*Journal of Institute of Science and Technology*, 30(2), 127-135 (2025)
*ISSN: 2467-9062 (print), e-ISSN: 2467-9240*
https://doi.org/10.3126/jist.v30i2.84384


*Research Article*

# IMPACT OF JOIN ORDER AND MAIN MEMORY BUFFER SIZE IN QUERY OPTIMIZATION WITH MATERIALIZATION

***Nawaraj Paudel[1*], Manish Pokharel[2], Bal Krishna Bal[2]***
*[1]Central Department of Computer Science and Information Technology, Tribhuvan University*
*[2]Department of Computer Science and Engineering, Kathmandu University*
*[*]Correspondence: nawarajpaudel@cdcsit.edu.np*
***(Received:** September 14, 2025; **Revised:** December 02, 2025; **Accepted:** December 02, 2025)*

## ABSTRACT

Query optimization is one of the crucial steps during query processing in any database management system that determines the best way to run a high-level query, such as SQL (Structured Query Language). To optimize any query, the query optimization process choses the most efficient execution plan from multiple execution plans. For a given high level query, the query optimization process selects an efficient execution plan to enhance performance of the database. Out of different factors that influence query optimization, the most crucial ones are join order and the use of main memory buffer and this paper focuses on the impact of these two factors in query optimization. In this paper, we consider all left deep tree join orders of relations and cost of each join order is calculated using different main memory buffer sizes with materialization. The cost of join-orders is calculated by considering number of secondary memory blocks accessed during query execution. The result of intermediate join operation is also materialized and the cost of the join orders is calculated. This study highlights that different join orders with different join selectivity and different main memory buffer sizes have significant impact on query optimization. The query with the value of join selectivity less than one and larger main memory buffer reduces the number of secondary memory blocks accessed and hence reduces the cost of join operation. The cost of join queries can further be reduced significantly by using indexes.

**Keywords:** Left-deep tree, Join cardinality, Join selectivity, Materialization

## INTRODUCTION

A database is an organized collection of data that allows easy access, retrieval and manipulation of data. Query processing in databases is used to execute a query like SQL (Structured Query Language) efficiently and return requested data to the user. Query optimization is the most important step of query processing which is used to select best execution strategy for any high-level query like SQL. A single query can have multiple execution strategies and query optimization process choses most efficient strategy to execute any query (Divya *et al.*, 2024). Query optimization is important for increasing the performance of any database management system as it selects best execution strategy for a given query from multiple strategies with different cost estimations. As there are different techniques, approaches and algorithms for query optimization in relational databases, studying these things is useful in the study of query optimization (Mor *et al.*, 2012; Paudel & Bhatta, 2019).

A single high-level query can have multiple tables and the cost of this query is also determined by the join order of these tables. Hence, selecting the best join order is an important task in query optimization. Join order optimization is the process of determining the order of tables in order to minimize intermediate result of Join operations. The minimized intermediate result reduces the computational load. Main memory buffer is used to execute the join operation and also to store intermediate results of join operations in multi-tale joins. The size of main memory buffer is crucial for query optimization and it plays a significant role to reduce total time to execute a database query. The main memory buffer also helps to keep frequently accessed data and intermediate results in memory during query processing. The main objective of considering join-order and main memory buffer is to reduce the total number of disk blocks accessed during query processing (Elmasri & Navathe, 2016). Materialization is the process of storing result of a subquery temporarily in the secondary memory and plays a significant role is query optimization. Pipelining on the other hand is the process of directly passing intermediate results to the next operation without storing the intermediate results in secondary memory (Graefe, 1993; Elmasri & Navathe, 2016). Instead of pipelining, this study considers materialization of intermediate results of Joins of multi-table join queries.

One of the most difficult problems in query optimization is estimating the cardinality of join queries while they are being executed. Hence, accurate cardinality estimation is important in query optimization problem (Qiu *et al.*, 2021). When calculating the cardinality of the join queries, neither learning-based nor classical approaches produce results that are sufficient. They either construct extensive models to comprehend the complex data distributions, which results in lengthy planning durations and a lack of generalizability across queries, or they rely on oversimplified assumptions that produce poor cardinality estimations (Wu *et al.*, 2023).

Query optimization, especially join query optimization, has been a fundamental field of research in databases and has a critical impact on database performance. Joins are often the most resource-intensive operations in query execution, and their efficiency depends heavily on factors such as join order, algorithm selection, and resource allocation such as main memory buffer. During past years, different algorithms and strategies have been proposed and used to address computational and disk I/O issues presented by join operations, including cost-based, heuristic, and adaptive techniques. Because of the popularity of in-memory databases and distributed databases with high volume and velocity of data, the studies have been also focused for the join query optimization for these environments.

Authors of (Jarke & Koch, 1984) focused their study in query optimization in centralized database systems. The authors presented different strategies to improve performance of query evaluation algorithms. The different strategies presented were logic-based and semantic transformations, quick implementations of fundamental operations, and combinatorial or heuristic algorithms for generating alternative access plans and choosing among them. Other optimization problems including the use of database machines, higher-level query evaluation, and optimizing queries in distributed databases are also covered.

