A Comparative Study of Two-Way and Multi-Way Equi-Join Algorithms in Hadoop MapReduce for Big Data Analytics

Ms. Nisha Jain[1] and Dr. Preeti Tiwari[2]
[1] Assistant Professor, S.S. Jain Subodh P.G. Mahila Mahavidyalaya, Jaipur, Rajasthan
[2]Associate Professor, International School of Informatics & Management, Jaipur, Rajasthan

Abstract: The rapid growth of big data has necessitated the development of efficient algorithms for processing large-scale datasets. Hadoop MapReduce, a widely used framework for distributed data processing, provides a robust environment for performing complex data operations like joins. This paper presents a comparative study of Two-way and Multi-way equi-join algorithms in Hadoop MapReduce, focusing on their performance in the context of big data analytics. Two-way equi-joins, which involve joining two datasets based on a common key, are the most common join operations in distributed systems. Multi-way joins, on the other hand, extend this concept by involving multiple datasets, resulting in more complex operations and increased computational overhead. The study evaluates these algorithms based on various performance metrics such as number of jobs, pre-processing, cost-effectiveness, execution time, strength and weakness when applied to large datasets. The results highlight the trade-offs between Two-way and Multi-way joins, providing insights into the optimization strategies for each type of operation in a MapReduce environment. By considering factors like number of joins, pre-processing, and cost-effectiveness, this research aims to guide practitioners in selecting the appropriate join algorithm for big data processing in Hadoop environments.

Keywords: Hadoop MapReduce, Big Data Analytics, Two-Way Equi-Join, Multi-Way Equi-Join, Join Algorithms

1. Introduction

The efficient processing and analysis of vast datasets are crucial for businesses, researchers, and organizations in the era of big data. Distributed computing frameworks like Hadoop MapReduce (Palla, K et al., 2009). have emerged as powerful tools for handling large-scale data processing tasks, especially when dealing with complex operations like joins. Two-way and Multi-way equi-joins (Chandar, J. et al.,2010). are common techniques for merging datasets based on shared attributes, enabling users to extract valuable insights from disparate data sources. As the volume and complexity of big data continue to expand, understanding the performance characteristics of these join algorithms has become increasingly important.

Hadoop (Pal, S. et al.,2016), an open-source distributed computing platform, is a cornerstone of big data storage and processing. Its core is the Hadoop Distributed File System (HDFS) and MapReduce (Veiga, J. et al., 2016), a programming model that processes data in parallel. Tools like Hive have been developed to simplify interaction with Hadoop, offering a SQL-like interface that abstracts the complexities of writing low-level MapReduce code. In relational data processing, joins, particularly Two-way and Multi-way equi-joins, are essential for combining data from multiple tables based on common attributes. This study aims to compare the performance of Two-way and Multi-way equi-join algorithms within Hadoop MapReduce environments, focusing on key factors such as number of jobs, pre-processing, cost-effectiveness, execution time, strength and weakness under various conditions. By understanding the strengths and limitations of these join algorithms, users can make informed decisions on the best strategies for their specific use cases, ensuring more efficient data processing and enhancing the performance of Hadoop-based big data systems.

2. Map Reduce Framework

The MapReduce programming model, the backbone of Hadoop frameworks, is a powerful tool for processing large datasets in parallel across a cluster of machines. It efficiently handles critical operations like joining data from multiple sources, which are typically represented as tables. The MapReduce Join algorithm splits the task into two main phases: map and reduce (Fig1).

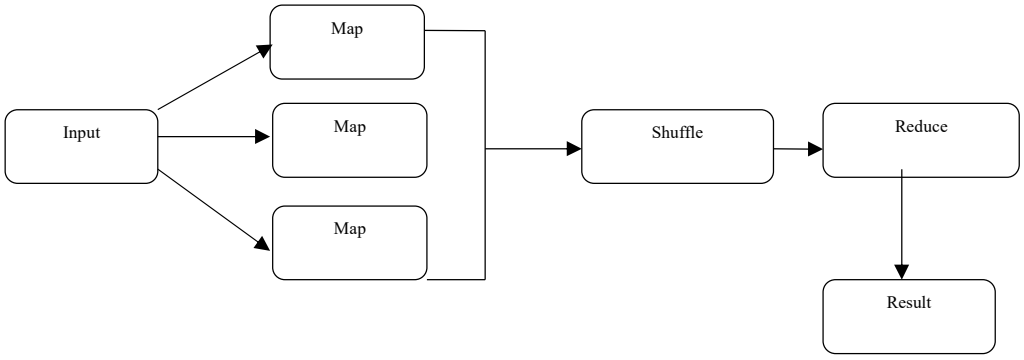

Fig 1: Diagram of Map-Reduce Framework (Al-Badarneh et al., 2022)

In the Map phase, input table data is distributed among mappers, processing portions of the data. For a join, related records from both datasets are sent to the same mapper. Mappers perform initial filtering or transformation to identify the relevant join key and prepare data for the subsequent reduction phase. This phase is highly parallel, as multiple mappers can process data simultaneously on different nodes of the cluster. In the Reduce phase, the reducer gathers and processes all related data for a particular join key, performing the actual join operation. The result is a merged dataset combining information from multiple sources based on the join condition (Blanas, S. et al., 2010).

