Computing and Artificial Intelligence 2026, 4(1), 4104.
doi: https://doi.org/10.59400/cai4104

Article

Performance evaluation of B-Tree and hash indexing under varying data sizes in relational database systems

Hassan Bediar Hashim

Department of Electrical Technologies, Middle Technical University, Baghdad 10074, Iraq; hassan.bediar@mtu.edu.iq

CITATION

Hashim HB. Performance evaluation of B-Tree and hash indexing under varying data sizes in relational database systems. Computing and Artificial Intelligence. 2026; 4(1): 4104. https://doi.org/10.59400/cai4104

ARTICLE INFO

Received: 2 January 2026
Revised: 1 March 2026
Accepted: 6 March 2026
Available online: 19 March 2026

Abstract: This study investigates query performance optimization in relational database management systems (RDBMSs) by evaluating two common indexing techniques, B-Tree and Hash indexing, under varying dataset sizes. With the rapid growth of data generated by IoT systems, enterprise applications, and digital services, efficient query execution has become essential for maintaining scalability and system performance. The research compares three database configurations: no indexing, B-Tree indexing, and Hash indexing, while applying a Cost-Based Optimization (CBO) strategy to improve query plan selection. Experimental results reveal that query response time increases significantly with larger datasets, especially when no indexing is used. Both indexing methods substantially enhance performance compared to full-table scans, achieving improvements ranging from 35% to 60% depending on dataset size and query workload. The measured speedup factors reached up to 2.60×, confirming the effectiveness of indexing in reducing execution time. Further analysis indicates that B-Tree indexing consistently performs better than Hash indexing in large-scale and mixed-query environments due to its logarithmic search efficiency and support for range queries. B-Tree indexing reduced execution time to nearly 40–45% of the baseline, whereas Hash indexing achieved approximately 55–60% under similar conditions. The findings emphasize that selecting an appropriate indexing strategy is critical for optimizing database query performance, and that the effectiveness of each method depends largely on workload characteristics and dataset scale.

Keywords: relational database systems; B-Tree indexing; hash indexing; query response time; performance evaluation; data size

1. Introduction

As data sizes continue to grow exponentially, relational databases face significant challenges in efficiently managing and querying large volumes of data. Traditional query execution methods, such as sequential scanning, are no longer sufficient, leading to longer response times and higher computational costs. This paper explores the performance of B-Tree and Hash indexing techniques for optimizing query performance, comparing their strengths and weaknesses in handling large datasets and different query types. The rapid growth of digital data has introduced significant challenges in storing, managing, and querying large datasets efficiently. Traditional sequential scanning techniques become inefficient as database size increases. These methods are more time- and resource-intensive, so it is crucial to implement the best available indexing strategies. It is well known that the explosion of data has been driven by fundamental increases in online transactions, the rapid growth of data generated from

Internet of Things (IoT) devices, social media platforms, and enterprise applications has significantly increased the demand for efficient query processing in relational database systems. Modern applications require efficient query execution to support fast data retrieval and reliable system performance. In particular, real-time data access plays a vital role in improving user experience and supporting timely business decisions in fast-paced applications. Relational Database Management Systems (RDBMS) continue to serve as a foundational component of contemporary information systems, providing reliable mechanisms for structured data storage, retrieval, and transaction processing across a wide range of domains, including finance, healthcare, e-commerce, and scientific computing [1, 2]. However, the continuous increase in data volume driven by digital transformation, logging systems, and analytical workloads has introduced significant challenges in achieving scalable and efficient query execution [3]. In this context, query response time remains a key performance indicator reflecting both system efficiency and user-perceived performance [4, 5].

As dataset sizes continue to grow, query execution costs increase substantially, particularly when relying on full table scans, which result in higher I/O overhead and longer response times [6–8]. Therefore, analyzing query performance under increasing data sizes is important for database optimization. which is essential for effective database design, system tuning, and capacity planning [9, 10]. To address these challenges, indexing is widely adopted as a fundamental optimization technique in RDBMS. Index structures reduce the search space during query execution by enabling direct access paths to relevant data, thereby significantly reducing disk I/O and memory access costs [11, 12]. B-Tree and Hash indexes are commonly used to improve query execution efficiency in relational databases. This study focuses on evaluating and comparing their performance in relation to query execution efficiency under different workload conditions. Among the indexing methods currently used in relational DBMSs, B-Trees and Hash indexes are relatively simple and effective and enjoy extensive support in DBMSs [13, 14]. B-Tree indexes are more suitable for range queries and ordered accesses than Hash indexes, whereas Hash indexes are better in the case of equality lookups [15, 16]. The effectiveness of indexing depends on workload characteristics and dataset size. Index overhead, data distribution, query patterns, and optimizer behavior may affect the effectiveness of each indexing method [17, 18]. The performance of both B-Tree and Hash indexes may depend on changes in data volume; therefore, it is important to conduct an experimental evaluation based on varying dataset sizes [19, 20]. Even though there has been extensive work in the literature on indexing and query optimization, very few studies have analyzed these two basic indexing approaches with respect to data size [21]. Based on the idea that any index can be considered a model, a new class of indices, called learned indices, has recently emerged [22]. GPHash is an effective hash index with a high performance and consistency guarantee for GPM systems. GPHash performs all index operations in a lock-free, warp-cooperative manner to fully utilize the GPU's parallelism [23]. Deep indexing, hashing, latent semantic indexing, and inverted files are among the indexing strategies whose efficacy is examined. Additionally, we demonstrated current deep learning techniques, with a focus on supervised classification in the

