Instacart Co-occurrence Analysis

Andrea Corradetti
University of Bologna
Department of Computer Science and Engineering
Bologna, Italy
andrea.corradetti2@studio.unibo.it

Abstract

This report documents the implementation and evaluation of a distributed system for co-purchase analysis using Apache Spark in Scala. The RDD and Dataset APIs are compared in terms of runtime and scalability. Results are tested on Google Cloud Dataproc and show that the Dataset API consistently outperforms the RDD API.

1 Introduction

The Instacart Online Grocery Basket Analysis dataset contains detailed purchase histories of Instacart customers. This report presents the implementation and performance evaluation of a Scala application that analyzes co-purchase patterns in a simplified version of the dataset.[1]

The application is developed using Apache Spark, follows a distributed computing approach, and is deployed on Google Cloud Dataproc. Apache Spark's newer Dataset API, as described in the official *SQL Programming Guide*[2], is compared against the classic Resilient Distributed Dataset (RDD) interface in terms of performance.

Code is available on Github.[3]

2 Implementation Overview

2.1 Project Structure

The project consists of two independent modules, each implementing the same logic using a different Spark API. One module uses the `RDD` interface, while the other relies on the `Dataset` API. Both can be compiled and executed separately.

2.2 Dataset Pipeline

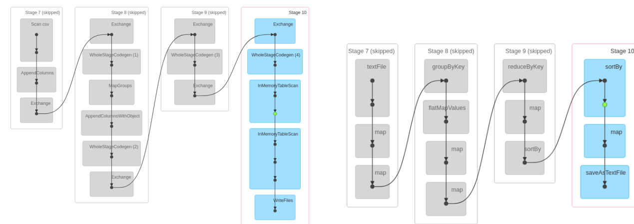
The input CSV file is parsed into a strongly typed `Dataset[Purchase]`. It is grouped by `orderId`, triggering a shuffle to co-locate all items from the same order.

From each group, duplicate products are removed and all unique unordered pairs are generated using `combinations(2)`. Pairs are normalized with `sortPair` to avoid duplicates such as `(a, b)` vs. `(b, a)`.

The pairs are counted via `groupByKey().count()`, sorted by frequency, and cached. This involves two more shuffles: one for aggregation and one for sorting.

Output is written in parallel across multiple files. The logic is implemented in `dataset.Main`, with pair normalization handled by `shared.Shared.sortPair`.

[1]The original dataset includes multiple CSV files with rich metadata on users, products, and orders. In this project, a minimal version was used containing only two columns: `order_id` and `product_id`, likely extracted from both `order_products__train.csv` and `order_products__prior.csv`.
[2]https://spark.apache.org/docs/latest/sql-programming-guide.html
[3]https://github.com/andrea-corradetti/instacart-map-reduce



(a) Dataset pipeline DAG **(b) RDD pipeline DAG**

Figure 1: Execution DAGs for Dataset (left) and RDD (right) pipelines. The stages consist of reading, grouping, reducing, and sorting

2.3 RDD Pipeline

The RDD version follows the same logic as the Dataset pipeline, but uses lower-level operations. After mapping purchases to `(orderId, itemId)`, data is grouped by key and unordered item pairs are generated per order using `toSeq.combinations(2)` and normalized.

Co-occurrence counts are aggregated with `reduceByKey`, then sorted by frequency using `sortBy`. The output is a sorted `RDD[(item1, item2, count)]`. The implementation resides in `rdd.Main` and reuses the same shared utilities.

2.4 Comparison

Both implementations produce the same output and follow the same dataflow: grouping, pair generation, aggregation, and sorting. The Dataset API offers higher-level abstractions and benefits from Catalyst optimizations, while the RDD version provides more explicit control over each step.

3 Cluster Configuration

Experiments were executed on Google Cloud Dataproc using static clusters. Each cluster was configured with the following settings:

- **Region:** `europe-west8`
- **Image version:** `2.2-debian11`
- **Metric sources:** `spark`
- **Component Gateway:** Enabled
- **Master machine type:** `n4-highmem-4`
- **Worker machine type:** `n4-standard-4`
- **Boot disk size:** 100 GB (for both master and workers)

Note. Dataproc submits jobs in `client` mode by default, rather than `cluster` mode, unless explicitly configured otherwise [3]. Because of this, the property `spark.default.parallelism` may not reflect the actual number of vCPUs available across the cluster [1]. To ensure correct partitioning of the input CSV file when using

RDDs, this value must be set manually at cluster creation time to match the total number of available vCores.

By default, Spark on Dataproc uses dynamic allocation and launches one executor per two vCores [2]. In single-node mode, we disable dynamic allocation, set the number of executor instances to zero, and assign four cores per executor. This forces a single executor thread and prevents memory pressure from co-locating the driver and multiple executors on a four-core machine.

4 Performance Comparison

4.1 Metrics: Speedup and Efficiency

We measure scalability using speedup $S(n) = \frac{T(1)}{T(n)}$ and strong scaling efficiency $\mathrm{SSE}(n) = \frac{S(n)}{n}$, where $T(n)$ is the runtime on n workers. These metrics help quantify how performance improves as more resources are added.

