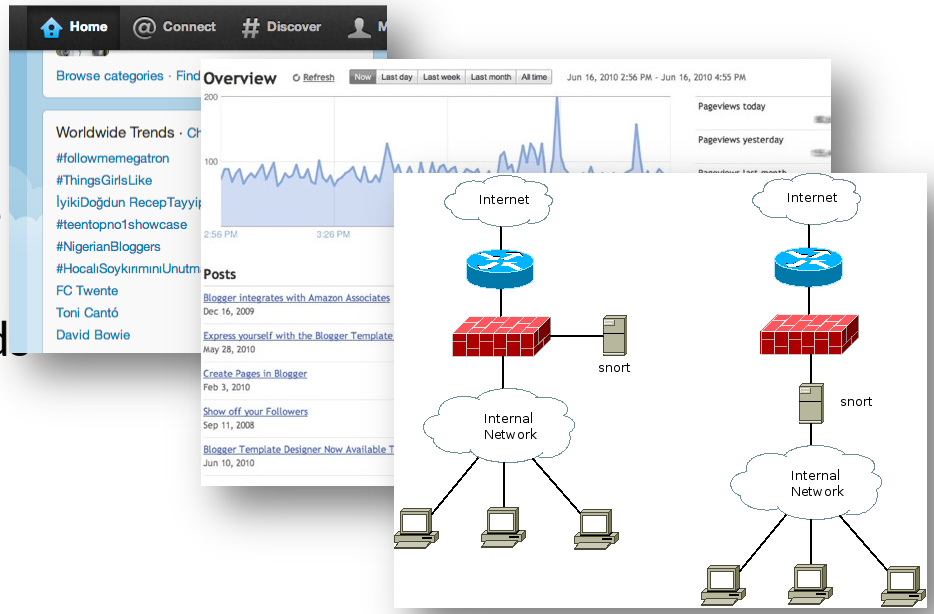
Chương 7

Các kĩ thuật xử lý luồng dữ liệu lớn

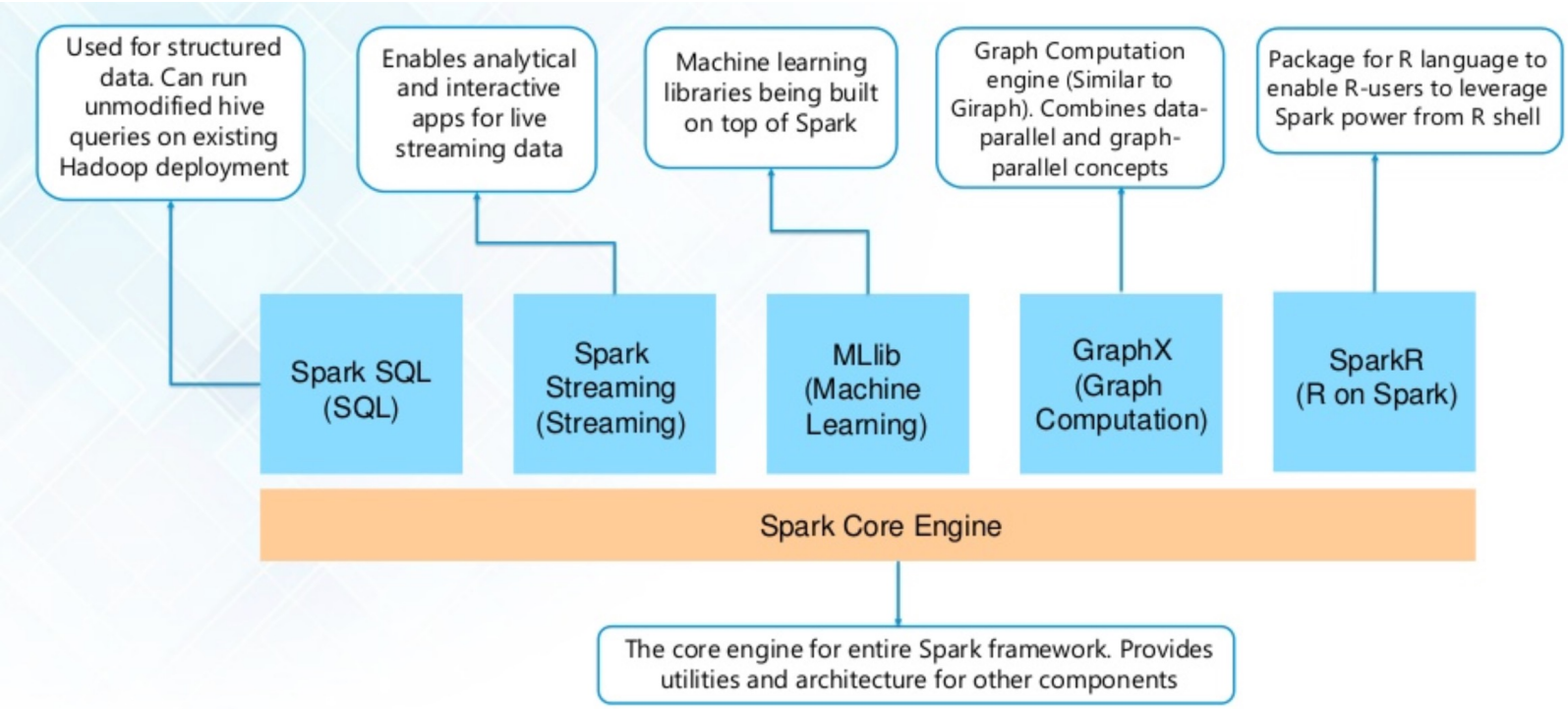
Spark streaming

Motivation

- Many important applications must process large streams of live data and provide results in near-real-time
  - Social network trends
  - Website statistics
  - Intrustion detection systems
  - etc.
- Require large clusters to handle
- Require latencies of few seconds



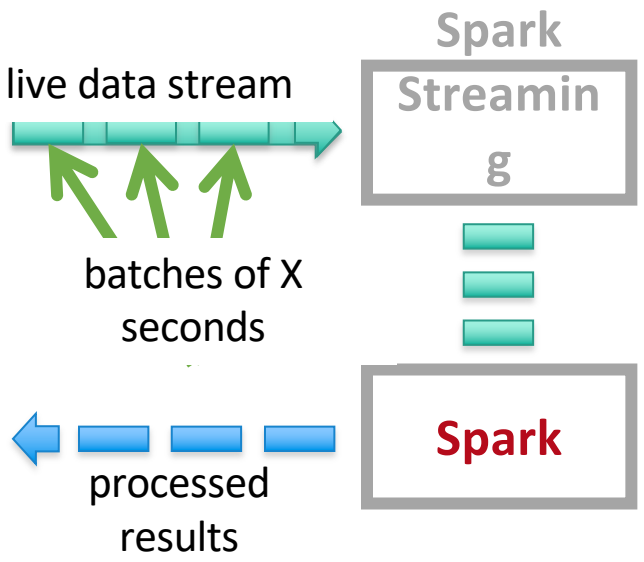
Apache Spark ecosystem



Discretized Stream Processing

Run a streaming computation as a **series of very small, deterministic batch jobs**

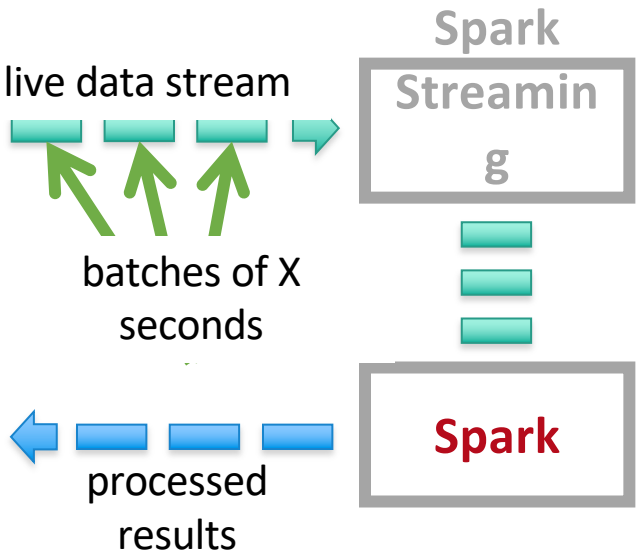
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



Discretized Stream Processing

Run a streaming computation as a **series of very small, deterministic batch jobs**

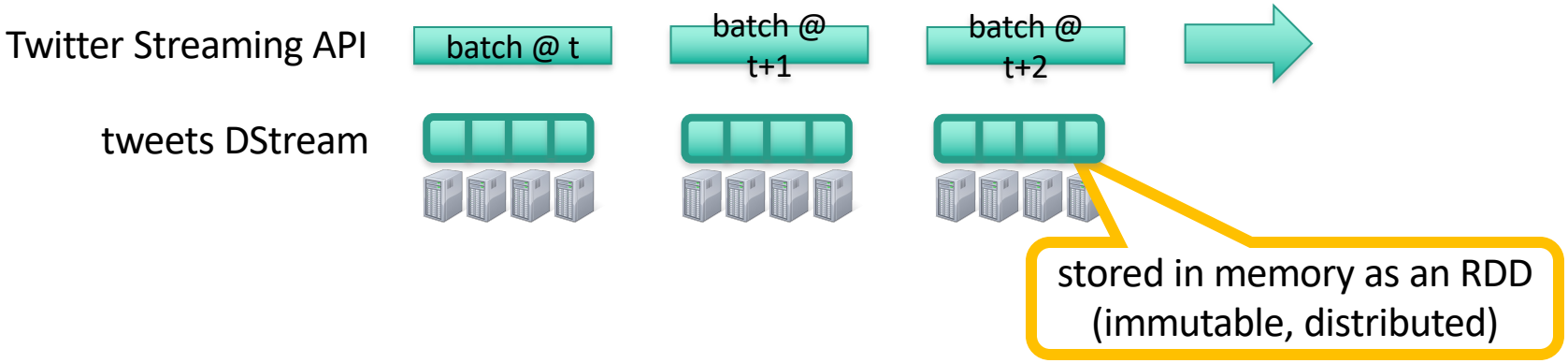
- Batch sizes as low as ½ second, latency ~ 1 second
- Potential for combining batch processing and streaming processing in the same system



Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
```

DStream: a sequence of RDD representing a stream of data

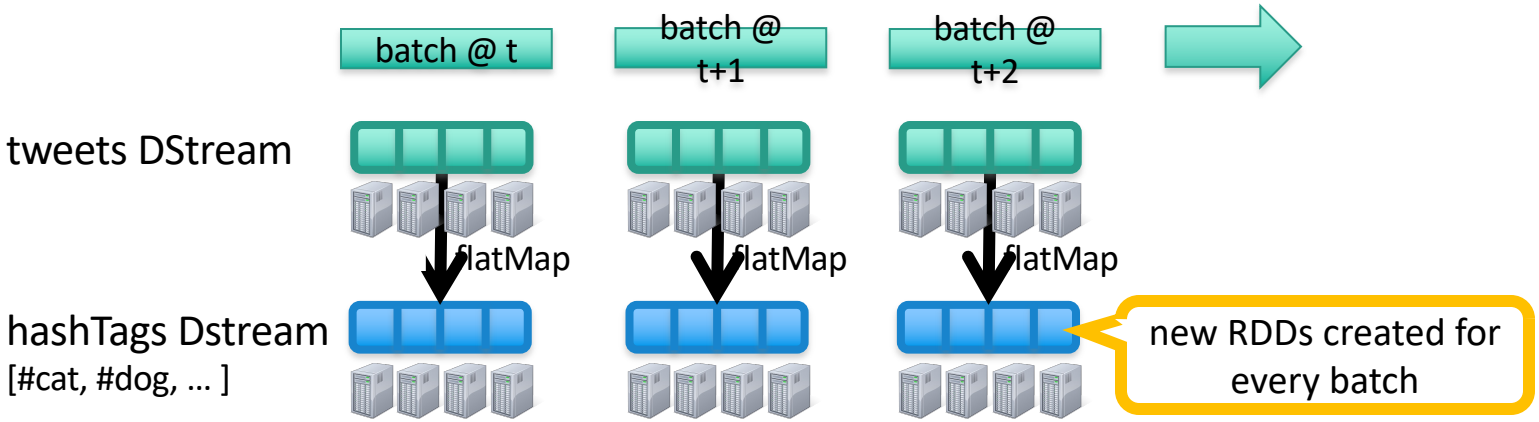


Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
```

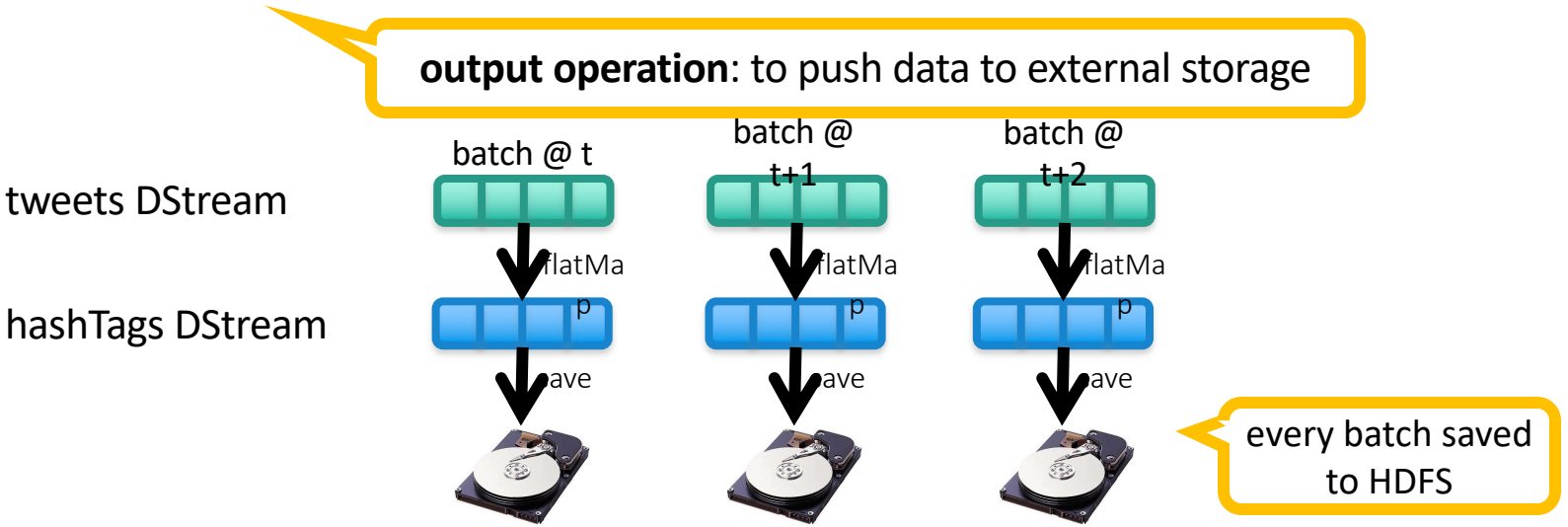
new DStream

transformation: modify data in one Dstream to create another DStream



Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```



Java Example

Scala

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)

val hashTags = tweets.flatMap (status => getTags(status))

hashTags.saveAsHadoopFiles("hdfs://...")
```

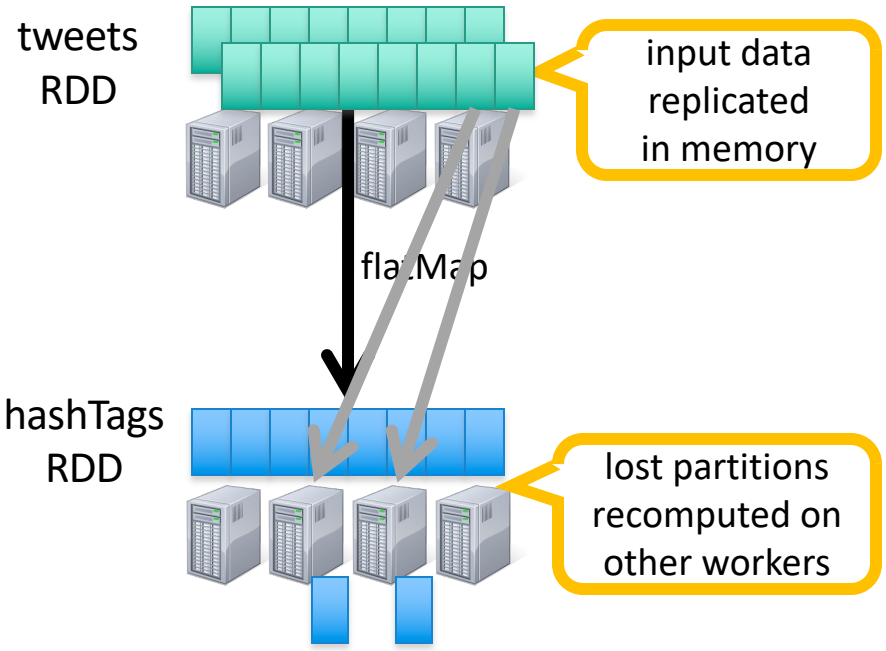
Java

