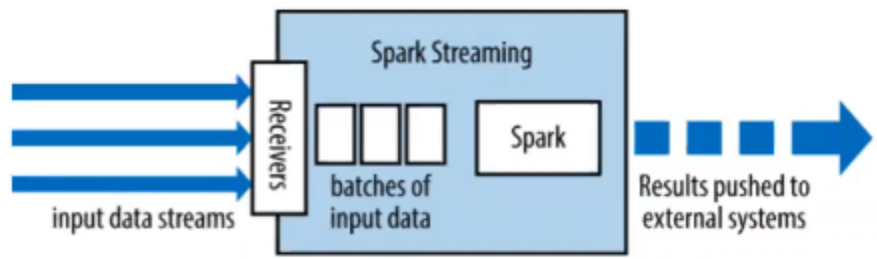
Spark Structured Streaming

Streaming Data Examples

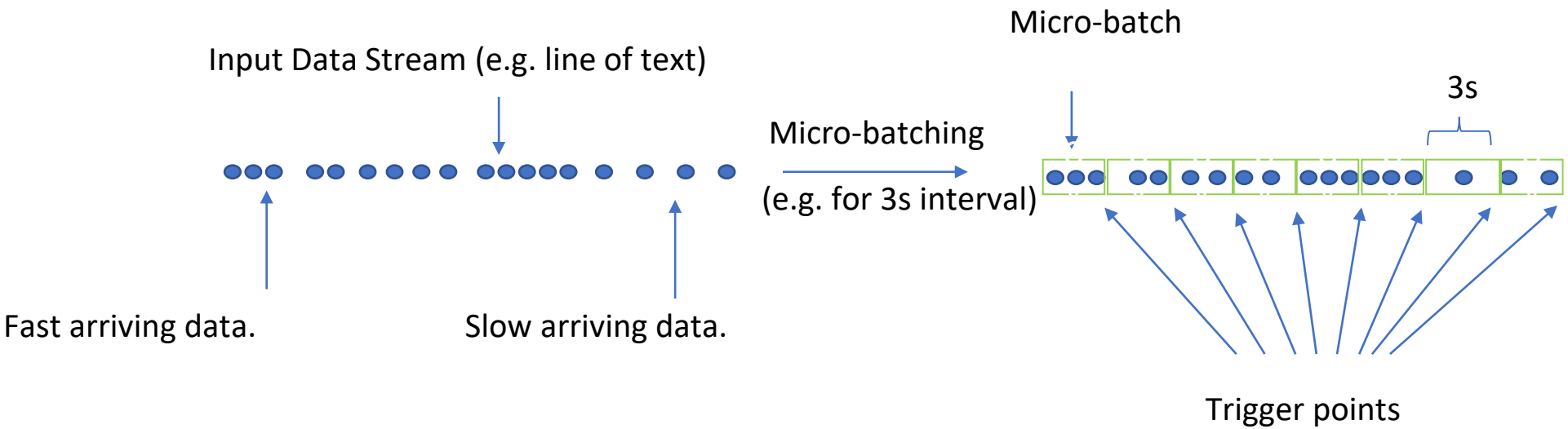
- Sensors in transportation vehicles, industrial equipment, and farm machinery send data to a streaming application. The application monitors performance, detects any potential defects in advance, and places a spare part order automatically preventing equipment down time.
- A financial institution tracks changes in the stock market in real time, computes value-at-risk, and automatically rebalances portfolios based on stock price movements.
- A media publisher streams billions of clickstream records from its online properties, aggregates and enriches the data with demographic information about users, and optimizes content placement on its site, delivering relevancy and better experience to its audience.

Spark Streaming High Level Architecture

- Streaming Data is data that is generated continuously by one or more data sources (sometimes thousands), repeatedly, typically at intervals of seconds to ms.
- This data needs to be processed sequentially and incrementally on a record-by-record basis or in batches, and used for a wide variety of analytics.
- Systems which process streaming data are called streaming data systems.



Spark Streaming - Micro Batches



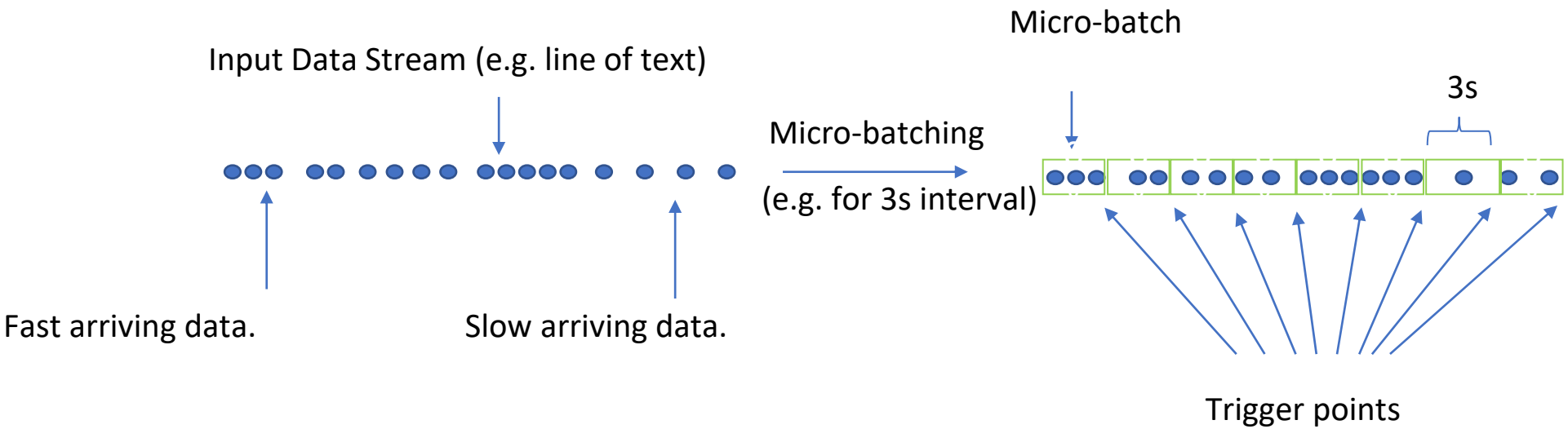
Micro Batch: A group of streaming records on which analysis is done..

Trigger Points: A Trigger point is a point when Spark Streaming processes all the data (micro-batch) received since the last trigger point.

Batch Interval: Time between two triggers – 3s in the above example.

Spark Streaming processes one micro-batch at a time (and not one unit of streaming data).

Spark Streaming - Micro Batches



Micro Batch: A group of streaming records on which analysis is done..

