Introduction to Streaming

Spark Streaming (1.0+)

[Optional] Spark Streaming in Depth (1.0+)

Spark Structured Streaming (2.0+)

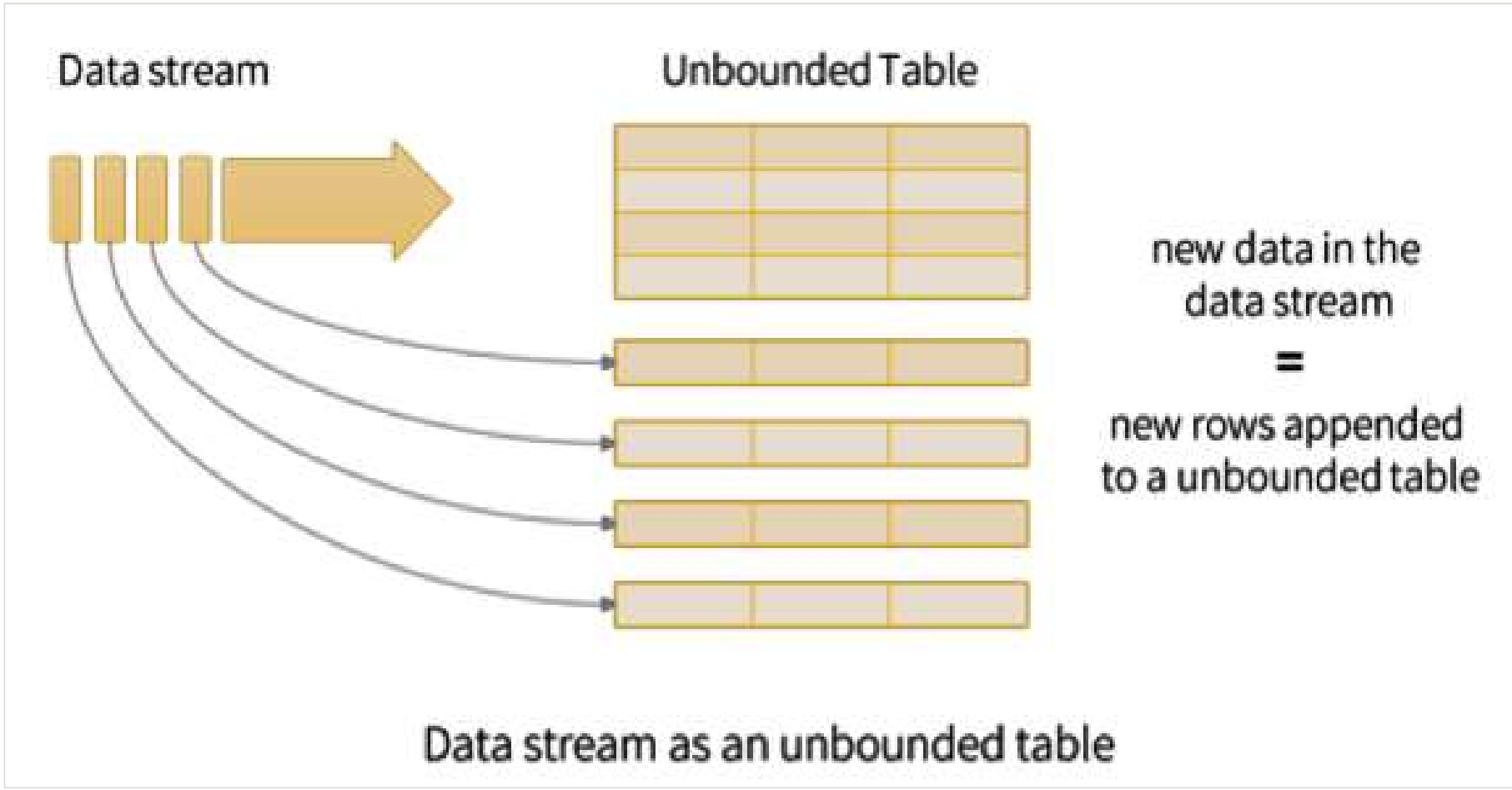
Consuming Kafka Data

Key Concepts

- Designed to support **continuous applications**:
  - End-to-end application that reacts to data in real-time
- Built over DataFrames — higher level than Spark Streaming
  - Streaming API is same as batch API !
- Important new features to support continuous applications:
  - **Streaming job consistent with batch jobs**:
    - Written using DataFrame API
    - Output guaranteed to be same as running batch job on prefix of data
  - **Transactional integration with storage systems**:
    - Process data exactly once
    - Updates output sinks transactionally
  - **Integrates with rest of Spark**
    - Spark SQL, ML, etc.
    - Goal: Every library in Spark runs incrementally on Structured Streaming

