Lab 2

Big Data Analytics on Cloud Computing Infrastructures

Azad Shokrollahi, Mahtab Jamali

Stream processing

Stream processing is a data processing paradigm that involves the real-time or near-real-time analysis and manipulation of data as it flows through a system. It is particularly useful for handling continuous data streams, such as those generated by sensors, applications, websites, and other sources. Here are some key characteristics and concepts related to stream processing:

1. **Continuous Data Streams**: Stream processing is designed to handle data that is continuously generated and arrives in small, often unbounded, chunks over time. Examples include IoT sensor data, social media updates, financial transactions, and clickstream data.

2. **Low Latency**: Stream processing systems aim to process data with low latency, meaning they analyze and respond to incoming data quickly, often within milliseconds or seconds. This real-time aspect is crucial for applications like fraud detection, recommendation engines, and monitoring systems.

3. **Event-Driven**: Stream processing is inherently event-driven, meaning it reacts to events or data points as they occur. Each data point or event triggers processing and potentially generates new events or data.

4. **Stateful and Stateless Processing**: Stream processing can be stateful or stateless. Stateful processing maintains information about the stream's history, allowing for context-aware analysis. Stateless processing treats each event in isolation.

5. **Windowing**: Time-based or count-based windows are used to group and analyze data over specific intervals. This is essential for aggregations and calculations over a sliding or tumbling window of time.

6. **Fault Tolerance**: Stream processing systems are designed to be fault-tolerant. They can recover from failures and ensure that no data is lost during processing.

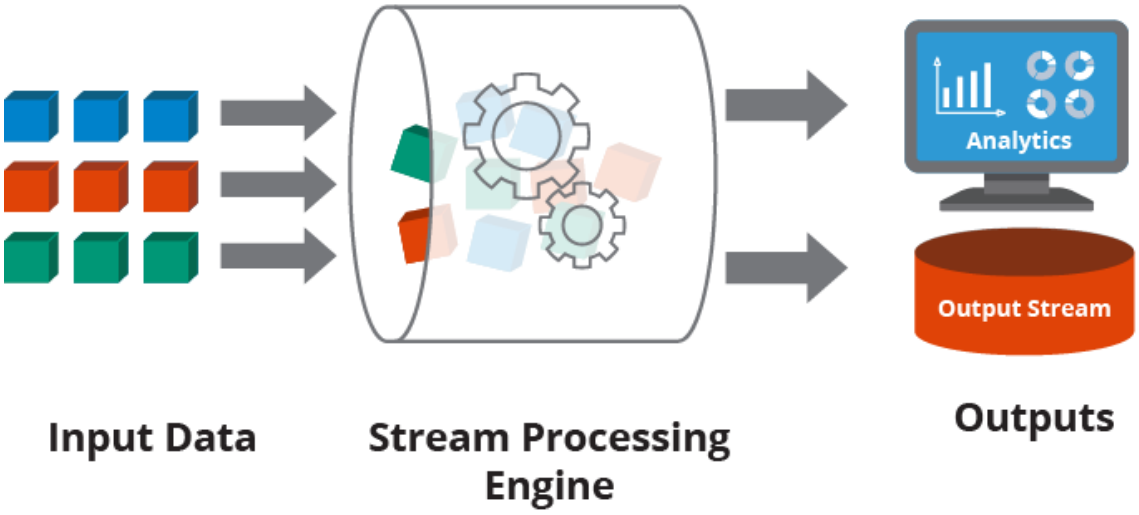
7. **Parallelism**: Stream processing frameworks often support parallelism to handle high data volumes efficiently. Processing can be distributed across multiple workers or nodes.

8. **Scalability**: Stream processing systems can scale horizontally to accommodate increasing data volumes and processing requirements. This is particularly important for large-scale applications.

9. **Complex Event Processing (CEP)**: CEP is a subfield of stream processing that focuses on detecting complex patterns and correlations in data streams. It's used in applications like fraud detection and real-time analytics.

10. **Use Cases**: Stream processing is employed in various domains, including IoT data analysis, financial trading, online advertising, recommendation systems, social media monitoring, and network security.

11. **Tools and Frameworks**: There are specialized stream processing frameworks and tools (e.g., Apache Kafka Streams, Apache Flink, Apache Storm) that simplify the development of stream processing applications.



summarize stream processing framework

amework	Latency	Stateful Processing	Fault Tolerance	Parallelism	Scalability	Complex Event Processing	Use Cases
Apache Kafka	Low	Yes	Yes	Yes	High	Yes	Real-time analytics, event-driven apps
Apache Flink	Low	Yes	Yes	Yes	High	Yes	IoT, fraud detection, analytics
Apache Storm	Low	Yes	Yes	Yes	High	Yes	Real-time processing, event-driven apps
Apache Beam	Low	Yes	Yes	Yes	High	Yes	Data pipelines, batch/stream integration
Apache Samza	Low	Yes	Yes	Yes	High	Yes	Real-time data processing, stateful apps
Amazon Kinesis	Low	Yes	Yes	Yes	High	Limited	Real-time data processing, analytics
NATS Streaming	Low	No	Yes	Yes	High	Limited	Messaging, event streaming
Spark Streaming	Low	Yes	Yes	Yes	High	Limited	Real-time analytics, batch integration
Confluent Platform	Low	Yes	Yes	Yes	High	Yes	Real-time event-driven applications
IBM Streams	Low	Yes	Yes	Yes	High	Yes	Real-time analytics, IoT data processing

Structured Streaming

Structured Streaming is a stream processing framework in Apache Spark that allows you to process and analyze structured data streams in a scalable, fault-tolerant, and high-level manner. It's built on top of the Spark SQL engine, making it easy to work with structured data in real-time. Here's a comprehensive explanation of Structured Streaming:

1. **Structured Data**: Structured Streaming is designed for processing structured data streams. Structured data typically comes in well-defined formats with a schema, such as JSON, CSV, Parquet, Avro, and more. This differs from unstructured or semi-structured data that doesn't fit neatly into tabular structures.

2. **Real-Time Data Processing**: It enables real-time or near-real-time data processing. Instead of processing data in batch mode, you can continuously process incoming data streams as they arrive, allowing for quick insights and rapid responses.

3. **High-Level API**: Structured Streaming provides a high-level, declarative API for working with streaming data. You can express your data transformations and operations using SQL queries or DataFrame operations, making it accessible to those familiar with SQL or data frames.

4. **Event-Time Processing**: Structured Streaming supports event-time processing, which is essential for handling out-of-order data in real-time streams. You can define watermarks and windows to perform time-based operations like aggregations.

5. **Continuous Processing**: Unlike traditional micro-batch processing, Structured Streaming uses continuous processing. It processes data record by record, providing lower latency and more accurate results.

6. **Fault Tolerance**: It ensures fault tolerance by checkpointing streaming operations. This means that in the event of a failure, the system can recover and maintain data consistency.

7. **Integration**: Structured Streaming can seamlessly integrate with various data sources and sinks. Common sources include Apache Kafka, Apache Flume, and file systems, while sinks can be databases, file systems, dashboards, or other streaming platforms.

8. **Windowed Aggregations**: You can perform windowed aggregations on streaming data. Windows are time-based or count-based intervals that allow you to calculate statistics or perform operations over specific time frames.

9. **Exactly-Once Semantics**: Structured Streaming aims to provide exactly-once processing semantics, ensuring that each record is processed only once, even in the presence of failures.

10. **Scalability**: As part of the Apache Spark ecosystem, Structured Streaming inherits Spark's scalability benefits. It can distribute data processing across a cluster of machines, allowing you to scale horizontally.

11. **Adaptivity**: Structured Streaming can dynamically adapt to varying workloads and data arrival rates, automatically adjusting processing rates based on available resources.
12. **Monitoring and Visualization**: Tools like Spark Web UI and Grafana can be used to monitor and visualize the performance and progress of Structured Streaming applications.