```
JavaDStream<Status> tweets = ssc.twitterStream(<Twitter username>, <Twitter
password>)

JavaDstream<String> hashTags = tweets.fl

hashTags.saveAsHadoopFiles("hdfs://...")
```

Function object to define the transformation

Fault-tolerance

- RDDs are remember the sequence of operations that created it from the original fault-tolerant input data
- Batches of input data are replicated in memory of multiple worker nodes, therefore fault-tolerant
- Data lost due to worker failure, can be recomputed from input data

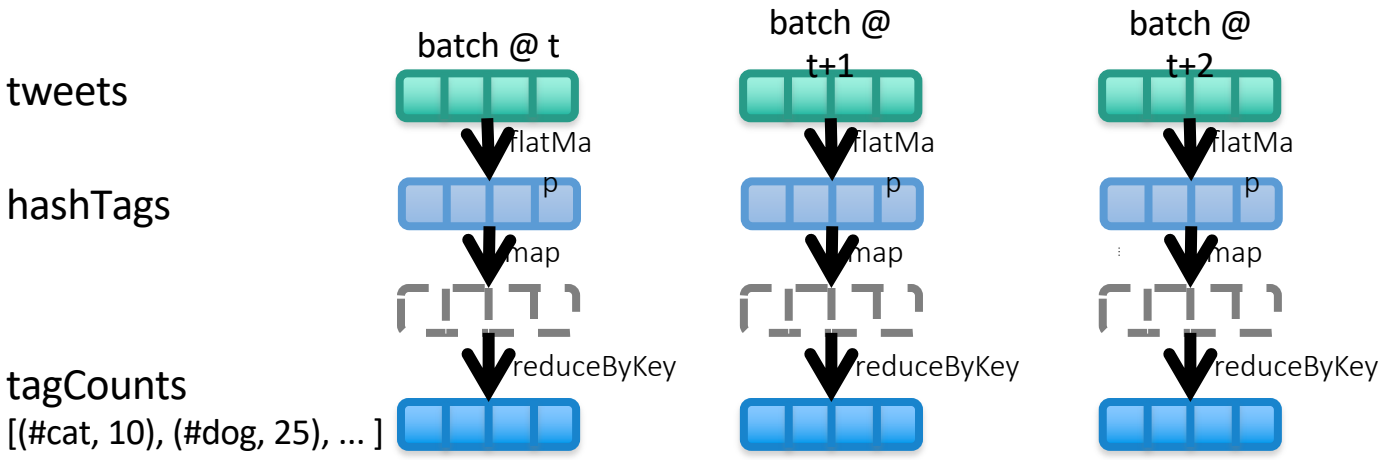


Key concepts

- **DStream** – sequence of RDDs representing a stream of data
  - Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets
- **Transformations** – modify data from on DStream to another
  - Standard RDD operations – map, countByValue, reduce, join, …
  - Stateful operations – window, countByValueAndWindow, …
- **Output Operations – send data to external entity**
  - saveAsHadoopFiles – saves to HDFS
  - foreach – do anything with each batch of results

Example 2 – Count the hashtags

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.countByValue()
```



Example 3 – Count the hashtags over last 10 mins

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```

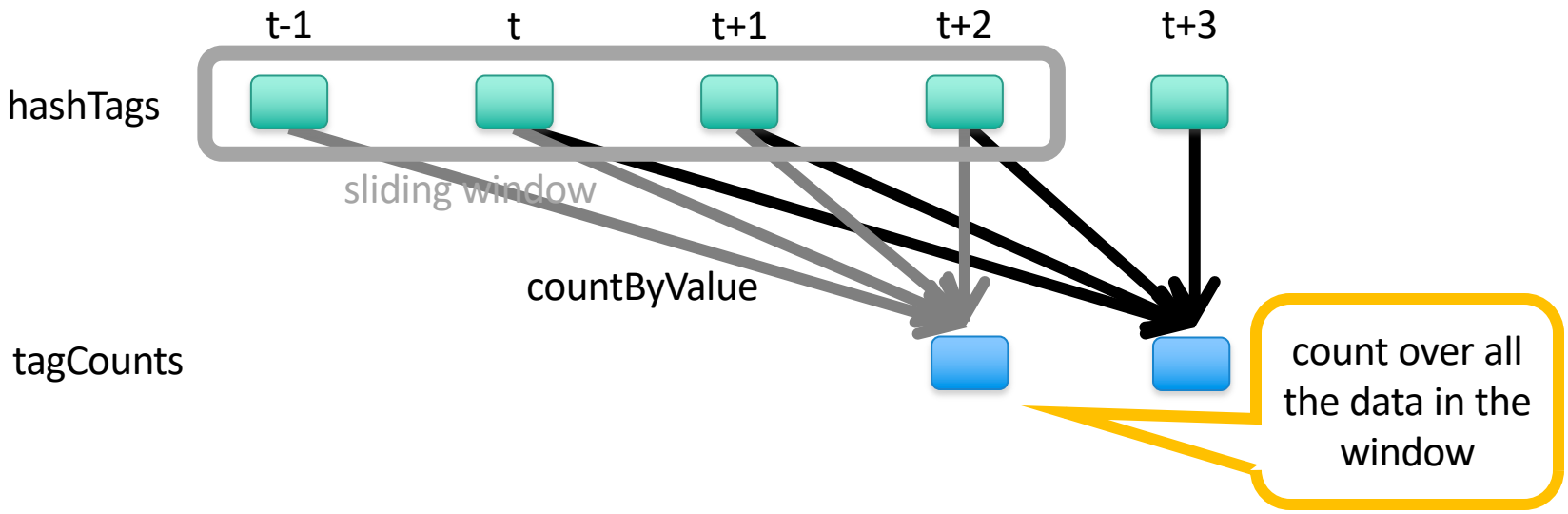
sliding window operation

window length

sliding interval

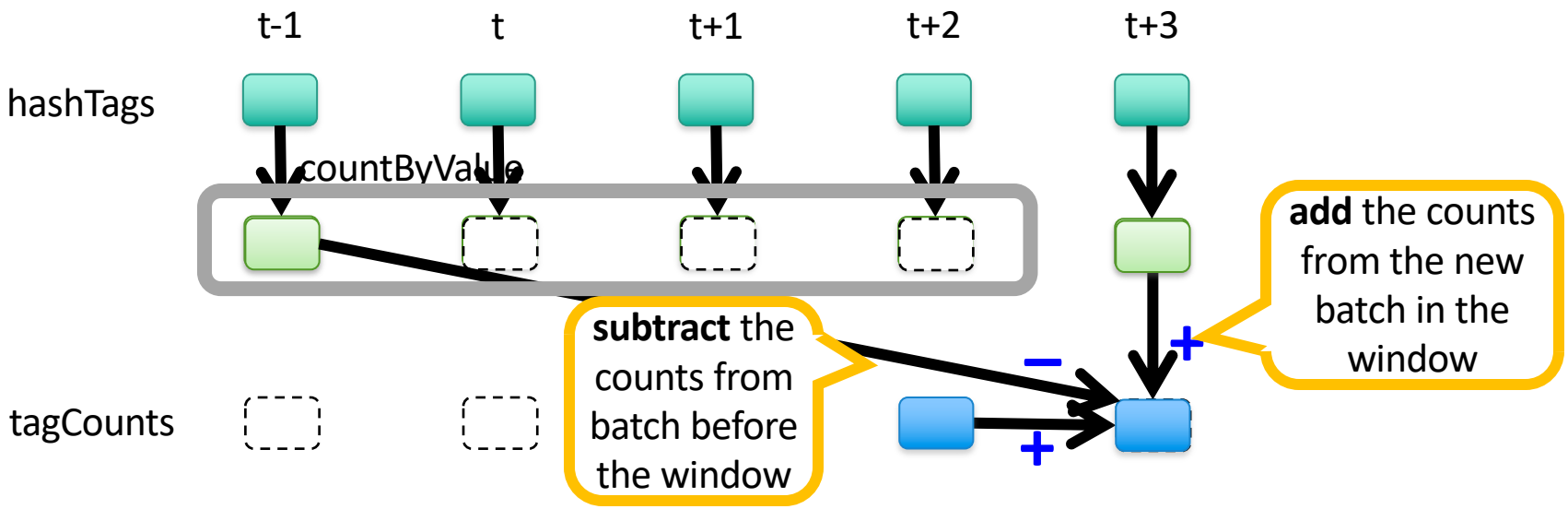
Example 3 – Counting the hashtags over last 10 mins

```
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



Smart window-based countByValue

```
val tagCounts = hashtags.countByValueAndWindow(Minutes(10), Seconds(1))
```



Smart window-based *reduce*

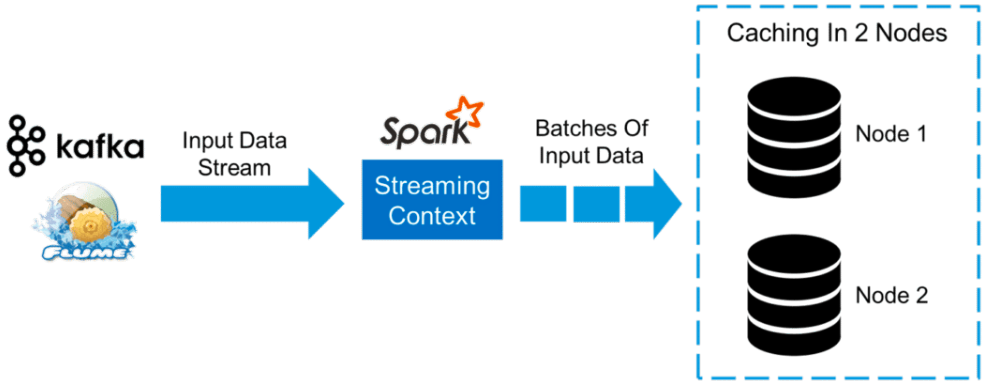
- Technique to incrementally compute count generalizes to many reduce operations
  - Need a function to “inverse reduce” (“subtract” for counting)

- Could have implemented counting as:

```
hashTags.reduceByKeyAndWindow(_ + _, _ - _, Minutes(1), …)
```

Caching/persistance

- DStream cho phép bảo lưu dữ liệu xuống vùng đệm hoặc lưu trữ lâu dài cho phép tái sử dụng các DStream này trong các biến đổi tiếp theo
- Sử dụng phương thức persist()



Accumulators

- Là các biến mà chỉ được thêm vào thông qua các phép toán kết hợp hoặc giao hoán (asociative, commutative)
- Được sử dụng cho phép toán count hoặc sum
- Các accumulators có thể được xem trên UI, rất có ích lợi trong hiểu quá trình thực thi các stage tính toán
- Spark mặc định hỗ trợ các accumulators số, có thể tạo accumulaftor có tên hoặc không tên

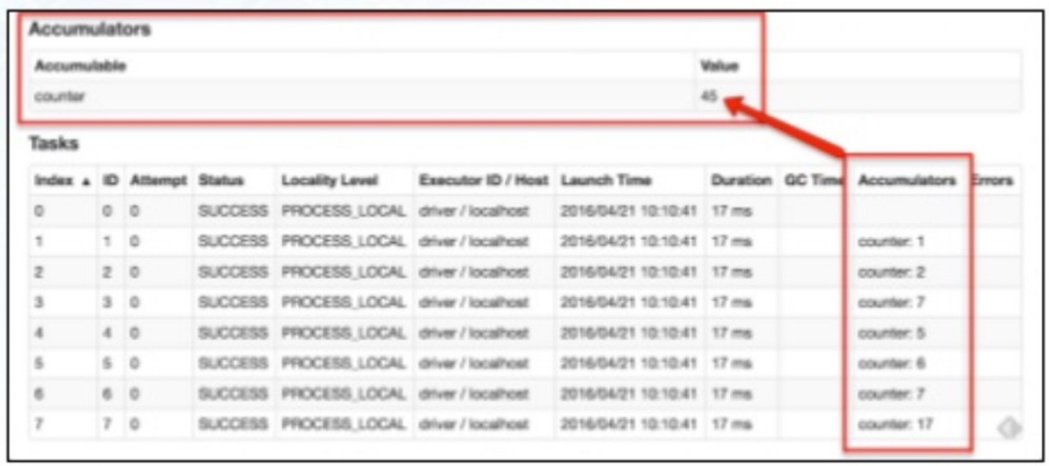


Figure: *Accumulators In Spark Streaming*

Broadcast Variables

- Các biến quảng bá cho phép nhà lập trình giữa một biến chỉ đọc trên mỗi node tính toán (cho phép chia sẻ dữ liệu trên tất cả các node)
- Spark sử dụng các giải thuật hiệu quả, tiết kiệm chi phí truyền thông để quảng bá các biến này

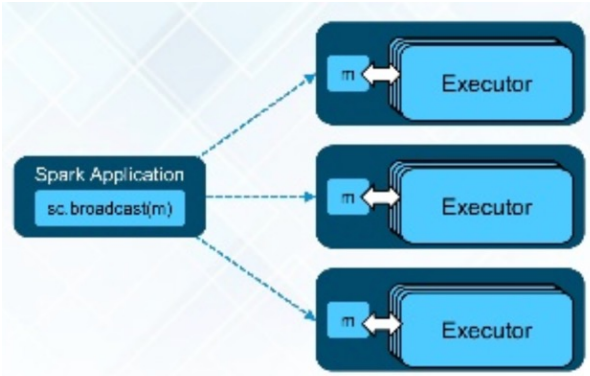


Figure: *Broadcasting A Value To Executors*

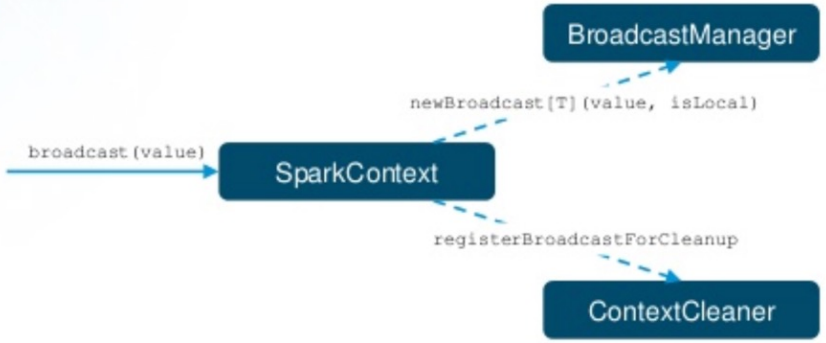
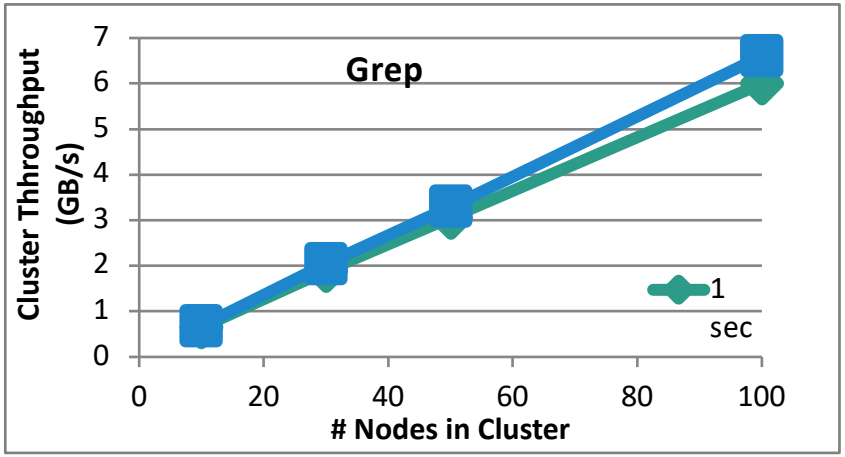


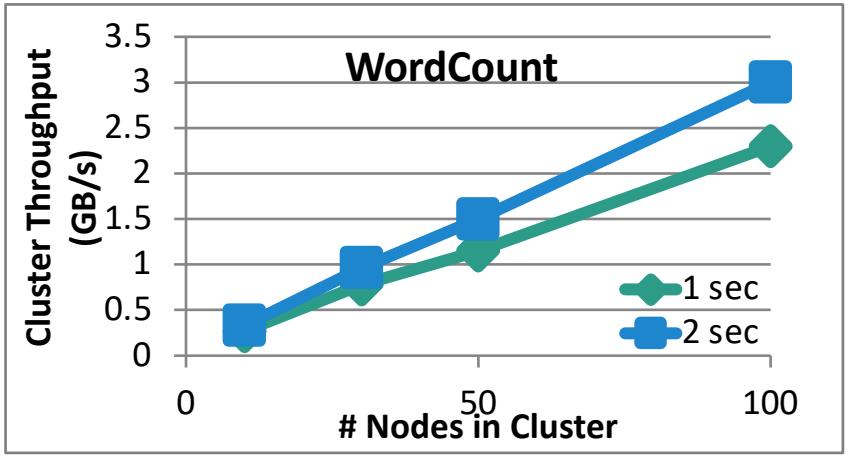
Figure: *SparkContext and Broadcasting*

Performance

Can process **6 GB/sec (60M records/sec)** of data on 100 nodes at **sub-second** latency

- Tested with 100 streams of data on 100 EC2 instances with 4 cores each

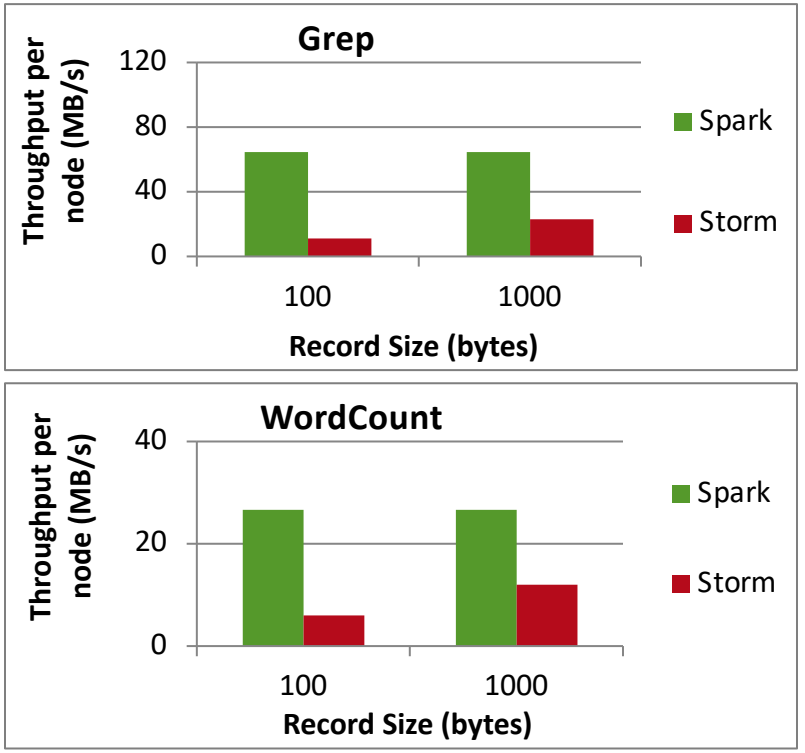




Comparison with Storm and S4

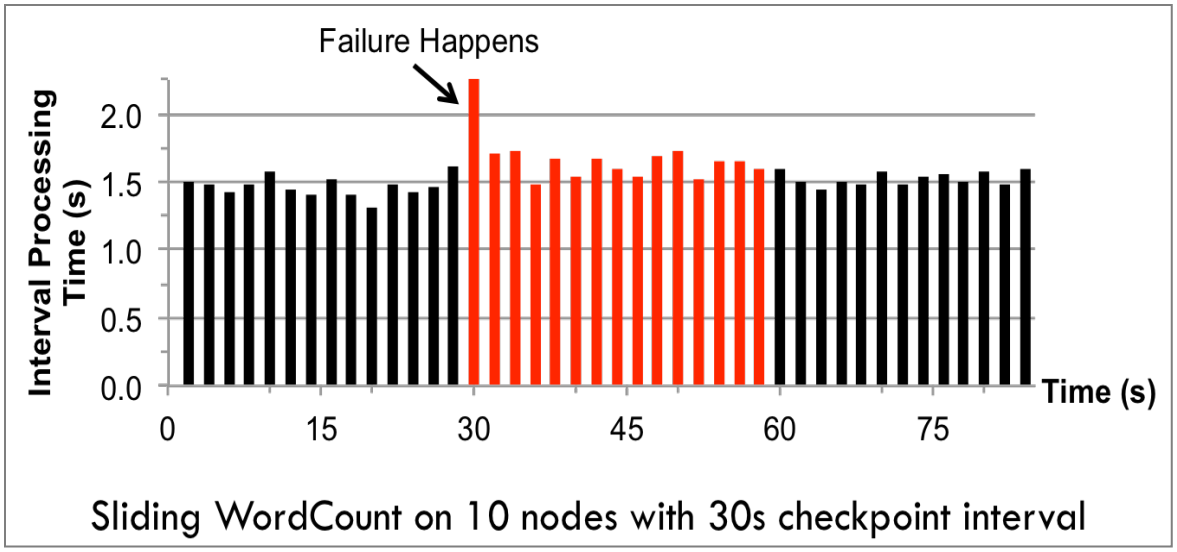
Higher throughput than Storm

- Spark Streaming: **670k** records/second/node
- Storm: **115k** records/second/node
- Apache S4: 7.5k records/second/node



Fast Fault Recovery

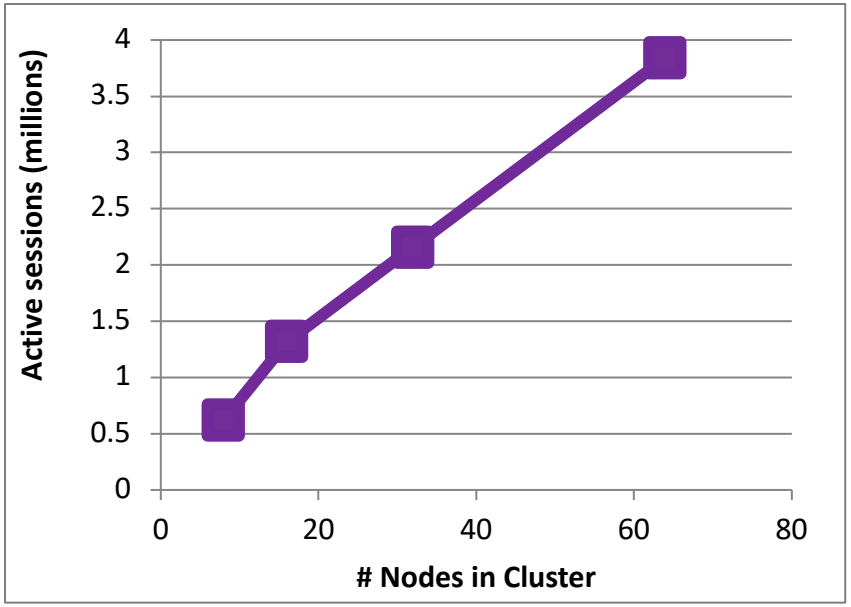
Recovers from faults/stragglers within **1 sec**



Real Applications: Conviva

Real-time monitoring of video metadata

- Achieved 1-2 second latency
- Millions of video sessions processed
- Scales linearly with cluster size



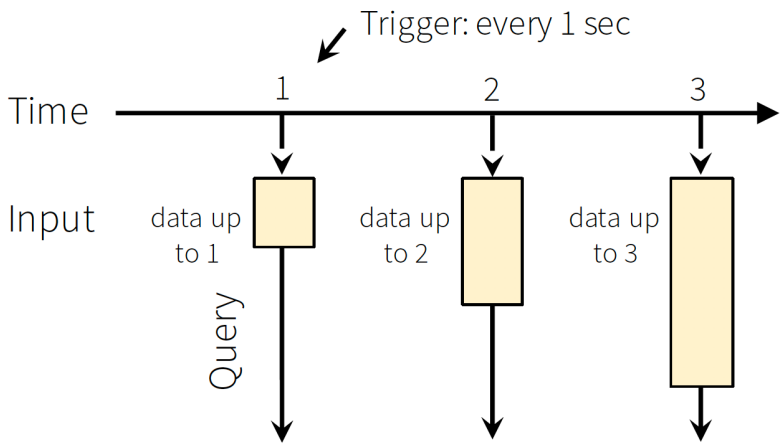
Structured streaming

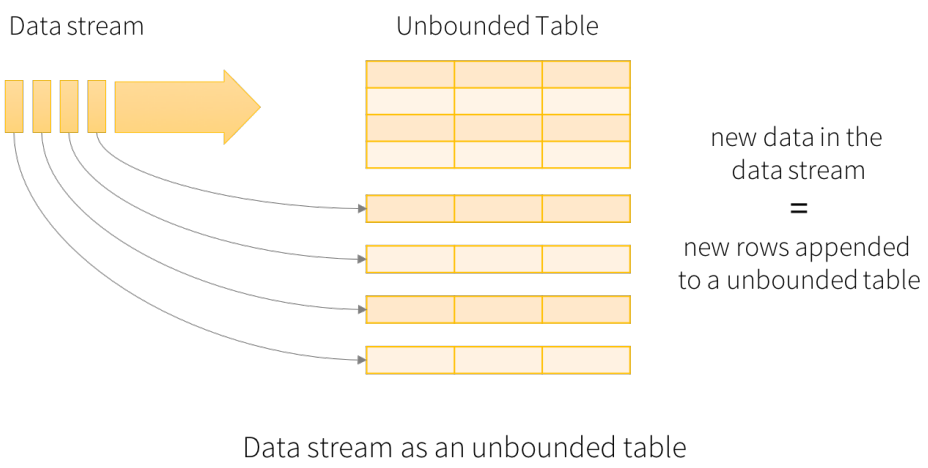
Pain points with DStreams

- Processing with event-time, dealing with late data
  - DStream API exposes batch time, hard to incorporate event-time
- Interoperate streaming with batch AND interactive
  - RDD/DStream has similar API, but still requires translation
- Reasoning about end-to-end guarantees
  - Requires carefully constructing sinks that handle failures correctly
  - Data consistency in the storage while being updated

New model

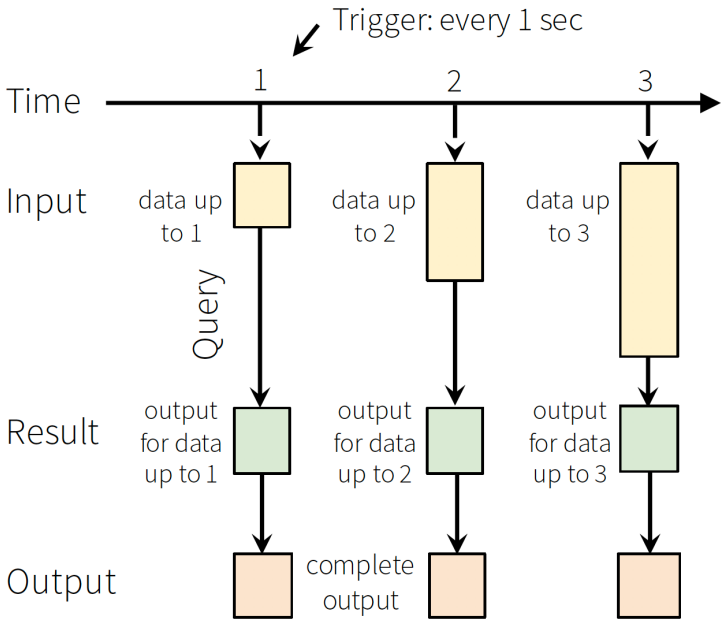
- Input: data from source as an append-only table
- Trigger: how frequently to check input for new data
- Query: operations on input usual map/filter/reduce new window, session ops





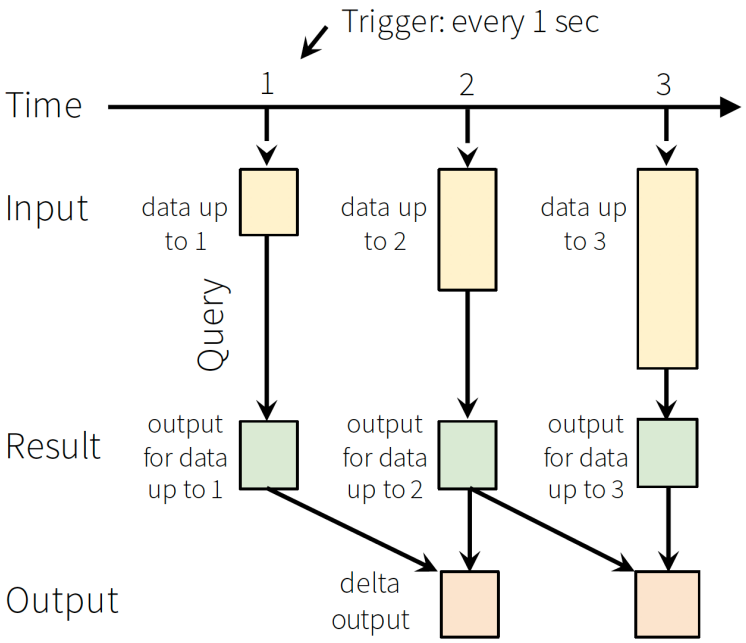
New model (2)

- Result: final operated table updated every trigger interval
- Output: what part of result to write to data sink after every trigger
- Complete output: Write full result table every time

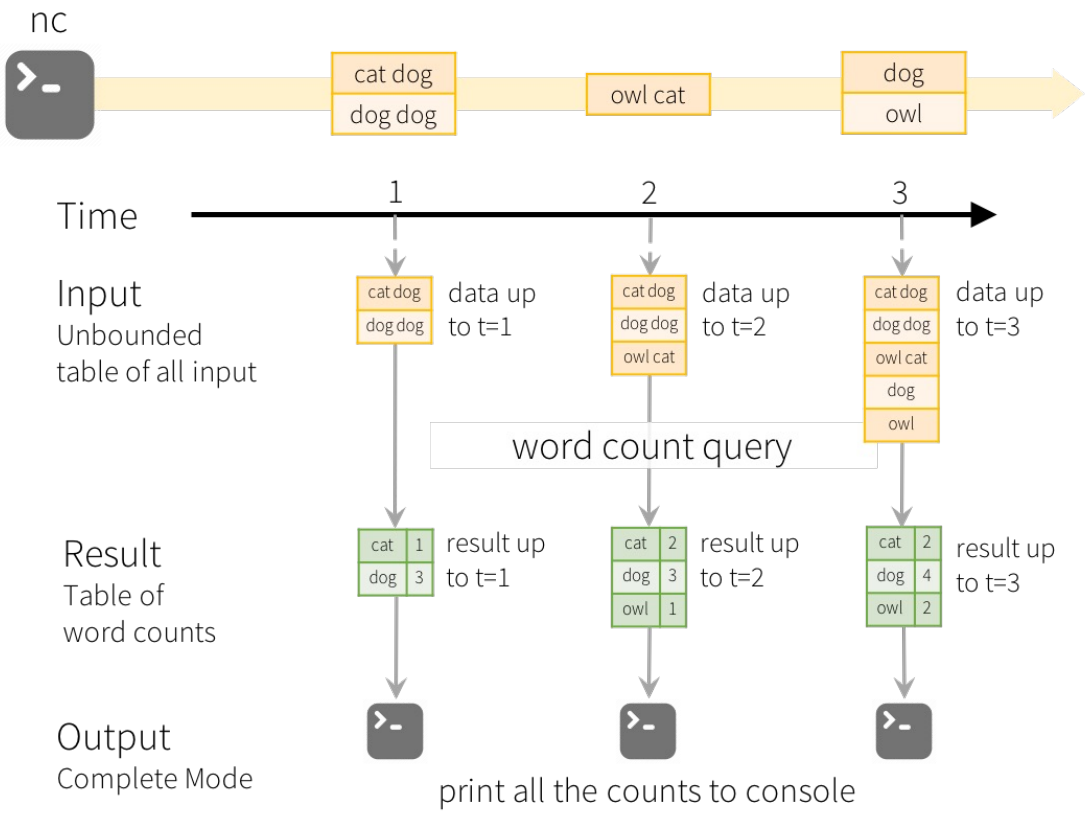


New model (3)

- Delta output: Write only the rows that changed in result from previous batch
- Append output: Write only new rows
- *Not all output modes are feasible with all queries

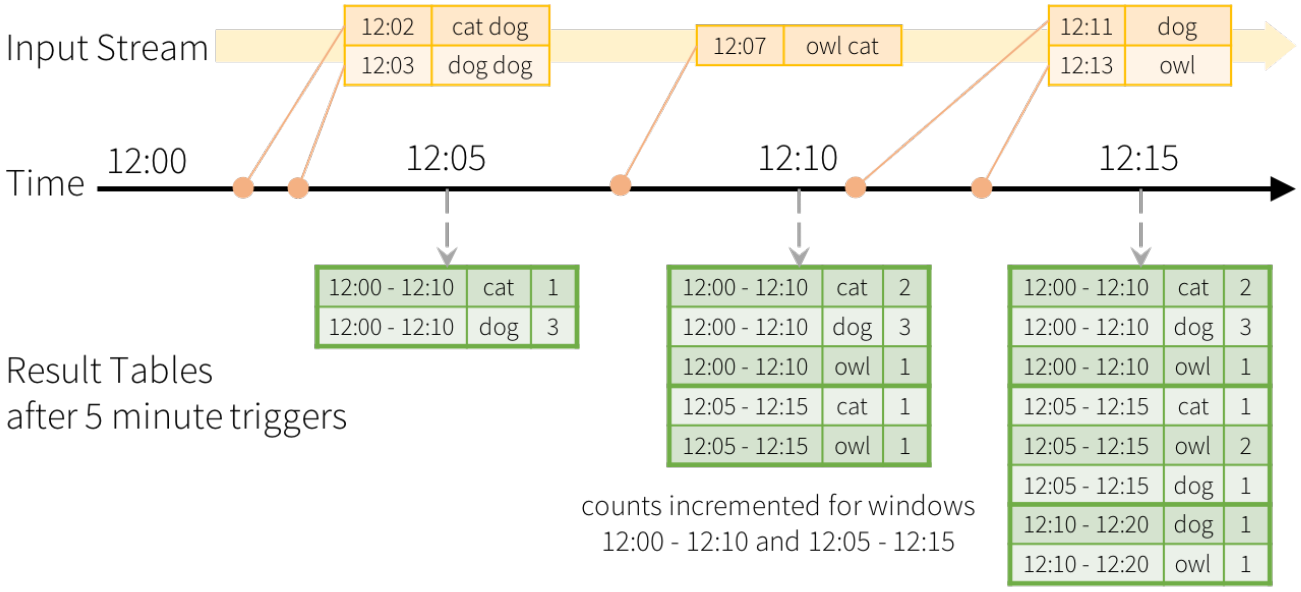


Example



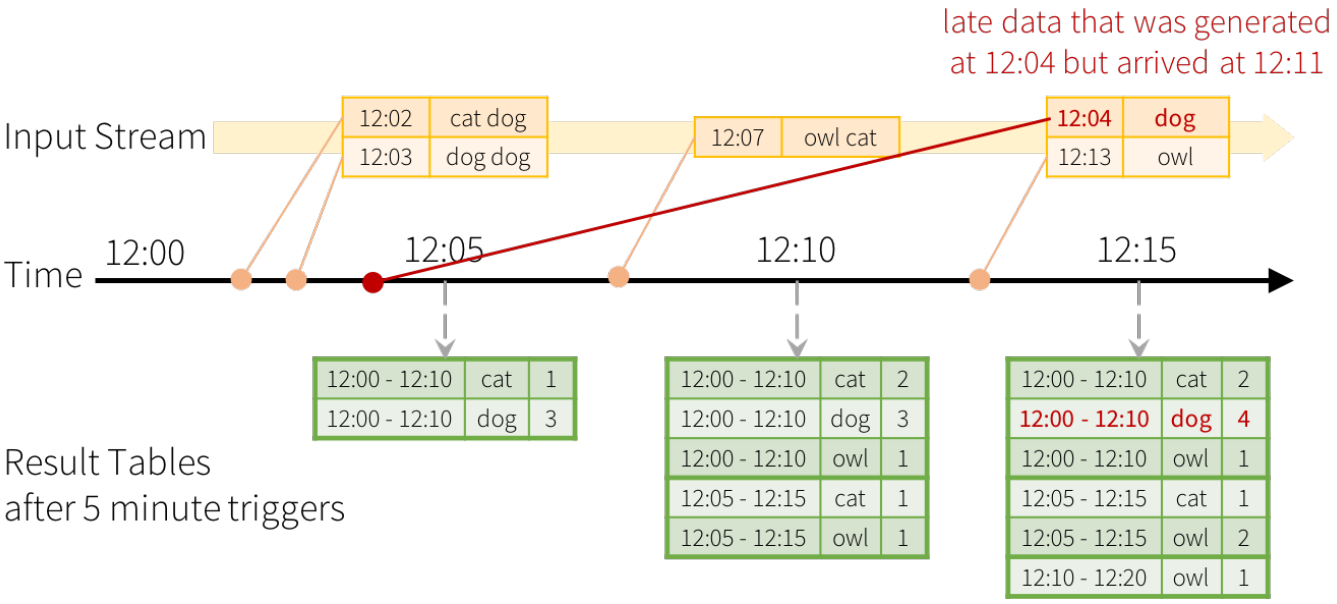
Model of the Quick Example

Window Operations on Event Time



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

Handling Late Data and Watermarking



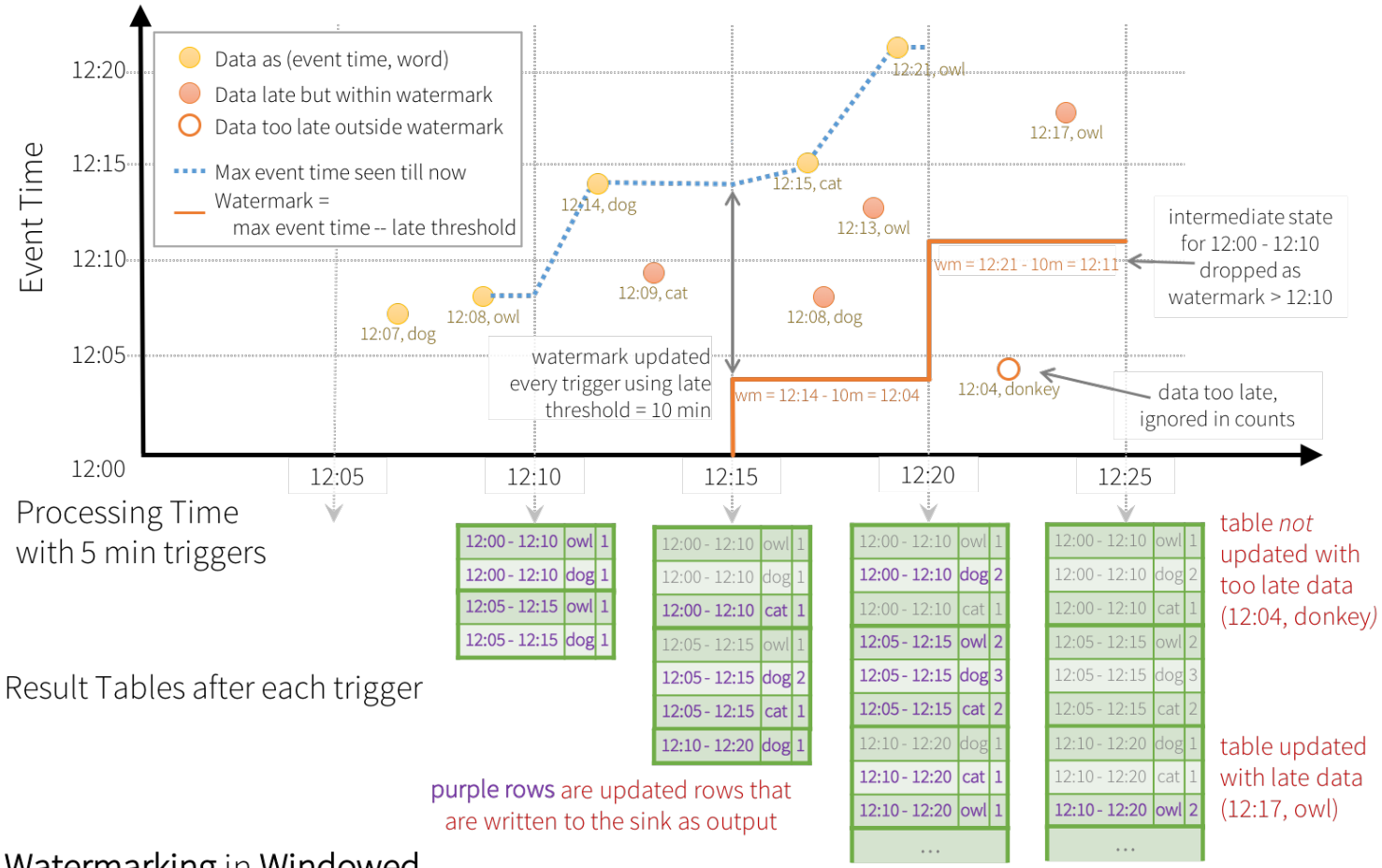
Late data handling in
Windowed Grouped Aggregation

Watermarking

- Event Time Concept
  - Event time is the timestamp associated with when an event occurred in the real world.
  - Example: Event time represents when a record was generated.
- Watermark
  - The threshold of how late is the data allowed to be.
  - Helps manage out-of-order data by determining when it's safe to finalize processing for a specific event time.
- Late Data Handling
  - Use withWatermark in Spark Structured Streaming to specify the maximum lateness allowed.

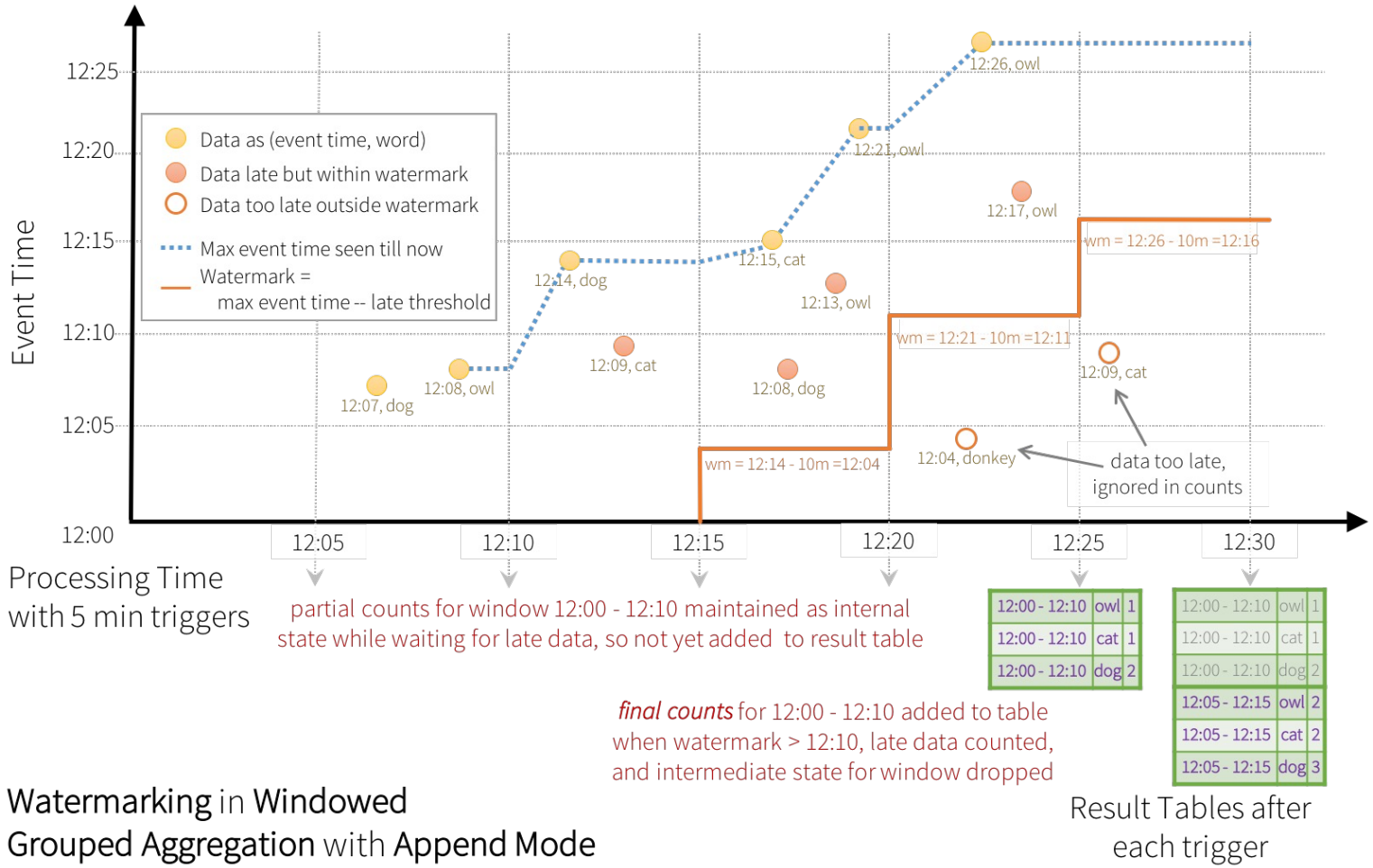
```
words = ...  # streaming DataFrame of schema { timestamp: Timestamp, word: String }

