Investigating Parallelization Techniques for Join Algorithms

Israt Jahan
ijahan1@memphis.edu
The University of Memphis
Memphis, Tennessee, USA

Dr. Xiaofei Zhang
xzhang12@memphis.edu
The University of Memphis
Memphis, Tennessee, USA

ABSTRACT

Parallel techniques play a crucial role in join algorithms, particularly in the context of large-scale data processing. These techniques enhance the efficiency and scalability of join operations, ensuring optimal utilization of resources in parallel computing environments. In the current landscape of database systems research, there exists a significant focus on advancing parallel techniques for join algorithms. This project investigates the performance of three parallel join algorithms—Sort-Merge Join, Broadcast Join, and Shuffled Hash Join—implemented in PySpark using the TPC-H dataset. Through extensive experimentation, the study focuses on assessing execution times under varying conditions. The results reveal that Sort-Merge Join consistently outperforms Broadcast Join and Shuffled Hash Join.

KEYWORDS

DBMS, Join Algorithm, Parallelization Technique, PySpark, TPC-H Dataset

ACM Reference Format:
Israt Jahan and Dr. Xiaofei Zhang. 2018. Investigating Parallelization Techniques for Join Algorithms. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In the era of expansive data and intricate analytical queries, the efficiency and speed of database operations are crucial. Join algorithms play a pivotal role in this process, facilitating the merging of data sets within relational databases. However, the exponential growth in data volumes poses a significant challenge to traditional sequentially executed join operations. Recognizing this challenge, contemporary Database Management Systems (DBMS) increasingly employ parallel techniques to optimize the efficiency and scalability of join algorithms. Parallelization strategies, involving concurrent processing across multiple computing units, have become essential for meeting the demands of modern computing environments [2]. Their implementation enhances the speed and performance of database operations, particularly in the context of large-scale data processing.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

Recent advancements in Database Management Systems (DBMS) underscore a shift towards leveraging parallelization for large-scale data processing. The escalating data volumes in modern applications necessitate streamlined and scalable techniques for query execution. Parallelization, as a key strategy, has become crucial for enhancing DBMS performance, especially concerning join algorithms. However, parallel join query processing encounters challenges, including uneven data distribution and skewed join key values leading to load imbalance. Issues such as communication overhead and synchronization among parallel processes can offset performance gains [17]. The selection of the right join algorithm, efficient memory management, and minimizing data movement costs are critical for optimal parallel processing. Dynamic workload balancing, fault tolerance mechanisms, and scalability concerns further contribute to the complexity of designing effective parallel database systems [10]. Addressing these challenges requires a combination of algorithmic optimizations, system architecture design, and consideration of data and query characteristics to ensure efficient and scalable parallel join operations.

This project focuses on a comparative analysis of three notable join algorithms—Sort-Merge Join [12], Broadcast Join [15], and Shuffled Hash Join [4]—implemented within the PySpark framework [16] in Google Colab [7]. PySpark stands out as a robust tool for efficient and scalable data processing. Leveraging Apache Spark's distributed computing capabilities, PySpark enables parallel execution of join queries across large datasets. The TPC-H dataset [5], a widely recognized benchmark for evaluating database system performance, forms the basis of our investigation.

The project's structure is as follows: Section 2 delves into related research and prior works on parallelization techniques for algorithms. Section 3 provides an exploration of our project's methodology. Section 4 describes the dataset and the environment settings for our project analysis. In Section 5, we present the results and engage in a concise discussion of our analysis findings. Finally, Section 6 concludes our research.

2 RELATED WORKS

In the literature, numerous research endeavors have explored parallel join techniques. Malek et al. [3] conducted a survey focusing on advancements in parallel join algorithms, employing the widely used MapReduce framework on the Hadoop distributed file system. Their work methodically outlined the merits and demerits of existing approaches in a chronological sequence. Additionally, they introduced a novel approach, demonstrating its performance improvements.

Phan et al. [13] extensively delved into diverse facets of join operations within Apache Spark. They provided a detailed examination of key strategies for implementing join operations, offering nuanced insights into the advantages and disadvantages of each

algorithm namely Map-side join, Broadcast join, Reduce-side join, Bloom join, and Intersection Filter-based Bloom join.

A survey on join algorithms was presented by Amer et al. [1]. The authors critically appraised various implementations of Spark join algorithms, including equi join, theta join, similarity join, k-NN, and top-k join. They proposed an optimization approach, showcasing its enhanced performance and contributing to the evolving landscape of join algorithms.

Phan et al. [14] systematically provide a theoretical and experimental assessment of two-way and recursive join algorithms in the Spark environment. The authors present the mathematical cost models for each algorithm and scrutinize the merits and drawbacks of these approaches.

