Comparative Analysis of Different Relational Database Join Strategies in Distributed Systems

Siddharth Samber
Master of Applied Computing
University of Windsor
Windsor, Canada
Email: siddharthsamber10@gmail.com

Nhan Nguyen
Master of Applied Computing
University of Windsor
Windsor, Canada
Email: Nguyen6n@uwindsor.ca

Aryan Chauhan
Master of Applied Computing
University of Windsor
Windsor, Canada
Email: Chauhanaryan2000@gmail.com

***Abstract*—This research presents a comprehensive comparative analysis of various relational database join strategies in the context of distributed systems. Join operations, as the foundation of query processing, play a critical role in determining the efficiency and responsiveness of database systems, especially in the era of exponential data growth. The study explores strategies such as Broadcast, Partitioned, Bucketed, Sort-Merge, MapReduce, and Bloom Filter Joins, aiming to provide insights into their strengths and limitations.**

***Index Terms*—Broadcast Join, Partitioned Join, Bucketed Join, Sort-Merge Join, MapReduce Join, Bloom Filter Join, Data Distribution Techniques, Range Partitioning, Hash Partitioning, Scalability, Query Processing, Distributed Systems**

I. Introduction and Motivation

Foundation of Query Processing: Join operations serve as the fundamental building blocks of database systems, integral to query processing. Their efficiency directly impacts the performance and responsiveness of database operations.

Importance in the Era of Growing Data: In the current technological landscape, characterized by exponential data growth, optimizing distributed join operations has become pivotal. The sheer volume of data demands streamlined and efficient join strategies to ensure timely and effective query execution.

Exploration of Various Join Techniques: This project aims to delve into an array of techniques for large-scale relational databases, including Broadcast, Partitioned, Bucketed, Sort-Merge, MapReduce, and Bloom Filter Joins. By comparing and contrasting these techniques, this research endeavors to provide a comprehensive understanding of their respective strengths and limitations.

Resource for Informed Decision-Making: The ultimate goal is to furnish a valuable resource for developers, database administrators (DBAs), businesses, and academic researchers. This resource will equip them with insights and knowledge crucial for making well-informed decisions regarding the selection of optimal join strategies tailored to their specific needs and scenarios.

II. Background Study and Related Works

Objectives of Prior Research: Numerous scholarly articles have focused on enhancing distributed join functionalities, emphasizing key goals such as:

Improving Query Performance: Streamlining join operations to boost the speed and efficiency of queries. Optimizing Resource Utilization: Efficiently managing system resources during distributed join operations. Enhancing Join Operations with Scaled Data: Tailoring join methods to handle large-scale data under various distribution scenarios. Significance of Data Distribution Techniques: Within these research studies, data distribution techniques, such as range partitioning and hash partitioning, have emerged as crucial elements. The choice of data distribution method significantly influences the effectiveness and efficiency of distributed join operations. Scholars have emphasized the impact of these techniques on minimizing data movement between nodes, a key consideration for optimizing distributed systems.

Focus on Minimizing Data Movement: A prominent trend in existing research has been the emphasis on reducing data transmission between nodes during join operations. This minimization approach targets the selective transfer of essential information, aiming to improve overall system performance.

Exploration in this Paper: This research endeavors to expand upon these established concepts by delving into a wider array of strategies for distributed joins. The objective is to provide a comprehensive examination of various join methodologies, encompassing conceptual explanations, performance evaluations, and tailored recommendations suited to specific use cases.

Related Works:

Corbett et al., Spanner: Google's Globally-Distributed Database: This work explores the challenges and techniques employed in managing globally distributed databases, shedding light on strategies for efficient distributed data handling. Cooper et al., Benchmarking Cloud Serving Systems with YCSB: This benchmarking study assesses the performance of cloud serving systems, offering insights into optimizing resource usage and query execution in distributed environments. Wang et al., A Performance Evaluation of Distributed Join Algorithms for Big Data Processing: This paper evaluates various distributed join algorithms, focusing on their performance in handling substantial volumes of data in distributed environments.

