Journal Homepage: - www.journalijar.com

INTERNATIONAL JOURNAL OF ADVANCED RESEARCH (IJAR)

Article DOI: 10.21474/IJAR01/23486
DOI URL: http://dx.doi.org/10.21474/IJAR01/23486

RESEARCH ARTICLE

A COMPARATIVE STUDY OF INDEXING STRATEGIES FOR BOOSTING POSTGRESQL QUERY PERFORMANCE

Ali Hamadou[1], Abdoul Aziz Issaka Hassane[1], Harouna Naroua[2], Abdoul Nasser Abdou Hama, Abdou Salame Salissou[2] and Ibrahim Mouazamou Laoualy Chaharou[1]

1. Department of Mathematics, Dan Dicko Dankoulodo University of Maradi, Maradi, Niger.
2. Department of Mathematics and Computer Science, Abdou Moumouni University, Niamey, Niger.

Manuscript Info

Manuscript History
Received: 12 March 2026
Final Accepted: 14 April 2026
Published: May 2026

Key words:-
Indexing; Simple index; Composite index; Text index; Query performance; PostgreSQL

Abstract

In the digital age, the exponential growth of data volumes in modern information systems has made query performance a critical challenge for organizations relying on relational database management systems (RDBMS). While indexing is widely recognized as an effective optimiz ation strategy, the comparative impact of different index types on PostgreSQL performance across multiple operation categories remains insufficiently studied in the literature. This study provides an experime ntal evaluation of three indexing strategies — simple index, composite index, and text index — applied to PostgreSQL 18.0 on a synthetic dataset of 500,000 records. Performance is assessed across three fundamental data manipulation operations (retrieval, updating, and deletion) using four metrics: execution time, CPU utilization, memory consumption, and index storage size. Results demonstrate substantial performance gains for retrieval and update operations across all index types: 96.5% and 98.0% for the simple index, 96.3% and 98.7% for the composite index, and 98.8% and 99.9% for the text index. Deletion operations showed more moderate gains (51.4%–60.3%), attributed to index maintenance overhead. These findings offer concrete guidelines for practitioners seeking to optimize large-scale data management in PostgreSQL-based information systems.

Introduction:-

At a time when digital transformation is profoundly reshaping how organizations operate, the volume of data generated by information systems is experiencing unprecedented growth. Every day, enormous quantities of data are produced, collected, and used to support decision-making processes, optimize operations, and enhance business competitiveness. In this context, database management systems (DBMS) have become essential technological pillars, ensuring the structured storage, organization, and efficient exploitation of these vast amounts of information.

However, this exponential data growth poses significant challenges. As databases reach critical sizes, fundamental operations such as data retrieval, updating, and deletion become increasingly costly in terms of processing time. This

Corresponding Author:- Ali Hamadou
Address:-Department of Mathematics, Dan Dicko Dankoulodo University of Maradi, Maradi, Niger.

performance degradation can significantly slow down companies' analytical and operational workflows. Studies have shown that poor database system responsiveness can lead to an organizational productivity decline of up to 30% [1] — not to mention direct financial losses linked to service interruptions or processing delays [2].

Given the scale of these challenges, choosing a high-performance DBMS capable of handling large data volumes is a strategic decision for both researchers and practitioners. In this landscape, PostgreSQL has established itself as a leading reference. This robust, extensible, and proven open-source relational system is now widely adopted worldwide. This prominence is confirmed by the Stack Overflow 2024 survey, which ranks it as the most widely used database among developers [3]. This recognition reflects its technological maturity and the trust it has earned within the developer community. However, even with a powerful system such as PostgreSQL, traditional query techniques can prove insufficient when applied to massive datasets. To address this issue, indexing techniques emerge as one of the most effective and accessible optimization strategies. By intelligently structuring data access, indexes significantly reduce response times and improve processing efficiency without requiring a full system architectural redesign.

Despite the extensive literature on database optimization, comparative experimental studies specifically evaluating the performance of multiple index types across heterogeneous operation categories (retrieval, updating, and deletion) in PostgreSQL remain scarce. Most existing work either focuses on a single index type or does not provide quantitative improvement rates across all three operation types simultaneously. It is within this perspective that the present study is situated, aiming to analyze and compare the effectiveness of three types of indexes applied to PostgreSQL: simple index, composite index, and text index. The experiments, conducted on a synthetic dataset of 500,000 records, focus on three key data manipulation operations: retrieval, updating, and deletion.