How Does it Work?

- Consider the input data stream as an input table
  - New data that arrives is like a new row appended to the table

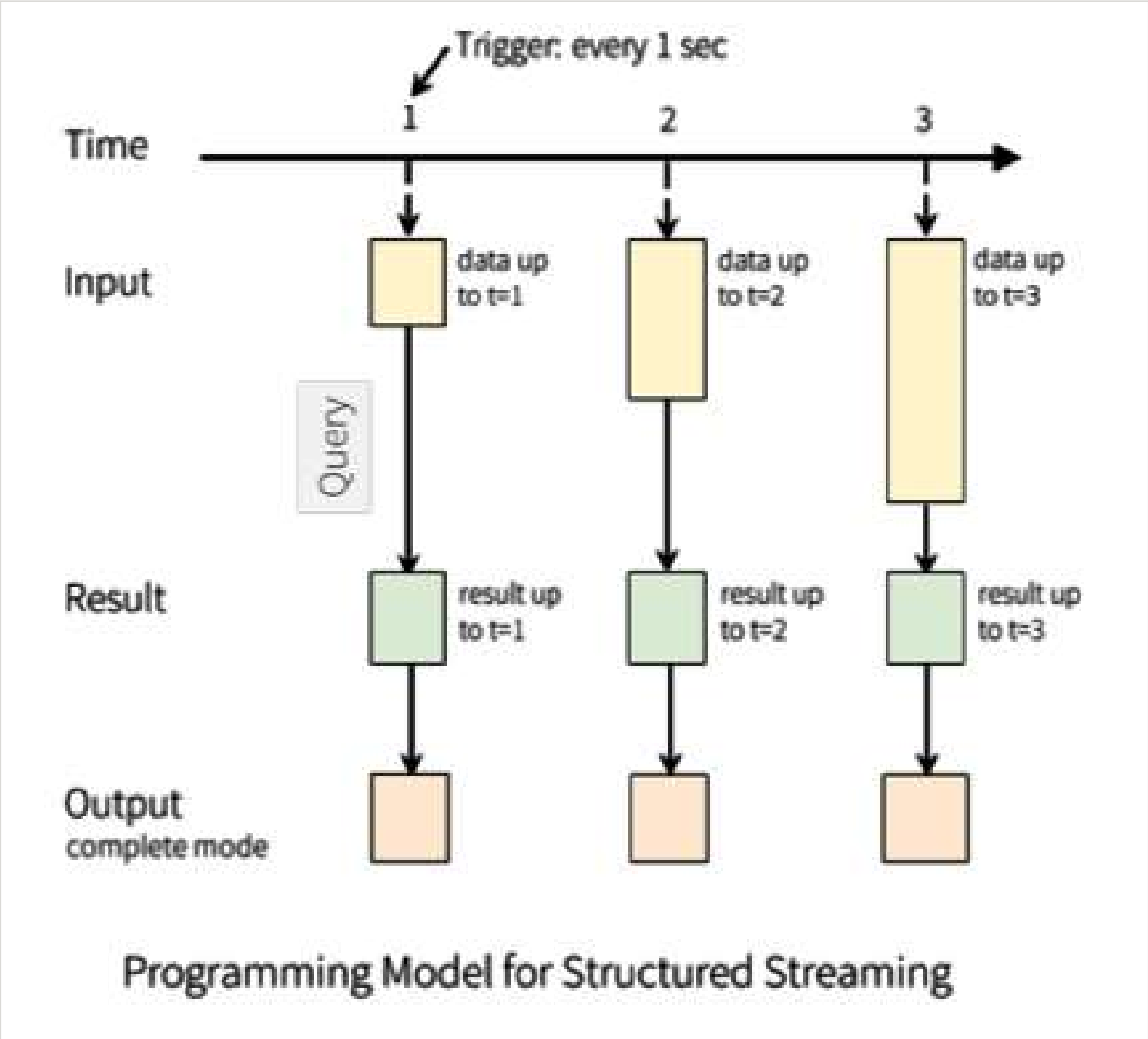


Result Table

- A query on the input creates a result table
  - For each trigger interval (e.g. 1 sec), new rows get appended to the input table
  - Eventually, the result table is updated
  - New data is then processed by your query transformations

- Supports three **output modes**:
  - **complete**: Entire updated table is output
  - **append**: New rows appended since last trigger are written
  - **update**: Rows updated since last trigger are written

Result Table Illustrated



Steps for Structured Streaming

- Set up input DataFrame
  - Use `SparkSession.readStream()` to create `DataStreamReader`
  - Specify the input source via `format()`
    - Currently file, Kafka or socket sources are supported
  - Set input source options (depends on source type)

- **Execute query** to start streaming
  - Use `DataSet.writeStream()` to create `DataStreamWriter`
  - Specify **trigger interval** (how often data is sampled)
    - Default — as soon as possible after data is available
  - Specify **output sink details** (data format, location, etc.)
  - Specify **output mode**
  - **Start** the query processing

Sample Program — Overview

- This simple program does the same processing as our previous Streaming (1.x) example
  - Sets up an input stream, with a source that reads from a socket
    - Using a trigger interval of 5 sec.
  - Filters out all input lines, except those containing string "Scala"
    - Using standard DataFrame operations
  - Writes the filtered input to the console
    - Using an output stream
- The input data is created via the **nc** (`netcat`) program as previously

Sample Code (1 of 2 — Initialization)

- Gets a `DataStreamReader` from session via `readStream()`
  - Uses `format()` to specify input data source format
    - Socket in this example
  - Uses `option()` to specify any options for data source
    - Host and port in this example
