Effect of JOIN Type on Query Performance

Andrea Meleková, Michal Kvet
University of Zilina
Žilina, Slovakia
Andrea.Melekova@uniza.sk,Michal.Kvet@uniza.sk

***Abstract*—Optimal query performance is crucial for database systems that operate in real-time or process large volumes of queries. In this paper, we present a comparative study of different SQL JOIN operations in an Oracle environment, with an emphasis on comparing INNER, LEFT, RIGHT and FULL OUTER JOIN, as well as the impact of using ON and USING clauses. We employ a cost-based optimizer method and measure query execution times for different dataset sizes and different indexing strategies, while also tracking CPU utilization, I/O operations, and memory consumption. Our findings indicate that INNER JOIN benefits most from indexing. These findings can help database administrators and developers choose an appropriate table join strategy, thereby reducing query latency and making resource utilization more efficient in a high-load environment.**

I. Introduction

SQL query optimization is critical for both research and practical applications, particularly in high-load environments where execution speed directly impacts system performance [1]. In the case of the Oracle database, which is one of the commercial systems with extensive optimization possibilities, even small differences in syntax or type of JOIN used often make a difference [1]. If two different query formulations return an identical set of records, it does not necessarily mean that they will be executed equally fast or that the cost-based optimizer will use the same table join method. This study analyzes how INNER, LEFT, RIGHT, and FULL OUTER JOIN types affect execution time and optimizer decisions, considering index usage and query structure.

Interest in this area is also growing because most modern databases offer multiple ways to write the same join logic. The presence or absence of indexes, the size of the dataset, and the setting of the cost-based optimizer are other factors that can affect the final query execution plan. With this paper, we aim to contribute to a better understanding of the behavior of Oracle databases under different variants of JOINs, to highlight different optimization approaches, and to offer an empirical comparison that can serve as a practical guide for administrators and developers.

In this study, we selected Oracle as the database system for our experiments due to its advanced cost-based optimizer and extensive indexing capabilities, which make it a suitable environment for analyzing the impact of different JOIN strategies. Additionally, Oracle is used as a partner in our research project. However, the techniques and insights presented in this paper are generally applicable across relational database systems, as modern SQL optimizers in other DBMSs, such as PostgreSQL and MySQL, use similar principles for query execution planning and optimization.

II. State of the Art

SQL query optimization plays a crucial role in database performance, particularly when processing large datasets under high concurrency. Modern database systems use cost-based optimizers (CBOs) that analyze different possible execution plans and select the one with the lowest estimated performance [2] . As reported by Mehta et al. [3], cost-based optimizers rely on statistical data on tables and indexes and aim to minimize the number of operations required to process a query. In Oracle databases, the optimization process is highly sophisticated and involves various factors such as selectivity of constraints, cardinality of result sets, and the use of indexes.

Before CBO became the dominant approach, database systems primarily relied on Rule-Based Optimization (RBO), where predefined heuristics determined the execution plan, rather than cost estimates derived from table statistics [2]. RBO prioritized specific access paths (such as indexed scans over full table scans) based on fixed rules rather than dynamically adjusting to data distribution. While CBO is now the standard in modern databases, RBO remains relevant in certain scenarios, such as when statistics are unavailable or outdated, or when database administrators need deterministic query execution plans. For example, in some legacy Oracle systems, RBO is manually enforced to ensure consistent performance when dealing with static workloads or predefined indexing strategies [2]. Understanding RBO is essential for performance tuning, as it highlights cases where cost-based decisions might lead to suboptimal execution paths due to incorrect cardinality estimates or unpredictable optimizer behavior.

CBO determines query execution plans by analyzing table statistics, index availability, and estimated row retrieval costs. It calculates the total execution cost based on factors such as I/O operations, CPU usage, and memory consumption.

In a study by Mehta et al. [3] was investigated how the order of processing JOIN operations and how they are written using ON and USING clauses affect the performance of SQL queries. Experimental comparisons showed that even small syntactic differences can affect the choice of execution plan and thus the overall query processing time. Leis et al. [4] demonstrated that cost-based optimizers often misestimate cardinality, leading to inefficient JOIN execution plans. Their findings highlight the optimizer's difficulty in accurately predicting result set sizes, particularly for complex queries. Cardinality estimates play a crucial role in choosing between different JOIN strategies, and their incorrectness can lead to

inadequate utilization of the available indices and a significant increase in query execution time.

