BTS | Barcelona Technology School

Real Time Data Analysis

Master in Big Data Solutions 2020-2021

Ankit Tewari
Course Instructor, Real Time Data Analysis (2020-21)
ankit.tewari@bts.tech

Today's Agenda

What are we going to discuss today?

- Understand the concept of data streams
- Perform basic transformations using data streams
- Discuss major challenges and propose solutions while processing data streams
- Explore production grade methods for stream processing applications

Contents

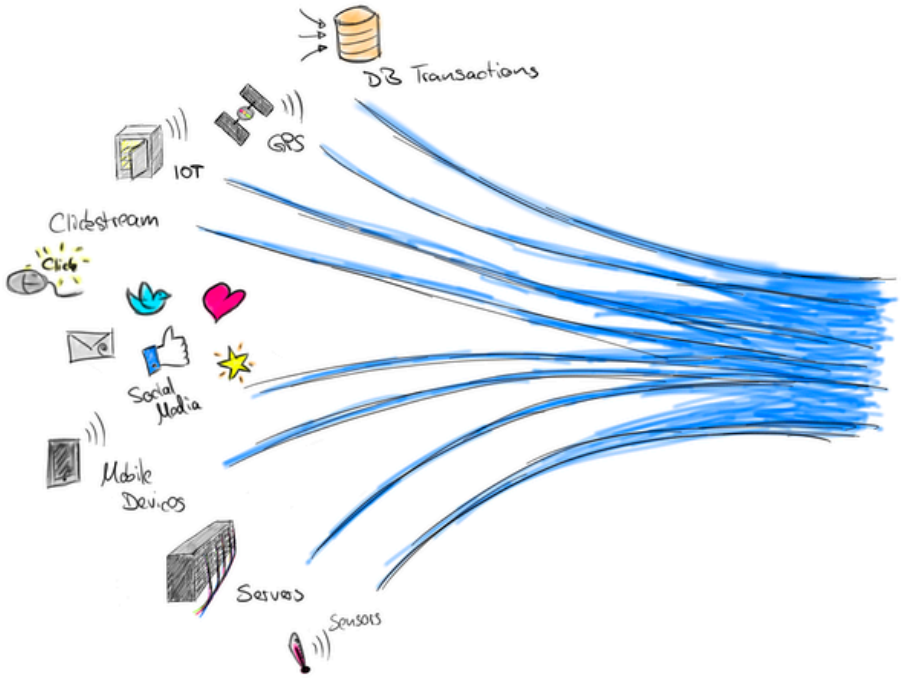
Today's topics at a glance

Introduction to Data Streams

What do we mean by data streams?

Streaming data is the continuous flow of data generated by various sources. By using stream processing technology, data streams can be processed, stored, analysed, and acted upon as it's generated in real-time.

Streams of data can be generated from various sources such as sensors onboard an aircraft or a smart watch, social media updates, e-commerce transactions, machine logs etc.



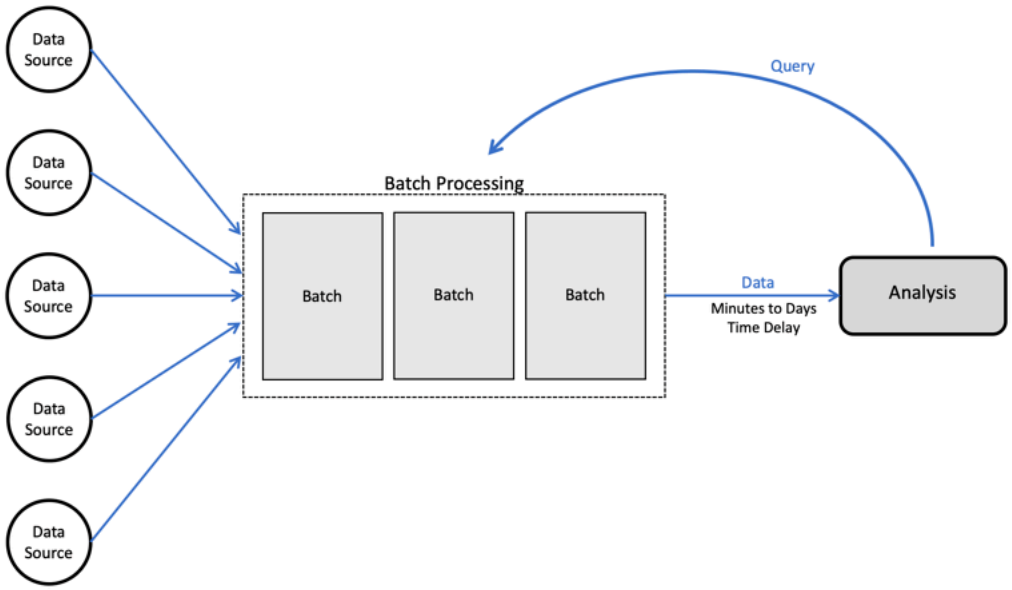
Introduction to Data Streams

Comparison between batches and streams of data

	Batch Data Processing	Real-Time Data Processing	Streaming Data
Hardware	Most storage and processing resources requirement to process large batches of data.	Less storage required to process the current or recent set of data packets. Less computational requirements.	Less storage required to process current data packets. More processing resources required to "stay awake" in order to meet real-time processing guarantees
Performance	Latency could be minutes, hours, or days	Latency needs to be in seconds or milliseconds	Latency must be guaranteed in milliseconds
Data set	Large batches of data	Current data packet or a few of them	Continuous streams of data
Analysis	Complex computation and analysis of a larger time frame	Simple reporting or computation	Simple reporting or computation

