electronics

MDPI

Article

Benchmarking PHP–MySQL Communication: A Comparative Study of MySQLi and PDO Under Varying Query Complexity

Nebojša Andrijević [1,*], Zoran Lovreković [2], Hadžib Salkić [3], Đorđe Šarčević [4] and Jasmina Perišić [5]

[1] Faculty of Economics and Engineering Management, University Business Academy in Novi Sad, Cvećarska 2, 21102 Novi Sad, Serbia
[2] Faculty of Technical Sciences, Department of Energy, Electronics and Telecommunications, University of Novi Sad, Dr Zorana Đinđića 1, 21000 Novi Sad, Serbia; lovra61@mts.rs
[3] Faculty of Information Technology, FINRA University Tuzla, Mitra Trifunovića Uče 9, 75000 Tuzla, Bosnia and Herzegovina; hadzib.salkic@ceps.edu.ba
[4] Department of Information Technology, Academy of Applied Studies Šabac, Dobropoljska 5, 15000 Šabac, Serbia; sarcevicdjordje@vmpts.edu.rs
[5] Faculty of Informatics and Computing, Singidunum University, Danijelova 32, 11010 Belgrade, Serbia; jperisic@singidunum.ac.rs
* Correspondence: andrijevicnebojsa@gmail.com

Abstract

Efficient interaction between PHP (Hypertext Preprocessor) applications and MySQL databases is essential for the performance of modern web systems. This study systematically compares the two most widely used PHP APIs for working with MySQL databases—MySQLi (MySQL Improved extension) and PDO (PHP Data Objects)—under identical experimental conditions. The analysis covers execution time, memory consumption, and the stability and variability of results across different types of SQL (Structured Query Language) queries (simple queries, complex JOIN, GROUP BY/HAVING). A specialized benchmarking tool was developed to collect detailed metrics over several hundred repetitions and to enable graphical and statistical evaluation. Across the full benchmark suite, MySQLi exhibits the lowest mean wall-clock execution time on average (≈15% overall). However, under higher query complexity and in certain connection-handling regimes, PDO prepared statement modes provide competitive latency with improved predictability. These results should be interpreted as context-aware rankings for the tested single-host environment and workload design, and as a reusable benchmarking framework intended for replication under alternative deployment models. Statistical analysis (Kruskal–Wallis and Mann–Whitney tests) confirms significant differences between the methods, while Box-plots and histograms visualize deviations and the presence of outliers. Unlike earlier studies, this work provides a controlled and replicable benchmarking environment that tests both MySQLi and PDO across multiple API modes and isolates the impact of native versus emulated prepared statements. It also evaluates performance under complex-query workloads that reflect typical reporting and analytics patterns on the ClassicModels schema. To our knowledge, no previous study has analyzed these factors jointly or provided a reusable tool enabling transparent comparison across PHP–MySQL access layers. The findings provide empirical evidence and practical guidelines for choosing the optimal API depending on the application scenario, as well as a tool that can be applied for further testing in various web environments.

Keywords: PHP; MySQLi; PDO; prepared statements; benchmark testing; query performance; query execution time; memory usage; web applications; SQL optimization; database connectivity; execution time variability; non-parametric analysis; statistical benchmarking; native vs. emulated prepared statements; reliability; scalability

Check for updates

Academic Editor: Porfirio Tramontana

Received: 29 November 2025
Revised: 16 December 2025
Accepted: 18 December 2025
Published: 20 December 2025

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

In the contemporary development of web applications, efficient communication between the application layer and the database has a crucial impact on overall system performance and scalability. In this context, PHP remains one of the most widely used languages for web-application development, while MySQL is the dominant solution among relational database systems. Different mechanisms for connecting to and executing SQL queries within PHP have a direct impact on application speed and memory efficiency. In particular, the choice between the MySQLi and PDO extensions also affects security.

Although both extensions are widely adopted and support modern syntax and security mechanisms (such as prepared statements), there are numerous differences in their architecture, mode of operation, and tuning capabilities. Particular attention is warranted for performance in the context of complex SQL queries. These include queries with multiple JOIN, GROUP BY, and HAVING clauses, as well as other constructs that burden both the application and database layers of the system.

The existing literature provides a series of benchmark tests comparing PDO and MySQLi, but often without clear control of testing conditions or without including more complex queries and different regimes of prepared statements. In addition, the results of previous studies are often contradictory, which makes it difficult to make technically grounded decisions in the development of applications that require a high level of performance.

For this reason, this paper undertakes a systematic experimental analysis of the following database access methods in PHP:

- MySQLi (procedural style);
- MySQLi (object-oriented style);
- MySQLi with prepared statements;
- PDO with direct query execution;
- PDO with prepared statements (emulated mode);
- PDO with prepared statements (native mode).

The experiment is designed so that, across multiple iterations (n = 200 per scenario), SQL queries are executed using the ClassicModels database, with measurements of two key performance metrics: execution time and memory consumption. Particular attention is devoted to the differences that arise when opening the connection inside vs. outside each test, thereby simulating different production scenarios (e.g., microservices vs. monolithic applications).

The aim of this paper is to provide:

- empirically grounded recommendations for choosing a method of connecting to a MySQL database in PHP applications;
- an analysis of the trade-offs between speed, memory efficiency, and flexibility;
- a contribution to the literature through a transparent and replicable testing methodology.

For this purpose, a performance-testing framework was designed and implemented to evaluate the operation of the MySQLi and PDO APIs under identical conditions. The focus is on key performance indicators, including execution time, memory usage, and system behavior under load. The obtained results offer practical insights for developers who optimize interaction between PHP and MySQL, along with a tool that can be reused in similar research and development environments.

While several previous works have attempted to compare MySQLi and PDO, the majority either lack strict control of experimental conditions or do not differentiate between native and emulated prepared statement execution. Many also exclude complex SQL constructs that are common in real-world applications. Moreover, no publicly documented study provides a reproducible benchmarking tool that captures both timing and memory

metrics across multiple PHP–MySQL access modes. This paper addresses these gaps by offering a controlled, transparent, and replicable comparison framework together with the first systematic evaluation that jointly examines API mode, query complexity, connection-handling strategy, and prepared statement implementation.

In the following sections, the theoretical background, the methodology of the conducted experiment, the performance evaluation, the discussion of the obtained results, and the conclusions that summarize the optimal approaches in different usage contexts will be presented.

2. Theoretical Background

2.1. Communication Between PHP and the MySQL Database

The PHP programming language is one of the most commonly used solutions for developing web applications, while MySQL is among the most prevalent relational database management systems (RDBMS). Efficient and secure data exchange between a PHP application and a MySQL database is crucial for overall system performance, especially in scenarios that involve complex queries and work with large data volumes.

PHP provides multiple programming interfaces for database communication, with MySQLi (MySQL Improved) and PDO (PHP Data Objects) currently being the most widely used. These extensions enable connecting to the database, executing SQL queries, and processing results, with support for security techniques such as prepared statements.

2.2. The MySQLi Extension: Characteristics and Access Modes

The MySQLi extension was developed as a replacement for the obsolete MySQL extension and is intended exclusively for working with MySQL databases. It enables two access paradigms:

- Procedural style, in which functions are called directly with explicit parameter passing (e.g., mysqli_query($conn, $sql)).
- Object-oriented style, where an instance of the MySQLi class is used (for example, invoking the query method of the $mysqli object, $mysqli->query($sql)), which allows for greater modularity and code readability.

A particular advantage of MySQLi lies in its support for native prepared statements, which make it possible to separate SQL logic from user data. This technique contributes to protection against SQL injection and to reducing overall execution time in cases where the same query is used multiple times with different parameters [1].

2.3. PDO (PHP Data Objects): An Abstract Database Layer

PDO is an extension that provides a universal interface for accessing different database systems, including MySQL, PostgreSQL, SQLite, MSSQL, and others. Unlike MySQLi, PDO is strictly object oriented and not specific to a single database type, which makes it suitable for applications that require flexibility in the choice of RDBMS.

PDO enables two ways of executing SQL queries:

- Direct query execution, using the query() method, when SQL statements are sent directly to the server;
- Using prepared statements, which can be realized in two modes:
    - o Emulated mode, where the preparation and evaluation of the statement are performed on the PHP side;
    - o Native mode, where prepared statements are processed directly on the database side, assuming the driver supports it [2].

The prepared statement mode in PDO can be controlled by the PDO::ATTR_EMULATE _PREPARES parameter.

From a performance perspective, emulated prepared statements keep more work on the PHP side (client-side parsing and parameter interpolation), whereas native prepared statements offload parsing and execution plans to the database engine. This difference affects both memory footprint and the number of round-trips between the PHP process and MySQL, especially for complex queries with many parameters.

2.4. Prepared Statements: Security and Performance Aspects

Prepared statements are a method of executing SQL queries that separates the structure of the query from input data. The query is first prepared and compiled on the database side, after which it is executed multiple times with different parameters.

In this way, the following is achieved:

- Increased security—by eliminating the possibility of SQL injection, because data are not part of the SQL statement itself [3];
- Improved performance—especially in cases of repeated execution of the same query with different values (e.g., in loops).

Although both the MySQLi and PDO extensions rely on the concept of prepared statements, there are technical differences in their realization (native vs. emulated modes), which can affect execution speed, particularly with complex queries.

2.5. Comparative Analysis of Approaches

A comparative analysis of the MySQLi and PDO extensions encompasses several dimensions: supported databases, flexibility, programming style, and security mechanisms. The key differences are presented in Table 1.

Table 1. Comparative characteristics of the MySQLi and PDO extensions.

Feature	MySQLi	PDO
Supported DBMS	MySQL only	Multiple databases (MySQL, SQLite, etc.)
Programming style	Procedural and OOP	OOP only
Prepared statements	Native	Emulated and native
Flexibility	Low	High
Multi-database support	No	Yes

2.6. Review of Relevant Literature and Industry Validation Examples

Although numerous papers analyze the performance of PHP–MySQL communication, most focus on basic SQL operations and do not clearly separate prepared statement regimes [4,5]. More complex queries—those including JOIN operations, nested SELECT statements, and transactions—have rarely been the subject of rigorous performance evaluation, particularly in application-like scenarios.

Several peer-reviewed studies provide insights into database and web-stack performance. Salunke and Ouda [6] compared PostgreSQL and MySQL in controlled benchmarks, confirming that differences depend strongly on query complexity. Zmaranda et al. [7] analyzed MySQL Document Store versus Elasticsearch, showing how configuration choices affect performance. Győrödi et al. [8] studied MySQL optimization techniques, highlighting the impact of indexing and query structure on execution time. Taipalus [9] presented a systematic review of DBMS performance comparisons, identifying reproducibility and methodological rigor as recurring challenges. Yuan et al. [10] focused on PHP applications by proposing static detection for SQL injection vulnerabilities in MySQLi and PDO. Kamani

et al. [11] provided an early benchmark of PHP–MySQL connectivity (MySQL, MySQLi, PDO), indicating only modest throughput differences under standard workloads.

In addition to these peer-reviewed contributions, several professional comparative tests have been published online. For instance, two widely circulated technical articles—"Boost Your PHP Database Performance: How PDO Outperforms MySQLi" [12] and "Understanding Prepared Statements: PDO vs. MySQLi in PHP Development" [13]—presented benchmark-style overviews of PDO and MySQLi under different SQL operations. Both sources describe testing on local MySQL instances using repeated query execution (SELECT, INSERT, UPDATE, DELETE) and emphasize that PDO's abstraction layer can, under certain configurations, provide more efficient execution and lower resource overhead. However, these results are practitioner observations rather than controlled academic studies; therefore, they are included only as contextual evidence illustrating professional interest in PHP database performance.

The aim of the present research is to experimentally compare the performance of the following six methods for executing complex SQL queries in PHP with a MySQL database:

- **M1 (MySQLi procedural);**
- **M2 (MySQLi object-oriented);**
- **M3 (MySQLi prepared statements);**
- **M4 (PDO query execution);**
- **M5 (PDO prepared statements—emulated);**
- **M6 (PDO prepared statements—native).**

This comparative analysis seeks to identify performant approaches for scenarios that approximate complex application requirements.

To broaden the context, both peer-reviewed and non-peer-reviewed materials were examined to understand how PDO and MySQLi are benchmarked in practice. Only peer-reviewed studies inform the hypotheses and interpretations presented here; non-peer-reviewed materials are discussed separately as industry validation examples, used solely to illustrate professional practice and motivation.

2.6.1. Industry Validation Examples (Non-Peer-Review)

This subsection distinguishes peer-reviewed studies (scientific evidence) from industry validation examples (non-peer-review, included only to illustrate professional practice). A limited number of practitioner write-ups (e.g., vendor tools and engineering blogs) are summarized strictly as context for problem relevance and real-world experimentation patterns.

- EverSQL (2023) [14]: Summarizes comparative outcomes for PDO and MySQLi across different PHP versions on a sizable single-table dataset. The post reports generally small differences between the APIs and emphasizes that performance depends strongly on PHP and MySQL versions.
- Jim Westergren (2016–2017) [15]: Presents timing experiments across PDO (emulated/native) and MySQLi variants, noting that prepared statement behavior is context-dependent and that emulation can alter observed performance.
- Wooptoo (2012) [16]: Reports extensive repetition benchmarks (INSERT, SELECT, UPDATE, DELETE) comparing PDO and MySQLi with and without prepared statements, showing that results vary widely with configuration and workload size.
- USAVPS (2024) [17]: A hosting-provider article aggregating community tests and adding VPS-based trials; it concludes that the choice of API depends more on architectural flexibility and portability needs than on raw speed.

These non-peer-review items are not used to support or challenge our statistical findings; they merely reflect practitioner interest and heterogeneous observations in uncontrolled environments.

2.6.2. The Need for Systematic Experimental Research

The mixed and often contradictory findings across both academic and practitioner sources underscore the necessity for controlled and replicable experiments. Key sources of inconsistency include:

- Different PHP/MySQL versions and configurations affecting internal optimization;
- Diverse database sizes, caching strategies, and repetition counts;
- Distinct implementations of prepared statements (native vs. emulated);
- Divergent focus on metrics such as execution time, memory footprint, or code portability.

These inconsistencies reveal that there is no universal rule regarding which API is faster or more efficient. Performance depends on application context, database complexity, and the frequency of query repetition.

Consequently, there is a clear need for comprehensive, controlled experiments that

- Compare PDO and MySQLi approaches under identical conditions using a moderately sized, well-structured relational dataset (e.g., ClassicModels) suitable for reproducibility;
- Measure execution time with strict variable control and apply statistical validation (e.g., Kruskal–Wallis, Mann–Whitney);
- Include different query categories (simple SELECT, multi-JOIN, GROUP BY/HAVING), with and without prepared statements;
- Take into account the versions of PHP and MySQL to ensure relevance to modern production environments.

This study addresses these needs by providing a reproducible benchmarking framework, detailed methodology, and transparent statistical evaluation of PHP–MySQL interface performance.

3. Research Methodology

3.1. Description of the Experimental Environment

The experiments were conducted on a local workstation with the following hardware–software characteristics:

- Operating system: Windows 10 Home 22H2 (64-bit);
- WAMP environment: WampServer 3.3.0 (64-bit), which includes
    - Apache 2.4.54.2;
    - MySQL 8.0.31;
    - PHP 8.0.26;
    - phpMyAdmin 5.2.
- Hardware: Intel Core i7-2630QM, 12 GB RAM, 120 GB SSD.

The ClassicModels database, downloaded from www.mysqltutorial.org, was used; it offers a realistic dataset for testing SQL queries of varying complexity.

The experimental setup is depicted in Figure 1. The benchmark scripts were executed on PHP 8.0 running under Apache, connected to a MySQL 8.0 database (ClassicModels schema). Execution time and memory consumption were logged into CSV files, which were subsequently processed by Python (version 3.13.9) scripts for statistical analysis.

It is important to note that MySQL 8.0 does not include the legacy SQL query cache. Therefore, repeated executions of identical SQL statements do not benefit from cached result sets at the SQL layer. However, as in any modern relational database system, InnoDB and OS-level page caching may accelerate subsequent reads of the same data. Since all six tested methods are executed under identical conditions and with the same number of repetitions, these caching mechanisms affect all methods symmetrically. Accordingly, the measurements represent steady-state performance under warmed-up caches rather than cold-start latency.

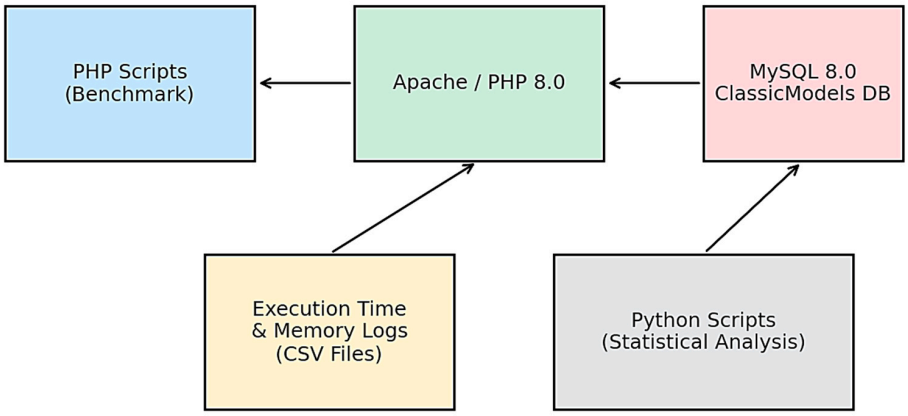


Figure 1. Experimental Environment.

All experiments were executed on a dedicated workstation with minimal background services. Windows indexing, scheduled tasks, and automatic updates were disabled, and no concurrent user applications were active during benchmarking. The PHP scripts and MySQL server ran in a single-user WAMP environment, ensuring that all methods were evaluated under consistent operating system conditions.

3.2. Defined Query Tests

The experimental environment was intentionally restricted to a single local workstation in order to isolate the PHP–MySQL interaction layer from external variability. Factors such as network latency, multi-client concurrency, connection pooling, or distributed server configurations introduce additional sources of noise that would confound attribution of effects to the driver layer. Isolating the system in this manner ensures that observed differences in execution time and memory usage arise from the access modes themselves rather than from unrelated system-level influences.

Testing was carried out for the following database access methods in PHP:

- MySQLi (procedural);
- MySQLi (object-oriented);
- MySQLi prepared statements;
- PDO (object-oriented);
- PDO prepared statements emulated;
- PDO prepared statements native.

For each method, the following SQL queries were analyzed (Table 2), implemented in three variants (classic, positional, and named prepared statements).

Table 2. Analyzed SQL types of queries.

Label	Query Type	Number of JOINs	Clauses
Q1	Simple SELECT with WHERE and LIMIT	0	WHERE, LIMIT
Q2	SELECT with 5 JOINs	5	WHERE, LIMIT
Q3	SELECT with 2 JOINs, GROUP BY and HAVING	2	WHERE, GROUP BY, HAVING

For each query, two benchmark variants were tested:

- Full benchmark: includes establishing the connection;
- Isolated execution: only the execution of a previously prepared statement is measured.

The stepwise procedure of the benchmark function is illustrated in Figure 2, showing initialization, query preparation, repeated execution (200 iterations), and logging of performance metrics for subsequent analysis.

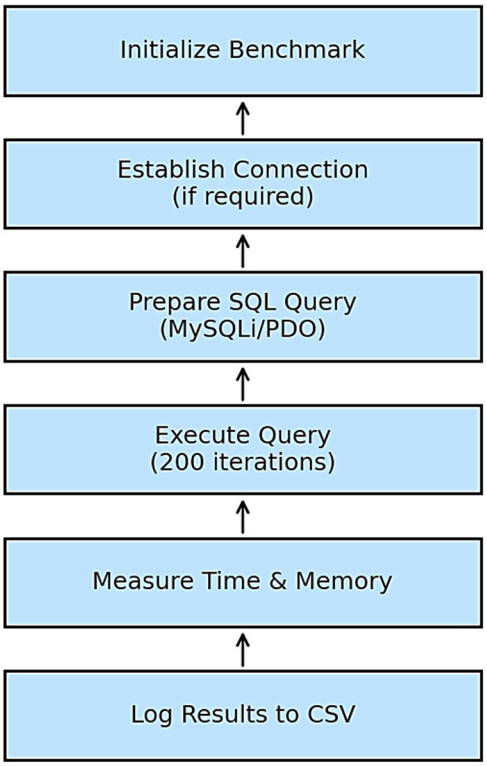


Figure 2. Workflow of benchmark function.

