UNIT-4

SPARK SQL

- Spark SQL brings native support for SQL to Spark and streamlines the process of querying data stored both in RDDs (Spark’s distributed datasets) and in external source.
- Unifying these powerful abstractions makes it easy for developers to intermix SQL commands querying external data with complex analytics, all within in a single application.
- Spark SQL will allow developers to:
  - Import relational data from Parquet files and Hive tables
  - Run SQL queries over imported data and existing RDDs
  - Easily write RDDs out to Hive tables or Parquet files

Spark sql data frames

- Limitations with RDD’s:
  - When working with structured data, there was no inbuilt optimization engine.
  - There was no provision to handle structured data.
- Data frames can be used to overcome the above limitations.
- Dataframe:
  - It is a distributed collection of data ordered into named columns. Concept wise it is equal to the table in a relational database or a data frame in **R**/ Python.

We can create DataFrame using:

- Structured data files
- Tables in Hive
- External databases
- Using existing RDD

Spark SQL Datasets

- **Spark Dataset** is an interface added in version Spark 1.6. it is a distributed collection of data.
- Dataset provides the **benefits of RDDs** along with the benefits of Apache Spark SQL's optimized execution engine.
- A Dataset can be made using JVM objects and after that, it can be manipulated using functional transformations (map, filter etc.).
- The Dataset API is accessible in *Scala* and *Java*. Dataset API is not supported by Python.

Spark Catalyst Optimizer

- The *optimizer* used by Spark SQL is **Catalyst optimizer**. It optimizes all the queries written in Spark SQL and DataFrame DSL.
- **Spark Catalyst** is a library built as a rule-based system. And each rule focusses on the specific optimization.
- Uses of Apache Spark SQL
- It executes SQL queries.
- We can read data from existing **Hive installation** using SparkSQL.
- When we run SQL within another programming language we will get the result as Dataset/DataFrame.

1. Load the dataset as a DataFrame:

```
      df =
spark.read.json("/home/spark/sampledata/json/cdrs.json")
```

2. Print the top 20 records from the data frame:

```
      df.show(20)
```

3. Display Schema:

```
      df.printSchema()
```

4. Count and view the total number of calls originating from London:

```
      df.filter("Origin = 'London'").count()
      df.filter("Origin = 'London'").show()
```

5. Count total revenue with calls originating from revenue and terminating in Manchester:

```
      df.filter("Origin = 'London'").filter("Dest =
'Manchester'").show()
      df.filter("Origin = 'London'").filter("Dest =
        'Manchester'").agg({"CallCharge":"sum"}).show()
```

6. Register the dataset as a table to be operated on using SQL:

```
      df.createOrReplaceTempView("calldetails")
      spark.sql("select Dest, count(*) as callCnt from
calldetails
      group by Dest Order by callCnt Desc").show()
```

Reverting to an RDD from a DataFrame

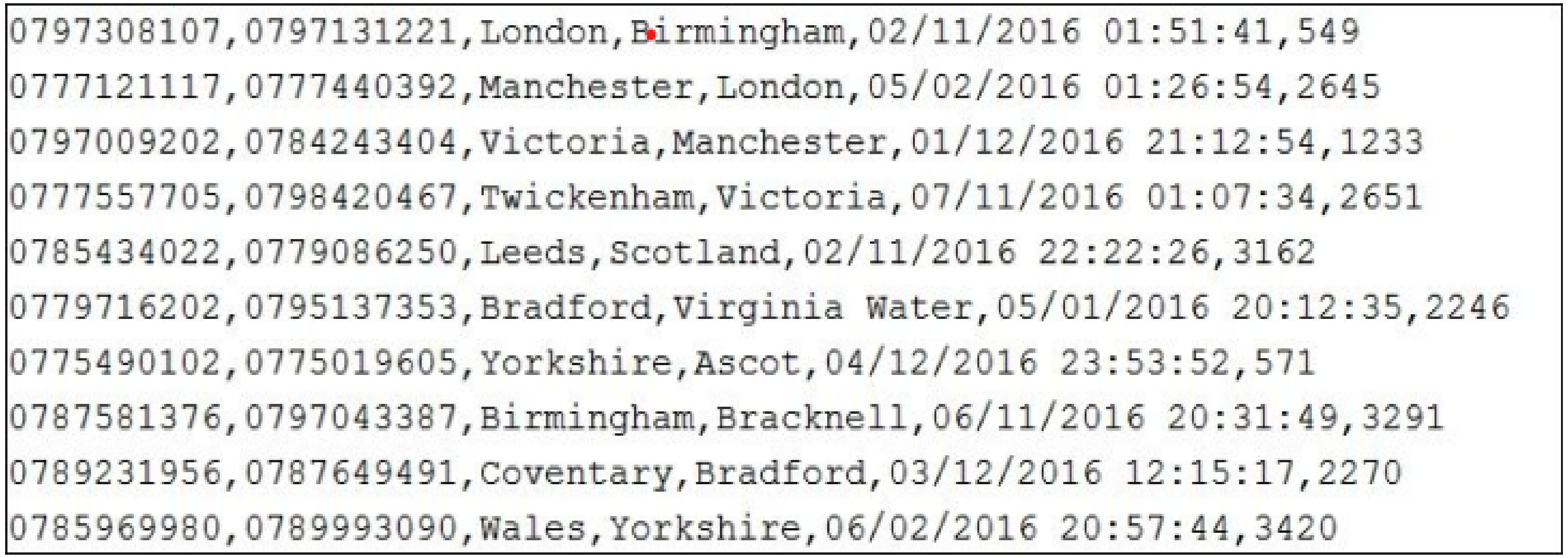
- There might be cases where you might want to switch back to working with RDD. Fortunately, Spark provides the ability to switch back to RDD interface in Scala, Python, and Java. There is no way to revert back to an RDD from the R API.
- Example:Reverting to RDD using Python API

```
callDetailRecordsRDD = callDetailsDF.rdd;
```

Converting an RDD to a DataFrame

If you are working with RDDs and feel that for a certain set of operations a DataFrame would be more suitable, you can switch to a DataFrame. A DataFrame needs a bit more information than an RDD can provide so you will need to provide the schema to the Spark Framework

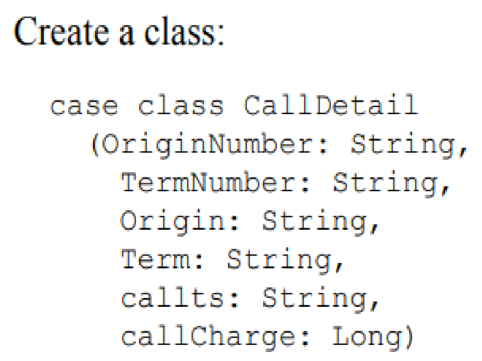
Example: consider sample data in csv format



```
0797308107,0797131221,London,Birmingham,02/11/2016 01:51:41,549
0777121117,0777440392,Manchester,London,05/02/2016 01:26:54,2645
0797009202,0784243404,Victoria,Manchester,01/12/2016 21:12:54,1233
0777557705,0798420467,Twickenham,Victoria,07/11/2016 01:07:34,2651
0785434022,0779086250,Leeds,Scotland,02/11/2016 22:22:26,3162
0779716202,0795137353,Bradford,Virginia Water,05/01/2016 20:12:35,2246
0775490102,0775019605,Yorkshire,Ascot,04/12/2016 23:53:52,571
0787581376,0797043387,Birmingham,Bracknell,06/11/2016 20:31:49,3291
0789231956,0787649491,Coventary,Bradford,03/12/2016 12:15:17,2270
0785969980,0789993090,Wales,Yorkshire,06/02/2016 20:57:44,3420
```

Converting RDD to DataFrame using Scala API

We are going to use the case class syntax to create a call detail class, and then use that to create rows of CallDetails from the data file.