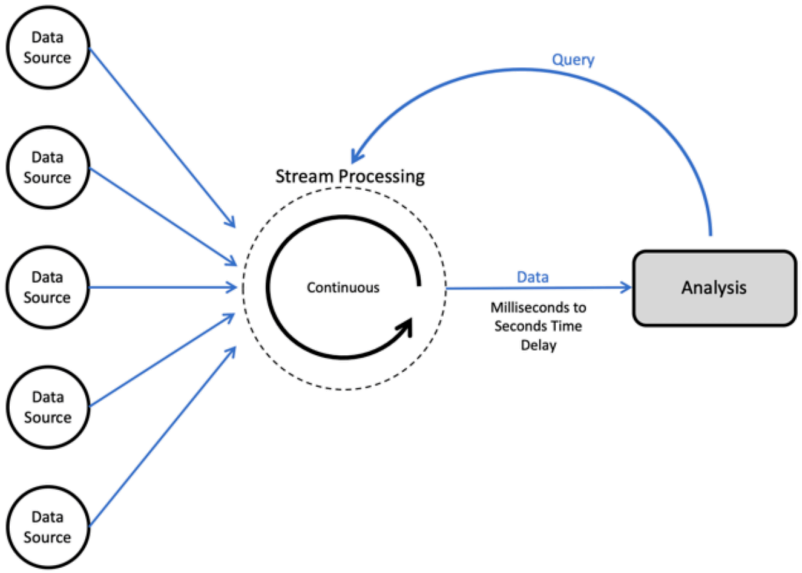
Introduction to Data Streams

Data batches



Introduction to Data Streams

Data streams



BTS | Barcelona Technology School

Stream Processing Use-Cases

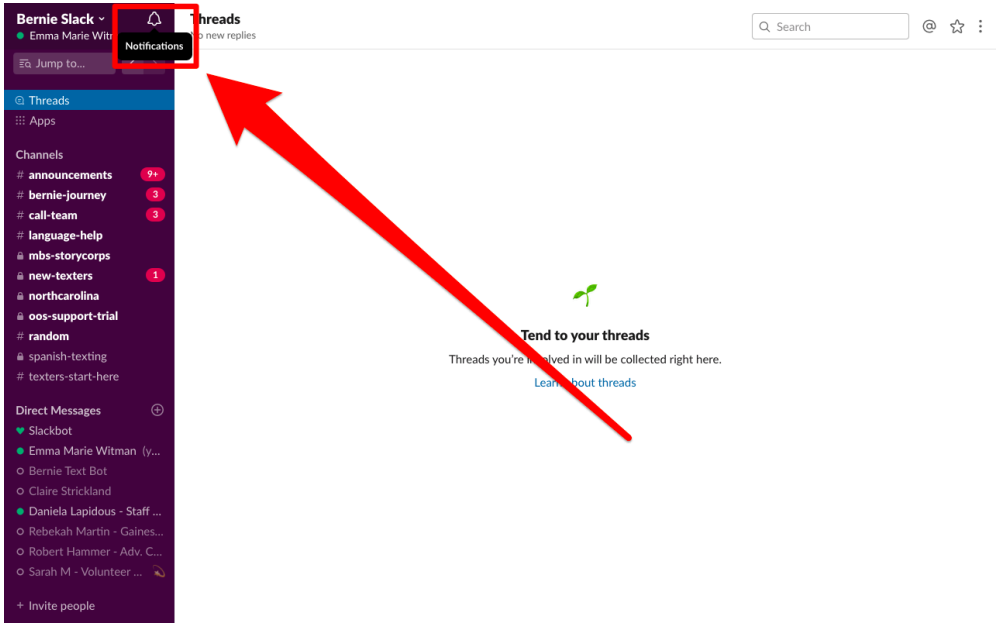
Use-Cases

Notifications and Alerts

Probably the most obvious streaming use case involves notifications and alerting. Given some series of events, a notification or alert should be triggered if some sort of event or series of events occurs. This doesn't necessarily imply autonomous or pre-programmed decision making;

Alerting can also be used to notify a human counterpart of some action that needs to be taken. An example might be driving an alert to an employee at a fulfilment centre that they need to get a certain item from a location in the warehouse and ship it to a customer. In either case, the notification needs to happen quickly;

Here, we provide the example how slack uses real time notifications to inform the users about events and actions;

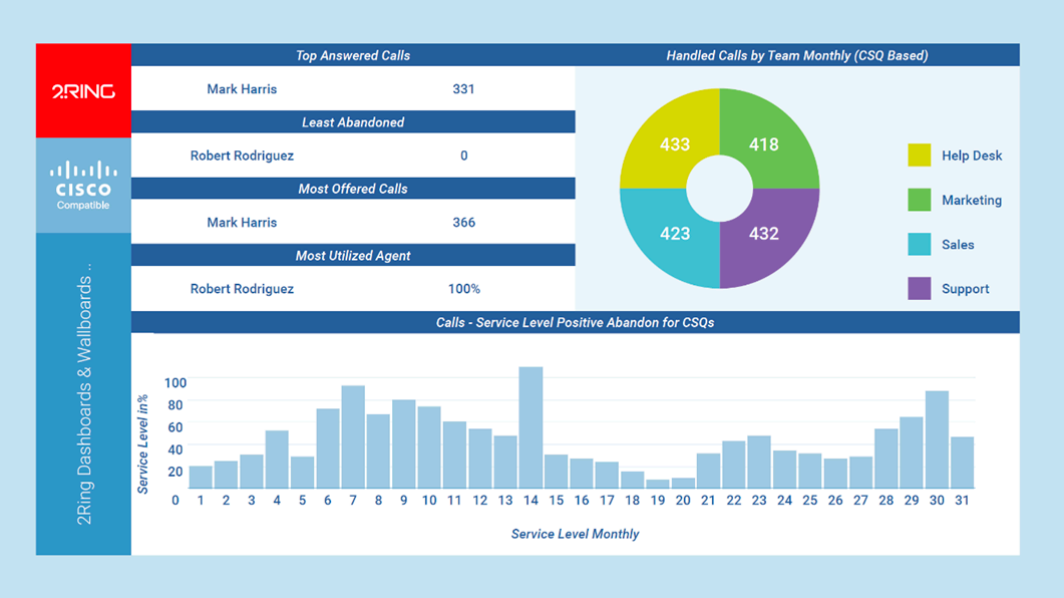


Use-Cases

Real Time Reporting

One of the most common streaming applications is to reduce the latency companies must endure while retrieving information into a data warehouse—in short, “my batch job, but streaming.” Spark batch jobs are often used for Extract, Transform, and Load (ETL) workloads that turn raw data into a structured format like Parquet to enable efficient queries. Using Structured Streaming, these jobs can incorporate new data within seconds, enabling users to query it faster downstream;

In this use case, it is critical that data is processed exactly once and in a fault-tolerant manner: we don’t want to lose any input data before it makes it to the warehouse, and we don’t want to load the same data twice. Moreover, the streaming system needs to make updates to the data warehouse transactionally so as not to confuse the queries running on it with partially written data;



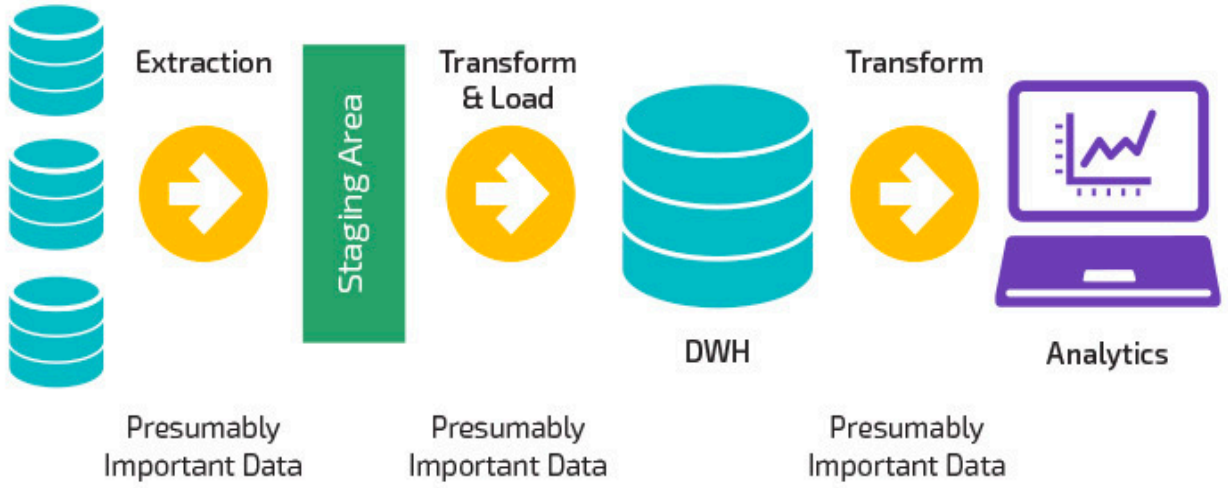
Use-Cases

Incremental Extract-Transform-Load (ETL) Operations

One of the most common streaming applications is to reduce the latency companies must endure while retrieving information into a data warehouse—in short, “my batch job, but streaming.” Spark batch jobs are often used for Extract, Transform, and Load (ETL) workloads that turn raw data into a structured format like Parquet to enable efficient queries. Using Structured Streaming, these jobs can incorporate new data within seconds, enabling users to query it faster downstream;

In this use case, it is critical that data is processed exactly once and in a fault-tolerant manner: we don’t want to lose any input data before it makes it to the warehouse, and we don’t want to load the same data twice;

Moreover, the streaming system needs to make updates to the data warehouse transactionally so as not to confuse the queries running on it with partially written data;



Use-Cases

Online Machine Learning

Being a Data Scientist, you may have faced the problem of change in your existing training dataset due to arrival of more data;
In order to overcome such challenges, there are two approaches adopted in general-

a) Retrain based on the addition of new datapoints in a modified dataset;
b) Schedule training on new data by defining a training period and automatically deploy the updated model;
c) Create a learning strategy to allow the model to learn as soon as it sees new data in near real time;

There are many new frameworks that propose the training machine learning algorithms on data streams. Some of the most important ones are described as follows-