3 METHODOLOGY

3.1 Existing Join Algorithms

In the contemporary era, various techniques are employed to achieve parallelization for join algorithms, and among them, several popular approaches stand out:

Parallel Hash Join: This method optimizes join operations by employing a hash function to partition tables, enabling the simultaneous processing of corresponding partitions [11] [4] [8].

Parallel Sort-Merge Join: This technique expedites join processes by parallelizing sorting and merging steps in distributed computing environments, enhancing efficiency [12] [2].

MapReduce Technique: Leveraging a parallel programming model, MapReduce facilitates join operations by mapping keys from different tables and reducing results. Despite its versatility, the iterative nature of MapReduce may introduce overhead in specific join scenarios [9].

Broadcast Join: This strategy efficiently distributes smaller tables to all nodes in a distributed system, minimizing data movement. It particularly excels in improving join performance in scenarios with skewed data distributions [15].

3.2 Proposed Work

In this study, we conduct an evaluation of broadcast join, shuffled-hash join [6], and sort-merge join within the PySpark framework [16], considering various Spark session configurations as well as partition and thread considerations. We implement inner join, left outer join, right outer join, full outer join, semi join, and anti join. Our primary aim is to assess performance based on execution time. The TPC-H benchmark dataset serves as the foundation for our analytical exploration, and detailed insights into the analysis are presented in the subsequent sections.

4 EXPERIMENT SETUP & DATA DESCRIPTION

4.1 Environment Setup

In our PySpark implementation on Google Colab, we initially install all the necessary dependencies. Following that, we establish a PySpark session with diverse configurations to evaluate the performance of different algorithms. Our configuration adjustments encompass variables such as driver memory, executor memory, and memory fraction.

Specifically, we experiment with distinct thread pools and analyze the impact of both partitioned and non-partitioned data. We conduct tests under several different Apache Spark configurations. Our evaluation is presented for two configurations.

Initially, we conduct join operations using thread pools of 3 and 4 without partitioning the data. Subsequently, within the same configuration (referred to as Config 1), we replicate the join operations with thread pool sizes of 3 and 4, incorporating data partitioning with 10 partitions. Throughout these experiments, we maintain a consistent setup, setting the driver memory at 2g, executor memory at 2g, and a memory fraction of 0.6.

Following that, we undertake another set of tasks involving three thread pools, considering both partitioned and non-partitioned data. This is executed under a distinct Apache Spark session configuration named Config 2, with parameters set as follows: the driver memory at 3g, executor memory at 3g, and a memory fraction of 0.7.

4.2 TPC-H Dataset

The TPC-H (Transaction Processing Performance Council - Decision Support) dataset serves as a widely utilized benchmark for assessing the performance of relational database systems in decision support applications. Comprising eight tables, namely: nation, orders, supplier, partsupp, region, lineitem, part, and customer, the dataset is designed to evaluate system performance comprehensively.

For our analysis, we specifically focus on the customer and orders tables. The customer table, featuring 88,144 rows and 8 columns, has the customer key as its primary key, which serves as a foreign key in the orders table. The orders table, on the other hand, consists of 730,734 rows and 9 columns.

5 RESULTS AND DISCUSSIONS

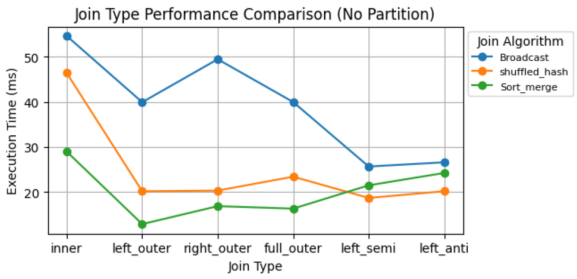
In this section, we elaborate on the insights gained from our analysis of join operations. Specifically, we execute join operations on the "customer" and "orders" tables, utilizing broadcast join, shuffled hash join, and sort-merge join across a spectrum of join types, encompassing inner, left outer, right outer, full outer, left semi, and anti semi join.

The execution time results for configuration 1 are shown in figure 1 and figure 2. Results for both thread 3 and thread 4 without data partitioning are illustrated in Figure 1. Figure 1a specifically represents the performance for thread 3, wherein the shuffled-hash join exhibits superior performance for left semi join, while sort-merge join excels in all other scenarios.

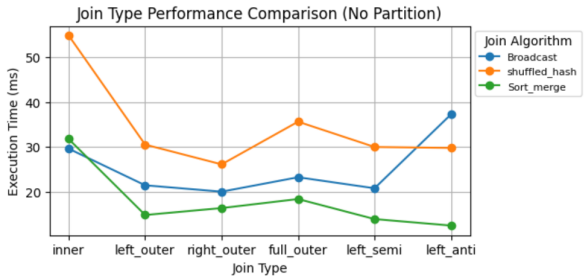
Moving to Figure 1b, which showcases the outputs for thread pool 4, broadcast join demonstrates minimal execution time for inner join. However, across all join types, sort-merge join consistently exhibits the lowest execution time.

