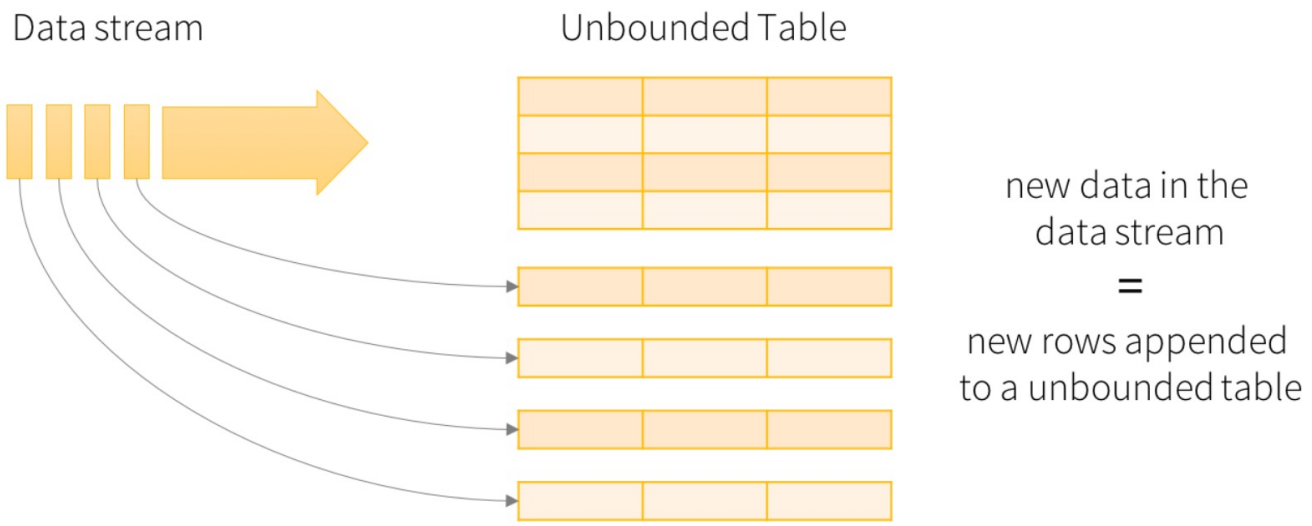
Structured Streaming Introduction (Reference)

Structured Streaming

- Structured Streaming is a scalable and fault-tolerant **stream processing engine built on the Spark SQL engine**.
- **Use the DataFrame API** to express streaming aggregations, event-time windows, stream-to-batch joins, etc.
- The Spark SQL engine will take care of **running it incrementally and continuously and updating the final result** as streaming data continues to arrive

Programing Model

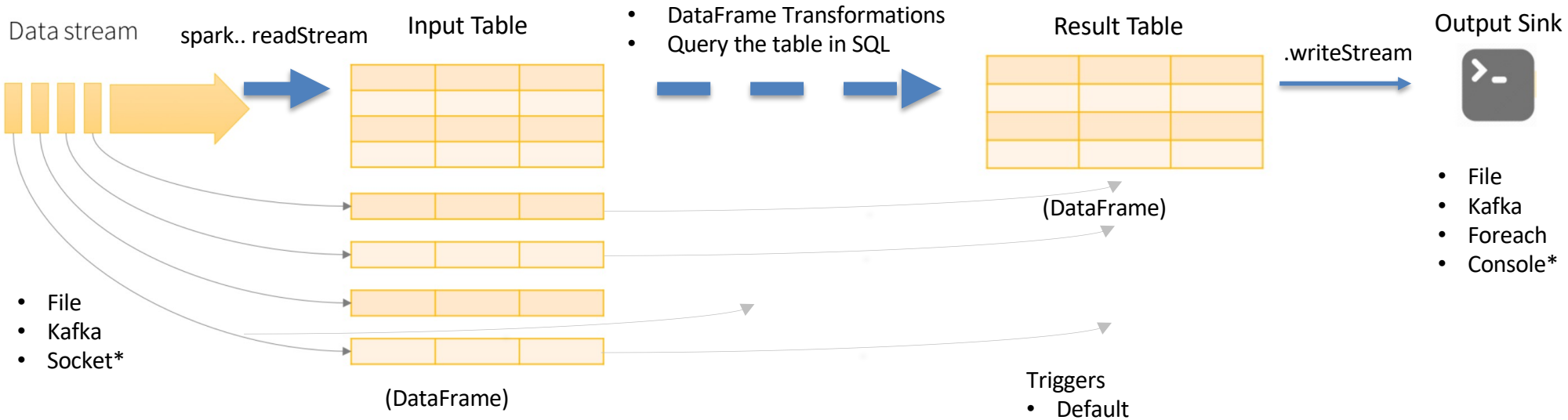
- Treat a live data stream as a input table that is being continuously appended.
- Every data item that is arriving on the stream is like a new row being appended to the Table.



