SPARK STREAMING

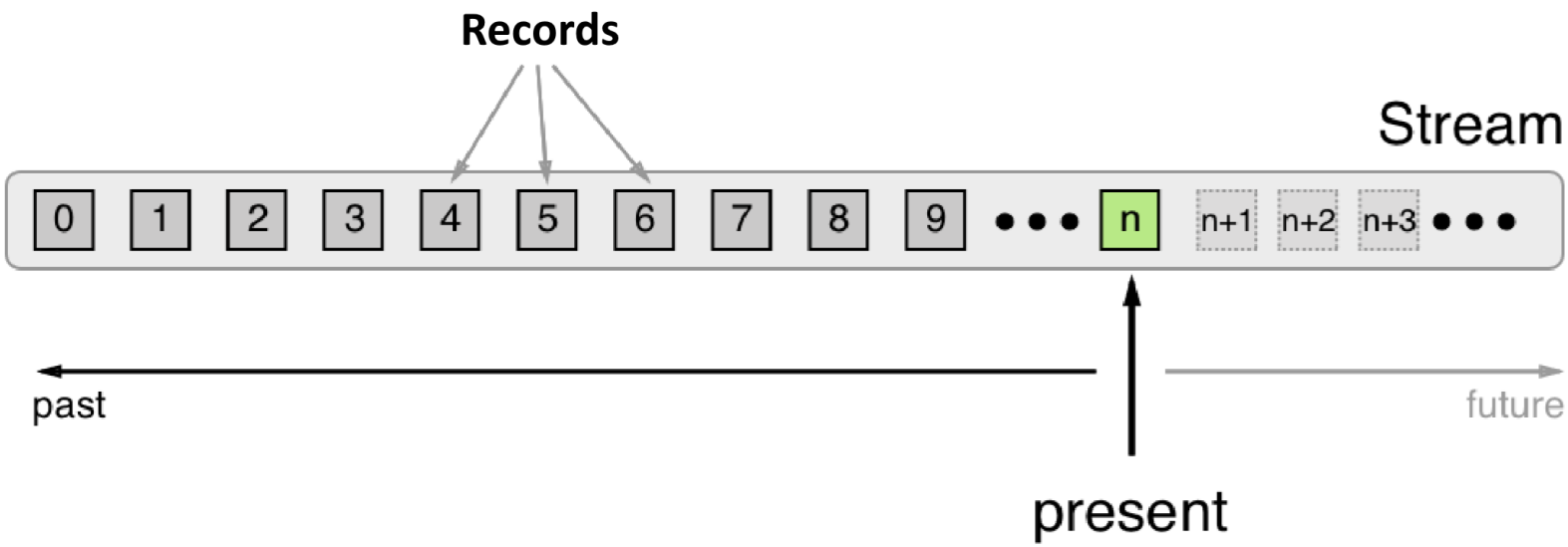
Presented by: Le Ngoc Thanh

Outline

- **Stream Processing**
- Event and processing time, types of execution
- Spark Streaming
- Input and Output
- Structured Streaming Query
- Windows
- Handling Late Data
- Metrics and Monitoring

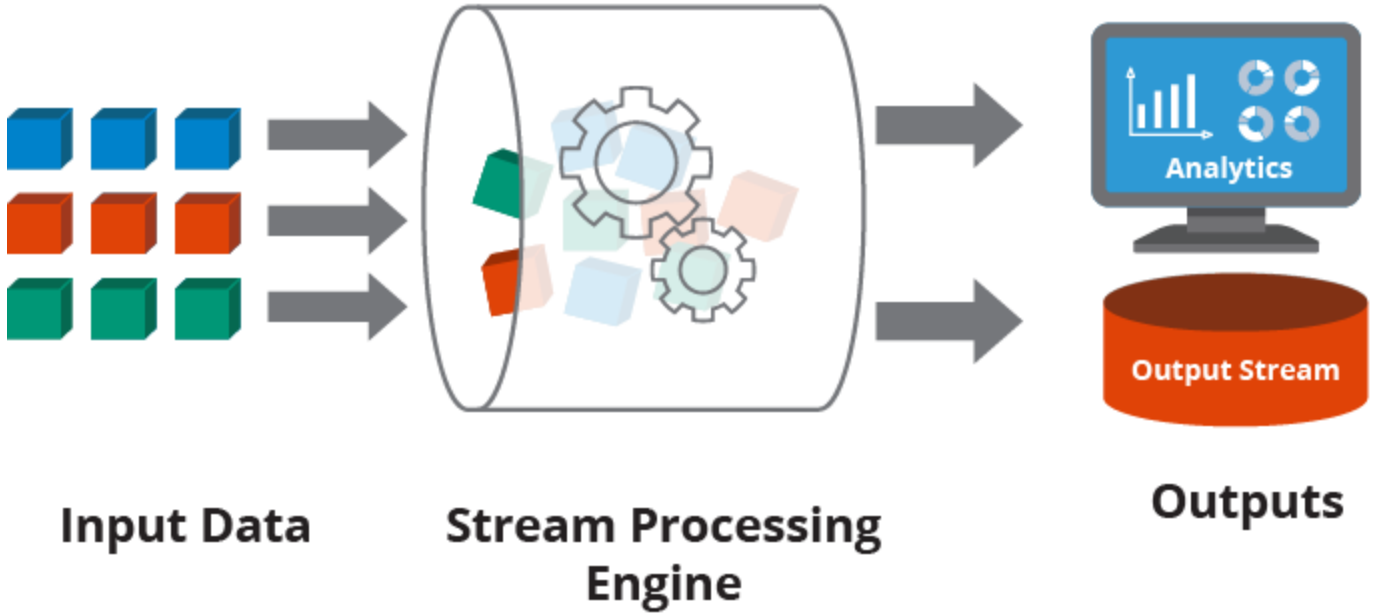
Stream

- A stream is a partitioned, immutable, append-only collection that represents a series of historical facts.
  - For example, the rows of a stream could model a sequence of financial transactions, like "Alice sent $100 to Bob", followed by "Charlie sent $50 to Bob".



Stream Proccessing

- Stream processing is the incremental processing of unbounded datasets.



When use stream processing

fit@hcmus

- There are 6 common use cases:
  - Notification and alerting should be trigger if some event occurs.
  - Real-time reporting: show real-time dashboard to monitor activities in your organization
  - Incremental ETL: reduce the latency while retrieving information into a data warehouse
  - Update data to serve in real time: for example, Google Analytics continuously track the number of page visits to serve users when they want to query the lastest counts.
  - Real-time decision marking: analyzing new input and automatically update business logic (e.g. verify a new transaction in credit card payment)
  - Online maching learning: train a model on a combination of streaming and historical data.

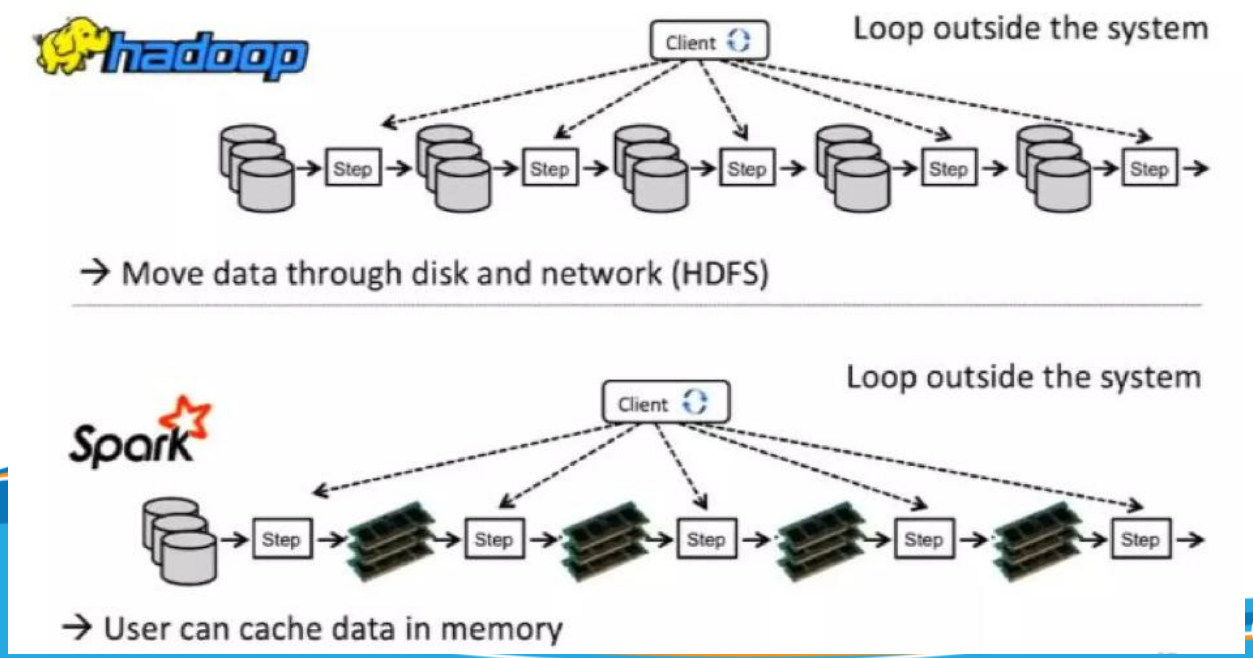
Advantages of Stream Processing

fit@hcmus

- Some advantages of stream processing:
  - Lower latency
    - Keep state in memory to respond quickly
  - Efficient in updating a result
    - Automatically incrementalizes the computation.
    - Only need to process new data since the last report.

Streaming processing with Map-Reduce

- Map/Reduce is inherently batch processing system, not suitable for streaming:
  - Need all data in available.
  - Stream needs multiple transformation
  - Overhead in lauch of a new M/R job



Challenges of stream processing

fit@hcmus

- Some challenges:
  - Data might to the application out-of-order (because delays or retransmission).
    - For example, a sensor reports its value at different times:

```
{value: 1, time: "2017-04-07T00:00:00"}
{value: 2, time: "2017-04-07T01:00:00"}
{value: 5, time: "2017-04-07T02:00:00"}
{value: 10, time: "2017-04-07T01:30:00"}
{value: 7, time: "2017-04-07T03:00:00"}
```

    - If you want only to trigger some action based on a specific sequence of values received (2 -> 10 -> 5), it is challenge for stream processing (but not challenge in batch processing).
  - Joining with external data
  - How to update as new events arrive
  - ….

Outline

- Stream Processing
- **Event and processing time, types of execution**
- Spark Streaming
- Input and Output
- Structured Streaming Query
- Windows
- Handling Late Data
- Metrics and Monitoring

Event time vs Processing time

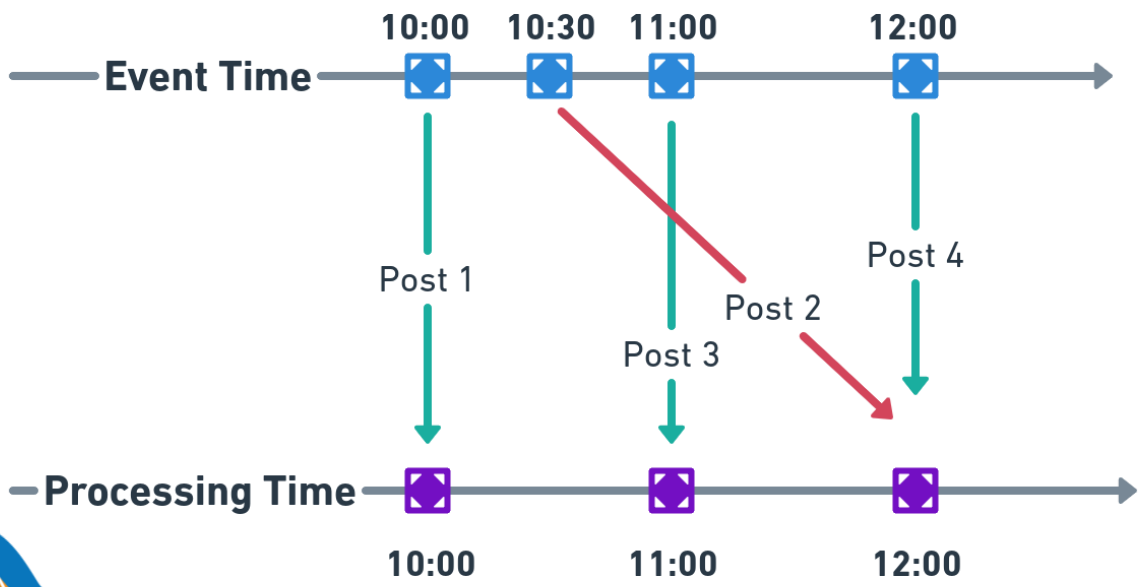
fit@hcmus

- Event time: timestamps inserted into each record at the source.

```
{value: 1, time: "2017-04-07T00:00:00"}
{value: 2, time: "2017-04-07T01:00:00"}
{value: 5, time: "2017-04-07T02:00:00"}
{value: 10, time: "2017-04-07T01:30:00"}
{value: 7, time: "2017-04-07T03:00:00"}
```