# Group the data by window and word and compute the count of each group
windowedCounts = words \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(
        window(words.timestamp, "10 minutes", "5 minutes"),
        words.word) \
    .count()
```

Group aggregation with update mode



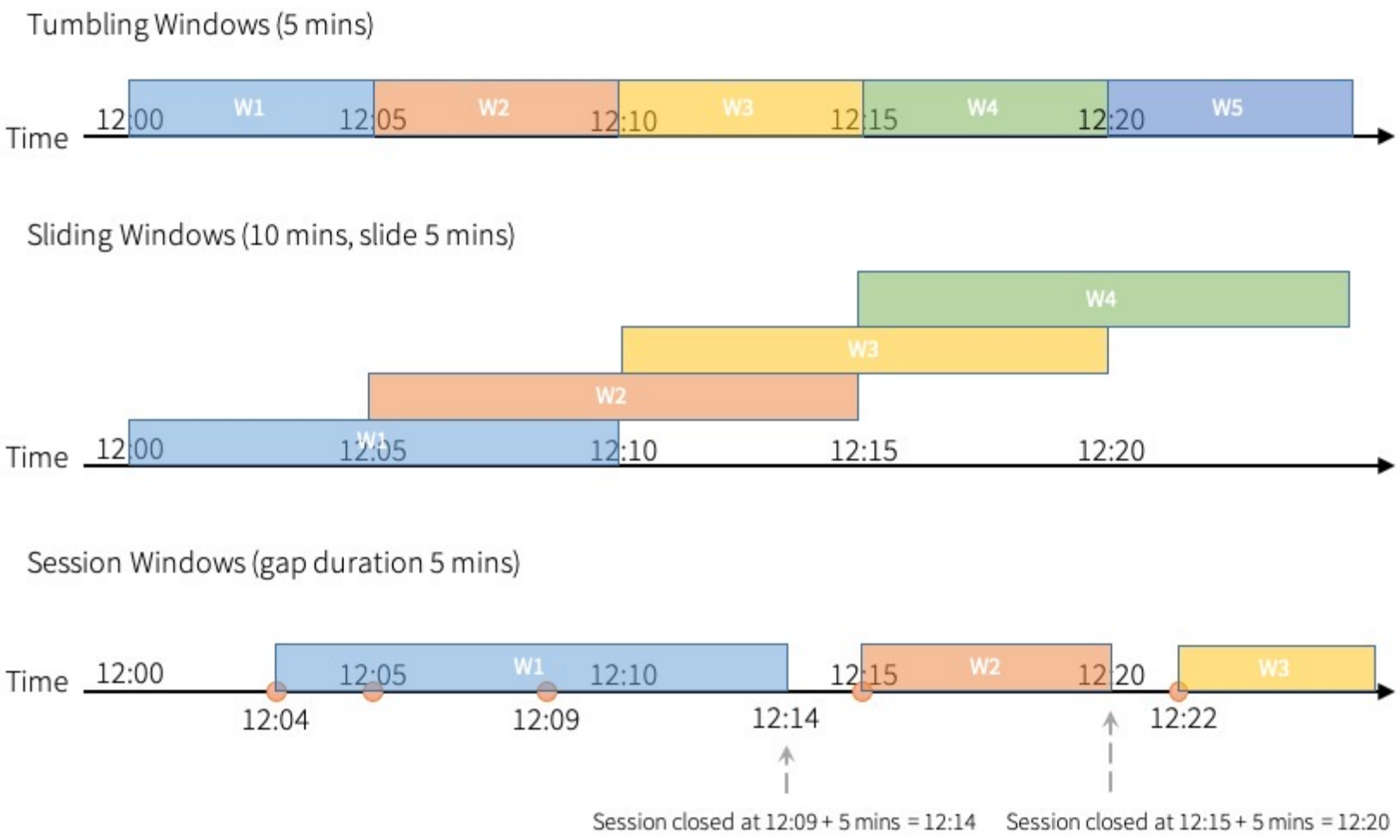
Watermarking in **Windowed Grouped Aggregation** with **Update Mode**

Group aggregation with append mode



Watermarking in **Windowed Grouped Aggregation** with **Append Mode**

Types of time windows



Batch ETL with DataFrames

