# Intro to Structured Streaming

# Why do we need stream processing?

- Streaming data can come in faster than it can be consumed when using traditional batch-related processing techniques.
  - Bank transactions
  - IoT devices
  - Online games
  - …
- A stream of data is treated as a table to which data is continuously appended.

# High level components of a streaming system

- Input source
  - Kafka
  - **Azure Event Hubs**
  - IoT Hub
  - Network sockets
  - ...
- Stream processing
  - **Structured Streaming**
  - forEach sinks
  - Memory sinks
  - ...



Antra

# Azure Databricks structured streaming

- Fast, scalable, fault tolerant stream processing API
- **Near** real-time analytics on streaming data
- Processing streaming data is very similar to processing static data
- The API continuously increments and updates the final data
- Can work with Azure Event Hubs and analyze data using structured streaming query and Spark SQL

# Spark uses a single API to handle batch and streaming data..

## Batch ETL with DataFrames

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

Read from Json file

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

Select some devices

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

Write to parquet file

## Streaming ETL with DataFrames

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

Read from Json file stream
Replace `load()` with `stream()`

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

Select some devices
Code does not change

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

Write to Parquet file stream
Replace `save()` with `startStream()`



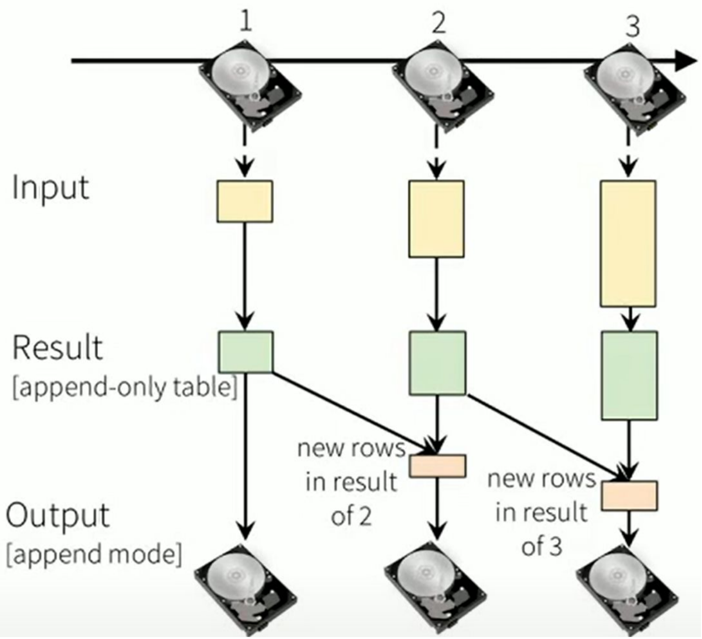
Antra

- read...stream() creates a streaming dataframe, but does not start any of the computation
- write...startStream() defines where and how to output the data and starts the processing
  - Basically a query that runs repeatedly, updating the output each time

# Append mode

- This is a simple query without any aggregation..

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

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

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



# What if we have Aggregations?

- Continuous windowed aggregations

```
input.avg("signal")
```

Continuously compute *average* signal *across all devices*

```
input.groupBy("device-type")
     .avg("signal")
```

Continuously compute *average* signal of *each type of device*

- Continuous windowed aggregations

```
input.groupBy(
        $"device-type",
        window($"event-time-col", "10 min"))
     .avg("signal")
```

Continuously compute *average* signal of *each type of device* in last 10 minutes using *event-time*

# Output Modes

- Different output modes make sense for different queries
- **Append mode** for **non-aggregation queries**

```
input.select("device", "signal")
      .write
      .outputMode("append")
      .format("parquet")
      .startStream("dest-path")
```

- **Complete mode** for **aggregation queries**

```
input.agg(count("*"))
      .write
      .outputMode("complete")
      .format("parquet")
      .startStream("dest-path")
```

