Experiment 10

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

Theory:

1. What is streaming? Explain batch and stream data.

Data Processing Paradigms: Batch vs Stream

Data processing systems generally fall into two major paradigms: batch processing and stream processing. These approaches differ fundamentally in how they handle data and when they produce results.

Batch Processing refers to the technique of collecting data over a period of time, storing it, and then processing it as a single unit or "batch." In this approach:

- Data is accumulated over time before processing begins
- Processing occurs on a predefined schedule (hourly, daily, weekly)
- Results are generated after the entire dataset has been processed
- It's well-suited for applications where insights aren't needed in real-time
- Examples include end-of-day financial reports, customer segmentation analysis, and periodic business intelligence processing

Batch processing is analogous to doing laundry - you collect dirty clothes for several days, then wash them all at once when the basket is full.

Stream Processing handles data as it arrives, processing each data item individually or in small time-windowed batches. Key characteristics include:

- Data is processed continuously as it's generated
- Results are updated in near real-time
- Processing must handle unbounded datasets (potentially infinite data flows)
- It's ideal for applications requiring immediate insights or actions
- Examples include fraud detection, IoT sensor monitoring, real-time recommendations, and social media trend analysis

Stream processing resembles a flowing river where water (data) is constantly moving and being processed as it passes through various points.

The primary differences between batch and stream processing center around timing (when processing occurs), data scope (finite vs. infinite), latency requirements (delayed vs. immediate results), and processing model (stateless vs. stateful computation).

2. How data streaming takes place using Apache Spark.

Data Streaming with Apache Spark

Apache Spark offers robust capabilities for stream processing through its Spark Streaming and newer Structured Streaming APIs. The streaming process in Spark follows these key steps:

Step 1: Setting Up the Streaming Environment

- Import necessary libraries and create a SparkSession configured for streaming

For Structured Streaming (recommended):

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \

 .appName("Spark Streaming Application") \

 .config("spark.streaming.stopGracefullyOnShutdown", "true") \

 .getOrCreate()
```

Step 2: Defining Input Sources

- Connect to streaming data sources such as:
    - Apache Kafka for message queuing
    - Kinesis for AWS environments
    - Socket connections for simple testing
    - File-based streaming (monitoring directories)

Example with Kafka source:

```
df = spark.readStream \

 .format("kafka") \

 .option("kafka.bootstrap.servers", "host:port") \

 .option("subscribe", "topic_name") \

 .load()
```

Step 3: Data Transformation

Apply schema to structured data:

```
from pyspark.sql.types import StructType, StructField, StringType, TimestampType
```

schema = StructType([...])

parsed_df = df.selectExpr("CAST(value AS STRING)") \

.select(from_json(col("value"), schema).alias("data")) \

.select("data.*")

- Perform transformations using DataFrame operations:
    - Filter records with `filter()` or `where()`
    - Aggregate data with `groupBy()` and aggregation functions
    - Enrich data by joining with static datasets
    - Apply window functions for time-based analysis

Step 4: Processing Models Spark Streaming offers two processing models:

1. **Micro-batch Processing** (Default):
    - Processes data in small, discrete time-windowed batches
    - Provides exactly-once processing semantics
    - Offers higher throughput at the cost of slightly higher latency
2. **Continuous Processing** (Low-latency mode):
    - Processes each record as it arrives
    - Provides at-least-once semantics
    - Achieves lower latency (as low as 1ms) but with reduced throughput

Step 5: State Management

Maintain state across batches for complex analytics: # Word count example with state

wordCounts = df.groupBy("word") \

.count()

Define watermarks for handling late data:

df = df.withWatermark("timestamp", "10 minutes") \

.groupBy(window("timestamp", "5 minutes"), "id") \

.count()

Step 6: Output Sinks

Define where processed results should be written:

query = wordCounts.writeStream \

.outputMode("complete") \

.format("console") \

.start()

- Common output options include:
    - Console (for debugging)
    - Memory (for querying from another thread)
    - File sinks (Parquet, CSV, etc.)
    - Kafka topics
    - Custom sinks via the ForeachBatch API

Step 7: Stream Execution and Monitoring

- Start the streaming query:
  query.awaitTermination()

Monitor streaming statistics:
query.status

query.lastProgress

query.recentProgress

Step 8: Handling Failures and Recovery

Implement checkpoint directories for fault tolerance:
query = df.writeStream \

.outputMode("append") \

.format("parquet") \

.option("checkpointLocation", "/path/to/checkpoint/dir") \

.start("/output/path")

- Spark automatically recovers from failures using checkpoint information

Conclusion:

Apache Spark provides a comprehensive framework for processing both batch and streaming data at scale. The experiment demonstrates how Spark's unified processing model bridges these two paradigms, allowing organizations to implement solutions that handle historical data analysis alongside real-time data processing using largely the same codebase.

Spark Streaming, particularly the newer Structured Streaming API, significantly simplifies building resilient streaming applications by abstracting away many of the complexities associated with stream processing such as fault tolerance, exactly-once semantics, and state management. By treating streaming computations as continuous queries on unbounded tables, Spark makes stream processing more accessible to data engineers and scientists already familiar with batch processing concepts.

The integration of streaming capabilities within the broader Spark ecosystem enables powerful hybrid architectures that combine batch processing for historical analysis with stream processing for real-time insights. This flexibility allows organizations to implement Lambda or Kappa architectural patterns depending on their specific requirements.

For production deployments, organizations should carefully consider configuration parameters related to micro-batch intervals, memory allocation, and checkpoint strategies to optimize performance and reliability. Additionally, integration with monitoring systems is essential for tracking streaming application health and performance metrics over time.

As data volumes continue to grow and real-time decision-making becomes increasingly important across industries, mastering tools like Spark Streaming represents an essential capability for modern data teams seeking to derive timely insights from continuously flowing data.