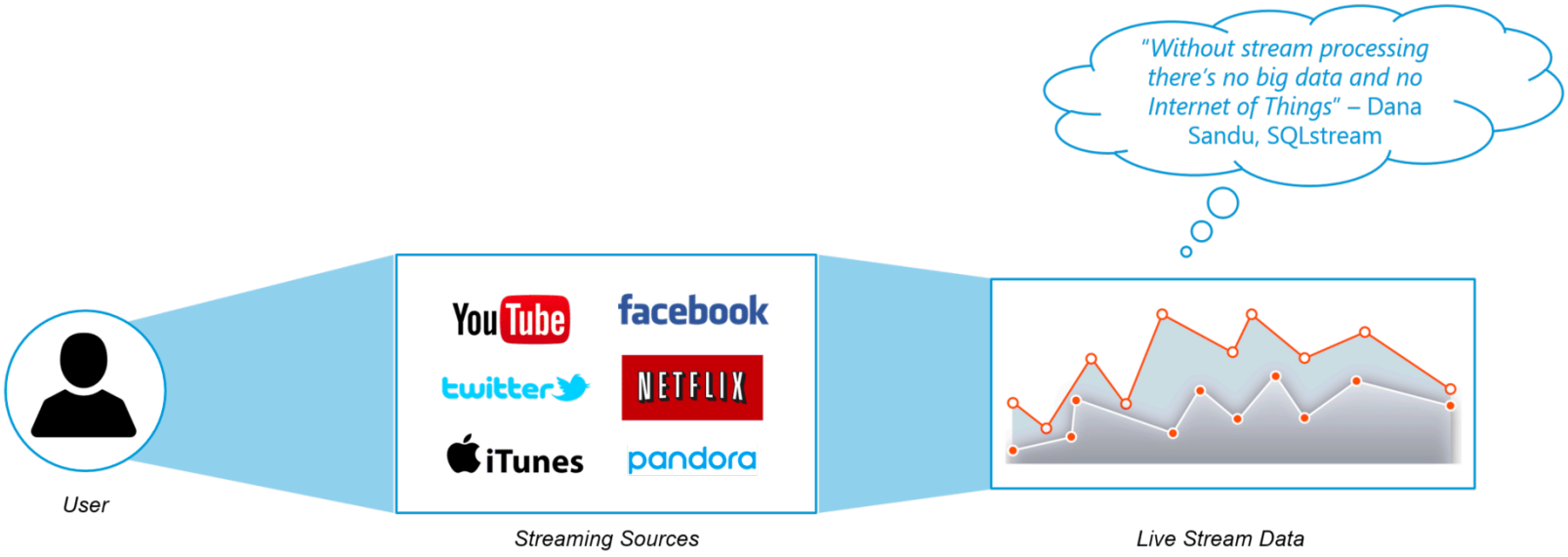
Spark streaming

Viet-Trung Tran

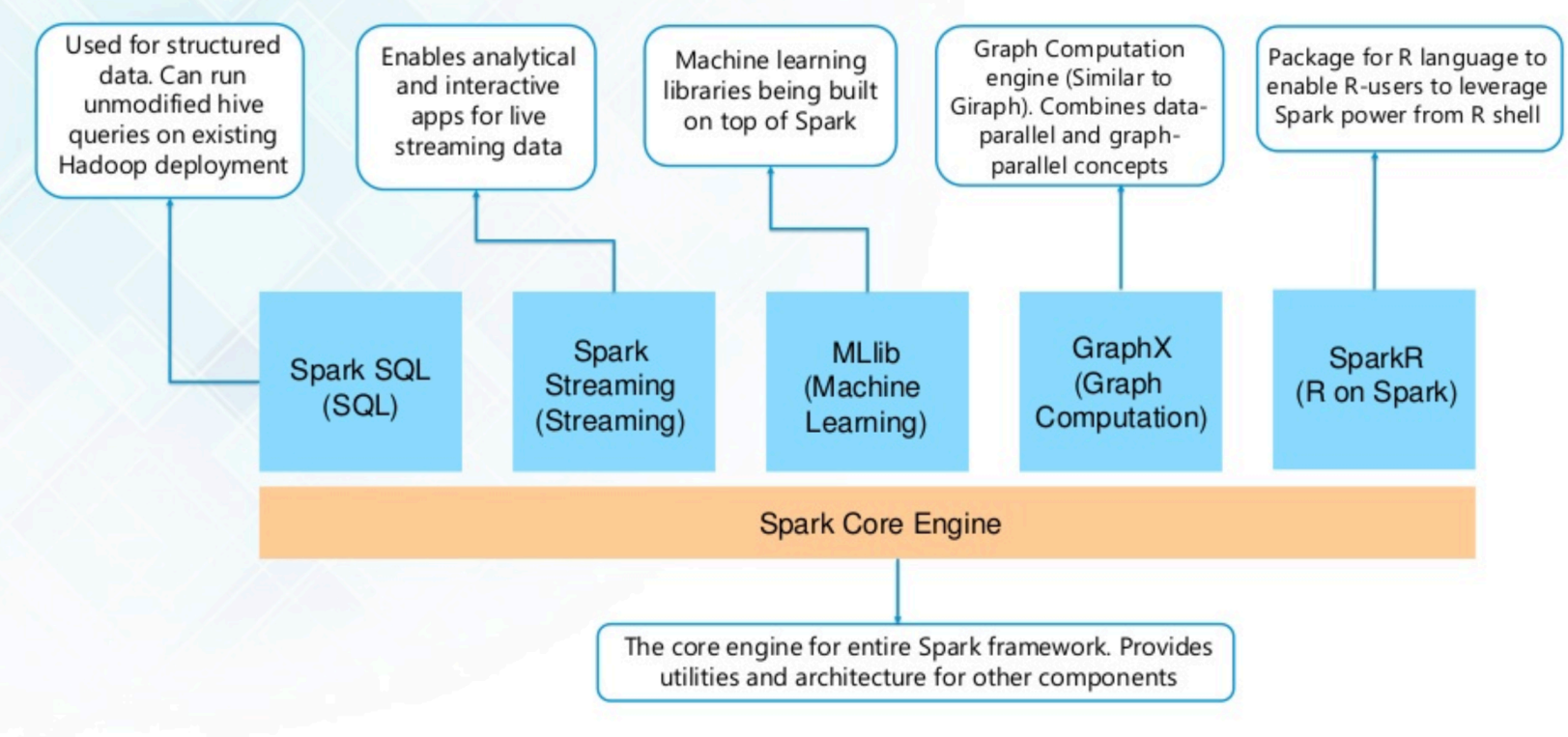
School of Information and Communication Technology

Data streaming

- Data Streaming là một kỹ thuật để thuyên chuyển dữ liệu dưới dạng luồng liên tục và bền vững
- Kỹ thuật streaming ngày càng trở nên quan trọng và phổ biến cùng với sự tăng trưởng của dữ liệu số

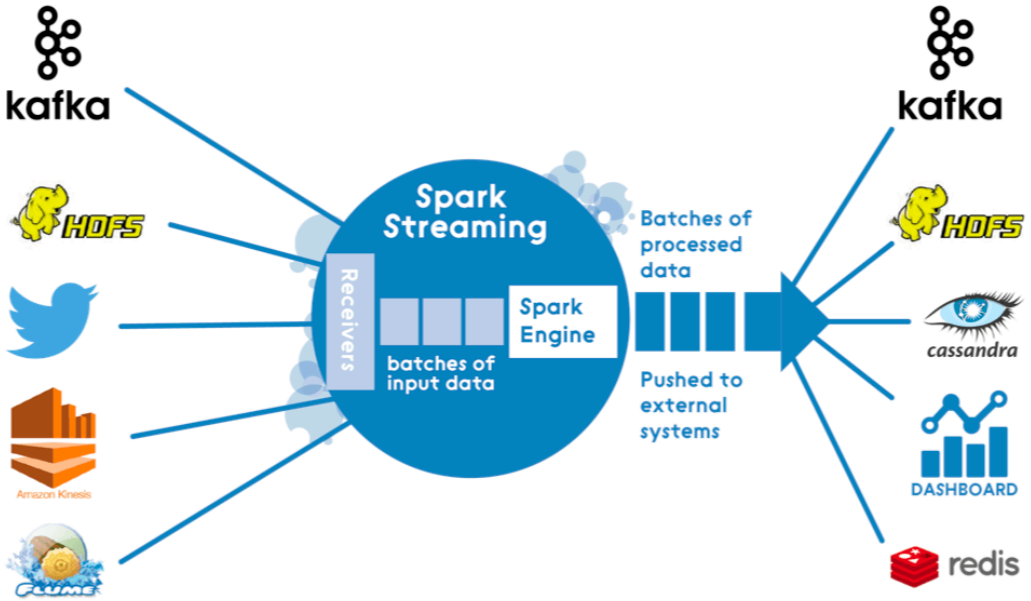


Hệ sinh thái Apache Spark



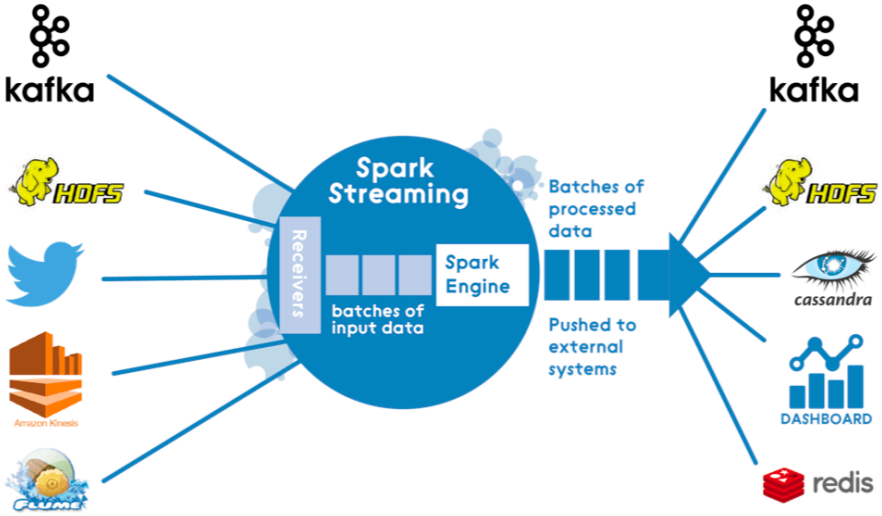
Why spark streaming

- Spark Streaming được sử dụng để thuyên chuyển dữ liệu thời gian thực (real-time data) từ nhiều nguồn khác nhau như Twitter, Facebook, IoT, và cho phép thực thi các phân tích dữ liệu mạnh mẽ từ các luồng dữ liệu này



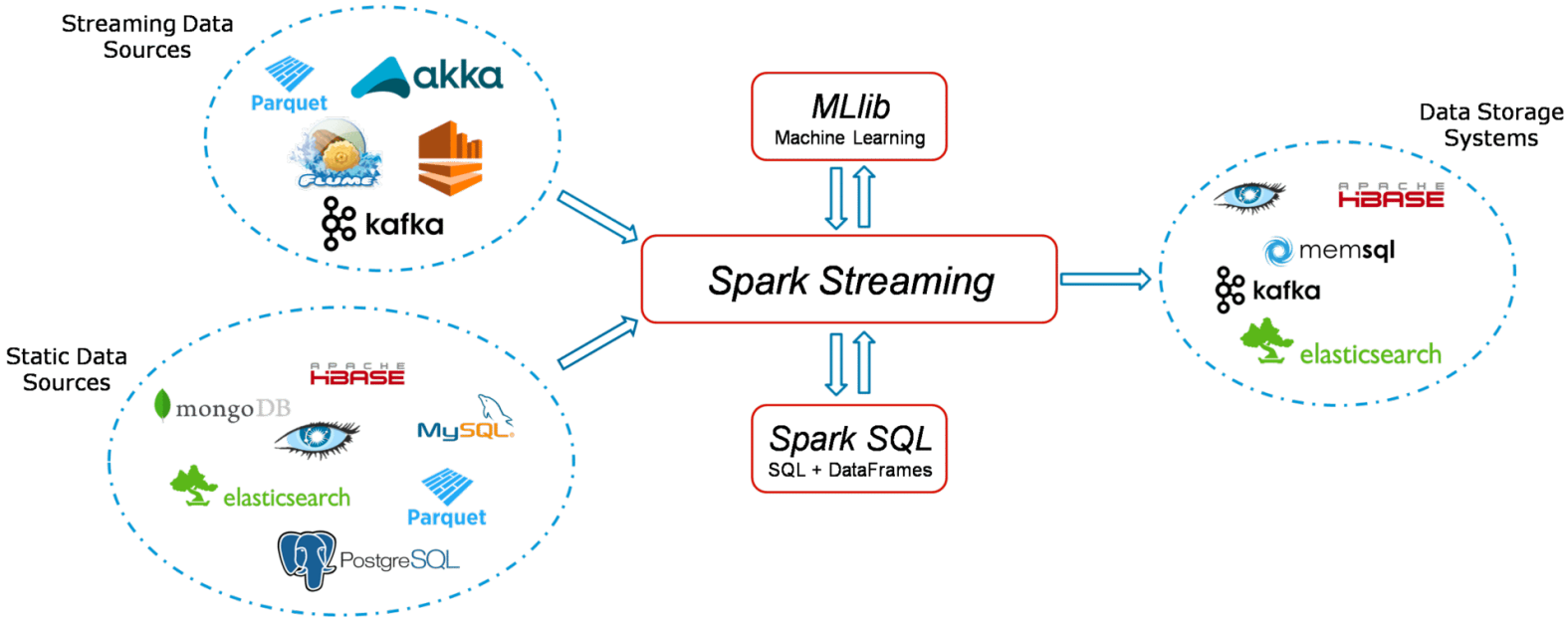
Tổng quan về Spark streaming

- Spark Streaming được sử dụng để xử lý luồng dữ liệu thời gian thực
- Là thành phần quan trọng trong hệ sinh thái Spark, bên cạnh Spark core API
- Spark Streaming cho phép xử lý luồng dữ liệu với thông lượng lớn (hight-throughput) và có khả năng chịu lỗi (fault-tolerant)
- Spark Streaming gọi luồng là Dstream, mỗi luồng là một chuỗi các RDD cần phải xử lý trực tuyến