More specifically, this research pursues the following objectives:

- Evaluate and compare the performance impact of simple, composite, and text indexes on PostgreSQL query execution;
- Quantify improvement rates for retrieval, updating, and deletion operations under each indexing strategy;
- Provide practical guidelines for index selection based on operation type and data characteristics.

The rem ainder of this paper is organized as follows: Section 2 presents related work. Section 3 describes the materials and methods, including dataset description, hardware and software configuration, and experimental protocol. Section 4 presents the experimental results and their analysis. Section 5 discusses these results, and Section 6 concludes the paper.

Related Work

Several research works have been published to guide the community toward appropriate indexing techniques capable of significantly accelerating query execution across various DBMSs. However, the literature shows that relatively few studies have truly highlighted the impact and importance of using indexes to improve query performance in PostgreSQL.

In this regard, the design of a new suffix-tree-based index in PostgreSQL has been investigated, despite the mismatch between the theoretical assumptions of this structure and the actual architecture of the system [4]. That study therefore detailed the main architectural components of PostgreSQL that have a direct impact on index design. It showed that integrating a new index structure into a DBMS such as PostgreSQL goes beyond simply adapting an algorithm to a programming interface; it also requires compatibility with the system's internal invariants. Furthermore, the authors of [5] proposed a comparative study of the six native PostgreSQL index types, namely B-Tree, Hash, GiST, SP-GiST, GIN, and BRIN, based on previous work. The results indicate that the B-Tree index is the most efficient solution for transactional workloads. In contrast, GIN provides the best performance for full-text search as well as for queries on JSONB data. The study also highlights the effectiveness of GiST and SP-GiST indexes for geometric and spatial workloads, while BRIN demonstrates strong scalability in analytical workloads and time-series data. The authors therefore concluded that the choice of an index type in PostgreSQL must be determined by the nature of the workload, data distribution, and update frequency. In the same vein, performance analyses of B-Tree, GIN, and BRIN indexes in PostgreSQL have been proposed for SELECT, UPDATE, and INSERT operations across different dataset sizes [6]. The results show that B-Tree indexes offer the best overall performance for SELECT and UPDATE operations, while GIN indexes are particularly well suited for JSONB data, despite their high resource consumption. The study also reveals that BRIN indexes are efficient for very large

databases due to their low overhead and compactness. Additionally, a study [7] proposed an approach focused on query optimization in PostgreSQL 15 aimed at supporting IT professionals using this DBMS. This research introduced several query categories as well as optimization strategies adapted to each of them, particularly through the use of indexing and appropriate join algorithms. Finally, the authors of [8] presented PostLearn, a native integration of a learned index based on ALEX+ into PostgreSQL to improve performance. The results indicate that this approach outperforms B+-Tree indexes for point queries and short-range scans; however, performance gains diminish due to increasing overheads.

Beyond PostgreSQL-specific studies, a broader body of literature addresses indexing and query performance in relational and multi-database contexts. Abbasi et al. [9] conducted a comprehensive experimental study on hardware-conscious indexing strategies, examining B-tree, hash, bitmap, and composite indexes across varying dataset sizes and access patterns using the TPC-H benchmark. Their results showed query execution time improvements ranging from 32.4% to 48.6% depending on the hardware configuration, underscoring that index performance is sensitive not only to the index type chosen but also to the underlying hardware architecture. This finding complements our study by highlighting that software-level indexing choices interact closely with hardware constraints.

In a comparative perspective, Salunke et al. [10] benchmarked PostgreSQL against MySQL on large-scale datasets, showing that PostgreSQL outperformed MySQL by a factor of up to 13 on primary key lookups, confirming the performance maturity of PostgreSQL and lending additional relevance to studies that seek to further optimize it through indexing. Similarly, a study comparing read performance between PostgreSQL and MongoDB in e-commerce scenarios [11] found that PostgreSQL executed complex analytical queries 1.6 to 15.1 times faster, with 65–80% shorter execution times for multi-criteria queries. These cross-system comparisons highlight the importance of selecting both the right DBMS and the appropriate indexing strategy for a given workload.