```
input = spark.read
        .format("json")
        .load("source-path")
result = input
        .select("device", "signal")
        .where("signal > 15")
result.write
        .format("parquet")
        .save("dest-path")
```

- Read from Json file
- Select some devices
- Write to parquet file

Streaming ETL with DataFrames

```
input = spark.read
            .format("json")
            .stream("source-path")
result = input
            .select("device", "signal")
            .where("signal > 15")
result.write
            .format("parquet")
            .startStream("dest-path")
```

- Read from Json file stream
  - Replace load() with stream()
- Select some devices
  - Code does not change
- Write to Parquet file stream
  - Replace save() with startStream()

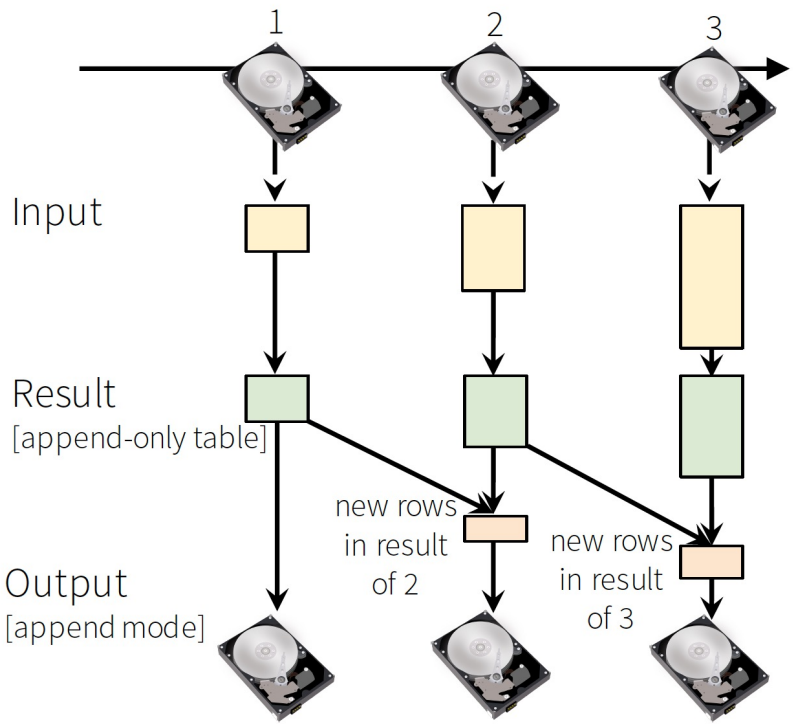
Streaming ETL with DataFrames

```
input = spark.read
        .format("json")
        .stream("source-path")
result = input
        .select("device", "signal")
        .where("signal > 15")
result.write
        .format("parquet")
        .startStream("dest-
        path")
```

- read…stream() creates a streaming DataFrame, does not start any of the computation
- write…startStream() defines where & how to output the data and starts the processing

Streaming ETL with DataFrames

```
input = spark.read
            .format("json")
            .stream("source-path")
result = input
            .select("device", "signal")
            .where("signal > 15")
result.write
            .format("parquet")
            .startStream("dest-
            path")
```



Continuous Aggregations

```
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

Continuous Windowed Aggregations

```
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

- Simplifies event-time stream processing (not possible in DStreams) Works on both, streaming and batch jobs

Joining streams with static data

```
kafkaDataset = spark.read
        .kafka("iot-updates")
        .stream()

staticDataset = ctxt.read
        .jdbc("jdbc://", "iot-
device-info")

joinedDataset =
        kafkaDataset.join(
        staticDataset,
"device-type")
```

- Join streaming data from Kafka with static data via JDBC to enrich the streaming data …
- … without having to think that you are joining streaming data

Output modes

Defines what is outputted every time there is a trigger
Different output modes make sense for different queries

- Append mode with non-aggregation queries

```
input.select("device", "signal")
        .write
        .outputMode("append")
        .format("parquet")
        .startStream("dest-path")
```

- Complete mode with aggregation queries

```
input.agg(count("*"))
        .write
        .outputMode("complete"
)
        .format("parquet")
        .startStream("dest-path")
```

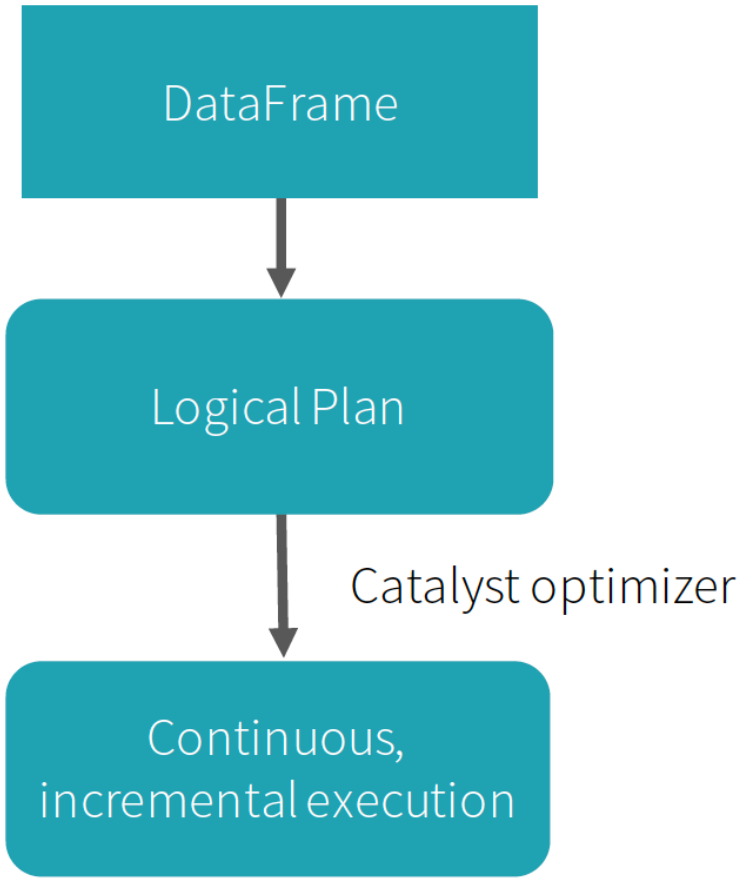
Query Management

```
query = result.write
        .format("parquet")
        .outputMode("append")
        .startStream("dest-path")

query.stop()
query.awaitTermination()
query.exception()

