Electronics and Information Technologies, 2026, 33, 97–112
http://publications.lnu.edu.ua/collections/index.php/electronics/index

UDC: 004.652.4

INTELLIGENT ANALYSIS OF PERFORMANCE RESULTS BASED ON OBJECT-RELATIONAL MAPPING STRATEGIES AND FOREIGN KEY CONSTRAINTS IN SQL DATABASES

Oleksandra Ryback , ***Oleh Husak*** , ***Roman Mysiuk*** *
Department of System Design
Ivan Franko National University of Lviv,
50 Drahomanova St., 79005 Lviv, Ukraine

Rybak, O., Husak, O., & Mysiuk, R. (2026). Intelligent Analysis of Performance Results Based on Object-Relational Mapping Strategies and Foreign Key Constraints in SQL Databases. *Electronics and Information Technologies*, 33, 97–112. https://doi.org/10.30970/eli.33.8

ABSTRACT

Background. The rapid expansion of data-driven applications has increased the importance of efficient query execution in relational database systems, where even minor inefficiencies can significantly affect overall performance. Although Object-Relational Mapping (ORM) frameworks simplify development and improve maintainability, their abstraction layer can introduce measurable overhead, and the impact of foreign key constraints on execution speed remains a practical concern, particularly in microservice architectures that follow the “Database per Service” principle.

Materials and Methods. An experimental information system is developed using a relational database and the SQLAlchemy ORM framework, with a schema that includes one-to-one, one-to-many, and many-to-many relationships tested both with and without foreign key constraints. Three representative queries retrieving booking details, aggregating related records, and calculating total payments are executed using raw SQL and ORM approaches, while an intelligent algorithm analyzed performance, detected potential *N*+1 query risks, and recommended optimal strategies such as explicit JOINs.

Results and Discussion. Raw SQL consistently demonstrated superior performance across all scenarios. The most significant disparity occurred in ORM implementations affected by the *N*+1 problem, where execution time exceeded that of equivalent SQL queries by more than an order of magnitude. Aggregation queries showed smaller yet consistent overhead. The presence or absence of foreign key constraints had a negligible influence on raw SQL performance, with differences remaining within experimental variance. Explicit JOIN usage in ORM substantially reduced overhead compared to implicit relationship navigation. The intelligent analysis accurately predicted high-risk queries and provided effective strategy recommendations, confirmed by empirical results.

Conclusion. ORM frameworks improve productivity and maintainability but introduce measurable overhead. Raw SQL remains preferable for performance-critical tasks, while foreign key constraints do not significantly degrade execution speed. Intelligent performance analysis supports balanced decisions between efficiency and maintainability in complex relational systems.

Keywords: relational databases, SQL performance, ORM, decision support system, intelligent analysis, database design

INTRODUCTION

Relational databases remain a fundamental component of modern information systems due to their strong consistency guarantees and well-defined data relationships. As applications scale, query performance becomes a critical factor, particularly in systems with complex inter-table relationships. Developers frequently rely on Object-Relational Mapping (ORM) frameworks to simplify database access, improve code maintainability, and reduce development time. However, ORMs introduce additional abstraction layers that may negatively affect query performance. A well-known issue in ORM-based systems is the *N*+1 query problem, which can lead to significant inefficiencies when navigating relationships. The essence of this problem is that when performing an initial database query that returns *N* objects, another query may automatically be executed for each of these objects, resulting in *N* additional queries. This creates a significant performance penalty, especially in applications that use ORMs, since iterating over a collection of objects in object-oriented language code naturally generates additional database queries. One common method is the use of eager loading or batch fetching, when the required data is loaded using a single combined or batch query, reducing the number of database calls. Also in practice, SQL query optimization and code refactoring are used to avoid iterations over collections with separate calls to ORM methods. For example, in article [1] an automated approach to refactoring was proposed that uses static analysis to detect and eliminate *N*+1 cases, transforming many small queries into one efficient one, which significantly improves performance.

At the same time, modern software architectures such as microservices promote the principle of Database per Service, emphasizing isolated and independently managed databases. This raises practical questions regarding the necessity and performance impact of foreign key constraints and ORM-managed relationships in high-performance systems. The goal of this study is to experimentally evaluate how different raw SQL and ORM query approaches behave in terms of performance under varying relationship configurations, and to quantify the overhead introduced by ORM abstractions.

This study aims to conduct a quantitative performance evaluation of ORM-based queries and raw SQL in relational database systems with inter-table relationships, focusing on execution time, resource usage, and the impact of ORM-managed relationships.

A quantitative performance evaluation comparing Prisma ORM and raw SQL demonstrated that raw SQL consistently outperformed Prisma across all query types. The composite performance index showed that raw SQL is approximately 2.26 times more efficient, particularly in nested and bulk operations where ORM abstraction introduced measurable latency and higher resource consumption [2].

The Cosmos-specific ODM outperformed raw SQL and generalized ORM-style approaches across all evaluated metrics, including task completion time, error frequency, and perceived cognitive load [3].

Comparative analyses of ORM tools have demonstrated measurable differences in execution performance, memory usage, and query optimization across platforms and frameworks [4, 5]. A comparative analysis of Entity Framework Core, Dapper, and LINQ to DB indicated that Dapper provided the best performance and the lowest memory consumption among the evaluated systems [6, 7]. Across all three experiments in [8], SQL consistently achieved faster execution times than ORM while maintaining nearly identical memory usage. Although ORM occasionally exhibited longer maximum execution times, SQL proved to be the more efficient solution for performance optimization. A query repository with automatic SQL query classification is presented to enforce data-privacy directives [9] and implemented optimizations to reduce the performance overhead caused by intercepting queries through a JDBC proxy, achieving classification latencies as low as 0.35 ms while still using graph-based metadata for cases requiring higher precision.

ORM tools can simplify development but may negatively affect relational query performance, producing inefficient query patterns and necessitating further investigation into mitigation strategies [10].

Summarizing this analysis of sources, it is worth noting that [2-10] largely evaluates ORM performance in isolation or with simple metrics, without considering the combined impact of foreign key constraints, relationship-heavy queries, and modern architectures. This work fills this gap by providing an intelligent, quantitative analysis of ORM strategies alongside FK effects for practical performance optimization.

MATERIALS AND METHODS

The experimental evaluation is conducted using a relational database designed to model real-world inter-table relationships, including one-to-one, one-to-many, and many-to-many associations. Two database configurations are implemented: one with enforced foreign key constraints and one without foreign keys, enabling the assessment of their impact on query performance.

