Experiment 10

Aim: To perform Batch and Streamed Data Analysis using Apache Spark.

Theory:

1. What is Streaming? Batch vs. Stream Data

Batch Data Processing

Definition: Processing a large volume of data at once (in batches) at scheduled intervals.

Characteristics:

- Data is collected over time and processed in chunks.
- High latency (delay between data collection and processing).
- Suitable for applications where real-time analysis is not required (e.g., daily sales reports).

Example:

Running an end-of-day report on transactions stored in a database.

Stream Data Processing

Definition: Processing data in real-time as it is generated.

Characteristics:

- Data is processed continuously with minimal latency.
- Used for real-time analytics (e.g., fraud detection, live dashboards).

Example:

Analyzing live Twitter feeds for trending topics.

Feature	**Batch Processing**	**Stream Processing**
Data Input	Collected over time	Continuous, real-time
Latency	High (minutes/hours)	Low (seconds/milliseconds)
Use Cases	Reports, historical analysis	Fraud detection, live monitoring
Tools	Hadoop MapReduce, Spark Batch	Spark Streaming, Kafka, Flink

2. How Data Streaming Works in Apache Spark

Apache Spark provides Spark Streaming (now part of Structured Streaming) for real-time data processing.

Key Concepts in Spark Streaming:

- DStream (Discretized Stream):
  - A sequence of RDDs (Resilient Distributed Datasets) representing data in small time intervals (micro-batches).
- Structured Streaming:
  - A higher-level API built on Spark SQL for real-time processing with DataFrame/Dataset abstractions.

Steps for Batch Processing

1. Initialize Spark Session

```
from pyspark.sql import SparkSession
# Create a SparkSession
spark = SparkSession.builder \
    .appName("BatchProcessingExample") \
    .getOrCreate()
```

2. Read Batch Data (CSV, JSON, Parquet, etc.)

```
# Reading from a CSV file
batch_df = spark.read \
    .format("csv") \
    .option("header", "true") \
    .option("inferSchema", "true") \
    .load("path/to/data.csv")
```

3. Apply Transformations (Filtering, Aggregations, Joins, etc.)

```
from pyspark.sql.functions import col, count, avg
# Example: Filtering and aggregation
filtered_df = batch_df.filter(col("age") > 30)
# GroupBy and Aggregation (e.g., average salary by department)
agg_df = batch_df.groupBy("department") \
    .agg(
        count("*").alias("employee_count"),
        avg("salary").alias("avg_salary")
    )
```

4. Output the Processed Data (Save to Disk, Database, etc.)
   ```
   # Write to CSV
   agg_df.write \
      .format("csv") \
      .mode("overwrite") \  # Options: "append", "overwrite", "ignore"
      .option("header", "true") \
      .save("output/path")
   ```

5. Stop Spark Session
   ```
   spark.stop()
   ```

Steps for Stream Processing

1. Initialize Spark Session
   ```
   from pyspark.sql import SparkSession
   spark = SparkSession.builder.appName("StreamingExample").getOrCreate()
   ```

2. Read Streaming Data (from Kafka, Socket, Files, etc.)
   ```
   # Reading from a socket (for testing)
   streaming_df = spark.readStream.format("socket").option("host",
   "localhost").option("port", 9999).load()
   ```

3. Apply Transformations (Filtering, Aggregations, etc.)
   ```
   # Example: Word count on streaming text
   from pyspark.sql.functions import explode, split
   words = streaming_df.select(explode(split("value", " ")).alias("word"))
   word_counts = words.groupBy("word").count()
   ```

4. Output the Stream (Console, Kafka, HDFS, etc.)
   ```
   query =
   word_counts.writeStream.outputMode("complete").format("console").start()
   ```

5. Start and Manage the Stream
   ```
   query.awaitTermination()  # Keeps the stream running
   ```

Conclusion:

In this experiment, we explored Batch and Stream Data Processing using Apache Spark.

Batch Processing

- Used for static, large-scale datasets (e.g., historical data).
- Processes data in fixed intervals (e.g., hourly/daily jobs).
- Ideal for ETL, analytics, and reporting.
- Example: spark.read.csv() → Transformations → df.write.save().

Stream Processing

- Handles real-time, continuous data (e.g., live logs, IoT sensors).
- Processes data in micro-batches or event-by-event.
- Used for fraud detection, live monitoring, alerts.
- Example: spark.readStream → Transformations → writeStream.start().