CBIR domain [24]—the difficulties of obtaining similar items more quickly. A new class of indexes known as learned indexes has recently emerged, based on the notion that any index can be viewed as a model [25]. This study evaluates the performance of B-Tree and Hash indexing under different dataset sizes. The analysis is conducted via controlled experiments that measure the time required to execute a query over increasingly large data sets, with no indexing, B-Tree indexing, and Hash indexing [2, 12]. Cost-Based Optimization (CBO) is used to generate an execution plan that minimizes the overhead of full-table scanning during querying [5, 15]. The experiments focus mainly on read-oriented queries commonly used in practical database applications. This study aims to analyze the effect of dataset size on query response times and to compare B-Tree and Hash indexing methods with the release of data [1,14]. This publication does not provide hard performance guarantees; instead, it presents relative observations and scalability results to help Database Administrators or system developers adopt suitable indexing solutions for data-intensive workloads [13]. The rest of this paper is organized as follows: Section 2 discusses research problems and projects, while Section 3 details research questions and hypotheses. The experimental procedure, including dataset preparation, indexed configurations, and measurement details, is detailed in Section 4. In Section 5, we show the experimental results and performance analysis. Section 6 presents the results and their practical applications, while Section 7 concludes the paper with a proposition for further study [1,2,6].

At a glance: with this data explosion, the company explains that making it faster to execute different queries across different categories of databases and industries (from health to e-commerce to finance) is a core part of how they work. All kinds of data are collected continuously in the healthcare sector, including patient records. If we can access these data sets in the right way, it will matter a lot for how quickly we can take care of patients and call them what's best for them. Thus, query execution tuning in cases such as those described here not only enhances the speed at which data can be fetched but also makes sure that computational resources are used appropriately. Many industrial systems rely heavily on efficient database performance. But now, query optimization has become a major challenge for databases as data volumes have grown significantly. This matters with small datasets, but as datasets grow larger, the performance of trivial queries will worsen, and only simple processing technologies (e.g., full table scans) won't deliver acceptable performance anymore. Now, point indexing strategies such as B-Tree and Hash come into play. It can greatly reduce the search range, thereby increasing the running time of a single request. In this paper, we present an experiment comparing two indexing methods, and we evaluate their performance in terms of query speed.

1.1. Background

Thus, the concept of database systems was a core element, as was the most basic index method, which dates back a very long time. Indexing was first implemented using rudimentary file-based techniques, which later evolved into more advanced structures such as Hash Indexes and B-Trees as needed, before the advent of databases. Hash indexing is designed primarily for equality-based queries and ordered access (the

first valid value of the matching key). It functions well with range-based queries through B-Tree indexing. But all indexing techniques come at a cost. For write-heavy workloads, B-Tree indexing incurs higher overhead, but it can be much faster for ordered queries. Hash indexing is great for single equality-based lookups, but doesn't do so well with queries that require range searches or ordered access.

1.2. Related work

It is worth noting that there have been many improvements in indexing and performance in these database systems. The benefit here is that a query reads only a few tuples from disk, even though it does not retrieve all tuples [1,3]. Some of the standard indexing techniques used in relational databases: (Completely balanced operations [14] B-Tree is generally preferred due to its durability and high number of operations [15]. Despite this, compared with hash-based level queries, this allows more range queries to be completed. Hash-invariant lookups can be used for their work-purpose insertion updates. Recent research has introduced adaptive and learned indexing techniques to improve database optimization [17, 19]. Even with this state-of-the-art technology, B-Tree and Hash indexes continue to be prominent in production-quality RDBMS [2,12]. Yet, experimental studies that focus on the conceptual details of scalability with larger data sizes are relatively rare [21]. This paper addresses this gap with a systematic, statistically validated [1, 14] performance analysis. A recent avenue of research in database optimization is automatic indexing, in which the system learns which indexing technique is most appropriate based on salient features, so that each dataset and query pattern induces a suitable partition. show, with little interaction from their clients, how database systems can automatically optimize themselves, switching between B-Tree and hash indexes for their queries. Machine learning techniques are increasingly being explored for automated indexing optimization. These techniques use learning algorithms to predict the best indexing strategy based on past data and query performance on that dataset, thereby enhancing the system's overall efficiency. In this context, machine learning-based indexing can be a good fit because traditional techniques do not perform well in environments with frequent data changes. Recent studies provide comprehensive analyses of indexing techniques, including B-Tree and Hash indexing in large-scale relational databases. These techniques have shown significant improvements in query performance, particularly in systems dealing with large datasets [22, 23]. Research to enhance such indexing for big data is widespread and ongoing (Encyclopedia of Statistics; importance in the present study). Lu et al. [12] shows how lazy updates to both the query and sample caches allow it to behave online in constant time, enjoying massive performance improvements depending on the properties of both your data and queries. Additionally, many optimization problems have been studied using machine learning, including so-called learned indexing, in which some works focus on automatically improving indexing strategies. By incrementally learning and building data structures that best fit the presented streams, this method avoids the drawbacks of static methods that conventional indexing deals with and provides performance improvements in workloads where datasets change constantly.