Query execution is implemented using two approaches: raw SQL queries and ORM-based queries developed with SQLAlchemy. For the ORM approach, both lazy loading (default relationship navigation) and explicit join strategies are applied. All SQL queries are manually optimized and executed using the same database engine to ensure consistency. Each experiment included three representative query types: a complex multi-table join, a count aggregation query, and a sum aggregation query.

Performance measurements are obtained by executing each query multiple times under identical conditions and recording execution time at the application level. The mean execution time and standard deviation are calculated for each query and configuration. Additional scalability experiments are performed by increasing the dataset size to evaluate the impact of data volume on query performance. All experiments are executed on the same hardware and software environment to eliminate external variability.

Software and environment

The experimental study is conducted using Python 3.13.0 and MySQL Server 8.0+. Database interactions are implemented via SQLAlchemy as the ORM layer and PyMySQL as the database driver. Data processing and statistical analysis are performed using Pandas, while visualization is carried out with Matplotlib. Synthetic test data are generated using the Faker library. All experiments are executed on a local machine equipped with an 11th Gen Intel® Core™ i3-1115G4 (3.00 GHz) processor, 8 GB RAM, and a 238 GB SSD, running Windows 11 (64-bit, x64 architecture). The database server is deployed locally to ensure consistent benchmarking conditions.

Database design

A relational database schema representing a transportation booking system is designed. The dataset consisted of synthetic but structurally realistic records representing clients, bookings, trips, payments, vehicles, and related entities. The scheme supported 1:1, 1:*N*, and *M*:*N* relationships. Two database configurations are created with and without foreign key constraints.

As part of the experiment, the trips_db database (**Fig. 1**) is filled with a controlled set of test data created using SQLAlchemy's ORM by completely cleaning the tables and adding a fixed number of records (2 for each entity), which allows you to focus on analyzing the impact of ORM abstractions and cross-table relationships on the performance of nested and multi-table queries regardless of the amount of data.

The tables in the studied database are indexed. Each table contains a primary key attribute (e.g., *booking_id*, *trip_id*, *client_id*, *vehicle_id*, *driver_id*), which is automatically indexed by the database management system. These attributes are used in foreign key relationships between tables such as bookings, routes, payments, and trips, which enables

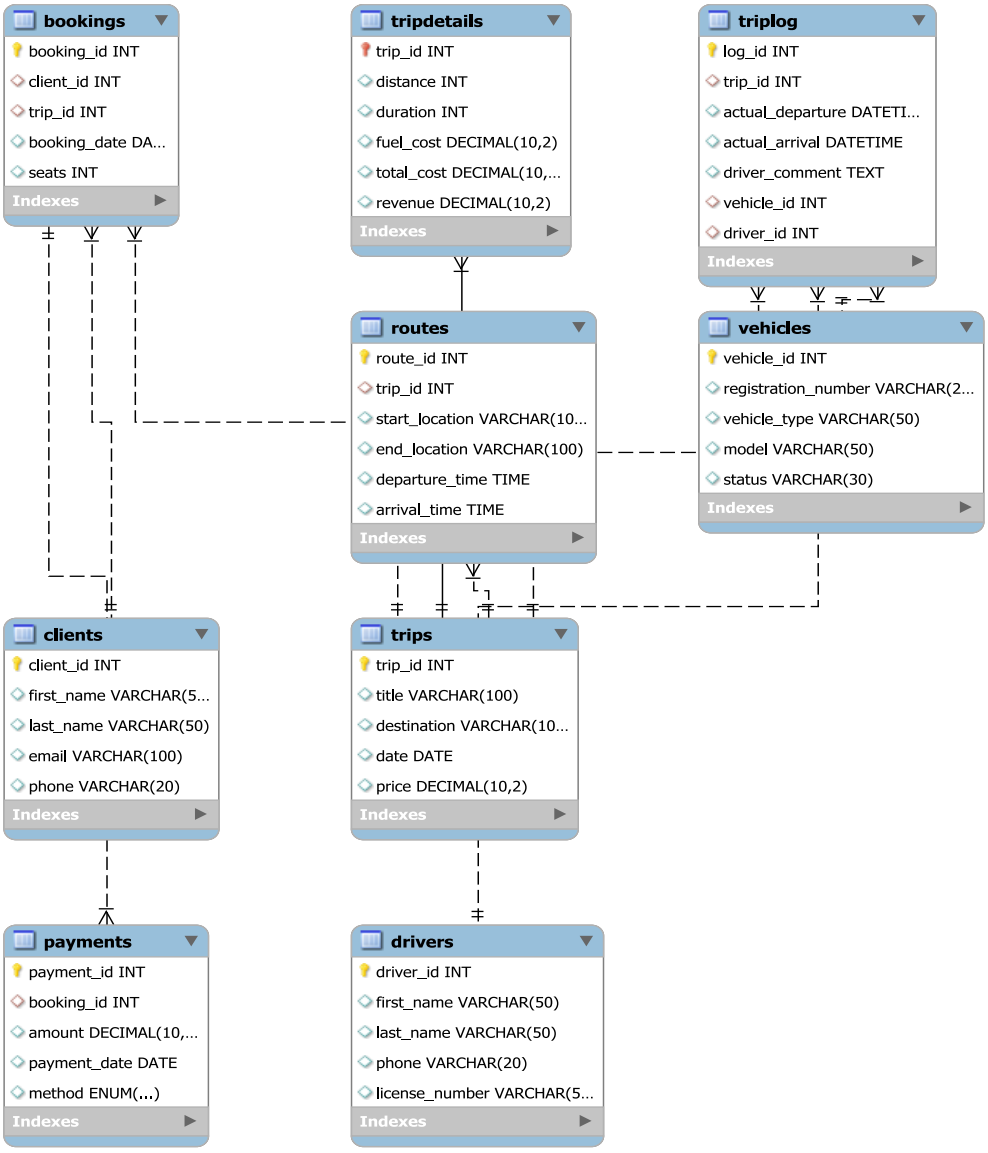


Fig. 1. Entity-relationship diagram of the *trips_db* database.

efficient join operations and improves query performance. The presence of indexes in the tables of the database under study has a critical impact on performance, since in the *N*+1 query problem, the absence of indexes leads to numerous full table scans.