Data stream as an unbounded table

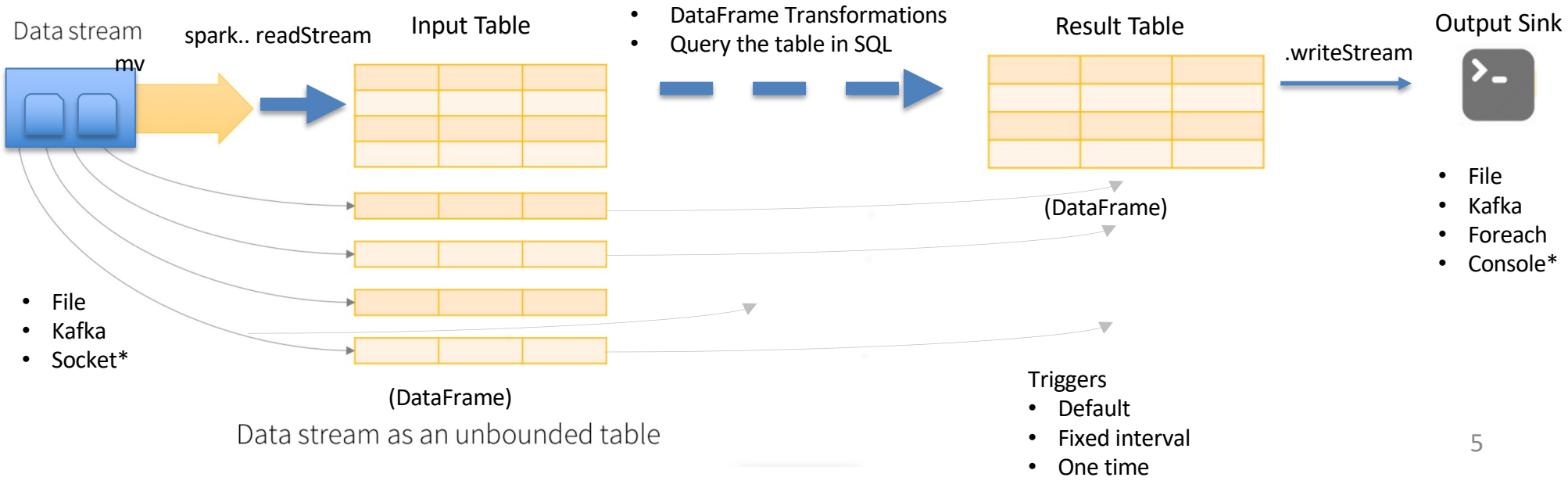
Programing Model

- A query on the input will generate the “Result Table”
- Every trigger interval, new rows get appended to the Input Table are processed and updates the Result Table. Then write the changed result rows to an external sink.