Ưu điểm của Spark streaming

Luồng hoạt động của Spark streaming



Chi tiết luồng hoạt động

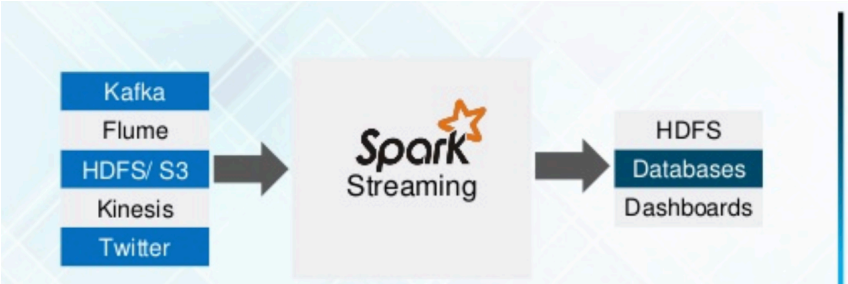


Figure: *Data from a variety of sources to various storage systems*

Figure: *Incoming streams of data divided into batches*

Figure: *Input data stream divided into discrete chunks of data*

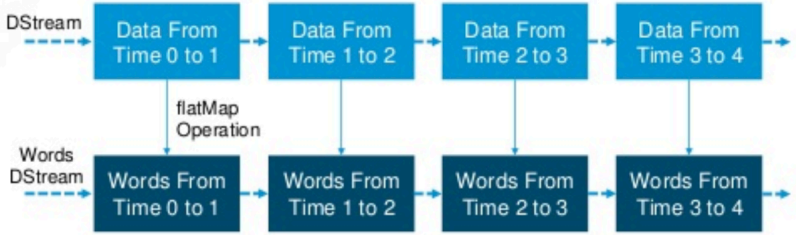
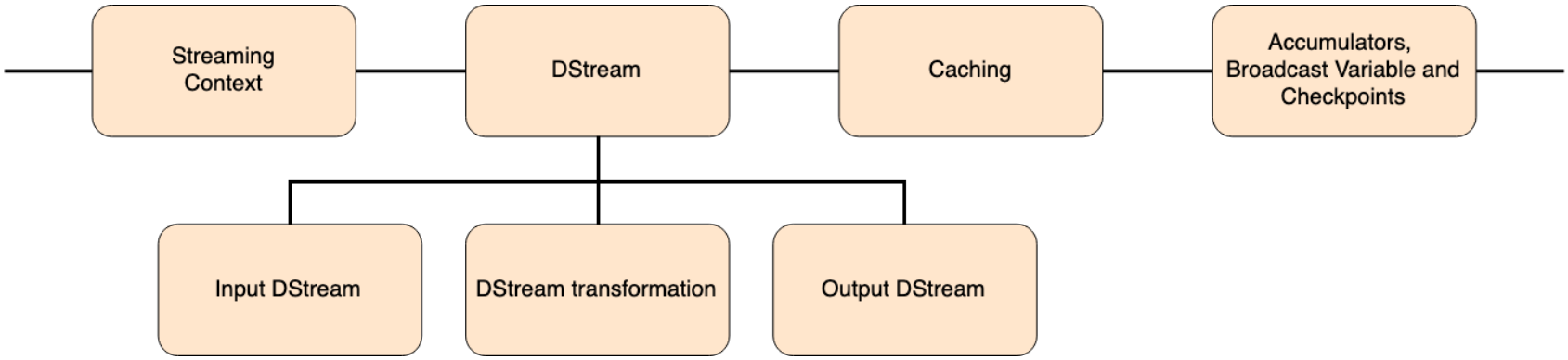


Figure: *Extracting words from an InputStream*

Streaming fundamentals



Streaming context

- Là entry point của chương trình Spark
  - Đầu vào là luồng dữ liệu nguồn, gọi là InputDStream và trả về một đối tượng Receiver
- Spark cung cấp một loạt các cài đặt có sẵn để kết nối với nguồn dữ liệu như Twitter, Akka Actor, ZeroMQ

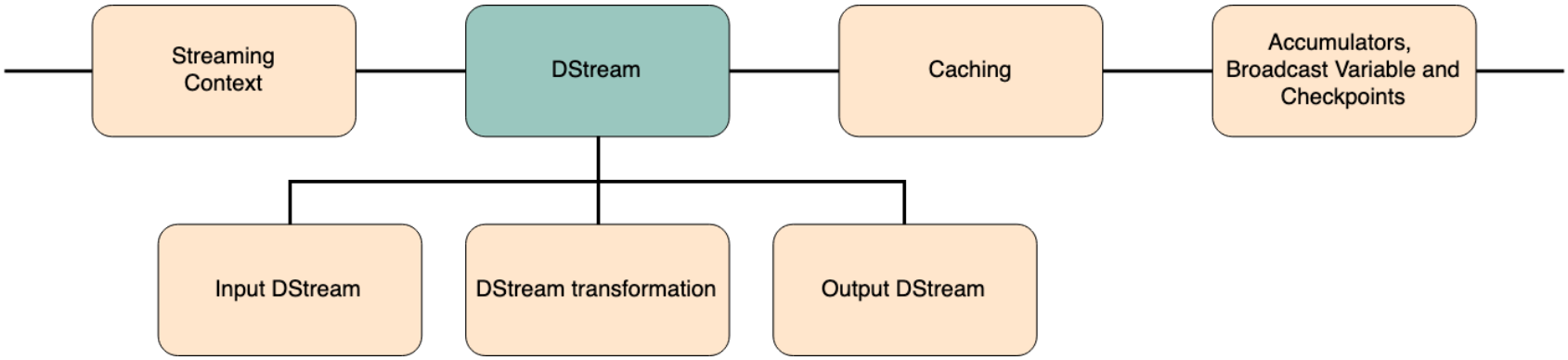
Figure: *Spark Streaming Context*

Figure: *Default Implementation Sources*

Khởi tạo - Initialization

- Một đối tượng StreamingContext có thể được tạo ra từ một đối tượng SparkContext
- Một SparkContext thể hiện một kế nối tới 1 cụm Spark mà có thể được dùng để khởi tạo RDDs, accumulators và broadcast variables trên cụm cluster đó

```
import org.apache.spark._
import org.apache.spark.streaming._
var ssc = new StreamingContext(sc,Seconds(1))
```


Streaming Context
DStream
Caching
Accumulators, Broadcast Variable and Checkpoints
Input DStream
DStream transformation
Output DStream

DStream

- Luồng rời rạc(discretized stream) là cách trừu tượng hóa dữ liệu trong Spark Streaming
- Dữ liệu của DStream đến từ các nguồn, hệ thống sinh dữ liệu bên ngoài hoặc là kết quả trả về từ các phép biến đổi (transformation) trên một nguồn dữ liệu đầu vào
- Dstream xem xét luồng dữ liệu như là một chuỗi liên tục của các RDDs, mỗi RDD là một phân đoạn dữ liệu của luồng dữ liệu đầu vào

Figure: *Input data stream divided into discrete chunks of data*

DStream operation

- Bất kỳ một phép toán nào áp dụng trên Dstream được chuyển đổi thành các phép toán trên các phân đoạn của luồng là các RDDs thành phần
- Ví dụ, cần chuyển đổi một luồng dữ liệu gồm các dòng thành luồng các từ, phép toán flatMap được thực thi trên mỗi RDD thành phần của DStream, từ đó tạo ra chuỗi các RDD của luồng DStream của các từ

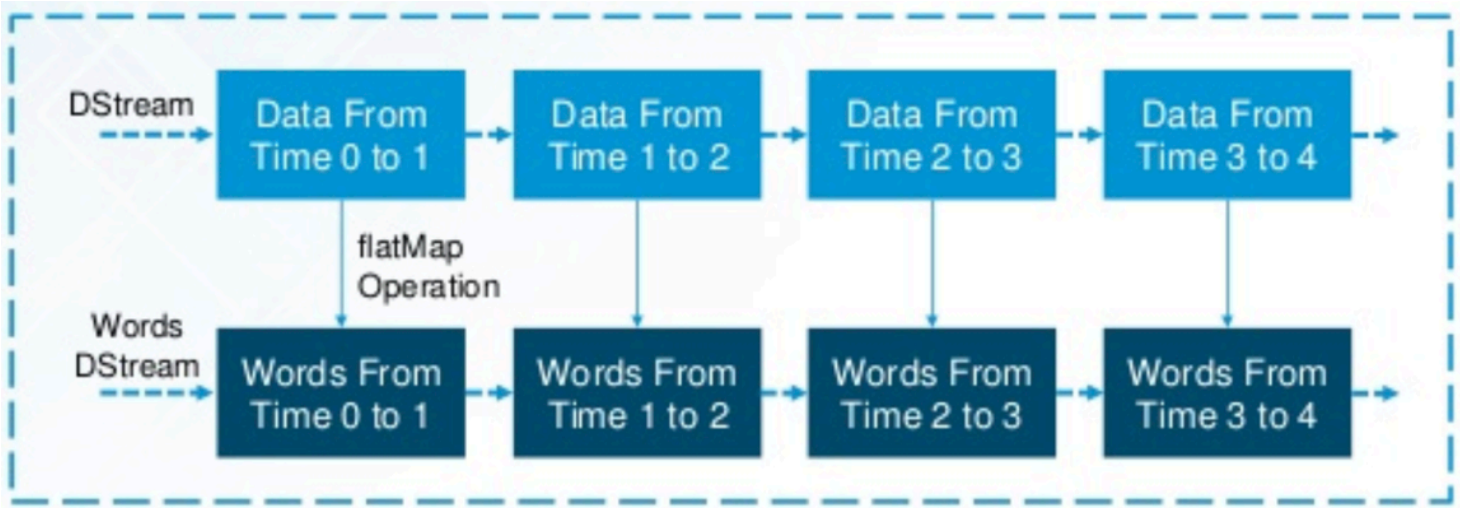
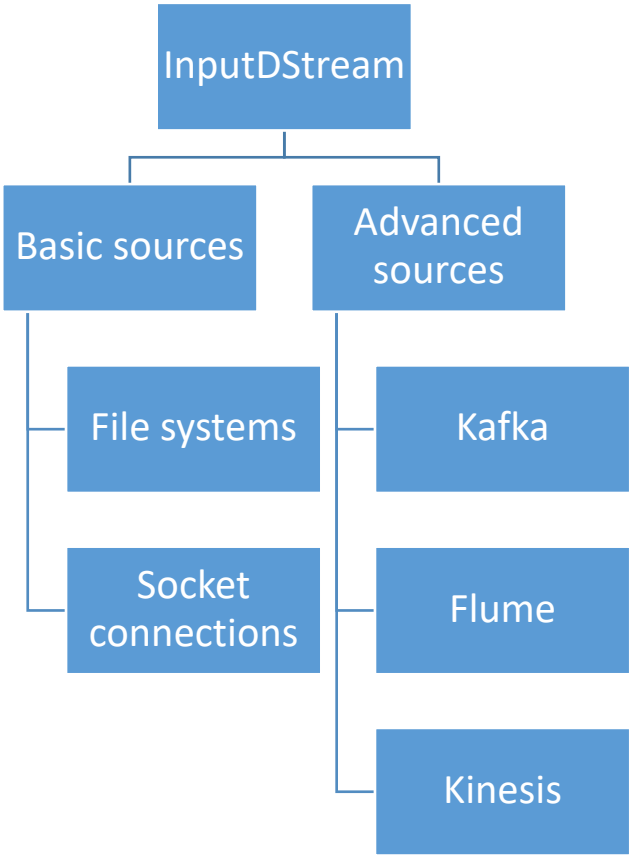


