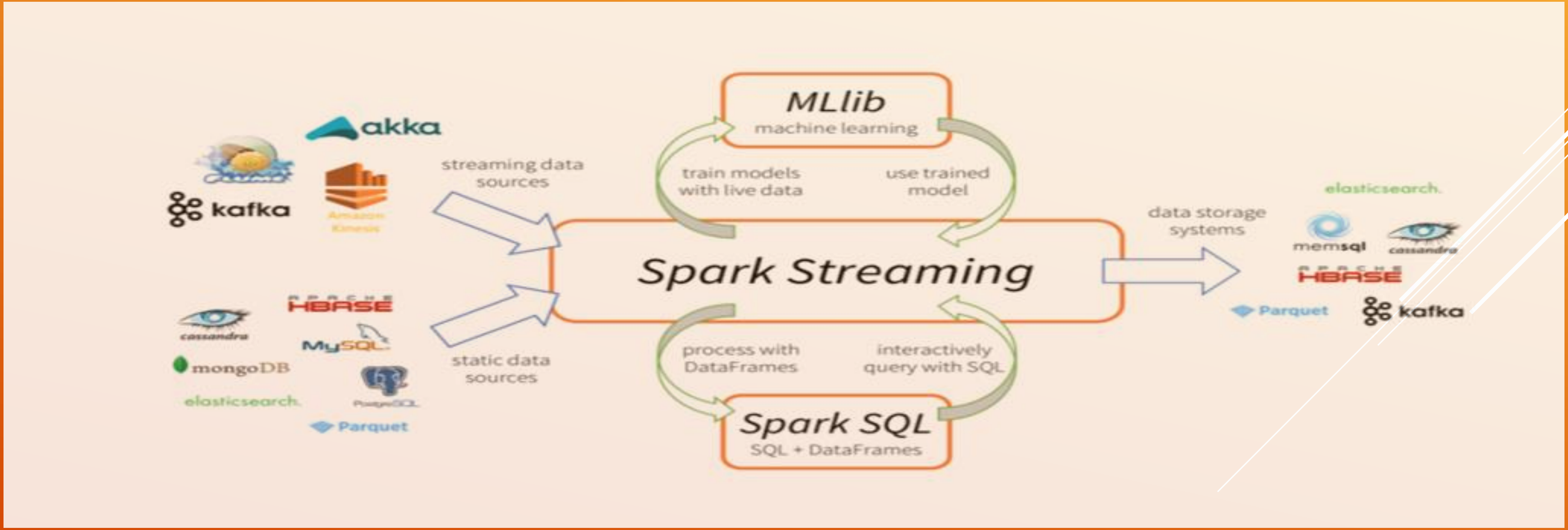
Spark Streaming

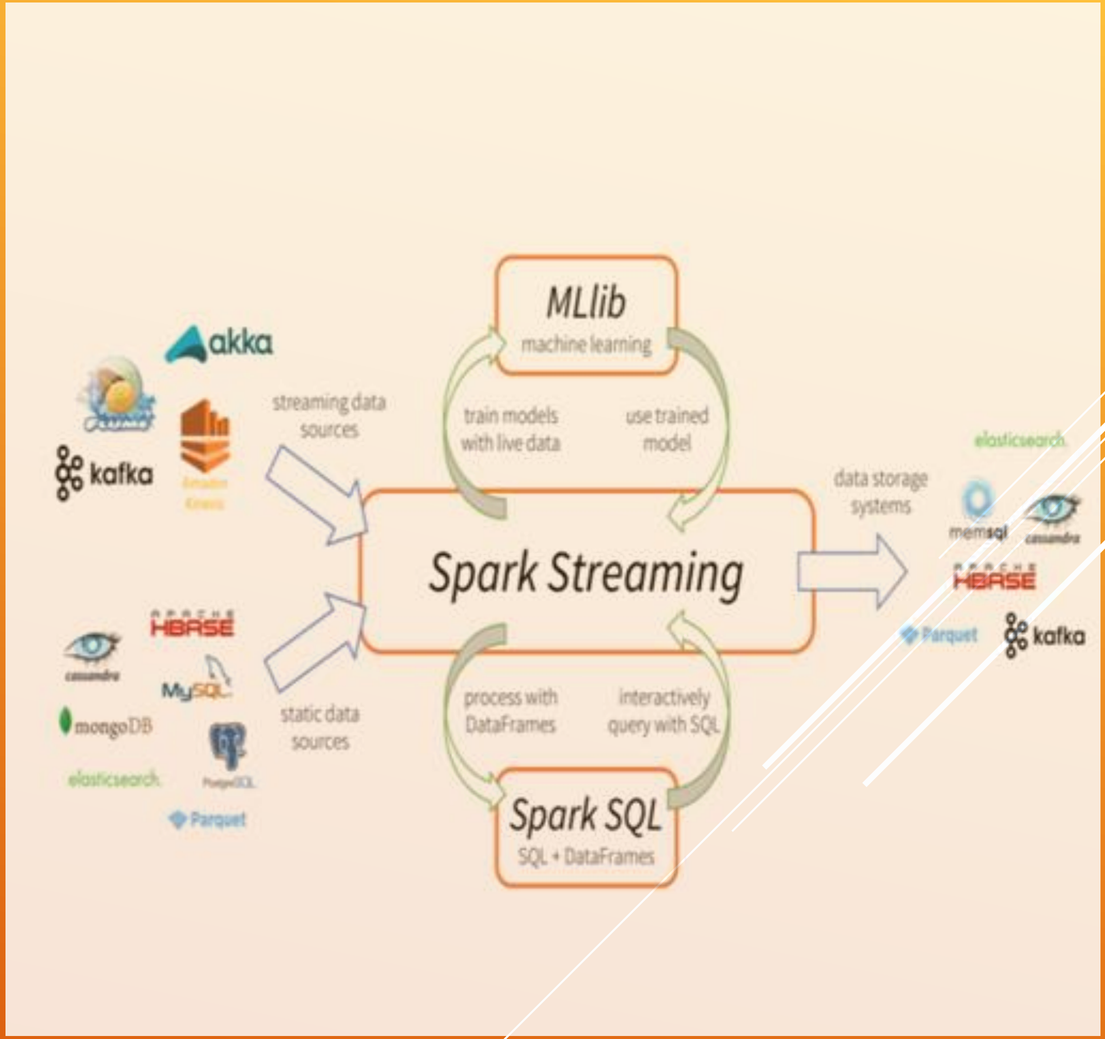
The Need for Streaming Architecture

- “Every company is now a software company” equates to companies collecting more data than ever and wanting to get value from that data in real-time.