III. Concepts

A. Broadcast Join

1) Introduction:: Broadcast Join is a method used in distributed systems for joining tables where one of the tables (typically the smaller one) is broadcasted to all the worker nodes. This means that a copy of the smaller table is sent to every node in the cluster. When the broadcasted table reaches each node, it is stored in memory for quick access. Then, each worker node performs an in-memory join with the broadcasted table and the partitioned table. This is particularly efficient because the join operation happens in memory, which is much faster than disk-based operations.

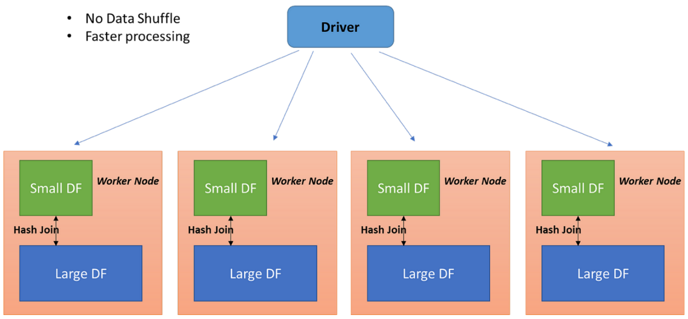


Fig. 1. Broadcast

2) Usage Scenarios::

- **Small Build Side:** It works well when one of the tables is significantly smaller than the other. Broadcasting the smaller table reduces the amount of data that needs to be transferred across the network.
- **High Network Bandwidth:** It's efficient when the network bandwidth between the nodes is high, as broadcasting a table can be expensive in terms of network traffic.

3) Limitations::

- **Limited by Memory:** The broadcasted table must fit in the memory of each worker node. If the table is too large, this method may not be feasible.
- **Skewness:** If the join keys are skewed, meaning that a few keys have a much larger number of matching records than others, this can lead to uneven distribution of work and potentially slower performance.
- **Inefficient for Large Tables:** If both tables are large, broadcasting can cause significant network congestion.

B. Partitioned Join

1) Introduction:: Partitioned Join is a strategy where both tables are partitioned based on their join keys, and these partitions are distributed across the worker nodes in the cluster. Each node then performs a local join with its partitions. After the local join, the nodes exchange the necessary information to complete the join operation. This exchange step ensures that the correct records are combined across partitions.

2) Usage Scenarios::

- **Large Tables:** Partitioned Join is well-suited for scenarios where both tables are large and can be evenly partitioned. It allows for parallel processing of data across nodes.
- **Join Key Distribution:** When the join keys are evenly distributed, partitioned join can be very efficient.

3) Limitations::

- **Skewed Data:** If the join keys are heavily skewed, meaning that certain keys have significantly more matches than others, it can lead to uneven processing and potentially slower performance.
- **Expensive Shuffle Operations:** If the join keys are not properly distributed, it can lead to costly data shuffling between nodes, impacting performance.

C. Bucketed Join

1) Introduction:: Bucket Join is a join operation that leverages pre-partitioning of data into buckets or partitions. This pre-partitioning is based on a common attribute or key shared between the tables to be joined. The basic steps involved in a Bucket Join are as follows:

- **Data Partitioning:** Tables are divided into buckets based on a shared key. Each bucket contains data with the same key.
- **Data Redistribution:** Buckets with the same key are sent to the same node in the distributed system.
- **Local Joins:** On each node, local joins are performed, which are much faster due to the data locality.
- **Result Aggregation:** The results of local joins are aggregated to obtain the final join result.

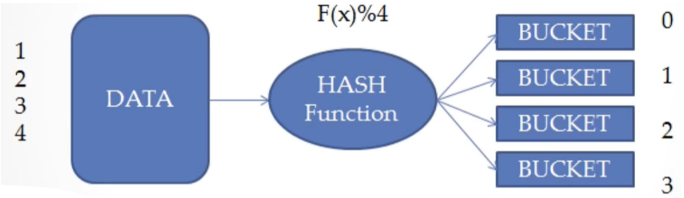


Fig. 2. Bucket Join

2) Usage Scenarios::

- **Large Datasets:** When dealing with large datasets, the distribution of data into buckets can significantly reduce the amount of data that needs to be shuffled across the network.
- **Common Join Keys:** When tables to be joined share a common key that is well-suited for partitioning, Bucket Join becomes highly efficient.
- **Clustered Data:** If the data exhibits clustering based on the join key, Bucket Join can take advantage of this natural data distribution.