Structured Streaming simplifies the development of real-time data processing applications by providing a familiar API and leveraging Spark's distributed computing capabilities. It is widely used in various industries, including finance, e-commerce, healthcare, and more, for applications requiring real-time data analysis and insights.

Starting with Structured Streaming

Starting with Structured Streaming in Apache Spark involves a series of steps, from setting up your development environment to writing and running streaming applications. Here's a step-by-step guide to get you started:

1. Set Up Your Environment:

- Install Apache Spark: Download and install Apache Spark on your local machine or cluster. You can find installation instructions on the Apache Spark website.
- Install a Compatible IDE: You can use IDEs like IntelliJ IDEA, PyCharm, or Jupyter Notebook for Spark development.

2. Create a SparkSession:

- In Structured Streaming, you work with a **SparkSession**, which is the entry point to Spark functionality. Create a SparkSession in your code:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("StructuredStreamingExample") \
    .getOrCreate()
```

3. Define a Source:

- Choose a streaming data source. Common sources include Apache Kafka, file systems (e.g., file or directory paths), or socket connections. Configure your source accordingly.

```
source_df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "topic_name") \
    .load()
```

In Structured Streaming, you can define a source to ingest data streams from various types of sources. Here are some common types of sources you can use in Structured Streaming:

1. **File Sources**:
   - You can read data streams from various file-based sources such as:
     - **File System**: Streaming data from directories where new files are added.
     - **Apache Kafka**: Reading data from Kafka topics.
     - **Cloud Storage**: Streaming data from cloud storage systems like AWS S3, Google Cloud Storage, or Azure Blob Storage.
     - **HDFS**: Streaming data from Hadoop Distributed File System (HDFS).
2. **Socket Sources**:
   - You can create a socket source to ingest data from network sockets. This is useful for testing and experimentation.
3. **Structured Network Sources**:
   - Spark Structured Streaming provides structured sources for working with various data formats. Some examples include:
     - **Kafka Source**: Streaming data from Apache Kafka topics.
     - **Socket Source**: Reading data from network sockets.
     - **Rate Source**: Generating artificial data streams at a specified rate for testing.
     - **File Source**: Reading data from files in a specified directory.
     - **Console Source**: Reading data from the console for testing and development.
4. **External Systems and Databases**:
   - You can connect Spark Structured Streaming to external systems and databases using connectors or libraries specific to those systems. This allows you to ingest data from and write data to external data sources. Examples include databases like MySQL, PostgreSQL, and NoSQL databases like Cassandra or MongoDB.

Source Name	Description	Example Usage
File	Reads data from files in a directory.	spark.readStream.format("parquet").load("/path/to/dir")
Kafka	Consumes messages from Kafka topics.	spark.readStream.format("kafka").option("subscribe", "topic_name").load()
Delta Lake	Reads from Delta Lake tables, which offer ACID transactions on data lakes.	spark.readStream.format("delta").load("/path/to/delta/table")
Kinesis	Consumes data from Amazon Kinesis streams.	spark.readStream.format("kinesis").option("streamName", "kinesis_stream_name").load()
Rate (for Testing)	Generates data at a specified rate for testing purposes.	spark.readStream.format("rate").option("rowsPerSecond", "10").load()
JDBC (3rd party)	Reads from JDBC databases (with third-party connectors). Less common for streaming.	Dependent on third-party library specifics.
Socket	Reads text data from a TCP socket. Primarily for testing and not for production use.	spark.readStream.format("socket").option("host", "localhost").option("port", "9999").load()

4. Define Data Transformations:

- Use Spark DataFrame operations or SQL queries to define the transformations you want to apply to your streaming data. These can include filtering, aggregating, joining, or other data manipulations.

```
transformed_df = source_df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
```

5. Define Output Sink:

- Specify where you want to send the processed data. Common output sinks include console, file systems (e.g., HDFS), databases, or external services.

```
query = transformed_df.writeStream \
    .outputMode("append") \
    .format("console") \
    .start()
```

6.Start the Streaming Query:

Start the streaming query using the start() method on your query object.

```
query.awaitTermination()
```

7. Run Your Application:

- Run your Spark application. Depending on your development environment, you may run it from the command line, IDE, or Jupyter Notebook.

8. Monitor Your Application:

- Use Spark's built-in monitoring tools, web UI, and logging to monitor the progress and performance of your Structured Streaming application.

9. Handle Errors and Failures:

- Implement error handling and fault tolerance mechanisms to ensure the reliability of your streaming application. Structured Streaming provides features like checkpointing and recovery for this purpose.

10. Stop Your Application: - When you're done with your streaming application, stop it using the **query.stop()** method.

How to run code(First example using Socket Sources)?

1.Start master and worker based on previous lab.

2.Go to bin folder and open new python file by nano(nano stream1_Example.py)

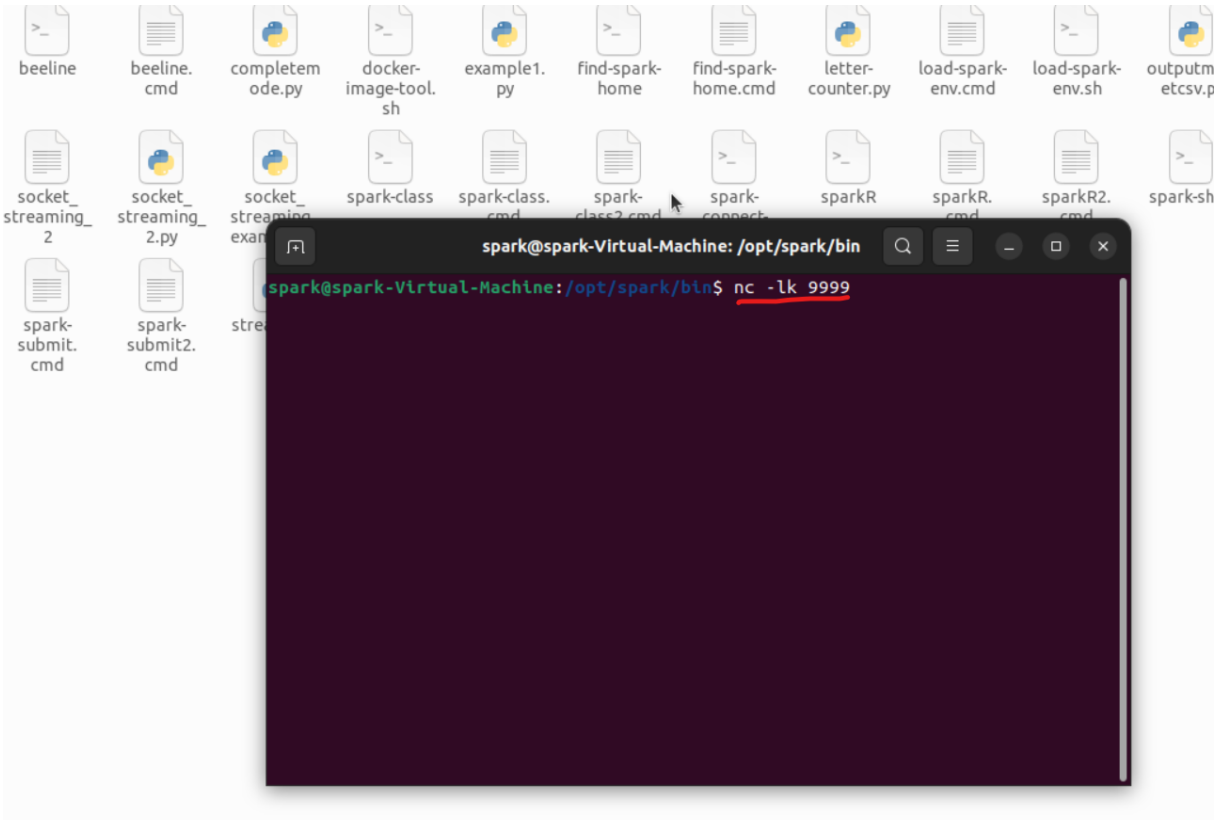
3.And copy past bellow code in this file and press ctrl+x in order save file

- The program listens for any data being sent to a specific port (**9999**) on **localhost** (the computer where the program is running).
- When data is received on this port, **the program immediately displays this data on the console.**
- It keeps listening and displaying until the program is stopped.