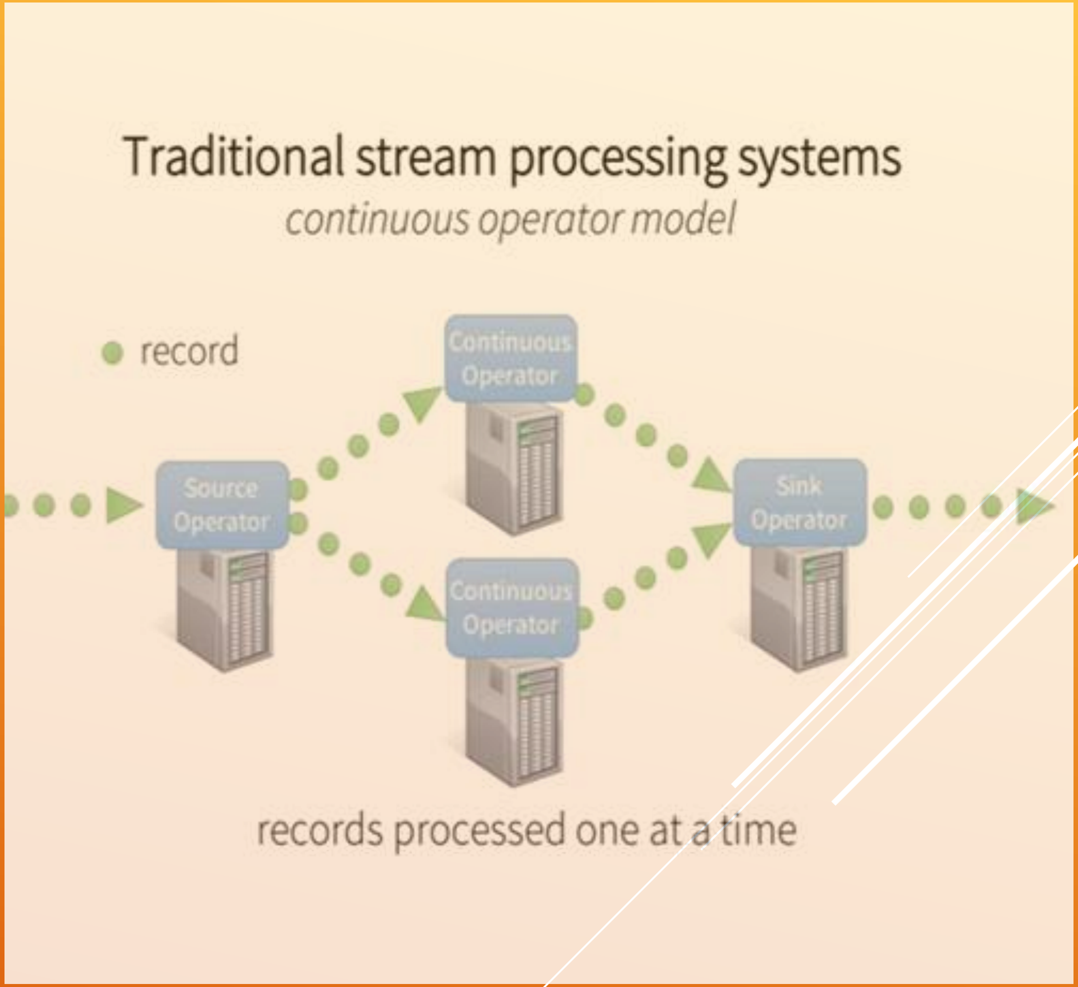
The Need for Streaming Architecture

- Sensors, IoT devices, social networks, and online transactions are all generating data that needs to be monitored constantly and acted upon quickly.
- Making a purchase online means that all the associated data (e.g. date, time, items, price) need to be stored and made ready for organizations to analyze and make prompt decisions based on the customer's behavior.
- Fraudulent bank transactions requires testing transactions against pre-trained fraud models as the transactions occur (i.e. as data streams) to quickly stop fraud in its track.



Stream Processing Architecture

- At high level, modern distributed stream processing pipelines execute as follows:
- Receive streaming data from data sources (e.g. live logs, system telemetry data, IoT device data, etc.) into some data ingestion system like Apache Kafka, Amazon Kinesis, etc.
- Process the data in parallel on a cluster. This is what stream processing engines are designed to do, as we will discuss this in detail in the next coming sessions.
- Output - The results out to downstream systems like HBase, Cassandra, Kafka, etc.


Traditional stream processing systems
continuous operator model
record
Source Operator
Continuous Operator
Continuous Operator
Sink Operator
records processed one at a time

Challenges

- Continuous operators are a simple and natural model. However, with today’s trend towards large-scale and more complex real-time analytics, this traditional architecture has also met some challenges. We designed Spark Streaming to satisfy the following requirements:
- Fast Failure and Straggler Recovery – With greater scale, there is a higher likelihood of a cluster node failing or unpredictably slowing down (i.e. Stragglers). The system must be able to automatically recover from failures and stragglers to provide results in real-time. Unfortunately, the static allocation of continuous operators to worker nodes makes it challenging for traditional systems to recover quickly from faults and stragglers.
- Load balancing – Uneven allocation of the processing load between the workers can cause bottlenecks in a continuous operator system. This is more likely to occur in large clusters and dynamically varying workloads. The system needs to be able to dynamically adapt the resource allocation based on the workload.