Figure: *Extracting words from an InputStream*

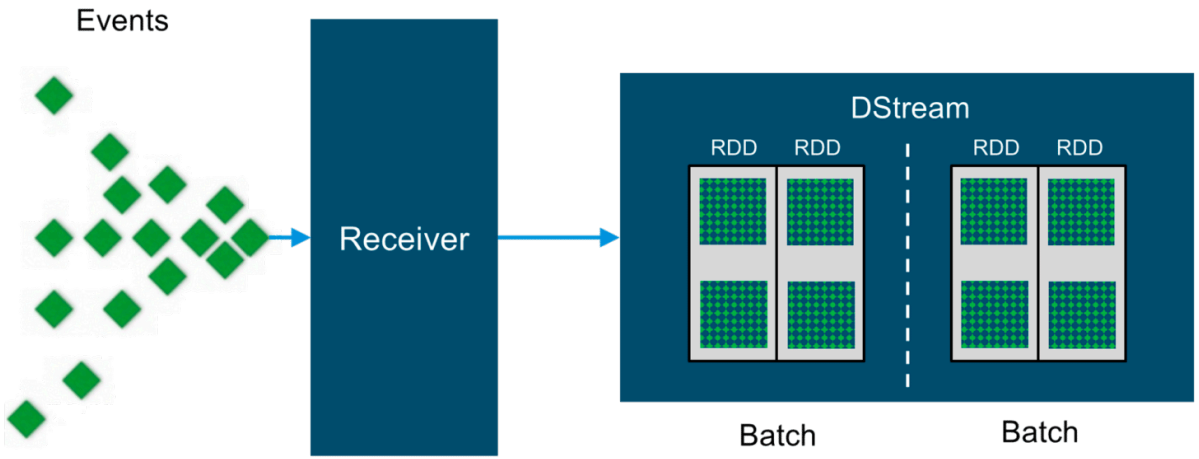
InputDStreams

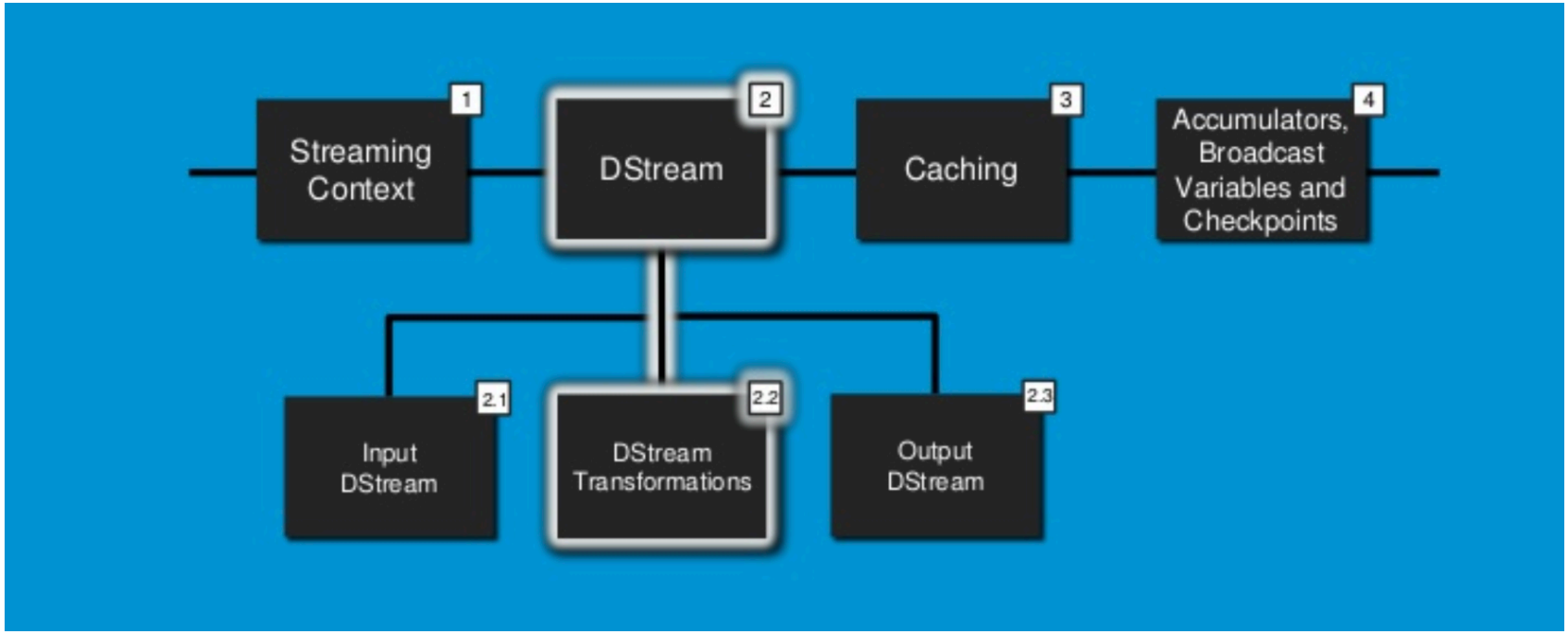
- InputDStreams là các DStreams khởi tạo từ luồng dữ liệu đến từ các nguồn bên ngoài Apache Spark



Receiver

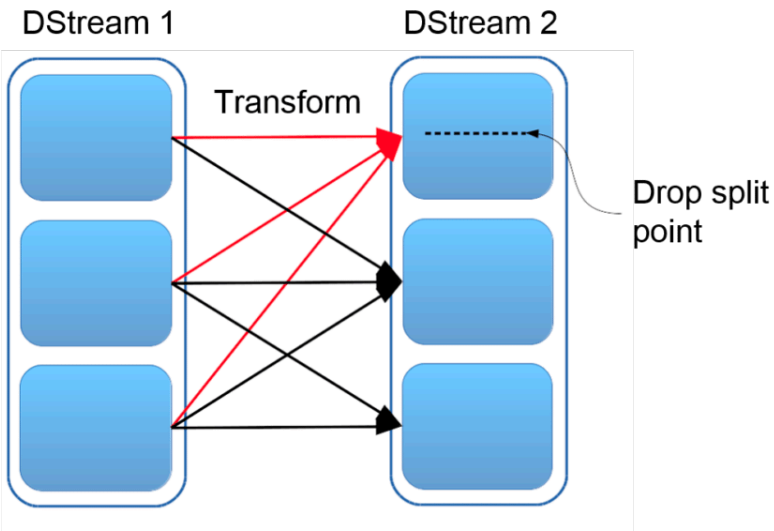
- Mỗi InputDStream được gán cho một đối tượng Receiver chịu trách nhiệm rời rạc hóa luồng thành các khối (batch) lưu trong bộ nhớ, mỗi khối bao gồm các RDD phân đoạn của luồng đầu vào




1
Streaming Context
2
DStream
3
Caching
4
Accumulators, Broadcast Variables and Checkpoints
2.1
Input DStream
2.2
DStream Transformations
2.3
Output DStream

Transformation trên DStreams

- Transformations cho phép biến đổi dữ liệu trên DStream giống như trên RDD
- DStream hỗ trợ nhiều dạng transformation khác nhau tương tự như với Spark RDD
- Các transformation phổ biến trên DStream
  - map, flatMap, filter, reduce, groupBy



Map(func) vs. Flatmap(func)

- Map(func): Trả về một DStream mới bằng cách biến đổi mỗi phần tử trong DStream nguồn qua một hàm *func*
- Flatmap(func): Tương tự như map(func) nhưng mỗi phần tử có thể được ánh xạ ra 0 hoặc nhiều phần tử đầu ra thông qua hàm *func*
  - Ví dụ
    - Input: [[1,2,3],[4,5,6],[7,8,9]]
    - Output: [1,2,3,4,5,6,7,8,9]

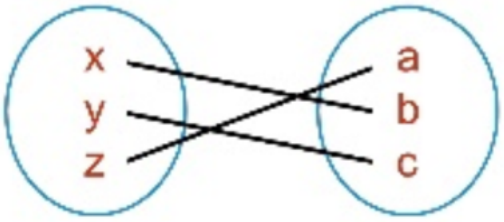


Figure: *Map Function*

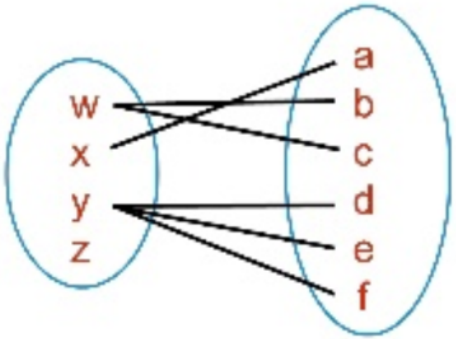


Figure: *flatMap Function*

Filter(func) & reduce(func)

- Filter: trả về một DStream mới bằng cách giữ lại các bản ghi thuộc DStream nguồn mà hàm func trên bản ghi đó trả về True
- Reduce: trả về một DStream là tập các RDD chỉ 1 phần tử bằng cách kết tập các phần tử trong mỗi RDD của DStream nguồn sử dụng hàm *func*

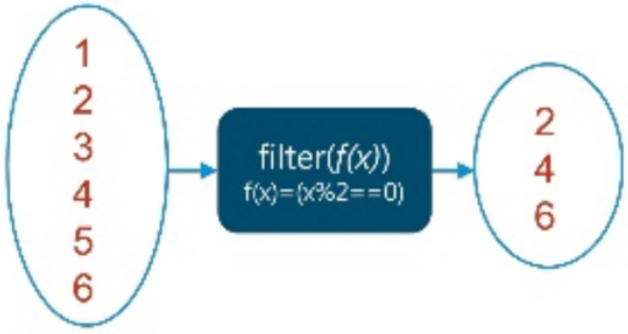


Figure: *Filter Function For Even Numbers*

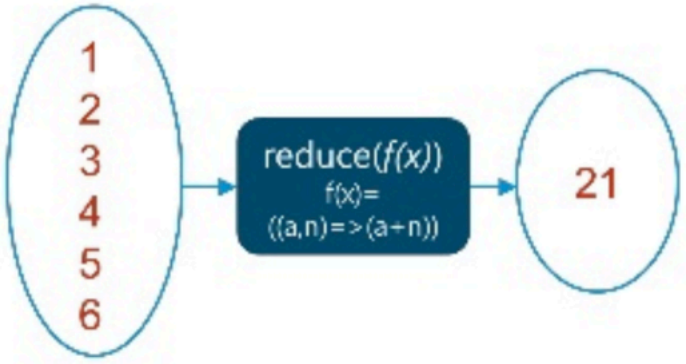


Figure: *Reduce Function To Get Cumulative Sum*

groupBy(func)

- Trả về RDD mới mà cơ bản được tạo thành bằng cách nhóm các phần tử cùng khóa thành một nhóm xác định bởi *func*

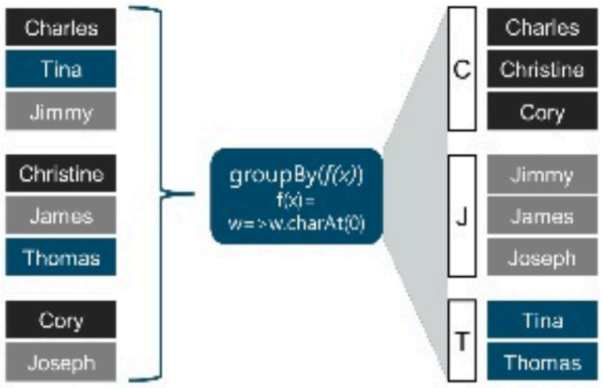


Figure: *Grouping By First Letters*

Dstream window

- Spark Streaming cung cấp cơ chế cho phép thực thi các transformation trên một cửa sổ trượt trên luồng dữ liệu

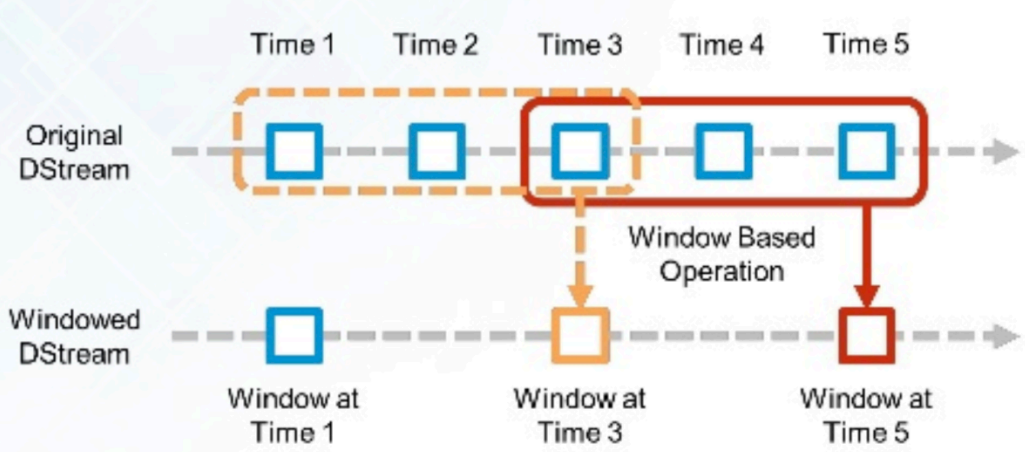
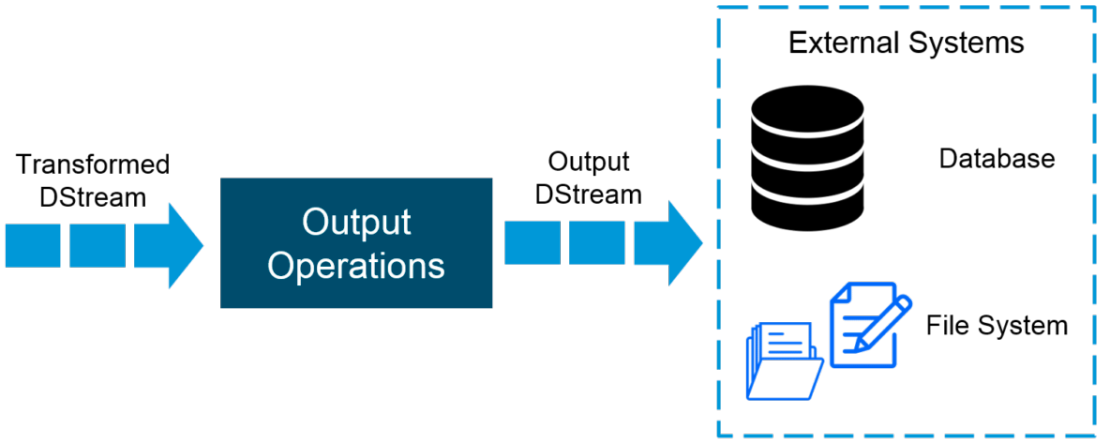


Figure: *DStream Window Transformation*

Đưa dữ liệu từ DStreams ra ngoài

- Output operations cho phép dữ liệu của DStream được ghi ra các hệ thống bên ngoài như CSDL, hệ thống tệp tin
- Output operations kích hoạt sự thực thi của các DStream transformations.