3. Map Reduce Join Algorithms

MapReduce, a big data management system, does not natively support direct join algorithms (Pigul, A. et al., 2012)., so it uses Two-way and Multi-way join algorithms to improve query execution and reduce I/O costs. These algorithms combine data from multiple tables based on specified conditions. Common techniques for MapReduce Join include replicated join, partitioned join, and sort-merge join (Shaikh, A. et al.,2012).

3.1 Two-Way Equi Join Algorithms for Map-Reduce

Two-way Equi Join is a crucial operation in relational databases and data processing frameworks that merges two datasets based on a common attribute or key, resulting in a new dataset with matching rows from both tables. When two datasets join in the following equation:

$$X(A,B) \bowtie Y(B,C) \quad \ldots\ldots\ldots\ldots\ldots\ldots (1)$$, X and Y are two datasets (tables).

Two-way join algorithms, such as Map Side Join, Broadcast Join, Map–Merge Join, Repartition Join, and Bloom Filter Join, are optimized for query optimization (Lee, K et al., 2012).

3.1.1 Standard Two-Way Equi Join Algorithms

Standard MapReduce's join algorithms are classified into map-side and reduce-side joins, which are performed in the map phase or reduce phase. Map-side joins generate the join result in the map phase, while reduce-side joins send a large number of intermediate records and generate the join result in the reduce phase.

Table 1: Standard Two-Way Equi Join Algorithms: Map-Side Join Vs. Reduce-Side Join (Lee, T. et al., 2014)

MapReduce's Map-Side Join Algorithm	**MapReduce's Reduce-Side Join Algorithm**
• Performs join operation in map phase, avoiding reduce phase. • Efficient for smaller datasets, broadcasted to all mappers. • Reduces execution time and network traffic. • Involves one MapReduce job, with smaller dataset broadcasted and join performed locally on larger dataset.	• Combines large datasets based on a common key. • Operates in three phases: Map, Shuffle and Sort, Reduce. • Map phase: Datasets processed independently by mappers. • Shuffle and Sort phase: Redistributes data to group by key. • Reduce phase: Reducers apply join logic to merge records by key. • Essential for managing large/skewed datasets, efficient distribution, and complex joins.
Steps in the Map-Side Join • Input: Dataset R (Large dataset), Dataset S (Small dataset) and Join Key: A common key (e.g., User_ID) present in both datasets • Load the Smaller Dataset (Dataset S) into Memory: Distributed Cache is used to load the smaller dataset into memory, enabling quick access without expensive disk reads for each cluster node. • Mapper Function: processes Dataset R by extracting the join key and comparing it to Dataset S, joining the results and emitting a key-value pair. • Emission of the join results, which are the key and the combined record from both datasets. This eliminates the need for further shuffling or reducing. • No Reduce Phase, as the join operation is completed during the map phase, eliminating the need for additional sorting, shuffling, or reducing. • Final output, consists of the key-value pairs emitted by the mapper, representing the join result.	**Steps in the Reduce-Side Join** • Prepare input data, which includes two or more datasets (e.g., Dataset R and Dataset S). • Map Phase, involves a join operation between two or more datasets, each with a common join key like User_ID. The key is the join key, and the value is a tagged record identifying the dataset. • Shuffle and Sort Phase, which automatically performs the shuffle and sort operation. This ensures that matching records from both datasets are brought together and processed by the same reducer. • Reduce Phase, the reducer processing key-value pairs from both datasets, iterating through values and joining matching records, even if no match is found in Dataset S. • Output of the reduce phase, consists of the joined records, stored as key-value pairs, where the key is the join key and the value is the merged data from both datasets, ready for further processing or storage.
Advantages and Limitations • Fast, efficient algorithm avoiding data shuffle between mappers and reducers. • Saves network and disk I/O. • Effective for small, memory-fitting datasets. • Best suited for equi-joins, joining based on equality of keys. • Limitations: memory constraints, not suitable for large datasets.	**Advantages and Limitations** • Flexible and scalable for joins in MapReduce. • Suitable for large datasets that cannot fit into memory. • No need for pre-sorting datasets. • Limitations: High shuffle and sort overhead.
Techniques of Map-Side Join Algorithms • Map-Side Merge Join Algorithm • Map-Side Partition Join Algorithm • Broadcast Join Algorithm • Fragment Replicated Join Algorithm • Reverse Map Join Algorithm	**Techniques of Reduce-Side Join Algorithms** • Repartition Join Algorithm • Improved Reparation Join Algorithm • Hybrid Hadoop Join Algorithm

I. **Techniques of Map-Side Join Algorithms:** The Map-Side Merge Join algorithm (Lee, T. et al., 2014) is an efficient technique used in MapReduce frameworks to join large, sorted datasets during the map phase. It eliminates the need for a reduce phase, reducing data shuffling and network overhead. The algorithm performs the join by merging pre-sorted