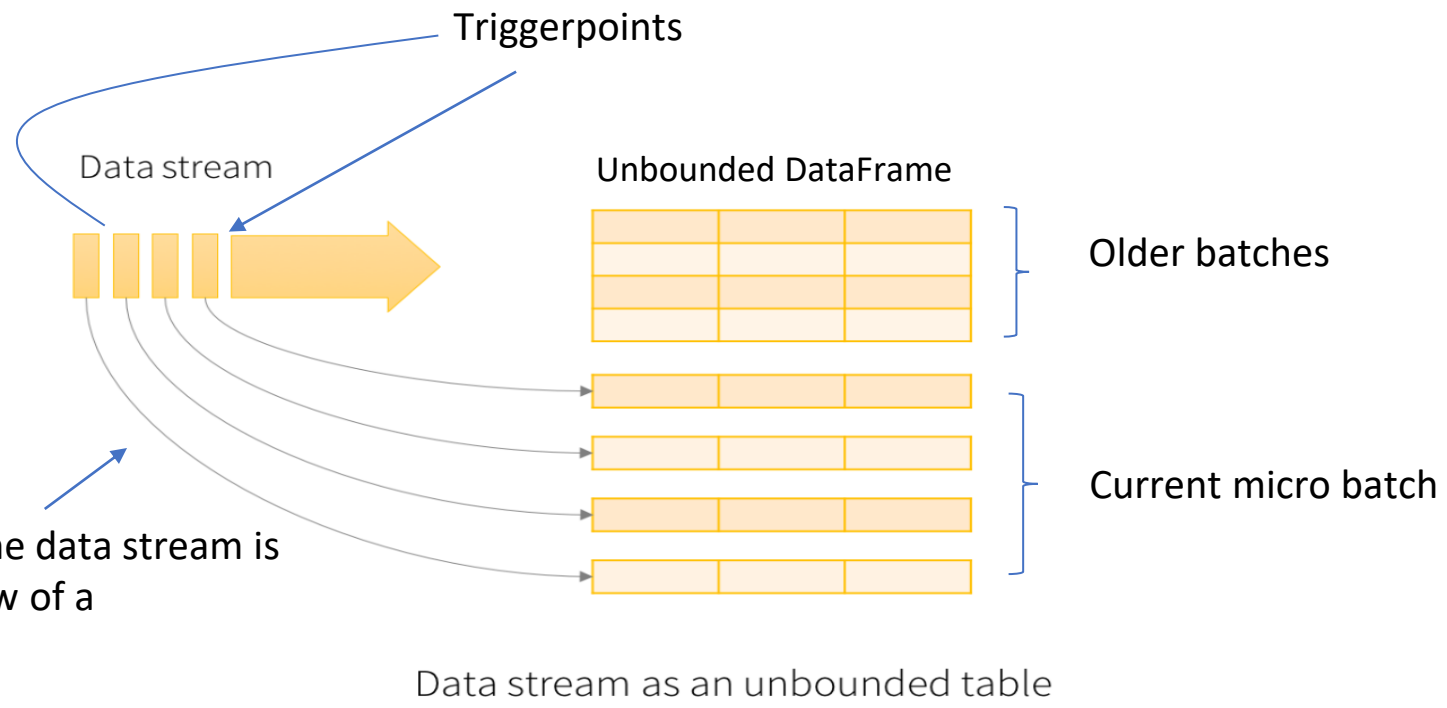
Trigger Points: A Trigger point is a point when Spark Streaming processes all the data (micro-batch) received since the last trigger point.

Batch Interval: Time between two triggers – 3s in the above example.

Spark Streaming processes one micro-batch at a time (and not one unit of streaming data).

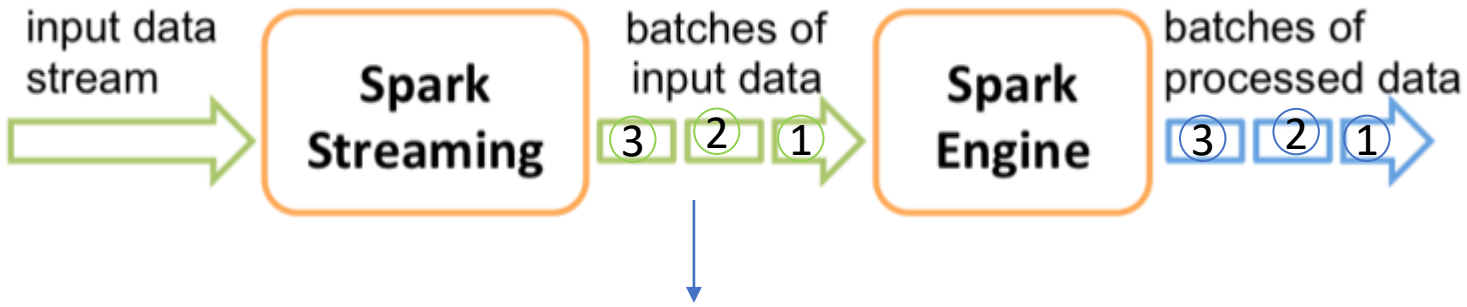
Spark Structured Streaming

Structured Streaming: Treats each unit of stream data as a record or row of a DataFrame.



A Dstream data structure (micro batch as a RDD instead of DataFrame) is also available in Spark Streaming.

Stream Processing in Spark



(Micro) batches can be RDD's or DataFrame's. We will discuss DataFrame's only.

Programming using Dstream

1. Initialize the streaming context

```
import org.apache.spark._
import org.apache.spark.streaming._
val conf = new SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, Seconds(1)
```

2.Define the input sources by creating input DStreams.
3. Define the streaming computations by applying transformation and output operations to DStreams.
4.Start receiving data and processing it using ssc.start().
5.Wait for the processing to be stopped (manually or due to any error) using ssc.awaitTermination().
6.The processing can be manually stopped using ssc.stop().

Transformation on Dstream

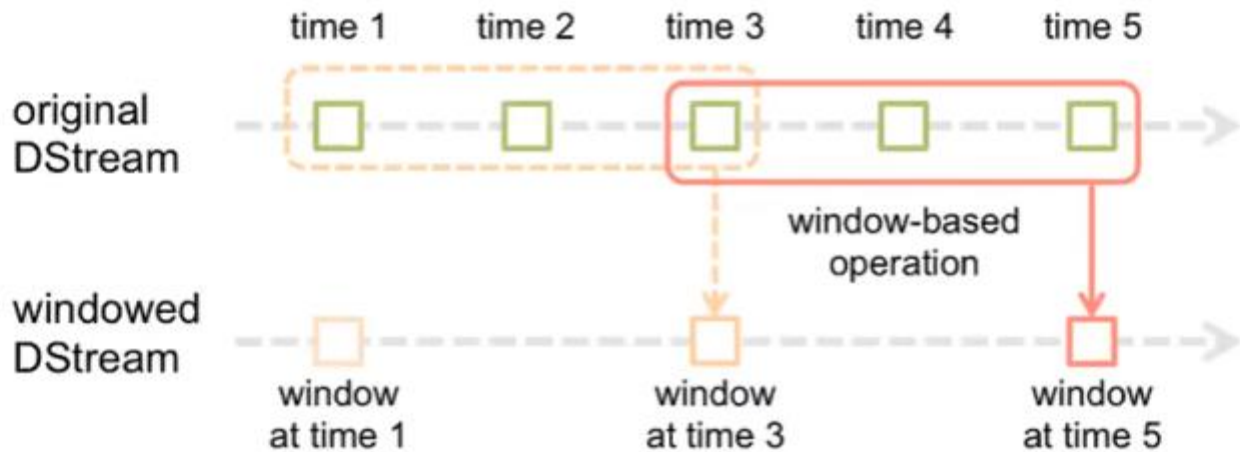
Categorized into 2 types:

1. **Stateless transformations** the processing of each batch does not depend on the data of its previous batches. They include the common RDD transformations , like map(), filter(), and reduceByKey().

2. **Stateful transformations**, in contrast, use data or intermediate results from previous batches to compute the results of the current batch. They include transformations based on sliding windows and on tracking state across time.

Window Operation

- Allow you to apply transformations over a sliding window of data.



- Every time the window *slides* over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream.

Any window operation needs to specify two parameters.

- *window length* - The duration of the window
- *sliding interval* - The interval at which the window operation is performed

Transformations on DStreams

Transformation	Meaning
map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items.
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.
repartition(*numPartitions*)	Changes the level of parallelism in this DStream by creating more or fewer partitions.
union(*otherStream*)	Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
reduce(*func*)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative and commutative so that it can be computed in parallel.
countByValue()	When called on a DStream of elements of type K, return a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
reduceByKey(*func*, [*numTasks*])	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function. **Note:** By default, this uses Spark's default number of parallel tasks (2 for local mode, and in cluster mode the number is determined by the config property `spark.default.parallelism`) to do the grouping. You can pass an optional `numTasks` argument to set a different number of tasks.
join(*otherStream*, [*numTasks*])	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.
cogroup(*otherStream*, [*numTasks*])	When called on a DStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples.