Output operations

Output Operation	Meaning
print()	Prints the first ten elements of every batch of data in a DStream on the driver node running the streaming application. This is useful for development and debugging. Python API This is called **pprint()** in the Python API.
saveAsTextFiles(*prefix*, [*suffix*])	Save this DStream's contents as text files. The file name at each batch interval is generated based on *prefix* and *suffix*: "*prefix-TIME_IN_MS[.suffix]*".
saveAsObjectFiles(*prefix*, [*suffix*])	Save this DStream's contents as `SequenceFiles` of serialized Java objects. The file name at each batch interval is generated based on *prefix* and *suffix*: "*prefix-TIME_IN_MS[.suffix]*". Python API This is not available in the Python API.
saveAsHadoopFiles(*prefix*, [*suffix*])	Save this DStream's contents as Hadoop files. The file name at each batch interval is generated based on *prefix* and *suffix*: "*prefix-TIME_IN_MS[.suffix]*". Python API This is not available in the Python API.
foreachRDD(*func*)	The most generic output operator that applies a function, *func*, to each RDD generated from the stream. This function should push the data in each RDD to an external system, such as saving the RDD to files, or writing it over the network to a database. Note that the function *func* is executed in the driver process running the streaming application, and will usually have RDD actions in it that will force the computation of the streaming RDDs.

Output operations example - foreachRDD

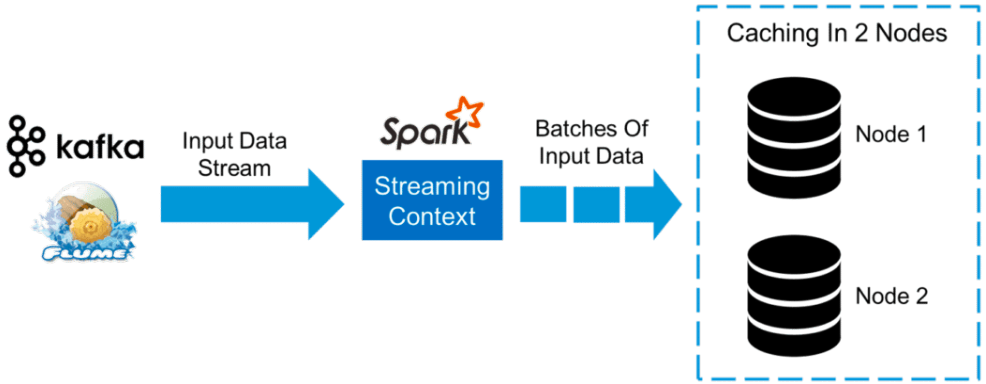
foreachRDD

- dstream.foreachRDD is a powerful primitive that allows data to be sent out to external systems.
- The lazy evaluation achieves the most efficient transfer of data.

```
dstream.foreachRDD { rdd =>
 rdd.foreachPartition { partitionOfRecords =>

// ConnectionPool is a static, lazily initialized pool of connections
val connection = ConnectionPool.getConnection()
 partitionOfRecords.foreach(record => connection.send(record))

// Return to the pool for future reuse
ConnectionPool.returnConnection(connection)
 }
}
```

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()
- Với input streams nhận dữ liệu từ mạng như Kafka, Flume, Sockets, cấu hình mức bảo lưu dữ liệu thường chỉ định nhân bản dữ liệu tới ít nhất 2 nút máy chủ để đảm bảo chịu lỗi



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

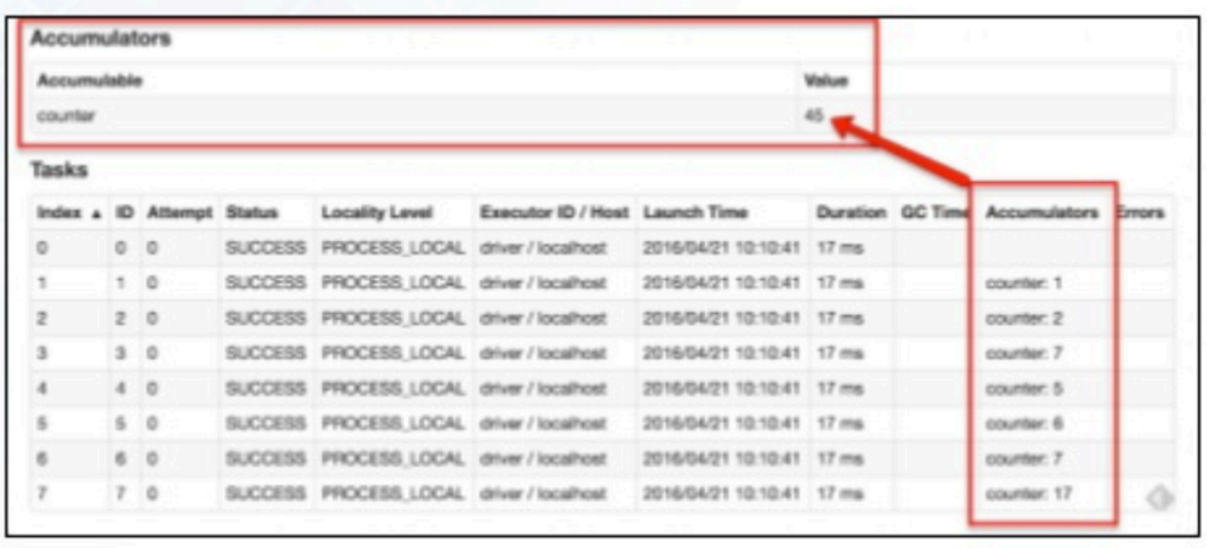


Figure: *Accumulators In Spark Streaming*

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

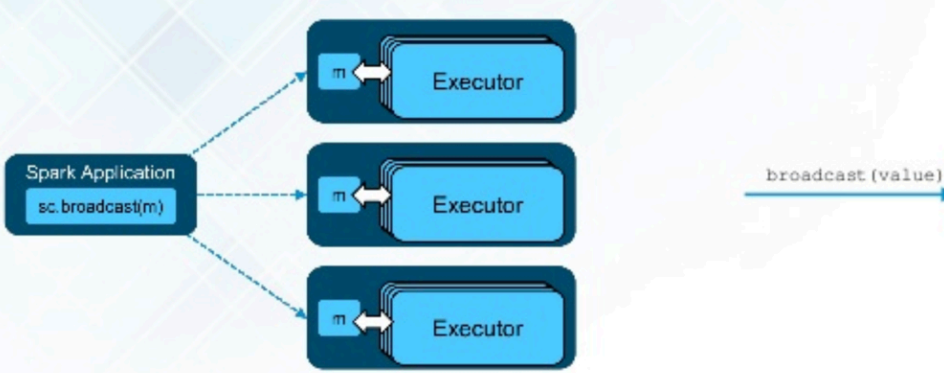


Figure: *Broadcasting A Value To Executors*

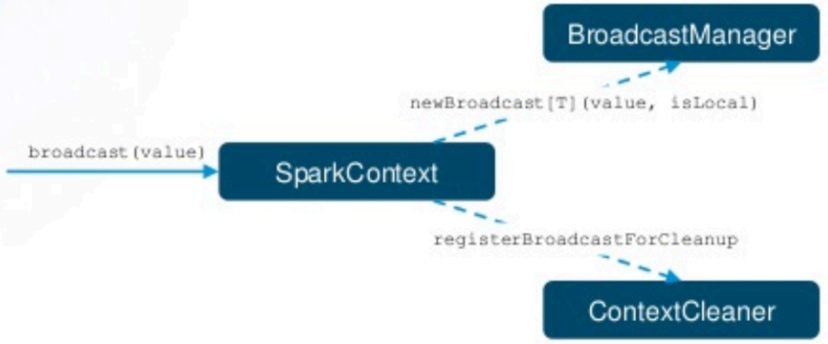


Figure: *SparkContext and Broadcasting*

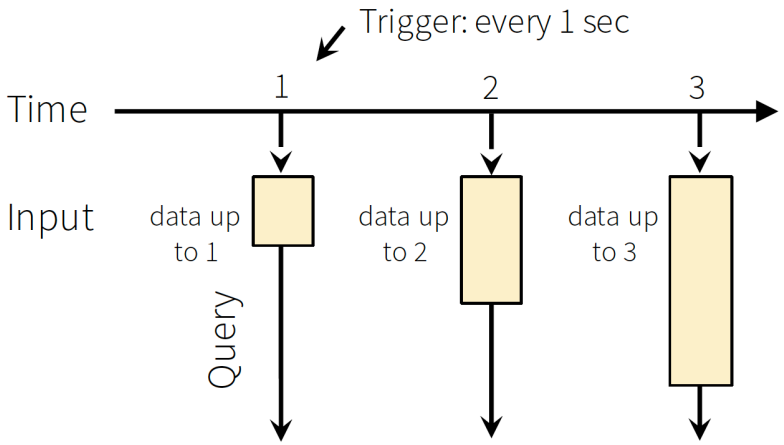
Structured streaming

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

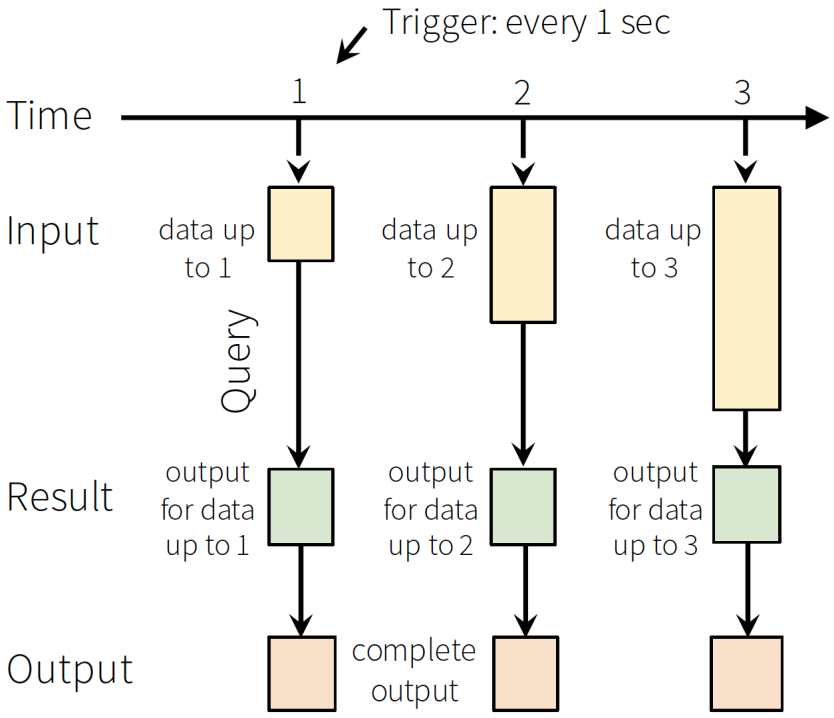
New model

- 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



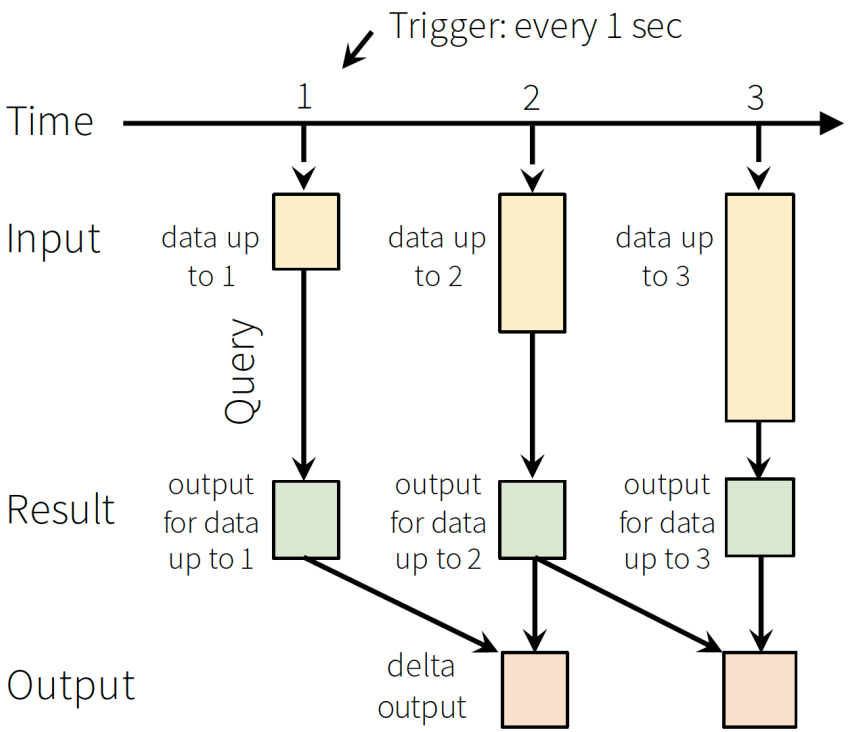
New model (2)

- 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



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()