query.sourceStatuses()
query.sinkStatus()
```

- query: a handle to the running streaming computation for managing it
  - Stop it, wait for it to terminate
  - Get status
  - Get error, if terminated
- Multiple queries can be active at the same time
- Each query has unique name for keeping track

Query execution

- Logically
  - Dataset operations on table (i.e. as easy to understand as batch)
- Physically
  - Spark automatically runs the query in streaming fashion (i.e. incrementally and continuously)

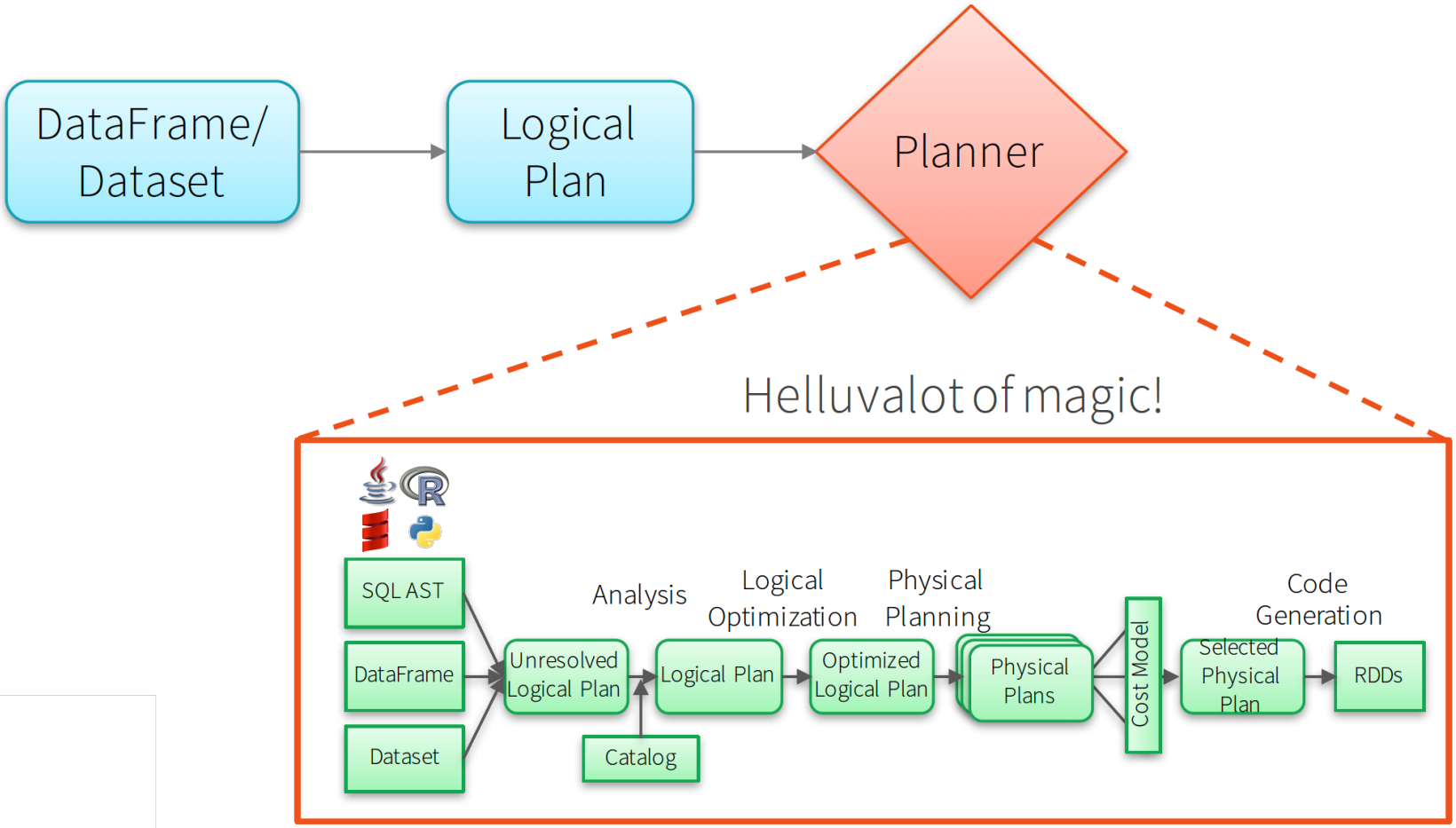


Structured Streaming

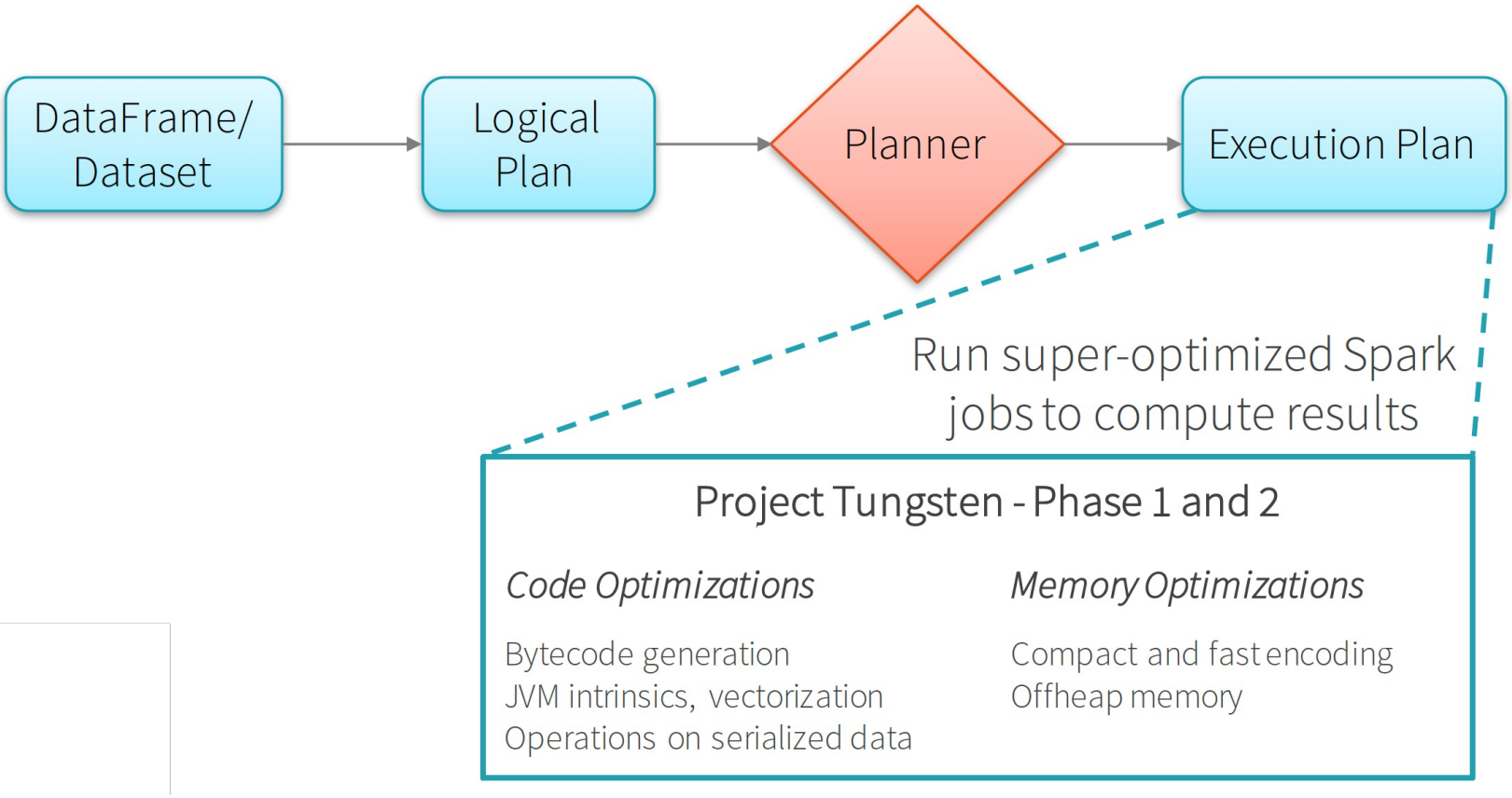
- High-level streaming API built on Datasets/DataFrames
  - Event time, windowing, sessions, sources & sinks
  - End-to-end exactly once semantics
- Unifies streaming, interactive and batch queries
  - Aggregate data in a stream, then serve using JDBC
  - Add, remove, change queries at runtime
  - Build and apply ML models
- https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Internal execution

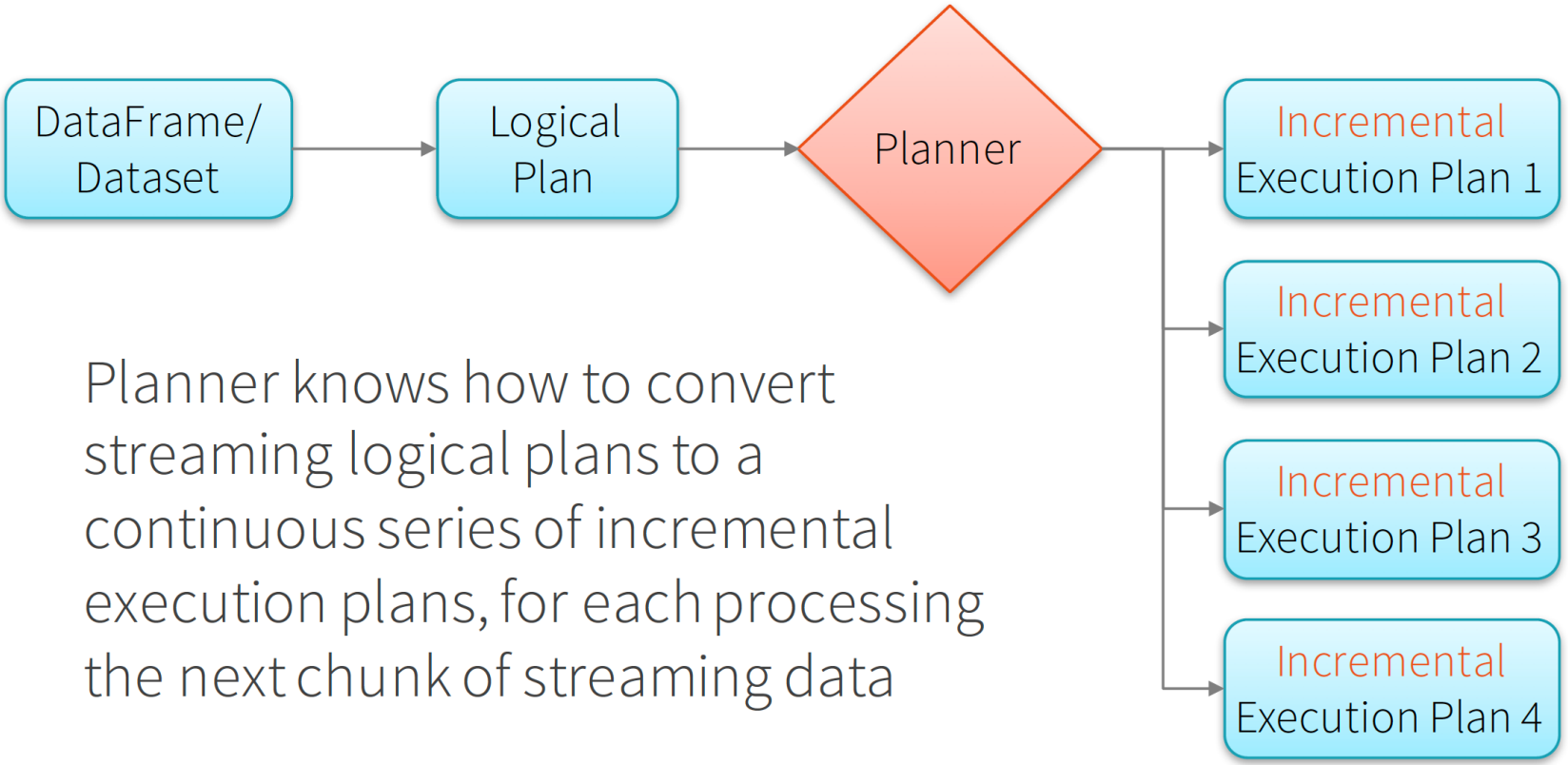
Batch Execution on Spark SQL



Batch Execution on Spark SQL

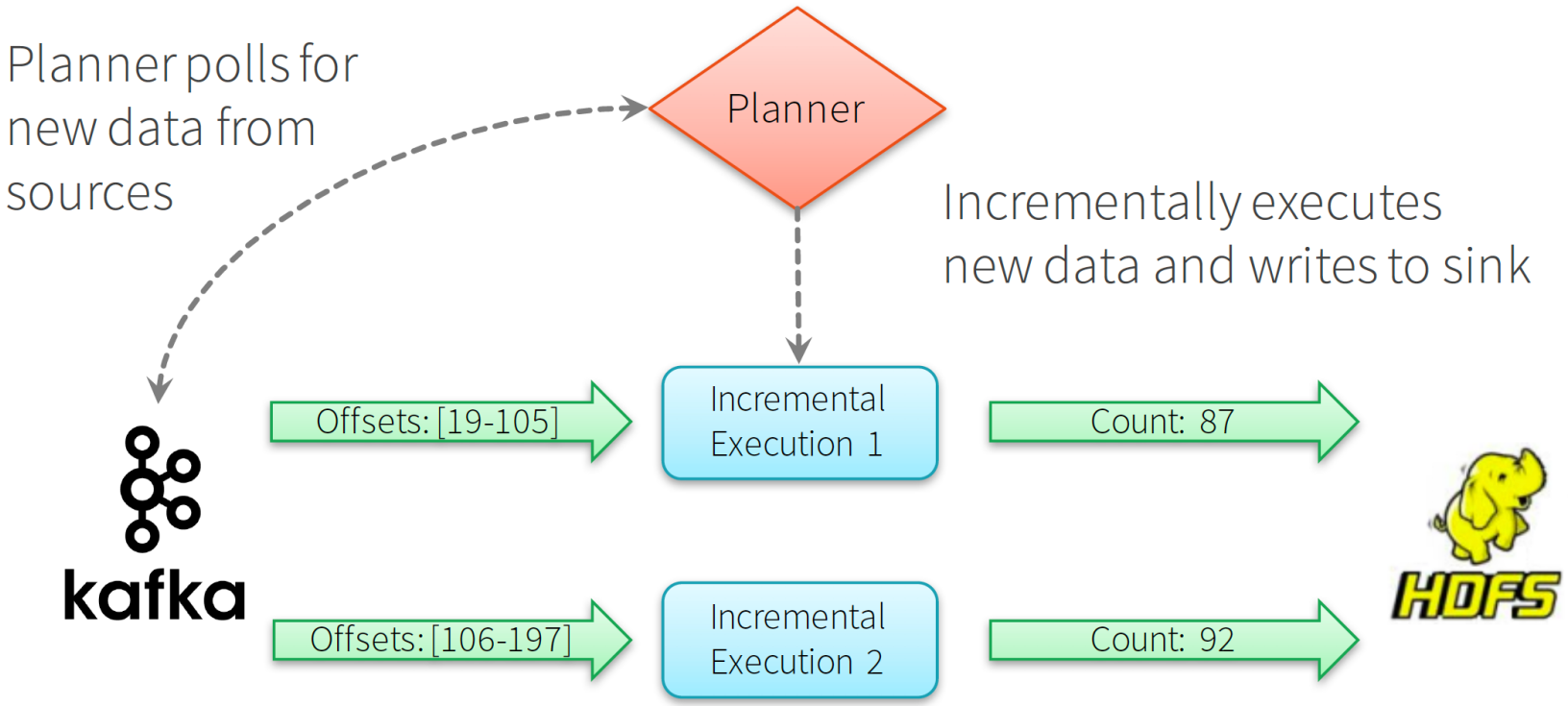


Continuous Incremental Execution



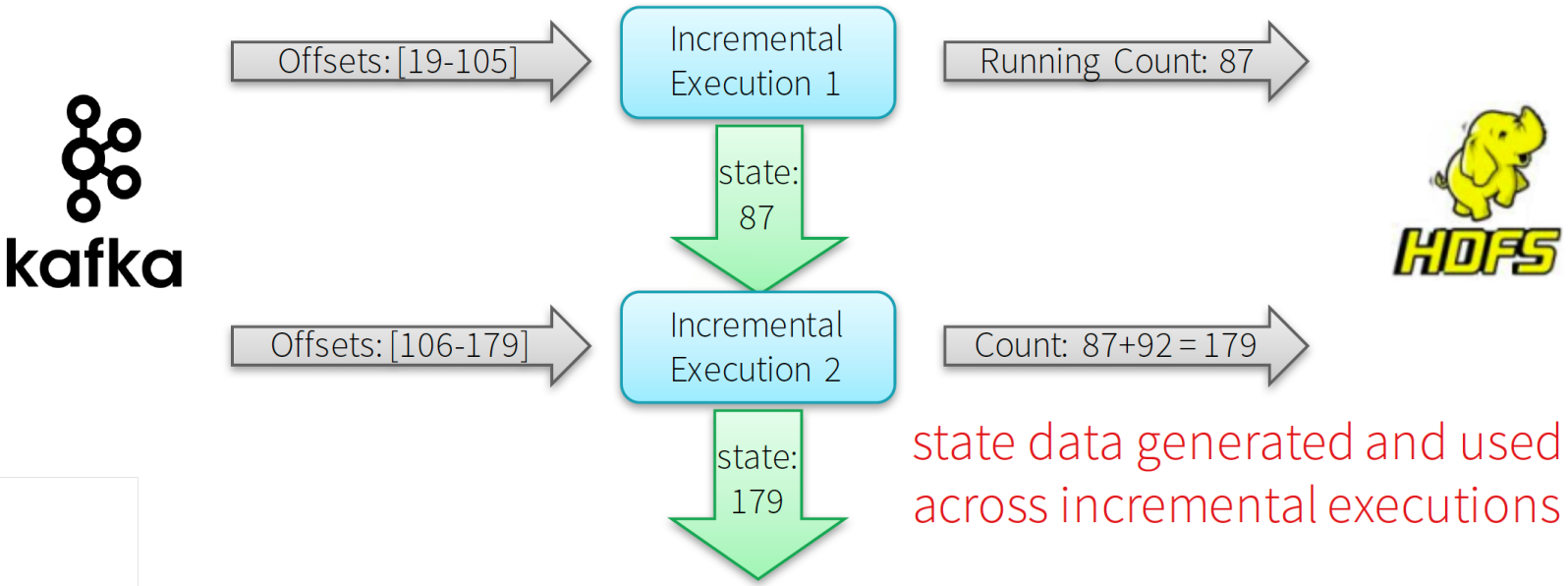
Planner knows how to convert streaming logical plans to a continuous series of incremental execution plans, for each processing the next chunk of streaming data

Continuous Incremental Execution



Continuous Aggregations

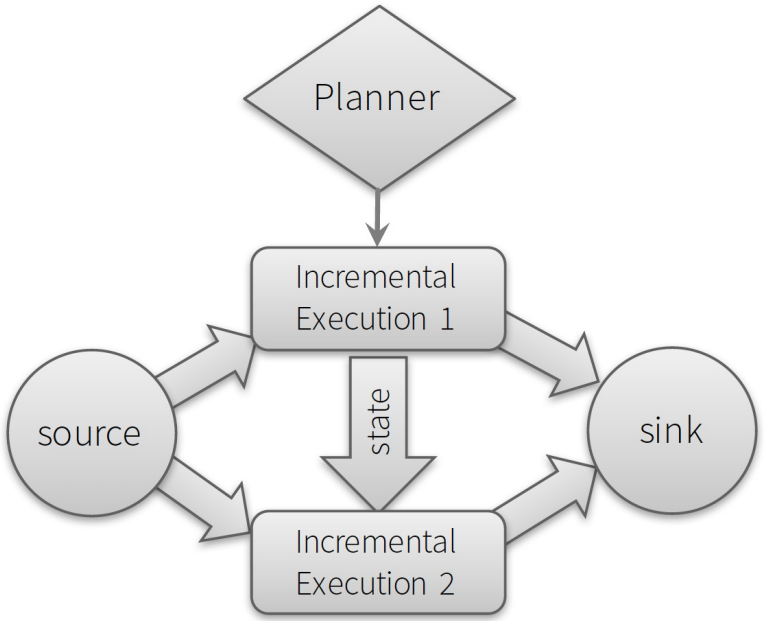
Maintain running aggregate as in-memory state backed by WAL in file system for fault-tolerance



state data generated and used across incremental executions

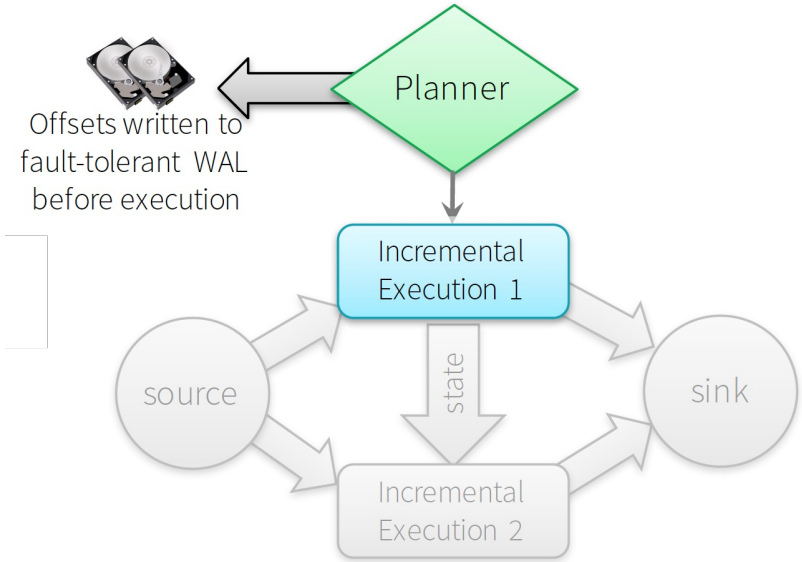
Fault-tolerance

- All data and metadata in the system needs to be recoverable / replayable



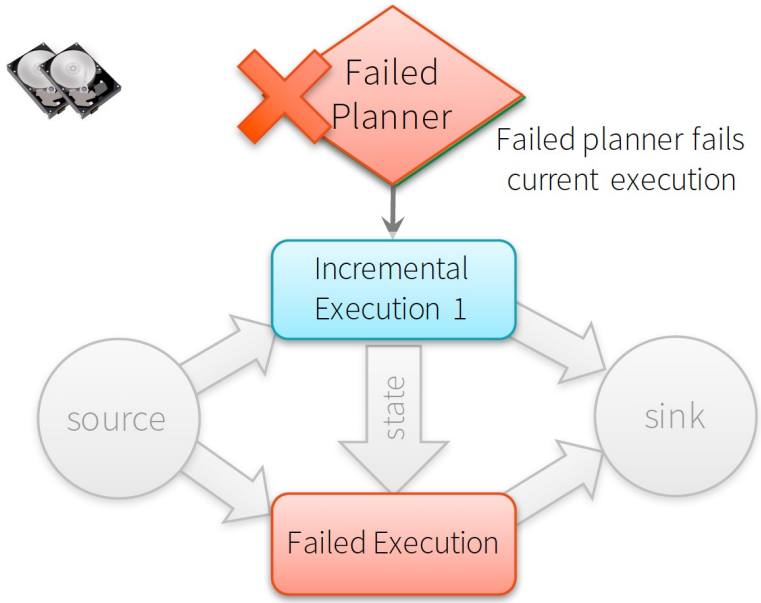
Fault-tolerant Planner

- Tracks offsets by writing the offset range of each execution to a write ahead log (WAL) in HDFS



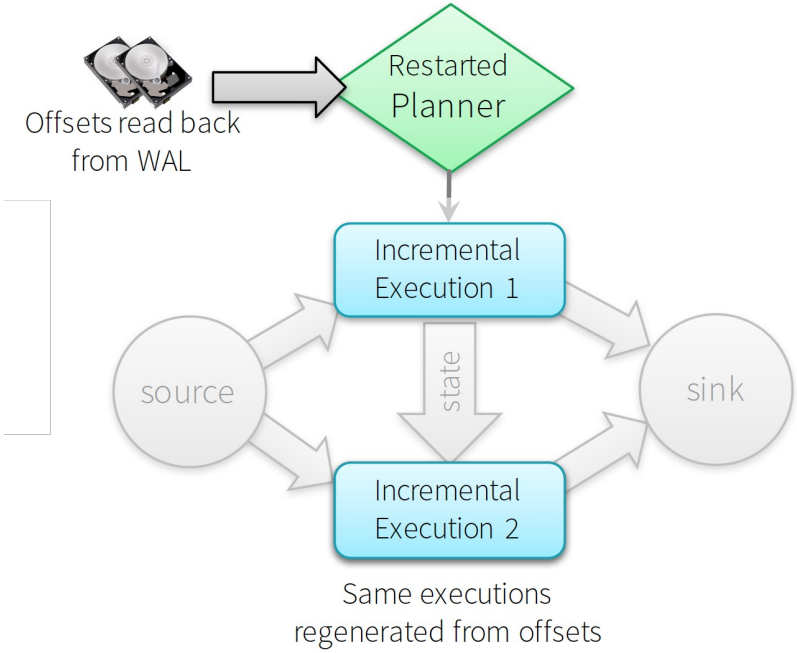
Fault-tolerant Planner

- Tracks offsets by writing the offset range of each execution to a write ahead log (WAL) in HDFS



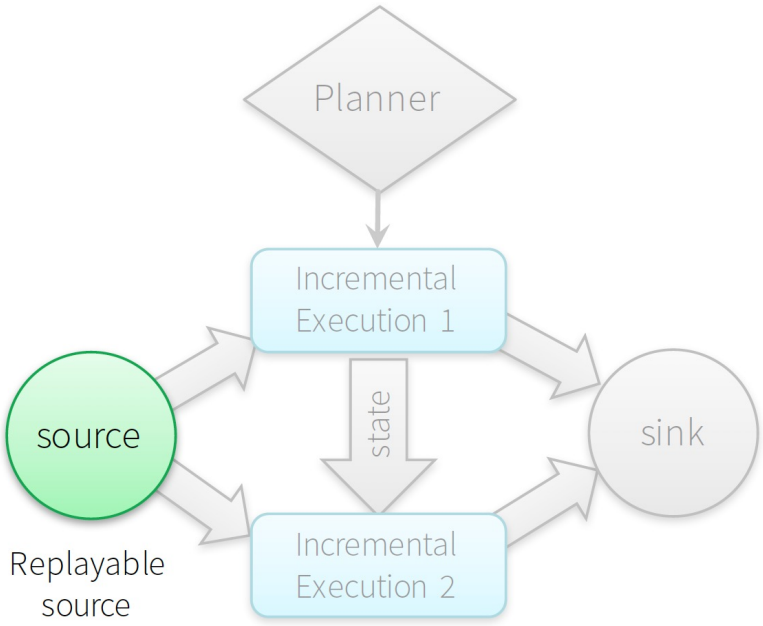
Fault-tolerant Planner

- Tracks offsets by writing the offset range of each execution to a write ahead log (WAL) in HDFS
- Reads log to recover from failures, and re-execute exact range of offsets



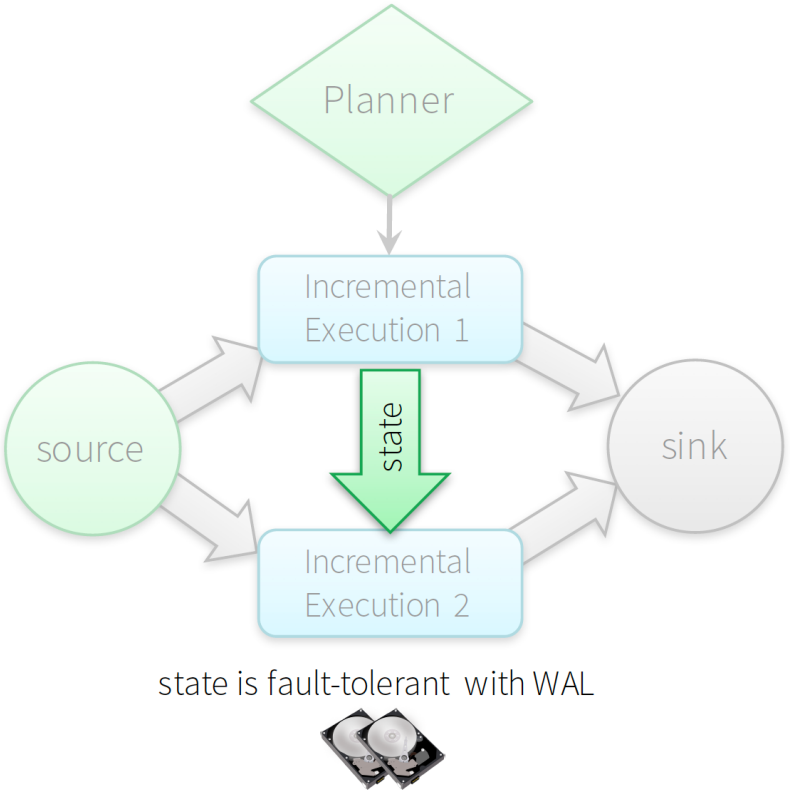
Fault-tolerant Sources

- Structured streaming sources are by design replayable (e.g. Kafka, Kinesis, files) and generate the exactly same data given offsets recovered by planner



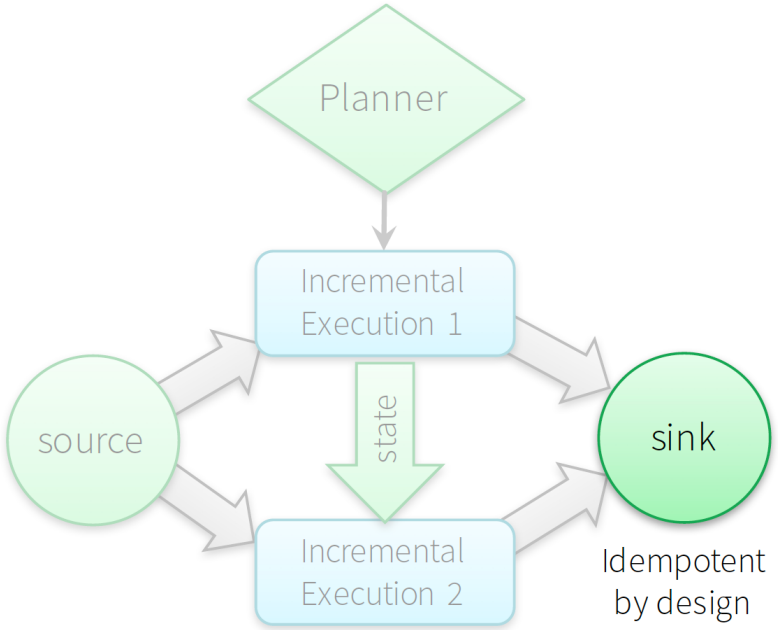
Fault-tolerant State

- Intermediate "state data" is a maintained in versioned, keyvalue maps in Spark workers, backed by HDFS
- Planner makes sure "correct version" of state used to reexecute after failure



Fault-tolerant Sink

- Sink are by design idempotent (deterministic), and handles re-executions to avoid double committing the output



Fault-tolerance

offset tracking in WAL

+

state management

+

fault-tolerant sources and sinks

=

end-to-end

exactly-once

guarantees

Structured streaming

Fast, fault-tolerant, exactly-once
stateful stream processing
without having to reason about streaming

Usecase

- https://mapr.com/blog/real-time-analysis-popular-uber-locations-spark-structured-streaming-machine-learning-kafka-and-mapr-db/

Usecase – Twitter sentiment analysis

Trending Topics can be used to create campaigns and attract larger audience.

Sentiment Analytics helps in crisis management, service adjusting and target marketing.

- Sentiment refers to the emotion behind a social media mention online.
- Sentiment Analysis is categorising the tweets related to particular topic and performing data mining using Sentiment Automation Analytics Tools.
- We will be performing Twitter Sentiment Analysis as our Use Case for Spark Streaming.

Figure: *Facebook And Twitter Trending Topics*

Problem statement

Problem Statement

To design a Twitter Sentiment Analysis System where we populate real time sentiments for crisis management, service adjusting and target marketing

Sentiment Analysis is used to:

- ❑ Predict the success of a movie
- ❑ Predict political campaign success
- ❑ Decide whether to invest in a certain company
- ❑ Targeted advertising
- ❑ Review products and services

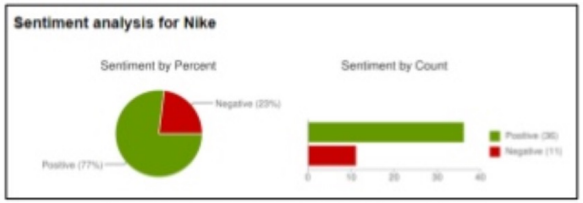


Figure: *Twitter Sentiment Analysis For Nike*

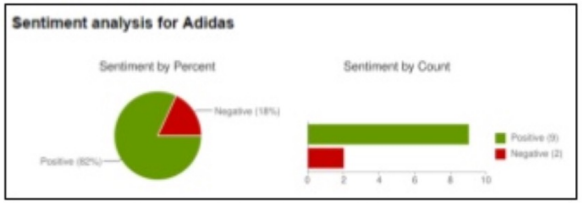


Figure: *Twitter Sentiment Analysis For Adidas*

Importing packages