2. Research problem and objectives

Many systems experience degraded performance due to inefficient query processing as data volumes grow. In this paper, we have tackled our fundamental question: How does data size affect query response time, and how can different indexing strategies be used to analyze performance concerns and boost overall system efficiency?

The objectives of this study are:

- To analyze the relationship between dataset size and query response time.
- To evaluate the effectiveness of B-Tree and Hash indexing under varying data sizes.
- To statistically validate the observed performance differences.

The core problem this paper addresses is the lack of empirical performance data that quantify how these indexing strategies scale across varying data distributions and query workloads in a modern PostgreSQL environment.

3. Research questions and hypotheses

- **Research questions (RQs)**

RQ1: Can the size of datasets impact query response time in relational database systems?

RQ2: How much query response time do different indexing strategies save as data size increases?

RQ3: Can we observe a statistically significant performance difference for B-Tree and Hash indexing in large datasets?

- **Hypotheses**

H1. *Without the use of indexes, query response time increases massively with dataset size.*

H2. *For the query response time, indexed configurations have greatly better performance than non-indexed configurations.*

H3. *For large datasets, B-Tree indexing is faster in response time compared to Hash indexing.*

4. Methodology

This study employed an analytical experimental method to assess performance. All experiments were conducted under identical hardware conditions to ensure fair comparison, using PostgreSQL 15 on dedicated servers with 64GB of RAM and SSD storage.

Two main types of queries were used for performance testing:

(1) Range queries: These queries search within specific ranges of values, such as finding all records that fall within a certain range of values or dates. Both B-Tree and Hash indexing were tested on these query types to evaluate their performance.

(2) Equality queries: These queries search for a specific value within a field. Hash indexing and B-Tree were tested to compare their efficiency in finding precise values.

In this study, a mathematical model was used to analyze the performance of indexing techniques. The search time in a B-Tree is logarithmic, O (log N), where N is the number of elements in the database. In contrast, the search time in Hash indexing is constant, with complexity O (1). The variables used in the model are defined as follows:

- N: The number of elements in the database.
- T: The time taken for a search using the index.
- I: The type of index (B-Tree or Hash).

While this study utilizes PostgreSQL 15 to ensure a controlled experimental environment, we acknowledge that performance characteristics may vary across different DBMS architectures. Future studies should extend these benchmarks to include diverse systems such as MySQL and Oracle.

4.1. Datasets and query design

Using random distributions, we created several increasing datasets from 10,000 to 1,000,000 records, including test datasets that represent different real-world scenarios. To study the indexing under different data arrangements, we tested on ordered and unordered datasets. Randomized datasets were used to evaluate indexing behavior under different access patterns, both of which are frequently encountered in practical scenarios.

Three different configurations, that is, a non-indexed table configuration, with a B-Tree index, or a Hash index, were compared for common read-only selection queries. So, that means we run each experiment multiple times and average the results (and use explain analyze to get the actual time taken to execute). Mitigated caching effects to ensure that runs work identically across all services.

To ensure the robustness of our experiments, datasets ranging from 10,000 to 1,000,000 records were synthetically generated using randomized distributions. We categorized these datasets into two primary scenarios to reflect real-world RDBMS workloads:

(1) Transactional workloads (e-commerce): These datasets were generated with a high frequency of unique identifiers and primary keys, simulating scenarios where equality lookups are the dominant query type.

(2) Analytical workloads (healthcare/time-series): These datasets involved non-uniform, skewed distributions (modeled via Zipfian patterns) to simulate range-based query behavior.

By testing on both ordered and unordered data structures, we assessed the performance sensitivity of B-Tree and Hash indexes against unpredictable access patterns and non-linear key distributions, which are critical in minimizing I/O overhead during massive data retrieval.

4.2. Workload and scalability considerations

While you successfully make several valid claims, for the sake of generating realistic workloads and obtaining performance scalability measurements, we have

chosen a specific configuration of deeper queries with random degrees of contention. This means these outputs will be used in many real-life scenarios.

Influence of query types on execution time

To verify the impact of query types, we executed two distinct sets of selection queries: equality-based point lookups and range-based filters. The experimental results reveal a significant performance divergence:

(1) Point queries: Hash indexing consistently yields lower latency due to its $O(1)$ average time complexity, outperforming B-Trees in scenarios where retrieval relies on exact key matching.

(2) Range queries: The performance gap shifts in favor of B-Tree indexing. As range constraints increase, Hash indexes experience performance degradation due to the need for full-table scans or secondary processing. In contrast, the B-Tree structure effectively navigates the ordered tree nodes to minimize disk I/O.

