇒ Unit`)
- Encoder (`ExpressionEncoder[T]`)

Adding Batch — `addBatch` Method

```
addBatch(batchId: Long, data: DataFrame): Unit
```

Note

`addBatch` is a part of [Sink Contract](#) to "add" a batch of data to the sink.

addBatch ...FIXME

MemorySinkV2

MemorySinkV2 is...FIXME

MemorySink

`MemorySink` is a streaming [Sink](#) that stores records in memory. It is particularly useful for testing.

`MemorySink` is used for `memory` format and requires a query name (by `queryName` method or `queryName` option).

```
val spark: SparkSession = ???  
val logs = spark.readStream.textFile("logs/*.out")  
  
scala> val outStream = logs.writeStream  
    .format("memory")  
    .queryName("logs")  
    .start()  
outStream: org.apache.spark.sql.streaming.StreamingQuery = org.apache.spark.sql.execution.streaming.StreamingQueryWrapper@690337df  
  
scala> sql("select * from logs").show(truncate = false)
```

Note

`MemorySink` was introduced in the pull request for [SPARK-14288][SQL] [Memory Sink for streaming](#).

Use `toDebugString` to see the batches.

Its aim is to allow users to test streaming applications in the Spark shell or other local tests.

You can set `checkpointLocation` using `option` method or it will be set to `spark.sql.streaming.checkpointLocation` property.

If `spark.sql.streaming.checkpointLocation` is set, the code uses `$location/$queryName` directory.

Finally, when no `spark.sql.streaming.checkpointLocation` is set, a temporary directory `memory.stream` under `java.io.tmpdir` is used with `offsets` subdirectory inside.

Note

The directory is cleaned up at shutdown using `ShutdownHookManager.registerShutdownDeleteDir`.

It creates `MemorySink` instance based on the schema of the DataFrame it operates on.

It creates a new DataFrame using `MemoryPlan` with `MemorySink` instance created earlier and registers it as a temporary table (using `DataFrame.registerTempTable` method).

Note	At this point you can query the table as if it were a regular non-streaming table using <code>sql</code> method.
------	--

A new `StreamingQuery` is started (using `StreamingQueryManager.startQuery`) and returned.

Table 1. MemorySink's Internal Registries and Counters

Name	Description
batches	FIXME Used when...FIXME

Tip	Enable <code>DEBUG</code> logging level for <code>org.apache.spark.sql.execution.streaming.MemorySink</code> logger to see what happens in <code>MemorySink</code> .
-----	--

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.execution.streaming.MemorySink=DEBUG
```

Refer to [Logging](#).

addBatch Method

```
addBatch(batchId: Long, data: DataFrame): Unit
```

`addBatch` checks if `batchId` has already been committed (i.e. added to `batches` internal registry).

If `batchId` was already committed, you should see the following DEBUG message in the logs:

```
DEBUG Skipping already committed batch: [batchId]
```

Otherwise, if the `batchId` is not already committed, you should see the following DEBUG message in the logs:

```
DEBUG Committing batch [batchId] to [this]
```

For `Append` and `Update` output modes, `addBatch` collects records from `data` and registers `batchId` (i.e. adds to `batches` internal registry).

Note

`addBatch` uses `collect` operator to collect records. It is when the records are "downloaded" to the driver's memory.

For `Complete` output mode, `addBatch` collects records (as for the other output modes), but before registering `batchId` clears `batches` internal registry.

When the output mode is invalid, `addBatch` reports a `IllegalArgumentException` with the following error message.

```
Output mode [outputMode] is not supported by MemorySink
```

Note

`addBatch` is a part of [Sink Contract](#) to "add" a batch of data to the sink.

MemoryStream

`MemoryStream` is a streaming [Source](#) that produces values to memory.

`MemoryStream` uses the internal [batches](#) collection of [datasets](#).

Caution

This source is **not** for production use due to design constraints, e.g. infinite in-memory collection of lines read and no fault recovery.

`MemoryStream` is designed primarily for unit tests, tutorials and debugging.

```
val spark: SparkSession = ???  
  
implicit val ctx = spark.sqlContext  
  
import org.apache.spark.sql.execution.streaming.MemoryStream  
// It uses two implicits: Encoder[Int] and SQLContext  
val intsIn = MemoryStream[Int]  
  
val ints = intsIn.toDF  
.withColumn("t", current_timestamp())  
.withWatermark("t", "5 minutes")  
.groupBy(window($"t", "5 minutes") as "window")  
.agg(count("*") as "total")  
  
import org.apache.spark.sql.streaming.{OutputMode, Trigger}  
import scala.concurrent.duration._  
val totalsOver5mins = ints.  
writeStream.  
format("memory").  
queryName("totalsOver5mins").  
outputMode(OutputMode.Append).  
trigger(Trigger.ProcessingTime(10.seconds)).  
start  
  
scala> val zeroOffset = intsIn.addData(0, 1, 2)  
zeroOffset: org.apache.spark.sql.execution.streaming.Offset = #0  
  
totalsOver5mins.processAllAvailable()  
spark.table("totalsOver5mins").show  
  
scala> intsOut.show  
+---+  
|value|  
+---+  
| 0|  
| 1|  
| 2|  
+---+  
  
memoryQuery.stop()
```

```

17/02/28 20:06:01 DEBUG StreamExecution: Starting Trigger Calculation
17/02/28 20:06:01 DEBUG StreamExecution: getOffset took 0 ms
17/02/28 20:06:01 DEBUG StreamExecution: triggerExecution took 0 ms
17/02/28 20:06:01 DEBUG StreamExecution: Execution stats: ExecutionStats(Map(),List(),
Map(watermark -> 1970-01-01T00:00:00.000Z))
17/02/28 20:06:01 INFO StreamExecution: Streaming query made progress: {
  "id" : "ec5adda-0e46-4c3c-b2c2-604a854ee19a",
  "runId" : "d850cab-94d0-4931-8a2d-e054086e39c3",
  "name" : "totalsOver5mins",
  "timestamp" : "2017-02-28T19:06:01.175Z",
  "numInputRows" : 0,
  "inputRowsPerSecond" : 0.0,
  "durationMs" : {
    "getOffset" : 0,
    "triggerExecution" : 0
  },
  "eventTime" : {
    "watermark" : "1970-01-01T00:00:00.000Z"
  },
  "stateOperators" : [ ],
  "sources" : [ {
    "description" : "MemoryStream[value#1]",
    "startOffset" : null,
    "endOffset" : null,
    "numInputRows" : 0,
    "inputRowsPerSecond" : 0.0
  }],
  "sink" : {
    "description" : "MemorySink"
  }
}

```

Tip

Enable `DEBUG` logging level for
`org.apache.spark.sql.execution.streaming.MemoryStream` logger to see what happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.execution.streaming.MemoryStream=DEBUG
```

Refer to [Logging](#).

Creating MemoryStream Instance

```
apply[A : Encoder](implicit sqlContext: SQLContext): MemoryStream[A]
```

`MemoryStream` object defines `apply` method that you can use to create instances of `MemoryStream` streaming sources.

Adding Data to Source (addData methods)

```
addData(data: A*): Offset
addData(data: TraversableOnce[A]): Offset
```

`addData` methods add the input `data` to `batches` internal collection.

When executed, `addData` adds a `DataFrame` (created using `toDS` implicit method) and increments the internal `currentOffset` offset.

You should see the following DEBUG message in the logs:

```
DEBUG MemoryStream: Adding ds: [ds]
```

Generating Next Streaming Batch — `getBatch` Method

Note	<code>getBatch</code> is a part of Streaming Source contract .
------	--

When executed, `getBatch` uses the internal `batches` collection to return requested offsets.

You should see the following DEBUG message in the logs:

```
DEBUG MemoryStream: MemoryBatch [[startOrdinal], [endOrdinal]]: [newBlocks]
```

StreamingExecutionRelation Logical Plan

`MemoryStream` uses `StreamingExecutionRelation` logical plan to build `Datasets` or `DataFrames` when requested.

```
scala> val ints = MemoryStream[Int]
ints: org.apache.spark.sql.execution.streaming.MemoryStream[Int] = MemoryStream[value#13]

scala> ints.toDS.queryExecution.logical.isStreaming
res14: Boolean = true

scala> ints.toDS.queryExecution.logical
res15: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan = MemoryStream[value#13]
```

Schema (schema method)

`MemoryStream` works with the data of the [schema](#) as described by the [Encoder](#) (of the [Dataset](#)).

Micro-Batch Stream Processing (Structured Streaming V1)

Micro-Batch Stream Processing is a stream processing model in Spark Structured Streaming (often referred as **Structured Streaming V1**) that is used for [Trigger.Once](#) and [Trigger.ProcessingTime](#) triggers.

Micro-batch stream processing uses [MicroBatchExecution](#) stream execution engine and [MicroBatchReadSupport](#) data sources.

```
import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._

val sq = spark
  .readStream
  .format("rate")
  .load
  .writeStream
  .format("console")
  .option("truncate", false)
  .trigger(Trigger.ProcessingTime(1.minute)) // <-- Uses MicroBatchExecution for execution
  .queryName("rate2console")
  .start

assert(sq.isActive)

scala> sq.explain
== Physical Plan ==
WriteToDataSourceV2 org.apache.spark.sql.execution.streaming.sources.MicroBatchWriter@678e6267
+- *(1) Project [timestamp#54, value#55L]
  +- *(1) ScanV2 rate[timestamp#54, value#55L]

// sq.stop
```

MicroBatchExecution — Stream Execution Engine of Micro-Batch Stream Processing

`MicroBatchExecution` is a [stream execution engine](#) of Micro-Batch Stream Processing.

`MicroBatchExecution` is [created](#) when `StreamingQueryManager` is requested to [create a streaming query](#) (when `DataStreamWriter` is requested to [start an execution of the streaming query](#)) with the following:

- Any type of [sink](#) but [StreamWriterSupport](#)
- Any type of [trigger](#) but [ContinuousTrigger](#)

```
import org.apache.spark.sql.streaming.Trigger
val query = spark
  .readStream
  .format("rate")
  .load
  .writeStream
  .format("console")
  .option("truncate", false)
  .trigger(Trigger.Once) // <-- Gives MicroBatchExecution
  .queryName("rate2console")
  .start

// The following gives access to the internals
// And to MicroBatchExecution
import org.apache.spark.sql.execution.streaming.StreamingQueryWrapper
val engine = query.asInstanceOf[StreamingQueryWrapper].streamingQuery
import org.apache.spark.sql.execution.streaming.StreamExecution
assert(engine.isInstanceOf[StreamExecution])

import org.apache.spark.sql.execution.streaming.MicroBatchExecution
val microBatchEngine = engine.asInstanceOf[MicroBatchExecution]
assert(microBatchEngine.trigger == Trigger.Once)
```

`MicroBatchExecution` defines [streaming.sql.batchId](#) local property for the current **batch** or **epoch IDs** that is used for the following:

- `DataWritingSparkTask` is requested to run
- `MicroBatchExecution` is requested to [run a single streaming micro-batch](#)

Creating MicroBatchExecution Instance

`MicroBatchExecution` takes the following to be created:

- `SparkSession`
- Name of the streaming query
- Path of the checkpoint directory
- Analyzed logical query plan of the streaming query (`LogicalPlan`)
- [Streaming sink](#)
- [Trigger](#)
- Trigger clock (`clock`)
- [Output mode](#)
- Extra options (`Map[String, String]`)
- `deleteCheckpointOnStop` flag to control whether to delete the checkpoint directory on stop

`MicroBatchExecution` initializes the [internal properties](#).

MicroBatchExecution and TriggerExecutor — triggerExecutor Property

```
triggerExecutor: TriggerExecutor
```

`triggerExecutor` is the [TriggerExecutor](#) of the streaming query that is how micro-batches are executed at regular intervals.

`triggerExecutor` is initialized based on the given [Trigger](#) (that was used to create the `MicroBatchExecution`):

- [ProcessingTimeExecutor](#) for `Trigger.ProcessingTime`
- [OneTimeExecutor](#) for [OneTimeTrigger](#) (aka `Trigger.Once` trigger)

`triggerExecutor` throws an `IllegalStateException` when the [Trigger](#) is not one of the built-in implementations.

```
Unknown type of trigger: [trigger]
```

Note

`triggerExecutor` is used exclusively when `StreamExecution` is requested to run an activated streaming query (at regular intervals).

Running Activated Streaming Query

— `runActivatedStream` Method

```
runActivatedStream(  
    sparkSessionForStream: SparkSession): Unit
```

Note `runActivatedStream` is part of [StreamExecution Contract](#) to run the activated streaming query.

`runActivatedStream` simply requests the [TriggerExecutor](#) to execute micro-batches using the [batch runner](#) (until `MicroBatchExecution` is no longer [active](#)).

TriggerExecutor's Batch Runner

The batch runner (of the [TriggerExecutor](#)) is executed as long as the `MicroBatchExecution` is [active](#).

Note *trigger* and *batch* are considered equivalent and used interchangeably.

The batch runner [initializes query progress for the new trigger](#).

In [triggerExecution time-tracking section](#), the batch runner [populates start offsets \(from the offsets and commits metadata logs\)](#) (when `currentBatchId` is the default `-1` that is right after `MicroBatchExecution` was [created](#)).

The batch runner prints out the following INFO message to the logs:

```
Stream started from [committedOffsets]
```

The batch runner sets the human-readable description of any Spark job submitted (that streaming sources may submit to get new data) to the [batch description](#).

The batch runner [constructs the next streaming micro-batch](#) (when the `isCurrentBatchConstructed` internal flag is off).

The batch runner [records trigger offsets](#) (with the `committed` and `available` offsets).

The batch runner updates the [current StreamingQueryStatus](#) with the `isNewDataAvailable` for `isDataAvailable` property.

With the `isCurrentBatchConstructed` flag enabled (`true`), the batch runner [updates the status message](#) to one of the following (per `isNewDataAvailable`) and [runs the streaming micro-batch](#).

```
Processing new data
```

```
No new data but cleaning up state
```

With the `isCurrentBatchConstructed` flag disabled (`false`), the batch runner simply [updates the status message](#) to the following:

```
Waiting for data to arrive
```

The batch runner [finalizes query progress for the trigger](#) (with a flag that indicates whether the current batch had new data).

With the `isCurrentBatchConstructed` flag enabled (`true`), the batch runner increments the `currentBatchId` and turns the `isCurrentBatchConstructed` flag off (`false`).

With the `isCurrentBatchConstructed` flag disabled (`false`), the batch runner simply sleeps (as long as configured using the `spark.sql.streaming.pollingDelay` configuration property).

In the end, the batch runner [updates the status message](#) to the following status and returns whether the `MicroBatchExecution` is [active](#) or not.

```
Waiting for next trigger
```

Resolving Analyzed Logical Plan of Streaming Query — `logicalPlan` Property

```
logicalPlan: LogicalPlan
```

Note

`logicalPlan` is part of [StreamExecution Contract](#) to resolve the analyzed logical plan of a streaming query.

`logicalPlan` resolves (*replaces*) [StreamingRelation](#), [StreamingRelationV2](#) logical operators to [StreamingExecutionRelation](#) logical operators.

Note

`logicalPlan` is a Scala lazy value and so the resolution happens only once at the first access and is cached for later use afterwards.

Internally, `logicalPlan` ...FIXME

Populating Start Offsets From Checkpoint (Resuming from Checkpoint) — `populateStartOffsets` Internal Method

```
populateStartOffsets(  
    sparkSessionToRunBatches: SparkSession): Unit
```

`populateStartOffsets` requests the [Offset Write-Ahead Log](#) for the [latest committed batch id with metadata](#) (i.e. [OffsetSeq](#)).

Note	The batch id could not be available in the write-ahead log when a streaming query started with a new log or no batch was not persisted (added) to the log before.
------	---

`populateStartOffsets` branches off based on whether the latest committed batch was [available or not](#).

Note	<code>populateStartOffsets</code> is used exclusively when MicroBatchExecution is requested to run an activated streaming query (before the first micro-batch) .
------	--

Latest Committed Batch Available

When the latest committed batch id with the metadata was available in the [Offset Write-Ahead Log](#), `populateStartOffsets` (re)initializes the internal state as follows:

- Sets the [current batch ID](#) to the latest committed batch ID found
- Turns the [isCurrentBatchConstructed](#) flag on (`true`)
- Sets the [available offsets](#) to the offsets found in the metadata

When the latest batch ID found is greater than `0`, `populateStartOffsets` requests the [Offset Write-Ahead Log](#) for the [second latest batch ID with metadata](#) or throws an [IllegalStateException](#) if not found.

```
batch [latestBatchId - 1] doesn't exist
```

`populateStartOffsets` sets the [committed offsets](#) to the second latest committed offsets.

`populateStartOffsets` updates the offset metadata.

Caution	FIXME Describe me
---------	-------------------

`populateStartOffsets` requests the [Offset Commit Log](#) for the [latest committed batch id with metadata](#) (i.e. [CommitMetadata](#)).

Caution	FIXME Describe me
---------	-------------------

When the latest committed batch id with metadata was found which is exactly the latest batch ID (found in the [Offset Commit Log](#)), `populateStartOffsets` ...FIXME

When the latest committed batch id with metadata was found, but it is not exactly the second latest batch ID (found in the [Offset Commit Log](#)), `populateStartOffsets` prints out the following WARN message to the logs:

```
Batch completion log latest batch id is [latestCommittedBatchId], which is not trailing batchid [latestBatchId] by one
```

When no commit log present in the [Offset Commit Log](#), `populateStartOffsets` prints out the following INFO message to the logs:

```
no commit log present
```

In the end, `populateStartOffsets` prints out the following DEBUG message to the logs:

```
Resuming at batch [currentBatchId] with committed offsets [committedOffsets] and available offsets [availableOffsets]
```

No Latest Committed Batch

When the latest committed batch id with the metadata could not be found in the [Offset Write-Ahead Log](#), it is assumed that the streaming query is started for the very first time (or the [checkpoint location](#) changed).

`populateStartOffsets` prints out the following INFO message to the logs:

```
Starting new streaming query.
```

`populateStartOffsets` sets the [current batch ID](#) to `0` and creates a new [WatermarkTracker](#).

Constructing (Or Skipping) Next Streaming Micro-Batch — `constructNextBatch` Internal Method

```
constructNextBatch(  
    noDataBatchesEnabled: Boolean): Boolean
```

Note

`constructNextBatch` will only be executed when the `isCurrentBatchConstructed` internal flag is enabled (`true`).

`constructNextBatch` performs the following steps:

1. Requesting the latest offsets from every streaming source (of the streaming query)
2. Updating `availableOffsets` `StreamProgress` with the latest available offsets
3. Updating batch metadata with the current event-time watermark and batch timestamp
4. Checking whether to construct (or skip) the next micro-batch

In the end, `constructNextBatch` returns whether the next streaming micro-batch was constructed or skipped.

Note

`constructNextBatch` is used exclusively when `MicroBatchExecution` is requested to run the activated streaming query.

Requesting Latest Offsets from Streaming Sources

`constructNextBatch` firstly requests the latest available offsets from every `streaming source`.

Note

`constructNextBatch` checks out the latest offset in every streaming data source sequentially, i.e. one data source at a time.

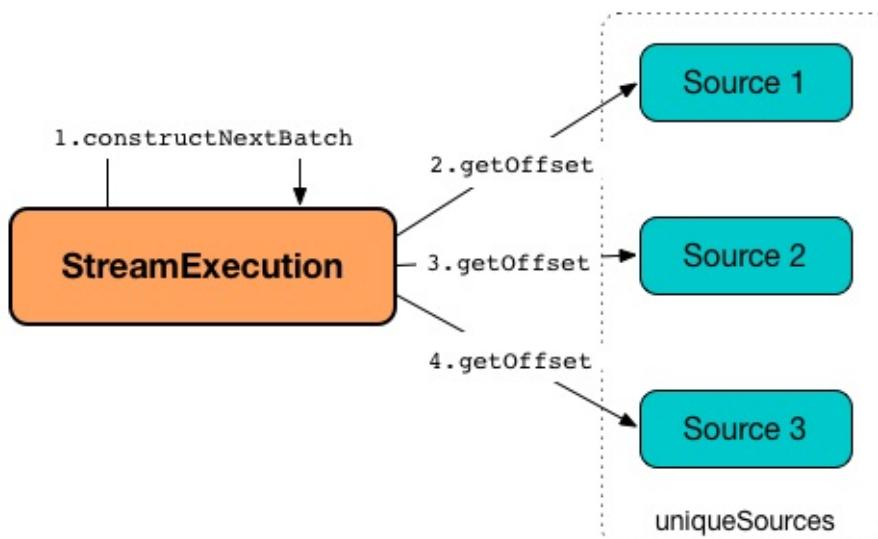


Figure 1. MicroBatchExecution's Getting Offsets From Streaming Sources

For every `streaming source` (Data Source API V1), `constructNextBatch` updates the status message to the following:

Getting offsets from [source]

In [getOffset time-tracking section](#), `constructNextBatch` requests the `source` for the [latest offset](#).

For every [MicroBatchReader](#) (Data Source API V2), `constructNextBatch` [updates the status message](#) to the following:

```
Getting offsets from [source]
```

In [setOffsetRange time-tracking section](#), `constructNextBatch` finds the available offsets of the source (in the [available offset](#) internal registry) and, if found, requests the [MicroBatchReader](#) to [deserialize the offset](#) (from [JSON format](#)). `constructNextBatch` requests the [MicroBatchReader](#) to [set the desired offset range](#).

In [getEndOffset time-tracking section](#), `constructNextBatch` requests the [MicroBatchReader](#) for the [end offset](#).

Updating availableOffsets StreamProgress with Latest Available Offsets

`constructNextBatch` updates the [availableOffsets StreamProgress](#) with the latest reported offsets.

Updating Batch Metadata with Current Event-Time Watermark and Batch Timestamp

`constructNextBatch` updates the [batch metadata](#) with the current [event-time watermark](#) (from the [WatermarkTracker](#)) and the batch timestamp.

Checking Whether to Construct or Skip Next Micro-Batch

`constructNextBatch` checks whether or not the next streaming micro-batch should be constructed (`lastExecutionRequiresAnotherBatch`).

`constructNextBatch` uses the [last IncrementalExecution](#) if the [last execution requires another micro-batch](#) (using the [batch metadata](#)) and the given `noDataBatchesEnabled` flag is enabled (`true`).

`constructNextBatch` also [checks out whether new data is available](#) (based on [available](#) and [committed offsets](#)).

Note	<code>shouldConstructNextBatch</code> local flag is enabled (<code>true</code>) when there is new data available (based on offsets) or the last execution requires another micro-batch (and the given <code>noDataBatchesEnabled</code> flag is enabled).
------	---

`constructNextBatch` prints out the following TRACE message to the logs:

```
noDataBatchesEnabled = [noDataBatchesEnabled],  
lastExecutionRequiresAnotherBatch =  
[lastExecutionRequiresAnotherBatch], isNewDataAvailable =  
[isNewDataAvailable], shouldConstructNextBatch =  
[shouldConstructNextBatch]
```

`constructNextBatch` branches off per whether to `constructs` or `skip` the next batch (per `shouldConstructNextBatch` flag in the above TRACE message).

Constructing Next Micro-Batch — `shouldConstructNextBatch` Flag Enabled

Note	<code>FIXME (if (shouldConstructNextBatch) ...)</code>
------	--

With the `shouldConstructNextBatch` flag enabled (`true`), `constructNextBatch` ...`FIXME`

Skipping Next Micro-Batch — `shouldConstructNextBatch` Flag Disabled

With the `shouldConstructNextBatch` flag disabled (`false`), `constructNextBatch` turns the `noNewData` flag on (`true`) and wakes up (`notifies`) all threads waiting for the `awaitProgressLockCondition` lock.

OLD / Review

New Data Available

If `lastExecution` is available (which may not when `constructNextBatch` is executed the very first time), `constructNextBatch` takes the executed physical plan (i.e. `SparkPlan`) and collects all `EventTimewindowExec` physical operators with the count of `eventTimeStats` greater than `0`.

Note	The executed physical plan is available as <code>executedPlan</code> property of <code>IncrementalExecution</code> (which is a custom <code>QueryExecution</code>).
------	--

`constructNextBatch` prints out the following DEBUG message to the logs:

```
Observed event time stats: [eventTimeStats]
```

`constructNextBatch` calculates the difference between the maximum value of `eventTimeStats` and `delayMs` for every `EventTimeWatermarkExec` physical operator.

Note

The maximum value of `eventTimeStats` is the youngest time, i.e. the time the closest to the current time.

`constructNextBatch` then takes the first difference (if available at all) and uses it as a possible new event time watermark.

If the event time watermark candidate is greater than the current watermark (i.e. later time-wise), `constructNextBatch` prints out the following INFO message to the logs:

```
Updating eventTime watermark to: [newWatermarkMs] ms
```

`constructNextBatch` creates a new [OffsetSeqMetadata](#) with the new event-time watermark and the current time.

Otherwise, if the `eventTime` watermark candidate is not greater than the current watermark, `constructNextBatch` simply prints out the following DEBUG message to the logs:

```
Event time didn't move: [newWatermarkMs] <= [batchWatermarkMs]
```

`constructNextBatch` creates a new [OffsetSeqMetadata](#) with just the current time.

Note

Although `constructNextBatch` collects all the `EventTimeWatermarkExec` physical operators in the executed physical plan of `lastExecution`, only the first matters if available.

Note

A physical plan can have as many `EventTimeWatermarkExec` physical operators as `withWatermark` operators used in a streaming query.

Note

[Streaming watermark](#) can be changed between a streaming query's restarts (and be different between what is checkpointed and the current version of the query).

FIXME True? Example?

`constructNextBatch` then adds the offsets to metadata log.

`constructNextBatch` updates the status message to [Writing offsets to log](#).

In [walCommit](#) time-tracking section, `constructNextBatch` adds the offsets in the batch to [OffsetSeqLog](#).

	<p>While writing the offsets to the metadata log, <code>constructNextBatch</code> uses the following internal registries:</p> <ul style="list-style-type: none"> • <code>currentBatchId</code> for the current batch id • <code>StreamProgress</code> for the available offsets • <code>sources</code> for the streaming sources • <code>OffsetSeqMetadata</code>
--	---

`constructNextBatch` reports a `AssertionError` when writing to the metadata log has failed.

```
Concurrent update to the log. Multiple streaming jobs detected for [currentBatchId]
```

	<p>Use <code>StreamingQuery.lastProgress</code> to access <code>walCommit</code> duration.</p>
Tip	<pre>scala> :type sq org.apache.spark.sql.streaming.StreamingQuery sq.lastProgress.durationMs.get("walCommit")</pre>

	<p>Enable INFO logging level for <code>org.apache.spark.sql.execution.streaming.StreamExecution</code> logger to be notified about <code>walCommit</code> duration.</p>
Tip	<pre>17/08/11 09:04:17 INFO StreamExecution: Streaming query made progress: { "id" : "ec8f8228-90f6-4e1f-8ad2-80222affed63", "runId" : "f605c134-cfb0-4378-88c1-159d8a7c232e", "name" : "rates-to-console", "timestamp" : "2017-08-11T07:04:17.373Z", "batchId" : 0, "numInputRows" : 0, "processedRowsPerSecond" : 0.0, "durationMs" : { "addBatch" : 38, "getBatch" : 1, "getOffset" : 0, "queryPlanning" : 1, "triggerExecution" : 62, "walCommit" : 19 // <-- walCommit }, }</pre>

`constructNextBatch` commits the offsets for the batch (only when `current batch id` is not `0`, i.e. when the `query` has just been started and `constructNextBatch` is called the first time).

`constructNextBatch` takes the previously-committed batch (from `OffsetSeqLog`), extracts the stored offsets per source.

	<p><code>constructNextBatch</code> uses <code>OffsetSeq.toStreamProgress</code> and <code>sources</code> registry to extract the offsets per source.</p>
--	--

`constructNextBatch` requests every streaming source to [commit the offsets](#)

Note	<code>constructNextBatch</code> uses the <code>Source</code> contract to commit the offsets (using <code>Source.commit</code> method).
------	--

`constructNextBatch` reports a `IllegalStateException` when [current batch id](#) is `0`.

`batch [currentBatchId]` doesn't exist

Running Single Streaming Micro-Batch — `runBatch` Internal Method

```
runBatch(sparkSessionToRunBatch: SparkSession): Unit
```

`runBatch` performs the following steps (aka *phases*):

1. `getBatch` Phase — Requesting New (and Hence Unprocessed) Data From Streaming Sources
2. `withNewSources` Phase — Replacing `StreamingExecutionRelations` (in Logical Plan) With Relations With New Data or Empty `LocalRelation`
3. `triggerLogicalPlan` Phase — Transforming Catalyst Expressions
4. `queryPlanning` Phase — Creating `IncrementalExecution` for Current Streaming Batch
5. `nextBatch` Phase — Creating Dataset (with `IncrementalExecution` for New Data)
6. `addBatch` Phase — Adding Current Streaming Batch to Sink
7. `awaitBatchLock` Phase — Waking Up Threads Waiting For Stream to Progress

Note	<code>runBatch</code> is used when...FIXME
------	--

getBatch Phase — Requesting New (and Hence Unprocessed) Data From Streaming Sources

Internally, `runBatch` first requests the [streaming sources](#) for unprocessed data (and stores them as `DataFrames` in `newData` internal registry).

In [getBatch time-tracking section](#), `runBatch` goes over the [available offsets per source](#) and processes the offsets that [have not been committed yet](#).

`runBatch` then requests [every source for the data](#) (as `DataFrame` with the new records).

Note

`runBatch` requests the streaming sources for new DataFrames sequentially, source by source.

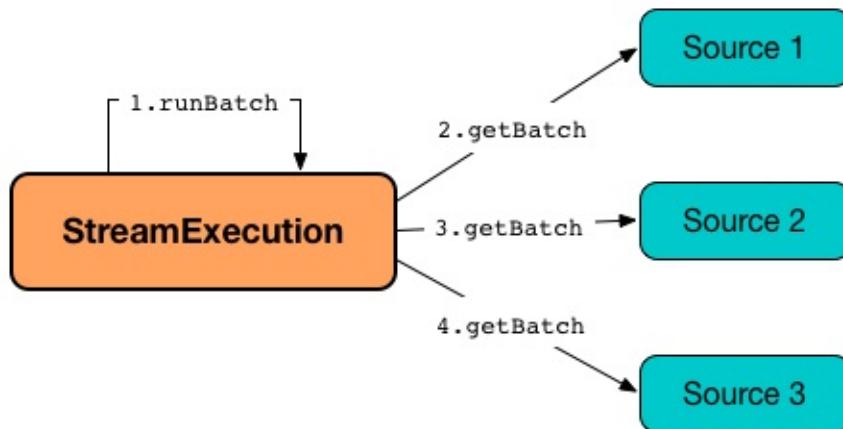


Figure 2. StreamExecution's Running Single Streaming Batch (getBatch Phase)

`runBatch` prints out the following DEBUG message to the logs:

```
Retrieving data from [source]: [current] -> [available]
```

`runBatch` prints out the following DEBUG message to the logs:

```
getBatch took [timeTaken] ms
```

withNewSources Phase — Replacing StreamingExecutionRelations (in Logical Plan) With Relations With New Data or Empty LocalRelation

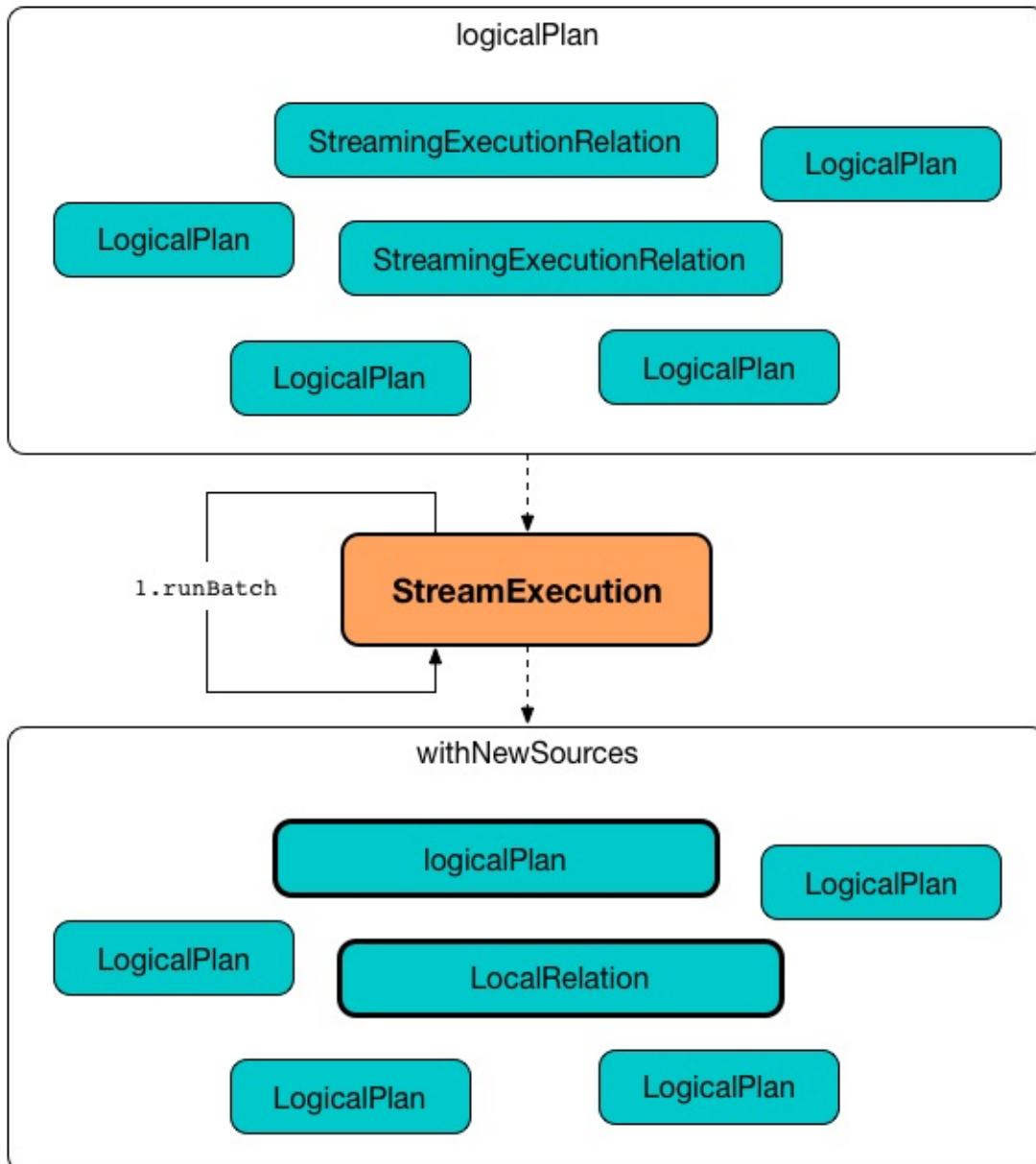


Figure 3. StreamExecution's Running Single Streaming Batch (withNewSources Phase)

In **withNewSources** phase, `runBatch` transforms **logical query plan** and replaces every **StreamingExecutionRelation** logical operator with the logical plan of the `DataFrame` with the input data in a batch for the corresponding streaming source.

Note

StreamingExecutionRelation logical operator is used to represent a streaming source in the **logical query plan** of a streaming `Dataset`.

`runBatch` finds the corresponding `DataFrame` (with the input data) per streaming source in `newData` internal registry. If found, `runBatch` takes the logical plan of the `DataFrame`. If not, `runBatch` creates a `LocalRelation` logical relation (for the output schema).

Note

`newData` internal registry contains entries for streaming sources that have new data available in the current batch.

While replacing `StreamingExecutionRelation` operators, `runBatch` records the output schema of the streaming source (from `StreamingExecutionRelation`) and the `DataFrame` with the new data (in `replacements` temporary internal buffer).

`runBatch` makes sure that the output schema of the streaming source with a new data in the batch has not changed. If the output schema has changed, `runBatch` reports...FIXME

triggerLogicalPlan Phase — Transforming Catalyst Expressions

`runBatch` transforms Catalyst expressions in `withNewSources` new logical plan (using `replacements` temporary internal buffer).

- Catalyst `Attribute` is replaced with one if recorded in `replacements` internal buffer (that corresponds to the attribute in the `DataFrame` with the new input data in the batch)
- `CurrentTimestamp` and `CurrentDate` Catalyst expressions are replaced with `CurrentBatchTimestamp` expression (with `batchTimestampMs` from `OffsetSeqMetadata`).

