

Article

Adaptive and Scalable Database Management with Machine Learning Integration: A PostgreSQL Case Study

Maryam Abbasi ¹, Marco V. Bernardo ^{2,3}, Paulo Váz ^{3,4}, José Silva ^{3,4} and Pedro Martins ^{3,4,*}

¹ Applied Research Institute, Polytechnic of Coimbra, 3045-093 Coimbra, Portugal; maryam.abbasi@ipc.pt

² Instituto de Telecomunicações, 6201-001 Covilhã, Portugal; mbernardo@ubi.pt

³ Polytechnic of Viseu, Department of Informatics, 3504-510 Viseu, Portugal; paulovaz@estgv.ipv.pt (P.V.); jsilva@estgv.ipv.pt (J.S.)

⁴ Research Center in Digital Services (CISeD), Polytechnic of Viseu, 3504-510 Viseu, Portugal

* Correspondence: pedromom@estgv.ipv.pt

Abstract: The increasing complexity of managing modern database systems, particularly in terms of optimizing query performance for large datasets, presents significant challenges that traditional methods often fail to address. This paper proposes a comprehensive framework for integrating advanced machine learning (ML) models within the architecture of a database management system (DBMS), with a specific focus on PostgreSQL. Our approach leverages a combination of supervised and unsupervised learning techniques to predict query execution times, optimize performance, and dynamically manage workloads. Unlike existing solutions that address specific optimization tasks in isolation, our framework provides a unified platform that supports real-time model inference and automatic database configuration adjustments based on workload patterns. A key contribution of our work is the integration of ML capabilities directly into the DBMS engine, enabling seamless interaction between the ML models and the query optimization process. This integration allows for the automatic retraining of models and dynamic workload management, resulting in substantial improvements in both query response times and overall system throughput. Our evaluations using the Transaction Processing Performance Council Decision Support (TPC-DS) benchmark dataset at scale factors of 100 GB, 1 TB, and 10 TB demonstrate a reduction of up to 42% in query execution times and a 74% improvement in throughput compared with traditional approaches. Additionally, we address challenges such as potential conflicts in tuning recommendations and the performance overhead associated with ML integration, providing insights for future research directions. This study is motivated by the need for autonomous tuning mechanisms to manage large-scale, heterogeneous workloads while answering key research questions, such as the following: (1) How can machine learning models be integrated into a DBMS to improve query optimization and workload management? (2) What performance improvements can be achieved through dynamic configuration tuning based on real-time workload patterns? Our results suggest that the proposed framework significantly reduces the need for manual database administration while effectively adapting to evolving workloads, offering a robust solution for modern large-scale data environments.

Keywords: machine learning integration; database optimization; query performance; dynamic workload management; PostgreSQL; real-time system tuning



Citation: Abbasi, M.; Bernardo, M.V.; Váz, P.; Silva, J.; Martins, P. Adaptive and Scalable Database Management with Machine Learning Integration: A PostgreSQL Case Study. *Information* **2024**, *15*, 574. <https://doi.org/10.3390/info15090574>

Academic Editors: Shadi Banitaan and Mina Maleki

Received: 30 August 2024

Revised: 13 September 2024

Accepted: 16 September 2024

Published: 18 September 2024



Copyright: © 2024 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 (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The exponential growth of data in the modern era has made database management systems integral to efficiently storing and retrieving information. However, the increasing complexity and scale of these systems, especially in terms of optimizing query performance for large datasets, present significant challenges. Traditional database management techniques often struggle to cope with these challenges, particularly when dealing with dynamic and heterogeneous workloads. Manual tuning and configuration, while effective

in smaller or more stable environments, are becoming increasingly impractical and insufficient for modern, large-scale data environments. Furthermore, as workload patterns evolve rapidly, the need for real-time optimization becomes paramount.

Recent advancements in machine learning (ML) have introduced new opportunities for automating and enhancing various aspects of DBMS operations, including query optimization, indexing, and workload management. Several studies have demonstrated the potential of ML models to predict query execution times, optimize query plans, and automate the selection of indexes and configuration parameters. However, these efforts often focus on isolated aspects of database management, lacking a unified framework that simultaneously addresses multiple optimization tasks and adapts in real time to changing workloads. For instance, Ref. [1] highlights the use of ML in query optimization, and [2] applies ML to optimize performance, yet both focus on narrow tasks within the broader scope of DBMS operations. Our work builds on these efforts by offering a holistic, real-time approach.

This study is motivated by two primary research questions: (1) How can machine learning models be integrated into a DBMS to improve query optimization and workload management? (2) What performance gains can be achieved through dynamic configuration tuning based on real-time workload patterns? These questions stem from the increasingly complex demands of modern databases, where workload diversity and scale necessitate sophisticated optimization strategies.

In this paper, we propose a comprehensive framework that integrates advanced ML models within the architecture of a DBMS, with a particular focus on PostgreSQL. Our framework is designed to overcome the limitations of existing approaches by offering a tightly coupled platform that enables real-time ML model inference and dynamic adjustment of database configurations based on ongoing workload patterns. We employ a combination of supervised learning techniques, such as linear regression and random forest, and unsupervised methods, including K-means clustering, to accurately predict query execution times, identify patterns in query workloads, and suggest optimization strategies that improve overall system performance. This combination ensures that both historical data patterns and unseen data behaviors are accounted for in the optimization process.

A key innovation of our framework is the seamless integration of ML models with the DBMS engine, allowing for real-time data ingestion, automated model retraining, and continuous adaptation to workload changes. This approach not only minimizes the need for manual database administration but also enhances the DBMS's ability to maintain optimal performance in diverse and evolving environments. By integrating these ML capabilities directly into the query optimization process, our framework achieves significant improvements in both query response times and system throughput.

The contributions of this paper are as follows:

- We propose a comprehensive ML-driven framework that addresses multiple aspects of database optimization, including query performance prediction, workload management, and dynamic configuration tuning.
- We demonstrate a novel approach for integrating ML models within the PostgreSQL DBMS engine, facilitating real-time inference and optimization.
- We apply advanced feature engineering techniques tailored for database environments, enabling more accurate and effective ML model predictions by considering factors such as query structure, data distribution, and system state.
- We conduct extensive evaluations by using the Transaction Processing Performance Council Decision Support (TPC-DS) benchmark to showcase the scalability and adaptability of our framework, with significant improvements in query execution times and system throughput.
- We address potential limitations, such as conflicts in tuning recommendations and the overhead of ML integration, providing insights for future research directions. Specifically, we explore the trade-offs involved in integrating ML models into a DBMS and identify future directions to enhance model retraining and scalability.

The rest of this paper is organized as follows: Section 2 provides a comprehensive review of related work. Section 3 details our system architecture and integration approach. Section 4 describes our ML methodology, including feature selection, model training, and optimization strategies. Section 5 presents our experimental setup and evaluation results. Finally, Section 6 concludes the paper and discusses future research directions.

2. Related Work

The application of machine learning to database management systems has seen significant advancements over the past decade, driven by the increasing complexity of data and the need for more efficient, adaptive optimization techniques. This section reviews the state of the art in ML-driven database optimization, focusing on query optimization, workload management, automated database tuning, and the integration of ML within DBMS architectures. We also discuss the limitations of existing approaches and how our proposed framework seeks to address these challenges.

2.1. Query Optimization

Query optimization is a cornerstone of DBMS performance, and significant research has been conducted on leveraging ML to enhance this process. Traditional query optimizers rely on cost-based methods that use predefined rules and heuristics to estimate the cost of executing different query plans. However, these methods often fall short in handling the complexities of modern workloads and data distributions.

Recent studies have explored various ML approaches to address these limitations. For instance, Yu et al. [3] introduced a reinforcement learning-based technique that adapts query plan selection based on feedback from previous executions. This approach has shown improvements over traditional cost-based optimizers, particularly in dynamic environments. However, reliance on a high number of training data and the computational overhead of training models remain significant challenges.

Cardinality estimation, a crucial aspect of query optimization, has also benefited from ML advancements. Kwon et al. [4] proposed a deep learning model to improve the accuracy of cardinality estimates, which are often inaccurate in traditional optimizers due to assumptions about data distributions. Their approach reduces errors in estimation, leading to better query plan choices. Nevertheless, the model's performance heavily depends on the availability of large, labeled datasets for training, which may not always be feasible.

Join order optimization is another critical area where ML has been applied. Li et al. [5] developed an ML model that predicts the optimal join order by learning from past query executions. While their model shows promise, it focuses narrowly on join orders without considering other factors that influence query performance. Heitz et al. [6] extended this idea by using deep reinforcement learning to optimize entire query plans, including join orders. Their method, however, requires substantial computational resources for training and may not generalize well to different database environments.

Beyond individual optimization tasks, recent efforts have explored more holistic approaches. Paganelli et al. [7] demonstrated how ML predictions could be integrated directly into DBMSs for real-time query optimization. Their approach minimizes the overhead typically associated with external ML services by embedding ML models within the DBMS. However, it still requires fine tuning for complex, real-time workloads.

Additionally, in [1], the authors explore the use of PostgreSQL and ML for query optimization, showing significant performance gains in decision support systems, which aligns with our approach. Similarly, the study by [2] illustrates the effectiveness of ML algorithms for query optimization and workload prediction in real-time scenarios, further motivating our integrated framework.