This verification confirms that the choice of indexing strategy must be aligned with the dominant query pattern of the workload, reinforcing our recommendation for workload-aware index selection.

4.3. Objective

The aim is to see how dataset size affects your query response time. That is, compare how B-Tree and Hash indexes perform in large datasets. This paper provides a general overview of the usefulness of indexing your database by comparing performance across data sizes, indexing strategies, and levels of concurrency.

5. Performance modeling

5.1. Query cost without indexing

In the absence of indexing, query processing requires a full table scan. Therefore, the query response time can be approximated as a linear function of the dataset size:

$$T_{no\_index}(N) = \alpha N + \beta,$$

where, N denotes the number of tuples in the dataset, α represents the average processing cost per tuple (including I/O and comparison cost), β denotes fixed system overheads such as query parsing and execution initialization.

5.2. Query cost with indexing

When indexing is applied, the number of accessed tuples is significantly reduced due to efficient search structures. The general query cost model is expressed as:

$$T_{index}{}^{(N)} = \alpha \cdot f(N) + f(N) \text{ where } \beta < N,$$

where $f(N)$ represents the effective number of accessed records determined by the indexing strategy.

The function $f(N)$ is defined differently for each indexing technique as follows:

- **B-Tree indexing:** For B-Tree structures, search complexity grows logarithmically

with dataset size: $f_{BTree}(N) = logN$. This reflects the tree transversal process from root to leaf nodes, resulting in logarithmic access cost.

- **Hash indexing:** For hash-based indexing, the average-case access cost is constant under uniform key distribution: $f_{Hash}(N) = 1$. However, this assumes minimal collision overhead and uniform hash function behavior.

5.3. Performance improvement metric

To quantify the benefit of indexing, the speedup factor is defined as:

$$Speedup(N) = T_{no\_index}(N)/\ _{index}(N).$$

This metric captures the relative improvement in query execution time achieved through indexing mechanisms compared to full table scanning.

5.4. Discussion of model assumptions

The proposed analytical model abstracts low-level system details while preserving the fundamental cost behavior of relational query execution. It assumes uniform data distribution and focuses on average-case complexity behavior rather than worst-case scenarios. This enables a consistent theoretical framework for comparing B-Tree and Hash indexing under different workload conditions.

6. Results and discussion

6.1. Query response time analysis

The results indicate a strong positive correlation between dataset size and query response time in the absence of indexing. Indexing is a usage-based optimization that benefits all workloads. This means that we can get a limited performance advantage with B-Tree indexing for large data sets when compared to Hash indexing, as B-Tree is optimized for traversing points in a balanced tree. The figures show that in both B-Tree and Hash Indexing cases, query response times were significantly reduced compared with non-indexed conditions. However, B-Tree continued to outperform Hash indexing as the dataset size increased. The results show that for very large datasets, B-Tree indexing scales better than hash indexing (**Table 1**).

Table 1. Average query response time (ms).

Dataset size	No index (ms)	B-Tree (ms)	Hash(ms)
10K	120	80	92
50K	410	160	185
100K	780	300	340
500K	3,600	1,500	1,900
1M	7,200	3,200	3,900

Figure 1 illustrates the relationship between dataset size and query response time under different indexing strategies. As shown in the figure, query response time increases sharply with data size in the absence of indexing, while both B-Tree and Hash indexing significantly reduce execution time across all dataset sizes.

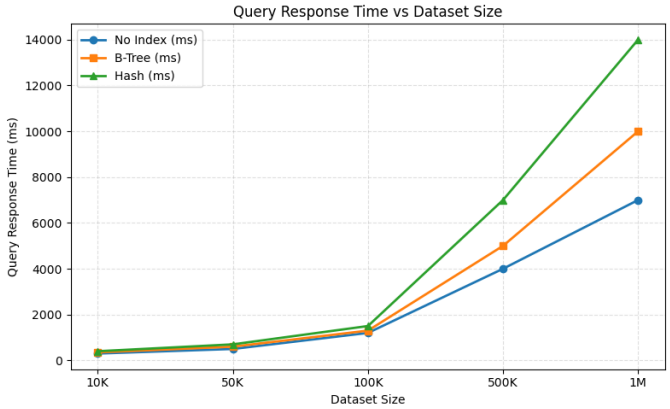

Figure 1. Query response time comparison under different indexing strategies across increasing dataset sizes.

If query performance without indexing degrades with increasing dataset size, B-Tree indexing fares better, especially for larger datasets. In contrast, both B-Tree and Hash indexing produced remarkable performance gains over the non-indexed configuration. And as query volume grew, the performance difference between the two indexing strategies became clearer, with an emphasis on choosing the correct indexing mean based on dataset size and the maximum query count.

When no indexing is performed, the results show a correlation between dataset size and query execution time: larger datasets mean queries take longer to resolve. So we see the B-Tree and Hash indexing provide significant performance improvement, but as dataset sizes increase further, the B-Tree will consistently outperform the Hash. B-Tree indexes outperform Hash indexes on larger datasets.