Use-Cases

Online Machine Learning

Being a Data Scientist, you may have faced the problem of change in your existing training dataset due to arrival of more data;
In order to overcome such challenges, there are two approaches adopted in general-

a) Retrain based on the addition of new datapoints in a modified dataset;
b) Schedule training on new data by defining a training period and automatically deploy the updated model;
c) Create a learning strategy to allow the model to learn as soon as it sees new data in near real time;

Offline Learning	Features	Online Learning
Less complex as model is constant	**Complexity**	Dynamic complexity as the model keeps evolving over time
Fewer computations, single time batch-based training	**Computational Power**	Continuous data ingestions result in consequent model refinement computations
Easier to implement	**Use in Production**	Difficult to implement and manage
Image Classification or anything related to Machine Learning - where data patterns remains constant without sudden concept drifts	**Applications**	Used in finance, economics, heath where new data patterns are constantly emerging
Industry proven tools. E.g. Sci-kit, TensorFlow, Pytorch, Keras, Spark Mlib	**Tools**	Active research/New project tools: E.g. MOA, SAMOA, scikit-multiflow, streamDM

Use-Cases

Advantages of Stream Processing

Following are some of the major advantages of stream processing-

- Stream processing can also be more efficient in updating a result than repeated batch jobs, because it automatically incrementalise the computation
- Stream processing enables lower latency
- Ability to process data in batch allows for vastly higher data processing throughput than many streaming systems

Use-Cases

Disadvantages of Stream Processing

Following are some of the most important disadvantages of stream processing-

- Processing out-of-order data based on application timestamps (also called event time)
- Maintaining large amounts of state
- Supporting high-data throughput
- Processing each event exactly once despite machine failures
- Handling load imbalance and stragglers
- Responding to events at low latency
- Joining with external data in other storage systems
- Determining how to update output sinks as new events arrive
- Writing data to output systems by means of transactions
- Updating your application’s business logic at runtime

Use-Cases

Stream Processing Design Points

The most important factors while designing a stream processing system are-

1. High throughput,
2. Low latency, and
3. Ability to process out-of-order data

Some of the most common design patterns are described as follows-

1. **Record-at-a-time vs Declarative**:
   Basically, in a record-at-a-time system, the processing happens as soon as a record is received by the system which means the system ingests stream in terms of one record at a time while in a declarative system, the application specifies what to compute but not how to compute it in response to each new event and how to recover from failure.

1. **Event Time vs Processing Time**
2. **Continuous vs Micro-Batch Execution**

Use-Cases

Stream Processing Design Points

The most important factors while designing a stream processing system are-

1. High throughput,
2. Low latency, and
3. Ability to process out-of-order data

Some of the most common design patterns are described as follows-

1. **Record-at-a-time vs Declarative**
2. **Event Time vs Processing Time**:
   It is the idea of processing data based on timestamps inserted into each record at the source (event time), as opposed to the time when the record is received at the streaming application (processing time)

3. **Continuous vs Micro-Batch Execution**

Use-Cases

Stream Processing Design Points

The most important factors while designing a stream processing system are-

1. High throughput,
2. Low latency, and
3. Ability to process out-of-order data

Some of the most common design patterns are described as follows-

1. **Record-at-a-time vs Declarative**
2. **Event Time vs Processing Time**
3. **Continuous vs Micro-Batch Execution**:
   Continuous processing-based systems, each node in the system is continually listening to messages from other nodes and outputting new updates to its child nodes;

   Micro-batch systems wait to accumulate small batches of input data (say, 500 ms worth), then process each batch in parallel using a distributed collection of tasks, similar to the execution of a batch job in Spark

BTS | Barcelona Technology School

Transformations

Transformations

Understanding the Activity Recognition Data

The Heterogeneity Human Activity Recognition (HHAR) dataset from Smartphones and Smartwatches is a dataset devised to benchmark human activity recognition algorithms (classification, automatic data segmentation, sensor fusion, feature extraction, etc.) in real-world contexts;

specifically, the dataset is gathered with a variety of different device models and use-scenarios, in order to reflect sensing heterogeneities to be expected in real deployments.

Data Set Characteristics:	Multivariate, Time-Series	**Number of Instances:**	43930257	**Area:**	Computer
Attribute Characteristics:	Real	**Number of Attributes:**	16	**Date Donated**	2015-10-26
Associated Tasks:	Classification, Clustering	**Missing Values?**	Yes	**Number of Web Hits:**	112700

The dataset contains the readings of two motion sensors commonly found in smartphones. Reading were recorded while users executed activities scripted in no specific order carrying smartwatches and smartphones.

1. **Activities**: ‘Biking’, ‘Sitting’, ‘Standing’, ‘Walking’, ‘Stair Up’ and ‘Stair down’.
2. **Sensors**: Sensors: Two embedded sensors, i.e., Accelerometer and Gyroscope, sampled at the highest frequency the respective device allows.
3. **Devices**: 4 smartwatches (2 LG watches, 2 Samsung Galaxy Gears)
4. **8 smartphones** (2 Samsung Galaxy S3 mini, 2 Samsung Galaxy S3, 2 LG Nexus 4, 2 Samsung Galaxy S+)

Transformations

Data Exploration

The first step before we begin to play with our structured streaming examples is to explore the data available to us. The data frame below is a spark data frame that consists of activity recognition data.

```
static= spark.read.json("./activity-data/")
```

```
static.show(4)
```

```
+-------------+-----------------+--------+-----+------+----+-----+------------+------------+------------+
| Arrival_Time|      Creation_Time|  Device|Index| Model|User|   gt|           x|           y|           z|
+-------------+-----------------+--------+-----+------+----+-----+------------+------------+------------+
|1424686735090|1424686733090638193|nexus4_1|   18|nexus4|   g|stand| 3.356934E-4|-5.645752E-4|-0.018814087|
|1424686735292|1424688581345918092|nexus4_2|   66|nexus4|   g|stand|-0.005722046| 0.029083252| 0.005569458|
|1424686735500|1424686733498505625|nexus4_1|   99|nexus4|   g|stand|   0.0078125|-0.017654419| 0.010025024|
|1424686735691|1424688581745026978|nexus4_2|  145|nexus4|   g|stand|-3.814697E-4|   0.0184021|-0.013656616|
+-------------+-----------------+--------+-----+------+----+-----+------------+------------+------------+
only showing top 4 rows
```

Transformations

Selections and filtering

All select and filter transformations are supported in Structured Streaming, as are all Data Frame functions and individual column manipulations

```
simpleTransform = streaming.withColumn("stairs", f.expr("gt like '%stairs%'"))\
.where("stairs")\
.where("gt is not null")\
.select("gt", "model", "arrival_time", "creation_time")\
.writeStream\
.queryName("simple_transform")\
.format("memory")\
.outputMode("append")\
.start()
```

Transformations

Selections and filtering

All select and filter transformations are supported in Structured Streaming, as are all Data Frame functions and individual column manipulations

```
simpleTransform = streaming.withColumn("stairs", f.expr("gt like '%stairs%'"))\
.where("stairs")\
.where("gt is not null")\
.select("gt", "model", "arrival_time", "creation_time")\
.writeStream\
.queryName("simple_transform")\
.format("memory")\
.outputMode("append")\
.start()
```

```
spark.sql("SELECT * FROM simple_transform").show(5)
```

```
+--------+------+-------------+-------------------+
|      gt| model| arrival_time|      creation_time|
+--------+------+-------------+-------------------+
|stairsup|nexus4|1424687983801|1424689829851420571|
|stairsup|nexus4|1424687984163|1424687982169917952|
|stairsup|nexus4|1424687984571|1424687982572835163|
|stairsup|nexus4|1424687984972|1424687982975667195|
|stairsup|nexus4|1424687985370|1424687983379305060|
+--------+------+-------------+-------------------+
only showing top 5 rows
```

Transformations

Aggregations

Structured Streaming has excellent support for aggregations. We can specify arbitrary aggregations, as we can do in case of Structured APIs.

```
deviceModelStats = streaming.cube("gt", "model").avg()\
.drop("avg(Arrival_time)")\
.drop("avg(Creation_Time)")\
.drop("avg(Index)")\
.writeStream.queryName("device_counts").format("memory")\
.outputMode("complete")\
.start()
```

