Scalable and Cloud programming - Co-Purchase-Analysis Technical Report

1st Michele Dinelli
dept. of Computer Science and Engineering, University of Bologna

***Abstract*—This document is the technical report for the scalable and cloud programming project developed for the university course held in Bologna (a.y 24/25). Source code is available online as a GitHub repository.**

***Index Terms*—cloud programming, scala, apache spark, google cloud platform, dataproc**

I. Introduction

This document is a technical report that describes the implementation and scaling evaluation of a co-purchase-analysis script written in Scala. The system is required to read purchases data from a dataset [1] and it must identify co-purchased products by calculating, for each pair of products, the number of orders in which they were purchased together.

II. Implementation

The solution used Apache Spark [2] and runs on distributed nodes using DataProc. Versions used are Scala 2.12.18 and Spark 3.5.5. Implementation follows the map-reduce approach with an eye to scaling efficiency.

TABLE I: Dataset Format

Order ID	Product ID
2	33120
2	28985
2	9327
⋮	⋮
3421083	5020

First the input is read and mapped to a `RDD[T]`. Resilient Distributed Dataset (RDD) are distributed data structures that are accessible as one entity but are shared across nodes. In this case we store couples of `Int` values in the RDD (`RDD[(Int, Int)]`). This comes from the fact that the dataset consists of rows in the format presented in Table I.

After reading the input, the dataset is partitioned across nodes using a `HashPartitioner`. This partitioner assigns each key-value pair to a partition based on the hash of the key. Specifically, a pair (key, value) is sent to the partition with index:

$$\texttt{hash}(key) \mod n \tag{1}$$

where $\texttt{hash}(key)$ is an hash function on the key and n is the number of partitions. This ensures that all pairs with the same key end up in the same partition (and thus on the same node), which is crucial for operations like `groupByKey` to work correctly without requiring additional shuffling.

Data are then grouped by order identifier, producing a `RDD[(Int, Iterable[Int])]` where the key is the order identifier. This is done using `groupByKey` which is an operation that triggers shuffling, but since the dataset was previously partitioned using a `HashPartitioner` based on the same key (order ID), all entries with the same key are already co-located in the same partition. As a result, the shuffle still occurs but is significantly optimized.

Next, for each grouped order, the list of product IDs is converted to a distinct list to remove duplicates. All unique pairs of products (i, j) such that $i < j$ are generated from this list, and each pair is associated with the value 1. This results in a new RDD of the form `RDD[((Int, Int), 1)]`, representing one co-purchase occurrence per pair.

Finally, all identical product pairs are aggregated using `reduceByKey`, summing the counts to compute the total number of times each pair was co-purchased. The results are formatted and written to the output directory using `saveAsTextFile`. A `coalesce(1)` is used to write the result into a single output file.

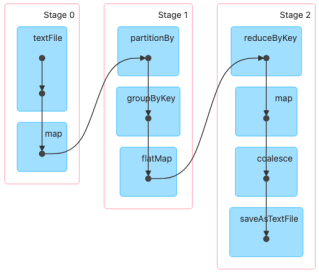


Fig. 1: DAG of the implemented solution

The steps are represented as a Directed Acyclic Graph (DAG) (Fig. 1) produced by the SparkUI.

III. Analytical Metrics

A. Speedup

We define $T(n)$ as the execution time of a parallel program with n nodes. The speedup $S(n)$ is defined by the formula

$$S(n) = \frac{T(1)}{T(n)} \tag{2}$$

Ideally, the program with n nodes requires $1/n$ the time of the program with 1 node. $S(n) = n$ is a linear speedup but in

practice it's sublinear $S(n) \leq n$. This limitation is captured by Amdahl's Law [3], which states that if a task consists of a fraction f hat is inherently sequential (i.e., cannot be parallelized), and the remaining fraction $1-f$ can be sped up by a factor of P then the maximum achievable speedup is

$$\frac{1}{f + \frac{1-f}{P}} < \frac{1}{f} \tag{3}$$

Even if we could infinitely speed up the parallelizable part (i.e., $P \to \infty$), the overall speedup would still be limited by the sequential portion f.

B. Scaling Efficiency

To evaluate the impact of Amdahl's law (3) on a distributed system we introduce Strong Scaling Efficiency (SSE) and Weak Scaling Efficiency (WSE).

1) SSE: measures how increasing the number of nodes impacts the speedup while keeping the total amount of work fixed. SSE is defined by

$$SSE(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)} \tag{4}$$

The total amount of work remains constant, while the amount of work for each processor decreases as n increases. SSE is limited by the constant $\frac{1}{f}$ so it tends to zero.

$$\lim_{n\to\infty} SSE(n) = \lim_{n\to\infty} \frac{T(1)}{nT(n)} = \lim_{n\to\infty} \frac{1}{fn} = 0 \tag{5}$$

IV. Local Results

Local testing was done on an Apple Mac Mini with an Apple M4 chip (10-core CPU: 4 performance cores, 6 efficiency cores; 10-core GPU; 120GB/s memory bandwidth; 16GB unified memory). The code was compiled and packaged using sbt, a build tool for Java and Scala.