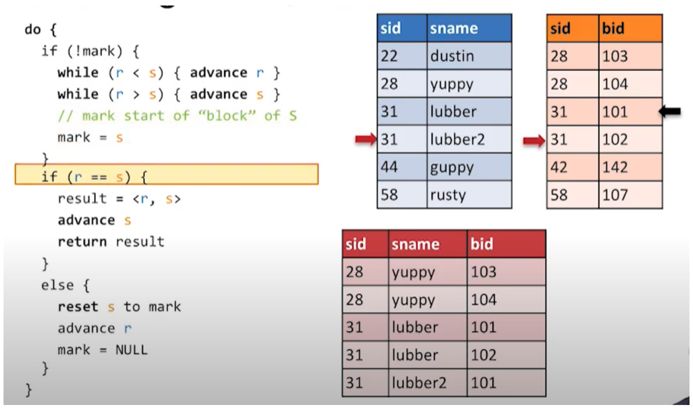
3) Limitations::

- **Skewed Data Distribution:** If the data is not evenly distributed across buckets, some nodes may become overwhelmed with data, leading to performance issues.
- **Limited Key Choices:** The efficiency of Bucket Join is highly dependent on the choice of the join key. In cases where an appropriate key is not available, other join strategies may be more suitable.

D. Sort-Merge Join

1) Introduction:: Sort Merge Join is an algorithm used to combine two data sets by first sorting them and then merging them based on a join condition. The fundamental steps involved in a Sort Merge Join are as follows:

- **Data Sorting:** The data sets to be joined are sorted based on the join key, typically using an efficient sorting algorithm.
- **Merge Step:** The sorted data sets are then merged together, comparing the join key values.
- **Result Formation:** When matches are found during the merge, the join operation forms the result set by combining the corresponding rows.



```
do {
  if (!mark) {
    while (r < s) { advance r }
    while (r > s) { advance s }
    // mark start of “block” of S
    mark = s
  }
  if (r == s) {
    result = <r, s>
    advance s
    return result
  }
  else {
    reset s to mark
    advance r
    mark = NULL
  }
}
```

sid	sname
22	dustin
28	yuppy
31	lubber
31	lubber2
44	guppy
58	rusty

sid	bid
28	103
28	104
31	101
31	102
42	142
58	107

sid	sname	bid
28	yuppy	103
28	yuppy	104
31	lubber	101
31	lubber	102
31	lubber2	101

Fig. 3. Sort Merge

2) Usage Scenarios::

- **Large Data Sets:** When dealing with large datasets, the efficiency of the Sort Merge Join becomes apparent, as sorting and merging are efficient operations.
- **Join on Unindexed Keys:** When joining on non-indexed keys or attributes that do not have efficient access paths, Sort Merge Join can outperform other join methods.
- **Optimized Data Processing:** In scenarios where data sorting is a common operation, using Sort Merge Join can lead to optimized data processing.

3) Limitations::

- **High Initial Cost:** Sorting data sets can be computationally expensive, especially for large datasets. The initial sorting step can be a bottleneck.
- **Limited Use for Small Data:** In situations where data sets are small, the overhead of sorting and merging may outweigh the benefits, making other join methods more suitable.

E. MapReduce Join

1) Introduction:: MapReduce Join leverages the MapReduce framework to perform joins, involving two MapReduce jobs. The first job maps both tables to emit a key-value pair, and the second job performs the actual join operation. In the first job, the data from both tables is transformed into key-value pairs, where keys are typically the join keys, and values are the corresponding records. Then, in the second job, the records with the same key are grouped together, allowing for the actual join operation.