Transformations

Aggregations

Structured Streaming has excellent support for aggregations. We can specify arbitrary aggregations, as we can do in case of Structured APIs.

```
deviceModelStats = streaming.cube("gt", "model").avg()\
.drop("avg(Arrival_time)")\
.drop("avg(Creation_Time)")\
.drop("avg(Index)")\
.writeStream.queryName("device_counts").format("memory")\
.outputMode("complete")\
.start()
```

```
spark.sql("SELECT * FROM device_counts").show(5)
```

```
+---------+------+--------------------+--------------------+--------------------+
|       gt| model|              avg(x)|              avg(y)|              avg(z)|
+---------+------+--------------------+--------------------+--------------------+
|     null|nexus4|0.001243822064214...|-0.00508318886798...|-0.01029860468302...|
|     null|  null|-0.00534777761728...|-0.00471625131602...|0.001053548924143...|
|stairsdown|  null|0.022206868836056656| -0.0326125139584748| 0.11849359875695885|
|      sit|  null|-5.48643225000002E-4|-1.75231850243742...|-2.21252465063372...|
|stairsdown|nexus4|0.022206868836056656| -0.0326125139584748| 0.11849359875695885|
+---------+------+--------------------+--------------------+--------------------+
only showing top 5 rows
```

Transformations

Aggregations

Now, the essence of streaming is felt if we are calling the streaming API with the assumption of fresh data being added to our staging folder. The following code demonstrates such an activity-

```
from time import sleep
for x in range(0, 5):
    display(spark.sql(f"SELECT * from device_counts").toPandas().dropna().head(3))
    sleep(1)

    #clear_output(wait=True)
else:
    print("Live view ended...")
```

Transformations

Aggregations

Now, the essence of streaming is felt if we are calling the streaming API with the assumption of fresh data being added to our staging folder. The following code demonstrates such an activity-

```
from time import sleep
for x in range(0, 5):
    display(spark.sql(f"SELECT * from device_counts").toPandas().dropna().head(3))
    sleep(1)

    #clear_output(wait=True)
else:
    print("Live view ended...")
```

	gt	model	avg(x)	avg(y)	avg(z)
4	stairsdown	nexus4	0.021614	-0.032490	0.120359
5	bike	nexus4	0.022689	-0.008779	-0.082510
6	null	nexus4	-0.008477	-0.000730	0.003091

	gt	model	avg(x)	avg(y)	avg(z)
4	stairsdown	nexus4	0.021614	-0.032490	0.120359
5	bike	nexus4	0.022689	-0.008779	-0.082510
6	null	nexus4	-0.008477	-0.000730	0.003091

	gt	model	avg(x)	avg(y)	avg(z)
4	stairsdown	nexus4	0.021614	-0.032490	0.120359
5	bike	nexus4	0.022689	-0.008779	-0.082510
6	null	nexus4	-0.008477	-0.000730	0.003091

Here, you can observe that the outputs corresponding to our code above are exhibited. In every subsequent output, do you observe that the result of aggregations are nearly the same?

Why the results for these aggregations are same? How could we improve it to show actual different aggregations?

Transformations

Joins

Structured Streaming supports joining streaming DataFrames to static DataFrames. We can perform multiple column joins and supplement streaming data with that from static data sources.

```
historicalAgg = static.groupBy("gt", "model").avg()
deviceModelStats = streaming.drop("Arrival_Time", "Creation_Time", "Index")\
.cube("gt", "model").avg()\
.join(historicalAgg, ["gt", "model"])\
.writeStream.queryName("device_counts_join").format("memory")\
.outputMode("complete")\
.start()
```

Transformations

Joins

Structured Streaming supports joining streaming DataFrames to static DataFrames. We can perform multiple column joins and supplement streaming data with that from static data sources.

```
historicalAgg = static.groupBy("gt", "model").avg()
deviceModelStats = streaming.drop("Arrival_Time", "Creation_Time", "Index")\
.cube("gt", "model").avg()\
.join(historicalAgg, ["gt", "model"])\
.writeStream.queryName("device_counts_join").format("memory")\
.outputMode("complete")\
.start()
```

```
spark.sql("SELECT * FROM device_counts_join").show(5)
```

```
+----------+------+--------------------+--------------------+--------------------+--------------------+------------
--------+--------------------+--------------------+--------------------+
|        gt| model|              avg(x)|              avg(y)|              avg(z)|   avg(Arrival_Time)|  avg(Creati
on_Time)|        avg(Index)|              avg(x)|              avg(y)|              avg(z)|
+----------+------+--------------------+--------------------+--------------------+--------------------+------------
--------+--------------------+--------------------+--------------------+--------------------+
|      bike|nexus4|0.022964785314061254|-0.00883802096885...|-0.08280852157940341|1.424751134339985...|1.4247521273
69587...| 326459.6867328154|0.022688759550866796|-0.00877912156368...| -0.0825100166341234|
|      null|nexus4|-0.00861985194739502|-0.00118695209268...|0.002615053985997819|1.424749002876339...|1.4247499194
82128...| 219276.9663669269|-0.00847688860109...|-7.30455258739192...|0.003090601491419929|
|stairsdown|nexus4|0.021940372352212245|-0.03145080962306262| 0.12130916214488122|1.424744591412857...|1.4247455036
35638...|230452.44623187225| 0.02161390866916545|-0.03249018824752...| 0.12035922691504071|
|     stand|nexus4|-3.05486659249782...|2.950288503954850...|2.881832138007999E-4|1.424743637921210...|1.4247445795
47463...|31317.877585550017|-3.11082189691709...|3.218461665975353E-4|2.141300040636499...|
|      walk|nexus4|-0.00383499087252...|0.001711204986843...|-0.00152407149813...|1.424746420641790...|1.4247473510
60674...|149760.09974990616|-0.00390116006094...|0.001052508689953...|-6.95435553042996...|
+----------+------+--------------------+--------------------+--------------------+--------------------+------------
--------+--------------------+--------------------+--------------------+--------------------+
only showing top 5 rows
```

BTS | Barcelona Technology School

Input and Output

Input and Output

What do we mean by time series?

This section dives deeper into the details of how sources, sinks, and output modes work in Structured Streaming. Specifically, we discuss how, when, and where data flows into and out of the system. As of this writing, Structured Streaming supports several sources and sinks, including Apache Kafka, files, and several sources and sinks for testing and debugging.

Input and Output

Source and Sinks

Structured Streaming supports several production sources and sinks (files and Apache Kafka), as well as some debugging tools like the memory table sink.

Files Sources and Sinks

- It is considered the simplest source.
- It's easy to reason about and understand.
- While essentially any file source should work, the ones that we see in practice are Parquet, text, JSON, and CSV.

The only difference between using the file source/sink and Spark's static file source is that with streaming, *we can control the number of files that we read in during each trigger via the maxFilesPerTrigger option* that we saw earlier.

Keep in mind that any files you add into an input directory for a streaming job need to appear in it atomically. Otherwise, Spark will process partially written files before you have finished. On file systems that show partial writes, such as local files or HDFS, this is best done by writing the file in an external directory and moving it into the input directory when finished. On Amazon S3, objects normally only appear once fully written.

Input and Output

Source and Sinks

Structured Streaming supports several production sources and sinks (files and Apache Kafka), as well as some debugging tools like the memory table sink.

Kafka Source and Sinks

- Apache Kafka is a distributed publish-and-subscribe system for streams of data.
- Kafka lets you publish and subscribe to streams of records like you might do with a message queue—these are stored as streams of records in a fault-tolerant way.
- Think of Kafka like a distributed buffer. Kafka lets you store streams of records in categories that are referred to as topics.
- Each record in Kafka consists of a key, a value, and a timestamp.
- Topics consist of immutable sequences of records for which the position of a record in a sequence is called an offset.
- Reading data is called subscribing to a topic and writing data is as simple as publishing to a topic.
- Spark allows you to read from Kafka with both batch and streaming DataFrames.