Event time vs Processing time

- Event time: timestamps inserted into each record at the source.
- Processing time: timestamps when the record is received at the streaming application

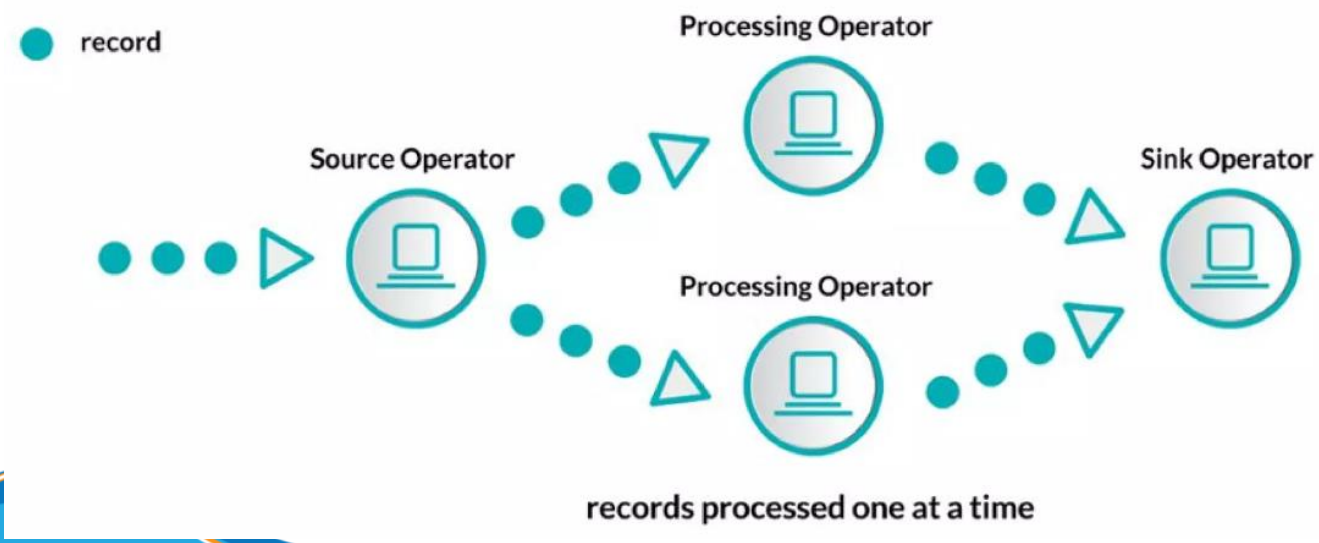


When use types of time

- For event time:
  - If you want to get the number of events generated by IoT devices every minute, then use the time when the data was generated.

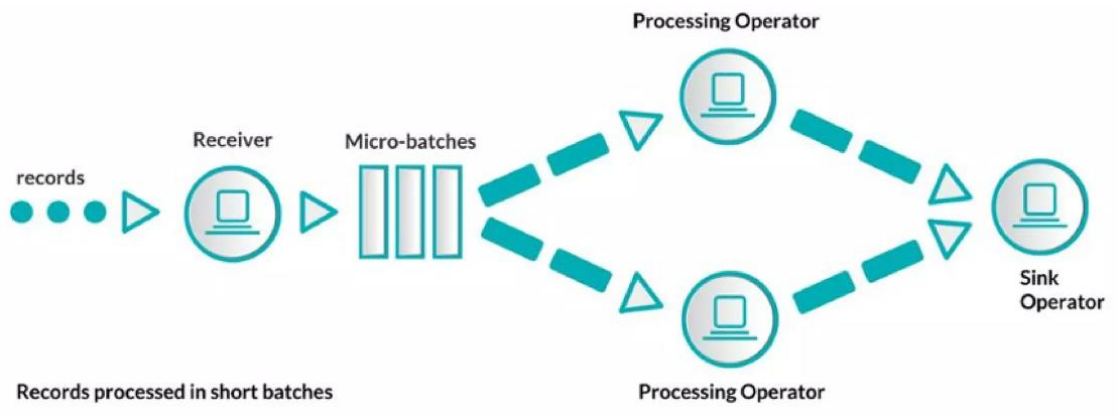
Continuous versus micro-batch excecution

- In continuous processing, each record of input is computed.
  - Pros: lowest possible latency
  - Cons: lower maximum throughput (overhead pre-record), cannot be moved at runtime without stopping the whole system.

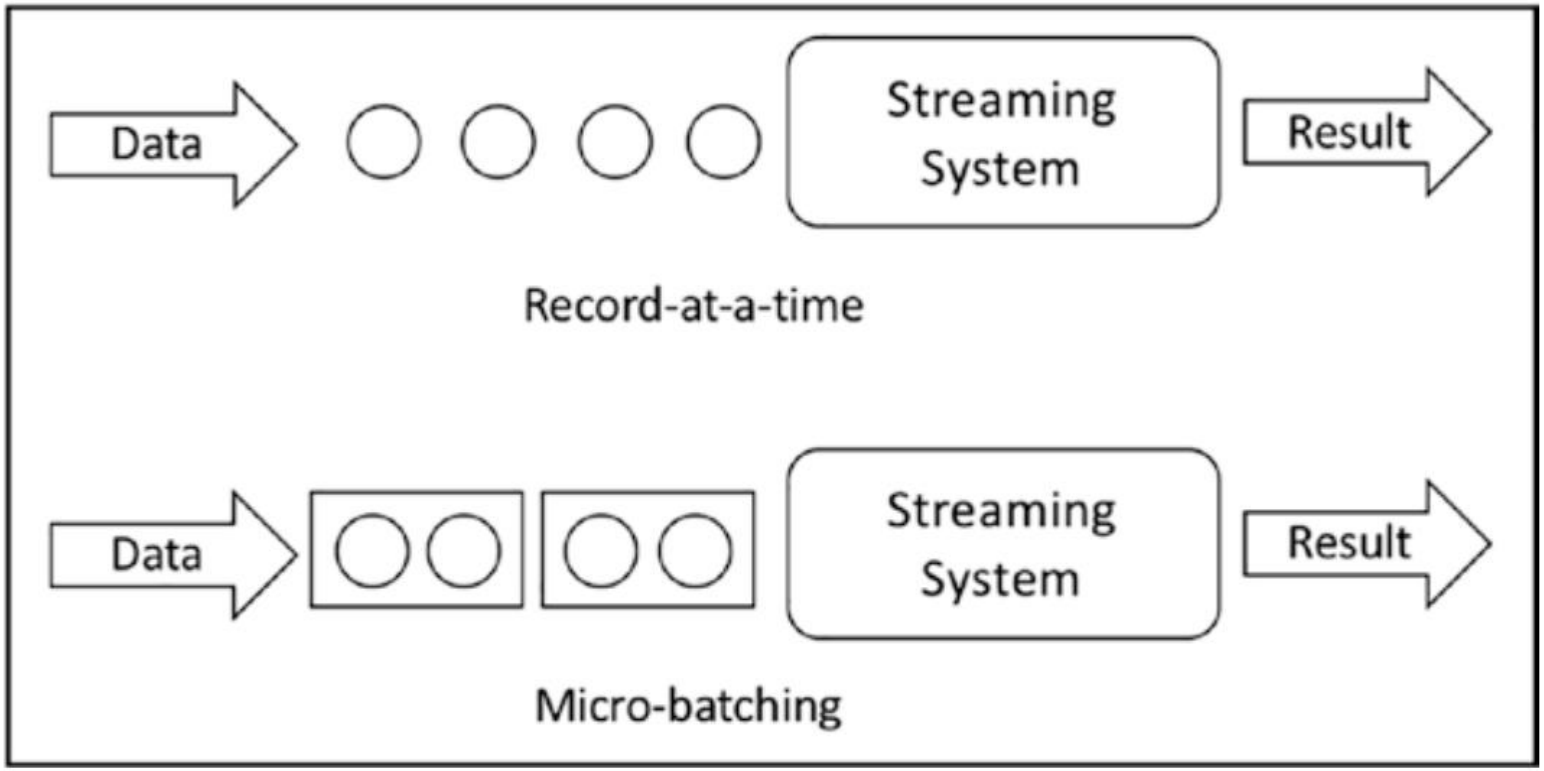


Continuous versus micro-batch execution

- In continuous processing, each record of input is computed.
- In micro-batch precessing, wait to accumulate small batches of input data, then process each batch in parallel.
  - Pros: high throughput
  - Cons: higher latency (100 ms to a second)



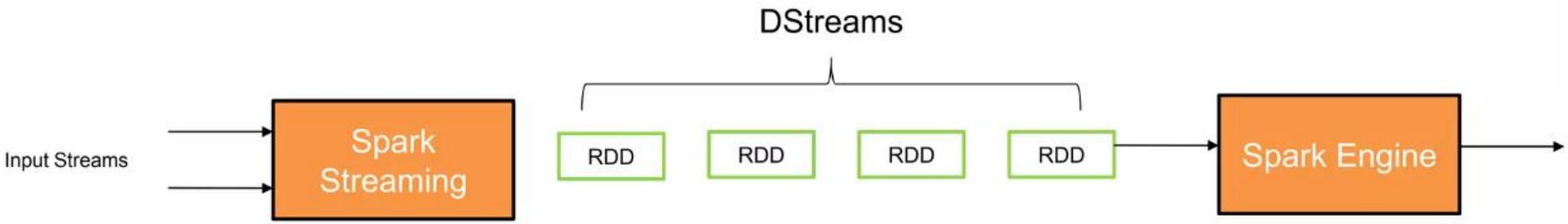
Continuous versus micro-batch execution



Outline

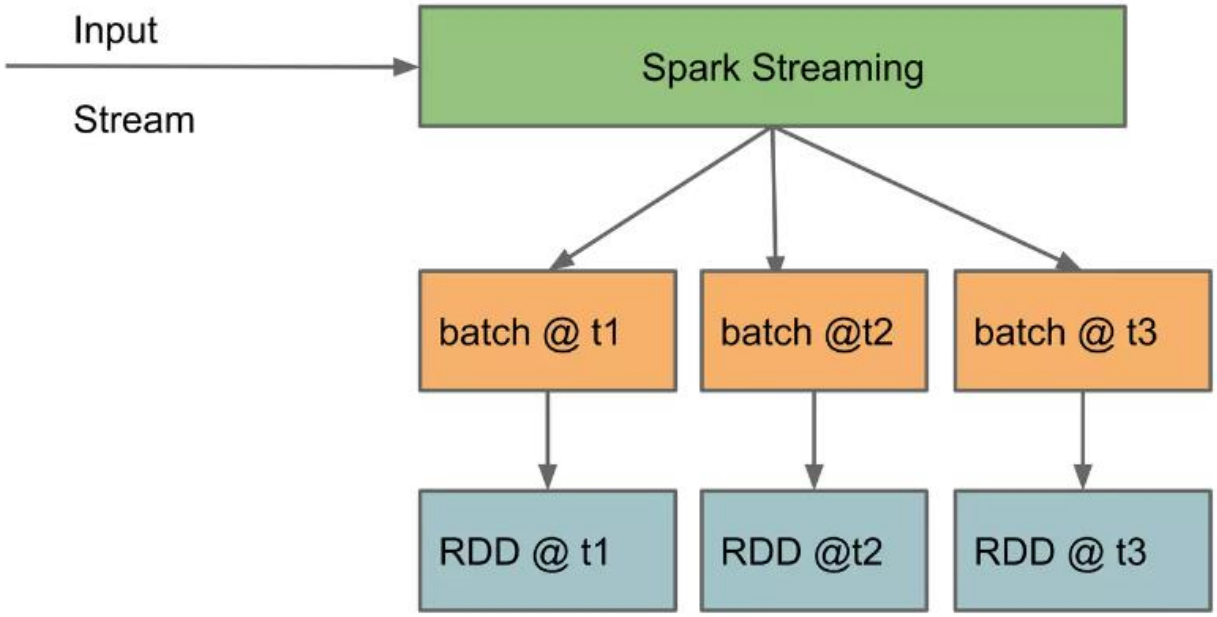
- Stream Processing
- Event and processing time, types of execution
- **Spark Streaming**
- Input and Output
- Structured Streaming Query
- Windows
- Handling Late Data
- Metrics and Monitoring

Spark Streaming



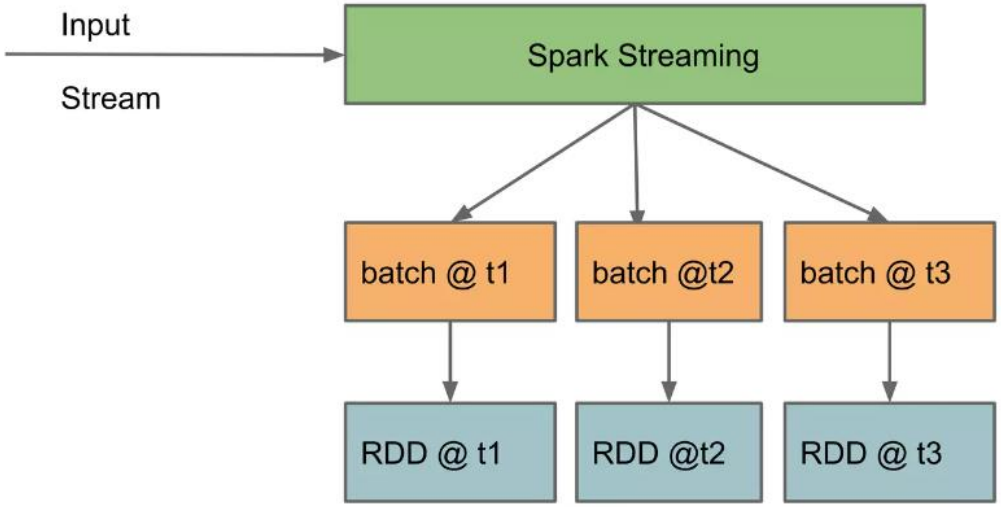
Spark's streaming API

- Spark includes 2 streaming APIs:
  - DStream API (earlier): purely micro-batch oriented.
    - Discretized streams
    - Collect stream data into small batch (1s to hours) and run batch processing.



