**Day 5 – Lab Document**

**Track:** Apache Spark Advanced Concepts  
**Focus:** DAG, RDD vs DataFrame, UDF Optimization, Cache vs Checkpoint, Cloud Reads, Pipeline Tuning

**Lab 1: DAG & Shuffle Analysis**

**Objective**  
Analyze Spark DAG and identify shuffle stages using groupBy operations.

**Dataset**

* transactions.csv (transaction\_id, product\_id, user\_id, amount, timestamp, region)

**Tasks**

1. Load transactions.csv into a Spark DataFrame.
2. Perform groupBy("product\_id").agg(sum("amount")).
3. Open Spark UI and identify the DAG stages and shuffle boundaries.
4. Optimize the job by applying repartition() before aggregation.
5. Discuss shuffle partitioning and its impact on performance.

**Expected Output**  
A DAG diagram with clear shuffle stages and insights on partition strategy.

**Lab 2: RDD vs DataFrame Performance Benchmark**

**Objective**  
Compare execution time and resource usage between RDD and DataFrame APIs.

**Dataset**

* user\_logs.json (user\_id, activity\_time, activity\_type, duration)

**Tasks**

1. Load JSON using both RDD and DataFrame APIs.
2. Aggregate total duration per user using both approaches.
3. Measure execution time using Python’s time module or Spark UI.
4. Compare logical and physical plans.
5. Explain why DataFrame is generally faster.

**Expected Output**  
A report comparing execution times and efficiency of both APIs.

**Lab 3: UDF vs pandas\_udf Profiling**

**Objective**  
Compare the performance of regular UDF and pandas\_udf in Spark.

**Dataset**

* fraud\_transactions.csv (txn\_id, amount, country, merchant, user\_id, timestamp)

**Tasks**

1. Write a fraud scoring function (e.g., return 1 if amount > threshold).
2. Implement using standard UDF.
3. Re-implement using pandas\_udf.
4. Use Spark UI to track stage and task execution.
5. Record and compare execution times and serialization overhead.

**Expected Output**  
Profiling metrics and evidence that pandas\_udf is faster for vectorized operations.

**Lab 4: Cache vs Checkpoint in a Multi-stage Pipeline**

**Objective**  
Compare Spark caching and checkpointing with respect to performance and lineage.

**Dataset**

* customer\_churn.csv (customer\_id, join\_date, last\_active, region, churned)

**Tasks**

1. Load dataset and apply multiple transformations (filter, groupBy, join).
2. Cache the intermediate DataFrame and examine reuse.
3. Replace cache with checkpoint and observe lineage truncation.
4. Use .explain() and Spark UI to assess differences.
5. Compare stage reuse, time taken, and task execution.

**Expected Output**  
Performance and lineage differences with caching vs checkpointing.

**Lab 5: Reading External Partitioned Data from Azure Blob**

**Objective**  
Read data from Azure Blob Storage and demonstrate partition pruning.

**Dataset**

* sales\_data/year=2023/month=07/\*.parquet

**Tasks**

1. Configure Spark to read from Azure Blob using spark.conf.set(...).
2. Load the partitioned Parquet folder.
3. Filter data for a single month (e.g., July).
4. Use df.explain(True) to verify partition pruning.
5. Confirm that only the relevant files are read.

**Expected Output**  
Filtered read with reduced I/O confirmed via explain plan.

**Lab 6: Optimize a Data Pipeline for 2× Speed-Up**

**Objective**  
Build and optimize a Spark ETL pipeline with tuning techniques.

**Dataset**

* big\_txns.csv (txn\_id, merchant\_id, txn\_date, amount, status)

**Tasks**

1. Build full ETL pipeline: read → filter → aggregate.
2. Record baseline execution time.
3. Apply optimizations:
   * broadcast() for small lookup joins
   * cache() reused DataFrames
   * coalesce() to reduce number of partitions
4. Measure and compare performance after optimization.
5. Capture before and after screenshots from Spark UI.

**Expected Output**  
Optimized Spark job with improved execution time by ~2×.

**Bonus Lab: Join Skew and Salting Strategy**

**Objective**  
Handle skewed joins using salting or broadcast techniques.

**Dataset**

* skewed\_data.csv (user\_id, product\_id, count)
* product\_info.csv (product\_id, category, price)

**Tasks**

1. Simulate skew by having one product\_id with high frequency.
2. Perform a join and observe shuffle delay in Spark UI.
3. Apply salting by appending/modifying join keys.
4. Alternatively, use broadcast() if product\_info is small.
5. Compare task duration and stage time before and after fix.

**Expected Output**  
Improved performance with reduced shuffle skew in joins.