Theoretical insight into the optimization of JOIN operations was provided by Atserias et al. [5], who addressed the problem of estimating the size of result sets when performing multiple JOIN operations. Their research showed that the size of the result set of a JOIN operation is closely related to the hypergraph models of the database schema and that proper selection of the order of JOIN operations can significantly reduce the computational cost [6]. They also addressed the maximum hypergraph density as a metric that can help predict the difficulty of executing complex SQL queries. These theoretical models are important in the design of efficient optimization strategies and can help in the development of new heuristic approaches for scheduling JOIN operations.

Indexing is another key aspect of optimizing SQL queries, especially when joining large datasets. Proper index selection can dramatically impact query performance, as it enables faster retrieval of relevant rows without the need to perform full table scan operations. As shown in the study by Atserias et al. [5], the use of efficient indexing strategies can reduce the number of scanned rows and minimize I/O operations. A study by Mehta et al. [3] pointed out that the use of semi-joins in specific cases can be more efficient than standard JOIN operations. In the case of FULL OUTER JOIN, indexing is less efficient since this operation requires processing entire tables, but the use of index scan techniques can at least minimize the negative impact on performance.

Research in SQL optimization shows that they influence the choice of execution plan in JOIN operations. The accuracy of the cost-based optimizer is a key factor, while inaccurate cardinality estimates can lead to inefficient plans and unnecessarily high system resource utilization [7]. The choice of the order of JOIN operations can be optimized using theoretical models such as hypergraph analysis and methods for predicting the size of result sets. Proper choice of indexing strategies can significantly speed up the processing of JOIN operations, but for some types of joins, such as FULL OUTER JOIN, additional techniques must be used to minimize the performance impact [8]. Although there is a large body of research on SQL optimization, there are still open questions regarding the efficient scheduling of JOIN operations in Oracle databases, especially for different indexing strategies. This work seeks to contribute a better understanding of these aspects through a detailed experimental analysis of the performance of INNER, LEFT, RIGHT and FULL OUTER JOIN in an Oracle environment.

III. Methodology and Experimental Design

A. Dataset description and table structure

For an experimental comparison of different SQL JOIN operations, we prepared a dataset in the Oracle 19c environment. We use a real-world dataset of traffic accidents in the Czech Republic to provide a more accurate picture of the experimental conditions, we selected three representative tables from the full dataset of 71 relational tables. The table CISI contains approximately 250,000 rows and stores accident records with a primary key on id and a foreign key on id_okres. The reference table OKRESY_TAB holds around 80 rows and is indexed on the id_okres column, while KRAJE_TAB stores about 15 rows and is indexed on id_kraj. The tables vary in number and types of attributes, including numeric, string, and date fields. We used default B-tree indexes on primary and foreign keys, but in selected experiments, we temporarily removed or added indexes to simulate real-world conditions with varying levels of optimization. This diversity in table size, attribute structure, and indexing allowed us to observe JOIN performance under realistic and scalable conditions.

Tables differ not only in name, but also in the number and type of columns. Some contain only a few attributes, while others may have dozens of columns, including numeric, string, or date types. In terms of the number of records, they cover a wide range - from tables with a few dozen rows to those that can contain thousands of records, allowing us to observe the impact of different dataset sizes on the performance of JOIN operations. Another important aspect is the indexes.

This diversity provides realistic conditions for exploring INNER, LEFT, RIGHT, and FULL OUTER JOIN, as well as different notations of join conditions (ON vs. USING). In the following subsections, we describe the specific queries constructed to measure performance, in which scenarios we the presence or absence of indices varied, and what type of metrics (execution time, CPU utilization, cost in the execution plan) tracked in our experiments. In this way, we will be able to evaluate to what extent the choice of a particular JOIN type and the syntax details of the notation are important factors in query optimization in a high-load, high-volume environment with large volumes of accident data from the Czech Republic.

B. Experimental setup

To ensure consistent benchmarking of SQL JOIN performance, all experiments were conducted in an Oracle Database 19c environment running on Windows 10 Home 64-bit. The hardware setup included an Intel Core i5-8300H processor (4 cores, 8 threads) with 16 GB of RAM. The dataset used in this study is publicly available at https://nehody.cdv.cz/.