<p>Note</p> <p><code>CurrentTimestamp</code> Catalyst expression corresponds to <code>current_timestamp</code> function.</p> <p>Find more about <code>current_timestamp</code> function in Mastering Apache Spark 2 gitbook.</p>
<p>Note</p> <p><code>CurrentDate</code> Catalyst expression corresponds to <code>current_date</code> function.</p> <p>Find more about <code>current_date</code> function in Mastering Apache Spark 2 gitbook.</p>

queryPlanning Phase — Creating IncrementalExecution for Current Streaming Batch

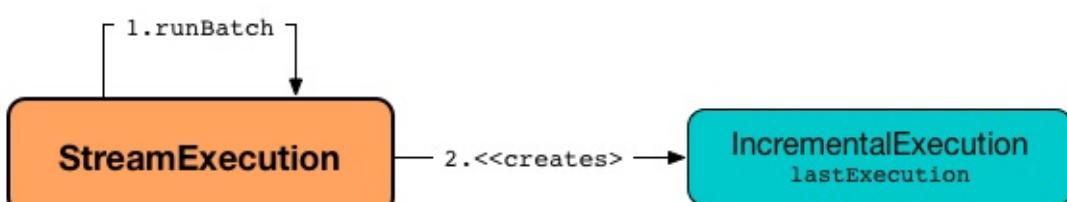


Figure 4. StreamExecution's Query Planning (queryPlanning Phase)

In [queryPlanning time-tracking section](#), `runBatch` creates a new `IncrementalExecution` with the following:

- Transformed [logical query plan](#) with [logical relations](#) for every streaming source and corresponding attributes

- the streaming query's `output mode`
- `state checkpoint directory` for managing state
- `current run id`
- `current batch id`
- `OffsetSeqMetadata`

The new `IncrementalExecution` is recorded in `lastExecution` property.

Before leaving `queryPlanning` section, `runBatch` forces preparation of the physical plan for execution (i.e. requesting `IncrementalExecution` for `executedPlan`).

Note	<code>executedPlan</code> is a physical plan (i.e. <code>sparkPlan</code>) ready for execution with <code>state optimization rules</code> applied.
------	---

nextBatch Phase — Creating Dataset (with IncrementalExecution for New Data)

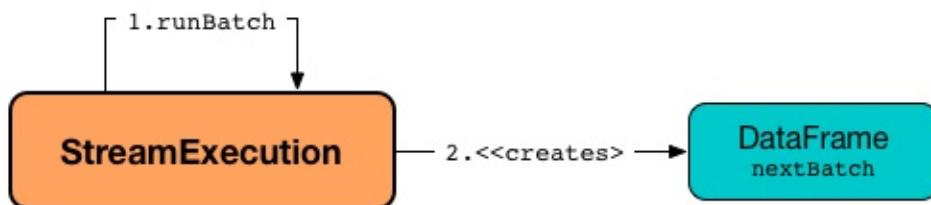


Figure 5. StreamExecution Creates DataFrame with New Data

`runBatch` creates a `DataFrame` with the new `IncrementalExecution` (as `QueryExecution`) and its analyzed output schema.

Note	The new <code>DataFrame</code> represents the result of a streaming query.
------	--

addBatch Phase — Adding Current Streaming Batch to Sink

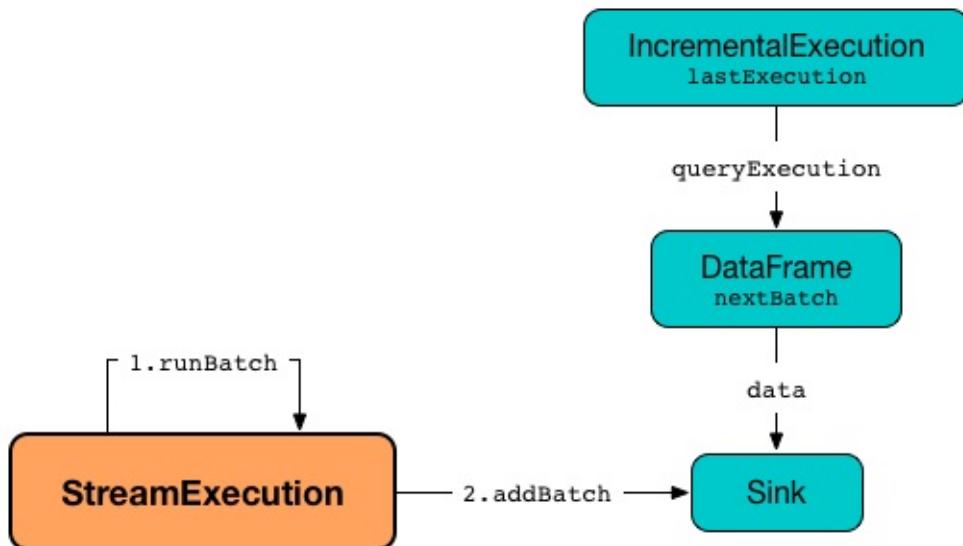


Figure 6. StreamExecution Creates DataFrame with New Data

In [addBatch time-tracking section](#), `runBatch` requests the one and only streaming [Sink](#) to [add the results of a streaming query](#) (as the `DataFrame` created in [nextBatch Phase](#)).

Note	<code>runBatch</code> uses Sink.addBatch method to request the <code>Sink</code> to add the results.
Note	<code>runBatch</code> uses <code>SQLExecution.withNewExecutionId</code> to execute and track all the Spark actions (under one execution id) that <code>Sink</code> can use when requested to add the results.
Note	The new <code>DataFrame</code> will only be executed in <code>Sink.addBatch</code> .
Note	<code>SQLExecution.withNewExecutionId</code> posts a <code>SparkListenerSQLExecutionStart</code> event before executing <code>Sink.addBatch</code> and a <code>SparkListenerSQLExecutionEnd</code> event right afterwards.
Tip	Register <code>SparkListener</code> to get notified about the SQL execution events. You can find more information on <code>SparkListener</code> in Mastering Apache Spark 2 gitbook.

awaitBatchLock Phase — Waking Up Threads Waiting For Stream to Progress

In [awaitBatchLock](#) code block (it is not a time-tracking section), `runBatch` acquires a lock on [awaitProgressLock](#), wakes up all waiting threads on [awaitProgressLockCondition](#) and immediately releases [awaitProgressLock](#) lock.

Note	<code>awaitProgressLockCondition</code> is used mainly when <code>StreamExecution</code> <code>processAllAvailable</code> (and also when <code>awaitOffset</code> , but that seems mainly for testing).
------	---

Stopping Stream Processing (Execution of Streaming Query) — `stop` Method

```
stop(): Unit
```

Note `stop` is part of the [StreamingQuery Contract](#) to stop a streaming query.

`stop` sets the [state](#) to [TERMINATED](#).

When the [stream execution thread](#) is alive, `stop` requests the current [SparkContext](#) to [cancelJobGroup](#) identified by the [runId](#) and waits for this thread to die. Just to make sure that there are no more streaming jobs, `stop` requests the current [SparkContext](#) to [cancelJobGroup](#) identified by the [runId](#) again.

In the end, `stop` prints out the following INFO message to the logs:

```
Query [prettyIdString] was stopped
```

Checking Whether New Data Is Available (Based on Available and Committed Offsets) — `isNewDataAvailable` Internal Method

```
isNewDataAvailable: Boolean
```

`isNewDataAvailable` checks whether there is a streaming source (in the [available offsets](#)) for which [committed offsets](#) are different from the available offsets or not available (committed) at all.

`isNewDataAvailable` is positive (`true`) when there is at least one such streaming source.

Note `isNewDataAvailable` is used when [MicroBatchExecution](#) is requested to [run an activated streaming query](#) and [construct the next streaming micro-batch](#).

Internal Properties

Name	Description
	<p>Flag to control whether to run a streaming micro-batch (<code>true</code>) or not (<code>false</code>)</p> <p>Default: <code>false</code></p>

	<ul style="list-style-type: none"> When disabled (<code>false</code>), changed to whatever constructing the next streaming micro-batch gives back when running activated streaming query Disabled (<code>false</code>) after running a streaming micro-batch (when enabled after constructing the next streaming micro-batch) Enabled (<code>true</code>) when populating start offsets (when running an activated streaming query) and re-starting a streaming query from a checkpoint (using the Offset Write-Ahead Log) Disabled (<code>false</code>) when populating start offsets (when running an activated streaming query) and re-starting a streaming query from a checkpoint when the latest offset checkpointed (written) to the offset write-ahead log has been successfully processed and committed to the Offset Commit Log 		
<code>readerToDataSourceMap</code>	<code>(Map[MicroBatchReader, (DataSourceV2, Map[String, String])])</code>		
<code>sources</code>	<p>Streaming sources (of the StreamingExecutionRelations of the analyzed logical query plan of the streaming query)</p> <p>Default: (empty)</p> <table border="1" data-bbox="611 1179 1389 1313"> <tr> <td data-bbox="611 1179 738 1313">Note</td> <td data-bbox="738 1179 1389 1313"> <code>sources</code> is part of the ProgressReporter Contract for the streaming sources of the streaming query. </td> </tr> </table> <ul style="list-style-type: none"> Initialized when <code>MicroBatchExecution</code> is requested for the transformed logical query plan <p>Used when:</p> <ul style="list-style-type: none"> Populating start offsets (for the available and committed offsets) Constructing or skipping next streaming micro-batch (and persisting offsets to write-ahead log) 	Note	<code>sources</code> is part of the ProgressReporter Contract for the streaming sources of the streaming query.
Note	<code>sources</code> is part of the ProgressReporter Contract for the streaming sources of the streaming query.		
<code>watermarkTracker</code>	WatermarkTracker that is created when <code>MicroBatchExecution</code> is requested to populate start offsets (when requested to run an activated streaming query)		

MicroBatchWriter — DataSourceWriter for StreamWriters (Data Source V2)

`MicroBatchWriter` is a `DataSourceWriter` that uses the given batch ID as the epoch when requested to commit, abort and create a `WriterFactory` for a given `StreamWriter`.

Tip

Read up on [DataSourceWriter](#) in [The Internals of Spark SQL book](#).

`MicroBatchWriter` is created exclusively when `MicroBatchExecution` is requested to run a [streaming batch](#) (with a `StreamWriterSupport` streaming sink).

MicroBatchReadSupport Contract — Data Sources with MicroBatchReaders

`MicroBatchReadSupport` is the abstraction of data sources with a `MicroBatchReader` in Micro-Batch Stream Processing.

`ContinuousReadSupport` defines a single `createMicroBatchReader` method to create a `MicroBatchReader`.

```
MicroBatchReader createMicroBatchReader(
    Optional<StructType> schema,
    String checkpointLocation,
    DataSourceOptions options)
```

`createMicroBatchReader` is used when:

- `MicroBatchExecution` is requested for the analyzed logical plan (and creates a `StreamingExecutionRelation` for a `StreamingRelationV2` with a `MicroBatchReadSupport` data source)
- `DataStreamReader` is requested to create a streaming query for a `MicroBatchReadSupport` data source

Table 1. MicroBatchReadSupports

MicroBatchReadSupport	Description
KafkaSourceProvider	Data source provider for kafka format
RateStreamProvider	
TextSocketSourceProvider	

MicroBatchReader Contract — Data Source Readers in Micro-Batch Stream Processing

`MicroBatchReader` is the [extension](#) of Spark SQL's `DataSourceReader` (and `BaseStreamingSource`) contracts for [data source readers](#) in [Micro-Batch Stream Processing](#).

`MicroBatchReader` is part of the novel Data Source API V2 in Spark SQL.

Tip	Read up on Data Source API V2 in The Internals of Spark SQL book.
-----	---

Table 1. MicroBatchReader Contract

Method	Description
<code>commit</code>	<pre style="background-color: #f0f0f0; padding: 5px;">void commit(Offset end)</pre> <p>Used when...FIXME</p>
<code>deserializeOffset</code>	<pre style="background-color: #f0f0f0; padding: 5px;">Offset deserializeOffset(String json)</pre> <p>Deserializes the <code>offset</code> (from JSON format) Used when...FIXME</p>
<code>getEndOffset</code>	<pre style="background-color: #f0f0f0; padding: 5px;">Offset getEndOffset()</pre> <p>End <code>offset</code> for this reader Used when...FIXME</p>
<code>getStartOffset</code>	<pre style="background-color: #f0f0f0; padding: 5px;">Offset getStartOffset()</pre> <p>Used when...FIXME</p>
<code>setOffsetRange</code>	<pre style="background-color: #f0f0f0; padding: 5px;">void setOffsetRange(Optional<Offset> start, Optional<Offset> end)</pre> <p>Sets the desired offset range for input partitions created from this reader (for data scan) Used when...FIXME</p>

Table 2. MicroBatchReaders

MicroBatchReader	Description
KafkaMicroBatchReader	
MemoryStream	
RateStreamMicroBatchReader	
TextSocketMicroBatchReader	

WatermarkTracker

`WatermarkTracker` tracks the event-time watermark of a streaming query (across `EventTimeWatermarkExec` operators in a physical query plan) based on a given `MultipleWatermarkPolicy`.

`WatermarkTracker` is used exclusively in `MicroBatchExecution`.

`WatermarkTracker` is created (using the factory method) when `MicroBatchExecution` is requested to populate start offsets (when requested to run an activated streaming query).

`WatermarkTracker` takes a single `MultipleWatermarkPolicy` to be created.

`MultipleWatermarkPolicy` can be one of the following:

- `MaxWatermark` (alias: `min`)
- `MinWatermark` (alias: `max`)

Enable ALL logging level for `org.apache.spark.sql.execution.streaming.WatermarkTracker` to see what happens inside.

Tip

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.execution.streaming.WatermarkTracker=ALL
```

Refer to [Logging](#).

Creating WatermarkTracker — apply Factory Method

```
apply(conf: RuntimeConfig): WatermarkTracker
```

`apply` uses the `spark.sql.streaming.multipleWatermarkPolicy` configuration property for the global watermark policy (default: `min`) and creates a `WatermarkTracker`.

Note

`apply` is used exclusively when `MicroBatchExecution` is requested to populate start offsets (when requested to run an activated streaming query).

setWatermark Method

```
setWatermark(newWatermarkMs: Long): Unit
```

`setWatermark` simply updates the [global event-time watermark](#) to the given `newWatermarkMs`.

Note

`setWatermark` is used exclusively when `MicroBatchExecution` is requested to [populate start offsets](#) (when requested to [run an activated streaming query](#)).

Updating Event-Time Watermark — `updateWatermark` Method

`updateWatermark(executedPlan: SparkPlan): Unit`

`updateWatermark` requests the given physical operator (`SparkPlan`) to collect all [EventTimeWatermarkExec](#) unary physical operators.

`updateWatermark` simply exits when no `EventTimeWatermarkExec` was found.

`updateWatermark` ...FIXME

Note

`updateWatermark` is used exclusively when `MicroBatchExecution` is requested to [run a single streaming batch](#) (when requested to [run an activated streaming query](#)).

Internal Properties

Name	Description
<code>globalWatermarkMs</code>	<p>Current global event-time watermark per MultipleWatermarkPolicy (across all EventTimeWatermarkExec operators in a physical query plan)</p> <p>Default: <code>0</code></p> <p>Used when...FIXME</p>
<code>operatorToWatermarkMap</code>	<p>Event-time watermark per EventTimeWatermarkExec physical operator (<code>mutable.HashMap[Int, Long]</code>)</p> <p>Used when...FIXME</p>

Offsets and Metadata Checkpointing

Every time a streaming query is started, [stream execution engines](#) use checkpoint location to resume stream processing and get so-called **start offsets** (to start query processing from).

`StreamExecution` resumes (populates the start offsets) from the latest checkpointed offsets from the [offset log](#) that may have already been processed (and committed to the [batch commit log](#) so they are used as the current [committed offsets](#)).

- Hadoop DFS-based metadata storage of [OffsetSeqs](#)
- [OffsetSeq](#) and [StreamProgress](#)
- [StreamProgress](#) and [StreamExecutions](#) ([committed](#) and [available offsets](#))

Micro-Batch Stream Processing

In [Micro-Batch Stream Processing](#), the [available offsets](#) registry is populated with the [latest offsets](#) from the [Write-Ahead Log \(WAL\)](#) when `MicroBatchExecution` stream processing engine is requested to [populate start offsets from checkpoint](#).

The [available offsets](#) are then [added](#) to the [committed offsets](#) when the latest batch ID available (as described above) is exactly the [latest batch ID](#) committed to [Offset Commit Log](#) when `MicroBatchExecution` stream processing engine is requested to [populate start offsets from checkpoint](#).

When a streaming query is resumed (restarted) from a checkpoint, `MicroBatchExecution` prints out the following INFO message:

```
Resuming at batch [currentBatchId] with committed offsets
[committedOffsets] and available offsets [availableOffsets]
```

Every time `MicroBatchExecution` is requested to [check whether a new data is available](#) (in any of the streaming sources)...FIXME

When `MicroBatchExecution` is requested to [construct the next streaming micro-batch](#) (when `MicroBatchExecution` requested to [run the activated streaming query](#)), every [streaming source](#) is requested for the latest offset available that are [added](#) to the [availableOffsets](#) registry. [Streaming sources](#) report some offsets or none at all (if this source has never received any data). Streaming sources with no data are excluded (*filtered out*).

`MicroBatchExecution` prints out the following TRACE message to the logs:

```
noDataBatchesEnabled = [noDataBatchesEnabled],  
lastExecutionRequiresAnotherBatch =  
[lastExecutionRequiresAnotherBatch], isNewDataAvailable =  
[isNewDataAvailable], shouldConstructNextBatch =  
[shouldConstructNextBatch]
```

With `shouldConstructNextBatch` internal flag enabled, `MicroBatchExecution` commits (adds) the available offsets for the batch to the [Write-Ahead Log \(WAL\)](#) and prints out the following INFO message to the logs:

```
Committed offsets for batch [currentBatchId]. Metadata  
[offsetSeqMetadata]
```

When [running a single streaming micro-batch](#), `MicroBatchExecution` requests every [Source](#) and [MicroBatchReader](#) (in the [availableOffsets](#) registry) for unprocessed data (that has not been [committed](#) yet and so considered unprocessed).

In the end (of [running a single streaming micro-batch](#)), `MicroBatchExecution` commits (adds) the available offsets (to the [committedOffsets](#) registry) so they are considered processed already.

`MicroBatchExecution` prints out the following DEBUG message to the logs:

```
Completed batch [currentBatchId]
```

Limitations (Assumptions)

It is assumed that the order of [streaming sources](#) in a [streaming query](#) matches the order of the [offsets](#) of [OffsetSeq](#) (in [offsetLog](#)) and [availableOffsets](#).

In other words, a streaming query can be modified and then restarted from a checkpoint (to maintain stream processing state) only when the number of streaming sources and their order are preserved across restarts.

MetadataLog Contract — Metadata Storage

`MetadataLog` is the abstraction of metadata storage that can persist, retrieve, and remove metadata (of type `T`).

Table 1. MetadataLog Contract

Method	Description
<code>add</code>	<pre>add(batchId: Long, metadata: T): Boolean</pre> <p>Persists (<i>adds</i>) metadata of a streaming batch</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>KafkaMicroBatchReader</code> is requested to <code>getOrCreateInitialPartitionOffsets</code> • <code>KafkaSource</code> is requested for the <code>initialPartitionOffsets</code> • <code>CompactibleFileStreamLog</code> is requested for the <code>store metadata of a streaming batch</code> and to <code>compact</code> • <code>FileStreamSource</code> is requested to <code>fetchMaxOffset</code> • <code>FileStreamSourceLog</code> is requested to <code>store (add) metadata of a streaming batch</code> • <code>ManifestFileCommitProtocol</code> is requested to <code>commitJob</code> • <code>MicroBatchExecution</code> stream execution engine is requested to <code>construct a next streaming micro-batch</code> and <code>run a single streaming micro-batch</code> • <code>ContinuousExecution</code> stream execution engine is requested to <code>addOffset</code> and <code>commit an epoch</code> • <code>RateStreamMicroBatchReader</code> is created (<code>creationTimeMs</code>)
<code>get</code>	<pre>get(batchId: Long): Option[T] get(startId: Option[Long], endId: Option[Long]): Array[(Long, T)]</pre> <p>Retrieves (<i>gets</i>) metadata of one or more batches</p> <p>Used when...FIXME</p>

getLatest	<code>getLatest(): Option[(Long, T)]</code> Retrieves the latest-committed metadata (if available) Used when...FIXME
purge	<code>purge(thresholdBatchId: Long): Unit</code> Used when...FIXME

Note	<p>HDFSMetadataLog is the only direct implementation of the MetadataLog Contract in Spark Structured Streaming.</p>
------	---

HDFSMetadataLog — Hadoop DFS-based Metadata Storage

`HDFSMetadataLog` is a concrete [metadata storage](#) (of type `T`) that uses Hadoop DFS for fault-tolerance and reliability.

`HDFSMetadataLog` uses the given [path](#) as the **metadata directory** with metadata logs. The path is immediately converted to a Hadoop [Path](#) for file management.

`HDFSMetadataLog` uses [Json4s](#) with the [Jackson](#) binding for metadata [serialization](#) and [deserialization](#) (to and from JSON format).

`HDFSMetadataLog` is further customized by the [extensions](#).

Table 1. HDFSMetadataLogs (Direct Extensions Only)

HDFSMetadataLog	Description
<code>Anonymous</code>	<code>HDFSMetadataLog</code> of KafkaSourceOffsets for KafkaSource
<code>Anonymous</code>	<code>HDFSMetadataLog</code> of LongOffsets for RateStreamMicroBatchReader
<code>CommitLog</code>	Offset commit log of streaming query execution engines
<code>CompactibleFileStreamLog</code>	Compactible metadata logs (that compact logs at regular interval)
<code>KafkaSourceInitialOffsetWriter</code>	<code>HDFSMetadataLog</code> of KafkaSourceOffsets for KafkaSource
<code>OffsetSeqLog</code>	Write-Ahead Log (WAL) of stream execution engines

Creating HDFSMetadataLog Instance

`HDFSMetadataLog` takes the following to be created:

- `SparkSession`
- Path of the metadata log directory

While being [created](#) `HDFSMetadataLog` creates the [path](#) unless exists already.

Serializing Metadata (Writing Metadata in Serialized Format) — `serialize` Method

```
serialize(
  metadata: T,
  out: OutputStream): Unit
```

`serialize` simply writes the log data (serialized using [Json4s](#) (with Jackson binding) library).

Note

`serialize` is used exclusively when `HDFSMetadataLog` is requested to [write metadata of a streaming batch to a file \(metadata log\)](#) (when [storing metadata of a streaming batch](#)).

Deserializing Metadata (Reading Metadata from Serialized Format) — `deserialize` Method

```
deserialize(in: InputStream): T
```

`deserialize` deserializes a metadata (of type `T`) from a given `InputStream`.

Note

`deserialize` is used exclusively when `HDFSMetadataLog` is requested to [retrieve metadata of a batch](#).

Retrieving Metadata Of Streaming Batch — `get` Method

```
get(batchId: Long): Option[T]
```

Note

`get` is part of the [MetadataLog Contract](#) to get metadata of a batch.

`get` ...FIXME

Retrieving Metadata of Range of Batches — `get` Method

```
get(
  startId: Option[Long],
  endId: Option[Long]): Array[(Long, T)]
```

Note

`get` is part of the [MetadataLog Contract](#) to get metadata of range of batches.

```
get ...FIXME
```

Persisting Metadata of Streaming Batch — `add` Method

```
add(  
    batchId: Long,  
    metadata: T): Boolean
```

Note `add` is part of the [MetadataLog Contract](#) to persist metadata of a streaming batch.

`add` returns `true` when the metadata of the streaming batch was not available and persisted successfully. Otherwise, `add` returns `false`.

Internally, `add` looks up metadata of the given streaming batch (`batchId`) and returns `false` when found.

Otherwise, when not found, `add` creates a metadata log file for the given `batchId` and writes metadata to the file. `add` returns `true` if successful.

Latest Committed Batch Id with Metadata (When Available) — `getLatest` Method

```
getLatest(): Option[(Long, T)]
```

Note `getLatest` is a part of [MetadataLog Contract](#) to retrieve the recently-committed batch id and the corresponding metadata if available in the metadata storage.

`getLatest` requests the internal [FileManager](#) for the files in [metadata directory](#) that match [batch file filter](#).

`getLatest` takes the batch ids (the batch files correspond to) and sorts the ids in reverse order.

`getLatest` gives the first batch id with the metadata which could be found in the metadata storage.

Note It is possible that the batch id could be in the metadata storage, but not available for retrieval.

Removing Expired Metadata (Purging) — `purge` Method

```
purge(thresholdBatchId: Long): Unit
```

Note

`purge` is part of the [MetadataLog Contract](#) to...FIXME.

`purge` ...FIXME

Creating Batch Metadata File — `batchIdToPath` Method

```
batchIdToPath(batchId: Long): Path
```

`batchIdToPath` simply creates a Hadoop [Path](#) for the file called by the specified `batchId` under the [metadata directory](#).

Note

`batchIdToPath` is used when:

- `CompatibleFileStreamLog` is requested to [compact](#) and [allFiles](#)
- `HDFSMetadataLog` is requested to [add](#), [get](#), [purge](#), and [purgeAfter](#)

`isBatchFile` Method

```
isBatchFile(path: Path): Boolean
```

`isBatchFile` ...FIXME

Note

`isBatchFile` is used exclusively when `HDFSMetadataLog` is requested for the [PathFilter of batch files](#).

`pathToBatchId` Method

```
pathToBatchId(path: Path): Long
```

`pathToBatchId` ...FIXME

Note

`pathToBatchId` is used when:

- `CompatibleFileStreamLog` is requested for the [compact interval](#)
- `HDFSMetadataLog` is requested to [isBatchFile](#), [get metadata of a range of batches](#), [getLatest](#), [getOrderedBatchFiles](#), [purge](#), and [purgeAfter](#)

verifyBatchIds Object Method

```
verifyBatchIds(  
    batchIds: Seq[Long],  
    startId: Option[Long],  
    endId: Option[Long]): Unit
```

verifyBatchIds ...FIXME

Note

`verifyBatchIds` is used when:

- `FileStreamSourceLog` is requested to `get`
- `HDFSMetadataLog` is requested to `get`

Retrieving Version (From Text Line) — parseVersion Internal Method

```
parseVersion(  
    text: String,  
    maxSupportedVersion: Int): Int
```

parseVersion ...FIXME

Note

`parseVersion` is used when:

- `KafkaSourceInitialOffsetWriter` is requested to `deserialize metadata`
- `KafkaSource` is requested for the `initial partition offsets`
- `CommitLog` is requested to `deserialize metadata`
- `CompactibleFileStreamLog` is requested to `deserialize metadata`
- `OffsetSeqLog` is requested to `deserialize metadata`
- `RateStreamMicroBatchReader` is requested to `deserialize metadata`

purgeAfter Method

```
purgeAfter(thresholdBatchId: Long): Unit
```

purgeAfter ...FIXME

Note

`purgeAfter` seems to be used exclusively in tests.

Writing Batch Metadata to File (Metadata Log)

— `writeBatchToFile` Internal Method

```
writeBatchToFile(  
    metadata: T,  
    path: Path): Unit
```

`writeBatchToFile` requests the [CheckpointFileManager](#) to [createAtomic](#) (for the specified `path` and the `overwriteIfPossible` flag disabled).

`writeBatchToFile` then [serializes the metadata](#) (to the `CancellableFSDataOutputStream` output stream) and closes the stream.

In case of an exception, `writeBatchToFile` simply requests the `CancellableFSDataOutputStream` output stream to `cancel` (so that the output file is not generated) and re-throws the exception.

Note	<code>writeBatchToFile</code> is used exclusively when <code>HDFSMetadataLog</code> is requested to store (persist) metadata of a streaming batch .
------	---

Retrieving Ordered Batch Metadata Files

— `getOrderedBatchFiles` Method

```
getOrderedBatchFiles(): Array[FileStatus]
```

`getOrderedBatchFiles` ...FIXME

Note	<code>getOrderedBatchFiles</code> does not seem to be used at all.
------	--

Internal Properties

Name	Description
batchFilesFilter	<p>Hadoop's PathFilter of batch files (with names being long numbers)</p> <p>Used when:</p> <ul style="list-style-type: none">• <code>CompactibleFileStreamLog</code> is requested for the <code>compactInterval</code>• <code>HDFSMetadataLog</code> is requested to get batch metadata, getLatest, getOrderedBatchFiles, purge, and purgeAfter
fileManager	<p>CheckpointFileManager</p> <p>Used when...FIXME</p>

CommitLog — HDFSMetadataLog for Offset Commit Log

`CommitLog` is an [HDFSMetadataLog](#) with [CommitMetadata](#) metadata.

`CommitLog` is [created](#) exclusively for the [offset commit log](#) of [StreamExecution](#).

`CommitLog` uses `CommitMetadata` for the metadata with **nextBatchWatermarkMs** attribute (of type `Long` and the default `0`).

`CommitLog` [writes](#) commit metadata to files with names that are offsets.

```
$ ls -tr [checkpoint-directory]/commits
0 1 2 3 4 5 6 7 8 9

$ cat [checkpoint-directory]/commits/8
v1
{"nextBatchWatermarkMs": 0}
```

`CommitLog` uses **1** for the version.

`CommitLog` (like the parent [HDFSMetadataLog](#)) takes the following to be created:

- `SparkSession`
- Path of the metadata log directory

Serializing Metadata (Writing Metadata to Persistent Storage) — `serialize` Method

```
serialize(
    metadata: CommitMetadata,
    out: OutputStream): Unit
```

Note	<code>serialize</code> is part of HDFSMetadataLog Contract to write a metadata in serialized format.
------	--

`serialize` writes out the [version](#) prefixed with `v` on a single line (e.g. `v1`) followed by the given `CommitMetadata` in JSON format.

Deserializing Metadata — `deserialize` Method

```
deserialize(in: InputStream): CommitMetadata
```

Note

`deserialize` is part of [HDFSMetadataLog Contract](#) to deserialize a metadata (from an `InputStream`).

`deserialize` simply reads (`deserializes`) two lines from the given `InputStream` for `version` and the `nextBatchWatermarkMs` attribute.

add Method

```
add(batchId: Long): Unit
```

`add` ...FIXME

Note

`add` is used when...FIXME

add Method

```
add(batchId: Long, metadata: String): Boolean
```

Note

`add` is part of [MetadataLog Contract](#) to...FIXME.

`add` ...FIXME

CommitMetadata

CommitMetadata is...FIXME

OffsetSeqLog — Hadoop DFS-based Metadata Storage of OffsetSeqs

`OffsetSeqLog` is a [Hadoop DFS-based metadata storage](#) for `OffsetSeq` metadata.

`OffsetSeqLog` uses `OffsetSeq` for metadata which holds an ordered collection of offsets and optional metadata (as `OffsetSeqMetadata` for event-time watermark).

`OffsetSeqLog` is [created](#) exclusively for the [write-ahead log \(WAL\)](#) of offsets of [stream execution engines](#) (i.e. `ContinuousExecution` and `MicroBatchExecution`).

`OffsetSeqLog` uses `1` for the version when [serializing](#) and [deserializing](#) metadata.

Creating OffsetSeqLog Instance

`OffsetSeqLog` (like the parent [HDFSMetadataLog](#)) takes the following to be created:

- `SparkSession`
- Path of the metadata log directory

Serializing Metadata (Writing Metadata in Serialized Format) — `serialize` Method

```
serialize(  
    offsetSeq: OffsetSeq,  
    out: OutputStream): Unit
```

Note	<code>serialize</code> is part of HDFSMetadataLog Contract to serialize metadata (write metadata in serialized format).
------	---

`serialize` firstly writes out the `version` prefixed with `v` on a single line (e.g. `v1`) followed by the [optional metadata](#) in JSON format.

`serialize` then writes out the `offsets` in JSON format, one per line.

Note	No offsets to write in <code>offsetSeq</code> for a streaming source is marked as <code>-</code> (a dash) in the log.
------	---

```
$ ls -tr [checkpoint-directory]/offsets
0 1 2 3 4 5 6

$ cat [checkpoint-directory]/offsets/6
v1
{"batchWatermarkMs":0,"batchTimestampMs":1502872590006,"conf":{"spark.sql.shuffle.partitions":"200","spark.sql.streaming.stateStore.providerClass":"org.apache.spark.sql.execution.streaming.state.HDFSBackedStateStoreProvider"}}
51
```

Deserializing Metadata (Reading OffsetSeq from Serialized Format) — `deserialize` Method

`deserialize(in: InputStream): OffsetSeq`

Note	<code>deserialize</code> is part of HDFSMetadataLog Contract to deserialize metadata (read metadata from serialized format).
------	--

`deserialize` firstly parses the [version](#) on the first line.

`deserialize` reads the optional metadata (with an empty line for metadata not available).

`deserialize` creates a [SerializedOffset](#) for every line left.

In the end, `deserialize` creates a [OffsetSeq](#) for the optional metadata and the [SerializedOffsets](#).

When there are no lines in the [InputStream](#), `deserialize` throws an [IllegalStateException](#):

Incomplete log file

OffsetSeq

`OffsetSeq` is the metadata managed by [Hadoop DFS-based metadata storage](#).

`OffsetSeq` is [created](#) (possibly using the `fill` factory methods) when:

- `OffsetSeqLog` is requested to [deserialize metadata](#) (retrieve metadata from a persistent storage)
- `StreamProgress` is requested to [convert itself to OffsetSeq](#) (most importantly when `MicroBatchExecution` stream execution engine is requested to [construct the next streaming micro-batch](#) to [commit available offsets for a batch to the write-ahead log](#))
- `ContinuousExecution` stream execution engine is requested to [get start offsets](#) and [addOffset](#)

Creating OffsetSeq Instance

`OffsetSeq` takes the following when created:

- Collection of optional [Offsets](#) (with `None` for [streaming sources with no new data available](#))
- Optional [OffsetSeqMetadata](#) (default: `None`)

Converting to StreamProgress — `toStreamProgress` Method

```
toStreamProgress(  
    sources: Seq[BaseStreamingSource]): StreamProgress
```

`toStreamProgress` creates a new [StreamProgress](#) and adds the [streaming sources](#) for which there are new [offsets](#) available.

Note	Offsets is a collection with <i>holes</i> (empty elements) for streaming sources with no new data available.
------	--

`toStreamProgress` throws an `AssertionError` if the number of the input `sources` does not match the [offsets](#):

```
There are [[offsets.size]] sources in the checkpoint offsets and now there are [[sources.size]] sources requested by the query. Cannot continue.
```

Note

- `toStreamProgress` is used when:
- `MicroBatchExecution` is requested to [populate start offsets from offsets and commits checkpoints](#) and [construct \(or skip\) the next streaming micro-batch](#)
 - `ContinuousExecution` is requested for [start offsets](#)

Textual Representation — `toString` Method

```
toString: String
```

Note

`toString` is part of the [java.lang.Object](#) contract for the string representation of the object.

`toString` simply converts the [Offsets](#) to JSON (if an offset is available) or `-` (a dash if an offset is not available for a streaming source at that position).

Creating OffsetSeq Instance — `fill` Factory Methods

```
fill(
  offsets: Offset*): OffsetSeq (1)
fill(
  metadata: Option[String],
  offsets: Offset*): OffsetSeq
```

1. Uses no metadata (`None`)

`fill` simply creates an [OffsetSeq](#) for the given variable sequence of [Offsets](#) and the optional [OffsetSeqMetadata](#) (in JSON format).

Note

- `fill` is used when:
- `OffsetSeqLog` is requested to [deserialize metadata](#)
 - `ContinuousExecution` stream execution engine is requested to [get start offsets](#) and [addOffset](#)

CompactibleFileStreamLog Contract— Compactible Metadata Logs

`CompactibleFileStreamLog` is the [extension](#) of the `HDFSMetadataLog` contract for [compactible metadata logs](#) that [compactLogs](#) every [compact interval](#).

`CompactibleFileStreamLog` uses `spark.sql.streaming.minBatchesToRetain` configuration property (default: `100`) for [deleteExpiredLog](#).

`CompactibleFileStreamLog` uses `.compact` suffix for `batchIdToPath`, `getBatchIdFromFileName`, and the `compactInterval`.

Table 1. CompactibleFileStreamLog Contract (Abstract Methods Only)

Method	Description
<code>compactLogs</code>	<code>compactLogs(logs: Seq[T]): Seq[T]</code> Used when <code>CompactibleFileStreamLog</code> is requested to compact and allFiles
<code>defaultCompactInterval</code>	<code>defaultCompactInterval: Int</code> Default compaction interval Used exclusively when <code>CompactibleFileStreamLog</code> is requested for the <code>compactInterval</code>
<code>fileCleanupDelayMs</code>	<code>fileCleanupDelayMs: Long</code> Used exclusively when <code>CompactibleFileStreamLog</code> is requested to deleteExpiredLog
<code>isDeletingExpiredLog</code>	<code>isDeletingExpiredLog: Boolean</code> Used exclusively when <code>CompactibleFileStreamLog</code> is requested to store (add) metadata of a streaming batch

Table 2. CompactibleFileStreamLogs

CompactibleFileStreamLog	Description
FileStreamSinkLog	
FileStreamSourceLog	CompactibleFileStreamLog (of <code>FileEntry</code> metadata) of FileStreamSource

Creating CompactibleFileStreamLog Instance

`CompactibleFileStreamLog` takes the following to be created:

- Metadata version
- `SparkSession`
- Path of the metadata log directory

Note	<code>CompactibleFileStreamLog</code> is a Scala abstract class and cannot be created directly. It is created indirectly for the concrete CompactibleFileStreamLogs .
------	---

batchIdToPath Method

