Replacing Hash Join with Sort-Merge Join in Neo4j

Muskan Moti Hassanandani
University of Southern California
hassanan@usc.edu

Aishwarya Krishnamurthy
University of Southern California
ak73043@usc.edu

Nisarg Shihora
University of Southern California
shihora@usc.edu

***Abstract*—Neo4j, a leading graph database, utilizes hash joins as its default join strategy, optimized for equality-based operations. However, hash joins face significant challenges when applied to graph data involving range queries, large datasets, and pre-sorted data. These limitations can lead to high memory consumption, inefficient query processing, and performance bottlenecks. To overcome these challenges, we implemented a sort-merge join strategy tailored to Neo4j's architecture. Sort-merge joins leverage sorted datasets to efficiently merge results, reducing memory overhead by eliminating the need for in-memory hash tables. This approach significantly improves the performance of range queries, optimizes the execution of pre-sorted queries, and scales better with large graphs. Sort-merge joins are particularly advantageous for applications where data can be naturally ordered, such as time-series analysis, logistics, and social network recommendations. This report explores the design and implementation of sort-merge joins in Neo4j, presenting specific use cases where this method outperforms hash joins. Additionally, we discuss limitations such as sorting overhead and propose solutions like hybrid join strategies and indexing optimizations. By incorporating these strategies, Neo4j can achieve more efficient query execution, better memory utilization, and enhanced scalability, addressing the needs of modern graph-based applications**

I. Introduction

Neo4j is a leading graph database management system designed to store and manage highly connected data efficiently. Unlike traditional relational databases, Neo4j models data as nodes, representing entities, and relationships, representing connections between these entities, with both nodes and relationships containing properties as metadata. This structure allows Neo4j to excel at tasks involving relationship-heavy queries such as social networks, fraud detection, recommendation engines, and network analysis. Neo4j uses the Labeled Property Graph (LPG) model, providing a natural way to represent real-world relationships. One of its key features is ACID compliance, ensuring reliable transactions, and index-free adjacency, which enables rapid graph traversals. Neo4j leverages the declarative Cypher query language, designed to be intuitive and human-readable, allowing users to perform complex pattern-matching operations on graph data efficiently. Cypher's expressiveness and ease of use make querying and updating graph data seamless, supporting tasks like finding shortest paths, identifying communities, and analyzing networks. Neo4j's scalability, with support for clustering and sharding, makes it suitable for handling large-scale datasets. However, while Neo4j uses hash joins by default for combining datasets, this approach can lead to high memory consumption and inefficiencies with range-based or pre-sorted data. To address these challenges, implementing sort-merge joins in Neo4j offers improved performance by leveraging sorted datasets, reducing memory overhead, and enhancing query efficiency for large graphs and complex traversals. These improvements make Neo4j a versatile and powerful tool for applications that rely on understanding and analyzing interconnected data.

This report is divided into three main sections.

1) Proposed Solution outlines the motivation for introducing sort-merge joins in Neo4j to address the inefficiencies of the default hash join approach. This section discusses how sort-merge joins improve performance for range queries, pre-sorted data, and memory-intensive operations by leveraging sorted datasets and reducing the need for in-memory hash tables.
2) Methodology details the implementation process of sort-merge joins in Neo4j, including the design considerations, algorithmic steps, and integration with Neo4j's query execution engine. It also covers the techniques used to optimize sorting, handle large datasets, and dynamically select join strategies based on query patterns and data characteristics.
3) Results presents a comprehensive evaluation of the implemented sort-merge join strategy. This section includes performance benchmarks, memory usage comparisons, and specific use cases demonstrating the improvements in query efficiency. The results highlight the advantages of sort-merge joins over hash joins in scenarios involving range queries, large-scale datasets, and pre-sorted data, providing insights into how this approach enhances Neo4j's overall performance and scalability.

A. Motivation

Neo4j's reliance on hash joins for query execution poses challenges when dealing with large datasets, range-based queries, and pre-sorted data, which are common in real-world applications. In **customer purchase analysis** [1], retail companies analyzing large transaction datasets benefit from reduced memory consumption and improved join efficiency compared to hash joins. In **social networks** [2], where friend recommendations rely on connection data sorted by timestamps or activity scores, sort-merge joins eliminate redundant sorting, making these queries faster and more efficient. For IoT sensor networks, where time-series data needs to be queried within specific time ranges, sort-merge joins efficiently merge sorted sensor readings, enhancing performance for range-based

queries. In **fraud detection** [3], banks analyzing financial transactions within specific amount ranges experience better memory utilization and improved scalability with sort-merge joins over hash joins. Lastly, in logistics and pathfinding, sort-merge joins optimize route calculations by efficiently handling travel time intervals, improving route optimization for transportation networks.

