My system is on low end so adding all the tech stack would have made my system lag I have given the implementation please look at it

In [ ]:
```
from pyspark.sql import SparkSession

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("Streaming JSON Files") \
    .getOrCreate()

# Directory where the JSON files are stored locally
input_path = "./data"

# Read JSON files as a static DataFrame to infer schema
static_df = spark.read \
    .format("json") \
    .option("inferSchema", "true") \
    .load(input_path)

# Extract the inferred schema from the static DataFrame
schema = static_df.schema

# Read JSON files as a streaming DataFrame with inferred schema
streaming_df = spark.readStream \
    .schema(schema) \
    .format("json") \
    .option("maxFilesPerTrigger", 1) \
    .load(input_path)

# Define the output Delta Lake path
output_path = "./delta_files"

# Write the streaming DataFrame to the Delta Lake path
query = streaming_df.writeStream \
    .format("delta") \
    .outputMode("append") \
    .option("checkpointLocation", "./checkpoint_dir") \
    .option("path", output_path) \
```

```
    .start()

# Await termination to keep the streaming query running
query.awaitTermination()
```

```
---------------------------------------------------------------------------
Py4JJavaError                             Traceback (most recent call last)
Cell In[26], line 36
     28 output_path = "./delta_files"
     30 # Write the streaming DataFrame to the Delta Lake path
     31 query = streaming_df.writeStream \
     32     .format("delta") \
     33     .outputMode("append") \
     34     .option("checkpointLocation", "./checkpoint_dir") \
     35     .option("path", output_path) \
---> 36     .start()
     38 # Await termination to keep the streaming query running
     39 query.awaitTermination()

File c:\Python312\Lib\site-packages\pyspark\sql\streaming\readwriter.py:1527, in DataStreamWriter.start(self, path, format, outputMode, partitionBy, queryName, **options)
   1525     self.queryName(queryName)
   1526 if path is None:
-> 1527     return self._sq(self._jwrite.start())
   1528 else:
   1529     return self._sq(self._jwrite.start(path))

File c:\Python312\Lib\site-packages\py4j\java_gateway.py:1322, in JavaMember.__call__(self, *args)
   1316 command = proto.CALL_COMMAND_NAME +\
   1317     self.command_header +\
   1318     args_command +\
   1319     proto.END_COMMAND_PART
   1321 answer = self.gateway_client.send_command(command)
-> 1322 return_value = get_return_value(
   1323     answer, self.gateway_client, self.target_id, self.name)
   1325 for temp_arg in temp_args:
   1326     if hasattr(temp_arg, "_detach"):

File c:\Python312\Lib\site-packages\pyspark\errors\exceptions\captured.py:179, in capture_sql_exception.<locals>.deco(*a, **kw)
    177 def deco(*a: Any, **kw: Any) -> Any:
    178     try:
--> 179         return f(*a, **kw)
    180     except Py4JJavaError as e:
    181         converted = convert_exception(e.java_exception)
```

```
File c:\Python312\Lib\site-packages\py4j\protocol.py:326, in get_return_value(answer, gateway_client, target_id, name)
    324 value = OUTPUT_CONVERTER[type](answer[2:], gateway_client)
    325 if answer[1] == REFERENCE_TYPE:
--> 326     raise Py4JJavaError(
    327         "An error occurred while calling {0}{1}{2}.\n".
    328         format(target_id, ".", name), value)
    329 else:
    330     raise Py4JError(
    331         "An error occurred while calling {0}{1}{2}. Trace:\n{3}\n".
    332         format(target_id, ".", name, value))

Py4JJavaError: An error occurred while calling o232.start.
: org.apache.spark.SparkClassNotFoundException: [DATA_SOURCE_NOT_FOUND] Failed to find the data source: delta. Please find pack
ages at `https://spark.apache.org/third-party-projects.html`.

        at org.apache.spark.sql.errors.QueryExecutionErrors$.dataSourceNotFoundError(QueryExecutionErrors.scala:724)

        at org.apache.spark.sql.execution.datasources.DataSource$.lookupDataSource(DataSource.scala:647)

        at org.apache.spark.sql.streaming.DataStreamWriter.startInternal(DataStreamWriter.scala:370)

        at org.apache.spark.sql.streaming.DataStreamWriter.start(DataStreamWriter.scala:251)

        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)

        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)

        at java.lang.reflect.Method.invoke(Unknown Source)

        at py4j.reflection.MethodInvoker.invoke(MethodInvoker.java:244)

        at py4j.reflection.ReflectionEngine.invoke(ReflectionEngine.java:374)

        at py4j.Gateway.invoke(Gateway.java:282)

        at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)

        at py4j.commands.CallCommand.execute(CallCommand.java:79)
```

```
        at py4j.ClientServerConnection.waitForCommands(ClientServerConnection.java:182)

        at py4j.ClientServerConnection.run(ClientServerConnection.java:106)

        at java.lang.Thread.run(Unknown Source)

Caused by: java.lang.ClassNotFoundException: delta.DefaultSource

        at java.net.URLClassLoader.findClass(Unknown Source)

        at java.lang.ClassLoader.loadClass(Unknown Source)

        at java.lang.ClassLoader.loadClass(Unknown Source)

        at org.apache.spark.sql.execution.datasources.DataSource$.$anonfun$lookupDataSource$5(DataSource.scala:633)

        at scala.util.Try$.apply(Try.scala:213)

        at org.apache.spark.sql.execution.datasources.DataSource$.$anonfun$lookupDataSource$4(DataSource.scala:633)

        at scala.util.Failure.orElse(Try.scala:224)

        at org.apache.spark.sql.execution.datasources.DataSource$.lookupDataSource(DataSource.scala:633)

        ... 14 more
```