```
batchIdToPath(batchId: Long): Path
```

Note	<code>batchIdToPath</code> is part of the HDFSMetadataLog Contract to...FIXME.
------	--

`batchIdToPath` ...FIXME

pathToBatchId Method

```
pathToBatchId(path: Path): Long
```

Note	<code>pathToBatchId</code> is part of the HDFSMetadataLog Contract to...FIXME.
------	--

`pathToBatchId` ...FIXME

isBatchFile Method

```
isBatchFile(path: Path): Boolean
```

Note`isBatchFile` is part of the [HDFSMetadataLog Contract](#) to...FIXME.`isBatchFile` ...FIXME

Serializing Metadata (Writing Metadata in Serialized Format) — `serialize` Method

```
serialize(
  logData: Array[T],
  out: OutputStream): Unit
```

Note`serialize` is part of the [HDFSMetadataLog Contract](#) to serialize metadata (write metadata in serialized format).

`serialize` firstly writes the version header (`v` and the `metadataLogVersion`) out to the given output stream (in `UTF_8`).

`serialize` then writes the log data (serialized using [Json4s \(with Jackson binding\)](#) library). Entries are separated by new lines.

Deserializing Metadata — `deserialize` Method

```
deserialize(in: InputStream): Array[T]
```

Note`deserialize` is part of the [HDFSMetadataLog Contract](#) to...FIXME.`deserialize` ...FIXME

Storing Metadata Of Streaming Batch — `add` Method

```
add(
  batchId: Long,
  logs: Array[T]): Boolean
```

Note`add` is part of the [HDFSMetadataLog Contract](#) to store metadata for a batch.`add` ...FIXME

allFiles Method

```
allFiles(): Array[T]
```

`allFiles` ...FIXME

Note

- `allFiles` is used when:
- `FileStreamSource` is created
 - `MetadataLogFileIndex` is created

compact Internal Method

```
compact(
  batchId: Long,
  logs: Array[T]): Boolean
```

`compact` [getValidBatchesBeforeCompactionBatch](#) (with the streaming batch and the [compact interval](#)).

`compact` ...FIXME

In the end, `compact` [compactLogs](#) and requests the parent `HDFSMetadataLog` to [persist](#) metadata of a streaming batch (to a metadata log file).

Note

`compact` is used exclusively when `CompactibleFileStreamLog` is requested to [persist metadata of a streaming batch](#).

getValidBatchesBeforeCompactionBatch Object Method

```
getValidBatchesBeforeCompactionBatch(
  compactionBatchId: Long,
  compactInterval: Int): Seq[Long]
```

`getValidBatchesBeforeCompactionBatch` ...FIXME

Note

`getValidBatchesBeforeCompactionBatch` is used exclusively when `CompactibleFileStreamLog` is requested to [compact](#).

isCompactionBatch Object Method

```
isCompactionBatch(batchId: Long, compactInterval: Int): Boolean
```

`isCompactionBatch` ...FIXME

Note

- `isCompactionBatch` is used when:
 - `CompactibleFileStreamLog` is requested to `batchIdToPath`, store the `metadata of a batch`, `deleteExpiredLog`, and `getValidBatchesBeforeCompactionBatch`
 - `FileStreamSourceLog` is requested to `store the metadata of a batch` and get

getBatchIdFromFile Name Object Method

```
getBatchIdFromFile Name(fileName: String): Long
```

`getBatchIdFromFile Name` simply removes the `.compact` suffix from the given `fileName` and converts the remaining part to a number.

Note

- `getBatchIdFromFile Name` is used when `CompactibleFileStreamLog` is requested to `pathToBatchId`, `isBatchFile`, and `deleteExpiredLog`.

deleteExpiredLog Internal Method

```
deleteExpiredLog(
  currentBatchId: Long): Unit
```

`deleteExpiredLog` does nothing and simply returns when the current batch ID incremented (`currentBatchId + 1`) is below the `compact interval` plus the `minBatchesToRetain`.

`deleteExpiredLog` ...FIXME

Note

- `deleteExpiredLog` is used exclusively when `CompactibleFileStreamLog` is requested to `store metadata of a streaming batch`.

Internal Properties

Name	Description
<code>compactInterval</code>	Compact interval

FileStreamSourceLog

`FileStreamSourceLog` is a concrete `CompactibleFileStreamLog` (of `FileEntry` metadata) of `FileStreamSource`.

`FileStreamSourceLog` uses a fixed-size `cache` of metadata of compaction batches.

`FileStreamSourceLog` uses `spark.sql.streaming.fileSource.log.compactInterval` configuration property (default: `10`) for the `default` compaction interval.

`FileStreamSourceLog` uses `spark.sql.streaming.fileSource.log.cleanupDelay` configuration property (default: `10` minutes) for the `fileCleanupDelayMs`.

`FileStreamSourceLog` uses `spark.sql.streaming.fileSource.log.deletion` configuration property (default: `true`) for the `isDeletingExpiredLog`.

Creating FileStreamSourceLog Instance

`FileStreamSourceLog` (like the parent `CompactibleFileStreamLog`) takes the following to be created:

- Metadata version
- `SparkSession`
- Path of the metadata log directory

Storing (Adding) Metadata of Streaming Batch — `add` Method

```
add(
  batchId: Long,
  logs: Array[FileEntry]): Boolean
```

Note

`add` is part of the `MetadataLog Contract` to store (`add`) metadata of a streaming batch.

`add` requests the parent `CompactibleFileStreamLog` to `store` metadata (possibly `compacting` logs if the batch is `compaction`).

If so (and this is a compaction batch), `add` adds the batch and the logs to `fileEntryCache` internal registry (and possibly removing the eldest entry if the size is above the `cacheSize`).

get Method

```
get(  
  startId: Option[Long],  
  endId: Option[Long]): Array[(Long, Array[FileEntry])]
```

Note

`get` is part of the [MetadataLog Contract](#) to...FIXME.

`get` ...FIXME

Internal Properties

Name	Description
<code>cacheSize</code>	<p>Size of the fileEntryCache that is exactly the compact interval</p> <p>Used when the fileEntryCache is requested to add a new entry in add and get a compaction batch</p>
<code>fileEntryCache</code>	<p>Metadata of a streaming batch (FileEntry) per batch ID (LinkedHashMap[Long, Array[FileEntry]]) of size configured using the cacheSize</p> <ul style="list-style-type: none"> • New entry added for a compaction batch when storing (adding) metadata of a streaming batch <p>Used when get (for a compaction batch)</p>

OffsetSeqMetadata

`OffsetSeqMetadata` holds the metadata for the current streaming batch:

- Event-time watermark
- Batch timestamp (in millis)
- **Streaming configuration** with `spark.sql.shuffle.partitions` and `spark.sql.streaming.stateStore.providerClass` Spark properties

Note

`OffsetSeqMetadata` is used mainly when `IncrementalExecution` is created.

`OffsetSeqMetadata` considers some configuration properties as **relevantSQLConfs**:

- `SHUFFLE_PARTITIONS`
- `STATE_STORE_PROVIDER_CLASS`
- `STREAMING_MULTIPLE_WATERMARK_POLICY`
- `FLATMAPGROUPSWITHSTATE_STATE_FORMAT_VERSION`
- `STREAMING_AGGREGATION_STATE_FORMAT_VERSION`

`relevantSQLConfs` are used when `OffsetSeqMetadata` is created and is requested to `setSessionConf`.

Creating OffsetSeqMetadata — `apply` Factory Method

```
apply(
  batchWatermarkMs: Long,
  batchTimestampMs: Long,
  sessionConf: RuntimeConfig): OffsetSeqMetadata
```

`apply ...FIXME`

Note

`apply` is used when...FIXME

`setSessionConf` Method

```
setSessionConf(metadata: OffsetSeqMetadata, sessionConf: RuntimeConfig): Unit
```

`setSessionConf ...FIXME`

Note	<code>setSessionConf</code> is used when...FIXME
-------------	--

CheckpointFileManager Contract

`CheckpointFileManager` is the abstraction of [checkpoint managers](#) that manage checkpoint files (metadata of streaming batches) on Hadoop DFS-compatible file systems.

`CheckpointFileManager` is created per [spark.sql.streaming.checkpointFileManagerClass](#) configuration property if defined before reverting to the available [checkpoint managers](#).

`CheckpointFileManager` is used exclusively by [HDFSMetadataLog](#), [StreamMetadata](#) and [HDFSBackedStateStoreProvider](#).

Table 1. CheckpointFileManager Contract

Method	Description
<code>createAtomic</code>	<pre>createAtomic(path: Path, overwriteIfPossible: Boolean): CancellableFSDataOutputSteam</pre> <p>Used when:</p> <ul style="list-style-type: none"> • <code>HDFSMetadataLog</code> is requested to store metadata for a batch (that writeBatchToFile) • <code>StreamMetadata</code> helper object is requested to persist metadata • <code>HDFSBackedStateStore</code> is requested for the deltaFileStream • <code>HDFSBackedStateStoreProvider</code> is requested to writeSnapshotFile
<code>delete</code>	<pre>delete(path: Path): Unit</pre> <p>Deletes a path recursively (if it exists)</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>RenameBasedFSDataOutputStream</code> is requested to cancel • <code>CompactibleFileStreamLog</code> is requested to store metadata for a batch (that deleteExpiredLog) • <code>HDFSMetadataLog</code> is requested to remove expired metadata and purgeAfter • <code>HDFSBackedStateStoreProvider</code> is requested to do maintenance (that cleans up)

	<code>exists(path: Path): Boolean</code>
<code>exists</code>	Used when <code>HDFSMetadataLog</code> is created (to create the metadata directory) and requested for metadata of a batch
<code>isLocal</code>	<code>isLocal: Boolean</code> Does not seem to be used.
<code>list</code>	<code>list(path: Path): Array[FileStatus] (1)</code> <code>list(path: Path, filter: PathFilter): Array[FileStatus]</code> 1. Uses <code>PathFilter</code> that accepts all files in the path Used when: <ul style="list-style-type: none">• <code>HDFSBackedStateStoreProvider</code> is requested for all delta and snapshot files• <code>CompactibleFileStreamLog</code> is requested for the compact interval and to <code>deleteExpiredLog</code>• <code>HDFSMetadataLog</code> is requested for metadata of one or more batches, the latest committed batch, ordered batch metadata files, to remove expired metadata and <code>purgeAfter</code>
<code>mkdirs</code>	<code>mkdirs(path: Path): Unit</code> Used when: <ul style="list-style-type: none">• <code>HDFSMetadataLog</code> is created• <code>HDFSBackedStateStoreProvider</code> is requested to initialize
<code>open</code>	<code>open(path: Path): FSDataInputStream</code> Opens a file (by the given path) for reading Used when: <ul style="list-style-type: none">• <code>HDFSMetadataLog</code> is requested for metadata of a batch• <code>HDFSBackedStateStoreProvider</code> is requested to retrieve the state store for a specified version (that <code>updateFromDeltaFile</code>), and <code>readSnapshotFile</code>

Table 2. CheckpointFileManagers

CheckpointFileManager	Description
FileContextBasedCheckpointFileManager	Default CheckpointFileManager that uses Hadoop's FileContext API for managing checkpoint files (unless spark.sql.streaming.checkpointFileManagerClass configuration property is used)
FileSystemBasedCheckpointFileManager	Basic CheckpointFileManager that uses Hadoop's FileSystem API for managing checkpoint files (that assumes that the implementation of <code>FileSystem.rename()</code> is atomic or the correctness and fault-tolerance Structured Streaming is not guaranteed)

Creating CheckpointFileManager Instance — `create` Object Method

```
create(
  path: Path,
  hadoopConf: Configuration): CheckpointFileManager
```

`create` finds [spark.sql.streaming.checkpointFileManagerClass](#) configuration property in the `hadoopConf` configuration.

If found, `create` simply instantiates whatever `CheckpointFileManager` implementation is defined.

If not found, `create` creates a [FileContextBasedCheckpointFileManager](#).

In case of `UnsupportedFileSystemException`, `create` prints out the following WARN message to the logs and creates (*falls back on*) a [FileSystemBasedCheckpointFileManager](#).

```
Could not use FileContext API for managing Structured Streaming checkpoint files at [path]. Using FileSystem API instead for managing log files. If the implementation of FileSystem.rename() is not atomic, then the correctness and fault-tolerance of your Structured Streaming is not guaranteed.
```

Note

`create` is used when:

- `HDFSMetadataLog` is [created](#)
- `StreamMetadata` helper object is requested to [write metadata to a file](#) (when `StreamExecution` is [created](#))
- `HDFSBackedStateStoreProvider` is requested for the [CheckpointFileManager](#)

FileContextBasedCheckpointFileManager

FileContextBasedCheckpointFileManager is...FIXME

FileSystemBasedCheckpointFileManager — CheckpointFileManager on Hadoop's FileSystem API

`FileSystemBasedCheckpointFileManager` is a [CheckpointFileManager](#) that uses Hadoop's [FileSystem](#) API for managing checkpoint files:

- `list` uses [FileSystem.listStatus](#)
- `mkdirs` uses [FileSystem.mkdirs](#)
- `createTempFile` uses [FileSystem.create](#) (with overwrite enabled)
- `createAtomic` uses [RenameBasedFSDataOutputStream](#)
- `open` uses [FileSystem.open](#)
- `exists` uses [FileSystem.getFileStatus](#)
- `renameTempFile` uses [FileSystem.rename](#)
- `delete` uses [FileSystem.delete](#) (with recursive enabled)
- `isLocal` is `true` for the [FileSystem](#) being [LocalFileSystem](#) or [RawLocalFileSystem](#)

`FileSystemBasedCheckpointFileManager` is [created](#) exclusively when [CheckpointFileManager](#) helper object is requested for a [CheckpointFileManager](#) (for [HDFSMetadataLog](#), [StreamMetadata](#) and [HDFSBackedStateStoreProvider](#)).

`FileSystemBasedCheckpointFileManager` is a [RenameHelperMethods](#) for [atomicity](#) by "write-to-temp-file-and-rename".

Creating FileSystemBasedCheckpointFileManager Instance

`FileSystemBasedCheckpointFileManager` takes the following to be created:

- Checkpoint directory (Hadoop's [Path](#))
- Configuration (Hadoop's [Configuration](#))

`FileSystemBasedCheckpointFileManager` initializes the [internal properties](#).

Internal Properties

Name	Description
fs	Hadoop's FileSystem of the checkpoint directory

Offset — Read Position of Streaming Query

`offset` is the [base](#) of [stream positions](#) that represent progress of a streaming query in [json](#) format.

Table 1. Offset Contract (Abstract Methods Only)

Method	Description
<code>json</code>	<pre>String json()</pre> <p>JSON string encoding</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>MicroBatchExecution</code> stream execution engine is requested to construct the next streaming micro-batch and run a streaming micro-batch (with <code>MicroBatchReader</code> sources) • <code>offsetSeq</code> is requested for the textual representation • <code>offsetSeqLog</code> is requested to serialize metadata (write metadata in serialized format) • <code>ProgressReporter</code> is requested to record trigger offsets • <code>ContinuousExecution</code> stream execution engine is requested to run a streaming query in continuous mode and commit an epoch

Table 2. Offsets

Offset	Description
<code>ContinuousMemoryStreamOffset</code>	
<code>FileStreamSourceOffset</code>	
<code>KafkaSourceOffset</code>	
<code>LongOffset</code>	
<code>RateStreamOffset</code>	
<code>SerializedOffset</code>	JSON-serialized offset
<code>TextSocketOffset</code>	

StreamProgress — Collection of Offsets per Streaming Source

`StreamProgress` is a collection of [Offsets](#) per streaming source.

`StreamProgress` is [created](#) when:

- `StreamExecution` is [created](#) (and creates [committed](#) and [available](#) offsets)
- `OffsetSeq` is requested to [convert to StreamProgress](#)

`StreamProgress` is an extension of Scala's `scala.collection.immutable.Map` with [streaming sources](#) as keys and their [Offsets](#) as values.

Creating StreamProgress Instance

`StreamProgress` takes the following to be created:

- Optional collection of [offsets](#) per [streaming source](#) (`Map[BaseStreamingSource, Offset]`) (default: empty)

Looking Up Offset by Streaming Source — `get` Method

```
get(key: BaseStreamingSource): Option[Offset]
```

Note	<code>get</code> is part of the Scala's <code>scala.collection.MapLike</code> to...FIXME.
------	---

`get` simply looks up an [Offsets](#) for the given [BaseStreamingSource](#) in the [baseMap](#).

`++` Method

```
++(  
  updates: GenTraversableOnce[(BaseStreamingSource, Offset)]): StreamProgress
```

`++` simply creates a new [StreamProgress](#) with the [baseMap](#) and the given [updates](#).

Note	<code>++</code> is used exclusively when <code>OffsetSeq</code> is requested to convert to StreamProgress .
------	---

Converting to OffsetSeq — `toOffsetSeq` Method

```
toOffsetSeq(  
    sources: Seq[BaseStreamingSource],  
    metadata: OffsetSeqMetadata): OffsetSeq
```

`toOffsetSeq` creates a [OffsetSeq](#) with offsets that are [looked up](#) for every [BaseStreamingSource](#).

Note	<p><code>toOffsetSeq</code> is used when:</p> <ul style="list-style-type: none">• <code>MicroBatchExecution</code> stream execution engine is requested to construct the next streaming micro-batch (to commit available offsets for a batch to the write-ahead log)• <code>StreamExecution</code> is requested to run stream processing (that failed with a Throwable)
------	--

Continuous Stream Processing (Structured Streaming V2)

Continuous Stream Processing is one of the two stream processing engines in [Spark Structured Streaming](#) that is used for execution of structured streaming queries with [Trigger.Continuous](#) trigger.

Note

The other feature-richer stream processing engine is [Micro-Batch Stream Processing](#).

Continuous Stream Processing execution engine uses the novel **Data Source API V2** (Spark SQL) and for the very first time makes stream processing truly **continuous**.

Tip

Read up on [Data Source API V2](#) in [The Internals of Spark SQL](#) book.

Because of the two innovative changes Continuous Stream Processing is often referred to as **Structured Streaming V2**.

```
import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._

val sq = spark
  .readStream
  .format("rate")
  .load
  .writeStream
  .format("console")
  .option("truncate", false)
  .trigger(Trigger.Continuous(15.seconds)) // <-- Uses ContinuousExecution for execution
  .queryName("rate2console")
  .start

scala> :type sq
org.apache.spark.sql.streaming.StreamingQuery

assert(sq.isActive)

// sq.stop
```

Under the covers, Continuous Stream Processing uses [ContinuousExecution](#) stream execution engine. When requested to [run an activated streaming query](#), [ContinuousExecution](#) adds [WriteToContinuousDataSourceExec](#) physical operator as the top-level operator in the physical query plan of the streaming query.

```

scala> :type sq
org.apache.spark.sql.streaming.StreamingQuery

scala> sq.explain
== Physical Plan ==
WriteToContinuousDataSource ConsoleWriter[numRows=20, truncate=false]
+- *(1) Project [timestamp#758, value#759L]
  +- *(1) ScanV2 rate[timestamp#758, value#759L]

```

From now on, you may think of a streaming query as a soon-to-be-generated [ContinuousWriteRDD](#) - an RDD data structure that Spark developers use to describe a distributed computation.

When the streaming query is started (and the top-level `writeToContinuousDataSourceExec` physical operator is requested to [execute and generate a recipe for a distributed computation \(as an `RDD\[InternalRow\]`\)](#)), it simply requests the underlying `ContinuousWriteRDD` to collect.

That collect operator is how a Spark job is run (as tasks over all partitions of the RDD) as described by the [ContinuousWriteRDD.compute](#) "protocol" (a recipe for the tasks to be scheduled to run on Spark executors).

Job Id (Job Group)	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0 (5b9edde1-e8cb-4ed5-a487-e841af09eba6)	rate2console id = b3074d1c-adbd-456f-a730-8aa0fc5bfc02 runId = 5b9edde1-e8cb-4ed5-a487-e841af09eba6 batch = init start at <console>:33	2019/06/03 11:13:27	2.3 min	0/1	0/5 (5 running)

Figure 1. Creating Instance of StreamExecution

While the [tasks are computing partitions](#) (of the `ContinuousWriteRDD`), they keep running [until killed or completed](#). And that's the *ingenious design trick* of how the streaming query (as a Spark job with the distributed tasks running on executors) runs continuously and indefinitely.

When `DataStreamReader` is requested to [create a streaming query for a `ContinuousReadSupport` data source](#), it creates...FIXME

ContinuousExecution — Stream Execution Engine of Continuous Stream Processing

`ContinuousExecution` is the [stream execution engine](#) of [Continuous Stream Processing](#).

`ContinuousExecution` is [created](#) when `StreamingQueryManager` is requested to [create a streaming query](#) with a `StreamWriterSupport` sink and a `ContinuousTrigger` (when `DataStreamWriter` is requested to [start an execution of the streaming query](#)).

`ContinuousExecution` can only run streaming queries with `StreamingRelationV2` with `ContinuousReadSupport` data source.

`ContinuousExecution` supports one `ContinuousReader` only in a [streaming query](#) (and asserts it when [addOffset](#) and [committing an epoch](#)). When requested for available [streaming sources](#), `ContinuousExecution` simply gives the [single ContinuousReader](#).

```
import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._

val sq = spark
  .readStream
  .format("rate")
  .load
  .writeStream
  .format("console")
  .option("truncate", false)
  .trigger(Trigger.Continuous(1.minute)) // <-- Gives ContinuousExecution
  .queryName("rate2console")
  .start

import org.apache.spark.sql.streaming.StreamingQuery
assert(sq.isInstanceOf[StreamingQuery])

// The following gives access to the internals
// And to ContinuousExecution
import org.apache.spark.sql.execution.streaming.StreamingQueryWrapper
val engine = sq.asInstanceOf[StreamingQueryWrapper].streamingQuery
import org.apache.spark.sql.execution.streaming.StreamExecution
assert(engine.isInstanceOf[StreamExecution])

import org.apache.spark.sql.execution.streaming.continuous.ContinuousExecution
val continuousEngine = engine.asInstanceOf[ContinuousExecution]
assert(continuousEngine.trigger == Trigger.Continuous(1.minute))
```

When [created](#) (for a streaming query), `ContinuousExecution` is given the [analyzed logical plan](#). The analyzed logical plan is immediately transformed to include a `ContinuousExecutionRelation` for every `StreamingRelationV2` with `ContinuousReadSupport` data source (and is the [logical plan](#) internally).

Note

`ContinuousExecution` uses the same instance of `continuousExecutionRelation` for the same instances of `StreamingRelationV2` with `ContinuousReadSupport` data source.

When requested to [run the streaming query](#), `ContinuousExecution` collects `ContinuousReadSupport` data sources (inside `ContinuousExecutionRelation`) from the analyzed logical plan and requests each and every `continuousReadSupport` to [create a `ContinuousReader`](#) (that are stored in `continuousSources` internal registry).

`ContinuousExecution` uses `__epoch_coordinator_id` local property for...FIXME

`ContinuousExecution` uses `__continuous_start_epoch` local property for...FIXME

`ContinuousExecution` uses `__continuous_epoch_interval` local property for...FIXME

Tip

Enable `ALL` logging level for `org.apache.spark.sql.execution.streaming.continuous.ContinuousExecution` to see what happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.execution.streaming.continuous.ContinuousExecution=INFO
```

Refer to [Logging](#).

Running Activated Streaming Query

— `runActivatedStream` Method

```
runActivatedStream(sparkSessionForStream: SparkSession): Unit
```

Note

`runActivatedStream` is part of [StreamExecution Contract](#) to run a streaming query.

`runActivatedStream` simply [runs the streaming query in continuous mode](#) as long as the state is [ACTIVE](#).

Running Streaming Query in Continuous Mode

— `runContinuous` Internal Method

```
runContinuous(sparkSessionForQuery: SparkSession): Unit
```

`runContinuous` initializes the `continuousSources` internal registry by traversing the `analyzed logical plan` to find `ContinuousExecutionRelation` leaf logical operators and requests their `ContinuousReadSupport` data sources to create a `ContinuousReader` (with the `sources` metadata directory under the `checkpoint` directory).

`runContinuous` initializes the `uniqueSources` internal registry to be the `continuousSources` distinct.

`runContinuous` gets the start offsets (they may or may not be available).

`runContinuous` transforms the `analyzed logical plan`. For every `ContinuousExecutionRelation` `runContinuous` finds the corresponding `ContinuousReader` (in the `continuousSources`), requests it to deserialize the start offsets (from their JSON representation), and then `setStartOffset`. In the end, `runContinuous` creates a `StreamingDataSourceV2Relation` (with the read schema of the `ContinuousReader` and the `ContinuousReader` itself).

`runContinuous` rewrites the transformed plan (with the `StreamingDataSourceV2Relation`) to use the new attributes from the source (the reader).

Note

`CurrentTimestamp` and `CurrentDate` expressions are not supported for continuous processing.

`runContinuous` requests the `StreamWriterSupport` to create a `StreamWriter` (with the run ID of the streaming query).

`runContinuous` creates a `WriteToContinuousDataSource` (with the `StreamWriter` and the transformed logical query plan).

`runContinuous` finds the only `ContinuousReader` (of the only `StreamingDataSourceV2Relation`) in the query plan with the `writeToContinuousDataSource`.

In `queryPlanning` time-tracking section, `runContinuous` creates an `IncrementalExecution` (that becomes the `lastExecution`) that is immediately executed (i.e. the entire query execution pipeline is executed up to and including `executedPlan`).

`runContinuous` sets the following local properties:

- `__is_continuous_processing` as `true`
- `__continuous_start_epoch` as the `currentBatchId`
- `__epoch_coordinator_id` as the `currentEpochCoordinatorId`, i.e. `runId` followed by `--` with a random UUID

- `__continuous_epoch_interval` as the interval of the `ContinuousTrigger`

`runContinuous` uses the `EpochCoordinatorRef` helper to create a remote reference to the `EpochCoordinator` RPC endpoint (with the `StreamWriter`, the `ContinuousReader`, the `currentEpochCoordinatorId`, and the `currentBatchId`).

Note

The `EpochCoordinator` RPC endpoint runs on the driver as the single point to coordinate epochs across partition tasks.

`runContinuous` creates a daemon `epoch update thread` and starts it immediately.

In `runContinuous` time-tracking section, `runContinuous` requests the physical query plan (of the `IncrementalExecution`) to execute (that simply requests the physical operator to `doExecute` and generate an `RDD[InternalRow]`).

Note

`runContinuous` is used exclusively when `ContinuousExecution` is requested to run an activated streaming query.

Epoch Update Thread

`runContinuous` creates an `epoch update thread` that...FIXME

Getting Start Offsets From Checkpoint — `getStartOffsets` Internal Method

```
getStartOffsets(sparkSessionToRunBatches: SparkSession): OffsetSeq
```

`getStartOffsets` ...FIXME

Note

`getStartOffsets` is used exclusively when `ContinuousExecution` is requested to run a streaming query in continuous mode.

Committing Epoch — `commit` Method

```
commit(epoch: Long): Unit
```

In essence, `commit` adds the given epoch to `commit log` and the `committedOffsets`, and requests the `ContinuousReader` to commit the corresponding offset. In the end, `commit` removes old log entries from the `offset` and `commit` logs (to keep `spark.sql.streaming.minBatchesToRetain` entries only).

Internally, `commit recordTriggerOffsets` (with the from and to offsets as the `committedOffsets` and `availableOffsets`, respectively).

At this point, `commit` may simply return when the `stream execution thread` is no longer alive (died).

`commit` requests the `commit log` to store a metadata for the epoch.

`commit` requests the single `ContinuousReader` to `deserialize the offset` for the epoch (from the `offset write-ahead log`).

`commit` adds the single `ContinuousReader` and the offset (for the epoch) to the `committedOffsets` registry.

`commit` requests the single `ContinuousReader` to `commit the offset`.

`commit` requests the `offset` and `commit` logs to `remove log entries` to keep `spark.sql.streaming.minBatchesToRetain` only.

`commit` then acquires the `awaitProgressLock`, wakes up all threads waiting for the `awaitProgressLockCondition` and in the end releases the `awaitProgressLock`.

Note	<code>commit</code> supports only one continuous source (registered in the <code>continuousSources</code> internal registry).
------	---

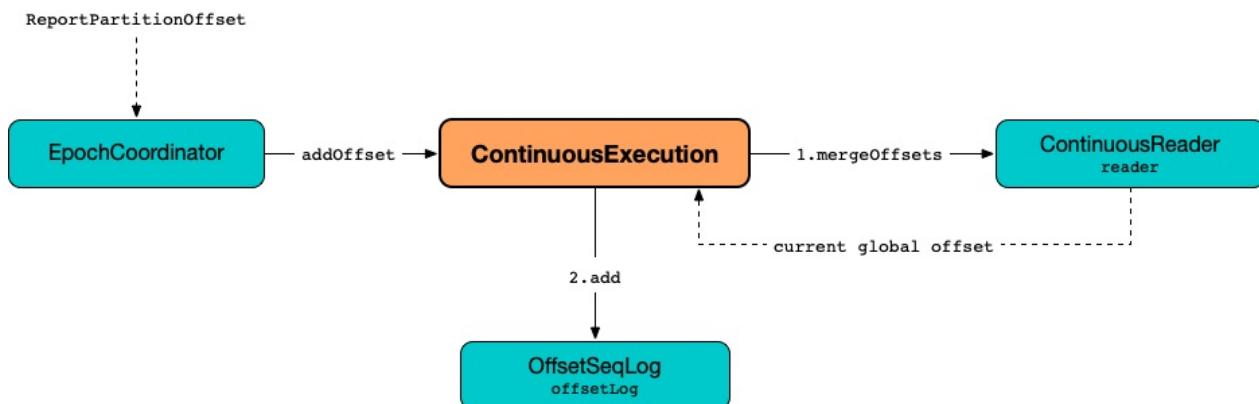
`commit` asserts that the given epoch is available in the `offsetLog` internal registry (i.e. the offset for the given epoch has been reported before).

Note	<code>commit</code> is used exclusively when <code>EpochCoordinator</code> is requested to <code>commitEpoch</code> .
------	---

addOffset Method

```
addOffset(
  epoch: Long,
  reader: ContinuousReader,
  partitionOffsets: Seq[PartitionOffset]): Unit
```

In essence, `addOffset` requests the given `ContinuousReader` to `mergeOffsets` (with the given `PartitionOffsets`) and then requests the `OffsetSeqLog` to `register the offset with the given epoch`.

Figure 1. `ContinuousExecution.addOffset`

Internally, `addOffset` requests the given `ContinuousReader` to `mergeOffsets` (with the given `PartitionOffsets`) and to get the current "global" offset back.

`addOffset` then requests the `OffsetSeqLog` to `add` the current "global" offset for the given epoch .

`addOffset` requests the `OffsetSeqLog` for the offset at the previous epoch.

If the offsets at the current and previous epochs are the same, `addOffset` turns the `noNewData` internal flag on.

`addOffset` then acquires the `awaitProgressLock`, wakes up all threads waiting for the `awaitProgressLockCondition` and in the end releases the `awaitProgressLock`.

Note	<code>addOffset</code> supports exactly one continuous source.
------	--

Note	<code>addOffset</code> is used exclusively when <code>EpochCoordinator</code> is requested to handle a <code>ReportPartitionOffset</code> message.
------	--

Analyzed Logical Plan of Streaming Query — `logicalPlan` Property

logicalPlan: LogicalPlan

Note	<code>logicalPlan</code> is part of <code>StreamExecution Contract</code> that is the analyzed logical plan of the streaming query.
------	---

`logicalPlan` resolves `StreamingRelationV2` leaf logical operators (with a `ContinuousReadSupport` source) to `ContinuousExecutionRelation` leaf logical operators.

Internally, `logicalPlan` transforms the `analyzed logical plan` as follows:

1. For every `StreamingRelationV2` leaf logical operator with a `ContinuousReadSupport` source, `logicalPlan` looks it up for the corresponding `ContinuousExecutionRelation` (if available in the internal lookup registry) or creates a `ContinuousExecutionRelation` (with the `ContinuousReadSupport` source, the options and the output attributes of the `StreamingRelationV2` operator)
2. For any other `streamingRelationV2`, `logicalPlan` throws an `UnsupportedOperationException`:

Data source [name] does not support continuous processing.

Creating ContinuousExecution Instance

`ContinuousExecution` takes the following when created:

- `SparkSession`
- The name of the structured query
- Path to the checkpoint directory (aka *metadata directory*)
- Analyzed logical query plan (`LogicalPlan`)
- `StreamWriterSupport`
- `Trigger`
- `Clock`
- `Output mode`
- Options (`Map[String, String]`)
- `deleteCheckpointOnStop` flag to control whether to delete the checkpoint directory on stop

`ContinuousExecution` initializes the `internal properties`.

Stopping Stream Processing (Execution of Streaming Query) — `stop` Method

`stop(): Unit`

Note `stop` is part of the `StreamingQuery Contract` to stop a streaming query.

`stop` transitions the streaming query to `TERMINATED` state.

If the `queryExecutionThread` is alive (i.e. it has been started and has not yet died), `stop` interrupts it and waits for this thread to die.

In the end, `stop` prints out the following INFO message to the logs:

```
Query [prettyIdString] was stopped
```

Note	<code>prettyIdString</code> is in the format of <code>queryName [id = [id], runId = [runId]]</code> .
------	---

awaitEpoch Internal Method

```
awaitEpoch(epoch: Long): Unit
```

`awaitEpoch ...FIXME`

Note	<code>awaitEpoch</code> seems to be used exclusively in tests.
------	--

Internal Properties

Name	Description		
continuousSources	<p>continuousSources: Seq[ContinuousReader]</p> <p>Registry of ContinuousReaders (in the analyzed logical plan of the streaming query)</p> <p>As asserted in commit and addOffset there could only be exactly one <code>ContinuousReaders</code> registered.</p> <p>Used when <code>ContinuousExecution</code> is requested to commit, getStartOffsets, and runContinuous</p> <p>Use sources to access the current value</p>		
currentEpochCoordinatorId	<p>FIXME</p> <p>Used when...FIXME</p>		
triggerExecutor	<p>TriggerExecutor for the Trigger:</p> <ul style="list-style-type: none"> • <code>ProcessingTimeExecutor</code> for ContinuousTrigger <p>Used when...FIXME</p> <table border="1"> <tr> <td>Note</td> <td>StreamExecution throws an <code>IllegalStateException</code> when the Trigger is not a ContinuousTrigger.</td> </tr> </table>	Note	StreamExecution throws an <code>IllegalStateException</code> when the Trigger is not a ContinuousTrigger .
Note	StreamExecution throws an <code>IllegalStateException</code> when the Trigger is not a ContinuousTrigger .		

ContinuousReadSupport Contract — Data Sources with Continuous Readers

`ContinuousReadSupport` is the abstraction of data sources with a `ContinuousReader` in [Continuous Stream Processing](#).

`ContinuousReadSupport` defines a single `createContinuousReader` method to create a `ContinuousReader`.

```
ContinuousReader createContinuousReader(
    Optional<StructType> schema,
    String checkpointLocation,
    DataSourceOptions options)
```

`createContinuousReader` is used when:

- `ContinuousExecution` is requested to [run a streaming query](#) (and finds `ContinuousExecutionRelations` in the [analyzed logical plan](#))
- `DataStreamReader` is requested to [create a streaming query for a ContinuousReadSupport data source](#)

Table 1. ContinuousReadSupports

ContinuousReadSupport	Description
ContinuousMemoryStream	
KafkaSourceProvider	
RateStreamProvider	
TextSocketSourceProvider	

ContinuousReader Contract — Data Source Readers in Continuous Stream Processing

`ContinuousReader` is the [extension](#) of Spark SQL's `DataSourceReader` (and `BaseStreamingSource`) contracts for [data source readers](#) in [Continuous Stream Processing](#).

`ContinuousReader` is part of the novel Data Source API V2 in Spark SQL.