Output Operations on DStreams

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.
saveAsTextFiles(*prefix*, [*suffix*])	Save this DStream's contents as text files.
saveAsObjectFiles(*prefix*, [*suffix*])	Save this DStream's contents as SequenceFiles of serialized Java objects.
saveAsHadoopFiles(*prefix*, [*suffix*])	Save this DStream's contents as Hadoop files.
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. 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.

Caching / Persistence

- DStreams also allow developers to persist the stream's data in memory
- Using **persist()** method Dstream can automatically persist every RDD of that Dstream in memory
- Dstream generated by window based operations are automatically persisted in memory without using persist()

Checkpointing

```
aggDF
  .writeStream
  .outputMode("complete")
  .option("checkpointLocation", "path/to/HDFS/dir")
  .format("memory")
   .start()
```

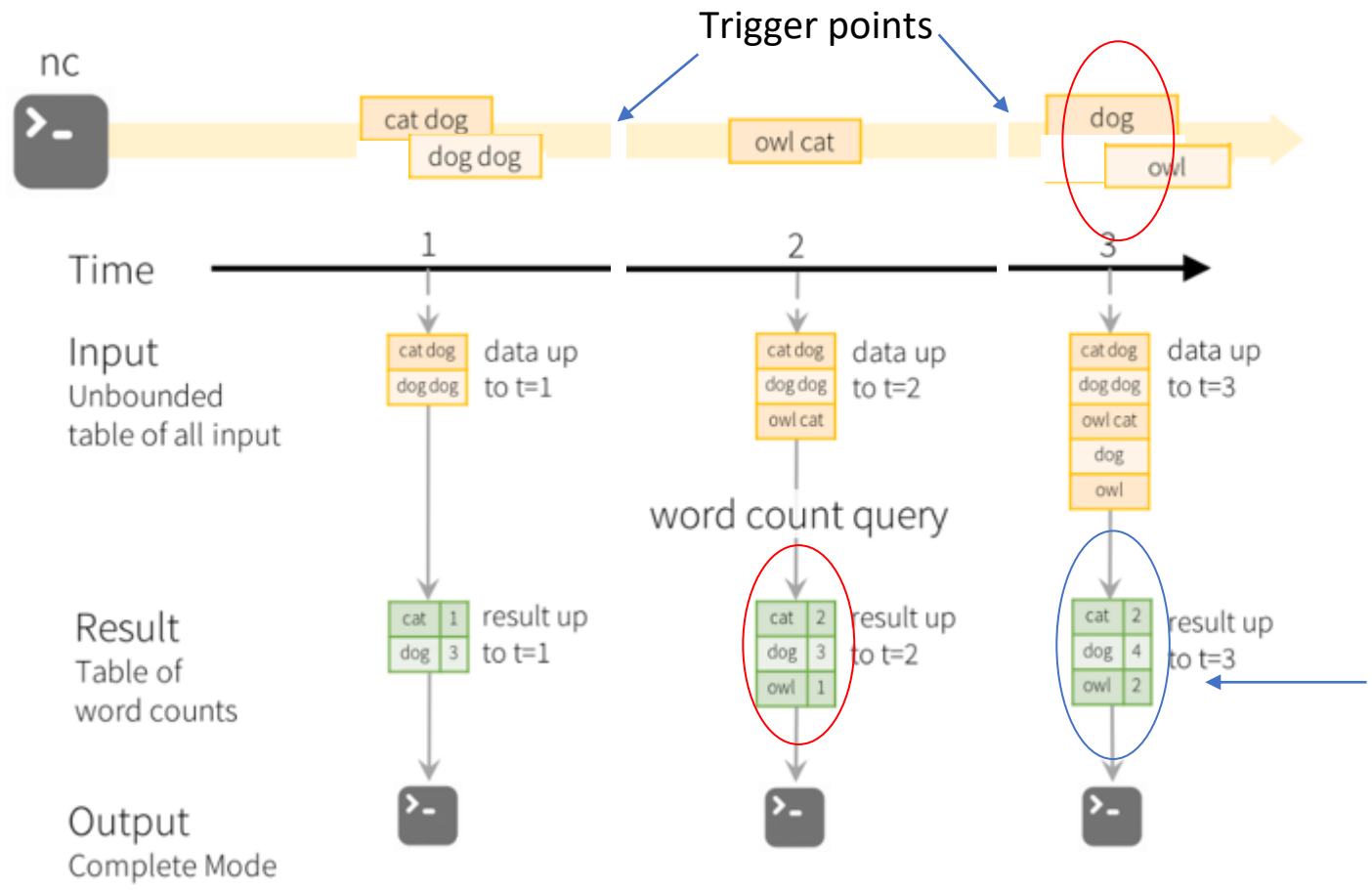
- Need to set up for fault tolerance
- It allows Spark Streaming to periodically save data about the application to a reliable storage system
- checkpointing serves two purposes:
  - Limiting the state that must be recomputed on failure. Spark Streaming can recompute state using the lineage graph of transformations, but checkpointing controls how far back it must go.
  - Providing fault tolerance for the driver. If the driver program in a streaming application crashes, you can launch it again and tell it to recover from a checkpoint.

Performance Tuning

- The parameters and configurations that can be tuned to improve the performance of you application. At a high level, you need to consider two things:
  - Reducing the processing time of each batch of data by efficiently using cluster resources.
  - Setting the right batch size such that the batches of data can be processed as fast as they are received (that is, data processing keeps up with the data ingestion).

Word Count Stream Processing

Example: Given an input word stream, print word count of words seen.



What data is required to compute output for batch i ?

Result of batch (i-1) and input of batch i required.

Cores / Threads required for Stream Processing

- It is possible to have multiple parallel data streams.
- Number of cores should be atleast 1 greater than number of input streams.

Streaming vs Batch Processing

	Batch processing	Stream processing
Data scope	Queries or processing over all or most of the data in the dataset.	Queries or processing over data within a rolling time window, or on just the most recent data record.
Data size	Large batches of data.	Individual records or micro batches consisting of a few records.
Performance	Latencies in minutes to hours.	Requires latency in the order of seconds or milliseconds.
Analyses	Complex analytics.	Simple response functions, aggregates, and rolling metrics.

Word Count Streaming Code

```scala
object StructuredNetworkWordCount {
  def main(args: Array[String]) {
    if (args.length < 2) {
      System.err.println("Usage: StructuredNetworkWordCount <hostname> <port>")
      System.exit(1)
    }

    val host = args(0)
    val port = args(1).toInt

    val spark = SparkSession.builder.appName("StructuredNetworkWordCount").getOrCreate()

    import spark.implicits._

    // Create DataFrame representing the stream of input lines from connection to localhost:9999
    val lines = spark.
      readStream.
      format("socket").option("host", host).option("port", port).load()

    // Split the lines into words
    val words = lines.as[String].flatMap(_.split(" ")).filter(!_.isEmpty)

    // Generate running word count
    val wordCounts = words.groupBy("value").count()

    // Start running the query that prints the running counts to the console
    val query = wordCounts.writeStream.outputMode("complete").format("console").
      trigger(Trigger.ProcessingTime("5 seconds")).start()

    query.awaitTermination()
  }
}
```

1. Need SparkSession to access streaming functionality.
2. Connect to a input stream. In this case to port “9999” on localhost. readStream creates a DataFrame. Hence, “lines” is a DataFrame.
3. Processing for streaming data. “wordCounts” contains the aggregation and is the “result” for a batch.
4. Start the streaming computations.
5. Write output to console.
6. Continue until termination signal received.