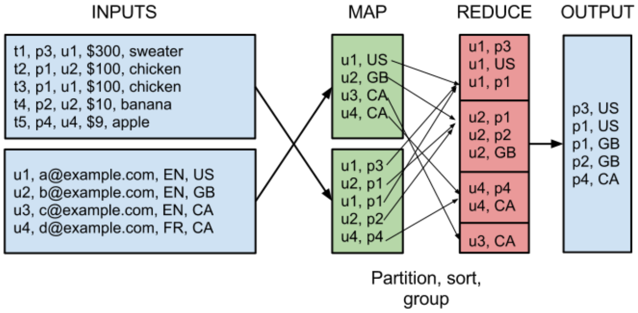


Fig. 4. Map reduce

2) Usage Scenarios::

- **Hadoop Ecosystem:** Commonly used in environments where the Hadoop ecosystem is prevalent and efficient disk-based processing is required.
- **Batch Processing:** MapReduce joins are well-suited for batch processing scenarios, where the focus is on processing large volumes of data.

3) Limitations::

- **High Latency:** MapReduce jobs can have high startup times and significant disk I/O, leading to higher latency compared to other join methods.
- **Not Suitable for Interactive Queries:** Due to its batch processing nature, MapReduce is not suitable for low-latency, interactive query scenarios.

F. Bloom Filter Join

1) Introduction:: Bloom Filter Join is a probabilistic join strategy that employs a data structure known as a Bloom filter to quickly filter out non-matching records before performing a more expensive join operation. A Bloom filter is a memory-efficient data structure capable of rapidly determining whether an element is definitely not in a set (false negatives are not possible) or possibly in the set (with a controlled false positive rate). In join operations, it aids in reducing the number of unnecessary join operations.

2) Usage Scenarios::

- **Reducing Disk I/O:** Useful when aiming to reduce the amount of disk I/O required for a join operation, as it can quickly eliminate non-matching records.
- **Skewed Data:** Effective in scenarios where join keys are skewed, as it helps in reducing the number of unnecessary join operations.

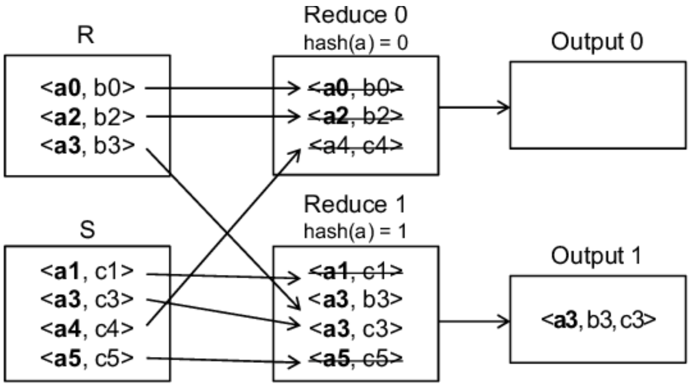


Fig. 5. Bloom Filter

3) Limitations::

- **False Positives:** Bloom filters can have false positives, indicating a match when there isn't one. A subsequent check is needed to verify the result, adding computational overhead.
- **Memory Consumption:** Bloom filters require additional memory, and the size of the filter is a trade-off between false positive rate and memory usage.

IV. Results and Illustrations

A. Join Strategy Average Time (ms) and Average Memory Usage (MB)

B. Findings and Analysis of Join Strategies

1) Broadcast Join: .

Average Time (ms): 1459
Average Memory Usage (MB): 93248
Findings: The Broadcast Join demonstrates a relatively lower time requirement but exhibits higher memory usage compared to some other methods.

2) Partitioned Join: .

Average Time (ms): 2743
Average Memory Usage (MB): 94165
Findings: While the Partitioned Join method shows a higher time requirement compared to Broadcast Join, its memory usage remains similar, indicating an efficient memory management aspect.

3) Bucketed Join: .

Average Time (ms): 2761
Average Memory Usage (MB): 61867
Findings: Bucketed Join method shows similar time requirements to the Partitioned Join but displays substantially lower memory usage, implying better memory efficiency.