Tip	Read up on Data Source API V2 in The Internals of Spark SQL book.
-----	---

Table 1. ContinuousReader Contract

Method	Description		
commit	<pre>void commit(offset end)</pre> <p>Commits the specified offset Used exclusively when <code>continuousExecution</code> is requested to commit an epoch</p>		
deserializeOffset	<pre>Offset deserializeOffset(String json)</pre> <p>Deserializes an offset from JSON representation Used when <code>ContinuousExecution</code> is requested to run a streaming query and commit an epoch</p>		
getStartOffset	<pre>Offset getStartOffset()</pre> <table border="1"> <tr> <td>Note</td> <td>Used exclusively in tests.</td> </tr> </table>	Note	Used exclusively in tests.
Note	Used exclusively in tests.		
mergeOffsets	<pre>Offset mergeOffsets(PartitionOffset[] offsets)</pre> <p>Used exclusively when <code>continuousExecution</code> is requested to addOffset</p>		
needsReconfiguration	<pre>boolean needsReconfiguration()</pre> <p>Indicates that the reader needs reconfiguration (e.g. to generate new input partitions) Used exclusively when <code>continuousExecution</code> is requested to run a streaming query in continuous mode</p>		
setStartOffset	<pre>void setStartOffset(Optional<Offset> start)</pre> <p>Used exclusively when <code>continuousExecution</code> is requested to run the streaming query in continuous mode.</p>		

Table 2. ContinuousReaders

ContinuousReader	Description
ContinuousMemoryStream	
KafkaContinuousReader	
RateStreamContinuousReader	
TextSocketContinuousReader	

ContinuousMemoryStream

ContinuousMemoryStream is...FIXME

RateStreamContinuousReader

RateStreamContinuousReader is a [ContinuousReader](#) that...FIXME

EpochCoordinator RPC Endpoint — Coordinating Epochs and Offsets Across Partition Tasks

`EpochCoordinator` is a `ThreadSafeRpcEndpoint` that tracks offsets and epochs (*coordinates epochs*) by handling [messages](#) (in [fire-and-forget one-way](#) and [request-response two-way](#) modes) from...FIXME

`EpochCoordinator` is created (using `create` factory method) when `ContinuousExecution` is requested to run a streaming query in continuous mode.

Table 1. EpochCoordinator RPC Endpoint's Messages

Message	Description
CommitPartitionEpoch <ul style="list-style-type: none"> • Partition ID • Epoch • DataSource API V2's <code>WriterCommitMessage</code> 	Sent out (in one-way asynchronous mode) exclusively when <code>ContinuousWriteRDD</code> is requested to compute a partition (after all rows were written down to a streaming sink)
GetCurrentEpoch	Sent out (in request-response synchronous mode) exclusively when <code>EpochMarkerGenerator</code> thread is requested to run
IncrementAndGetEpoch	Sent out (in request-response synchronous mode) exclusively when <code>ContinuousExecution</code> is requested to run a streaming query in continuous mode (and start a separate epoch update thread)
ReportPartitionOffset <ul style="list-style-type: none"> • Partition ID • Epoch • PartitionOffset 	Sent out (in one-way asynchronous mode) exclusively when <code>ContinuousQueuedDataReader</code> is requested for the next row to be read in the current epoch, and the epoch is done
SetReaderPartitions <ul style="list-style-type: none"> • Number of partitions 	Sent out (in request-response synchronous mode) exclusively when <code>DataSourceV2ScanExec</code> leaf physical operator is requested for the input RDDs (for a ContinuousReader and is about to create a ContinuousDataSourceRDD) The number of partitions is exactly the number of <code>InputPartitions</code> from the <code>ContinuousReader</code> .
SetWriterPartitions <ul style="list-style-type: none"> • Number of partitions 	Sent out (in request-response synchronous mode) exclusively when <code>WriteToContinuousDataSourceExec</code> leaf physical operator is requested to execute and generate a recipe for a distributed computation (as an <code>RDD[InternalRow]</code>) (and requests a ContinuousWriteRDD to collect that simply never finishes...and that's the <i>trick</i> of continuous mode)
StopContinuousExecutionWrites	Sent out (in request-response synchronous mode) exclusively when <code>ContinuousExecution</code> is requested to run a streaming query in continuous mode (and it finishes successfully or not)

Enable `ALL` logging level for `org.apache.spark.sql.execution.streaming.continuous.EpochCoordinatorRef` logger to see what happens inside.

Add the following line to `conf/log4j.properties`:

Tip

```
log4j.logger.org.apache.spark.sql.execution.streaming.continuous.EpochCoordinatorRef=ALL
```

Refer to [Logging](#).

Receiving Messages (Fire-And-Forget One-Way Mode)

— `receive` Method

```
receive: PartialFunction[Any, Unit]
```

Note

`receive` is part of the `RpcEndpoint` Contract in Apache Spark to receive messages in fire-and-forget one-way mode.

`receive` handles the following messages:

- [CommitPartitionEpoch](#)
- [ReportPartitionOffset](#)

With the `queryWritesStopped` turned on, `receive` simply *swallows* messages and does nothing.

Receiving Messages (Request-Response Two-Way Mode)

— `receiveAndReply` Method

```
receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit]
```

Note

`receiveAndReply` is part of the `RpcEndpoint` Contract in Apache Spark to receive and reply to messages in request-response two-way mode.

`receiveAndReply` handles the following messages:

- [GetCurrentEpoch](#)
- [IncrementAndGetEpoch](#)
- [SetReaderPartitions](#)

- [SetWriterPartitions](#)
- [StopContinuousExecutionWrites](#)

resolveCommitsAtEpoch Internal Method

```
resolveCommitsAtEpoch(epoch: Long): Unit
```

resolveCommitsAtEpoch ...FIXME

Note

`resolveCommitsAtEpoch` is used exclusively when `EpochCoordinator` is requested to handle [CommitPartitionEpoch](#) and [ReportPartitionOffset](#) messages.

commitEpoch Internal Method

```
commitEpoch(
  epoch: Long,
  messages: Iterable[WriterCommitMessage]): Unit
```

commitEpoch ...FIXME

Note

`commitEpoch` is used exclusively when `EpochCoordinator` is requested to [resolveCommitsAtEpoch](#).

Creating EpochCoordinator Instance

`EpochCoordinator` takes the following to be created:

- [StreamWriter](#)
- [ContinuousReader](#)
- [ContinuousExecution](#)
- Start epoch
- [SparkSession](#)
- [RpcEnv](#)

`EpochCoordinator` initializes the [internal properties](#).

Registering EpochCoordinator RPC Endpoint— `create` Factory Method

```
create(
    writer: StreamWriter,
    reader: ContinuousReader,
    query: ContinuousExecution,
    epochCoordinatorId: String,
    startEpoch: Long,
    session: SparkSession,
    env: SparkEnv): RpcEndpointRef
```

`create` simply [creates a new EpochCoordinator](#) and requests the `RpcEnv` to register a RPC endpoint as **EpochCoordinator-[id]** (where `id` is the given `epochCoordinatorId`).

`create` prints out the following INFO message to the logs:

```
Registered EpochCoordinator endpoint
```

Note	<code>create</code> is used exclusively when <code>ContinuousExecution</code> is requested to run a streaming query in continuous mode .
------	--

Internal Properties

Name	Description
queryWritesStopped	<p>Flag that indicates whether to drop messages (<code>true</code>) or not (<code>false</code>) when requested to handle one synchronously</p> <p>Default: <code>false</code></p> <p>Turned on (<code>true</code>) when requested to handle a synchronous StopContinuousExecutionWrites message</p>

EpochCoordinatorRef

`EpochCoordinatorRef` is...FIXME

Creating Remote Reference to EpochCoordinator RPC Endpoint— `create` Factory Method

```
create(
    writer: StreamWriter,
    reader: ContinuousReader,
    query: ContinuousExecution,
    epochCoordinatorId: String,
    startEpoch: Long,
    session: SparkSession,
    env: SparkEnv): RpcEndpointRef
```

`create` ...FIXME

Note

`create` is used exclusively when `ContinuousExecution` is requested to run a streaming query in continuous mode.

Getting Remote Reference to EpochCoordinator RPC Endpoint— `get` Factory Method

```
get(id: String, env: SparkEnv): RpcEndpointRef
```

`get` ...FIXME

Note

`get` is used when:

- `DataSourceV2ScanExec` leaf physical operator is requested for the input RDDs (and creates a `ContinuousDataSourceRDD` for a `ContinuousReader`)
- `ContinuousQueuedDataReader` is created (and initializes the `epochCoordEndpoint`)
- `EpochMarkerGenerator` is created (and initializes the `epochCoordEndpoint`)
- `ContinuousWriteRDD` is requested to compute a partition
- `WriteToContinuousDataSourceExec` is requested to execute and generate a recipe for a distributed computation (as an `RDD[InternalRow]`)

EpochTracker

EpochTracker is...FIXME

Current Epoch — `getCurrentEpoch` Method

```
getCurrentEpoch: Option[Long]
```

getCurrentEpoch ...FIXME

Note

`getCurrentEpoch` is used when...FIXME

Advancing (Incrementing) Epoch — `incrementCurrentEpoch` Method

```
incrementCurrentEpoch(): Unit
```

incrementCurrentEpoch ...FIXME

Note

`incrementCurrentEpoch` is used when...FIXME

ContinuousQueuedDataReader

`ContinuousQueuedDataReader` is created exclusively when `ContinuousDataSourceRDD` is requested to compute a partition.

`ContinuousQueuedDataReader` uses two types of continuous records:

- `EpochMarker`
- `ContinuousRow` (with the `InternalRow` at `PartitionOffset`)

Fetching Next Row — `next` Method

```
next(): InternalRow
```

`next` ...FIXME

Note	<code>next</code> is used when...FIXME
------	--

Closing ContinuousQueuedDataReader — `close` Method

```
close(): Unit
```

Note	<code>close</code> is part of the java.io.Closeable to close this stream and release any system resources associated with it.
------	---

`close` ...FIXME

Creating ContinuousQueuedDataReader Instance

`ContinuousQueuedDataReader` takes the following to be created:

- `ContinuousDataSourceRDDPartition`
- `TaskContext`
- Size of the [data queue](#)
- `epochPollIntervalMs`

`ContinuousQueuedDataReader` initializes the [internal properties](#).

Internal Properties

Name	Description
coordinatorId	Epoch Coordinator Identifier Used when...FIXME
currentOffset	PartitionOffset Used when...FIXME
dataReaderThread	DataReaderThread daemon thread that is created and started immediately when <code>ContinuousQueuedDataReader</code> is created Used when...FIXME
epochCoordEndpoint	<code>RpcEndpointRef</code> of the EpochCoordinator per <code>coordinatorId</code> Used when...FIXME
epochMarkerExecutor	java.util.concurrent.ScheduledExecutorService Used when...FIXME
epochMarkerGenerator	EpochMarkerGenerator Used when...FIXME
reader	InputPartitionReader Used when...FIXME
queue	java.util.concurrent.ArrayBlockingQueue of ContinuousRecords (of the given <code>data size</code>) Used when...FIXME

DataReaderThread

DataReaderThread is...FIXME

EpochMarkerGenerator Thread

EpochMarkerGenerator is...FIXME

run Method

run(): Unit

Note

run is part of the [java.lang.Runnable](#) Contract to be executed upon starting a thread.

run ...FIXME

PartitionOffset

PartitionOffset is...FIXME

ContinuousExecutionRelation Leaf Logical Operator

`ContinuousExecutionRelation` is a `MultiInstanceRelation` leaf logical operator.

Tip	Read up on Leaf Logical Operators in The Internals of Spark SQL book.
-----	---

`ContinuousExecutionRelation` is [created](#) (to represent `StreamingRelationV2` with `ContinuousReadSupport` data source) when `ContinuousExecution` is [created](#) (and requested for the [logical plan](#)).

`ContinuousExecutionRelation` takes the following to be created:

- [ContinuousReadSupport](#) source
- Options (`Map[String, String]`)
- Output attributes (`seq[Attribute]`)
- `SparkSession`

WriteToContinuousDataSource Unary Logical Operator

`WriteToContinuousDataSource` is a unary logical operator (`LogicalPlan`) that is created exclusively when `ContinuousExecution` is requested to run a streaming query in continuous mode (to create an `IncrementalExecution`).

`WriteToContinuousDataSource` is planned (*translated*) to a `WriteToContinuousDataSourceExec` unary physical operator (when `DataSourceV2Strategy` execution planning strategy is requested to plan a logical query).

Tip

Read up on [DataSourceV2Strategy Execution Planning Strategy](#) in [The Internals of Spark SQL book](#).

`WriteToContinuousDataSource` takes the following to be created:

- [StreamWriter](#)
- Child logical operator (`LogicalPlan`)

`WriteToContinuousDataSource` uses empty output schema (which is exactly to say that no output is expected whatsoever).

WriteToContinuousDataSourceExec Unary Physical Operator

`WriteToContinuousDataSourceExec` is a unary physical operator that [creates a ContinuousWriteRDD for continuous write](#).

Note

A unary physical operator (`UnaryExecNode`) is a physical operator with a single [child](#) physical operator.

Read up on [UnaryExecNode](#) (and physical operators in general) in [The Internals of Spark SQL](#) book.

`WriteToContinuousDataSourceExec` is [created](#) exclusively when `DataSourceV2Strategy` execution planning strategy is requested to plan a [WriteToContinuousDataSource](#) unary logical operator.

Tip

Read up on [DataSourceV2Strategy Execution Planning Strategy](#) in [The Internals of Spark SQL](#) book.

`WriteToContinuousDataSourceExec` takes the following to be created:

- [StreamWriter](#)
- Child physical operator (`SparkPlan`)

`WriteToContinuousDataSourceExec` uses empty output schema (which is exactly to say that no output is expected whatsoever).

Tip

Enable `ALL` logging level for

```
org.apache.spark.sql.execution.streaming.continuous.WriteToContinuousDataSourceExec
```

happens inside.

Add the following line to `conf/log4j.properties` :

```
log4j.logger.org.apache.spark.sql.execution.streaming.continuous.WriteToContinuousDa
```

Refer to [Logging](#).

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note `doExecute` is part of `SparkPlan` Contract to generate the runtime representation of an physical operator as a distributed computation over internal binary rows on Apache Spark (i.e. `RDD[InternalRow]`).

`doExecute` requests the [StreamWriter](#) to create a `DataWriterFactory`.

`doExecute` then requests the [child physical operator](#) to execute (that gives a `RDD[InternalRow]`) and uses the `RDD[InternalRow]` and the `DataWriterFactory` to create a [ContinuousWriteRDD](#).

`doExecute` prints out the following INFO message to the logs:

```
Start processing data source writer: [writer]. The input RDD has [partitions] partitions.
```

`doExecute` requests the `EpochCoordinatorRef` helper for a [remote reference to the EpochCoordinator RPC endpoint](#) (using the `_epoch_coordinator_id` local property).

Note The [EpochCoordinator RPC endpoint](#) runs on the driver as the single point to coordinate epochs across partition tasks.

`doExecute` requests the EpochCoordinator RPC endpoint reference to send out a [SetWriterPartitions](#) message synchronously.

In the end, `doExecute` requests the `ContinuousWriteRDD` to collect (which simply runs a Spark job on all partitions in an RDD and returns the results in an array).

Note Requesting the `ContinuousWriteRDD` to collect is how a Spark job is ran that in turn runs tasks (one per partition) that are described by the `ContinuousWriteRDD.compute` method. Since executing `collect` is meant to run a Spark job (with tasks on executors), it's in the discretion of the tasks themselves to decide when to finish (so if they want to run indefinitely, so be it). *What a clever trick!*

ContinuousWriteRDD — RDD of WriteToContinuousDataSourceExec Unary Physical Operator

`ContinuousWriteRDD` is a specialized `RDD` (`RDD[Unit]`) that is used exclusively as the underlying `RDD` of `writeToContinuousDataSourceExec` unary physical operator to [write records continuously](#).

`ContinuousWriteRDD` is [created](#) exclusively when `writeToContinuousDataSourceExec` unary physical operator is requested to [execute](#) and generate a recipe for a distributed computation (as an `RDD[InternalRow]`).

`ContinuousWriteRDD` uses the [parent RDD](#) for the partitions and the partitioner.

`ContinuousWriteRDD` takes the following to be created:

- Parent `RDD` (`RDD[InternalRow]`)
- Write task (`DataWriterFactory[InternalRow]`)

Computing Partition — `compute` Method

```
compute(  
    split: Partition,  
    context: TaskContext): Iterator[Unit]
```

Note	<code>compute</code> is part of the <code>RDD</code> Contract to compute a partition.
------	---

`compute` requests the `EpochCoordinatorRef` helper for a [remote reference to the EpochCoordinator RPC endpoint](#) (using the `__epoch_coordinator_id` local property).

Note	The EpochCoordinator RPC endpoint runs on the driver as the single point to coordinate epochs across partition tasks.
------	---

`compute` uses the `EpochTracker` helper to [initializeCurrentEpoch](#) (using the `__continuous_start_epoch` local property).

`compute` then executes the following steps (in a loop) until the task (as the given `TaskContext`) is killed or completed.

`compute` requests the [parent RDD](#) to compute the given partition (that gives an `Iterator[InternalRow]`).

`compute` requests the [DataWriterFactory](#) to create a `DataWriter` (for the partition and the task attempt IDs from the given `TaskContext` and the [current epoch](#) from the `EpochTracker` helper) and requests it to write all records (from the `Iterator[InternalRow]`).

`compute` prints out the following INFO message to the logs:

```
Writer for partition [partitionId] in epoch [epoch] is committing.
```

`compute` requests the `DataWriter` to commit (that gives a `WriterCommitMessage`).

`compute` requests the EpochCoordinator RPC endpoint reference to send out a [CommitPartitionEpoch](#) message (with the `WriterCommitMessage`).

`compute` prints out the following INFO message to the logs:

```
Writer for partition [partitionId] in epoch [epoch] is committed.
```

In the end (of the loop), `compute` uses the `EpochTracker` helper to [incrementCurrentEpoch](#).

In case of an error, `compute` prints out the following ERROR message to the logs and requests the `DataWriter` to abort.

```
Writer for partition [partitionId] is aborting.
```

In the end, `compute` prints out the following ERROR message to the logs:

```
Writer for partition [partitionId] aborted.
```

ContinuousDataSourceRDD — Input RDD of DataSourceV2ScanExec Physical Operator with ContinuousReader

`ContinuousDataSourceRDD` is a specialized `RDD` (`RDD[InternalRow]`) that is used exclusively for the only input RDD (with the input rows) of `DataSourceV2ScanExec` leaf physical operator with a [ContinuousReader](#).

`ContinuousDataSourceRDD` is [created](#) exclusively when `DataSourceV2ScanExec` leaf physical operator is requested for the input RDDs (which there is only one actually).

`ContinuousDataSourceRDD` uses [spark.sql.streaming.continuous.executorQueueSize](#) configuration property for the size of the data queue.

`ContinuousDataSourceRDD` uses [spark.sql.streaming.continuous.executorPollIntervalMs](#) configuration property for the epochPollIntervalMs.

`ContinuousDataSourceRDD` takes the following to be created:

- `SparkContext`
- Size of the data queue
- `epochPollIntervalMs`
- `InputPartition[InternalRow] S`

`ContinuousDataSourceRDD` uses `InputPartition` (of a `ContinuousDataSourceRDDPartition`) for preferred host locations (where the input partition reader can run faster).

Computing Partition — `compute` Method

```
compute(
  split: Partition,
  context: TaskContext): Iterator[InternalRow]
```

Note

`compute` is part of the RDD Contract to compute a given partition.

`compute` ...FIXME

getPartitions Method

```
getPartitions: Array[Partition]
```

Note	getPartitions is part of the <code>RDD</code> Contract to specify the partitions to compute .
-------------	---

```
getPartitions ...FIXME
```

StreamExecution — Base of Stream Execution Engines

`StreamExecution` is the [base of stream execution engines](#) (aka *streaming query processing engines*) that can [run a structured query](#) (on a [stream execution thread](#)).

Note	Continuous query, streaming query, continuous Dataset, streaming Dataset are all considered high-level synonyms for an executable entity that stream execution engines run using the analyzed logical plan internally.
------	---

Table 1. StreamExecution Contract (Abstract Methods Only)

Property	Description
<code>logicalPlan</code>	<pre>logicalPlan: LogicalPlan</pre> <p>Analyzed logical plan of the streaming query to execute Used when <code>StreamExecution</code> is requested to run stream processing</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <p>Note <code>logicalPlan</code> is part of ProgressReporter Contract and the only purpose of the <code>logicalPlan</code> property is to change the access level from <code>protected</code> to <code>public</code>.</p> </div>
<code>runActivatedStream</code>	<pre>runActivatedStream(sparkSessionForStream: SparkSession): Unit</pre> <p>Executes (<i>runs</i>) the activated streaming query Used exclusively when <code>StreamExecution</code> is requested to run the streaming query (when transitioning from <code>INITIALIZING</code> to <code>ACTIVE</code> state)</p>

Streaming Query and Stream Execution Engine

```
scala> :type query
org.apache.spark.sql.streaming.StreamingQuery

import org.apache.spark.sql.execution.streaming.StreamingQueryWrapper
val se = query.asInstanceOf[StreamingQueryWrapper].streamingQuery

scala> :type se
org.apache.spark.sql.execution.streaming.StreamExecution
```

`StreamExecution` uses the `spark.sql.streaming.minBatchesToRetain` configuration property to allow the `StreamExecutions` to discard old log entries (from the `offset` and `commit` logs).

Table 2. StreamExecutions

StreamExecution	Description
ContinuousExecution	Used in Continuous Stream Processing
MicroBatchExecution	Used in Micro-Batch Stream Processing

Note	<code>StreamExecution</code> does not support adaptive query execution and cost-based optimizer (and turns them off when requested to run stream processing).
------	--

`StreamExecution` is the **execution environment** of a single streaming query (aka *streaming Dataset*) that is executed every `trigger` and in the end adds the results to a `sink`.

Note	<code>StreamExecution</code> corresponds to a single streaming query with one or more streaming sources and exactly one streaming sink .
------	--

```

import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._

val q = spark.
  readStream.
  format("rate").
  load.
  writeStream.
  format("console").
  trigger(Trigger.ProcessingTime(10.minutes)).
  start

scala> :type q
org.apache.spark.sql.streaming.StreamingQuery

// Pull out StreamExecution off StreamingQueryWrapper
import org.apache.spark.sql.execution.streaming.{StreamExecution, StreamingQueryWrapper}
}
val se = q.asInstanceOf[StreamingQueryWrapper].streamingQuery
scala> :type se
org.apache.spark.sql.execution.streaming.StreamExecution

```

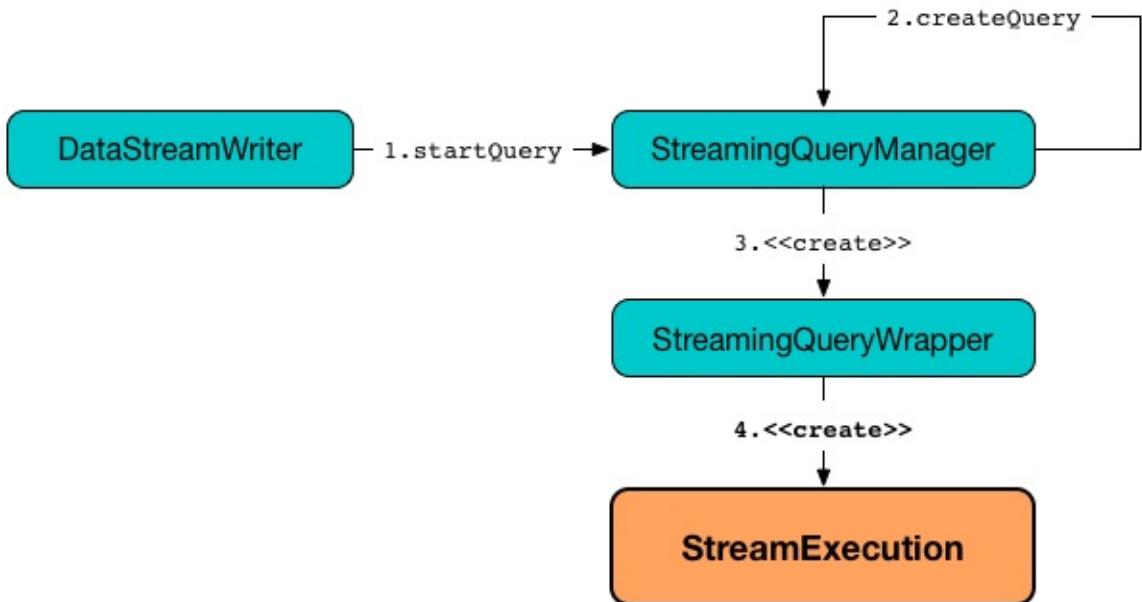


Figure 1. Creating Instance of StreamExecution

Note

`DataStreamWriter` describes how the results of executing batches of a streaming query are written to a streaming sink.

When `started`, `StreamExecution` starts a `stream execution thread` that simply `runs stream processing` (and hence the streaming query).

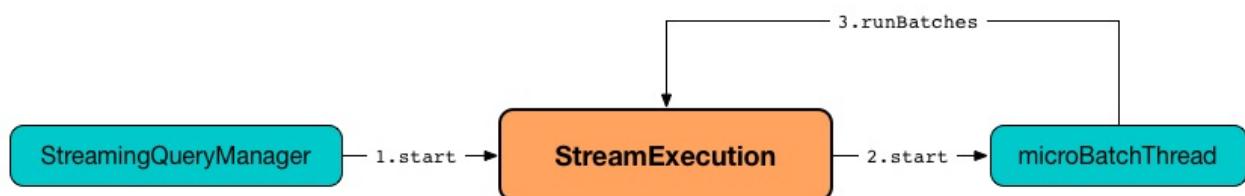


Figure 2. StreamExecution's Starting Streaming Query (on Execution Thread)

`StreamExecution` is a `ProgressReporter` and `reports status of the streaming query` (i.e. when it starts, progresses and terminates) by posting `StreamingQueryListener` events.

```

import org.apache.spark.sql.streaming.Trigger
import scala.concurrent.duration._

val sq = spark
  .readStream
  .text("server-logs")
  .writeStream
  .format("console")
  .queryName("debug")
  .trigger(Trigger.ProcessingTime(20.seconds))
  .start

// Enable the log level to see the INFO and DEBUG messages
// log4j.logger.org.apache.spark.sql.execution.streaming.StreamExecution=DEBUG

17/06/18 21:21:07 INFO StreamExecution: Starting new streaming query.
17/06/18 21:21:07 DEBUG StreamExecution: getOffset took 5 ms
17/06/18 21:21:07 DEBUG StreamExecution: Stream running from {} to {}
17/06/18 21:21:07 DEBUG StreamExecution: triggerExecution took 9 ms
17/06/18 21:21:07 DEBUG StreamExecution: Execution stats: ExecutionStats(Map(),List(),
Map())
17/06/18 21:21:07 INFO StreamExecution: Streaming query made progress: {
  "id" : "8b57b0bd-fc4a-42eb-81a3-777d7ba5e370",
  "runId" : "920b227e-6d02-4a03-a271-c62120258cea",
  "name" : "debug",
  "timestamp" : "2017-06-18T19:21:07.693Z",
  "numInputRows" : 0,
  "processedRowsPerSecond" : 0.0,
  "durationMs" : {
    "getOffset" : 5,
    "triggerExecution" : 9
  },
  "stateOperators" : [ ],
  "sources" : [ {
    "description" : "FileStreamSource[file:/Users/jacek/dev/oss/spark/server-logs]",
    "startOffset" : null,
    "endOffset" : null,
    "numInputRows" : 0,
    "processedRowsPerSecond" : 0.0
  }],
  "sink" : {
    "description" : "org.apache.spark.sql.execution.streaming.ConsoleSink@2460208a"
  }
}
17/06/18 21:21:10 DEBUG StreamExecution: Starting Trigger Calculation
17/06/18 21:21:10 DEBUG StreamExecution: getOffset took 3 ms
17/06/18 21:21:10 DEBUG StreamExecution: triggerExecution took 3 ms
17/06/18 21:21:10 DEBUG StreamExecution: Execution stats: ExecutionStats(Map(),List(),
Map())

```

`StreamExecution` tracks streaming data sources in `uniqueSources` internal registry.

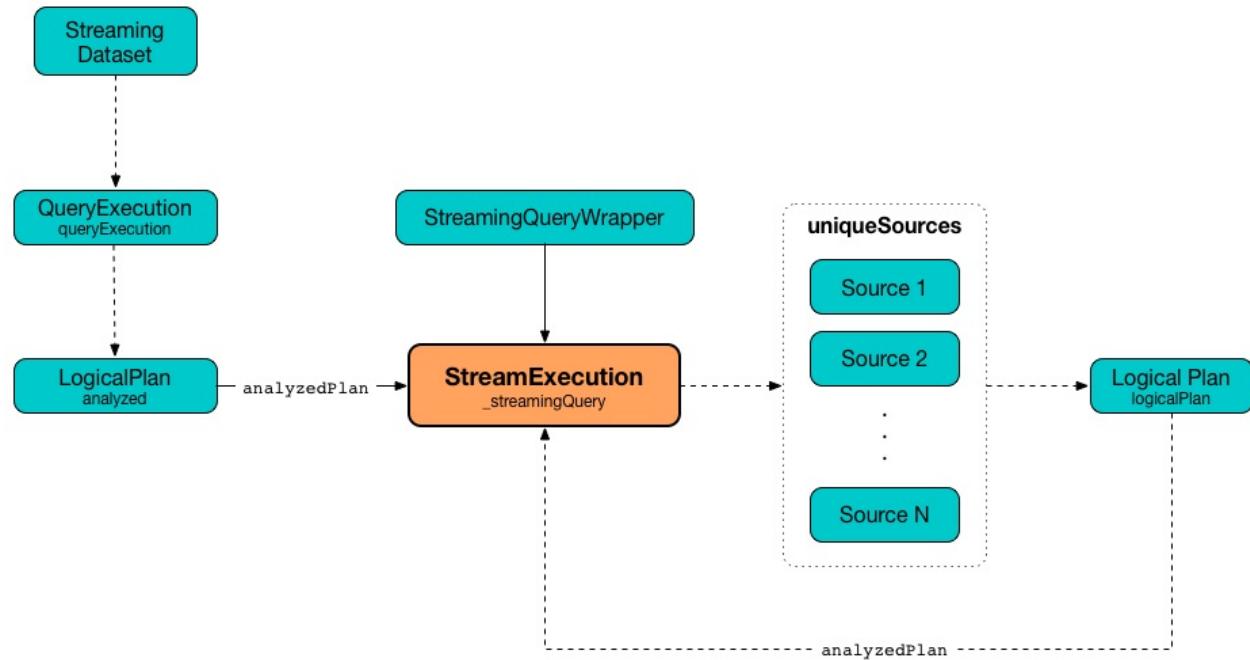


Figure 3. StreamExecution’s uniqueSources Registry of Streaming Data Sources

`StreamExecution` collects `durationMs` for the execution units of streaming batches.

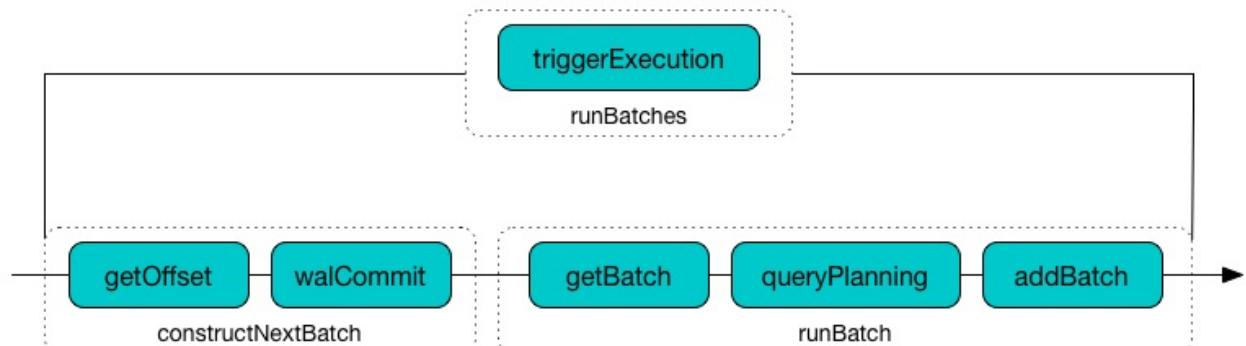


Figure 4. StreamExecution’s durationMs

```

scala> :type q
org.apache.spark.sql.streaming.StreamingQuery

scala> println(q.lastProgress)
{
  "id" : "03fc78fc-fe19-408c-a1ae-812d0e28fcee",
  "runId" : "8c247071-afba-40e5-aad2-0e6f45f22488",
  "name" : null,
  "timestamp" : "2017-08-14T20:30:00.004Z",
  "batchId" : 1,
  "numInputRows" : 432,
  "inputRowsPerSecond" : 0.9993568953312452,
  "processedRowsPerSecond" : 1380.1916932907347,
  "durationMs" : {
    "addBatch" : 237,
    "getBatch" : 26,
    "getOffset" : 0,
    "queryPlanning" : 1,
    "triggerExecution" : 313,
    "walCommit" : 45
  },
  "stateOperators" : [ ],
  "sources" : [ {
    "description" : "RateSource[rowsPerSecond=1, rampUpTimeSeconds=0, numPartitions=8]"
  },
  {
    "startOffset" : 0,
    "endOffset" : 432,
    "numInputRows" : 432,
    "inputRowsPerSecond" : 0.9993568953312452,
    "processedRowsPerSecond" : 1380.1916932907347
  } ],
  "sink" : {
    "description" : "ConsoleSink[numRows=20, truncate=true]"
  }
}

```

`StreamExecution` uses [OffsetSeqLog](#) and [BatchCommitLog](#) metadata logs for **write-ahead log** (to record offsets to be processed) and that have already been processed and committed to a streaming sink, respectively.

Tip

Monitor `offsets` and `commits` metadata logs to know the progress of a streaming query.

`StreamExecution` delays polling for new data for 10 milliseconds (when no data was available to process in a batch). Use `spark.sql.streaming.pollingDelay` Spark property to control the delay.

Every `StreamExecution` is uniquely identified by an **ID of the streaming query** (which is the `id` of the `StreamMetadata`).

Note

Since the `StreamMetadata` is persisted (to the `metadata` file in the `checkpoint directory`), the streaming query ID "survives" query restarts as long as the checkpoint directory is preserved.

`StreamExecution` is also uniquely identified by a **run ID of the streaming query**. A run ID is a randomly-generated 128-bit universally unique identifier (UUID) that is assigned at the time `StreamExecution` is created.

Note

`runId` does not "survive" query restarts and will always be different yet unique (across all active queries).

Note

The `name`, `id` and `runId` are all unique across all active queries (in a `StreamingQueryManager`). The difference is that:

- `name` is optional and user-defined
- `id` is a UUID that is auto-generated at the time `StreamExecution` is created and persisted to `metadata` checkpoint file
- `runId` is a UUID that is auto-generated every time `StreamExecution` is created

`StreamExecution` uses a `StreamMetadata` that is **persisted** in the `metadata` file in the `checkpoint directory`. If the `metadata` file is available it is **read** and is the way to recover the **ID** of a streaming query when resumed (i.e. restarted after a failure or a planned stop).

`StreamExecution` uses **`__is_continuous_processing`** local property (default: `false`) to differentiate between `ContinuousExecution` (`true`) and `MicroBatchExecution` (`false`) which is used when `statestoreRDD` is requested to **compute a partition** (and **finds a StateStore** for a given version).