Challenges

- Unification of Streaming, Batch and Interactive Workloads – In many use cases, it is also attractive to query the streaming data interactively (after all, the streaming system has it all in memory), or to combine it with static datasets (e.g. pre-computed models). This is hard in continuous operator systems as they are not designed to the dynamically introduce new operators for ad-hoc queries. This requires a single engine that can combine batch, streaming and interactive queries.
- Advanced Analytics like Machine Learning and SQL Queries – More complex workloads require continuously learning and updating data models, or even querying the “latest” view of streaming data with SQL queries. Again, having a common abstraction across these analytic tasks makes the developer’s job much easier.

Basic Sources

- In the session Spark Streaming, Stream from text data received over a TCP socket connection. Besides sockets, the StreamingContext API provides methods for creating Streams from files and Akka actors as input sources.
- • File Streams: For reading data from files on any file system compatible with the HDFS API (that is, HDFS, S3, NFS, etc.)

Receiver Reliability

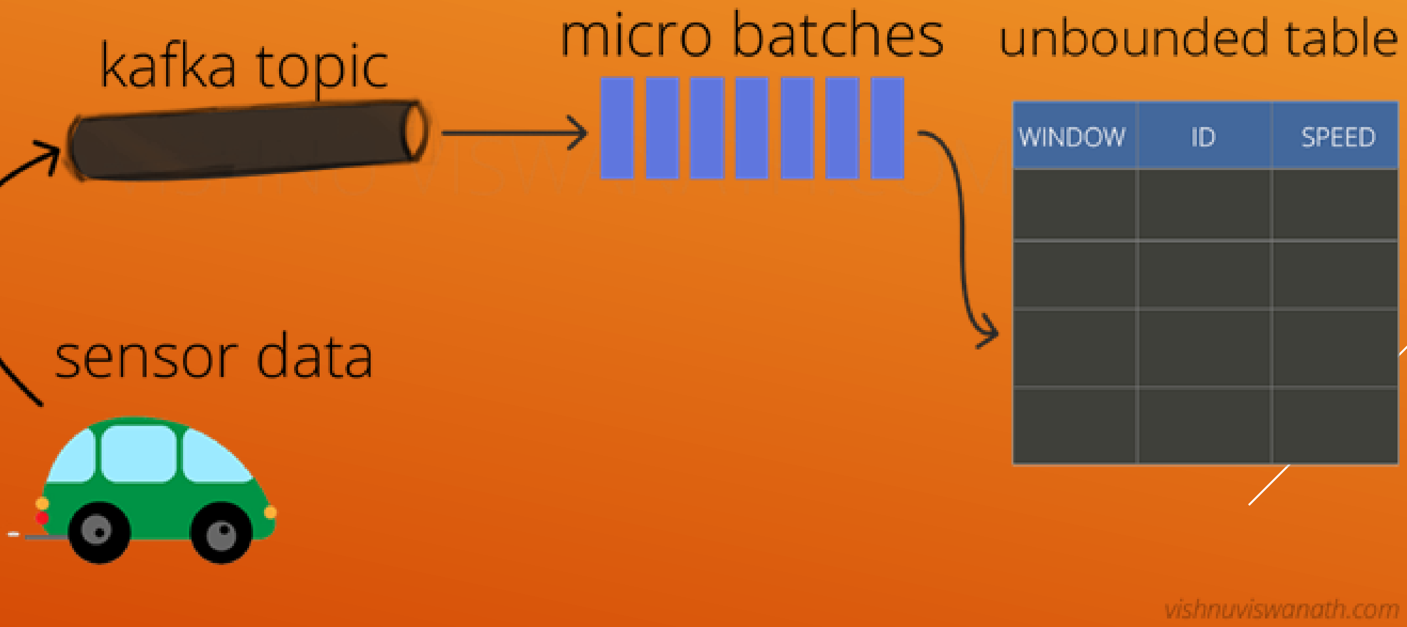
- There are two kinds of data sources based on their reliability.
- Sources like Kafka and Flume allow the transferred data to be acknowledged.
- If the system receiving data from these reliable sources acknowledges the received data correctly, it can be ensured that no data will be lost due to any kind of failure. This leads to two kinds of receiver:
  - Reliable Receiver – Correctly sends acknowledgment to a reliable source when the data has been received and stored in Spark with Replication
  - Unreliable Receiver – Doesn’t send acknowledgement to a source. This can be used for sources that do not support acknowledgement, or even for reliable sources when one does not want or need to go into the complexity of acknowledgement.

Performance Tuning

- Getting the best performance out of a Spark Streaming application on a cluster requires a bit of tuning. At a high level, you need to consider two things:
  - Reducing the processing time of each batch of data by efficiently using the cluster resources
  - Setting the right batch size such that batches of data can be processed as fast as they are received (that is, data processing keeps up with the data ingestion)