Create a class:

```
case class CallDetail
  (OriginNumber: String,
    TermNumber: String,
    Origin: String,
    Term: String,
    callts: String,
    callCharge: Long)
```

- Create a DataFrame from the RDD by splitting the data using the separator comma and then applying the CallDetail class. We then use the individual attributes and map them to the CallDetail object. Finally, we use the toDF method to create a DataFrame:

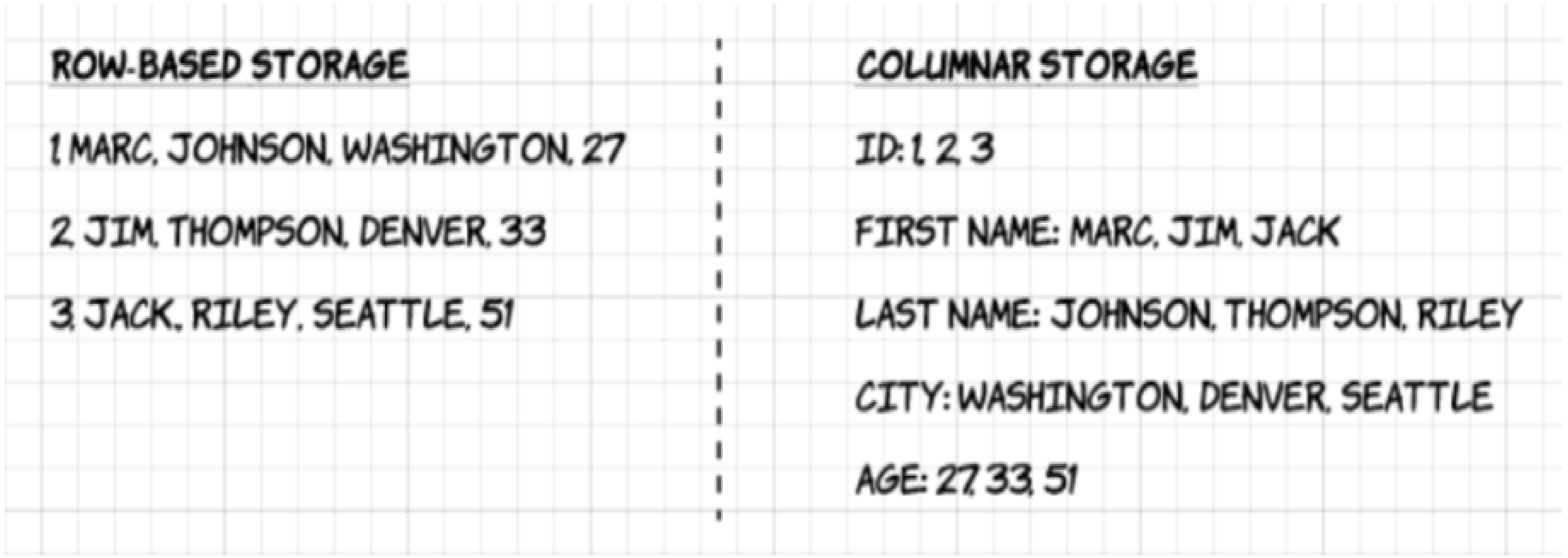
```
val callDetailsDataFrame =
sc.textFile("/home/spark/sampledata/cdrs.csv")
   .map(callDetail => callDetail.split(","))
   .map(attributes => CallDetail
      (attributes(0),
      attributes(1),
      attributes(2),
      attributes(3),
      attributes(4),
      attributes(5).toInt))
   .toDF()
      callDetailsDataFrame.show()
```

Parquet files

- Apache Parquet is an open source, column-oriented data file format designed for efficient data storage and retrieval.
- It provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk.
- Apache Parquet is designed to be a common interchange format for both batch and interactive workloads

Apache Parquet is a file format designed to support fast data processing for complex data, with several notable **characteristics**:

- **Columnar:** Unlike row-based formats such as CSV or Avro, Apache Parquet is column-oriented – meaning the values of each table column are stored next to each other, rather than those of each record:



ROW-BASED STORAGE

1 MARC, JOHNSON, WASHINGTON, 27

2 JIM, THOMPSON, DENVER, 33

3 JACK, RILEY, SEATTLE, 51

COLUMNAR STORAGE

ID: 1, 2, 3

FIRST NAME: MARC, JIM, JACK

LAST NAME: JOHNSON, THOMPSON, RILEY

CITY: WASHINGTON, DENVER, SEATTLE

AGE: 27, 33, 51

- **Open-source:** Parquet is free to use and open source under the Apache Hadoop license, and is compatible with most Hadoop data processing frameworks.
- **Self-describing**: In addition to data, a Parquet file contains metadata including schema and structure. Each file stores both the data and the standards used for accessing each record – making it easier to decouple services that write, store, and read Parquet files.

Advantages of parquet

- **Compression**

In Parquet, compression is performed column by column and it is built to support flexible compression options and extendable encoding schemas per data type – e.g., different encoding can be used for compressing integer and string data.

- **Performance**

Parquet is optimized for performance. When running queries on your Parquet-based file-system, you can focus only on the relevant data very quickly. Moreover, the amount of data scanned will be way smaller and will result in less I/O usage.

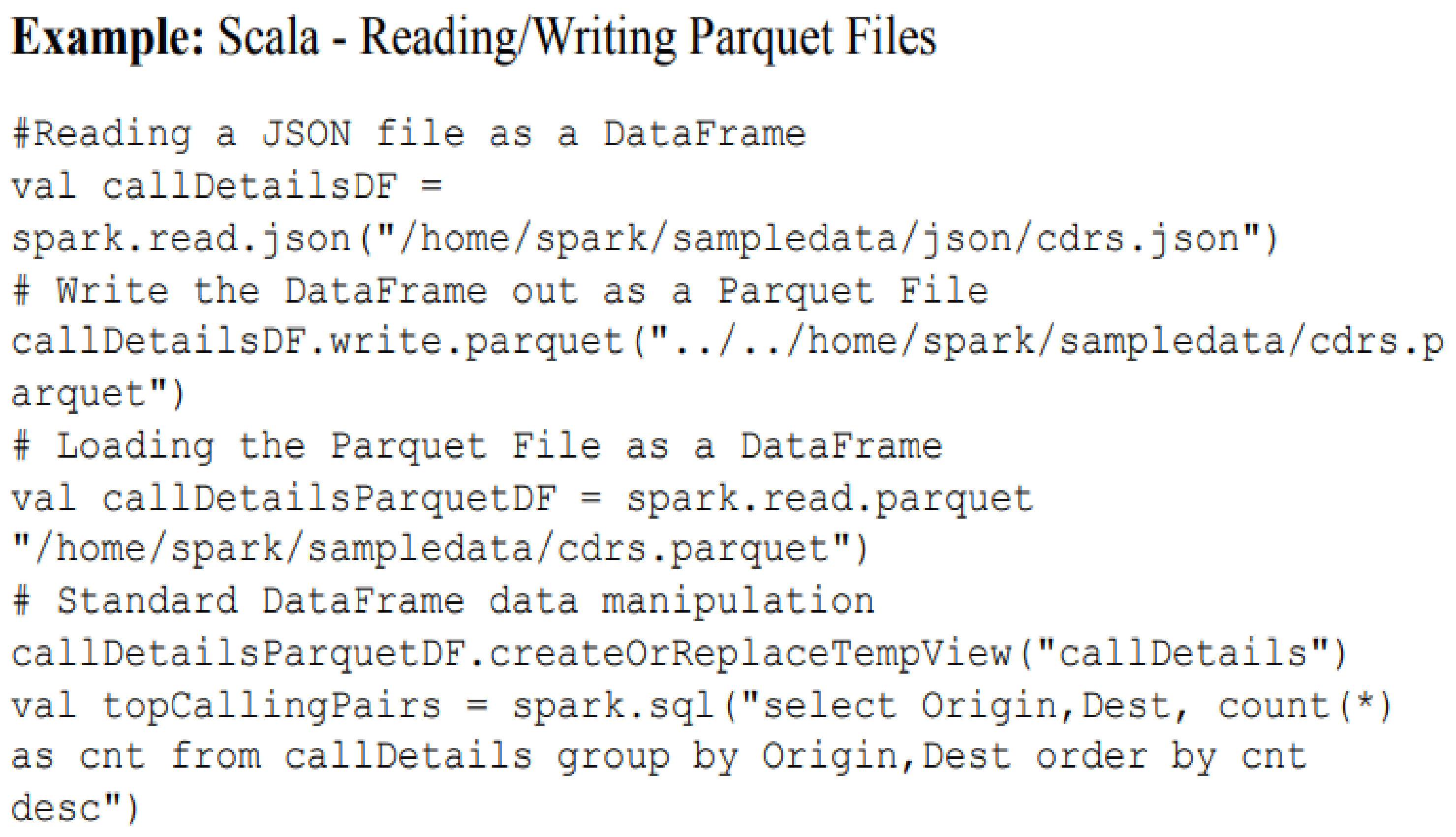
- **Schema evolution**

  When using columnar file formats like Parquet, users can start with a simple schema, and gradually add more columns to the schema as needed. In this way, users may end up with multiple Parquet files with different but mutually compatible schemas. In these cases, Parquet supports automatic schema merging among these files.