Materials and Methods:-

This section describes the materials and methods used in this study. It presents the dataset used, the hardware and software configuration, and the experimental protocol established to conduct all experiments in a controlled and reproducible environment.

Dataset Description:-

The experimental dataset comprises 500,000 records, each entry representing a product entity defined by a structured set of attributes. Each record is characterized by eight descriptive attributes: a unique product identifier (idproduit), product name, category, weight, color, dimensions, a textual description, and price. The data was transferred in CSV (Comma-Separated Values) format, thereby ensuring optimal compatibility and simplifying its integration into various database environments for experimental purposes. To ensure a high diversity of data and to optimize the use of indexing mechanisms under realistic conditions, several high-cardinality attributes were designed. This is particularly the case for the fields idproduit, name, weight, price, and description, whose values are unique to each record. In contrast, attributes such as color have limited cardinality, which facilitates the study of index behavior across different data distribution contexts and improves the comparative analysis of optimization approaches.

In addition, each product is associated with a specific textual description composed of six automatically generated words. This design choice aims to simulate the presence of large volumes of textual fields and to evaluate the effectiveness of text-based queries at scale. This dataset was therefore developed as a controlled and reproducible experimental environment, enabling a detailed analysis of the influence of various indexing methods and query optimization strategies on large-scale data volumes.

Hardware and Software Configuration:-

Our experiments were conducted on an HP All-in-One 24-cr0xxx desktop computer equipped with 16 GB of RAM and a 932 GB NVMe SSD. Regarding the database management system, we used PostgreSQL 18.0. This hardware and software configuration enabled the efficient execution of all queries on the previously described dataset, allowing us to obtain reliable and significant performance results.

Experimental Protocol:-
All experiments were conducted in a controlled and reproducible environment. For each index type (simple, composite, and text), three categories of queries were executed: retrieval (SELECT), updating (UPDATE), and deletion (DELETE). Each query was run using PostgreSQL's EXPLAIN ANALYZE command, which provides actual execution time, CPU usage, and memory consumption metrics. To ensure measurement reliability, each query was executed five times and the median execution time was retained. The PostgreSQL buffer cache was cleared between each run to avoid cache warm-up effects. Index size was measured using the pg_relation_size() function after index creation. The improvement rate for each operation was computed using the following formula: Improvement Rate = (Time without index – Time with index) / Time without index × 100.

Experimental Results and Analysis:-
This section presents the results obtained from a series of experiments conducted on PostgreSQL, focusing on the evaluation of the impact of indexing techniques on query performance. The tests were carried out on three fundamental categories of data manipulation operations, namely: data retrieval, data updating, and data deletion. The results are organized and presented in tabular form, providing a clear and structured comparative view of the measured performance indicators.

For each type of operation and each indexing strategy, the tables provide the following metrics:
- Query execution time, expressed in milliseconds (ms);
- Percentage of CPU utilization;
- Memory space consumed during query execution;
- Memory space allocated for the creation of each index.

Impact of the Simple Index on Query Performance:-
This subsection is dedicated to the comparative analysis of PostgreSQL query performance, evaluated first in the absence and then in the presence of a simple index. The objective is to accurately measure the contribution of this indexing technique to data processing efficiency.

Table 1 summarizes all results obtained by presenting both sets of performance indicators: those of queries executed without a simple index, which reflect the system's native behavior when handling retrieval, update, and deletion operations, and those measured after applying the simple index.
Table 2 summarizes and quantifies the performance improvement rates achieved through the use of the simple index by comparing the results of the two previous configurations. These rates constitute a key indicator for evaluating the effectiveness and relevance of this optimization technique in a large-scale data management context.

Table 1:Query Execution Performance — Simple Index

Operation	Index Condition	Time (ms)	CPU (%)	RAM (MB)	Size (MB)
Retrieval	Without a simple index	74.51	160.54	0.0	0.0
Update		30.01	165.86	0.0	0.0
Deletion		54.49	168.52	0.0	0.0
Retrieval	With a simple index	2.54	156.32	6.22	15.99
Update		0.58	160.01	0.0	15.99
Deletion		26.43	162.87	0.0	15.99

Table 2: Performance Improvement Rates — Simple Index

Operation	Improvement Rate
Retrieval	96.5%
Update	98.0%
Deletion	51.4%

The results presented in Tables 1 and 2 show that the use of a simple index in PostgreSQL significantly improves query performance. The execution times of retrieval, update, and deletion operations decrease substantially after indexing. The highest performance gains are observed for data retrieval (96.5%) and data updating (98.0%), while data deletion shows a more moderate improvement (51.4%). CPU usage remains relatively stable, with a slight additional cost in memory and storage space associated with index creation.