Processing Time Trigger

```
// Start running the query that prints the running counts to the console
val query = wordCounts.writeStream.outputMode("complete").format("console").
  trigger(Trigger.ProcessingTime("5 seconds")).outputMode("complete").start()
```

Trigger determines the batch interval.

```
subhrajit@subhrajit-VirtualBox:~/sparkProjects/streamProcessing2$ nc -lk 9999
apache spark
```

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

18/09/01 14:06:46 WARN ProcessingTimeExecutor: Current batch is falling behind.
The trigger interval is 5000 milliseconds, but spent 16602 milliseconds
```

Indicates that batch interval (5 s) is less than time taken to process batch (16 s).

Setting the Batch Interval

- It batch interval > batch processing time:
  - reduce batch interval.
  - smaller batch interval implies faster stream processing, i.e. more real time response.
- If batch interval < batch processing time:
  - Try to reduce batch processing time by optimizing the computations or increasing number of threads / cores available.
  - Increase batch interval, but it will imply slower stream processing, i.e. less real time response.

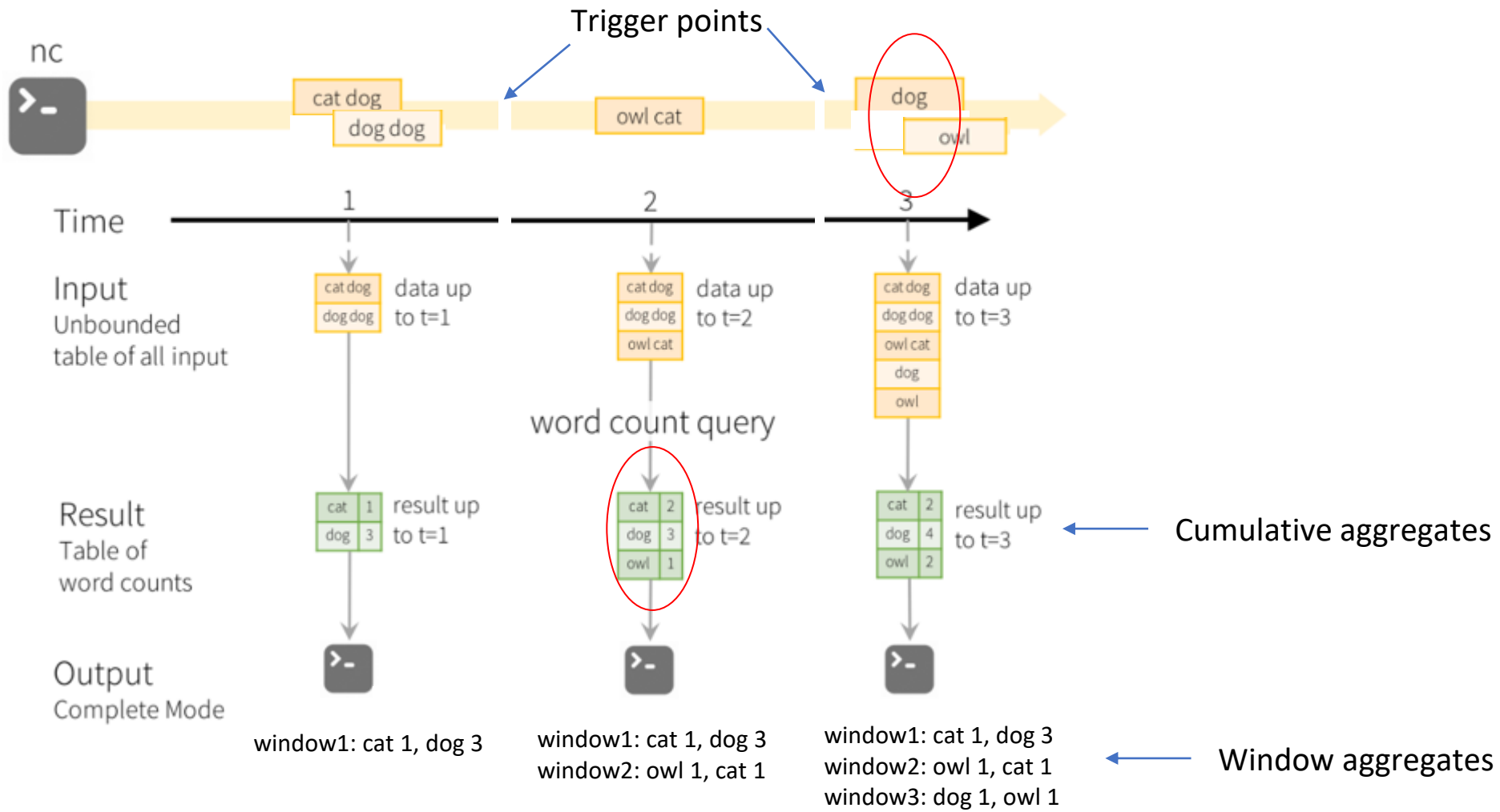
Optimizing batch interval is beyond the scope of this course.

Review Questions 1

1. What is the difference between output modes “complete”, “append” and “update” ? What are restrictions if any on the output modes. Check the following information source: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