4) Sort-Merge Join: .

Average Time (ms): 1494
Average Memory Usage (MB): 37408
Findings: Sort-Merge Join exhibits relatively lower time requirements and significantly reduced memory usage compared to most other methods, indicating its efficiency in handling both time and memory.

5) MapReduce Join: .

Average Time (ms): 13709
Average Memory Usage (MB): 90293
Findings: MapReduce Join displays considerably higher time requirements and memory usage compared to other strategies, indicating potential inefficiencies in handling these aspects.

6) Bloom Filter Join: .

Average Time (ms): 4748
Average Memory Usage (MB): 78368
Findings: Bloom Filter Join shows moderate time requirements but relatively higher memory usage, positioning it in the mid-range in terms of efficiency.

C. Comments on the Results

Efficiency Trade-offs: The findings suggest a trade-off between time and memory usage among different join strategies. Strategies like Sort-Merge Join exhibit balanced efficiency in both aspects, whereas MapReduce Join shows notable inefficiency in time consumption.

Memory Utilization Variation: Noticeable differences exist in memory usage across different join strategies. Strategies like Bucketed Join demonstrate better memory efficiency compared to others, potentially due to their data partitioning mechanisms.

Performance Implications: The analysis highlights the performance implications of each join strategy, providing valuable insights for selecting appropriate strategies based on specific use cases and resource constraints.

Optimization Opportunities: Strategies like MapReduce Join might benefit from optimization in memory utilization, considering their relatively higher resource requirements compared to others.

Recommendations: Based on these findings, recommendations for optimal join strategies could be proposed tailored to diverse scenarios, assisting developers and database administrators in making informed decisions.

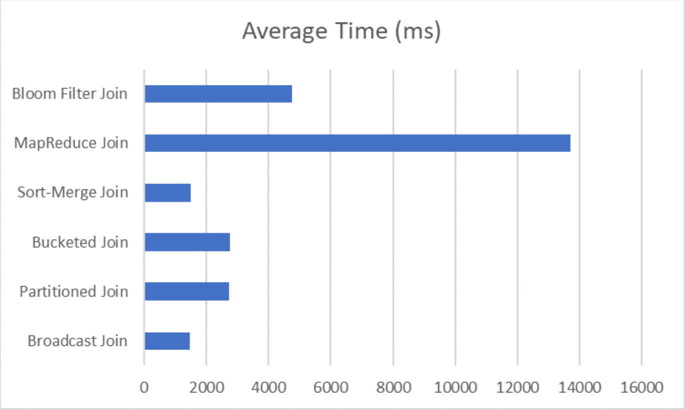


Fig. 6. Average Time

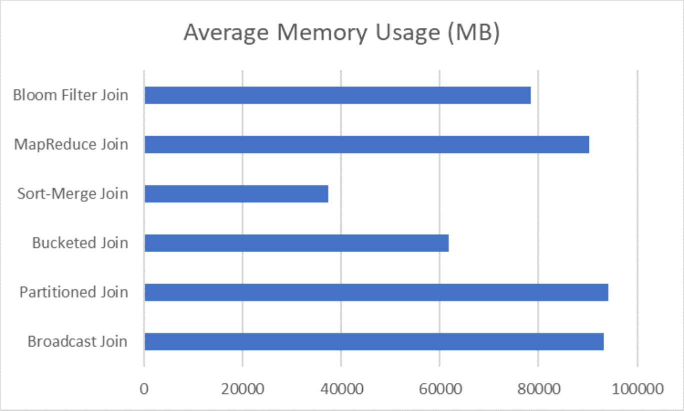


Fig. 7. Average Memory

V. Methodology

A. Data Extraction and Database Setup

Initiating our research involves extracting two primary databases from the MovieLens 25M dataset. The ’movie’ table comprises over 62,000 records, while the ’rating’ database contains an extensive dataset of movie ratings, totaling over 5 million records. This dual-database structure aligns with the microservice architecture, ensuring separate databases for individual services.