Example

- A streaming application runs for 12 hours continuously.
- Suppose the batch interval is of 5 minutes.
  - It means that a new RDD will be created every 5 minutes.
- How many RDD will we have?
  - 144



DStream

- Advantages:
  - Spark's agile task scheduling can very quickly and efficiently recover from failures and straggler executors.
  - The deterministic nature of the tasks ensures that the output data is the same no matter how many times the task is reexecuted.
- Disadvantages:
  - The micro-batch model cannot achieve millisecond-level latencies.
  - Lack of native support for event-time windows.
  - Lack of separation between logical and physical plans.

DStream

- When to use DStream?
  - The pipeline does not need latencies lower than a few seconds.
    - For example, when the streaming output is only going to be read by hourly jobs, it is not useful to generate output with subsecond latencies.
  - There are larger delays in other parts of the pipeline.
    - For example, if the writes by a sensor into Apache Kafka are batched to achieve higher throughput, then no amount of optimization in the downstream processing systems can make the end-to-end latency lower than the batching delays

Spark's streaming API

- Spark includes 2 streaming APIs:
  - DStream API (earlier): purely micro-batch oriented.
  - Structured Streaming API (newer): support continuous processing.

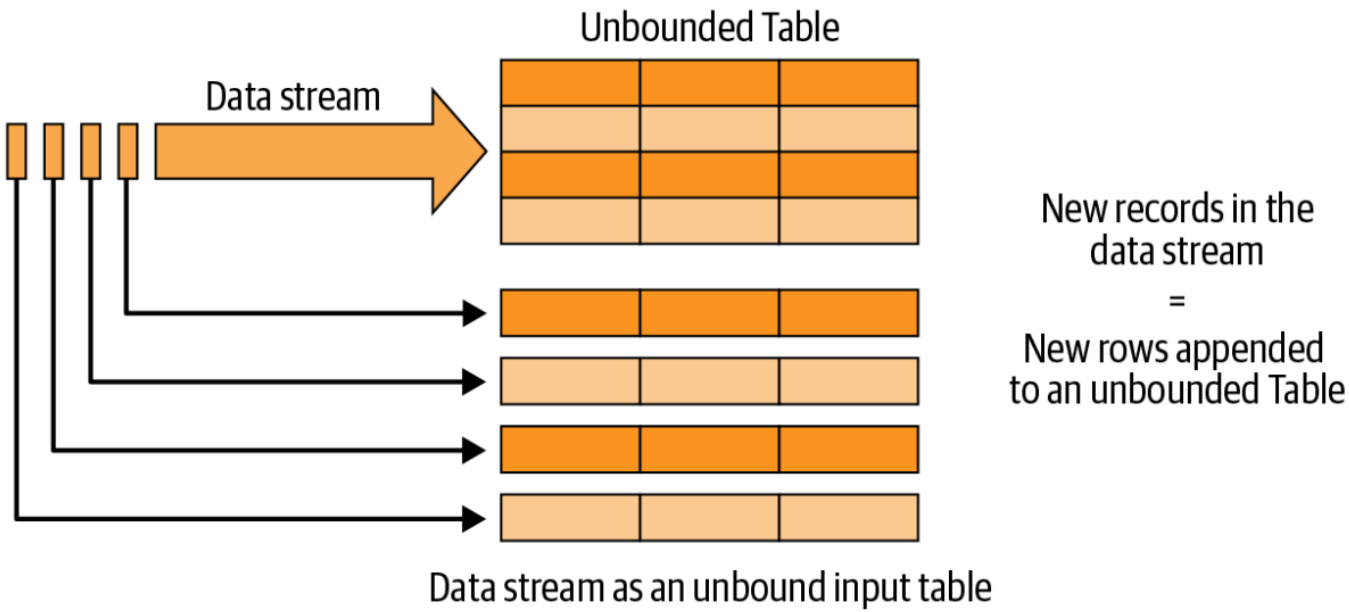
Structured Streaming API

fit@hcmus

- Structured Streaming API:
  - Use the existing structured APIs in Spark (DataFrame, Dataset, SQL)
    - Users express a streaming computation in the same way write a batch computation on static data.
    - Structured Streaming engine take care of running the query incrementally and continuously as new data arrives.
  - Include other features for streaming: end-to-end, exactly-once processing, fault-tolerance.

Structured Streaming API

- Main idea:
  - Treat a stream of data as a table.
  - Data is continuously appended to this table.

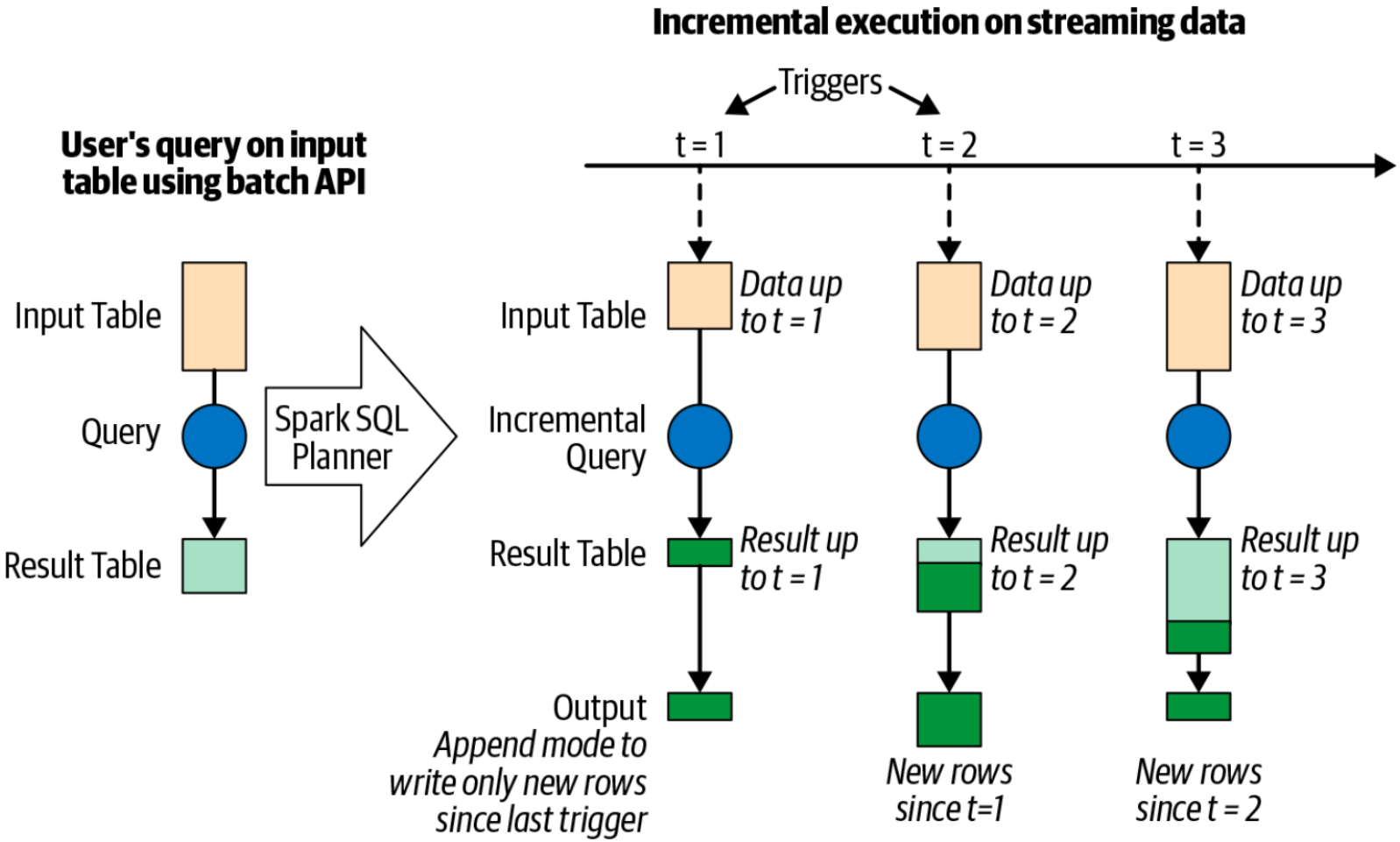


Structured Streaming API

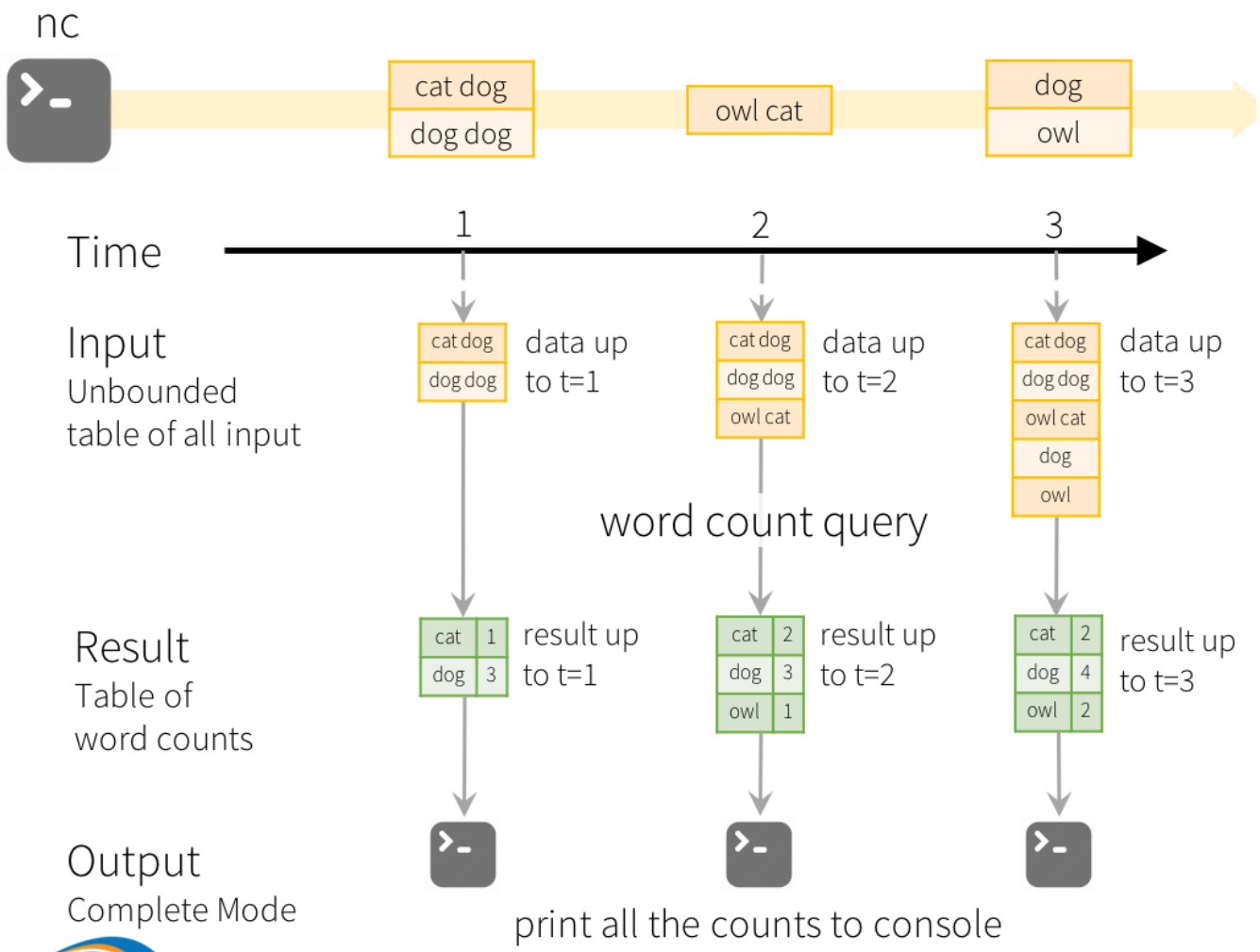
fit@hcmus

- Main idea:
  - Treat a stream of data as a table.
  - Data is continuously appended to this table.
- Therefore:
  - Should not have to change your code when doing batch or streaming processing
  - Only specify whether to run in a batch or streaming mode.