2. Run the code for streaming word count for the above output modes and explain your observations.

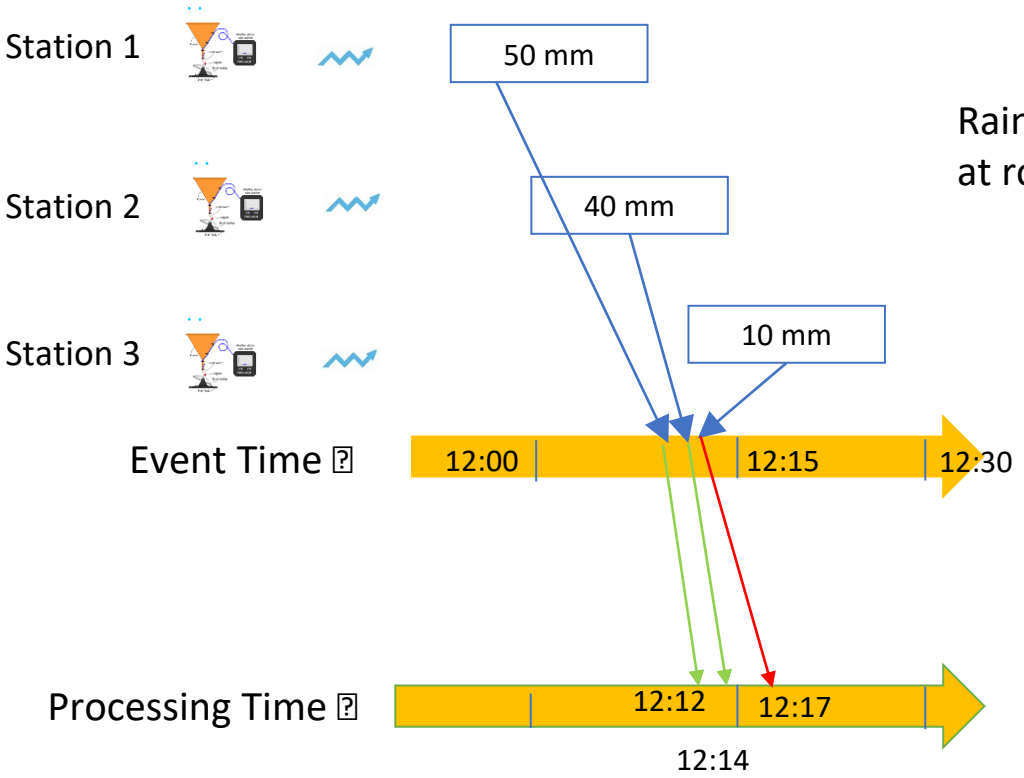
Stream Word Counting: Cumulative versus Window Based

Example: Given an input word stream, print word count of words seen.



Time Windows: Event Time versus Processing Time

Rainfall measured in Bangalore every 15 minutes at 3 locations. Need to report averages for every 15 minute interval.



Rainfall events measured at roughly the same time.

Streaming events arrive with some (network) delay at data center.

Event time window is natural and preferred way of reporting.

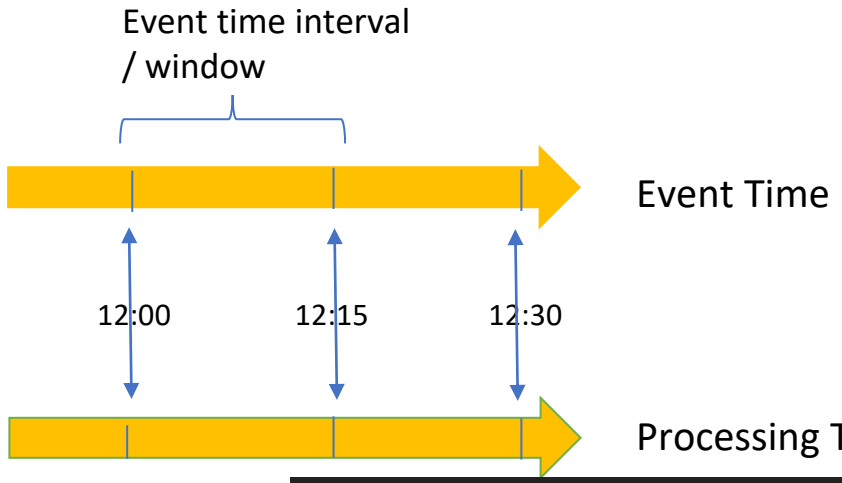
Event Time Windows	Inputs (so far)	Average
12:00 - 12:15	50, 10, 40	33.33
12:15 – 12:30	0	0

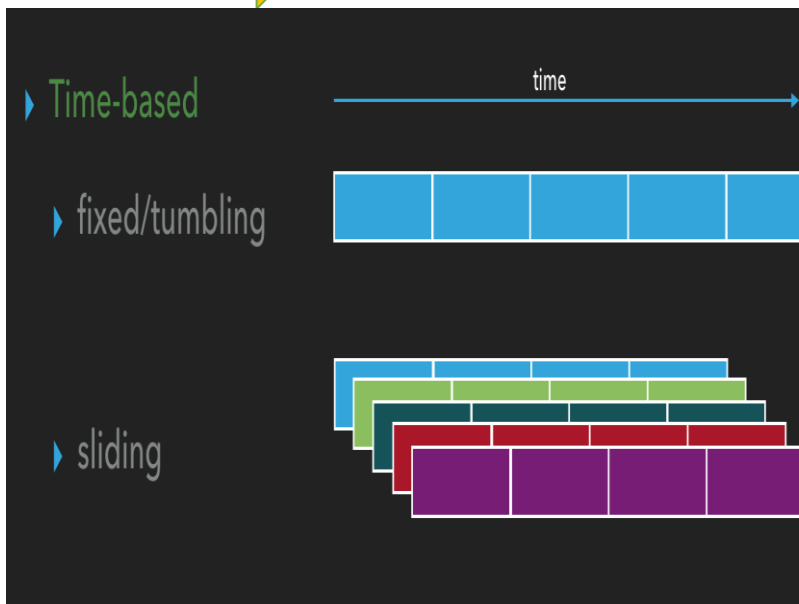
Avg using Processing Time Windows	Inputs (so far)	Average
12:00 - 12:15	50, 40	45
12:15 – 12:30	10	10

Time Window Based Processing

Event Time Window: A time window defined w.r.t when event occurred.

In Spark Streaming, window based aggregations are defined over event time based windows.





For event time window based processing, event time must be included in

Streaming Rainfall Data Example

Three stations

1 record of the streaming data

Station	Event Time	Rainfall (mm)	Average (w/ windowing)
Bengaluru-1	12:15	100	
Bengaluru-2	12:15	80	
Bengaluru-3	12:15	120	100
Bengaluru-1	12:30	20	
Bengaluru-2	12:30	10	
Bengaluru-3	12:30	30	20
Bengaluru-1	12:45	100	
Bengaluru-2	12:45	80	
Bengaluru-3	12:45	120	100

Time event recorded.

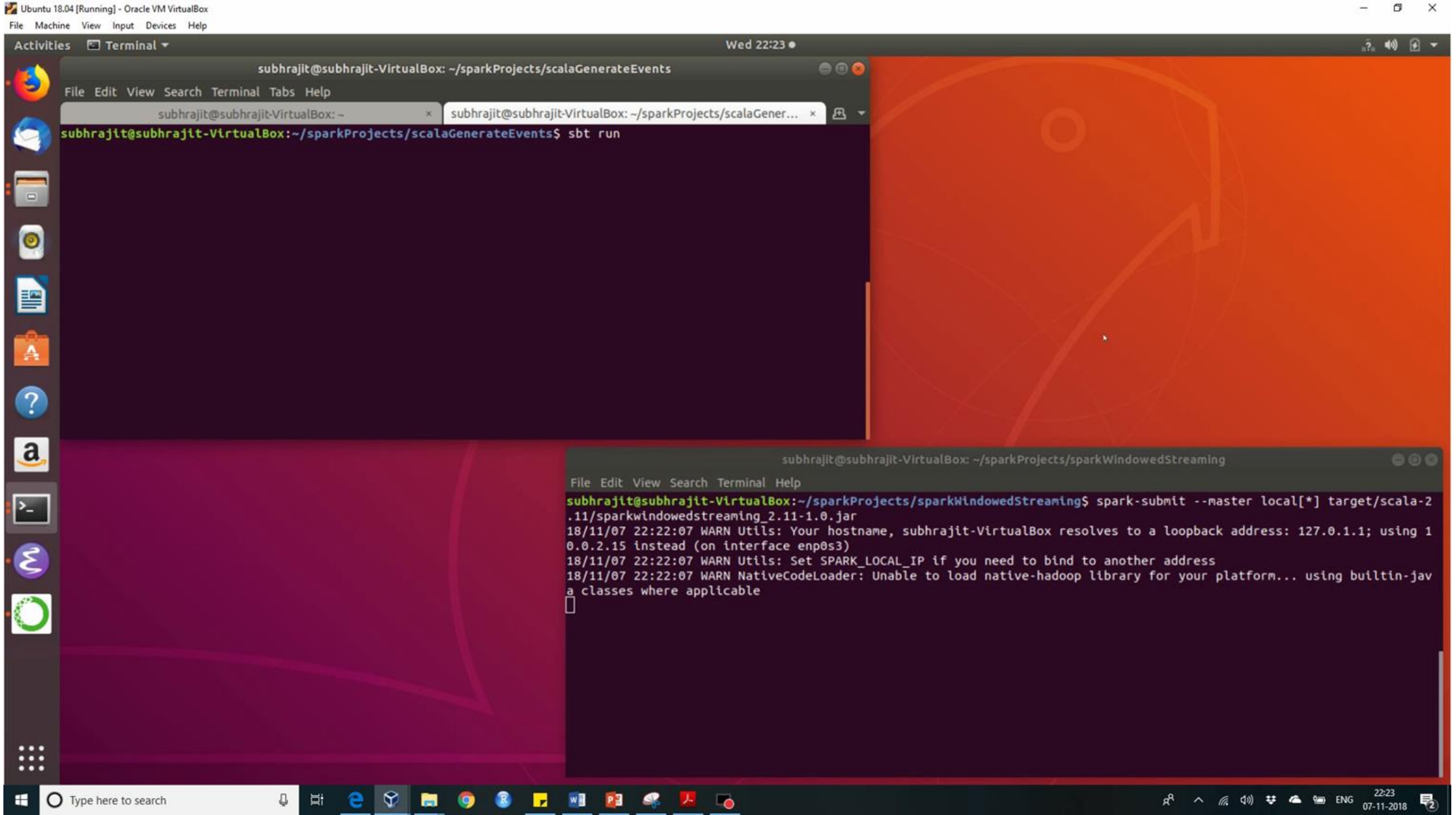
Rainfall event.