Results for thread pools 3 and 4 with data partitioning are presented in Figure 2, with Figure 2a depicting results for thread pool 3 and Figure 2b for thread pool 4. In both cases, sort-merge join outperforms other join algorithms.

The outcomes for Configuration 2 are illustrated in figure 3 concerning thread pool 3, encompassing both partitioned and non-partitioned data. In scenarios involving non-partitioned data, the

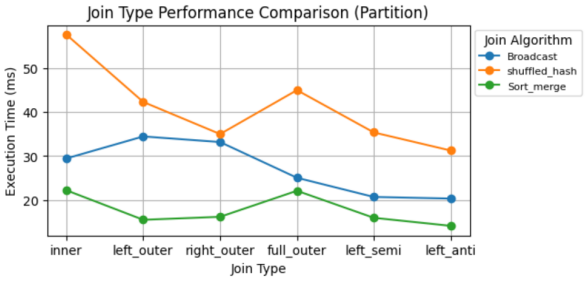


(a) Thread Pool 3

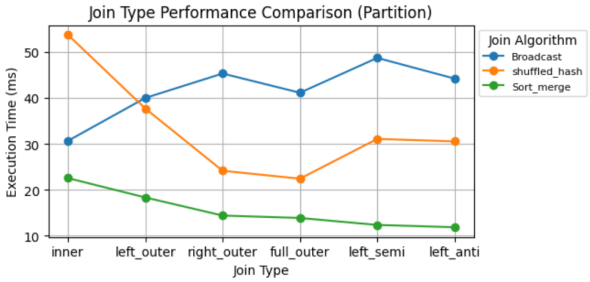


(b) Thread Pool 4

Figure 1: Execution Time for Join Types with No Partition for Config. 1

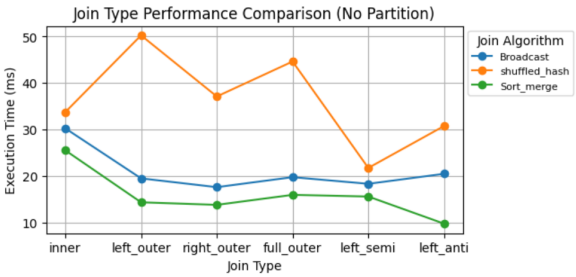


(a) Thread Pool 3

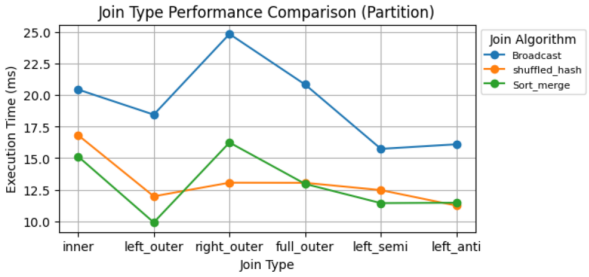


(b) Thread Pool 4

Figure 2: Execution Time for Join Types for Config. 1



(a) Thread Pool 3 without Partition



(b) Thread Pool 3 with Partition

Figure 3: Execution Time for Join Types with Partition for Config. 2

broadcast operation yields superior results compared to the shuffled hash, although sort-merge surpasses both, as depicted in figure 3a. Conversely, with partitioned data, the shuffled hash join algorithm competes closely with the sort-merge algorithm in certain cases, as illustrated in figure 3b.

Upon this analysis, our findings consistently affirm that sort-merge join surpasses alternative join methods across diverse join types.

6 CONCLUSIONS

In conclusion, our analysis of join operations on the "customer" and "orders" tables within the PySpark framework has provided valuable insights into the performance of broadcast join, shuffled hash join, and sort-merge join across various join types. Through this examination of execution times under different configurations, including different thread pools and data partitioning scenarios, we have consistently observed that the sort-merge join algorithm outperforms its counterparts. In our future endeavors, we intend to broaden our comparative analysis to encompass a wider array of configurations and delve into the performance of join operations within alternative frameworks, including Apache Hadoop. While our current study focuses on binary join operations, we plan to extend our research to incorporate multi-way join operations using diverse datasets. This expansion is designed to provide a more comprehensive insight into the behavior of join algorithms across

varied settings, contributing to informed decision-making in the domain of distributed data processing

ACKNOWLEDGMENTS

I express my gratitude to Dr. Xiaofei Zhang, my instructor, for the invaluable support provided during this project.