C. Preparation of test queries

In designing the test queries, our goal was to obtain multiple scenarios in which we could observe the impact of changing the JOIN type (INNER, LEFT, RIGHT, and FULL OUTER) and the way the JOIN condition is formulated (ON vs. USING) on performance. We selected table pairs and triples where JOIN operations return sufficiently large non-zero result sets, ensuring realistic performance evaluation. Each query variant was tested with INNER, LEFT, RIGHT, and FULL OUTER JOIN, maintaining identical filtering conditions to isolate the impact of JOIN type. We then created a simple INNER JOIN query for each group of tables, specifying what columns and filters we would use in the WHERE clause. For performance comparison, we rewrote the same query into variants with LEFT, RIGHT, and FULL OUTER JOIN, keeping the filtering logic identical so that each version potentially returned the same result set. In this way, we were able to eliminate factors other than the difference in JOIN type.

To ensure the reliability and objectivity of our measurements, each query variant was executed ten times consecutively under the same conditions. The first execution was discarded to reduce the influence of cold cache effects, and the remaining nine measurements were averaged. This approach minimized the impact of transient background processes and system-level optimizations that could skew single-run results. Furthermore, before each new batch of experiments, we manually flushed the Oracle buffer cache and shared pool using the commands ALTER SYSTEM FLUSH BUFFER_CACHE and ALTER SYSTEM FLUSH SHARED_POOL to reset internal memory structures and ensure a fair baseline for each measurement.

Figure 1 shows the execution plan for an INNER JOIN query without indexes, where the optimizer defaults to a HASH JOIN strategy, leading to a full table scan.

```
SELECT c1.kodc1,
       c1.popiscl,
       c1.id_okres,
       o.nazov_okres
FROM cisi c1
JOIN okresy_tab o ON c1.id_okres = o.id_okres
WHERE c1.id_okres IS NOT NULL;
```

Fig. 1. INNER JOIN without Indexes

In the next phase, we extended the test queries with USING clauses if both joined tables contained a matching named column. In cases where the column names in the two tables differed, we stuck to the traditional form with the ON clause so as not to violate the basic join conditions. We followed the same procedure for queries with three or more tables, adding smaller tables to the from clause, to which we associated the main table TC_FORM_TAB. In these more complex queries, we also observed the extent to which the cost-based optimizer changes the order of joins or the preferred JOIN method (Hash, Nested Loop, Merge) as the number of tables processes increased.

Figure 2 presents the execution plan before introducing an index. The execution plan for an INNER JOIN query without indexes demonstrates how the optimizer selects the HASH JOIN strategy. Since no indexes are available, the optimizer performs a full table scan on both tables, leading to higher I/O operations and buffer usage.

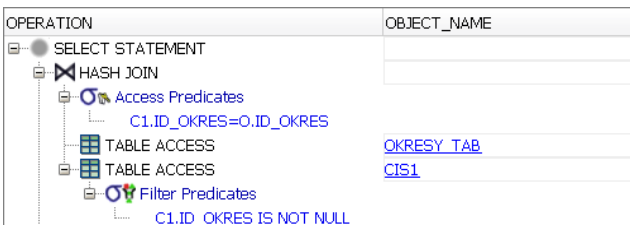

Fig. 2. Execution Plan for INNER JOIN without Indexes

During the preparation of the test queries, we varied the presence of indices in several variants too. First, we experimented with a situation where only basic indexes were created on the primary key of each table. Later, we temporarily removed some of these indexes or added new indexes specifically on foreign keys to simulate real database scenarios that may have varying degrees of optimization. The resulting query set thus included all four JOIN types in different indexing configurations, and we were able to record the extent to which these factors affect execution time, cost, CPU and I/O load, or other aspects of the execution plan. Thus, this range of test queries provided us with a comprehensive view of how individual changes to JOIN and join condition writes affect performance in a real Oracle database environment.

IV. Experimental Results and Analysis

This section evaluates how indexing influences SQL JOIN performance based on execution plans, execution time, and resource consumption. We compare optimizer decisions for INNER, LEFT, RIGHT, and FULL OUTER JOINs using different indexing strategies. Results are measured in terms of execution time, buffer usage, and CPU utilization to understand when indexing improves performance and when it introduces additional overhead.

A. Influence of Indexing on SQL JOIN Performance

The performance of SQL JOIN operations in Oracle databases varies significantly based on factors such as indexing, execution plan selection, table size, and query structure. Our experimental results show that the optimizer's choice of join strategy depends on the availability of indexes, the selectivity of the filtering conditions, and the number of records involved in the join operation. The primary join strategies observed include HASH JOIN, MERGE JOIN, and NESTED LOOPS, each with distinct performance characteristics.

After adding indexes, the optimizer switches to MERGE JOIN, taking advantage of sorted access paths. However, while this reduces full table scans, it may introduce sorting overhead, potentially increasing execution time depending on data distribution.

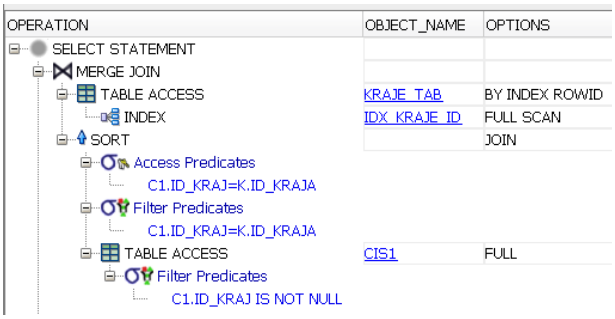

Fig. 3. Execution Plan for INNER JOIN with Indexes

This is particularly beneficial for large datasets, as hash joins allow efficient in-memory processing without requiring index lookups. In contrast, when indexes are present, the optimizer tends to favor MERGE JOIN or NESTED LOOPS, depending on the query structure. MERGE JOIN is often selected when both tables are sorted on the join key, whereas NESTED LOOPS is used when an indexed lookup can

efficiently retrieve matching rows. However, in cases where the dataset is large, indexed lookups can introduce additional overhead, leading to slightly higher execution times compared to hash joins.

In relational databases, a buffer refers to a memory structure used to store data blocks retrieved from the disk. The number of buffers required for query execution indicates how efficiently the database accesses data. Higher buffer usage generally implies increased I/O operations, which can impact query performance. While indexing typically improves query efficiency, its effect on buffer usage varies depending on the execution plan chosen by the optimizer. The following figure illustrates how buffer consumption changes for INNER JOIN, FULL OUTER JOIN, and COUNT() aggregation when indexes are present.

This figure illustrates how indexing affects buffer usage in different JOIN operations and aggregation queries. Oracle measures buffer usage by counting the number of database blocks that a query accesses during execution. While aggregation queries like COUNT() benefit from indexing, reducing buffer reads significantly, JOIN queries exhibit mixed behavior depending on the chosen execution plan.

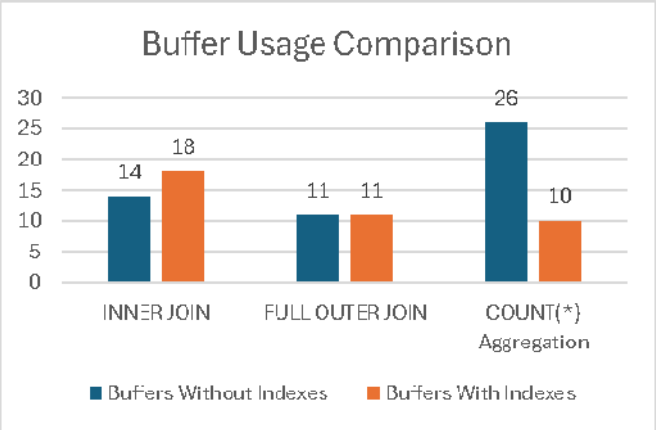

Fig. 4. Buffer usage comparison

Applying an index to the aggregation query COUNT(*) reduced buffer usage from 26 buffers to 10 buffers, a 61.54% improvement. This confirms that indexed access paths significantly reduce memory consumption for aggregate functions. However, for INNER JOIN, buffer usage increased from 14 to 18 buffers (+28.57% overhead) due to sorting operations introduced in MERGE JOIN. FULL OUTER JOIN showed no change in buffer usage (11 buffers in both cases), indicating that indexing has minimal impact when all records must be processed.. However, for INNER JOIN, the number of buffers slightly increases, suggesting that while indexing enables efficient lookups, it may also introduce additional processing steps, such as sorting for MERGE JOIN. Meanwhile, FULL OUTER JOIN exhibits minimal differences in buffer usage, reinforcing that indexing has limited benefits in scenarios where all records, including unmatched rows, must be retained. These findings suggest that indexing should be applied selectively, depending on the query type and workload characteristics. By analyzing buffer usage alongside execution time, database administrators can make more informed indexing decisions to optimize overall performance.

A detailed comparison of INNER JOIN performance highlights these differences. Without indexes, the optimizer consistently selects HASH JOIN, resulting in full table scans but allowing for efficient bulk processing. The recorded execution time for this approach was 0.365 seconds. When indexes were introduced, the optimizer switched to MERGE JOIN, utilizing an index scan on the joined column. Surprisingly, the execution time increased slightly to 0.375 seconds, indicating that the cost of sorting and merging indexed results did not always yield a performance gain. This suggests that for large tables with high cardinality, full table scans with hash joins can outperform indexed lookups, particularly when the query is designed to retrieve a significant portion of the data.

B. LEFT and RIGHT JOI and tndexing trade-offs

LEFT JOIN operations exhibit a similar trend. Without indexes, Oracle relies on HASH JOIN RIGHT OUTER, which ensures that unmatched rows from the left table are preserved. The observed execution time for this approach was 0.362 seconds, with full table scans on both participating tables.

After introducing an index on the join key, the optimizer shifted to MERGE JOIN, but the performance did not improve significantly, with execution times averaging around 0.372 seconds. This is because LEFT JOINs inherently involve additional processing overhead to retain non-matching records, which can offset the benefits of indexed lookups. The best performance gains from indexing in LEFT JOIN scenarios occur when the right-side table is large and highly selective queries are used to filter results.

For RIGHT JOIN operations, the observed behavior closely mirrors that of LEFT JOINs, with hash joins being the preferred strategy when no indexes are present. However, in practical applications, RIGHT JOIN is often replaced by LEFT JOIN with reversed table order, as many SQL developers prefer this structure for readability and consistency.

FULL OUTER JOIN remains the most expensive operation due to its requirement to retain unmatched rows from both tables, leading to full table scans and large hash table operations even with indexing. Since FULL OUTER JOIN must retain unmatched records from both tables, it often results in full table scans and large hash table operations. Even when indexes were available, the optimizer's execution plan remained inefficient due to the necessity of retrieving and combining all records.

C. Aggregation queries and indexing

Aggregation queries further highlight the impact of indexing on performance. Table 1 presents the execution time changes for different JOIN types when indexing is applied. Table 1 shows that INNER JOIN, LEFT JOIN, and RIGHT JOIN benefit significantly from indexing, achieving execution time reductions of over 47%, while FULL OUTER JOIN remains unaffected. This method efficiently groups records in memory while processing the join in a single pass. However,

after introducing indexes, the optimizer opted for MERGE JOIN instead of HASH JOIN, improving performance due to reduced full table scans. The impact of indexing is especially noticeable in aggregation queries like AVG and COUNT, where indexed lookups reduce execution time by up to 52.4%.

As shown in Table I, the introduction of indexes generally results in significant performance gains for most JOIN operations, except for FULL OUTER JOIN, where indexing does not provide any measurable benefit. Index scans improve access to individual rows and significantly reduce execution time for INNER, LEFT, and RIGHT JOINs. Additionally, indexing enhances aggregation queries like AVG and COUNT, reducing execution time by over 40%. These findings highlight the importance of selecting the appropriate execution plan and indexing strategy based on query type and workload characteristics.

TABLE I. EXECUTION TIME FOR JOIN OPERATIONS

JOIN Type	Execution Time without Indexes (s)	Execution Time with Indexes (s)	Performance Change
INNER JOIN	0.021	0.010	-52.4%
LEFT JOIN	0.022	0.011	-50.0%
RIGHT JOIN	0.023	0.012	-47.8%
FULL OUTER JOIN	0.030	0.030	No Change
AVG	0.021	0.010	-52.4%
COUNT	0.032	0.019	-40.6%

Table II highlights the impact of indexing on execution plan selection, where MERGE JOIN is typically used with indexes, but FULL OUTER JOIN consistently incurs high cost.