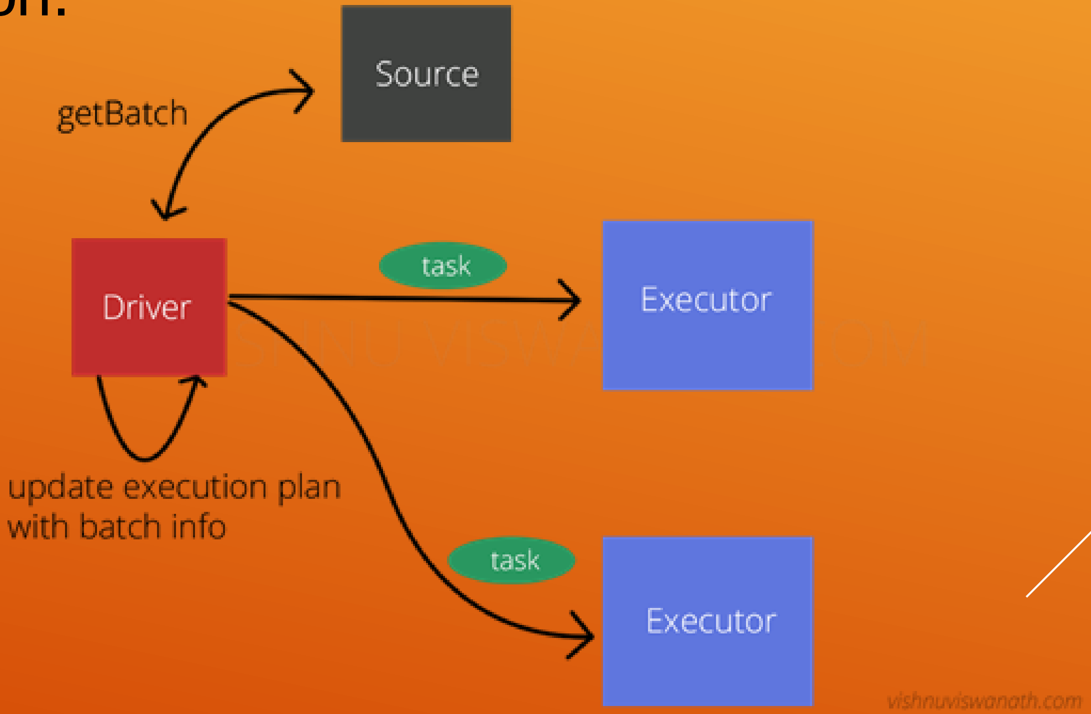
SPARK STRUCTURED STREAMING

- Structured Streaming is Apache Spark’s streaming engine which can be used for doing near real-time analytics. Lets explore Structured Streaming by going through a very simple use case. Imagine you started a ride hailing company and need to check if the vehicles are over-speeding. We will create a simple near real-time streaming application to calculate the average speed of vehicles every few seconds



MICRO BATCH BASED STREAMING

- Structured Streaming in Spark, uses **micro-batching** to do streaming. That is, spark waits for a very small interval say 1 second (or even 0 seconds - i.e., as soon as possible) and batches together all the events that were received during that interval into a micro batch. This micro batch is then scheduled by the Driver to be executed as Tasks at the Executors. After a micro-batch execution is complete, the next batch is collected and scheduled again. This scheduling is done frequently to give an impression of streaming execution.



CONTINUOUS

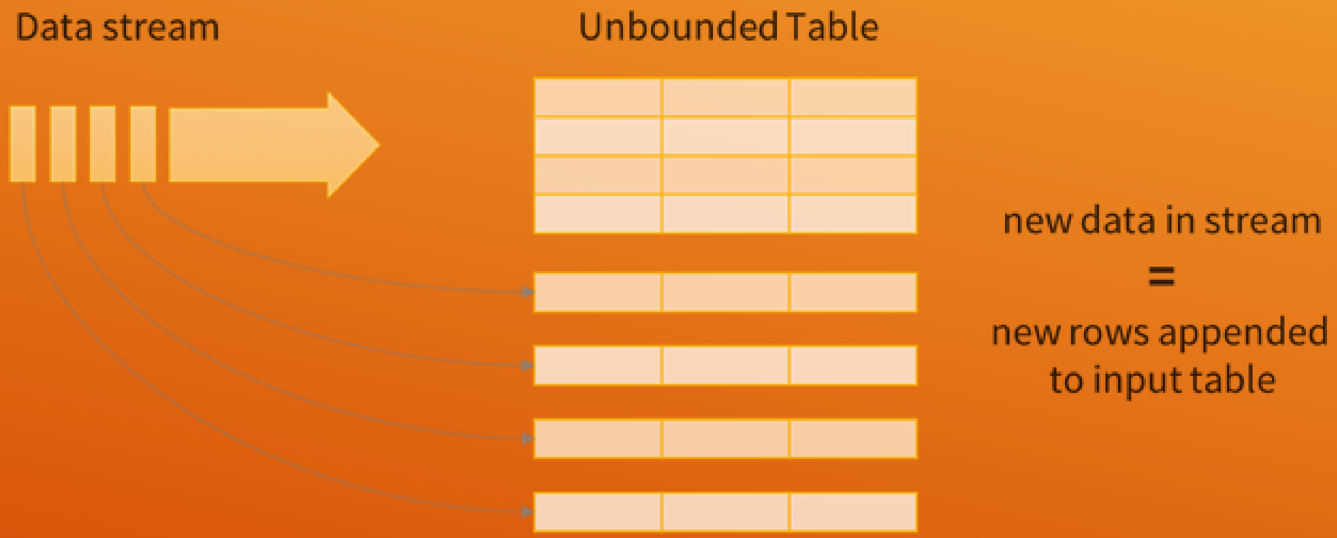
- In version 2.3, Spark released a new execution engine called Continuous Processing, which does not do micro-batching. Instead, it launches long runnings tasks that read and process incoming data continuously.

```
.writeStream
    .format("kafka")

.option("kafka.bootstrap.servers","localhost:9092")
    .option("topic", "fastcars")
    .option("checkpointLocation",
"/tmp/sparkcheckpoint/")
    .queryName("kafka spark streaming kafka")
    .outputMode("update")
    .trigger(Trigger.Continuous("10 seconds")) //10
seconds is the checkpoint interval.
    .start()
```

MODEL DETAILS

- Conceptually, Structured Streaming treats all the data arriving as an unbounded **input table**. Each new item in the stream is like a row appended to the input table. We won’t actually retain all the input, but our results will be equivalent to having all of it and running a batch job.