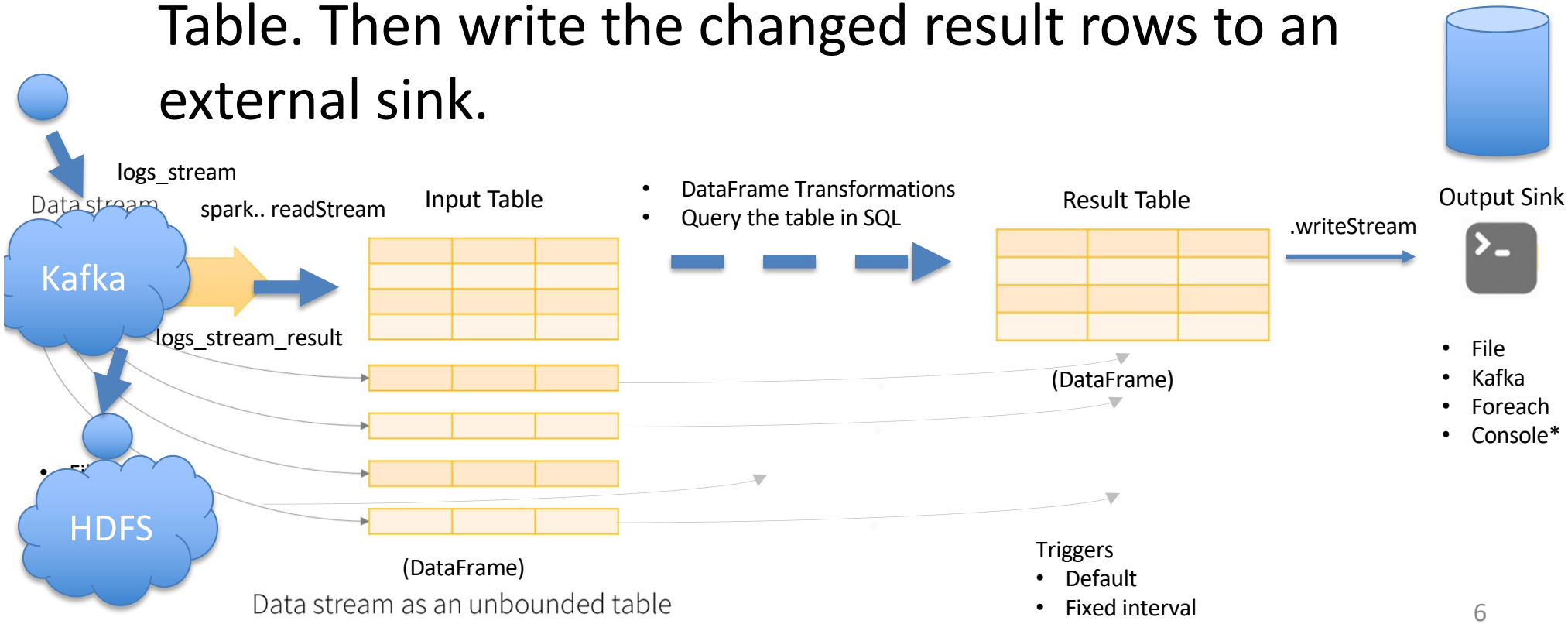
Programing Model

- A query on the input will generate the “Result Table”
- Every trigger interval, new rows get appended to the Input Table are processed and updates the Result Table. Then write the changed result rows to an external sink.