data streams based on the join key, making it fast and resource-efficient. However, its limitation lies in the requirement that both datasets must be pre-sorted by the join key, which can add overhead if not done beforehand, and may not scale well for very large datasets. It is best suited for smaller, pre-sorted datasets. The Map-Side Partition Merge Join algorithm (Al-Badarneh et al., 2022) is similar but optimized for partitioned datasets. When both datasets are partitioned by the join key, map tasks can process them concurrently, reducing network traffic. However, it requires that both datasets be partitioned in the same way, which can lead to imbalanced workloads or difficulties in partitioning the data. This method is more efficient than the Map-Side Merge Join when partitioning is feasible. The Broadcast Join algorithm (Shanoda, M et al., 2014) is used when one dataset is small enough to fit into memory. It broadcasts the smaller dataset to all nodes in the cluster, minimizing the need for a reduce phase and reducing network overhead. While this method avoids sorting and shuffling, it is limited by the memory capacity of each node, and large datasets may cause memory overflow issues. The Fragment Replicate Join algorithm (Shaikh, A. et al.,2012). is similar to the Broadcast Join but replicates the smaller dataset across all mapper nodes, where each node processes it alongside the larger dataset. It reduces network overhead and simplifies processing but faces issues with high memory usage and storage requirements. The Reverse Map Join (Al-Badarneh et al., 2022)is effective when one dataset is much smaller than the other, as it broadcasts the smaller dataset to all mappers, similar to the Broadcast Join. This reduces the need for data shuffling and sorting. However, it is also limited by memory capacity and can lead to high memory usage in mappers.

Table 2: Process-Steps of different techniques of Map-Side Join Algorithms

Map-Side Merge Join Algorithm	**Map-Side Partition Merge Join algorithm**	**Broadcast Join Algorithm**	**Fragment Replicate Join Algorithm**	**Reverse Map Join Algorithm**
Sort Input Datasets: Both datasets are pre-sorted by the join key (before or during the map phase). **Map Phase:** Mappers process datasets, emitting key-value pairs with the join key and corresponding records. Data is emitted in sorted order. **Merge Phase in Map:** Mappers merge the sorted datasets in parallel based on the join key, emitting combined records for matching keys. **No Shuffle Phase:** No shuffle phase is required since data is already sorted, and merging happens within the map phase. **Output Phase:** Mappers directly emit the final joined records as output.	**Partition Datasets:** Both datasets are partitioned into smaller chunks based on the join key, typically using a hash partitioning strategy. **Map Phase:** Each mapper processes a specific partition from both datasets, emitting key-value pairs (join key, corresponding records) for each dataset. **Merge Phase in Map:** The mapper merges the two partitions, matching records based on the join key and emitting combined results when keys match. **No Shuffle Phase:** As data is already partitioned, no shuffle phase is required, and the merging occurs directly within the map phase. **Output Phase:** The final joined results are emitted by the mapper for each partition.	**Identify Smaller Dataset:** Identify the smaller dataset that can be broadcasted to all worker nodes. **Broadcast Smaller Dataset:** The smaller dataset is sent to all nodes in the cluster. **Map Phase:** Each node has the smaller dataset and performs the join with its portion of the larger dataset. **Join Operation:** On each node, the join is performed locally between the broadcasted smaller dataset and the local partition of the larger dataset. **Output Phase:** The joined results are returned from the nodes.	**Fragment the Larger Dataset:** The larger dataset is divided into smaller fragments. **Replicate Fragments:** Each fragment of the larger dataset is replicated and distributed to all worker nodes. **Map Phase:** Each node has access to all fragments of the larger dataset and the smaller dataset. **Join Operation:** Each node performs a local join between its fragment of the larger dataset and the smaller dataset. **Output Phase:** The final joined results are emitted from each node.	**Identify Larger Dataset:** The larger dataset is processed first. **Map Phase:** The larger dataset is emitted in key-value pairs, with the key being the join key. **Broadcast Smaller Dataset:** The smaller dataset is broadcasted to all mappers. **Join in Map Phase:** Each mapper uses the broadcasted smaller dataset and performs the join locally with the larger dataset partition it processes. **Output Phase:** The final joined records are emitted by each mapper.

II. Techniques of Reduce-Side Join Algorithms: The Repartition Join (Sun, X. et al., 2014) is a crucial technique in MapReduce that optimizes the performance of join operations between two datasets that may not be partitioned similarly. It addresses the issue of increased latency and resource consumption by partitioning both datasets based on the join keys, ensuring all records with the same key are sent to the same node during the shuffle phase. Once the data is correctly aligned, the actual join operation is performed on the nodes, enabling efficient parallel processing. However, traditional Repartition Join can still encounter performance bottlenecks, especially with large datasets or skewed data distributions. To address these limitations, the Improved Repartition Join algorithm (Barhoush, M. et al., 2019) was introduced. This enhanced method employs several strategies to optimize both the shuffle and join phases, including handling skewed data distributions, incorporating adaptive partitioning strategies, and leveraging a broadcast join approach in cases where one dataset is substantially smaller than the other. The Hybrid Hadoop Join is a data processing technique that combines map-side and reduce-side joins in Hadoop MapReduce. It initially uses a map-side join method, loading a smaller dataset into memory as a hash table for fast in-memory lookups. When datasets grow too large, it transitions to a reduce-side join, partitioning data based on join keys and shuffling it to reducers. This hybrid (Mohamed, M. et al., 2018) approach maximizes efficiency and minimizes data movement, striking a balance between speed and resource utilization. Unlike traditional repartition algorithms, the hybrid approach uses intelligent sampling and optimized partitioning strategies to reduce data volume.