- **Open source and non-proprietary**

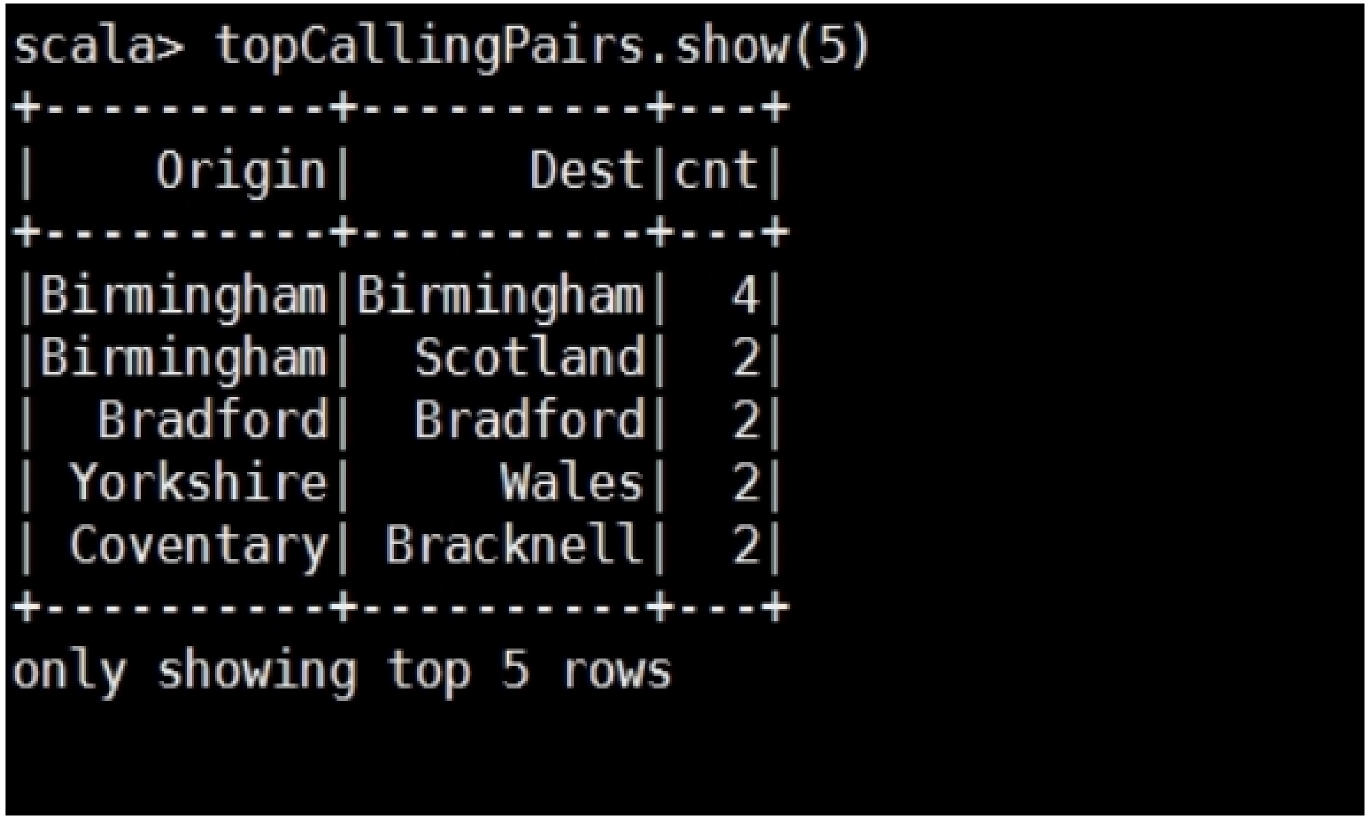
  Apache Parquet is part of the open-source Apache Hadoop ecosystem. Development efforts around it are active, and it is being constantly improved and maintained by a strong community of users and developers

- Reading/writing parquet files



Example: Scala - Reading/Writing Parquet Files

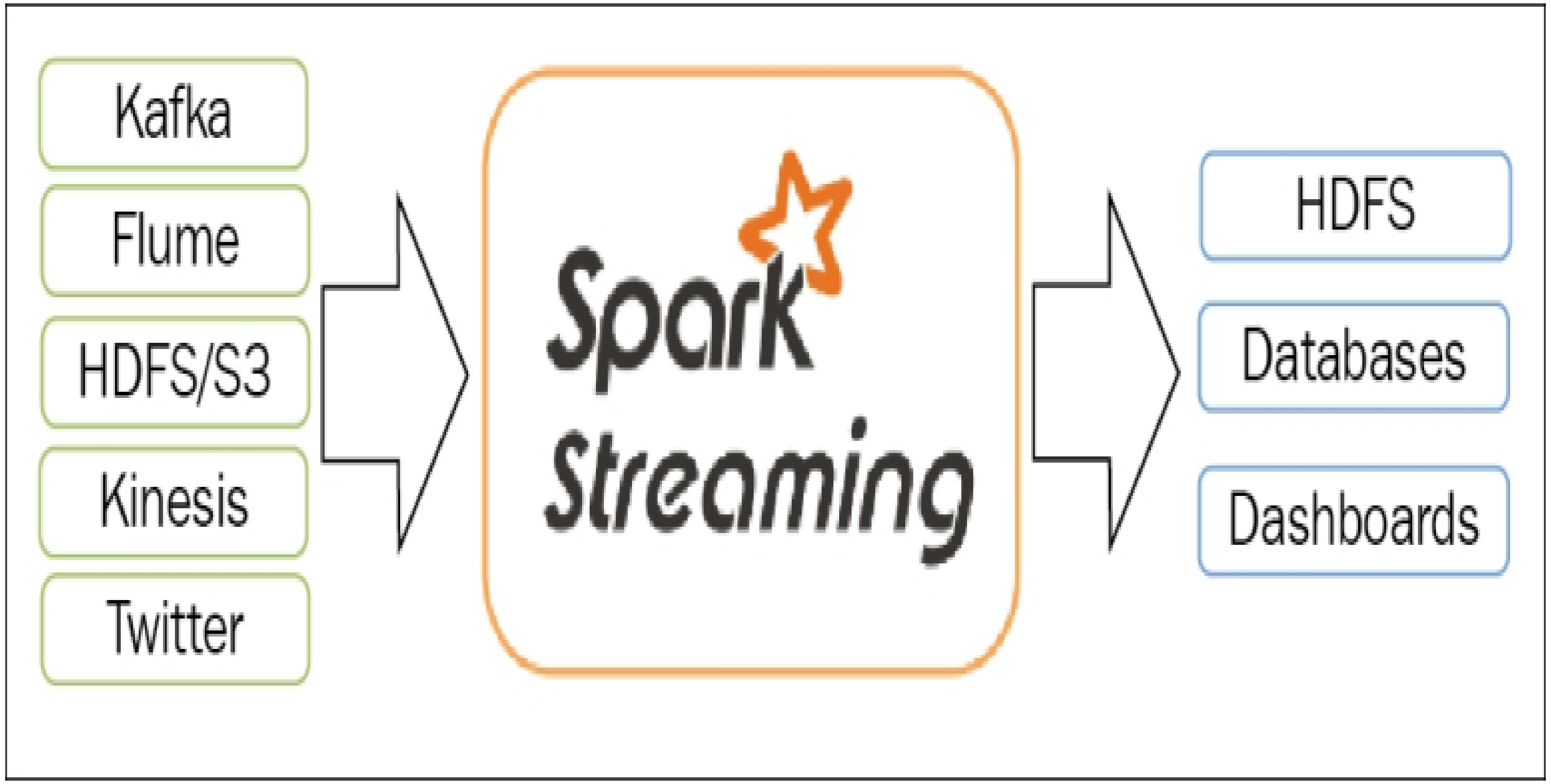
```
#Reading a JSON file as a DataFrame
val callDetailsDF =
spark.read.json("/home/spark/sampledata/json/cdrs.json")
# Write the DataFrame out as a Parquet File
callDetailsDF.write.parquet("../../home/spark/sampledata/cdrs.p
arquet")
# Loading the Parquet File as a DataFrame
val callDetailsParquetDF = spark.read.parquet
"/home/spark/sampledata/cdrs.parquet")
# Standard DataFrame data manipulation
callDetailsParquetDF.createOrReplaceTempView("callDetails")
val topCallingPairs = spark.sql("select Origin,Dest, count(*)
as cnt from callDetails group by Origin,Dest order by cnt
desc")
```



```
scala> topCallingPairs.show(5)
+----------+----------+---+
|    Origin|      Dest|cnt|
+----------+----------+---+
|Birmingham|Birmingham|  4|
|Birmingham|  Scotland|  2|
|  Bradford|  Bradford|  2|
| Yorkshire|     Wales|  2|
| Coventary| Bracknell|  2|
+----------+----------+---+
only showing top 5 rows
```

Spark streaming

- Spark Streaming is an API that provides you the ability to work with streams of data.
- A data stream is an unbounded sequence of data arriving continuously. Streaming divides continuously flowing input data into discrete units for further processing.
- **spark Streaming** was added to Apache Spark in 2013, an extension of the core Spark API that provides scalable, high-throughput and fault-tolerant stream processing of live data streams.
- Data ingestion can be done from many sources like Kafka, Apache flume, Amazon Kinesis or TCP sockets and processing can be done using complex algorithms that are expressed with high-level functions like map, reduce, join and window.
- Finally, processed data can be pushed out to filesystems, databases and live dashboards.
- Live input data streams is received and divided into batches by Spark streaming, these batches are then processed by the Spark engine to generate the final stream of results in batches.