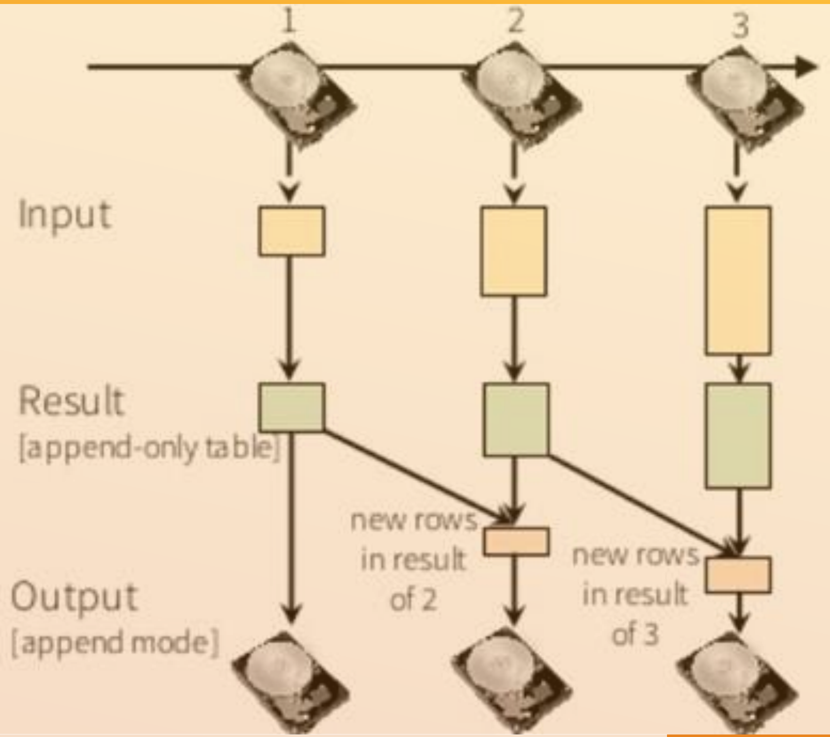
Data stream as an unbounded Input Table

READ , PROCESS, WRITE

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

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

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



```
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*

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

Continuously compute *average* signal *across all devices*

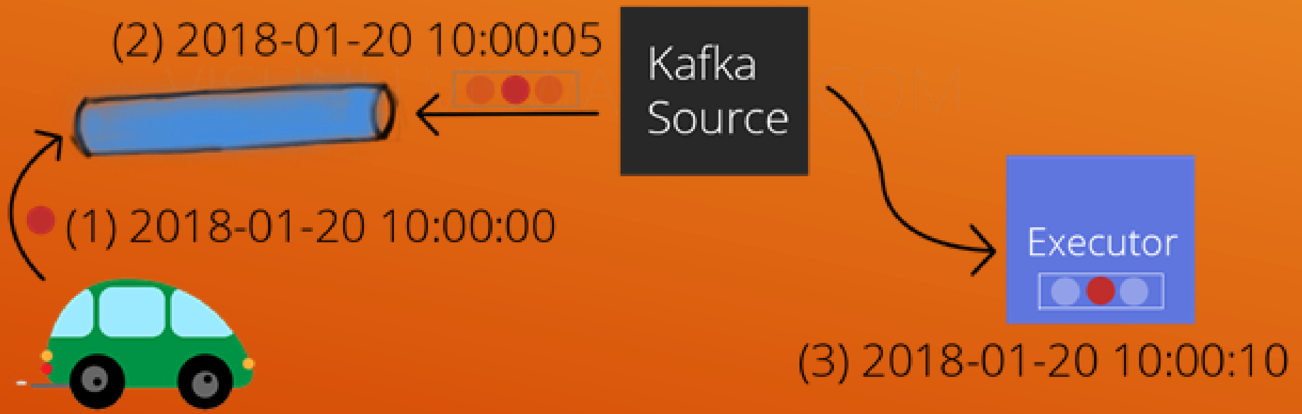
```
input.groupBy("device-type")
    .avg("signal")
```

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

EVENT TIME & PROCESSING TIME

EventTime is the time at which an event is generated at its source, whereas a ProcessingTime is the time at which that event is processed by the system. There is also one more time which some stream processing systems account for, that is IngestionTime - the time at which event/message was ingested into the System. It is important to understand the difference between EventTime and ProcessingTime.

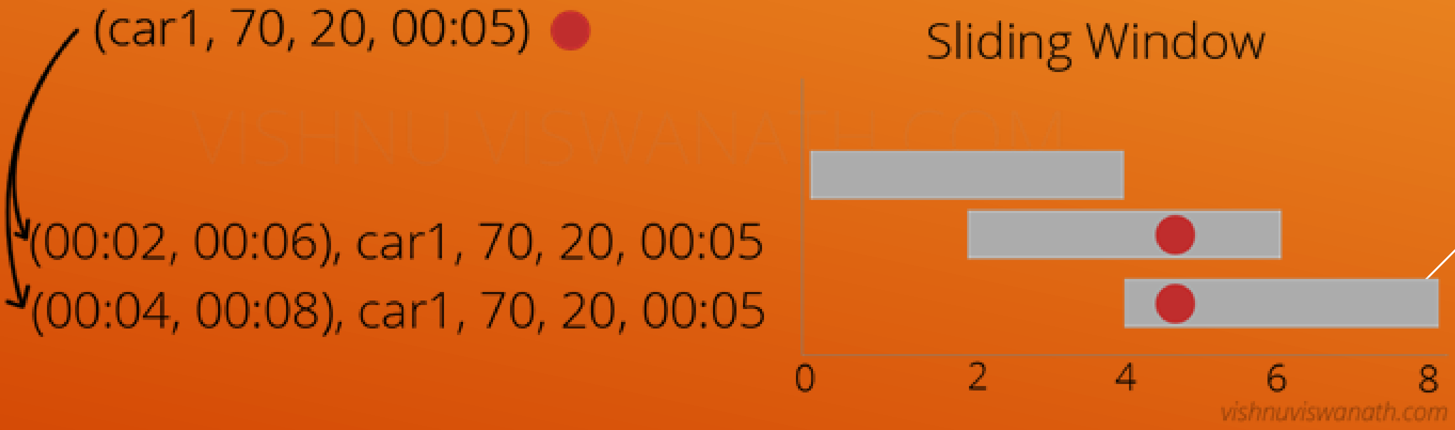
The red dot in the above image is the message, which originates from the vehicle, then flows through the Kafka topic to Spark's Kafka source and then reaches executor during task execution. There could be a slight