II. Proposed solution

Neo4j is a leading graph database, designed for high-performance querying and flexibility when navigating graph relationships. The architecture of Neo4j consists of two important components: one is a graph-optimized Storage Engine, and the second is a Query Processor utilizing Cypher, Neo4j's Graph Query Language. In order to reduce computational overhead, the query processor breaks down queries into operators that handle data in a pipelined fashion. This also results in increased performance

For simpler joins, Neo4j efficiently explores graph relations using pattern matching techniques like MATCH (a)-[r]− >(b) [4]. However, for more complex queries that require explicit joins, Neo4j relies on the hash join algorithm. Existing join strategies in Neo4j are ValueHashJoin and NodeHashJoin, which are primarily triggered based on equality conditions

A. Value-Based Joins

Here, node values act as join keys. For example, when two datasets are joined based on a common attribute, such as product price. This method creates a hash table and then probes it for matches

B. Node-Based Joins

In certain queries, the join is performed on node identities rather than properties, so the node variables are used as keys. Such queries arise, for example, when we match (o1: Order)-[:CONTAINS]− >(p: Product) and (o2: Order)-[:CONTAINS]− >(p: Product) to find a common product node p. Node-based joins are necessary to combine these disparate match results, where the node p itself serves as the join key.

Limitations and how we resolve them:

1) **Sorting Overhead:**
   Sort-merge joins require datasets to be sorted before merging, which can introduce significant computational overhead when dealing with unsorted data. This overhead can be mitigated by leveraging Neo4j's indexing capabilities to ensure datasets are pre-sorted whenever possible. For frequently queried datasets, maintaining sorted copies or caching previously sorted results can reduce the need for repeated sorting operations, improving overall efficiency.
2) **Sorting Overhead:**
   When dealing with small datasets, the sorting step required by sort-merge joins can negate the performance benefits compared to hash joins, which are faster for smaller data sizes. To address this, Neo4j can adopt a hybrid join strategy where the system dynamically selects between hash joins and sort-merge joins based on the dataset size and query complexity. This approach ensures that the most efficient join method is applied, optimizing query performance across different use cases.
3) **Increased Disk I/O for Large Datasets:**
   Sorting large datasets that do not fit entirely in memory can lead to excessive disk I/O, slowing down query execution. This issue can be mitigated by employing efficient external sorting algorithms that minimize disk access and by optimizing Neo4j's storage configuration for large-scale datasets. Additionally, implementing batch processing techniques can help reduce disk I/O by processing data in manageable chunks, thereby maintaining performance even when working with very large graphs.
4) **Limited Support for Non-Comparable Data:**
   Sort-merge joins require data that can be ordered, making them unsuitable for non-comparable or complex data types, such as unstructured text or nested properties. To overcome this limitation, preprocessing data to convert complex types into comparable formats (e.g., converting text to standardized numeric codes) can make sort-merge joins feasible. In cases where data cannot be ordered effectively, Neo4j can fall back to using hash joins to ensure flexibility and efficiency.
5) **Query Planner Integration:**
   Neo4j's query planner may not automatically optimize for sort-merge joins, particularly if it has been primarily designed for hash joins. Enhancing the query planner to recognize range conditions, pre-sorted datasets, and other scenarios where sort-merge joins would be advantageous can improve query execution. Additionally, providing manual query hints for advanced users to specify the desired join method can offer greater control and optimize performance for complex queries.

III. Methodology

The core of this project was to integrate the SortMergeJoinOperator into Neo4j's existing query execution engine. Here, we chose a straight replacement approach, which is replacing the HashJoinOperator in the execution pipeline with the SortMergeJoinOperator. This method allowed for an instant performance evaluation of the new operator while minimizing changes in the planner code

We paid special attention to integrating this, so the new operator remains compatible with the rest of the already existing execution pipeline-to have good interaction between pre- and future operators like scan, filtering, and aggregate operators.Additionally, it maintains compatibility with slot mappings, operator interfaces, and existing data structures (Slotted Row). Our modifications were made primarily in

