Large Scale Data Management Systems – Project

Alexandros Panagiotis Stylos

Student ID: 03003260

PhD Student, School of Electrical & Computer Engineering

June 9, 2025

Project Repository on Github:

https://github.com/alexandrosst/Large-Scale-Data-Management-Systems

Query 1

The performance evaluation was conducted on the lab Spark cluster with a static configuration to ensure consistency across runs: 2 executors were used, each allocated 1 CPU core and 2 GB of memory. Each query was executed three times and the average runtime was recorded.

Spark DataFrame operations without user-defined functions (UDFs) are the most efficient, averaging 27.67 seconds. Introducing UDFs increases execution time slightly to 28 seconds—a negligible difference—likely due to limited query optimization for custom functions. In contrast, implementing the same logic with RDDs significantly extends runtime, averaging over 1 minute. Since RDDs work at a lower level and lack logical optimization, they are less efficient for structured data processing. The results are presented in Table 1.

Trial	DataFrame APIs (sec)	DataFrame with UDFs (sec)	RDDs (min)
1st	30	29	1
2nd	26	28	1.1
3rd	27	27	1
Average	**27.67**	**28**	**1.03**

Table 1: Query 1 - Execution Time Comparison for DataFrame APIs (with and without UDFs) & RDDs

Query 2

For fair comparison across the different cases, we have used 2 executors, each allocated 1 CPU core and 2 GB of memory. Each query was executed 3 times and the average runtime was calculated.

Our results indicate that both the DataFrame and SQL APIs achieve efficient execution times—20.33 seconds and 37 seconds, respectively—thanks to Catalyst's logical query optimization. In contrast, the RDD approach is noticeably slower, taking 1.67 minutes on average. These findings, presented in Table 2, highlight the superior performance of the DataFrame and SQL API.

Trial	DataFrame APIs (sec)	RDDs (min)	SQL APIs (sec)
1st	20	1.7	37
2nd	20	1.6	35
3rd	21	1.7	39
Average	**20.33**	**1.67**	**37**

Table 2: Query 2 - Execution Time Comparison for DataFrame APIs, RDDs & SQL APIs

Query 3

We evaluate the performance of reading data from CSV and Parquet file formats using Spark, comparing both DataFrames and RDDs under a fixed configuration: 2 executors, each allocated 1 CPU core and 2 GB of RAM. For each combination of file format and processing method, we conducted three trials.

As shown in Table 3, DataFrames exhibited faster read times with Parquet (25.33 seconds) compared to CSV (29.33 seconds). A similar trend was observed with RDDs, as shown in Table 4, where Parquet took 27 seconds versus 30.33 seconds for CSV.

These results align with expectations, as Parquet, a columnar format optimized for analytical workloads, provides superior read efficiency. Unlike CSV, which stores data in a row-based manner, Parquet organizes data by columns, allowing Spark to read only the necessary columns rather than scanning the entire file. This reduces I/O overhead and improves query performance, especially when working with large datasets. Additionally, Parquet applies efficient compression and encoding techniques, further minimizing storage footprint and read time. CSV, by contrast, lacks built-in metadata and requires full row scans, leading to additional parsing overhead that slightly increases execution time.

In Query 3, the optimizer chose the BroadcastHashJoin strategy for the join operation, as shown in the physical plan below. This decision was driven by the fact that the query involves two small datasets —"Median Household Income by Zip Code" and "2010 Census Populations by Zip Code" — which are well-suited for broadcasting.

```
== Physical Plan ==
AdaptiveSparkPlan (20)
+- == Final Plan ==
   CollectLimit (12)
   +- * Project (11)
      +- * BroadcastHashJoin Inner BuildLeft (10)
         :- BroadcastQueryStage (6), Statistics(sizeInBytes=1040.0 KiB, rowCount=284)
         :  +- BroadcastExchange (5)
         :     +- * Project (4)
         :        +- * Filter (3)
         :           +- * ColumnarToRow (2)
         :              +- Scan parquet  (1)
         +- * Filter (9)
            +- * ColumnarToRow (8)
               +- Scan parquet  (7)
```