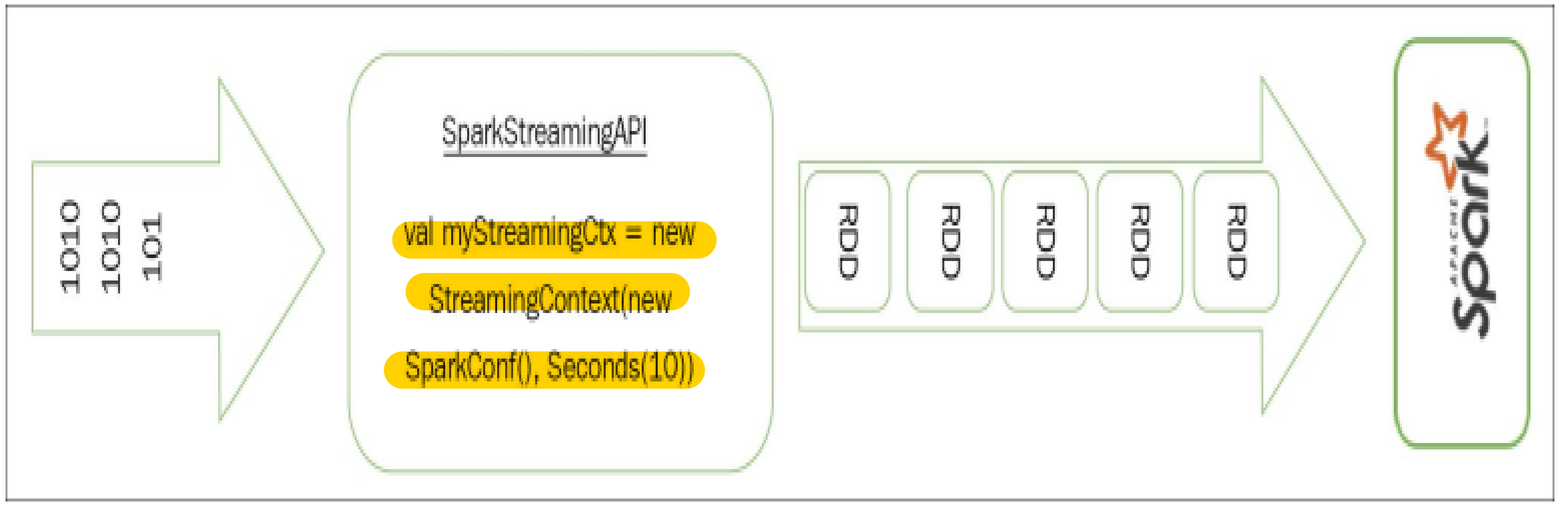

Kafka
Flume
HDFS/S3
Kinesis
Twitter
Spark Streaming
HDFS
Databases
Dashboards

- Spark provides two different programming models for streaming:
  - Spark Streaming as Discretized Streams (Dstream)
  - Structured Streaming

DStream

- Spark DStream, which represents a stream of data divided into small batches.
- DStream is essentially dividing a live stream of data into smaller batches of n seconds, and processing each individual batch as an RDD in Spark.
- DStreams are built on Spark RDDs. This allows Streaming in Spark to seamlessly integrate with any other Apache Spark components like Spark MLlib and Spark SQL.

- In the following figure, the StreamingContext uses a 10 second interval to join the data arriving and creating an RDD, which is passed onto Spark for further processing as per the user's program declarations.



```
val myStreamingCtx = new
StreamingContext(new
SparkConf(), Seconds(10))
```

Streaming context

- The StreamingContext is the main entrance point to Spark Streaming applications. The StreamingContext is configured the same way as SparkContext, but includes the additional parameter - the batch duration, which can be in milliseconds, seconds, or minutes.
- Spark streaming example:
- Let us cinsider an example whre we will use the netcat utility to write the data onto a network socket. Our streaming application will listen to the network socket to read the data from the socket, and perform a word count on the incoming data.

- Scala version of streaming example

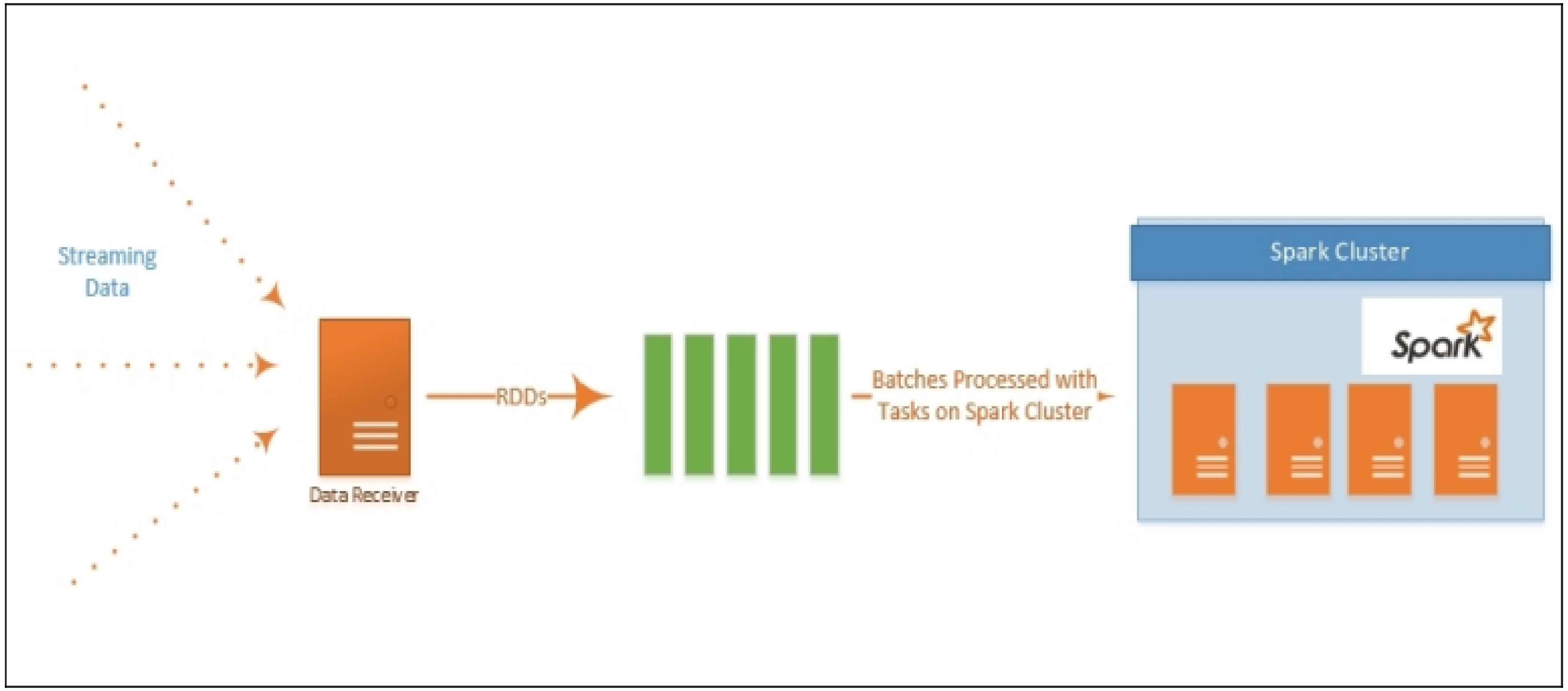
```
import org.apache.spark._
import org.apache.spark.streaming._
object StreamingWordCount {
 def main(args: Array[String]){
 // Create a streaming context - Donot use local[1] 
when
running locally
 val conf = new
SparkConf().setMaster("local[2]").
setAppName("NetworkWordCount")
 val ssc = new StreamingContext(conf, Seconds(5))
 // Create a DStream that connects to hostname:
port and
fetches
 information at the start of streaming.
 val lines = ssc.socketTextStream("localhost", 9988)
```

```
// Operate on the DStream, as you would
operate on a regular
stream
 val words = lines.flatMap(_.split(" "))
 // Count each word in each batch
 val pairs = words.map(word => (word, 1))
 val wordCounts = pairs.reduceByKey((x, y) => x +
y)

 // Print on the console
 wordCounts.print()
 ssc.start() // Start the computation
 ssc.awaitTermination() // Wait for the
computation to
terminate
 }
}
```

Architecture of Spark Streaming

- A distributed stream processing engine uses the following execution model:
  - Receive data from other data sources
  - Apply business logic
  - Once you have applied your business rules: You would potentially want to store the results in an external storage system (such as HBase, HDFS, or Cassandra).