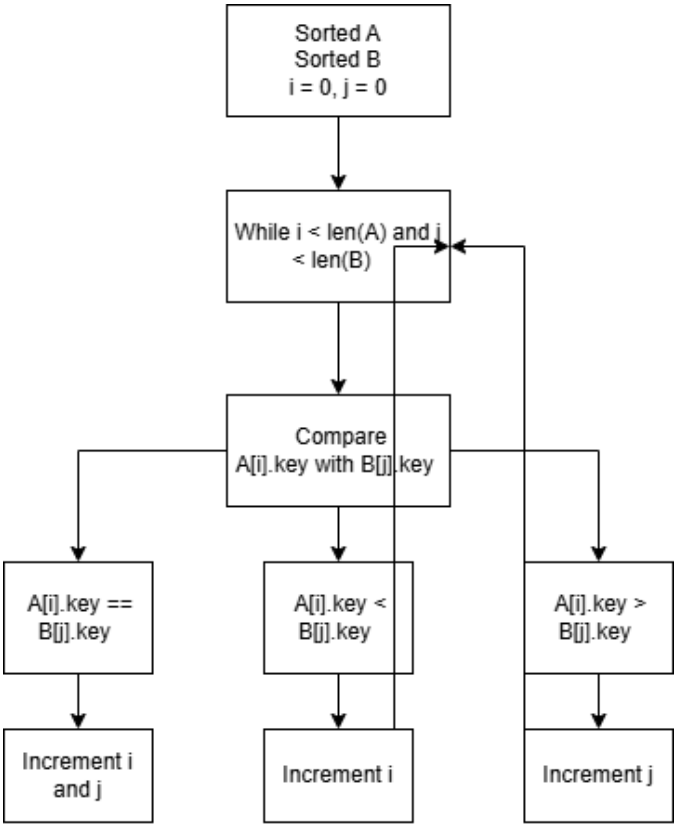

Fig. 1: methodology

SlottedPipeMapper.scala and the newly introduced pipes (ValueSortMergeJoinSlottedPipe and NodeSortMergeJoinSlottedPipe). We internally route queries that would typically result in HashJoin to our Sort-Merge Join implementation. Our algorithm of Sort-Merge Join as shown in Fig 1

A. Input Extraction

We first retrieve both left and right inputs.In node-based joins, we refer to the IDs of the node variables, but in value-based joins, we extract property values as join keys. We define the order of the selected keys using Neo4j's built-in Values.COMPARATOR.

```
val lhsKeyedRows = lhsInput.flatMap { row =>
        val key = computeKey(row, leftSide, state
        ↪ )
        key.map(k => (k, row))
    }

val rhsKeyedRows = rhsInput.flatMap { row =>
        val key = computeKey(row, rightSide,
        ↪ state)
        key.map(k => (k, row))
    }
```

B. Sorting Phase

Each input list is sorted on their own with respect to the join keys. This may be by node or property values. Sorting is performed using Scala's native sorting facilities. After this step, both sets of data are in ascending order.

```
val sortedLeft = lhsKeyedRows.sortBy(_._1)(
↪ ValueComparator)
val sortedRight = rhsKeyedRows.sortBy(_._1)(
↪ ValueComparator)
```

C. Merge Phase

With both inputs sorted, we proceed as illustrated in the flow diagram (see Figure 1). We maintain pointers i and j for the left and right sorted lists, respectively:

- Compare left[i].key to right[j].key.
- If left[i].key == right[j].key, we output merged rows left[i] and right[j]
- If left[i].key $<$ right[j].key, we increment i to catch up.
- If left[i].key $>$ right[j].key, we increment j.

```
override def hasNext: Boolean = {
  if (nextOutput.isDefined) return true

  while (leftRowOpt.isDefined && (rightRowOpt.
  ↪ isDefined || rightBuffer.nonEmpty)) {
    if (rightBuffer.isEmpty) {
      val (leftKey, leftRow) = leftRowOpt.get
      val (rightKey, rightRow) = rightRowOpt.get

      val cmp = ValueComparator.compare(leftKey,
      ↪ rightKey)
      if (cmp < 0) {
        advanceLeft()
      } else if (cmp > 0) {
        advanceRight()
      } else {
        // Matching keys
        currentLeftKey = leftKey
        rightBuffer = collectMatchingRightRows(
        ↪ rightKey)
        if (rightBuffer.nonEmpty) {
          nextOutput = Some(joinRows(leftRow,
          ↪ rightBuffer.head))
          rightBuffer = rightBuffer.tail
          return true
        } else {
          advanceLeft()
        }
      }
    } else {
      // We have buffered right rows with
      ↪ matching keys
      val leftRow = leftRowOpt.get._2
      nextOutput = Some(joinRows(leftRow,
      ↪ rightBuffer.head))
      rightBuffer = rightBuffer.tail
      if (rightBuffer.isEmpty) advanceLeft()
      return true
    }
  }

  false
}
```