Query optimization problem with many joins was examined by the authors of (Swami & Gupta, 1988). The authors used heuristic approach for selecting best join order of the query. For the big join query, they used simulated annealing and iterative improvement methods and analyzed the effectiveness of these methods using ANOVA (analysis of variance) and factorial experiments. The outcome of this study showed that iterative improvement outperformed simulated annealing approach.

Authors of (Azhir *et al.*, 2022) used a hybrid Multi-Objective Genetic Algorithm with BAT (MOGABAT) to generate best query execution plan. This algorithm was also compared with Multi-Objective BAT (MOBAT) and Non-dominated Sorting Genetic Algorithm II (NSGA-II) on different join graph structures. The final outcome of this study showed that the MOGABAT outperformed MOBAT and NSGA-II to generate query execution plans with different join graph structures. However, the runtime of MOGABAT was longer than that of the MOBAT and NSGA-II.

Bayesian optimization algorithm was used to quickly estimate the cardinality of complex multi-join queries with multiple tables. To minimize the error rate of the estimation, two-layer sampling method was then used in a limited amount of time. After comparing with random search algorithm, the final result showed that the suggested approach lowered the error rate in the estimation of cardinality for complex queries involving join operations with multiple tables. The experiment was conducted on TCP-H dataset and the result showed that the Bayesian optimization algorithm reduces the cardinality estimate error rate by 54.8% to 60.2% when compared with the random search approach (Qian *et al.*, 2022).

Authors of (Schuh *et al.*, 2016) studied thirteen different equi-join implementations with various workloads and hardware configurations. They also considered partitioned and non-partitioned has joins in this study. This study revealed that no single join algorithm outperformed others and the performance of each algorithm varies significantly depending on the factors, such as, data distribution, input sizes, hardware architecture, and parallelism. The result also revealed that non-partitioned hash joins outperformed when the probing side fits in the CPU cache or when input relations are imbalanced and partitioned joins outperformed on multi-core systems with balanced input sizes. The study highlighted that even small system-level tweaks, like making the system aware of NUMA or using write-combine buffers, can make a big difference.

In recent years, deep reinforcement learning techniques has produced encouraging results for choosing best join order of the queries with multiple tables. Authors of (Ji *et al.*, 2023) proposed Dynamic Double Deep Q-Network Optimization Strategy (DDOS). This technique was especially designed for join order optimization with multiple tables. This approach uses Markov Decision Process (MDP) for join order selection and Double Deep Q-Network (DDQN) to learn effective join orders. The authors also proposed a dynamic progressive exploration technique to enhance exploration and speed up learning. The experiment was carried out on Join Order Benchmark (JOB) dataset and final result showed that DDOS outperformed dynamic programming, greedy heuristics, and other reinforcement learning methods in both reducing cost and improving execution performance.

Since, few studies have examined the impact of join order and main memory buffer size, this study aims to identify the impact of these factors during query execution. Here, we have also considered materialization of the intermediate result of join operations during query processing which involves creating and storing intermediate results during query processing, which can significantly affect the performance of the query. In this study, we have only considered left-deep-tree join orders to simplify the join enumeration process and reduce the complexity of the optimization space.

## MATERIALS AND METHODS

### Database Used

The database used to in this research is the World database (MySQL AB, 2019). The World Database is a sample database provided by MySQL, containing real-world geographical and population data. It is commonly used for learning and testing SQL queries in MySQL database. As shown in the database schema in Figure 1, this database contains only three tables: City table with 5 attributes, Country table with 15 attributes, and *CountryLanguage* table with 4 attributes. The Country table stores data related to countries, the City table stores data related to cities, and the CountryLanguage table stores data related to languages spoken in each country.