Tip

Enable `ALL` logging level for `org.apache.spark.sql.execution.streaming.StreamExecution` to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.execution.streaming.StreamExecution=ALL
```

Refer to [Logging](#).

Creating StreamExecution Instance

`StreamExecution` takes the following to be created:

- `SparkSession`
- Name of the streaming query (can also be `null`)
- Path of the checkpoint directory (aka *metadata directory*)
- Streaming query (as an analyzed logical query plan, i.e. `LogicalPlan`)
- **Streaming sink**
- **Trigger**
- `Clock`
- **Output mode**
- `deleteCheckpointOnStop` flag (to control whether to delete the checkpoint directory on stop)

`StreamExecution` initializes the [internal properties](#).

Note	<code>StreamExecution</code> is a Scala abstract class and cannot be created directly. It is created indirectly when the concrete StreamExecutions are.
------	---

Write-Ahead Log (WAL) of Offsets — `offsetLog` Property

<pre>offsetLog: OffsetSeqLog</pre>

`offsetLog` is a [Hadoop DFS-based metadata storage](#) (of `OffsetSeqs`) with [offsets metadata directory](#).

`offsetLog` is used as **Write-Ahead Log of Offsets** to [persist offsets](#) of the data about to be processed in every trigger.

Note	Metadata log or metadata checkpoint are synonyms and are often used interchangeably.
------	--

The number of entries in the `offsetSeqLog` is controlled using `spark.sql.streaming.minBatchesToRetain` configuration property (default: `100`). [Stream execution engines](#) discard ([purge](#)) offsets from the `offsets` metadata log when the [current batch ID](#) (in [MicroBatchExecution](#)) or the [epoch committed](#) (in [ContinuousExecution](#)) is above the threshold.

Note	<p><code>offsetLog</code> is used when:</p> <ul style="list-style-type: none"> • <code>ContinuousExecution</code> stream execution engine is requested to commit an epoch, getStartOffsets, and addOffset • <code>MicroBatchExecution</code> stream execution engine is requested to populate start offsets and construct (or skip) the next streaming micro-batch
------	--

State of Streaming Query Execution — `state` Property

```
state: AtomicReference[State]
```

`state` indicates the internal state of execution of the streaming query (as [java.util.concurrent.atomic.AtomicReference](#)).

Table 3. States

Name	Description
ACTIVE	<code>StreamExecution</code> has been requested to run stream processing (and is about to run the activated streaming query)
INITIALIZING	<code>StreamExecution</code> has been created
TERMINATED	Used to indicate that: <ul style="list-style-type: none"> • <code>MicroBatchExecution</code> has been requested to stop • <code>ContinuousExecution</code> has been requested to stop • <code>StreamExecution</code> has been requested to run stream processing (and has finished running the activated streaming query)
RECONFIGURING	Used only when <code>ContinuousExecution</code> is requested to run a streaming query in continuous mode (and the <code>ContinuousReader</code> indicated that needs reconfiguration)

Available Offsets (StreamProgress) — `availableOffsets` Property

```
availableOffsets: StreamProgress
```

`availableOffsets` is a collection of offsets per streaming source to track what data (by offset) is available for processing for every streaming source in the streaming query (and have not yet been committed).

`availableOffsets` works in tandem with the `committedOffsets` internal registry.

`availableOffsets` is empty when `StreamExecution` is created (i.e. no offsets are reported for any streaming source in the streaming query).

Note	<p><code>availableOffsets</code> is used when:</p> <ul style="list-style-type: none"> • <code>MicroBatchExecution</code> stream execution engine is requested to resume and fetch the start offsets from checkpoint, check whether new data is available, construct the next streaming micro-batch and run a single streaming micro-batch • <code>ContinuousExecution</code> stream execution engine is requested to commit an epoch • <code>StreamExecution</code> is requested for the internal string representation
------	--

Committed Offsets (StreamProgress) — `committedOffsets` Property

`committedOffsets: StreamProgress`

`committedOffsets` is a collection of offsets per streaming source to track what data (by offset) has already been processed and committed (to the sink or state stores) for every streaming source in the streaming query.

`committedOffsets` works in tandem with the `availableOffsets` internal registry.

Note	<p><code>committedOffsets</code> is used when:</p> <ul style="list-style-type: none"> • <code>MicroBatchExecution</code> stream execution engine is requested for the start offsets (from checkpoint), to check whether new data is available and run a single streaming micro-batch • <code>ContinuousExecution</code> stream execution engine is requested for the start offsets (from checkpoint) and to commit an epoch • <code>StreamExecution</code> is requested for the internal string representation
------	---

Fully-Qualified (Resolved) Path to Checkpoint Root Directory — `resolvedCheckpointRoot` Property

```
resolvedCheckpointRoot: String
```

`resolvedCheckpointRoot` is a fully-qualified path of the given [checkpoint root directory](#).

The given [checkpoint root directory](#) is defined using **checkpointLocation** option or the `spark.sql.streaming.checkpointLocation` configuration property with `queryName` option.

`checkpointLocation` and `queryName` options are defined when `StreamingQueryManager` is requested to [create a streaming query](#).

`resolvedCheckpointRoot` is used when [creating the path to the checkpoint directory](#) and when `StreamExecution` finishes [running streaming batches](#).

`resolvedCheckpointRoot` is used for the [logicalPlan](#) (while transforming [analyzedPlan](#) and planning `StreamingRelation` logical operators to corresponding `StreamingExecutionRelation` physical operators with the streaming data sources created passing in the path to `sources` directory to store checkpointing metadata).

	You can see <code>resolvedCheckpointRoot</code> in the INFO message when <code>StreamExecution</code> started.
--	--

Tip

Starting [prettyIdString]. Use [resolvedCheckpointRoot] to store the query check

Internally, `resolvedCheckpointRoot` creates a Hadoop `org.apache.hadoop.fs.Path` for [checkpointRoot](#) and makes it qualified.

Note	<code>resolvedCheckpointRoot</code> uses <code>SparkSession</code> to access <code>SessionState</code> for a Hadoop configuration.
------	--

`resolvedCheckpointRoot` uses `SparkSession` to access `SessionState` for a Hadoop configuration.

Offset Commit Log — `commits` Metadata Checkpoint Directory

`StreamExecution` uses **offset commit log** ([CommitLog](#) with `commits` [metadata checkpoint directory](#)) for streaming batches successfully executed (with a single file per batch with a file name being the batch id) or committed epochs.

Note	Metadata log or metadata checkpoint are synonyms and are often used interchangeably.
------	--

`Metadata log` or `metadata checkpoint` are synonyms and are often used interchangeably.

`commitLog` is used by the [stream execution engines](#) for the following:

- `MicroBatchExecution` is requested to [run an activated streaming query](#) (that in turn requests to [populate the start offsets](#) at the very beginning of the streaming query execution and later regularly every [single batch](#))
- `ContinuousExecution` is requested to [run an activated streaming query in continuous mode](#) (that in turn requests to [retrieve the start offsets](#) at the very beginning of the streaming query execution and later regularly every [commit](#))

Stopping Streaming Sources and Readers

— `stopSources` Method

```
stopSources(): Unit
```

`stopSources` requests every [streaming source](#) (in the [streaming query](#)) to [stop](#).

In case of an non-fatal exception, `stopSources` prints out the following WARN message to the logs:

```
Failed to stop streaming source: [source]. Resources may have leaked.
```

Note

`stopSources` is used when:

- `StreamExecution` is requested to [run stream processing](#) (and [terminates successfully or not](#))
- `ContinuousExecution` is requested to [run the streaming query in continuous mode](#) (and terminates)

Running Stream Processing — `runStream` Internal Method

```
runStream(): Unit
```

`runStream` simply prepares the environment to [execute the activated streaming query](#).

Note

`runStream` is used exclusively when the [stream execution thread](#) is requested to start (when `DataStreamWriter` is requested to [start an execution of the streaming query](#)).

Internally, `runStream` sets the job group (to all the Spark jobs started by this thread) as follows:

- `runId` for the job group ID
- `getBatchDescriptionString` for the job group description (to display in web UI)
- `interruptOnCancel` flag on

Note	<p><code>runStream</code> uses the SparkSession to access <code>SparkContext</code> and assign the job group id.</p> <p>Read up on SparkContext.setJobGroup method in The Internals of Apache Spark book.</p>
------	---

`runStream` sets `sql.streaming.queryId` local property to `id`.

`runStream` requests the `MetricsSystem` to register the `MetricsReporter` when `spark.sql.streaming.metricsEnabled` configuration property is on (default: off / `false`).

`runStream` notifies `StreamingQueryListeners` that the streaming query has been started (by posting a new `QueryStartedEvent` event with `id`, `runId`, and `name`).

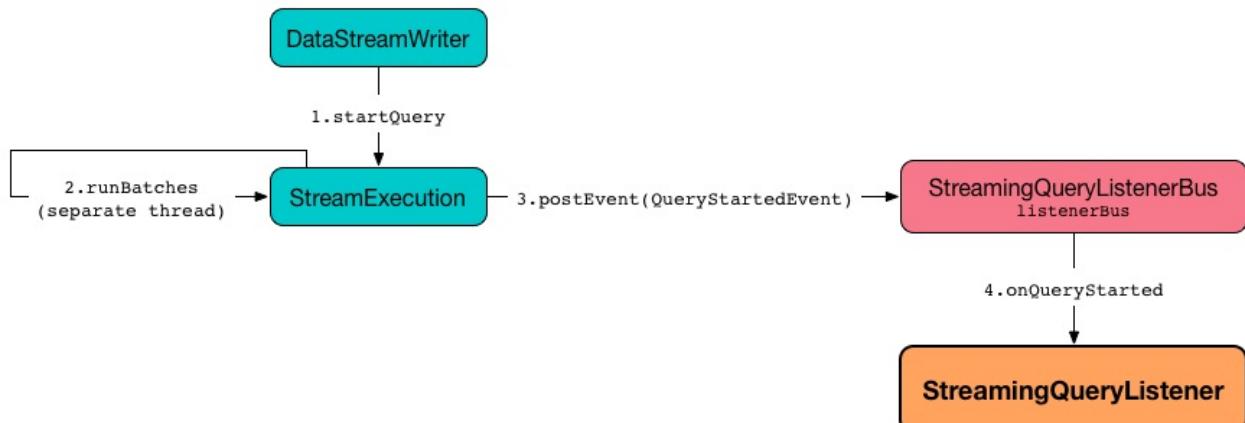


Figure 5. `StreamingQueryListener` Notified about Query's Start (`onQueryStarted`)

`runStream` unblocks the [main starting thread](#) (by decrementing the count of the `startLatch` that when `0` lets the starting thread continue).

Caution	FIXME A picture with two parallel lanes for the starting thread and daemon one for the query.
---------	---

`runStream` updates the status message to be **Initializing sources**.

`runStream` initializes the [analyzed logical plan](#).

Note	The analyzed logical plan is a lazy value in Scala and is initialized when requested the very first time.
------	---

`runStream` disables **adaptive query execution** and **cost-based join optimization** (by turning `spark.sql.adaptive.enabled` and `spark.sql.cbo.enabled` configuration properties off, respectively).

`runStream` creates a new "zero" `OffsetSeqMetadata`.

(Only when in `INITIALIZING` state) `runStream` enters `ACTIVE` state:

- Decrements the count of `initializationLatch`
- Executes the activated streaming query (which is different per `StreamExecution`, i.e. `ContinuousExecution` or `MicroBatchExecution`).

Note	<code>runBatches</code> does the main work only when first started (i.e. when <code>state</code> is <code>INITIALIZING</code>).
------	--

`runStream` ...FIXME (describe the failed and stop states)

Once `TriggerExecutor` has finished executing batches, `runBatches` updates the status message to `Stopped`.

Note	<code>TriggerExecutor</code> finishes executing batches when <code>batch runner</code> returns whether the streaming query is stopped or not (which is when the internal <code>state</code> is not <code>TERMINATED</code>).
------	---

Caution	FIXME Describe <code>catch</code> block for exception handling
---------	--

Running Stream Processing — `finally` Block

`runStream` releases the `startLatch` and `initializationLatch` locks.

`runStream` `stopSources`.

`runStream` sets the `state` to `TERMINATED`.

`runStream` sets the `StreamingQueryStatus` with the `isTriggerActive` and `isDataAvailable` flags off (`false`).

`runStream` removes the `stream metrics reporter` from the application's `Metricssystem` .

`runStream` requests the `StreamingQueryManager` to handle termination of a streaming query.

`runStream` creates a new `QueryTerminatedEvent` (with the `id` and `run id` of the streaming query) and `posts it`.

With the `deleteCheckpointOnStop` flag enabled and no `StreamingQueryException` reported, `runStream` deletes the `checkpoint directory` recursively.

In the end, `runStream` releases the `terminationLatch` lock.

TriggerExecutor's Batch Runner

Batch Runner (aka `batchRunner`) is an executable block executed by `TriggerExecutor` in `runBatches`.

`batchRunner` starts trigger calculation.

As long as the query is not stopped (i.e. `state` is not `TERMINATED`), `batchRunner` executes the streaming batch for the trigger.

In `triggerExecution` time-tracking section, `runBatches` branches off per `currentBatchId`.

Table 4. Current Batch Execution per currentBatchId

<code>currentBatchId < 0</code>	<code>currentBatchId >= 0</code>
<ol style="list-style-type: none"> 1. <code>populateStartOffsets</code> 2. Setting Job Description as <code>getBatchDescriptionString</code> <pre>DEBUG Stream running from [committedOffsets] to [availableOffsets]</pre>	<ol style="list-style-type: none"> 1. Constructing the next streaming micro-batch

If there is `data available` in the sources, `batchRunner` marks `currentStatus` with `isDataAvailable` enabled.

Note	<p>You can check out the status of a streaming query using <code>status</code> method.</p> <pre>scala> spark.streams.active(0).status res1: org.apache.spark.sql.streaming.StreamingQueryStatus = { "message" : "Waiting for next trigger", "isDataAvailable" : false, "isTriggerActive" : false }</pre>
------	---

`batchRunner` then updates the status message to **Processing new data** and runs the current streaming batch.

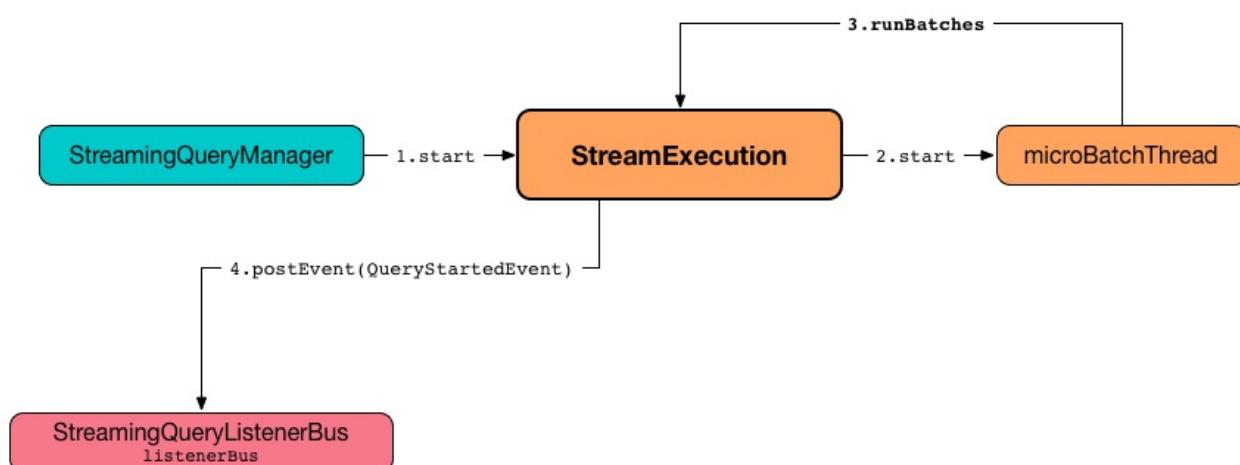


Figure 6. StreamExecution's Running Batches (on Execution Thread)

After `triggerExecution` section has finished, `batchRunner` finishes the streaming batch for the trigger (and collects query execution statistics).

When there was `data available` in the sources, `batchRunner` updates committed offsets (by adding the current batch id to `BatchCommitLog` and adding `availableOffsets` to `committedOffsets`).

You should see the following DEBUG message in the logs:

```
DEBUG batch $currentBatchId committed
```

`batchRunner` increments the current batch id and sets the job description for all the following Spark jobs to include the new batch id.

When no `data was available` in the sources to process, `batchRunner` does the following:

1. Marks `currentStatus` with `isDataAvailable` disabled
2. Updates the status message to **Waiting for data to arrive**
3. Sleeps the current thread for `pollingDelayMs` milliseconds.

`batchRunner` updates the status message to **Waiting for next trigger** and returns whether the query is currently active or not (so `TriggerExecutor` can decide whether to finish executing the batches or not)

Human-Readable Description of Spark Jobs — `getBatchDescriptionString` Method

```
getBatchDescriptionString: String
```

```
getBatchDescriptionString ...FIXME
```

Note

- | | |
|--|---|
| | <p><code>getBatchDescriptionString</code> is used when:</p> <ul style="list-style-type: none"> • <code>MicroBatchExecution</code> is requested to <code>runActivatedStream</code> (and sets the job description) • <code>StreamExecution</code> is requested to <code>runStream</code> (and sets job group) |
|--|---|

Starting Streaming Query (on Stream Execution Thread) — `start` Method

```
start(): Unit
```

When called, `start` prints out the following INFO message to the logs:

```
Starting [prettyIdString]. Use [resolvedCheckpointRoot] to store the query checkpoint.
```

`start` then starts the [stream execution thread](#) (as a daemon thread).

Note	<code>start</code> uses Java's java.lang.Thread.start to run the streaming query on a separate execution thread.
------	--

Note	When started, a streaming query runs in its own execution thread on JVM.
------	--

In the end, `start` pauses the main thread (using the `startLatch` until `StreamExecution` is requested to [run the streaming query](#) that in turn sends a [QueryStartedEvent](#) to all streaming listeners followed by decrementing the count of the `startLatch`).

Note	<code>start</code> is used exclusively when <code>StreamingQueryManager</code> is requested to start a streaming query (when <code>DataStreamWriter</code> is requested to start an execution of the streaming query).
------	---

Path to Checkpoint Directory — `checkpointFile` Internal Method

```
checkpointFile(name: String): String
```

`checkpointFile` gives the path of a directory with `name` in [checkpoint](#) directory.

Note	<code>checkpointFile</code> uses Hadoop's <code>org.apache.hadoop.fs.Path</code> .
------	--

Note	<code>checkpointFile</code> is used for streamMetadata , OffsetSeqLog , BatchCommitLog , and lastExecution (for runBatch).
------	---

Posting StreamingQueryListener Event — `postEvent` Method

```
postEvent(event: StreamingQueryListener.Event): Unit
```

Note	<code>postEvent</code> is a part of ProgressReporter Contract .
------	---

`postEvent` simply requests the `StreamingQueryManager` to [post](#) the input event (to the `StreamingQueryListenerBus` in the current `SparkSession`).

Note	<code>postEvent</code> uses <code>SparkSession</code> to access the current <code>StreamingQueryManager</code> .
------	--

Note	<code>postEvent</code> is used when: <ul style="list-style-type: none"> • <code>ProgressReporter</code> reports update progress (while finishing a trigger) • <code>StreamExecution</code> runs streaming batches (and announces starting a streaming query by posting a <code>QueryStartedEvent</code> and query termination by posting a <code>QueryTerminatedEvent</code>)
------	---

Waiting Until No Data Available in Sources or Query Has Been Terminated — `processAllAvailable` Method

```
processAllAvailable(): Unit
```

Note	<code>processAllAvailable</code> is a part of StreamingQuery Contract .
------	---

`processAllAvailable` [reports `streamDeathCause`](#) exception if defined (and returns).

Note	<code>streamDeathCause</code> is defined exclusively when <code>StreamExecution</code> runs streaming batches (and terminated with an exception).
------	---

`processAllAvailable` returns when [isActive](#) flag is turned off (which is when `StreamExecution` is in `TERMINATED` state).

`processAllAvailable` acquires a lock on `awaitProgressLock` and turns `noNewData` flag off.

`processAllAvailable` keeps waiting 10 seconds for `awaitProgressLockCondition` until `noNewData` flag is turned on or `StreamExecution` is no longer [active](#).

Note	<code>noNewData</code> flag is turned on exclusively when <code>StreamExecution</code> constructs the next streaming micro-batch (and finds that no data is available).
------	--

In the end, `processAllAvailable` releases `awaitProgressLock` lock.

Stream Execution Thread — `queryExecutionThread` Property

```
queryExecutionThread: QueryExecutionThread
```

`queryExecutionThread` is a Java thread of execution (`java.util.Thread`) that runs the structured query when started.

`queryExecutionThread` uses the name **stream execution thread for [id]** (that uses `prettyIdString` for the id, i.e. `queryName [id = [id], runId = [runId]]`).

`queryExecutionThread` is a `queryExecutionThread` that is really a custom `UninterruptibleThread` from Apache Spark with `runUninterruptibly` method for running a block of code without being interrupted by `Thread.interrupt()`.

`queryExecutionThread` is started (as a daemon thread) when `StreamExecution` is requested to [start](#).

When started, `queryExecutionThread` sets the thread-local properties as the [call site](#) and [runs the streaming query](#).

	Use Java's jconsole or jstack to monitor the streaming threads.
--	---

Tip

```
$ jstack <driver-pid> | grep -e "stream execution thread"
"stream execution thread for kafka-topic1 [id =...]
```

Internal String Representation — `toDebugString` Internal Method

	<code>toDebugString(includeLogicalPlan: Boolean): String</code>
--	---

`toDebugString` ...FIXME

Note

	<code>toDebugString</code> is used exclusively when <code>StreamExecution</code> is requested to run stream processing (and an exception is caught).
--	--

Current Batch Metadata (Event-Time Watermark and Timestamp) — `offsetSeqMetadata` Internal Property

	<code>offsetSeqMetadata: OffsetSeqMetadata</code>
--	---

`offsetSeqMetadata` is a [OffsetSeqMetadata](#).

Note

	<code>offsetSeqMetadata</code> is part of the ProgressReporter Contract to hold the current event-time watermark and timestamp.
--	---

`offsetSeqMetadata` is used to create an [IncrementalExecution](#) in the [queryPlanning](#) phase of the [MicroBatchExecution](#) and [ContinuousExecution](#) execution engines.

`offsetSeqMetadata` is initialized (with `0` for `batchWatermarkMs` and `batchTimestampMs`) when `StreamExecution` is requested to [run stream processing](#).

`offsetSeqMetadata` is then updated (with the current event-time watermark and timestamp) when `MicroBatchExecution` is requested to [construct the next streaming micro-batch](#).

Note

`MicroBatchExecution` uses the [WatermarkTracker](#) for the current event-time watermark and the [trigger clock](#) for the current batch timestamp.

`offsetSeqMetadata` is stored (*checkpointed*) in the [walCommit](#) phase of [MicroBatchExecution](#) (and printed out as INFO message to the logs).

FIXME INFO message

`offsetSeqMetadata` is restored (*re-created*) from a checkpointed state when `MicroBatchExecution` is requested to [populate start offsets](#).

isActive Method

`isActive: Boolean`

Note

`isActive` is part of the [StreamingQuery Contract](#) to indicate whether a streaming query is active (`true`) or not (`false`).

`isActive` is enabled (`true`) as long as the [State](#)

exception Method

`exception: Option[StreamingQueryException]`

Note

`exception` is part of the [StreamingQuery Contract](#) to indicate whether a streaming query...FIXME

`exception` ...FIXME

Internal Properties

Name	Description

awaitProgressLock	Java's fair reentrant mutual exclusion java.util.concurrent.locks.ReentrantLock (that favors granting access to the longest-waiting thread under contention)
awaitProgressLockCondition	Lock
callSite	
currentBatchId	<p>ID (number) of the current streaming batch</p> <ul style="list-style-type: none"> • Starts at <code>-1</code> when <code>StreamExecution</code> is created • <code>0</code> when <code>StreamExecution</code> populates start offsets (and OffsetSeqLog is empty, i.e. no offset files in <code>offsets</code> directory in checkpoint) • Incremented when <code>StreamExecution</code> runs streaming batches and finishes a trigger that had data available from sources (right after committing the batch).
initializationLatch	
lastExecution	IncrementalExecution from the very recent (<i>last</i>) execution
newData	<p><code>newData: Map[BaseStreamingSource, LogicalPlan]</code></p> <p>Registry of the streaming sources (in the logical query plan) that have new data available in the current batch. The new data is a streaming <code>DataFrame</code>.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <p>Note <code>newData</code> is part of the ProgressReporter Contract.</p> </div> <p>Set exclusively when <code>StreamExecution</code> is requested to requests unprocessed data from streaming sources (while running a single streaming batch).</p> <p>Used exclusively when <code>StreamExecution</code> replaces StreamingExecutionRelations in a logical query plan with relations with new data (while running a single streaming batch).</p>
noNewData	<p>Flag that indicates whether there are new offsets (<i>data</i>) available for processing or not</p> <ul style="list-style-type: none"> • Enabled (<code>true</code>) when constructing the next streaming micro-batch when no new offsets are available

<code>pollingDelayMs</code>	<p>Time delay before polling new data again when no data was available</p> <p>Set to spark.sql.streaming.pollingDelay Spark property.</p> <p>Used when <code>StreamExecution</code> has started running streaming batches (and no data was available to process in a trigger).</p>
<code>prettyIdString</code>	<p>Pretty-identified string for identification in logs (with <code>name</code> if defined).</p> <pre>queryName [id = xyz, runId = abc] [id = xyz, runId = abc]</pre>
<code>startLatch</code>	<p>Java's java.util.concurrent.CountDownLatch with count 1.</p> <p>Used when <code>StreamExecution</code> is requested to start to pause the main thread until <code>StreamExecution</code> was requested to run the streaming query.</p>
<code>streamDeathCause</code>	<p><code>StreamingQueryException</code></p>
<code>streamMetrics</code>	<p>MetricsReporter with spark.streaming.[name or id] source name</p> <p>Uses <code>name</code> if defined (can be <code>null</code>) or falls back to <code>id</code></p>
<code>uniqueSources</code>	<p>Unique streaming sources (after being collected as <code>StreamingExecutionRelation</code> from the logical query plan).</p> <div data-bbox="624 1432 1394 1673" style="border: 1px solid #ccc; padding: 10px;"> <p>Note StreamingExecutionRelation is a leaf logical operator (i.e. <code>LogicalPlan</code>) that represents a streaming data source (and corresponds to a single StreamingRelation in analyzed logical query plan of a streaming Dataset).</p> </div> <p>Used when <code>StreamExecution</code>:</p> <ul style="list-style-type: none"> • Constructs the next streaming micro-batch (and gets new offsets for every streaming data source) • Stops all streaming data sources

StreamingQueryWrapper — Serializable StreamExecution

StreamingQueryWrapper is a serializable interface of a StreamExecution.

Demo: Any Streaming Query is StreamingQueryWrapper

```
import org.apache.spark.sql.execution.streaming.StreamingQueryWrapper
val query = spark
  .readStream
  .format("rate")
  .load
  .writeStream
  .format("memory")
  .queryName("rate2memory")
  .start
assert(query.isInstanceOf[StreamingQueryWrapper])
```

StreamingQueryWrapper has the same StreamExecution API and simply passes all the method calls along to the underlying StreamExecution.

StreamingQueryWrapper is created when StreamingQueryManager is requested to create a streaming query (when DataStreamWriter is requested to start an execution of the streaming query).

TriggerExecutor

`TriggerExecutor` is the [interface](#) for **trigger executors** that `StreamExecution` uses to execute a batch runner.

Note

Batch runner is an executable code that is executed at regular intervals. It is also called a **trigger handler**.

```
package org.apache.spark.sql.execution.streaming

trait TriggerExecutor {
  def execute(batchRunner: () => Boolean): Unit
}
```

Note

`StreamExecution` reports a `IllegalStateException` when `TriggerExecutor` is different from the [two built-in implementations](#): `OneTimeExecutor` or `ProcessingTimeExecutor`.

Table 1. `TriggerExecutor`'s Available Implementations

TriggerExecutor	Description		
<code>OneTimeExecutor</code>	Executes <code>batchRunner</code> exactly once.		
<code>ProcessingTimeExecutor</code>	<p>Executes <code>batchRunner</code> at regular intervals (as defined using <code>ProcessingTime</code> and <code>DataStreamWriter.trigger</code> method).</p> <pre>ProcessingTimeExecutor(processingTime: ProcessingTime, clock: Clock = new SystemClock())</pre> <table border="1"> <tr> <td>Note</td> <td>Processing terminates when <code>batchRunner</code> returns <code>false</code>.</td> </tr> </table>	Note	Processing terminates when <code>batchRunner</code> returns <code>false</code> .
Note	Processing terminates when <code>batchRunner</code> returns <code>false</code> .		

notifyBatchFallingBehind Method

Caution

FIXME

IncrementalExecution — QueryExecution of Streaming Queries

`IncrementalExecution` is the `QueryExecution` of streaming queries.

Tip

Read up on [QueryExecution](#) in [The Internals of Spark SQL book](#).

`IncrementalExecution` is [created](#) (and becomes the `StreamExecution.lastExecution`) when:

- `MicroBatchExecution` is requested to [run a single streaming micro-batch](#) (in `queryPlanning` phase)
- `ContinuousExecution` is requested to [run a streaming query in continuous mode](#) (in `queryPlanning` phase)
- `Dataset.explain` operator is executed (on a streaming query)

`IncrementalExecution` uses the `statefulOperatorId` internal counter for the IDs of the stateful operators in the [optimized logical plan](#) (while applying the [preparations](#) rules) when requested to prepare the plan for execution (in `executedPlan` phase).

Preparing Logical Plan (of Streaming Query) for Execution — `optimizedPlan` and `executedPlan` Phases of Query Execution

When requested for the optimized logical plan (of the [logical plan](#)), `IncrementalExecution` transforms `CurrentBatchTimestamp` and `ExpressionWithRandomSeed` expressions with the timestamp literal and new random seeds, respectively. When transforming `CurrentBatchTimestamp` expressions, `IncrementalExecution` prints out the following INFO message to the logs:

```
Current batch timestamp = [timestamp]
```

Once [created](#), `IncrementalExecution` is immediately executed (by the `MicroBatchExecution` and `ContinuousExecution` stream execution engines in the `queryPlanning` phase) and so the entire query execution pipeline is executed up to and including `executedPlan`. That means that the [extra planning strategies](#) and the [state preparation rule](#) have been applied at this point and the [streaming query](#) is ready for execution.

Creating IncrementalExecution Instance

`IncrementalExecution` takes the following to be created:

- `SparkSession`
- `Logical plan (LogicalPlan)`
- `OutputMode` (as specified using `DataStreamWriter.outputMode` method)
- `State checkpoint location`
- Run ID of a streaming query (`UUID`)
- Batch ID
- `OffsetSeqMetadata`

State Checkpoint Location (Directory)

When `created`, `IncrementalExecution` is given the `checkpoint location`.

For the two available execution engines ([MicroBatchExecution](#) and [ContinuousExecution](#)), the checkpoint location is actually `state` directory under the `checkpoint root directory`.

```
val queryName = "rate2memory"
val checkpointLocation = s"file:/tmp/checkpoint-$queryName"
val query = spark
  .readStream
  .format("rate")
  .load
  .writeStream
  .format("memory")
  .queryName(queryName)
  .option("checkpointLocation", checkpointLocation)
  .start

// Give the streaming query a moment (one micro-batch)
// So lastExecution is available for the checkpointLocation
import scala.concurrent.duration._
query.awaitTermination(1.second.toMillis)

import org.apache.spark.sql.execution.streaming.StreamingQueryWrapper
val stateCheckpointDir = query
  .asInstanceOf[StreamingQueryWrapper]
  .streamingQuery
  .lastExecution
  .checkpointLocation
val stateDir = s"$checkpointLocation/state"
assert(stateCheckpointDir equals stateDir)
```

State checkpoint location is used exclusively when `IncrementalExecution` is requested for the [state info of the next stateful operator](#) (when requested to optimize a streaming physical plan using the [state preparation rule](#) that creates the stateful physical operators:

[StateStoreSaveExec](#), [StateStoreRestoreExec](#), [StreamingDeduplicateExec](#), [FlatMapGroupsWithStateExec](#), [StreamingSymmetricHashJoinExec](#), and [StreamingGlobalLimitExec](#)).

Number of State Stores (`spark.sql.shuffle.partitions`)

— `numStateStores` Internal Property

`numStateStores: Int`

`numStateStores` is the **number of state stores** which corresponds to `spark.sql.shuffle.partitions` configuration property (default: `200`).

Tip

Read up on [spark.sql.shuffle.partitions](#) configuration property (and the others) in [The Internals of Spark SQL book](#).

Internally, `numStateStores` requests the [OffsetSeqMetadata](#) for the `spark.sql.shuffle.partitions` configuration property (using the [streaming configuration](#)) or simply takes whatever was defined for the given [SparkSession](#) (default: `200`).

`numStateStores` is initialized right when `IncrementalExecution` is [created](#).

`numStateStores` is used exclusively when `IncrementalExecution` is requested for the [state info of the next stateful operator](#) (when requested to optimize a streaming physical plan using the [state preparation rule](#) that creates the stateful physical operators: [StateStoreSaveExec](#), [StateStoreRestoreExec](#), [StreamingDeduplicateExec](#), [FlatMapGroupsWithStateExec](#), [StreamingSymmetricHashJoinExec](#), and [StreamingGlobalLimitExec](#)).

Extra Planning Strategies for Streaming Queries

— `planner` Property

`IncrementalExecution` uses a custom `SparkPlanner` with the following **extra planning strategies** to plan the [streaming query](#) for execution:

- [StreamingJoinStrategy](#)
- [StatefulAggregationStrategy](#)
- [FlatMapGroupsWithStateStrategy](#)

- [StreamingRelationStrategy](#)
- [StreamingDeduplicationStrategy](#)
- [StreamingGlobalLimitStrategy](#)

TipRead up on [SparkPlanner](#) in [The Internals of Spark SQL](#) book.

State Preparation Rule For Batch-Specific Configuration — `state` Property

```
state: Rule[SparkPlan]
```

`state` is a custom physical preparation rule (`Rule[SparkPlan]`) that can transform a streaming physical plan (`SparkPlan`) with the following physical operators:

- [StateStoreSaveExec](#) with any unary physical operator (`UnaryExecNode`) with a [StateStoreRestoreExec](#)
- [StreamingDuplicateExec](#)
- [FlatMapGroupsWithStateExec](#)
- [StreamingSymmetricHashJoinExec](#)
- [StreamingGlobalLimitExec](#)

`state` simply transforms the physical plan with the above physical operators and fills out the execution-specific configuration:

- [nextStatefulOperationStateInfo](#) for the state info
- [OutputMode](#)
- [batchWatermarkMs](#) (through the [OffsetSeqMetadata](#)) for the event-time watermark
- [batchTimestampMs](#) (through the [OffsetSeqMetadata](#)) for the current timestamp

`state` rule is used (as part of the physical query optimizations) when [IncrementalExecution](#) is requested to [optimize \(prepare\) the physical plan of the streaming query](#) (once for [ContinuousExecution](#) and every trigger for [MicroBatchExecution](#) in their [queryPlanning](#) phases).

TipRead up on [Physical Query Optimizations](#) in [The Internals of Spark SQL](#) book.

`nextStatefulOperationStateInfo` Internal Method

```
nextStatefulOperationStateInfo(): StatefulOperatorStateInfo
```

`nextStatefulOperationStateInfo` simply creates a new `StatefulOperatorStateInfo` with the [state checkpoint location](#), the [run ID](#) (of the streaming query), the next [statefulOperator ID](#), the [current batch ID](#), and the [number of state stores](#).

Note

The only changing part of `StatefulOperatorStateInfo` across executions of the `nextStatefulOperationStateInfo` method is the the next [statefulOperator ID](#).

All the other properties (the [state checkpoint location](#), the [run ID](#), the [current batch ID](#), and the [number of state stores](#)) are the same within a single `IncrementalExecution` instance.

The only two properties that may ever change are the [run ID](#) (after a streaming query is restarted from the checkpoint) and the [current batch ID](#) (every micro-batch in `MicroBatchExecution` execution engine).

Note

`nextStatefulOperationStateInfo` is used exclusively when `IncrementalExecution` is requested to optimize a streaming physical plan using the [state preparation rule](#) (and creates the stateful physical operators: `StateStoreSaveExec`, `StateStoreRestoreExec`, `StreamingDeduplicateExec`, `FlatMapGroupsWithStateExec`, `StreamingSymmetricHashJoinExec`, and `StreamingGlobalLimitExec`).

Checking Out Whether Last Execution Requires Another Non-Data Micro-Batch — `shouldRunAnotherBatch` Method

```
shouldRunAnotherBatch(newMetadata: OffsetSeqMetadata): Boolean
```

`shouldRunAnotherBatch` is positive (`true`) if there is at least one `StateStoreWriter` operator (in the `executedPlan` physical query plan) that requires another non-data batch (per the given `OffsetSeqMetadata` with the event-time watermark and the batch timestamp).

Otherwise, `shouldRunAnotherBatch` is negative (`false`).

Note

`shouldRunAnotherBatch` is used exclusively when `MicroBatchExecution` is requested to [construct the next streaming micro-batch](#) (and checks out whether the last batch execution requires another non-data batch).

Demo: State Checkpoint Directory

```
// START: Only for easier debugging
// The state is then only for one partition
```

```

// which should make monitoring easier
import org.apache.spark.sql.internal.SQLConf.SHUFFLE_PARTITIONS
spark.sessionState.conf.setConf(SHUFFLE_PARTITIONS, 1)

assert(spark.sessionState.conf.numShufflePartitions == 1)
// END: Only for easier debugging

val counts = spark
  .readStream
  .format("rate")
  .load
  .groupBy(window($"timestamp", "5 seconds") as "group")
  .agg(count("value") as "value_count") // <-- creates an Aggregate logical operator
  .orderBy("group") // <-- makes for easier checking

assert(counts.isStreaming, "This should be a streaming query")