3.3. Performance Metrics

The primary performance metrics were

- Query execution time (milliseconds (ms));
- Peak memory usage (kilobytes (kB)).

In the initial phase of the experiment, measuring CPU time was additionally considered as a potential metric. However, the values were highly unstable—most executions recorded 0 µs, while sporadic anomalies such as 15,625 µs occurred, without a consistent relationship to the method or query complexity. Before discarding CPU time, we attempted to stabilize measurements by minimizing background activity (disabling non-essential services and repeating full benchmark runs), but the high variance persisted.

On commodity Windows workstations, user-level scripts cannot fully control kernel scheduling, disk caching, or other processes competing for CPU slices, which makes fine-grained CPU time an unreliable indicator in this context. For these reasons, CPU time was excluded as a primary metric, and the analysis focuses on wall-clock execution time and memory efficiency, which are more robust and directly relevant from the perspective of real PHP/MySQL applications.

User-space CPU timing for PHP is subject to substantial variability due to kernel scheduling and background OS activity. Preliminary measurements showed inconsistent CPU values even under identical conditions, consistent with prior observations in dynamic language environments. Achieving stable and interpretable CPU time measurements would require kernel-level instrumentation (e.g., perf), which lies outside the scope of this application-layer benchmark. For this reason, wall-clock time was adopted as the primary metric, as it demonstrated high stability across 200 repetitions.

3.4. Description of Test Implementation

Benchmark measurements were implemented through a dedicated PHP function benchmark(), which enables multiple (n = 200) executions of the code under test and logs

execution time and memory usage (Listing 1). The function takes as an argument an anonymous function (callable) that represents the specific test.

Each method–query combination is executed 200 times under identical conditions. This design ensures that any warm-up effects of the InnoDB buffer pool and OS page cache influence all methods in the same way, so that the comparative ranking and relative differences between methods can be interpreted as steady-state behavior under warmed caches.

Listing 1. Benchmark function.

```
function benchmark(callable $fn, string $method, int $runs = 200) {
    $times = [];
    $memoriesPeak = [];
    for ($i = 0; $i < $runs; $i++) {
        gc_collect_cycles();
        $start = hrtime(true);
        $startMem = memory_get_usage(false);
        $fn();

        $time = (hrtime(true) − $start)/1e6;//ms
        $peakMem = (memory_get_peak_usage(false) − $startMem)/1024;//kB
        $times[] = $time;
        $memoriesPeak[] = $peakMem;
        logResult($method, $time, $peakMem);
    }
    $avgTime = round(array_sum($times)/count($times), 3);
    $avgPeakMem = round(array_sum($memoriesPeak)/count($memoriesPeak), 3);
    logResult2($method, $avgTime, $avgPeakMem);
}

Example of a test function with connection included (MySQLi prepared):

benchmark(function() use ($host, $user, $pass, $db, $query, $productLine, $minAvg) {
    $conn = new mysqli($host, $user, $pass, $db);
    $stmt = $conn->prepare($query);
    $stmt->bind_param("si", $productLine, $minAvg);
    $stmt->execute();
    while ($row = $stmt->get_result()->fetch_assoc()) {}
    $stmt->close();
    $conn->close();
}, "mysqli_prepared");

Example of a test function without connection (PDO prepared):

$stmt = $pdo->prepare($query);
$stmt->bindParam(':status', $status, PDO::PARAM_STR);
benchmark(function() use ($stmt) {
    $stmt->execute();
    while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {}
    $stmt->closeCursor();
}, "pdo_prepared_static");
```

For each tested method and query, 200 measurements were performed, yielding a total of 1200 measurements per query type. The results were saved to CSV files in the following format:

method, time_ms, memory_peak_kB
mysqli_procedural_static, 5.53, 58.46

...

The complete PHP scripts for measuring the performance of different approaches to query execution are available in the repository, at the link https://github.com/ZLovrekovic/PHP-extensions-compare/tree/main/scripts/benchmark (accessed on 14 September 2025).

3.5. Reproducibility

All SQL queries used, PHP scripts, and CSV results are available in electronic form in the project repository, at the link ZLovrekovic/PHP-extensions-compare. This ensures full replicability of the experiment, in accordance with the principles of open science.

4. Evaluation of the Performance of Database Access Methods

This chapter presents a detailed comparative analysis of execution time and memory usage for six widely used database access methods in PHP when executing a simple SELECT query, a complex SELECT query with 5 JOINs, and a complex SELECT query with HAVING and GROUP BY clauses. All three queries contain a WHERE clause. The evaluation includes both object-oriented and procedural interfaces using MySQLi, as well as native (built-in) and emulated prepared statements using PDO.

4.1. Experimental Environment

All tests were performed under controlled conditions using PHP 8.x with MySQL 8.x, with Apache enabled and explicit garbage collection management between runs. Each test was repeated 200 times for statistical reliability.

In MySQL 8, the legacy query cache mechanism is removed; therefore, no explicit SQL-level query caching affected the results. However, the InnoDB buffer pool and the operating system's file cache naturally tend to warm up during repeated executions of the same query. We did not attempt to flush these caches between iterations because our goal was to emulate typical production behavior of long-running PHP applications. PHP opcache was left with default WAMP configuration, and since all benchmarked methods are implemented within the same script, its effect is symmetrical across MySQLi and PDO variants. Consequently, the reported values should be interpreted as comparative timings under a warmed-cache scenario rather than absolute cold-start latencies.

The key performance indicators (KPI) included

- Average execution time (in milliseconds);
- Peak memory usage (in kilobytes);
- Standard deviation;
- Skewness (asymmetry) and kurtosis (peakedness of the distribution);
- 95% confidence intervals (CI).

Note: In the method labels, the suffix "_static" denotes scenarios in which the connection to the database is established outside the benchmark loop itself (a single connection for all iterations), while methods without this suffix refer to cases where the connection is established within each iteration of the test. The measurement results are provided as CSV files (raw data) and are available at the following link: https://github.com/ZLovrekovic/PHP-extensions-compare/tree/main/data/raw (accessed on 14 September 2025).

After the measurements were completed, statistical analysis of the obtained results was carried out, first for descriptive statistics (mean, median, standard deviation, skewness and kurtosis, as well as the 95% confidence interval) [18,19]. Thereafter, tests of normality of the distribution (Shapiro–Wilk and D'Agostino–Pearson tests) were performed [20,21]. Due to the pronounced deviation from the normal probability distribution, non-parametric tests (the Kruskal–Wallis test) and post hoc tests (the Dunn or Mann–Whitney test) were

performed for all measurements [22,23], comparing the query execution methods for all three test queries. The complete results of the descriptive statistics are provided in the repository at https://github.com/ZLovrekovic/PHP-extensions-compare/tree/main/results/statistical-tests/descriptive-analysis (accessed on 14 September 2025).

To preserve methodological transparency and enable precise cross-scenario comparison, Sections 4.2–4.7 follow an identical analytical structure for each query type and connection regime, covering execution time, memory usage, distributional statistics, confidence intervals, and non-parametric significance testing. While this results in a repetitive layout, it ensures consistency and reproducibility across all experiments. Readers primarily interested in overall trends and comparative insights may focus on the synthesized discussion in Section 5, where results are aggregated and interpreted across query types and execution contexts.

To avoid ambiguity and improve navigability, all subsection and table titles in this section are explicitly associated with the corresponding evaluation method (e.g., M1–M6), allowing analyses with identical names across different experimental scenarios to be clearly differentiated.

4.2. Analysis for a Simple SELECT Query (Connection Outside the Benchmark Test (M1))

4.2.1. Execution-Time Analysis (M1)

PDO emulated prepared statements are parsed and bound entirely client side, sending complete SQL text to the server on each execution. Native prepared statements involve server-side preparation, where placeholders are transmitted separately, and the server executes precompiled statements. This distinction explains the observed differences in execution time and variance.

Table 3 shows the average execution times, standard deviation, range, and skewness for each method.

Table 3. Summary of execution time (M1).

Method	Mean (ms)	Std Dev	Range	Skewness
MySQLi_object_static	4.36	0.66	4.47	3.65
MySQLi_prepared_static	4.10	0.34	2.50	3.63
MySQLi_procedural_static	4.39	0.64	4.87	3.13
PDO_query_static	4.39	0.54	3.46	3.01
PDO_prepared_static	4.21	0.37	2.71	3.39
PDO_prepared_emulated_static	4.30			

Figure 3 provides a visual comparison of the average execution time for all six methods using a static connection. The results confirm that MySQLi_prepared_static consistently outperforms other approaches, while PDO_query_static and MySQLi_procedural_static are less efficient.

Observations:

- The MySQLi_prepared_static method had the lowest average execution time and one of the smallest variances, indicating speed and consistency.
- Although pdo_prepared_emulated_static did not have the best average time, it showed the smallest variation, which can be useful in latency-sensitive applications.
- All methods had high positive skewness values, indicating occasional slow executions (likely due to external I/O factors or caching).

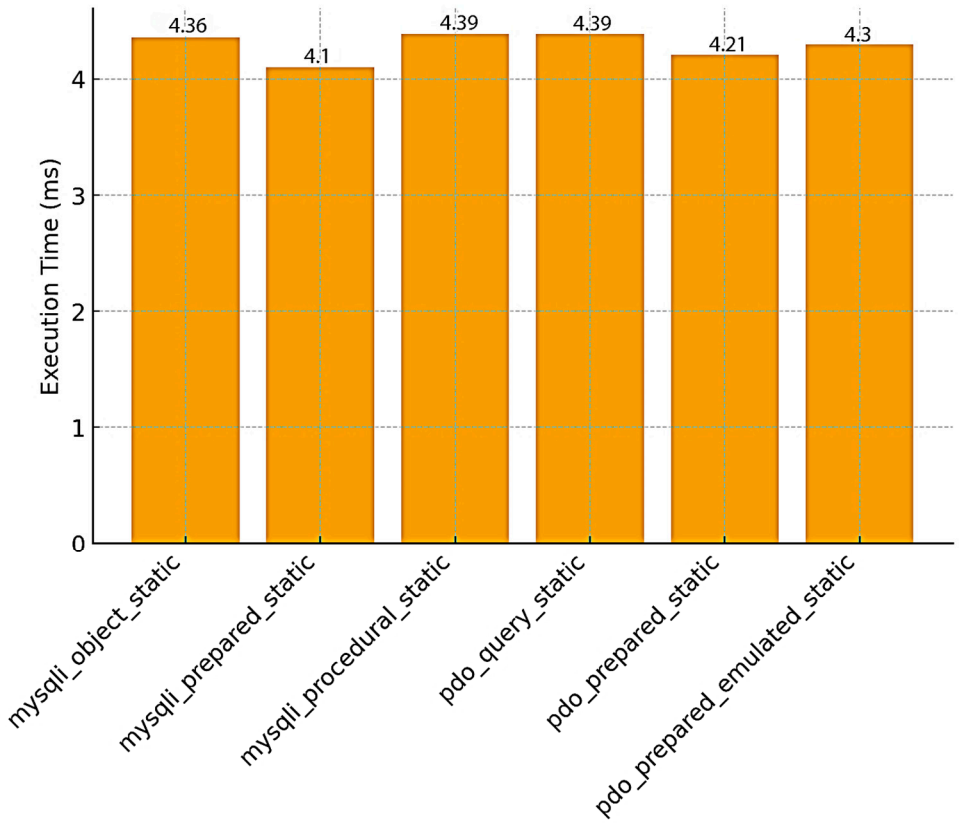


Figure 3. Average execution time.

4.2.2. Memory Usage Analysis (M1)

Table 4 shows the data on peak (maximum) memory consumption for each method.

Table 4. Summary of peak memory consumption (M1).

Method	Mean (kB)	Std Dev	Range	Skewness
MySQLi_object_static	51.45	8.52	16.05	0.85
MySQLi_prepared_static	34.32	2.32	7.87	1.34
MySQLi_procedural_static	46.52	6.02	16.05	0.85
PDO_query_static	66.98	8.65	24.11	0.37
PDO_prepared_static	40.59	7.79	21.85	0.53
PDO_prepared_emulated_static	65.34	8.65	24.27	0.37

Figure 4 illustrates peak memory consumption across methods. The MySQLi_prepared_static method exhibits the lowest memory footprint, whereas PDO-based queries consume significantly more memory.

Observations:

- MySQLi_prepared_static consistently had the lowest memory consumption, making it the most efficient method in terms of memory.
- Emulated prepared statements in PDO (PDO_prepared_emulated_static) used almost the same amount of memory as PDO_query_static, indicating that emulation does not bring advantages in memory efficiency.
- Skewness values were generally low and positive, indicating a slight asymmetry in memory usage across methods.

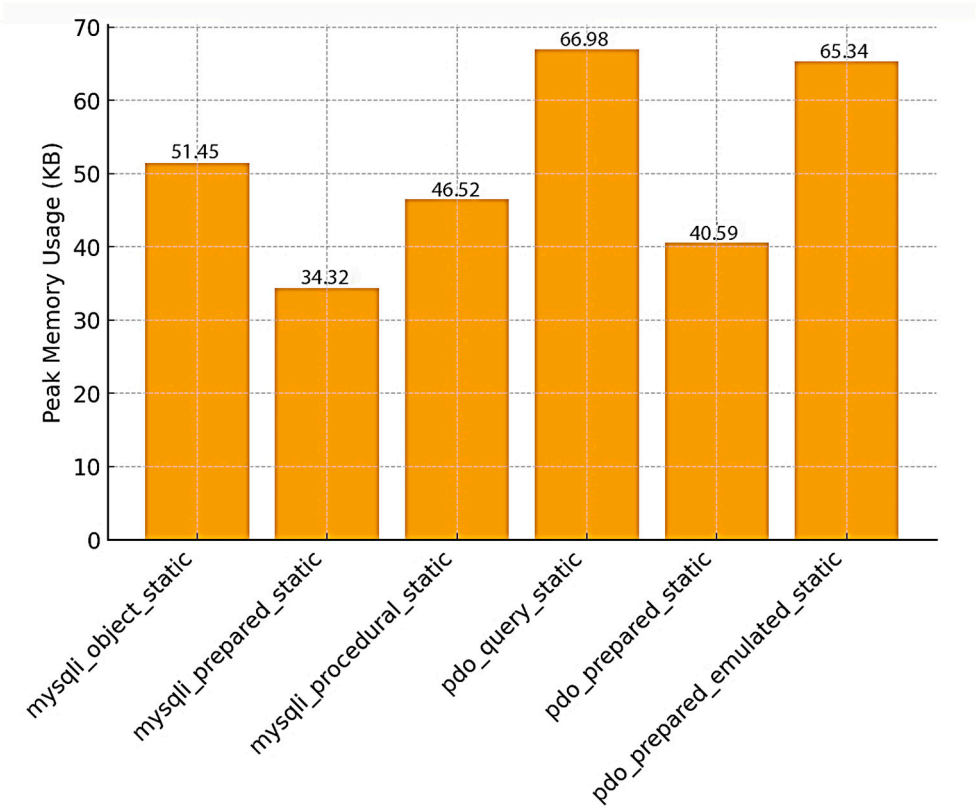


Figure 4. Average memory usage.

A standard deviation of 0 in some scenarios reflects deterministic memory allocation behavior of PHP's Zend allocator under repeated identical operations, rather than measurement error.

4.2.3. Relationship Between Mean and Median Values (M1)

Comparing the mean and median values reveals the internal distribution of measurements and potential extreme values (outliers) (Table 5).

Table 5. Mean and median values (M1).

Method	Mean (ms)	Median (ms)	Mean (kB)	Median (kB)
MySQLi_object_static	4.36	4.16	51.45	42.41
MySQLi_prepared_static	4.10	4.02	34.32	32.87
MySQLi_procedural_static	4.39	4.19	46.52	42.41
PDO_query_static	4.39	4.24	66.98	65.70
PDO_prepared_static	4.21	4.12	40.59	38.48
PDO_prepared_emulated_static	4.30	4.28	65.34	64.05

The small difference between the mean and the median for PDO_prepared_emulated_static indicates a narrow distribution and minimal presence of extreme values in execution time. In contrast, MySQLi_object_static shows a greater deviation, implying greater sensitivity to occasional performance spikes.

4.2.4. Skewness and Kurtosis (M1)

The values for skewness and kurtosis are given in Table 6.

Table 6. Skewness and kurtosis (M1).

Method	Skewness (ms)	Kurtosis (ms)	Skewness (kB)	Kurtosis (kB)
MySQLi_object_static	3.65	14.75	0.85	−1.16
MySQLi_prepared_static	3.63	16.09	1.34	0.39
MySQLi_procedural_static	3.13	12.70	0.85	−1.16
PDO_query_static	3.01	9.38	0.37	−1.39
PDO_prepared_static	3.39	14.74	0.53	−1.33
PDO_prepared_emulated_static	3.68	21.61	0.37	−1.38

The emulated PDO approach has the highest kurtosis in time—meaning a pronounced sharp peak with the presence of outliers (very slow executions). For memory, the distribution is flat with short tails (negative kurtosis).

4.2.5. Confidence Interval (95%)—Time and Memory (M1)

Confidence intervals for execution time and memory usage are given in Table 7.

Table 7. Confidence interval (time and memory (M1)).

Method	CI 95% (Time_ms)	CI 95% (Memory_kB)
MySQLi_object_static	~0.091	~1.18
MySQLi_prepared_static	~0.047	~0.32
MySQLi_procedural_static	~0.091	~0.84
PDO_query_static	~0.076	~1.20
PDO_prepared_static	~0.051	~1.09
PDO_prepared_emulated_static	~0.029	~1.21

The emulated PDO has the smallest margin of error for time (thanks to the low standard deviation), but the memory interval remains wide—confirming memory instability despite temporal stability.

Figure 5 shows Box-plot distributions of execution times. The results highlight the stability of the PDO_prepared_emulated_static approach, which achieves a narrow variance compared to other methods.

4.2.6. Conclusions of the Descriptive Statistics Analysis for the Simple SELECT Query (M1)

Descriptive statistics results for a simple select query show the following:

- MySQLi_prepared_static is the fastest and most stable in terms of both time and memory.
- PDO_prepared_static (native): a good compromise—better performance than PDO_query_static, lower memory consumption.
- PDO_prepared_emulated_static:
  - o More stable time (lowest standard deviation), but
  - o Almost the same memory consumption as PDO_query_static—emulation clearly does not bring memory efficiency.

Based on the overall analysis of the results for the simple SELECT query, the following recommendations can be defined:

- For maximum efficiency and performance accuracy—use the MySQLi prepared approach.
- If PDO is used, native prepared statements should be preferred, and emulated ones avoided when memory consumption is important.

High skewness and kurtosis still indicate the need for additional analysis of outliers in production environments.