Table 3: Process-Steps of different techniques of Reduce-Side Join Algorithms

Repartition Join	**Improved Repartition Join**	**Hybrid Hadoop Join**
Map Phase: Input Processing: Each mapper processes records from datasets A and B. Emit Key-Value Pairs: For dataset A, emit (join_key, record_A). For dataset B, emit (join_key, record_B). **Shuffle and Sort Phase:** Shuffle: Group key-value pairs by the join key. All records with the same key are sent to the same reducer. Sort: Sort records within each partition by the join key to ensure that all records with the same key are grouped together. **Reduce Phase:** Process Key-Value Pairs: The reducer receives all records associated with the same join key. Join Operation: For each key, perform the join (e.g., nested loop join) on records from dataset A and dataset B. **Emit Joined Result:** Output the joined records as (join_key, joined_record).	**Pre-Processing:** Data Partitioning: Use a custom partitioner to ensure records with the same join key from datasets A and B are sent to the same reducer **Map Phase:** Input Processing: Process records from datasets A and B. Emit Key-Value Pairs: Emit (join_key, record_A) for A and (join_key, record_B) for B. **Shuffle and Sort Phase:** Shuffle: Group data by the join key based on the custom partitioning. Sort: Sort records within each partition by the join key. **Reduce Phase:** Process Key-Value Pairs: The reducer processes records with the same join key from A and B. Join Operation: Perform the join on the records (e.g., nested loop join). **Emit Joined Result:** Output the result as (join_key, joined_record).	**Pre-processing Data:** Data Splitting: Divide data into smaller chunks for parallel processing. Partitioning Data: Partition data by join key to group related data from both tables. **Map-Side Join (if applicable):** Load the smaller dataset into memory if it fits. Map function processes each record and finds matching keys in the smaller dataset. **Shuffling and Sorting:** Hadoop shuffles data to ensure records with the same key go to the same reducer. Data is sorted by keys. **Reduce-Side Join:** If datasets are too large for memory, the join is performed in the reduce phase on matching keys. **Output:** The final joined results are written to the output (e.g., HDFS).

3.1.2 Filter-Based Two-Way Equi Join Algorithms

Filter-based Equi-Joins in MapReduce (Lee, T. et al., 2014) are a significant improvement in processing large datasets. They address the inefficiencies of traditional equi-joins by minimizing data processing during the join operation. The approach operates in two phases: the map phase and the reduce phase. In the map phase, each mapper processes its input data, applying a filtering criterion to eliminate irrelevant records. Only records likely to match during the join are retained and sent to reducers. The remaining records are shuffled and distributed to the appropriate reducers based on the join key. In the reduce phase, reducers receive only the relevant data subsets, making the actual equi-join faster and easier. Filter-based equi-joins offer scalability as datasets grow in size, reducing the amount of data needed for shuffled and processed. They also enhance resource utilization, allowing better management of memory and processing power within the Hadoop ecosystem.

Techniques of Filter-Based Two-Way Join Algorithms

The Semi Join Algorithm (Mohamed, M. et al., 2018) is a distributed system operation that filters rows from one dataset (R) based on the presence of matching keys in another dataset (S). This technique reduces data transfer in distributed environments by only returning rows with a corresponding key in S. The semi-join process involves two steps: the mapper phase, which processes the dataset (S) and emits keys, and the reducer phase, which combines all keys from S and prepares a compact structure for broadcasting. The final step is to emit all unique keys of S. The Pre-Semi Join Algorithm (Barhoush, M. et al., 2019) addresses the limitations of the semi-join algorithm by introducing a preprocessing step to reduce data size and ensure efficient filtering. The Bloom Join algorithm (Al Badarneh et al., 2022) uses Bloom filters to optimize join operations in MapReduce, creating a probabilistic data structure to check if an element is a member of a set.

Table 4: Process-Steps of different techniques of Filter-based Two-way Join Algorithms

Semi Join Algorithm	Pre-Semi Join Algorithm	Bloom Join Algorithm
Mapper Phase: The mapper processes dataset S. Emits the keys from S to be used for filtering dataset R. **Shuffling Phase:** The keys emitted by the mappers from S are shuffled and grouped together, ensuring all relevant keys are sent to the same reducer. **Reducer Phase:** The reducer receives all the keys from S and prepares a compact structure. The reducer then broadcasts this compact structure to the mappers. **Final Filtering:** The mappers filter dataset R, keeping only those rows that have a matching key in S.	**Initial Filtering (Pre-Join Phase):** Dataset S is processed to extract only the keys needed for the join. R is then filtered based on the extracted keys from S. **Mapper Phase:** The filtered dataset R (after applying the pre-semi join filter) is processed by mappers. The mappers only pass relevant rows of R that match the keys from S. **Shuffling Phase:** The relevant keys and rows from R are shuffled and grouped by the join key. **Reducer Phase:** The reducer performs the join operation on the filtered data, merging the rows from R with matching rows from S.	**Bloom Filter Creation (Mapper Phase):** A Bloom filter is created for the smaller dataset (S) in the map phase. The Bloom filter stores the keys of S in a compact and memory-efficient way. **Filtering Data (Mapper Phase):** The larger dataset (R) is processed by the mappers. For each record in R, the Bloom filter is used to test whether a matching key exists in S. **Shuffling Phase:** Only records from R that have a potential match (as indicated by the Bloom filter) are sent to the reducers, reducing unnecessary data shuffle. **Reducer Phase:** The reducer performs the actual join operation, matching records from R with the corresponding keys in S.