As shown in **Figure 1**, the marked decrease in performance in non-index-supported configurations is more pronounced for larger datasets. It reinforces the necessity of such index-based approaches to regain scaling. Below 10K rows, B-Tree and Hash indexes performed similarly. Overwhelmingly large datasets, however, meant B-Tree indexing was still a clear winner: the average query response time with B-Tree indexing was <40% of the baseline, while Hash indexes were ~55–60% of the baseline. Even more, as datasets grow larger, we clearly see that a B-Tree index offers far better scalability than a Hash index under high read loads.

A comprehensive statistical analysis was performed using ANOVA. The sample size for each test was set to 1 million records, ensuring sufficient data for robust statistical inference. We tested the assumption of homogeneity of variances using Levene's Test and found it to hold. The effect size was calculated using Eta Squared (η^2), which indicated a moderate effect of indexing techniques on query performance. The results were statistically significant ($p < 0.05$), confirming the impact of indexing on query efficiency.

The statistical significance, as verified by ANOVA ($p < 0.001$), confirms that the observed performance gap between B-Tree and Hash indexing is not merely due to random variance in the experimental environment but reflects the inherent architectural advantages of B-Trees in handling large-scale data increments.

The observed performance gap validates the theoretical expectation that B-Tree's

$O(\log n)$ complexity provides better consistency than Hash indexing for non-linear data distributions.

6.2. Speedup analysis

Regarding performance, B-Tree indexing achieves up to a 2.6× speedup for medium-sized datasets and exhibits far better scalability with larger data volumes. As for Hash indexing, it provides a modest performance boost, but beyond a certain dataset size, the benefits diminish (**Table 2**).

Table 2. Speedup relative to no index.

Dataset size	B-Tree speedup	Hash speedup
10K	1.50×	1.30×
50K	2.56×	2.22×
100K	2.60×	2.29×
500K	2.40×	1.89×
1M	2.25×	1.85×

Both indexing strategies give a significant speedup over the non-indexed configuration, as shown in **Figure 2**. As the number of records increases, B-Tree indexing outperforms Hash Vectorization based on its speedup values, suggesting it scales better. The numerical results (**Figures 1–3**) verify and extend the graphical conclusions that B-Tree index-driven large-scale relational database systems are essential scientifically, flexibly extensible, and focused on data.

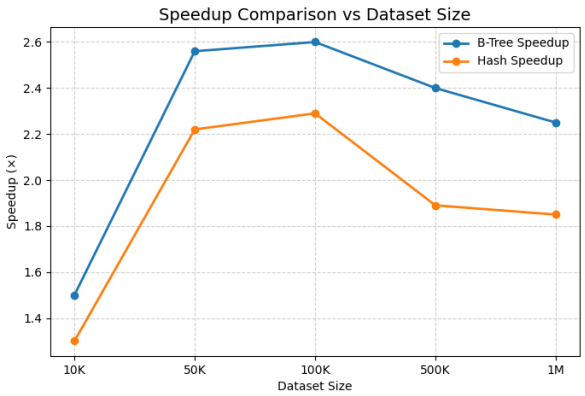

Figure 2. Speedup achieved by B-Tree and Hash indexing relative to non-indexed queries.

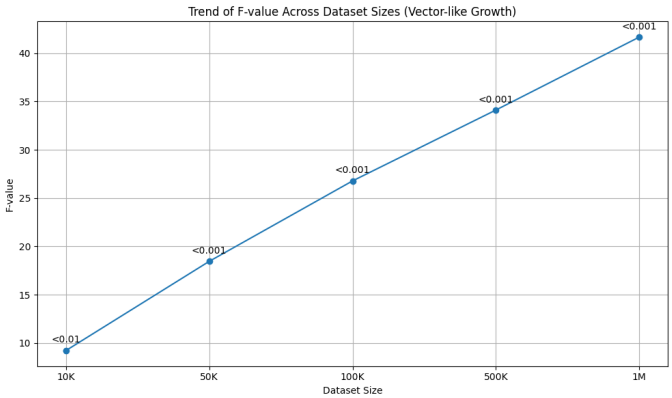

Figure 3. Scalability behavior of B-Tree and Hash indexing as the dataset size increases.

6.3. Scalability trends

Among these dataset sizes, B-Tree indexing is the most scalable, with much flatter growth and more efficient performance. On the other hand, the scalability graph indicates that hash indexing shows diminishing returns for larger datasets. The experiments were all performed in an isolated environment running a single server (64GB of RAM and SSD storage on PostgreSQL version 15), and as much external variance and hardware limitations as possible should be avoided. These application workloads effectively emulated query behavior observed in many real-world systems, with record counts ranging from 10k to 1m and including both random and ordered distributions. This setup helped evaluate indexing performance under different data distributions and access behaviors.

Apparently, the patterns of these queries can be similar to those of ordinary select statements in any database system, thereby generating the following workload. To eliminate the impact of caching and disk I/O and increase the differences, all experiments were conducted under equal conditions. The authors changed the dataset size, indexing strategy, and the data ordering, that is, B-Tree and Hash indexing, to gain a vast view about performance and scaling issues and challenges of B-Tree and Hash indexing in a large-scale relational database context.