delay (or maybe a long delay if there is any network connectivity issue) between these points. The time at the source is what is called an EventTime, the time at the executor is what is called the ProcessingTime. You can think of the ingestion time as the time at when it was first read into the system at the Kafka source (IngestionTime is not relevant for spark).



TUMBLING WINDOW & SLIDING WINDOW

A tumbling window is a non-overlapping window, that tumbles over every “window-size”. e.g., for a Tumbling window of size 4 seconds, there could be window for [00:00 to 00:04), [00:04: 00:08), [00:08: 00:12) etc (ignoring day, hour etc here). If an incoming event has EventTime 00:05, that event will be assigned the window - [00:04 to 00:08)

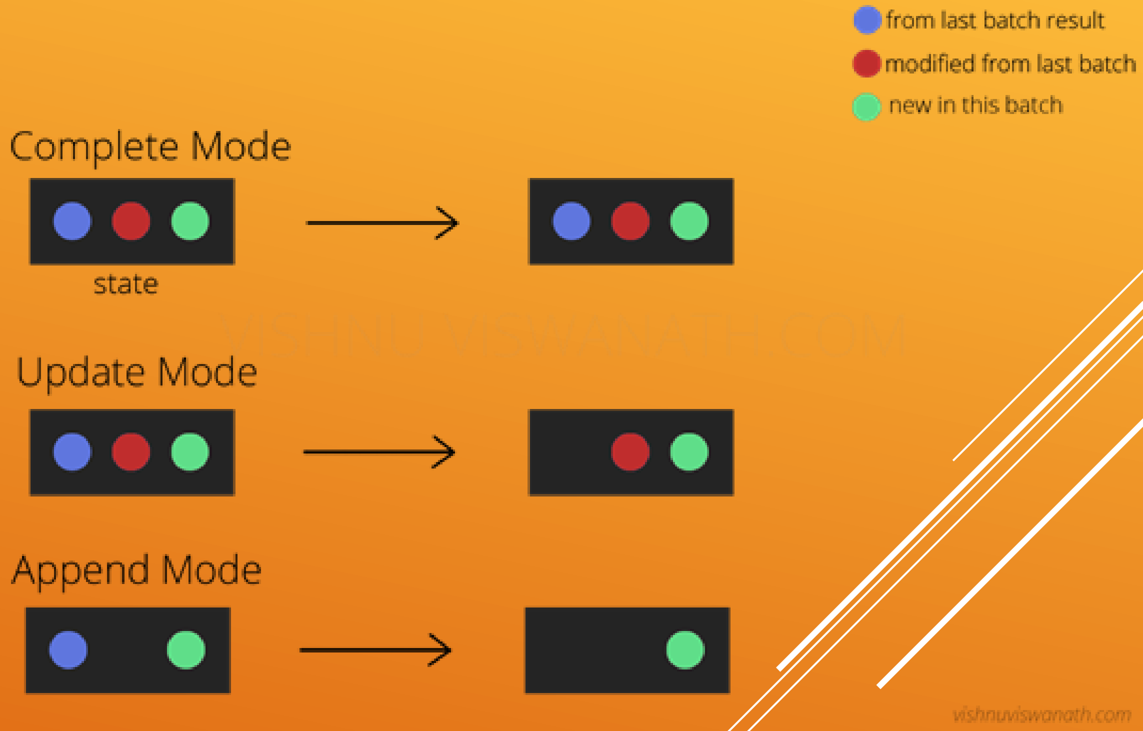
A SlidingWindow is a window of a given size(say 4 seconds) that slides every given interval (say 2 seconds). That means a sliding window could overlap with another window. For a window of size 4 seconds, that slides every 2 seconds there could windows [00:00 to 00:04), [00:02 to 00:06), [00:04 to 00:08) etc. Notice that the windows 1 and 2 are overlapping here. If an event with EventTime 00:05 comes in, that event will belong to the windows [00:02 to 00:06) and [00:04 to 00:08).