The database is populated with a minimal yet structurally complete synthetic dataset covering all entities and relationships required for controlled performance evaluation. Data covers all 9 tables and all types of relationships (1:1, 1:*M*, *M*:*N*); the scenario is deterministic (same results on each run); the goal is to compare query approaches, not stress-test the database.

The Triplog entity acts as a central point for recording trip performance, unifying all key resources, with the Trips, Drivers, and Vehicles tables linked to Triplog in a one-to-many (1:*M*) relationship.

Test scenarios

Three representative queries are selected:

- Retrieval of booking and trip execution details for a specific client (multi-table join).
- Counting the number of bookings per trip (aggregation with grouping).
- Calculation of the total sum of payments (aggregation).

Each query is executed using four approaches ORM with foreign keys (automatic relationship navigation), raw SQL with foreign keys, ORM without foreign keys (explicit JOINs).

All tests are executed multiple times on the same dataset, and the recorded times are averaged to minimize measurement noise. Query execution time is measured at the application level by recording timestamps immediately before and after query execution using the datetime library.

In the implementation [11], Data Definition Language (DDL) elements are represented through ORM models (Client, Trip, Booking, Triplog, Driver, Vehicle, Payment), where the declarative approach is used to define database tables, primary and foreign keys, attribute data types, and inter-table relationships. Overall, the core functionality belongs to Data Manipulation Language (DML), as it focuses on data retrieval and aggregation through SELECT, JOIN, GROUP BY, and SUM operations implemented both in pure SQL and through ORM abstractions (count, sum).

Intelligent query strategy decision

The rules that are implemented based on the ORM slowdown factor, defined as: $S = T_{ORM}/T_{SQL}$, where T_{ORM} and T_{SQL} denote the mean execution times of the ORM-based and raw SQL queries, respectively (**Fig. 2**). This metric quantifies the relative performance degradation introduced by ORM abstractions to assess the relative inefficiency of different data access strategies.

Based on the measured slowdown factors, queries are assigned to one of three distinct risk levels to systematically evaluate the performance impact of ORM strategies. Queries with a slowdown factor $S > 10$ are classified as Critical Inefficiency (N+1 Risk),

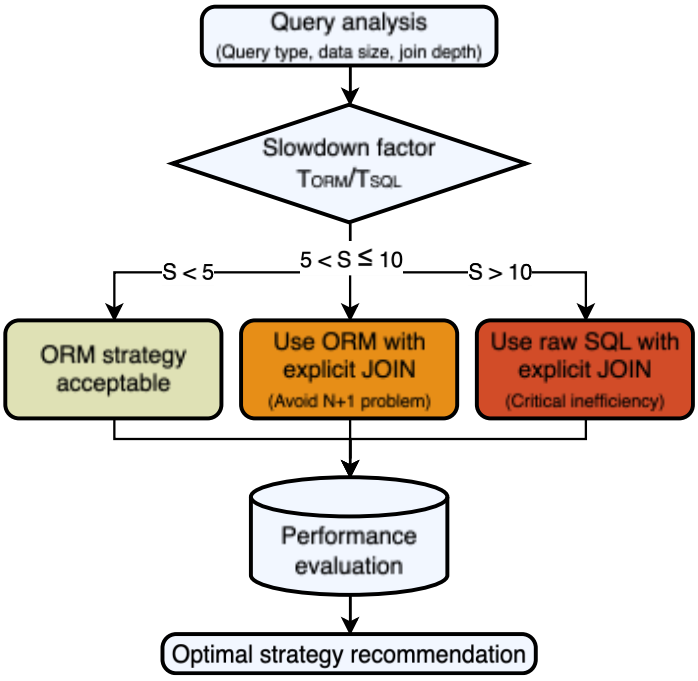


Fig. 2. Intelligent query strategy decision framework.

indicating that lazy-loading ORM patterns (*N*+1 queries) substantially increase execution time and are prone to severe performance degradation, particularly in large-scale datasets. Queries with $5 < S \leq 10$ are classified as Moderate Overhead, representing cases where ORM abstractions introduce noticeable performance loss, which can often be mitigated through explicit JOINs or eager-loading strategies. Finally, queries with $S \leq 5$ are considered Acceptable Performance, indicating low-risk scenarios where ORM overhead is minimal.

A decision tree is mathematically designed to model exactly this kind of conditional branching and provides robustness on small datasets, minimal hyperparameter tuning, and the ability to model categorical outputs directly. It also operates independently of feature normalization and naturally supports future feature expansion, such as join depth, table count, row volume, index availability, and aggregation type. While it provides interpretable, threshold-based rule extraction with minimal preprocessing, decision trees may overfit and become unstable on extremely small datasets, and alternative models like linear regression, logistic regression, neural networks, and SVMs were avoided due to limited interpretability or incompatibility with threshold-based decision logic. From a technical perspective, we use *DecisionTreeClassifier*, which is trained on the experimental dataset. The model automatically learned threshold-based rules for strategy selection. The trained tree is exported as a vector-based SVG diagram using *Graphviz* for interpretability and reproducibility. This approach transforms deterministic rule logic into a data-driven adaptive optimization model. By providing a structured and interpretable metric, the risk classification framework serves as a reliable tool for both benchmarking and predictive performance analysis of relational database queries.

Statistical Hypothesis Testing

To validate the observed differences in query execution times and assess their statistical significance [12, 13], paired t-tests and one-way ANOVA are conducted. The paired t-test is used to compare the mean execution time of ORM queries to the corresponding SQL queries under identical conditions, with the null hypothesis stating that no significant difference exists between ORM and SQL mean execution times. Execution times are recorded over multiple repeated runs for each query type, and paired t-tests are applied using the function *ttest_rel*( ) from SciPy library of Python. The one-way ANOVA is applied to evaluate whether mean execution times differed across multiple query execution strategies, including ORM with foreign keys, SQL with foreign keys, ORM with explicit JOINs, and SQL without foreign keys. In this case, the null hypothesis posited that all group means are equal, and the test is implemented using the function *f_oneway*( ) from SciPy library of Python.

These statistical tests provide rigorous validation of the performance measurements. While raw execution times suggest that ORM may incur higher overhead, the t-test and ANOVA determine whether these differences are statistically meaningful or likely attributable to random variability in repeated measurements. The inclusion of hypothesis testing therefore, enhances the reliability of performance conclusions and informs the adaptive strategy recommendation framework.