3.2 Multi-Way Equi Join Algorithms for Map-Reduce

Multi-Way Equi-Joins (Afrati, F. et al., 2011) are essential in databases, data warehousing, and large-scale distributed data processing systems. They are more complex than traditional two-way joins due to the need to manage data partitioning, shuffling, sorting, and combining across multiple datasets**.** When join more than two datasets in the following equations:

$$X(A,B) \bowtie Y(B,C) \bowtie Z(C,D) \ldots\ldots\ldots (2)$$ X, Y and Z are datasets (tables)

Multi-way join algorithms, Reduce-Side One-Short Join , Reduce and Reduce-Side Cascade Join are optimized for query optimization. MapReduce is a distributed framework that provides a powerful and scalable method for multi-way equi-joins. However, designing efficient algorithms for these operations is challenging. Some effective algorithms include Reduce-Side One-Shot Join and Reduce-Side Cascade Join (Mohamed, M. et al., 2018). These operations are necessary when joining multiple datasets based on common attributes, like in e-commerce scenarios. MapReduce's join operation is divided into Map and Reduce phases: Map Phase, where the mapper reads input data, Shuffle Phase, where records with the same join key are sent to the same reducer, and Reduce Phase, where the reducer performs the actual join operation

Techniques for Multi-Way Equi Join Algorithms

The Reduce-Side One-Shot Join is a MapReduce algorithm (Barhoush, M. et alt., 2019) that efficiently joins two large datasets that cannot fit into memory. It is performed in the reduce phase, with the mapper reading both input datasets and emitting key-value pairs. Hadoop shuffles and sorts these pairs, ensuring all records with the same join key are grouped together and sent to the same reducer. The reducer phase processes the grouped records, performs the actual join operation on matching keys, and outputs the result. This one-shot method minimizes data transfer by allowing the join to be done in a single step at the reducer. The Reduce-Side Cascade Join algorithm is an advanced MapReduce technique used to join multiple datasets efficiently, especially when the datasets are large and cannot be handled entirely in memory (kumar Sahu et al).

Table 5: Process-Steps of different techniques of Multi -way Join Algorithms

Reduce-Side One-Shot Join Algorithm	**Reduce-Side Cascade Join Algorithm**
Map Phase: Each dataset emits key-value pairs: (key, "Dataset: record"). Example: (key, "A: record from A"), (key, "B: record from B"), (key, "C: record from C"). **Shuffle Phase:** Records with the same key are grouped together. Ensures that all records with the same join key go to the same reducer. **Reduce Phase:** Reducer receives a list of values for each key (e.g., customer_id). Performs the join operation based on the join condition. Outputs the joined records**.**	**First Map Phase:** Each dataset emits key-value pairs, using only the first dataset's join key for partitioning. **First Shuffle and Sort Phase:** Data is shuffled by the join key, sending intermediate results of the first dataset to the reducer. **First Reduce Phase:** Joins the first and second datasets, emitting intermediate results with a new join key. **Second Map and Reduce Phases:** Intermediate results are passed to subsequent stages, joining with additional datasets one at a time. **Final Output:** After all datasets are joined, the final result is output by the last reducer.

4. A Comparative Analysis of Join Algorithm in MapReduce

In this section, we analyze the characteristics of Two-way Equi join algorithms and Multi-way Equi join algorithms (Fig: 2) in MapReduce based on several key parameters that influence overall performance. The comparison is made between the two-way map-side join algorithm and the two-way reduce-side join algorithm (Blanas, S. et al., 2010). in terms of efficiency, number of jobs, pre-processing requirements, cost-effectiveness, execution time, strengths, and weaknesses. This analysis will help readers choose the most suitable join algorithm for their specific applications. The findings are summarized in the following table.