DataFrame APIs		
Trial	**Parquet (sec)**	**CSV (sec)**
1st	25	29
2nd	25	30
3rd	26	29
Average	**25.33**	**29.33**

Table 3: Query 3 - DataFrame APIs Execution Time Comparison for Parquet and CSV Files

RDDs		
Trial	**Parquet (sec)**	**CSV (sec)**
1st	28	31
2nd	27	30
3rd	26	30
Average	**27**	**30.33**

Table 4: Query 3 - RDDs Execution Time Comparison for Parquet and CSV Files

Query 4

For calculating the average execution time in each configuration setup, we carried out three trials.
As shown in Table 5, our evaluation of Spark scaling configurations reveals a notable trend: increasing the number of executors (horizontal scaling) results in longer execution times. Specifically, execution time rose from 33.33 seconds with 2 executors to 43.33 seconds with 4 executors and 52.67 seconds with 8 executors. This slowdown likely stems from the overhead associated with managing additional executor processes and increased task fragmentation, which outweigh the benefits of parallelism.
On the contrary, vertical scaling—by increasing CPU cores and memory per executor—had minimal impact on performance. This suggests that the workload's resource demands are adequately met with fewer executors, and additional resources per executor do not yield significant performance improvements.

Executors	CPU (cores)	Memory (GB)	Avg Time (sec)
2	4	8	33.33
4	2	4	43.33
8	1	2	52.67
2	1	2	32.67
2	2	4	33.33
2	4	8	33.33

Table 5: Query 4 - Execution Time Comparison for Different Configurations (Horizontal & Vertical Scaling)

In Query 4, the optimizer once again selected the BroadcastHashJoin strategy for both join operations, as reflected in the physical plan below. The query processes a large dataset, "Los Angeles Crime Data", which is joined with two smaller datasets — "LA Police Stations" and "MO Codes". Since these smaller datasets are compact enough to be efficiently broadcasted, this approach minimizes shuffle overhead and enhances performance.

```
== Physical Plan ==
AdaptiveSparkPlan (55)
+- == Final Plan ==
   TakeOrderedAndProject (33)
   +- * HashAggregate (32)
      +- AQEShuffleRead (31)
         +- ShuffleQueryStage (30), Statistics(sizeInBytes=8.5 KiB, rowCount=168)
            +- Exchange (29)
               +- * HashAggregate (28)
                  +- * Project (27)
                     +- * BroadcastHashJoin Inner BuildRight (26)
                        :- * Filter (19)
                        :  +- * HashAggregate (18)
                        :     +- AQEShuffleRead (17)
                        :        +- ShuffleQueryStage (16), Statistics(sizeInBytes=40.0 MiB, rowCount=5.24E+5)
                        :           +- Exchange (15)
                        :              +- * HashAggregate (14)
                        :                 +- * Project (13)
                        :                    +- * BroadcastHashJoin Inner BuildRight (12)
                        :                       :- * Project (5)
                        :                       :  +- * Generate (4)
                        :                       :     +- * Filter (3)
                        :                       :        +- * ColumnarToRow (2)
                        :                       :           +- Scan parquet  (1)
                        :                       +- BroadcastQueryStage (11), Statistics(sizeInBytes=2.0 MiB, rowCount=9)
                        :                          +- BroadcastExchange (10)
                        :                             +- * Project (9)
                        :                                +- * Filter (8)
                        :                                   +- * ColumnarToRow (7)
                        :                                      +- Scan parquet  (6)
                        +- BroadcastQueryStage (25), Statistics(sizeInBytes=1024.2 KiB, rowCount=21)
                           +- BroadcastExchange (24)
                              +- * Project (23)
                                 +- * Filter (22)
                                    +- * ColumnarToRow (21)
                                       +- Scan parquet  (20)
```