Programming Model for Structured Streaming



Example

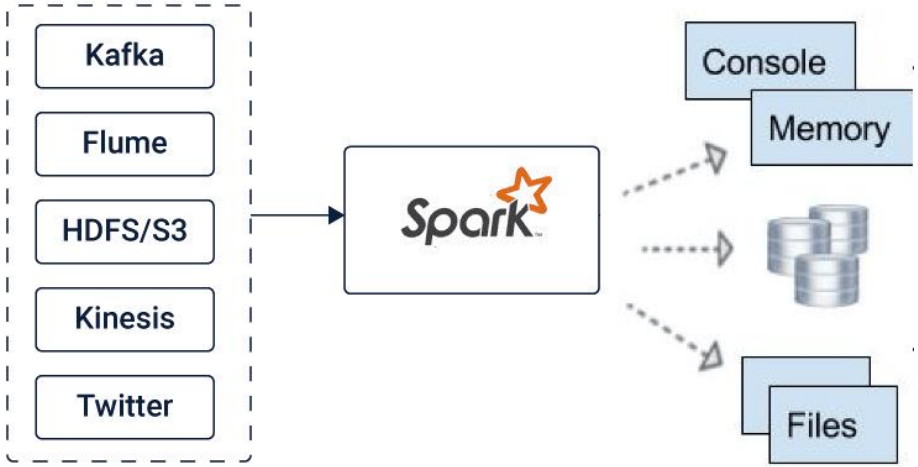

nc
cat dog
dog dog
owl cat
dog
owl
Time
1
2
3
Input
Unbounded
table of all input
cat dog
dog dog
data up
to t=1
cat dog
dog dog
owl cat
data up
to t=2
cat dog
dog dog
owl cat
dog
owl
data up
to t=3
word count query
Result
Table of
word counts
cat 1
dog 3
result up
to t=1
cat 2
dog 3
owl 1
result up
to t=2
cat 2
dog 4
owl 2
result up
to t=3
Output
Complete Mode
print all the counts to console

Outline

- Stream Processing
- Event and processing time, types of execution
- Spark Streaming
- **Input and Output**
- Structured Streaming Query
- Windows
- Handling Late Data
- Metrics and Monitoring

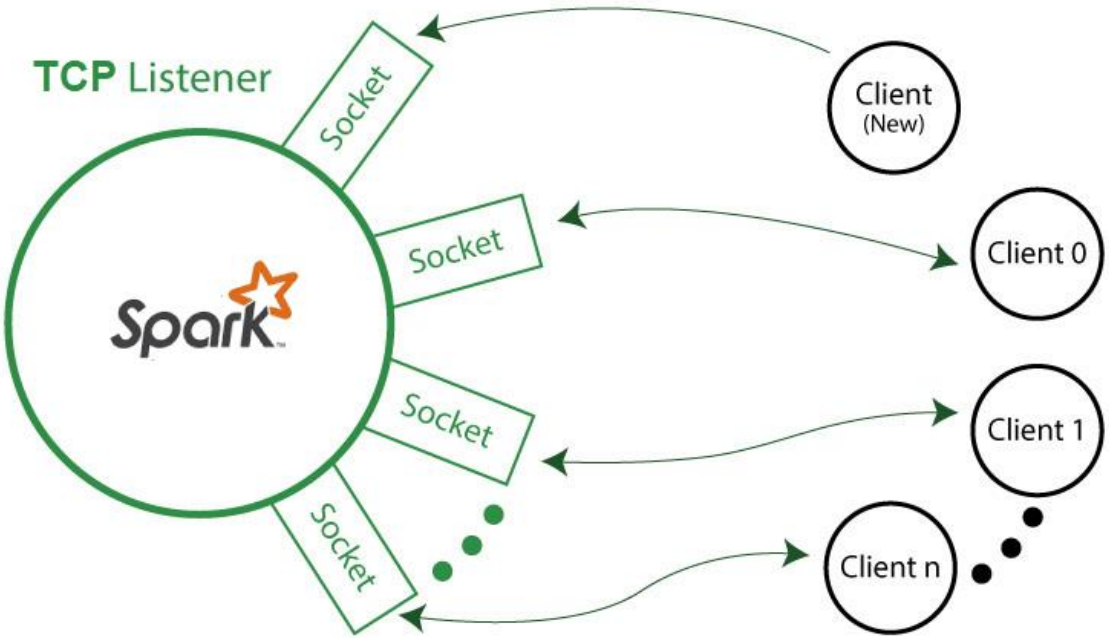
Source of Streaming

- Data can be constructed from:
  - Sockets
  - HDFS files
  - Kafka
  - Custom receivers
- Sink after processing:
  - Console
  - Memory
  - Files
  - Kafka
  - Custom



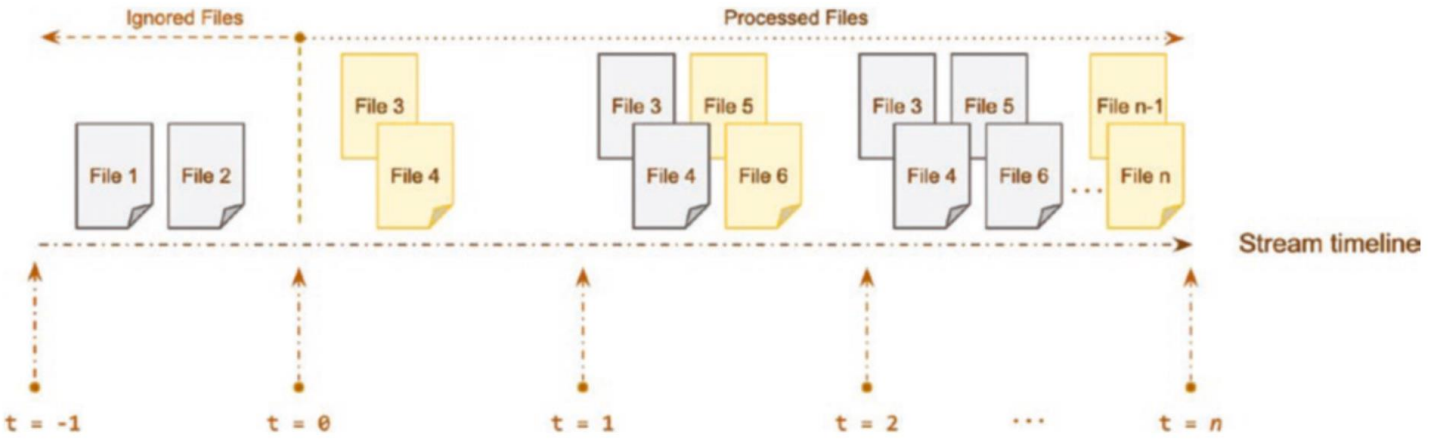
Socket Stream

- Listen to any socket on remote machines
- Need to configure host and port



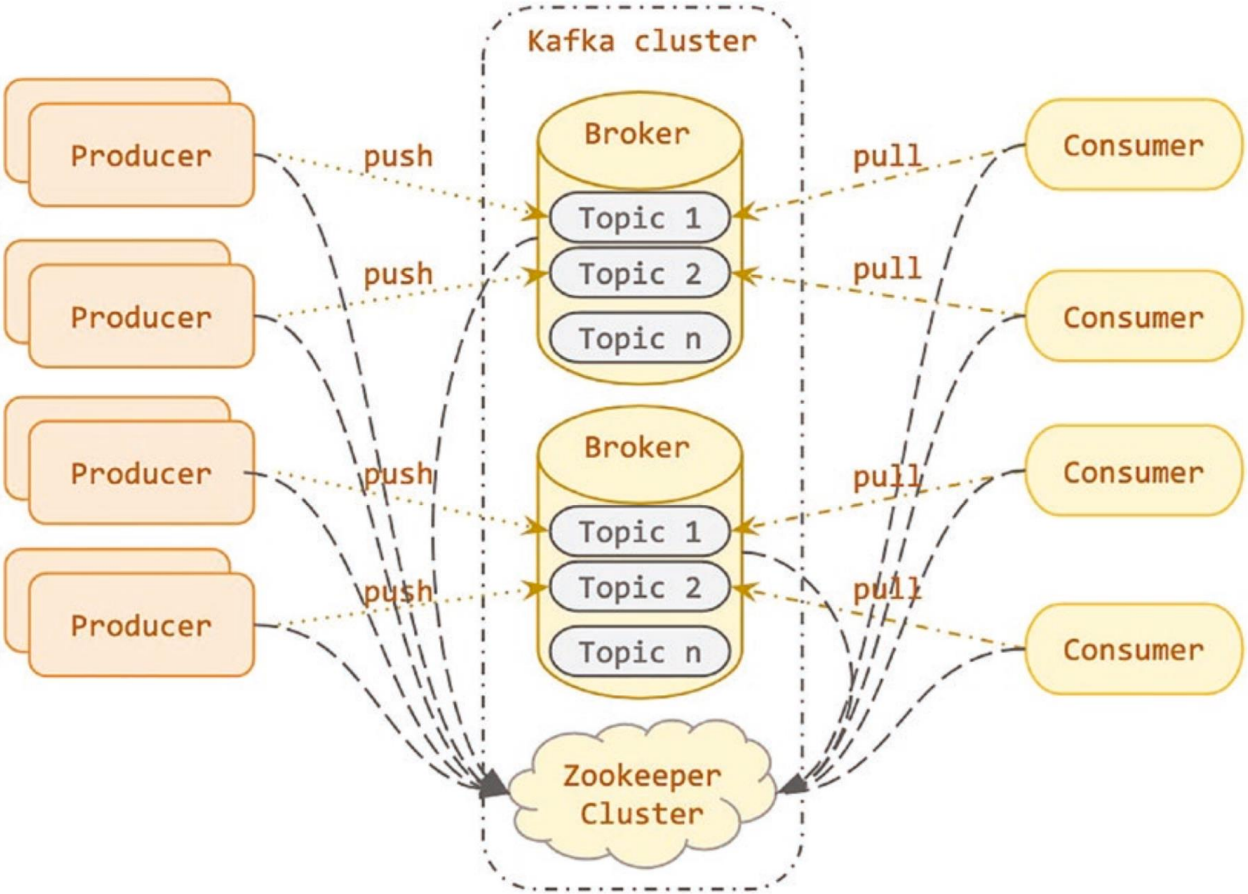
File Stream

- Track new files in a given directory on HDFS
  - Whenever there is new file appears, spark streaming will pick it up.
  - Only works for new files, modification for existing files will not be considered.
  - Tracked using file creation time.