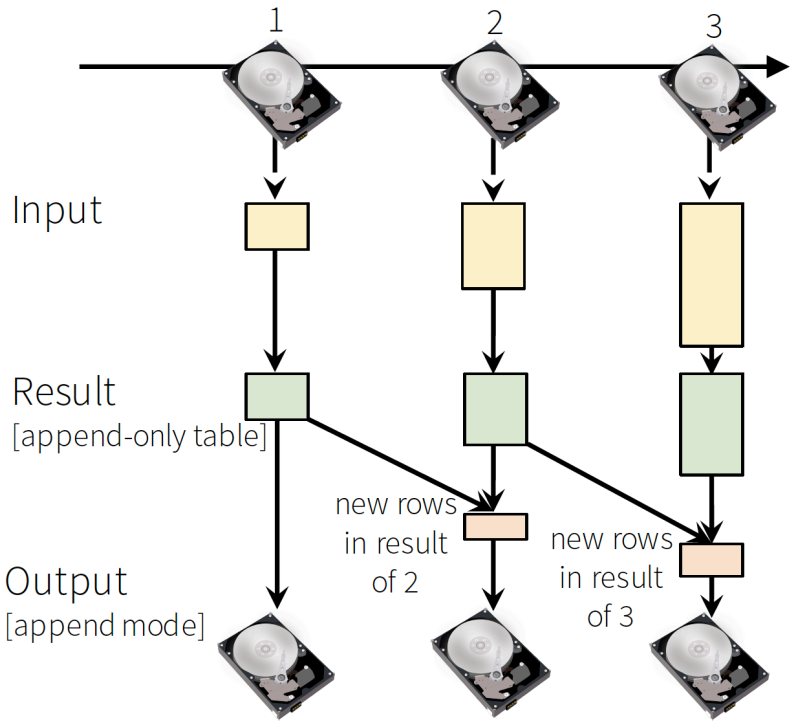
Streaming ETL with DataFrames

```
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")
```