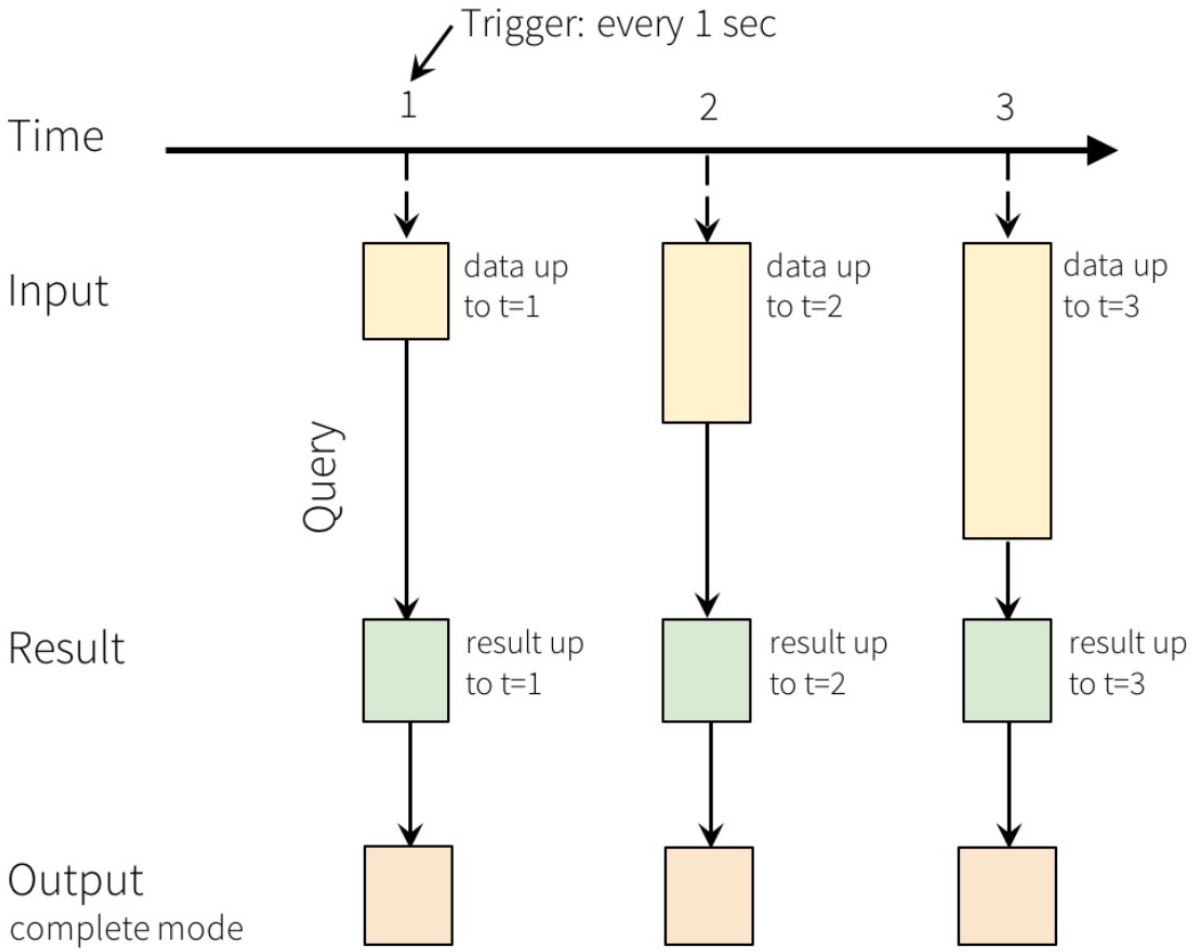


Programing Model

- A query on the input will generate the “Result Table”
- Every trigger interval, new rows get appended to the Input Table are processed and updates the Result Table. Then write the changed result rows to an external sink.