```
//Import the necessary packages into the Spark Program
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.SparkContext._
import org.apache.spark.streaming.twitter._
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark._
import org.apache.spark.rdd._
import org.apache.spark.rdd.RDD
import org.apache.spark.SparkContext._
import org.apache.spark.sql
import org.apache.spark.storage.StorageLevel
import scala.io.Source
import scala.collection.mutable.HashMap
import java.io.File
```

Twitter token authorization

```
object mapr {

 def main(args: Array[String]) {
 if (args.length < 4) {
 System.err.println("Usage: TwitterPopularTags <consumer key>
<consumer secret> " +
 "<access token> <access token secret> [<filters>]")
 System.exit(1)
 }

 StreamingExamples.setStreamingLogLevels()
 //Passing our Twitter keys and tokens as arguments for authorization
 val Array(consumerKey, consumerSecret, accessToken,
accessTokenSecret) = args.take(4)
 val filters = args.takeRight(args.length - 4)
```

Dstream transformation

```
// Set the system properties so that Twitter4j library used by twitter stream
// Use them to generate OAuth credentials
System.setProperty("twitter4j.oauth.consumerKey", consumerKey)
System.setProperty("twitter4j.oauth.consumerSecret", consumerSecret)
System.setProperty("twitter4j.oauth.accessToken", accessToken)
System.setProperty("twitter4j.oauth.accessTokenSecret",
accessTokenSecret)

val sparkConf = new
SparkConf().setAppName("Sentiments").setMaster("local[2]")
val ssc = new StreamingContext(sparkConf, Seconds(5))
val stream = TwitterUtils.createStream(ssc, None, filters)

//Input DStream transformation using flatMap
val tags = stream.flatMap { status =>
status.getHashtagEntities.map(_.getText)}
```

Generating tweet data

```
//RDD transformation using sortBy and then map function
tags.countByValue()
 .foreachRDD { rdd =>
 val now = org.joda.time.DateTime.now()
 rdd
 .sortBy(_._2)
 .map(x => (x, now))
 //Saving our output at ~/twitter/ directory
 .saveAsTextFile(s"~/twitter/$now")
 }

//DStream transformation using filter and map functions
val tweets = stream.filter {t =>
 val tags = t.getText.split("
").filter(_.startsWith("#")).map(_.toLowerCase)
 tags.exists { x => true }
}
```

Extracting sentiments

```
val data = tweets.map { status =>
val sentiment = SentimentAnalysisUtils.detectSentiment(status.getText)
val tagss = status.getHashtagEntities.map(_.getText.toLowerCase)
(status.getText, sentiment.toString, tagss.toString())
}

data.print()
//Saving our output at ~/ with filenames starting like twitterss
data.saveAsTextFiles("~/twitterss","20000")