(car1, 70, 20, 00:05)
Sliding Window
(00:02, 00:06), car1, 70, 20, 00:05
(00:04, 00:08), car1, 70, 20, 00:05
0
2
4
6
8
vishnuviswanath.com

OUTPUT MODES

The last part of the model is **output modes**. Each time the result table is updated, the developer wants to write the changes to an external system, such as S3, HDFS, or a database. We usually want to write output incrementally. For this purpose, Structured Streaming provides three output modes:

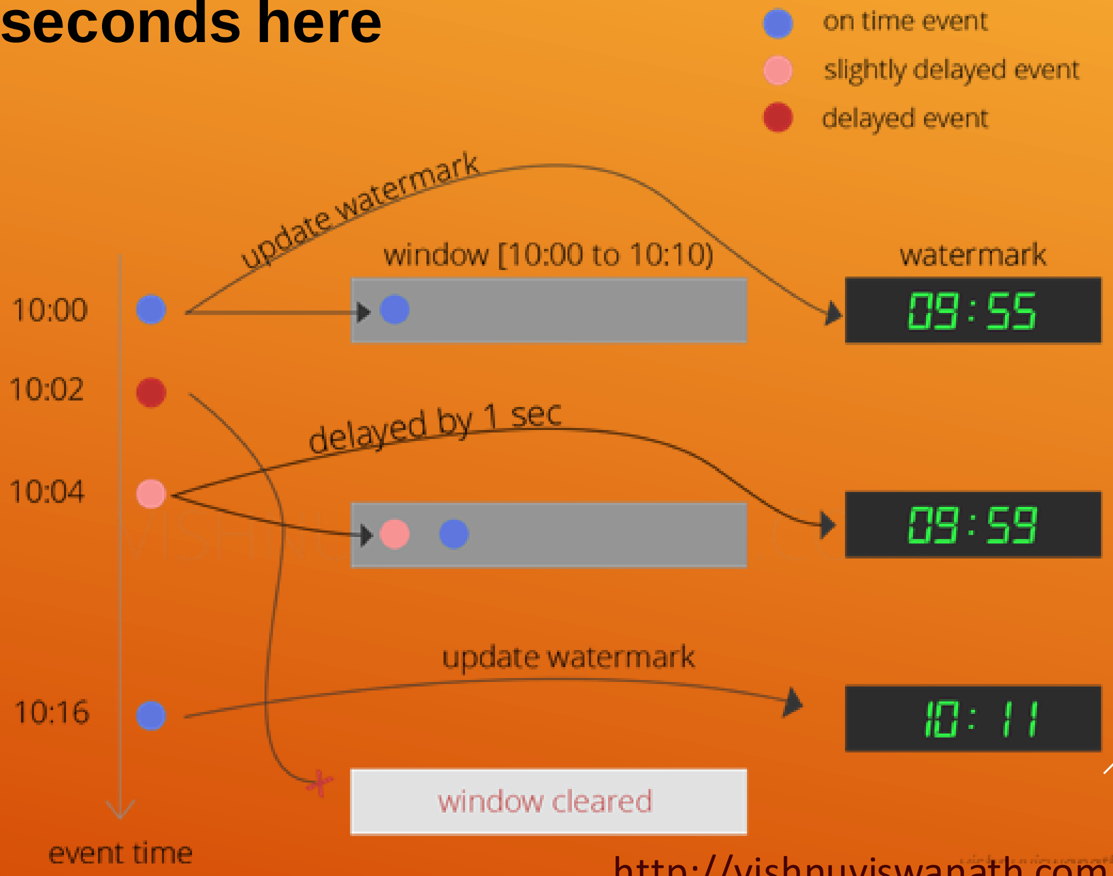
- **Append:** Only the new rows appended to the result table since the last trigger will be written to the external storage. This is applicable only on queries where existing rows in the result table cannot change (e.g. a map on an input stream).
- **Complete**: The entire updated result table will be written to external storage.
- **Update**: Only the rows that were updated in the result table since the last trigger will be changed in the external storage. This mode works for output sinks that can be updated in place, such as a MySQL table.



WATERMARK

In Spark, Watermark is used to decide when to clear a state based on current maximum event time. Based on the **delay** you specify, Watermark lags behind the maximum event time seen so far. e.g., if dealy is 3 seconds and current max event time is 10:00:45 then the watermark is at 10:00:42. This means that Spark will keep the state of windows who's end time is less than 10:00:42.