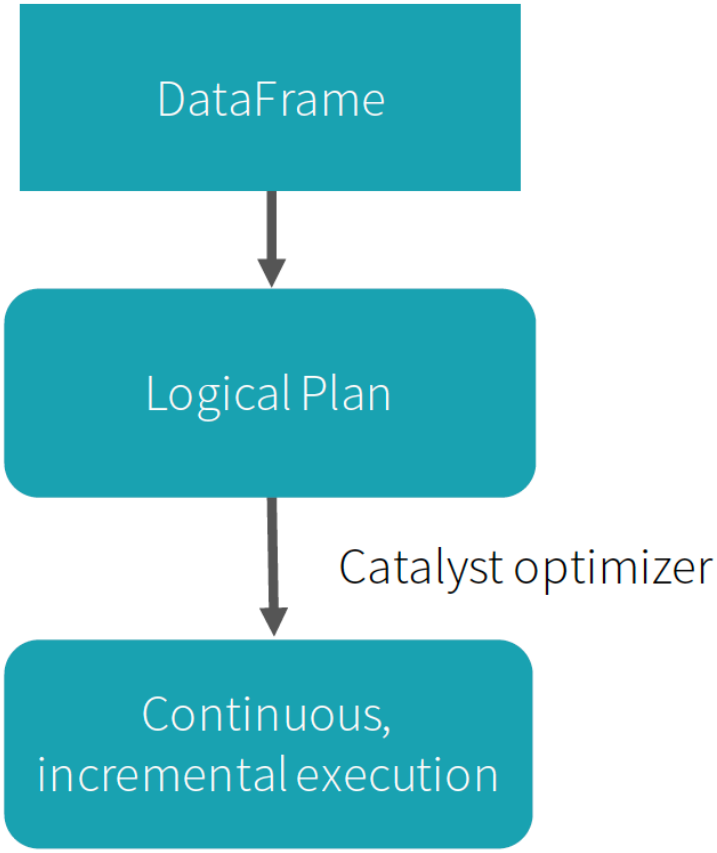
Query Management

```
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)

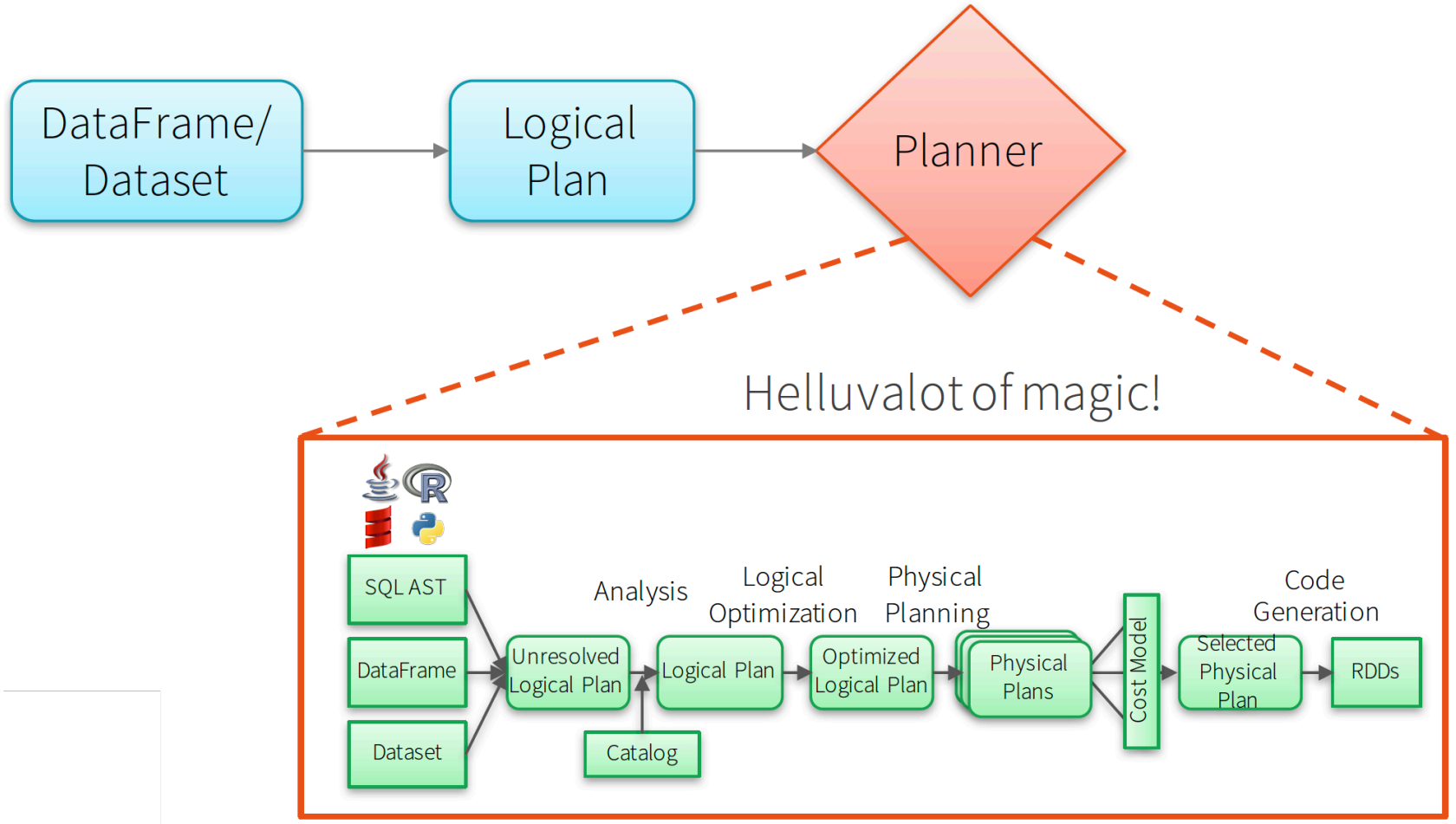


Structured Streaming

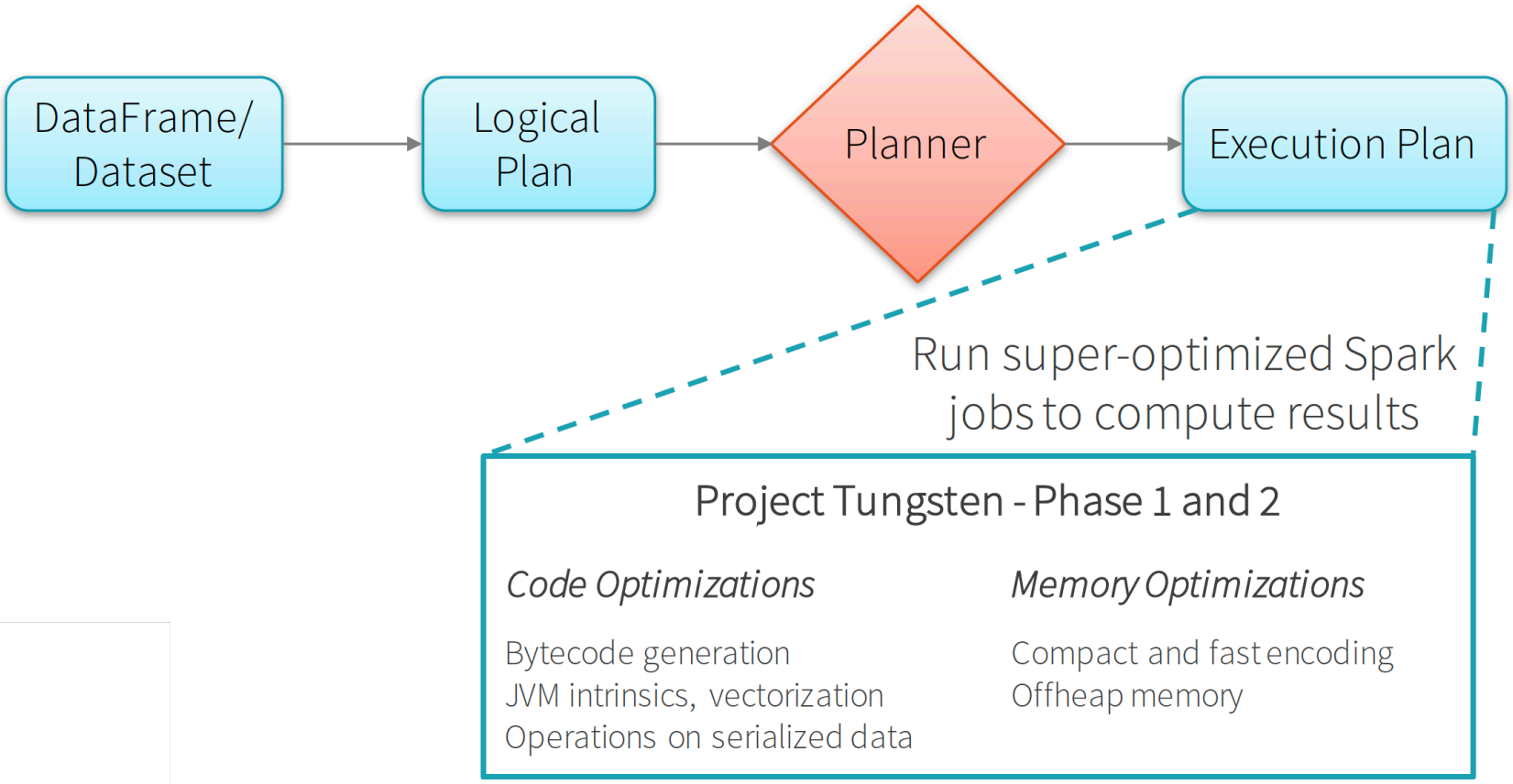
- High-level streaming API built on Datasets/DataFrames
  - Event time, windowing, sessions, sources & sinks
  - End-to-end exactly once semantics
- Unifies streaming, interactive and batch queries
  - Aggregate data in a stream, then serve using JDBC
  - Add, remove, change queries at runtime
  - Build and apply ML models

Internal execution

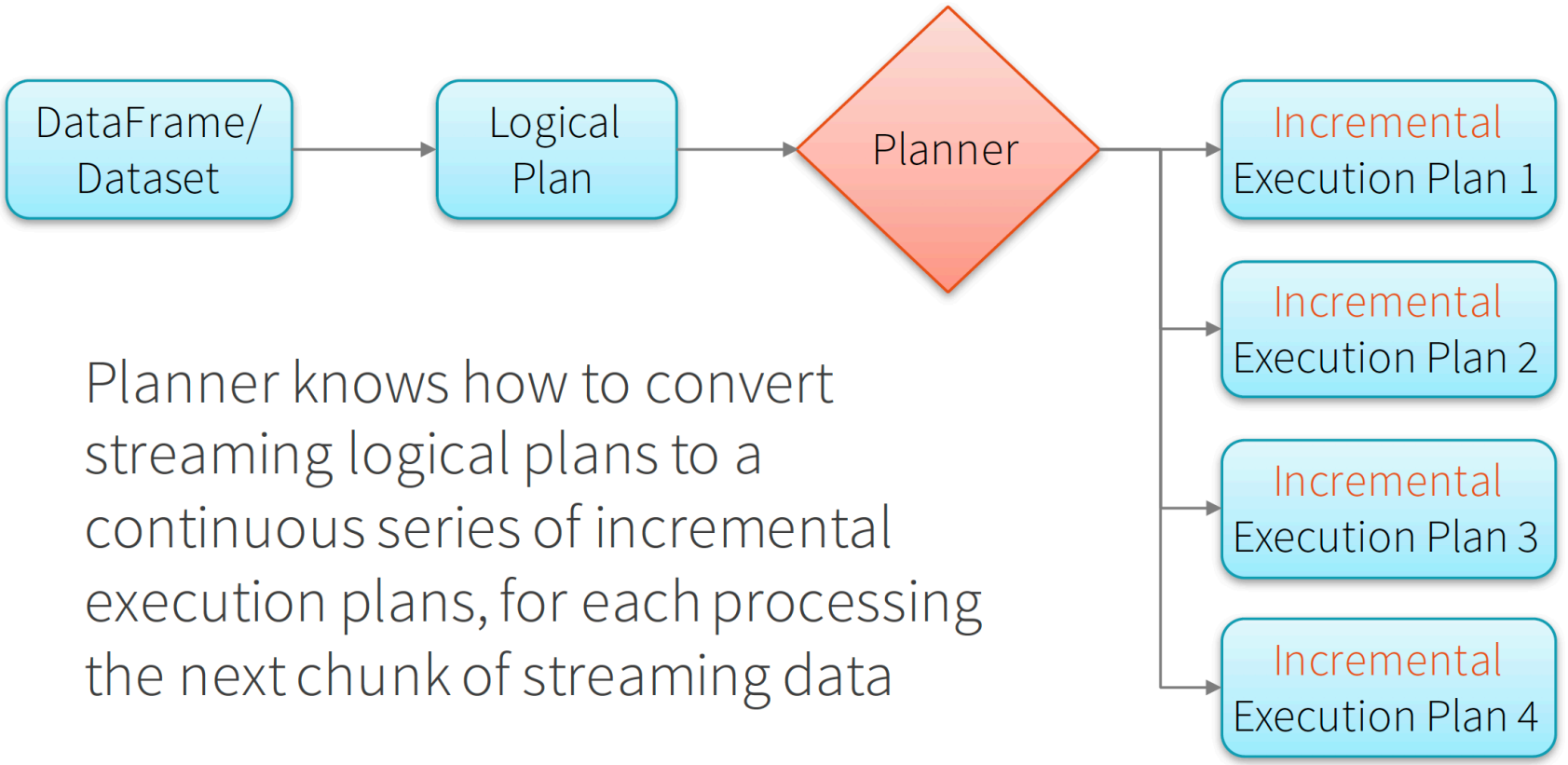
Batch Execution on Spark SQL



Batch Execution on Spark SQL

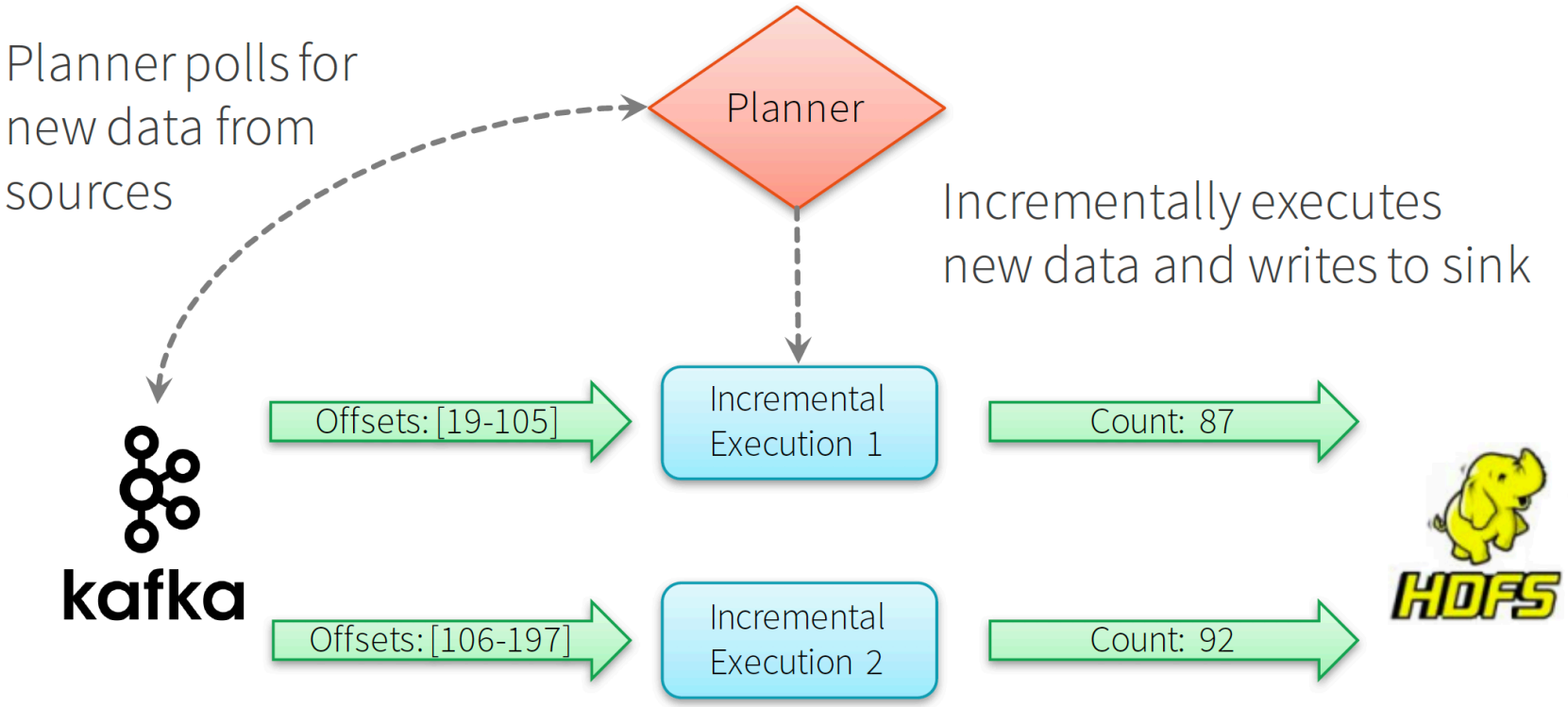


Continuous Incremental Execution



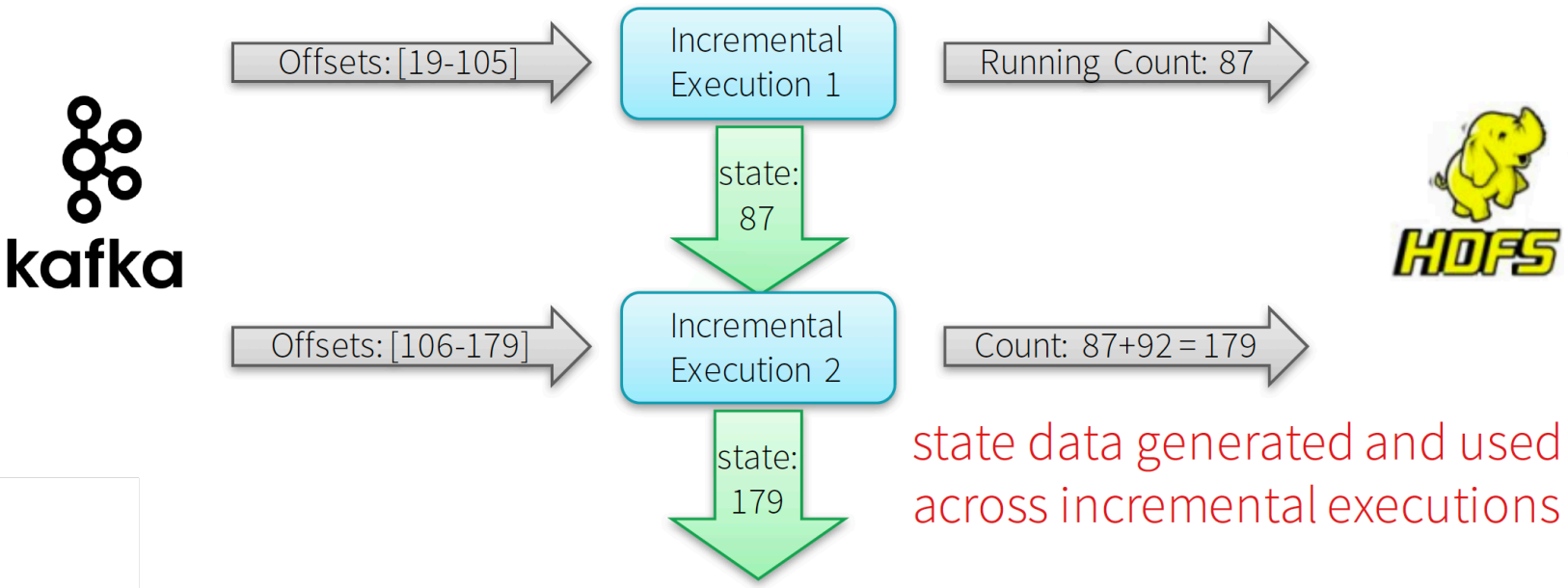
Planner knows how to convert streaming logical plans to a continuous series of incremental execution plans, for each processing the next chunk of streaming data

Continuous Incremental Execution



Continuous Aggregations

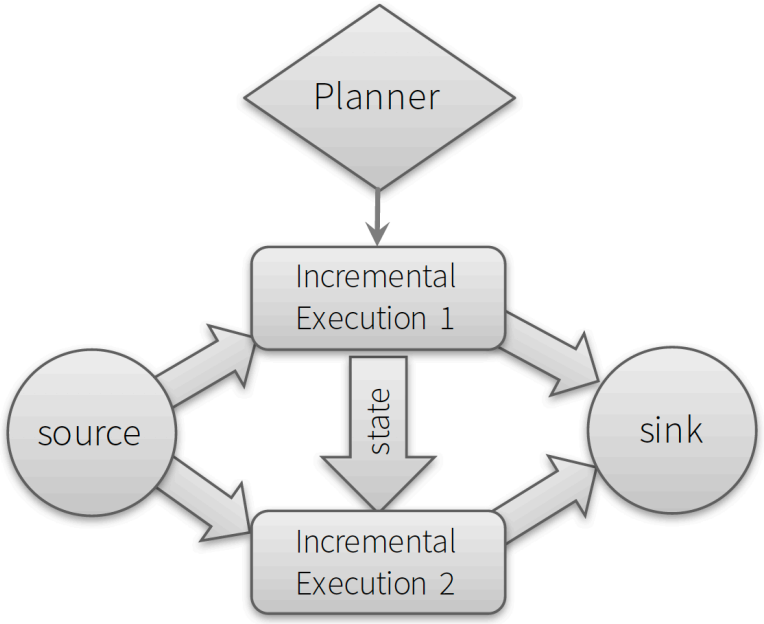