Streaming Data
Data Receiver
RDDs
Batches Processed with Tasks on Spark Cluster
Spark Cluster
Spark

- Spark has adopted a distributed processing framework by discretizing the streaming data into micro-batches (based on your specified time-interval).
- The data is received in parallel by receivers before being buffered into the work node memory. Spark then performs the computation based on the locality info of the data available in the worker memory, enabling the architecture to load-balance and provide optimum fault recovery.
- In addition, each batch of data is a Resilient Distributed Dataset (RDD).Once data is available as an RDD, it can be processed as a traditional Spark RDD.

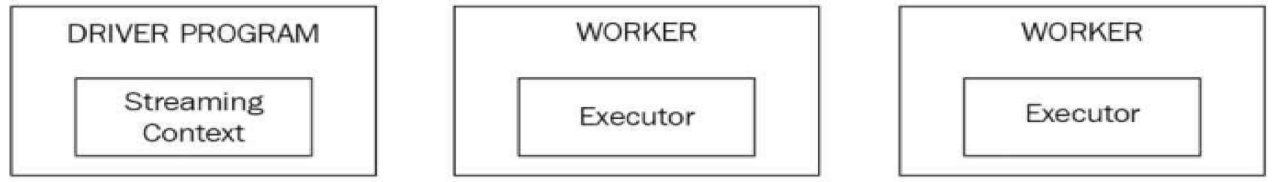
- Spark will split the incoming stream of data into micro-batches to allow for fine-grained allocations of computation to resources.
- For a simple job where a data stream is partitioned by key, the job's tasks will be naturally load balanced across the workers - some workers will process a few longer tasks, others will process more of the shorter tasks.
- Since the computation is divided into small tasks, failed nodes can be relaunched in parallel on all the other nodes in the cluster, thus evenly redistributing all the recomputations across many nodes and hence providing a swift recovery from any node failure.

- The following streaming job flow clearly shows the execution of SparkStreaming within components of the Spark framework:

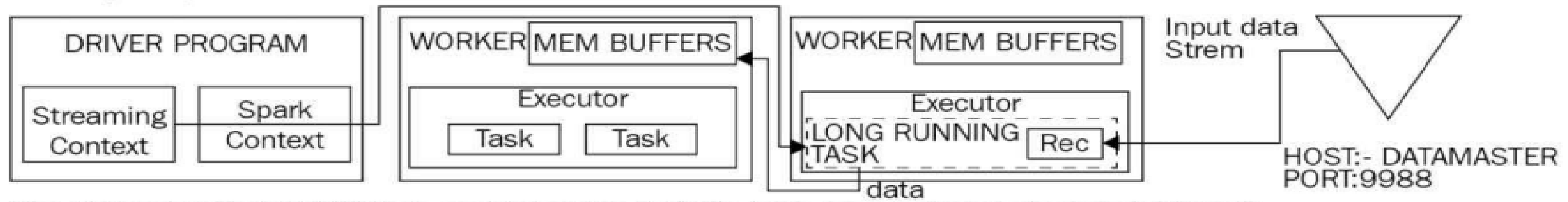
1. A streaming context launches receivers, which are tasks running within the Executors responsible for data collection from input source and saving it as RDDs. Remember that there are two types of receivers: reliable receivers, which send an acknowledgement to the reliable source when the data has been received and stored in Spark with replication, and unreliable receivers, which do not send any acknowledgement to the source.
2. The input data is cached in memory and also replicated to other executors for fault tolerance. By default, it is replicated across two nodes. Spark can therefore survive single node failures.
3. Streaming context would then run jobs based on the batch interval to process this data.
4. The output operations in the job then output the job in batches.

The flow of a Spark Streming job

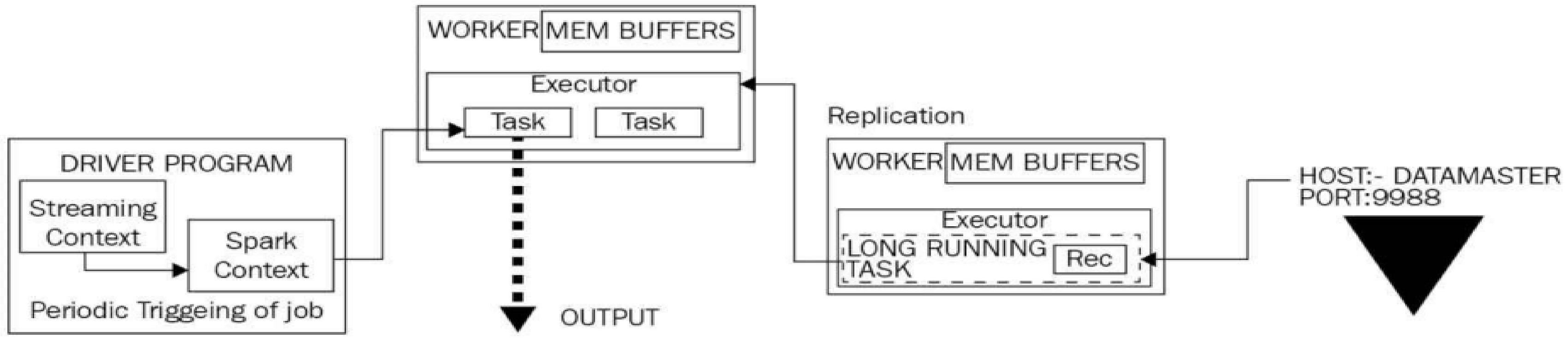
1. A Streaming Context is launched.



2. Using Streaming Context, the user indicates the intention to load data from one of the Streaming Sources e.g Socket Text Stream. Spark will launch receives inside the executor of worker nodes. Spark Streaming receivers will Accept data in parallel and buffer it in Memory of Sparks worker nodes.



3. The Streaming Context will then start jobs on a Periodic basis (depending on the batch internal) and to process the available data



let's look at the three major areas of Spark Streaming, which include:

- Input sources
- Transformations
- Output operations

Input sources

Core/basic sources :

- Spark has built-in support for a number of different sources and these are often referred to as core sources.
- In the very first example, we looked at the socket text stream, which is one of the popular sources; however, streaming context offers options to read data from network or file-based sources using operations such as: binaryRecordStream(), fileStream(), queueStream(), receiverStream(), socketStream(), and textFileStream()

Advanced sources :

In addition to the core sources, the strong eco-system around Spark provides support for an ever-growing list of data sources.These advanced sources include:

- Kafka
- Flume
- Kinesis

Transformations

DStreams transformations can be viewed as two distinct groups based on their state. The two groups can be stated as follows:

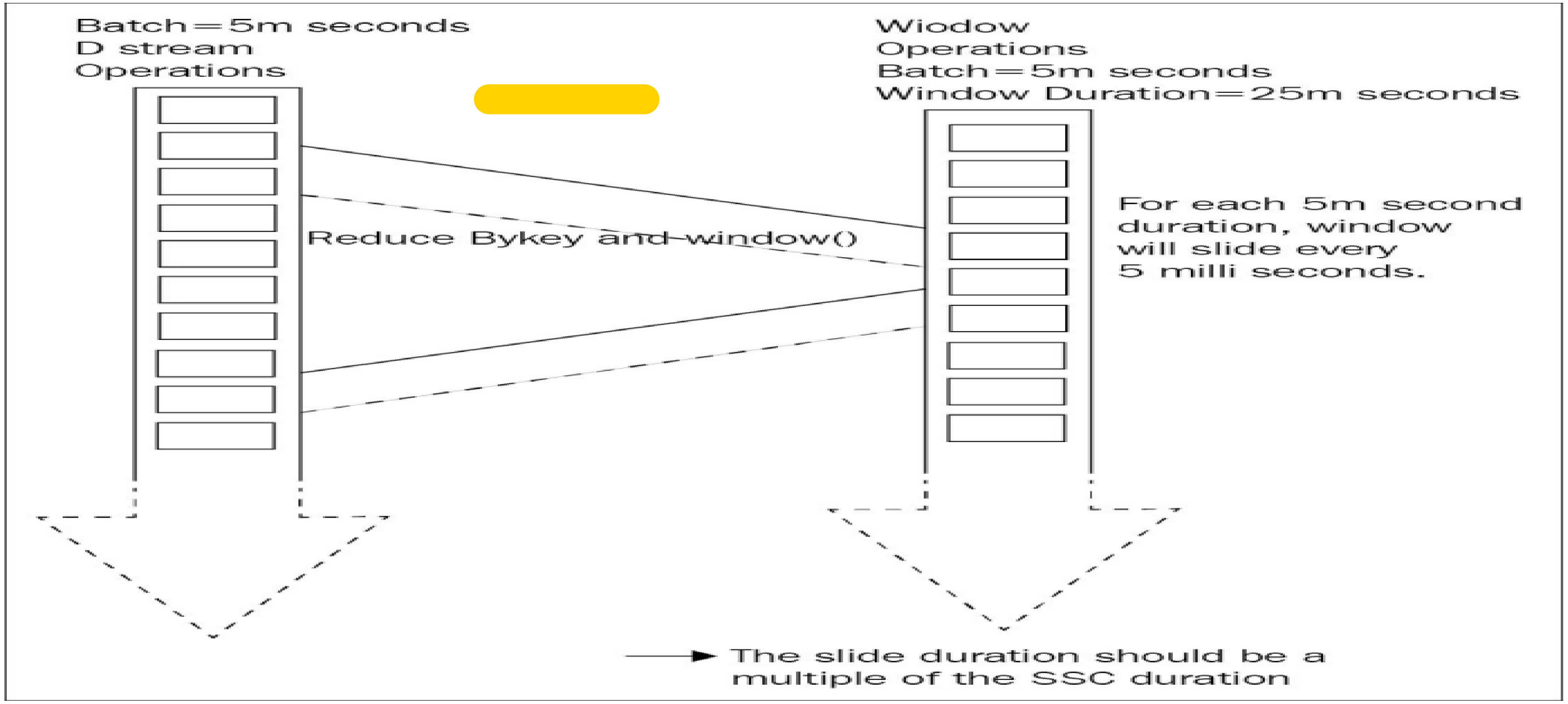
- Stateless transformations: are transformations that can be operated on individual batches without regard for the previous state of the batch.
- Each operation is applied to an individual RDD within the Dstream. For example, in our previous example, a DStream is composed of multiple RDDs, with each RDD representing a line of text being received from the socket connection, and each transformation is applied on an individual RDD rather than all the RDDs within the DStream.

- Stateful transformations:are transformations where the results from previous batches are used to compute the results of the current batch.
- For example, in a telecom marketing campaign you might want to see if the user has sent 50 SMS's over the past hour before you can make them an offer of 5 free SMS's or an upgrade to an SMS bundle.
- Stateful transformations include transformations that act over a sliding window of time periods and updateStateByKey(), which is used to track states across events for each key.

- Sliding window operations
- A sliding window is a popular technique when you want to operate on RDDs over a given duration compared to operating based on the duration given during the ssc configuration.
- As seen earlier, a batch interval basically means the interval after which the system generates RDDs based on the data received. For example, if you set the batch interval as 1 second, every 1 second, the system will generate RDDs from the received data.
- A window operator however is defined by two parameters rather than one, both of which are rather self explanatory:
  - Window length: Window length is the duration of the window .
  - Slide interval: Slide interval is the interval at which the window will slide or move forward
  - These two intervals must be a multiple of the batches internal of the source DStream

- Example:

Let's say you are collecting tweets about a particular topic, for example, your store's annual sale with the Hashtag #PPSaleOffer every 1 second, but you would like to know the sentiment of people reacting to your sale, and you would like to know the sentiment of people about the sale. You could use the sliding window functions to achieve that.


Batch=5m seconds
D stream
Operations
Wiodow
Operations
Batch=5m seconds
Window Duration=25m seconds
Reduce Bykey and window()
For each 5m second duration, window will slide every 5 milli seconds.
The slide duration should be a multiple of the SSC duration

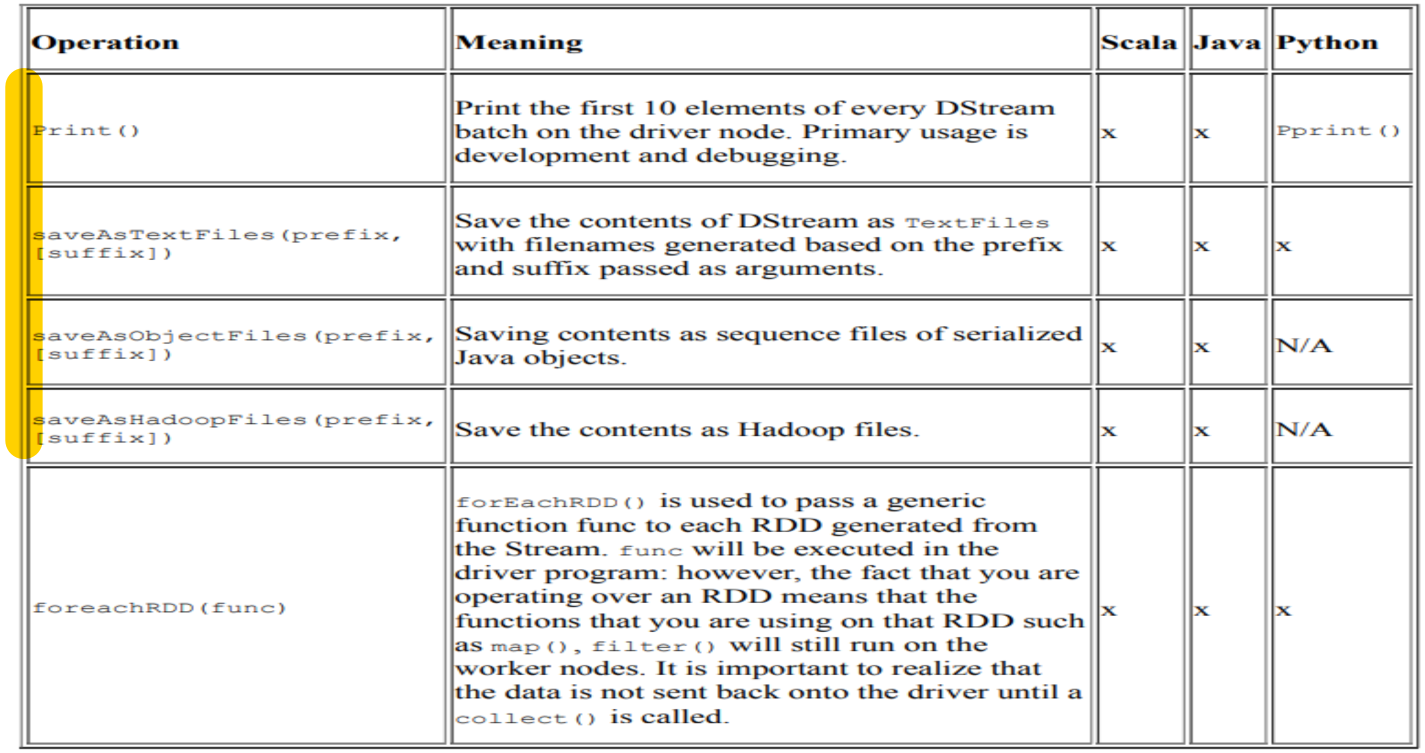
- The default behavior of window operations is to slide at the same interval as the ssc duration. Spark allows you to define any slide duration, as long as it is a multiple of your batch duration.
- The two main window operations include:
  - **countByWindow( )** : returns count of the elements in the stream.
  - **reduceByKeyAndWindow():** has two variants:
    - reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks]): As the name indicates this is essentially a reduce function, and the reduce function is passed in as a parameter. When the function is called on a discretized stream of key/value pairs, it returns a new discretized stream which returns the values aggregated using the func operating over the batches in the window defined by the windowLength and slideInterval.
    - reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks]): This is a rather more efficient version of reduceByKeyAndWindow() as the reduce is calculated incrementally using the output

Output operations

Output operations allow the data computed via transformations to be pushed out to external systems.

Output operations start the actual execution of DStream transformations just like actions within RDDs.

At the time of writing, the following output operations are supported:



Operation	Meaning	Scala	Java	Python
`Print()`	Print the first 10 elements of every DStream batch on the driver node. Primary usage is development and debugging.	x	x	`Pprint()`
`saveAsTextFiles(prefix, [suffix])`	Save the contents of DStream as `TextFiles` with filenames generated based on the prefix and suffix passed as arguments.	x	x	x
`saveAsObjectFiles(prefix, [suffix])`	Saving contents as sequence files of serialized Java objects.	x	x	N/A
`saveAsHadoopFiles(prefix, [suffix])`	Save the contents as Hadoop files.	x	x	N/A
`foreachRDD(func)`	`forEachRDD()` is used to pass a generic function func to each RDD generated from the Stream. `func` will be executed in the driver program: however, the fact that you are operating over an RDD means that the functions that you are using on that RDD such as `map()`, `filter()` will still run on the worker nodes. It is important to realize that the data is not sent back onto the driver until a `collect()` is called.	x	x	x

X = available in the API N/A = not available in the API

Caching and persistence

- Caching and persistence are two key areas that developers can use to improve performance of Spark applications.
- The persist() method on a DStream will persist all RDDs within the DStream in memory. This is especially useful if the computation happens multiple times on a Dstream and is especially true in window-based operations.
- Developers do not explicitly need to call a persist() on window-based operations and they are automatically persisted.