This linear scan and comparison allow us to process large data sets efficiently, avoiding random lookups in hash tables. All potential matches have been generated by the time the merging phase reaches the end of one list.

IV. Design

A Cypher query is converted by Neo4j's execution pipeline into a logical plan, which is then converted into a physical plan made up of a pipeline of operators, or pipes (Fig 2).Each operator processes data as rows, and passes intermediate results downstream which could be an abstract syntax tree or rows. Neo4j mentions two runtime implementations: the interpreted runtime, which uses a more straightforward, evaluation-based approach to execute queries, and the slotted runtime, which employs a slot-based data representation for improved performance and memory efficiency [5]. To improve outcomes, data is filtered and aggregated after joining as done in HashJoin where each row is organized into fixed slots, with each slot intended for a specific piece of data. This fixed format makes it possible to get and manipulate data quickly. [6]

V. Benchmarking

It was necessary to understand how the replacement of **Hash Join** with **Sort-Merge Join** in Neo4j is going to perform and how that modification is going to be reflected for a set of four different query types. These query types have been carefully chosen to best represent the wide range of real-world use cases that can be used to evaluate the efficiency and effectiveness of the join strategy. Key metrics assessed in performance evaluation are query execution time, memory consumption, and scalability considering variable data size and complexity.

Each query type was explained in detail, having in view its role within the analysis and in connection with practical database operations. Comparing execution speeds with the resource utilization brings out the strengths and weaknesses of each technique. Trends and insights are provided using charts and graphs that, for clarity, shed light on the trade-offs. The discussion is structured and analytical, ensuring the findings are relevant and can be acted upon, both for research and practical purposes.

A. Dataset Description

For testing we used Northwind dataset [7]: This is one of the most accessible and popular sample databases on the web. Northwind, being a realistic business model - a trading company - is thus quite ideal for testing join performance. The dataset consists of a number of tables like Customers, Orders, Products, Suppliers, and Order Details, among others. These exhibit a wide range of diverse attributes and relationships, hence allowing a wide range of various possible joins that can be tested. Features in the dataset include:

- Customers Table: This would include demographic and geographic information about a customer.
- Orders table: Customer orders details.