RESULTS AND DISCUSSION

The performance of ORM-based queries and raw SQL is compared. The results are showing that SQL consistently executes faster, especially for complex multi-table joins where ORM lazy loading triggers the *N*+1 problem. Scalability analysis demonstrates that ORM execution time grows more rapidly with increasing dataset sizes, while SQL scales more linearly, highlighting potential bottlenecks in high-load scenarios. Intelligent strategy decisions classify queries by risk level, recommending raw SQL for critical inefficiencies and ORM with explicit joins for moderate overhead, providing actionable guidance for developers. Statistical hypothesis testing, including paired t-tests and ANOVA, confirms that the observed performance differences are significant and not due to random variation.

Overall, the section integrates empirical measurements, predictive modeling, and intelligent analysis to support evidence-based optimization of query execution strategies.

Comparison of ORM and SQL performance

A comparative experiment is conducted to assess query efficiency using four data access approaches:

- Pure SQL with explicit JOINs and foreign key constraints.
- ORM with relationship-based navigation (automatic joins via SQLAlchemy).
- SQL without foreign key constraints (manual joins).
- ORM without relationships (explicit joins defined in code).

The results in **Table 1** indicate a consistent performance advantage of pure SQL over ORM-based implementations when foreign keys are present.

Table 1. **Comparison of query performance using Foreign Keys**

#	Description	Results		
		ORM (with FK), ms	SQL (with FK), ms	SQL faster, times
Q1	Details of the trip (5 tables, N+1)	32.25	1.93	~16.7
Q2	Number of bookings per trip (JOIN, COUNT)	5.03	0.63	~8.0
Q3	Total Payments (SUM)	5.52	0.68	~8.1

The largest discrepancy is observed in the complex multi-table query (Q1), where ORM with relationship navigation exhibited the *N*+1 query pattern and required 32.25 ms compared to 1.93 ms for SQL, making SQL approximately 16.7 times faster. For aggregation queries (Q2 and Q3), SQL remained about eight times faster than ORM. These findings confirm that although ORM improves abstraction and developer productivity, it may introduce significant performance overhead in complex relational queries, particularly when implicit loading strategies are used.

Table 2 presents the performance comparison of ORM and pure SQL queries executed without enforcing foreign key constraints. The results demonstrate that even in the absence of foreign keys, pure SQL maintains a consistent performance advantage across all query types. For the complex multi-table retrieval (Q1), ORM with explicit JOIN requires 13.72 ms, whereas SQL completes the same operation in 4.00 ms, making SQL approximately 3.4 times faster. Similar trends are observed for aggregation queries: in Q2 (COUNT with JOIN), SQL is about 3.6 times faster, and in Q3 (SUM aggregation), the performance gap increases to approximately six times. Importantly, compared to the earlier *N*+1 ORM implementation, the explicit use of JOIN within ORM significantly reduces

Table 2. **Comparison of query performance without Foreign Keys**

#	Description	Results		
		ORM (without FK, with JOIN), ms	SQL (without FK), ms	SQL faster, times
Q1	Details of the trip (5 tables, N+1)	13.72	4.00	~3.4
Q2	Number of bookings per trip (JOIN, COUNT)	5.78	1.59	~3.6
Q3	Total Payments (SUM)	5.31	0.89	~6.0

execution time, confirming that query structure has a critical impact on performance. Nevertheless, SQL remains more efficient due to lower abstraction overhead.

Table 3 provides a consolidated comparison of all four experimental scenarios, enabling a holistic evaluation of abstraction level and foreign key enforcement effects. Scenario I (ORM with foreign keys and relationship navigation) shows the highest execution times, particularly for Q1, reflecting the impact of the *N*+1 query pattern. Scenario II (SQL with foreign keys) demonstrates the best overall performance. Scenario III (ORM with explicit joins and without foreign keys) substantially improves over Scenario I, confirming that manual join specification mitigates inefficiencies caused by implicit loading. Scenario IV (SQL without foreign keys) remains faster than both ORM-based approaches, though slightly slower than SQL with foreign keys. Overall, the table confirms two central findings: explicit JOIN usage in ORM significantly enhances efficiency compared to *N*+1 navigation, and pure SQL consistently delivers superior performance across all tested conditions.

Table 3. **Consolidated comparison of all approaches**

Scenario	Description	Results		
		Q1 (JOIN), ms	Q2 (COUNT), ms	Q3 (SUM), ms
I	ORM (with FK)	32.25	5.03	5.52
II	SQL (with FK)	1.93	0.63	0.68
III	ORM (explicit JOIN) without FK	13.72	5.78	5.31
IV	SQL without FK	4.00	1.59	0.89

This confirms that replacing implicit navigation with explicit joins significantly enhances ORM efficiency. However, even after optimization, ORM remains slower than pure SQL in all scenarios. For example, in Q1 without foreign keys, ehe ORM runs approximately 3.4 times slower than raw SQL. Similar gaps persist in Q2 and Q3.

The substantiates that while explicit join usage in ORM enhances performance, pure SQL consistently outperforms ORM across all measured scenarios, both in terms of execution time and stability (**Fig. 3**).

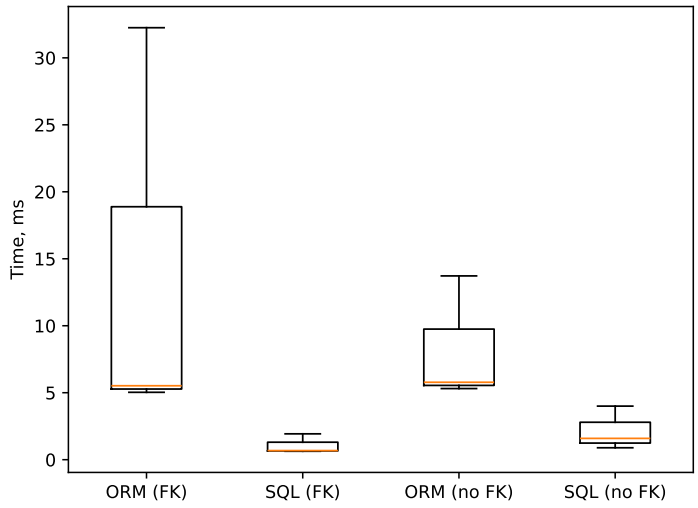


Fig. 3. Distribution of Query Execution Time for ORM and SQL Approaches.

The ORM *N*+1 queries can severely degrade performance on complex queries with many related records (Q1), while explicit JOINs are faster, though for simple queries (Q2, Q3) both strategies perform similarly (**Fig. 4**).

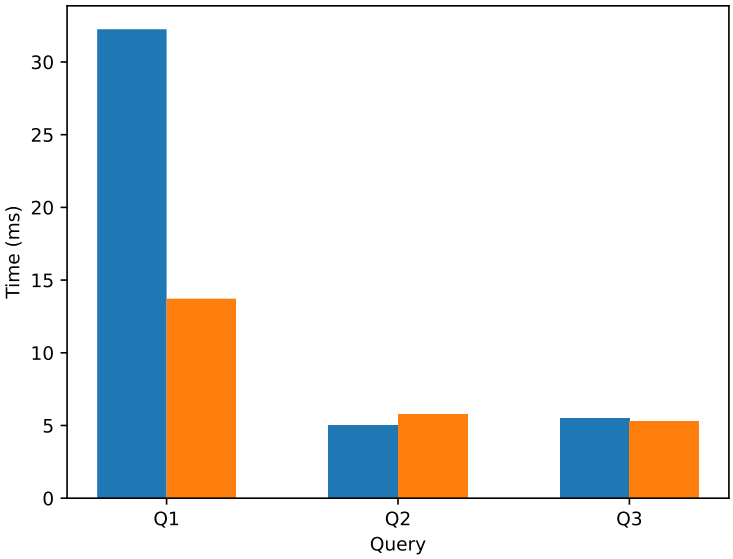


Fig. 4. ORM Strategy Comparison.

Table 4 shows that using foreign keys consistently improves SQL query performance, and while ORM JOINs mitigate N+1 overhead, raw SQL remains faster.

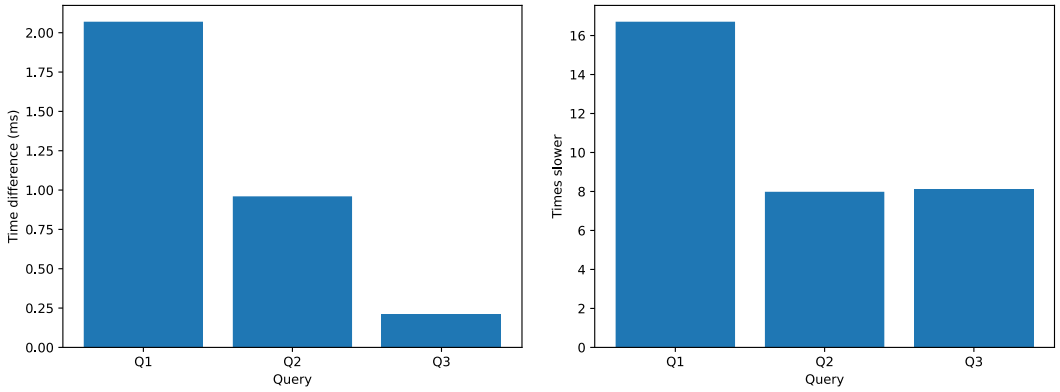
The experiments demonstrate that while explicit JOINs in ORM substantially reduce *N*+1 query overhead, pure SQL (with foreign keys) consistently delivers superior performance across all query types and complexities.

A quantitative analysis of query performance under different database access strategies executed with highlighting the influence of foreign key enforcement and ORM abstraction (**Fig. 5**).

The presence of foreign key constraints substantially improves SQL execution efficiency, with the most pronounced reduction observed in the complex multi-table query Q1, whereas simpler aggregation queries Q2 and Q3 benefit to a lesser extent (**Fig. 5a**). The performance drawback (**Fig. 5b**) associated with ORM-based access relative to pure SQL, ORM with foreign keys exhibits a pronounced slowdown for Q1, attributable to the *N*+1 query pattern, while the overhead for Q2 and Q3 is comparatively minor. These results underscore two principal findings: first, foreign key enforcement materially enhances SQL query performance, particularly in queries involving multiple related tables; second, despite the use of relationship navigation, ORM introduces significant execution overhead in complex query scenarios, reinforcing the trade-off between abstraction convenience and raw performance.

Table 4. **Relative Impact of Foreign Keys on SQL Query Performance**

#	Results	
	SQL with FK, ms	SQL without FK, ms
Q1	1.93	4.00
Q2	0.63	1.59
Q3	0.68	0.89



(a) (b)

Fig. 5. Analysis of performance results: a) Impact of Foreign Keys on SQL Performance (difference between SQL with FK and SQL without FK), b) ORM Slowdown Factor (ORM with FK to SQL with FK).

The experiments demonstrate that raw SQL consistently outperforms ORM-based queries across all tested scenarios. The most significant performance gap occurs in complex multi-table joins (Q1) affected by the *N*+1 problem, where ORM execution time exceeds SQL by more than an order of magnitude. Aggregation queries (Q2 and Q3) show moderate ORM overhead, while the presence or absence of foreign key constraints has a negligible impact on SQL performance. Using explicit JOINs in ORM significantly reduces overhead compared to lazy relationship navigation. Furthermore, the intelligent algorithm accurately identifies high-risk queries, quantifies performance degradation, and recommends optimal execution strategies, validating its utility as a decision-support tool for database optimization.

Scalability analysis

The experiments are repeated to evaluate scalability with increasing dataset size. To analyze scalability, the scenario of increasing the number of records in the key tables of the trips_db database is considered, namely: table clients 1,100 records, trips 500 records, bookings 1,200 records, and payments 1,200 records. The results, shown in **Figure 3**, reveal fundamentally different growth patterns. ORM queries affected by the *N*+1 problem exhibit non-linear growth in execution time, with performance degradation becoming critical at larger scales. In contrast, raw SQL queries scale almost linearly with data size, maintaining low execution times even for larger datasets. Eager loading strategies show improved scalability compared to lazy loading, following a near-linear trend. However, their execution time remains consistently higher than that of raw SQL. These findings confirm that the choice of data access strategy has a direct impact on system scalability. **Table 5** presents the statistical analysis of repeated query executions.