```scala
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions.{window, col}
import org.apache.spark.sql.types.StructType

object sparkWindowedStreaming {
  def main(args: Array[String]) {

    val spark = SparkSession.builder.appName("sparkWindowedStreaming").getOrCreate()

    import spark.implicits._

    spark.conf.set("spark.sql.shuffle.partitions", 5)

    val userSchema = new StructType().
      add("Creation_Time", "double").
      add("Station", "string").
      add("Rainfall", "float")

    val streaming = spark.
      readStream.
      schema(userSchema).
      json("/home/subhrajit/sparkProjects/data/event-data/threeWindows")

    val withEventTime = streaming.selectExpr(
      "*",
      "cast(cast(Creation_Time as double)/1000000000 as timestamp) as event_time")

    val events_per_window = withEventTime
      .groupBy(window(col("event_time"), "15 minutes"))
      .agg(avg("Rainfall"),count("Station"))
      .writeStream
      .queryName("events_per_window")
      .format("console")
      .outputMode("complete")
      .option("truncate", false)
      .start()

    events_per_window.awaitTermination()

  }
}
```

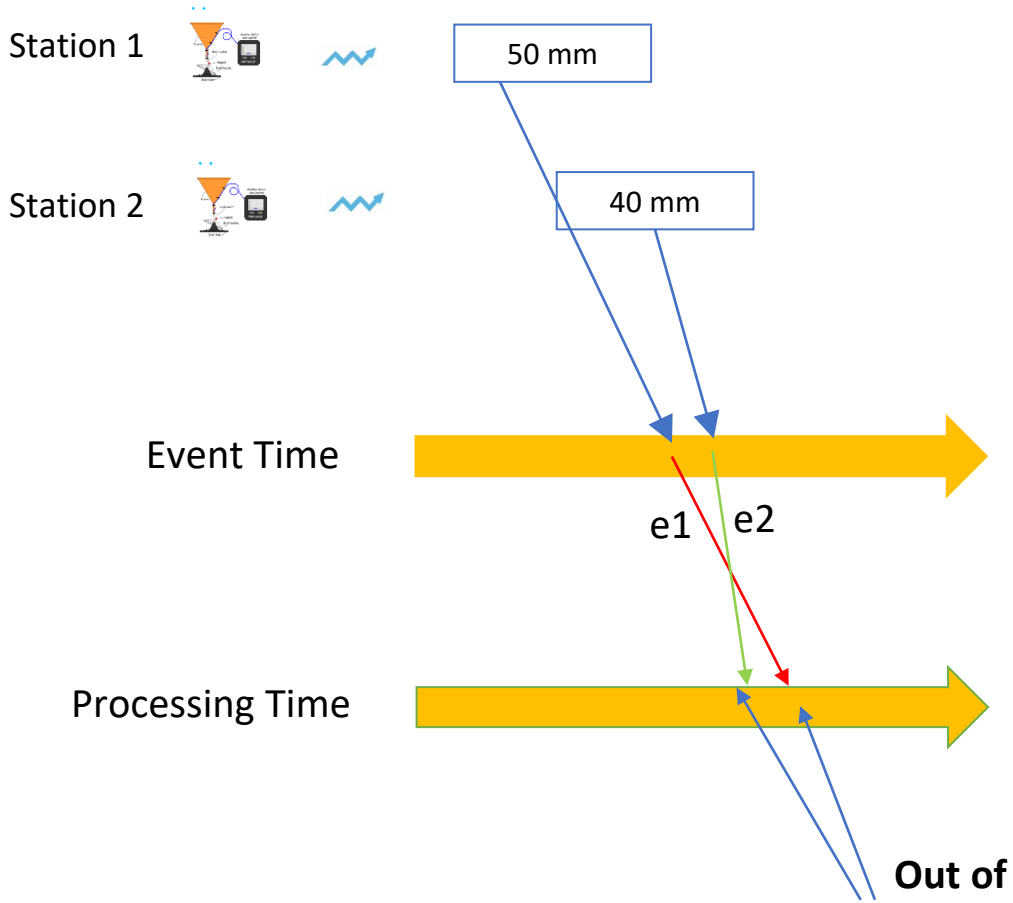
Code for Average Rainfall With 15 minutes Windows

Station	Event Time	Rainfall (mm)	Average (w/ windowing)
Bengaluru-1	12:15	100	
Bengaluru-2	12:15	80	
Bengaluru-3	12:15	120	100
Bengaluru-1	12:30	20	
Bengaluru-2	12:30	10	
Bengaluru-3	12:30	30	20
Bengaluru-1	12:45	100	
Bengaluru-2	12:45	80	
Bengaluru-3	12:45	120	100

Spark Streaming with Windowing for Rainfall Data - Demo



Out of Order (OOO) Events

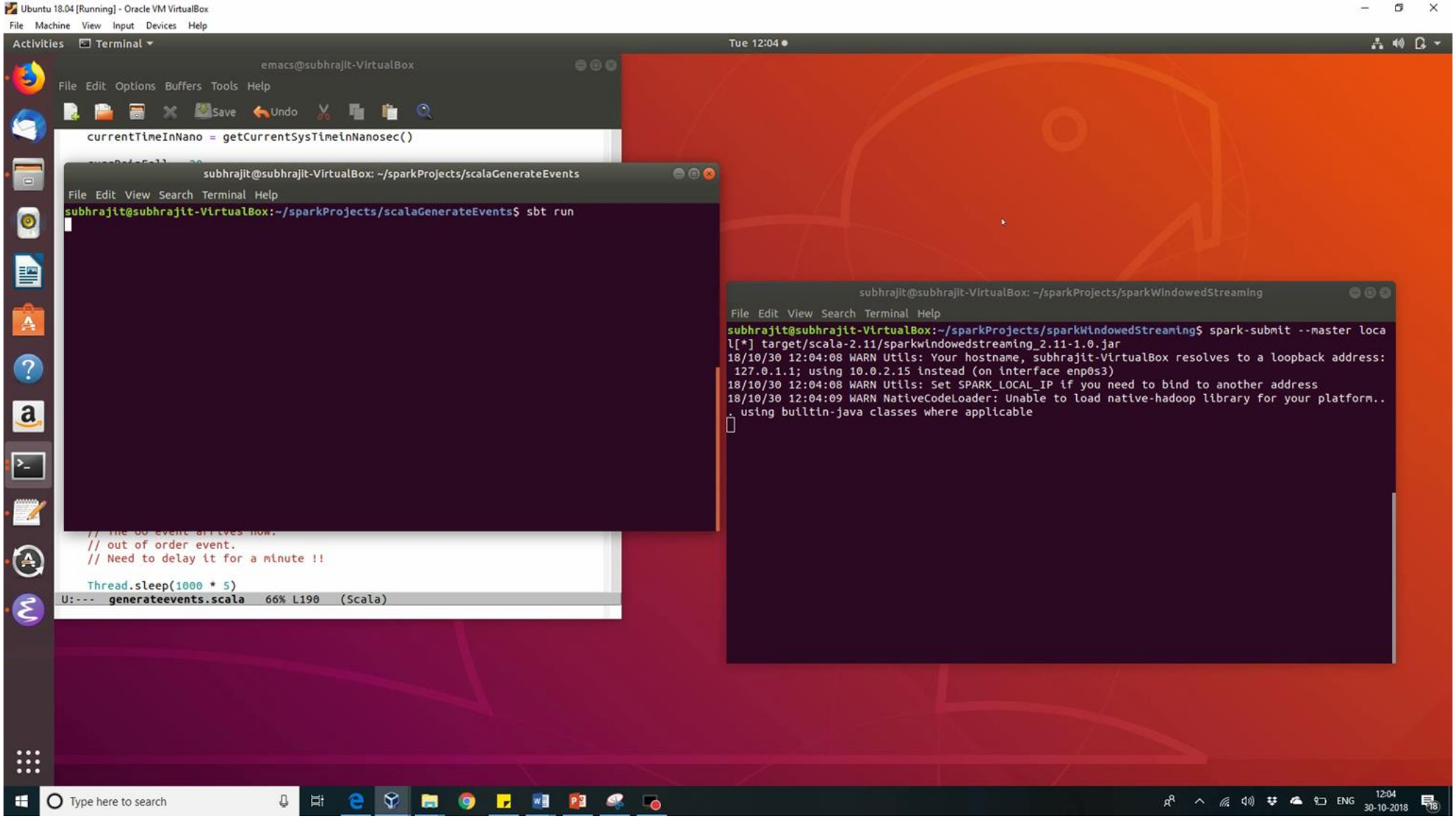


Streaming events arrive with some (network) delay at data center.

Out of order events: For two events e1 and e2, e1 occurs before e2 but arrives after e2, then e1 is out of order.

OOO Event Processing

- Until out of order event arrives, aggregate for corresponding window is incomplete / incorrect.
- When out of order event arrives, window aggregate is updated.
- To update window aggregates in presence of OOO events, window aggregates have to be carried forward as state.