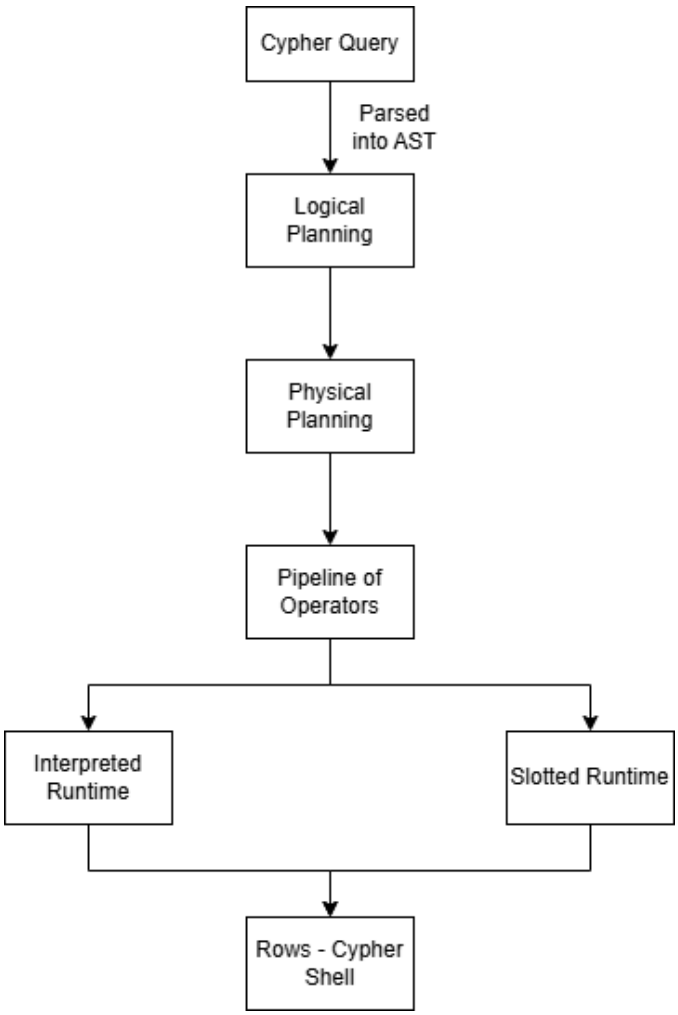

Fig. 2: flow

- Products Table: Catalog information of the products with the respective supplier details.
- Suppliers table: Vendors that supply these products.
- Order Details Table: This is a kind of junction table that provides the details for each order at the atom level.

It finds the dataset particularly suitable for the study because it has primary and foreign key relationships, hence allowing different kinds of joins such as equality, range, and pre-sorted.

B. Query Types and Performance

The evaluation of replacing the hash join with the sort-mix join in Neo4j has been done based on four types of queries. These are selected as representatives of general operations/scenarios that may occur during the operation of the database. Each query type reflects aspects that both join algorithms will have positive and negative sides depending on the characteristic of the dataset and requirements of queries. Details of each type with queries are given below.

1) **Joining on Unique Identifiers:**
This query type involves joining datasets based on

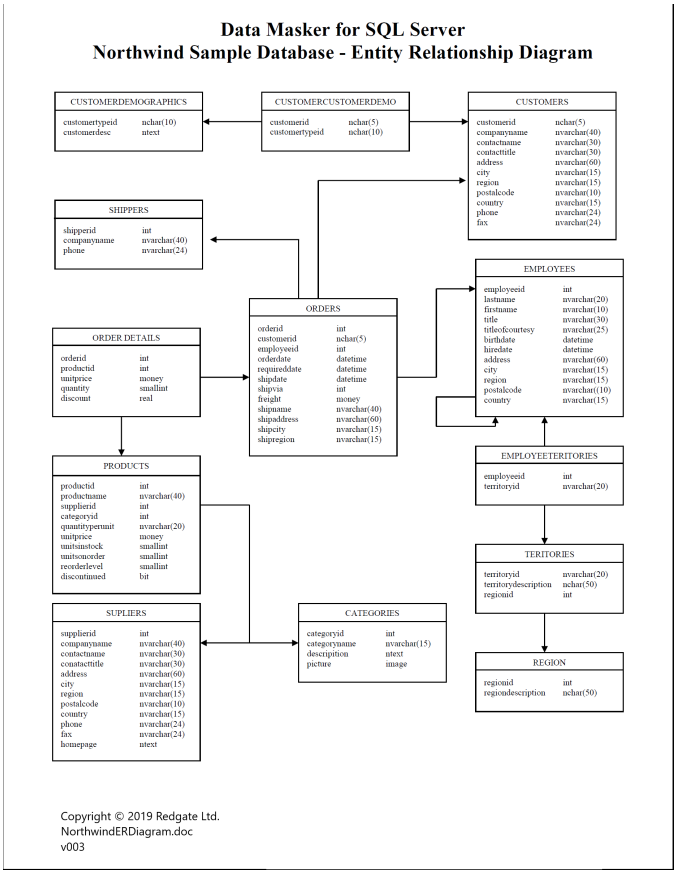

Fig. 3: NorthWind ER

unique identifier fields, such as user IDs or product codes. The uniqueness of the identifiers ensures a direct and efficient matching process. Hash Join demonstrates optimal performance due to its ability to quickly locate matches using hash tables.

```
PROFILE MATCH (c:Customer), (o:Order)
WHERE c.customerID = o.customerID
RETURN
c.companyName AS Customer,
o.orderID AS OrderID;
```

2) **High Selectivity Joins:**
This query operate on conditions that filter out a big part of the dataset, returning small-sized results. Because of fast lookups, Hash Join can do this discarding of non-matching rows fast and therefore is most suitable for such queries.

```
PROFILE MATCH (e:Employee), (o:Order)
WHERE e.employeeID = o.employeeID
AND o.freight > 1000
RETURN
e.firstName, e.lastName, o.orderID, o.freight;
```