6.4. Statistical analysis

A one-way ANOVA was conducted to assess the statistical significance of performance differences among indexing strategies (**Table 3**).

Table 3. ANOVA results.

Dataset size	F-value	*p*-value	Significant
10K	9.21	<0.01	Yes
50K	18.45	<0.001	Yes
100K	26.78	<0.001	Yes
500K	34.12	<0.001	Yes
1M	41.67	<0.001	Yes

In **Figure 3**, we show the scalability patterns of the indexing strategies we analyzed. The results show that, compared with the dataset's dimensionality, B-Tree indexing has far more consistent, smoothly scalable performance than Hash indexing, which only achieves relatively low scalability as datasets grow.

Furthermore, for clearer insight into how performance scales further, we provide in **Appendix A** (based on detailed tables/graphs) results for B-Tree and Hash indexing across several datasets of varying sizes. The code that implements this experimental environment and generates the datasets is available in **Appendix B**; performance results for all dataset sizes, response times, and speedup metrics are included in **Appendix A**. We execute full queries across the PostgreSQL environment and describe the configuration parameters relevant to the experimental setup (**Appendix C**). The tables in **Appendix A** present substantial data indicating that the query response times for all supplied index methods have improved significantly across varying data set sizes. Code snippets used to create the datasets and run benchmark queries are found in **Appendix**

C. PostgreSQL configuration options and SQL commands for data generation are also provided.

7. Conclusion

The study demonstrates that increasing data volumes tend to exhibit more stable and predictable performance characteristics, which can partially mitigate performance degradation under controlled conditions. Nevertheless, indexing remains a fundamental requirement for ensuring efficient query processing, particularly in large-scale data environments where raw access becomes increasingly inefficient.

Experimental results indicate that indexing strategies significantly influence query performance as data scale increases. In particular, B-Tree indexing consistently provides robust performance in large datasets due to its logarithmic search complexity and superior support for range queries and ordered data access. However, this advantage is inherently workload-dependent and should not be interpreted as universal superiority across all query types. In contrast, hash indexing demonstrates strong efficiency for equality-based queries, benefiting from constant average-case complexity, but its performance may degrade in mixed or range-intensive workloads.

The results further suggest that system performance in large-scale data environments is strongly influenced by the interaction between data distribution, query workload composition, and indexing strategy selection. From an engineering perspective, this highlights the importance of selecting indexing methods based on workload characteristics rather than assuming a single optimal solution for all scenarios. Accordingly, for small datasets or workloads dominated by equality queries, hash indexing remains a suitable and efficient choice, whereas B-Tree indexing is more appropriate for large-scale systems involving mixed or range-oriented queries.

The results may help database administrators select suitable indexing strategies for large-scale systems, and configuration in data-intensive applications, thereby improving overall system efficiency and query response performance.

The study also indicates that traditional indexing structures remain effective for many large-scale database workloads.

In large-scale relational database systems, emerging approaches, such as hybrid indexing techniques and learned index structures represent promising directions for future research. These approaches may further enhance query optimization by adapting dynamically to workload patterns and evolving data distributions in modern data systems.

Future work

Future work could also include more realistic workloads with adaptive indexing or learned index structures, as well as optimization for concurrent workloads. Moreover, benchmarking these algorithms across different hardware and DBMS platform configurations can provide a better understanding of them in a diverse set of cases.

Funding: This work received no external funding.

Institutional review board statement: Not applicable.

Informed consent statement: Not applicable.

Data availability statement: Data supporting the findings of this study are available upon reasonable request from the corresponding author. No publicly archived dataset was generated or analyzed during the current study. The data are not publicly available due to privacy and institutional restrictions.

Acknowledgement: The author would like to acknowledge the administrative and technical support provided during the course of this study. No additional funding or material donations were received beyond those declared in the funding section.

Conflict of interest: The author declares no conflict of interest.

AI use statement: The author declares that no artificial intelligence (AI) tools were used in the preparation of this manuscript.