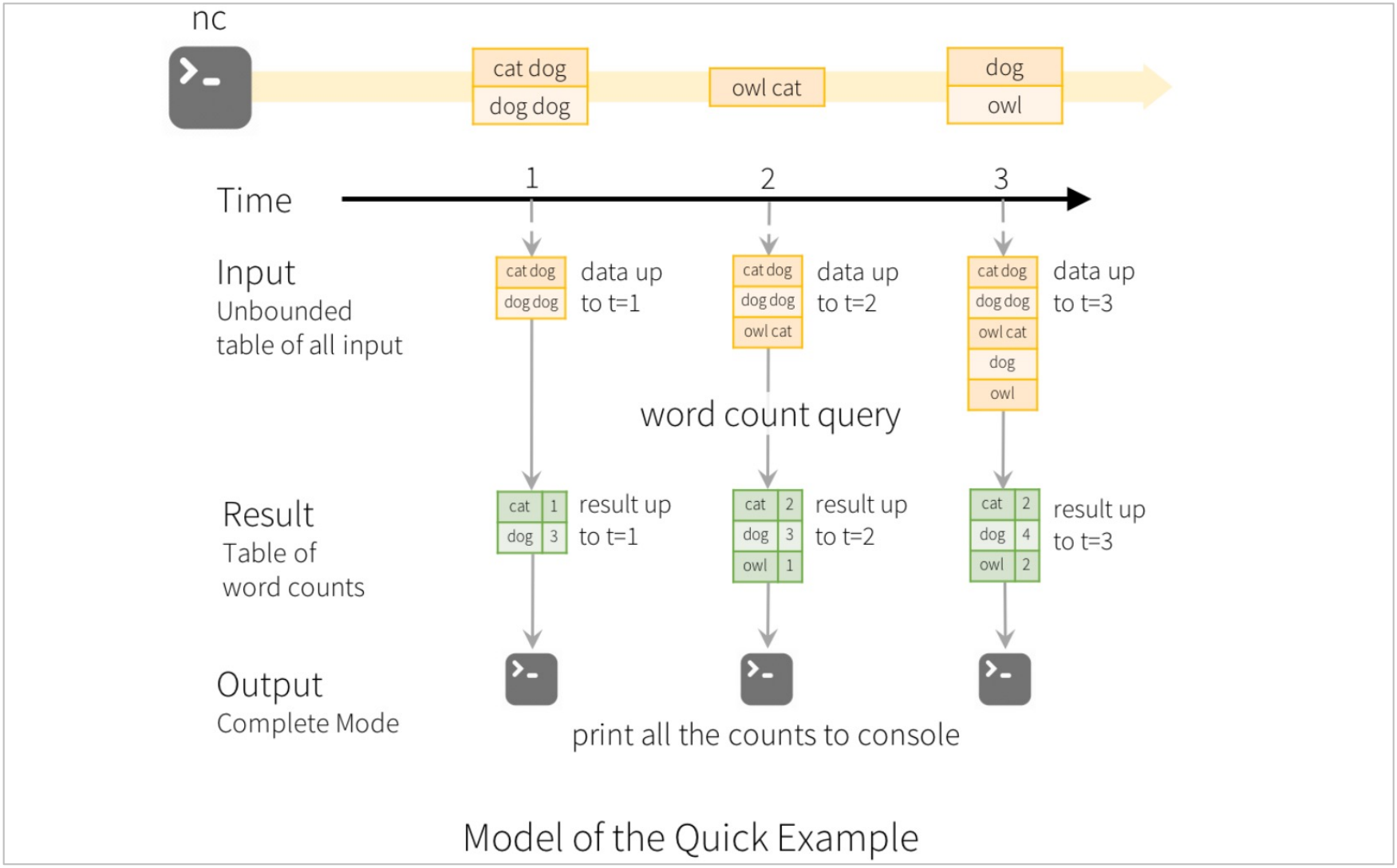


Programing Model



Programming Model for Structured Streaming

Structured Network WordCount Example



Model of the Quick Example

Output Modes

- Complete Mode
  - The entire updated Result Table will be written to the external storage
- Append Mode
  - Only the new rows appended in the Result Table since the last trigger will be written to the external storage.
- Update Mode
  - Only the rows that were updated in the Result Table since the last trigger will be written to the external storage
- Each mode is applicable on certain types of queries.