// Search for "checkpoint = <unknown>" in the following output
// Looks for StateStoreSave and StateStoreRestore
scala> counts.explain
== Physical Plan ==
*(5) Sort [group#5 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(group#5 ASC NULLS FIRST, 1)
  +- *(4) HashAggregate(keys=[window#11], functions=[count(value#1L)])
    +- StateStoreSave [window#11], state info [ checkpoint = <unknown>, runId = 558b
f725-accb-487d-97eb-f790fa4a6138, opId = 0, ver = 0, numPartitions = 1], Append, 0, 2
    +- *(3) HashAggregate(keys=[window#11], functions=[merge_count(value#1L)])
      +- StateStoreRestore [window#11], state info [ checkpoint = <unknown>, run
Id = 558bf725-accb-487d-97eb-f790fa4a6138, opId = 0, ver = 0, numPartitions = 1], 2
      +- *(2) HashAggregate(keys=[window#11], functions=[merge_count(value#1L
)])
        +- Exchange hashpartitioning(window#11, 1)
          +- *(1) HashAggregate(keys=[window#11], functions=[partial_count(
value#1L)])
            +- *(1) Project [named_struct(start, precisetimestampconversion(
(((((CASE WHEN (cast(CEIL((cast((precisetimestampconversion(timestamp#0, TimestampType
, LongType) - 0) as double) / 5000000.0)) as double) = (cast((precisetimestampconversion(
(timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0)) THEN (CEIL((cas
t((precisetimestampconversion(timestamp#0, TimestampType, LongType) - 0) as double) /
5000000.0)) + 1) ELSE CEIL((cast((precisetimestampconversion(timestamp#0, TimestampType
, LongType) - 0) as double) / 5000000.0)) END + 0) - 1) * 5000000) + 0), LongType, Tim
estampType), end, precisetimestampconversion(((CASE WHEN (cast(CEIL((cast((preciseti
mestampconversion(timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0))
as double) = (cast((precisetimestampconversion(timestamp#0, TimestampType, LongType) -
0) as double) / 5000000.0)) THEN (CEIL((cast((precisetimestampconversion(timestamp#0,
TimestampType, LongType) - 0) as double) / 5000000.0)) + 1) ELSE CEIL((cast((preciseti
mestampconversion(timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0))
END + 0) - 1) * 5000000) + 5000000, LongType, TimestampType)) AS window#11, value#1L]
            +- *(1) Filter isnotnull(timestamp#0)
              +- StreamingRelation rate, [timestamp#0, value#1L]

// Start the query to access lastExecution that has the checkpoint resolved
import scala.concurrent.duration._
```

```
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
val t = Trigger.ProcessingTime(1.hour) // should be enough time for exploration
val sq = counts
  .writeStream
  .format("console")
  .option("truncate", false)
  .option("checkpointLocation", "/tmp/spark-streams-state-checkpoint-root")
  .trigger(t)
  .outputMode(OutputMode.Complete)
  .start

// wait till the first batch which should happen right after start

import org.apache.spark.sql.execution.streaming._
val lastExecution = sq.asInstanceOf[StreamingQueryWrapper].streamingQuery.lastExecution
scala> println(lastExecution.checkpointLocation)
file:/tmp/spark-streams-state-checkpoint-root/state
```

StreamingQueryListenerBus — Notification Bus for Streaming Events

`StreamingQueryListenerBus` is an event bus (i.e. `ListenerBus`) to dispatch streaming events to `StreamingQueryListener` streaming event listeners.

`StreamingQueryListenerBus` is created when `StreamingQueryManager` is created (as the internal `listenerBus`).

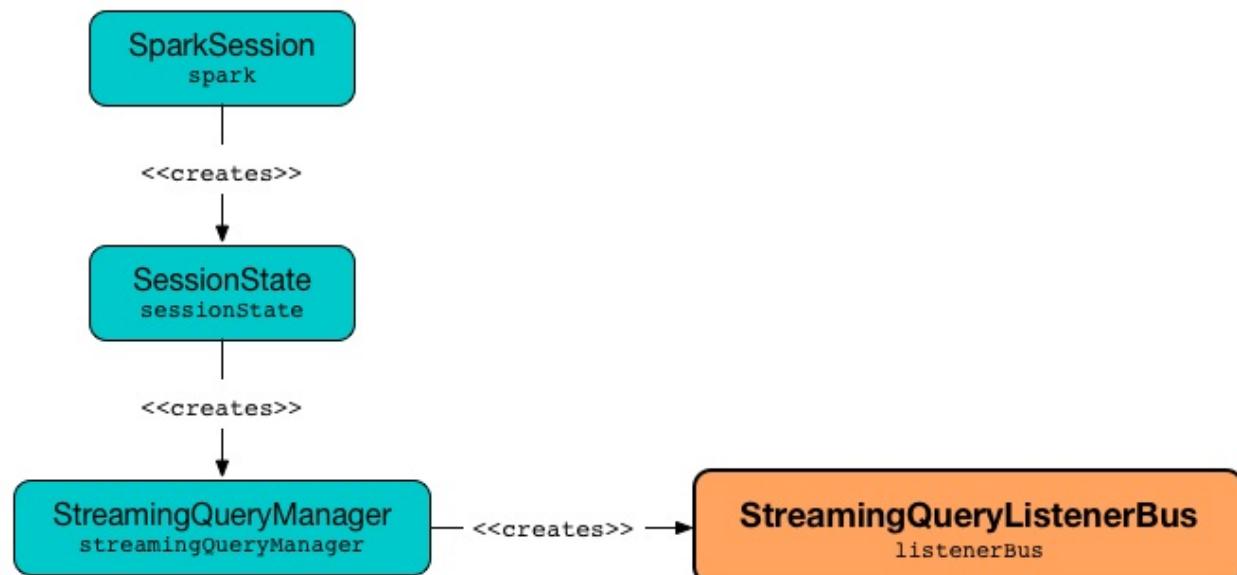


Figure 1. `StreamingQueryListenerBus` is Created Once In `SparkSession`

`StreamingQueryListenerBus` is also a `SparkListener` and registers itself with `LiveListenerBus` to intercept a `QueryStartedEvent`.

Table 1. `StreamingQueryListenerBus`'s Internal Registries and Counters

Name	Description
<code>activeQueryRunIds</code>	<p>Collection of active streaming queries by their <code>runIds</code>.</p> <ul style="list-style-type: none"> <code>runId</code> is added when <code>StreamingQueryListenerBus</code> posts a <code>QueryStartedEvent</code> event to <code>LiveListenerBus</code> <code>runId</code> is removed when <code>StreamingQueryListenerBus</code> postToAll a <code>QueryTerminatedEvent</code> event <p>Used mainly when <code>StreamingQueryListenerBus</code> dispatches an event to listeners (for queries started in the same <code>SparkSession</code>).</p>

Posting `StreamingQueryListener` Events to `LiveListenerBus` — `post` Method

```
post(event: StreamingQueryListener.Event): Unit
```

`post` simply posts the input `event` straight to [LiveListenerBus](#) except [QueryStartedEvent](#) events.

For `QueryStartedEvent` events, `post` adds the query's `runId` to [activeQueryRunIds](#) registry first before posting the event to [LiveListenerBus](#) followed by `postToAll`.

Note

`post` is used exclusively when [StreamingQueryManager](#) posts [StreamingQueryListener](#) event.

onOtherEvent Method

Caution**FIXME**

doPostEvent Method

Caution**FIXME**

postToAll Method

Caution**FIXME**

Creating StreamingQueryListenerBus Instance

`StreamingQueryListenerBus` takes the following when created:

- `LiveListenerBus`

`StreamingQueryListenerBus` registers itself with [LiveListenerBus](#).

`StreamingQueryListenerBus` initializes the [internal registries and counters](#).

StreamMetadata

`StreamMetadata` is a metadata associated with a [StreamingQuery](#) (indirectly through [StreamExecution](#)).

`StreamMetadata` takes an ID to be created.

`StreamMetadata` is [created](#) exclusively when [StreamExecution](#) is created (with a randomly-generated 128-bit universally unique identifier (UUID)).

`StreamMetadata` can be [persisted](#) to and [unpersisted](#) from a JSON file. `StreamMetadata` uses [json4s-jackson](#) library for JSON persistence.

```
import org.apache.spark.sql.execution.streaming.StreamMetadata
import org.apache.hadoop.fs.Path
val metadataPath = new Path("metadata")

scala> :type spark
org.apache.spark.sql.SparkSession

val hadoopConf = spark.sessionState.newHadoopConf()
val sm = StreamMetadata.read(metadataPath, hadoopConf)

scala> :type sm
Option[org.apache.spark.sql.execution.streaming.StreamMetadata]
```

Unpersisting StreamMetadata (from JSON File) — `read` Object Method

```
read(
  metadataFile: Path,
  hadoopConf: Configuration): Option[StreamMetadata]
```

`read` unpersists `StreamMetadata` from the given `metadataFile` file if available.

`read` returns a `StreamMetadata` if the metadata file was available and the content could be read in JSON format. Otherwise, `read` returns `None`.

Note	<code>read</code> uses <code>org.json4s.jackson.Serialization.read</code> for JSON deserialization.
------	---

Note	<code>read</code> is used exclusively when <code>StreamExecution</code> is created (and tries to read the <code>metadata</code> checkpoint file).
------	---

Persisting Metadata — `write` Object Method

```
write(  
  metadata: StreamMetadata,  
  metadataFile: Path,  
  hadoopConf: Configuration): Unit
```

`write` persists the given `StreamMetadata` to the given `metadataFile` file in JSON format.

Note `write` uses `org.json4s.jackson.Serialization.write` for JSON serialization.

Note `write` is used exclusively when `streamExecution` is [created](#) (and the metadata checkpoint file is not available).

EventTimeWatermark Unary Logical Operator

— Streaming Watermark

`EventTimeWatermark` is a unary logical operator that is [created](#) to represent `Dataset.withWatermark` operator in a logical query plan of a streaming query.

Note	<p>A unary logical operator (<code>UnaryNode</code>) is a logical operator with a single child logical operator.</p> <p>Read up on UnaryNode (and logical operators in general) in The Internals of Spark SQL book.</p>
------	---

`EventTimeWatermark` is resolved (aka *planned*) to [EventTimeWatermarkExec](#) physical operator in [StatefulAggregationStrategy](#) execution planning strategy.

Note	<p><code>EliminateEventTimeWatermark</code> logical optimization rule (i.e. <code>Rule[LogicalPlan]</code>) removes <code>EventTimeWatermark</code> logical operator from a logical plan if the child logical operator is streaming, i.e. when <code>Dataset.withWatermark</code> operator is used on a batch query.</p> <pre> val logs = spark. read. // <-- batch non-streaming query that makes `EliminateEventTimeWatermark` applicable format("text"). load("logs") // logs is a batch Dataset assert(!logs.isStreaming) val q = logs. withWatermark(eventTime = "timestamp", delayThreshold = "30 seconds") // <-- EventTimeWatermark scala> println(q.queryExecution.logical.numberedTreeString) // <-- no EventTimeWatermark as it was removed immediately 00 Relation[value#0] text </pre>
------	---

Creating EventTimeWatermark Instance

`EventTimeWatermark` takes the following to be created:

- Name of the column for [event-time watermark](#)
- Delay (`CalendarInterval`)
- Child logical operator (`LogicalPlan`)

Output Schema — `output` Property

```
output: Seq[Attribute]
```

Note

`output` is part of the `QueryPlan` Contract to describe the attributes of (the schema of) the output.

`output` finds `eventTime` column in the output schema of the `child` logical operator and updates the `Metadata` of the column with `spark.watermarkDelayMs` key and the milliseconds for the delay.

`output` removes `spark.watermarkDelayMs` key from the other columns.

```
// FIXME How to access/show the eventTime column with the metadata updated to include
spark.watermarkDelayMs?
import org.apache.spark.sql.catalyst.plans.logical.EventTimeWatermark
val etw = q.queryExecution.logical.asInstanceOf[EventTimeWatermark]
scala> etw.output.toStructType.printTreeString
root
| -- timestamp: timestamp (nullable = true)
| -- value: long (nullable = true)
```

spark.watermarkDelayMs Metadata Key

`EventTimeWatermark` uses `spark.watermarkDelayMs` key (in the `Metadata` of the `output attributes`) to hold the event-time watermark delay (as a so-called *watermark attribute* or *eventTime watermark*).

The **event-time watermark delay** is used to calculate the difference between the event time of an event (modeled as a row in the streaming Dataset) and the time in the past.

FlatMapGroupsWithState Unary Logical Operator

`FlatMapGroupsWithState` is a unary logical operator that is [created](#) to represent the following operators in a logical query plan of a streaming query:

- [KeyValueGroupedDataset.mapGroupsWithState](#)
- [KeyValueGroupedDataset.flatMapGroupsWithState](#)

Note	A unary logical operator (<code>UnaryNode</code>) is a logical operator with a single child logical operator. Read up on UnaryNode (and logical operators in general) in The Internals of Spark SQL book .
------	---

`FlatMapGroupsWithState` is resolved (*planned*) to:

- [FlatMapGroupsWithStateExec](#) unary physical operator for streaming datasets (in [FlatMapGroupsWithStateStrategy](#) execution planning strategy)
- [MapGroupsExec](#) physical operator for batch datasets (in [BasicOperators](#) execution planning strategy)

Creating `SerializeFromObject` with `FlatMapGroupsWithState` — `apply` Factory Method

```
apply[K: Encoder, V: Encoder, S: Encoder, U: Encoder](
  func: (Any, Iterator[Any], LogicalGroupState[Any]) => Iterator[Any],
  groupingAttributes: Seq[Attribute],
  dataAttributes: Seq[Attribute],
  outputMode: OutputMode,
  isMapGroupsWithState: Boolean,
  timeout: GroupStateTimeout,
  child: LogicalPlan): LogicalPlan
```

`apply` [creates](#) a `SerializeFromObject` logical operator with a `FlatMapGroupsWithState` as its child logical operator.

Internally, `apply` [creates](#) `SerializeFromObject` object consumer (aka unary logical operator) with `FlatMapGroupsWithState` logical plan.

Internally, `apply` finds `ExpressionEncoder` for the type `s` and creates a `FlatMapGroupsWithState` with `UnresolvedDeserializer` for the types `k` and `v`.

In the end, `apply` creates a `SerializeFromObject` object consumer with the `FlatMapGroupsWithState`.

Note `apply` is used in [KeyValueGroupedDataset.flatMapGroupsWithState](#) operator.

Creating FlatMapGroupsWithState Instance

`FlatMapGroupsWithState` takes the following to be created:

- State function (`(Any, Iterator[Any], LogicalGroupState[Any]) ⇒ Iterator[Any]`)
- Key deserializer Catalyst expression
- Value deserializer Catalyst expression
- Grouping attributes
- Data attributes
- Output object attribute
- State `ExpressionEncoder`
- [Output mode](#)
- `isMapGroupsWithState` flag (default: `false`)
- [GroupStateTimeout](#)
- Child logical operator (`LogicalPlan`)

Deduplicate Unary Logical Operator

`Deduplicate` is a unary logical operator (i.e. `LogicalPlan`) that is [created](#) to represent `dropDuplicates` operator (that drops duplicate records for a given subset of columns).

`Deduplicate` has [streaming](#) flag enabled for streaming Datasets.

```
val uniqueRates = spark.  
  readStream.  
  format("rate").  
  load.  
  dropDuplicates("value") // <-- creates Deduplicate logical operator  
// Note the streaming flag  
scala> println(uniqueRates.queryExecution.logical.numberedTreeString)  
00 Deduplicate [value#33L], true // <-- streaming flag enabled  
01 +- StreamingRelation DataSource(org.apache.spark.sql.SparkSession@4785f176, rate, List  
(,), None, List(), None, Map(), None), rate, [timestamp#32, value#33L]
```

Caution

FIXME Example with duplicates across batches to show that `Deduplicate` keeps state and [withWatermark](#) operator should also be used to limit how much is stored (to not cause OOM)

Note

`UnsupportedOperationChecker` [ensures](#) that `dropDuplicates` operator is not used after the following code is not supported in Structured Streaming and results in an `AnalysisException`:

```
val counts = spark.  
  readStream.  
  format("rate").  
  load.  
  groupBy(window($"timestamp", "5 seconds") as "group").  
  agg(count("value") as "value_count").  
  dropDuplicates // <-- after groupBy  
  
import scala.concurrent.duration._  
import org.apache.spark.sql.streaming.{OutputMode, Trigger}  
val sq = counts.  
  writeStream.  
  format("console").  
  trigger(Trigger.ProcessingTime(10.seconds)).  
  outputMode(OutputMode.Complete).  
  start  
org.apache.spark.sql.AnalysisException: dropDuplicates is not supported after a
```

Note	<p><code>Deduplicate</code> logical operator is translated (aka <i>planned</i>) to:</p> <ul style="list-style-type: none">• <code>StreamingDeduplicateExec</code> physical operator in <code>StreamingDeduplicationStrategy</code> execution planning strategy for streaming Datasets (aka <i>streaming plans</i>)• <code>Aggregate</code> physical operator in <code>ReplaceDeduplicatewithAggregate</code> execution planning strategy for non-streaming/batch Datasets (aka <i>batch plans</i>)
------	---

The output schema of `Deduplicate` is exactly the `child`'s output schema.

Creating Deduplicate Instance

`Deduplicate` takes the following when created:

- Attributes for keys
- Child logical operator (i.e. `LogicalPlan`)
- Flag whether the logical operator is for streaming (enabled) or batch (disabled) mode

MemoryPlan Logical Operator

`MemoryPlan` is a leaf logical operator (i.e. `LogicalPlan`) that is used to query the data that has been written into a `MemorySink`. `MemoryPlan` is created when [starting continuous writing](#) (to a `MemorySink`).

Tip

See the example in [MemoryStream](#).

```
scala> intsOut.explain(true)
== Parsed Logical Plan ==
SubqueryAlias memstream
+- MemoryPlan org.apache.spark.sql.execution.streaming.MemorySink@481bf251, [value#21]

== Analyzed Logical Plan ==
value: int
SubqueryAlias memstream
+- MemoryPlan org.apache.spark.sql.execution.streaming.MemorySink@481bf251, [value#21]

== Optimized Logical Plan ==
MemoryPlan org.apache.spark.sql.execution.streaming.MemorySink@481bf251, [value#21]

== Physical Plan ==
LocalTableScan [value#21]
```

When executed, `MemoryPlan` is translated to `LocalTableScanExec` physical operator (similar to `LocalRelation` logical operator) in `BasicOperators` execution planning strategy.

StreamingRelation Leaf Logical Operator for Streaming Source

`StreamingRelation` is a leaf logical operator (i.e. `LogicalPlan`) that represents a [streaming source](#) in a logical plan.

`StreamingRelation` is [created](#) when `DataStreamReader` is requested to load data from a [streaming source](#) and creates a streaming `Dataset`.



Figure 1. `StreamingRelation` Represents Streaming Source

```

val rate = spark.
  readStream.      // <-- creates a DataStreamReader
  format("rate").
  load("hello")   // <-- creates a StreamingRelation
scala> println(rate.queryExecution.logical.numberedTreeString)
00 StreamingRelation DataSource(org.apache.spark.sql.SparkSession@4e5dcc50,rate,List(),
  None,None,Map(path -> hello),None), rate, [timestamp#0, value#1L]
  
```



`isStreaming` flag is always enabled (i.e. `true`).

```

import org.apache.spark.sql.execution.streaming.StreamingRelation
val relation = rate.queryExecution.logical.asInstanceOf[StreamingRelation]
scala> relation.isStreaming
res1: Boolean = true
  
```

`toString` gives the [source name](#).

```

scala> println(relation)
rate
  
```

Note

`StreamingRelation` is [resolved](#) (aka *planned*) to [StreamingExecutionRelation](#) (right after `StreamExecution` starts running batches).

Creating `StreamingRelation` for `DataSource` — `apply` Factory Method

```

apply(dataSource: DataSource): StreamingRelation
  
```

`apply` creates a `StreamingRelation` for the input streaming `DataSource` and the short name and the schema of the streaming source (behind the `DataSource`).

Note	<code>apply</code> creates a <code>StreamingRelation</code> logical operator (for the input <code>DataSource</code>) that represents a streaming source.
Note	<code>apply</code> is used exclusively when <code>DataStreamReader</code> is requested to load data from a streaming source to a streaming <code>Dataset</code> .

Creating StreamingRelation Instance

`StreamingRelation` takes the following when created:

- `DataSource`
- Short name of the streaming source
- Output attributes of the schema of the streaming source

StreamingRelationV2 Leaf Logical Operator

`StreamingRelationV2` is a `MultiInstanceRelation` leaf logical operator that represents `MicroBatchReadSupport` or `ContinuousReadSupport` streaming data sources in a logical plan of a streaming query.

Tip

Read up on [Leaf logical operators](#) in [The Internals of Spark SQL](#) book.

`StreamingRelationV2` is [created](#) when:

- `DataStreamReader` is requested to "load" data as a `streaming DataFrame` for `MicroBatchReadSupport` and `ContinuousReadSupport` streaming data sources
- `ContinuousMemoryStream` is created

`isStreaming` flag is always enabled (i.e. `true`).

```
scala> :type sq
org.apache.spark.sql.DataFrame

import org.apache.spark.sql.execution.streaming.StreamingRelationV2
val relation = sq.queryExecution.logical.asInstanceOf[StreamingRelationV2]
assert(relation.isStreaming)
```

`StreamingRelationV2` is resolved (*replaced*) to the following leaf logical operators:

- `ContinuousExecutionRelation` when `ContinuousExecution` stream execution engine is requested for the [analyzed logical plan](#)
- `StreamingExecutionRelation` when `MicroBatchExecution` stream execution engine is requested for the [analyzed logical plan](#)

Creating StreamingRelationV2 Instance

`StreamingRelationV2` takes the following to be created:

- `DataSourceV2`
- Name of the data source
- Options (`Map[String, String]`)
- Output attributes (`seq[Attribute]`)
- Optional `StreamingRelation`

- `SparkSession`

StreamingExecutionRelation Leaf Logical Operator for Streaming Source At Execution

`StreamingExecutionRelation` is a leaf logical operator (i.e. `LogicalPlan`) that represents a [streaming source](#) in the logical query plan of a streaming `Dataset`.

The main use of `StreamingExecutionRelation` logical operator is to be a "placeholder" in a logical query plan that will be replaced with the real relation (with new data that has arrived since the last batch) or an empty `LocalRelation` when `StreamExecution` [runs a single streaming batch](#).

`StreamingExecutionRelation` is [created](#) for a `StreamingRelation` in [analyzed logical query plan](#) (that is the execution representation of a streaming `Dataset`).

Note

Right after `StreamExecution` [has started running streaming batches](#) it initializes the streaming sources by transforming the analyzed logical plan of the streaming `Dataset` so that every `StreamingRelation` logical operator is replaced by the corresponding `StreamingExecutionRelation`.

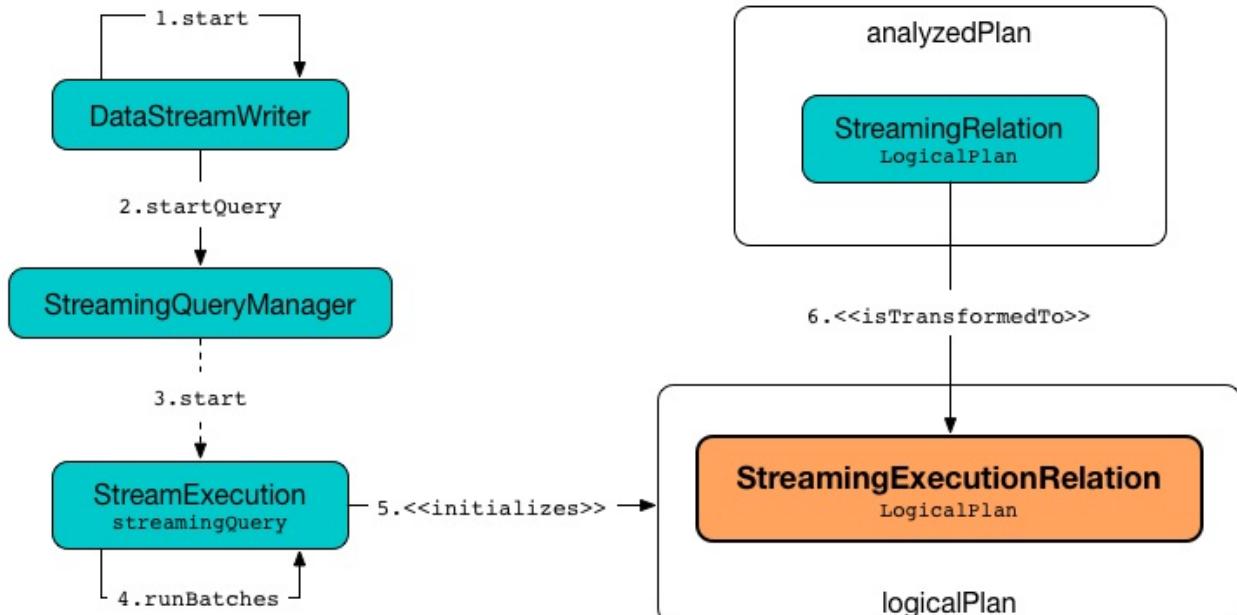


Figure 1. `StreamingExecutionRelation` Represents Streaming Source At Execution

Note

`StreamingExecutionRelation` is also resolved (aka *planned*) to a `StreamingRelationExec` physical operator in `StreamingRelationStrategy` execution planning strategy only when [explaining](#) a streaming `Dataset`.

Creating `StreamingExecutionRelation` Instance

`StreamingExecutionRelation` takes the following when created:

- [Streaming source](#)
- [Output attributes](#)

Creating `StreamingExecutionRelation` (for `MemoryStream` Source) — `apply` Factory Method

```
apply(source: Source): StreamingExecutionRelation
```

`apply` creates a `StreamingExecutionRelation` for the input `source` and with the attributes of the `schema` of the `source`.

Note

`apply` is used exclusively when `MemoryStream` is created (and the logical plan initialized).

EventTimeWatermarkExec Unary Physical Operator

`EventTimeWatermarkExec` is a unary physical operator that represents [EventTimeWatermark](#) logical operator at execution time.

Note

A unary physical operator (`UnaryExecNode`) is a physical operator with a single [child](#) physical operator.

Read up on [UnaryExecNode](#) (and physical operators in general) in [The Internals of Spark SQL](#) book.

The purpose of the `EventTimeWatermarkExec` operator is to simply extract (*project*) the values of the [event-time watermark column](#) and add them directly to the [EventTimeStatsAccum](#) internal accumulator.

Note

Since the execution (data processing) happens on Spark executors, the only way to establish communication between the tasks (on the executors) and the driver is to use an accumulator.

Read up on [Accumulators](#) in [The Internals of Apache Spark](#) book.

`EventTimeWatermarkExec` uses [EventTimeStatsAccum](#) internal accumulator as a way to send the statistics (the maximum, minimum, average and update count) of the values in the [event-time watermark column](#) that is later used in:

- `ProgressReporter` for [creating execution statistics](#) for the most recent query execution (for monitoring the `max`, `min`, `avg`, and `watermark` event-time watermark statistics)
- `StreamExecution` to observe and possibly update event-time watermark when [constructing the next streaming batch](#).

`EventTimeWatermarkExec` is [created](#) exclusively when [StatefulAggregationStrategy](#) execution planning strategy is requested to plan a logical plan with [EventTimeWatermark](#) logical operators for execution.

Tip

Check out [Demo: Streaming Watermark with Aggregation in Append Output Mode](#) to deep dive into the internals of [Streaming Watermark](#).

Creating EventTimeWatermarkExec Instance

`EventTimeWatermarkExec` takes the following to be created:

- **Event time column** - the column with the (event) time for event-time watermark
- Delay interval (`CalendarInterval`)
- Child physical operator (`SparkPlan`)

While `being created`, `EventTimeWatermarkExec` registers the `EventTimeStatsAccum` internal accumulator (with the current `SparkContext`).

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note	<code>doExecute</code> is part of <code>SparkPlan</code> Contract to generate the runtime representation of an physical operator as a distributed computation over internal binary rows on Apache Spark (i.e. <code>RDD[InternalRow]</code>).
------	--

Internally, `doExecute` executes the `child` physical operator and maps over the partitions (using `RDD.mapPartitions`).

`doExecute` creates an unsafe projection (one per partition) for the `column with the event time` in the output schema of the `child` physical operator. The unsafe projection is to extract event times from the (stream of) internal rows of the child physical operator.

For every row (`InternalRow`) per partition, `doExecute` requests the `eventTimeStats` accumulator to `add the event time`.

Note	The event time value is in seconds (not millis as the value is divided by <code>1000</code>).
------	--

Output Attributes (Schema) — `output` Property

```
output: Seq[Attribute]
```

Note	<code>output</code> is part of the <code>QueryPlan</code> Contract to describe the attributes of the output (aka <i>schema</i>).
------	---

`output` requests the `child` physical operator for the output attributes to find the `event time column` and any other column with metadata that contains `spark.watermarkDelayMs` key.

For the `event time column`, `output` updates the metadata to include the `delay` interval for the `spark.watermarkDelayMs` key.

For any other column (not the event time column) with the `spark.watermarkDelayMs` key, `output` simply removes the key from the metadata.

```
// FIXME: Would be nice to have a demo. Anyone?
```

Internal Properties

Name	Description				
<code>delayMs</code>	<p>Delay interval - the <code>delay</code> interval in milliseconds</p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>EventTimeWatermarkExec</code> is requested for the <code>output</code> attributes • <code>WatermarkTracker</code> is requested to update the event-time watermark 				
<code>eventTimeStats</code>	<p><code>EventTimeStatsAccum</code> accumulator to accumulate <code>eventTime</code> values from every row in a streaming batch (when <code>EventTimeWatermarkExec</code> is executed).</p> <table border="1"> <tr> <td>Note</td><td><code>EventTimeStatsAccum</code> is a Spark accumulator of <code>EventTimeStats</code> from <code>Longs</code> (i.e. <code>AccumulatorV2[Long, EventTimeStats]</code>).</td></tr> <tr> <td>Note</td><td>Every Spark accumulator has to be registered before use, and <code>eventTimeStats</code> is registered when <code>EventTimeWatermarkExec</code> is created.</td></tr> </table>	Note	<code>EventTimeStatsAccum</code> is a Spark accumulator of <code>EventTimeStats</code> from <code>Longs</code> (i.e. <code>AccumulatorV2[Long, EventTimeStats]</code>).	Note	Every Spark accumulator has to be registered before use, and <code>eventTimeStats</code> is registered when <code>EventTimeWatermarkExec</code> is created.
Note	<code>EventTimeStatsAccum</code> is a Spark accumulator of <code>EventTimeStats</code> from <code>Longs</code> (i.e. <code>AccumulatorV2[Long, EventTimeStats]</code>).				
Note	Every Spark accumulator has to be registered before use, and <code>eventTimeStats</code> is registered when <code>EventTimeWatermarkExec</code> is created.				

FlatMapGroupsWithStateExec Unary Physical Operator

`FlatMapGroupsWithStateExec` is a unary physical operator that represents `FlatMapGroupsWithState` logical operator at execution time.

Note

A unary physical operator (`UnaryExecNode`) is a physical operator with a single `child` physical operator.

Read up on [UnaryExecNode](#) (and physical operators in general) in [The Internals of Spark SQL](#) book.

Note

`FlatMapGroupsWithState` unary logical operator represents `KeyValueGroupedDataset.mapGroupsWithState` and `KeyValueGroupedDataset.flatMapGroupsWithState` operators in a logical query plan.

`FlatMapGroupsWithStateExec` is [created](#) exclusively when `FlatMapGroupsWithStateStrategy` execution planning strategy is requested to plan a `FlatMapGroupsWithState` logical operator for execution.

`FlatMapGroupsWithStateExec` is an `ObjectProducerExec` physical operator and so produces a [single output object](#).

Tip

Read up on [ObjectProducerExec—Physical Operators With Single Object Output](#) in [The Internals of Spark SQL](#) book.

Tip

Check out [Demo: Internals of FlatMapGroupsWithStateExec Physical Operator](#).

Note

`FlatMapGroupsWithStateExec` is given an `OutputMode` when created, but it does not seem to be used at all. Check out the question [What's the purpose of OutputMode in flatMapGroupsWithState? How/where is it used?](#) on StackOverflow.

Tip

Enable `ALL` logging level for `org.apache.spark.sql.execution.streaming.FlatMapGroupsWithStateExec` to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.execution.streaming.FlatMapGroupsWithStateExec=
```

Refer to [Logging](#).

Creating FlatMapGroupsWithStateExec Instance

FlatMapGroupsWithStateExec takes the following to be created:

- **User-defined state function** that is applied to every group (of type `(Any, Iterator[Any], LogicalGroupState[Any]) → Iterator[Any]`)
- Key deserializer expression
- Value deserializer expression
- Grouping attributes (as used for grouping in [KeyValueGroupedDataset](#) for `mapGroupsWithState` or `flatMapGroupsWithState` operators)
- Data attributes
- Output object attribute (that is the reference to the single object field this operator outputs)
- [StatefulOperatorStateInfo](#)
- State encoder (`ExpressionEncoder[Any]`)
- State format version
- [OutputMode](#)
- [GroupStateTimeout](#)
- [Batch Processing Time](#)
- [Event-time watermark](#)
- Child physical operator

FlatMapGroupsWithStateExec initializes the [internal properties](#).

Performance Metrics (SQLMetrics)

FlatMapGroupsWithStateExec uses the performance metrics of [StateStoreWriter](#).

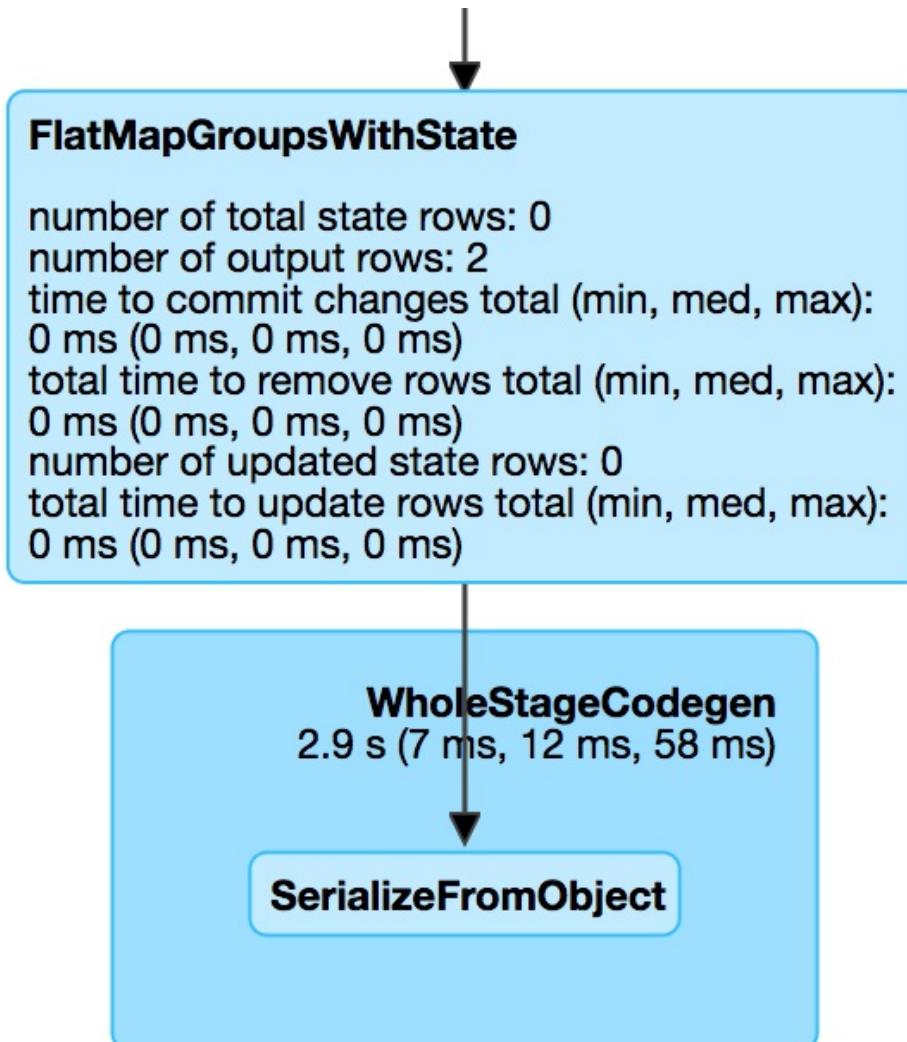


Figure 1. FlatMapGroupsWithStateExec in web UI (Details for Query)

FlatMapGroupsWithStateExec as StateStoreWriter

`FlatMapGroupsWithStateExec` is a [stateful physical operator that can write to a state store](#)(and `MicroBatchExecution` requests [whether to run another batch or not](#) based on the `GroupStateTimeout`).

`FlatMapGroupsWithStateExec` uses the `GroupStateTimeout` (and possibly the updated `metadata`) when asked [whether to run another batch or not](#) (when `MicroBatchExecution` is requested to [construct the next streaming micro-batch](#) when requested to [run the activated streaming query](#)).