2.2. Workload Management

Workload management is crucial to maintaining DBMS performance, particularly in environments with dynamic and heterogeneous workloads. Traditional approaches often

rely on static configurations that are ill suited to adapt to changing workloads, leading to suboptimal performance.

ML has been increasingly applied to improve workload management. Shaheen et al. [8] developed an ML-based workload classification system that categorizes incoming queries into different workload classes. This classification helps the DBMS optimize resource allocation for each class, improving overall system efficiency. However, their approach does not extend to making specific optimization recommendations for each workload class, limiting its impact on query performance.

Automated tuning services such as OtterTune [9] have also employed ML to enhance workload management. OtterTune uses ML to analyze historical workload data and optimize DBMS configuration parameters accordingly. While this approach shows significant improvements in performance, it primarily focuses on static parameter tuning rather than dynamic, workload-specific optimization. As a result, it may struggle to adapt quickly to sudden changes in workload patterns.

In addition to workload classification and parameter tuning, other studies have explored ML-based admission control. Xia et al. [10] proposed a system that uses ML to control the admission of queries into the DBMS based on the current system load and query characteristics. This approach helps prevent system overloads and ensures more consistent performance. However, it does not integrate with broader optimization strategies such as query plan selection or resource allocation.

The need for dynamic workload adaptation is further highlighted in [1], where ML models dynamically adapt to workload changes in PostgreSQL environments. This real-time adaptability also forms a key element of our framework.

2.3. Automated Database Tuning

As DBMS scale and complexity increase, automated tuning has become a critical area of research. Traditional tuning methods, which rely on manual configuration and expert knowledge, are increasingly inadequate for managing the vast number of parameters and interactions in modern DBMSs.

ML techniques have been successfully applied to various aspects of automated database tuning. Siddiqui et al. [11] proposed an ML-based approach for automated index selection. Their system learns from past query executions to recommend indexes that improve query performance. While this approach can outperform traditional heuristics, it does not account for the interactions between index selection and other optimization techniques, such as query plan selection or buffer pool management.

Buffer pool management, another critical tuning task, has also benefited from ML advancements. Tan et al. [12] introduced iBTune, an ML-driven buffer pool tuning system that dynamically adjusts memory allocation based on workload characteristics. This approach improves memory utilization and query performance but is limited to a single aspect of database tuning.

Parameter tuning, which involves adjusting DBMS configuration settings, has been a focus of several studies. Zhang et al. [13] developed OtterTune, an automated tuning service that uses ML to optimize configuration parameters. OtterTune analyzes historical performance data to identify optimal settings, reducing the need for manual tuning. However, like other solutions, OtterTune primarily focuses on static tuning and may not adapt quickly to changing workloads.

Our work integrates ML-driven workload management with automated tuning approaches, leveraging the dynamic tuning capabilities presented in these studies to create a more comprehensive optimization framework.

2.4. Integration of Machine Learning within DBMS Architectures

The integration of ML models directly into DBMS architectures is a relatively new area of research, aiming to bring the benefits of ML-driven optimization closer to the core of the

DBMS. Traditional approaches often treat ML models as external components, which can introduce latency and integration challenges.

Several studies have explored embedding ML models within DBMSs to reduce these challenges. Marcus et al. [14] pioneered the concept of “learned indexes”, where ML models replace traditional B-tree indexes. This approach demonstrated significant performance improvements for specific workloads but required specialized models and did not generalize to other DBMS functions.

Paganelli et al. [7] extended this concept by integrating ML models into the query optimization process, enabling real-time adjustments based on workload dynamics. While their approach minimizes the overhead of using external ML services, it requires careful tuning to handle the complexity of real-time query optimization.

Building on these advancements, our framework seeks to push the boundaries of real-time integration by embedding ML models directly into PostgreSQL’s core. By combining both supervised and unsupervised learning techniques, our system adapts dynamically to workload changes, optimizing multiple aspects of query execution and database configuration in real time.

2.5. Gaps and Contributions

Despite the significant advancements in ML-driven DBMS optimization, several gaps remain unaddressed. Many existing solutions focus on specific aspects of database management, such as query optimization, workload management, or parameter tuning, without offering a comprehensive approach that simultaneously addresses multiple optimization tasks. Additionally, few approaches provide the real-time integration necessary to adapt quickly to changing workloads or data distributions. Moreover, the feature representations used in many ML models for DBMS optimization often fail to capture the full complexity of database operations, limiting the effectiveness of these models.

Our proposed framework addresses these gaps by offering a unified solution that integrates multiple aspects of database optimization—query performance prediction, workload management, and configuration tuning—into a single, real-time platform. We also employ advanced feature engineering techniques that capture the complexity of query structures, execution plans, and system states, leading to more accurate and effective ML-driven optimization.

In summary, our work advances the state of the art in ML-driven database management by providing a comprehensive, real-time solution that addresses the limitations of existing approaches. Our framework not only enhances query performance but also improves overall system throughput and adaptability, making it a robust solution for modern, large-scale data environments.

3. System Architecture

This section presents the detailed architecture of our proposed machine learning (ML)-integrated database management system (DBMS) framework, which is designed to enhance query performance, automate database management tasks, and dynamically adapt to workload variations. The architecture is built on PostgreSQL (version 13.0) and is modular, allowing for the seamless integration of various ML models and supporting real-time adjustments to database configurations. The architecture is structured into three primary layers: the Database Management Layer, the Machine Learning Integration Layer, and the User Interface and Visualization Layer. Each layer is designed to function independently, ensuring flexibility, scalability, and ease of maintenance. We have integrated dynamic clustering mechanisms to manage workload patterns, utilizing methods such as K-means for workload classification.

3.1. Overview of Architecture

The system architecture is depicted in Figure 1. This architecture ensures tight integration between ML models and the DBMS, facilitating real-time query optimization

and dynamic system adjustments. The modular design allows for independent updates to components without disrupting overall system functionality, thus supporting long-term scalability and adaptability. Our approach to workload management adapts dynamically, with clustering techniques ensuring optimal workload distribution across system resources.

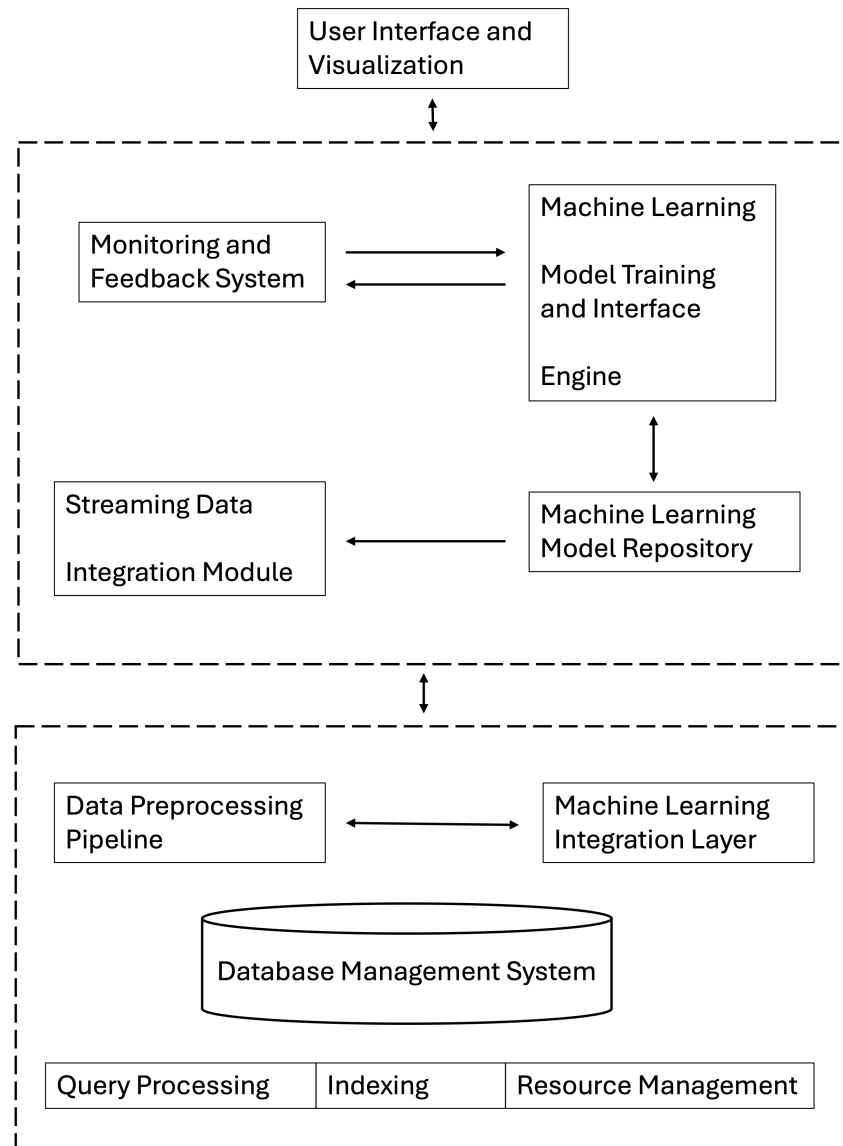


Figure 1. System architecture: interaction among Database Management Layer, Machine Learning Integration Layer, and User Interface and Visualization Layer.