A Dive-in Demo

```
1  from pyspark.sql import SparkSession
2  from pyspark.sql.functions import explode
3  from pyspark.sql.functions import split
4
5  if __name__ == "__main__":
6
7      spark = SparkSession\
8          .builder\
9          .appName("Structured Network Wordcount")\
10         .getOrCreate()
11
12     # Create DataFrame representing the stream of input lines from connection to localhost:9999
13     lines = spark \
14         .readStream \
15         .format("socket") \
16         .option("host", "localhost") \
17         .option("port", 9999) \
18         .load()
19
20     # Split the lines into words
21     words = lines.select(
22         explode(
23             split(lines.value, " ")
24         ).alias("word")
25     )
26
27     # Generate running word count
28     wordCounts = words.groupBy("word").count()
29
30
31      # Start running the query that prints the running counts to the console
32     query = wordCounts \
33         .writeStream \
34         .outputMode("complete") \
35         .format("console") \
36         .start()
37
38     query.awaitTermination()
```

A Dive-in Demo

```
# TERMINAL 1:
# Running Netcat

$ nc -lk 9999
apache spark
apache hadoop

...
```

Scala | Java | **Python** | R

```
# TERMINAL 2: RUNNING structured_network_wordcount.py

$ ./bin/spark-submit examples/src/main/python/sql/streaming/structured_network_wordcount.py localhost 9999

-------------------------------------------
Batch: 0
-------------------------------------------
+------+-----+
| value|count|
+------+-----+
|apache|    1|
| spark|    1|
+------+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+------+-----+
| value|count|
+------+-----+
|apache|    2|
| spark|    1|
|hadoop|    1|
+------+-----+
...
```

Streaming DataFrame

- In structured streaming, streaming data is expressed as a DataFrame just like batch data

```
df.isStreaming()
```

  - True if the DataFrame is a streaming DataFrame
  - False if not

Streaming DataFrame Operations

- Use SQL or DataFrame API to transform streaming DataFrames just like batch data
  - Use SQL

```
df.createOrReplaceTempView("updates")
spark.sql("select count(*) from updates")  # returns another streaming DF
```

  - Use DataFrame Transformations

```
df = ...  # streaming DataFrame with IOT device data with schema { device: string, deviceTyp
e: string, signal: double, time: DateType }

# Select the devices which have signal more than 10
df.select("device").where("signal > 10")

# Running count of the number of updates for each device type
df.groupBy("deviceType").count()
```

Unsupported Operations

- **Multiple streaming aggregations** (i.e. a chain of aggregations on a streaming DF) **are not yet supported** on streaming Datasets.
- **Limit and take first N rows** are **not supported** on streaming Datasets.
- **Distinct operation** on streaming Datasets are **not supported**.
- **Sorting operations are supported** on streaming Datasets **only after an aggregation and in Complete Output Mode**.
- **Few types of outer joins** on streaming Datasets **are not supported.**
  - https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#support-matrix-for-joins-in-streaming-queries

Creating streaming DataFrames

- **File source**
  - **Reads files written in a directory as a stream of data**. Supported file formats are **text, csv, json, orc, parquet**.
  - Note that the files must be atomically placed in the given directory, which in most file systems, can be achieved by **file move operations**.
- **Kafka source**
  - Reads data from Kafka.
  - It's compatible with Kafka broker versions **0.10.0 or higher**.
- **Socket source (for testing)**
  - **Reads UTF8 text data from a socket connection**.
  - **The listening server socket is at the driver.**
  - Note that this should be used only for testing as this does not provide end-to-end fault-tolerance guarantees.