```
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder \
   .appName("SimpleStructuredStreaming") \
   .getOrCreate()

# Define the Input Source (Socket)
streamingInputDF = spark.readStream \
   .format("socket") \
   .option("host", "localhost") \
   .option("port", 9999) \
   .load()

# Define the Output Sink (Console) with " append " output mode
query = streamingInputDF \
   .writeStream \
   .outputMode("append") \
   .format("console") \
   .start()

# Start and Await Termination
query.awaitTermination()
```

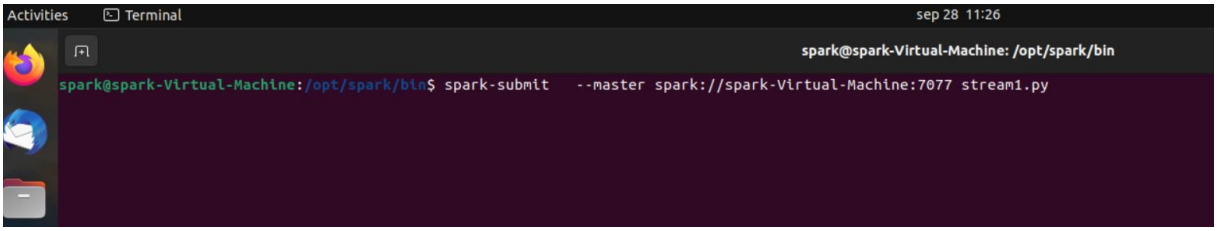
```
 1 from pyspark.sql import SparkSession
 2
 3 # Create a SparkSession
 4 spark = SparkSession.builder \
 5     .appName("SimpleStructuredStreaming") \
 6     .getOrCreate()
 7
 8 # Define the Input Source (Socket)
 9 streamingInputDF = spark.readStream \
10     .format("socket") \
11     .option("host", "localhost") \
12     .option("port", 9999) \
13     .load()
14
15 # Define the Output Sink (Console) with "complete"
16 query = streamingInputDF \
17     .writeStream \
18     .outputMode("append") \
19     .format("console") \
20     .start()
21
22 # Start and Await Termination
23 query.awaitTermination()
24
```

4. run nc -lk 9999 without closing terminal



nc -lk 9999: This command starts the netcat utility in listening mode (-l) and keeps it listening for incoming connections (-k) on port 9999. Essentially, it sets up a simple server that you can use to manually send strings of text to any client connected to that port.

5. open another terminal in bin folder and Then submit/run the Spark program look like bellow pictures.

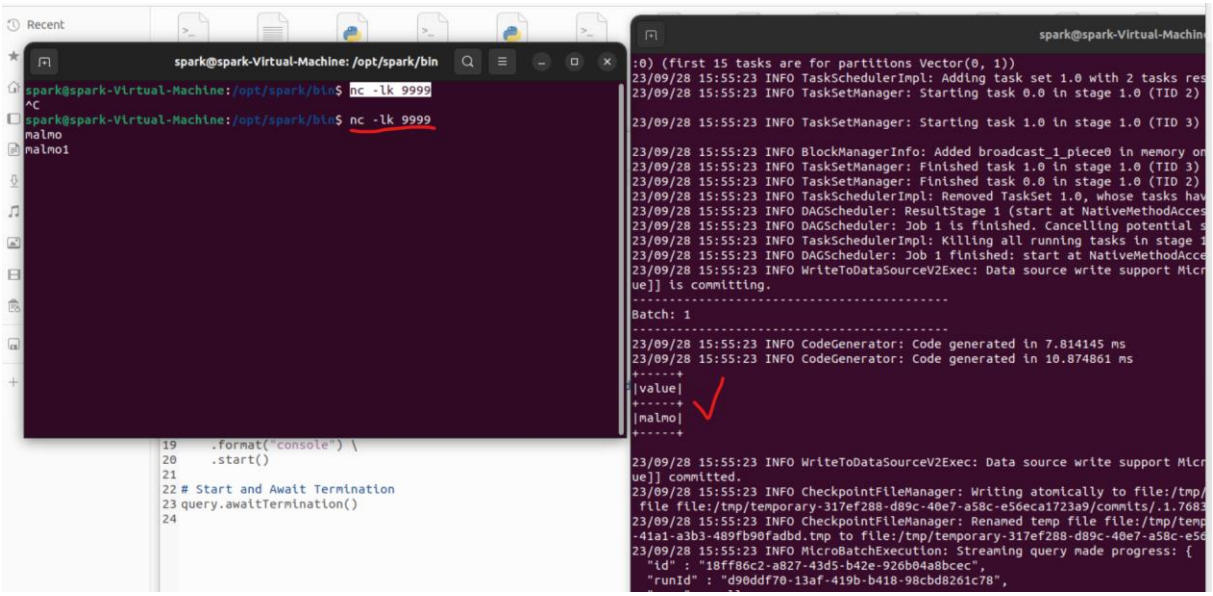


1. When you submit/run the Spark program after starting netcat, the program will connect to the server set up by netcat (because we've specified localhost and port 9999 in the Spark code). Once this connection is established, any text you enter into the netcat terminal will be captured by the Spark Structured Streaming job and then printed to the console (based on the logic we've provided in the Spark code).

So, starting netcat first ensures there's an active source of data for the Spark program to connect to when it's run. **If you try to run the Spark program first, it might not be able to find the data source and could throw an error.**

6.Based on code we type in first terminal, and we see the result on another terminal

Continue typing in left terminal and press enter.



Explain the code

Initializing a SparkSession:

Setting up the Input Data Stream:

```
streamingInputDF = spark.readStream \
   .format("socket") \
   .option("host", "localhost") \
   .option("port", 9999) \
   .load()
```

Explanation: Defines a streaming source, where the data comes from.

- **spark.readStream**: Indicates the intention to read from a streaming source.
- **.format("socket")**: Specifies that the source format is a TCP socket.
- **.option("host", "localhost")**: Sets the host address of the TCP socket to "localhost".
- **.option("port", 9999)**: Sets the port of the TCP socket to 9999. This means the program will listen for incoming data on port 9999 of the localhost.
- **.load()**: Completes the command to create a streaming DataFrame (**streamingInputDF**).

Setting up the Output Data Stream:

```
query = streamingInputDF \
   .writeStream \
   .outputMode("append") \
   .format("console") \
   .start()
```

Explanation: Configures how and where the processed streaming data will be outputted.

- **streamingInputDF.writeStream**: Indicates the intention to write the streaming DataFrame to an output.
- **.outputMode("append")**: Defines the output mode as "append". In this mode, only the new rows added to the output sink are displayed, not the entire DataFrame.
- **.format("console")**: Specifies that the output (or sink) should be the console. The data will be displayed in the terminal or command prompt.
- **.start()**: Starts the streaming query. This initiates the continuous process of checking for new data, processing it, and sending it to the console.

Keep the Streaming Application Running:

```
query.awaitTermination()
```

Explanation: Ensures that the streaming application continues to run until it's either manually terminated or encounters an error. This command holds the program in a waiting state, allowing data to be continually read, processed, and outputted as long as the program is running.

outputMode

In Apache Spark's structured streaming, outputMode defines how the output of a streaming query is written to the output sink in terms of what data should be written to the external storage. There are three primary output modes:

1. Complete Mode (outputMode("complete")):
    - In this mode, the entire updated Result Table is written to the sink after every trigger. This means that the sink will have the complete and latest result after every trigger.
    - This mode is mostly used with aggregation queries where you want to see the entire updated result every time, not just the recent changes.
2. Append Mode (outputMode("append")):
    - In this mode, only the new rows added to the Result Table since the last trigger are written to the sink.
    - This is the default mode for many sinks.
    - It's ideal for use cases where only new data (rows) are of interest, and there's no need to rewrite unchanged old rows.
    - A critical thing to note is that using append mode on a query with aggregations will throw an exception unless the aggregation is watermarked. This is because Spark needs to ensure that no old data will be seen again and added to the result.
3. Update Mode (outputMode("update")):
    - In this mode, only the rows in the Result Table that were updated since the last trigger will be written to the sink.
    - It's useful when you want to see changes in your aggregated results but don't need to see the entire result table every time.
    - Rows that are not changed (from the previous trigger to the current trigger) will not be written to the sink.

When choosing an outputMode, consider the requirements of your use case and the nature of your query. Not all output modes are supported by all sinks, so always check compatibility with the specific sink you are using. For example, the console sink, which is commonly used for testing and debugging, supports all output modes. However, other sinks like parquet or kafka might have restrictions on which output modes can be used.

Example for Update mode

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split

# Create a SparkSession
spark = SparkSession.builder \
    .appName("UpdateModeExample") \
    .getOrCreate()

# Define the Input Source (Socket)
streamingInputDF = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Tokenize each line into words and count them
words = streamingInputDF.select(explode(split(streamingInputDF.value, " ")).alias("word"))
wordCounts = words.groupBy("word").count()

# Define the Output Sink (Console) with "update" output mode
query = wordCounts \
    .writeStream \
    .outputMode("update") \
    .format("console") \
    .start()

# Start and Await Termination
```

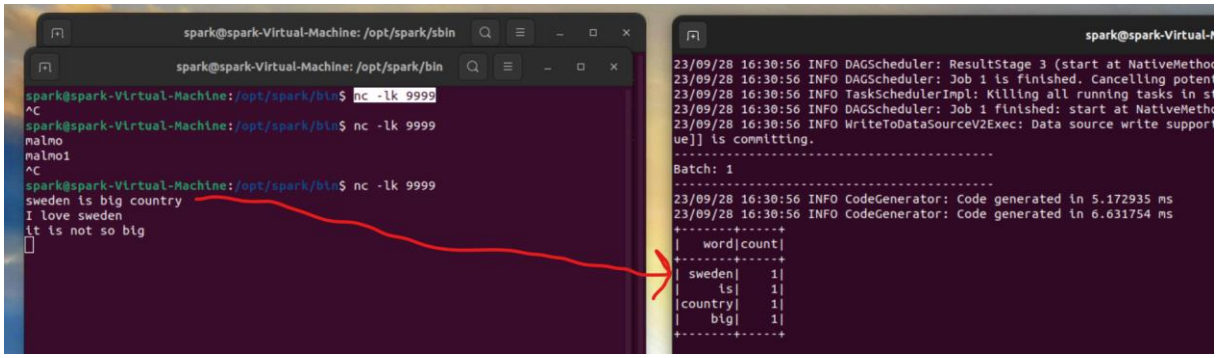
Explanation:

- **split(streamingInputDF.value, " "): This uses the split function to divide each line of the DataFrame streamingInputDF into individual words. The splitting is based on space (" "). The result of this operation for each line is an array of words.**
- **explode(...): The explode function is used to transform a row that has an array of multiple elements into multiple rows, with each row having one element from the array. In this context, it's turning the array of words from each line into multiple rows with individual words.**
- **.alias("word"): This renames the column resulting from the explode function to "word".**

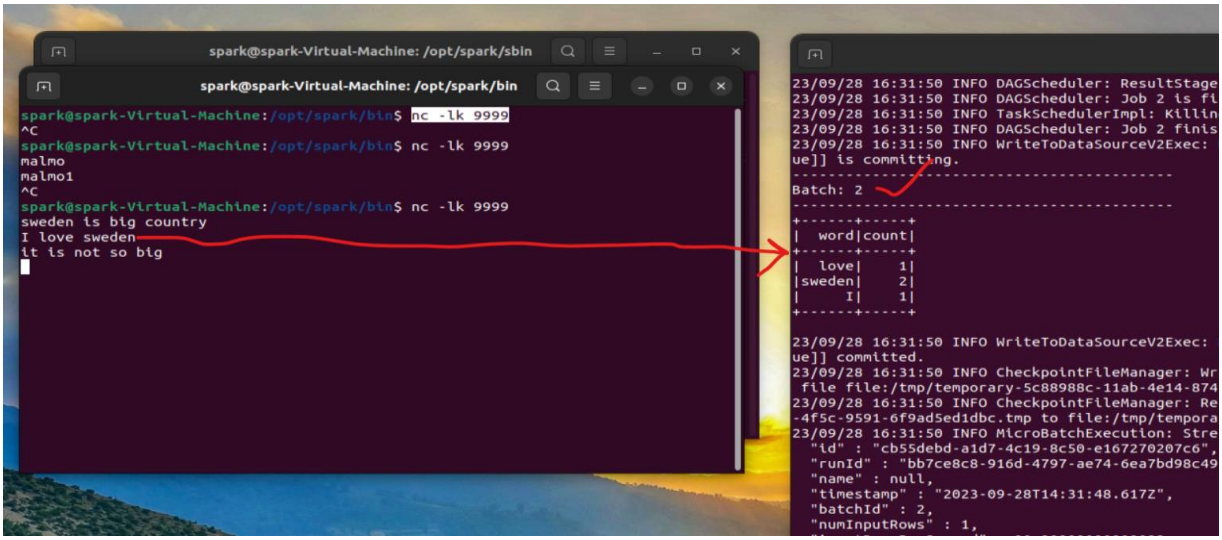
The outcome of this line is a new DataFrame words where each row contains a single word from the input data.

Writing first sentences:

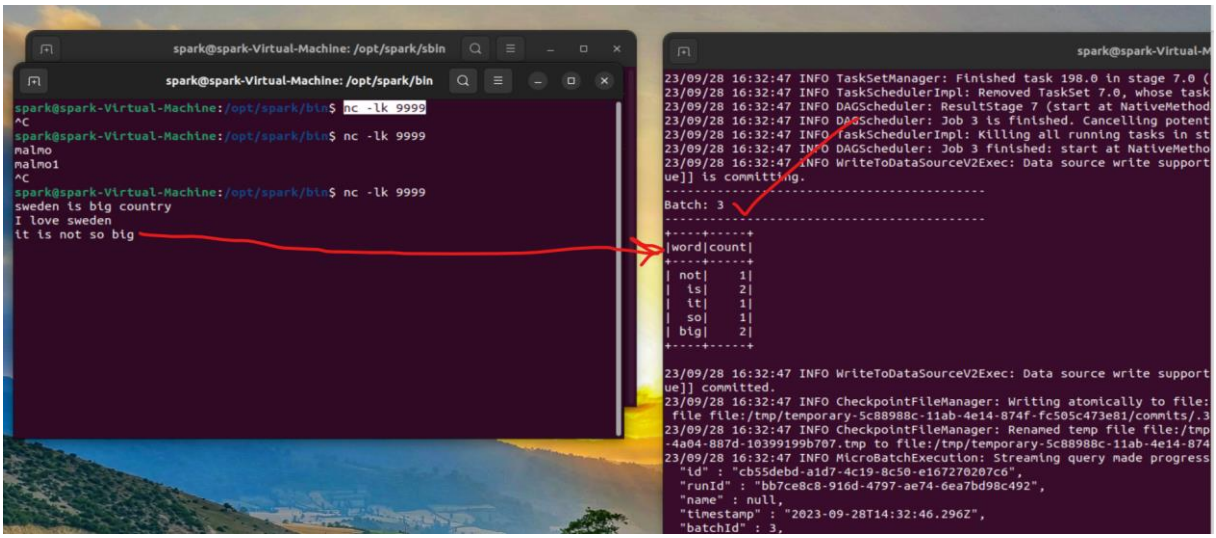
If you type sentences in the first terminal and press enter, you can see the result on the other side. If you continue writing and press enter, you can see the result on the other side based on update mode.