The improvement rate was calculated using the following formula:

$$\text{Improvement Rate} = \frac{\text{Time without index} - \text{Time with index}}{\text{Time without index}} \times 100$$

Figures 1, 2, and 3 provide a graphical representation of the results presented in Table 1.

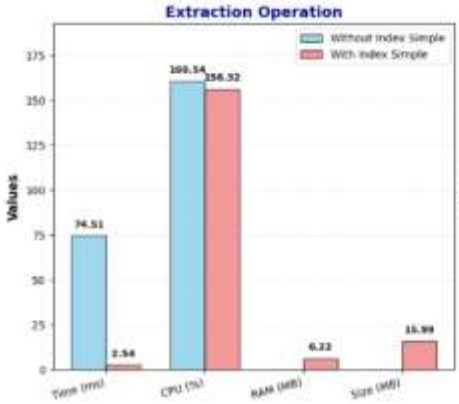


Figure 1: Query execution time — Retrieval operation, Simple Index

Figure 2: Query execution time — Update operation, Simple Index

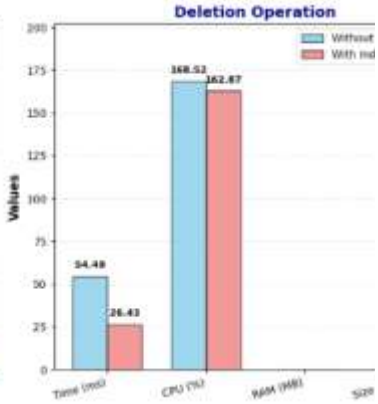


Figure 3: Query execution time — Deletion operation, Simple Index

Figure 4 illustrates the performance improvement rates for the three operations using the simple index.

Figure 4: Performance improvement rates — Simple Index

Impact of the Composite Index on Query Performance:-

This subsection is devoted to a comparative analysis of PostgreSQL query performance, evaluated both without and with a composite index. This allows a precise assessment of the contribution of this indexing method to data processing efficiency.

Table 3 summarizes all the results obtained, presenting both the performance indicators of queries executed without a composite index — reflecting the system's intrinsic behavior during retrieval, update, and deletion operations — and the performance measured after implementing the composite index.

Table 4 summarizes and evaluates the performance improvement levels achieved through the composite index by comparing the results of the two previous configurations.

Table 3: Query Execution Performance — Composite Index

Operation	Index Condition	Time (ms)	CPU (%)	RAM (MB)	Size (MB)
Retrieval	Without a composite index	72.33	164.68	0.0	0.0
Update		45.54	165.86	0.0	0.0
Deletion		72.15	168.61	0.0	0.0
Retrieval	With composite index	2.61	157.14	0.0	23.8
Update		0.57	160.08	0.0	23.8
Deletion		28.58	163.04	0.0	23.8

Table 4: Performance Improvement Rates — Composite Index

Operation	Improvement Rate
Retrieval	96.3%
Update	98.7%
Deletion	60.3%

The results from Tables 3 and 4 show that the use of a composite index in PostgreSQL leads to a significant improvement in query performance.

The highest gains are observed for update operations (98.7%) and retrieval operations (96.3%), indicating near-optimal optimization of these processes after indexing. Deletion also shows a substantial improvement, although more moderate, with a gain of 60.3%.These results confirm the effectiveness of the composite index in reducing response times, particularly for read and update operations, while improving overall system performance.

Figures 5, 6, and 7 provide a graphical representation of the results presented in Table 3.