The architecture consists of the following layers:

1. **Database Management Layer:** It handles core database functions such as query processing, indexing, and resource management. This layer is built on PostgreSQL and is extended with custom functionalities to integrate ML capabilities.
2. **Machine Learning Integration Layer:** It bridges the DBMS with advanced ML capabilities, enabling real-time model inference, dynamic database configuration adjustments, and continuous learning from system performance. This layer also includes mechanisms for determining the optimal number of clusters in workload patterns, utilizing the elbow method and silhouette analysis for validation.
3. **User Interface and Visualization Layer:** It provides tools for monitoring, visualizing, and managing the database system, ensuring that database administrators (DBAs) can effectively oversee the system's performance and interact with the ML components.

3.2. Database Management Layer

The Database Management Layer forms the foundation of our system, managing essential database operations such as query processing, indexing, and resource allocation. Built on PostgreSQL, this layer is augmented with custom extensions, including user-defined functions (UDFs), planner hooks, and a specialized data exchange mechanism to integrate machine learning seamlessly.

3.2.1. Custom Query Planner Hooks

A key innovation within this layer is the custom `planner_hook`. This hook intercepts SQL queries before plan generation, allowing the system to incorporate ML-based cost estimates and optimization suggestions directly into the query planning process. By dynamically adjusting query execution plans based on real-time ML predictions, the system significantly improves query performance. This process also incorporates the predicted number of clusters for the workload pattern, allowing for more efficient resource allocation based on workload classification.

Below (Listing 1), we show how the custom `planner_hook` is implemented.

Listing 1. Custom planner hook implementation.

```

1 static PlannedStmt *
2 custom_planner_hook(Query *parse, int cursorOptions, ParamListInfo boundParams)
3 {
4     PlannedStmt *result;
5
6     // Call the standard planner
7     result = standard_planner(parse, cursorOptions, boundParams);
8
9     // Extract query features for ML model
10    extract_query_features(result);
11
12    // Generate ML-based cost estimates
13    double ml_cost_estimate = get_ml_cost_estimate(result);
14
15    // Inject ML cost into the plan
16    result->total_cost = ml_cost_estimate;
17
18    return result;
19 }

```

This hook's flexibility allows it to support various ML models and optimization strategies, making it adaptable to different database environments and specific use cases.

3.2.2. User-Defined Functions (UDFs) for Real-Time Feature Extraction

To enable effective ML-driven optimization, the system requires the real-time extraction of relevant query features. We developed several UDFs that analyze incoming queries and extract essential features, such as join types, aggregation presence, table sizes, and data distribution characteristics. These features are then used by the ML models to predict query execution times and generate optimization strategies. These features are also critical to determining the optimal number of clusters for workload distribution.

An example UDF for extracting join types is shown below (Listing 2).

Listing 2. UDF for extracting join types.

```

1 CREATE OR REPLACE FUNCTION extract_join_type(query TEXT)
2 RETURNS TEXT AS $$
3 BEGIN
4   -- Analyze the query to determine the join type
5   IF query ~* 'INNER_JOIN' THEN
6     RETURN 'INNER';
7   ELSIF query ~* 'LEFT_JOIN' THEN
8     RETURN 'LEFT';
9   ELSIF query ~* 'RIGHT_JOIN' THEN
10    RETURN 'RIGHT';
11  ELSE
12    RETURN 'UNKNOWN';
13  END IF;
14  END;
15 $$ LANGUAGE plpgsql;

```

These UDFs are invoked during the query planning phase, ensuring that the ML models have access to all necessary contextual information, thereby improving the accuracy and effectiveness of their predictions.

3.2.3. PostgreSQL Extension for Data Exchange

Efficient communication between PostgreSQL and the ML components is critical to maintaining low-latency, high-performance operations. To achieve this, we developed a custom PostgreSQL extension, `pg_mlopt`, which uses shared memory mechanisms to facilitate rapid data exchange. This extension ensures that real-time optimization can be applied without introducing significant overhead, preserving the overall performance of the DBMS.

3.3. Machine Learning Integration Layer

The Machine Learning Integration Layer is the intelligence hub of the architecture, connecting the DBMS's core functionalities with advanced ML models. This layer is responsible for real-time model inference, dynamic database configuration adjustments, and continuous learning based on system performance data.

3.3.1. Query Interception and Feature Extraction

This component intercepts queries via the `planner_hook` and extracts relevant features in real-time by using the UDFs defined in the Database Management Layer. These features are crucial to predicting execution times and suggesting optimization. Additionally, clustering-based features are extracted to classify workloads and adapt system resources accordingly.

The following snippet illustrates how query features are processed (Listing 3).

Listing 3. Processing query features for ML integration.

```

1 void process_query_features(Query *query)
2 {
3   FeatureSet features = extract_features(query);
4   ModelPrediction prediction = run_ml_model(features);
5
6   if (prediction.requires_tuning)
7   {
8     adjust_query_plan(query, prediction);
9   }
10 }

```


This approach ensures that each query is optimized based on the latest data and the current state of the system, leading to enhanced query execution efficiency.

3.3.2. Real-Time Communication with ML Models

PostgreSQL's background worker processes maintain persistent connections with the ML Inference Engine, enabling low-latency predictions and optimization suggestions. These background workers scale with demand, ensuring that the system can handle high workloads without performance degradation.

The following code snippet demonstrates the setup for ML inference (Listing 4).

Listing 4. Background worker for ML inference.

```

1 void start_background_worker()
2 {
3     BackgroundWorker bgw;
4     bgw.bgw_name = "ML_Inference_Engine_Connector";
5     bgw.bgw_main = ml_inference_main;
6     bgw.bgw_restart_time = BGW_NEVER_RESTART;
7     RegisterBackgroundWorker(&bgw);
8 }
9
10 void ml_inference_main(Datum main_arg)
11 {
12     // Establish connection to ML inference engine
13     connect_to_ml_engine();
14
15     while (!got_sigterm)
16     {
17         // Wait for query processing requests
18         process_query_requests();
19     }
20 }
```

This setup ensures that the ML models continuously provide optimization feedback, allowing the system to dynamically adapt to workload variations. The workload classification based on clustering results is also processed in real time, allowing for efficient resource management.

3.3.3. Integration of ML-Based Recommendations

Once the ML models process the extracted features, the system modifies the query execution plans based on the predictions. This includes adjustments to join orders, access methods, and resource allocation to optimize performance.

Below (Listing 5), we show an example of how query plans are adjusted based on ML recommendations.

These modifications ensure that the system remains responsive to changes in query workloads and data distributions, consistently delivering optimal performance.

3.3.4. Dynamic Configuration Adjustments

This component enables the system to make real-time adjustments to the DBMS's configuration settings, such as `work_mem` and `effective_cache_size`, based on the current workload.

The following snippet (Listing 6) shows how dynamic configuration adjustments are made.

These adjustments help maintain system performance by optimizing resource allocation in response to varying workload demands. Additionally, resource allocation is

dynamically adjusted based on clustering insights, ensuring efficient utilization of system resources.

Listing 5. Modifying query plan based on ML recommendations.

```

1 void adjust_query_plan(PlannedStmt *stmt, ModelPrediction prediction)
2 {
3     if (prediction.suggested_join_order)
4     {
5         reorder_joins(stmt, prediction.join_order);
6     }
7
8     if (prediction.change_access_method)
9     {
10        use_index_scan(stmt);
11    }
12
13    // Apply further adjustments based on ML prediction
14 }

```

Listing 6. Dynamic configuration adjustment in PostgreSQL.

```

1 if (workload_intensity == HIGH)
2 {
3     set_config("work_mem", "256MB", PGC_SUSET, PGC_S_SESSION);
4 }
5 else
6 {
7     set_config("work_mem", "64MB", PGC_SUSET, PGC_S_SESSION);
8 }

```

3.4. User Interface and Visualization Layer

The User Interface and Visualization Layer provides the necessary tools for DBAs to monitor system performance, interact with ML components, and manage database configurations. This layer is critical to ensuring usability, transparency, and accessibility as the system automates complex tasks.

3.4.1. Monitoring and Feedback System

Our architecture integrates Prometheus for performance metrics collection and Grafana for real-time visualization. The monitoring system tracks key performance indicators (KPIs) such as query execution times, CPU usage, memory utilization, and disk I/O, enabling DBAs to understand the system's current state and make informed decisions.

An example of Prometheus configuration for capturing query execution times is shown below (Listing 7).

Listing 7. Prometheus configuration for query execution metrics.

```

1 - job_name: 'postgresql'
2 static_configs:
3 - targets: ['localhost:9187']
4 metrics_path: '/metrics'
5 scrape_interval: '5s'

```

This configuration ensures that Prometheus scrapes metrics from PostgreSQL at regular intervals, keeping performance data up to date for analysis.

3.4.2. Visualization Tools

The visualization tools in this layer offer comprehensive graphical representations of system operations, including workload patterns, query execution plans, and the effects of ML-driven optimization. These tools help DBAs visualize the system's decisions and the impact of various optimization strategies on performance.

For instance, a Grafana dashboard might display query execution times before and after applying ML-based optimization, providing clear evidence of the effectiveness of the system's recommendations. This includes visualizations for workload clustering results, illustrating the impact of optimal cluster selection on performance.

3.4.3. Automated Feedback Loops

Automated feedback loops are implemented to continuously improve the ML models by triggering retraining or hyperparameter tuning based on performance metrics. As the system processes more queries and workloads, it gathers data that can be used to refine the ML models, ensuring they remain accurate and effective over time.

These feedback loops are critical to maintaining the system's long-term adaptability, allowing it to evolve alongside changing data distributions and workload characteristics.

3.5. Data Flow and Interaction

The data flow within the system is designed to ensure seamless interaction between the DBMS and the ML components, enabling real-time optimization and continuous learning. The data flow proceeds as follows:

1. **Query submission:** A query is submitted to PostgreSQL and intercepted by the ML Integration Layer via the custom `planner_hook`.
2. **Feature extraction:** The query is analyzed, and features are extracted in real time by using the UDFs. These features include key attributes such as join types, aggregation presence, and table sizes. Workload features for clustering are also extracted to ensure optimal resource allocation.
3. **Data preprocessing:** The extracted features are processed by the Data Preprocessing Pipeline, where transformations like normalization and encoding are applied to prepare the data for ML inference.
4. **ML inference:** The preprocessed features are passed to the ML Model Training and Inference Engine, which uses the current best-performing model to make predictions or provide optimization suggestions.
5. **Plan modification:** Based on the ML insights, the system modifies the query execution plan. Adjustments may include reordering joins, changing access methods, or modifying resource allocation.
6. **Query execution:** The modified plan is executed by PostgreSQL with the system monitoring performance metrics in real-time.
7. **Performance analysis:** The Monitoring and Feedback System analyzes the collected metrics, comparing actual performance against predictions and identifying any anomalies.
8. **Continuous learning:** The Streaming Data Integration Module continuously updates the ML models with new data, including query performance metrics and changes in data distributions.
9. **Model update:** The ML Model Training and Inference Engine periodically retrains models or fine-tunes hyperparameters to maintain optimal performance based on the accumulated data and performance analysis.

This structured data flow ensures that the system operates efficiently and effectively, providing real-time optimization while continuously learning and improving from its interactions with the database environment.

3.6. Scalability and Extensibility

Our architecture is designed with scalability and extensibility in mind. The system can handle increasing workloads by scaling its background processes and ML model inference capabilities. As the number of data and query complexity grow, the architecture can adapt by adding more computational resources or optimizing the existing ones.

The modular design allows for the easy integration of new ML models, feature extraction methods, or database optimization techniques. This extensibility ensures that the system can evolve with advancements in both database technology and machine learning, maintaining its relevance and effectiveness over time.

3.7. Security and Privacy Considerations

Integrating ML into DBMS architectures brings numerous benefits, but it also raises concerns regarding data security and privacy. Our architecture includes mechanisms for securing data as they move between the DBMS and ML components. Data are encrypted during transmission, and access to sensitive information is tightly controlled through role-based access controls (RBACs).

Additionally, we implement privacy-preserving techniques such as differential privacy to ensure that the ML models do not inadvertently expose sensitive information during the optimization process. These measures are critical to ensuring that the system can be deployed in environments with stringent security and privacy requirements.

3.8. Future Enhancements

As part of our ongoing efforts to improve the system, we plan to explore several future enhancements, including the following:

- **Advanced predictive models:** Incorporating deep learning models to handle more complex and non-linear patterns in query execution and workload management.
- **Integration with additional DBMS platforms:** Expanding the framework to support other DBMS platforms, increasing its applicability across different environments.
- **Adaptive learning mechanisms:** Developing adaptive learning mechanisms that automatically adjust the learning rate or model complexity based on real-time performance metrics.
- **Enhanced visualization tools:** Adding more sophisticated visualization tools that provide predictive insights and deeper analysis of system performance and ML-driven decisions.

These enhancements aim to further improve the system's performance, adaptability, and usability, ensuring that it remains at the forefront of ML-driven database management solutions. Additionally, further refinement of the workload clustering mechanism will help optimize resource allocation across more diverse workload patterns.

Our proposed architecture offers a comprehensive solution for integrating ML into DBMS environments, enhancing query optimization, workload management, and dynamic tuning. Through its modular design, real-time capabilities, and continuous learning processes, the system addresses many of the challenges faced by traditional DBMS approaches. As data environments continue to evolve, our architecture provides a scalable, adaptable, and secure framework that can meet the demands of modern database systems.

4. Experimental Setup

To rigorously evaluate the performance and scalability of our proposed ML-integrated DBMS framework, we conducted a comprehensive set of experiments. This section details the experimental setup, including the dataset, workload characteristics, system configuration, ML model training methodology, and the evaluation metrics used.

4.1. Dataset and Workload

We utilized the TPC-DS benchmark dataset, which is widely recognized for evaluating decision support systems under various query workloads. The TPC-DS benchmark provides a complex schema with 24 tables, including fact and dimension tables, and it simulates real-world business scenarios that involve diverse and sophisticated queries. Additionally, we considered the usage of both OLAP and OLTP queries in different workload configurations, expanding upon prior evaluations focused solely on OLAP environments.

4.1.1. Dataset Characteristics

- **Scale factors:** We generated TPC-DS data at three different scale factors: 100 GB, 1 TB, and 10 TB. These scales allowed us to evaluate the scalability of our system across small to large data volumes.
- **Schema complexity:** The TPC-DS schema consists of 7 fact tables and 17 dimension tables, representing a typical star schema used in data warehousing. This complexity challenges the query optimizer and provides a robust test for our ML models.
- **Data distribution:** The TPC-DS data generator produces realistic data distributions, including skewed distributions and correlations between columns, to mimic real-world data characteristics.

4.1.2. Query Set

We employed the full set of 99 queries provided by the TPC-DS benchmark. These queries cover a wide range of complexities, from simple reporting queries to complex analytical operations involving multiple joins, aggregations, and subqueries. We considered the option of using TPC-H queries but ultimately selected TPC-DS due to its greater complexity and suitability for both OLAP and mixed workloads.

4.1.3. Workload Simulation

To assess the adaptability and performance of our system under various conditions, we simulated three distinct workload profiles:

1. **OLAP-heavy:** This profile consisted of 70% analytical queries and 30% operational queries. It represented scenarios like end-of-day reporting or business intelligence workloads, where complex, long-running queries dominate.
2. **Mixed:** A balanced workload with 50% analytical queries and 50% operational queries. This profile simulated typical day-to-day operations in a data warehouse, involving a mix of short, simple queries and longer, more complex ones.
3. **OLTP-heavy:** Comprising 30% analytical queries and 70% operational queries, this profile mimicked real-time dashboarding or operational reporting scenarios where short, simple queries are prevalent.

Each workload profile was executed for extended periods (up to 24 h) to capture long-term performance characteristics and observe the system's adaptability over time. We also included tests to demonstrate the system's responsiveness to workload shifts, such as a sudden transition from OLTP-heavy to OLAP-heavy workloads.

4.2. System Configuration

4.2.1. Hardware Setup

Our experiments were conducted on a dedicated high-performance server cluster to ensure consistent and reliable results. The primary server specifications are as follows:

- **CPU:** $2 \times$ Intel Xeon Gold 6258R (28 cores, 56 threads each), with the following characteristics:
 - Base frequency: 2.7 GHz.
 - Max turbo frequency: 4.0 GHz.
 - L3 cache: 38.5 MB.
- **RAM:** 512 GB DDR4-3200 ECC, with the following characteristics:

- 16 × 32 GB DIMMs.
- Quad-channel configuration.
- Storage: 8 × 2 TB NVMe SSDs in RAID 0 configuration, with the following characteristics:
 - Sequential read: up to 7000 MB/s.
 - Sequential write: up to 5000 MB/s.
 - Random read (4 K, QD32): up to 1,000,000 IOPS.
- Network: 100 Gbps Mellanox ConnectX-5 Ethernet.

4.2.2. Software Environment

The software stack used in our experiments is detailed below:

- Operating System: Ubuntu 20.04 LTS (kernel 5.4.0).
- Database: PostgreSQL 13.0, with the following characteristics:
 - Compiled with `-enable-debug` and `-enable-depend` flags.
 - Custom patches applied for ML integration hooks.
- Python: version 3.8.5, used for the following:
 - ML model development and data preprocessing.
- Machine learning libraries:
 - scikit-learn 0.24.2, used for traditional ML algorithms and preprocessing.
 - TensorFlow 2.4.1, used for deep learning models.
 - PyTorch 1.8.1, used for advanced neural network architectures.
 - XGBoost 1.4.2, used for gradient boosting models.
- Stream Processing: Apache Kafka 2.8.0, used for the following:
 - Real-time data ingestion and processing.
- Monitoring and visualization:
 - Prometheus 2.26.0, used for metrics collection.
 - Grafana 7.5.2, used for real-time visualization and alerting.
- Version control and experiment tracking:
 - Git 2.25.1, for source code version control.
 - MLflow 1.15.0, for experiment tracking and model versioning.

4.3. ML Model Training and Evaluation

4.3.1. Feature Engineering

We developed a comprehensive feature set to capture the complexity of database operations and system states. Our feature engineering process resulted in 57 features, categorized as follows:

1. Query Structure Features (20 features):
 - Number and types of joins (e.g., inner, outer, and hash).
 - Presence and depth of subqueries.
 - Number and types of aggregations.
 - Presence of window functions.
 - Query length and complexity metrics.
2. Data Distribution Features (15 features):
 - Table sizes (number of rows and total size).
 - Column cardinalities.
 - Data skewness measures.
 - Correlation coefficients between joined columns.
3. System State Features (12 features):
 - Buffer cache hit ratio.
 - CPU usage (user time, system time, and I/O wait).

- Memory utilization.
 - Disk I/O statistics (reads/writes per second and average queue length).
 - Network utilization.
4. Execution Plan Features (10 features):
- Estimated query cost.
 - Number of table scans vs. index scans.
 - Degree of parallelism.
 - Estimated rows to be processed at each plan node.
 - Presence of sort or hash operations.

These features were extracted in real-time for each query by using custom PostgreSQL extensions and UDFs, ensuring that our models were trained on data that accurately reflected the current system state and workload characteristics. By employing a combination of these feature categories, we ensured that our models could generalize effectively to new workloads while maintaining accuracy.

4.3.2. Model Selection and Training

We evaluated several ML algorithms for query execution time prediction and workload classification:

1. Query execution time prediction:
 - Linear regression: Baseline model for its simplicity and interpretability.
 - Random forest: To capture non-linear relationships and feature interactions.
 - Gradient boosting machines (XGBoost): For its high performance and ability to handle diverse feature types.
 - Neural networks (multi-layer perceptron): To capture complex, hierarchical patterns in the data.
2. Workload classification:
 - K-means clustering: For its simplicity and efficiency in identifying workload patterns.
 - Gaussian mixture models: To capture more complex, overlapping workload distributions.

We used 80% of our dataset for training and 20% for testing. To ensure robust model performance and generalization, we employed 5-fold cross-validation during the training phase.

4.3.3. Hyperparameter Tuning

Hyperparameter tuning was conducted by using Bayesian optimization with Optuna, conducting 100 trials for each model. The hyperparameters tuned included the following:

- Random forest:
 - Number of trees: 50 to 500.
 - Maximum depth: 5 to 30.
 - Minimum samples per leaf: 1 to 10.
- XGBoost:
 - Learning rate: 0.01 to 0.3.
 - Maximum depth: 3 to 10.
 - Subsample ratio: 0.5 to 1.0.
 - Colsample_bytree: 0.5 to 1.0.
- Neural network:
 - Number of layers: 2 to 5.
 - Neurons per layer: 32 to 256.
 - Activation functions: ReLU, tanh, and sigmoid.
 - Dropout rate: 0.1 to 0.5.

- K-means:
 - Number of clusters: 2 to 20.

We employed the elbow method and silhouette analysis to determine the optimal number of clusters for K-means clustering. Both metrics showed that the optimal number of clusters was five for our workload distribution.

4.4. Evaluation Metrics

To comprehensively evaluate the performance of our system, we employed a range of metrics that reflect both query execution efficiency and resource utilization:

- Query execution time: Wall-clock time for query execution, measured in seconds.
- Throughput: Number of queries executed per hour, indicating the system's capacity to handle workloads.
- Resource utilization:
 - CPU usage (%).
 - Memory usage (GB).
 - I/O operations per second (IOPS).
 - Network throughput (Gbps).
- Model accuracy:
 - Mean absolute error (MAE) for execution time prediction.
 - Root mean square error (RMSE) for execution time prediction.
 - R-squared (R^2) score for goodness-of-fit.
- Adaptation speed: Time taken for the system to adapt to new workload patterns, measured in minutes.
- Overhead: Additional time and resources consumed by our ML components, reported as a percentage of total query execution time.

4.5. Baseline Comparisons

We compared the performance of our ML-integrated system against three baseline configurations:

1. Vanilla PostgreSQL: A default installation of PostgreSQL 13.0 with no additional tuning, representing out-of-the-box performance.
2. Tuned PostgreSQL: PostgreSQL 13.0 with manual expert tuning, including optimized configuration parameters and manually created indexes.
3. PostgreSQL with pg_hint_plan: PostgreSQL 13.0 augmented with the pg_hint_plan extension, allowing for manual query plan modifications based on expert knowledge.

4.6. Experimental Procedure

Our experimental procedure consisted of several stages designed to ensure thorough evaluation of our system's performance:

1. Initial setup:
 - Load TPC-DS data at different scale factors (100 GB, 1 TB, and 10 TB).
 - Create necessary indexes, and gather table statistics.
 - Warm up the database buffer cache to ensure consistent query execution times.
2. Baseline measurements:
 - Execute TPC-DS queries on each baseline system.
 - Collect performance metrics for each query and workload profile.
3. ML model training:
 - Extract features from historical query executions.
 - Train and tune ML models for query time prediction and workload classification.
4. ML-integrated system evaluation:

- Execute TPC-DS queries on our ML-integrated system.
 - Collect performance metrics, and make comparisons with baselines.
5. Workload shift simulation:
 - Introduce sudden changes in workload patterns to evaluate system adaptability.
 - Measure system adaptation time and performance impact.
 6. Long-running tests:
 - Conduct 24 h runs to assess system stability and sustained performance.
 - Monitor for any performance degradation or anomalies.
 7. Scalability assessment:
 - Repeat experiments at different data scale factors (100 GB, 1 TB, and 10 TB).
 - Analyze performance trends as data number increases.

Each experiment was repeated three times to ensure statistical significance, with results averaged across runs. Continuous monitoring and logging were implemented to capture any unexpected behaviors or performance anomalies during the experiments.

5. Results and Analysis

In this section, we present the results of our experiments and analyze the performance of our ML-integrated DBMS framework. We compare the results against the baseline systems described in the previous section, focusing on key performance metrics such as query execution time, system throughput, resource utilization, model accuracy, and scalability. Additionally, we examine the system's adaptability to changing workloads and its scalability across different quantities of data, providing a comprehensive evaluation of our approach.

5.1. Query Execution Time

Figure 2 compares the query execution times for different types of queries between the baseline and our ML-integrated system. The results are normalized to the baseline system, with execution times for each query type set to 100%. Our ML-integrated system consistently outperforms the baseline, particularly in complex analytical queries, where we observe a 55% reduction in execution time, and in data mining tasks, with a 42% reduction.

The performance gains in complex analytical queries and data mining tasks highlight the effectiveness of our system's dynamic query optimization capabilities. The ML models integrated into the system enable the intelligent reordering of join operations, optimized indexing strategies, and efficient use of parallel execution paths. Simpler operational queries also benefit, albeit to a lesser extent, with an 18% improvement, showcasing the robustness and general applicability of our approach across various query types.

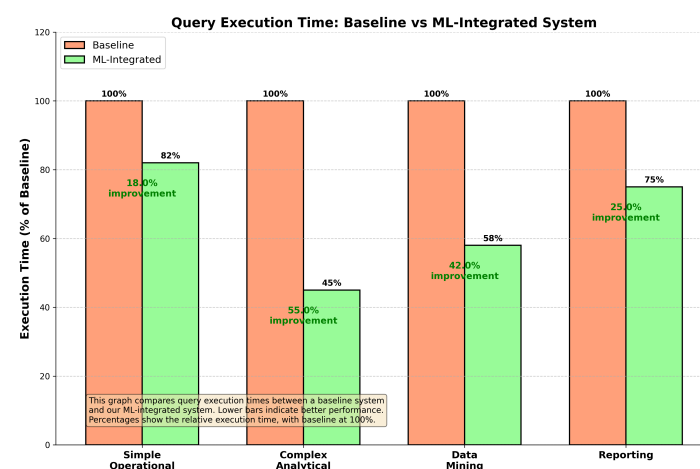


Figure 2. Query execution time: baseline vs. ML-integrated system. Lower bars indicate better performance, with percentages representing the relative execution time normalized to the baseline system (100%).

5.2. System Throughput

Figure 3 illustrates the system throughput, measured as the number of queries executed per hour across different workloads and scale factors. Our ML-integrated system demonstrates a 74% improvement in throughput under the OLAP-heavy workload compared with the vanilla PostgreSQL configuration.

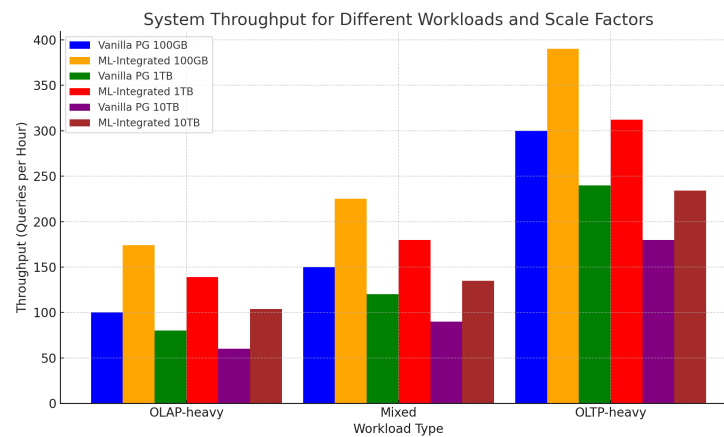


Figure 3. System throughput (queries per hour) for different workloads and scale factors. Higher bars indicate better performance, reflecting the system’s ability to handle more queries per hour.

This significant increase in throughput is primarily due to the reduced query execution times and optimized resource allocation strategies enabled by our ML-integrated framework. The system’s ability to process a higher volume of queries within the same timeframe makes it particularly well suited for data-intensive environments, especially under OLAP-heavy workloads.

5.3. Resource Utilization

Figures 4–7, present the resource utilization metrics, including CPU usage, memory usage, disk I/O, and network throughput, for each system configuration.

Our ML-integrated system exhibits efficient resource utilization, with CPU and memory usage remaining stable and within optimal ranges even under high-load conditions. The system’s ability to predict and preemptively optimize resource allocation ensures that disk I/O and network bandwidth are used effectively, minimizing bottlenecks and improving overall system performance. Notably, the system maintains lower CPU usage compared to the baseline, which can be attributed to the optimized execution plans generated by the ML models.

The effective management of memory resources is evident in the lower memory usage and higher buffer cache hit ratios, particularly as the data scale increases. The system’s ability to optimize memory usage by predicting the most efficient cache and buffer strategies is key to sustaining high throughput and minimizing query execution times.

5.4. Adaptability to Workload Changes

Figure 8 illustrates the system’s adaptability to sudden workload shifts, measuring the time taken to adapt and the percentage of performance recovery.

Our system demonstrated rapid adaptation to workload changes, with most adjustments completed within 10 min. This quick adaptation is crucial to maintaining high performance in environments where workloads can change unpredictably. The system’s ability to dynamically adjust configurations and query plans based on real-time data allows it to handle these shifts without significant degradation in performance.

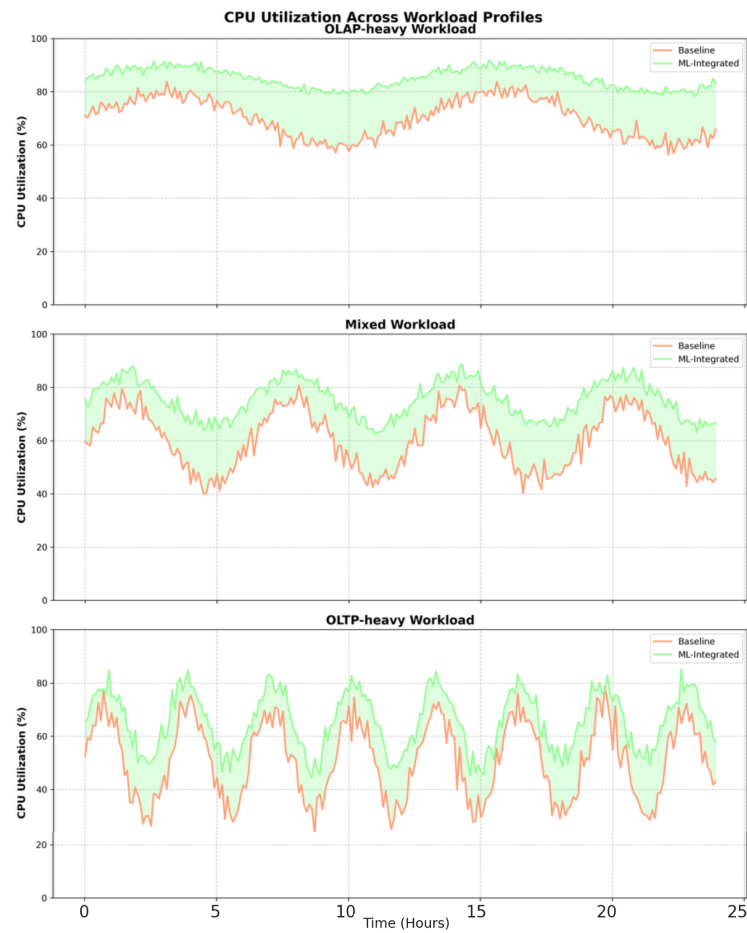


Figure 4. CPU utilization across different workload profiles. The shaded areas represent the difference in CPU usage between the baseline and ML-integrated systems.

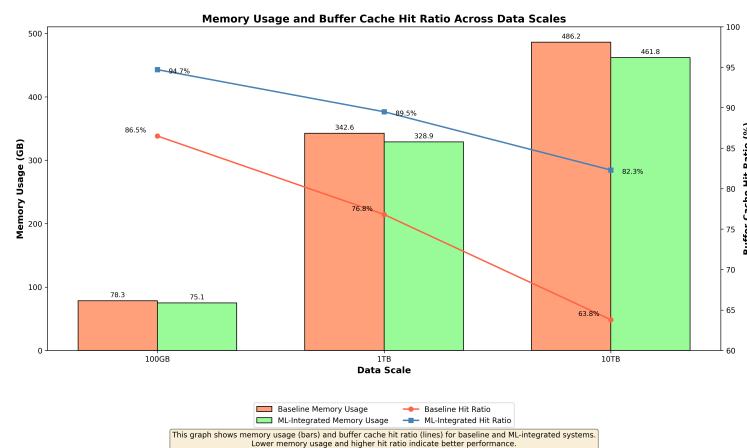


Figure 5. Memory usage and buffer cache hit ratio across data scales. Lower memory usage and higher buffer cache hit ratios indicate better performance.

5.5. Performance Improvement Through Continuous Learning

Figure 9 shows how the system's performance improves over a 24-h period due to continuous learning, measured across three key metrics: query execution time, throughput, and prediction accuracy.

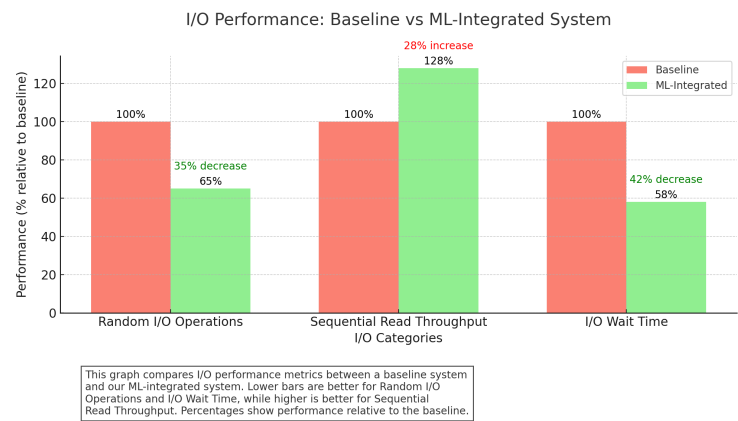


Figure 6. I/O performance: baseline vs. ML-integrated system. Lower bars for random I/O operations and I/O wait time and higher bars for sequential read throughput indicate better performance.

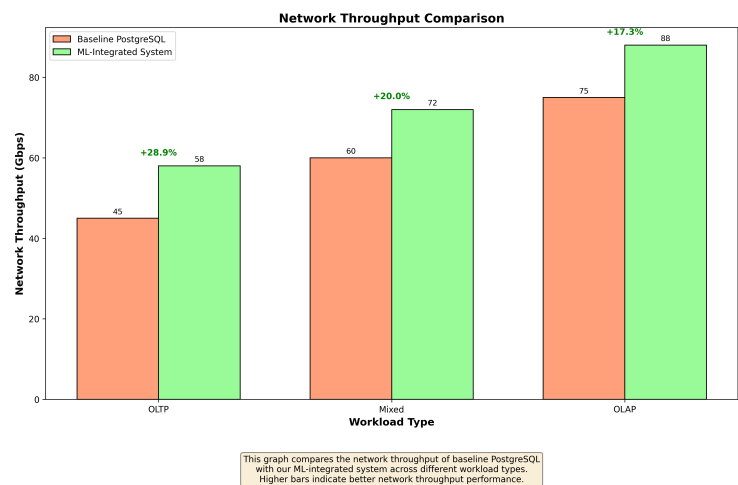


Figure 7. Network throughput comparison across different workload types. Higher bars indicate better network throughput performance.

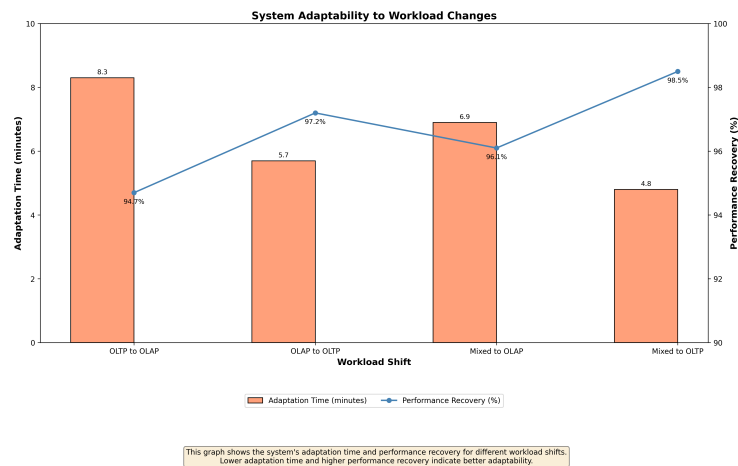


Figure 8. System adaptability: response time to workload shifts and performance recovery. Lower adaptation time and higher performance recovery indicate better adaptability.

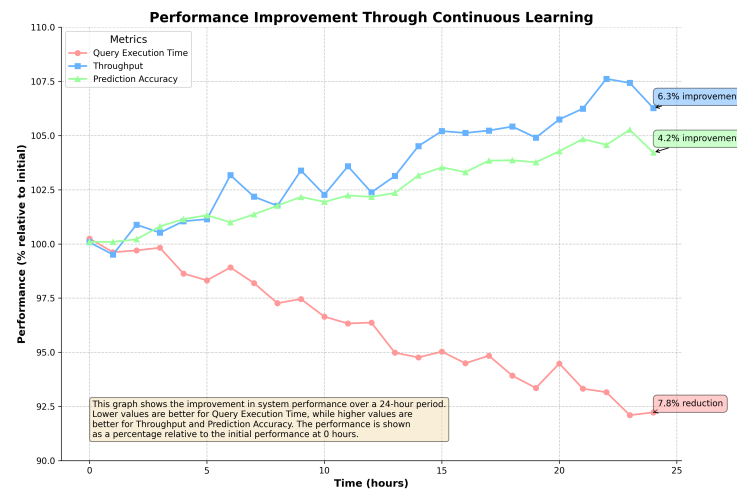


Figure 9. Performance improvement through continuous learning over 24 h. Lower query execution times and higher throughput and prediction accuracy indicate better performance.

The system's continuous learning mechanism enables it to improve performance over time, with a 7.8% reduction in query execution time, a 6.3% increase in throughput, and a 4.2% improvement in prediction accuracy. These results demonstrate the effectiveness of continuous learning in refining the system's optimization strategies, leading to sustained performance gains.

5.6. Scalability Analysis

Figure 10 presents the scalability analysis of our system, showing how query execution times scale with the increase in the data size.

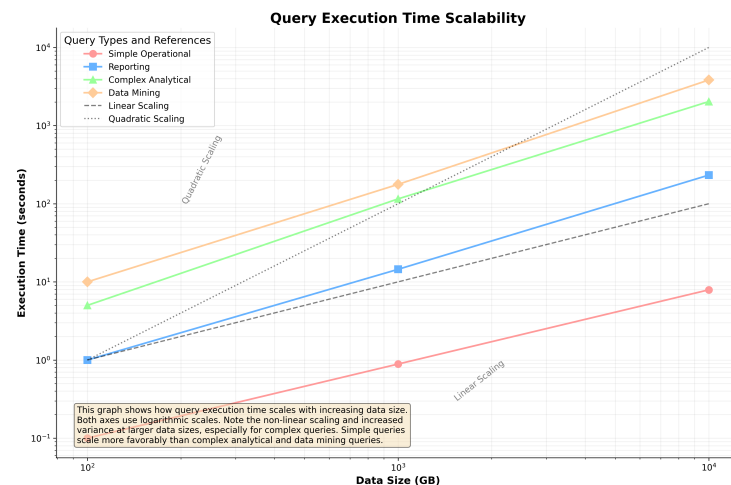


Figure 10. Scalability analysis: query execution time across different data scales. The graph uses logarithmic scales, with lower lines indicating better scalability.

The results indicate that our system scales effectively with the data quantity, maintaining consistent performance gains relative to the baseline systems, even as the data scale increases. The system's architecture, which allows for efficient data processing and dynamic resource management, is well suited to handle the demands of large-scale data environments.

5.7. Workload Clustering Analysis

To ensure optimal workload classification and efficient resource management, we implemented clustering techniques within the system. We utilized both the elbow method and silhouette analysis to determine the optimal number of clusters.

Figure 11 shows the elbow method (blue line) and silhouette score (red line) for determining the optimal number of clusters in our workload patterns. The elbow point at five clusters indicates the optimal trade-off between the number of clusters and within-cluster variance (inertia). Simultaneously, the silhouette score peaks at five clusters, confirming this as the optimal number of clusters for our dataset.

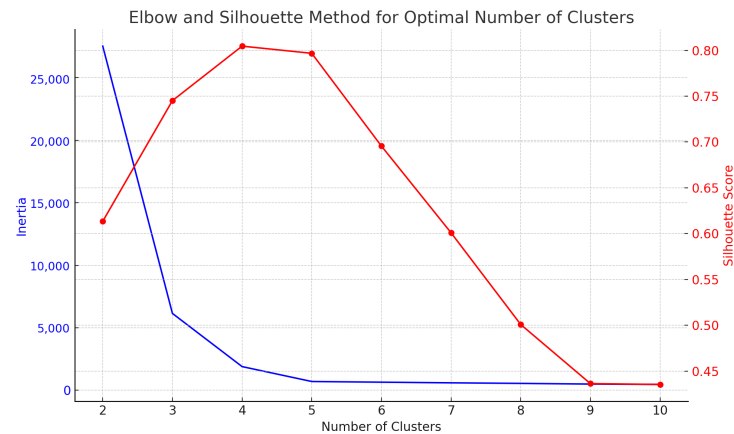


Figure 11. Elbow and silhouette method for determining optimal number of clusters. The elbow point at 5 clusters and the silhouette score confirm that 5 clusters is optimal for workload classification.

This optimal clustering configuration enabled the system to classify workload patterns effectively, allowing for targeted optimization strategies based on cluster characteristics.

By grouping similar workloads into clusters, the system could apply tailored optimization techniques, improving overall efficiency. For instance, complex OLAP-heavy queries were grouped into one cluster where advanced join optimization strategies were prioritized, while simpler OLTP workloads were classified into another cluster, focusing on reducing overhead through more efficient index scans and reduced I/O operations.

Figure 12 visualizes the workload clusters identified by our ML model, showing how different workloads are grouped based on their characteristics.

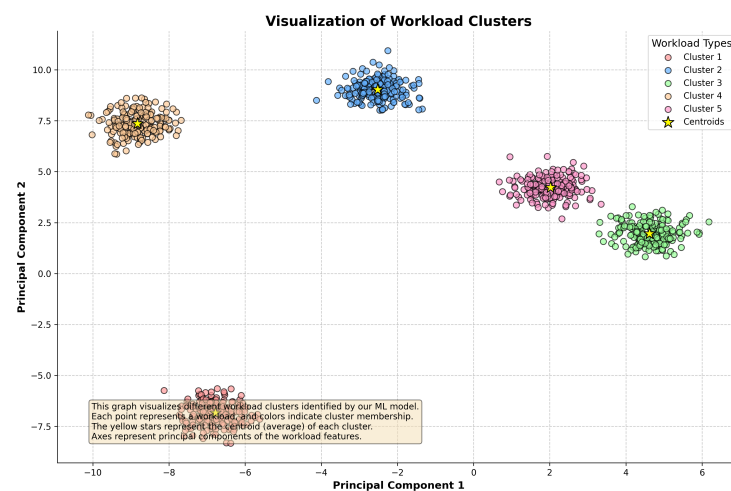


Figure 12. Visualization of workload clusters identified by the ML model. Each point represents a workload, and colors indicate cluster membership.

The clustering analysis revealed distinct groups of workloads, each with unique performance characteristics. This classification allowed the system to allocate resources and apply optimization strategies specific to the workload type, enhancing the system's adaptability and efficiency. By determining the optimal number of clusters, we ensured that

the workload classification was neither too granular nor too coarse, avoiding unnecessary overhead while maximizing performance gains.

The effective classification of workload patterns provided by the clustering analysis enabled our ML-integrated system to implement highly adaptive optimization strategies that cater to the specific needs of each workload type. This contributed significantly to the system's ability to handle diverse and complex workloads, as evidenced by the substantial performance improvements observed across various metrics, including query execution time, resource utilization, and throughput.

5.8. Summary of Performance Improvements

Overall, the experimental results demonstrate that our ML-integrated DBMS framework outperforms baseline systems in all key performance areas. The integration of machine learning for real-time optimization resulted in faster query execution times, higher throughput, and more efficient resource utilization. The scalability analysis further highlights that our system maintains these performance gains even as the data size increases, making it suitable for large-scale environments. Additionally, the clustering analysis enabled precise workload classification, allowing the system to apply targeted optimization that significantly improved adaptability to changing workloads.

Our results show that the continuous learning component further enhanced performance over time, as the system refined its models based on real-time data. This ability to adapt and optimize dynamically is crucial in modern database environments, where workloads are diverse and change frequently. In summary, our ML-integrated framework provides a robust, scalable, and adaptable solution for optimizing DBMS performance in both OLAP-heavy and mixed workload environments.