BTS | Barcelona Technology School

Triggers

Triggers

What are triggers?

Triggers are used to control when data is output to our sink. By default, Structured Streaming will start data as soon as the previous trigger completes processing.

We can use triggers to ensure that we do not overwhelm our output sink with too many updates or to try and control file sizes in the output. Currently, there is one periodic trigger type, based on processing time, as well as a “once” trigger to manually run a processing step once.

Triggers

What are triggers?

Triggers are used to control when data is output to our sink. By default, Structured Streaming will start data as soon as the previous trigger completes processing.

We can use triggers to ensure that we do not overwhelm our output sink with too many updates or to try and control file sizes in the output. Currently, there is one periodic trigger type, based on processing time, as well as a “once” trigger to manually run a processing step once.

- **Processing time trigger**:

  The Processing Time trigger will wait for multiples of the given duration in order to output data. In order to execute it, we simply specify the duration as a string.

  For example, with a trigger duration of one minute, the trigger will fire at 12:00, 12:01, 12:02, and so on. If a trigger time is missed because the previous processing has not yet completed, then Spark will wait until the next trigger point (i.e., the next minute), rather than firing immediately.

- **Once trigger**

Triggers

What are triggers?

Triggers are used to control when data is output to our sink. By default, Structured Streaming will start data as soon as the previous trigger completes processing.

We can use triggers to ensure that we do not overwhelm our output sink with too many updates or to try and control file sizes in the output. Currently, there is one periodic trigger type, based on processing time, as well as a “once” trigger to manually run a processing step once.

- **Processing time trigger**

- **Once trigger**:
  We can also just run a streaming job once by setting that as the trigger.

  This might seem like a weird case, but it’s actually extremely useful in both development and production. During development, we can test the application on just one trigger’s worth of data at a time. During production, the Once trigger can be used to run our job manually at a low rate (e.g., import new data into a summary table just occasionally).

  Because Structured Streaming still fully tracks all the input files processed and the state of the computation, this is easier than writing your own custom logic to track this in a batch job, and saves a lot of resources over running a continuous job 24/7

Triggers

What are triggers?

Triggers are used to control when data is output to our sink. By default, Structured Streaming will start data as soon as the previous trigger completes processing.

We can use triggers to ensure that we do not overwhelm our output sink with too many updates or to try and control file sizes in the output. Currently, there is one periodic trigger type, based on processing time, as well as a “once” trigger to manually run a processing step once.

```
activityCounts.writeStream.trigger(once=True)\
.format("console").outputMode("complete").start()
```

```
<pyspark.sql.streaming.StreamingQuery at 0x13190afd0>
```

```
activityCounts.writeStream.trigger(processingTime='5 seconds')\
.format("console").outputMode("complete").start()
```

```
<pyspark.sql.streaming.StreamingQuery at 0x13190a710>
```

BTS | Barcelona Technology School

Output Modes

Output Modes

Structured Streaming Output Modes

Now, we know where the data can go, so let's discuss how the result Dataset will look when it gets there. We can control how this happens using Output Modes. There are essentially three output models described as follows-

1. **Append Mode**:
   When new rows are added to the result table, they will be output to the sink based on the trigger that you specify. each row is output once (and only once), assuming that you have a fault-tolerant sink; Why do assume fault tolerance in this case?

2. **Complete Mode**

3. **Update Mode**

Output Modes

Structured Streaming Output Modes

Now, we know where the data can go, so let's discuss how the result Dataset will look when it gets there. We can control how this happens using Output Modes. There are essentially three output models described as follows-

1. **Append Mode**
2. **Complete Mode**:
   Complete mode will output the entire state of the result table to your output sink. It is useful when we're working with some stateful data for which all rows are expected to change over time or the sink you are writing does not support row-level updates.

3. **Update Mode**

Output Modes

Structured Streaming Output Modes

Now, we know where the data can go, so let's discuss how the result Dataset will look when it gets there. We can control how this happens using Output Modes. There are essentially three output models described as follows-

1. **Append Mode**:
2. **Complete Mode**
3. **Update Mode**:
   Update mode is similar to complete mode except that only the rows that are different from the previous write are written out to the sink. Note that, If the query doesn't contain aggregations, this is equivalent to append mode. How?

Output Modes

Structured Streaming Output Modes

The table below provides with a brief understanding of which output mode to use under what situations. Remember that it totally depends from the business case we are working on presently-

Query Type	Query type (continued)	Supported Output Modes	Notes
Queries with aggregation	Aggregation on event-time with watermark	Append, Update, Complete	Append mode uses watermark to drop old aggregation state. This means that as new rows are brought into the table, Spark will only keep around rows that are below the “watermark”. Update mode also uses the watermark to remove old aggregation state. By definition, complete mode does not drop old aggregation state since this mode preserves all data in the Result Table.
	Other aggregations	Complete, Update	Since no watermark is defined (only defined in other category), old aggregation state is not dropped. Append mode is not supported as aggregates can update thus violating the semantics of this mode.
Queries with mapGroupsWithState		Update	
Queries with flatMapGroupsWithState	Append operation mode	Append	Aggregations are allowed after flatMapGroupsWithState.
	Update operation mode	Update	Aggregations not allowed after flatMapGroupsWithState.
Other queries		Append, Update	Complete mode not supported as it is infeasible to keep all unaggregated data in the Result Table.

BTS | Barcelona Technology School

Event Time
&
Stateful Processing

Event Time & Stateful Processing

Event Related Times

At a higher level, in stream processing systems there are effectively two relevant times for each event: the time at which it actually occurred (event time), and the time that it was processed or reached the stream processing system (processing time).

- **Event Time**:
  It is the time that is embedded in the data itself. It is most often, though not required to be, the time that an event actually occurs. This is important to use because *it provides a more robust way of comparing events against one another*. The challenge here is that event data can be late or out of order. This means that the stream processing system must be able to handle out-of-order or late data.

- **Processing Time**:
  It is *the time at which the stream-processing system actually receives data*. This is usually less important than event time because when it's processed is largely an implementation detail. This can't ever be out of order because it's a property of the streaming system at a certain time (not an external system like event time).

Event Time & Stateful Processing

Event Related Times

Suppose that we have a data-centre located in San Francisco. An event occurs in two places at the same time: one in Ecuador, the other in Virginia.

Due to the location of the data-centre, the event in Virginia is likely to show up in our data-centre before the event in Ecuador.

Event Time & Stateful Processing

Event Related Times

Suppose that we have a data-centre located in San Francisco. An event occurs in two places at the same time: one in Ecuador, the other in Virginia.

Due to the location of the data-centre, the event in Virginia is likely to show up in our data-centre before the event in Ecuador.

If we were to analyse this data based on processing time, it would appear that the event in Virginia occurred before the event in Ecuador: something that we know to be wrong. However, if we were to analyse the data based on event time (largely ignoring the time at which it's processed), we would see that these events occurred at the same time.

The key fundamental idea is that the order of the series of events in the processing system does not guarantee an ordering in event time.

Computer networks are unreliable. That means that events can be dropped, slowed down, repeated, or be sent without issue. Because individual events are not guaranteed to suffer one fate or the other, we must acknowledge that any number of things can happen to these events on the way from the source of the information to our stream processing system.

We need to operate on event time and look at the overall stream with reference to this information contained in the data rather than on when it arrives in the system. This means that we hope to compare events based on the time at which those events occurred.