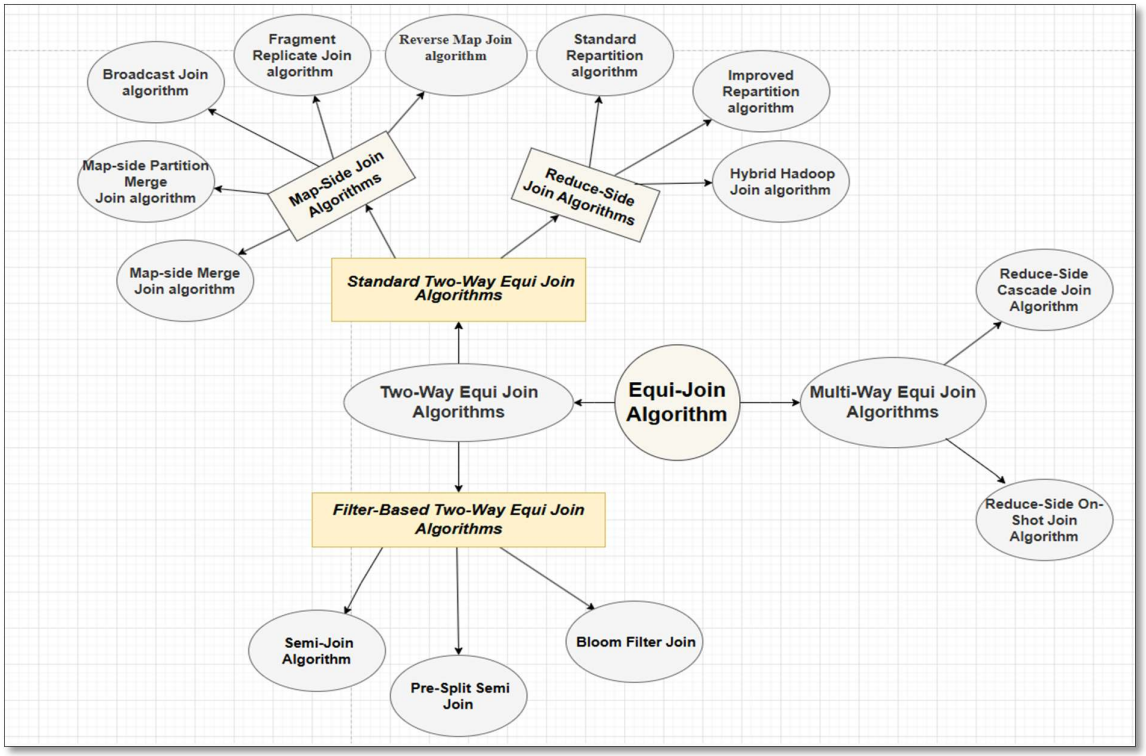

Fig 2: Two-way Equi join algorithms and Multi-way Equi join algorithms in MapReduce (Mohamed, M. et al., 2018)

The map-side join is more efficient for smaller datasets, while the reduce-side join algorithm (Pigul, A. et al., 2012)., is better suited for handling large datasets. The map-side join is more efficient when one dataset is small enough to fit into memory, as it eliminates the need for the reduce phase, resulting in fewer jobs, reduced data transfer, and faster execution times. However, its performance decreases when both datasets are large, as the smaller dataset may not always fit into memory.

Table 6: Comparison of various Two-Way Equi Map-Side Join Algorithms

Join Algorithm	Number of Jobs	Pre-processing	Cost Effectiveness	Execution Time	Strengths	Weaknesses
Map-Side Merge Join	1 job (if data is sorted)	Data must be pre-sorted by join key	Efficient when datasets are sorted; minimal cost	Very fast if data is sorted	Efficient for sorted data, no need for reduce phase	Requires pre-sorted data; inefficient for unsorted or large datasets
Map-Side Partition Merge Join	1 job	Data is partitioned based on the join key	Cost-effective for partitioned data	Faster than reduce-side join	Good for partitioned datasets, reduces shuffle overhead	Requires data partitioning; less effective for unsorted data

Broadcast Join	1 job	Broadcast smaller dataset to all mappers	High cost for large datasets (broadcasting overhead)	Fast for small datasets	Ideal for small datasets, reduces need for shuffling	Inefficient for large datasets due to high network overhead
Fragment Replicate Join	1 job (typically)	Replicate smaller dataset to each mapper	Cost-effective for small replicable datasets	Fast for small datasets	Effective for small replicated datasets	Replication overhead for large datasets; memory intensive
Reverse Map Join	2 jobs (mapper + reducer)	Requires reverse mapping or inverting keys	Cost-effective for certain join types	Slower than map-side join	Efficient for certain reversed key join conditions	More complex, requires additional mapping steps

As per Table 6, the Map-Side Merge Join is effective when pre-sorted datasets are present, but struggles with unsorted data. It can handle partitioned data, improving performance when partitioning is feasible. Broadcast Join and Fragment Replicate Join reduce network overhead by broadcasting smaller datasets, but face memory limitations. Reverse Map Join efficiently handles smaller datasets but has high memory usage for larger ones. Each algorithm overcomes specific limitations by leveraging strategies like partitioning or broadcasting. Broadcast Join is the best choice for smaller datasets in distributed environments due to its speed, simplicity, and efficiency. However, the choice of join algorithm should be based on the dataset's characteristics, such as size, partitioning, and memory capacity (Veiga, J et al., 2016).

On the other hand, the reduce-side join is more scalable for large datasets, handling the join in the reduce phase. Although it requires more jobs and higher execution time due to data transfer overhead, it can handle larger datasets that cannot be loaded into memory. Pre-processing is necessary to align join keys for the reduce phase. While slower, it is more adaptable to large datasets but less efficient for small datasets.

Table 7: Comparison of various Two-Way Equi Reduce-Side Join Algorithms

Criteria	**Repartition Join**	**Improved Repartition Join**	**Hybrid Hadoop Join**
Number of Jobs	2 MapReduce Jobs (for Repartition & Joining)	2 MapReduce Jobs (optimized for Skewed data)	3 MapReduce Jobs (for Repartition, Joining & Final Output)
Pre-processing	Requires partitioning of datasets	Minimal pre-processing (dynamic partitioning and redistribution of skewed data)	Extensive pre-processing (need partitioned or indexed)
Cost Effectiveness	Cost-effective for balanced datasets	More Cost-effective for skewed datasets	More cost-effective: one dataset is small, due to hybrid techniques (e.g., broadcast join)
Execution Time	High latency, slow execution with skewed datasets.	Faster execution with dynamic partitioning, balancing	Fast for small datasets, slow for large
Strengths	Simple, works well for balanced data, reduces network overhead.	Efficient with skewed data, dynamic partitioning.	Combines join techniques (e.g., Broadcast Join and Repartition Join), optimal for diverse dataset
Weakness	Inefficient with skewed data	Complex, requires tuning, inefficient for large datasets	Complex, computationally expensive for large datasets

As per Table7, Repartition Join (Lee, T. et al., 2014) is a simple and efficient method for balanced datasets, but struggles with skewed data. Improved Repartition Join handles skewed data and dynamic partitioning, making it effective for uneven distributions but more complex. Hybrid Hadoop Join, combining Broadcast and Repartition Join, is more flexible but computationally expensive and complex. The best algorithm depends on the dataset's size, distribution, and available processing resources. The filter-based two-way equi-join algorithm (Mohamed, M. et al., 2018) in MapReduce overcomes the limitations of the reduce-based two-way equi-join by efficiently reducing the amount of data shuffled across the network. In a reduce-based approach, all the matching keys need to be sent to the

same reducer, resulting in significant data transfer overhead, especially when the datasets are large. In contrast, the filter-based algorithm pre-filters the data using a smaller, representative subset (or filter) of the join key, which reduces the dataset size before the join operation. This minimizes the amount of data sent to the reducers, thereby improving the overall efficiency of the join process, reducing network congestion, and lowering the execution time.