To get a sense of how the program scales, tests were run locally by gradually increasing the number of cores available to Spark. The resulting Jar, when submitted to Spark, produces the following results:

TABLE II: Local Results

Number of Cores	Partitions	Time (min)
1	2	5,5
2	4	4,3
4	8	3,0
8	16	2,5
10	30	2,3

The full list of run is shown in Fig. 4. Results are not sorted.

For local testing we notice that the Speedup is sublinear since for $n = 10$

$$S(n) = \frac{T(1)}{T(n)} = \frac{5.5}{2.3} \approx 2.39 \tag{6}$$

And SSE for $n = 10$ is

$$SSE(n) = \frac{T(1)}{nT(n)} = \frac{5.5}{n2.3} \approx 0,239 \tag{7}$$

This gave a rough idea of how the system handles parallel execution. However, since everything runs on the same machine, it doesn't reflect the full complexity of a real distributed setup. For example there's no network delay or data transfer between nodes.

V. Results

Final testing was done using Google Cloud Dataproc. The Jar produced and the input file have been stored in a bucket and then a Spark Job was submitted increasing workers from 1 to 4. Results are shown in Table III.

TABLE III: Results

Workers (n)	Time (s)	S(n) (ratio)	SSE(n)	Machine
1	753	/	/	n2-standard-4
2	337	2,23	1,11	
	336	2,24	1,12	
3	222	3,39	1,13	
	226	3,33	1,11	
4	195	3,86	0.96	
	181	4,16	1,04	

The results turned out quite encouraging, showing that the solution scales well as more workers are added. Speedup and efficiency remained strong up to four nodes, which suggests that the workload benefits from parallel execution. That said, Amdahl's Law reminds us that this improvement won't continue forever, there's a point where adding more workers won't help much due to the parts of the task that can't be parallelized. The n2-standard-4 machines performed very good and seem like a goof choice for future tests. Unfortunately, we couldn't explore beyond this setup due to limited Google Cloud credits.

The product pair with the highest number of co-purchases is (13176, 47209), appearing together 62.341 times.

References

[1] Kaggle. [Online]. Available: https://www.kaggle.com/datasets/psparks/instacart-market-basket-analysis

[2] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: a unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, p. 56–65, Oct. 2016. [Online]. Available: https://doi.org/10.1145/2934664

[3] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, 1967, p. 483–485. [Online]. Available: https://doi.org/10.1145/1465482.1465560

Appendix

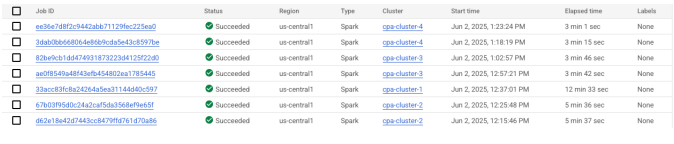
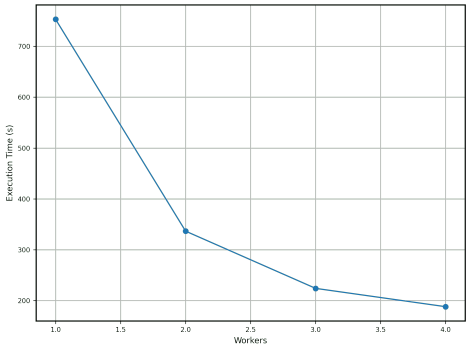
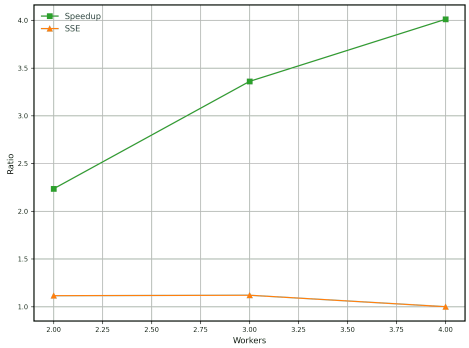


Fig. 2: Dataproc testing results



(a) Execution Time vs Workers



(b) SpeedUp and SSE vs Workers

Fig. 3: Performance metrics as the number of workers increases. (a) shows execution time vs number of workers. As the number of workers increases, execution time decreases. (b) shows speedup and SSE vs number of workers. While speedup improves with more workers, SSE eventually declines due to Amdahl's law [3].

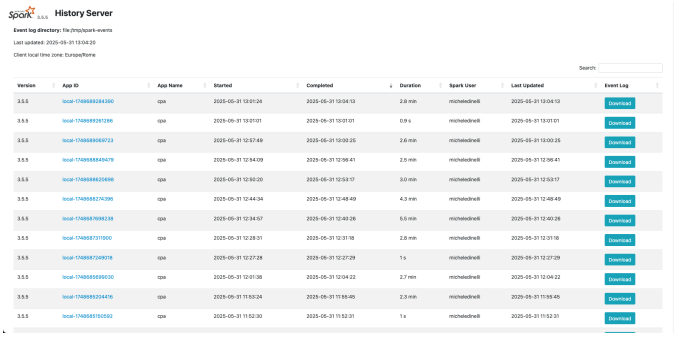


Fig. 4: Local testing results