FlatMapGroupsWithStateExec with Streaming Event-Time Watermark Support (WatermarkSupport)

`FlatMapGroupsWithStateExec` is a [physical operator that supports streaming event-time watermark](#).

`FlatMapGroupsWithStateExec` is given the [optional event time watermark](#) when created.

The [event-time watermark](#) is initially undefined (`None`) when planned to for execution (in [FlatMapGroupsWithStateStrategy](#) execution planning strategy).

Note

`FlatMapGroupsWithStateStrategy` converts [FlatMapGroupsWithState](#) unary logical operator to `FlatMapGroupsWithStateExec` physical operator with undefined [StatefulOperatorStateInfo](#), [batchTimestampMs](#), and [eventTimeWatermark](#).

The [event-time watermark](#) (with the [StatefulOperatorStateInfo](#) and the [batchTimestampMs](#)) is only defined to the [current event-time watermark](#) of the given [OffsetSeqMetadata](#) when [IncrementalExecution](#) query execution pipeline is requested to apply the [state](#) preparation rule (as part of the [preparations](#) rules).

Note

The [preparations](#) rules are executed (applied to a physical query plan) at the [executedPlan](#) phase of Structured Query Execution Pipeline to generate an optimized physical query plan ready for execution).

Read up on [Structured Query Execution Pipeline](#) in [The Internals of Spark SQL](#) book.

[IncrementalExecution](#) is used as the [lastExecution](#) of the available [streaming query execution engines](#). It is created in the [queryPlanning](#) phase (of the [MicroBatchExecution](#) and [ContinuousExecution](#) execution engines) based on the current [OffsetSeqMetadata](#).

Note

The [optional event-time watermark](#) can only be defined when the [state](#) preparation rule is executed which is at the [executedPlan](#) phase of Structured Query Execution Pipeline which is also part of the [queryPlanning](#) phase.

FlatMapGroupsWithStateExec and StateManager — stateManager Property

`stateManager: StateManager`

While being created, `FlatMapGroupsWithStateExec` creates a [StateManager](#) (with the [state encoder](#) and the [isTimeoutEnabled](#) flag).

A `stateManager` is [created](#) per [state format version](#) that is given while creating a `FlatMapGroupsWithStateExec` (to choose between the [available implementations](#)).

The [state format version](#) is controlled by `spark.sql.streaming.flatMapGroupsWithState.stateFormatVersion` internal configuration property (default: `2`).

Note	<code>StateManagerImplV2</code> is the default <code>StateManager</code> .
------	--

The `StateManager` is used exclusively when `FlatMapGroupsWithStateExec` physical operator is [executed](#) (to generate a recipe for a distributed computation as an `RDD[InternalRow]`) for the following:

- [State schema](#) (for the [value schema](#) of a `StateStoreRDD`)
- State data for a key in a `StateStore` while processing new data
- All state data (for all keys) in a `StateStore` while processing timed-out state data
- Removing the state for a key from a `StateStore` when [all rows have been processed](#)
- Persisting the state for a key in a `StateStore` when [all rows have been processed](#)

keyExpressions Method

```
keyExpressions: Seq[Attribute]
```

Note	<code>keyExpressions</code> is part of the WatermarkSupport Contract to...FIXME.
------	--

`keyExpressions` simply returns the [grouping attributes](#).

Executing Physical Operator (Generating RDD[InternalRow]) — doExecute Method

```
doExecute(): RDD[InternalRow]
```

Note	<code>doExecute</code> is part of <code>SparkPlan</code> Contract to generate the runtime representation of an physical operator as a distributed computation over internal binary rows on Apache Spark (i.e. <code>RDD[InternalRow]</code>).
------	--

`doExecute` first initializes the [metrics](#) (which happens on the driver).

`doExecute` then requests the `child` physical operator to execute and generate an `RDD[InternalRow]`.

`doExecute` uses `StateStoreOps` to create a `StateStoreRDD` with a `storeUpdateFunction` that does the following (for a partition):

1. Creates an `InputProcessor` for a given `StateStore`

2. (only when the `GroupStateTimeout` is `EventTimeTimeout`) Filters out late data based on the `event-time watermark`, i.e. rows from a given `Iterator[InternalRow]` that are older than the `event-time watermark` are excluded from the steps that follow
3. Requests the `InputProcessor` to create an iterator of a new data processed from the (possibly filtered) iterator
4. Requests the `InputProcessor` to create an iterator of a timed-out state data
5. Creates an iterator by concatenating the above iterators (with the new data processed first)
6. In the end, creates a `CompletionIterator` that executes a completion function (`completionFunction`) after it has successfully iterated through all the elements (i.e. when a client has consumed all the rows). The completion method requests the given `StateStore` to commit changes followed by setting the store-specific metrics.

Checking Out Whether Last Batch Execution Requires Another Non-Data Batch or Not

— `shouldRunAnotherBatch` Method

```
shouldRunAnotherBatch(newMetadata: OffsetSeqMetadata): Boolean
```

Note

`shouldRunAnotherBatch` is part of the `StateStoreWriter Contract` to indicate whether `MicroBatchExecution` should run another non-data batch (based on the updated `OffsetSeqMetadata` with the current event-time watermark and the batch timestamp).

`shouldRunAnotherBatch` uses the `GroupStateTimeout` as follows:

- With `EventTimeTimeout`, `shouldRunAnotherBatch` is positive (`true`) only when the `event-time watermark` is defined and is older (below) the `event-time watermark` of the given `OffsetSeqMetadata`
- With `NoTimeout` (and other `GroupStateTimeouts` if there were any), `shouldRunAnotherBatch` is always negative (`false`)
- With `ProcessingTimeTimeout`, `shouldRunAnotherBatch` is always positive (`true`)

Internal Properties

Name	Description
<code>isTimeoutEnabled</code>	<p>Flag that says whether the <code>GroupStateTimeout</code> is not <code>NoTimeout</code></p> <p>Used when:</p> <ul style="list-style-type: none"> • <code>FlatMapGroupsWithStateExec</code> is created (and creates the internal <code>StateManager</code>) • <code>InputProcessor</code> is requested to <code>processTimedOutState</code>
<code>stateAttributes</code>	
<code>stateDeserializer</code>	
<code>stateSerializer</code>	
<code>timestampTimeoutAttribute</code>	
<code>watermarkPresent</code>	<p>Flag that says whether the <code>child</code> physical operator has a <code>watermark attribute</code> (among the output attributes).</p> <p>Used exclusively when <code>InputProcessor</code> is requested to <code>callFunctionAndUpdateState</code></p>

StateStoreRestoreExec Unary Physical Operator — Restoring Streaming State From State Store

`StateStoreRestoreExec` is a unary physical operator that [restores](#) (reads) a streaming state from a [state store](#) (for the keys from the [child](#) physical operator).

Note	<p>A unary physical operator (<code>UnaryExecNode</code>) is a physical operator with a single child physical operator.</p> <p>Read up on UnaryExecNode (and physical operators in general) in The Internals of Spark SQL book.</p>
------	---

`StateStoreRestoreExec` is [created](#) exclusively when [StatefulAggregationStrategy](#) execution planning strategy is requested to plan a [streaming aggregation](#) for execution ([Aggregate](#) logical operators in the logical plan of a streaming query).

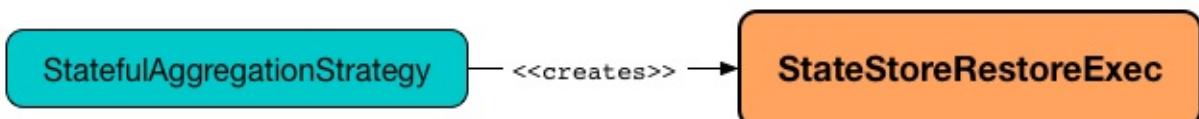


Figure 1. StateStoreRestoreExec and StatefulAggregationStrategy

The optional [StatefulOperatorStateInfo](#) is initially undefined (i.e. when `StateStoreRestoreExec` is [created](#)). `StateStoreRestoreExec` is updated to hold the streaming batch-specific execution property when `IncrementalExecution` [prepares a streaming physical plan for execution](#) (and [state](#) preparation rule is executed when `StreamExecution` [plans a streaming query](#) for a streaming batch).



Figure 2. StateStoreRestoreExec and IncrementalExecution

When [executed](#), `StateStoreRestoreExec` executes the [child](#) physical operator and [creates a StateStoreRDD to map over partitions](#) with `storeUpdateFunction` that restores the state for the keys in the input rows if available.

The output schema of `StateStoreRestoreExec` is exactly the [child](#)'s output schema.

The output partitioning of `StateStoreRestoreExec` is exactly the [child](#)'s output partitioning.

Performance Metrics (SQLMetrics)

Key	Name (in UI)	Description
numOutputRows	number of output rows	The number of input rows from the <code>child</code> physical operator (for which <code>StateStoreRestoreExec</code> tried to find the state)

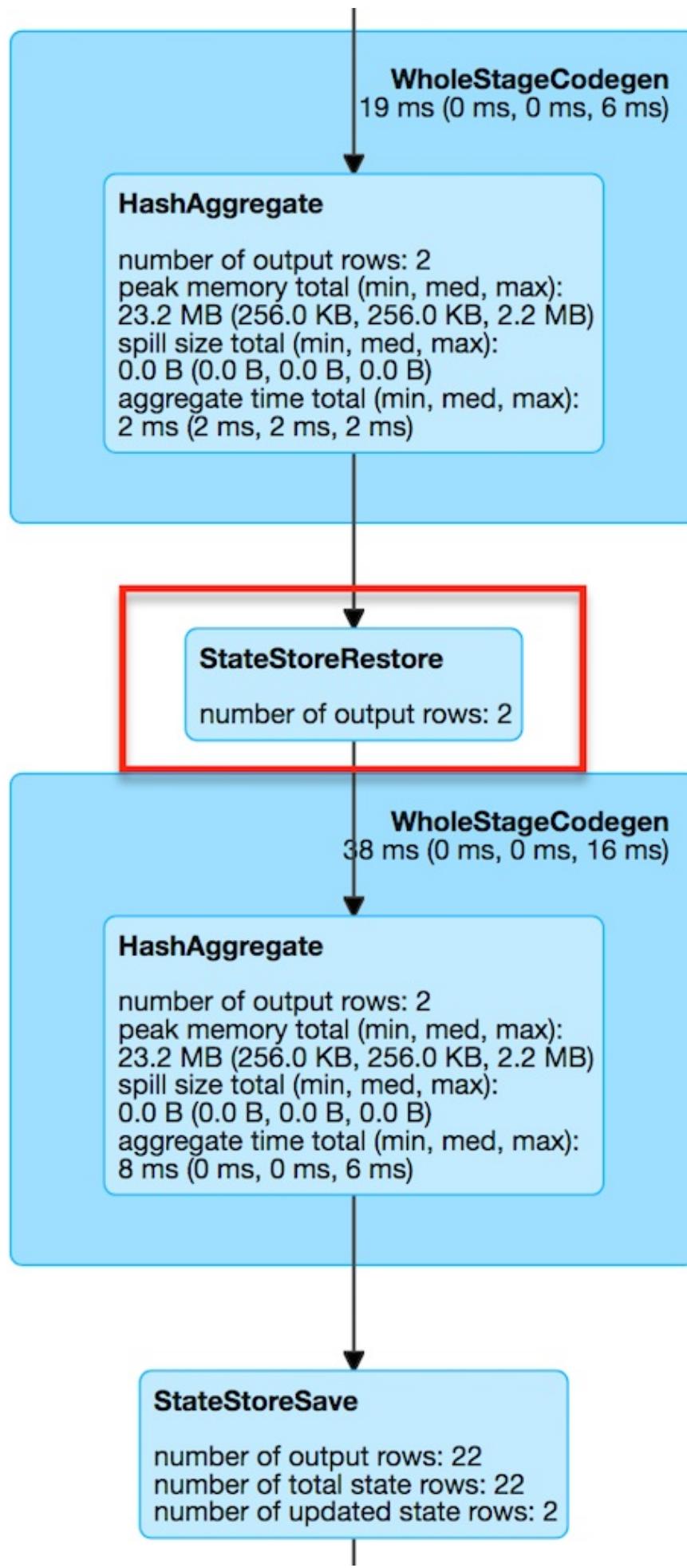


Figure 3. StateStoreRestoreExec in web UI (Details for Query)

Creating StateStoreRestoreExec Instance

`StateStoreRestoreExec` takes the following to be created:

- **Key expressions**, i.e. Catalyst attributes for the grouping keys
- Optional `StatefulOperatorStateInfo` (default: `None`)
- Version of the state format (based on the `spark.sql.streaming.aggregation.stateFormatVersion` configuration property)
- Child physical operator (`SparkPlan`)

StateStoreRestoreExec and StreamingAggregationStateManager — `stateManager` Property

```
stateManager: StreamingAggregationStateManager
```

`stateManager` is a `StreamingAggregationStateManager` that is created together with `StateStoreRestoreExec`.

The `StreamingAggregationStateManager` is created for the `keys`, the output schema of the `child` physical operator and the `version of the state format`.

The `StreamingAggregationStateManager` is used when `StateStoreRestoreExec` is requested to generate a recipe for a distributed computation (as a `RDD[InternalRow]`) for the following:

- Schema of the values in a state store
- Extracting the columns for the key from the input row
- Looking up the value of a key from a state store

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note

`doExecute` is part of `SparkPlan` Contract to generate the runtime representation of an physical operator as a distributed computation over internal binary rows on Apache Spark (i.e. `RDD[InternalRow]`).

Internally, `doExecute` executes `child` physical operator and creates a `StateStoreRDD` with `storeUpdateFunction` that does the following per `child` operator's RDD partition:

1. Generates an unsafe projection to access the key field (using `keyExpressions` and the output schema of `child` operator).
2. For every input row (as `InternalRow`)
 - Extracts the key from the row (using the unsafe projection above)
 - Gets the saved state in `StateStore` for the key if available (it might not be if the key appeared in the input the first time)
 - Increments `numOutputRows` metric (that in the end is the number of rows from the `child` operator)
 - Generates collection made up of the current row and possibly the state for the key if available

Note

The number of rows from `stateStoreRestoreExec` is the number of rows from the `child` operator with additional rows for the saved state.

Note

There is no way in `StateStoreRestoreExec` to find out how many rows had associated state available in a state store. You would have to use the corresponding `StateStoreSaveExec` operator's `metrics` (most likely `number of total state rows` but that could depend on the output mode).

StateStoreSaveExec Unary Physical Operator — Saving Streaming State To State Store

`StateStoreSaveExec` is a unary physical operator that saves a streaming state to a state store with support for streaming watermark.

Note	A unary physical operator (<code>UnaryExecNode</code>) is a physical operator with a single child physical operator. Read up on UnaryExecNode (and physical operators in general) in The Internals of Spark SQL book.
------	--

`StateStoreSaveExec` is created exclusively when `StatefulAggregationStrategy` execution planning strategy is requested to plan a streaming aggregation for execution (Aggregate logical operators in the logical plan of a streaming query).

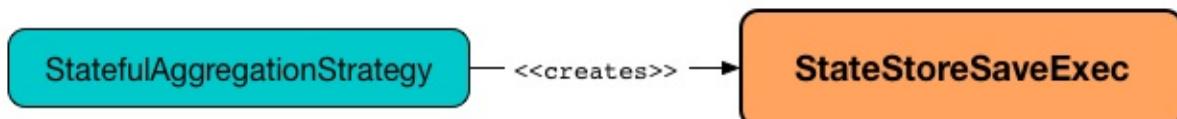


Figure 1. StateStoreSaveExec and StatefulAggregationStrategy

The optional properties, i.e. the `StatefulOperatorStateInfo`, the `output mode`, and the `event-time watermark`, are initially undefined when `StateStoreSaveExec` is created.

`StateStoreSaveExec` is updated to hold execution-specific configuration when `IncrementalExecution` is requested to prepare the logical plan (of a streaming query) for execution (when the state preparation rule is executed).



Figure 2. StateStoreSaveExec and IncrementalExecution

Note	Unlike <code>StateStoreRestoreExec</code> operator, <code>stateStoreSaveExec</code> takes output mode and event time watermark when created.
------	--

When executed, `StateStoreSaveExec` creates a `StateStoreRDD` to map over partitions with `storeUpdateFunction` that manages the `StateStore`.



Figure 3. StateStoreSaveExec creates StateStoreRDD

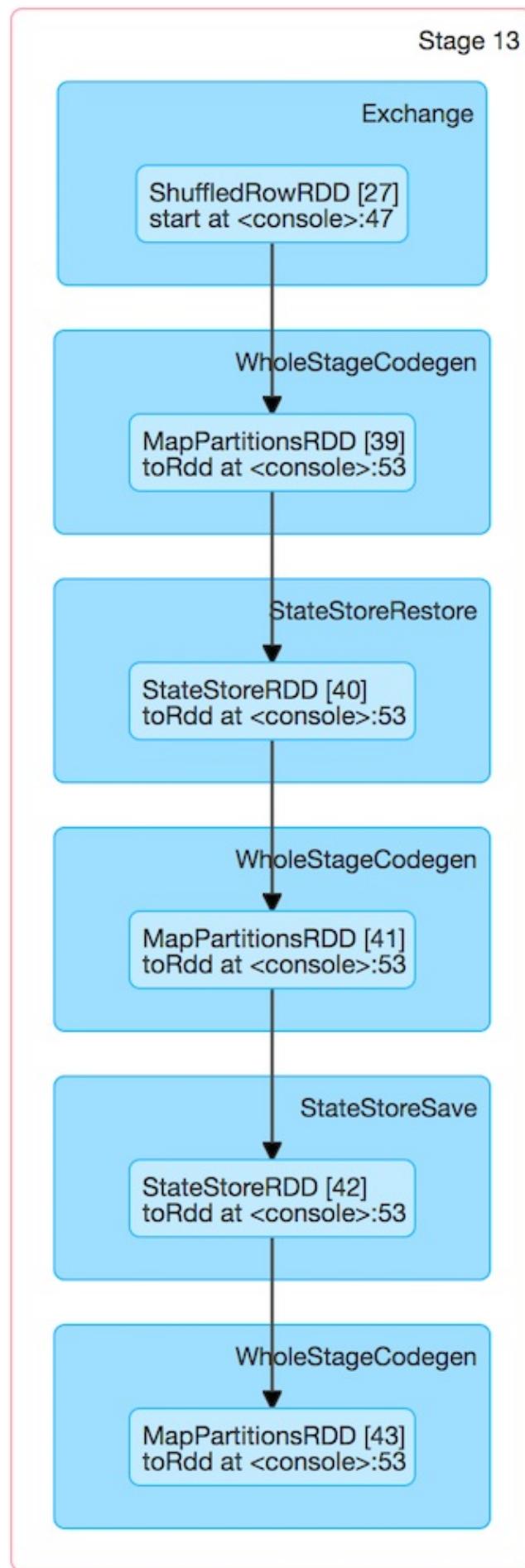


Figure 4. StateStoreSaveExec and StateStoreRDD (after streamingBatch.toRdd.count)

Note	The number of partitions of <code>StateStoreRDD</code> (and hence the number of Spark tasks) is what was defined for the <code>child</code> physical plan. There will be that many <code>stateStores</code> as there are partitions in <code>StateStoreRDD</code> .
------	--

Note	<code>StateStoreSaveExec</code> behaves differently per output mode.
------	--

When `executed`, `StateStoreSaveExec` executes the `child` physical operator and creates a `StateStoreRDD` (with `storeUpdateFunction` specific to the output mode).

The output schema of `StateStoreSaveExec` is exactly the `child`'s output schema.

The output partitioning of `StateStoreSaveExec` is exactly the `child`'s output partitioning.

`StateStoreRestoreExec` uses a `StreamingAggregationStateManager` (that is created for the `keyExpressions`, the output of the `child` physical operator and the `stateFormatVersion`).

Tip	<p>Enable <code>ALL</code> logging level for <code>org.apache.spark.sql.execution.streaming.StateStoreSaveExec</code> to see what happens inside.</p> <p>Add the following line to <code>conf/log4j.properties</code> :</p> <pre>log4j.logger.org.apache.spark.sql.execution.streaming.StateStoreSaveExec=ALL</pre> <p>Refer to Logging.</p>
-----	---

Performance Metrics (SQLMetrics)

`StateStoreSaveExec` uses the performance metrics of the parent `StateStoreWriter`.

Key	Name (in UI)	Description				
allUpdatesTimeMs						
allRemovalsTimeMs						
commitTimeMs						
numOutputRows						
numTotalStateRows		<p>Number of the state keys in the state store</p> <p>Corresponds to <code>numRowsTotal</code> in <code>stateOperators</code> in StreamingQueryProgress (and is available as <code>sq.lastProgress.stateOperators(0).numRowsTotal</code> for <code>0</code>)</p>				
numUpdatedStateRows		<p>Number of the state keys that were stored as updates in the trigger and for the keys in the result rows of the upstream operator.</p> <ul style="list-style-type: none"> In Complete output mode, <code>numUpdatedStateRows</code> is the number of output rows (which should be exactly the number of output rows from the upstream operator) <table border="1"> <tr> <td>Caution</td> <td>FIXME</td> </tr> </table> <ul style="list-style-type: none"> In Append output mode, <code>numUpdatedStateRows</code> is the number of output rows with keys that have not expired yet (per requirement). In Update output mode, <code>numUpdatedStateRows</code> is equal to the number of output rows, i.e. the number of keys that have not been updated since the watermark has been defined at all (which is optional). <table border="1"> <tr> <td>Caution</td> <td>FIXME</td> </tr> </table> <p>Note</p> <p>You can see the current value as <code>numRowsUpdated</code> in <code>stateOperators</code> in StreamingQueryProgress as <code>StreamingQuery.lastProgress.stateOperators(n).numRowsUpdated</code> for <code>n</code> th operator).</p>	Caution	FIXME	Caution	FIXME
Caution	FIXME					
Caution	FIXME					
stateMemory		Memory used by the StateStore				

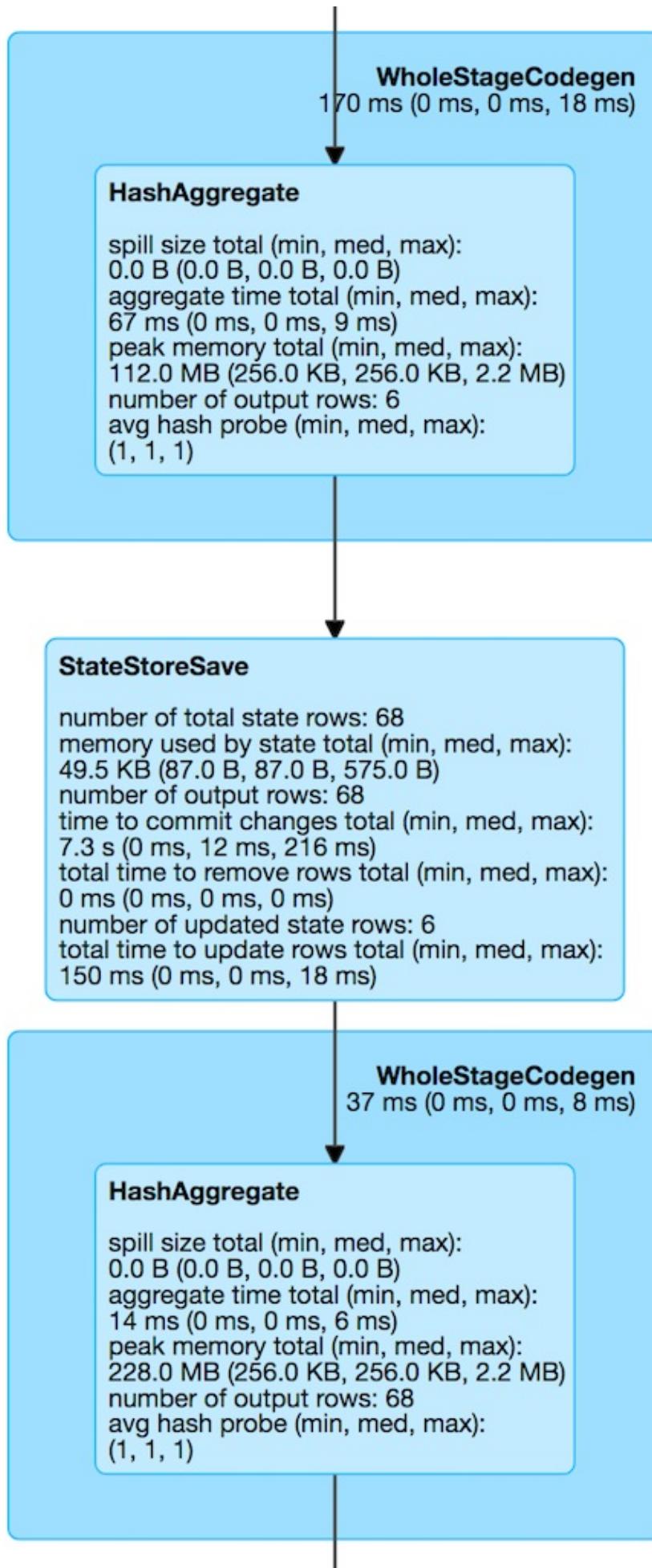


Figure 5. StateStoreSaveExec in web UI (Details for Query)

Creating StateStoreSaveExec Instance

`StateStoreSaveExec` takes the following to be created:

- **Key expressions**, i.e. Catalyst attributes for the grouping keys
- Execution-specific `StatefulOperatorStateInfo` (default: `None`)
- Execution-specific `output mode` (default: `None`)
- `Event-time watermark` (default: `None`)
- Version of the state format (based on the `spark.sql.streaming.aggregation.stateFormatVersion` configuration property)
- Child physical operator (`SparkPlan`)

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note	<code>doExecute</code> is part of <code>SparkPlan</code> Contract to generate the runtime representation of an physical operator as a distributed computation over internal binary rows on Apache Spark (i.e. <code>RDD[InternalRow]</code>).
------	--

Internally, `doExecute` initializes `metrics`.

Note	<code>doExecute</code> requires that the optional <code>outputMode</code> is at this point defined (that should have happened when <code>IncrementalExecution</code> had prepared a streaming aggregation for execution).
------	---

`doExecute` executes `child` physical operator and creates a `StateStoreRDD` with `storeUpdateFunction` that:

1. Generates an unsafe projection to access the key field (using `keyExpressions` and the output schema of `child`).
2. Branches off per `output mode`.

Table 1. `doExecute`'s Behaviour per Output Mode

Output Mode	<code>doExecute</code> 's Behaviour

Note

`Append` is the default output mode when not specified explicitly.

Note

`Append` output mode requires that a streaming query defines event time watermark (using `withWatermark` operator) on the event time column that is used in aggregation (directly or using `window` function).

1. Finds late (aggregate) rows from `child` physical operator (that have expired per `watermark`)
2. Stores the late rows in the state store (and increments `numUpdatedStateRows` metric)
3. Gets all the added (late) rows from the state store
4. Creates an iterator that removes the late rows from the state store when requested the next row and in the end commits the state updates

Note

`numUpdatedStateRows` metric is the number of rows that...FIXME

Tip

Refer to [Demo: StateStoreSaveExec with Append Output Mode](#) for an example of `StateStoreSaveExec` in `Append` output mode.

Append

Caution

FIXME When is "Filtering state store on:" printed out?

Caution

FIXME Track `numUpdatedStateRows` metric

-
1. Uses `watermarkPredicateForData` predicate to exclude matching rows and (like in `Complete` output mode) stores all the remaining rows in `stateStore`.
 2. (like in `Complete` output mode) While storing the rows, increments `numUpdatedStateRows` metric (for every row) and records the total time in `allUpdatesTimeMs` metric.
 3. Takes all the rows from `stateStore` and returns a `NextIterator` that:
 - In `getNext`, finds the first row that matches `watermarkPredicateForKeys` predicate, removes it from `stateStore`, and returns it back.

	<p>If no row was found, <code>getNext</code> also marks the iterator as finished.</p> <ul style="list-style-type: none"> ◦ In <code>close</code>, records the time to iterate over all the rows in <code>allRemovalsTimeMs</code> metric, commits the updates to <code>StateStore</code> followed by recording the time in <code>commitTimeMs</code> metric and recording StateStore metrics. 				
	<ol style="list-style-type: none"> 1. Takes all <code>unsafeRow</code> rows (from the parent iterator) 2. Stores the rows by key in the state store eagerly (i.e. all rows that are available in the parent iterator before proceeding) 3. Commits the state updates 4. In the end, <code>doExecute</code> reads the key-row pairs from the state store and passes the rows along (i.e. to the following physical operator) <p>The number of keys stored in the state store is recorded in <code>numUpdatedStateRows</code> metric.</p>				
Complete	<table border="1"> <tr> <td>Note</td><td>In <code>Complete</code> output mode <code>numOutputRows</code> metric is exactly <code>numTotalStateRows</code> metric.</td></tr> <tr> <td>Tip</td><td>Refer to Demo: StateStoreSaveExec with Complete Output Mode for an example of <code>StateStoreSaveExec</code> in <code>Complete</code> output mode.</td></tr> </table>	Note	In <code>Complete</code> output mode <code>numOutputRows</code> metric is exactly <code>numTotalStateRows</code> metric.	Tip	Refer to Demo: StateStoreSaveExec with Complete Output Mode for an example of <code>StateStoreSaveExec</code> in <code>Complete</code> output mode.
Note	In <code>Complete</code> output mode <code>numOutputRows</code> metric is exactly <code>numTotalStateRows</code> metric.				
Tip	Refer to Demo: StateStoreSaveExec with Complete Output Mode for an example of <code>StateStoreSaveExec</code> in <code>Complete</code> output mode.				
	<ol style="list-style-type: none"> 1. Stores all rows (as <code>UnsafeRow</code>) in <code>stateStore</code>. 2. While storing the rows, increments <code>numUpdatedStateRows</code> metric (for every row) and records the total time in <code>allUpdatesTimeMs</code> metric. 3. Records <code>0</code> in <code>allRemovalsTimeMs</code> metric. 4. Commits the state updates to <code>stateStore</code> and records the time in <code>commitTimeMs</code> metric. 5. Records StateStore metrics. 6. In the end, takes all the rows stored in <code>stateStore</code> and increments <code>numOutputRows</code> metric. 				
	Returns an iterator that filters out late aggregate rows (per <code>watermark</code> if defined) and stores the "young" rows in the state store (one by one, i.e. every <code>next</code>). With no more rows available, that removes the late rows from the state store (all at once) and commits the state updates .				

Update	<p>Tip</p> <p>Refer to Demo: StateStoreSaveExec with Update Output Mode for an example of <code>StateStoreSaveExec</code> in <code>Update</code> output mode.</p>
<p>Returns <code>Iterator</code> of rows that uses <code>watermarkPredicateForData</code> predicate to filter out late rows.</p> <p>In <code>hasNext</code>, when rows are no longer available:</p> <ol style="list-style-type: none"> 1. Records the total time to iterate over all the rows in <code>allUpdatesTimeMs</code> metric. 2. <code>removeKeysOlderThanWatermark</code> and records the time in <code>allRemovalsTimeMs</code> metric. 3. <code>Commits the updates</code> to <code>StateStore</code> and records the time in <code>commitTimeMs</code> metric. 4. Records <code>StateStore metrics</code>. <p>In <code>next</code>, stores a row in <code>StateStore</code> and increments <code>numOutputRows</code> and <code>numUpdatedStateRows</code> metrics.</p>	

`doExecute` reports a `UnsupportedOperationException` when executed with an invalid output mode.

Invalid output mode: [outputMode]

Checking Out Whether Last Batch Execution Requires Another Non-Data Batch or Not

— `shouldRunAnotherBatch` Method

```
shouldRunAnotherBatch(newMetadata: OffsetSeqMetadata): Boolean
```

Note	<p><code>shouldRunAnotherBatch</code> is part of the StateStoreWriter Contract to indicate whether MicroBatchExecution should run another non-data batch (based on the updated OffsetSeqMetadata with the current event-time watermark and the batch timestamp).</p>
---	--

`shouldRunAnotherBatch` is positive (`true`) when all of the following are met:

- `OutputMode` is either [Append](#) or [Update](#)
- [Event-time watermark](#) is defined and is older (below) the [event-time watermark](#) of the given `offsetSeqMetadata`

Otherwise, `shouldRunAnotherBatch` is negative (`false`).

StreamingDeduplicateExec Unary Physical Operator for Streaming Deduplication

`StreamingDeduplicateExec` is a unary physical operator that writes state to `StateStore` with support for streaming watermark.

Note	A unary physical operator (<code>UnaryExecNode</code>) is a physical operator with a single child physical operator. Read up on UnaryExecNode (and physical operators in general) in The Internals of Spark SQL book .
------	---

`StreamingDeduplicateExec` is created exclusively when `StreamingDeduplicationStrategy` plans Deduplicate unary logical operators.

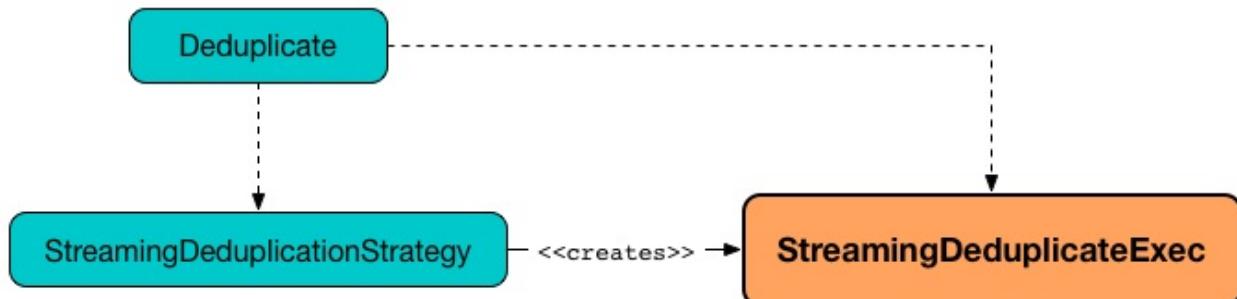


Figure 1. `StreamingDeduplicateExec` and `StreamingDeduplicationStrategy`

```

val uniqueValues = spark.
  readStream.
  format("rate").
  load.
  dropDuplicates("value") // <-- creates Deduplicate logical operator

scala> println(uniqueValues.queryExecution.logical.numberedTreeString)
00 Deduplicate [value#214L], true
01 +- StreamingRelation DataSource(org.apache.spark.sql.SparkSession@4785f176, rate, List(),
(), None, List(), None, Map(), None), rate, [timestamp#213, value#214L]

scala> uniqueValues.explain
== Physical Plan ==
StreamingDeduplicate [value#214L], StatefulOperatorStateInfo(<unknown>, 5a65879c-67bc-4
e77-b417-6100db6a52a2, 0, 0), 0
+- Exchange hashpartitioning(value#214L, 200)
  +- StreamingRelation rate, [timestamp#213, value#214L]

// Start the query and hence StreamingDeduplicateExec
import scala.concurrent.duration._
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
val sq = uniqueValues.
  