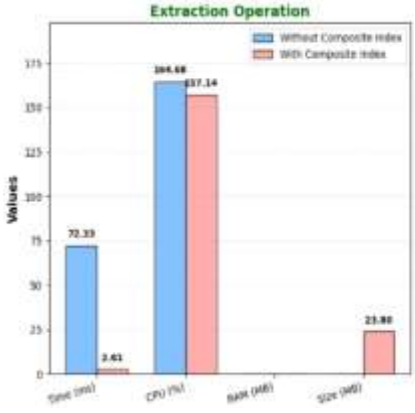


Figure 5: Query execution time — Retrieval operation, Composite Index

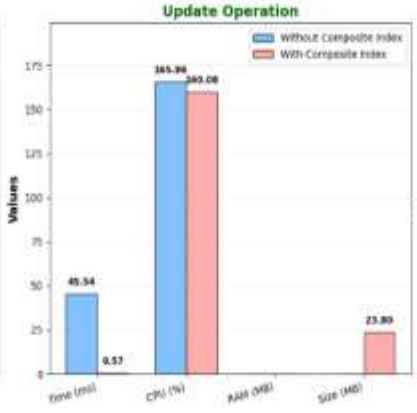


Figure 6: Query execution time — Update operation, Composite Index

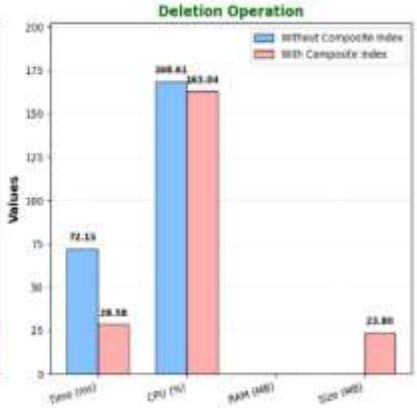


Figure 7: Query execution time — Deletion operation, Composite Index

Figure 8 illustrates the performance improvement rates for the three operations using the composite index.

Figure 8: Performance improvement rates — Composite Index

Impact of the Text Index on Query Performance:-

This subsection focuses on the study of PostgreSQL query performance before and after the integration of a text index. The main objective is to evaluate in detail the impact of this specialized indexing method on the optimization of textual data processing. Table 5 presents all the experimental results obtained under the two configurations: performance observed without a text index, illustrating the system's standard behavior during retrieval, update, and deletion operations applied to textual data, and performance recorded after the implementation of the text index. Table 6 summarizes the performance improvement rates achieved through the use of the text index, comparing the results of the two configurations.

Table 5: Query Execution Performance — Text Index

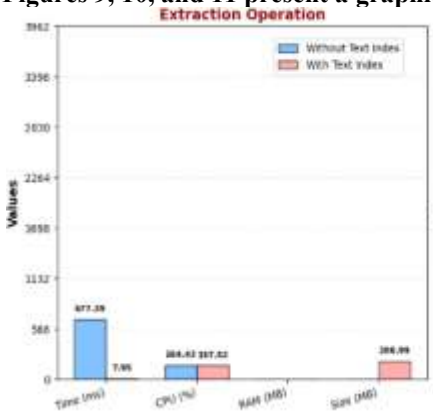
Operation	Index Condition	Time (ms)	CPU (%)	RAM (MB)	Size (MB)
Retrieval	Without a text index	677.39	164.43	0.0	0.0
Update		1676.96	162.43	0.0	0.0
Deletion		3445.22	167.01	0.0	0.0
Retrieval	With a text index	7.95	157.52	0.0	206.99
Update		0.8	159.57	0.0	206.99
Deletion		1615.86	161.81	0.0	207.0

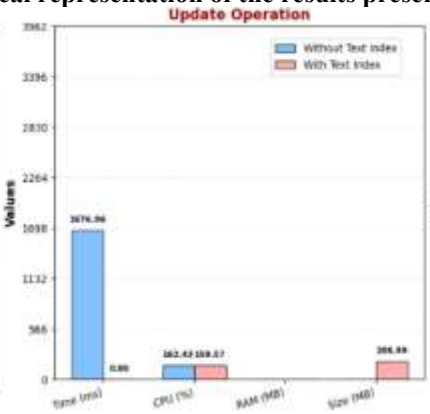
Table 6: Performance Improvement Rates — Text Index

Operation	Improvement Rate
Retrieval	98.8%
Update	99.9%
Deletion	53.0%

The results presented in Tables 5 and 6 highlight a notable improvement in PostgreSQL query performance through the use of a text index. The most significant gains concern update operations (99.9%) and retrieval operations (98.8%), reflecting near-complete optimization of these processes after indexing. Deletion operations also show improvement, although more limited, with a gain of 53.0%. Overall, these observations confirm that the text index effectively reduces response times, particularly by optimizing update and retrieval operations, while generally enhancing overall system performance.