Event Time & Stateful Processing

Event Times

Basically, the code here is reading the JSON files from the same activity recognition folder having datasets about activities and prints the schema of the data.

```
spark.conf.set("spark.sql.shuffle.partitions", 5)
static = spark.read.json("./activity-data")
streaming = spark\
.readStream\
.schema(static.schema)\
.option("maxFilesPerTrigger", 10)\
.json("./activity-data")

streaming.printSchema()
```

```
root
 |-- Arrival_Time: long (nullable = true)
 |-- Creation_Time: long (nullable = true)
 |-- Device: string (nullable = true)
 |-- Index: long (nullable = true)
 |-- Model: string (nullable = true)
 |-- User: string (nullable = true)
 |-- gt: string (nullable = true)
 |-- x: double (nullable = true)
 |-- y: double (nullable = true)
 |-- z: double (nullable = true)
```

Event Time & Stateful Processing

Windows on Event Times

Our first step in event-time analysis is to *convert the timestamp column into the proper Spark SQL timestamp type*. Our current column is Unix-time nanoseconds (represented as a long)

```
withEventTime = streaming\
.selectExpr("*", "cast(cast(Creation_Time as double)/1000000000 as timestamp) as event_time")
```

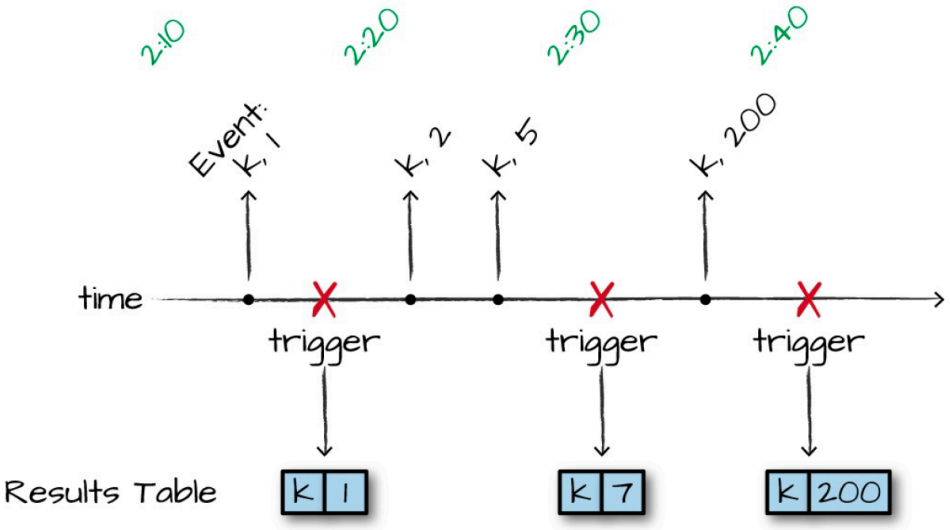
We're now prepared to do arbitrary operations on event time!

Note how this experience is just like we'd do in batch operations—there's no special API. We simply use columns, just like we might in batch, the aggregation, and we're working with event time.

Event Time & Stateful Processing

Tumbling Windows

Let us begin with the simplest operation to simply count the number of occurrences of an event in a given window. We are simply performing an aggregation of keys over a window of time. Once we have results, we can update the result table (depending on the output mode) when every trigger runs, which will operate on the data received since the last trigger.



Event Time & Stateful Processing

Tumbling Windows

Let us begin with the simplest operation to simply count the number of occurrences of an event in a given window. We are simply performing an aggregation of keys over a window of time. Once we have results, we can update the result table (depending on the output mode) when every trigger runs, which will operate on the data received since the last trigger.

```
from pyspark.sql.functions import window, col
withEventTime.groupBy(window(col("event_time"), "10 minutes")).count()\
.writeStream\
.queryName("pyevents_per_window")\
.format("memory")\
.outputMode("complete")\
.start()
```

```
<pyspark.sql.streaming.StreamingQuery at 0x131ea1fd0>
```

```
spark.sql("SELECT * FROM pyevents_per_window").printSchema()
```

```
root
 |-- window: struct (nullable = false)
 |    |-- start: timestamp (nullable = true)
 |    |-- end: timestamp (nullable = true)
 |-- count: long (nullable = false)
```

In our example, we will do so in 10-minute windows without any overlap between them. We're writing out to the in-memory sink for debugging, so we can query it with SQL after we have the stream running.

This will update in real time, as well, meaning that if new events were being added upstream to our system, Structured Streaming would update those counts accordingly. This is the complete output mode, Spark will output the entire result table regardless of whether we've seen the entire dataset.

Event Time & Stateful Processing

Tumbling Windows

Let us begin with the simplest operation to simply count the number of occurrences of an event in a given window. We are simply performing an aggregation of keys over a window of time. Once we have results, we can update the result table (depending on the output mode) when every trigger runs, which will operate on the data received since the last trigger.

```
from pyspark.sql.functions import window, col
withEventTime.groupBy(window(col("event_time"), "10 minutes")).count()\
.writeStream\
.queryName("pyevents_per_window")\
.format("memory")\
.outputMode("complete")\
.start()
```

```
<pyspark.sql.streaming.StreamingQuery at 0x131ea1fd0>
```

```
spark.sql("SELECT * FROM pyevents_per_window").show(5)
```

```
+--------------------+------+
|              window| count|
+--------------------+------+
|{2015-02-24 17:20...|150773|
|{2015-02-24 18:30...|133323|
|{2015-02-23 18:00...|100853|
|{2015-02-23 15:50...| 99178|
|{2015-02-24 18:00...|125679|
+--------------------+------+
only showing top 5 rows
```

Event Time & Stateful Processing

Tumbling Windows

Let us begin with the simplest operation to simply count the number of occurrences of an event in a given window. We are simply performing an aggregation of keys over a window of time. Once we have results, we can update the result table (depending on the output mode) when every trigger runs, which will operate on the data received since the last trigger.

```
from pyspark.sql.functions import window, col
withEventTime.groupBy(window(col("event_time"), "10 minutes"), "User").count()\
.writeStream\
.queryName("events_per_window")\
.format("memory")\
.outputMode("complete")\
.start()
```

```
<pyspark.sql.streaming.StreamingQuery at 0x1319ead50>
```

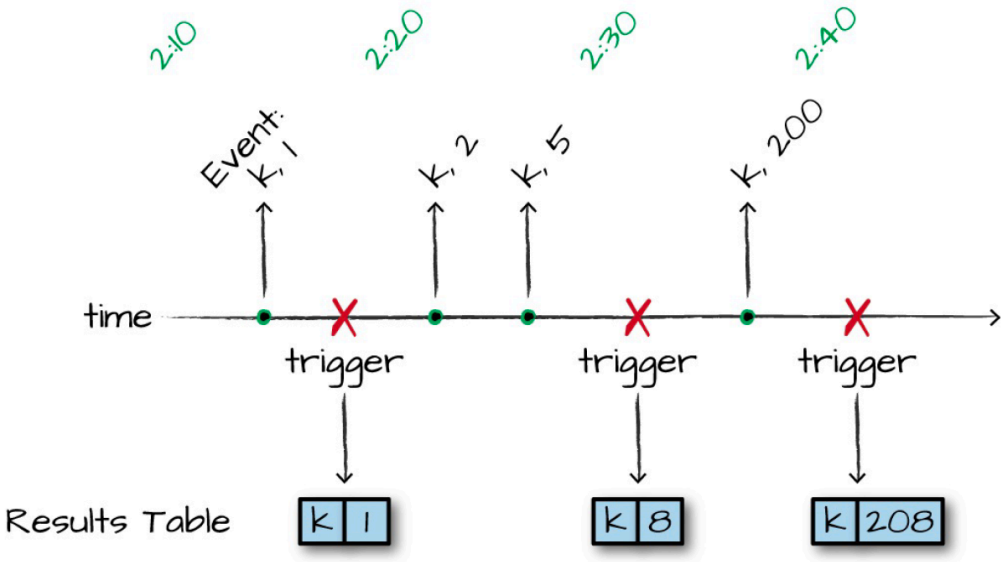
```
spark.sql("SELECT * FROM events_per_window").show(5)
```

```
+--------------------+----+------+
|              window|User| count|
+--------------------+----+------+
|{2015-02-24 17:50...|   f|133623|
|{2015-02-24 18:30...|   f| 33366|
|{2015-02-24 20:20...|   e|126282|
|{2015-02-23 20:00...|   h| 94669|
|{2015-02-24 19:40...|   e| 67577|
+--------------------+----+------+
only showing top 5 rows
```

Event Time & Stateful Processing

Sliding Windows

In this example, we have 10-minute windows, starting every 5 minutes. Therefore, each event will fall into two different windows. Why did we do that?