The difference between cache() and persist() are:

- cache(): Persists the RDDs of the DStream with the default storage level (MEMORY_ONLY_SER). Cache() under the hood and calls the persist() method with the default storage level.
- persist() is overloaded with two variants:
  - You can either use persist() without parameters which will persist the RDDs of this DStream with the default storage level.
  - You can also use perisist(level: StorageLevel), which can persist the RDDs of the DStream with the given storage level.
  - You cannot change the storage level of the DStream after the streaming context has been started.

Checkpointing

- Spark Streaming provides the option to checkpoint information to a fault-tolerant storage system so that it can recover from failures.
- Checkpointing consist of two major types:
  - Metadata checkpointing
  - Data checkpointing
- Metadata checkpointing:

  Metadata checkpoint is essential if you would like to recover from driver program failures.

  A typical application metadata will include the following:

  1. Configuration: The initial configuration used to define the application.
  2. DStream operations: The operations that constitute the particular DStream.
  3. Incomplete batches: Currently running batches whose jobs are queued, but have not completed yet.

- Data checkpointing:
  - Data checkpointing is necessary when you would want to recover from failures of stateful transformations.
  - RDD’s are periodically checkpointed to reliable storage (for example, HDFS) to cut off the dependency chains.

Setting up checkpointing

- To configure checkpointing, you will need to set up a checkpoint location on a reliable and fault-tolerant filesystem such as HDFS or S3.
- Steps:
  - First run the program.
  - A new streaming context is set up, and then a start() call is made.
  - Restart from failure.
  - Streaming context is recreated from the checkpoint data in the checkpoint director.

Setting up checkpointing with Python

Let us look into an example of setting up checkpointing using Python:

```
# Function to create and setup a new StreamingContext
def createStreamingContext():
ssc = new StreamingContext(...)
lines = ssc.socketTextStream(...) # Create DStreams from your
preferred data Stream, in this case a socket connection
...
ssc.checkpoint(checkpointDirectory) # set checkpoint directory
return ssc
# Will get a streaming context from checkpoint if restarted
# Will get a new Streaming context if first time.
streamingCtx =
StreamingContext.getOrCreate(checkpointDirectory,
functionToCreateContext)
# Additional stream setup operations
streamingCtx. ...
# Start the context
streamingCtx.start()
streamingCtx.awaitTermination()
```

Fault tolerance

- Fault refers to failure, thus fault tolerance in Apache Spark is the capability to operate and to recover loss after a failure occurs.
- If we want our system to be fault tolerant, it should be redundant because we require a redundant component to obtain the lost data.
- In a distributed publish-subscribe messaging system, the computers that make up the system can always fail independently of one another.
- Depending on the action the producer takes to handle such a failure, you can get different semantics.

- **At-least-once semantics**: If the producer receives an acknowledgement (ack) it means that the message has been written exactly once . However, if a producer ack times out or receives an error, it might retry sending the message assuming that the message was not written. If the receiver had failed right before it sent the ack but after the message was successfully written, this retry leads to the message being written twice and hence delivered more than once to the end consumer.
- **At-most-once semantics**: If the receiver is configured as unreliable, then there will not be any acknowledgement to source/producer. In this case there is no duplication of input data received.
- **exactly once:** Most difficult to implement. Requires cooperation.

Fault tolerance

In a streaming application there are typically three types of guarantees available, as follows:

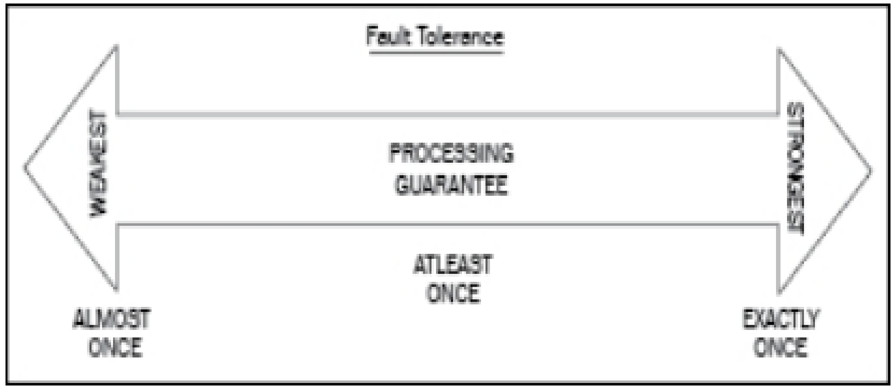


Figure 5.11: Typical guarantees offered by a streaming application

- In a streaming application, which generally comprises of data receivers, transformers, and components, producing different output failures can happen.

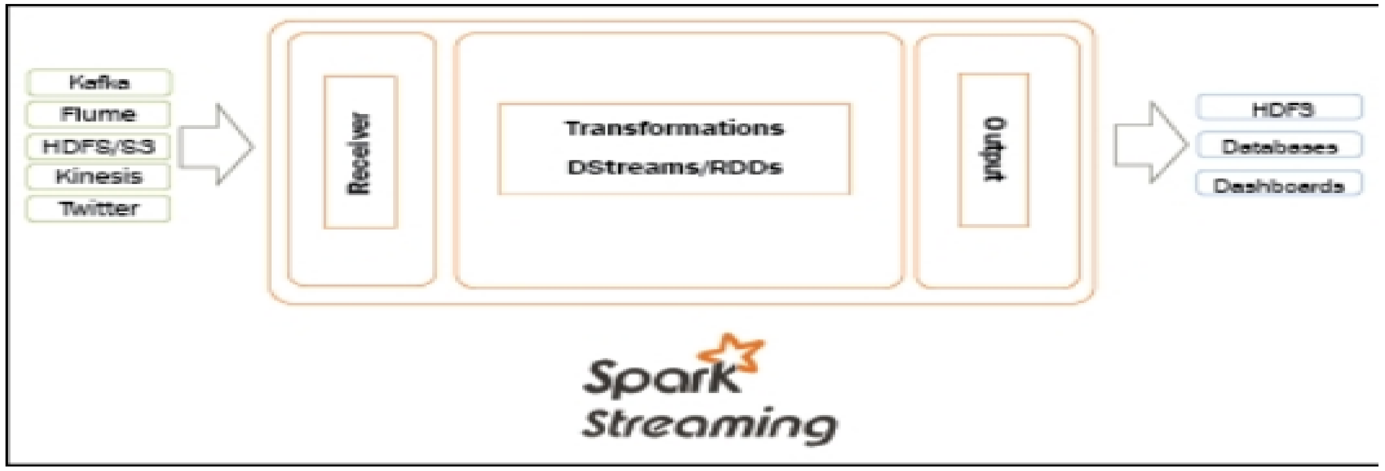


Figure 5.12: Components of a streaming application

worker failure impact on receivers

- When a Spark worker fails, it can impact the receiver that might be in the midst of reading data from a source.
- Suppose you are working with a source that can be either a reliable filesystem or a messaging system such as Kafka/Flume, and the worker running the receiver responsible for getting the data from the system and replicating it within the cluster dies.
- Spark has the ability to recover from failed receivers, but its ability depends on the type of data source and can range from at least once to exactly once semantics.