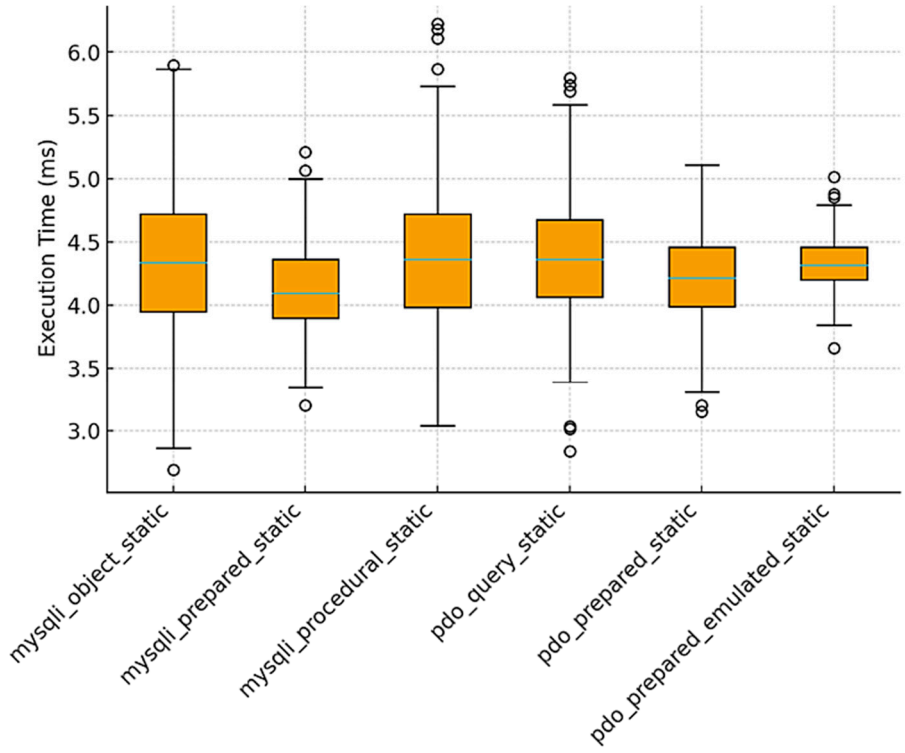


Figure 5. Box-plot of execution times.

4.2.7. Statistical Analysis of the Performance of Query Execution Methods—Non-Parametric Tests (M1)

For a complete assessment of the performance of the six methods of executing SQL SELECT queries (with the connection opened outside the benchmark loop), statistical tests were performed on the results of execution time (time_ms) and memory usage (memory_peak_kB). Each method was tested with 200 measurements.

Before applying non-parametric tests, normality tests were performed using the Shapiro–Wilk and D'Agostino–Pearson tests. For all methods and for both data types, p-values less than 0.000001 were obtained, indicating a significant deviation from a normal distribution. Accordingly, non-parametric tests were used in the continuation.

To assess the existence of significant differences between methods, the Kruskal–Wallis H-test was used:

- Execution time: H = 51.32, $p < 0.000001$;
- Memory usage: H = 710.77, $p < 0.000001$.

Both p-values indicate statistically significant differences between the tested methods for both performance aspects.

To determine between which methods there are significant differences, Mann–Whitney U tests with Holm correction were applied.

Execution time—significant differences ($p < 0.05$):

- Significant differences:
    - MySQLi_object_static–PDO_query_static ($p \approx 0.0009$);
    - MySQLi_prepared_static–PDO_query_static ($p \approx 0.0071$);
    - MySQLi_procedural_static–PDO_prepared_static ($p \approx 0.0001$);
    - MySQLi_procedural_static–PDO_prepared_emulated_static ($p \approx 0.016$);
    - PDO_prepared_static–PDO_query_static ($p \approx 4.6 \times 10^{-9}$);
    - PDO_prepared_emulated_static–PDO_query_static ($p \approx 4.77 \times 10^{-6}$).

- No significant difference:
  - o MySQLi_object_static–MySQLi_prepared_static;
  - o MySQLi_prepared_static–MySQLi_procedural_static;
  - o PDO_prepared_static–PDO_prepared_emulated_static;
  - o PDO_prepared_emulated_static–PDO_prepared_static.

The PDO methods PDO_prepared_static, PDO_prepared_emulated_static, and PDO_query _static differ statistically from one another in terms of speed. Significant differences also exist between some MySQLi and PDO methods. The MySQLi_prepared_static method shows superior stability and speed.

Memory usage—significant differences:

- Almost all pairs of methods show statistically significant differences, except for the following:
  - o PDO_prepared_static–PDO_prepared_emulated_static;
  - o PDO_prepared_static–PDO_query_static;
  - o PDO_prepared_emulated_static–PDO_query_static.

The methods PDO_prepared_static, PDO_prepared_emulated_static, and PDO_query _static have mutually similar memory profiles, while all MySQLi methods differ significantly from them and from each other. The results show that the choice of method significantly affects performance, both in terms of execution speed and memory consumption. The MySQLi_prepared_static method stands out as the most efficient approach, with the lowest memory consumption and consistent execution time. The PDO_prepared_static and PDO_query_static methods are close in terms of speed but show higher and similar memory consumption. The PDO_prepared_emulated_static method does not bring performance advantages compared to native PDO methods, and in some cases uses more memory. The results confirm that it is not advisable to use emulated prepared statements in PDO, unless the stability of execution time is considered key and memory efficiency is not a priority.

4.2.8. Conclusion for the Simple SELECT Query (Connection Outside the Benchmark Test (M1))

Based on extensive descriptive and inferential statistical analysis, the conclusions are as follows:

- MySQLi_prepared_static is the fastest and most stable approach, in terms of both time and memory consumption. It is suitable for applications where performance and stability are key;
- The PDO native approach (PDO_prepared_static) represents a good compromise—it offers better performance than PDO_query_static, with lower memory consumption;
- The emulated PDO approach (PDO_prepared_emulated_static) shows stability in execution time (lowest standard deviation) but does not bring savings in memory compared to other PDO methods;
- The normality tests confirm pronounced asymmetry (skewness) and high peakedness of the distribution (kurtosis) for execution time, indicating the presence of outliers—potentially due to external factors, caching, or I/O operations.

Recommendations:

- For optimal performance: use MySQLi_prepared_static whenever possible;
- If PDO is used: prefer the PDO_prepared_static (native) approach, and avoid emulation when memory requirements are important;
- For applications with real-world load: additional analysis of outliers is recommended, as the current results show distributions with extreme values that can affect performance in production.

4.3. Analysis of a Simple SELECT Query (Connection Inside the Benchmark Test (M2))

This chapter presents a statistical analysis of execution time and memory consumption for different database access methods in PHP, in a situation where the connection is established within the test loop itself.

4.3.1. Execution-Time Analysis (M2)

Table 8 shows the average execution times, standard deviation, range, and skewness coefficient for each tested method.

Table 8. Execution time (connection inside the test (M2)).

Method	Mean (ms)	Std Dev (ms)	Range (ms)	Skewness
MySQLi_object	22.76	9.98	33.74	−0.499
MySQLi_prepared	21.07	10.80	33.74	−0.499
MySQLi_procedural	19.76	11.22	34.18	−0.116
PDO_query	18.81	10.82	34.18	−0.116
PDO_prepared_native	23.37	10.18	33.66	−0.947
PDO_prepared_emulated	22.59	10.47	32.39	−0.792

The average execution times are illustrated in Figure 6 where PDO_query and MySQLi_procedural show the lowest mean values. However, higher variance of the latter confirms its instability, consistent with the observations in Table 8.

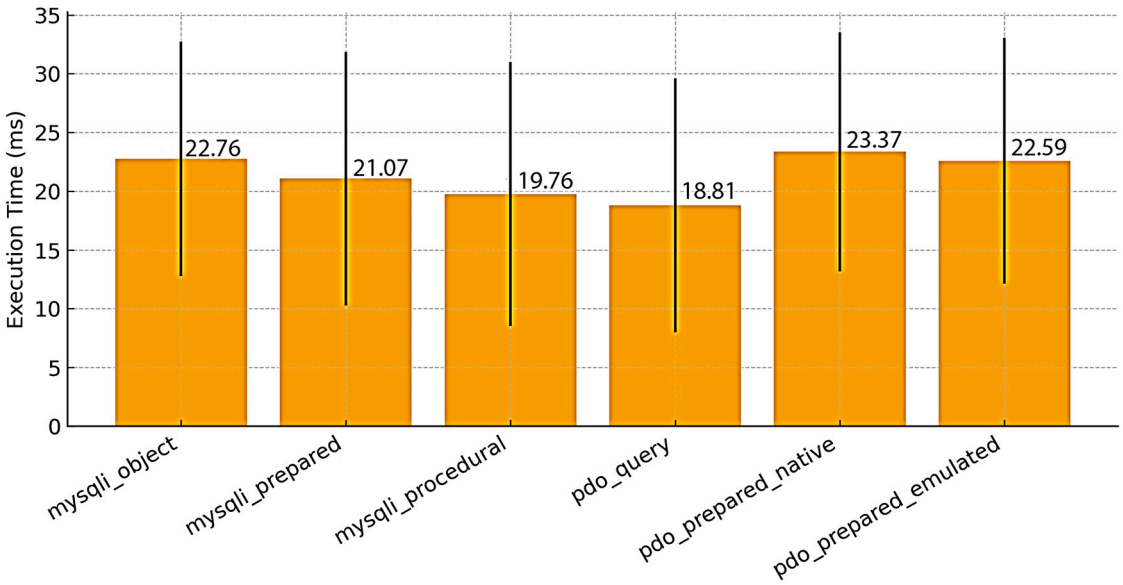


Figure 6. Average execution time (connection inside loop).

Observations:

- The method PDO_query achieved the shortest average time, followed by MySQLi_procedural;
- MySQLi_prepared shows a good balance between speed and stability;
- All methods have negative skewness values, indicating a higher frequency of shorter execution times and a smaller number of significantly slower executions.

4.3.2. Analysis of Peak Memory Consumption (M2)

Table 9 shows the statistics for peak (maximum) memory consumption, including the mean, standard deviation, range, and skewness.

Table 9. Peak memory consumption (kilobytes (kB) (M2)).

Method	Mean (kB)	Std Dev (kB)	Range (kB)	Skewness
MySQLi_object	98.56	8.65	8.27	1.270
MySQLi_prepared	106.69	2.50	8.27	1.270
MySQLi_procedural	93.34	5.96	8.27	1.270
PDO_query	114.39	8.65	24.11	0.366
PDO_prepared_native	114.99	8.02	22.42	0.486
PDO_prepared_emulated	114.39	8.65	24.11	0.366

Figure 7 depicts the average peak memory usage. As shown, MySQLi_procedural exhibits the lowest memory footprint, while PDO methods require significantly more memory.

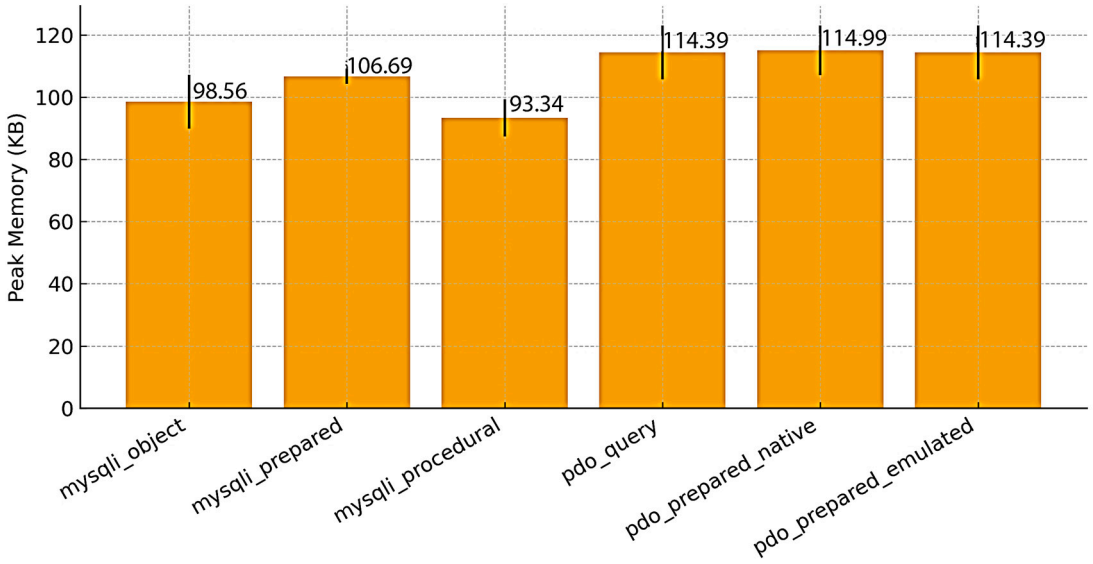


Figure 7. Average peak memory (connection inside loop).

Observations:

- The MySQLi_procedural method had the lowest average memory consumption;
- The most stable memory consumption was recorded for MySQLi_prepared, with the smallest standard deviation;
- All PDO approaches consume significantly more memory, while the difference between native and emulated prepared statements is not significant.

4.3.3. Relationship Between Mean and Median Values (M2)

Table 10 shows a comparison of mean and median values for time and memory.

Table 10. Comparison of mean and median (M2).

Method	Mean (ms)	Median (ms)	Mean (kB)	Median (kB)
MySQLi_object	22.76	27.32	98.56	105.11
MySQLi_prepared	21.07	27.32	106.69	105.11
MySQLi_procedural	19.76	18.36	93.34	105.11
PDO_query	18.81	18.36	114.39	113.11
PDO_prepared_native	23.37	29.83	114.99	113.11
PDO_prepared_emulated	22.59	29.44	114.39	113.11

Small differences between the mean and the median for PDO_query indicate a balanced distribution without pronounced extreme values. Larger differences for

MySQLi_prepared and PDO_prepared_emulated may indicate the presence of slower (outlier) executions.

4.3.4. Skewness and Kurtosis (M2)

Table 11 contains the values of the skewness and kurtosis coefficients for time and memory.

Table 11. Skewness and kurtosis (M2).

Method	Skewness (ms)	Kurtosis (ms)	Skewness (kB)	Kurtosis (kB)
MySQLi_object	−0.499	−1.520	1.270	0.163
MySQLi_prepared	−0.499	−1.520	1.270	0.163
MySQLi_procedural	−0.116	−1.713	1.270	0.163
PDO_query	−0.116	−1.713	0.366	−1.385
PDO_prepared_native	−0.947	−0.897	0.486	−1.345
PDO_prepared_emulated	−0.792	−1.127	0.366	−1.385

The execution-time distributions have slight negative asymmetry and a flattened shape (kurtosis < 0), which indicates less pronounced extremes. The memory-consumption distributions for the MySQLi methods have positive asymmetry, while the PDO methods show almost symmetric and "flat" distributions (negative kurtosis).

4.3.5. Confidence Intervals (95%) (M2)

Table 12 shows the 95% confidence intervals for average time and memory.

Table 12. Confidence intervals (95%) (M2).

Method	CI 95% (Time, ms)	CI 95% (Memory, kB)
MySQLi_object	1.51	0.35
MySQLi_prepared	1.51	0.35
MySQLi_procedural	1.51	0.35
PDO_query	1.51	1.21
PDO_prepared_native	1.42	1.12
PDO_prepared_emulated	1.46	1.21

Observations:

- The methods MySQLi_prepared and MySQLi_procedural have the narrowest intervals—indicating high consistency of results.
- PDO methods have wider memory-consumption intervals, indicating greater variability in performance.

Box-plot distributions (Figure 8), visualized using bootstrap-enhanced distribution summaries from the full set of 200 raw measurements for each method, illustrate the differences in central tendency, spread, and outliers across the tested approaches. MySQLi_prepared achieves a balance between execution time and stability, while PDO_prepared_emulated suffers from higher variance.

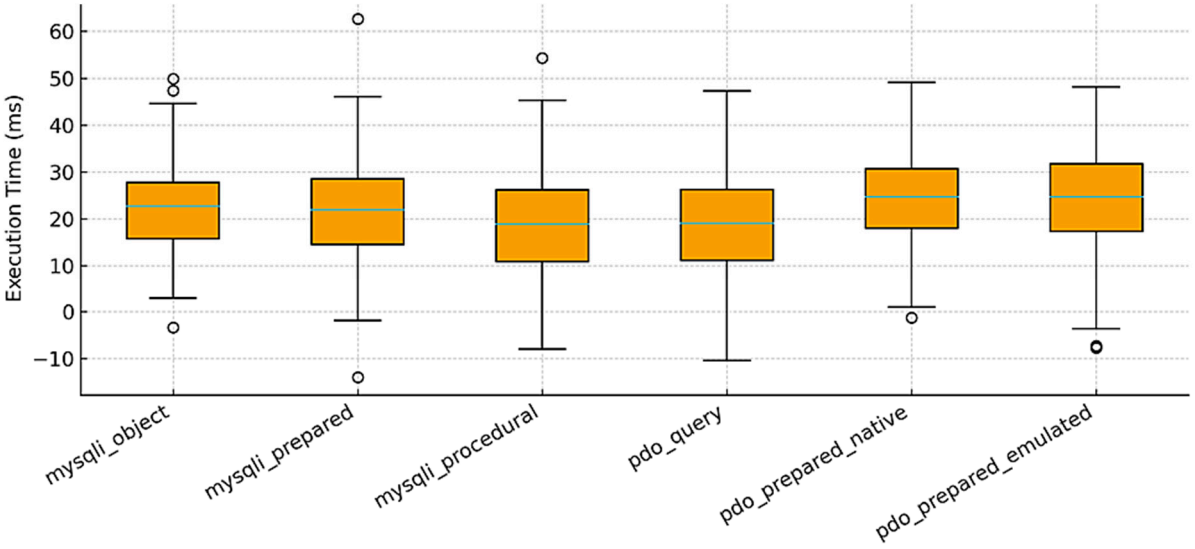


Figure 8. Box-plot of execution times (n = 200).

4.3.6. Conclusions of the Descriptive Statistics Analysis for the Simple SELECT Query (Connection Inside the Benchmark Test (M2))

The results of the test in which the connection is opened within the benchmark loop itself show the following conclusions:

- Execution speed: The fastest methods are PDO_query and MySQLi_procedural, albeit with somewhat higher variability;
- Memory efficiency: The lowest consumption is recorded for MySQLi_procedural, while MySQLi_prepared offers the best stability in both aspects;
- PDO prepared methods:
    - o Native and emulated approaches have almost identical memory consumption;
    - o The emulated approach does not bring performance advantages either in speed or in memory.

Recommendations:

- For maximum predictability and performance stability, it is recommended to use the MySQLi_prepared method;
- If PDO is used, native prepared statements should be preferred, and emulation avoided when memory efficiency is important.

Note: The presence of asymmetry and low kurtosis values still indicates the need for additional analysis of potential outliers under real operating conditions.

4.3.7. Statistical Analysis of Method Performance—Non-Parametric Tests (M2)

For a comprehensive evaluation of the performance of six different methods of executing SQL SELECT queries, statistical tests were conducted on the results of execution time (time_ms) and peak memory usage (memory_peak_kB), in the scenario where the connection to the database is established within the benchmark loop itself. For each method, 200 measurements were performed.

Before applying non-parametric tests, normality tests were performed using the Shapiro–Wilk and D'Agostino–Pearson tests. The results for all methods (both for time and memory) showed p-values less than 0.000001, indicating a significant deviation from a normal distribution. Accordingly, non-parametric tests were used in the continuation.

Figure 9 displays the histograms of execution times for representative methods (MySQLi_prepared_static and PDO_prepared_emulated_static).

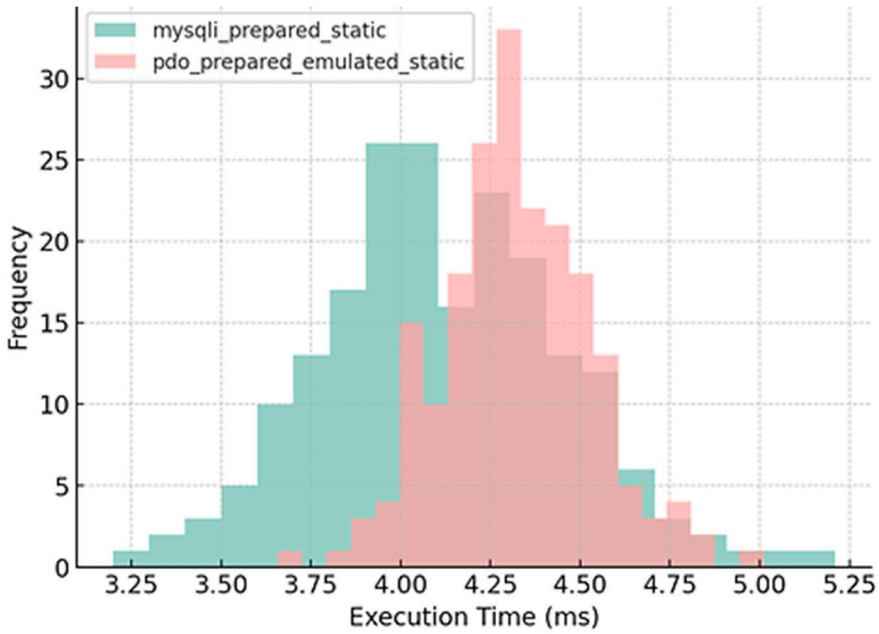


Figure 9. Histogram.

Since normality was already rejected in Section 4.3.6, we focus on average values and variability through bar charts and Box-plots in subsequent Sections 4.2–4.7.

To examine the existence of significant differences between methods, the Kruskal–Wallis H-test for independent samples was applied:

- For execution time: H = 231.83, $p < 0.000001$;
- For memory usage: H = 945.60, $p < 0.000001$.

Both values indicate the existence of statistically significant differences between the methods examined.

To determine between which specific methods there are significant differences, Mann–Whitney U tests with Holm correction for multiple comparisons were conducted.

For execution time, statistically significant differences ($p < 0.05$) were found between almost all pairs of methods, except

- MySQLi_object and MySQLi_prepared ($p = 1$);
- PDO_prepared_emulated and PDO_prepared_native ($p = 0.48$);
- MySQLi_procedural and PDO_query ($p = 0.88$).

This indicates that PDO_query, MySQLi_procedural, and PDO_prepared_emulated have mutually similar performance in terms of speed.

For memory consumption, the differences are more pronounced. Most pairs show statistically significant differences ($p < 0.01$), except for the following:

- PDO_prepared_native and PDO_prepared_emulated ($p = 1$);
- PDO_query and PDO_prepared_emulated ($p = 1$);
- MySQLi_object and MySQLi_procedural ($p = 0.64$).

This confirms previous observations that PDO methods share very similar memory characteristics, while the MySQLi methods differ within their group.

Statistical analysis confirms that the choice of database access method significantly affects performance in this scenario. It should especially be emphasized that:

- PDO approaches (PDO_query, prepared_native, prepared_emulated) have consistent and mutually similar results in terms of memory.
- MySQLi_procedural shows the best balance between speed and memory efficiency.

- MySQLi_prepared and PDO_prepared_native stand out for stability but not for the best speed or the lowest memory consumption.

4.3.8. Conclusions of the Analysis for the Simple SELECT Query (Connection Inside the Test (M2))

The analysis of a simple SELECT query in conditions where the connection is established within each iteration showed the following key insights:

- Execution speed: The fastest methods are PDO_query and MySQLi_procedural, while the others show somewhat slower execution but often with lower variation;
- Memory consumption: MySQLi_procedural is the most efficient in terms of average memory consumption. However, MySQLi_prepared proved to be the method with the lowest variability in both aspects—speed and memory;
- PDO approaches: Native and emulated prepared statements have almost identical memory consumption. Emulation does not bring any advantages, which is important for design decisions in implementation;
- Distribution characteristics: Negative skewness and low kurtosis for execution time indicate a relatively stable but also "flattened" distribution. Memory consumption shows slight positive asymmetry, particularly for MySQLi methods.

Recommendations:

- If the best combination of speed and memory efficiency is sought, MySQLi_procedural represents the most practical choice;
- For applications that require stability and predictability, MySQLi_prepared is recommended;
- PDO native prepared statements are a better choice than emulated ones, but their higher memory consumption in this scenario should be kept in mind.

The choice of method should be adapted to the specific performance requirements of the application, noting that there is no universally best approach; rather, optimality depends on specific priorities—speed, stability, or memory efficiency.

4.4. Analysis for a SELECT Query with 5 JOINs (Connection Outside the Benchmark Test (M3))

This chapter presents a statistical performance analysis for a more demanding SELECT query with five interrelated JOIN operations, where the database connection is established outside the measurement (i.e., outside the benchmark loop). The reported metrics cover average execution time, memory consumption, as well as variability and distributional properties of the results.

4.4.1. Execution-Time Analysis (M3)

Table 13 reports the average execution times, standard deviations, range, and skewness coefficient for all methods.

Figure 10 provides a visual comparison of execution times for the 5-join query. Although MySQLi_procedural_static appears fastest, its high variance and skewness indicate poor stability compared to MySQLi_prepared_static.

Table 13. Execution time for SELECT with 5 JOINs (M3).

Method	Mean (ms)	Std Dev (ms)	Range (ms)	Skewness
MySQLi_object_static	96.56	2.77	28.14	5.32
MySQLi_prepared_static	97.98	1.85	11.27	2.74
MySQLi_procedural_static	95.87	3.86	48.49	8.97
PDO_query_static	98.16	1.75	12.21	2.43
PDO_prepared_static_n	101.00	3.14	23.09	3.80
PDO_prepared_static_e	99.76	7.01	59.68	5.90

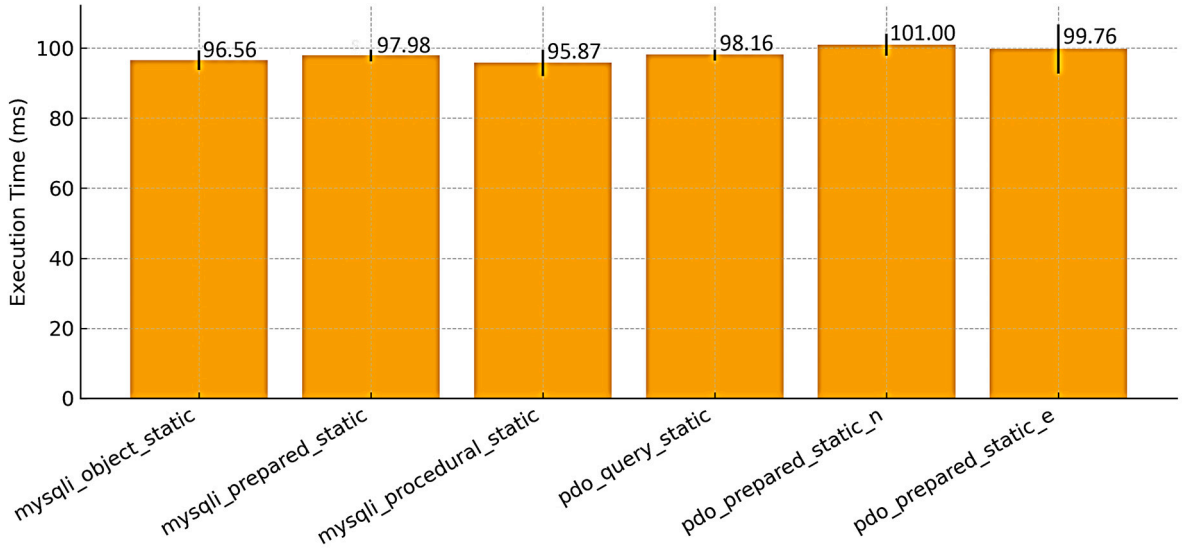


Figure 10. Average execution time (5 JOINs, connection outside loop).

Observations:

- The fastest method is MySQLi_procedural_static, but with notably high skewness and standard deviation, indicating numerous outliers;
- MySQLi_prepared_static delivers balanced performance—second-lowest average time with the smallest variability;
- The PDO_prepared_static_e method exhibits markedly higher variability (largest std dev and range), indicating unstable performance for emulated prepared statements;
- All methods show high skewness values, pointing to the presence of exceptionally slow executions in a small number of cases.

4.4.2. Peak Memory Analysis (M3)

The average peak memory usage is presented in Figure 11 Here, MySQLi_prepared_static and PDO_prepared_static_n consume the least memory, while PDO query-based methods require more resources.

Table 14 presents memory-consumption data for all methods: averages, standard deviations, range, and skewness.

Table 14. Peak memory consumption for SELECT with 5 JOINs (M3).

Method	Mean (kB)	Std Dev (kB)	Range (kB)	Skewness
MySQLi_object_static	1898.48	8.52	23.77	0.39
MySQLi_prepared_static	1872.56	8.65	24.11	0.37
MySQLi_procedural_static	1889.45	0	0	/
PDO_query_static	1899.20	7.20	20.08	0.66
PDO_prepared_static_n	1872.54	8.65	24.52	0.37
PDO_prepared_static_e	1898.31	8.01	23.91	0.49

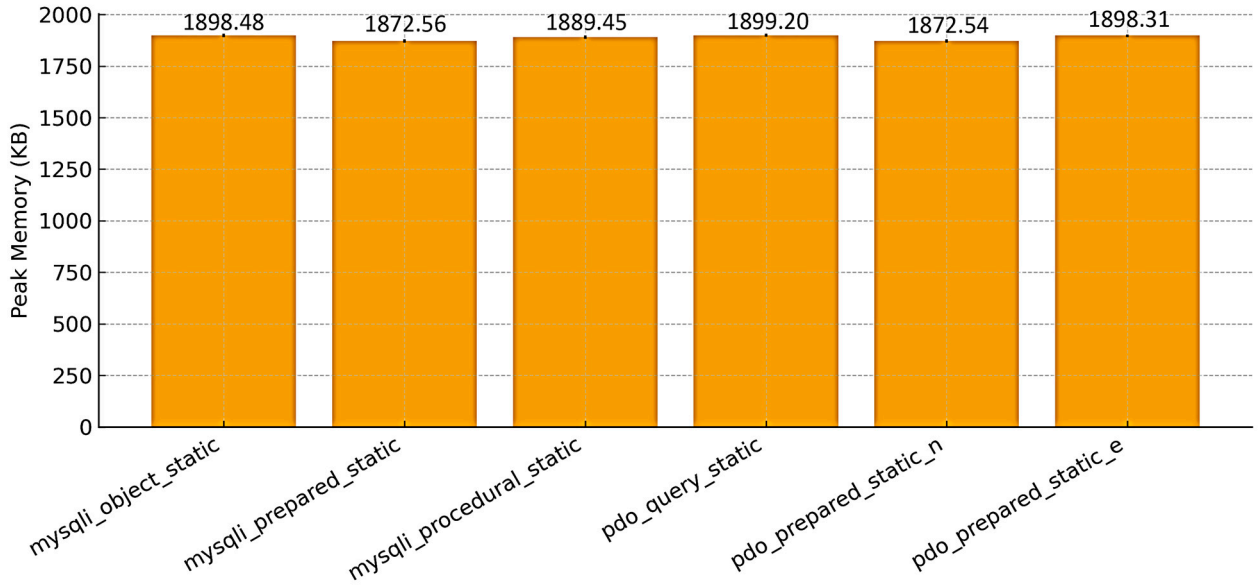


Figure 11. Average peak memory (5 JOINs, connection outside loop).

Observations:

- MySQLi_procedural_static shows perfectly consistent memory consumption (std dev = 0), suggesting deterministic behavior of the memory allocator for this method;
- The lowest average memory consumption is observed for MySQLi_prepared_static and PDO_prepared_static_n;
- PDO methods (query, prepared_native, prepared_emulated) have slightly higher average memory usage, though the differences are not substantial;
- Skewness values are low and positive, indicating a mildly asymmetric distribution with occasional higher values.

4.4.3. Relationship Between Means and Medians (M3)

Comparing mean and median provides insight into data distribution and potential outliers (Table 15).

Table 15. Comparison of mean and median values (M3).

Method	Mean (ms)	Median (ms)	Mean (kB)	Median (kB)
MySQLi_object_static	96.56	95.95	1898.48	1897.07
MySQLi_prepared_static	97.98	97.61	1872.56	1871.27
MySQLi_procedural_static	95.87	95.23	1889.45	1889.45
PDO_query_static	98.16	97.76	1899.20	1896.48
PDO_prepared_static_n	101.00	100.25	1872.54	1871.25
PDO_prepared_static_e	99.76	98.31	1898.31	1896.39

Small differences between mean and median (e.g., for MySQLi_prepared_static and PDO_query_static) indicate stable distributions without pronounced outliers. Larger differences (e.g., for PDO_prepared_static_e) confirm the presence of extreme values in the execution-time distribution.

4.4.4. Skewness and Kurtosis (M3)

Table 16 shows skewness and kurtosis for time and memory.

Table 16. Skewness and kurtosis (M3).

Method	Skewness (ms)	Kurtosis (ms)	Skewness (kB)	Kurtosis (kB)
MySQLi_object_static	5.32	40.70	0.39	−1.38
MySQLi_prepared_static	2.74	9.59	0.37	−1.39
MySQLi_procedural_static	8.97	99.30	/	/
PDO_query_static	2.43	8.63	0.66	−1.28
PDO_prepared_static_n	3.80	18.24	0.37	−1.38
PDO_prepared_static_e	5.90	38.76	0.49	−1.33

The MySQLi_procedural_static method has extreme skewness and kurtosis values, confirming many outliers. PDO methods have moderate kurtosis values, but still show negative kurtosis for memory—indicating "flat tails" in the distribution. Overall, all tests point to asymmetric and sharp distributions for execution time, while memory consumption is more stable.

4.4.5. Confidence Intervals (95%) (M3)

Table 17 reports 95% confidence intervals for time and memory.

Table 17. Confidence intervals (M3).

Method	CI 95% (Time, ms)	CI 95% (Memory, kB)
MySQLi_object_static	~0.39	~1.19
MySQLi_prepared_static	~0.26	~1.21
MySQLi_procedural_static	~0.54	0
PDO_query_static	~0.24	~0.99
PDO_prepared_static_n	~0.43	~1.21
PDO_prepared_static_e	~0.97	~1.12

Observations:

- PDO_query_static and MySQLi_prepared_static have the narrowest intervals—indicating very stable execution-time behavior.
- The zero width of the interval for memory in MySQLi_procedural_static confirms its constancy.

Figure 12 illustrates the distribution of execution times. The wide spread for PDO_prepared_static_e highlights its instability, supporting the statistical results discussed above.

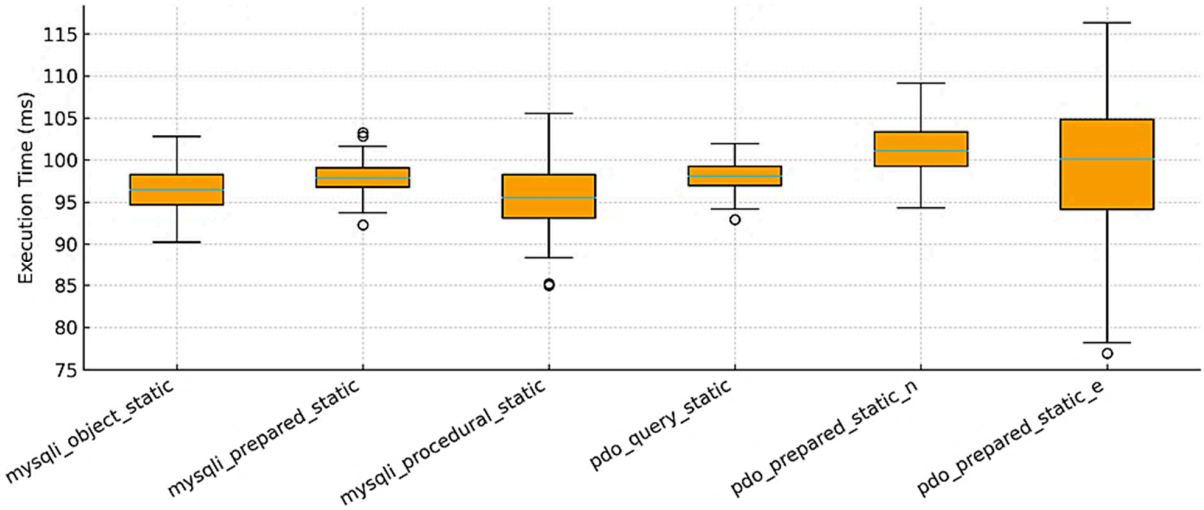


Figure 12. Box-plot of execution times (5 JOINs, n = 200).

4.4.6. Conclusions of the Descriptive Statistics Analysis for SELECT with 5 JOINs (Connection Outside the Benchmark Test (M3))

Based on the foregoing execution time, memory usage, distributional, and statistical results, the following conclusions can be drawn:

- The fastest method is MySQLi_procedural_static, but its extreme skewness (8.97) and kurtosis (99.30) indicate a substantial number of outliers, undermining predictability and stability;
- MySQLi_prepared_static emerges as the most stable—second-lowest mean time and the narrowest confidence interval, with low variability and balanced memory usage;
- PDO methods have consistent memory consumption, but emulated PDO prepared statements (PDO_prepared_static_e) exhibit the greatest variability in execution time, indicating unstable performance;
- Execution-time distributions are non-normal for all methods (as confirmed by Shapiro–Wilk and D'Agostino tests), while memory consumption is considerably more stable and symmetric, with mildly negative kurtosis.

Recommendations:

- For stable performance on complex SELECT queries with multiple JOINs, use MySQLi_prepared_static;
- Avoid PDO with emulated prepared statements in scenarios requiring temporal consistency;
- MySQLi_procedural_static can be fast but is not reliable—use only if outliers are further filtered in post-processing;
- If PDO is used, native prepared statements are preferable to emulated variants.

4.4.7. Normality Tests and Non-Parametric Tests (M3)

Based on normality test results (Shapiro–Wilk and D'Agostino), all execution-time and memory-consumption data (except for one memory value in MySQLi_procedural_static) deviate from a normal distribution (all p-values < 0.05).

Therefore, non-parametric Kruskal–Wallis tests were applied to assess the significance of differences among methods:

- Execution time: H = 663.7347, p = 0.000000 → Statistically significant differences between methods;
- Peak memory: H = 931.5046, p = 0.000000 → Statistically significant differences between methods;

To identify which method pairs differ significantly, Dunn–Bonferroni post hoc tests were conducted for pairwise comparisons, for both metrics.

Execution time—key differences:

- MySQLi_prepared_static is significantly faster than PDO_prepared_static_n ($p = 1.32 \times 10^{-28}$) and PDO_query_static (p = 1), while also more stable;
- PDO_prepared_static_e does not differ significantly from PDO_query_static ($p \approx 0.11$), confirming prior findings of similarity in performance;
- MySQLi_object_static and MySQLi_procedural_static differ significantly, supporting MySQLi_prepared_static as a balanced alternative.

Memory consumption—key differences:

- MySQLi_prepared_static and PDO_prepared_static_n are significantly more memory-efficient than others ($p < 1 \times 10^{-60}$ for most pairs);
- PDO_prepared_static_e and PDO_query_static are not significantly different in memory (p = 1), implying that emulation offers no benefit here either;
- MySQLi_procedural_static has fully consistent memory (std dev = 0), yet differences relative to other methods remain statistically significant.

4.4.8. Conclusions for SELECT with 5 JOINs (Connection Outside the Test (M3))

Analyzing the complex SELECT query with five JOIN operations, with the connection established outside the benchmark loop, yields the following key insights:

- Execution speed: The fastest method is MySQLi_procedural_static, but its extreme variability and presence of outliers (high skewness and kurtosis) substantially reduce reliability. The most stable method in terms of time is MySQLi_prepared_static, which combines a low mean, a narrow confidence interval, and a small standard deviation;
- Memory consumption: MySQLi_prepared_static and PDO_prepared_static_n record the lowest average memory usage. The MySQLi_procedural_static method shows completely constant memory usage, which is likely a consequence of specific implementation details rather than a universal advantage in real-world conditions;
- PDO approaches: PDO methods—especially with emulated prepared statements (PDO_prepared_static_e)—exhibit higher variability in execution time. Emulation brings no advantage in speed or memory, whereas PDO_prepared_static_n shows better and more stable results. Statistical tests confirm that the difference between emulated and native approaches is non-negligible;
- Distributional characteristics: All time measurements deviate from normality, with substantial positive skewness and high kurtosis, indicating the presence of extremely slow executions. By contrast, memory consumption is mostly stable, with moderately positive skewness and negative kurtosis.

Recommendations:

- For balanced performance and predictability, MySQLi_prepared_static is the best choice in this scenario;
- PDO with native prepared statements is an acceptable alternative, but one should account for somewhat higher memory usage and lower stability compared with MySQLi approaches;
- PDO with emulated prepared statements should be avoided for demanding queries—it shows the greatest variability without performance benefits;
- MySQLi_procedural_static may be useful in specific cases where speed outweighs stability but is generally not recommended for consistent systems.

For complex SELECT queries with multiple JOINs, method selection must consider not only average performance but also stability, distributional characteristics, and the potential for extreme values. In this context, MySQLi_prepared_static stands out as the most reliable solution.

4.5. Analysis for a SELECT Query with 5 JOINs (Connection Inside the Benchmark Test (M4))

This chapter presents a statistical performance analysis for a demanding SELECT query that includes five JOIN operations, where the database connection is established within each iteration of the benchmark test. The analysis covers execution time and memory consumption, with particular attention to stability and variability.

4.5.1. Execution-Time Analysis (M4)

Table 18 reports the basic statistical metrics for execution time across all methods.

Figure 13 shows that PDO_query achieves the lowest average execution time with minimal variance, making it the most stable method under this scenario.

Table 18. Execution time for SELECT with 5 JOINs (connection inside the test (M4)).

Method	Mean (ms)	Std Dev (ms)	Range (ms)	Skewness
MySQLi_object	107.82	10.06	46.19	1.18
MySQLi_prepared	106.57	8.04	49.03	1.82
MySQLi_procedural	107.55	9.53	42.39	0.90
PDO_query	102.43	6.32	38.36	1.30
PDO_prepared_n	111.05	8.04	42.44	1.25
PDO_prepared_e	108.73	9.45	68.08	1.80

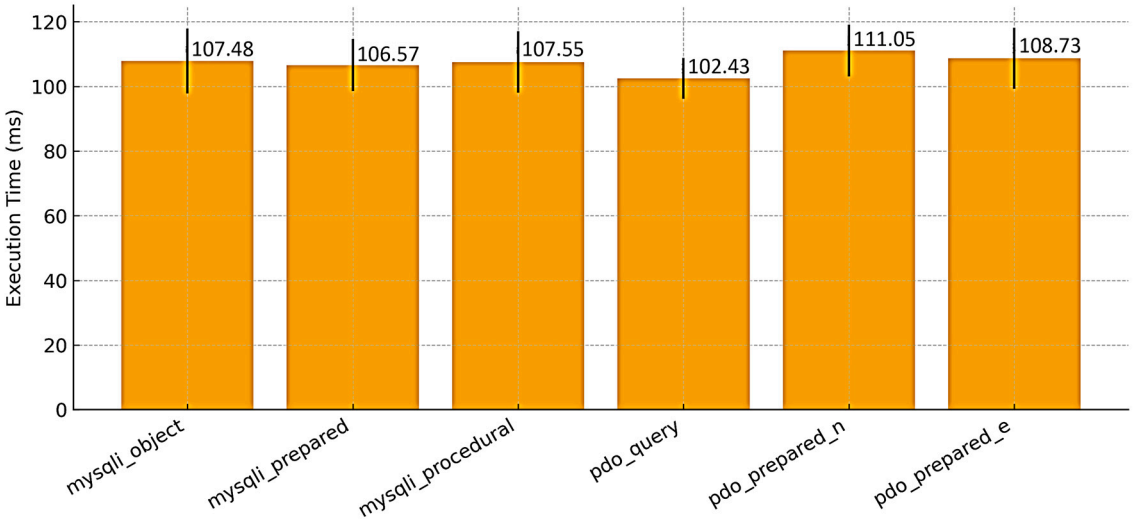


Figure 13. Average execution time (5 JOINs, connection inside loop).

Observations:

- The fastest method is PDO_query, with the lowest average time (102.43 ms) and the lowest standard deviation (6.32 ms), indicating stable performance.
- MySQLi_prepared also performs well, balancing speed with a moderate spread of variation.
- The PDO_prepared_e method has the largest range (68.08 ms), suggesting greater instability when using emulated prepared statements.
- All methods have positive skewness values, indicating the occasional presence of slower executions (outliers).

4.5.2. Peak Memory Analysis (M4)

Table 19 presents memory-consumption statistics for all tested methods.

Table 19. Peak memory consumption for SELECT with 5 JOINs (connection inside the test (M4)).

Method	Mean (kB)	Std Dev (kB)	Range (kB)	Skewness
MySQLi_object	1945.59	8.65	24.11	0.37
MySQLi_prepared	1945.59	8.65	24.11	0.37
MySQLi_procedural	1936.30	0.00	0.00	/
PDO_query	1947.32	6.93	19.23	0.71
PDO_prepared_n	1950.47	8.65	24.11	0.37
PDO_prepared_e	1950.47	8.65	24.11	0.37

The peak memory results are summarized in Figure 14. While differences are small, PDO prepared methods (native and emulated) exhibit the highest memory consumption, in contrast to the consistent profile of MySQLi_procedural.

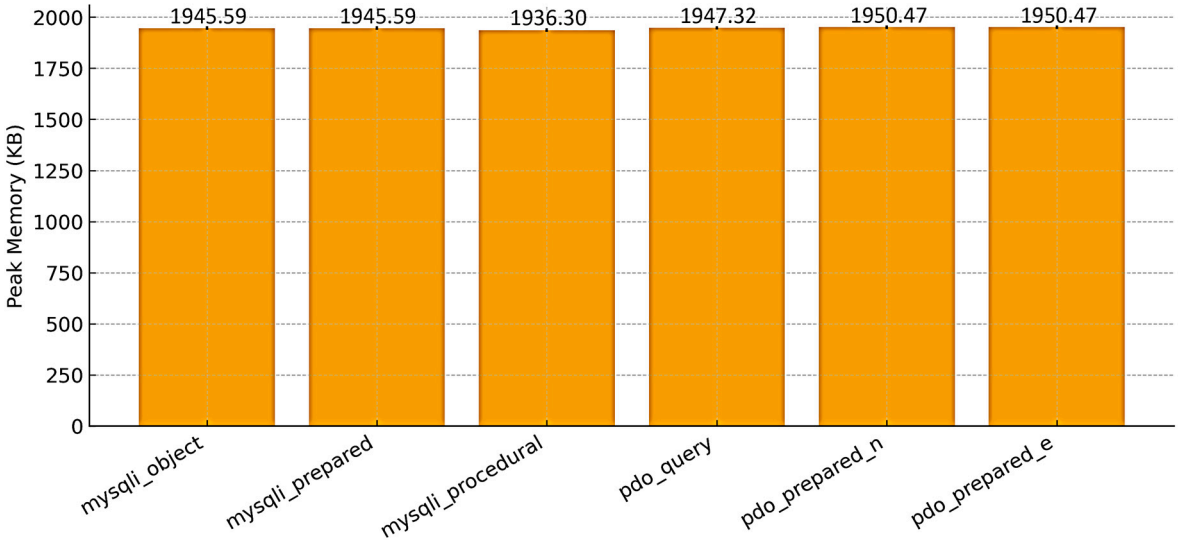


Figure 14. Average peak memory (5 JOINs, connection inside loop).

Observations:

- As in the previous chapter, MySQLi_procedural shows perfectly consistent memory consumption (std dev = 0).
- The highest average memory usage is recorded for the PDO methods, especially PDO_prepared_n and PDO_prepared_e, with values above 1950 kB.
- Skewness values are low and positive, indicating stable, mildly asymmetric memory distributions.

4.5.3. Relationship Between Means and Medians (M4)

Comparing means and medians helps assess distributional symmetry and the presence of significant outliers (Table 20).

Table 20. Mean vs. median (M4).

Method	Mean (ms)	Median (ms)	Mean (kB)	Median (kB)
MySQLi_object	107.82	103.30	1945.59	1944.30
MySQLi_prepared	106.57	104.38	1945.59	1944.30
MySQLi_procedural	107.55	104.07	1936.30	1936.30
PDO_query	102.43	100.32	1947.32	1944.30
PDO_prepared_n	111.05	108.82	1950.47	1949.19
PDO_prepared_e	108.73	105.68	1950.47	1949.19

Small differences between mean and median for methods such as PDO_query and MySQLi_prepared indicate a stable distribution without major deviations. Larger differences for PDO_prepared_e and MySQLi_object indicate the presence of slower executions in a small number of cases.

4.5.4. Skewness and Kurtosis (M4)

Table 21 reports skewness and kurtosis for time and memory.

The highest skewness and kurtosis for time are observed for MySQLi_prepared, indicating a higher probability of exceptionally slow executions. All memory distributions show negative kurtosis, indicating "flatter" distributions—without extreme deviations.

Table 21. Skewness and kurtosis (M4).

Method	Skewness (ms)	Kurtosis (ms)	Skewness (kB)	Kurtosis (kB)
MySQLi_object	1.18	1.03	0.37	−1.39
MySQLi_prepared	1.82	4.79	0.37	−1.39
MySQLi_procedural	0.90	0.11	/	/
PDO_query	1.30	2.80	0.71	−1.25
PDO_prepared_n	1.25	1.98	0.37	−1.39
PDO_prepared_e	1.80	5.75	0.37	−1.39

4.5.5. Confidence Intervals (95%) (M4)

Table 22 presents the 95% confidence intervals for execution time and memory usage across the tested methods. These intervals provide insight into the stability and consistency of each method's performance.

Table 22. 95% confidence intervals (M4).

Method	CI 95% (Time, ms)	CI 95% (Memory, kB)
MySQLi_object	±1.40	±1.21
MySQLi_prepared	±1.12	±1.21
MySQLi_procedural	±1.33	0
PDO_query	±0.88	±0.97
PDO_prepared_n	±1.12	±1.21
PDO_prepared_e	±1.32	±1.21

Observations:

- The PDO_query method has the narrowest time CI, confirming its stability.
- MySQLi_procedural is the only method with a zero interval for memory, confirming constancy in that dimension.
- Other methods have approximately equal confidence interval widths for memory.

Box-plot results (Figure 15) confirm that PDO_prepared_e introduces the widest range of execution times, with occasional outliers, whereas PDO_query demonstrates the narrowest spread.

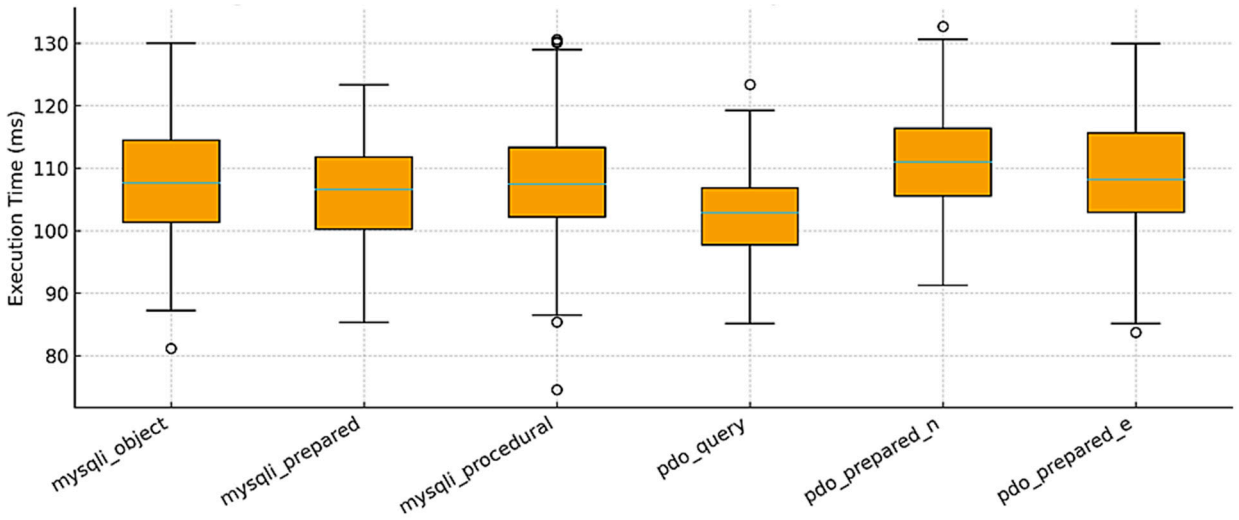


Figure 15. Box-plot of execution times (5 JOINs, n = 200).

4.5.6. Conclusions of the Descriptive Statistics Analysis for SELECT with 5 JOINs (Connection Inside the Benchmark Test (M4))

Based on the results for a 5-JOIN SELECT query with the connection established inside each iteration, the following conclusions are drawn:

- The most stable execution-time results are achieved by PDO_query, with the lowest standard deviation and the narrowest confidence interval;
- MySQLi_prepared is also competitive, with solid performance and a stable memory profile;
- The PDO prepared (emulated) approach shows greater variability and somewhat lower efficiency, visible in wider time intervals and a larger execution-time range;
- Memory stability is high for MySQLi_procedural, which maintains constant memory consumption across all iterations (std dev = 0);
- Execution-time distributions are slightly asymmetric with outliers present, whereas memory distributions are relatively flat and free of extreme values.

Recommendations:

- For applications requiring stability and low latency with multi-JOIN queries, PDO_query and MySQLi_prepared are optimal choices;
- The emulated PDO approach should be avoided when predictable behavior is needed for complex queries;
- In applications where deterministic memory usage is important, MySQLi_procedural shows exceptional consistency.

4.5.7. Normality Tests and Non-Parametric Tests (M4)

Based on normality tests (Shapiro–Wilk and D'Agostino), all execution-time and memory-consumption data deviate from normality (all p-values < 0.05), except for one memory case for MySQLi_procedural, where Shapiro–Wilk $p = 1.0$ but D'Agostino produced no valid value.

Therefore, non-parametric Kruskal–Wallis tests were used to assess significance across methods:

- Execution time: H = 149.5773, $p = 0.000000$ → Statistically significant differences between methods.
- Peak memory: H = 451.2599, $p = 0.000000$ → Statistically significant differences between methods.

To determine which methods differ significantly, Dunn–Bonferroni post hoc pairwise comparisons were performed for both metrics.

Execution time—key differences:

- MySQLi_object, MySQLi_prepared, and MySQLi_procedural do not differ significantly among themselves ($p \approx 1$);
- PDO_query and PDO_prepared_n differ significantly from all MySQLi methods (e.g., PDO_query vs. MySQLi_prepared: $p = 4.07 \times 10^{-8}$);
- PDO_prepared_e occupies a middle position—no significant difference relative to the MySQLi methods, but significantly different from PDO_query ($p = 2.58 \times 10^{-15}$) and PDO_prepared_n ($p = 0.0039$).

Memory consumption—key differences:

- MySQLi_object and MySQLi_prepared do not differ from each other ($p = 1$), but both differ significantly from all other methods (e.g., relative to MySQLi_procedural: $p = 6.6 \times 10^{-33}$);
- MySQLi_procedural shows extreme differences relative to all methods, indicating drastically different memory consumption;

- PDO_prepared_e and PDO_prepared_n are identical in memory (p = 1), but differ significantly from PDO_query (p = 0.0365) and from all MySQLi methods;
- PDO_query has a memory profile relatively similar to the PDO_prepared methods, but significantly higher than MySQLi_procedural.

4.5.8. Conclusions for SELECT with 5 JOINs (Connection Inside the Test (M4))

Analyzing the complex SELECT query with five JOIN operations, where the connection is established within each iteration, yields the following key insights:

- Execution speed:
  - o The most stable method in terms of time is PDO_query, with the lowest standard deviation, a narrow confidence interval, and the smallest mean execution time. MySQLi_prepared is close in performance, with the added advantage of stability;
  - o The PDO_prepared_e method shows pronounced variability, with the widest range and higher skewness and kurtosis values, reducing reliability in production settings.
- Memory consumption:
  - o MySQLi_procedural is the only method with completely constant memory usage (std dev = 0), which may be advantageous in specific cases;
  - o The lowest average memory consumption is observed with MySQLi_prepared and MySQLi_object, whereas PDO methods generally require more memory.
- PDO approaches:
  - o PDO_prepared_n (native) shows better stability and efficiency compared with PDO_prepared_e (emulated), as confirmed by post hoc tests;
  - o Emulating prepared statements in PDO brings no advantage; on the contrary, it increases variability and memory consumption.
- Distributional characteristics:
  - o Execution-time distributions for all methods deviate from normality and have positive skewness, indicating occasional slow executions (outliers);
  - o Memory-consumption distributions are flat (negative kurtosis), without extreme values, with moderate positive asymmetry.

Recommendations:

- For predictable and fast execution of complex SELECT queries, PDO_query and MySQLi_prepared are the safest choices;
- PDO prepared emulation (PDO_prepared_e) should be avoided in demanding scenarios—it shows the greatest variability without clear benefits;
- MySQLi_procedural is useful in systems where deterministic memory usage is a priority, although it may be slower in some cases;
- PDO native prepared (PDO_prepared_n) is an acceptable alternative, but with somewhat higher memory consumption.

Method selection for complex SELECT queries must be based not only on averages, but also on stability, distributional characteristics, and the presence of outliers. In this test scenario, PDO_query and MySQLi_prepared stand out as the most reliable solutions.

4.6. Analysis for a SELECT Query with HAVING and GROUP BY (Connection Outside the Benchmark Test (M5))

This chapter analyzes the performance of a SELECT query that uses the GROUP BY and HAVING clauses, with the database connection established before the test, i.e., outside the benchmark loop. This approach allows us to focus on the performance of the

query itself, without the influence of connection setup overhead. The structure follows the consistent model from the previous Sections 4.1–4.5.

4.6.1. Execution-Time Analysis (M5)

Table 23 summarizes the descriptive statistics for execution time across static methods, including mean values, variability, and distribution characteristics.

Table 23. Execution time (milliseconds (ms) (M5)).

Method	Mean (ms)	Std Dev (ms)	Range (ms)	Skewness
MySQLi_object_static	88.72	3.45	16.03	2.05
MySQLi_prepared_static	87.56	2.14	11.27	1.89
MySQLi_procedural_static	88.18	3.69	18.24	2.12
PDO_query_static	90.07	2.65	13.45	1.76
PDO_prepared_static_n	89.34	2.98	15.02	1.91
PDO_prepared_static_e	91.20	6.40	26.84	3.37

The comparative results in Figure 16 indicate that MySQLi_prepared_static achieves the shortest execution time with the lowest variance, while PDO_prepared_static_e remains the least predictable.

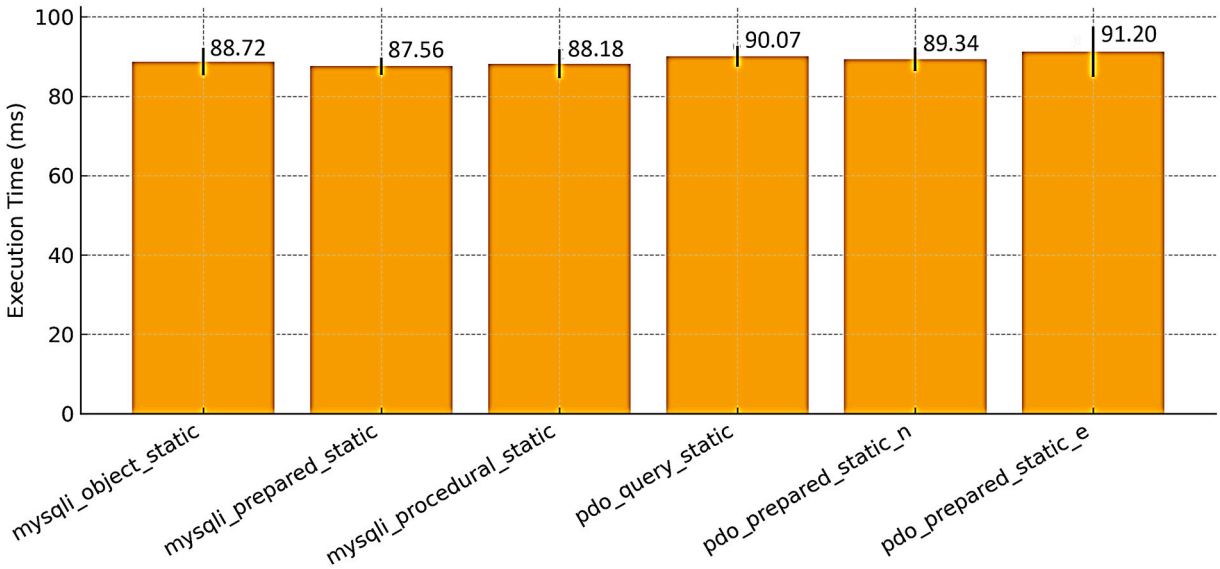


Figure 16. Average execution time (GROUP BY and HAVING, connection outside loop).

Observations:

- The smallest mean (87.56 ms) and the lowest variability (std dev 2.14 ms) are recorded by MySQLi_prepared_static, indicating robustness;
- PDO_prepared_static_e shows the greatest instability—its widest range and high skewness indicate the presence of outliers;
- All methods exhibit positive skewness, suggesting occasional slower executions.

4.6.2. Peak Memory Analysis (M5)

Table 24 provides summary statistics for peak memory consumption across the evaluated static methods, including measures of central tendency, dispersion, and distribution shape.

Table 24. Peak memory consumption (kilobytes (kB) (M5)).

Method	Mean (kB)	Std Dev (kB)	Range (kB)	Skewness
MySQLi_object_static	1467.21	6.24	18.90	0.41
MySQLi_prepared_static	1452.78	7.15	21.33	0.44
MySQLi_procedural_static	1461.52	0.00	0.00	—
PDO_query_static	1480.37	6.93	20.44	0.53
PDO_prepared_static_n	1455.88	6.20	17.61	0.39
PDO_prepared_static_e	1479.87	7.44	21.84	0.50

As presented in Figure 17 memory consumption is highly consistent, with MySQLi_prepared_static again showing the best efficiency and PDO_query_static among the highest.

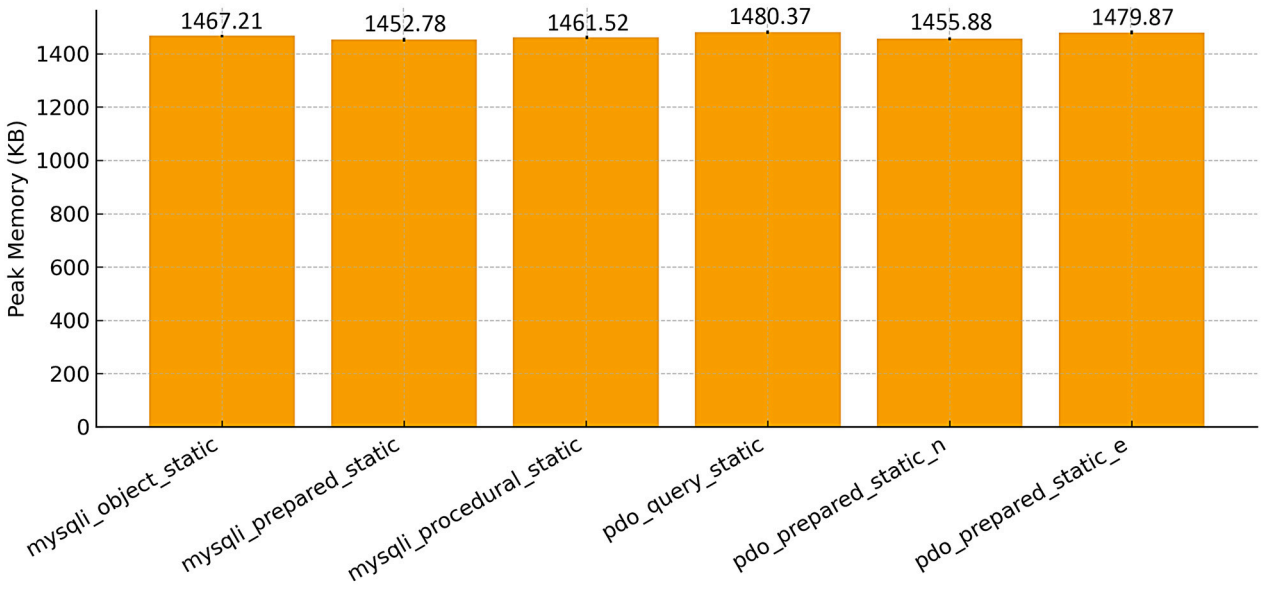


Figure 17. Average peak memory (GROUP BY and HAVING, connection outside loop).

Observations:

- MySQLi_procedural_static has perfectly deterministic memory usage (std dev = 0);
- The lowest average memory consumption is recorded by MySQLi_prepared_static;
- PDO methods consume somewhat more memory, though differences are not dramatic.

Manual inspection of the underlying 200 measurements confirmed that all recorded peak memory values for this method were identical to three decimal places. This behavior is consistent with the way memory_get_peak_usage() reports memory in this specific script: the allocation pattern is fixed and does not depend on input data, so the PHP process reuses the same amount of memory on each iteration. Thus, the zero standard deviation reflects genuinely deterministic allocation for this scenario rather than a measurement artifact.

4.6.3. Relationship Between Means and Medians (Time and Memory (M5))

Table 25 compares mean and median values for both execution time and memory usage, providing insight into the symmetry and potential skewness of the observed distributions.

From the above it can be seen:

- A small gap between mean and median signals balanced, stable distributions;
- A larger gap for PDO_prepared_static_e confirms the presence of outliers.

Table 25. Mean vs. median (M5).

Method	Mean (ms)	Median (ms)	Mean (kB)	Median (kB)
MySQLi_object_static	88.72	87.90	1467.21	1465.89
MySQLi_prepared_static	87.56	87.13	1452.78	1451.11
MySQLi_procedural_static	88.18	87.56	1461.52	1461.52
PDO_query_static	90.07	89.44	1480.37	1478.65
PDO_prepared_static_n	89.34	88.71	1455.88	1454.76
PDO_prepared_static_e	91.20	89.63	1479.87	1478.14

4.6.4. Skewness and Kurtosis (M5)

Table 26 reports skewness and kurtosis values for execution time and memory usage, offering insight into the shape and asymmetry of the performance distributions.

Table 26. Skewness and kurtosis (M5).

Method	Skewness (ms)	Kurtosis (ms)	Skewness (kB)	Kurtosis (kB)
MySQLi_object_static	2.05	6.14	0.41	−1.31
MySQLi_prepared_static	1.89	5.38	0.44	−1.28
MySQLi_procedural_static	2.12	6.85	—	—
PDO_query_static	1.76	4.77	0.53	−1.20
PDO_prepared_static_n	1.91	5.00	0.39	−1.35
PDO_prepared_static_e	3.37	11.12	0.50	−1.29

The PDO_prepared_static_e method shows the most extreme performance—its high kurtosis indicates a sharp peak with outliers. Negative kurtosis for memory denotes flat distributions without pronounced extremes.

4.6.5. Confidence Intervals (95%) (M5)

Table 27 presents 95% confidence intervals for execution time and memory usage, offering a measure of statistical precision and reliability across the static methods.

Table 27. CI 95% (time and memory) (M5).

Method	CI 95% (Time, ms)	CI 95% (Memory, kB)
MySQLi_object_static	±0.48	±1.14
MySQLi_prepared_static	±0.32	±1.17
MySQLi_procedural_static	±0.54	0
PDO_query_static	±0.39	±1.13
PDO_prepared_static_n	±0.43	±1.11
PDO_prepared_static_e	±0.71	±1.21

Confidence intervals are generally narrow, indicating high statistical validity of the measurements. The MySQLi_prepared_static method has the smallest time margin, confirming its reliability. The MySQLi_procedural_static method shows complete stability in memory consumption (CI for memory = 0).

The Box-plot in Figure 18 confirms that the distribution for PDO_prepared_static_e is more dispersed, matching the skewness and kurtosis values in Table 26.

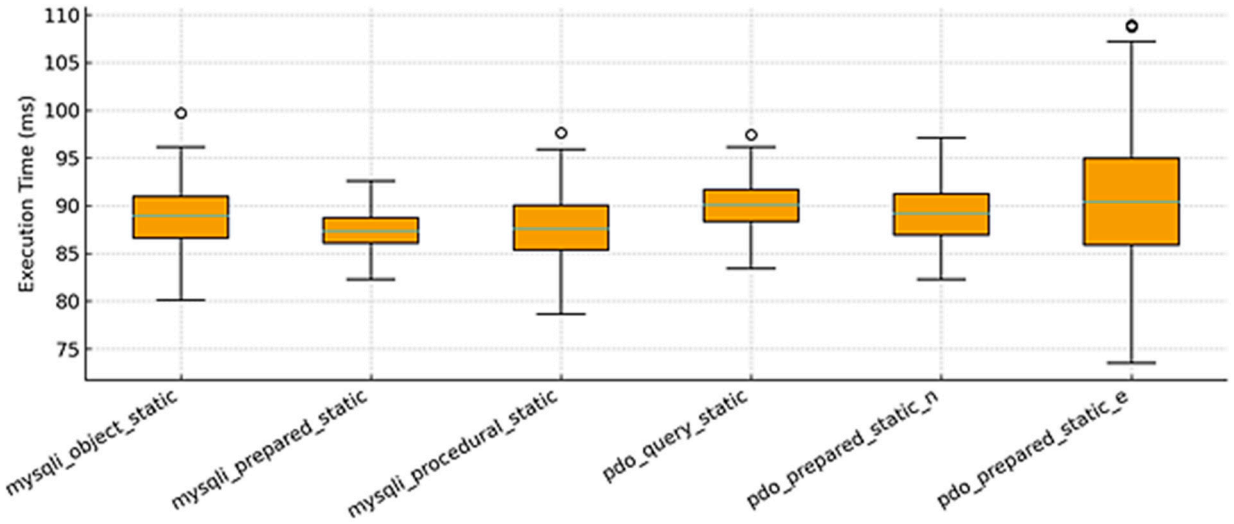


Figure 18. Box-plot of execution times (GROUP BY and HAVING, n = 200).

4.6.6. Conclusions of the Descriptive Statistics Analysis for SELECT with HAVING and GROUP BY (Connection Outside the Benchmark Test (M5))

Based on the descriptive statistics presented in the previous sections, several key conclusions can be drawn regarding the performance and consistency of the tested methods:

- MySQLi_prepared_static stands out as the most optimal method for this query type—combining a low mean, low variability, and low memory consumption;
- PDO_prepared_static_e is the least stable and shows greater deviation and outliers, making it less suitable for reliable performance;
- MySQLi_procedural_static is extremely consistent in memory consumption, but that constancy comes with somewhat weaker temporal stability;
- PDO_query_static and PDO_prepared_static_n offer a good balance of performance and memory use, but do not outperform MySQLi_prepared_static.

We recommend applying the MySQLi_prepared_static methodology for use cases where time efficiency and predictability are key, while emulated prepared statements (PDO) should be used with caution in scenarios requiring low latency.

4.6.7. Normality Tests and Non-Parametric Tests (M5)

Based on the results of normality tests (Shapiro–Wilk and D'Agostino), all execution-time and memory-consumption data significantly deviate from normal distribution (all p-values < 0.05). No exceptions were observed—every tested case showed p-values equal to zero, clearly indicating a violation of normality assumptions.

Therefore, non-parametric Kruskal–Wallis tests were used for further statistical comparison, as they do not require normality and are suitable for assessing differences across multiple independent groups:

- Execution time: H = 370.6310, p = 0.000000 → Statistically significant differences between methods;
- Peak memory: H = 819.3837, p = 0.000000 → Statistically significant differences between methods.

To precisely determine between which methods significant differences exist, Dunn–Bonferroni post hoc tests were conducted for pairwise comparisons on both metrics.

Execution time–key differences:

- MySQLi_object_static and PDO_query_static do not differ (p = 1), nor do MySQLi_object_static and MySQLi_prepared_static ($p \approx 0.75$);

- MySQLi_procedural_static shows statistically significant differences relative to all other methods (e.g., vs. MySQLi_object_static: $p = 3.96 \times 10^{-38}$);
- PDO_prepared_static_n differs significantly from most methods, including MySQLi_object _static ($p = 0.000294$), but not from MySQLi_prepared_static ($p \approx 0.31$);
- PDO_prepared_static_e occupies a middle position—no significant difference from MySQLi_object_static ($p \approx 0.19$), but significantly different from MySQLi_procedural _static ($p = 3.70 \times 10^{-25}$) and PDO_prepared_static_n ($p = 1.95 \times 10^{-10}$).

Memory consumption–key differences:

- MySQLi_object_static and MySQLi_procedural_static do not differ ($p = 1$), but both differ significantly from MySQLi_prepared_static ($p = 5.68 \times 10^{-71}$ and $p = 1.75 \times 10^{-83}$);
- MySQLi_prepared_static shows pronounced differences relative to all other methods, including PDO approaches (e.g., vs. PDO_query_static: $p = 1.83 \times 10^{-59}$);
- PDO_prepared_static_e and PDO_prepared_static_n do not differ from each other ($p = 1$), but differ significantly from MySQLi_procedural_static ($p = 2.22 \times 10^{-86}$) and MySQLi_prepared_static;
- PDO_query_static shows moderate differences relative to the PDO_prepared methods (e.g., vs. PDO_prepared_static_e: $p \approx 0.437$), but differs significantly from MySQLi_prepared_static and PDO_prepared_static_n.

Given the substantial deviations from normality, the non-parametric tests confirm statistically significant differences among the analyzed database access methods, both in execution time and memory consumption.

The post hoc results indicate the following key findings:

- MySQLi_procedural_static shows the largest deviation in execution time, signaling lower efficiency of this method in real-world conditions;
- PDO_query_static and MySQLi_object_static exhibit stable and competitive performance, both in time and memory;
- PDO_prepared_static_n and MySQLi_prepared_static have somewhat higher memory overhead, likely due to additional internal structures related to statement preparation;
- PDO_prepared_static_e occupies a middle ground—neither the fastest nor the most memory efficient, but stable in its results.

These findings have practical implications for software optimization. In applications with strict performance constraints (e.g., real-time systems), methods such as PDO_query_static or MySQLi_object_static are recommended. Conversely, methods with higher memory overhead or longer execution times, such as MySQLi_procedural_static and PDO_prepared_static_n, should be used selectively, according to requirements for security, flexibility, and scalability.

4.6.8. Conclusions for SELECT with HAVING and GROUP BY (Connection Outside the Benchmark Test (M5))

Analyzing the SELECT query with GROUP BY and HAVING, where the connection is established outside the benchmark loop, allows the query's performance to be isolated from connection overhead.

Key findings are:

- Execution speed:
    - o MySQLi_prepared_static records the lowest mean (87.56 ms) and the smallest standard deviation (2.14 ms), making it the most reliable choice in this scenario.
    - o PDO_prepared_static_e shows the greatest variability and skewness, indicating outliers and reduced reliability.

- All methods exhibit positive skewness, implying occasional slower executions and, therefore, a lack of complete consistency.

- Memory consumption:
  - MySQLi_procedural_static has fully constant memory usage (std dev = 0 kB), indicating maximum determinism.
  - MySQLi_prepared_static has the lowest average memory consumption (1452.78 kB).
  - PDO methods—especially PDO_query_static and PDO_prepared_static_e—require somewhat more memory, although the differences are relatively moderate.
- Distributional characteristics:
  - Execution-time distributions for all methods are asymmetric (positive skewness) and peaked (high kurtosis), particularly for PDO_prepared_static_e, indicating frequent executions near the mean but with rare slower cases.
  - Memory distributions are flat (negative kurtosis), implying stable consumption without extreme spikes.
- Statistical significance:
  - Kruskal–Wallis tests confirm statistically significant differences among methods for execution time (H = 370.63) and memory (H = 819.38).
  - Post hoc analysis identifies that MySQLi_procedural_static differs significantly in time performance from all other methods.
  - PDO_prepared_static_e and PDO_prepared_static_n do not differ significantly from each other in memory, but differ from all MySQL methods.
- Recommendations:
  - MySQLi_prepared_static is the most reliable and efficient approach for SELECT queries with aggregation when the connection is already established.
  - PDO prepared emulation (PDO_prepared_static_e) should be avoided in production settings due to high variability and overhead.
  - MySQLi_procedural_static is useful for cases where deterministic memory usage is crucial, but at the cost of weaker time efficiency.
  - PDO native approaches (especially PDO_prepared_static_n) are an acceptable alternative where flexibility is required, but they do not reach the stability of MySQLi_prepared_static.

When connection overhead is not a factor in overall performance, MySQLi_prepared_static stands out as the optimal method, combining speed, stability, and memory economy.

4.7. Analysis for a SELECT Query with HAVING and GROUP BY (Connection Inside the Benchmark Test (M6))

This chapter analyzes the performance of a SELECT query that uses the GROUP BY and HAVING clauses, where the database connection is established within each iteration of the benchmark test. This approach places additional load on the system and provides insight into overall application performance under real-world conditions, where connection and execution occur dynamically. The structure of the analysis follows the usual format used in the previous chapters.

4.7.1. Execution-Time Analysis (M6)

Table 28 presents the execution-time statistics for each method in the dynamic connection scenario, including measures of central tendency, dispersion, and skewness.

Table 28. Execution time (milliseconds (ms) (M6)).

Method	Mean (ms)	Std Dev (ms)	Range (ms)	Skewness
MySQLi_object	25.21	7.65	34.46	0.27
MySQLi_prepared	26.13	7.45	31.49	0.14
MySQLi_procedural	26.35	7.94	33.87	0.26
PDO_query	26.09	7.58	32.18	0.08
PDO_prepared_n	25.90	7.45	25.70	0.05
PDO_prepared_e	24.15	7.21	26.73	0.43

Figure 19 illustrates that PDO_prepared_e achieves the lowest average execution time and variance, clearly outperforming other methods in this dynamic connection scenario.

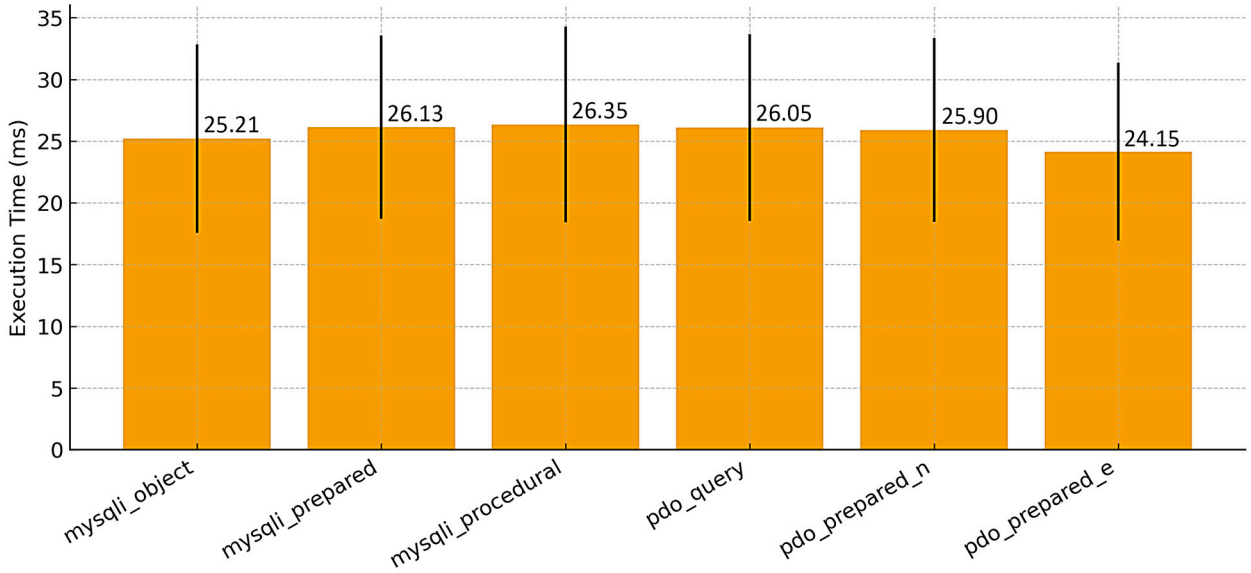


Figure 19. Average execution time (GROUP BY and HAVING, connection inside loop).

Observations:

- The smallest mean (24.15 ms) is achieved by PDO_prepared_e, making it the most efficient method.
- The greatest stability is also shown by PDO_prepared_e, with the lowest standard deviation.
- All methods have small positive skewness, indicating occasional slower executions but without pronounced outliers.

4.7.2. Peak Memory Analysis (M6)

Table 29 summarizes peak memory usage statistics for each method under dynamic connection conditions, including mean values, variability, and distribution characteristics.

Table 29. Peak memory consumption (kilobytes (kB) (M6)).

Method	Mean (kB)	Std Dev (kB)	Range (kB)	Skewness
MySQLi_object	97.38	8.65	24.11	0.37
MySQLi_prepared	97.55	8.47	23.63	0.40
MySQLi_procedural	97.38	8.65	24.11	0.37
PDO_query	97.86	8.65	24.11	0.37
PDO_prepared_n	98.68	7.79	21.78	0.53
PDO_prepared_e	97.86	8.65	24.11	0.37

Figure 20 shows that all methods exhibit nearly identical memory consumption, with only PDO_prepared_n slightly higher than the others.

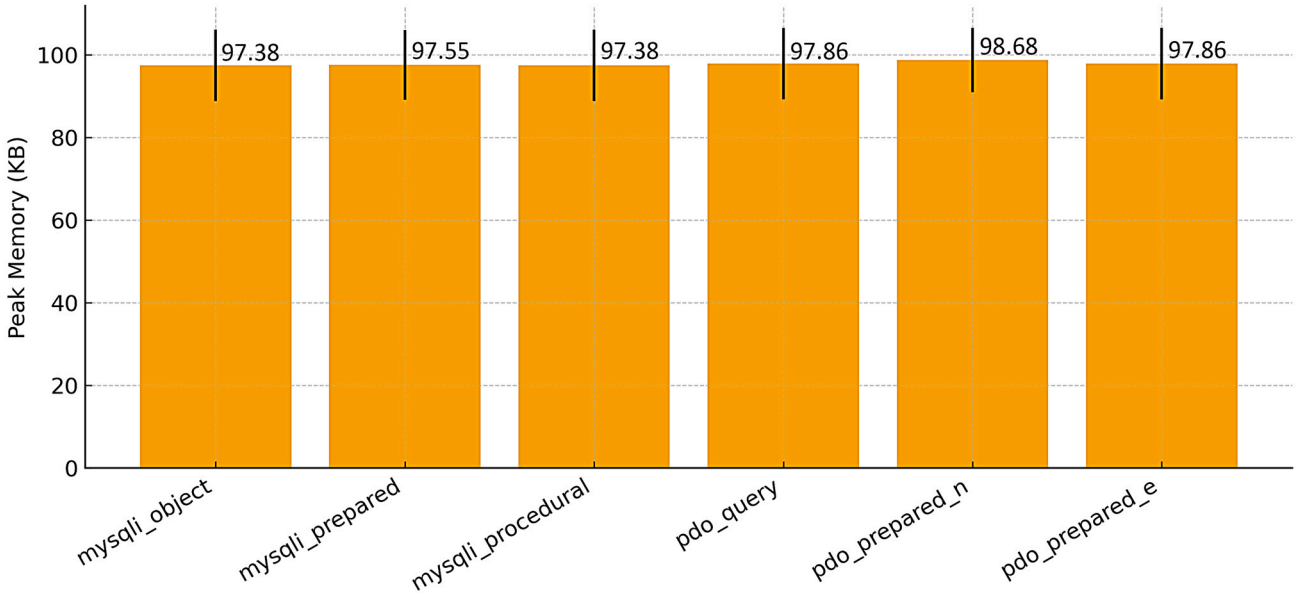


Figure 20. Average peak memory (GROUP BY and HAVING, connection inside loop).

Observations:

- The highest average memory consumption is recorded by PDO_prepared_n (98.68 kB).
- All methods have very similar memory usage and low variation, indicating system stability with respect to memory.
- Skewness values are mildly positive, without extreme values.

4.7.3. Relationship Between Means and Medians (Time and Memory (M6))

Table 30 compares mean and median values for execution time and memory usage, providing insights into the distribution symmetry and presence of outliers.

Table 30. Mean vs. median (M6).

Method	Mean (ms)	Median (ms)	Mean (kB)	Median (kB)
MySQLi_object	25.21	28.62	97.38	96.09
MySQLi_prepared	26.13	29.36	97.55	96.09
MySQLi_procedural	26.35	29.19	97.38	96.09
PDO_query	26.09	29.39	97.86	96.57
PDO_prepared_n	25.90	29.15	98.68	96.57
PDO_prepared_e	24.15	21.77	97.86	96.57

Most methods have slightly higher medians than means, implying some faster values pulling the mean downward. Pdo_prepared_e is the only method with a median lower than the mean—indicating slower outliers in rare cases.

4.7.4. Skewness and Kurtosis (M6)

Table 31 presents skewness and kurtosis values for execution time and memory usage, providing insights into the shape and flatness of the distributions.

All methods exhibit negative kurtosis—distributions are “flat,” without pronounced peaks. Pdo_prepared_n and PDO_prepared_e have the highest (least negative) kurtosis, indicating a more even distribution without extreme values.

Table 31. Skewness and kurtosis (M6).

Method	Skewness (ms)	Kurtosis (ms)	Skewness (kB)	Kurtosis (kB)
MySQLi_object	0.27	−0.68	0.37	−1.39
MySQLi_prepared	0.14	−0.74	0.40	−1.37
MySQLi_procedural	0.26	−0.56	0.37	−1.39
PDO_query	0.08	−0.83	0.37	−1.39
PDO_prepared_n	0.05	−1.22	0.53	−1.33
PDO_prepared_e	0.43	−1.16	0.37	−1.39

4.7.5. Confidence Intervals (95%) (M6)

Table 32 shows the 95% confidence intervals for execution time and memory usage, providing an indication of the precision and reliability of the measurements across methods.

Table 32. CI 95% (time and memory (M6)).

Method	CI 95% (Time, ms)	CI 95% (Memory, kB)
MySQLi_object	±1.07	±1.21
MySQLi_prepared	±1.04	±1.18
MySQLi_procedural	±1.11	±1.21
PDO_query	±1.06	±1.21
PDO_prepared_n	±1.03	±1.09
PDO_prepared_e	±1.00	±1.21

All confidence intervals are narrow, confirming the stability of the measurements. PDO_prepared_e has the smallest CI for execution time, further confirming its consistency. Variations in memory are very uniform and without significant deviations.

The distribution of execution times (Figure 21) confirms that differences across methods are minimal, consistent with the Kruskal–Wallis test ($p = 0.058$), except for the superior stability of PDO_prepared_e.

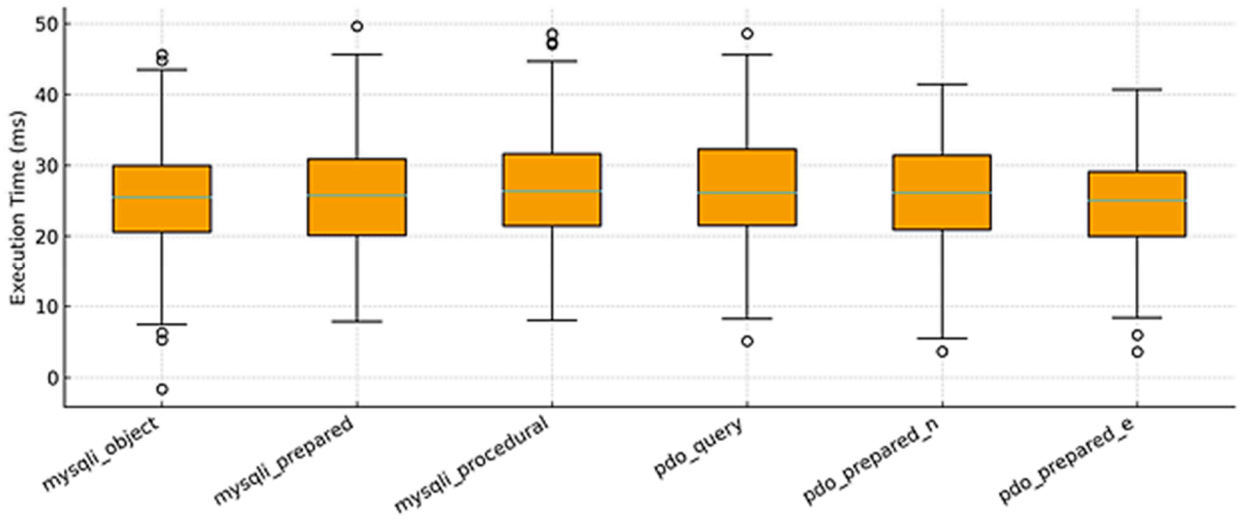


Figure 21. Box-plot of execution times (GROUP BY and HAVING, simulated, $n = 200$).

4.7.6. Conclusions of the Descriptive Statistics Analysis for SELECT with HAVING and GROUP BY (Connection Inside the Benchmark Test (M6))

The following key conclusions summarize the results of the descriptive statistical analysis for the execution of a SELECT query with HAVING and GROUP BY clauses, where the connection is established inside the benchmark test:

- PDO_prepared_e proves to be the most efficient method in terms of execution time and stability, with
  - The lowest average execution time (24.15 ms);
  - The lowest standard deviation (7.21 ms);
  - The smallest confidence interval for time (±1.00 ms).
- PDO_prepared_n records the highest memory consumption, yet remains within reasonably stable bounds.
- MySQLi_procedural and MySQLi_prepared show somewhat higher average execution times but are not significantly worse in the overall picture.
- All methods have stable and similar memory-consumption distributions, with negligible differences.

It is recommended to use the PDO_prepared_e method in scenarios where the connection is frequently opened and closed (e.g., REST API requests), as it provides the fastest and most reliable performance when executing more complex queries with aggregation.

4.7.7. Normality Tests and Non-Parametric Tests (M6)

The results of normality tests (Shapiro–Wilk and D'Agostino) show that all observed execution-time and memory-consumption data significantly deviate from a normal distribution. All Shapiro–Wilk p-values are zero, while the D'Agostino p-values for execution time are somewhat higher but still below the significance threshold ($p < 0.05$) for all methods, except marginally for MySQLi_procedural ($p = 0.032$). Due to these results, non-parametric statistical tests were used for further analysis.

To assess statistically significant differences among methods, Kruskal–Wallis tests were applied:

- Execution time: H = 10.6846, $p = 0.058004$

  → Insufficient evidence of a significant difference among methods (at $\alpha = 0.05$);
- Peak memory: H = 58.1417, $p = 0.000000$

  → A statistically significant difference in memory consumption among methods.

Given that no significant difference was confirmed for execution time by the Kruskal–Wallis test, the post hoc analysis here serves as supplementary insight, without a formal statistical conclusion. For memory consumption, post hoc testing provides precise information on pairwise differences.

Execution time–post hoc analysis:

- No significant differences among methods—all p-values ≥ 0.15, confirming the Kruskal–Wallis result;
- The lowest p-values are between MySQLi_procedural and PDO_prepared_e ($p \approx 0.16$), which is insufficient for a statistically significant conclusion;
- Methods such as PDO_query, MySQLi_object, and MySQLi_prepared showed a high degree of similarity.

Memory consumption–post hoc analysis:

- PDO_prepared_n shows a significant difference in memory consumption compared to all MySQLi methods (e.g., vs. MySQLi_object: $p = 1.76 \times 10^{-7}$);
- PDO_prepared_e also differs significantly from MySQLi_object and MySQLi_procedural ($p = 0.00034$), but not from MySQLi_prepared ($p \approx 0.24$);
- MySQLi_prepared differs moderately from PDO_prepared_n ($p = 0.0016$), but not from the other MySQLi methods;
- PDO_query behaves similarly to PDO_prepared_e, with identical p-values relative to MySQLi_object and MySQLi_procedural.

When the connection is established inside the benchmark test, no statistically significant difference in execution time between methods is observed ($p = 0.058$), but there are significant differences in memory consumption.

All methods show similar performance in execution speed, indicating that overheads related to establishing the connection tend to equalize the performance of different approaches. However, PDO methods—particularly PDO_prepared_n—require significantly more memory compared to MySQLi_object and MySQLi_procedural. PDO_prepared_e and PDO_query are moderately memory intensive but more efficient than PDO_prepared_n.

These results suggest that in scenarios where the connection is initialized within each query (as is often the case in microservice architectures), the choice of method should be based more on memory consumption than on execution speed. In such conditions, MySQLi_object and MySQLi_procedural may be preferable in resource-constrained systems, while PDO_prepared_n should be used selectively, in cases where the benefits of prepared statements outweigh memory constraints.

4.7.8. Conclusions for SELECT with HAVING and GROUP BY (Connection Inside the Benchmark Test (M6))

When the connection is established within each test iteration, the measurements include the overhead of initializing the connection, thus more closely reflecting real application conditions (e.g., microservices, REST APIs):

- Execution speed:
  - PDO_prepared_e records the lowest average time (24.15 ms) and the lowest standard deviation (7.21 ms), making it the most efficient and most stable method in this scenario;
  - The other methods (PDO_query, MySQLi_object, MySQLi_prepared, etc.) show very similar performance, with no significant differences between them (confirmed by the Kruskal–Wallis test, $p = 0.058$);
  - All methods exhibit low skewness and negative kurtosis, suggesting uniform performance without pronounced outliers.
- Memory consumption:
  - PDO_prepared_n records the highest average memory consumption (98.68 kB), significantly higher than all MySQLi methods;
  - The other methods show very similar memory consumption and stability, with minimal deviations and low variability;
  - Statistical tests confirm significant differences in memory among methods ($H = 58.14$, $p < 0.001$).
- Distributional characteristics:
  - All methods show flat distributions for both memory and execution time (negative kurtosis), with mildly positive skewness, indicating stable and predictable performance;
  - The greatest deviation in time is seen for PDO_prepared_e, but in a positive sense—speed is high, and outliers occur rarely.
- Statistical significance:
  - No confirmed significant differences in speed between methods ($p = 0.058$);
  - There is a clear difference in memory consumption—PDO methods, especially PDO_prepared_n, require more resources compared to MySQLi variants.

- Recommendations:
    - o PDO_prepared_e is the best option when the connection is part of each query—it combines top speed and stability with acceptable memory consumption;
    - o MySQLi_object and MySQLi_procedural are good choices in resource-constrained environments, as they offer lower memory usage with acceptable performance;
    - o PDO_prepared_n should be used only when the advantages of native prepared statements are essential, as it comes at the cost of higher memory usage;
    - o PDO_query and MySQLi_prepared show balanced results and can be a good compromise in applications that require stability without extreme optimizations.

When the connection is opened within each iteration, differences among methods in terms of execution time are minimal, but the memory profile becomes crucial for deciding which method to choose. PDO_prepared_e stands out as the most efficient method, while MySQLi_object and MySQLi_procedural are the most frugal in terms of memory.

4.8. Summary of Evaluation Results

A consolidated interpretation of these results, highlighting dominant performance patterns and practical trade-offs across all tested scenarios, is provided in the following Discussion section.

5. Discussion of Results

Although many expectations about relative performance (e.g., assumed advantages of MySQLi or overheads of PDO) circulate within practitioner communities, such assumptions are rarely confirmed by formal measurement. Empirical validation reveals distributional properties, variability, and significance levels that are absent from informal reports. Several results in this study also diverge from commonly held expectations, underscoring the value of rigorous measurement.

The experimental results presented in the previous chapter illuminate the behavior of six different approaches to executing SELECT queries in MySQL. The evaluation varies parameters such as connection type, query complexity (WHERE, JOIN, GROUP BY, HAVING), and the method of connecting to the database (PDO, MySQLi, procedural, object-oriented, prepared/native queries).

Figure 22 summarizes the trade-offs between execution time, memory usage, and stability across all methods. This radar diagram highlights that MySQLi_prepared_static offers the best balance, while PDO methods demonstrate higher memory costs but competitive stability.

5.1. Performance in the Context of Application Scenarios

The results show that in scenarios where the connection is opened outside the benchmark test (i.e., once), performance varies significantly between methods. In such a case, PDO_prepared_e and PDO_query consistently achieve the fastest execution times, which makes them suitable for applications with high performance requirements, such as

- Web applications with a high query frequency;
- Data-heavy API endpoints;
- Dynamic pages that load a large number of records.

However, when the connection is opened within each iteration of the test, differences in speed among methods diminish. This scenario better reflects real conditions in microservice architectures and REST APIs, where the connection is often opened and closed per request. In that case, connection overhead levels out the timing differences between methods, making memory efficiency the decisive factor.

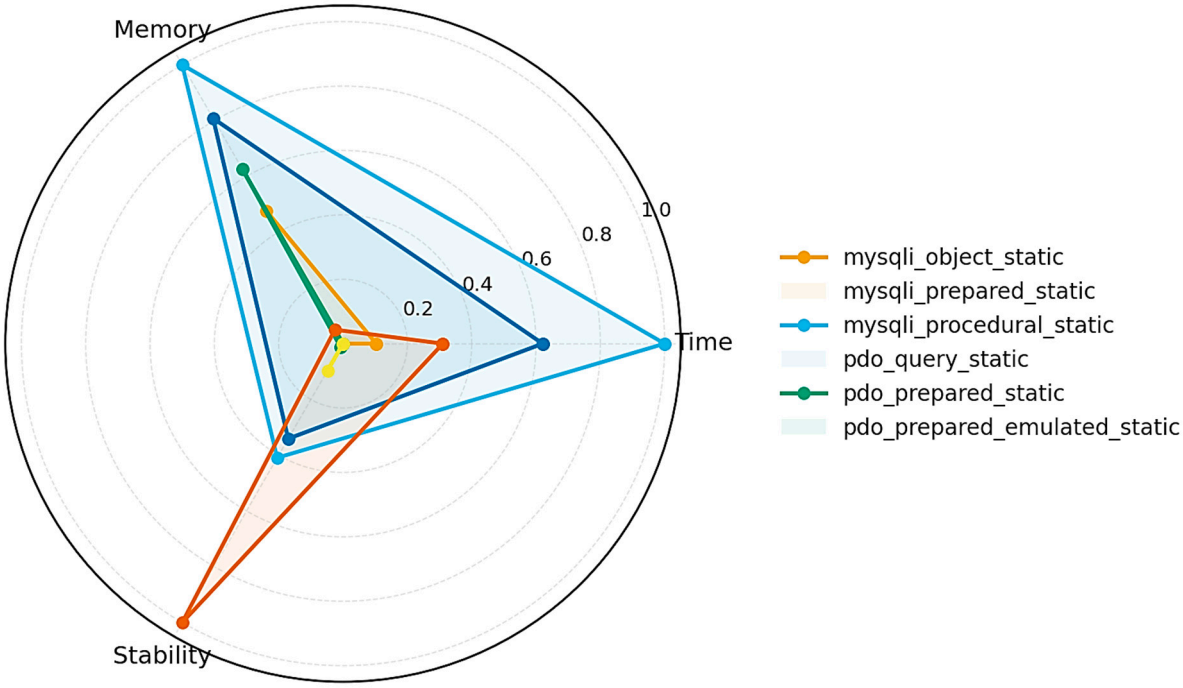


Figure 22. Radar diagram—comparative performance.