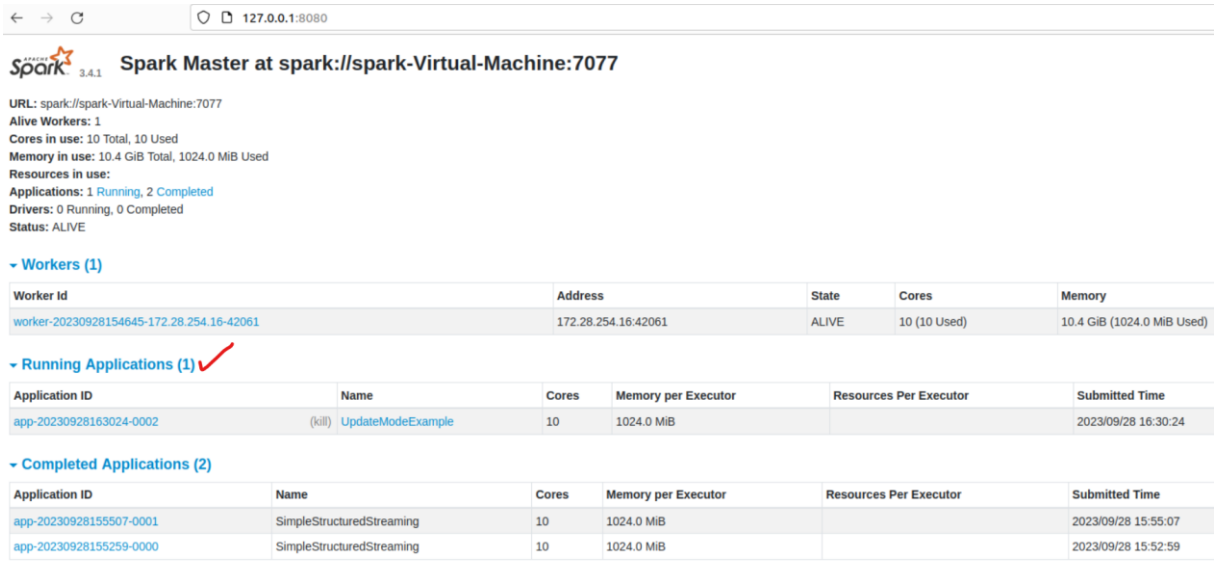
Writing second sentences



Writing third sentences



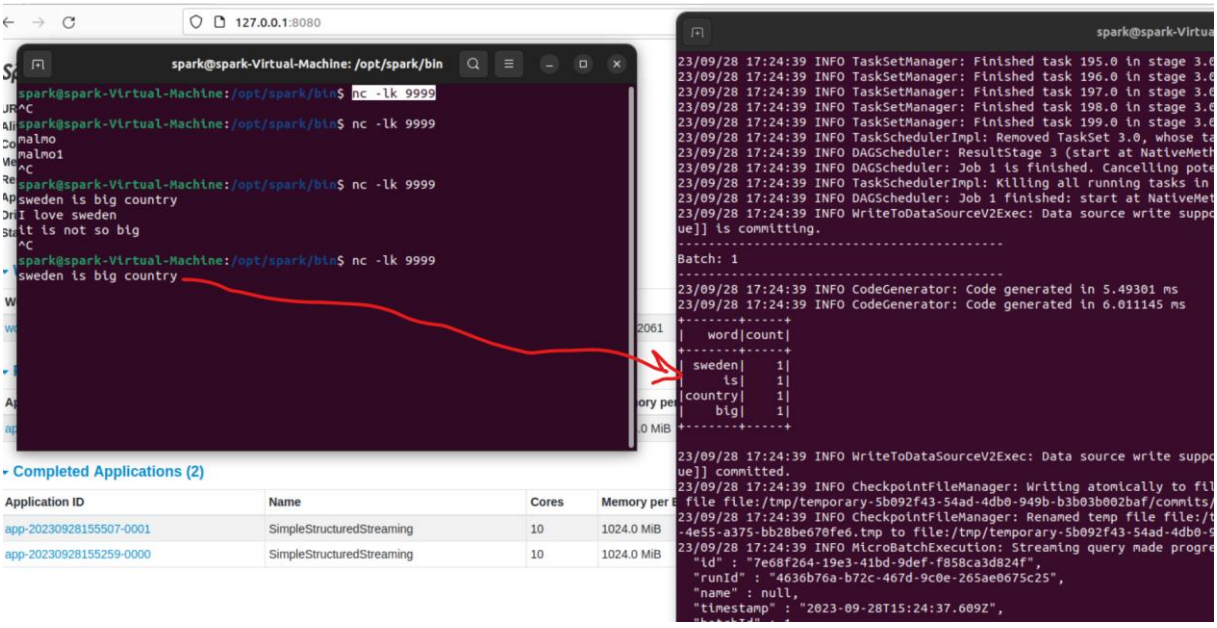
As you can see in bellow pictures , this program are running



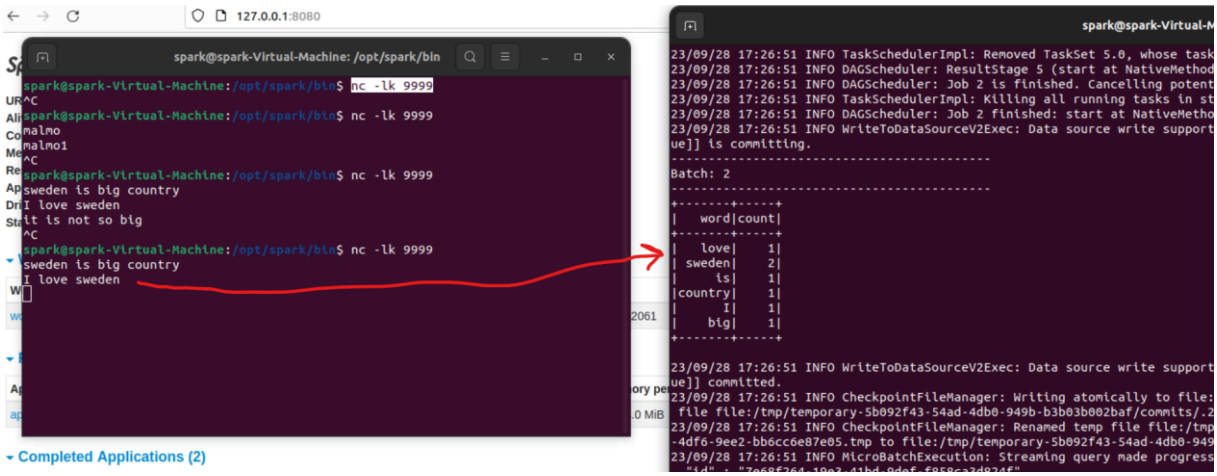
Example for Complete mode

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split

# Create a SparkSession
spark = SparkSession.builder \
    .appName("CompleteModeExample") \
    .getOrCreate()

# Define the Input Source (Socket)
streamingInputDF = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Tokenize each line into words and count them
words = streamingInputDF.select(explode(split(streamingInputDF.value, " ")).alias("word"))
wordCounts = words.groupBy("word").count()

# Define the Output Sink (Console) with "complete" output mode
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

# Start and Await Termination
query.awaitTermination()
```

Type the first sentences:



Type second sentences and you can see result in another terminal



More example

Letter counter(complete mode)

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, lower, split

# Initialize a Spark session
spark = SparkSession.builder \
    .appName("RealTimeLetterCounter") \
    .getOrCreate()

# Define the streaming input DataFrame to read data from a socket
inputStream = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Tokenize each line into letters and convert to lowercase
letters = inputStream.select(explode(split(lower(inputStream.value), "")).alias("letter"))

# Count occurrences of each letter
letterCounts = letters.groupBy("letter").count()

# Output the letter count to the console
query = letterCounts.writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

# Await termination of the query
query.awaitTermination()
```

split(..., "")
Explanation:

- **split** function is used to break a string into individual substrings based on a specified delimiter.
- In this case, the delimiter is an empty string (**""**). This means the function will split the string into individual characters.