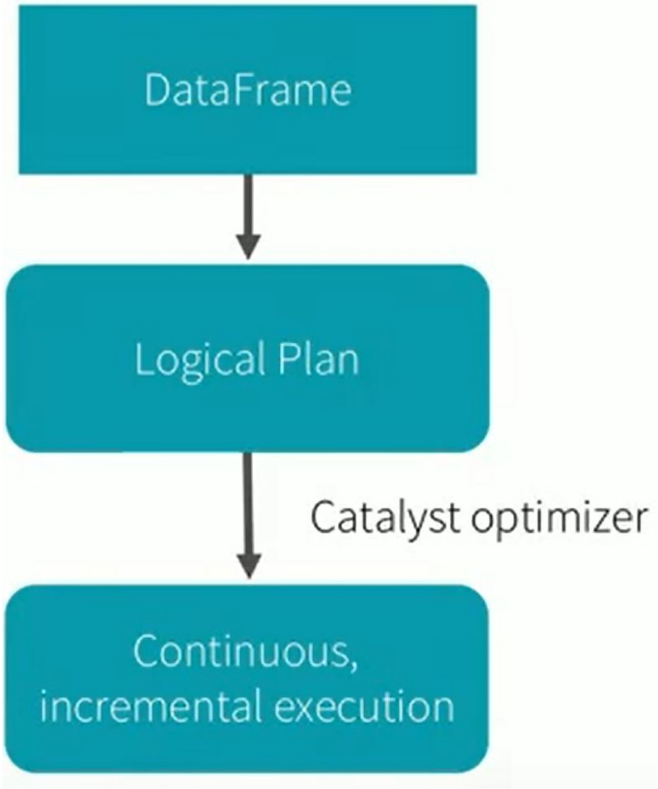
# Query Management

- Multiple queries can be active at the same time
- We can assign a query to a variable and manage it. We can:
  - stop execution, wait for it to terminate
  - Get its status
  - Get the exception if there is an error

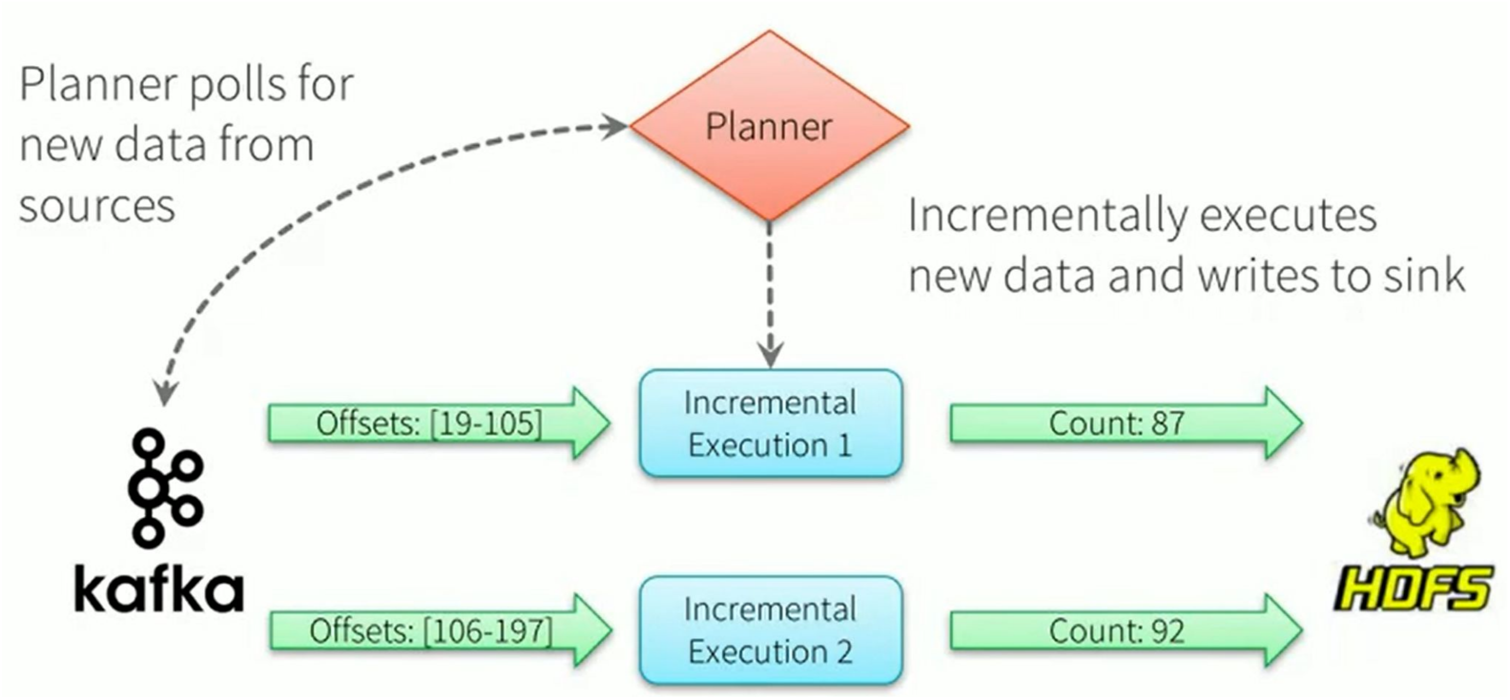
```
query = result.write
      .format("parquet")
      .outputMode("append")
      .startStream("dest-path")


query.stop()
query.awaitTermination()
query.exception()

query.sourceStatuses()
query.sinkStatus()
```