Maintain running aggregate as in-memory state backed by WAL in file system for fault-tolerance



state data generated and used across incremental executions

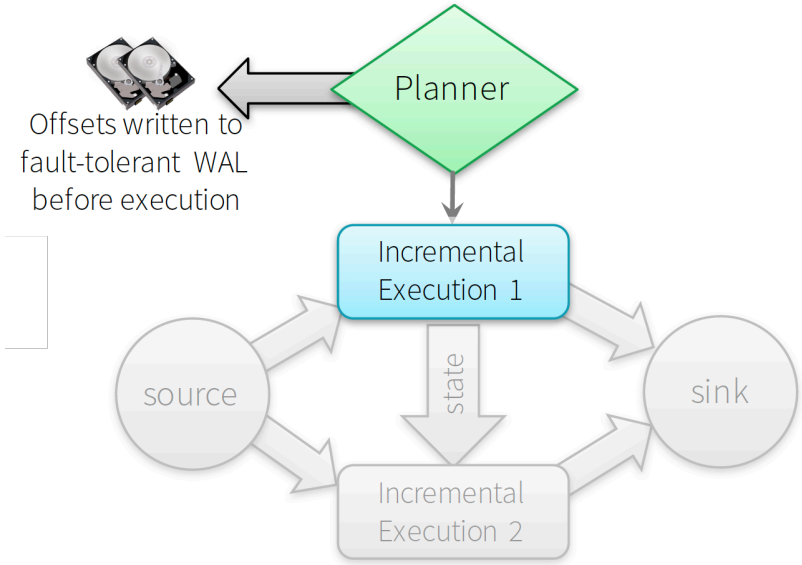
Fault-tolerance

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



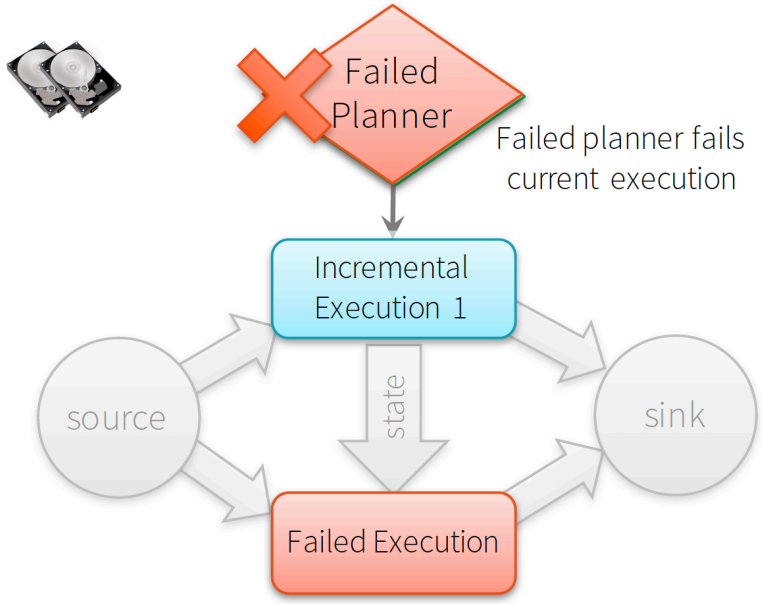
Fault-tolerant Planner

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



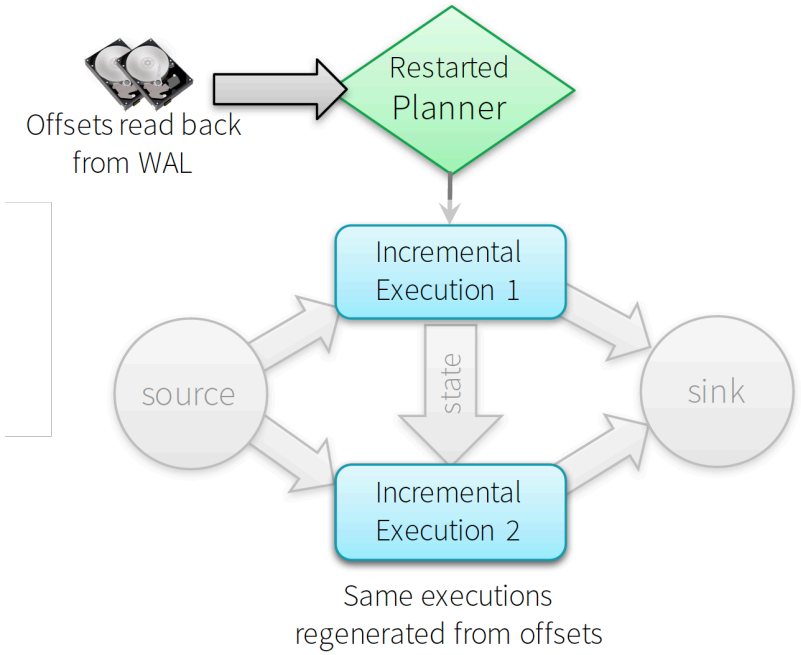
Fault-tolerant Planner

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



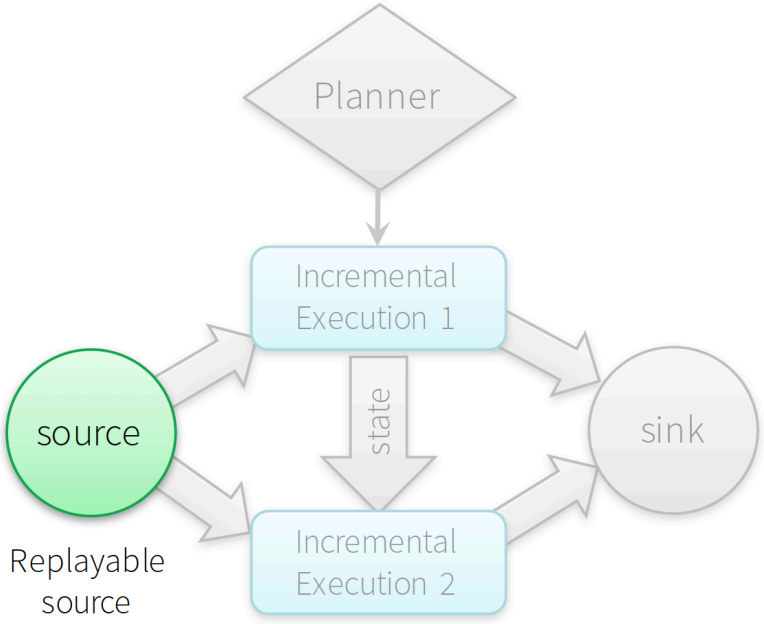
Fault-tolerant Planner

- 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



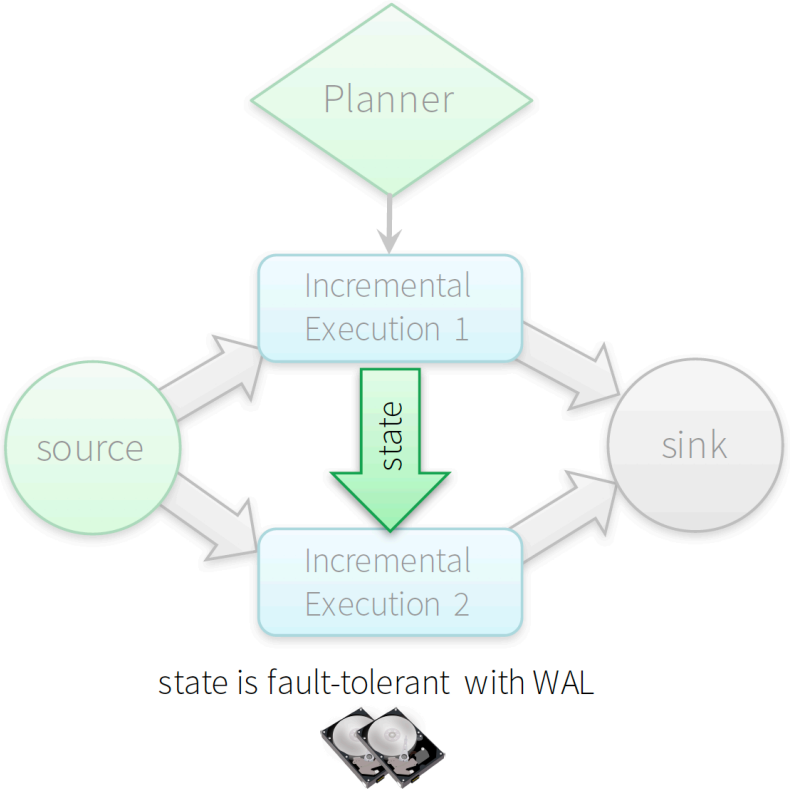
Fault-tolerant Sources

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



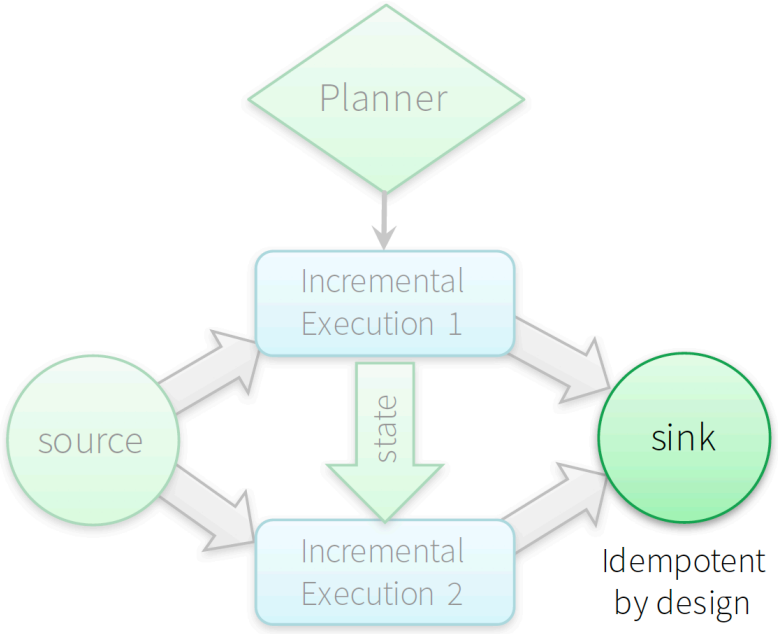
Fault-tolerant State

- 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

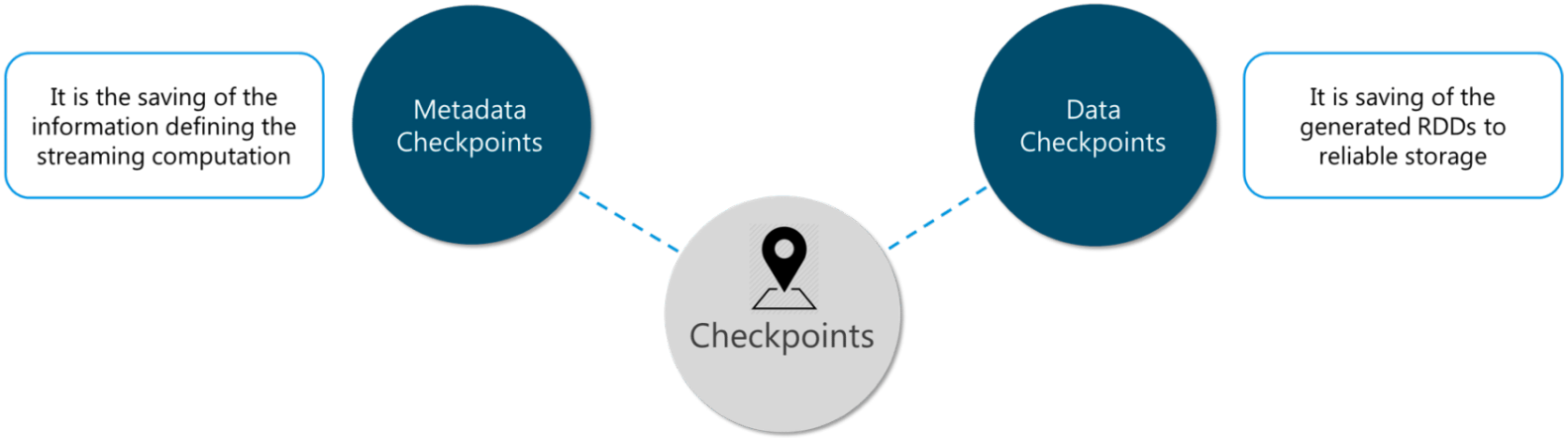
Fast, fault-tolerant, exactly-once

stateful stream processing

without having to reason about streaming