TABLE II. COMPARISON OF EXECUTION PLANS FOR JOIN TYPES

JOIN Type	Without Indexes (Plan Used)	With Indexes (Plan Used)
INNER JOIN	HASH JOIN (Full Table Scan)	MERGE JOIN (Index Scan)
LEFT JOIN	HASH JOIN RIGHT OUTER	MERGE JOIN
RIGHT JOIN	HASH JOIN RIGHT OUTER	MERGE JOIN
FULL OUTER JOIN	HASH JOIN (High Resource Cost)	HASH JOIN (Minimal Change)

A similar trend was observed in AVG(kodc1) calculations, where indexed queries were slightly slower than their full scan counterparts. The AVG(kodc1) function calculates the average value of the column 'kodc1', which represents accident code. In this case, the optimizer's decision to use MERGE JOIN with indexed lookups introduced unnecessary complexity, making the indexed query less efficient. This highlights an important consideration: indexes do not always improve query performance, particularly when aggregation functions require scanning a sizable portion of the table. Instead, hash joins with full scans can provide better performance in such scenarios.

The choice between ON and USING clauses had a negligible impact on execution performance. While USING simplifies query syntax when column names match both tables, it does not influence the optimizer's execution plan in a meaningful way. The optimizer still evaluates the same set of conditions for join selection, meaning that query performance remains identical whether ON or USING is used.

D. Practical recommendations for optimizing SQL queries

Our experimental findings lead to several practical recommendations for optimizing SQL JOIN performance in Oracle databases. INNER JOIN works best with indexes when filters are highly selective, but for large datasets with low selectivity, avoiding indexes and relying on HASH JOIN often yields better performance. LEFT JOIN benefits from indexing the right-hand table, especially when filtering is applied, while indexing the left table has limited effect. FULL OUTER JOIN remains the most resource-intensive join type and should be avoided whenever possible—preferably replaced with a combination of LEFT JOIN, RIGHT JOIN, and UNION operations. Aggregation functions such as COUNT and AVG show significant gains when using indexes on filtered columns, but may also introduce buffer overhead due to sorting. Finally, using ON or USING clauses has no impact on execution plans and should be chosen based on code readability rather than performance.

V. DISCUSSION

The experimental results confirm that the choice of JOIN type and indexing strategy significantly influences query performance in Oracle databases.

LEFT JOIN performs better when the right-side table is large and selective filtering is applied. RIGHT JOIN often mirrors the behavior of LEFT JOIN and can be rewritten in reverse order to improve readability without affecting performance. FULL OUTER JOIN, being computationally expensive, should be avoided whenever possible, particularly when an equivalent result can be achieved using a combination of LEFT JOIN, RIGHT JOIN, and UNION operations.Future work should investigate the impact of semi-joins and anti-joins on performance, as well as the effects of parallel execution plans in distributed database environments.

Although the experiments in this study were conducted using Oracle 19c, the observed performance trends can be partially generalized to other relational database management systems. Both PostgreSQL and MySQL use cost-based optimizers that evaluate execution plans based on table statistics and indexing, similar to Oracle. However, internal implementation details, such as index types, planner heuristics, and join algorithm preferences, can result in different behavior. For example, PostgreSQL supports additional join methods like parallel hash join and adaptive join strategies, while MySQL may prefer nested loop joins in simpler queries even when indexes are present.

In PostgreSQL, the query planner often relies heavily on up-to-date statistics and may be more sensitive to data distribution. This could affect JOIN type selection, especially for skewed datasets. On the other hand, MySQL's optimizer is generally simpler and may not perform as aggressively with regard to join reordering or indexing strategies. Unlike Oracle, which tends to favor hash joins for larger datasets without indexes, PostgreSQL may favor merge joins if sorted data or bitmap indexes are available.

While a direct experimental comparison is outside the scope of this paper, future work may involve replicating the current setup on open-source platforms to measure the differences empirically. Such a comparison would provide a more comprehensive understanding of JOIN performance across systems and help validate the generalizability of the results.