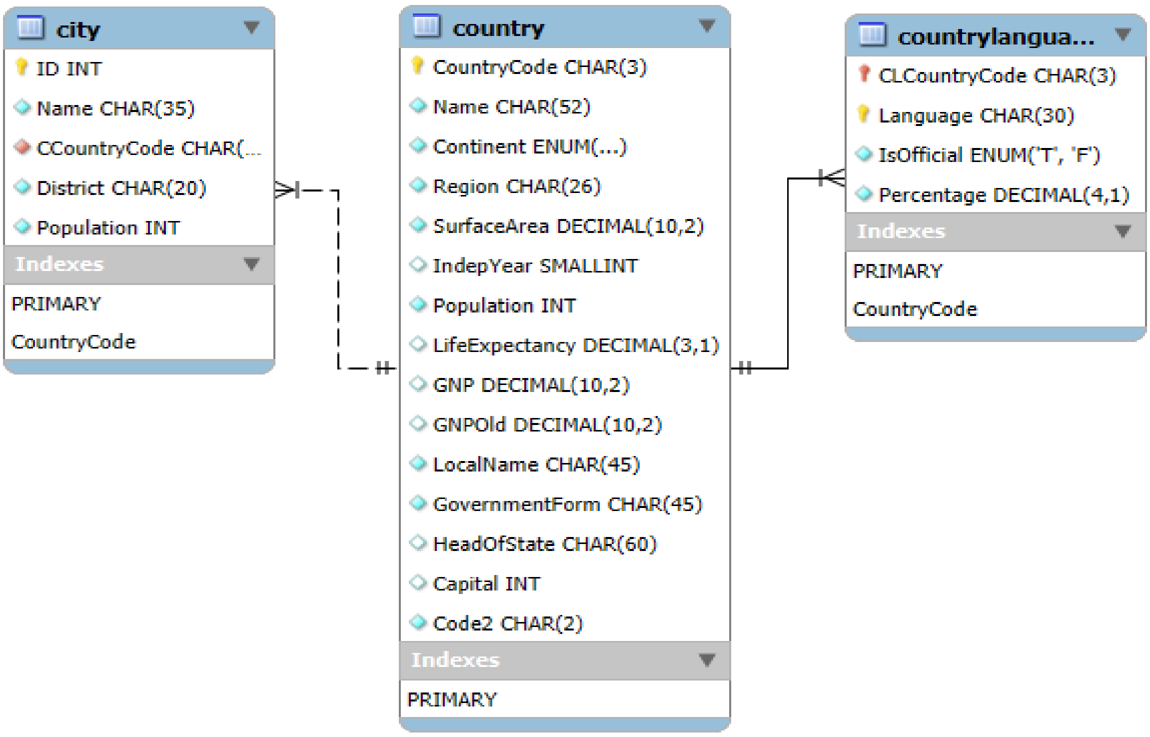


**Figure 1. Database schema of the World database.**

The Table 1 shows the storage related metadata of the tables City, Country, and CountryLanguage. Table 1 shows the detail information about number of records stored in, number of bytes consumed by, and number of secondary memory blocks allocated to each table. The number of blocks taken by the City, Country, and CountryLanguage tables in the MySQL world database depends on factors like storage engine, page size, and how MySQL organizes data internally. The storage engine used is InnoDB (Janjua *et al.*, 2022) which stores data in pages and the page size used is 16 KB (default page size of InnoDB). The number of blocks taken by City, Country, and CountryLanguage tables is 25, 7, and 6 respectively.

**Table 1. Number of rows, bytes, and blocks.**

| Table | Rows | Bytes | Blocks |
|---|---|---|---|
| City | 4046 | 409600 | 25 |
| Country | 239 | 114688 | 7 |
| CountryLanguage | 984 | 98404 | 6 |

Figure 2 presents metadata details for the tables in the database. The metadata are number of rows in the table and storage blocks occupied by each table in the secondary memory. This information provides insight into the size and storage footprint of each table, which can influence query performance and optimization strategies.

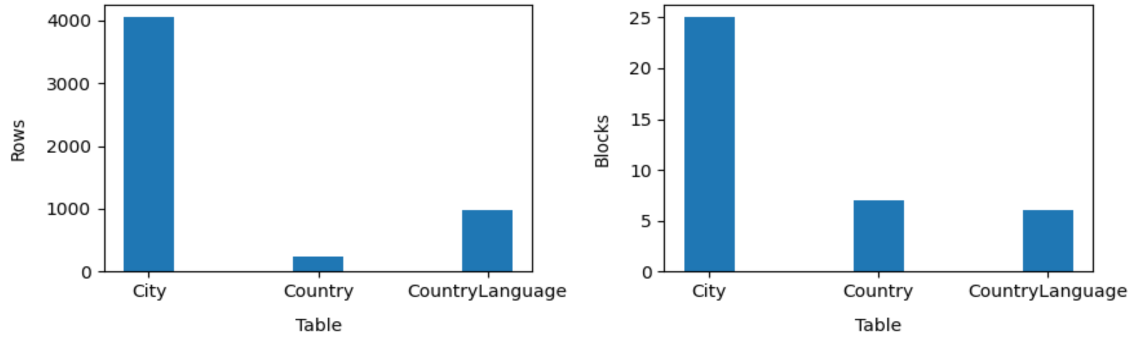


**Figure 2. Number of rows and blocks in each table**

Table 2 shows metadata information related to join columns. The Country table has *CountryCode* attribute as primary key and the City table has *CCountryCode* as foreign key. Similarly, the Country table has CountryCode attribute as primary key and the CountryLanguage table has *CLCountryCode* as foreign key. These key pairs are used as join columns in join operations. The tables City and CountryLanguage do not have join columns. Number of distinct values for join attributes CountryCode, CCountryCode, and CLCountryCode are 239, 232, and 233 respectively.