Figures 9, 10, and 11 present a graphical representation of the results presented in Table 5.





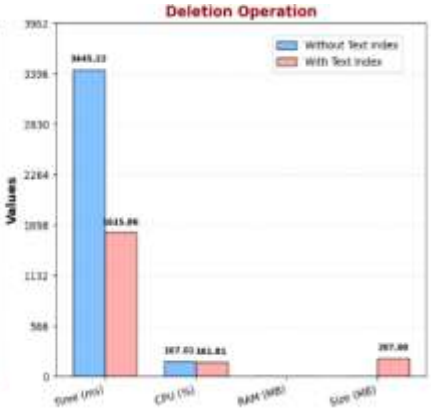


Figure 9: Query execution time — Retrieval operation, Text Index **Figure 10: Query execution time — Update operation, Text Index** **Figure 11: Query execution time — Deletion operation, Text Index**

Figure 12 illustrates the performance improvement rates for the three operations using the text index.

Figure 12: Performance improvement rates — Text Index

Discussion:-

The experimental results obtained in this study highlight the considerable contribution of indexing techniques to the performance of PostgreSQL queries, for all three evaluated index types: the simple index, the composite index, and the text index. In general, all three index types produce significant performance gains for retrieval and update operations, with improvement rates consistently exceeding 96%. These results can be explained by the ability of indexes to reduce the search space, thereby avoiding full sequential table scans, which constitute one of the main sources of latency in large-scale relational database management systems.

Index-Specific Performance Analysis:-

Regarding the simple index, the results reveal substantial improvements, with gains reaching 96.5% for retrieval and 98.0% for updates. These gains demonstrate the ability of this index to efficiently accelerate read and update operations on high-cardinality columns. Moreover, the low memory footprint generated by this index, estimated at 15.99 MB, makes it a particularly cost-effective optimization choice considering the performance benefits it provides.The composite index presents similar, and even slightly superior, results for update operations (98.7%), while occupying more memory space (23.8 MB). This index proves particularly suitable for queries involving multiple columns simultaneously, a common situation in complex information systems.

The text index stands out with exceptional gains: 98.8% for retrieval and 99.9% for updates. These results are explained by the very nature of textual queries, which, without indexing, require exhaustive scanning of text fields across all records—an especially costly operation on a dataset containing 500,000 entries. However, the storage footprint of this index is significantly larger (approximately 207 MB), which represents a trade-off to consider when deploying it in resource-constrained environments. These observations are consistent with the conclusions of previous studies [5], which emphasize the effectiveness of PostgreSQL GIN indexes for full-text search.

Write Operation Overhead and CPU Stability:-

For all evaluated indexes, deletion operations show more moderate gains, ranging from 51.4% to 60.3%. This trend can be explained by the overhead associated with maintaining index structures during record deletion: the system must not only remove the row from the table but also update the corresponding entries within the index.

Furthermore, CPU utilization indicators remain globally stable between configurations with and without indexes, consistently ranging between 156% and 169% across all experiments. Two points deserve clarification in this regard. First, values exceeding 100% reflect the cumulative measurement across all logical cores of the processor, rather than a single-core utilization percentage; on the HP All-in-One 24 used in this study, a value of approximately 160% corresponds to the utilization of roughly 1.6 out of 4 logical cores, or about 40% of total processing capacity. Second, the stability of CPU metrics despite dramatic reductions in execution time is consistent with the nature of the optimization achieved: sequential scans, which dominate unindexed query execution, are primarily I/O-bound operations that consume relatively little CPU while generating high latency. The performance gains delivered by indexing therefore manifest as reductions in disk access latency and data traversal time, rather than as reductions in active CPU processing load. This explains why execution time improves by up to 99.9% while CPU utilization varies by only 3 to 8 percentage points across all evaluated configurations.

Practical Guidelines for Index Selection:-

Ultimately, the choice of index type should be guided by the nature of the queries, the characteristics of the data (cardinality, type, and distribution), and the operational constraints of the system. The simple index constitutes a lightweight and versatile solution for high-cardinality columns. The composite index is preferable for frequent multi-column queries. Finally, the text index becomes essential whenever search operations involve large textual fields.