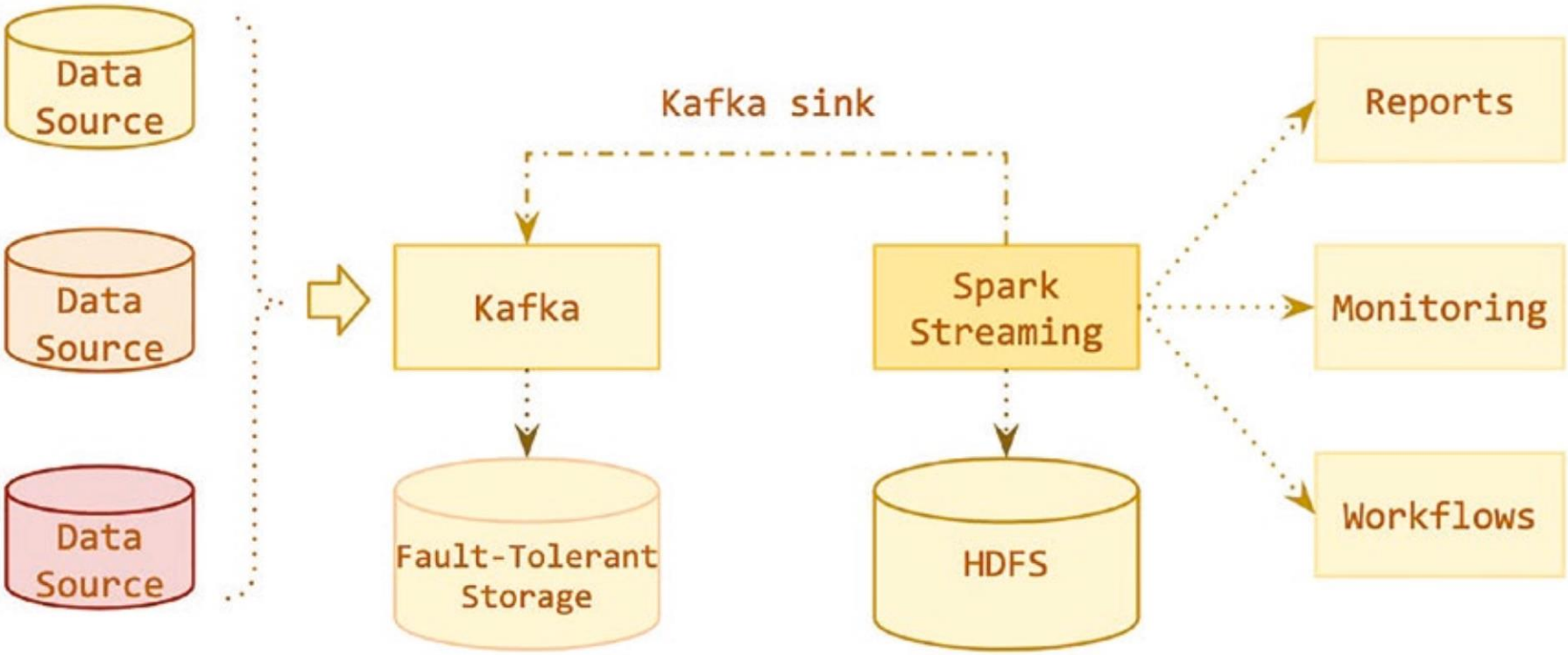
Kafka streaming

- Build on Producer-Subcriber from topics



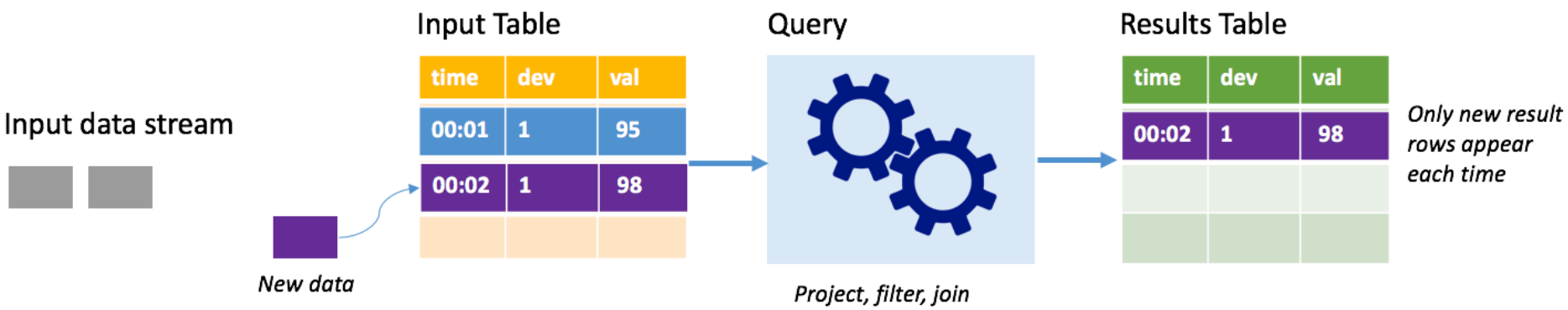
Kafka streaming

- Build on Producer-Subcriber from topics



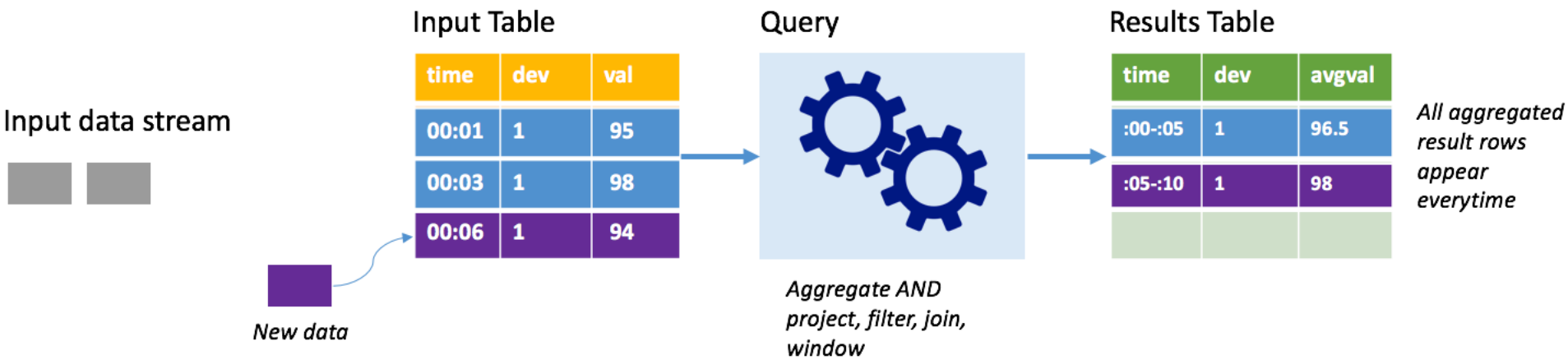
Output Modes

- There are three output modes:
  - Append mode (default): only new rows.



Output Modes

- There are three output modes:
  - Append mode (default): only new rows.
  - Complete mode: entire table is written.



Output Modes

- There are three output modes:
  - Append mode (default): only new rows.
  - Complete mode: entire table is written.
  - Update mode: only updated rows.

Outline

- Stream Processing
- Event and processing time, types of execution
- Spark Streaming
- Input and Output
- **Structured Streaming Query**
- Windows
- Handling Late Data
- Metrics and Monitoring

@lnthanh

Develop Structured Streaming Queries

fit@hcmus

- Five Steps to Define a Streaming Query:
  - Step 1: Define input sources
    - Define a DataFrame from a streaming source.
    - Use spark.readStream to create a DataStreamReader.

```
val spark = SparkSession...
val lines = spark
  .readStream.format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()
```

@lnthanh

Develop Structured Streaming Queries

fit@hcmus

- Five Steps to Define a Streaming Query:
  - Step 1: Define input sources
  - Step 2: Transform data
    - Apply the usual DataFrame operations.

```
import org.apache.spark.sql.functions._
val words = lines.select(split(col("value"), "\\s").as("word"))
val counts = words.groupBy("word").count()
```

Stateful vs Stateless operations

fit@hcmus

- Stateful operations maintain state across multiple batches.
  - For example, *count()* requires maintaining state to combine data across multiple rows.
- Stateless operations focus only current state.
  - Operations like *select()*, *filter()*, *map()* do not require any information from previous rows to process the next row; each row can be processed by itself.

Stateful vs Stateless operations

fit@hcmus

- Example: A stateful computation of the Fibonacci elements

```
scala> val ints = Stream.from(0)
ints: scala.collection.immutable.Stream[Int] = Stream(0, ?)

scala> val fibs = (ints.scanLeft((0, 1)){ case ((previous, current), index) =>
        (current, (previous + current))})

fibs: scala.collection.immutable.Stream[(Int, Int)] = Stream((0,1), ?)

scala> fibs.take(8).print
(0,1), (1,1), (1,2), (2,3), (3,5), (5,8), (8,13), (13,21), empty

Scala> fibs.map{ case (x, y) => x}.take(8).print
0, 1, 1, 2, 3, 5, 8, 13, empty
```

@lnthanh

Stateful vs Stateless operations

fit@hcmus

- Example: A stateless computation of the Fibonacci elements

```
scala> import scala.math.{pow, sqrt}
import scala.math.{pow, sqrt}

scala> val phi = (sqrt(5)+1) / 2
phi: Double = 1.618033988749895

scala> def fibonacciNumber(x: Int): Int =
  ((pow(phi,x) - pow(-phi,-x))/sqrt(5)).toInt
fibonacciNumber: (x: Int)Int

scala> val integers = Stream.from(0)
integers: scala.collection.immutable.Stream[Int] = Stream(0, ?)
scala> integers.take(10).print
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, empty

scala> val fibonacciSequence = integers.map(fibonacciNumber)
fibonacciSequence: scala.collection.immutable.Stream[Int] = Stream(0, ?)

scala>fibonacciSequence.take(8).print
0, 1, 1, 2, 3, 5, 8, 13, empty
```

Stateful vs Stateless operations

fit@hcmus

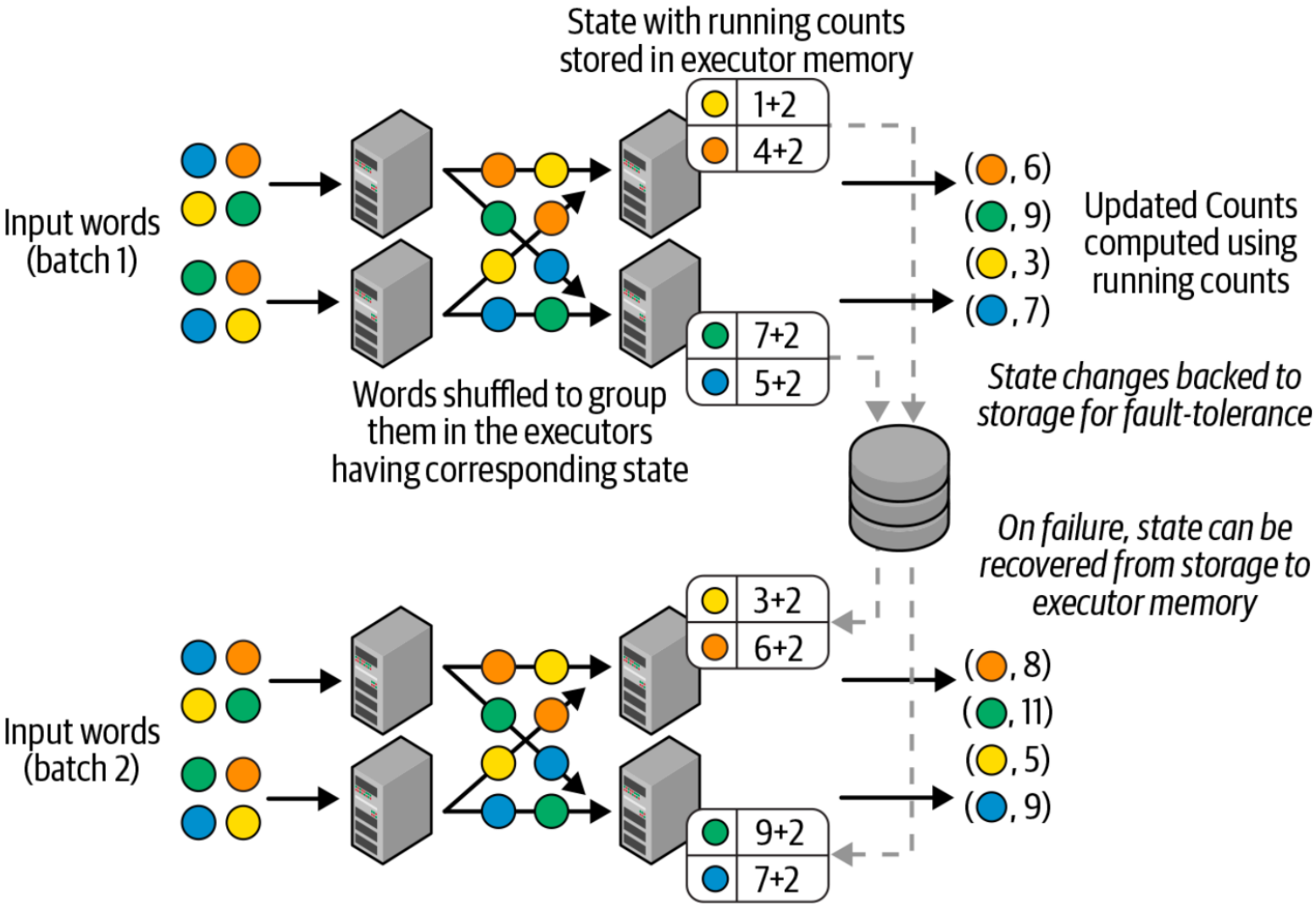
- Stateful operation usually happen in real analysis applications:
  - How long was the user session on our site?
  - What was the path the taxi used across the city?
  - What is the moving average of the pressure sensor on an industrial machine?