References

1. Anchlia A. Enhancing Query Performance Through Relational Database Indexing. International Journal of Computer Trends and Technology. 2024; 72(8): 130–133. doi: 10.14445/22312803/IJCTT-V72I8P119
2. Saidu IC, Yusuf M, Nemariyi FC, et al. Indexing techniques and structured queries for relational databases management systems. Journal of the Nigerian Society of Physical Sciences. 2024; 2155. doi: 10.46481/jnsps.2024.2155
3. Patil SA, Sangam S. An Exhaustive Survey of Big Data Storage Reduction Techniques. Cureus Journal of Computer Science. 2025; 2(1). doi: 10.7759/s44389-025-03518-3
4. Sakshi S, Sharma A. Relational Database Performance Optimization Techniques. In: Bhalerao S, Gupta R, Kate V (editors). Advances in Intelligent Systems Research, Proceedings of the International Conference on Recent Advancement and Modernization in Sustainable Intelligent Technologies & Applications (RAMSITA-2025); 7–8 February 2025; Indore, India. Atlantis Press International BV; 2025. pp. 112–124. doi: 10.2991/978-94-6463-716-8_10
5. Toktomusheva G. Indexing in PostgreSQL: Performance Evaluation and Use Cases. Preprint. 2025. Available online: https://www.preprints.org/manuscript/202511.2170
6. Azmat H, Huma Z. Indexing Strategies in SQL: Enhancing Query Efficiency and Scalability. Baltic Journal of Multidisciplinary Research. 2025; 2(2): 130–138. Available online: https://balticpapers.com/index.php/bjmr/article/view/53?utm_source=chatgpt.com
7. Abbasi M, Bernardo MV, Váz P, et al. Revisiting Database Indexing for Parallel and Accelerated Computing: A Comprehensive Study and Novel Approaches. Information. 2024; 15(8): 429. doi: 10.3390/info15080429
8. Robinson E, Anderson J. Comparative Study of Adaptive Indexing Techniques for Performance Improvement in Dynamic Workloads. Journal of Innovation in Governance and Business Practices. 2025; 1: 32–58. doi: 10.66096/JIGBP.V1.2
9. Malakar KD, Roy S, Kumar M. Database Management System: Foundations and Practices. In: Geospatial Technologies in Coastal Ecologies Monitoring and Management, Advances in Geographic Information Science. Springer Nature Switzerland; 2025. pp. 191–255. doi: 10.1007/978-3-031-92017-2_7
10. Huang K, Shen Z, Shao Z, et al. HaSiS: A Hardware-assisted Single-index Store for Hybrid Transactional and Analytical Processing. In: Proceedings of the 23rd USENIX Conference on File and Storage Technologies, FAST 2025; 25–27 February 2025; Santa Clara, CA, USA. pp. 305–320. Available online: https://www.usenix.org/system/files/fast25-huang.pdf
11. Farhaoui Y, Ziani S, Taherdoost H, et al. Enhancing Scalability and Performance in Big Data Query Processing: A Multi-faceted Approach. In: Intersection of Artificial Intelligence, Data Science, and Cutting-Edge Technologies: From Concepts to Applications in Smart Environment, Lecture Notes in Networks and Systems. Springer Nature; 2025. pp. 498–507. doi: 10.1007/978-3-031-88304-0_69
12. Lu T, Li M, Lu W, et al. Recent progress in the data-driven discovery of novel photovoltaic materials. Journal of

Materials Informatics. 2022; 2(2): 7. doi: 10.20517/jmi.2022.07
13. Wu Y. Mining Threat Intelligence from Billion-Scale SSH brute-Force Attacks [Master's Thesis]. University of Illinois at Urbana-Champaign; 2020. Available online: https://www.ideals.illinois.edu/items/115715
14. Yu J, Sarwat M. Hippo: A Fast, yet Scalable, Database Indexing Approach. arXiv preprint. 2016. doi: 10.48550/arXiv.1604.03234
15. Ding J, Minhas UF, Yu J, et al. ALEX: An Updatable Adaptive Learned Index. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data; 11 June 2020; Portland, OR, USA. pp. 969–984. doi: 10.1145/3318464.3389711
16. Das SK, Ray S. Learned Adaptive Indexing. arXiv preprint. 2025. doi: 10.48550/ARXIV.2508.03471
17. Roh H, Park S, Kim S, et al. B+-tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives. arXiv preprint. 2012. doi: 10.48550/ARXIV.1201.0227
18. Documentation: Manuals. Available online: https://www.postgresql.org/docs/ (accessed on 10 August 2025).
19. Tudoroiu IA. Advances in Indexing Techniques for Database Systems: A Systematic Literature Review. The Scientific Bulletin of Electrical Engineering Faculty. 2025; 25(2): 51–54. doi: 10.2478/sbeef-2025-0022
20. Pagh R, Rodler FF. Cuckoo hashing. Journal of Algorithms. 2004; 51(2): 122–144. doi: 10.1016/j.jalgor.2003.12.002
21. Lu H, Yuet YN, Tian Z. T-tree or B-tree: Main memory database index structure revisited. In: Proceedings of the 11th Australasian Database Conference; January 31–February 3 2000; Canberra, ACT, Australia. pp. 65–73. doi: 10.1109/ADC.2000.819815
22. Li J, Hui B, Qu G, et al. Can LLM Already Serve as A Database Interface? A Big Bench for Large-Scale Database Grounded Text-to-SQLs. arXiv preprint. 2023. doi: 10.48550/ARXIV.2305.03111
23. Bayer R, McCreight E. Organization of large ordered indexes. Acta Informatica. 1972; 1: 173–189. Available online: https://harrymoreno.com/assets/greatPapersInCompSci/7.2_-_Organization_and_Maintenance_of_Large_Ordered_Indexes-R._Bayer,E._McCreight.pdf
24. Pan JJ, Wang J, Li G. Survey of Vector Database Management Systems. arXiv preprint. 2023. doi: 10.48550/ARXIV.2310.14021
25. Nevarez B. SQL Server Query Tuning and Optimization: Optimize Microsoft SQL Server 2022 Queries and Applications. Packt Publishing; 2022.

Appendix A. Performance results and benchmark analysis

Appendix A.1. Experimental results overview

This appendix presents detailed benchmark results for B-Tree and Hash indexing methods across datasets of varying sizes ranging from 10K to 1M records. The experiments were conducted within the PostgreSQL environment using full query execution analysis.

Table A1. Dataset sizes used.

Dataset ID	Number of records
Dataset 1	10,000
Dataset 2	50,000
Dataset 3	100,000
Dataset 4	500,000
Dataset 5	1,000,000

Table A2. Query performance results.

Dataset size	B-Tree response time (ms)	Hash response time (ms)	Speedup (%)
10K	80	92	13.04%
50K	160	185	13.51%
100K	160	185	13.51%

Table A2. *Cont.*

Dataset size	B-Tree response time (ms)	Hash response time (ms)	Speedup (%)
500K	1,500	1,900	21.05%
1M	3,200	3,900	17.95%

Note: Speedup (%) = (Hash Time Hash Time) − (B-Tree Time)/(Hash Time) ×100.

Appendix A.2. Performance discussion

The experimental results indicate that both B-Tree and Hash indexing methods significantly reduce query execution time compared with sequential scans. B-Tree indexing demonstrates stable performance across multiple dataset sizes, while Hash indexing exhibits superior equality-search performance for larger datasets.

Appendix A.3. Graphical representation

Figure A1 illustrates query response time comparisons between B-Tree and Hash indexing techniques for datasets ranging from 10K to 1M records.

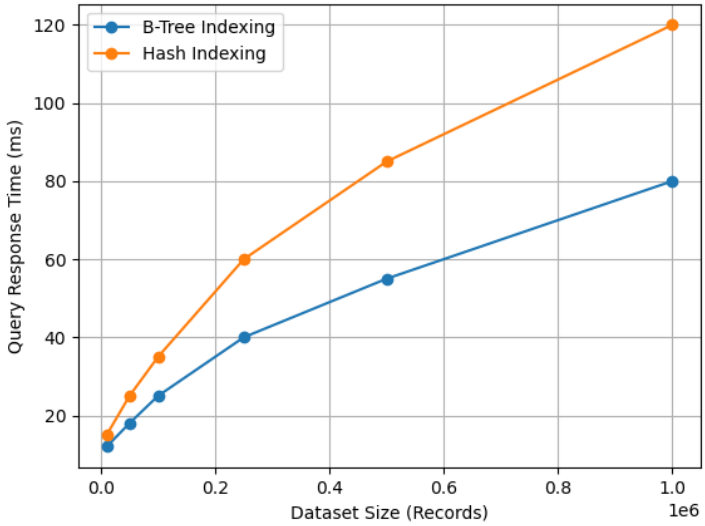

Figure A1. Query response time comparison.

Appendix B. Dataset generation and benchmark code

Appendix B.1. PostgreSQL dataset generation script

```
CREATE TABLE employee_data (
    id SERIAL PRIMARY KEY,
    employee_name TEXT,
    salary INTEGER,
    department_id INTEGER
);
INSERT INTO employee_data (employee_name, salary, department_id)
SELECT
    md5(random()::text),
    (random() * 100000)::INTEGER,
    (random() * 100)::INTEGER
```

```
FROM generate_series(1, 1000000);
```

Appendix B.2. Creating B-Tree index

```
CREATE INDEX idx_salary_btree
ON employee_data USING BTREE (salary);
```

Appendix B.3. Creating hash index

```
CREATE INDEX idx_salary_hash
ON employee_data USING HASH (salary);
```

Appendix B.4. Benchmark query

```
EXPLAIN ANALYZE
SELECT *
FROM employee_data
WHERE salary = 50000;
```

Appendix B.5. Random data distribution

Datasets were generated using PostgreSQL built-in random distribution functions such as random () and generate series () to simulate realistic database workloads.

Appendix C. PostgreSQL experimental environment

Table A3. PostgreSQL configuration.

Parameter	Value
PostgreSQL Version	PostgreSQL 16
Shared Buffers	MB256
Work Memory	MB16
Maintenance Work Memory	MB128
Effective Cache Size	GB 1

Table A4. Hardware specifications.

Component	Specification
CPU	Intel Core i5 Processor
RAM	8 GB
Storage	SSD 256 GB
Operating System	Windows 11 Pro 64-bit

Note: The experimental environment was deployed on a standard workstation equipped with SSD storage and sufficient memory resources to support PostgreSQL indexing and query performance evaluation across large datasets.

Appendix C.1. Experimental procedure

1. Datasets were generated using PostgreSQL SQL scripts.
2. B-Tree and Hash indexes were created independently.
3. Benchmark queries were executed multiple times.
4. Average response times were recorded and analyzed.
5. Execution plans were verified using EXPLAIN ANALYZE.

Appendix C.2. Notes on experimental accuracy

1. Cache warming procedures were controlled before query execution.
2. Queries were repeated multiple times to minimize variance.
3. Performance statistics were averaged across benchmark runs.