Scalable and Cloud Programming

Co-purchase Analysis Project

- **Introduction**

This project implements a distributed co-purchase analysis system using Apache Spark, designed to identify product affinity patterns from retail transaction data. The solution processes order records to determine how frequently pairs of items are purchased together, valuable information for recommendation systems and inventory optimization.

Apache Spark serves as the foundational framework for this analysis, providing:

- **Parallel processing** capabilities across clustered systems
- **In-memory computation** for iterative algorithms
- **Fault tolerance** through resilient distributed datasets (RDDs)
- **Scalable architecture** that handles growing data volumes efficiently

Subsequent sections will evaluate the system's performance and scalability characteristics when processing the datasets across different cluster configurations. The technical approach focuses on maintaining computational efficiency while ensuring result accuracy in distributed environments.

- **Approach Used:**

The solution processes co-purchase data through a distributed pipeline:

1. **Data Ingestion**
   Loads order-product pairs from CSV source files
2. **Order Consolidation**
   Groups all products by their originating order ID
3. **Pair Formation**
   Generates unique product combinations per order
   (ensures x < y to avoid duplicate pairs)
4. **Frequency Analysis**
   Counts co-occurrences across all orders
5. **Result Export**
   Stores final pair frequencies as CSV in cloud storage

In the next page, I show the commented code:

```scala
import org.apache.spark.sql.{SparkSession, functions => F}

object OrderProductsCoPurchaseAnalysis {

  def main(args: Array[String]): Unit = {
    // Define paths directly in code
    val gcsInputPath = "gs://bucket-scala-pablo/order_products.csv" // Path to CSV file in GCS
    val gcsOutputPath = "gs://bucket-scala-pablo/output/"           // Output path in GCS

    // Create Spark session
    val spark = SparkSession.builder()
      .appName("OrderProductsCoPurchaseAnalysis")
      .getOrCreate()

    // Read CSV file from GCS
    val data = spark.read
      .option("header", "false") // File has no headers
      .option("inferSchema", "true") // Infer schema automatically
      .csv(gcsInputPath)
      .toDF("order_id", "product_id") // Rename columns

    // Display loaded data (for debugging)
    data.show()
```

```scala
    // Generate product pairs for each order
    val productPairs = data
      .join(data, "order_id") // Combine products within each order
      .filter(F.col("product_id") < F.col("product_id_2")) // Avoid reverse pairs
      .select(F.col("product_id").alias("x"), F.col("product_id_2").alias("y"))

    // Count occurrences of each product pair
    val coPurchaseCounts = productPairs
      .groupBy("x", "y")
      .agg(F.count("*").alias("n"))

    // Save results as CSV in GCS
    coPurchaseCounts.write
      .option("header", "true") // Include headers
      .mode("overwrite") // Overwrite if exists
      .csv(s"$gcsOutputPath/co_purchase_analysis")
```

```scala
    // Stop Spark
    spark.stop()
  }
}
```

Scalability and Performance Analysis:

Execution time of the clusters:

Cluster 1	2 workers	7:13min
Cluster 2	3 workers	6:34min
Cluster 3	4 workers	6:12 min
Cluster 4	5 workers	6:08 min

Key Findings:

1. **Diminishing Returns**
   - Adding a 3rd worker brings 10% improvement
   - 4th worker adds only 6% more
   - 5th worker shows marginal gain (0.7%)
2. **Optimal Configuration**
   The 4-worker cluster offers the best balance:
   - 16% faster than baseline
   - Only 4 seconds slower than 5-worker setup
   - Better resource utilization than 5-worker cluster

Observed Patterns:

- **Non-linear Scaling**:
  Execution time improves sublinearly due to:
  - Communication overhead between workers
  - Data shuffling costs
  - Driver node bottleneck
- **Cluster Saturation**:
  Minimal improvement beyond 4 workers suggests:
  - The dataset size may be too small for 5 workers
  - Task granularity becomes too fine-grained
  - Network latency starts dominating

- **Conclusion**
  With bigger datasets, the system would show a better linear scaling.
  More data → Longer computation per worker → Better amortization of fixed overheads
  The workers would process larger partitions, and the CPU/Memory usage would remains consistently high**.**

  Link to the repository:
  https://github.com/pablosaez21/Scalable-and-Cloud-Programming-project