Table 8 : Comparison of various Filter-based Two-Way Equi Join Algorithms

Join Algorithms	Semi-Join Algorithm	Pre-Semi Join Algorithm	Bloom Join Algorithm
Number of Jobs	2 jobs	1 job (or 2 jobs in some cases)	2 jobs
Pre-processing	Requires the smaller dataset to be filtered and sent to the larger dataset	Pre-filters the smaller dataset before transmission	Requires building a Bloom filter on the smaller dataset
Cost Effectiveness	Cost-effective when one dataset is much smaller than the other	More cost-effective than semi-join when the smaller dataset is heavily filtered	Cost-effective for large datasets with probabilistic filtering
Execution Time	Faster than full join, but requires 2 jobs	Faster than semi-join in some cases due to reduced data transmission	Execution time can be faster due to reduced data size in the second job
Strengths	Reduces the amount of data transferred by only sending necessary rows	Reduces data transmission by pre-filtering before join	Efficient for large datasets, reduces false positives through filtering
Weaknesses	Requires two jobs; still involves significant data transfer overhead	Requires an additional pre-processing step, may not be as efficient for all scenarios	Bloom filter introduces a chance of false positives, may require tuning

As per Table 8, semi-join algorithm is ideal for small datasets where it can efficiently transfer relevant data to another, reducing network traffic and execution time. It is cost-effective for skewed datasets. The pre-semi-join algorithm requires additional preprocessing to extract relevant keys and filter out unnecessary data, reducing the dataset size before the join. It is more complex and has a higher memory footprint. The Bloom Join algorithm (Barhoush, M. et al., 2019) uses a Bloom filter to probabilistically filter out irrelevant data, reducing network traffic and memory usage. It is ideal for large datasets where reducing data transfer costs is crucial. Each algorithm is suited for different use cases, depending on dataset size, filtering efficiency, and the need for preprocessing or memory constraints. Semi-join is more efficient for smaller datasets, Pre-Semi-Join for preprocessing-reducing data transfer, and Bloom Join for large-scale datasets, provided false positives are managed appropriately.

Multi-way equi-join algorithms (Barhoush, M. et al., 2019) in MapReduce reveals a range of techniques designed to efficiently perform joins across multiple datasets in a distributed setting. Traditional pairwise equi-join methods, including both reduce-based and filter-based approaches, are extended to handle multi-way joins by leveraging MapReduce's parallel processing capabilities. One common approach is to use a "broadcasting" technique, where smaller datasets are broadcasted to all mappers, allowing them to filter and reduce the data before applying the join. Another strategy involves the use of partitioning schemes, where data is divided into partitions based on join keys, ensuring that related records from different datasets are grouped together during the shuffle phase. Some algorithms focus on optimizing the shuffle and sort phases to handle large-scale data more efficiently by minimizing data transfer and balancing workload across nodes. Recent advancements also explore hybrid models that combine the strengths of both filter-based and reduce-based techniques, applying local filtering before the final join operation to reduce the data size. The key challenges addressed in these algorithms include handling skewed data distributions, optimizing data locality, and improving fault tolerance in large-scale distributed environments.

Table 9: Comparison of Reduce-Side One-Shot Join and Reduce-Side Cascade Join

Factor	Reduce-Side One-Shot Join	Reduce-Side Cascade Join
Number of Phases	1 (Single map and reduce phase)	Multiple phases (multiple map and reduce jobs)
Memory Usage	High memory usage per reducer (may cause bottlenecks)	Reduced memory load by joining in stages
Shuffling Overhead	High shuffling cost for large datasets	Reduced shuffling cost (intermediate results)
Scalability	Scalable for a small number of datasets but can struggle with many datasets	More scalable for large multi-way joins with many datasets
Complexity	Simpler to implement	More complex due to multiple stages
Execution Time	Faster for simple joins, may be slower for large multi-way joins	Potentially slower due to multiple phases but more efficient for large datasets
Ideal Use Case	Best for smaller datasets or fewer join keys	Better for large datasets with many joins, especially when memory is a concern

As per Table 9, Both Reduce-Side One-Shot Join and Reduce-Side Cascade Join are effective algorithms for multi-way equi-joins in MapReduce. Reduce-Side One-Shot Join is simpler and suitable for smaller datasets, but may not scale efficiently for large multi-way joins due to high memory usage and shuffle overhead. Reduce-Side Cascade Join breaks the problem into multiple stages, improving memory efficiency and reducing shuffle overhead, but introducing more complexity and requiring multiple MapReduce jobs (Al-Badarneh et al., 2022). The choice between these algorithms depends on dataset size, number of joins, and available system resources.