Creating streaming DataFrames

Source	Options	Fault-tolerant	Notes
File source	`path`: path to the input directory, and common to all file formats. `maxFilesPerTrigger`: maximum number of new files to be considered in every trigger (default: no max) `latestFirst`: whether to processs the latest new files first, useful when there is a large backlog of files (default: false) `fileNameOnly`: whether to check new files based on only the filename instead of on the full path (default: false). With this set to `true`, the following files would be considered as the same file, because their filenames, "dataset.txt", are the same: "file:///dataset.txt" "s3://a/dataset.txt" "s3n://a/b/dataset.txt" "s3a://a/b/c/dataset.txt"	Yes	Supports glob paths, but does not support multiple comma-separated paths/globs.
Socket Source	`host`: host to connect to, must be specified `port`: port to connect to, must be specified	No	
Kafka Source	See the Kafka Integration Guide.	Yes	

Demo

- Crime data example

Trigger

- The trigger defines the timing of streaming data processing

Trigger Type	Description
unspecified (default)	If no trigger setting is explicitly specified, then by default, the query will be executed in micro-batch mode, where micro-batches will be generated as soon as the previous micro-batch has completed processing.
Fixed interval micro-batches	The query will be executed with micro-batches mode, where micro-batches will be kicked off at the user-specified intervals. • If the previous micro-batch completes within the interval, then the engine will wait until the interval is over before kicking off the next micro-batch. • If the previous micro-batch takes longer than the interval to complete (i.e. if an interval boundary is missed), then the next micro-batch will start as soon as the previous one completes (i.e., it will not wait for the next interval boundary). • If no new data is available, then no micro-batch will be kicked off.
One-time micro-batch	The query will execute *only one* micro-batch to process all the available data and then stop on its own. This is useful in scenarios you want to periodically spin up a cluster, process everything that is available since the last period, and then shutdown the cluster. In some case, this may lead to significant cost savings.

Trigger

- The trigger defines the timing of streaming data processing

```
# Default trigger (runs micro-batch as soon as it can)
df.writeStream \
  .format("console") \
  .start()

# ProcessingTime trigger with two-seconds micro-batch interval
df.writeStream \
  .format("console") \
  .trigger(processingTime='2 seconds') \
  .start()

# One-time trigger
df.writeStream \
  .format("console") \
  .trigger(once=True) \
  .start()
```

Demo

- Network wordcount with trigger settings

Output Sinks

Sink	Supported Output Modes	Options	Fault-tolerant
File Sink	Append	`path`: path to the output directory, must be specified. For file-format-specific options, see the related methods in DataFrameWriter (Scala/Java/Python/R). E.g. for "parquet" format options see `DataFrameWriter.parquet()`	Yes (exactly-once)
Kafka Sink	Append, Update, Complete	Read Kafka integration guide and examples https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html	Yes (at-least-once)
Foreach Sink	Append, Update, Complete	None	Depends on ForeachWriter implementation
Console Sink	Append, Update, Complete	`numRows`: Number of rows to print every trigger (default: 20) `truncate`: Whether to truncate the output if too long (default: true)	No

Output Sinks

- **File sink** - Stores the output to a directory.

```
writeStream
    .format("parquet")        // can be "orc", "json", "csv", etc.
    .option("path", "path/to/destination/dir")
    .start()
```

- **Kafka sink** - Stores the output to one or more topics in Kafka.

```
writeStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
    .option("topic", "updates")
    .start()
```

- **Foreach sink** - Runs arbitrary computation on the records in the output. See later in the section for more details.

```
writeStream
    .foreach(...)
    .start()
```

- **Console sink (for debugging)** - Prints the output to the console/stdout every time there is a trigger. Both, Append and Complete output modes, are supported. This should be used for debugging purposes on low data volumes as the entire output is collected and stored in the driver's memory after every trigger.

```
writeStream
    .format("console")
    .start()
```

Query Types and Output Modes

- Different types of streaming queries support different output modes
  - Queries with aggregation
    - Complete, Update
  - Queries with joins
    - Append
  - Other queries
    - Append, Update