State with and without OOO Events

If Out Of Order Events can occur:

- batch 0:
  - compute aggregate for window 0, w0.agg.
  - print w0.agg.
  - carry forward w0.agg.
- batch 1:
  - compute aggregate for window 1, w1.agg.
  - print w1.agg.
  - carry w1.agg, w2.agg forward.
- batch 2:
  - compute aggregate for window 2, w2.agg.
  - OOE for window 1 arrives. update aggregate w1.agg.
  - print w1.agg and w2.agg.
  - carry w0.agg, w1.agg, w2.agg, forward.

If Out Of Order Events can't occur:

- batch 0:
  - compute aggregate for window 0, w0.agg.
  - print w0.agg.
- batch 1:
  - compute aggregate for window 1, w1.agg.
  - print w1.agg.
- batch 2:
  - compute aggregate for window 2, w2.agg.
  - print w2.agg.

- If window events can arrive after inordinate amount of time, Spark may have to maintain window aggregates for infinite amount of time to update aggregates when out of order events arrive.
- This could increase storage requirements beyond acceptable limits.

Watermarking

- Until out of order event arrives, aggregate for corresponding window is incomplete / incorrect.
- Forces Spark to maintain window aggregates permanently as state.
- Watermarking limits state data:
  - Maintain window aggregates until a certain threshold delay on event time.
  - After that delay, assume no more OOO events will arrive.
  - Window aggregates discarded after given delay.
- Watermarking detail:
  - Let us assume we are ready to wait for additional time D for an event to arrive.
  - If event arrives after this delay, the event is discarded.
  - Let event e_o belongs to window [T1 – T2] and have timestamp T'.
  - Let latest e_L event to arrive have timestamp > T2 + D.
  - Then window [T1-T2] is assumed complete.
  - If e_o arrives after e_L, it is discarded.
  - The wait time D is called the **watermark delay**.

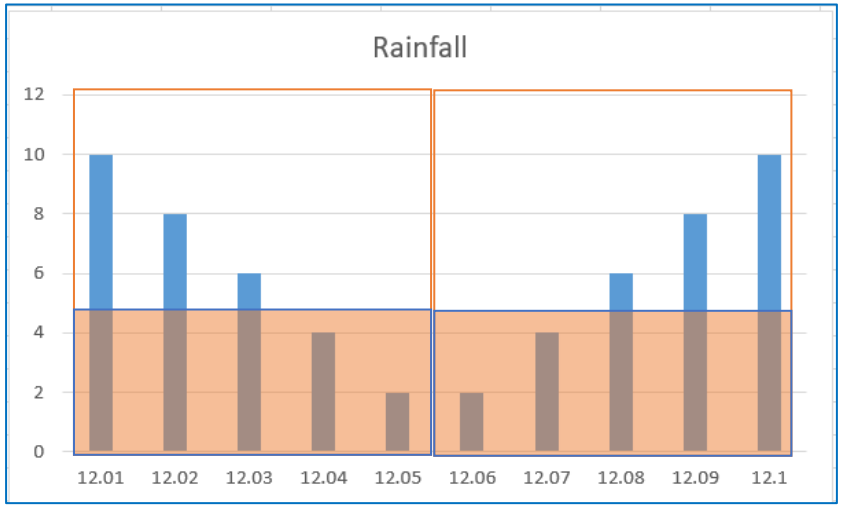
Specifying Watermark to Spark

Watermark or maximum out of order delay.

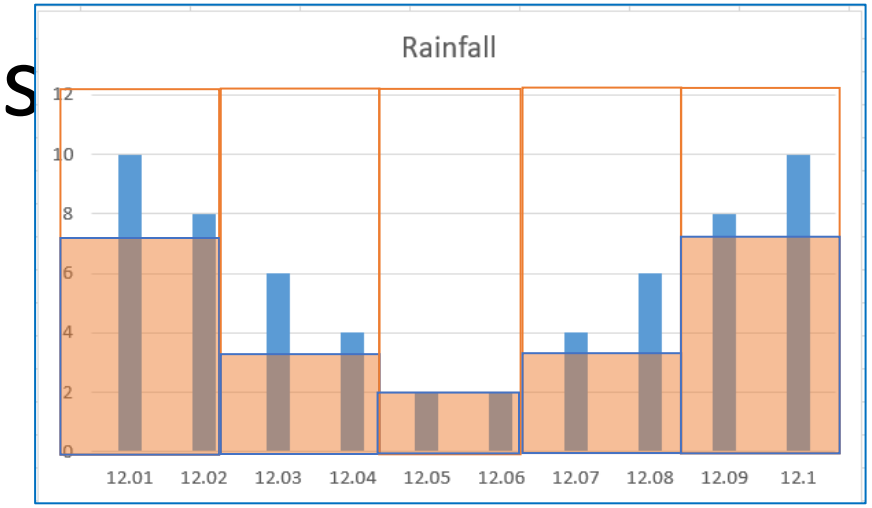
```
val withEventTime = streaming.selectExpr(
  "*",
  "cast(cast(Creation_Time as double)/1000000000 as timestamp) as event_time")

