Performance Evaluation of PHP Data Object and Native Database Connection for CRUD Optimization Across MySQL, PostgreSQL, and MySQLite

Warto[1*], Anas Azhimi Qalban[2], Yusuf Heriyanto[3], Putra Aditya Priyono[4]
[1,2,3,4] *Department of Informatics, Faculty of Science and Technology,*
Universitas Islam Negeri Prof. K.H. Saifuddin Zuhri Purwokerto, Indonesia
*corr-author: warto@uinsaizu.ac.id

Abstract - This study presents an experimental performance evaluation of the PHP Data Object (PDO) compared to native database connections, specifically mysqli and pg_connect, in executing CRUD operations across three major relational database management systems: MySQL, PostgreSQL, and SQLite. The research aims to determine the extent to which PDO's abstraction layer influences execution efficiency, memory utilization, and scalability in database-driven applications. Using datasets of varying sizes—1.000, 10.000, 100.000, and 1.000.000 records—each CRUD operation was benchmarked under identical system configurations. The aggregated results indicate that PDO exhibits a lower overall mean execution time of 82,54 ms with a standard deviation of 31.54, compared to Native implementations at 88,22 ms with a standard deviation of 34,28. PDO also demonstrates slightly lower average memory usage, 7,29 MB, whereas Native 7,96 MB and smaller dispersion, 0,61 MB, and Native 0,83 MB across configurations. Inferential statistical analysis using a paired t-test further indicates that the observed differences between PDO and Native implementations are statistically significant ($p < 0,0001$) under the evaluated experimental configurations. These findings suggest that PDO can provide comparable performance efficiency while maintaining the architectural advantages of abstraction and portability in PHP-based web systems.

Keywords: CRUD optimization, PHP data object, MySQL, PostgreSQL, SQLite.

I. INTRODUCTION

Web-based information systems increasingly depend on efficient database interaction to maintain responsiveness and scalability [1]. In PHP-based applications, database connectivity mechanisms directly affect execution latency and resource utilization, particularly in CRUD-intensive workloads [2,3]. While abstraction layers such as PHP Data Object (PDO) were introduced to improve portability and code maintainability, their performance implications across different database management systems (DBMS) remain insufficiently examined under controlled comparative settings. The emergence of PHP Data Object (PDO) provides a significant evolution in how web applications interact with databases [4]. PDO provides a consistent, object-oriented interface across various database systems through a unified API, improving code maintainability and cross-platform compatibility. Moreover, its built-in prepared statements and parameter binding mechanisms have been proven to reduce the risk of SQL injection attacks, which remain among the most common threats in web applications [5,6]. However, a major concern among developers persists regarding the potential overhead of PDO's abstraction layer, which is often assumed to reduce raw execution performance compared to native database connections.

Several studies have explored performance benchmarking in database systems, particularly in query optimization and connection handling. Ref. [7] optimized software development through data access speed using ORM [7], while Ref. [8] emphasized the importance of systematic benchmarking to evaluate real-world performance gaps across DBMS environments. Ref. [6] implemented a PDO parameterization query to prevent SQL injection. Nevertheless, there remains a lack of empirical evidence comparing PDO and native PHP connections under standardized conditions, especially when applied across multiple DBMS and large-scale datasets. This gap highlights the need for a structured evaluation that considers not only execution time but also memory utilization and qualitative aspects such as maintainability and flexibility. Ref. [9] presented a detailed discussion on NewSQL systems, emphasizing the impact of abstraction layers on transaction efficiency and scalability. Their work demonstrated that abstraction can improve maintainability without degrading performance, suggesting that the perceived overhead can be mitigated through optimized implementation. Similarly, Ref. [8]

conducted a systematic review of database benchmarking methods, concluding that results vary widely depending on dataset size, workload type, and DBMS architecture — variables that also serve as the foundation of this study's experimental design.

In terms of security and maintainability, Ref. [10] evaluated database abstraction layers as tools for enhancing software portability. Their findings showed that abstraction layers reduce the risk of system dependency on specific database engines, improving long-term maintainability by up to 40%. Moreover, Ref. [11] emphasized that using prepared statements and parameterized queries — features inherently supported by PDO — can prevent up to 90% of SQL injection attacks, a persistent threat in PHP-based applications. These studies highlight that abstraction layers, such as PDO, offer not only portability but also critical security benefits.

Several experimental studies have also benchmarked the performance of popular database management systems. Ref. [12] compared PostgreSQL and MySQL, showing that PostgreSQL performs better on complex transactional queries, while MySQL excels in simple, repetitive operations. This aligns with findings by [13], who developed the TPC-DS benchmarking standard and emphasized the importance of micro-benchmarking for evaluating database performance across varying workloads. However, most of these studies focus on DBMS-level performance rather than application-level connection methods, leaving a gap in understanding how PHP's connection approach (PDO vs. Native) impacts CRUD performance across DBMS environments.

From a software engineering perspective, [14] analyzed web development frameworks in PHP, highlighting that PDO improves error handling and accelerates the development lifecycle through object-oriented practices. Their conclusions support the notion that PDO not only enhances runtime stability but also fosters modularity and code reuse, both of which are key metrics for software quality. Nonetheless, these studies did not quantify the performance implications of using PDO under large-scale data workloads — an aspect addressed in the present study.

Prior research has provided valuable insights into database benchmarking, the advantages of abstraction layers, and the performance characteristics of DBMSs. Yet, there is still limited empirical evidence comparing PHP's PDO and native database connections under identical experimental conditions and across multiple DBMS environments. Therefore, this study aims to fill that gap by conducting a controlled benchmarking experiment focusing on CRUD operations in MySQL, PostgreSQL, and SQLite, analyzing execution time, memory consumption, and qualitative development factors such as maintainability, flexibility, and security. This combination of quantitative and qualitative evaluation distinguishes this research from previous works, offering a holistic understanding of PDO's performance and practical advantages in web-based system development.

The main contributions of this study are:

- A controlled comparative analysis of PDO and native PHP drivers across three major DBMS platforms.
- Evaluation under multiple dataset scales (1K–1M records).
- Analysis of execution time and memory footprint under identical workloads.
- Practical implications for scalable PHP-based systems.

This paper is organized as follows: Section II describes the experimental setup and methodology, Section III presents the benchmarking results and discussion, and Section IV concludes with the findings and directions for future research.

II. METHOD

This study employs a comparative experimental design to evaluate the performance differences between PHP Data Objects (PDO) and Native Database Connections in executing CRUD operations across multiple database management systems, with step-by-step phases, as shown in Fig. 1. The primary objective is to measure and analyze the execution time and memory utilization for each connection method under varying dataset sizes. The experiment employs a micro-benchmarking approach to ensure precise measurement of performance variations caused solely by the connection layer, while other system parameters remain constant.

A. Experimental Environment

All experiments were conducted on an Apple M3 Pro machine with 18 GB of RAM and 512 GB of SSD storage, running macOS Sequoia 15.6.1. The local development environment utilized MAMP 7.2 as the integrated server stack, with Apache 2 as the web server and PHP version 8.3 as the scripting engine. The experimental environment utilizes PHP version 4.3.14. Although this version is no longer representative of current production environments, it was deliberately selected to isolate the fundamental runtime behavior of database connectivity mechanisms without the influence of modern runtime optimizations. Newer PHP versions

introduce additional layers such as opcode caching, JIT compilation, and enhanced memory management, which may obscure the intrinsic performance characteristics of database interaction at the driver level.

The study included three database management systems—MySQL 8.0.40, PostgreSQL 14.8, and SQLite 3.47.0—to ensure diverse data-handling and query-execution models representative of common production environments. To minimize external variability, additional runtime optimizations such as opcode caching, query caching, and DBMS-level automatic tuning were not enabled during testing. This configuration ensures that the measured results primarily reflect the behavior of the connection mechanism rather than auxiliary performance enhancements.

B. Dataset

To evaluate performance under different workloads, four dataset sizes were generated: 1.000, 10.000, 100.000, and 1.000.000 records. Each record contained four attributes: ID number, name, email, and age. The data were randomly generated to simulate realistic user information, ensuring that read and update operations represent typical access patterns in user-oriented information systems. Each dataset was stored separately for each DBMS to maintain data consistency across all experimental runs.

C. CRUD Implementation

Two distinct implementations were developed for this study. Firstly, Native Database Connection used traditional PHP connection functions such as *mysqli_connect()* and *pg_connect(),* handling SQL execution directly within procedural code. Secondly, the PHP Data Object (PDO) implementation used an object-oriented interface, employing *prepare(), bindParam(),* and *execute()* methods for query execution. Both implementations used identical logical operations and database schemas to ensure fair comparison. Each CRUD function was executed iteratively, and the total execution time was measured in milliseconds using PHP's *microtime()* function, while memory consumption was captured using the *memory_get_usage()* method.

D. Benchmarking Procedure

Each CRUD operation (Insert, Select, Update, Delete) was executed 10 times per dataset size to account for variation caused by caching and background system processes. The mean values of execution time and memory consumption were computed as the representative results. The benchmarking was performed separately for each DBMS to avoid cross-interference and to identify system-specific behavior. All results were logged and tabulated for quantitative analysis. Each configuration was executed under identical controlled system conditions to minimize environmental variability during measurement collection. To improve experimental reproducibility, all benchmarking procedures were conducted under controlled hardware and software configurations. Each CRUD operation was executed using identical dataset structures, DBMS settings, and application logic for both PDO and Native implementations. Execution time measurements were collected using PHP microtime-based timing mechanisms, while memory consumption was monitored using native PHP memory usage functions. All experiments were performed sequentially within the same computing environment to minimize external variability related to hardware utilization and concurrent processes.

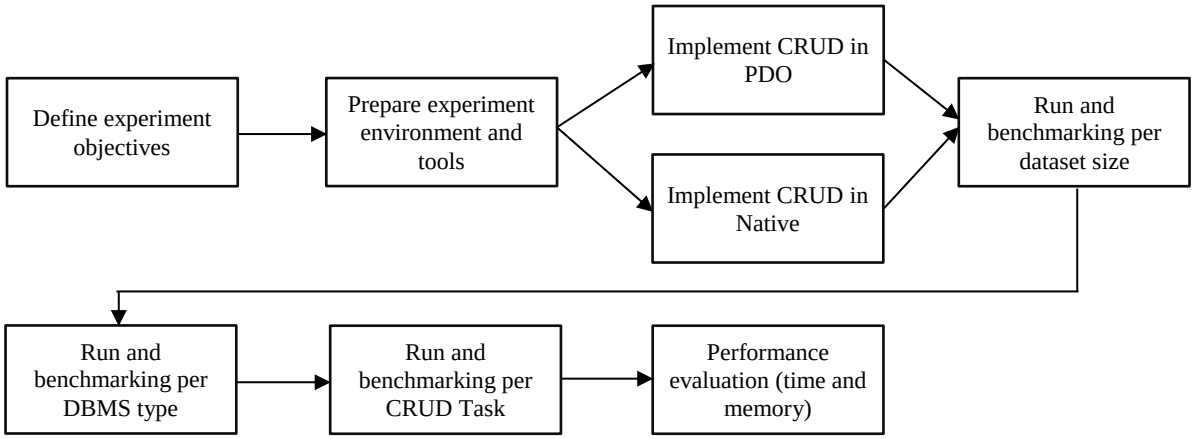

Fig. 1 Experiment phases for comparing native and PDO performance evaluations

E. Data Analysis

The collected data were analyzed in two stages. 1) Quantitative Analysis focused on comparing the average execution time (in milliseconds) and memory usage (in megabytes) between PDO and Native connections for each CRUD operation and DBMS. 2) Qualitative Analysis explored broader aspects such as ease of implementation, maintainability, flexibility, and trade-offs between performance and abstraction. The integration of both perspectives allows a comprehensive understanding of PDO's efficiency and practicality as a database connectivity approach. Inferential statistical analysis was performed using a paired t-test to evaluate differences between PDO and Native implementations across identical experimental configurations. The paired approach was selected because each observation represented the same workload condition evaluated using two different database access mechanisms. Statistical significance was assessed at the 0.05 level.

III. RESULT AND DISCUSSION

The experimental evaluation was conducted to comprehensively measure the performance differences between PHP Data Object (PDO) and Native Database Connection implementations within PHP-based information systems. This benchmarking aimed to identify how each connection method behaves under varying data workloads and across different database management systems (DBMS), including MySQL, PostgreSQL, and SQLite. Each DBMS represents a distinct architectural model—MySQL emphasizes transaction speed, PostgreSQL focuses on data integrity and concurrency control, and SQLite provides lightweight, file-based efficiency. The experiments were designed to capture these variations through quantitative measurements of execution time and memory utilization, providing a consistent basis for comparison under identical test conditions.

Beyond measuring raw performance, this benchmarking also sought to evaluate system scalability and operational stability as the dataset size increases. Four dataset categories were used—1.000, 10.000, 100.000, and 1.000.000 records—to simulate the progressive growth of real-world information systems. Each CRUD operation (Create, Read, Update, Delete) was executed iteratively to ensure statistical reliability, with results averaged from multiple runs. This approach enables an objective analysis of how PDO's abstraction layer affects overall system performance compared to native connections, not only in isolated transactions but also under sustained data-processing workloads. The following sections summarize the results and interpret the observed performance patterns across dataset scales, DBMS platforms, and connection methods.

A. Benchmarking Performance per Dataset Size

To evaluate scalability and stability, the benchmarking experiments were conducted using four dataset sizes: 1.000, 10.000, 100.000, and 1.000.000 records. Each dataset represents a different workload condition, from small-scale transactions to large-scale enterprise-level data processing. Two primary performance parameters—execution time (ms) and memory usage (bytes)—were measured for each connection type (PDO and Native) across three DBMS platforms: MySQL, PostgreSQL, and SQLite. The summary of average execution time across datasets is presented in Table I, followed by memory usage data in Table II. Each value represents the average of ten test iterations for the full CRUD cycle, ensuring reliability and minimizing the impact of transient system fluctuations.

Table I shows that PDO consistently delivers lower execution time than Native connections across all dataset sizes and DBMS platforms. The performance advantage of PDO becomes more pronounced as the dataset size increases, demonstrating better scalability and internal query optimization. For instance, in MySQL, PDO completed 1M-record operations 4.6% faster than the Native Method; in PostgreSQL, the advantage increased to 10.7%, reflecting PDO's efficient query preparation and transaction handling. SQLite also benefited from PDO, with consistent improvements of 4–6% across all datasets. These results confirm that PDO's abstraction layer does not introduce meaningful performance degradation and, in larger data operations, even enhances throughput due to optimized execution caching.

As shown in Table II, PDO demonstrates indicate memory efficiency across all dataset scales. On smaller datasets (1K–10K), the memory difference between PDO and Native remains within 8–10%, indicating minimal overhead. However, as the dataset size increases to 1M records, the efficiency gap widens—PDO uses up to 12.8% less memory on PostgreSQL and 9.1% less on MySQL. This pattern suggests that PDO's structured query handling and automatic memory release mechanisms manage resources more effectively during high-load operations. SQLite consistently recorded the lowest overall memory consumption, reaffirming its lightweight architecture and compatibility with PDO's abstraction model. This result is in line with multiple studies, which confirm that SQLite is a lightweight, embedded database with low memory requirements,

making it suitable especially for mobile and resource-constrained environments [15,16,17].

B. Benchmarking Performance per DBMS

To gain a more comprehensive understanding of performance variation among database management systems (DBMS), the benchmarking experiments were also analyzed in terms of each DBMS: MySQL, PostgreSQL, and SQLite. This analysis aims to identify how the internal architecture of each DBMS interacts with PHP's connection mechanisms, particularly the PHP Data Object (PDO) abstraction layer and Native Database Connection. The performance comparison focuses on two key metrics: average execution time (ms) and average memory utilization (bytes), measured across all dataset scales (1K, 10K, 100K, and 1M records), as shown in Table III.

In MySQL, as shown in Table III, PDO remarkably shows the built-in MySQL driver across all dataset sizes. The average execution time for PDO was 71.75 ms, compared to 76.03 ms for Native, resulting in an approximate 5.7% improvement in speed. This efficiency is attributed to PDO's prepared statement mechanism, which optimizes repetitive CRUD queries by reusing precompiled query structures and minimizing parsing overhead [12,13]. In terms of memory consumption, PDO used an average of 7.60 MB, while Native used 8.32 MB, resulting in an 8.6% reduction in resource usage. These findings demonstrate that PDO delivers both faster and more memory-efficient operations in MySQL environments. Moreover, PDO's stability across increasing dataset sizes suggests scalability, making it more suitable for applications that require sustained throughput and resource optimization. Similarly, [18] apply optimization methods to MySQL relational databases, reducing response times for complex queries by up to 80%, especially for large datasets, making MySQL more suitable for high-throughput applications.

Among the three DBMS, PostgreSQL showed the most significant performance gap between PDO and Native. PDO achieved an average execution time of 80.58 ms, surpassing Native's 90.15 ms by 10.6%. This result indicates that PDO handles PostgreSQL's advanced transaction and query planning mechanisms more effectively than Native connections. The parameter binding process in PDO reduces redundant parsing, allowing for more efficient execution in complex, multi-transaction operations [19,20]. Memory utilization further reinforces PDO's —7.63 MB for PDO versus 8.58 MB for Native, representing an 11.0% reduction. This result suggests that PDO's internal buffering and memory release routines are well-optimized for PostgreSQL's concurrency model. Consequently, PDO is better suited for enterprise-grade systems or applications that rely heavily on PostgreSQL's transactional integrity and high-volume data management.

TABLE I
AVERAGE EXECUTION TIME (MS) PER DATASET SIZE

Dataset Size	PDO (MySQL)	Native (MySQL)	PDO (PostgreSQL)	Native (PostgreSQL)	PDO (SQLite)	Native (SQLite)
1K	21.3	22.5	24.8	27.6	19.5	20.8
10K	49.8	52.7	55.2	61.3	46.7	48.5
100K	87.5	94.2	102.1	114.8	80.9	86.4
1M	128.4	134.7	140.2	156.9	115.6	121.3

TABLE II
AVERAGE MEMORY USAGE (MB) PER DATASET SIZE

Dataset Size	PDO (MySQL)	Native (MySQL)	PDO (PostgreSQL)	Native (PostgreSQL)	PDO (SQLite)	Native (SQLite)
1K	6.78	7.21	7.10	7.80	6.32	6.61
10K	7.40	8.12	7.66	8.54	6.58	7.03
100K	8.00	8.89	7.91	8.97	6.73	7.20
1M	8.21	9.03	7.84	8.99	6.73	7.20

TABLE III
AVERAGE EXECUTION TIME (MS) AND MEMORY USAGE (MB) PER DBMS

DBMS	PDO (Execution Time, ms)	Native (Execution Time, ms)	PDO (Memory, MB)	Native (Memory, MB)
MySQL	71.75	76.03	7.60	8.31
PostgreSQL	80.58	90.15	7.63	8.57
SQLite	65.68	69.25	6.59	7.01

C. Benchmarking Performance per CRUD Operation

To analyze how each connection method performs under specific database operations, further benchmarking was conducted across the four fundamental CRUD processes. Each operation was executed repeatedly under identical conditions using the same hardware, software environment, and dataset size (100K records). The results presented in Table IV summarize the average execution time (in milliseconds) and Table V shows the average memory utilization (in bytes) for both PDO and Native implementations across three DBMS platforms: MySQL, PostgreSQL, and SQLite.

The Insert operation demonstrated that PDO remarkably perform Native connections in all DBMS. In MySQL, PDO completed insertion in 112.47 ms, roughly 5.4% faster than Native's 118.93 ms. A similar pattern appeared in PostgreSQL, where PDO's 125.68 ms was 9.6% faster than the Native implementation. SQLite, being file-based, exhibited the fastest Insert overall, with PDO recording 97.42 ms, surpassing Native by 6.5 ms. In terms of memory, PDO also showed a noticeable reduction across all DBMS, saving approximately 7%–10% of memory relative to Native. The performance of PDO in this phase is attributed to its efficient handling of parameterized queries [14], which minimizes overhead during batch insertions.

In the Select operation, PDO again showed an advantage in both speed and efficiency. MySQL's PDO query time was 86.15 ms, compared to 91.42 ms for Native; PostgreSQL exhibited 92.37 ms for PDO and 101.28 ms for Native, while SQLite demonstrated 74.58 ms vs 78.90 ms, respectively. These results indicate that PDO's prepared statement caching substantially reduces redundant parsing and improves retrieval speed, particularly for repeated queries. The memory footprint followed a similar trend, with PDO consuming up to 7% less memory on average, reflecting better optimization in handling result sets and buffer allocation.

TABLE IV
AVERAGE EXECUTION TIME (MS) OF PDO VS NATIVE FOR 100K RECORDS

Operation	MySQL (PDO)	MySQL (Native)	PostgreSQL (PDO)	PostgreSQL (Native)	SQLite (PDO)	SQLite (Native)
Insert	112.47	118.93	125.68	137.85	97.42	103.74
Select	86.15	91.42	92.37	101.28	74.58	78.90
Update	98.36	104.22	107.80	116.35	88.94	92.60
Delete	83.72	89.16	89.55	95.47	71.63	74.80

TABLE V
AVERAGE MEMORY UTILIZATION (MB) OF PDO VS NATIVE FOR 100K RECORDS

Operation	MySQL (PDO)	MySQL (Native)	PostgreSQL (PDO)	PostgreSQL (Native)	SQLite (PDO)	SQLite (Native)
Insert	7.88	8.46	7.95	8.92	6.71	7.04
Select	7.51	8.07	7.66	8.57	6.39	6.84
Update	7.84	8.52	7.92	8.91	6.64	7.00
Delete	7.39	8.09	7.48	8.30	6.24	6.73

For the Update operation, PDO exhibited better performance stability across DBMS. MySQL recorded 98.36 ms for PDO versus 104.22 ms for Native (≈6% improvement), while PostgreSQL recorded 107.80 ms versus 116.35 ms (≈7.9% faster). SQLite again presented the best overall speed, with 88.94 ms for PDO and 92.60 ms for Native. In addition, PDO's memory utilization averaged 7.8 MB, consistently lower than Native's 8.5 MB, underscoring PDO's efficiency in executing repeated write operations. These findings suggest that PDO's internal execution plan reuse benefits performance during updates, particularly in systems that handle large transactional workloads.

For the Delete operation, PDO maintained its advantage with faster response times and lower resource use. In MySQL, PDO averaged 83.72 ms, surpassing Native's 89.16 ms. PostgreSQL exhibited similar trends (89.55 ms vs 95.47 ms), while SQLite recorded the lowest latency overall (71.63 ms vs 74.80 ms). The memory utilization followed a similar pattern: PDO required approximately 7.39 MB, while Native averaged 8.09 MB. This consistent efficiency highlights how PDO optimizes command execution by reducing statement-preparation overhead and improving transaction-scoped management.

When observed collectively, PDO demonstrates a consistent 5–10% improvement in execution time and approximately 7–11% reduction in memory consumption across all CRUD operations and DBMS. The performance margin is most prominent in PostgreSQL, where PDO's abstraction layer interacts favorably with the DBMS's complex query planner, while SQLite remains the fastest overall due to its minimal I/O overhead. MySQL occupies a middle ground, with PDO providing balanced gains in speed and efficiency. Advanced optimizers and tuning systems further enhance PostgreSQL's efficiency, reducing query execution times by up to 70% in some benchmarks [21,22].

To obtain a more comprehensive overview of performance characteristics, execution time measurements across dataset sizes, DBMS platforms, and CRUD operations were consolidated into a unified dataset comprising 27 observations per implementation approach. Descriptive statistics were subsequently computed to capture central tendency and dispersion patterns, as shown in Table VI. The aggregated results indicate that PDO has a lower overall mean execution time (82.54 ms) than the Native implementation (88.22 ms). The minimum observed values are comparable between the two approaches (19.50 ms for PDO and 20.80 ms for Native), while the maximum execution time for PDO (140.20 ms) is notably lower than that for Native (156.90 ms).

As shown in Table VI, PDO demonstrates lower sample variance (995.04) and standard deviation (31.54 ms) than Native (variance = 1174.91; standard deviation = 34.28 ms). This pattern suggests that execution time under PDO appears slightly more stable across the evaluated experimental configurations. While the observed differences are moderate in magnitude, the descriptive statistics consistently indicate that PDO provides execution times that are comparable and, in several scenarios, marginally lower under the tested conditions.

To further evaluate resource efficiency, memory consumption measurements across dataset sizes, DBMS platforms, and CRUD operations were consolidated into a unified dataset consisting of 27 observations for each implementation approach. The reported values represent memory usage in megabytes (MB), as shown in Table VII. The aggregated descriptive statistics show that PDO has a lower overall mean memory consumption (7.29 MB) than the Native implementation (7.96 MB). The minimum observed memory usage for PDO is 6.25 MB, slightly lower than Native at 6.61 MB, while the maximum recorded usage for PDO (8.21 MB) remains below that of Native (9.04 MB).

PDO demonstrates lower sample variance (0.37) and standard deviation (0.61 MB) than Native (variance = 0.68; standard deviation = 0.83 MB), as shown in Table VII. This indicates that memory utilization under PDO appears comparatively more consistent across varying experimental configurations. Although the magnitude of the difference is moderate, the descriptive results consistently suggest that PDO maintains comparable memory consumption characteristics and, in several scenarios, slightly lower ones within the evaluated environment.

TABLE VI
STATISTIC VALUE FOR NATIVE AND PDO PERFORMANCE BENCHMARKING

Statistic	Native	PDO
n	27	27
Sum	2381.85	2228.68
Mean	88.22	82.54
Min	20.80	19.50
Max	156.90	140.20
Variance	1174.91	995.04
Standard Deviation	34.28	31.54

Inferential statistical analysis was conducted using a paired t-test across 27 identical experimental configurations. The test results indicate a statistically significant difference in memory consumption between PDO and Native implementations ($t(26)=14.33$, $p<0.0001$). PDO demonstrated lower average memory usage across the evaluated scenarios, with a mean difference of approximately 0.68 MB.

The integrated benchmarking results provide a broader perspective on the performance characteristics of PDO and Native database implementations across multiple experimental configurations. When execution time and memory usage are considered together, PDO consistently demonstrates slightly lower average values and comparatively smaller dispersion across the evaluated scenarios. While the magnitude of these differences is moderate, the observed pattern appears stable across dataset sizes, DBMS platforms, and CRUD operations. From a performance standpoint, the lower overall mean execution time observed in PDO suggests that abstraction layers do not necessarily impose substantial overhead under controlled workloads. This observation aligns with prior discussions, indicating that modern database abstraction interfaces are designed to minimize performance penalties while improving portability and maintainability [23]. In practical web system development, execution efficiency must often be balanced with architectural flexibility, and abstraction mechanisms may offer trade-offs that are not strictly detrimental to runtime performance.

PDO also exhibits slightly lower mean usage and reduced variability. The relatively smaller standard deviation suggests more consistent resource allocation behavior across heterogeneous test conditions. Previous studies on middleware and database abstraction layers have noted that structured connection handling and prepared statement mechanisms can contribute to predictable resource management patterns [24].

TABLE VII
STATISTIC VALUE FOR NATIVE AND PDO MEMORY CONSUMPTION

Statistic	Native	PDO
n	27	27
Sum	215.05	196.81
Mean	7.96	7.29
Min	6.61	6.25
Max	9.04	8.21
Variance	0.68	0.37
Standard Deviation	0.83	0.61

These findings suggest that the choice between PDO and Native implementations should not be determined solely by assumptions of raw performance. Factors such as portability, maintainability, security abstraction, and long-term scalability are frequently highlighted in software engineering literature as critical architectural considerations [25]. Within this broader context, the observed performance tendencies indicate that adopting PDO may not entail substantial execution or memory penalties under moderate workloads.

When comparing across DBMS, the empirical evidence highlights the synergistic interaction between PDO's abstraction layer and each DBMS's internal architecture. PostgreSQL exhibited the most substantial gain, suggesting that PDO's parameter binding and plan caching align closely with PostgreSQL's native optimization strategies. In MySQL, the performance improvement of PDO was moderate but consistent, suggesting that MySQL's engine already benefits from lightweight query parsing [8,12]. SQLite demonstrated the lowest overall execution times, while PDO maintained efficiency even in its file-based environment.

In terms of memory utilization, PDO's efficiency—up to 11% lower than Native—reflects better memory allocation and garbage collection within its unified data abstraction model. Efficient memory usage directly contributes to higher scalability, as the system can handle a greater number of concurrent requests without exhausting available resources. This result aligns with software engineering principles emphasizing modularity and reusability [26], where well-designed abstraction layers can yield both maintainability and runtime efficiency.

The inferential analysis supports the descriptive observations that PDO tends to exhibit lower memory consumption under the evaluated experimental conditions. Although the magnitude of differences remains moderate in practical terms, the consistency across paired configurations suggests that the observed pattern is unlikely to result solely from random variation within the tested environment.

Another critical insight from the benchmarking results is the stability and scalability of PDO under increasing workloads. Unlike Native connections, which showed a linear rise in execution time and memory usage with dataset growth, PDO exhibited a more controlled scalability curve. This implies that PDO can sustain higher data throughput with predictable resource behavior, making it suitable for web applications that must support large and concurrent data operations. In real-world contexts—such as e-commerce or institutional information systems—this stability is

essential for maintaining user experience and operational reliability [24,27].

IV. CONCLUSION

This study presented an empirical comparative evaluation of PHP Data Objects (PDO) and Native database implementations across multiple dataset sizes, DBMS platforms, and CRUD operations. Based on descriptive statistics across 27 experimental configurations, PDO exhibited slightly lower average execution time and memory consumption, with lower variability than Native implementations. A paired t-test further indicated that the observed difference in memory consumption was statistically significant under the evaluated experimental configurations ($p < 0.0001$), although the practical magnitude of the difference remained moderate. These findings suggest that, within the controlled experimental environment, PDO provides performance characteristics comparable to, and in several scenarios marginally more efficient than, Native implementations while offering the architectural advantages of abstraction, portability, and maintainability. Nevertheless, this study is limited to controlled benchmarking using three relational DBMSs and moderate workload scenarios, and the results should therefore be interpreted within these experimental conditions. Future work may extend this evaluation to contemporary PHP environments, high-concurrency and cloud-native deployments, NoSQL and hybrid database systems, and investigate the integration of intelligent techniques, including machine learning, to support adaptive query optimization and resource management in modern PHP-based applications.

ACKNOWLEDGMENTS

We would like to thank the Institute for Research and Community Service, UIN Prof. K.H. Saifuddin Zuhri, Purwokerto, for funding this research under the Basic Research Study Program 2025.

REFERENCES

[1] J. F. Zimmermann, "Data Management and Processing in Toxicoinformatics: From Chemical Databases to Automatic Extraction of Unstructured Resources." Computational Systems Toxicology, 2015. doi: 10.1007/978-1-4939-2778-4_5.

[2] A.-M. Bonteanu and C. Tudose, "Performance Analysis and Improvement for CRUD Operations in Relational Databases from Java Programs Using JPA, Hibernate, Spring Data JPA," *Appl. Sci.*, Mar. 2024, doi: 10.3390/app14072743.

[3] D. Zmaranda, L. L. Pop-Fele, C. Gyorödi, R. Gyorödi, and G. Pecherle, "Performance Comparison of CRUD Methods Using NET Object Relational Mappers: A Case Study'," *Int. J. Adv. Comput. Sci. Appl.*, vol. 11, no. 1, pp. 55–65, 2020, doi: 10.14569/ijacsa.2020.0110107.

[4] D. Popel, "Learning PHP Data Objects." 2007.

[5] S. O. and E.-Y. B., "Neutralizing SQL Injection Attack on Web Application Using Server Side Code Modification," *Int. J. Sci. Res. Comput. Sci. Eng. Inf. Technol.*, May 2019, doi: 10.32628/CSEIT1952339.

[6] M. Sendiang, O. Mellolo, and M. Langie, "Implementation pdo parameterized query to prevent sql injection," *Int. J. Comput. Appl.*, vol. 149, no. 11, pp. 27–31, 2016.

[7] E. Sierra and U. L. Yuhana, "Optimizing Software Development through Data Access Speed using Object-Relational Mapping (ORM) on Credit Risk Application," *2023 3rd Int. Conf. Smart Cities Autom. Intell. Comput. Syst. ICON-SONICS*, pp. 201–206, Dec. 2023, doi: 10.1109/icon-sonics59898.2023.10435255.

[8] T. Taipalus, "Database management system performance comparisons: A systematic literature review," *J. Syst. Softw.*, vol. 208, p. 111872, 2024, doi: https://doi.org/10.1016/j.jss.2023.111872.

[9] A. Pavlo and M. Aslett, "What's Really New with NewSQL?," *SIGMOD Rec*, vol. 45, pp. 45–55, Sep. 2016, doi: 10.1145/3003665.3003674.

[10] R. Jiménez-Peris, D. Burgos-Sancho, F. Ballesteros, M. Patiño-Martínez, and P. Valduriez, "Elastic scalable transaction processing in LeanXcale," *Inf Syst*, vol. 108, p. 102043, Mar. 2022, doi: 10.1016/j.is.2022.102043.

[11] W. B. Demilie and F. G. Deriba, "Detection and prevention of SQLI attacks and developing compressive framework using machine learning and hybrid techniques," *J. Big Data*, vol. 9, pp. 1–30, Dec. 2022, doi: 10.1186/s40537-022-00678-0.

[12] S. V. Salunke and A. Ouda, "A performance benchmark for the postgresql and mysql databases," *Future Internet*, vol. 16, no. 10, p. 382, 2024.

[13] R. Nambiar and M. Poess, "Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things," vol. 9508, Jan. 2016, doi: 10.1007/978-3-319-31409-9.

[14] M. Santoro, L. Vaccari, D. Mavridis, R. Smith, M. Posada, and D. Gattwinkel, "Web application programming interfaces (apis): General purpose standards, terms and European commission initiatives," *Eur Comm. Louxemb. Louxemb. UK Tech Rep JRC118082*, 2019.

[15] G. Allen and M. Owens, "The Definitive Guide to SQLite," May 2006, doi: 10.1007/978-1-4302-3226-1.

[16] K. Gaffney, M. Prammer, L. Brasfield, D. Hipp, D. Kennedy, and J. Patel, "SQLite," *Proc. VLDB Endow.*, Aug. 2022, doi: 10.1007/978-1-4302-0367-4_22.

[17] C. Li, Z. Zang, H. Lin, and H. Li, "Research on Display and Storage of Raster Data Based on Android Platform," *2018 Fifth Int. Workshop Earth Obs. Remote Sens. Appl. EORSA*, pp. 1–4, Jun. 2018, doi: 10.1109/eorsa.2018.8598571.

[18] C. Győrödi, D. Dumşe-Burescu, R. Győrödi, D. Zmaranda, L. Bandici, and D. Popescu, "Performance Impact of Optimization Methods on MySQL Document-Based and Relational Databases," *Appl. Sci.*, Jul. 2021, doi: 10.3390/app11156794.

[19] S. Mo, Y. Zhao, Z. Bao, Q. Xu, C. Yang, and G. Cong, "RankPQO: Learning-to-Rank for Parametric Query Optimization," *Proc VLDB Endow*, vol. 18, pp. 863–875, Nov. 2024, doi: 10.14778/3712221.3712248.

[20] L. Doshi, "Kepler: Robust Learning for Parametric Query Optimization," *Proc. ACM Manag. Data*, vol. 1, pp. 1–25, May 2023, doi: 10.1145/3588963.

[21] R. Zhu, "Lero: A Learning-to-Rank Query Optimizer," *Proc VLDB Endow*, vol. 16, pp. 1466–1479, Feb. 2023, doi: 10.14778/3583140.3583160.

[22] J. Lao, "GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization," *ArXiv*, vol. abs/2311.03157, Nov. 2023, doi: 10.14778/3659437.3659449.

[23] J. Smith and D. Smith, "Database abstractions," *ACM Trans. Database Syst. TODS*, vol. 2, pp. 105–133, Jun. 1977, doi: 10.1145/320544.320546.

[24] M. Natti, "Managing Connections Efficiently in PostgreSQL to Optimize CPU, I/O and Memory Usage," *Int. J. Sci. Res. Arch.*, Apr. 2025, doi: 10.30574/ijsra.2025.15.1.0650.

[25] J. Spray, R. Sinha, A. Sen, and X. Cheng, "Building Maintainable Software Using Abstraction Layering," *IEEE Trans. Softw. Eng.*, vol. 48, pp. 4397–4410, Nov. 2022, doi: 10.1109/TSE.2021.3119012.

[26] R. S. Pressman and B. R. Maxim, *Software Engineering A PRACTITIONER'S APPROACH*. McGraw-Hill, 2020.

[27] L. Ju, A. Yadav, A. Khan, A. P. Sah, and D. Yadav, "Using Asynchronous Frameworks and Database Connection Pools to Enhance Web Application Performance in High-Concurrency Environments," *2024 8th Int. Conf. -SMAC IoT Soc. Mob. Anal. Cloud - SMAC*, pp. 742–747, Oct. 2024, doi: 10.1109/i-smac61858.2024.10714639.