**Table 2. Metadata related to join column of tables.**

| Table | Join column | Distinct values | Referenced table | Referenced column | Referenced distinct |
|---|---|---|---|---|---|
| Country | CountryCode | 239 | City | CCountryCode | 232 |
| Country | CountryCode | 239 | CountryLanguage | CLCountryCode | 233 |

Figure 3 illustrates number of unique values of join columns in different database tables. These distinct value counts are critical statistical metadata used by query optimization process in order to estimate *selectivity* of join operations. Selectivity refers to the ratio of number of rows that satisfy the join condition the total number of possible combinations of rows from joined tables. A lower selectivity implies fewer output rows, which typically results in faster query execution. Accurately estimating join selectivity is essential for choosing the most efficient join strategy and execution plan, especially in large-scale databases where performance can significantly vary depending on join order and method.

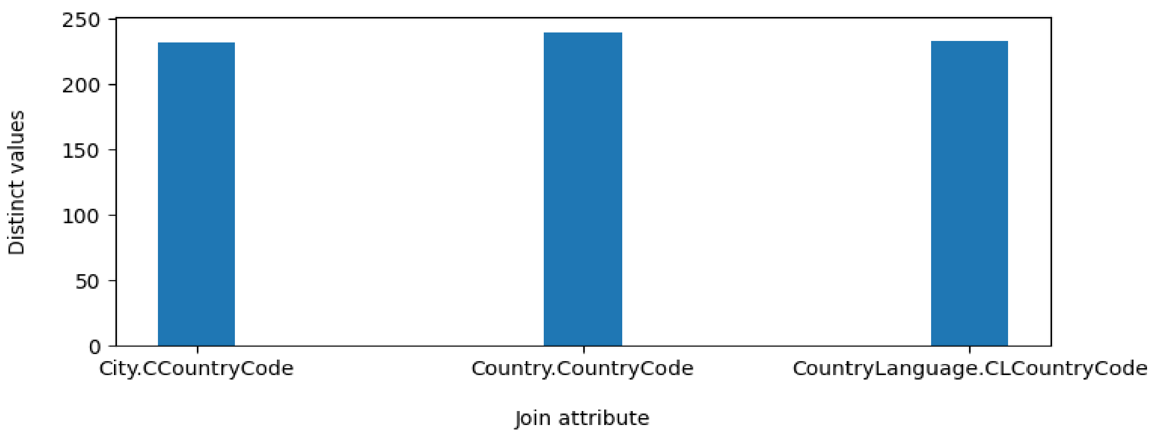


**Figure 3. Number of unique values in join attributes.**

**Cost Estimation**

For estimating the cost of a join operation, join selectivity is used. Join selectivity refers to the proportion of record pairs from two tables that satisfy the join condition. The value of the join selectivity is always between zero and one. The query optimizer in databases estimates join selectivity and then join cardinality to select best query execution plan (Repas *et al.*, 2023). The optimizer uses join selectivity as a major component to choose the best execution plan for join queries. The join selectivity is used to determine the cost of the Join operation between two tables. For two relations R and S and join condition A = B, where A is an attribute of R and B is an attribute of S, the join selectivity is calculated using basic formula given below.

$$Join\ Selectivity = \frac{|R \bowtie_{A=B} S|}{|R| \times |S|} \text{ ................. (1)}$$

Where:

- $|R|$ = number of rows in table R
- $|S|$ = number of rows in table S
- $|R \bowtie_{A=B} S|$ = number of rows after join

Assuming uniform data distribution and independent attributes, the join selectivity for equijoin operation given in the equation (1) can also be estimated using the approximation formula as given below.

$$Join\ Selectivity = \frac{1}{max(V(R,\ A),\ V(S,\ B))} \text{ ...... (2)}$$

Where:

- $V(R, A)$ = number of unique values of attribute A in table R
- $V(S, B)$ = number of unique values of attribute B in table S

The database management system then estimates join cardinality using the join selectivity. Join cardinality is the basis for cost-based query optimization (Chen *et al.*, 2021). This value is estimated using the formula given below.

$$Join\ cardinality = Join\ Selectivity\ \times |R| \times |S| \text{ .... (3)}$$

Where:

- $|R|$ = number of rows in table R
- $|S|$ = number of rows in table S

The estimated cost of the join is then calculated using nested loop join algorithm. This cost is the total number of secondary memory blocks accessed by the algorithm.

$$B(RES) = \frac{Join\ Cardinality}{BFR(RS)} \text{ ..................... (4)}$$

$$cost = B(R) + \left(\left\lceil\frac{B(R)}{(B(M)-2)}\right\rceil \times B(S)\right) + B(RES) \text{..(5)}$$

Where:

- $B(RES)$ = number of secondary memory blocks required to store result of the join between tables R and S
- $BRF(RS)$ = blocking factor of the join result between tables R and S
- $B(R)$ = number of secondary memory blocks occupied by the table R
- B(S) = number of secondary memory blocks occupied by the table S
- B(M) = number of main memory buffer blocks