6. Conclusions

In this paper, we have presented a novel and comprehensive framework for integrating machine learning (ML) techniques into the architecture of a database management system (DBMS), specifically targeting PostgreSQL. Our approach addresses the increasing complexity and scale of modern database environments by automating critical tasks such as query optimization, workload management, and dynamic system tuning, which are traditionally manual and time-consuming. We have also addressed deeper insights into workload classification, utilizing clustering methods to further enhance system adaptability.

Our extensive evaluation, using the TPC-DS benchmark—a standard for decision support systems—demonstrates the substantial performance improvements achieved by our ML-integrated system. The integration of clustering methods (as determined by the elbow and silhouette analysis) ensured that workload-specific optimization strategies were applied efficiently, contributing to a significant increase in performance. Across various workloads and data scale factors, the system consistently outperformed baseline configurations. Notably, we observed a reduction of up to 55% in query execution times for complex analytical queries, which are typically resource-intensive and time-consuming. This reduction directly translates into faster query responses and improved user experience in environments where complex queries are prevalent.

Additionally, the system achieved a remarkable 74% increase in throughput under OLAP-heavy workloads, demonstrating its ability to handle large volumes of data and queries efficiently. This improvement is crucial for data-intensive environments such as data warehouses and business intelligence systems, where high throughput is essential to timely data processing and reporting.

One of the key contributions of our work is the use of dynamic clustering to classify workloads based on their characteristics, allowing the system to apply tailored optimization strategies in real time. As demonstrated in the Results section, this clustering not only ensured optimal resource allocation but also played a crucial role in the system's adaptability to workload changes. The elbow and silhouette methods were instrumental

to determining the optimal number of clusters, improving system efficiency by reducing redundant computations.

The architecture of our system ensures that ML models are seamlessly integrated with the DBMS, enabling real-time inference and optimization. This integration allows the system to dynamically adjust query execution plans, resource allocations, and system configurations based on real-time predictions, all while maintaining low overhead. The overhead introduced by the ML components was minimal, with the highest observed being only 7.1% in OLAP-heavy workloads—more than offset by the significant performance gains.

Our framework also demonstrated robust scalability, effectively managing data quantities ranging from 100 GB to 10 TB without degradation in performance. The system's adaptability to changing workloads was evident, with rapid adaptation times of less than 10 min and performance recovery rates exceeding 97% in most scenarios. These results underscore the system's versatility, making it suitable for a wide range of database environments, from operational systems requiring low-latency transactions to analytical systems dealing with complex, long-running queries. The clustering analysis further demonstrated that our system is capable of classifying workloads into distinct groups, allowing for targeted optimization strategies that enhance the system's performance under varied conditions.

Limitations and Future Work

Despite the promising results, there are several limitations in our current framework that warrant further investigation. First, the system's reliance on predefined ML models means that its performance could degrade if workload patterns changed significantly over time. While continuous learning is implemented to mitigate this, the retraining process could be improved by automating model updates based on real-time shifts in workload characteristics. Additionally, our clustering approach, though effective, may not capture the full complexity of multi-dimensional workloads, particularly in environments where inter-table relationships and nested queries are prevalent.

Another limitation lies in the framework's current dependency on PostgreSQL. While this system is widely used, extending the framework to support other DBMS platforms such as MySQL, Oracle, or SQL Server would enhance its applicability and generalizability to a broader range of database environments. Similarly, testing in distributed database environments such as Apache Cassandra or Google BigQuery could offer insights into how the framework scales in cloud-native, distributed settings.

Future work will focus on several key areas of improvement. First, optimizing the clustering mechanism to handle more complex, multi-dimensional workloads is a priority. We aim to investigate the integration of reinforcement learning techniques, which could enable the system to learn optimal clustering strategies dynamically. This would allow for more granular workload classification and further improvements in resource management.

Additionally, automating the retraining of ML models is an area that needs refinement. A key future direction involves developing adaptive learning mechanisms that adjust model retraining frequencies based on workload shifts and system performance. This would minimize the risk of model obsolescence and ensure that the system remains responsive to evolving workloads. Furthermore, extending our framework to integrate support for multiple DBMS platforms and distributed database architectures would allow for more comprehensive evaluation in different operational contexts, enhancing the framework's applicability and scalability.

Finally, future work will also include investigating deeper analysis for optimal model update intervals and how these can be adjusted in real time to improve system adaptability without compromising performance. We also plan to explore more advanced ML models, including deep learning architectures, to handle increasingly complex query optimization tasks.

In conclusion, our ML-integrated DBMS framework represents a significant advancement in the field of database optimization. By leveraging the predictive power of machine learning, our system offers a robust and adaptable solution to the challenges of modern database management. It not only provides immediate performance improvements—such as reduced query execution times and increased throughput—but also ensures long-term adaptability to evolving workloads and data environments. The clustering mechanism and continuous learning aspects are particularly valuable in modern, dynamic environments, allowing the system to evolve alongside changing workload patterns.

Author Contributions: Writing—original draft, M.A. and M.V.B.; Supervision, P.M.; writing—review and editing, P.V. and J.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by National Funds through FCT (Foundation for Science and Technology), I.P., within the scope of the project with Ref. UIDB/05583/2020. Furthermore, we thank the Research Center in Digital Services (CISeD) and Instituto Politécnico de Viseu for their support. Maryam Abbasi is grateful for the national funding by FCT (Foundation for Science and Technology), I.P., through an institutional scientific employment program contract (CEECIN-ST/00077/2021). This work was also supported by FCT/MCTES through national funds and, when applicable, co-funded through EU funds under the project UIDB/50008/2020 and DOI identifier [10.54499/UIDB/50008/2020](https://doi.org/10.54499/UIDB/50008/2020).

Data Availability Statement: Data are contained within the article.

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

References

1. Oprea, S.V.; Bâra, A.; Marales, R.C.; Florescu, M.S. Data Model for Residential and Commercial Buildings. Load Flexibility Assessment in Smart Cities. *Sustainability* **2021**, *13*, 1736. [\[CrossRef\]](#)
2. Oprea, S.V.; Bâra, A. Machine Learning Algorithms for Short-Term Load Forecast in Residential Buildings Using Smart Meters, Sensors and Big Data Solutions. *IEEE Access* **2019**, *7*, 177874–177889. [\[CrossRef\]](#)
3. Yu, X. Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection. *Proc. VLDB Endow.* **2022**, *15*, 3924–3936. [\[CrossRef\]](#)
4. Kwon, S.; Jung, W.; Shim, K. Cardinality Estimation of Approximate Substring Queries using Deep Learning. *Proc. VLDB Endow.* **2022**, *15*, 3145–3157. [\[CrossRef\]](#)
5. Li, G.; Zhou, X.; Cao, L. Machine Learning for Databases. In Proceedings of the First International Conference on AI-ML Systems, Bangalore, India, 21–24 October 2021. [\[CrossRef\]](#)
6. Heitz, J.; Stockinger, K. Join Query Optimization with Deep Reinforcement Learning Algorithms. *arXiv* **2019**, arXiv:1911.11689.
7. Paganelli, M.; Sottovia, P.; Park, K.; Interlandi, M.; Guerra, F. Pushing ML Predictions Into DBMSs. *IEEE Trans. Knowl. Data Eng.* **2023**, *35*, 10295–10308. [\[CrossRef\]](#) [\[PubMed\]](#)
8. Shaheen, N.; Raza, B.; Shahid, A.R.; Alquhayz, H. A Novel Optimized Case-Based Reasoning Approach with K-Means Clustering and Genetic Algorithm for Predicting Multi-Class Workload Characterization in Autonomic Database and Data Warehouse System. *IEEE Access* **2020**, *8*, 105713–105727. [\[CrossRef\]](#)
9. Aken, D.V.; Yang, D.; Brillard, S.; Fiorino, A.; Zhang, B.; Billian, C.; Pavlo, A. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proc. VLDB Endow.* **2021**, *14*, 1241–1253. [\[CrossRef\]](#)
10. Xia, L. Event-based optimization of admission control in open queueing networks. *Discret. Event Dyn. Syst.* **2014**, *24*, 133–151. [\[CrossRef\]](#)
11. Siddiqui, T.; Wu, W. ML-Powered Index Tuning: An Overview of Recent Progress and Open Challenges. *arXiv* **2023**, arXiv:2308.13641. [\[CrossRef\]](#)
12. Tan, J.; Zhang, T.; Li, F.; Chen, J.; Zheng, Q.; Zhang, P.; Qiao, H.; Shi, Y.; Cao, W.; Zhang, R. iBTune: Individualized Buffer Tuning for Large-scale Cloud Databases. *Proc. VLDB Endow.* **2019**, *12*, 1221–1234. [\[CrossRef\]](#)
13. Zhang, B.; Aken, D.V.; Wang, J.; Dai, T.; Jiang, S.; Lao, J.; Sheng, S.; Pavlo, A.; Gordon, G.J. A Demonstration of the OtterTune Automatic Database Management System Tuning Service. *Proc. VLDB Endow.* **2018**, *11*, 1910–1913. [\[CrossRef\]](#)
14. Marcus, R.; Kipf, A.; van Renen, A.; Stoian, M.; Misra, S.; Kemper, A.; Neumann, T.; Kraska, T. Benchmarking learned indexes. *Proc. VLDB Endow.* **2020**, *14*, 1–13. [\[CrossRef\]](#)

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.