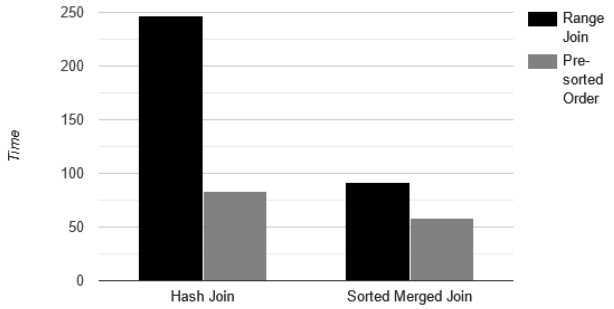

Fig. 4: Hash Join Vs Sorted Merge Join on Range and Pre-sorted order query

3) **Range Joins:**
Range joins are queries that have conditions on ranges, such as selecting data between a certain range of values. Sort-Merge Join is very effective in this context, since the sorted datasets enable the range conditions to be evaluated without additional hashing or filtering.

```
PROFILE MATCH (p1:Product), (p2:Product)
WHERE p1.unitPrice < p2.unitPrice
RETURN
p1.productName, p2.productName, p1.unitPrice,
p2.unitPrice;
```

4) **Pre-Sorted Order Joins:**
Sort-Merge Join is advantageous in scenarios where data is pre-sorted, as it avoids the overhead of sorting while ensuring efficient merging. This makes it an ideal choice for queries requiring ordered results or involving sorted datasets.

```
PROFILE MATCH (p:Product), (o:Order)
WHERE p.productID = o.productID
RETURN p.productName, o.orderID;
```

C. Visual Comparison

Fig 4. is the performance comparison between Hash Join and Sorted Merge Join for two key query types: Range Joins, Pre-Sorted Order Joins, High Selectivity Join and Joining on Unique Identifiers. which presents the execution times (in milliseconds) of the two join algorithms.

Fig 4 shows that the execution time of Hash Join is much higher, about **250 ms**, as compared to Sorted Merge Join, which takes only about **92 ms**, hence offering high performance improvement about **63%** for Sorted Merge Join. The reasons for such good performance of Sorted Merge Join in this case are as follows:

- Sorted datasets leverage the fact that the whole design

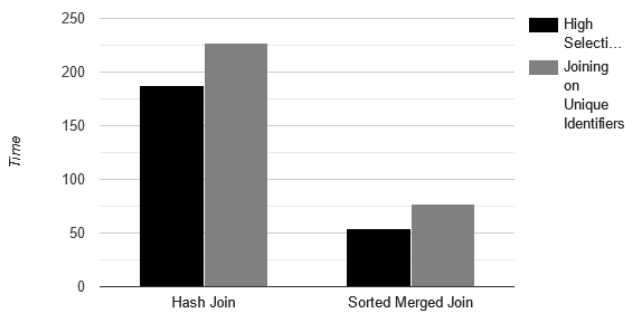

Fig. 5: Hash Join Vs Sorted Merge Join on High Selectivity and Unique Identifiers query

of Sorted Merge Join is to work on already sorted data. Because it iterates over the sorted input, it can evaluate range conditions ($<$, $>$, $<=$, $>=$) directly without any extra lookups or intermediate operations.
- Unlike the Hash Join, which has to compute hash values for the range keys and performs expensive lookups for each pair, Sorted Merge Join processes data in a sequential manner, avoiding computation overheads associated with creating and managing hash tables.
- Since its complexity depends on the amount of sorted input and not the number of hash operations, Sorted Merge Join scales much better when the size of the data set increases. This would prove most useful in queries containing very large datasets or for any operation that requires sorting from a disk.

In contrast, Hash Join has poor performance for range-based conditions owing to the nature of hash tables, which are intrinsically optimized for equality rather than comparisons of ranges. Because of this, Hash Join adds significant computational overhead to the process and accounts for higher execution time in such scenarios.

Important is the fact that Sorted Merge Join really stands out when Range Joins should be handled efficiently, which, if the sorted datasets are readily available or could be preprocessed with very minimal overhead.