The mean execution time, standard deviation, and coefficient of variation (CV) are reported for each query type [11]. The results indicate that ORM queries, particularly those affected by the *N*+1 problem, exhibit higher variability and longer execution times compared to equivalent SQL queries. The CV values highlight the relative dispersion of execution times, confirming that ORM *N*+1 queries are the least stable, while aggregation queries (SUM, COUNT) are more consistent. Q1 exhibits a critical inefficiency caused by the *N*+1 query problem, with a slowdown factor significantly exceeding the defined threshold. The recommended strategy is to use raw SQL with explicit JOINs to eliminate excessive query overhead. Q2 and Q3 show moderate ORM overhead, indicating that while ORM abstractions introduce some performance loss, it is not critical. For these queries, the system recommends using ORM with explicit JOINs to maintain ORM convenience while mitigating unnecessary query inflation.

Statistical Hypothesis Testing

Paired t-tests and one-way ANOVA are conducted to evaluate whether ORM and SQL execution times differed significantly. The paired t-test ($t = 0.993$, $p = 0.377$) indicates no significant difference between ORM and SQL for matched queries. Similarly, ANOVA across all tested approaches ($F = 1.666$, $p = 0.251$) shows that mean execution times do not differ significantly at the 5% significance level. These results suggest that, although ORM queries exhibit numerically higher execution times, the observed differences are not statistically significant for the sampled runs, highlighting the influence of measurement variability or limited sample size.

Table 5. **Relative Impact of Foreign Keys on SQL Query Performance**

Scenario	Results					Mean execution time, s	Standard Deviation, s	Coefficient of Variation
	1	2	3	4	5			
ORM Search (*N*+1)	0.382	0.002	0.002	0.001	0.002	0.078	0.1521	1.955
SQL Search (JOIN)	0.039	0.004	0.001	0.002	0.002	0.009	0.0147	1.535
ORM Grouping (Count)	0.028	0.005	0.009	0.006	0.009	0.011	0.0085	0.741
SQL Grouping (Count)	0.007	0.008	0.007	0.010	0.010	0.009	0.0014	0.161
ORM Aggregation (SUM)	0.008	0.001	0.001	0.001	0.001	0.002	0.0028	1.167
SQL Aggregation (SUM)	0.003	0.001	0.001	0.001	0.001	0.001	0.0008	0.571

Intelligent strategy decisions

The intelligent analysis (**Fig. 6**) illustrates the adaptive decision tree derived from experimental performance data, with the root node threshold Slowdown_ORM_vs_SQL_with_FK ≤ 12.414 separating queries into two branches. Queries below the threshold (Q2 and Q3) are classified as “Use ORM with explicit JOIN”, reflecting moderate overhead where ORM remains acceptable. Queries exceeding the threshold (Q1) fall into the “Use raw SQL with explicit JOIN” leaf, indicating severe ORM overhead and *N*+1 risk. Testing on the trips_db_no_fk database confirmed these predictions, with foreign key removal having minimal impact, demonstrating that the system provides a transparent, threshold-based decision-support tool for selecting query execution strategies.

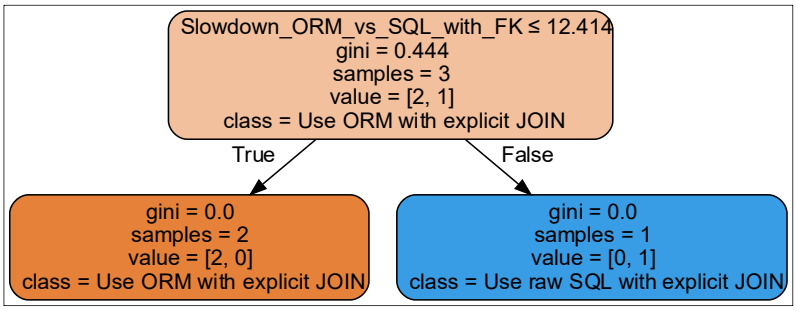


Fig. 6. Results based on queries.

Discussion of results

The quantitative analysis confirms that the primary source of performance degradation is not the presence of foreign key constraints but the ORM execution model itself. The increased execution time and variance observed in ORM-based approaches are caused by additional query generation, data fetching overhead, and object mapping operations. While eager loading significantly improves performance, raw SQL remains the most efficient and stable solution for complex and high-load query scenarios. From a practical perspective, the results suggest that ORM frameworks are suitable for CRUD (create, read, update, delete) operations and moderate workloads, while raw SQL is preferable for performance-critical components, particularly those involving complex joins and large datasets.

The adaptive decision tree effectively classifies queries based on the well-established ORM slowdown metric, with a threshold of 12.414 distinguishing moderate overhead (where ORM with explicit JOINs is acceptable) from severe overhead (favoring raw SQL to avoid *N*+1 issues). This provides developers with a transparent, reproducible decision-support tool to select query execution strategies, anticipate performance bottlenecks, and prevent inefficiencies. Validation on the trips_db_no_fk database confirmed the model's predictions, showing that foreign key removal had minimal impact and indicating robustness, although generalizability may be limited across different databases, ORMs, or query patterns. Future work could expand the approach to additional query types, incorporate runtime profiling for real-time decisions, and integrate with automated query optimization to enhance practical applicability.

Although the use of ORM greatly simplifies the process of developing and maintaining program code, this is achieved at the cost of reducing performance. At the same time, the presence of intertable relationships in the database ensures data integrity and, according to the results of the experiment, does not lead to a noticeable deterioration in the performance of raw SQL queries.

Based on [14–17] and our experimental results, we prepared a heatmap to demonstrate the feasibility of using ORM and direct SQL for different types of queries (**Fig. 7**). The heatmap shows query types on the Y-axis which include CRUD operations, aggregates, and complex multi JOIN queries and technology choice on the X-axis (ORM or SQL), with each cell ranging from 0 (less appropriate) to 1 (highly recommended) to indicate the suitability of a given technology for each query type. Therefore, it is advisable to use ORM in systems where development speed and maintainability are priorities, whereas direct SQL is more appropriate for high-load or latency-critical queries.

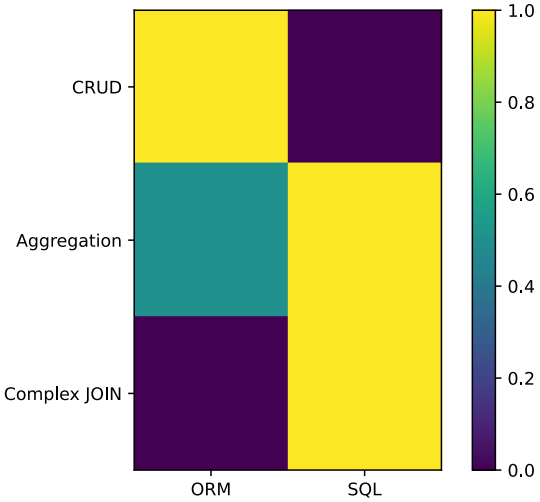


Fig. 7. Technology Sustainability Matrix.