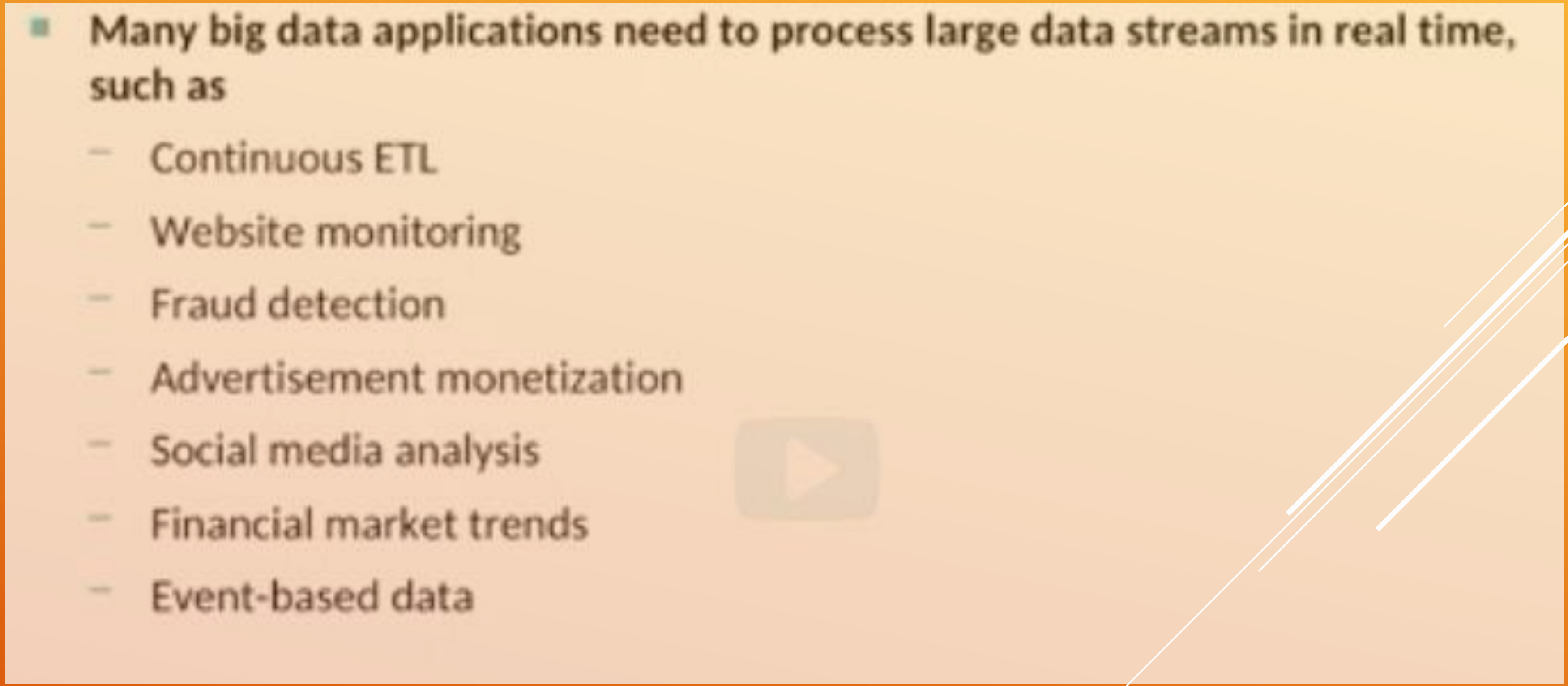
Watermark delay is 5 seconds here



http://vishnuviswanath.com/spark_structured_streaming.ht

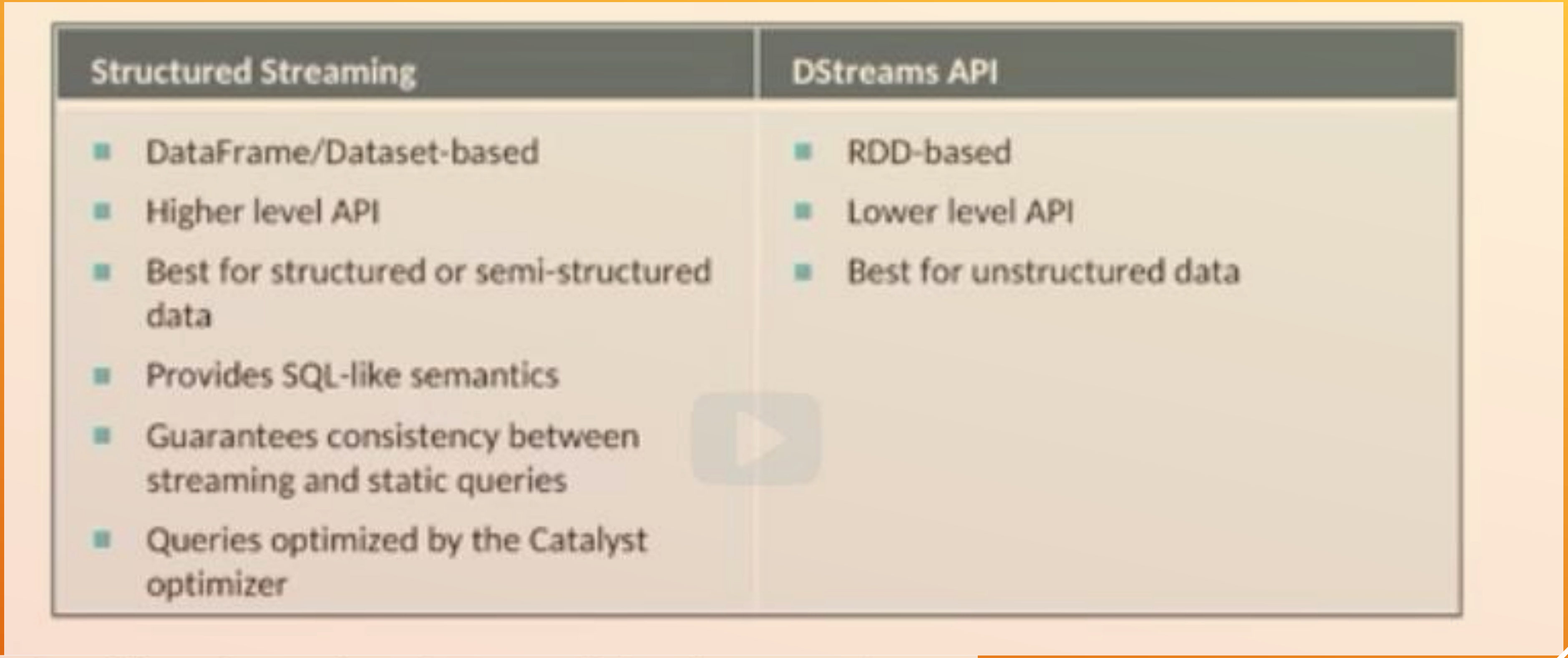
Spark Streaming

Use Cases:

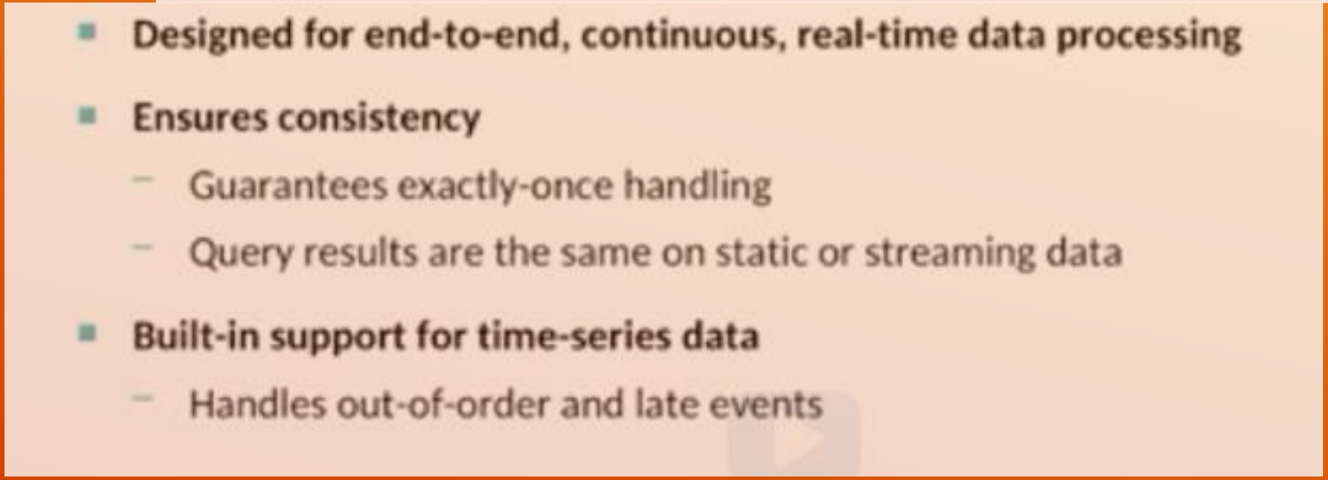


- Many big data applications need to process large data streams in real time, such as
  - Continuous ETL
  - Website monitoring
  - Fraud detection
  - Advertisement monetization
  - Social media analysis
  - Financial market trends
  - Event-based data

Spark Streaming v/s DStreams



Structured Streaming	DStreams API
DataFrame/Dataset-based	RDD-based
Higher level API	Lower level API
Best for structured or semi-structured data	Best for unstructured data
Provides SQL-like semantics	
Guarantees consistency between streaming and static queries	
Queries optimized by the Catalyst optimizer	



- **Designed for end-to-end, continuous, real-time data processing**
- **Ensures consistency**
  - Guarantees exactly-once handling
  - Query results are the same on static or streaming data
- **Built-in support for time-series data**
  - Handles out-of-order and late events

Thank you!