Fig 4 illustrates that the execution time of Hash Join is about **83 ms**, while for Sorted Merge Join, it is around **58 ms**, which means an increase in performance by **30%**. The difference is not as radical as in the case of Range Joins, but here, too, the efficiency of Sorted Merge Join is considerable. Sorted Merge Join performs better because of the following design advantages:

- In the case of already sorted inputs, Sorted Merge Join can immediately merge the inputs by comparing elements in order and without any extra sorting or intermediate computation.This reduces the query execution time as no extra operation is needed, as in the case of Hash Join, which needs to hash the join key and look up.
- Since Sorted Merge Join works on already sorted input, the result of such a join is naturally ordered and doesn't require further post-processing. It turns out to be very useful for queries with an ORDER BY clause, in which the join results shall be sorted according to a given field

On the other hand, Hash Join introduces additional overhead when processing pre-sorted data:

- Even though the data is already sorted, Hash Join requires creating a hash table for one of the datasets, an action computationally redundant in context.
- Hash Join does not preserve input order, and if results need to come ordered, additional sorting steps are required, further increasing execution time

Indeed, these observations reinforce the advantages of using Sorted Merge Join in the case of Pre-Sorted Order Joins. This, although quite effective in equality-based conditions within Hash Join, constitutes unnecessary overhead when operating over already sorted datasets; hence, Sorted Merge Join is preferable for such scenarios.

High Selectivity Join refers to the selectivity condition being so much restricted to shrink the sizes of a dataset produced in output significantly smaller. Such a query is performed very often in situations when only some subset of data is meaningful, targeting either analysis or selective reports.

Fig 5 shows that Hash Join has an execution time of **188 ms** while Sorted Merge Join has a slower execution time of **227 ms**, which is a performance difference of **20.7%** in favor of Hash Join. Hash Join performs better here because it can:

- Filter out the unselective filters efficiently: It builds a hash table and discards rows that do not match as early as possible, avoiding wasteful computation.
- Reduced Overhead of Sorting: Hash Join does not sort the dataset unnecessarily for such types of queries, whereas maintaining efficiency even in scenarios where the size of input data is huge.

In contrast, Sorted Merge Join has lower efficiency for high-selective queries because it requires input data sets to be sorted prior to joining, adding computation overhead when volumes of data are huge. Inefficiency with Sparse Output: Due to the low output size, the benefit of merging the sorted datasets is dominated by the overheads of sorting.

One class of queries in both transactional and relational databases involves joining data sets in conditions of equality, concerning unique fields, for instance, primary keys or User IDs. These queries are intrinsically efficient since the uniqueness of the keys applied to join can already allow for efficiency.

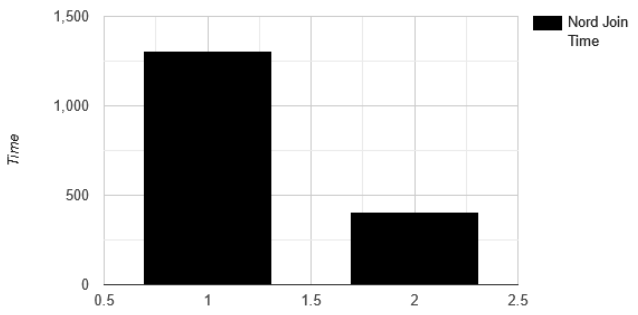

Fig. 6: Hash Join Vs Sorted Merge Join on Node Join

Fig 5 Execution time of Hash Join is **54 ms**, while Sorted Merge Join takes **77 ms**. Thus, Hash Join performs the task **42.6%** faster Hash Join is better in this case because of the following reasons:

- Hash Join creates a hash table for one dataset and then enables fast lookups of matching rows; this is extremely effective when the conditions are based on equality.
- Hash Join avoids sorting, hence minimizing extra steps in processing joins on unique identifiers.

Sorted Merge Join, even though efficient, lags behind because:

- Even though the datasets are joined on unique identifiers, the nature of Sorted Merge Join necessitates that the input datasets have to be sorted before merging-an action that serves to uselessly add to the computational overhead.
- Since a Sorted Merge Join has not had equality-based joins in mind in its design, hence it is less powerful as compared to Hash Join.

Fig 6. showing the performance of **Node Join** represented in terms of execution time in milliseconds. This chart has two bars, each representing two different scenarios or executions. Observations obtained from this chart are labeled below.