This study has been carried out using three tables R, S, and T. The cost of join in this case with materialization of the join result of first two tables R and S is estimated using the formula given below.

$$cost = B(R) + \left(\left\lceil\frac{B(R)}{(B(M)-2)}\right\rceil \times B(S)\right) + 2 \times B(RES) + \left(\left\lceil\frac{B(RES)}{(B(M)-2)}\right\rceil \times B(T)\right) \text{ ........ (6)}$$

Where:

- B(T) = number of secondary memory blocks occupied by the table T

**Experimental Setup**

This research was carried out using Python programming language, MySQL Connector/Python (Krogh, 2018), and the libraries such as Pandas, NumPy, Itertools, Math, and Matplotlib. The computer with Microsoft Windows 10 Pro, Intel(R) Core (TM) i5-10210U CPU @ 1.60GHz 2.11 GHz, and 8GB RAM configuration was used. The database used is the MySQL World database which is a sample database provided by MySQL for learning and practicing. (MySQL AB, 2019)

**RESULTS AND DISCUSSION**

The experimental evaluation demonstrates the critical role of join order and main memory buffer size in determining the performance of left-deep-tree join queries with materialization. This experiment was carried out using the metadata information related to tables, information related to join columns and all possible left-deep tree join order permutations of the tables in the World Database. In join query optimization, the order in which joins are executed can significantly affect performance of the database. The query optimizer explores different join orders to choose the most efficient execution plan. This experiment was conducted to access the performance of join orders with left-deep tree only. The Table 3 shows all possible join orders (or permutations) considering only left-deep join trees used during optimization. A left-deep join tree is a binary tree in which right child of each join is always a base relation and left child is either a base relation or the result of previous join. The tree with the least estimated cost will be selected by the optimizer for executing the query. The Figures 4 shows left-deep tree representations of all join order permutations in Table 3.

**Table 3. Join order permutations of left-deep trees only.**

| Join Order | Execution Order (Left to Right) |
|---|---|
| Join Order 1 (JO1) | City ⋈ Country ⋈ CountryLanguage |
| Join Order 2 (JO2) | City ⋈ CountryLanguage ⋈ Country |
| Join Order 3 (JO3) | Country ⋈ City ⋈ CountryLanguage |
| Join Order 4 (JO4) | Country ⋈ Countrylanguage ⋈ City |
| Join Order 5 (JO5) | CountryLanguage ⋈ City ⋈ Country |
| Join Order 6 (JO6) | CountryLanguage ⋈ Country ⋈ City |

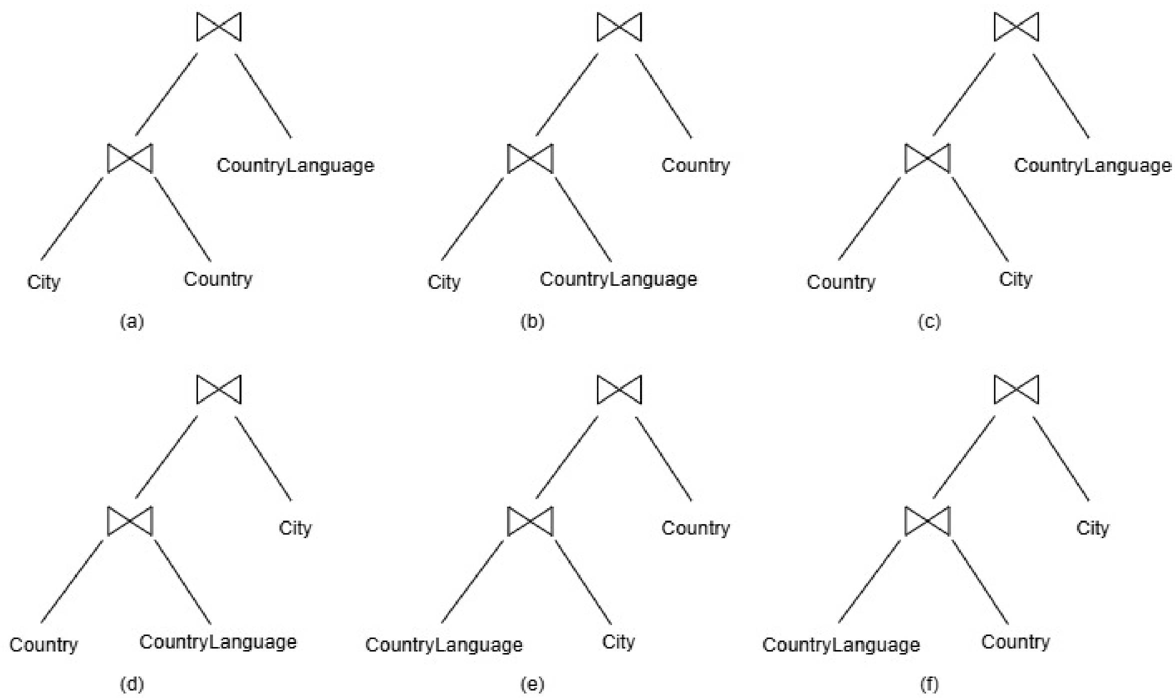


**Figure 4. (a) Left-deep tree of Join Order 1. (b) Left-deep tree of Join Order 2. (c) Left-deep tree of Join Order 3. (d) Left-deep tree of Join Order 4. (e) Left-deep tree of Join Order 5. (f) Left-deep tree of Join Order 6**