B. Simulated Distributed Environment

To replicate a distributed setting, we leverage Docker containers, a widely acknowledged virtualization tool, for hosting and managing our databases. This simulation enables the evaluation of distributed join strategies in a controlled yet realistic environment, mirroring real-world scenarios.

C. Development of Node.js Application

We’re crafting a tailored Node.js application specifically designed for this research context. This application serves as the testing ground for employing diverse distributed join strategies on the simulated databases. It aims to gauge the effectiveness and efficiency of various approaches in managing intricate join operations within a distributed framework.

D. Performance Metrics Evaluation

A meticulous recording of crucial performance metrics is fundamental throughout the experimentation phase. This encompasses capturing execution time, reflecting query processing speed, and monitoring memory utilization, which acts as a gauge for resource efficiency. These parameters will critically assess the practical applicability and scalability of each join strategy.

E. System Definition and Key Components

1) MySQL: An open-source RDBMS employing SQL for managing and manipulating data.

2) Node.js: An open-source runtime environment facilitating server-side JavaScript execution, commonly used for scalable network applications.

3) Docker: A platform enabling the creation, deployment, and execution of applications within isolated containers, ensuring high portability across diverse systems.

F. System Environment and Workflow

For this research, we employ a cloud-based virtual machine, specifically an AWS EC2 instance. This instance configuration consists of 1 vCPU and 1 GB RAM, widely preferred in the web industry for hosting various application types.

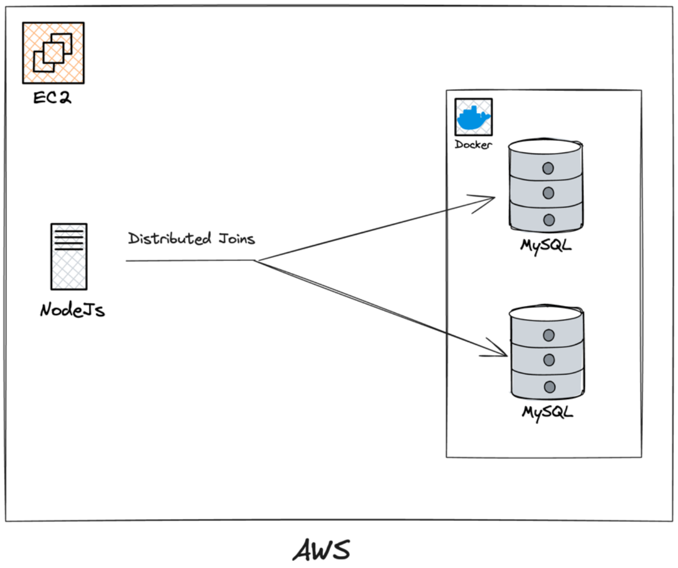


Fig. 8. Workflow

1) Workflow:

- Data Extraction: Utilize MovieLens 25M dataset to create ’movie’ and ’rating’ databases.
- Containerization: Docker hosts and manages databases, simulating a distributed environment.
- Application Development: Node.js application crafted to test distributed join strategies.
- Performance Evaluation: Rigorous metric measurement for execution time and memory utilization.

2) System Components Utilized: MySQL, Node.js, Docker.

3) Cloud-based Hosting: AWS EC2 instance with specified configurations.

4) Flow Diagram:

VI. Limitations/Challenges

A. Adaptive Scalability Requirements

1) Dynamic Workload Handling:

- Contextual Scaling: Adapting to varying workloads necessitates systems that can dynamically scale resources up or down based on specific contextual factors such as peak times, seasonal variations, or sudden spikes in user activity.
- Predictive Scaling Models: Developing predictive models to anticipate workload changes and scale resources proactively, optimizing resource utilization without compromising performance.