This Python script demonstrates how to use PySpark to process JSON files as a streaming DataFrame and write the output to Delta Lake:

Step-by-Step Explanation:

1. **Initialize SparkSession:**
   - `spark = SparkSession.builder \ ... \ .getOrCreate()` : Initializes a SparkSession with the name "Streaming JSON Files". SparkSession is the entry point to Spark functionality and provides a way to interact with various Spark APIs.
2. **Define Input and Output Paths:**
   - `input_path = "./data"` : Specifies the directory path where JSON files are stored locally.
   - `output_path = "./delta_files"` : Specifies the directory path where the Delta Lake files will be written.

3. **Read JSON Files as a Static DataFrame (Infer Schema):**

   - `static_df = spark.read \ ... \ .load(input_path)` : Reads JSON files from `input_path` into a static DataFrame (`static_df`). The `inferSchema` option is set to `true` , allowing Spark to automatically infer the schema of the JSON files.

4. **Extract the Inferred Schema:**

   - `schema = static_df.schema` : Extracts the schema inferred from the static DataFrame (`static_df`). This schema will be used to define the structure of the streaming DataFrame.

5. **Read JSON Files as a Streaming DataFrame:**

   - `streaming_df = spark.readStream \ ... \ .load(input_path)` : Reads JSON files as a streaming DataFrame (`streaming_df`). The schema inferred from the static DataFrame (`schema`) is applied to ensure consistent schema across batch and streaming processing.

6. **Write Streaming DataFrame to Delta Lake:**

   - `query = streaming_df.writeStream \ ... \ .start()` : Writes the streaming DataFrame (`streaming_df`) to Delta Lake format. Several options are specified:
     - `format("delta")` : Specifies the output format as Delta Lake.
     - `outputMode("append")` : Defines the output mode as append, which adds new data to the Delta table.
     - `option("checkpointLocation", "./checkpoint_dir")` : Provides the directory path for checkpointing. Checkpointing is essential for fault tolerance and ensuring exactly-once processing semantics.
     - `option("path", output_path)` : Specifies the directory path where the Delta Lake files will be stored.

7. **Await Termination of the Streaming Query:**

   - `query.awaitTermination()` : Initiates the streaming query (`query`) and waits for its termination. This keeps the streaming query running indefinitely until terminated manually.

Purpose:

This script is designed to handle streaming JSON data using Apache Spark's structured streaming capabilities. It leverages Delta Lake for reliable, scalable, and performant data storage. Delta Lake provides ACID transactions, schema enforcement, and compatibility with existing data lakes, making it suitable for handling real-time data ingestion and analytics use cases.

Considerations:

- **Schema Evolution**: Ensure that any changes in the structure of incoming JSON files are compatible with the inferred schema or handle schema evolution appropriately.
- **Performance**: Adjust parameters like `maxFilesPerTrigger` to control the rate at which new data is processed, balancing between latency and throughput requirements.
- **Checkpointing**: Proper checkpointing is crucial for fault tolerance in streaming applications. Ensure the checkpoint directory (`checkpointLocation`) is accessible and has sufficient storage capacity.

By following this approach, you can effectively process streaming JSON data with Spark, leverage Delta Lake for data reliability and performance, and integrate streaming analytics seamlessly into your data pipeline.