The Table 4 shows the estimated cost of all join orders with different main memory buffer blocks. The number of main memory buffer blocks used in this experiment are 3, 8, 13, 18, 23, 28, 33, 38, 43, and 48. To estimate the final cost, blocking factor value of 100 is used. As shown in the Table 4, the estimated cost of the Join Order 6 (JO6) is lowest and that of Join Order 2(JO2) is highest with all main memory buffer blocks. The Join Order 6 (JO6) is 1128, 1525, 1820, 1683, 1611, 1566, 1536, 1515, 1498, and 1486 times faster than Join Order 2 (JO2) with the number of main memory blocks 3, 8, 13, 18, 23, 28, 33, 38, 43, and 48 respectively. The result also shows that larger main memory buffer reduces the estimated cost of join orders significantly. Figure 5 is the graphical representation of Table 4 and shows the visual interpretation of estimated cost of all join orders with different main memory block sizes. For all join orders, this graph clearly shows that cost decreases significantly as the size of main memory increases and it reduces more significantly when the main memory block is increased from 3 to 8. It can also be clearly seen that the rate of decrease in cost is almost similar for all join orders.

**Table 4. Estimated cost of join orders with different main memory blocks.**

| Blocks | JO1 | JO2 | JO3 | JO4 | JO5 | JO6 |
|---|---|---|---|---|---|---|
| 3 | 526.92 | 358491.3 | 508.92 | 318.68 | 358472.3 | 317.68 |
| 8 | 182.92 | 126132.3 | 179.92 | 88.68 | 126108.3 | 82.68 |
| 13 | 150.92 | 105008.3 | 136.92 | 57.68 | 104996.3 | 57.68 |
| 18 | 137.92 | 97085.28 | 130.92 | 57.68 | 97079.28 | 57.68 |
| 23 | 131.92 | 92934.28 | 124.92 | 57.68 | 92928.28 | 57.68 |
| 28 | 124.92 | 90380.28 | 124.92 | 57.68 | 90380.28 | 57.68 |
| 33 | 124.92 | 88651.28 | 124.92 | 57.68 | 88651.28 | 57.68 |
| 38 | 124.92 | 87398.28 | 124.92 | 57.68 | 87398.28 | 57.68 |
| 43 | 118.92 | 86460.28 | 118.92 | 57.68 | 86460.28 | 57.68 |
| 48 | 118.92 | 85718.28 | 118.92 | 57.68 | 85718.28 | 57.68 |