VI. Conclusion

The optimizer selects different execution plans based on table size, available indexes, and filtering conditions, leading to variations in execution time, CPU usage, and buffer consumption. Without indexes, the optimizer selects HASH JOIN, which allows efficient memory processing but requires full table scans. With indexes, the optimizer shifts towards MERGE JOIN or NESTED LOOPS, depending on the query structure. In the case of INNER JOIN, indexing led to the selection of MERGE JOIN, but execution time slightly increased due to additional sorting overhead. For FULL OUTER JOIN, indexing provided minimal improvement, as this join type inherently requires processing entire tables, limiting the optimizer's ability to leverage indexes.

One of the most notable findings concerns buffer usage. In contrast, INNER JOIN with indexes exhibited a slight increase in buffer usage, suggesting that while indexes optimize lookups, they may introduce additional sorting steps. FULL OUTER JOIN showed negligible differences in buffer consumption, indicating that indexing is less effective for operations that must retain unmatched records from both tables. These results suggest that indexing should be applied selectively, as its impact varies depending on query structure and data distribution.

The results also highlight the trade-offs associated with different JOIN types. INNER JOIN is most effective with indexes when applied to datasets with high selectivity. LEFT JOIN performs better when the right-side table is large and selective filtering is applied. RIGHT JOIN often mirrors the behavior of LEFT JOIN and can be rewritten in reverse order to improve readability without affecting performance.

The most effective optimization strategies depend on dataset size, the availability of indexes, and query complexity. Large tables benefit from HASH JOIN due to its ability to process large datasets in memory, whereas smaller tables are better suited for MERGE JOIN or NESTED LOOPS. Aggregation queries can suffer from increased buffer reads when executed with indexed joins.

While our study provides insights into JOIN performance in Oracle databases, it is important to note some limitations. First, our experiments were conducted within Oracle 19c; optimizations may differ in other database management systems such as MySQL or PostgreSQL. Second, the dataset used consists of 71 tables related to transport data, and performance may vary in databases with different indexing structures or data distributions. Finally, we focused on INNER, LEFT, RIGHT, and FULL OUTER JOIN without analyzing the impact of semi-joins or anti-joins, which may present alternative optimization strategies.

Acknowledgment

This paper was supported by the VEGA 1/0192/24 project - Developing and applying advanced techniques for efficient processing of large-scale data in the intelligent transport systems environment.

References

[1] Oracle Documentation, Cost-Based Optimizer, Web: https://docs.oracle.com/cd/E98457_01/opera_5_6_core_help/cost_based_optimizer.htm.

[2] J. Lewis, *Cost-Based Oracle Fundamentals*, 1st ed. Berkeley, CA: Apress, 2006.

[3] S. Mehta, P. Kaur, P. Lodhi, and O. Mishra, “Empirical Evidence of Heuristic and Cost Based Query Optimizations in Relational Databases,” in *Proc. 11th Int. Conf. Contemporary Computing (IC3)*, Noida, India, 2018, pp. 1–3.

[4] B. Wagner, A. Kohn, P. Boncz, and V. Leis, “Incremental Fusion: Unifying Compiled and Vectorized Query Execution,” in *Proc. 40th IEEE Int. Conf. Data Eng. (ICDE)*, Utrecht, Netherlands, 2024, pp. 462–474.

[5] A. Atserias, M. Grohe, and D. Marx, “Size Bounds and Query Plans for Relational Joins,” in *Proc. 49th Annu. IEEE Symp. Foundations Comput. Sci. (FOCS)*, Philadelphia, PA, USA, 2008, pp. 739–748.

[6] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica, “Learning to Optimize Join Queries With Deep Reinforcement Learning,” *arXiv preprint arXiv:1808.03196*, 2019.

[7] T. Neumann, V. Leis, and A. Kemper, “The Complete Story of Joins (in HyPer),” in *Proc. Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, Bonn, Germany, 2017, pp. 31–50.

[8] C. Ordonez and J. García-García, “Evaluating Join Performance on Relational Database Systems,” *J. Comput. Sci. Eng.*, vol. 4, no. 4, pp. 276–290, Dec. 2010, doi: 10.5626/JCSE.2010.4.4.276.