ssc.start()
ssc.awaitTermination()
  }
}
```

Results

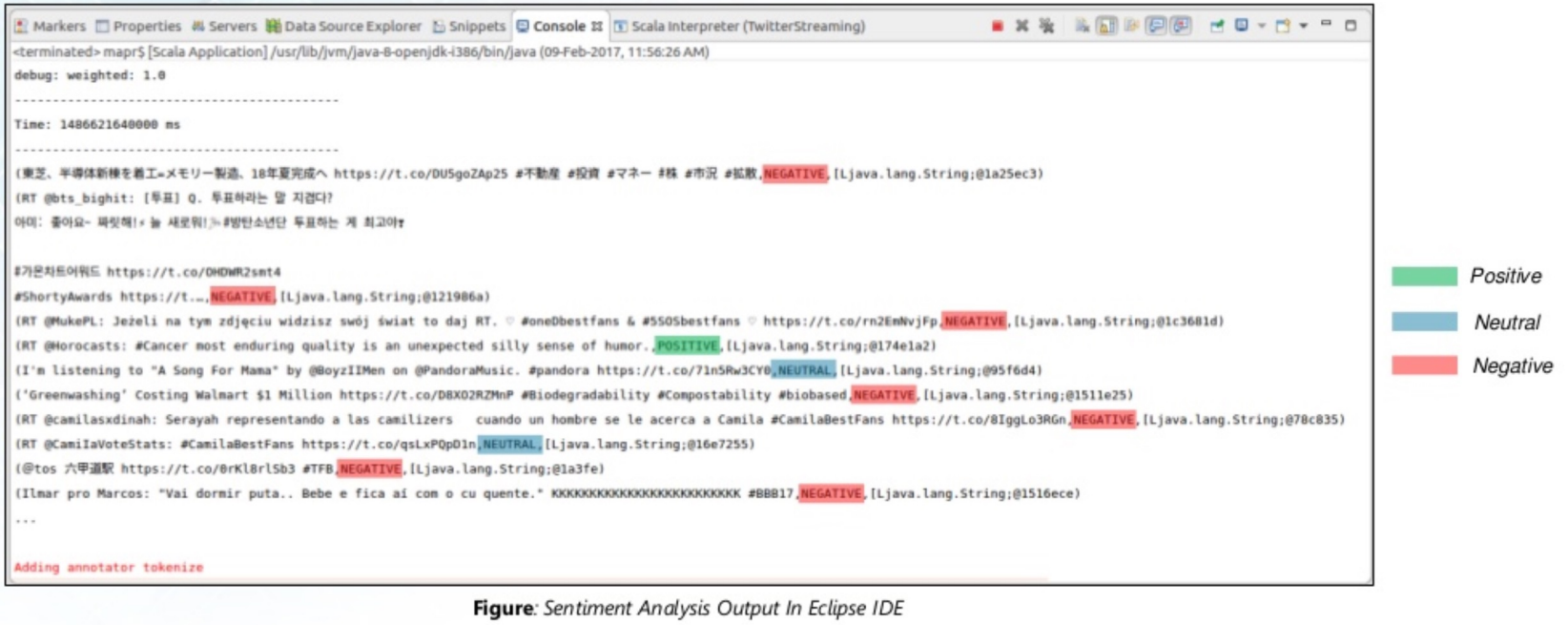


Figure: *Sentiment Analysis Output In Eclipse IDE*

Output directory

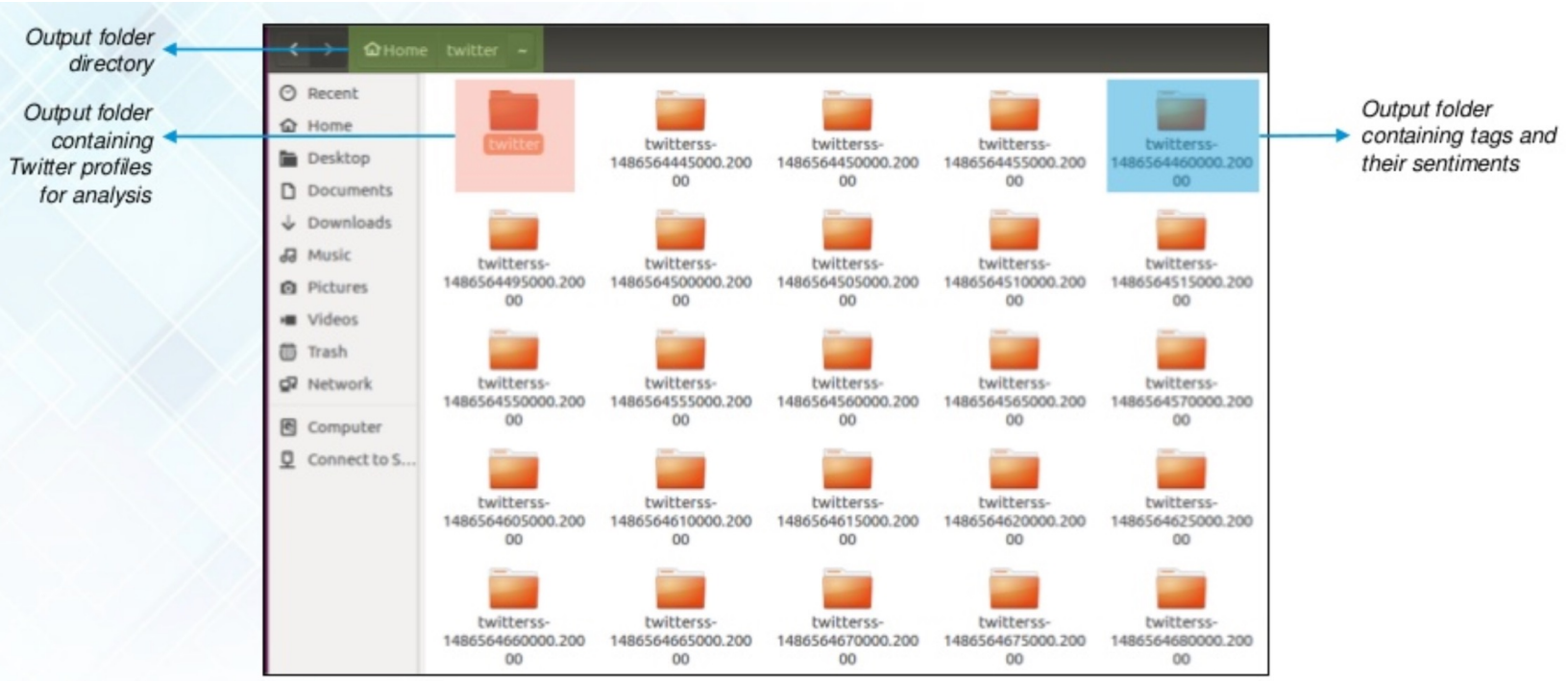


Figure: *Output folders inside our 'twitter' project folder*

Output usernames

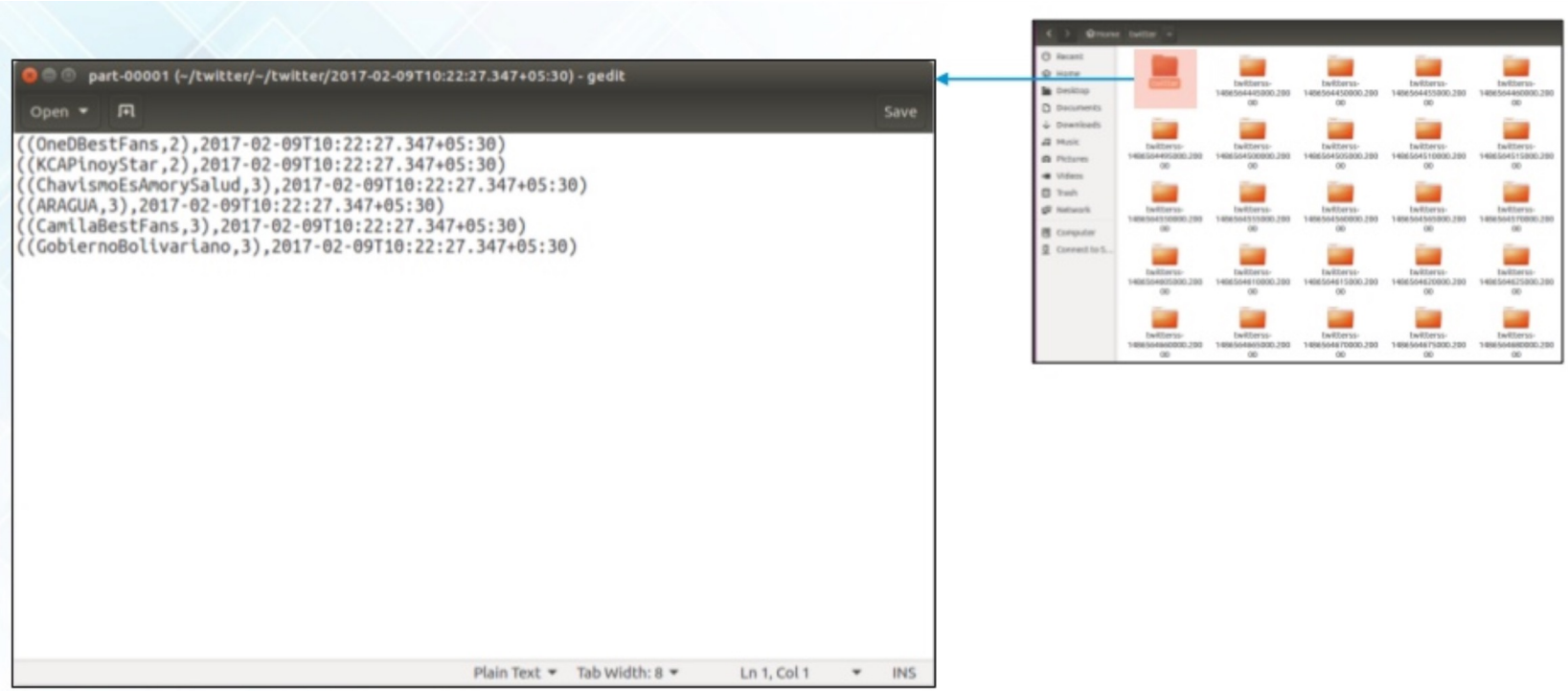


Figure: *Output file containing Twitter username and timestamp*

Output tweets and sentiments

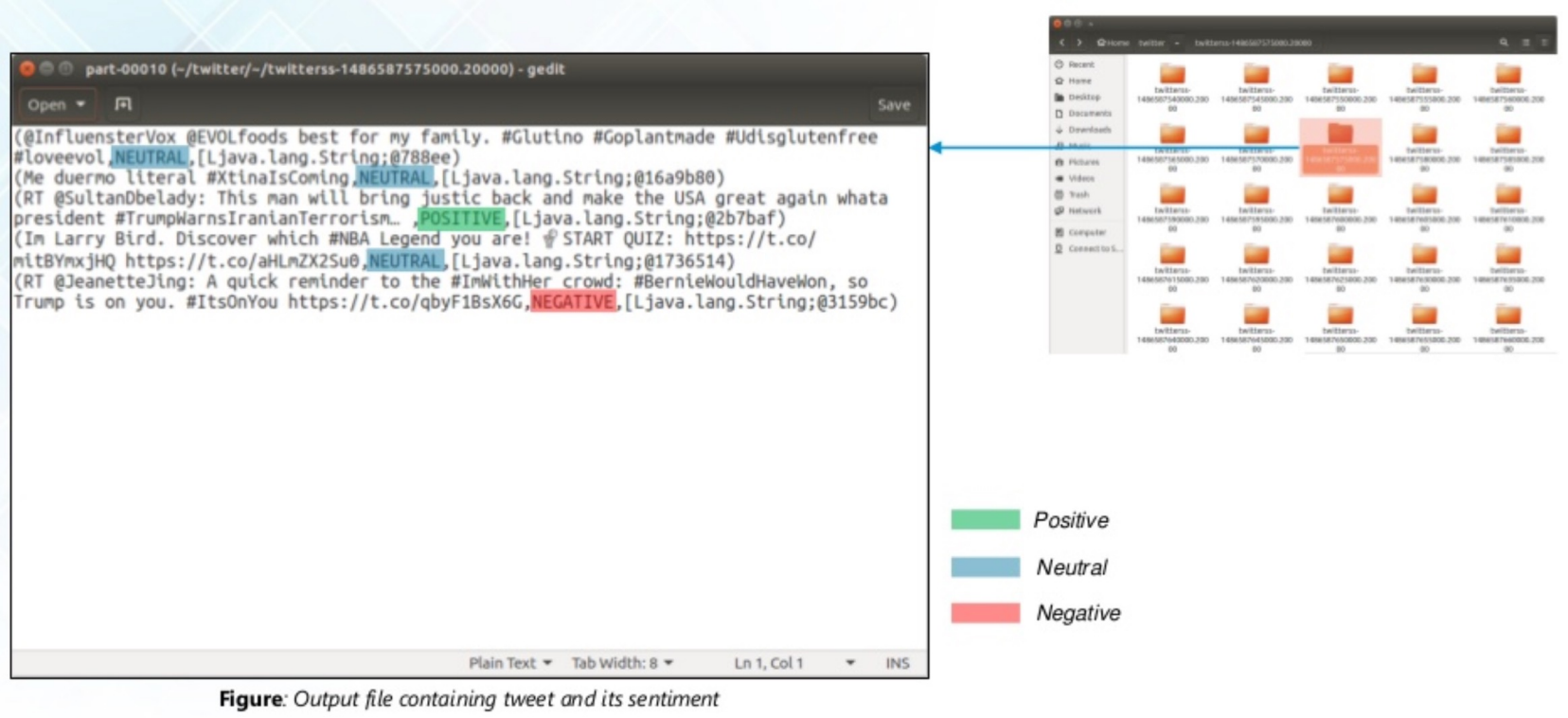


Figure: *Output file containing tweet and its sentiment*

Sentiments for Trump

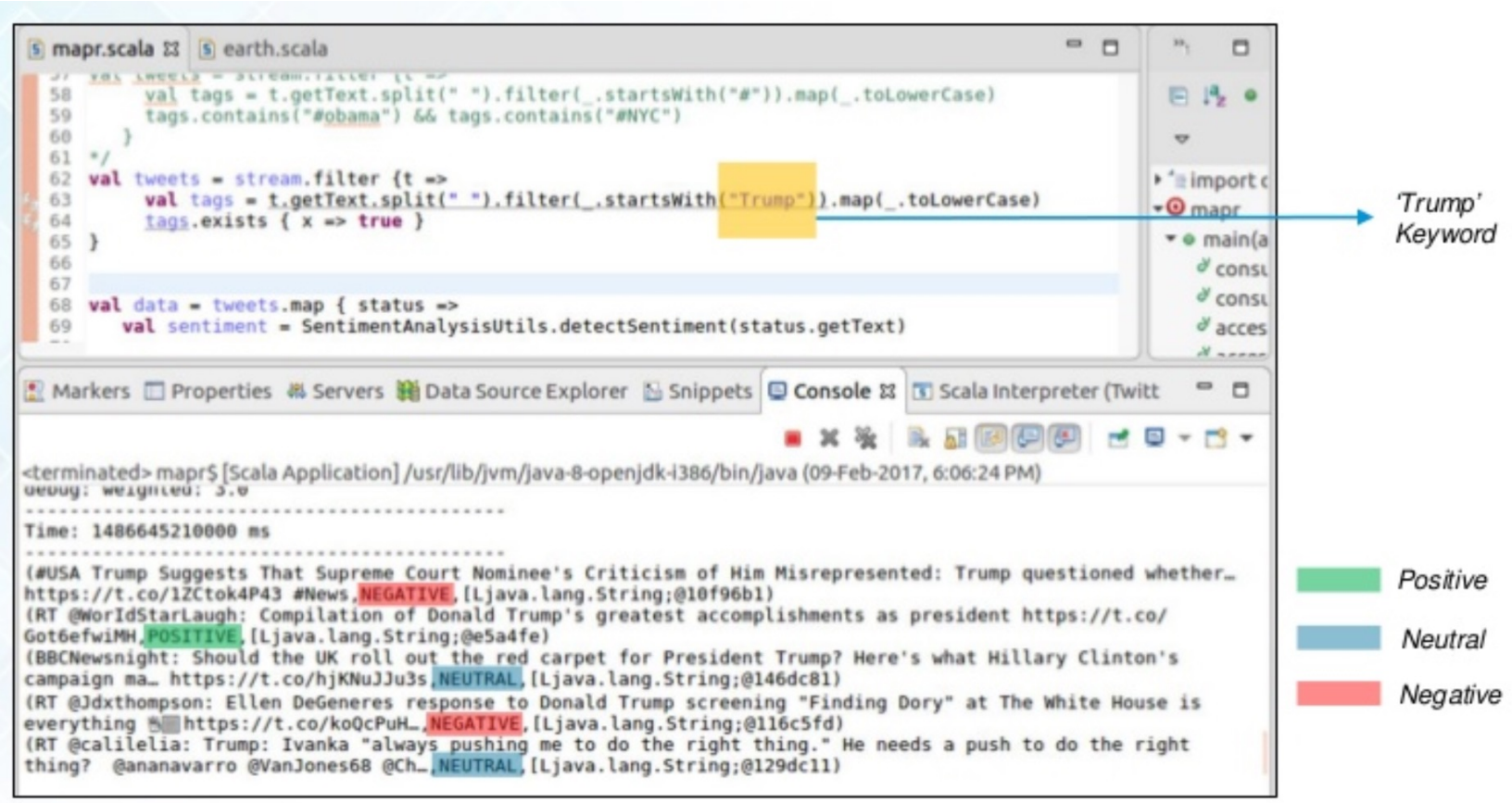


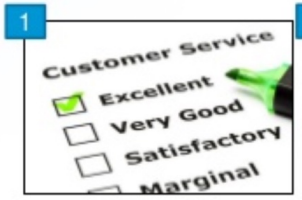
Figure: *Performing Sentiment Analysis on Tweets with 'Trump' Keyword*

Applying sentiment analysis

- As we have seen from our Sentiment Analysis demonstration, we can extract sentiments of particular topics just like we did for 'Trump'.
- Hence Sentiment Analytics can be used in crisis management, service adjusting and target marketing by companies around the world.

Companies using Spark Streaming for Sentiment Analysis have applied the same approach to achieve the following:

1. Enhancing the customer experience
2. Gaining competitive advantage
3. Gaining Business Intelligence
4. Revitalizing a losing brand



Thank you for your attention!
Q&A