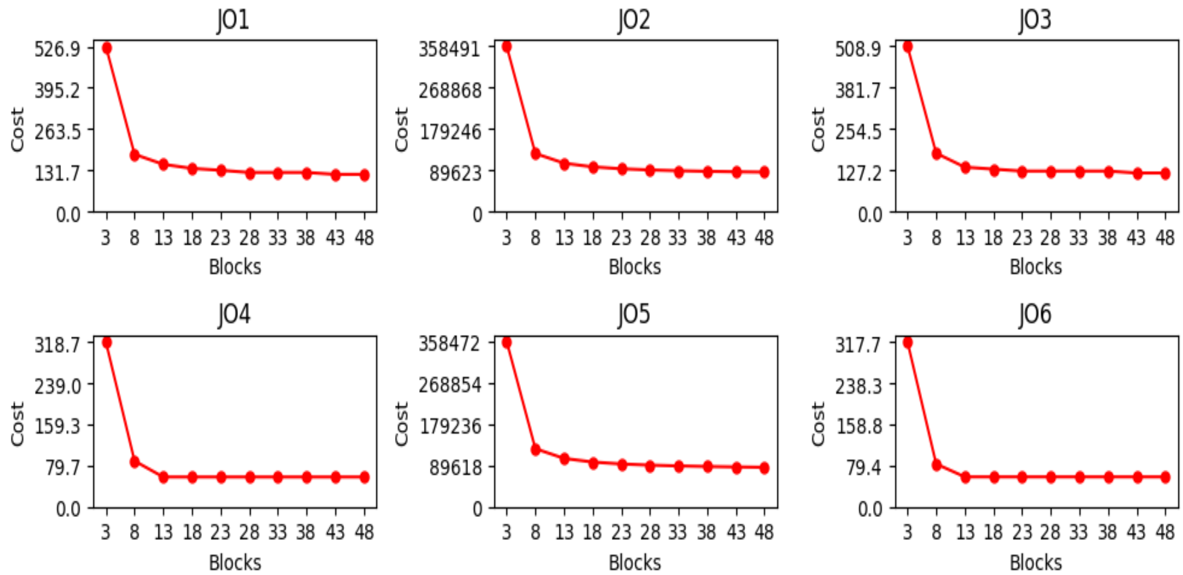


**Figure 5. Estimated cost of different join orders with different main memory blocks.**

Table 5 presents the percentage decrease in the estimated cost of all possible join orders as the number of available main memory blocks increases. This table quantitatively demonstrates how memory availability impacts the efficiency of join operations during query execution. It can also be clearly seen that the cost reduces significantly with change in main memory blocks from 3 to 8 than other observations. Figure 6 is the graphical representation of the data in Table 5, offering a visual interpretation of the cost reductions. This indicates that the optimizer is able to utilize additional memory to reduce intermediate result materialization, buffer more data pages, or select more efficient join strategies, such as, reducing the need for disk I/O operations. Beyond this range, the rate of cost reduction becomes more gradual, suggesting diminishing returns in performance improvement with further increases in memory. This observation highlights the critical role of memory availability in query optimization and its influence on overall query execution cost.

**Table 5. Percentage decrease in cost with change in memory blocks.**

| Change in Blocks | JO1 | JO2 | JO3 | JO4 | JO5 | JO6 |
|---|---|---|---|---|---|---|
| From 3 to 8 | 65.29 | 64.82 | 64.65 | 72.17 | 64.82 | 73.97 |
| From 8 to 13 | 17.49 | 16.75 | 23.90 | 34.96 | 16.74 | 30.24 |
| From 13 to 18 | 8.61 | 7.55 | 4.38 | 0.00 | 7.54 | 0.00 |
| From 18 to 23 | 4.35 | 4.28 | 4.58 | 0.00 | 4.28 | 0.00 |
| From 23 to 28 | 5.31 | 2.75 | 0.00 | 0.00 | 2.74 | 0.00 |
| From 28 to 33 | 0.00 | 1.91 | 0.00 | 0.00 | 1.91 | 0.00 |
| From 33 to 38 | 0.00 | 1.41 | 0.00 | 0.00 | 1.41 | 0.00 |
| From 38 to 43 | 4.80 | 1.07 | 4.80 | 0.00 | 1.07 | 0.00 |
| From 43 to 48 | 0.00 | 0.86 | 0.00 | 0.00 | 0.86 | 0.00 |

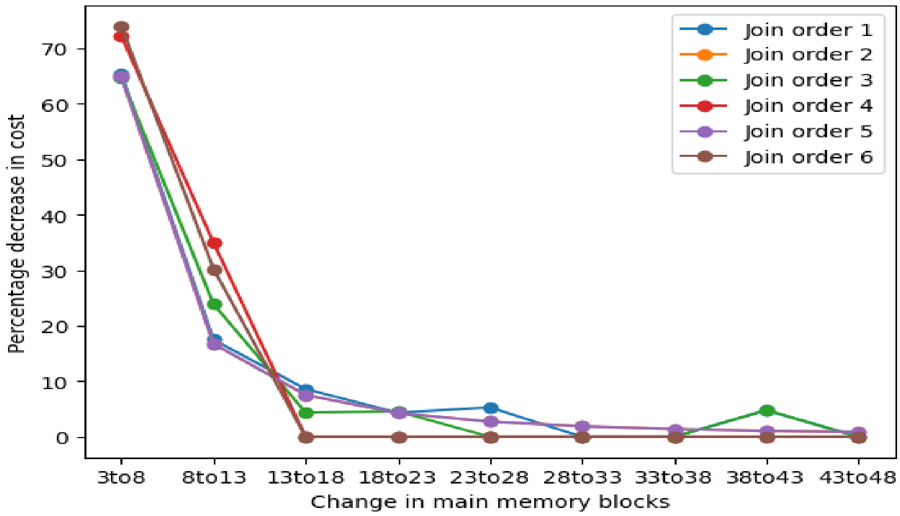


**Figure 6. Percentage decrease in cost with change in memory blocks.**

## CONCLUSIONS

This study highlights the critical role of join order selection and number of main memory buffer blocks allocated for the join operation in the performance of relational query processing, particularly under materialization-based execution strategies. Experimental results demonstrate that poor join order and number of main memory block choices can lead to significant increases in intermediate result sizes thereby increasing the overall cost of query execution. This study also focuses on left-deep join trees only as these trees are well suited for sequential processing and materialization, as they allow the right child to be a base relation, enabling efficient access patterns and reduced memory overhead. Therefore, an effective query optimizer must dynamically consider both join order and available buffer size to choose the most cost-effective execution plan.

This study focuses on analyzing the performance impact of join order and buffer size considering left-deep join trees and materialization. The main contribution of this study is to identify the impact of join order, main memory buffer size, and materialization during query processing. Several directions are still open for further exploration in this domain. Future work can explore join query optimization having more than two join operations, use of indexes, adaptive query optimization and machine learning based optimization that adapt better to dynamic execution environments and diverse workloads.