Line length count(append mode)

```
from pyspark.sql import SparkSession

# Initialize a Spark session
spark = SparkSession.builder \
    .appName("LineLengthCounter") \
    .getOrCreate()

# Define the streaming input DataFrame to read data from a socket
inputStream = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Compute the length of each line
lineLengths = inputStream.selectExpr("length(value) as line_length")

# Output the line lengths to the console
query = lineLengths.writeStream \
    .outputMode("append") \
    .format("console") \
    .start()

# Await termination of the query
query.awaitTermination()
```

In PySpark, the **selectExpr** method is a convenient function provided by DataFrames and Datasets that allows for SQL-like querying. It's especially useful for performing operations on columns using SQL expressions without needing to use the full SQL syntax or create a temporary table/view.

Here's a breakdown:

1. **Basic Idea**:
   - While **select** allows you to select columns or compute new ones using functions from the **pyspark.sql.functions** module, **selectExpr** enables you to express these operations using plain SQL-like strings.
2. **Usage**:
   - It accepts one or more SQL expressions as strings and returns a new DataFrame. This is great for users who come from an SQL background, as they can leverage their existing knowledge.
3. **Examples**:
   - **Renaming a Column**:

pythonCopy code
```
df.selectExpr("oldColumnName as newColumnName")
```

   - **Performing Arithmetic Operations**:

pythonCopy code
```
df.selectExpr("column1 + 5", "column2 - column3")
```

   - **Using SQL Functions**:

pythonCopy code
```
df.selectExpr("avg(column1)", "max(column2)", "count(distinct column3)")
```

4. **Advantages**:
   - **Compactness**: Instead of chaining multiple transformations, you can perform multiple operations in one **selectExpr**.
   - **Readability**: For those familiar with SQL, using SQL-like expressions can be more readable.
   - **Flexibility**: You can combine column operations, functions, and SQL-like queries together.
5. **Considerations**:
   - While it's powerful, **selectExpr** should be used judiciously. Over-reliance or extremely complex expressions can lead to code that's harder to debug and maintain compared to the more structured DataFrame API operations.
   - Always ensure SQL injections aren't possible when using dynamic content with **selectExpr**.

User-Defined Functions (UDFs)

Real-time Number Doubler:

Scenario: For each number sent via a socket, this Spark streaming application will double the number using a UDF.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf
from pyspark.sql.types import DoubleType

# Initialize a Spark session
spark = SparkSession.builder \
    .appName("NumberDoubler") \
    .getOrCreate()

# Define a simple UDF to double a number
def double_number(num):
    try:
        return float(num) * 2
    except:
        return None  # Return None if the conversion to float fails

# Register the UDF with Spark
double_udf = udf(double_number, DoubleType())

# Define the streaming input DataFrame to read data from a socket
inputStream = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Use the UDF to transform the input stream
doubledStream = inputStream.select(double_udf(inputStream.value).alias("doubled_value"))

# Output the transformed data to the console
query = doubledStream.writeStream \
    .outputMode("append") \
    .format("console") \
    .start()