val events_per_window = withEventTime
  .withWatermark("event_time", "1 minutes")
  .groupBy(window(col("event_time"), "15 minutes"))
  .agg(avg("Rainfall"))
  .writeStream
  .queryName("events_per_window")
  .format("console")
  .outputMode("complete")
  .option("truncate", false)
  .start()
```

Sliding/Overlapping Time Windows

Event Time	Rainfall
12.01	10
12.02	8
12.03	6
12.04	4
12.05	2
12.06	2
12.07	4
12.08	6
12.09	8
12.1	10

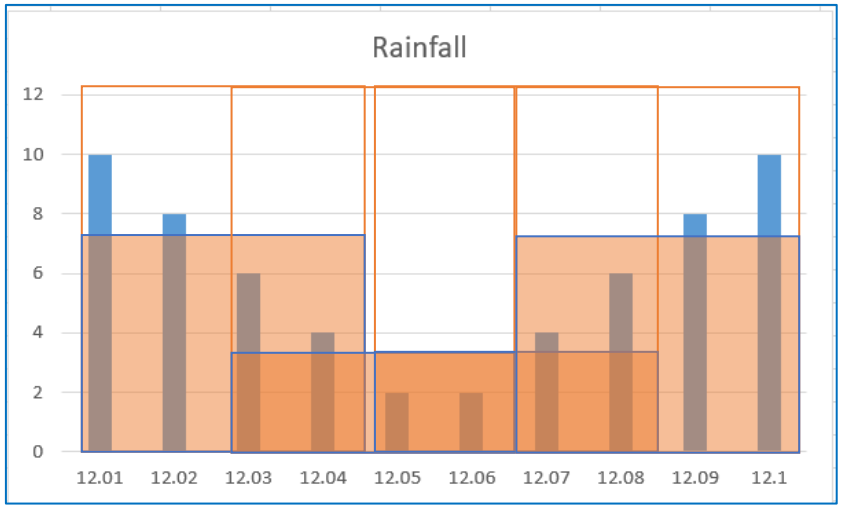


Non-overlapping Windows.



Non-Overlapping Windows (shorter)

Sliding Windows: Overlapping windows on event time are called sliding windows.



Sliding/Overlapping Windows.

Windows show averages over time, which is often more meaningful than instantaneous patterns. However, if the interval is too long, behaviour pattern gets averaged out and the pattern is lost. One can play with controlling window size, or by using sliding windows.

Directive For Sliding Window

Length of window.

Amount by which to slide window.

```
val withEventTime = streaming.selectExpr(
  "*",
  "cast(cast(Creation_Time as double)/1000000000 as timestamp) as event_time")

val events_per_window = withEventTime
  .withWatermark("event_time", "1 minutes")
  .groupBy(window(col("event_time"), "15 minutes", "5 minutes"))
  .agg(avg("Rainfall"))
  .writeStream
  .queryName("events_per_window")
  .format("console")
  .outputMode("complete")
  .option("truncate", false)
  .start()
```

Input Sources, Output Sinks

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

Stream Sinks	Output Modes
File Sink	Append
Kafka Sink	Append, Update, Complete
Foreach Sink	Append, Update, Complete
Console Sink	Append, Update, Complete
Memory Sink	Append, Complete

Some restrictions exist on the output modes in addition to sink type.

Output Operations

- Console Sink (for debugging)

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

- File Sink

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

- Kafka Sink
- Memory Sink (for debugging)
- Foreach Sink

Performance Tuning

- Batch Size Control
  - If batch processing time is more than batch interval, increase batch size.
- Increasing Parallelism
  - Increase number of receivers of streaming data. Perform union of multiple streams before aggregation.
- Sliding interval
  - Increase slide interval if required to reduce number of overlapping windows on which aggregation has to be performed.
- Java garbage collection
  - Optimize java garbage collection if this is the bottleneck.

Checkpointing

- Checkpointing is used to recover from node failures.
- Checkpointing saves the window aggregates dataframe and the offsets† in the streaming data.

```
val events_per_window = withEventTime
  .groupBy(window(col("event_time"), "1 minutes"))
  .agg(avg("Rainfall"),count("Station"))
  .writeStream
  .queryName("events_per_window")
  .format("console")
  .outputMode("complete")
  .option("checkpointLocation", "path/to/HDFS/dir")
  .option("truncate", false)
  .start()
```

path to checkpoint location.

† The position in the stream till which processing has been completed.

Preliminaries - row

Row is a generic Spark object with an ordered collection of fields.

```
import org.apache.spark.sql._

val row = Row(1, true, "a string", null)
row(0)
```

```
Any = 1
```

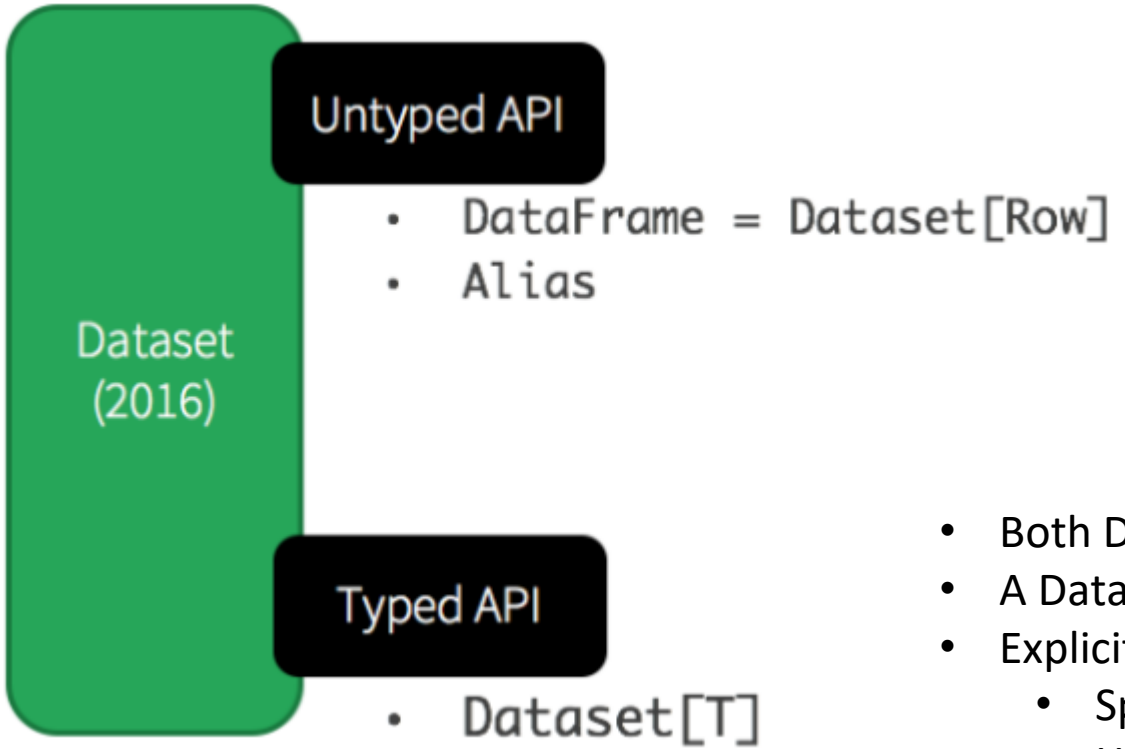
```
row.getInt(0)
```

```
Int = 1
```

If the type of row element is known, then the type can be changed from Any to the known type.

“Row” object can be used to store data without pre-specified type information. Type information can be generated or specified on demand.

Preliminaries – Datasets and Dataframes



IdNum	LName	FName	JobCode	Salary
1876	CHIN	JACK	TA1	$44,096
1114	GREENWALD	JANICE	ME3	$38,950
1556	PENNINGTON	MICHAEL	ME1	$31,054
1354	PARKER	MARY	FA3	$67,445
1130	WOOD	DEBORAH	PT2	$37,427

- Both DataFrames and Datasets are used to represent tables.
- A DataFrame is an alias for Dataset[Row].
- Explicit type definition is not required for DataFrames.
  - Spark generates type definitions or schemas at runtime.
  - Hence type checking can't be done during compile time.
  - But gives better performance due to "Row" type's optimized in-memory format for computation.

Metadata Refreshing in Spark SQL

Spark caches Hive table metadata for performance reasons.

Need to explicitly refresh metadata in case of change by other application.

```
spark.catalog.refreshTable("my_table")
```