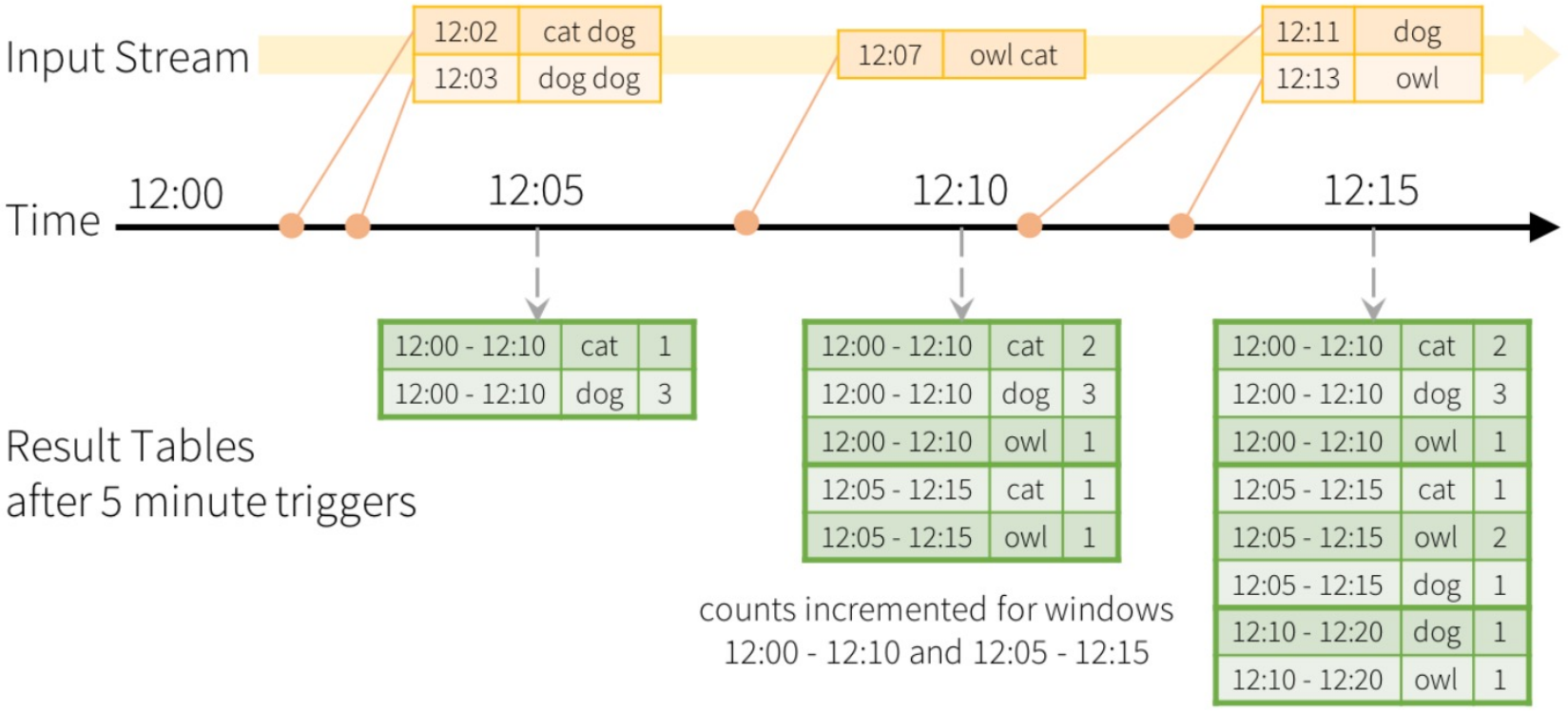
Demo

- Kafka source examples
- EC logs examples

Window Operations on Event Time

- Aggregations over a sliding event-time window are straightforward with Structured Streaming and are very similar to grouped aggregations
- In case of window-based aggregations, aggregate values are maintained for each window the event-time of a row falls into

Window Operations on Event Time



Windowed Grouped Aggregation
with 10 min windows, sliding every 5 mins

Demo

- Window operations on event time