Conclusion:-

This study presented a comparative experimental evaluation of three types of indexes applied to PostgreSQL, namely the simple index, the composite index, and the text index, using a synthetic dataset of 500,000 records. The experiments focused on three fundamental data manipulation operations: retrieval, updating, and deletion, measured through four performance indicators: execution time, CPU usage, memory consumption, and storage space allocated to the index.The results clearly demonstrate the effectiveness of indexing mechanisms in improving PostgreSQL query performance. The most significant gains concern retrieval and update operations, with improvement rates reaching 96.5% and 98.0% respectively for the simple index, 96.3% and 98.7% for the composite index, and 98.8% and 99.9% for the text index. Deletion operations, although improved, exhibit more moderate gains ranging from 51.4% to 60.3% depending on the type of index, due to the overhead associated with maintaining index structures.

These findings confirm that the careful selection and rigorous implementation of an indexing strategy constitute an essential optimization lever in any large-scale data management system. They also highlight that each type of index presents specific advantages depending on the nature of the operations and the characteristics of the processed data. As such, these results have considerable practical significance for organizations, providing them with concrete guidance for rapidly selecting the indexing strategy best suited to their needs, thereby accelerating operational decision-making in the face of growing data volumes.

Several directions could extend this work. First, it would be relevant to include other native PostgreSQL index types, such as GiST, GIN, BRIN, or SP-GiST, to broaden the comparison to more specialized use cases (geospatial data, time-series data, and JSON data). Second, replicating the experiments on real-world datasets of varying sizes — from a few thousand to several million records — would allow researchers to assess the scalability of each indexing strategy under more realistic and heterogeneous conditions, and to identify potential performance thresholds as data volumes grow. Finally, a comparative evaluation across other relational DBMSs would help position PostgreSQL's indexing performance within the broader landscape of modern data management systems.

References:-

1. Hamadou, A., Hassane, A. A. I., Naroua, H., Hama, A. N. A., Salissou, A. S., & Chaharou, I. M. L. A COMPARATIVE EXPERIMENTAL STUDY OF INDEX PERFORMANCE IN MONGODB AND POSTGRESQL. (2026). International Journal of Engineering Sciences & Research Technology, 15(4), 22-31. https://doi.org/10.64149/j.ijesrt.15.4.22-31
2. Roy, M. M. (2023, April 27). Le coût réel des performances des base de données faible. ManageEngine Blog. https://www.manageengine.com/fr/blog/general/le-cout-reel-dun-systeme-de-base-de-donnees-faible.html[Accessed May. 21, 2026]
3. Cavaillès, D. (2026, January 30). PostgreSQL : les chiffres qui montrent l'adoption du SGBD. WeLoveDevs.com. [Online]. Available: https://welovedevs.com/articles/les-chiffres-qui-montrent-ladoption-de-postgresql/ [Accessed May. 21, 2026]
4. Zvazhii, D. V., & Gorokhovsky, S. S. (2026). Implementing indexes in POSTGRESQL. PROBLEMS IN PROGRAMMING, (1), 14-24.
5. Toktomusheva, G. (2025). Indexing in PostgreSQL: Performance Evaluation and Use Cases.
6. Zolotukhina, D. Y. E. (2025). Comparative analysis of indexing strategies in PostgreSQL under various load scenarios. Software systems and computational methods, (1), 21-31.
7. Dombrovskaya, H., Novikov, B., & Bailliekova, A. (2021). PostgreSQL Query Optimization. Apress.
8. Fuad, A., Vashisth, S., Balmau, O., & Kemme, B. (2026, April). PostLearn: Towards A Learned Index For PostgreSQL. In Proceedings of the 6th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (pp. 20-25).
9. Abbasi, M., Bernardo, M. V., Vaz, P., Silva, J., & Martins, P. (2024). Revisiting Database Indexing for Parallel and Accelerated Computing: A Comprehensive Study and Novel Approaches. Information, 15(8), 429. MDPI.
10. Salunke, S. V., & Ouda, A. (2024). A performance benchmark for the PostgreSQL and MySQL databases. Future Internet, 16(10), 382.
11. Urnikienė, J., Steponavičienė, V., & Atanasov, S. (2026). Comparative Read Performance Analysis of PostgreSQL and MongoDB in E-Commerce: An Empirical Study of Filtering and Analytical Queries. Big Data and Cognitive Computing, 10(2), 66.