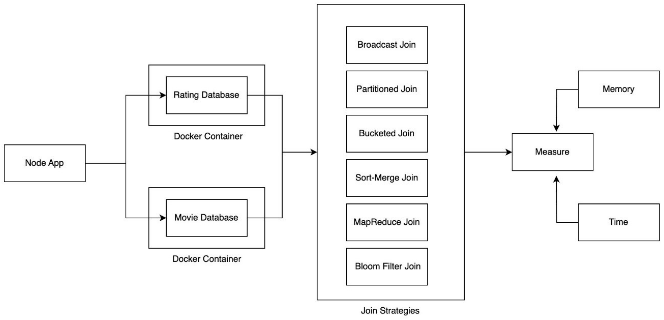


Fig. 9. Flow diagram

2) Automated Resource Allocation:

- AI-Driven Resource Allocation: Leveraging artificial intelligence (AI) and machine learning (ML) algorithms to autonomously allocate resources in real-time, considering factors like data volume, query complexity, and system performance metrics.
- Intelligent Resource Scheduling: Implementing intelligent scheduling mechanisms that consider workload priorities, data locality, and system constraints for efficient resource utilization.

B. Complex Data Variability

1) Unstructured Data Integration:

- Flexible Schema Integration: Developing adaptable schema designs and integration frameworks capable of handling diverse data structures, allowing seamless integration of unstructured or semi-structured data sources.
- Data Transformation Techniques: Employing advanced data transformation methodologies to convert unstructured data into a format compatible with structured databases, enabling effective joins.

2) Streaming Data Joins:

- Real-time Processing Architectures: Implementing stream processing architectures (e.g., Apache Kafka, Apache Flink) to enable continuous data ingestion, processing, and joining in real-time, ensuring low-latency processing for streaming data.

C. Enhanced Performance Demands

1) Real-time Query Optimization:

- Dynamic Query Tuning: Developing techniques for real-time optimization of query execution plans based on changing data statistics and workload patterns to reduce query response times.
- Caching and Preprocessing: Utilizing intelligent caching mechanisms and precomputed summaries to expedite query processing, especially for frequently accessed data subsets.

2) Adaptive Strategy Selection:

- Machine Learning-Driven Decision Making: Integrating machine learning models that continuously learn from historical data and adaptively choose the most efficient join strategy, considering various parameters like data distribution, query complexity, and system performance.

D. Security and Compliance Concerns

1) Data Privacy and Compliance:

- Privacy-Preserving Join Techniques: Developing secure multi-party computation or cryptographic protocols to perform joins without exposing sensitive data, ensuring compliance with data protection regulations (e.g., GDPR, HIPAA).

2) Secure Cross-Platform Joins:

- Secure Data Transmission: Implementing robust encryption, secure data transfer protocols, and access controls to maintain data integrity and confidentiality when joining data across diverse platforms or domains, preventing unauthorized access or breaches.

VII. Conclusion and Future Work

A. Conclusion

Conclusion: The exploration and comparison of diverse relational database join strategies in distributed systems have unveiled critical insights essential for the effective optimization of query processing. Through this extensive study, it becomes evident that each join strategy bears its strengths and limitations, significantly impacting the overall performance and efficiency of distributed databases.

From the meticulous analysis conducted in this research, several key observations and conclusions emerge:

Key Findings: Strategy-specific Advantages: Each join strategy, be it Broadcast, Partitioned, Bucketed, Sort-Merge, MapReduce, or Bloom Filter Joins, demonstrates particular advantages in distinct scenarios based on data distribution, size, and system configurations.

Performance Metrics Variation: Metrics such as execution time, memory utilization, and scalability varied significantly across different join strategies and database configurations. Importance of Contextual Suitability: The context of data distribution, network bandwidth, data size, and system resources significantly influences the optimal choice of join strategy. Implications:

Informed Decision-making: This comparative analysis provides a valuable resource for developers, database administrators, businesses, and academics, offering nuanced insights to guide the selection of appropriate join strategies in diverse distributed system contexts.

Continuous Evolution: The dynamic nature of distributed systems calls for ongoing adaptation and optimization of join strategies as systems evolve, data volumes grow, and new technologies emerge. Challenges and Opportunities: Addressing current challenges and anticipating future complexities in distributed join strategies will steer future research toward innovative solutions and advancements in this domain.