REFERENCES

[1] Amer Al-Badarneh. 2019. Join Algorithms under Apache Spark: Revisited. In *ICCTA '19: Proceedings of the 2019 5th International Conference on Computer and Technology Applications*. 56–62. https://doi.org/10.1145/3323933.3324094

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (sep 2013), 85–96. https://doi.org/10.14778/2732219.2732227

[3] Malek Mahmoud Barhoush, Anas Mohammad AlSobeh, and Ahmad Al Rawashdeh. 2019. A Survey on Parallel Join Algorithms Using MapReduce on Hadoop. In *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*. 381–388. https://doi.org/10.1109/JEEIT.2019.8717427

[4] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving Hash Join Performance Through Prefetching. *ACM Transactions on Database Systems (TODS)* 32, 3 (August 2007), 17–es. https://doi.org/10.1145/1272743.1272747

[5] Transaction Processing Performance Council. 25 November, 2023. *TPC Benchmark™ H (TPC-H).* https://www.tpc.org/tpch/

[6] Siddharth Ghosh. 8 December, 2023. *Apache Spark Join Strategies Deep Dive.* https://medium.com/@ghoshsiddharth25/apache-spark-join-strategies-deep-dive-26bf7e85db28

[7] Google. 7 December, 2023. *Google Colab.* https://colab.research.google.com/

[8] Wentao Huang, Yunhong Ji, Xuan Zhou, Bingsheng He, and Kian-Lee Tan. 2023. A Design Space Exploration and Evaluation for Main-Memory Hash Joins in Storage Class Memory. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1249–1263. https://doi.org/10.14778/3583140.3583144

[9] Hyunjo Lee, Jae-Woo Chang, and Cheoljoo Chae. 2021. KNN-Join Query Processing Algorithm on Mapreduce for Large Amounts of Data. In *2021 International Symposium on Electrical, Electronics and Information Engineering* (Seoul, Republic of Korea) *(ISEEIE 2021)*. Association for Computing Machinery, New York, NY, USA, 538–544. https://doi.org/10.1145/3459104.3459192

[10] J. Li, J. Naughton, and R. Nehme. 2017. Resource bricolage and resource selection for parallel database systems. *The VLDB Journal — The International Journal on Very Large Data Bases* 26, 1 (2017), 31–54. https://doi.org/10.1007/s00778-016-0435-4 Online publication date: 1-Feb-2017.

[11] Eugenio Marinelli and Raja Appuswamy. 2021. XJoin: Portable, parallel hash join across diverse XPU architectures with oneAPI. In *DAMON '21: Proceedings of the 17th International Workshop on Data Management on New Hardware*. 1–5. https://doi.org/10.1145/3465998.3466012

[12] Mehdi Moghaddamfar, Christian Färber, Wolfgang Lehner, and Norman May. 2020. Comparative Analysis of OpenCL and RTL for Sort-Merge Primitives on FPGA. In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (Portland, Oregon) *(DaMoN '20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 7 pages. https://doi.org/10.1145/3399666.3399897

[13] Anh-Cang Phan, Thuong-Cang Phan, and Thanh-Ngoan Trieu. 2020. A Comparative Study of Join Algorithms in Spark. In *Proceedings of the 7th International Conference on Future Data and Security Engineering (FDSE 2020)*. 185–198. https://doi.org/10.1007/978-3-030-63924-2_11

[14] Anh-Cang Phan, Thuong-Cang Phan, Thanh-Ngoan Trieu, and [et al.]. 2021. A Theoretical and Experimental Comparison of Large-Scale Join Algorithms in Spark. *SN Computer Science* 2, 352 (2021). https://doi.org/10.1007/s42979-021-00738-x

[15] Sebastian Schelter, Christoph Boden, Martin Schenck, Alexander Alexandrov, and Volker Markl. 2013. Distributed Matrix Factorization with Mapreduce Using a Series of Broadcast-Joins. In *Proceedings of the 7th ACM Conference on Recommender Systems* (Hong Kong, China) *(RecSys '13)*. Association for Computing Machinery, New York, NY, USA, 281–284. https://doi.org/10.1145/2507157.2507195

[16] Apache Spark. 8 December, 2023. *Apache Spark Python API Documentation.* https://spark.apache.org/docs/latest/api/python/index.html

[17] Y. Wang, Y. Zhong, Q. Ma, and G. Yang. 2018. Handling data skew in joins based on cluster cost partitioning for MapReduce. *Multiagent and Grid Systems* 14, 1 (2018), 103–123. https://doi.org/10.3233/MGS-180283 Online publication date: 1-Jan-2018.

A ASSOCIATED SOURCE CODE

The associated source code for this project is available in the following github link:
https://github.com/ijahan1/Parallel-Join-Algorithm

Received 8 December 2023; revised 8 December 2023; accepted 8 December 2023