# How are the queries executed?

- **Logically**, we are writing queries as if we are working on a static table
- **Physically**, Spark automatically runs the query *incrementally* and *continuously*

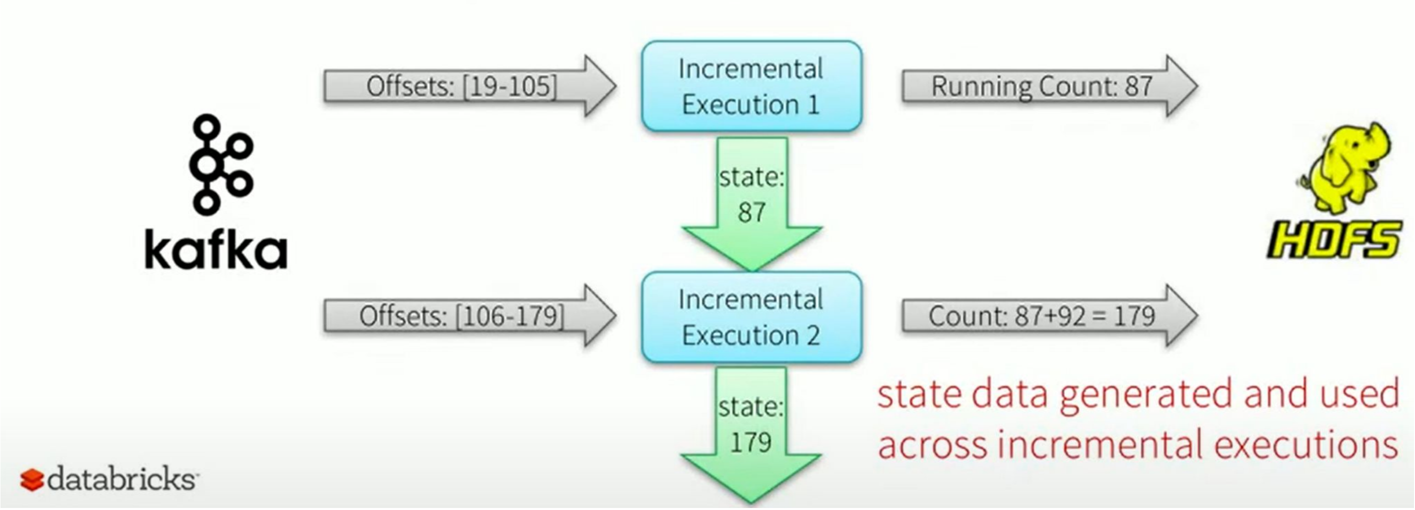


# How are the queries executed?



# How are the queries executed?

- Previous result is stored in memory and used for the next execution



# How is fault-tolerance achieved?

- Everytime the planner gets a offset of data, the offset is first stored before any execution
  - If execution fails, we can read back the offset and rerun the execution
    - This relies on the datasource to be able to replay the data based on the offset
    - By design, structured streaming sources are replayable(Kafka, kinesis, file...)
- Planner also makes sure to use the correct version of in-memory state for the execution(for aggregations)
- Sinks, by design, will make sure to handle re-executions correctly and avoid double committing the output.
- **Offset tracking + in-memory state management + fault-tolerant sources and sinks = end-to-end fault tolerance**

# Azure Event Hubs

- A scalable real-time data ingestion service
- Can receive large amounts of data from multiple sources and stream the prepared data to ADLS or Blob storage
- Can be integrated with  Spark Structured Streaming to perform stream processing

# Creating Event Hubs

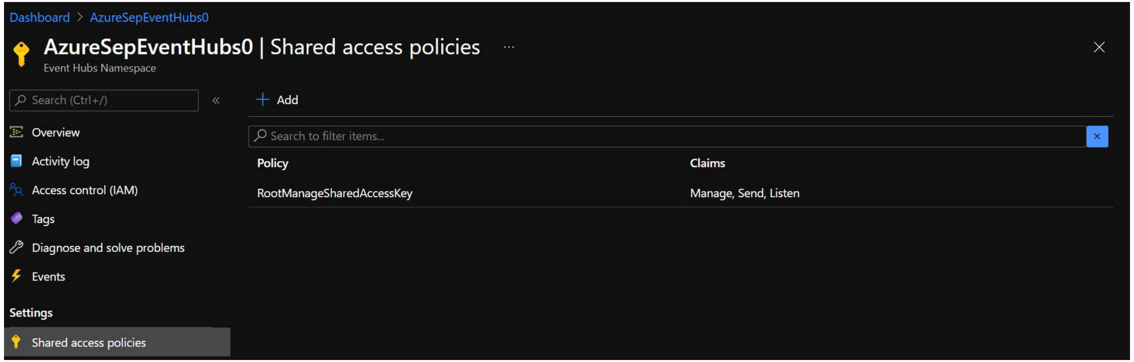
- Click “Create a resource” in the portal
- Search “Event Hubs” and create the resource
- Locate the service within the resource group you selected
- Click “+ Event Hub”

# Settings in Event Hub creation

- Partition count
  - At least one partition per Throughput Unit(TU)
  - TUs can be scaled independently from the partition count
  - Cost is calculated based on the TUs only and not on the number of partitions
  - More partitions make the processing more complex, and it's only beneficial when you have tens of substreams.
- Message Retention
  - Locked to 1 day for Basic tier namespace
  - Up to 7 days for Standard tier
- Capture
  - Not available for Basic tier
  - Specify a minimum size and time window to perform the capture
  - Captured data can be automatically saved to a Blob or ADLS account of your choice
  - Data will be in Apache Avro format

# Getting the connection string

- After the creation is finished, you can access the connection string from "Shared access policies"



# Reading Streams

- **SparkSession.readStream** returns a **DataStreamReader** used to configure the stream
  - Recall **SparkSession.read**, which returns a **DataFrameReader** that can be used to read data in as a DataFrame.
- Key things to configure:
  - Schema
    - Predefined for you in some Pub/Sub sources like Kafka and Event Hubs
    - User-defined only(no inferencing) for file-based streaming
  - Stream type
    - Kafka
    - Files
    - TCP/IP
  - Stream type specific configurations

# Reading Streams

- Creating a streaming df is very similar to creating a static df, except we have to specify the schema

Cmd 12

```
streaming_df = (spark
                .readStream
                .option("maxFilesPerTrigger",1)
                .schema(schema)
                .json(file_path)
               )
```

- We can check if a df is streaming with **df.isStreaming**, which returns **True** if the df is a streaming df(vs. A static df)

# Operations on Streaming Dataframes

- Most operations are identical to operations on a static df
- Some operations are not supported
  - Sorting -> only supported after aggregation and in Complete Output Mode
  - Limit / take(n)
  - Distinct
  - Certain types of joins
  - …
  - Check doc for more details: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#unsupported-operations

# Join types supported

| **Left Input** | **Right Input** | **Join Type** | |
|---|---|---|---|
| Static | Static | All types | Supported, since its not on streaming data even though it can be present in a streaming query |
| Stream | Static | Inner | Supported, not stateful |
| | | Left Outer | Supported, not stateful |
| | | Right Outer | Not supported |
| | | Full Outer | Not supported |
| | | Left Semi | Supported, not stateful |
| Static | Stream | Inner | Supported, not stateful |
| | | Left Outer | Not supported |
| | | Right Outer | Supported, not stateful |
| | | Full Outer | Not supported |
| | | Left Semi | Not supported |
| Stream | Stream | Inner | Supported, optionally specify watermark on both sides + time constraints for state cleanup |
| | | Left Outer | Conditionally supported, must specify watermark on right + time constraints for correct results, optionally specify watermark on left for all state cleanup |
| | | Right Outer | Conditionally supported, must specify watermark on left + time constraints for correct results, optionally specify watermark on right for all state cleanup |
| | | Full Outer | Conditionally supported, must specify watermark on one side + time constraints for correct results, optionally specify watermark on the other side for all state cleanup |
| | | Left Semi | Conditionally supported, must specify watermark on right + time constraints for correct results, optionally specify watermark on left for all state cleanup |

# Writing a Stream

- **DataFrame.writeStream** returns a **DataStreamWriter** used to configure the output of the stream.