B. Future Work

1. Advanced Performance Benchmarking: Continued research into developing more comprehensive benchmarking methodologies and tools that accurately simulate real-world scenarios and workload variations. Emphasize the incorporation of diverse performance metrics to gauge join strategy efficiency comprehensively.

2. Adaptive Join Strategy Selection: Focus on the development of intelligent, adaptive systems that autonomously select optimal join strategies based on dynamic changes in data distribution, workload, and system resources. This involves integrating machine learning algorithms for real-time strategy selection and optimization.

3. Enhanced Scalability and Dynamic Resource Allocation: Further exploration into scalable architectures that adapt to changing workloads, requiring automated resource allocation strategies. Investigate machine learning-driven models to predict resource demands and scale systems accordingly for efficient join operations.

4. Security-Centric Distributed Joins: Research initiatives aimed at enhancing security and privacy in distributed join operations. Develop secure multi-party computation techniques or privacy-preserving protocols for performing joins on sensitive data while complying with stringent data privacy regulations.

5. Real-time Stream Processing and Unstructured Data Joins: Investigate advanced methodologies for efficient joins in streaming data and the integration of unstructured or semi-structured data across distributed systems. Focus on low-latency processing and adaptable schema integration techniques.

Conclusion Remarks: In conclusion, the comparative analysis of relational database join strategies in distributed systems is an ongoing journey of exploration and innovation. By addressing current challenges and embarking on future research endeavors, the aim is to continually refine join strategies, ultimately contributing to the efficient operation of large-scale distributed databases in evolving technological landscapes.

The journey toward optimal join strategy selection in distributed systems remains open, offering numerous opportunities for innovation, optimization, and a deeper understanding of database query processing in distributed environments.

References

[1] J. C. Corbett, et al., *Spanner: Google's Globally-Distributed Database*, https://research.google/pubs/pub39966/ (accessed May 25, 2023).

[2] M. Takada, *Distributed Systems for Fun and Profit*, https://book.mixu.net/distsys/ (accessed May 25, 2023).

[3] B. F. Cooper, et al., *Benchmarking Cloud Serving Systems with YCSB*, https://dl.acm.org/doi/10.1145/1807128.1807152 (accessed May 25, 2023).

[4] O. Zimmermann, et al., *The Microservices Architectural Style: Decisions and Options in Service Design*, https://ieeexplore.ieee.org/document/7375438 (accessed May 25, 2023).

[5] C. Wang, et al., *A Performance Evaluation of Distributed Join Algorithms for Big Data Processing*, https://ieeexplore.ieee.org/document/7785169 (accessed May 25, 2023).

[6] M. T. Özsu, *Distributed Database Systems: Where Are We Now?*, https://dl.acm.org/doi/10.14778/1920841.1920845 (accessed May 25, 2023).

[7] A. Pavlo, et al., *"Self-Driving Database Management Systems." Communications of the ACM*, vol. 61, no. 4, 2018, pp. 50-57. DOI: 10.1145/3183587

[8] J. Dean, et al., *"Bigtable: A Distributed Storage System for Structured Data." ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, 2008, article 4. DOI: 10.1145/1365815.1365816

[9] D. J. Abadi, et al., *"The Design and Implementation of Modern Column-Oriented Database Systems." Foundations and Trends in Databases*, vol. 5, no. 3, 2012, pp. 197-280. DOI: 10.1561/1900000022

[10] S. Melnik, et al., *"Dremel: Interactive Analysis of Web-Scale Datasets." Proceedings of the 36th International Conference on Very Large Data Bases*, 2010, pp. 330-339.

[11] A. Kemper, et al., *"Hyper: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots." Proceedings of the VLDB Endowment*, vol. 4, no. 6, 2011, pp. 425-436. DOI: 10.14778/2023651.2023658

[12] F. Chang, et al., *"Bigtable: A Distributed Storage System for Structured Data." Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006, pp. 205-218.

[13] A. Lakshman, et al., *"Cassandra: A Decentralized Structured Storage System." ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, 2010, pp. 35-40. DOI: 10.1145/1773912.1773922