Event Time & Stateful Processing

Sliding Windows

In this example, we have 10-minute windows, starting every 5 minutes. Therefore, each event will fall into two different windows. Why did we do that?

```
from pyspark.sql.functions import window, col
withEventTime.groupBy(window(col("event_time"), "10 minutes", "5 minutes"))\
.count()\
.writeStream\
.queryName("pyevents_per_sliding_window")\
.format("memory")\
.outputMode("complete")\
.start()
```

```
<pyspark.sql.streaming.StreamingQuery at 0x1325d7cd0>
```

```
spark.sql("SELECT * FROM pyevents_per_sliding_window").show(5)
```

```
+--------------------+------+
|              window| count|
+--------------------+------+
|{2015-02-23 19:45...|107668|
|{2015-02-24 17:20...|150773|
|{2015-02-24 18:30...|133323|
|{2015-02-22 06:05...|    35|
|{2015-02-23 18:00...|100853|
+--------------------+------+
only showing top 5 rows
```

Event Time & Stateful Processing

Watermarking for handling delays

Now, let us revisit our data centre example. Imagine that we frequently see some amount of delay from our customers in Latin America. Now, if we specify a watermark of 10 minutes, instruct Spark that any event that occurs more than 10 “event-time” minutes past a previous event should be ignored.

Now, with the application of watermarking, Structured Streaming will wait until 30 minutes after the final timestamp of this 10-minute rolling window before it finalizes the result of that window.

Event Time & Stateful Processing

Watermarking for handling delays

```
from pyspark.sql.functions import window, col
withEventTime\
.withWatermark("event_time", "30 minutes")\
.groupBy(window(col("event_time"), "10 minutes", "5 minutes"))\
.count()\
.writeStream\
.queryName("watermarking_events")\
.format("memory")\
.outputMode("complete")\
.start()
```

```
<pyspark.sql.streaming.StreamingQuery at 0x1319c96d0>
```

```
spark.sql("SELECT * FROM watermarking_events").show(5)
```

```
+--------------------+-----+
|              window|count|
+--------------------+-----+
|{2015-02-23 19:45...|26984|
|{2015-02-24 17:20...|37578|
|{2015-02-24 18:30...|33292|
|{2015-02-22 06:05...|    6|
|{2015-02-23 18:00...|25170|
+--------------------+-----+
only showing top 5 rows
```

Event Time & Stateful Processing

Dropping duplicates

Now, imagine that our goal here will be to de-duplicate the number of events per user by removing duplicate events. It is an extremely essential operation. Consider that an Internet of Things (IoT) applications that have upstream producers generating messages in nonstable network environments, and the same message might end up being sent multiple times.

Now, we must operate on a batch of records at a time in order to find duplicates—there's a high coordination overhead in the processing system.

Structured Streaming makes it easy to take message systems that provide at-least once semantics, and convert them into exactly-once by dropping duplicate messages as they come in. To de-duplicate data, Spark will maintain a number of user specified keys and ensure that duplicates are ignored.

Event Time & Stateful Processing

Dropping duplicates

Now, imagine that our goal here will be to de-duplicate the number of events per user by removing duplicate events. It is an extremely essential operation. Consider that an Internet of Things (IoT) applications that have upstream producers generating messages in nonstable network environments, and the same message might end up being sent multiple times.

Notice how we need to specify the event time column as a duplicate column along with the column you should de-duplicate. One core assumption is that duplicate events will have the same timestamp as well as identifier.

```
from pyspark.sql.functions import expr
withEventTime\
.withWatermark("event_time", "5 seconds")\
.dropDuplicates(["User", "event_time"])\
.groupBy("User")\
.count()\
.writeStream\
.queryName("pydeduplicated")\
.format("memory")\
.outputMode("complete")\
.start()
```

```
spark.sql("SELECT * FROM pydeduplicated").show(5)
```

```
+----+-----+
|User|count|
+----+-----+
|   a|80854|
|   b|91239|
|   c|77155|
|   g|91673|
|   h|77326|
+----+-----+
only showing top 5 rows
```

BTS | Barcelona Technology School

Production Grade Structured Streaming

Production Grade Structured Streaming

Fault Tolerance and Checkpointing

The most important operational concern for a streaming application is failure recovery. Failures can be due to many reasons such as if we're going to lose a machine in the cluster, or a schema will change by accident without a proper migration, or the user may even intentionally restart the cluster or application.

Structured Streaming allows us to recover an application by just restarting it. We must configure the application to use checkpointing and write-ahead logs, both of which are handled automatically by the engine.

We must configure a query to write to a checkpoint location on a reliable file system (e.g., HDFS, S3, or any compatible filesystem).

Periodically save all relevant progress information (for instance, the range of offsets processed in a given trigger) as well as the current intermediate state values to the checkpoint location

In case of a failure scenario, we simply need to restart your application, making sure to point to the same checkpoint location, and it will automatically recover its state and start processing data where it left off.

Production Grade Structured Streaming

Fault Tolerance and Checkpointing

The most important operational concern for a streaming application is failure recovery. Therefore, Spark provides us with the option to control how to recover our processing once a fault is encountered.

```
# Checkpointing
static = spark.read.json("./activity-data")
streaming = spark\
.readStream\
.schema(static.schema)\
.option("maxFilesPerTrigger", 10)\
.json("./activity-data")\
.groupBy("gt")\
.count()

query = streaming\
.writeStream\
.outputMode("complete")\
.option("checkpointLocation", "./checkpointing/")\
.queryName("test_python_stream")\
.format("memory")\
.start()
```

Production Grade Structured Streaming

Updating Application

Following are the kind of updates:

- **Updates in Spark Versioning**:

  Structured Streaming applications *should be able to restart from an old checkpoint directory across patch version updates to Spark* (e.g., moving from Spark 2.2.0 to 2.2.1 to 2.2.2).

  The checkpoint format is designed to be forward-compatible, so the only way it may be broken is due to critical bug fixes. If a Spark release cannot recover from old checkpoints, this will be clearly documented in its release notes.

  The Structured Streaming developers also aim to keep the checkpoint format compatible across minor version updates (e.g., Spark 2.2.x to 2.3.x), but you should check the release notes to see whether this is supported for each upgrade. In either case, *if you cannot start from a checkpoint, you will need to start your application again using a new checkpoint directory*.

- **Updates in Streaming Application Code**
- **Resizing and Rescaling Application**

Production Grade Structured Streaming

Updating Application

Following are the kind of updates:

- **Updates in Spark Versioning**
- **Updates in Streaming Application Code**:

  Structured Streaming is designed to allow certain types of changes to the application code between application restarts.

  Most importantly, *you are allowed to change user-defined functions (UDFs) as long as they have the same type signature*. This feature can be very useful for bug fixes. For example, imagine that your application starts receiving a new type of data, and one of the data parsing functions in your current logic crashes. With Structured Streaming, you can recompile the application with a new version of that function and pick up at the same point in the stream where it crashed earlier.

  While *small adjustments like adding a new column or changing a UDF are not breaking changes and do not require a new checkpoint directory*, there are larger changes that do require an entirely new checkpoint directory. For example, *if you update your streaming application to add a new aggregation key or fundamentally change the query itself, Spark cannot construct the required state for the new query from an old checkpoint directory*. In these cases, Structured Streaming will throw an exception saying it cannot begin from a checkpoint directory, and you must start from scratch with a new (empty) directory as your checkpoint location.

- **Resizing and Rescaling Application**

Production Grade Structured Streaming

Updating Application

Following are the kind of updates:

- **Updates in Spark Versioning**
- **Updates in Streaming Application Code**
- **Resizing and Rescaling Application**:

  In general, the *size of your cluster should be able to comfortably handle bursts above your data rate*. The key metrics you should be monitoring in your application and cluster are discussed as follows-

  If you see that your *input rate is much higher than your processing rate (elaborated upon momentarily), it's time to scale up your cluster or application*. Depending on your resource manager and deployment, you may just be able to dynamically add executors to your application.