# Await termination of the query
query.awaitTermination()
```

type of output mode

the compatibility between output modes and sink formats is vital for effective usage of Spark structured streaming. Here's a detailed breakdown:

1. **Console Sink**:
    - **Append**: Supported
    - **Complete**: Supported

    - **Update**: Supported
2. **Memory Sink**:
    - **Append**: Supported
    - **Complete**: Supported
    - **Update**: Supported
3. **File Sink** (This includes sinks like Parquet, CSV, JSON, and ORC):
    - **Append**: Supported
    - **Complete**: Not Supported
    - **Update**: Not Supported
4. **Kafka Sink**:
    - **Append**: Supported
    - **Complete**: Not Supported
    - **Update**: Not Supported
5. **Delta Lake Sink**:
    - **Append**: Supported
    - **Complete**: Supported
    - **Update**: Supported
6. **JDBC Sink** (like writing to a database):
    - **Append**: Supported
    - **Complete**: Not Supported
    - **Update**: Not Supported

Please note that while some formats might technically support a particular mode, there can be restrictions based on the operations you're performing in the streaming query. For instance, as you've seen, aggregations without watermarking can't be written to CSV in "append" mode.

To summarize:

- **Append mode** is the most versatile and widely supported. You can add new records to the output sink, and it's supported by most formats.
- **Complete mode** rewrites the entire output sink every time and is mostly supported by in-memory and console sinks. Delta Lake, a recent and popular addition to the big data ecosystem, also supports this mode.
- **Update mode** only writes the rows that have changed since the last output, and its support is limited. Console and memory sinks support this mode. Delta Lake, again, provides broader support for this mode than most other file-based formats.

Always consider the specific requirements of your application and the characteristics of your data when choosing an output mode. For example, if your data includes late-arriving records, using watermarking with update mode can help manage and process that data correctly.

How prepare environment for csv output

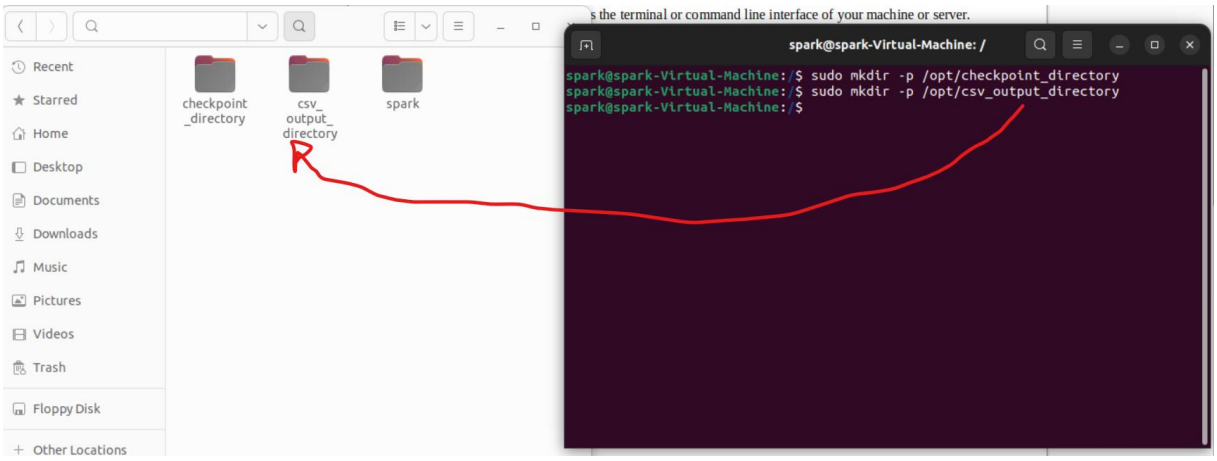
Csv output mode

1.Go to root directory, so you can run bellow commands in the terminal.

mkdir -p /opt/csv_output_directory

mkdir -p /opt/checkpoint_directory

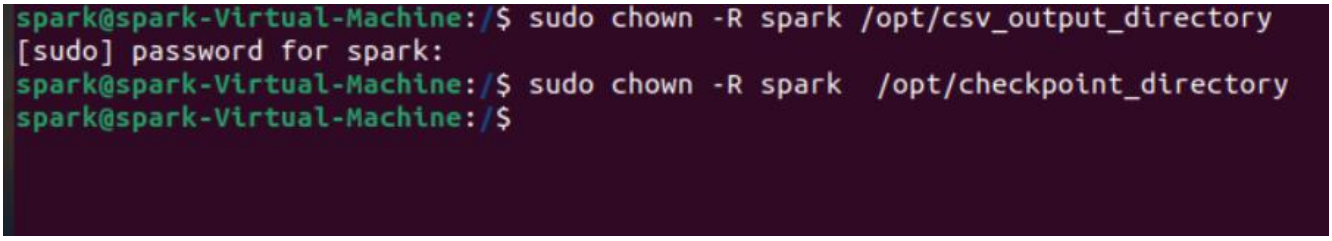
these to line create two folder as you can see in bellow



2.Set Permissions: If needed (for example, if you're running **Spark** as the spark user), set the necessary permissions:

sudo chown -R **spark** /opt/csv_output_directory

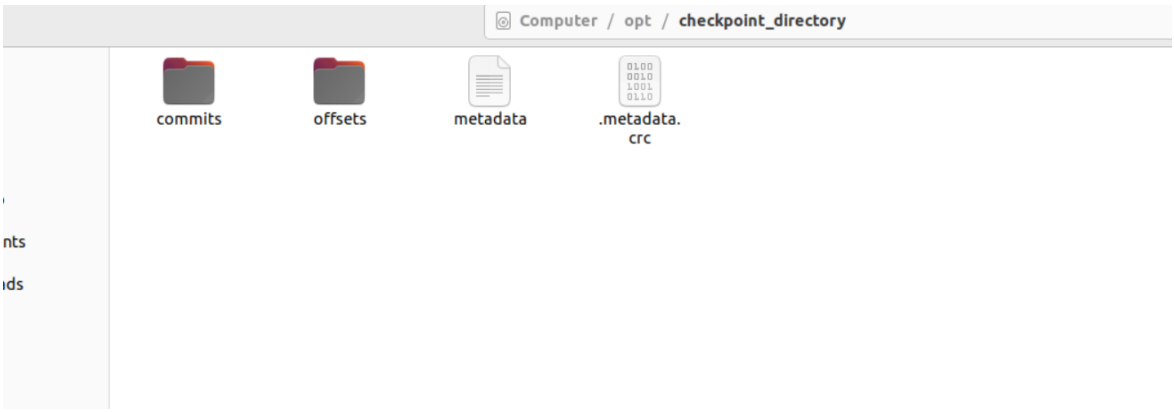
sudo chown -R **spark** /opt/checkpoint_directory



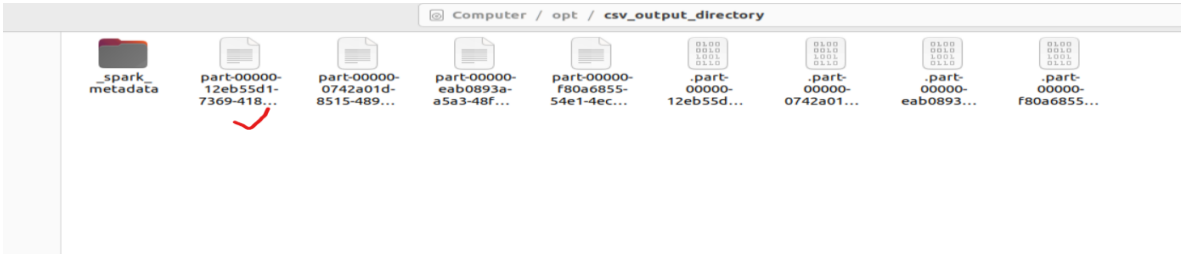
3.Run bellow code look like previous examples:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split

# Create a SparkSession
spark = SparkSession.builder \
    .appName("AppendModeCSVExampleRaw") \
    .getOrCreate()

# Define the Input Source (Socket)
streamingInputDF = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Tokenize each line into words without aggregation
words = streamingInputDF.select(explode(split(streamingInputDF.value, " ")).alias("word"))

# Define the Output Sink (CSV) with "append" output mode
query = words \
    .writeStream \
    .outputMode("append") \
    .format("csv") \
    .option("path", "/opt/csv_output_directory") \
    .option("checkpointLocation", "/opt/checkpoint_directory") \
    .start()

# Start and Await Termination
query.awaitTermination()
```

4.You can see result in bellow pictures



5.If you open these files you can see the result:



When you run stream code, you can see something like this I will explain about that

```
23/09/27 21:24:10 INFO MicroBatchExecution: Streaming query made progress: {
  "id" : "91ca89eb-6e4c-4dd7-9ea8-9d1c74a38323",
  "runId" : "da5e6b53-8f02-4f91-874b-986067b35c8c",
  "name" : null,
  "timestamp" : "2023-09-27T19:24:10.131Z",
  "batchId" : 11,
  "numInputRows" : 0,
  "inputRowsPerSecond" : 0.0,
  "processedRowsPerSecond" : 0.0,
  "durationMs" : {
    "latestOffset" : 0,
    "triggerExecution" : 0
  },
  "stateOperators" : [ ],
  "sources" : [ {
    "description" : "TextSocketV2[host: localhost, port: 9999]",
    "startOffset" : 9,
    "endOffset" : 9,
    "latestOffset" : 9,
    "numInputRows" : 0,
    "inputRowsPerSecond" : 0.0,
    "processedRowsPerSecond" : 0.0
  } ],
  "sink" : {
    "description" : "org.apache.spark.sql.execution.streaming.ConsoleTable$@240b03c1",
    "numOutputRows" : 0
  }
```

This is a JSON representation of the streaming query's progress in Spark:

1. **id** & **runId**:
    - These are unique identifiers for the query and its run. The id is consistent across multiple runs of a query (through restarts), while runId is unique for every start of the query.
2. **name**:
    - This would be the name of the query if you gave it one, but in your case, it's null.
3. **timestamp**:
    - This is the time when the progress report was generated.
4. **batchId**:
    - Each streaming source processes data in batches. This is the ID of the current batch.
5. **numInputRows** & **inputRowsPerSecond**:
    - These indicate how many rows of data were read in the current batch and the rate at which data was read, respectively.
6. **processedRowsPerSecond**:
    - This tells you the rate at which rows are being processed.
7. **durationMs**:
    - A dictionary that provides timings (in milliseconds) for various stages:
        - **latestOffset**: Time taken to get the latest data offsets.
        - **triggerExecution**: Time taken for the entire trigger to complete.
8. **stateOperators**:
    - If your query had stateful operations (like windowed aggregations or joins with watermarks), this list would contain metrics about them. In your case, it's empty.
9. **sources**:
    - This is a list of sources from which the streaming query is reading. For your query, there's only one source:
        - **description**: Describes the source, which in your case is a socket reading from localhost at port 9999.
        - **startOffset** & **endOffset**: Offsets indicating where reading began and ended for this batch.
        - **latestOffset**: The latest known offset in the source.
        - **numInputRows** & **inputRowsPerSecond**: Similar to the top-level keys but specific to this source.
        - **processedRowsPerSecond**: Same as the top-level key but specific to this source.
10. **sink**:
    - This is where the processed data is being sent. Your data is being sent to a console sink.
        - **description**: Description of the sink.
        - **numOutputRows**: Number of rows outputted in this batch.

From the looks of it, the report indicates that during this batch (batchId 11), no new rows were read, and hence no rows were processed or written to the sink (the console in your case).