Fault-tolerant state management

fit@hcmus

- For stateful stream processing queries, besides writing to sinks, each micro-batch of tasks generates intermediate state data which will be consumed by the next micro-batch.
  - This state data generation is completely partitioned and distributed, and it is cached in the executor memory for efficient consumption.
  - However, it is not sufficient to just keep this state in memory, as any failure will cause the in-memory state to be lost.
  - To avoid loss, we synchronously save the key/value state update as change logs in the checkpoint location provided by the user.

Fault-tolerant state management



Develop Structured Streaming Queries

fit@hcmus

- Five Steps to Define a Streaming Query:
  - Step 1: Define input sources
  - Step 2: Transform data
  - Step 3: Define output sink and output mode
    - Write the processed output data with DataFrame.writeStream.
    - Additional methods to specify the following:
      - Output writing details (where and how to write the output)
      - Processing details (how to process data and how to recover from failures)

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

@lnthanh

Develop Structured Streaming Queries

fit@hcmus

- Five Steps to Define a Streaming Query:
  - Step 1, 2, 3…
  - Step 4: Specify process details
    - Triggering details: indicates when to trigger the discovery and processing of newly available streaming data. There are 4 options:
      - *Default*: the next micro-batch is triggered as soon as the previous micro-batch has completed.
      - *Processing time with trigger interval*: the query will trigger micro-batches at that fixed interval.
      - *Available-now*: processes all the new data available in a single batch based on source options and then stops itself.
      - *Continuous*: process data continuously instead of in micro-batches.
    - Checkpoint location: this is a directory in any HDFS-compatible filesystem where a streaming query saves its progress information—that is, what data has been successfully processed.

```
import org.apache.spark.sql.streaming._
val checkpointDir = "..."
val writer2 = writer
  .trigger(Trigger.ProcessingTime("1 second"))
  .option("checkpointLocation", checkpointDir)
```

Develop Structured Streaming Queries

fit@hcmus

- Five Steps to Define a Streaming Query:
  - Step 1, 2, 3…
  - Step 4: Specify process details
  - Step 5: Start the query
    - The returned object of type streamingQuery represents an active query and can be used to manage the query.
    - start() is a nonblocking method, so it will return as soon as the query has started in the background.
      - If you want the main thread to block until the streaming query has terminated, you can use streamingQuery.awaitTermination().

```
val streamingQuery = writer2.start()
streamingQuery.awaitTermination()
```

WordCount Example

fit@hcmus

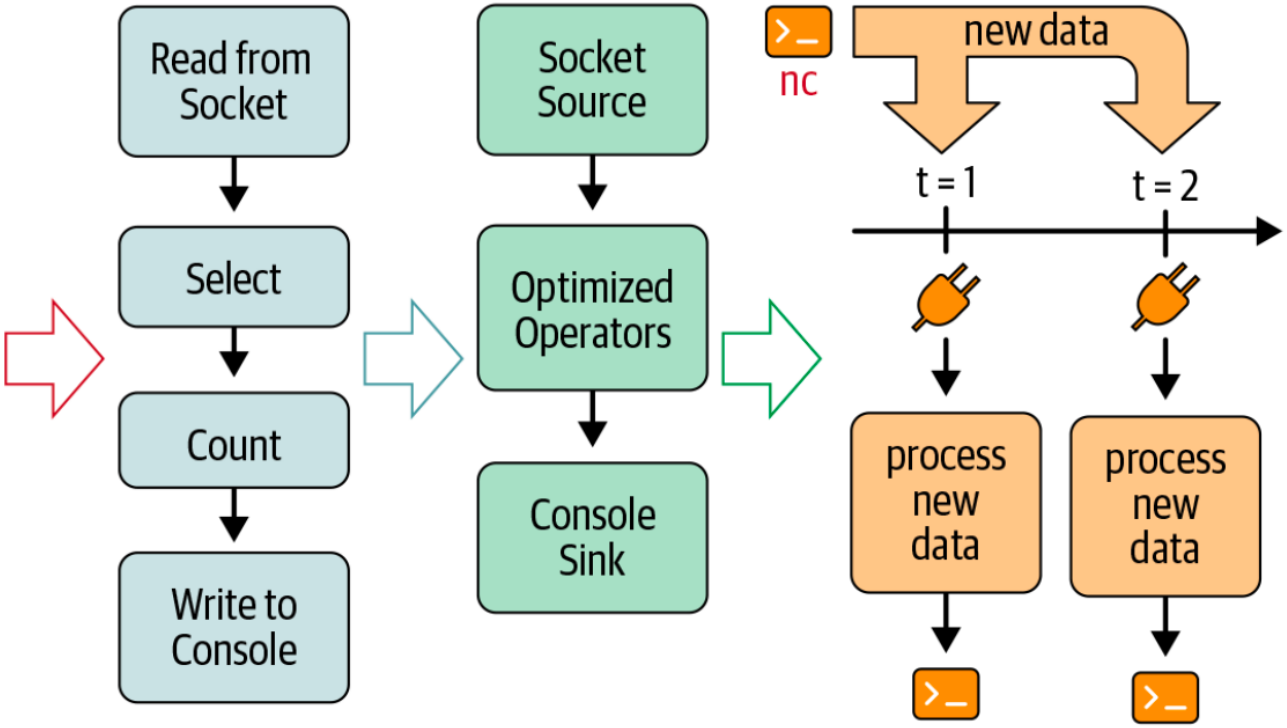
```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.streaming._
val spark = SparkSession...
val lines = spark
  .readStream.format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()

val words = lines.select(split(col("value"), "\\s").as("word"))
val counts = words.groupBy("word").count()

val checkpointDir = "..."
val streamingQuery = counts.writeStream
  .format("console")
  .outputMode("complete")
  .trigger(Trigger.ProcessingTime("1 second"))
  .option("checkpointLocation", checkpointDir)
  .start()
streamingQuery.awaitTermination()
```

Under the Hood

```
val lines = spark
   .readStream
   .format("socket")
   ...
val words = lines
   .select(...)
val counts = words
   .groupBy("word")
   .count()
val streamingQuery =
   counts
        .writeStream
        .format("console")
        ...
```



User Code

Logical Plan

Optimized Plan

Series of Incremental Execution Plans

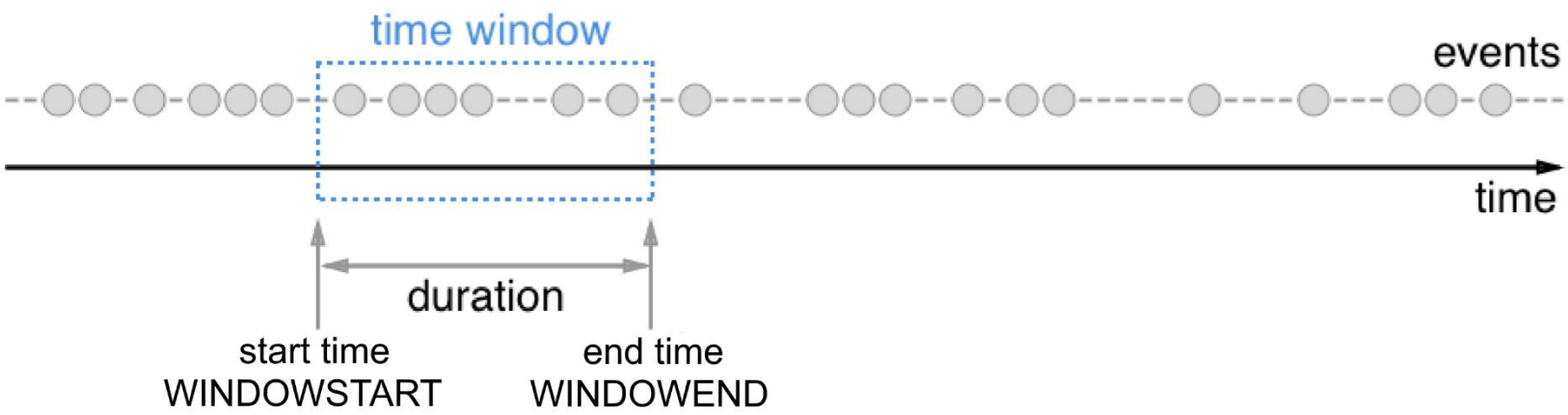
Outline

- Stream Processing
- Event and processing time, types of execution
- Spark Streaming
- Input and Output
- Structured Streaming Query
- **Windows**
- Handling Late Data
- Metrics and Monitoring

@lnthanh

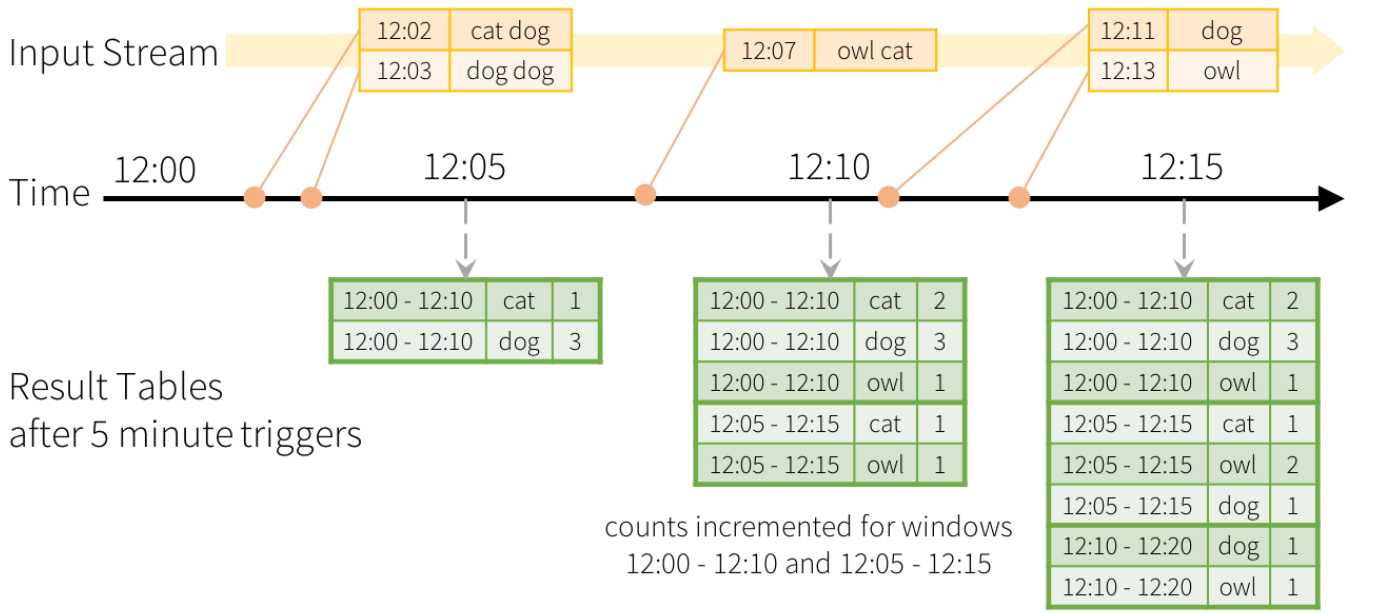
Event-time Windows

- In many cases, rather than running aggregations over the whole stream, you want aggregations over data bucketed by time windows.
  - For example, detect if any sensor is reporting an unusually high number of times.



Window Operations

- Scenario:
  - The stream contains lines along with the time when the line was generated.
  - We want to count words within 10 minute windows, updating every 5 minutes.