## ACKNOWLEDGEMENTS

The authors would like to express their sincere gratitude to all the faculty members of Department of Computer Science and Engineering, Kathmandu University and Central Department of Computer

Science and Information Technology, Tribhuvan University for their valuable suggestions and constructive feedback. We are especially thankful to the Graduate Research Coordinator of School of Science, Kathmandu University for valuable guidance, encouragement, and constructive feedback.

## AUTHOR CONTRIBUTION

Conceptualization: NP; Investigation: NP; Methodology: NP; Data curation: NP; Data analysis: NP, MP, BKB; Writing - original draft: NP; Writing - review and editing: NP, MP, BKB.

## CONFLICT OF INTEREST

The authors declare no conflict of interest.

## ETHICAL STATEMENT

The authors confirm that the study is unique and has not been published or submitted for publication anywhere else and the study was conducted in accordance with the ethical standards.

## DATA AVAILABILITY STATEMENT

Upon reasonable request, the corresponding author will provide the data generated, collected, and analyzed during the current study.

## REFERENCES

Azhir, E., Navimipour, N. J., Hosseinzadeh, M., Sharifi, A., Unal, M., & Darwesh, A. (2022). Join queries optimization in the distributed databases using a hybrid multi-objective algorithm. *Cluster Computing*, *25*(3). https://doi.org/10.1007/s10586-021-03451-9.

Chen, L., Huang, H., & Chen, D. (2021). Join cardinality estimation by combining operator-level deep neural networks. *Information Sciences*, *546*. https://doi.org/10.1016/j.ins.2020.09.065.

Divya, V. L., Job, P. A., & Mathew, P. K. (2024). Secure query processing and optimization in cloud environment: a review. In *Information Security Journal* (Vol. 33, Issue 2). https://doi.org/10.1080/19393555.2023.2270976.

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of Database Systems* (7th ed.). Pearson.

Graefe, G. (1993). Query Evaluation Techniques for Large Databases. *ACM Computing Surveys (CSUR)*, *25*(2). https://doi.org/10.1145/152610.152611.

Janjua, J. I., Khan, T. A., Zulfiqar, S., & Usman, M. Q. (2022). An Architecture of MySQL Storage Engines to Increase the Resource Utilization. *2022 International Balkan Conference on Communications and Networking, BalkanCom 2022*. https://doi.org/10.1109/BalkanCom55633.2022.9900616.

Jarke, M., & Koch, J. (1984). Query Optimization in Database Systems. *ACM Computing Surveys (CSUR)*, *16*(2). https://doi.org/10.1145/356924.356928.

Ji, L., Zhao, R., Dang, Y., Liu, J., & Zhang, H. (2023). Query Join Order Optimization Method Based on Dynamic Double Deep Q-Network. *Electronics (Switzerland)*, *12*(6). https://doi.org/10.3390/electronics12061504.

Krogh, J. W. (2018). MySQL Connector/Python Revealed: SQL and NoSQL Data Storage Using MySQL for Python Programmers. In *MySQL Connector/Python Revealed: SQL and NoSQL Data Storage Using MySQL for Python Programmers*. https://doi.org/10.1007/978-1-4842-3694-9.

Mor, J., Kashyap, I., & K. Rathy, R. (2012). Analysis of Query Optimization Techniques in Databases. *International Journal of Computer Applications*, *47*(15). https://doi.org/10.5120/7262-0127.

MySQL AB. (2019). *MySQL :: Other MySQL Documentation*. https://dev.mysql.com/doc/index-other.html.

Paudel, N., & Bhatta, J. (2019). Cost-Based Query Optimization in Centralized Relational Databases. *Journal of Institute of Science and Technology*, *24*(1). https://doi.org/10.3126/jist.v24i1.24627.

Qian, W., Jing, Y., Wang, X., & Wu, Z. (2022). Cardinality Estimation Method for Multitable JOIN Query Optimization. *Jisuanji Gongcheng/Computer Engineering*, *48*(6). https://doi.org/10.19678/j.issn.1000-3428.0061625.

Repas, D., Luo, Z., Schoemans, M., & Sakr, M. (2023). Selectivity Estimation of Inequality Joins in Databases. *Mathematics*, *11*(6). https://doi.org/10.3390/math11061383.

Schuh, S., Chen, X., & Dittrich, J. (2016). An experimental comparison of thirteen relational equi-joins in main memory. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, *26-June-2016*. https://doi.org/10.1145/2882903.2882917.

Swami, A., & Gupta, A. (1988). Optimization of large join queries. *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, 8–17. https://doi.org/10.1145/50202.50203.

Wu, Z., Negi, P., Alizadeh, M., Kraska, T., & Madden, S. (2023). FactorJoin: A New Cardinality Estimation Framework for Join Queries. *Proceedings of the ACM on Management of Data*, *1*(1). https://doi.org/10.1145/3588721.