1) Execution Time Comparison:
   First, the bar on the left shows around **1300 ms** execution time, which is just way too high. The second bar has an execution time of about **400 ms** showing performance improvement between the two executions.
2) Possible Causes of Performance Improvement:
   The difference in the performance of the two executions point to the effect of the caching and Query optimization mechanisms in the database:
   - Mechanism of Caching:
     The database might have done some extra work during the first execution, such as sorting, loading of data, creation of structures for the join, etc. In the second execution, either the results were cached or some intermediate steps were pre-computed that reduced the execution time significantly.
   - Query Optimization:
     The first execution could be costlier in terms of computation because of a lack of optimizations. Subsequent executions benefited from the runtime adjustments.
3) Resource Utilization:
   Although the chart does not show it, because the database operations are identical, it is likely that both executions processed the same amount of data and returned results over the same data. This would be consistent with the behavior of databases, where execution time is improved because data is cached or pre-sorted rather than resource usage changes
4) Implications:
   The Node Join exhibits efficient performance during subsequent executions, leveraging database caching and query preprocessing. However, the high initial execution time indicates that the join algorithm may rely on intermediate steps, such as sorting or indexing, that introduce overhead in the absence of precomputed results.

CONCLUSION

This Neo4j comparative analysis between Hash Join and Sort-Merge Join for different query types yields a good trade-off and relative efficiencies of each join strategy. Execution time, memory utilization, and scalability analyses are considered for four representative query types: Joining on Unique Identifiers, High Selectivity Joins, Range Joins, and Pre-Sorted Order Joins. The results emphasize that the choice of the joining strategy should be compatible with the query requirements and characteristics of the dataset under consideration.

Hash Join showed excellent performance in cases of Joining on Unique Identifiers and High Selectivity Joins, where the dominant conditions are equalities. It is ideal for these kinds of scenarios since it could hash lookups fast and remove the non-matching rows right from the start of the execution of the query. Its performance gets worse in range-based or ordered queries because of inherent limitations in hash tables. Another weakness of the former is that heavy reliance on hashing operations also introduces extra overhead, especially as the dataset size increases.

On the other hand, Sort-Merge Join was always superior to Hash Join in Range Joins and Pre-Sorted Order Joins. It easily handled both range-based conditions and ordered queries with amazing efficiency since it could take advantage of sorted datasets for direct merging. Sort-Merge Join has very good performance for large queries or queries that return sorted outputs, since it avoids extra hashing and maintains the order of the inputs. For this algorithm, performance is hit since sorted input is required. If the data happens to be unsorted, extra overhead may be required.

These results indicate the flexibility and strength of Sort-Merge Join when the joining attributes are already sorted or range-based conditions are present, while Hash Join remains very strong in equality-based joins where one or both of the joining datasets are small or highly selective. The study further shows how caching and query optimization bring down execution times, as can be seen from the improvement in performance in subsequent executions for both join strategies.

The important conclusion of this study is that modern graph databases, such as Neo4j, need adaptive query optimization mechanisms. Being able to dynamically choose the best join strategy given query patterns and characteristics of datasets will be of great value in balancing practical applications' performance, scalability, and resource efficiency. These will provide further insights into more advanced query optimization techniques, such as hybrid join strategies, which would mix and match the strengths of Hash Join and Sort-Merge Join.

REFERENCES

[1] Shoujin Wang, Liang Hu, Yan Wang, Xiangnan He, Quan Z. Sheng, Mehmet A. Orgun, Longbing Cao, Francesco Ricci, Philip S. Yu, "Graph Learning based Recommender Systems: A Review"

[2] Nitai B. Silva; Ing-Ren Tsang; George D. C. Cavalcanti; Ing-Jyh Tsang, "A graph-based friend recommendation system using Genetic Algorithm," Barcelona, Spain.

[3] Tahereh Pourhabibi, Kok-Leong Ong, Booi H. Kam, Yee Ling Boo, "Fraud detection: : A systematic literature review of graph-based anomaly detection approaches," Volume 133, Elsevier Science Publishers B. V.

[4] Jürgen Hölsch Tobias Schmidt Michael Grossniklaus, "On the Performance of Analytical and Pattern Matching Graph Queries in Neo4j and a Relational Database," .O. Box 188, 78457 Konstanz, Germany.

[5] Peter Boncz, Marcin Zukowski, Niels Nes, "MonetDB/X100: Hyper-Pipelining Query Execution" CWI, Kruislaan 413, Amsterdam, The Netherlands.

[6] "Neo4j, Neo4j Documentation", v5.0. [Online]. Available: https://neo4j.com/docs/. [Accessed: Dec. 10, 2024].'

[7] "YugabyteDB. (n.d.). Northwind Sample Dataset. YugabyteDB Documentation. Retrieved December 12, 2024, from https://docs.yugabyte.com/preview/sample-data/northwind/"