4.2 Results

Table 1: RDD vs Dataset Runtime on Dataproc

Workers	RDD Runtime	Dataset Runtime
1 (single-node)	9 min 2 sec	5 min 12 sec
2	5 min 14 sec	2 min 56 sec
3	3 min 55 sec	2 min 16 sec
4	3 min 21 sec	2 min 3 sec

Table 2: RDD Performance Metrics

Workers	Runtime (s)	Speedup	SSE
1 (single-node)	542	1.00	1.00
2	314	1.73	0.86
3	235	2.31	0.77
4	201	2.70	0.68

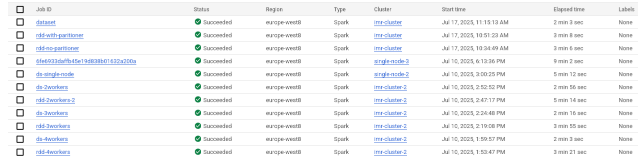
Table 3: Dataset Performance Metrics

Workers	Runtime (s)	Speedup	SSE
1 (single-node)	312	1.00	1.00
2	176	1.77	0.88
3	136	2.29	0.76
4	123	2.54	0.63

4.3 Discussion

The Dataset API consistently outperformed the RDD-based pipeline, benefiting from Catalyst query optimization and efficient memory handling. While both approaches exhibit improved runtimes as more workers are added, scaling is sublinear.

Both implementations perform the same number of shuffles and follow equivalent logical steps. Local testing also showed negligible performance differences when manually specifying a Partitioner before groupBy compared to relying on Spark's default behavior. This is expected, as groupBy internally applies a partitioner, and data must be shuffled once in either case.



Job ID	Status	Region	Type	Cluster	Start time	Elapsed time	Labels
dataset	Succeeded	europe-west8	Spark	imr-cluster	Jul 17, 2025, 11:15:13 AM	2 min 3 sec	None
rdd-with-paritioner	Succeeded	europe-west8	Spark	imr-cluster	Jul 17, 2025, 10:51:23 AM	3 min 8 sec	None
rdd-no-paritioner	Succeeded	europe-west8	Spark	imr-cluster	Jul 17, 2025, 10:34:49 AM	3 min 6 sec	None
6fe6933daffb45e19d838b01632a200a	Succeeded	europe-west8	Spark	single-node-3	Jul 10, 2025, 6:13:36 PM	9 min 2 sec	None
ds-single-node	Succeeded	europe-west8	Spark	single-node-2	Jul 10, 2025, 3:00:25 PM	5 min 12 sec	None
ds-2workers	Succeeded	europe-west8	Spark	imr-cluster-2	Jul 10, 2025, 2:52:52 PM	2 min 56 sec	None
rdd-2workers-2	Succeeded	europe-west8	Spark	imr-cluster-2	Jul 10, 2025, 2:47:17 PM	5 min 14 sec	None
ds-3workers	Succeeded	europe-west8	Spark	imr-cluster-2	Jul 10, 2025, 2:24:48 PM	2 min 16 sec	None
rdd-3workers	Succeeded	europe-west8	Spark	imr-cluster-2	Jul 10, 2025, 2:19:08 PM	3 min 55 sec	None
ds-4workers	Succeeded	europe-west8	Spark	imr-cluster-2	Jul 10, 2025, 1:59:57 PM	2 min 3 sec	None
rdd-4workers	Succeeded	europe-west8	Spark	imr-cluster-2	Jul 10, 2025, 1:53:47 PM	3 min 21 sec	None

Figure 2: Screenshot of Spark job execution results showing runtime.

5 Conclusion

This project implemented and compared co-purchase analysis using Spark's RDD and Dataset APIs on the Instacart dataset. The Dataset API proved more concise and slightly faster, benefiting from Spark's internal query optimizations. While both pipelines scaled reasonably well up to 4 workers, speedup was sublinear, and efficiency decreased with more nodes—reflecting typical limits of parallel processing.

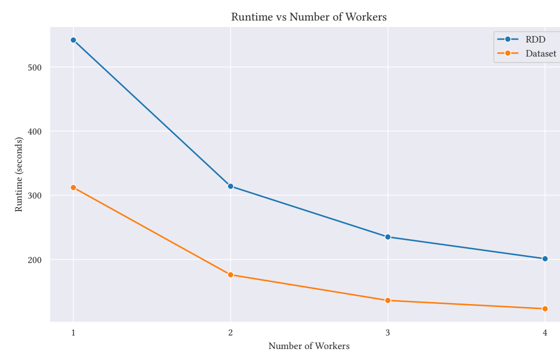


Figure 3: Runtime comparison

Figure 4: Speedup over baseline

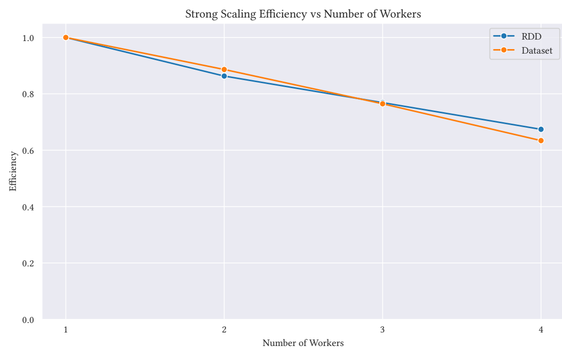


Figure 5: Strong scaling efficiency

References

[1] Apache Spark. 2024. Spark Configuration. https://spark.apache.org/docs/3.5.5/configuration.html#execution-behavior. Accessed: 2025-07-11.

[2] Google Cloud. 2024. Tuning Spark Jobs on Dataproc. https://cloud.google.com/dataproc/docs/support/spark-job-tuning#use_dynamic_allocation. Accessed: 2025-07-11.

[3] Google Cloud. 2024. Viewing and Understanding Dataproc Job Output. https://cloud.google.com/dataproc/docs/guides/dataproc-job-output. Accessed: 2025-07-11.