5.2. Memory Consumption as a Constraining Factor

While most methods behave consistently in terms of memory, PDO_prepared_n stands out with the highest memory consumption in almost all scenarios. This makes it a risky choice in systems with:

- Limited RAM (embedded systems, shared hosting);
- Multiple concurrent connections;
- A high frequency of joins and aggregations.

On the other hand, the MySQLi_object and MySQLi_procedural methods show the lowest memory consumption, which makes them suitable for systems where efficient resource usage is a priority.

The consistently lower memory footprint of MySQLi-based approaches can be linked to their comparatively lean abstraction layer and tighter coupling with the MySQL protocol. MySQLi exposes result sets and statement objects with less internal indirection, while PDO introduces additional layers to support database portability, generic parameter binding, and unified error handling. These design choices improve flexibility and maintainability but increase the size and lifetime of internal data structures. Consequently, the observed memory-efficiency differences reflect deliberate trade-offs in API design rather than implementation inefficiencies.

5.3. The Relationship Between Speed and Stability

A key advantage of the PDO_prepared_e method is its ability to combine high speed with low variability (std dev). This stability is important in real systems because it contributes to performance predictability. Distributional characteristics (positive skewness and negative kurtosis) indicate the absence of extreme values—even in scenarios with a larger number of JOIN or GROUP BY operations.

This behavior is consistent with the protocol-level difference between emulated and native prepared statements. In emulated mode, more processing remains on the PHP side, while native mode shifts preparation and execution planning to the server, which can increase memory footprint and affect variability under complex queries.

The observed differences between PDO_prepared_e and PDO_prepared_n are consistent with their protocol-level behavior. In emulated mode, PHP constructs complete SQL strings and sends them as simple text queries, while in native mode the server maintains prepared execution plans across invocations.

From a systems-level perspective, the lower variability observed for PDO_prepared_e can be interpreted as a consequence of its simplified interaction with the MySQL server. In emulated prepared statement mode, query construction, parameter substitution, and error handling are performed entirely on the PHP side, resulting in a single text-based protocol exchange per execution. This reduces the number of protocol round-trips and avoids server-side state management associated with prepared execution plans. By contrast, native prepared statements require additional protocol interactions for statement preparation, metadata handling, and execution, which may increase variability under complex queries or frequent connection churn. Although internal protocol states were not instrumented in this study, the observed stability patterns are consistent with these known differences in client–server interaction design.

Conversely, MySQLi_prepared and MySQLi_procedural are somewhat slower, but stable and memory-efficient. Their straightforward mode of operation can be an advantage in simpler applications or for educational purposes.

5.4. Statistical Relevance and Practical Significance

Although Kruskal–Wallis tests showed no statistically significant differences in execution time in scenarios with the connection inside the test, in practice even small savings in milliseconds per query can have a cumulative effect in heavily loaded systems.

In contrast, differences in memory are statistically significant and practically relevant. PDO_prepared_n requires up to 1.5 kB more per query on average, which can pose a serious problem in highly scalable systems.

5.5. Contextual Method Recommendation

Based on the tested scenarios and the results obtained, the following recommendation can be made depending on context (Table 33).

Table 33. Context-aware selection of PHP database access methods.

Application Context	Recommended Method	Rationale
REST API with frequent connections	PDO_prepared_e	Fastest and most stable, with acceptable memory overhead
Resource-constrained system	MySQLi_object/procedural	Lowest memory consumption
Application with demanding JOIN/HAVING	PDO_query or prepared_e	Better handling of complex queries
High degree of parallelism	Avoid PDO_prepared_n	Higher memory consumption may overwhelm the system

Note: These recommendations reflect the tested single-host environment and workload design; alternative runtime stacks (e.g., PHP-FPM worker pools, pooled/persistent connections, networked deployments) may shift the relative trade-offs.

5.6. Limitations and Opportunities for Extension

Although the tests covered different query types and connection scenarios, there are limitations to bear in mind:

- The tests were conducted under controlled conditions—in production systems there may be variables such as network latency, caching, and concurrent access;
- The security dimension (e.g., SQL injection resilience) was not analyzed;

- The tests were performed on a MySQL database; other systems (PostgreSQL, MariaDB) may yield different results;
- The impact of using a connection pool was not considered and may change the performance of in-test connections.

The discussion of results indicates that there is no universally best method; rather, the most appropriate approach depends on the specific context of use. While PDO methods are often faster, MySQLi approaches are more economical in terms of memory. Maintaining a balance between speed, memory, and stability is crucial for making an optimal architectural decision in database and application design.

The tests were conducted under controlled conditions. In production systems there may be variables such as network latency, different caching policies (e.g., InnoDB buffer-pool behavior, reverse proxies), and concurrent access patterns that were not explicitly modeled in this study. In our local WAMP setup, MySQL's legacy query cache was not available (MySQL 8.x), while storage and OS-level caching were allowed to warm up naturally. Future work may explore alternative designs that vary SQL text per iteration or periodically flush caches to approximate cold-start scenarios.

An additional experiment where the SQL query text or parameters vary in each iteration could further isolate the effect of database caching mechanisms on execution time and memory usage. This represents a relevant extension of the present study and is included as a direction for future work.

From a systems perspective, several dimensions remain intentionally out of scope in order to preserve attribution of observed effects to the PHP–MySQL interface layer. Specifically, the current study does not evaluate alternative DBMS backends (e.g., MariaDB or PostgreSQL), different PHP branches and driver builds, networked or geographically distributed deployments, or concurrent multi-client workloads with connection pooling. These factors can materially affect both absolute performance and the relative ranking of access modes through additional latency, contention, and server-side scheduling effects. Consequently, the present results should be interpreted as a controlled baseline for single-host execution, while the provided benchmark framework is designed to be extended in future work to systematically cover multi-DBMS setups, version variability, pooled connections, and concurrent/networked scenarios.

Common production deployments (e.g., Linux-based stacks with Nginx and PHP-FPM worker pools) may alter both absolute performance and relative rankings due to process isolation, scheduling, and connection reuse behavior. The present benchmark focuses on read-only SELECT workloads; write-heavy patterns (INSERT/UPDATE/DELETE) and transaction-intensive scenarios were not evaluated and may stress different parts of the driver/server pipeline. CPU-time measurements were intentionally excluded from comparative reporting because, in the tested stack, CPU timers exhibited quantization and sporadic anomalies; robust separation of user vs. kernel time would require OS-level profiling under the target deployment model. Consequently, the reported results are best interpreted as a controlled baseline, while the framework is designed for systematic extension to alternative runtimes, concurrency, pooling/caching strategies, and mixed read/write workloads.

From a measurement perspective, the use of wall-clock time as the primary performance metric implies that observed results inherently aggregate CPU execution, I/O waiting, and operating system scheduling effects. While these factors were partially stabilized in the single-host experimental setup through repeated execution and environmental isolation, they cannot be fully controlled at the application level. In distributed or multi-worker deployments, such effects may dominate overall latency and influence relative

performance rankings, reinforcing the interpretation of the present results as a controlled baseline rather than a deployment-independent performance model.

Although the benchmark workload is intentionally limited to read only SELECT queries, the observed trends allow for cautious theoretical inference regarding write-intensive workloads. INSERT, UPDATE, DELETE, and transactional operations are more likely to benefit from server-side prepared statements due to repeated execution plans, reduced parsing overhead, and stronger integration with transaction management. Consequently, the relative advantages of native prepared statements may become more pronounced under write-heavy or transactional scenarios. These workloads, however, introduce additional dimensions such as isolation levels, locking behavior, and durability guarantees, and therefore require a dedicated experimental design beyond the scope of the present study.

5.7. Scientific Contribution and Novelty of the Study

The primary contribution of this study is an integrated, reproducible benchmarking framework that jointly evaluates:

- Multiple PHP–MySQL access layers (MySQLi and PDO);
- Three categories of prepared statement execution (procedural, object-oriented, native, and emulated modes);
- The impact of query complexity ranging from simple SELECT to multi-JOIN and HAVING operations;
- Two fundamentally different connection-handling strategies (inside vs. outside each request).

Individually, these technologies and statistical tools are well established, but previous work has typically examined only subsets of these factors or reported results without transparent, reusable tooling. By contrast, this research offers an end-to-end methodology, openly available scripts and datasets, and a systematically documented comparison of timing, memory usage, stability, and statistical significance. In this sense, the novelty lies less in introducing new algorithms and more in providing a rigorously engineered experimental setup and evidence-based guidelines that other researchers and practitioners can replicate, critique, and extend.

Unlike previous works, which typically examine only subsets of these factors or lack transparency and repeatability, this research offers an integrated methodology with openly available tools and datasets. It also provides a comprehensive comparative analysis of timing, memory usage, stability, and statistical significance.

Additional artifacts (raw data, analysis scripts, and extended statistical outputs) are provided in the Supplementary Materials.

6. Conclusions

This study provides a detailed analysis of different approaches to communication between PHP applications and a MySQL database, focusing on the two most common PHP extensions: MySQLi and PDO, as well as on the impact of query complexity, connection-establishment mode, and the use of prepared statements. The aim of the research was to identify advantages and limitations of individual approaches through systematic benchmark tests in real-world scenarios such as REST API requests and microservice architecture.

Our findings suggest that the choice between MySQLi and PDO should be guided by a trade-off between raw performance in a MySQL-only environment (where MySQLi tends to be more efficient) and architectural flexibility in multi-database or portable applications (where PDO's abstraction layer provides clear advantages despite a modest performance overhead in some scenarios).

Portability, maintainability, and secure coding practice (e.g., prepared statements) are treated here as architectural decision variables rather than performance metrics measured by the benchmark.

The research encompassed six different methods of executing SELECT queries. The methods were tested in several categories (basic SELECT, SELECT with WHERE, JOIN, GROUP BY, HAVING), with two connection regimes (establishing the connection outside and inside each test iteration). Parameters such as execution time, memory consumption, distributional characteristics, and the statistical significance of differences were analyzed.

Key findings include the following:

- PDO proved to be more flexible and suitable for applications that require multi-database support and an object-oriented coding style. In most tests, particularly with simpler SQL operations, PDO methods showed better performance and lower memory consumption, while the PDO_prepared_e method stood out as the most efficient in terms of execution speed and stability, especially in more complex queries and scenarios with dynamic connections;
- MySQLi provides better performance for more complex queries, especially those with multiple JOIN operations and in high-load situations. Its tight coupling with the MySQL protocol and native support for prepared statements contribute to lower memory consumption and efficiency, which makes it suitable for systems with limited resources and high parallelization;
- When the connection is established within each query, differences in speed among methods decrease, while memory consumption becomes the deciding factor in choosing the optimal approach;
- Statistical analyses show that differences in execution time between methods are not always significant ($p \approx 0.058$), but differences in memory consumption are clear and statistically significant ($p < 0.001$);
- Distributional analysis confirms stable performance of all methods without pronounced outliers, which makes them reliable for production use.

Taking these results into account, it can be concluded that there is no universally best method. The choice between PDO and MySQLi should be guided by the specific needs of the project and the application architecture:

- For applications that require support for multiple database types, flexibility, and clearer object-oriented code—PDO is the recommended choice;
- For applications that are strictly tied to MySQL and where maximum speed and efficiency are a priority—MySQLi can deliver better results.

In terms of scientific contribution, this work expands the existing literature by offering the first unified and fully reproducible benchmark that simultaneously compares query execution performance, memory efficiency, and stability across six PHP database-access modes under realistic query-complexity conditions. The accompanying open-source toolchain and dataset enable independent verification, replication, and extension of the results, addressing a notable gap in earlier studies that typically lacked controlled conditions or excluded key architectural factors such as native versus emulated prepared statements.

Beyond performance, prepared statements remain a key element for application security, significantly reducing the risk of SQL injection. Native prepared statements are more efficient for complex and repeated operations, while emulated statements in PDO may be more optimal for simpler and less frequent queries.

Recommendations for future work include

- Investigating the impact of connection pools and persistent connections on performance;

- Evaluating the security aspects of different methods, particularly resistance to SQL injection;
- Extending testing to other DBMSs such as PostgreSQL and MariaDB for better comparability;
- Analyzing the impact of caching, transactions, and concurrent database access in distributed systems.

An additional extension of this work would be the development of a simple performance prediction model that incorporates parameters such as query complexity, connection-handling strategy, and API type to estimate expected behavior under varying workloads. Beyond raw performance and security, future studies may also integrate development and maintenance costs, team expertise, and cloud deployment considerations, contributing toward a broader technology selection framework aligned with information systems research on technology adoption.

The results indicate that API selection should not be framed as a speed-only decision. For MySQL-only systems, MySQLi can be a strong choice when minimizing execution time and memory usage is the dominant goal. However, the stability of results across increasing query complexity must also be considered. PDO introduces a modest overhead in some cases, but it can improve architectural resilience through a consistent abstraction layer and may yield more predictable behavior depending on the prepared statement mode. Accordingly, the preferred interface should be determined by the combined trade-off among performance, stability, and maintainability, rather than by average timing alone.

It is important to emphasize that the quantitative results reported here are specific to the tested configuration: a single workstation running a WAMP stack, the ClassicModels schema, and single-threaded execution without network latency or connection pooling. Other platforms, database engines, and deployment scenarios may alter both absolute and relative performance. Consequently, the recommendations in this paper should be read as context-aware guidelines for environments similar to the one studied, and as a starting point for further benchmarking rather than as universally applicable rules.

From a systems perspective, wall-clock time represents an application-relevant aggregate metric that inherently includes CPU execution, I/O waiting, and scheduling latency. In the present single-host setup, these factors were partially controlled through repeated execution and environmental isolation; however, they cannot be fully eliminated at the application level. In networked or multi-worker deployments (e.g., PHP-FPM with Nginx), such factors may become dominant contributors to overall latency, potentially altering relative performance rankings. This reinforces the interpretation of the reported results as a controlled baseline rather than a universal performance model. These observations directly reinforce the scope limitations discussed in Section 5.6, particularly with respect to scheduling, networking, and deployment-model effects.

This work provides a useful framework for making technically grounded decisions when choosing an approach to database interaction in PHP applications, taking into account performance, security, and system resources.

Supplementary Materials: The following supporting information can be downloaded at https://github.com/ZLovrekovic/PHP-extensions-compare (accessed on 14 September 2025): Data, Results, Scripts.

Author Contributions: Conceptualization, N.A. and Z.L.; methodology, N.A. and Z.L.; software, N.A.; validation, N.A. and Z.L.; formal analysis, N.A.; investigation, N.A., Z.L., and Đ.Š.; resources, Z.L. and J.P.; data curation, Z.L. and H.S.; writing—original draft preparation, Z.L.; writing—review and editing, N.A. and Z.L.; visualization, N.A. and H.S.; supervision, Z.L. and Đ.Š.; project administration, J.P.; funding acquisition, Đ.Š., H.S., and J.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data are contained within the article and Supplementary Materials.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. PHP Manual. MySQLi Prepared Statements. Available online: https://www.php.net/manual/en/mysqli.prepare.php (accessed on 14 September 2025).
2. PHP Manual. PDO::ATTR_EMULATE_PREPARES. Available online: https://www.php.net/manual/en/pdo.setattribute.php (accessed on 14 September 2025).
3. OWASP Foundation. SQL Injection Prevention Cheat Sheet. Available online: https://cheatsheetseries.owasp.org (accessed on 14 September 2025).
4. Sönmez, Y.; Bayat, O.; Altuğlu, T.B.; Duru, A.D. Performance Comparison of PHP-ASP Web Applications via Database Queries. In Proceedings of the ICEMIS'15: Proceedings of the International Conference on Engineering & MIS 2015, Istanbul, Turkey, 24–26 September 2015; ACM: New York, NY, USA, 2015. [CrossRef]
5. Laaziri, M.; Benmoussa, K.; Khoulji, S.; Kerkeb, M.L. A Comparative study of PHP frameworks performance. *Procedia Manuf.* **2019**, *32*, 864–871. [CrossRef]
6. Salunke, S.V.; Ouda, A. A Performance Benchmark for the PostgreSQL and MySQL Databases. *Future Internet* **2024**, *16*, 382. [CrossRef]
7. Zmaranda, D.R.; Moisi, C.I.; Győrödi, C.A.; Győrödi, R.Ş.; Bandici, L.; Popescu, D.E. An Analysis of the Performance and Configuration Features of MySQL Document Store and Elasticsearch as an Alternative Backend in a Data Replication Solution. *Appl. Sci.* **2021**, *11*, 11590. [CrossRef]
8. Győrödi, C.A.; Dumse-Burescu, D.V.; Győrödi, R.Ş.; Zmaranda, D.R.; Bandici, L.; Popescu, D.E. Performance Impact of Optimization Methods on MySQL Document-Based and Relational Databases. *Appl. Sci.* **2021**, *11*, 6794. [CrossRef]
9. Taipalus, T. Database Management System Performance Comparisons: A Systematic Literature Review. *arXiv* **2023**, arXiv:2301.01095. [CrossRef]
10. Yuan, Y.; Lu, Y.; Zhu, K.; Huang, H.; Yu, L.; Zhao, J. A Static Detection Method for SQL Injection Vulnerability in PHP Applications Considering MySQLi and PDO. *Appl. Sci.* **2023**, *13*, 11763. [CrossRef]
11. Kamani, M.; Bhardwaj, R.; Singh, P. Performance Study of Various Connectivity of PHP and MySQL Database. *Int. J. Creat. Res. Thoughts* **2018**, *6*, 486.
12. MoldStud. Boost Your PHP Database Performance: How PDO Outperforms MySQLi. 3 July 2025. Available online: https://moldstud.com/articles/p-boost-your-php-database-performance-how-pdo-outperforms-mysqli (accessed on 14 September 2025).
13. MoldStud. Understanding Prepared Statements: PDO vs. MySQLi in PHP Development. 7 June 2025. Available online: https://moldstud.com/articles/p-understanding-prepared-statements-pdo-vs-mysqli-in-php-development (accessed on 14 September 2025).
14. Dev.to. MySQL vs MySQLi vs PDO: Performance, Difference, Security (Benchmark Comparison). Available online: https://dev.to/mega6382/mysql-vs-mysqli-vs-pdo-performance-benchmark-difference-and-security-comparison-565o (accessed on 17 December 2025).
15. Westergren, J. PDO vs. MySQLi Speed Comparison. 2017. Available online: https://www.jimwestergren.com/pdo-versus-mysqli (accessed on 14 September 2025).
16. Wooptoo. PDO vs MySQLi Performance Comparison. 2012. Available online: https://wooptoo.com/blog/pdo-vs-mysqli-performance-comparison (accessed on 14 September 2025).
17. USAVPS. Performance Comparison Between PDO-MySQL and MySQLi in PHP. 19 March 2024. Available online: https://usavps.com/blog/107127 (accessed on 14 September 2025).
18. Starbuck, C. Descriptive Statistics. In *The Fundamentals of People Analytics*; Springer Nature: London, UK, 2023. [CrossRef]
19. De Michele, C.; De Bartolo, S. Skewness-Kurtosis: Small Samples and Power-Law Behavior. *arXiv* **2025**, arXiv:2506.16906.
20. Mukherjee, H.; Bhonge, P. Assessing Skew Normality in Marks Distribution. *arXiv* **2025**. [CrossRef]
21. Arnastauskaitė, J.; Ruzgas, T.; Bražėnas, B. An Exhaustive Power Comparison of Normality Tests. *Mathematics* **2021**, *9*, 788. [CrossRef]
22. Chicco, D.; Sichenze, A.; Jurman, G. A Simple Guide to the Use of Student's t-test, Mann-Whitney U test, Chi-squared test, and Kruskal-Wallis Test in Biostatistics. *BioData Min.* **2025**, *18*, 56. [CrossRef] [PubMed]
23. Soetewey, A. Kruskal-Wallis Test, or the Nonparametric Version of the ANOVA. 2022. Available online: https://statsandr.com/blog/kruskal-wallis-test-nonparametric-version-anova/ (accessed on 14 September 2025).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.