  - Calls `load()` to load input (evaluated lazily)
    - No work done until you consume the streaming data with a sink
  - Filters the data — standard `DataFrame.filter()`

```
// Code excerpt showing streaming code only

val lines = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()
val scalaLines = lines.filter('value.contains("Scala"))
```

Sample Code (2 of 2 — Consume Data)

- Creates `DataStreamWriter` via `writeStream()`
  - Sets trigger interval to be 5 seconds
  - Sets output mode to be `append` (appends new output)
  - Sets format to be `console` (outputs to console)
  - Starts processing via `start()`
    - Returns a `StreamingQuery` instance

```
import org.apache.spark.sql.streaming.ProcessingTime

// Code fragment
val query = scalaLines.writeStream
  .trigger(ProcessingTime("5 seconds"))
  .outputMode("append")
  .format("console")
  .start()

query.awaitTermination() // Standalone programs
```

Supported Sources and Sinks

- `DataStreamReader` currently supports these input sources
  - **Socket streams**: (Testing only) read input from socket
    - Via `format("socket")`
  - **File streams**: CSV, JSON, text, Parquet
    - Via `csv(), json(), parquet(), textFile()` functions
  - **Kafka**: Poll data from Kafka (1)
    - Via `format("kafka")`

- `DataStreamWriter` currently supports these output sinks
  - **Console sink**: (for debugging) — outputs to console
    - Via `format("console")`
  - **File sink**: CSV, JSON, text, Parquet
    - Via `format("XXX")` where XXX is "parquet", "json", etc.
  - **Memory sink**: (for debugging) — store output on in-memory table
    - Via `format("memory")`
  - **foreach sink**: Run arbitrary computation on output records
    - Via `foreach(…)`

Summary

- Spark Structured Streaming is better than Spark Streaming
  - Based on Spark SQL / DataFrames
  - Leverages all benefits — Catalyst, Tungsten, higher level API
  - Future enhancements will be done here
    - Spark Streaming is legacy now
- Spark Structured Streaming is an **Alpha** release
  - Not feature complete
  - May change before final release
  - Use with caution