  If needed, *you can scale-down your application in the same way- remove executors (potentially through your cloud provider) or restart your application with lower resource counts*. These changes will likely incur some processing delay (as data is recomputed or partitions are shuffled around when executors are removed).

  While making underlying infrastructure changes to the cluster or application are sometimes necessary, other times a change may only require a restart of the application or stream with a new configuration. For instance, changing *spark.sql.shuffle.partitions* is not supported while a stream is currently running (it won't actually change the number of shuffle partitions). This requires restarting the actual stream, not necessarily the entire application.

Production Grade Structured Streaming

Metrics and Monitoring

There are two key APIs we can leverage to query the status of a streaming query and see its recent execution progress. With these two APIs, we can get a sense of whether or not your stream is behaving as expected.

- **Query Status**:
  It is the most fundamental monitoring API, as well a great starting point.

  It aims to answer the question, “What processing is my stream performing right now?”

  This information is reported in the *status field of the query object returned by startStream*. For example, you might have a simple counts stream that provides counts of IOT devices defined by the following query (here we’re just using the same query from the previous chapter without the initialization code)

- **Recent Progress**

Production Grade Structured Streaming

Metrics and Monitoring

There are two key APIs we can leverage to query the status of a streaming query and see its recent execution progress. With these two APIs, we can get a sense of whether or not your stream is behaving as expected.

- **Query Status**:
  It is the most fundamental monitoring API, as well a great starting point.

  It aims to answer the question, “What processing is my stream performing right now?”

  This information is reported in the *status field of the query object returned by startStream*. For example, you might have a simple counts stream that provides counts of IOT devices defined by a query (here we’re just using the same query from the previous slides without the initialization code)

```
query.status
```

```
{'message': 'Waiting for data to arrive',
 'isDataAvailable': False,
 'isTriggerActive': False}
```

- **Recent Progress**

Production Grade Structured Streaming

Metrics and Monitoring

There are two key APIs we can leverage to query the status of a streaming query and see its recent execution progress. With these two APIs, we can get a sense of whether or not your stream is behaving as expected.

- **Query Status**
- **Recent Progress**:

  While the query's current status is useful to see, equally important is an ability to view the query's progress.

  The progress API allows us to answer questions like "At what rate am I processing tuples?" or "How fast are tuples arriving from the source?"

  By *running query.recentProgress, you'll get access to more time-based information like the processing rate and batch durations*. The streaming query progress also includes information about the input sources and output sinks behind your stream.

Production Grade Structured Streaming

Metrics and Monitoring

There are two key APIs we can leverage to query the status of a streaming query and see its recent execution progress. With these two APIs, we can get a sense of whether or not your stream is behaving as expected.

- **Query Status**
- **Recent Progress**:

  While the query's current status is useful to see, equally important is an ability to view the query's progress.

  The progress API allows us to answer questions like "At what rate am I processing tuples?" or "How fast are tuples arriving from the source?"

  By *running query.recentProgress, you'll get access to more time-based information like the processing rate and batch durations*. The streaming query progress also includes information about the input sources and output sinks behind your stream.

```
query.recentProgress[1]
```

```
{'id': '6556e144-3df9-4473-9e0a-26c18e5da84b',
 'runId': '5dfc0d17-2eaf-4011-8c9e-33b804ff5e99',
 'name': 'test_python_stream',
 'timestamp': '2021-05-31T02:25:17.391Z',
 'batchId': 8,
 'numInputRows': 0,
 'inputRowsPerSecond': 0.0,
 'processedRowsPerSecond': 0.0,
 'durationMs': {'latestOffset': 12, 'triggerExecution': 12},
 'stateOperators': [{'numRowsTotal': 7,
   'numRowsUpdated': 0,
   'memoryUsedBytes': 82024,
   'numRowsDroppedByWatermark': 0,
   'customMetrics': {'loadedMapCacheHitCount': 2800,
    'loadedMapCacheMissCount': 0,
    'stateOnCurrentVersionSizeBytes': 18896}}],
 'sources': [{'description': 'FileStreamSource[file:/Users/ankitbit/Documents/Lectures/Real-Time-Data-Analysis/Tuto
rials/activity-data]',
   'startOffset': {'logOffset': 7},
   'endOffset': {'logOffset': 7},
   'numInputRows': 0,
   'inputRowsPerSecond': 0.0,
   'processedRowsPerSecond': 0.0}],
 'sink': {'description': 'MemorySink', 'numOutputRows': 0}}
```

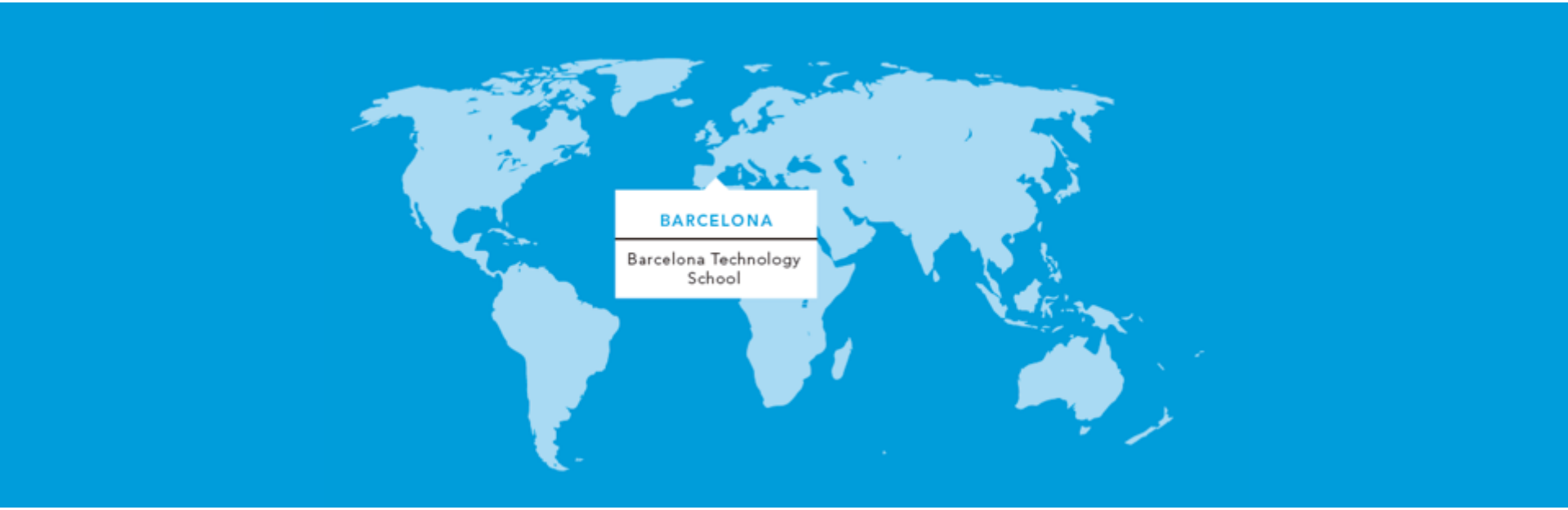
Production Grade Structured Streaming

Alerting

Understanding and looking at the metrics for your Structured Streaming queries is an important first step. However, this involves constantly watching a dashboard or the metrics in order to discover potential issues. Therefore, *there is a need to have a robust automatic alerting system to notify you when your jobs are failing or not keeping up with the input data rate without monitoring them manually*.

There are several ways to integrate existing alerting tools with Spark, generally building on the recent progress API we just discussed.

For example, you may directly feed the metrics to a monitoring system such as the open source Coda Hale Metrics library or Prometheus, or you may simply log them and use a log aggregation system like Splunk. In addition to monitoring and alerting on queries, we will also be willing to monitor and alert on the state of the cluster and the overall application (if you're running multiple queries together).


BARCELONA
Barcelona Technology School