Additional example

Real-time Line Reverser:

Scenario: For each line of text you send via a socket, this Spark streaming application will reverse the order of the characters in the line and display the reversed line.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import reverse

# Initialize a Spark session
spark = SparkSession.builder \
    .appName("RealTimeLineReverser") \
    .getOrCreate()

# Define the streaming input DataFrame to read data from a socket
inputStream = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Reverse each line's characters
reversedLines = inputStream.select(reverse(inputStream.value).alias("reversed_line"))

# Output the reversed lines to the console
query = reversedLines.writeStream \
    .outputMode("append") \
    .format("console") \
    .start()

# Await termination of the query
query.awaitTermination()
```

UDF example

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf, col, when, length
from pyspark.sql.types import StringType, IntegerType

# Initialize a more configured Spark session
spark = SparkSession.builder \
    .appName("EnhancedUDFExample") \
    .config("spark.executor.memory", "2g") \
    .config("spark.driver.memory", "2g") \
    .getOrCreate()

# Define a complex UDF that does text processing
def complex_text_processing(input_str):
    # Simple text processing: reversing and converting to upper case
    return "Processed: " + input_str[::-1].upper()

# Register the UDF with Spark
complex_text_udf = udf(complex_text_processing, StringType())

# Another UDF to calculate the length of a string
def string_length(input_str):
    return len(input_str)

length_udf = udf(string_length, IntegerType())

# Define the streaming input DataFrame to read data from a socket
inputStream = spark.readStream \
    .format("socket") \
```

```
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Use the UDFs to transform the input stream
transformedStream = inputStream.select(
    complex_text_udf(inputStream.value).alias("processed_text"),
    length_udf(inputStream.value).alias("original_length")
)

# Further transformation: marking if original length is > 10
transformedStream = transformedStream.withColumn(
    "long_text",
    when(col("original_length") > 10, "Yes").otherwise("No")
)

# Output the transformed data to the console
query = transformedStream.writeStream \
    .outputMode("append") \
    .format("console") \
    .start()

# Await termination of the query
query.awaitTermination()
```

Real-time Filter of Values Above 50:

Scenario: Accept numerical strings via a socket and filter out those values that are greater than 50. The result will be shown in real-time.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

spark = SparkSession.builder \
    .appName("ValueFilterAbove50") \
    .getOrCreate()

# Define the streaming input DataFrame to read data from a socket
inputStream = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Convert the string values to integers
integers = inputStream.selectExpr("CAST(value AS INT) AS number")

# Filter numbers greater than 50
filteredNumbers = integers.filter(col("number") > 50)

# Output the filtered numbers to the console
query = filteredNumbers.writeStream \
    .outputMode("append") \
    .format("console") \
    .start()

# Await termination of the query
query.awaitTermination()
```

1. **Setup Socket**: Start a netcat socket with the command: nc -lk 9999.

2. **Run the Spark Application**: Execute the provided Python script.
3. **Send Data**: Type numbers (as strings) into the nc terminal, one at a time (e.g., 45, 55, 10, 60, etc.).
4. **View Results**: The Spark application's console will display only the numbers that are greater than 50, i.e., from the above inputs, only 55 and 60 would be displayed in the console.

This example showcases how to perform a simple transformation on the data (casting a string to an integer) and then a filter operation to select specific values.

Real-time SQL Filtering on Streaming Data:

Scenario: use SQL to filter out and display only the cities that are experiencing temperatures above a specific threshold.

```
from pyspark.sql import SparkSession

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("StreamingSQLTemperatureFilter") \
    .getOrCreate()

# Connect to the socket source
# Assuming each line of input is in the format: city,temperature (e.g., "New York,75")
streamingInputDF = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Split the input lines into city and temperature columns
splitDF = streamingInputDF.selectExpr("split(value, ',')[0] as city", "split(value, ',')[1] as temperature")

# Register the DataFrame with split data as a temporary view
splitDF.createOrReplaceTempView("city_temperatures")

# Use SQL to filter cities with temperature above 80
hotCities = spark.sql("SELECT city, temperature FROM city_temperatures WHERE CAST(temperature AS INT) > 80")

# Output the results to console
query = hotCities.writeStream \
    .outputMode("append") \
    .format("console") \
    .start()

# Keep the application running until terminated
query.awaitTermination()
```

Instructions:

1. Start a socket using nc -lk 9999 before you run this Spark application.
2. Execute the provided Python code.
3. In the nc terminal, input city-temperature data in the format: city,temperature. For example: New York,85 or Los Angeles,78.
4. The Spark application's console will display only cities that have temperatures above 80 degrees.

Real-time SQL Aggregation on Streaming Data:

Scenario: We'll simulate receiving real-time data of sales transactions, each transaction indicating a product and its sold quantity. We'll group by product and sum up the quantities to see real-time sales per product.

```
from pyspark.sql import SparkSession

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("StreamingSQLProductSales") \
    .getOrCreate()

# Connect to the socket source
# Assuming each line of input is in the format: product,quantity (e.g., "Laptop,5")
streamingInputDF = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Split the input lines into product and quantity columns
splitDF = streamingInputDF.selectExpr("split(value, ',')[0] as product", "split(value, ',')[1] as quantity")

# Register the DataFrame with split data as a temporary view
splitDF.createOrReplaceTempView("product_sales")

# Use SQL to group by product and sum the quantities
salesPerProduct = spark.sql("SELECT product, SUM(CAST(quantity AS INT)) as total_sales FROM product_sales GROUP BY product")

# Output the results to console
query = salesPerProduct.writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

# Keep the application running until terminated
query.awaitTermination()
```

Instructions:

1. Start a socket using nc -lk 9999 before you run this Spark application.
2. Execute the provided Python code.
3. In the nc terminal, input sales data in the format: product,quantity. For example: Laptop,5 or Phone,3.
4. The Spark application's console will display accumulated sales per product in real-time.

count of total words received so far in real-time.

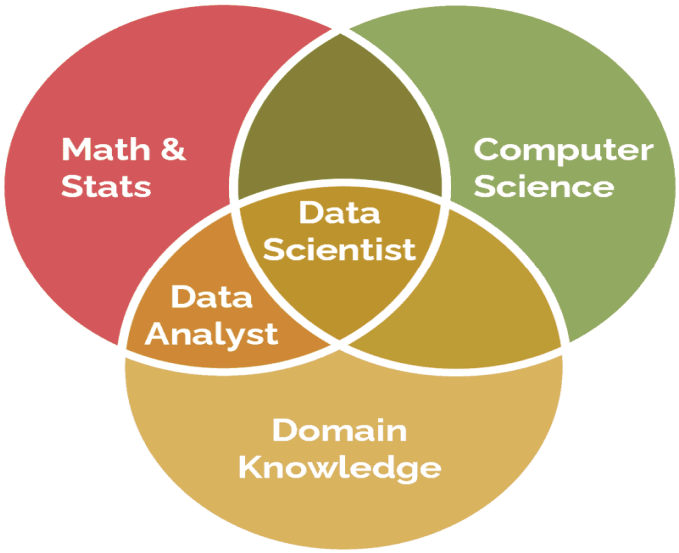
Real-time Total Word Count with "Complete" Mode:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("StreamingSQLCompleteWordCount") \
    .getOrCreate()

# Connect to the socket source
# Expecting lines of text from the input
streamingInputDF = spark.readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Tokenize the input lines into words
words = streamingInputDF.select(explode(split(streamingInputDF.value, " ")).alias("word"))

# Register the DataFrame of words as a temporary view
words.createOrReplaceTempView("words_table")

# Use SQL to count the total number of words received
wordCount = spark.sql("SELECT 'Total Words' as Description, COUNT(word) as Count FROM words_table")

# Output the results to console using the "complete" mode
query = wordCount.writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

# Keep the application running until terminated
query.awaitTermination()
```

Instructions:

1. Start a socket using nc -lk 9999 before you run this Spark application.
2. Execute the provided Python code.
3. In the nc terminal, input lines of text.
4. The Spark application's console will display the cumulative word count from the input data in real-time, updating with each new line of text you input.

Enjoy your LAB


Math &
Stats
Computer
Science
Data
Scientist
Data
Analyst
Domain
Knowledge