The matrix (**Fig. 7**) demonstrates that ORM is most efficient for simple CRUD operations, moderately suitable for aggregate queries, but less suitable for complex JOINs. On the other hand, direct SQL shows high feasibility for aggregations and complex JOIN queries, providing better performance when working with large amounts of data and complex relationships. The visualization (**Fig. 7**) allows you to quickly evaluate the optimal choice of technology depending on the type of request.

Experimental studies on relational database systems demonstrate that query execution time increases with the number of tables involved in JOIN operations due to the growth of intermediate results and the complexity of join ordering. The presence of indexes on foreign key attributes and efficient join planning significantly influence query performance.

The experiments presented in this study were conducted on a local computer environment, as it is mentioned in the software and environment of the method and materials section, in order to ensure controlled testing conditions and reproducibility of results. However, future work will extend the experimental setup to include remote database servers, which will allow evaluation of query performance under network latency and distributed deployment conditions that are closer to real-world systems.

Future work could expand the model to include additional query types, integrate runtime profiling, and support automated query optimization, extending its practical applicability.

CONCLUSION

This study presented a quantitative performance analysis of SQL and ORM-based query execution in relational databases with inter-table relationships. The experiments are conducted using three representative query types with a complex multi-table join, an aggregation query for counting records, and an aggregation query for summation.

This study also addresses a gap in existing research by systematically evaluating the interplay between ORM strategies and foreign key constraints in complex, relationship-heavy queries, providing actionable insights for developers on when to rely on ORM versus raw SQL for performance-critical operations. Performance is evaluated across four scenarios: ORM with foreign keys, raw SQL with foreign keys, ORM without foreign keys, and raw SQL without foreign keys.

The results demonstrate that raw SQL consistently outperforms ORM-based approaches in all tested scenarios. For the most complex query involving five joined tables, ORM with lazy loading exhibited an average execution time of 32.25 ms, while the equivalent raw SQL query executed in 1.93 ms, making SQL approximately 16.7 times faster. This performance gap is primarily attributed to the *N*+1 query problem inherent in lazy ORM navigation.

For aggregation queries, ORM also showed higher execution times. The booking count query required 5.03 ms using ORM with foreign keys compared to 0.63 ms using raw SQL, while the payment sum query took 5.52 ms with ORM and 0.68 ms with SQL. Thus, raw SQL is approximately 8 times faster for aggregation operations. When foreign key constraints are removed, raw SQL maintains its performance advantage. The complex join query executed in 4.00 ms using SQL without foreign keys, compared to 13.72 ms for ORM with explicit joins, resulting in a 3.4 speedup. Similar trends are observed for aggregation queries, where SQL outperformed ORM by factors ranging from 3.6× to 6.0×.

A comparative evaluation of ORM with explicit joins shows that execution time reduced from 32.25 ms to 13.72 ms, demonstrating a performance improvement of more than 2 times faster. However, even with optimized loading strategies, ORM did not reach the performance level of raw SQL. The analysis also revealed that the presence or absence of foreign key constraints has a negligible impact on raw SQL performance. Observed differences in execution time are below 1 ms and fall within the expected measurement

error range. This indicates that foreign keys primarily contribute to data integrity rather than query performance degradation.

Moreover, the results highlight the potential for intelligent performance analysis frameworks to guide ORM usage in distributed or microservice-based database architectures, where balancing maintainability and efficiency is critical.

To sum up, ORM frameworks provide substantial benefits in terms of development speed, code maintainability, and abstraction, but introduce measurable performance overhead due to additional query generation and object mapping. Raw SQL remains the most efficient solution for performance-critical operations, complex joins, and high-load systems where every millisecond is significant. These findings support a hybrid approach in real-world systems, where ORM is used for standard operations and raw SQL is employed for optimized, high-performance queries.

ACKNOWLEDGMENTS AND FUNDING SOURCES

The authors received no financial support for the research, writing, and/or publication of this article.

COMPLIANCE WITH ETHICAL STANDARDS

The authors declare that the research is conducted in the absence of any conflict of interest.

AUTHOR CONTRIBUTIONS

Conceptualization, [O. R]; methodology, [O.H.]; validation, [O. H.]; investigation, [R. M.]; writing – original draft preparation, [O. R.]; writing – review and editing, [R. M.]; visualization, [O. R.] supervision, [O. H., R. M.].

REFERENCES

[1] Turcotte, A., Aldrich, M. W., & Tip, F. (2023). Reformulator: Automated refactoring of the N+1 problem in database-backed applications. Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), Article 84, 1–12. Association for Computing Machinery. https://doi.org/10.1145/3551349.3556911

[2] Yusmita, J. C., Arya, R., Wijaya, J. M., Suryaningrum, K. M., & Siswanto, R. R. (2025). Optimizing database access strategy: A performance analysis comparison of raw SQL and Prisma ORM. Procedia Computer Science, 269, 1201–1210. https://doi.org/10.1016/j.procs.2025.09.061

[3] Singh, L. K., Tarai, S. K., Sharma, R., Godiyal, A., & others. (2025). A comparison of developer performance across raw SQL, ORM-inspired interfaces, and a Cosmos-specific ODM. International Journal of Scientific Research in Engineering and Management, 9(11), 1–9. https://doi.org/10.55041/IJSREM54453

[4] Hule, K. (2023). Analysis of Different ORM Tools for Data Access Object Tier Generation: A Brief Study. International Journal of Membrane Science and Technology. https://doi.org/10.15379/IJMST.V10I1.2842

[5] Güvercin, A. E., & Avenoglu, B. (2022). Performance analysis of object-relational mapping (ORM) tools in .NET 6 environment. Bilişim Teknolojileri Dergisi, 15(4), 453–465. https://doi.org/10.17671/gazibtd.1059516

[6] Hule, K., & Ranawat, R. (2023). Analysis of different ORM tools for data access object tier generation: A brief study. International Journal of Membrane Science and Technology, 10(1), 1277–1291. https://doi.org/10.15379/IJMST.V10I1.2842