```

```
writeStream.  
  format("console").  
  option("truncate", false).  
  trigger(Trigger.ProcessingTime(10.seconds)).  
  outputMode(OutputMode.Update).  
  start  
  
// sorting not supported for non-aggregate queries  
// and so values are unsorted  
  
-----  
Batch: 0  
-----  
+---+---+  
|timestamp|value|  
+---+---+  
+---+---+  
  
-----  
Batch: 1  
-----  
+---+---+  
|timestamp | value |  
+---+---+  
| 2017-07-25 22:12:03.018 | 0 |  
| 2017-07-25 22:12:08.018 | 5 |  
| 2017-07-25 22:12:04.018 | 1 |  
| 2017-07-25 22:12:06.018 | 3 |  
| 2017-07-25 22:12:05.018 | 2 |  
| 2017-07-25 22:12:07.018 | 4 |  
+---+---+  
  
-----  
Batch: 2  
-----  
+---+---+  
|timestamp | value |  
+---+---+  
| 2017-07-25 22:12:10.018 | 7 |  
| 2017-07-25 22:12:09.018 | 6 |  
| 2017-07-25 22:12:12.018 | 9 |  
| 2017-07-25 22:12:13.018 | 10 |  
| 2017-07-25 22:12:15.018 | 12 |  
| 2017-07-25 22:12:11.018 | 8 |  
| 2017-07-25 22:12:14.018 | 11 |  
| 2017-07-25 22:12:16.018 | 13 |  
| 2017-07-25 22:12:17.018 | 14 |  
| 2017-07-25 22:12:18.018 | 15 |  
+---+---+  
  
// Eventually...  
sq.stop
```

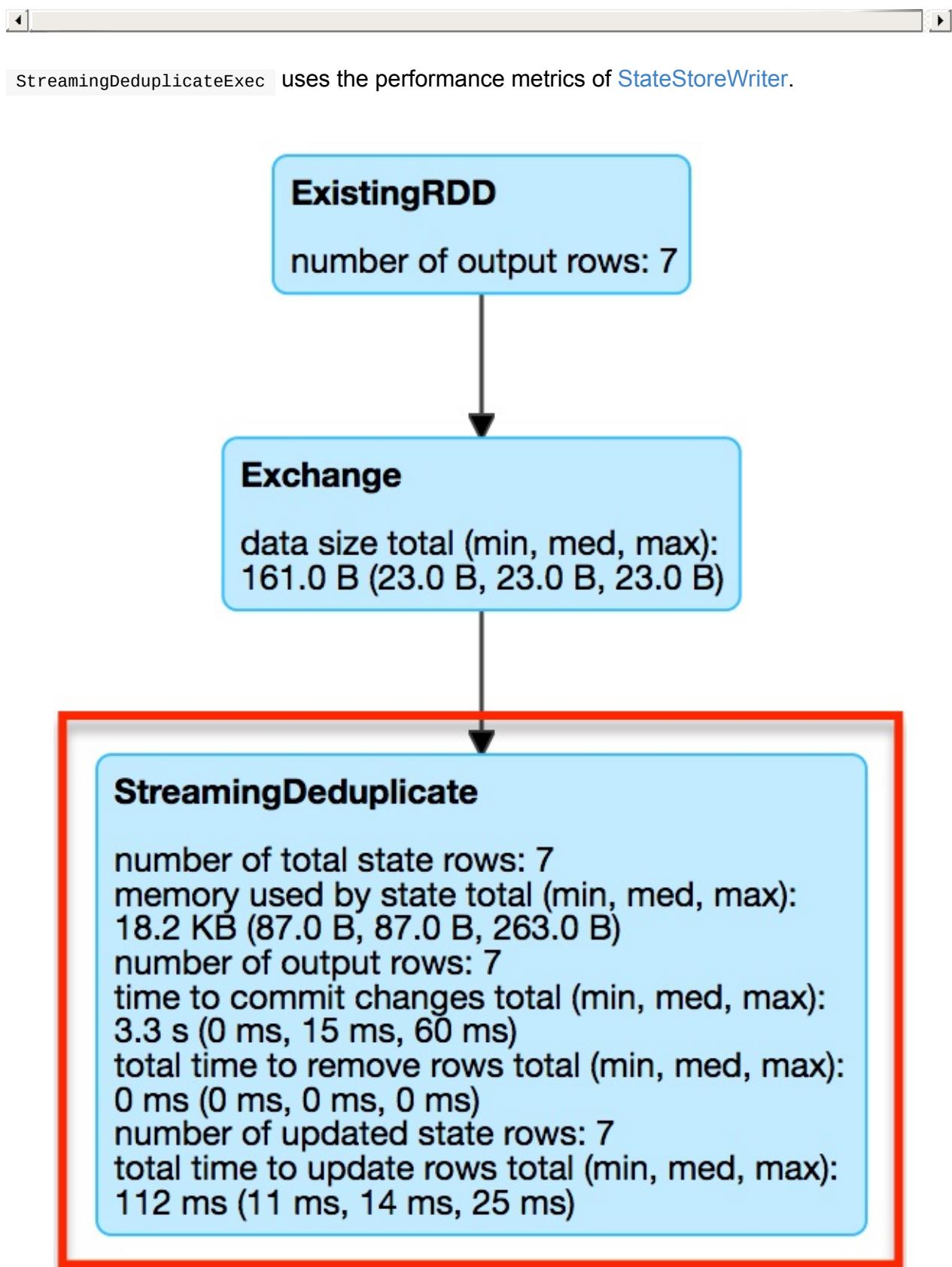


Figure 2. StreamingDeduplicateExec in web UI (Details for Query)

The output schema of `StreamingDeduplicateExec` is exactly the `child`'s output schema.

The output partitioning of `StreamingDeduplicateExec` is exactly the `child`'s output partitioning.

```
/*
// Start spark-shell with debugging and Kafka support
```

```

SPARK_SUBMIT_OPTS="-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=500
5" \
./bin/spark-shell \
--packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0-SNAPSHOT
*/
// Reading
val topic1 = spark.
readStream.
format("kafka").
option("subscribe", "topic1").
option("kafka.bootstrap.servers", "localhost:9092").
option("startingOffsets", "earliest").
load

// Processing with deduplication
// Don't use watermark
// The following won't work due to https://issues.apache.org/jira/browse/SPARK-21546
/**
val records = topic1.
  withColumn("eventtime", 'timestamp). // <-- just to put the right name given the pu
rpose
  withWatermark(eventTime = "eventtime", delayThreshold = "30 seconds"). // <-- use th
e renamed eventtime column
  dropDuplicates("value"). // dropDuplicates will use watermark
                            // only when eventTime column exists
  // include the watermark column => internal design leak?
  select('key cast "string", 'value cast "string", 'eventtime).
  as[(String, String, java.sql.Timestamp)]
*/

```

```

val records = topic1.
  dropDuplicates("value").
  select('key cast "string", 'value cast "string").
  as[(String, String)]

```

```

scala> records.explain
== Physical Plan ==
*Project [cast(key#0 as string) AS key#249, cast(value#1 as string) AS value#250]
+- StreamingDeduplicate [value#1], StatefulOperatorStateInfo(<unknown>,68198b93-6184-49
ae-8098-006c32cc6192,0,0), 0
  +- Exchange hashpartitioning(value#1, 200)
    +- *Project [key#0, value#1]
      +- StreamingRelation kafka, [key#0, value#1, topic#2, partition#3, offset#4L,
        timestamp#5, timestampType#6]

// Writing
import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration.-
val sq = records.
writeStream.
format("console").
option("truncate", false).
trigger(Trigger.ProcessingTime(10.seconds)).

```

```
queryName("from-kafka-topic1-to-console").
outputMode(OutputMode.Update).
start

// Eventually...
sq.stop
```

Tip Enable `INFO` logging level for `org.apache.spark.sql.execution.streaming.StreamingDeduplicateExec` to see what happens inside.

Add the following line to `conf/log4j.properties`:

```
log4j.logger.org.apache.spark.sql.execution.streaming.StreamingDeduplicateExec=INFO
```

Refer to [Logging](#).

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

`doExecute(): RDD[InternalRow]`

Note `doExecute` is part of `SparkPlan` Contract to generate the runtime representation of an physical operator as a distributed computation over internal binary rows on Apache Spark (i.e. `RDD[InternalRow]`).

Internally, `doExecute` initializes [metrics](#).

`doExecute` executes `child` physical operator and creates a `StateStoreRDD` with `storeUpdateFunction` that:

1. Generates an unsafe projection to access the key field (using `keyExpressions` and the output schema of `child`).
2. Filters out rows from `Iterator[InternalRow]` that match `watermarkPredicateForData` (when defined and `timeoutConf` is `EventTimeTimeout`)
3. For every row (as `InternalRow`)
 - Extracts the key from the row (using the unsafe projection above)
 - Gets the saved state in `StateStore` for the key
 - (when there was a state for the key in the row) Filters out (aka *drops*) the row

- (when there was *no* state for the key in the row) Stores a new (and empty) state for the key and increments `numUpdatedStateRows` and `numOutputRows` metrics.
4. In the end, `storeUpdateFunction` creates a `CompletionIterator` that executes a completion function (aka `completionFunction`) after it has successfully iterated through all the elements (i.e. when a client has consumed all the rows).

The completion function does the following:

- Updates `allUpdatesTimeMs` metric (that is the total time to execute `storeUpdateFunction`)
- Updates `allRemovalsTimeMs` metric with the time taken to remove keys older than the watermark from the StateStore
- Updates `commitTimeMs` metric with the time taken to commit the changes to the StateStore
- Sets StateStore-specific metrics

Creating StreamingDeduplicateExec Instance

`StreamingDeduplicateExec` takes the following when created:

- Duplicate keys (as used in `dropDuplicates` operator)
- Child physical operator (`SparkPlan`)
- `StatefulOperatorStateInfo`
- Event-time watermark

Checking Out Whether Last Batch Execution Requires Another Non-Data Batch or Not — `shouldRunAnotherBatch` Method

```
shouldRunAnotherBatch(newMetadata: OffsetSeqMetadata): Boolean
```

Note	<code>shouldRunAnotherBatch</code> is part of the <code>StateStoreWriter Contract</code> to indicate whether <code>MicroBatchExecution</code> should run another non-data batch (based on the updated <code>OffsetSeqMetadata</code> with the current event-time watermark and the batch timestamp).
------	--

`shouldRunAnotherBatch` ...FIXME

StreamingGlobalLimitExec Unary Physical Operator

`StreamingGlobalLimitExec` is a unary physical operator that represents a `Limit` logical operator of a streaming query at execution time.

Note

A unary physical operator (`UnaryExecNode`) is a physical operator with a single `child` physical operator.

Read up on [UnaryExecNode](#) (and physical operators in general) in [The Internals of Spark SQL](#) book.

`StreamingGlobalLimitExec` is created exclusively when [StreamingGlobalLimitStrategy](#) execution planning strategy is requested to plan a `Limit` logical operator (in the logical plan of a streaming query) for execution.

Note

`Limit` logical operator represents `Dataset.limit` operator in a logical query plan.

Read up on [Limit Logical Operator](#) in [The Internals of Spark SQL](#) book.

`StreamingGlobalLimitExec` is a [stateful physical operator](#) that can write to a state store.

`StreamingGlobalLimitExec` supports [Append](#) output mode only.

The optional properties, i.e. the [StatefulOperatorStateInfo](#) and the [output mode](#), are initially undefined when `StreamingGlobalLimitExec` is created. `StreamingGlobalLimitExec` is updated to hold execution-specific configuration when `IncrementalExecution` is requested to [prepare the logical plan \(of a streaming query\) for execution](#) (when the [state preparation rule](#) is executed).

Creating StreamingGlobalLimitExec Instance

`StreamingGlobalLimitExec` takes the following to be created:

- **Streaming Limit**
- Child physical operator (`SparkPlan`)
- [StatefulOperatorStateInfo](#) (default: `None`)
- [OutputMode](#) (default: `None`)

`StreamingGlobalLimitExec` initializes the [internal properties](#).

StreamingGlobalLimitExec as StateStoreWriter

`StreamingGlobalLimitExec` is a [stateful physical operator](#) that can write to a state store.

Performance Metrics

`StreamingGlobalLimitExec` uses the performance metrics of the parent [StateStoreWriter](#).

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note	<code>doExecute</code> is part of <code>SparkPlan</code> Contract to generate the runtime representation of an physical operator as a recipe for distributed computation over internal binary rows on Apache Spark (<code>RDD[InternalRow]</code>).
------	---

`doExecute` ...FIXME

Internal Properties

Name	Description
<code>keySchema</code>	FIXME Used when...FIXME
<code>valueSchema</code>	FIXME Used when...FIXME

StreamingRelationExec Leaf Physical Operator

StreamingRelationExec is a leaf physical operator (i.e. LeafExecNode) that...FIXME

StreamingRelationExec is created when StreamingRelationStrategy plans StreamingRelation and StreamingExecutionRelation logical operators.

```
scala> spark.version
res0: String = 2.3.0-SNAPSHOT

val rates = spark.
  readStream.
  format("rate").
  load

// StreamingRelation logical operator
scala> println(rates.queryExecution.logical.numberedTreeString)
00 StreamingRelation DataSource(org.apache.spark.sql.SparkSession@31ba0af0,rate,List(),
None,List(),None,Map(),None), rate, [timestamp#0, value#1L]

// StreamingRelationExec physical operator (shown without "Exec" suffix)
scala> rates.explain
== Physical Plan ==
StreamingRelation rate, [timestamp#0, value#1L]
```

StreamingRelationExec is not supposed to be executed and is used...FIXME

Creating StreamingRelationExec Instance

StreamingRelationExec takes the following when created:

- The name of a streaming data source
- Output attributes

StreamingSymmetricHashJoinExec Binary Physical Operator — Stream-Stream Joins

`StreamingSymmetricHashJoinExec` is a binary physical operator that represents a `Join` logical operator of two streaming queries at execution time.

Note

A binary physical operator (`BinaryExecNode`) is a physical operator with `left` and `right` child physical operators.

Read up on [BinaryExecNode](#) (and physical operators in general) in [The Internals of Spark SQL](#) book.

Note

`Join` logical operator represents `Dataset.join` operator in a logical query plan.

Read up on [Join Logical Operator](#) in [The Internals of Spark SQL](#) book.

`StreamingSymmetricHashJoinExec` requires that the `join type` be `Inner`, `LeftOuter`, or `RightOuter` with the same data types of the `left` and the `right` keys.

`StreamingSymmetricHashJoinExec` is created exclusively when `StreamingJoinStrategy` execution planning strategy is requested to plan a logical query plan with a `Join` logical operator of two streaming queries with equality predicates (`EqualTo` and `EqualNullSafe`)

`StreamingSymmetricHashJoinExec` is a stateful physical operator that writes to a state store.

The output schema of `StreamingSymmetricHashJoinExec` is...FIXME

The output partitioning of `StreamingSymmetricHashJoinExec` is...FIXME

Creating StreamingSymmetricHashJoinExec Instance

`StreamingSymmetricHashJoinExec` takes the following to be created:

- Catalyst expressions of the keys on the left side
- Catalyst expressions of the keys on the right side
- `JoinType`
- Join condition (`JoinConditionSplitPredicates`)
- `StatefulOperatorStateInfo`
- Event-time watermark

- State watermark (`JoinStateWatermarkPredicates`)
- Physical operator on the left side (`SparkPlan`)
- Physical operator on the right side (`SparkPlan`)

`StreamingSymmetricHashJoinExec` initializes the [internal properties](#).

Performance Metrics

`StreamingSymmetricHashJoinExec` uses the performance metrics of [StateStoreWriter](#).

Checking Out Whether Last Batch Execution Requires Another Non-Data Batch or Not — `shouldRunAnotherBatch` Method

```
shouldRunAnotherBatch(newMetadata: OffsetSeqMetadata): Boolean
```

Note

`shouldRunAnotherBatch` is part of the [StateStoreWriter Contract](#) to indicate whether [MicroBatchExecution](#) should run another non-data batch (based on the updated [OffsetSeqMetadata](#) with the current event-time watermark and the batch timestamp).

`shouldRunAnotherBatch` ...FIXME

Executing Physical Operator (Generating RDD[InternalRow]) — `doExecute` Method

```
doExecute(): RDD[InternalRow]
```

Note

`doExecute` is part of `SparkPlan` Contract to generate the runtime representation of an physical operator as a recipe for distributed computation over internal binary rows on Apache Spark (`RDD[InternalRow]`).

`doExecute` first requests the `StreamingQueryManager` for the [StateStoreCoordinatorRef](#) to the `StateStoreCoordinator` RPC endpoint (for the driver).

`doExecute` then requests the `SymmetricHashJoinStateManager` for the [names of the state stores](#) for the left and right side of the streaming join.

In the end, `doExecute` requests the `left` and `right` physical operators to execute (generate an RDD) and then `stateStoreAwareZipPartitions` with `processPartitions` (and with the `StateStoreCoordinatorRef` and the state stores).

Processing Partitions — `processPartitions` Internal Method

```
processPartitions(
    leftInputIter: Iterator[InternalRow],
    rightInputIter: Iterator[InternalRow]): Iterator[InternalRow]
```

`processPartitions` ...FIXME

Note

`processPartitions` is used exclusively when `StreamingSymmetricHashJoinExec` physical operator is requested to [execute](#).

Internal Properties

Name	Description
<code>hadoopConfBcast</code>	Hadoop Configuration broadcast (to the Spark cluster) Used exclusively to create a SymmetricHashJoinStateManager
<code>joinStateManager</code>	SymmetricHashJoinStateManager Used when <code>OneSideHashJoiner</code> is requested to <code>storeAndJoinWithOtherSide</code> , <code>removeOldState</code> , <code>commitStateAndGetMetrics</code> , and for the values for a given key
<code>nullLeft</code>	<code>GenericInternalRow</code> of the size of the output schema of the left physical operator
<code>nullRight</code>	<code>GenericInternalRow</code> of the size of the output schema of the right physical operator
<code>storeConf</code>	StateStoreConf Used exclusively to create a SymmetricHashJoinStateManager

FlatMapGroupsWithStateStrategy Execution Planning Strategy for FlatMapGroupsWithState Logical Operator

`FlatMapGroupsWithStateStrategy` is an execution planning strategy that can plan streaming queries with `FlatMapGroupsWithState` unary logical operators to `FlatMapGroupsWithStateExec` physical operator (with undefined `StatefulOperatorStateInfo`, `batchTimestampMs`, and `eventTimeWatermark`).

Tip Read up on [Execution Planning Strategies](#) in [The Internals of Spark SQL](#) book.

`FlatMapGroupsWithStateStrategy` is used exclusively when `IncrementalExecution` is requested to plan a streaming query.

Demo: Using FlatMapGroupsWithStateStrategy

```

import org.apache.spark.sql.streaming.GroupState
val stateFunc = (key: Long, values: Iterator[(Timestamp, Long)], state: GroupState[Long]) => {
  Iterator((key, values.size))
}
import java.sql.Timestamp
import org.apache.spark.sql.streaming.{GroupStateTimeout, OutputMode}
val numGroups = spark.
  readStream.
  format("rate").
  load.
  as[(Timestamp, Long)].
  groupByKey { case (time, value) => value % 2 }.
  flatMapGroupsWithState(OutputMode.Update, GroupStateTimeout.NoTimeout)(stateFunc)

scala> numGroups.explain(true)
== Parsed Logical Plan ==
'SerializeFromObject [assertnonnull(assertnonnull(input[0, scala.Tuple2, true]))._1 AS _1#267L, assertnonnull(assertnonnull(input[0, scala.Tuple2, true]))._2 AS _2#268]
+- 'FlatMapGroupsWithState <function3>, unresolveddeserializer(upcast(getcolumnbyordinal(0, LongType), LongType, - root class: "scala.Long"), value#262L), unresolveddeserializer(newInstance(class scala.Tuple2), timestamp#253, value#254L), [value#262L], [timestamp#253, value#254L], obj#266: scala.Tuple2, class[value[0]: bigint], Update, false, NoTimeout
  +- AppendColumns <function1>, class scala.Tuple2, [StructField(_1, TimestampType, true), StructField(_2, LongType, false)], newInstance(class scala.Tuple2), [input[0, bigint, false] AS value#262L]
    +- StreamingRelation DataSource(org.apache.spark.sql.SparkSession@38bcac50, rate, List(), None, List(), None, Map(), None), rate, [timestamp#253, value#254L]

...
== Physical Plan ==
*SerializeFromObject [assertnonnull(input[0, scala.Tuple2, true])._1 AS _1#267L, assertnonnull(input[0, scala.Tuple2, true])._2 AS _2#268]
+- FlatMapGroupsWithState <function3>, value#262: bigint, newInstance(class scala.Tuple2), [value#262L], [timestamp#253, value#254L], obj#266: scala.Tuple2, StatefulOperatorStateInfo(<unknown>, 84b5dccb-3fa6-4343-a99c-6fa5490c9b33, 0, 0), class[value[0]: bigint], Update, NoTimeout, 0, 0
  +- *Sort [value#262L ASC NULLS FIRST], false, 0
    +- Exchange hashpartitioning(value#262L, 200)
      +- AppendColumns <function1>, newInstance(class scala.Tuple2), [input[0, bigint, false] AS value#262L]
        +- StreamingRelation rate, [timestamp#253, value#254L]

```

StatefulAggregationStrategy Execution Planning Strategy — EventTimeWatermark and Aggregate Logical Operators

`StatefulAggregationStrategy` is an execution planning strategy that is used to plan streaming queries with the two logical operators:

- `EventTimeWatermark` logical operator (for `Dataset.withWatermark` operator)
- `Aggregate` logical operator (for `Dataset.groupBy` and `Dataset.groupByKey` operators, and `GROUP BY` SQL clause)

Tip Read up on [Execution Planning Strategies](#) in [The Internals of Spark SQL](#) book.

`StatefulAggregationStrategy` is used exclusively when `IncrementalExecution` is requested to plan a streaming query.

`StatefulAggregationStrategy` is available using `SessionState`.

```
spark.sessionState.planner.StatefulAggregationStrategy
```

Table 1. StatefulAggregationStrategy's Logical to Physical Operator Conversions

Logical Operator	Physical Operator
<code>EventTimeWatermark</code>	<code>EventTimeWatermarkExec</code>
<code>Aggregate</code>	<p>In the order of preference:</p> <ol style="list-style-type: none"> 1. <code>HashAggregateExec</code> 2. <code>ObjectHashAggregateExec</code> 3. <code>SortAggregateExec</code>
	<p>Tip Read up on Aggregation Execution Planning Strategy for Aggregate Physical Operators in The Internals of Spark SQL book.</p>

```

val counts = spark.
  readStream.
  format("rate").
  load.
  groupBy(window($"timestamp", "5 seconds") as "group").
  agg(count("value") as "count").
  orderBy("group")
scala> counts.explain
== Physical Plan ==
*Sort [group#6 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(group#6 ASC NULLS FIRST, 200)
  +- *HashAggregate(keys=[window#13], functions=[count(value#1L)])
    +- StateStoreSave [window#13], StatefulOperatorStateInfo(<unknown>, 736d67c2-6daa
-4c4c-9c4b-c12b15af20f4, 0, 0), Append, 0
    +- *HashAggregate(keys=[window#13], functions=[merge_count(value#1L)])
      +- StateStoreRestore [window#13], StatefulOperatorStateInfo(<unknown>, 736d
67c2-6daa-4c4c-9c4b-c12b15af20f4, 0, 0)
      +- *HashAggregate(keys=[window#13], functions=[merge_count(value#1L)])
        +- Exchange hashpartitioning(window#13, 200)
          +- *HashAggregate(keys=[window#13], functions=[partial_count(valu
e#1L)])
            +- *Project [named_struct(start, precisetimestampconversion(((CASE WHEN (cast(CEIL((cast((precisetimestampconversion(timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0)) as double) = (cast((precisetimestampconversion(timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0)) THEN (CEIL((cast((precisetimestampconversion(timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0)) + 1) ELSE CEIL((cast((precisetimestampconversion(timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0)) END + 0) - 1) * 5000000) + 0), LongType, Times
tampType), end, precisetimestampconversion((((CASE WHEN (cast(CEIL((cast((precisetime
stampconversion(timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0)) as
double) = (cast((precisetimestampconversion(timestamp#0, TimestampType, LongType) - 0
) as double) / 5000000.0)) THEN (CEIL((cast((precisetimestampconversion(timestamp#0, T
imestampType, LongType) - 0) as double) / 5000000.0)) + 1) ELSE CEIL((cast((precisetim
estampconversion(timestamp#0, TimestampType, LongType) - 0) as double) / 5000000.0)) E
ND + 0) - 1) * 5000000) + 5000000), LongType, TimestampType)) AS window#13, value#1L]
            +- *Filter isnullobject(timestamp#0)
              +- StreamingRelation rate, [timestamp#0, value#1L]

import org.apache.spark.sql.streaming.{OutputMode, Trigger}
import scala.concurrent.duration.-
val consoleOutput = counts.
  writeStream.
  format("console").
  option("truncate", false).
  trigger(Trigger.ProcessingTime(10.seconds)).
  queryName("counts").
  outputMode(OutputMode.Complete). // <-- required for groupBy
  start

// Eventually...
consoleOutput.stop

```

Selecting Aggregate Physical Operator Given Aggregate Expressions — `AggUtils.planStreamingAggregation` Internal Method

```
planStreamingAggregation(
    groupingExpressions: Seq[NamedExpression],
    functionsWithoutDistinct: Seq[AggregateExpression],
    resultExpressions: Seq[NamedExpression],
    child: SparkPlan): Seq[SparkPlan]
```

`planStreamingAggregation` takes the grouping attributes (from `groupingExpressions`).

Note	<code>groupingExpressions</code> corresponds to the grouping function in groupBy operator.
------	--

`planStreamingAggregation` creates an aggregate physical operator (called `partialAggregate`) with:

- `requiredChildDistributionExpressions` `undefined` (i.e. `None`)
- `initialInputBufferOffset` `as 0`
- `functionsWithoutDistinct` in `Partial` mode
- `child` operator as the input `child`

	<p><code>planStreamingAggregation</code> creates one of the following aggregate physical operators (in the order of preference):</p> <ol style="list-style-type: none"> 1. <code>HashAggregateExec</code> 2. <code>ObjectHashAggregateExec</code> 3. <code>SortAggregateExec</code> <p><code>planStreamingAggregation</code> uses <code>AggUtils.createAggregate</code> method to select an aggregate physical operator that you can read about in Selecting Aggregate Physical Operator Given Aggregate Expressions — <code>AggUtils.createAggregate</code> Internal Method in Mastering Apache Spark 2 gitbook.</p>
--	--

`planStreamingAggregation` creates an aggregate physical operator (called `partialMerged1`) with:

- `requiredChildDistributionExpressions` based on the input `groupingExpressions`
- `initialInputBufferOffset` as the length of `groupingExpressions`
- `functionsWithoutDistinct` in `PartialMerge` mode
- `child` operator as [partialAggregate](#) aggregate physical operator created above

`planStreamingAggregation` creates `StateStoreRestoreExec` with the grouping attributes, `statefulOperatorStateInfo`, and `partialMerged1` aggregate physical operator created above.

`planStreamingAggregation` creates an aggregate physical operator (called `partialMerged2`) with:

- `child` operator as `StateStoreRestoreExec` physical operator created above

Note	The only difference between <code>partialMerged1</code> and <code>partialMerged2</code> steps is the child physical operator.
------	---

`planStreamingAggregation` creates `StateStoreSaveExec` with:

- the grouping attributes based on the input `groupingExpressions`
- No `stateInfo`, `outputMode` and `eventTimeWatermark`
- `child` operator as `partialMerged2` aggregate physical operator created above

In the end, `planStreamingAggregation` creates the final aggregate physical operator (called `finalAndCompleteAggregate`) with:

- `requiredChildDistributionExpressions` based on the input `groupingExpressions`
- `initialInputBufferOffset` as the length of `groupingExpressions`
- `functionsWithoutDistinct` in `Final` mode
- `child` operator as `StateStoreSaveExec` physical operator created above

Note	<code>planStreamingAggregation</code> is used exclusively when <code>StatefulAggregationStrategy</code> plans a streaming aggregation.
------	--

StreamingDeduplicationStrategy Execution Planning Strategy for Deduplicate Logical Operator

`StreamingDeduplicationStrategy` is an execution planning strategy that can plan streaming queries with `Deduplicate` logical operators (over streaming queries) to `StreamingDeduplicateExec` physical operators.

Tip Read up on [Execution Planning Strategies](#) in [The Internals of Spark SQL](#) book.

Note `Deduplicate` logical operator represents `Dataset.dropDuplicates` operator in a logical query plan.

`StreamingDeduplicationStrategy` is available using `SessionState`.

```
spark.sessionState.planner.StreamingDeduplicationStrategy
```

Demo: Using StreamingDeduplicationStrategy

FIXME

StreamingGlobalLimitStrategy Execution Planning Strategy

`StreamingGlobalLimitStrategy` is an execution planning strategy that can plan streaming queries with `ReturnAnswer` and `Limit` logical operators (over streaming queries) with the `Append` output mode to `StreamingGlobalLimitExec` physical operator.

Tip

Read up on [Execution Planning Strategies](#) in [The Internals of Spark SQL](#) book.

`StreamingGlobalLimitStrategy` is used (and [created](#)) exclusively when `IncrementalExecution` is requested to plan a streaming query.

`StreamingGlobalLimitStrategy` takes a single `OutputMode` to be created (which is the `OutputMode` of the `IncrementalExecution`).

Demo: Using StreamingGlobalLimitStrategy

FIXME

StreamingJoinStrategy Execution Planning Strategy — Stream-Stream Equi-Joins

`StreamingJoinStrategy` is an execution planning strategy that can plan streaming queries with `Join` logical operators of two streaming queries to a `StreamingSymmetricHashJoinExec` physical operator.

Tip

Read up on [Execution Planning Strategies](#) in [The Internals of Spark SQL](#) book.

`StreamingJoinStrategy` is used exclusively when `IncrementalExecution` is requested to plan a streaming query.

Demo: Using StreamingJoinStrategy

```

scala> :type spark
org.apache.spark.sql.SparkSession

scala> :type spark.sessionState.planner
org.apache.spark.sql.execution.SparkPlanner

// FIXME How to use the strategy in the demo
val strategy = spark.sessionState.planner.StreamingJoinStrategy

val left = spark.readStream.format("rate").load
val right = spark.readStream.format("rate").load
val q = left.join(right, "value")
scala> q.explain(true)

...
== Optimized Logical Plan ==
Project [value#52L, timestamp#51, timestamp#55]
+- Join Inner, (value#52L = value#56L)
  :- Filter isnotnull(value#52L)
  :  +- StreamingRelationV2 org.apache.spark.sql.execution.streaming.sources.RateStre
amProvider@60b0015, rate, [timestamp#51, value#52L]
  +- Filter isnotnull(value#56L)
    +- StreamingRelationV2 org.apache.spark.sql.execution.streaming.sources.RateStre
amProvider@24d92fffc, rate, [timestamp#55, value#56L]

== Physical Plan ==
*(3) Project [value#52L, timestamp#51, timestamp#55]
+- StreamingSymmetricHashJoin [value#52L], [value#56L], Inner, condition = [ leftOnly
= null, rightOnly = null, both = null, full = null ], state info [ checkpoint = <unkno
wn>, runId = f254d136-d903-4b1c-9fd5-861b541848ab, opId = 0, ver = 0, numPartitions =
200], 0, state cleanup [ left = null, right = null ]
  :- Exchange hashpartitioning(value#52L, 200)
    :  +- *(1) Filter isnotnull(value#52L)
      :    +- StreamingRelation rate, [timestamp#51, value#52L]
    +- ReusedExchange [timestamp#55, value#56L], Exchange hashpartitioning(value#52L, 2
00)

```

StreamingRelationStrategy Execution Planning Strategy for StreamingRelation and StreamingExecutionRelation Logical Operators

`StreamingRelationStrategy` is an execution planning strategy that can plan streaming queries with `StreamingRelation`, `StreamingExecutionRelation`, and `StreamingRelationV2` logical operators to `StreamingRelationExec` physical operators.

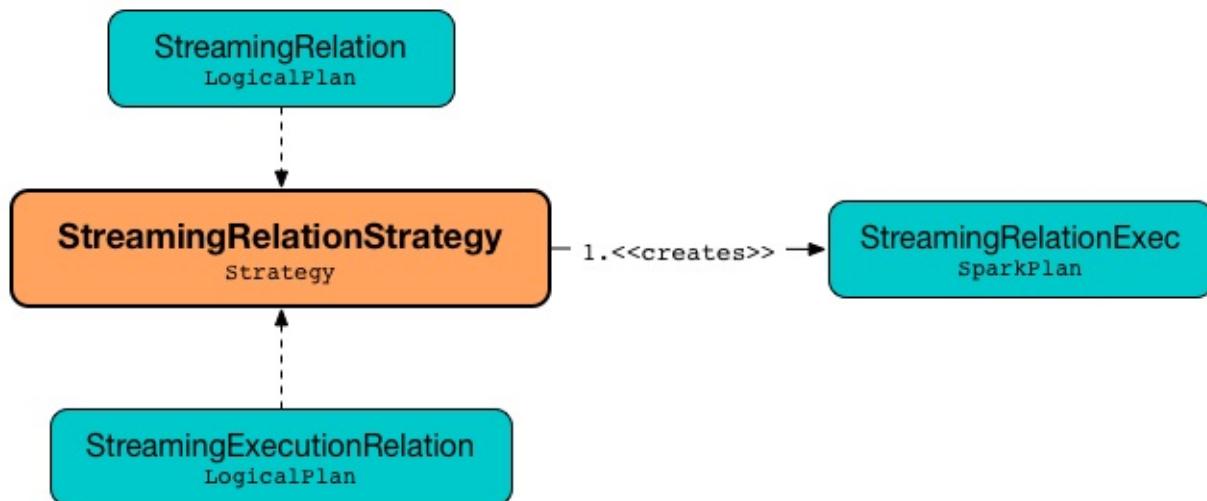


Figure 1. StreamingRelationStrategy, StreamingRelation, StreamingExecutionRelation and StreamingRelationExec Operators

Tip	Read up on Execution Planning Strategies in The Internals of Spark SQL book.
-----	--

`StreamingRelationStrategy` is used exclusively when `IncrementalExecution` is requested to plan a streaming query.

`StreamingRelationStrategy` is available using `SessionState` (of a `SparkSession`).

```
spark.sessionState.planner.StreamingRelationStrategy
```

Demo: Using StreamingRelationStrategy

```
val rates = spark.  
  readStream.  
  format("rate").  
  load // <-- gives a streaming Dataset with a logical plan with StreamingRelation logical operator  
  
// StreamingRelation logical operator for the rate streaming source  
scala> println(rates.queryExecution.logical.numberedTreeString)  
00 StreamingRelation DataSource(org.apache.spark.sql.SparkSession@31ba0af0, rate, List(),  
None, List(), None, Map(), None), rate, [timestamp#0, value#1L]  
  
// StreamingRelationExec physical operator (shown without "Exec" suffix)  
scala> rates.explain  
== Physical Plan ==  
StreamingRelation rate, [timestamp#0, value#1L]  
  
// Let's do the planning manually  
import spark.sessionState.planner.StreamingRelationStrategy  
val physicalPlan = StreamingRelationStrategy.apply(rates.queryExecution.logical).head  
scala> println(physicalPlan.numberedTreeString)  
00 StreamingRelation rate, [timestamp#0, value#1L]
```

UnsupportedOperationChecker

`UnsupportedOperationChecker` checks whether the logical plan of a streaming query uses supported operations only.

Note	<code>UnsupportedOperationChecker</code> is used exclusively when the internal <code>spark.sql.streaming.unsupportedOperationCheck</code> Spark property is enabled (which is by default).
------	--

Note	<code>UnsupportedOperationChecker</code> comes actually with two methods, i.e. <code>checkForBatch</code> and <code>checkForStreaming</code> , whose names reveal the different flavours of Spark SQL (as of 2.0), i.e. batch and streaming, respectively.
------	--

The Spark Structured Streaming gitbook is solely focused on `checkForStreaming` method.

checkForStreaming Method

```
checkForStreaming(plan: LogicalPlan, outputMode: OutputMode): Unit
```

`checkForStreaming` asserts that the following requirements hold:

1. Only one streaming aggregation is allowed
2. Streaming aggregation with Append output mode requires watermark (on the grouping expressions)
3. Multiple `flatMapGroupsWithState` operators are only allowed with Append output mode

`checkForStreaming ...FIXME`

`checkForStreaming` finds all streaming aggregates (i.e. `Aggregate` logical operators with streaming sources).

Note	<code>Aggregate</code> logical operator represents <code>Dataset.groupBy</code> and <code>Dataset.groupByKey</code> operators (and SQL's <code>GROUP BY</code> clause) in a logical query plan.
------	---

`checkForStreaming` asserts that there is exactly one streaming aggregation in a streaming query.

Otherwise, `checkForStreaming` reports a `AnalysisException`:

Multiple streaming aggregations are not supported with streaming DataFrames/Datasets

`checkForStreaming` asserts that `watermark` was defined for a streaming aggregation with `Append` output mode (on at least one of the grouping expressions).

Otherwise, `checkForStreaming` reports a `AnalysisException`:

Append output mode not supported when there are streaming aggregations on streaming DataFrames/DataSets without watermark

Caution	FIXME
---------	-------

`checkForStreaming` counts all `FlatMapGroupsWithState` logical operators (on streaming Datasets with `isMapGroupsWithState` flag disabled).

Note	<code>FlatMapGroupsWithState</code> logical operator represents <code>KeyValueGroupedDataset.mapGroupsWithState</code> and <code>KeyValueGroupedDataset.flatMapGroupsWithState</code> operators in a logical query plan.
Note	<code>FlatMapGroupsWithState.isMapGroupsWithState</code> flag is disabled when... FIXME

`checkForStreaming` asserts that multiple `FlatMapGroupsWithState` logical operators are only used when:

- `outputMode` is `Append` output mode
- `outputMode` of the `FlatMapGroupsWithState` logical operators is also `Append` output mode

Caution	FIXME Reference to an example in <code>flatMapGroupsWithState</code>
---------	--

Otherwise, `checkForStreaming` reports a `AnalysisException`:

Multiple `flatMapGroupsWithStates` are not supported when they are not all in append mode or the output mode is not append on a streaming DataFrames/Datasets

Caution	FIXME
---------	-------

Note	<p><code>checkForStreaming</code> is used exclusively when <code>StreamingQueryManager</code> is requested to create a <code>StreamingQueryWrapper</code> (for starting a streaming query), but only when the internal <code>spark.sql.streaming.unsupportedOperationCheck</code> Spark property is enabled (which is by default).</p>
------	--