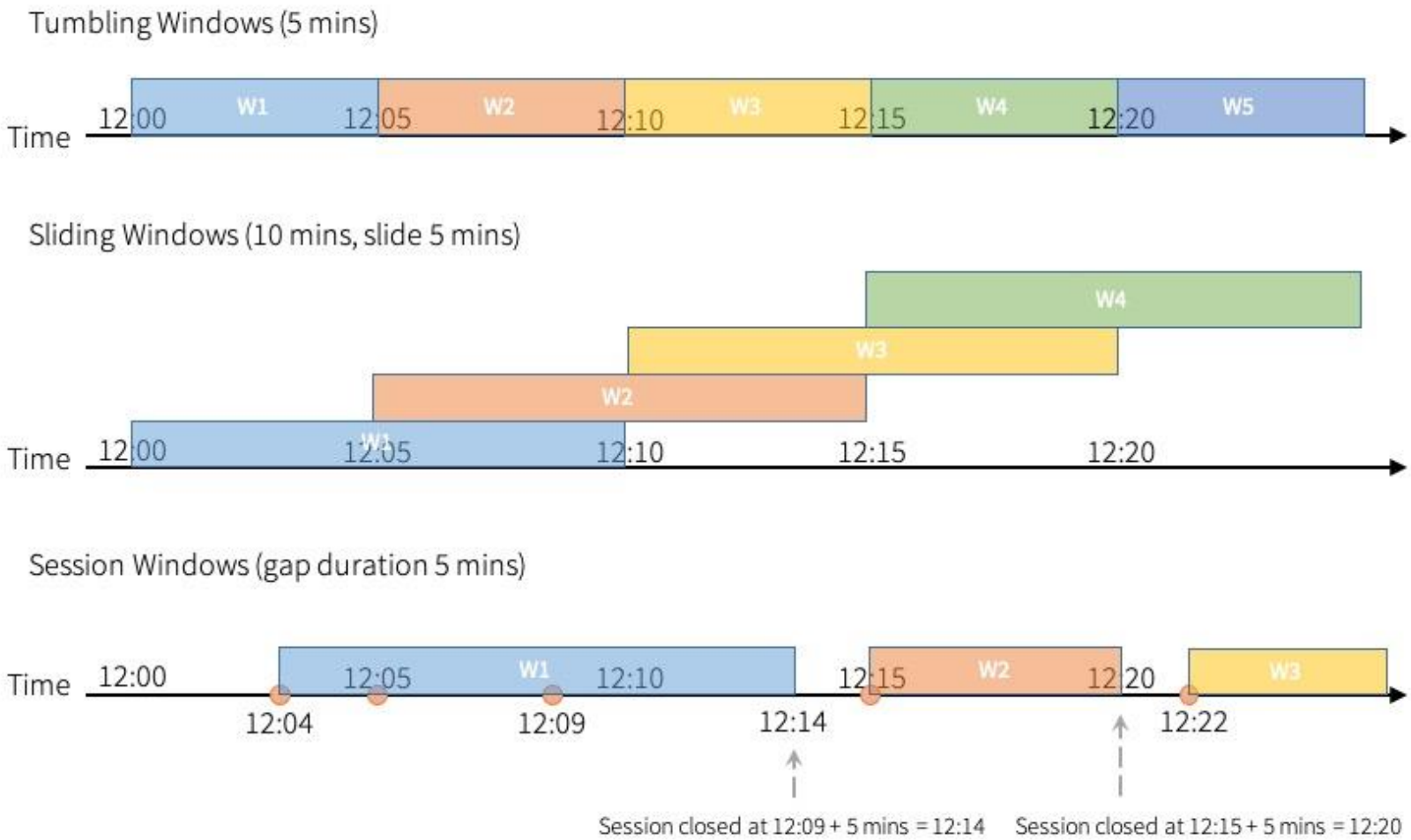
Windowed Grouped Aggregation
with 10 min windows, sliding every 5 mins

Window Operations

```
import org.apache.spark.sql.functions.*
sensorReadings
  .groupBy("sensorId", window("eventTime", "5 minute"))
  .count()
```

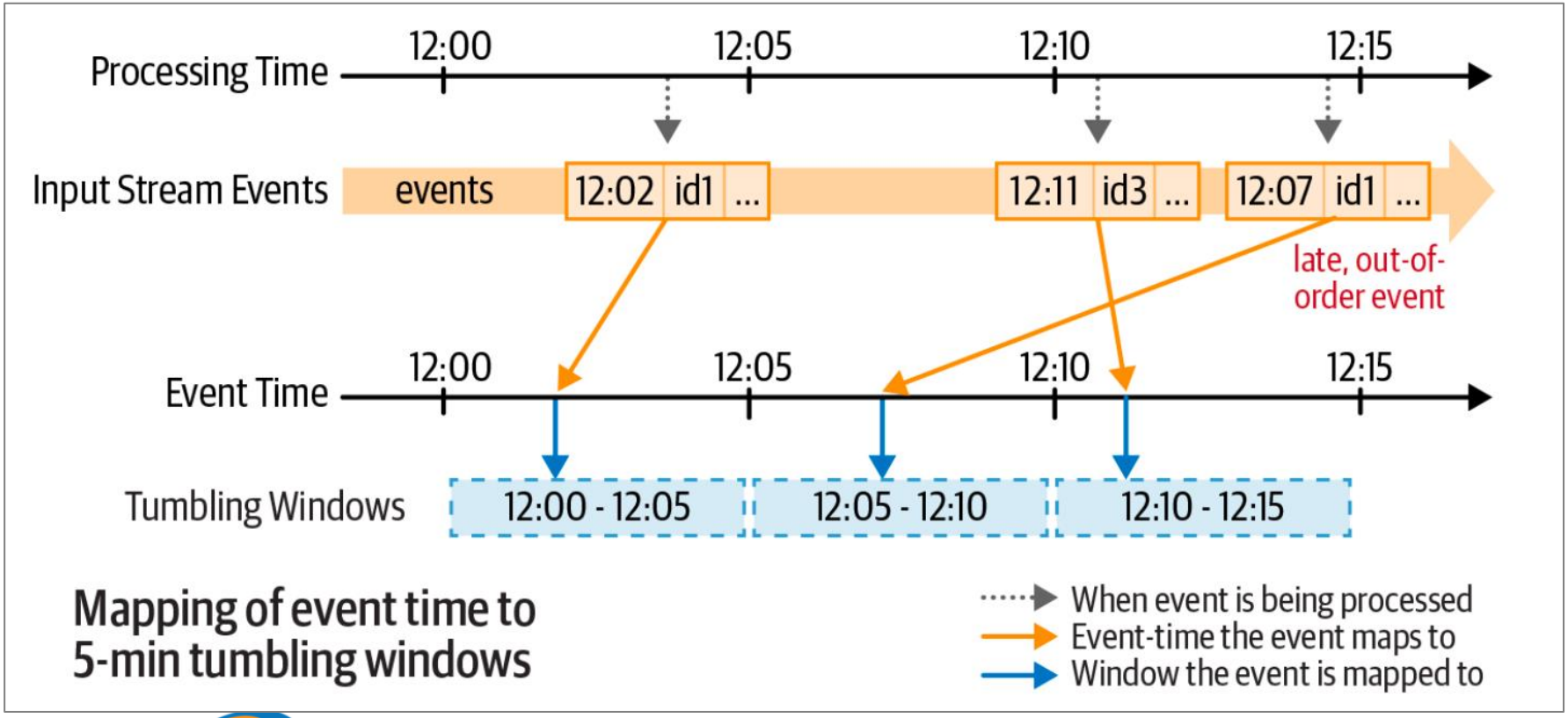
- The window() function express the five-minute windows as a dynamically computed grouping column.
- When started, this query do the following for each sensor reading:
  - Use the eventTime value to compute the five-minute time window the sensor reading falls into.
  - Group the reading based on the composite group (<computed window>, SensorId).
  - Update the count of the composite group.

Types of time windows



Tumbling Windows

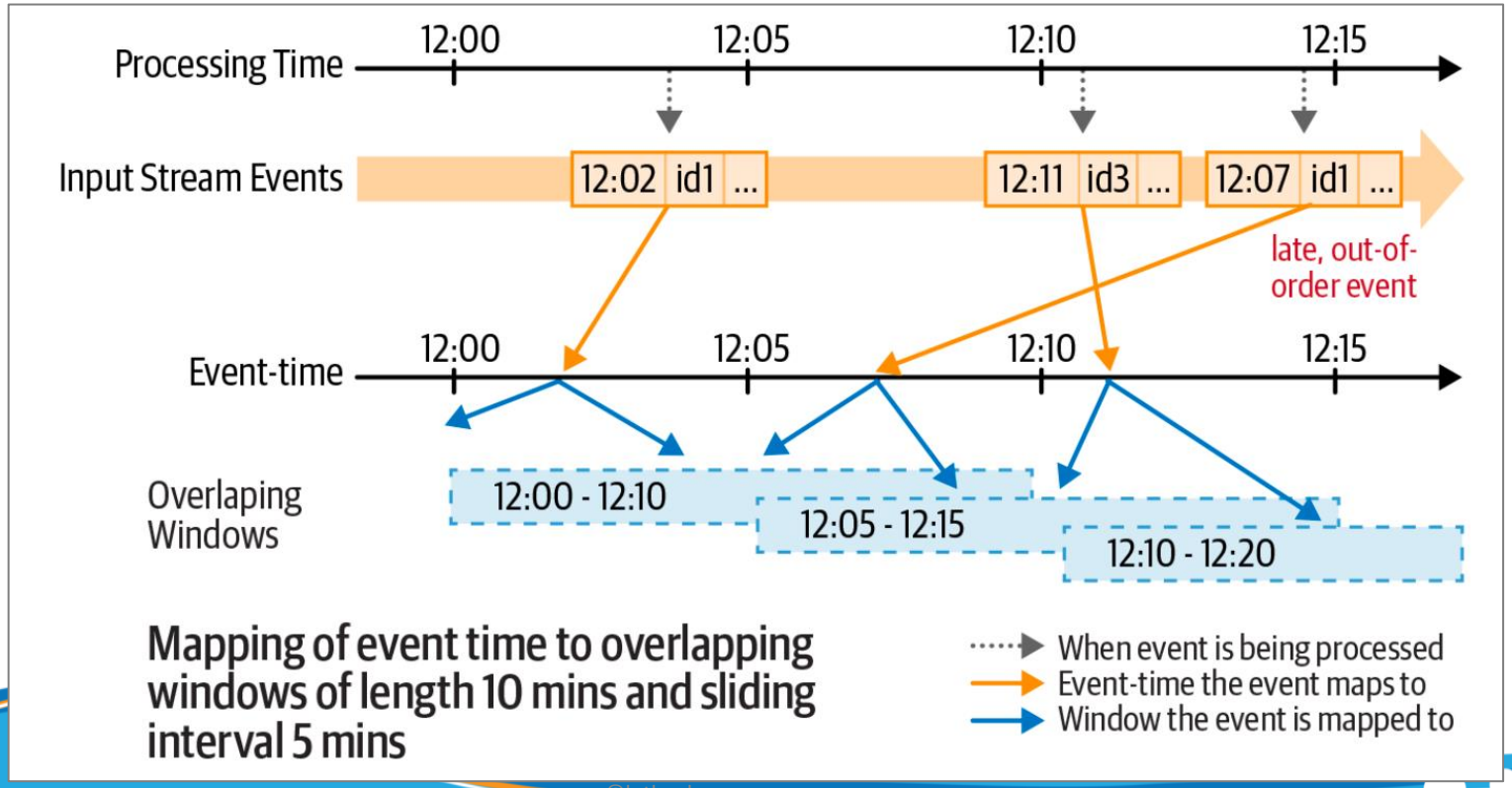
- Mapping of event time to tumbling windows



Sliding Windows

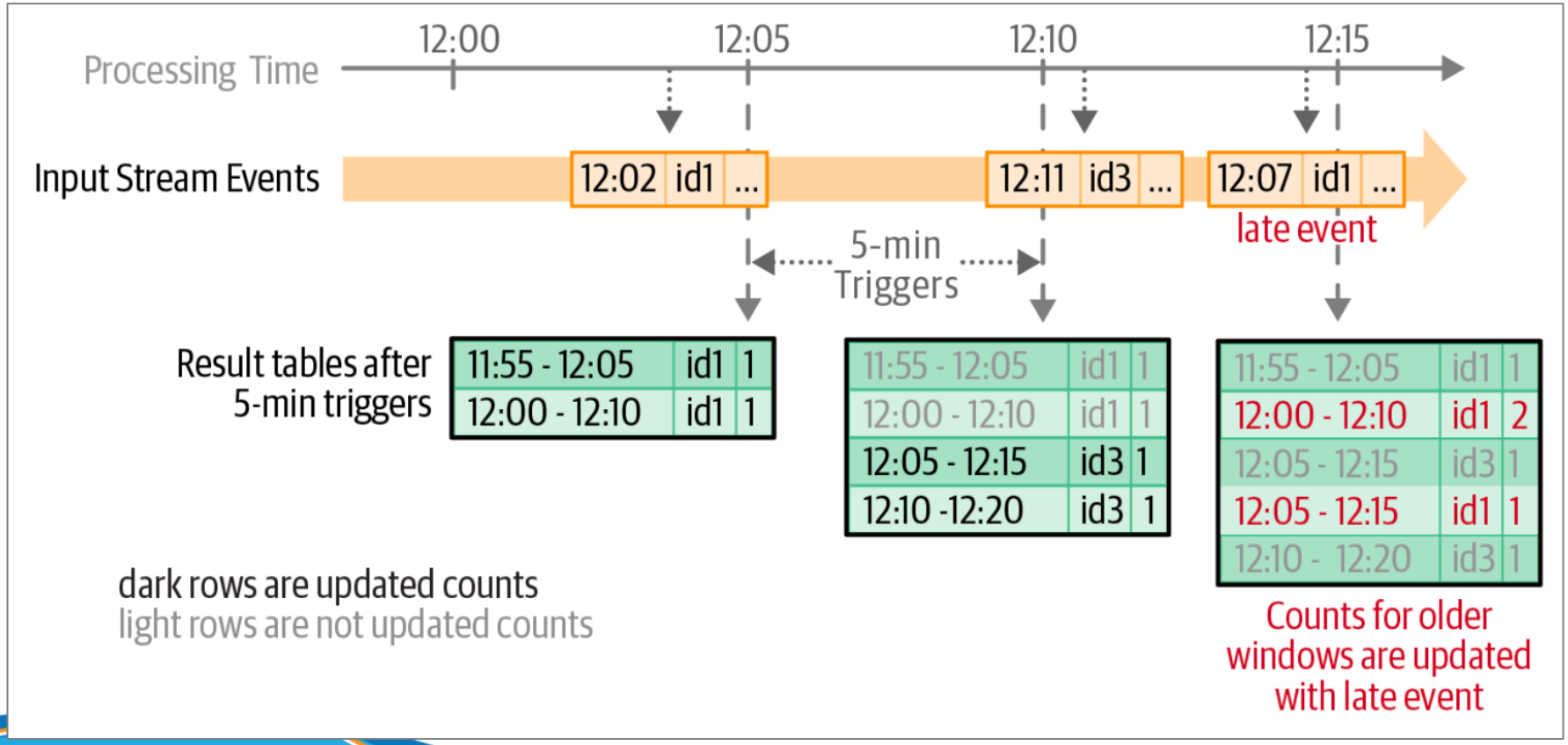
- Mapping of event time to multiple overlapping

```
sensorReadings
  .groupBy("sensorId", window("eventTime", "10 minute", "5 minute"))
  .count()
```