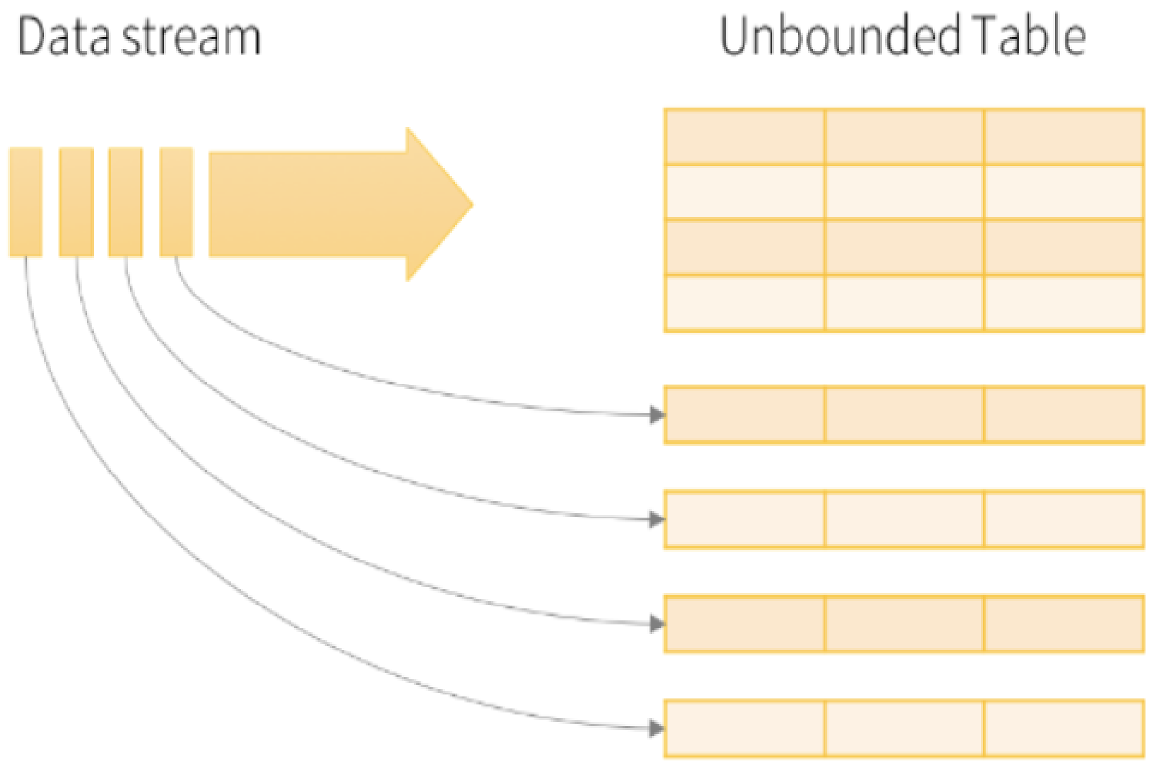
- However, if data is based on reliable sources, the fault-tolerance mechanics depend on the type of receiver and the kind of failure.
- In the event of a reliable receiver, generally an acknowledgement is sent after the replication to other nodes, and in case the failure happens before replication has occurred, no acknowledgement will be sent. This will result in the data being sent to the receiver again, and hence no data loss. This type of receiver provides at least once semantics.
- In the event of an unreliable receiver, the data that has not yet been replicated will be lost.
- **Failure of driver node –** If there is a failure of the driver node that is running the Spark Streaming application, then SparkContent losses and all executors lose their in-memory data.

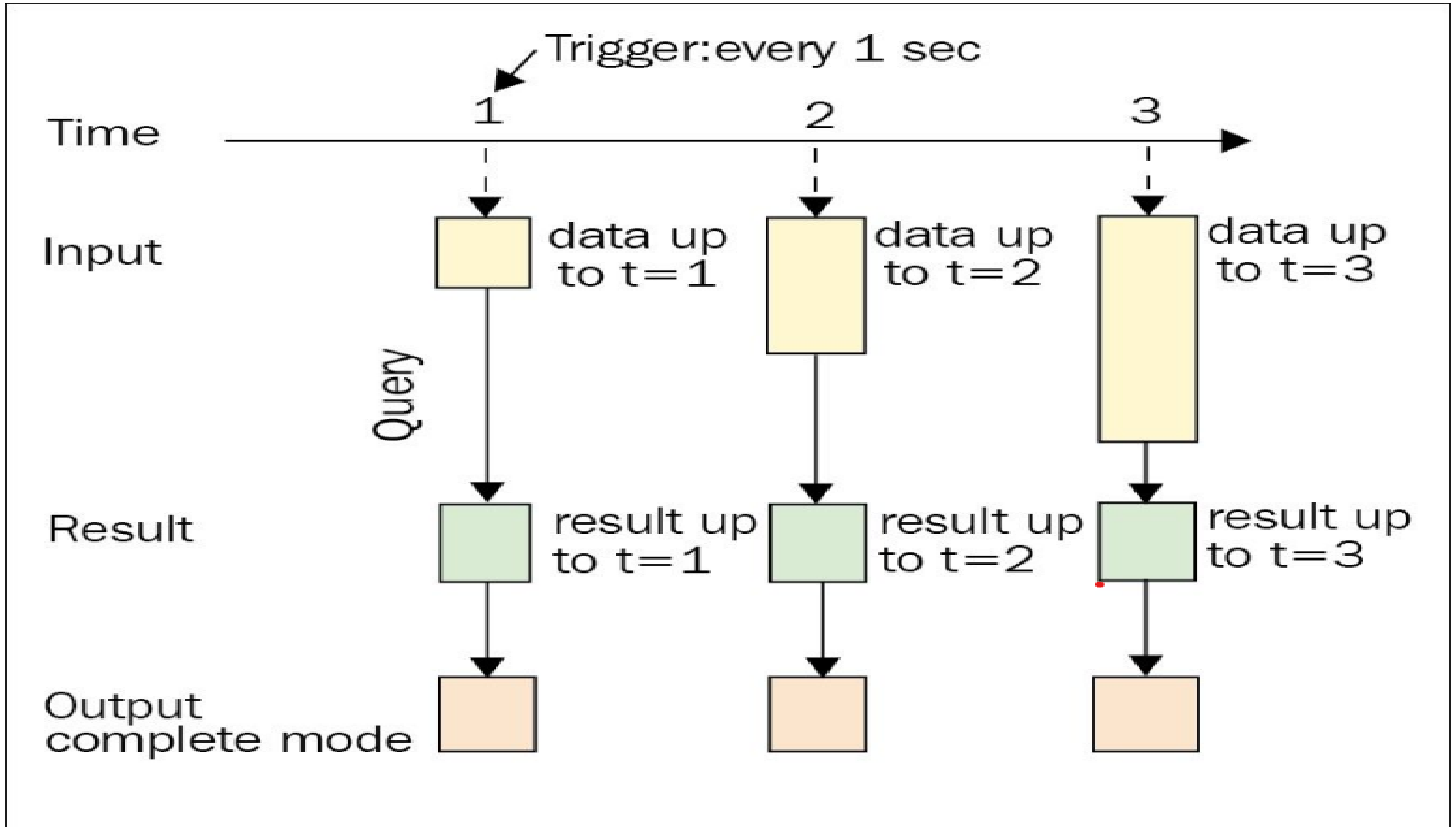
- **Worker failure impact on RDDs/DStreams**
  - RDDs and DStreams provide semantics exactly once. We know that RDD can recompute from its lineage; as long as the input data is available there is no loss of data.
- **Worker failure impact on output operations**
  - In the event of a worker failure, the output operations by default provide semantics at least once, as the operation that saves data to external systems might execute more than once.
  - however, if you want to achieve semantics exactly once you will need to engineer the system to make use of transactions, so that each update gets pushed only once.
  - Spark's saveAs****File() operations automatically take care of multiple updates by ensuring that only one copy of the output file exists.

what is Structured Streaming?

- The following issues with Dstreams lead to introduction of structured streaming.
  - DStreams can work with the batch time, but not the event time inside the data.
  - The Streaming API was different to RDD API in the sense that you cannot take a Batch job and start running it as a streaming application.
  - To handle fault tolerance especially for drivers using the checkpointing or output operations and make sure that you either use transactions or idempotent updates to achieve semantics exactly once.
- In Structured Streaming, a data stream is treated as a table that is being continuously appended.
- This leads to a stream processing model that is very similar to a batch processing model. You express your streaming computation as a standard batch-like query as on a static table, but Spark runs it as an incremental query on the unbounded input table.

Consider the input data stream as the input table. Every data item that is arriving on the stream is like a new row being appended to the input table.





Programming Model for Structured Streaming

- A query on the input generates a result table. At every trigger interval (say, every 1 second), new rows are appended to the input table, which eventually updates the result table.
- Whenever the result table is updated, the changed result rows are written to an external sink. The output is defined as what gets written to external storage.
- The output can be configured in different modes:
- **Complete Mode**: The entire updated result table is written to external storage. It is up to the storage connector to decide how to handle the writing of the entire table.
- **Append Mode**: Only new rows appended in the result table since the last trigger are written to external storage. This is applicable only for the queries where existing rows in the Result Table are not expected to change.
- **Update Mode**: Only the rows that were updated in the result table since the last trigger are written to external storage. This is different from Complete Mode in that Update Mode outputs only the rows that have changed since the last trigger. If the query doesn't contain aggregations, it is equivalent to Append mode.

Output sinks

- In Spark Streaming, output sinks store results into external storage. There are a few types of built-in output sinks.
  - File sink: Stores the contents of a DataFrame in a file within a directory. Supported file formats are csv, json, and parquet.
  - **Foreach sink**: Applies to each row of a DataFrame and can be used when writing custom logic to store data.
  - Console sink: Displays the content of the DataFrame to console. Very useful for debugging, as it prints everything to the console.
  - Memory Sink: Memory sink stores the output in memory on the driver side in an in-memory table providing both append and complete modes: