Structured Streaming

Why

Read first streams

Structured Streaming I/O

High-level API

- ease of development
- interoperable with other Spark APIs
- auto-optimizations

Structured Streaming Principles

Lazy evaluation

Transformations and Actions

- transformations describe of how new DFs are obtained
- actions start executing/running Spark code

Input sources

- Kafka, Flume
- a distributed file system
- sockets

Output sinks

- a distributed file system
- databases
- Kafka
- testing sinks e.g. console, memory

Streaming I/O

Output modes

- append = only add new records
- update = modify records in place  if query has no aggregations, equivalent with append
- complete = rewrite everything

Not all queries and sinks support all output modes

- example: aggregations and append mode

Triggers = when new data is written

- default: write as soon as the current micro-batch has been processed
- once: write a single micro-batch and stop
- processing-time: look for new data at fixed intervals
- continuous (currently experimental)

Takeaways

Streaming DFs can be read via the Spark session

```
val lines = spark.readStream // instead of read
     .format("socket")
     .option("host", "localhost")
     .option("port", 12345)
     .schema(mySchema)
     .load()
```

Streaming DFs have identical API to non-streaming DFs

```
val query = persons.select(
          col("person.name").as("name"),
          col("person.age").as("age")
     )
```

Streaming DFs can be written via a call to start

```
val writer = query.writeStream
     .format("console")
     .outputMode("complete")
     .start()

// need to wait for the stream to finish
writer.awaitTermination()
```

Spark rocks