Sliding Windows

- Updated counts in the result table after each five-minute trigger

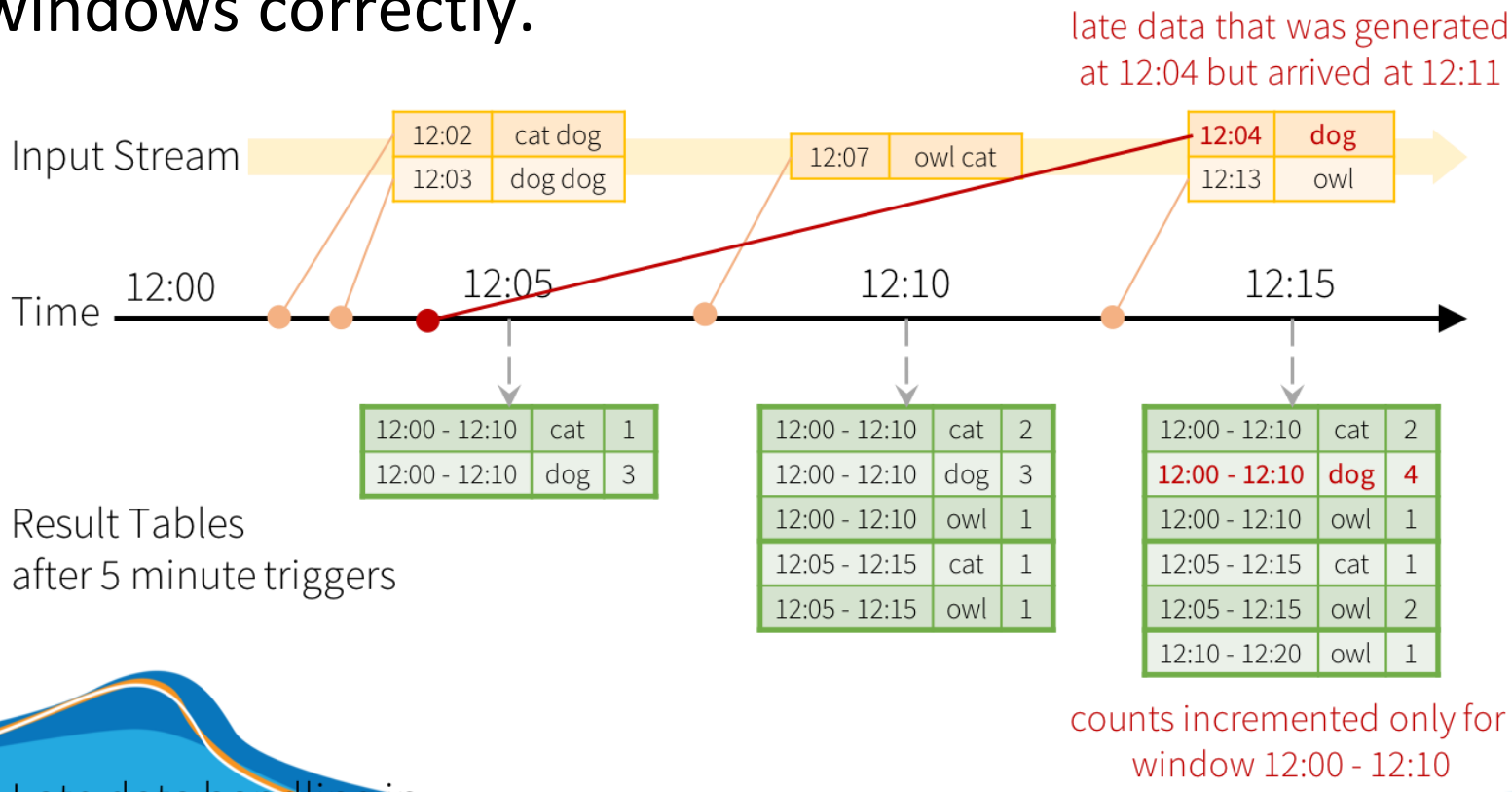


Outline

- Stream Processing
- Event and processing time, types of execution
- Spark Streaming
- Input and Output
- Structured Streaming Query
- Windows
- **Handling Late Data**
- Metrics and Monitoring

Handling Late Data

- Structured Streaming can maintain the intermediate state for partial aggregates for a long period of time such that late data can update aggregates of old windows correctly.



Late data handling in Windowed Grouped Aggregation

Watermarking

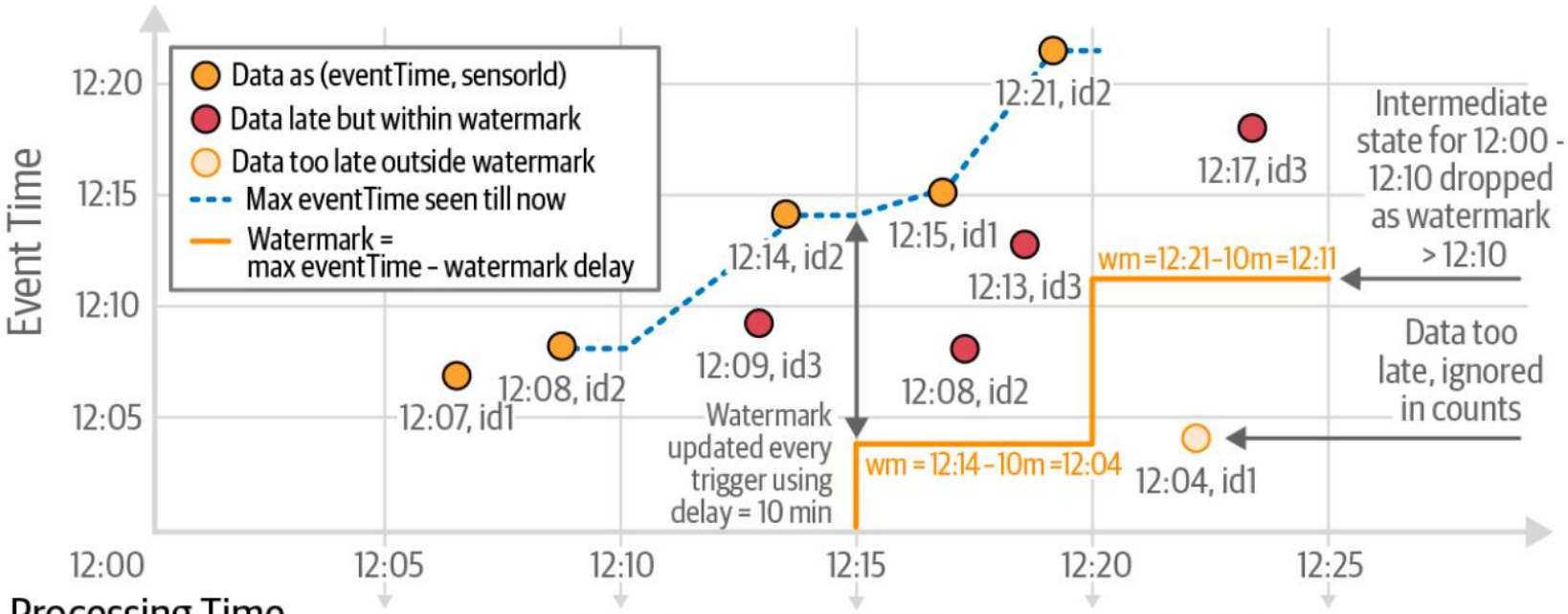
- The system to bound the amount of intermediate in-memory state it accumulates.
  - The system needs to know when an old aggregate can be dropped from the in-memory state because the application is not going to receive late data for that aggregate any more.

=> **watermarking**

```
sensorReadings
  .withWatermark("eventTime", "10 minutes")
  .groupBy("sensorId", window("eventTime", "10 minutes", "5 minute"))
  .mean("value")
```

Note that you must call withWatermark() before the groupBy() and on the same timestamp column as that used to define windows.

Watermarking



Processing Time with 5 min triggers

Result Tables after each trigger

12:00 - 12:10	id1	1
12:00 - 12:10	id2	1
12:05 - 12:15	id1	1
12:05 - 12:15	id2	1

12:00 - 12:10	id1	1
12:00 - 12:10	id2	1
12:00 - 12:10	id3	1
12:05 - 12:15	id1	1
12:05 - 12:15	id2	2
12:05 -12:15	id3	1
12:10 -12:20	id2	1

dark rows are updated counts

12:00 - 12:10	id1	1
12:00 - 12:10	id2	2
12:00 - 12:10	id3	1
12:05 - 12:15	id1	2
12:05 - 12:15	id2	3
12:05 - 12:15	id3	2
12:10 - 12:20	id2	1
12:10 - 12:20	id1	1
12:10 - 12:20	id3	1
...		

12:00 - 12:10	id1	1
12:00 - 12:10	id2	2
12:00 - 12:10	id3	1
12:05 - 12:15	id1	2
12:05 - 12:15	id2	3
12:05 - 12:15	id3	2
12:10 - 12:20	id2	1
12:10 - 12:20	id1	1
12:10 - 12:20	id3	2
...		

Table *not* updated with too late data (12:04, id1)

Table updated with late data (12:17, id3)

Watermarking in Windowed Grouped Counts

Outline

- Stream Processing
- Event and processing time, types of execution
- Spark Streaming
- Input and Output
- Structured Streaming Query
- Windows
- Handling Late Data
- **Metrics and Monitoring**

@lnthanh

Query Status

- To get the status of a given query, simply running the command query.status will return the current status of the stream.

```
query.status
```

```
{
  "message" : "Getting offsets from ...",
  "isDataAvailable" : true,
  "isTriggerActive" : true
}
```

Recent process

- By running query.recentProgress, get access to more time-based information like the processing rate, batch durations, input sources and output sinks.

```
query.recentProgress
```


```
Array({
  "id" : "d9b5eac5-2b27-4655-8dd3-4be626b1b59b",
  "runId" : "f8da8bc7-5d0a-4554-880d-d21fe43b983d",
  "name" : "test_stream",
  "timestamp" : "2017-08-06T21:11:21.141Z",
  "numInputRows" : 780119,
  "processedRowsPerSecond" : 19779.89350912779,
  "durationMs" : {
    "addBatch" : 38179,
    "getBatch" : 235,
    "getOffset" : 518,
    "queryPlanning" : 138,
    "triggerExecution" : 39440,
    "walCommit" : 312
  },
  "stateOperators" : [ {
    "numRowsTotal" : 7,
    "numRowsUpdated" : 7
  } ],
  "sources" : [ {
    "description" : "FileStreamSource[/some/stream/source/]",
    "startOffset" : null,
    "endOffset" : {
      "logOffset" : 0
    },
    "numInputRows" : 780119,
    "processedRowsPerSecond" : 19779.89350912779
  } ],
  "sink" : {
    "description" : "MemorySink"
  }
})
```


@lnthanh

THE END