Checkpoints

- Spark hỗ trợ cơ chế checkpointing
  - Metadata
  - Data



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*

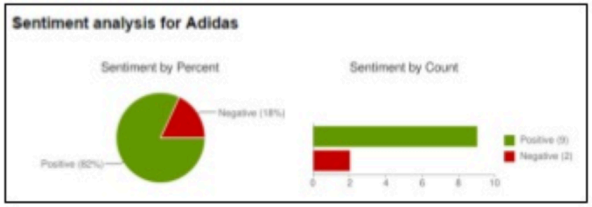


Figure: *Twitter Sentiment Analysis For Adidas*

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

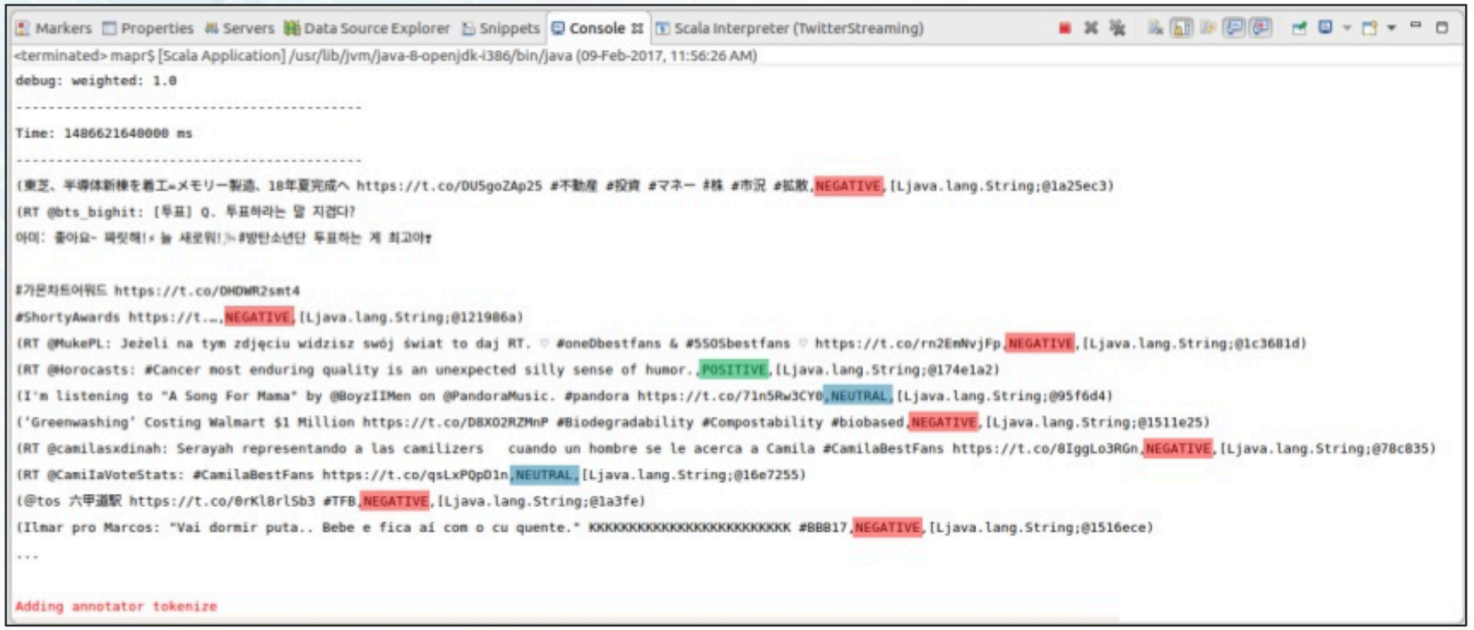


Figure: *Sentiment Analysis Output In Eclipse IDE*

Output directory

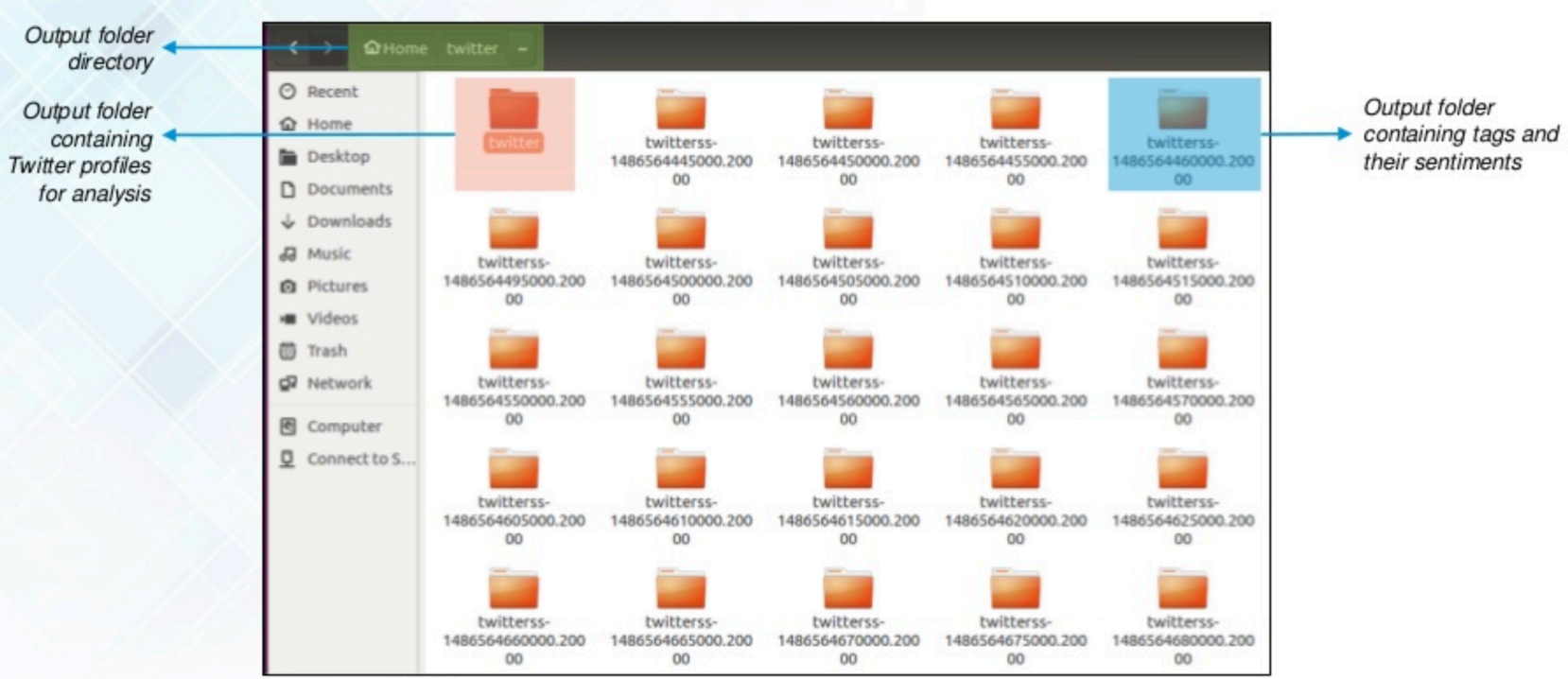
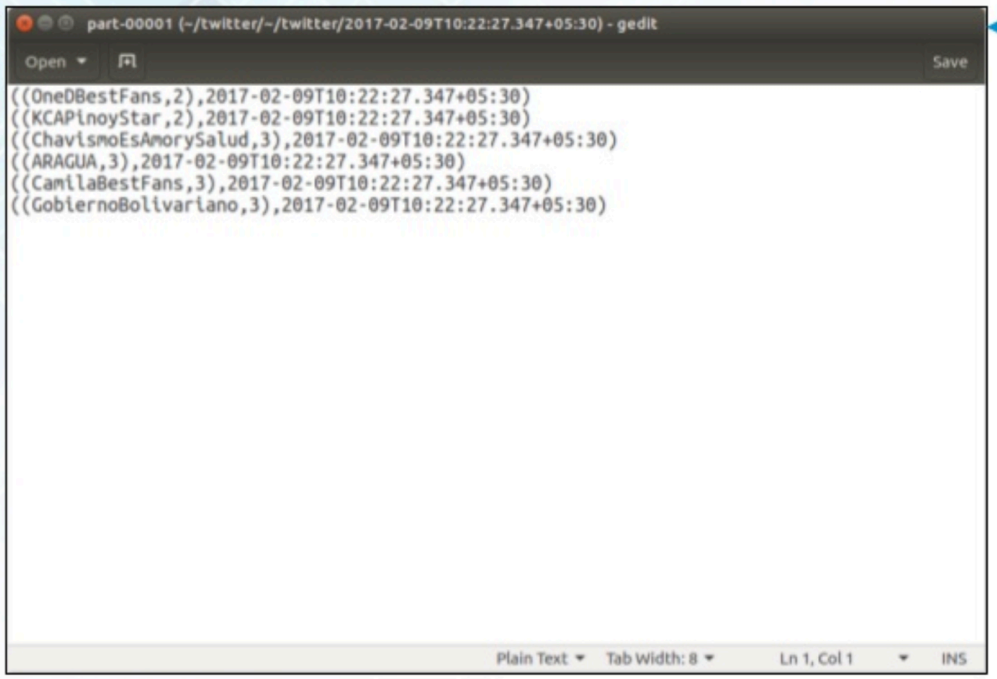


Figure: *Output folders inside our 'twitter' project folder*

Output usernames



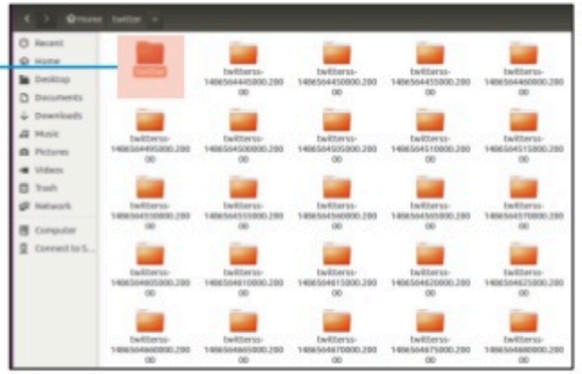


Figure: *Output file containing Twitter username and timestamp*

Output tweets and sentiments

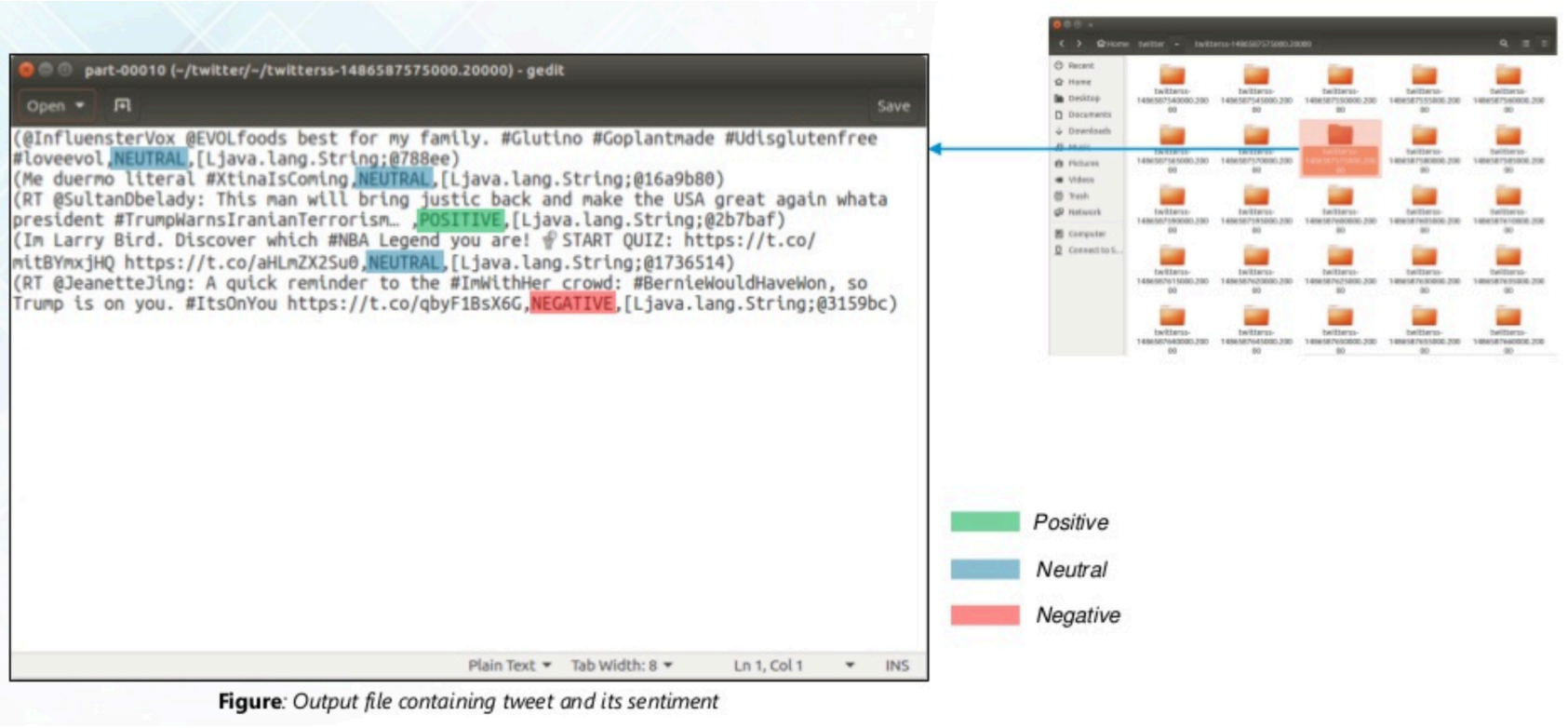


Figure: *Output file containing tweet and its sentiment*

Sentiments for Trump

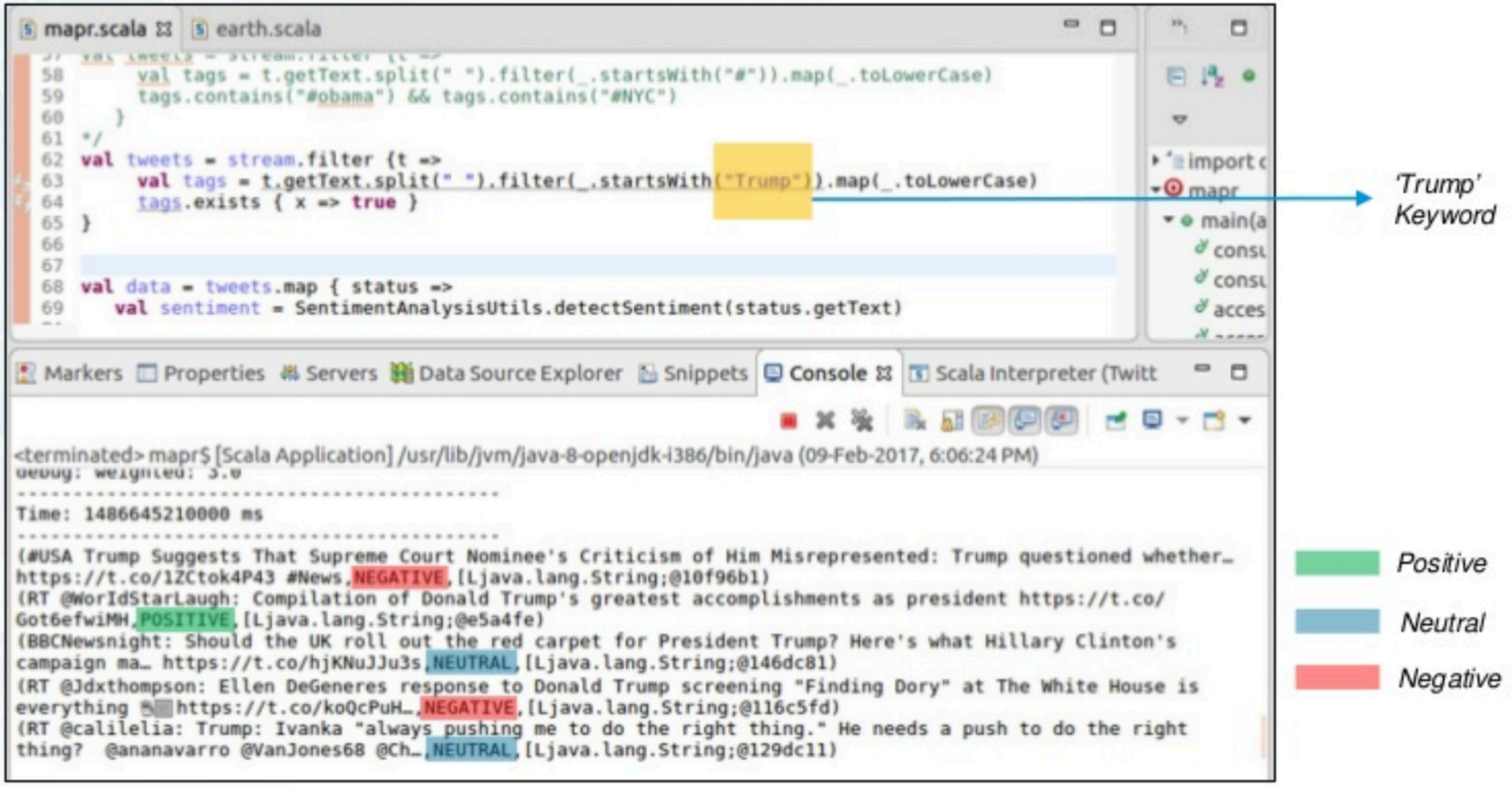


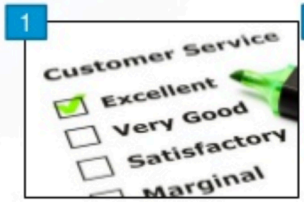
Figure: *Performing Sentiment Analysis on Tweets with 'Trump' Keyword*

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