[7] Wiatrowski, T. (2024). Comparative Analysis of ORM Systems for the .NET Platform. Journal of Computer Sciences Institute, 31, 97–102. https://doi.org/10.35784/jcsi.6012

[8] Hermanto, B., Parabi, M. I., Sakethi, D., & Azhar, N. (2025). Performance Comparison of Object-Relational Mapping (ORM) and SQL Query in Developing the Booking Service API at PT Tunas Dwipa Matra. Jurnal Pepadun, 6(2), 113–119. https://doi.org/10.23960/pepadun.v6i2.263
[9] Schwab, P.K., Röckl, J., Langohr, M.S. et al. (2021) Performance Evaluation of Policy-Based SQL Query Classification for Data-Privacy Compliance. Datenbank Spektrum 21, 191–201. https://doi.org/10.1007/s13222-021-00385-9
[10] Colley, D., Stanier, C., & Asaduzzaman, M. (2018). The impact of object-relational mapping frameworks on relational query performance. In 2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE) (pp. 47–52). IEEE. https://doi.org/10.1109/iCCECOME.2018.8659222
[11] coursework_trips_db (Version latest) [GitHub repository]. GitHub. https://github.com/SashaRyback007/coursework_trips_db/tree/master
[12] Dobson, R. (2018). One-Way Analysis of Variance Test Add-on for SQL Statistics Package. MSSQLTips. https://www.mssqltips.com/sqlservertip/5712/a-oneway-analysis-of-variance-test-addon-for-the-sql-statistics-package/
[13] Pangesa, D. M., Astriani, M. S., & Manuaba, I. (2024). Study on object-relational mapping (ORM) data model performance effects in the oil and gas industry. In Proceedings of the 2024 2nd International Conference on Technology Innovation and Its Applications (ICTIIA), 1–6. https://doi.org/10.1109/ICTIIA61827.2024.10761468
[14] Bonteanu, A. M., & Tudose, C. (2024). Performance analysis and improvement for CRUD operations in relational databases from Java programs using JPA, Hibernate, Spring Data JPA. Applied Sciences, 14(7), Article 2743. https://doi.org/10.3390/app14072743
[15] Zhadko-Bazilevych, S. (2025). Analysis of ORM framework approaches for Node.js. Journal of Computer Sciences Institute, 37, 426–430. https://doi.org/10.35784/jcsi.7951
[16] Güvercin, A. E., & Avenoglu, B. (2022). Performance Analysis of Object-Relational Mapping (ORM) Tools in .Net 6 Environment. Bilişim Teknolojileri Dergisi, 15(4), 453-465. https://doi.org/10.17671/gazibtd.1059516
[17] Marchuk, I., Dyyak, I., & Makar, I. (2023). Performance analysis of database access: Comparison of direct connection, ORM, REST API and GraphQL approaches. In 2023 IEEE 13th International Conference on Electronics and Information Technologies (ELIT) (pp. 174–176). IEEE. https://doi.org/10.1109/ELIT61488.2023.10310748

ІНТЕЛЕКТУАЛЬНИЙ АНАЛІЗ РЕЗУЛЬТАТІВ ПРОДУКТИВНОСТІ НА ОСНОВІ СТРАТЕГІЙ ОБ'ЄКТНО-РЕЛЯЦІЙНОГО ВІДОБРАЖЕННЯ ТА ОБМЕЖЕНЬ ЗОВНІШНЬОГО КЛЮЧА В БАЗАХ ДАНИХ SQL

Олександра Рибак **, *Олег Гусак*** **, *Роман Мисюк*** *****
Кафедра системного проектування
Львівський національний університет імені Івана Франка,
вул. Драгоманова 50, м. Львів, 79005, Україна

АНОТАЦІЯ

Вступ. Швидке зростання застосунків, які використовують велику кількість даних, підвищує важливість ефективного виконання запитів у реляційних базах даних. Хоча системи об'єктно-реляційного відображення (ОРВ) спрощують розробку та

покращують підтримуваність коду, їхній рівень абстракції може створювати вимірювані затримки, а вплив обмежень зовнішніх ключів на швидкість виконання запитів залишається практичною проблемою, особливо в мікросервісних архітектурах, що дотримуються принципу «База даних на сервіс».

Матеріали та методи. Інформаційну систему було розроблено на основі реляційної бази даних та програмного каркасу ОРВ SQLAlchemy, зі схемою, що включала зв'язки один-до-одного, один-до-багатьох та багато-до-багатьох, протестовані як з обмеженнями зовнішніх ключів, так і без них. Було виконано три типові запити: отримання даних бронювання, агрегація пов'язаних записів та обчислення загальних платежів, використовуючи підходи необробленого SQL та ОРВ, при цьому інтелектуальний алгоритм аналізував продуктивність, виявляв потенційні проблеми *N*+1 запитів та рекомендував оптимальні стратегії, наприклад явні операції з'єднання таблиць JOIN.

Результати. Запити на raw SQL стабільно показували кращу продуктивність у всіх сценаріях. Найбільша різниця спостерігалася в ОРВ -запитах, уражених проблемою *N*+1 запитів, де час виконання перевищував аналогічні SQL-запити більш ніж у десятки разів. Агрегаційні запити демонстрували менші, але стабільні затримки. Наявність або відсутність обмежень зовнішніх ключів майже не впливала на продуктивність необробленого SQL. Використання явних операцій JOIN в ОРВ суттєво зменшувало затримки порівняно з неявною навігацією по зв'язках. Інтелектуальний аналіз точно визначав запити з високим рівнем ризику та пропонував ефективні рекомендації, підтверджені експериментально.

Висновки. ОРВ-системи покращують продуктивність розробки та підтримуваність, але вводять вимірювані затримки. Для критично важливих завдань продуктивності перевагу слід надавати SQL, а обмеження зовнішніх ключів незначно впливають на швидкість виконання. Інтеграція інтелектуального аналізу продуктивності дозволяє приймати зважені рішення між ефективністю та підтримуваністю у складних реляційних системах.

Ключові слова: реляційні бази даних, продуктивність SQL, ОРВ, система підтримки рішень, інтелектуальний аналіз, проєктування баз даних

Received / Одержано	Revision / Доопрацьовано	Accepted / Прийнято	Published / Опубліковано
13 February, 2026	10 March, 2026	16 March, 2026	30 March, 2026