5. Conclusion

Two-way equi-join and multi-way equi-join algorithms are essential in distributed data processing, with broad applications in big data analytics, data warehousing, and more. While two-way joins, involving only two datasets, are simple, efficient, and work well for smaller datasets, multi-way joins, which handle three or more datasets, are more complex due to the increased memory requirements, shuffle phase overhead, and the challenge of integrating multiple datasets. The shuffle phase overhead for multi-way joins is significantly higher, as it involves more data exchanges between nodes, leading to increased memory usage compared to the moderate requirements of two-way joins. Performance tends to be faster for two-way joins on smaller datasets, but it becomes slower for multi-way joins, particularly as the number of datasets increases. Two-way joins scale well for small to medium-sized datasets, whereas multi-way joins require optimizations to handle larger datasets efficiently. Common use cases for two-way joins are simple joins between two datasets, while multi-way joins are more suited for complex data integration across multiple sources. Addressing data skew in multi-way joins is critical, with techniques like skew-insensitive joins, salting, and data partitioning helping to mitigate this issue. Advanced techniques such as bloom filters reduce data transfer during the shuffle phase, improving performance for large-scale joins. Hybrid joins, combining map-side and reduce-side processing, help balance load, and distributed query engines like Apache Hive and Apache Spark have optimized methods for handling multi-way equi-joins, abstracting complexity while applying advanced optimizations. As datasets grow, memory-efficient approaches like streaming or chunked processing are likely to become more widespread. Future developments in these algorithms are expected to focus on enhancing optimization techniques, reducing memory skew, and improving query engines to enable efficient multi-way joins across vast datasets.

6. References

1. Palla, K. (2009). A comparative analysis of join algorithms using the hadoop map/reduce framework. *Master of science thesis. School of informatics, University of Edinburgh*.
2. Afrati, F. N., & Ullman, J. D. (2011). Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering*, *23*(9), 1282-1298.
3. Chandar, J. (2010). Join algorithms using map/reduce. *Magisterarb. University of Edinburgh*.
4. Leu, J.S., Yee, Y.S. and Chen, W.L. (2010) 'Comparison of map-reduce and SQL on large-scale data processing', *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pp.244–248.
5. Blanas, S., Patel, J. M., Ercegovac, V., Rao, J., Shekita, E. J., & Tian, Y. (2010, June). A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (pp. 975-986).
6. Pigul, A. (2012). Comparative Study parallel join algorithms for MapReduce environment. *Труды Института системного программирования РАН*, *23*, 285-306.
7. Shaikh, A., & Jindal, R. (2012). Join query processing in mapreduce environment. In *Advances in Communication, Network, and Computing: Third International Conference, CNC 2012, Chennai, India, February 24-25, 2012, Revised Selected Papers 3* (pp. 275-281). Springer Berlin Heidelberg.
8. Lee, K. H., Lee, Y. J., Choi, H., Chung, Y. D., & Moon, B. (2012). Parallel data processing with MapReduce: a survey. *AcM sIGMoD record*, *40*(4), 11-20.
9. Lee, T., Im, D. H., Kim, H., & Kim, H. J. (2014). Application of filters to multiway joins in MapReduce. *Mathematical Problems in Engineering*, *2014*(1), 249418.
10. Shanoda, M. S., Senbel, S. A., & Khafagy, M. H. (2014, December). Jomr: Multi-join optimizer technique to enhance map-reduce job. In *2014 9th International Conference on Informatics and Systems* (pp. PDC-80). IEEE.
11. Sun, X. H., Qu, W., Stojmenovic, I., Zhou, W., Li, Z., Guo, H., ... & Liu, L. (Eds.). (2014, August). Algorithms and Architectures for Parallel Processing: 14th International Conference, ICA3PP 2014, Dalian, China, August 24-27, 2014. Proceedings, Part II. In *International Conference on Algorithms and Architectures for Parallel Processing 14*. Cham: Springer International Publishing.
12. Veiga, J., Expósito, R. R., Pardo, X. C., Taboada, G. L., & Tourifio, J. (2016, December). Performance evaluation of big data frameworks for large-scale data analytics. In *2016 IEEE International Conference on Big Data (Big Data)* (pp. 424-431). IEEE
13. Pal, S. (2016). SQL on Big Data: Technology, Architecture, and Innovation. Apress.
14. Mohamed, M. H., Khafagy, M. H., & Ibrahim, M. H. (2018). From Two-Way to Multi-Way: A Comparative Study for Map-Reduce Join Algorithms. *WSEAS Transactions on Communications*, *17*, 129-141.
15. Barhoush, M. M., AlSobeh, A. M., & Al Rawashdeh, A. (2019, April). A survey on parallel join algorithms using MapReduce on Hadoop. In *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)* (pp. 381-388). IEEE.
16. kumar Sahu, S., Chhualsingh, S., & Dora, S. G. Multi-Way Join using Map Reduce for Big Data Applications.
17. Pang, Z.,Wu, S., Huang, H., Hong, Z., & Xie, Y. (2021). AQUA+: Query Optimization for Hybrid Database-MapReduce System. *Knowledge and Information Systems*, *63*(4), 905-938.
18. Al-Badarneh, A. F., & Rababa, S. A. (2022). An analysis of two-way equi-join algorithms under MapReduce. *Journal of King Saud University-Computer and Information Sciences*, *34*(4), 1074-1085.