An Efficient Primary Indexing Method with Sibling Pointers for Large-Scale Database Systems

Mohammad Al Khaldy
Department of Business Intelligence and Data Analytics, University of Petra, Amman, Jordan
mohammad.alkhaldy@uop.edu.jo (corresponding author)

Ameen Shaheen
Department of Software Engineering, Al-Zaytoonah University, Amman, Jordan
a.shaheen@zuj.edu.jo

Wael Alzyadat
Department of Software Engineering, Al-Zaytoonah University, Amman, Jordan
wael.alzyadat@zuj.edu.jo

Aysh Alhroob
Department of Software Engineering, Al-Zaytoonah University, Amman, Jordan
aysh@zuj.edu.jo

Received: 18 April 2025 | Revised: 21 May 2025 | Accepted: 25 May 2025

 DOI: https://doi.org/10.48084/etasr.11575

ABSTRACT

Efficient data retrieval is critical in modern database systems where data volumes are increasing. Although traditional indexing with B+ tree and bitmap indexes optimizes query performance, it introduces storage overhead, has inefficient mechanisms for handling duplicate keys, and proves challenging to scale. This study presents a new primary index approach that minimizes query execution time and eliminates extraneous lookups by adding sibling pointers, enabling efficient management and retrieval of duplicate keys. Based on a thorough experimental study utilizing a MySQL database of up to 10 million records, the proposed approach achieved significantly faster query execution times (up to 33.5%) and reduced storage overhead (up to 25%) compared to classical techniques. The proposed method provides stable effectiveness regardless of query type and improves scalability over large databases, advancing the field of indexing techniques by providing a low-cost, scalable, and storage-friendly solution applicable to high-traffic workloads and very large datasets. Future directions include its use in distributed and cloud-based environments, with opportunities for improvement through adaptive and AI-driven indexing approaches.

Keywords-database optimization; B+ tree; storage efficiency; large-scale data systems

I. INTRODUCTION AND BACKGROUND

As modern applications generate data on an unprecedented scale, there is a growing need for effective indexing techniques that can deliver high query performance without compromising scalability. In various fields, including financial systems, healthcare, e-commerce, and more, databases have become the basis for rapid access to accurate data to improve operational efficiency [1]. These systems store massive amounts of data with exponential growth in complexity, making traditional sequential search techniques inefficient for managing large-scale datasets [2]. This drives research on the design of advanced indexing techniques to improve query performance, response time, and storage cost [3].

Indexing techniques are one of the key aspects of how data are structured and stored to allow fast and efficient retrieval [4]. Traditional methods such as B+ trees and bitmap indexes have long been used to improve database performance due to their ordered hierarchical structure and range query efficiency [5]. However, they introduce additional storage and maintenance overhead, especially in systems with frequent inserts, deletes, and updates [6]. Bitmap indexes enable fast filtering and bitwise operations to accelerate query execution; however, although they can be efficient in read-heavy applications, they suffer from poor update performance and are not suitable for high transaction-rate environments due to the frequent need for reorganization [7].

As databases scale in size and complexity, the limitations of classic indexing approaches become more evident [8]. The demand for real-time data retrieval in various domains, including financial trading systems, machine learning models, and Internet of Things (IoT) applications, has led to the emergence of adaptive and hybrid indexing methods [9]. Modern database systems must handle all types of workloads where read speed, storage management, and continuously changing datasets present a balancing challenge. These issues have prompted researchers to explore hybrid indexing approaches, parallel and distributed indexing methods, and machine learning-based indexing models that adaptively adjust to workload patterns to improve query performance and reduce computational overhead [10]. With the rapid development of cloud computing and distributed databases, ındexing is facing new challenges, such as data replication, synchronization, and multi-node query processing. Cloud databases need scalable indexing methods to better handle low latency and high availability [11]. To improve performance in large-scale storage systems, several key technologies have been developed, such as Distributed Hash Tables (DHTs) and sharded indexing, to eliminate the delay caused by querying cloud environments. [12].

In response to these challenges, this study presents an efficient primary indexing method to increase the execution speed of queries and reduce redundant lookups, as well as optimize indexing structures to handle large-scale databases. The proposed approach ensures a balance of speed, adaptability, and storage efficiency, making it particularly suitable for next-generation systems operating in high-throughput and large-data scenarios. This work builds on existing indexing methods by incorporating insights from traditional indexing models, hybrid approaches, and distributed indexing techniques to deliver improved query efficiency with lower storage overhead.

In line with the growing demand for efficient and scalable indexing mechanisms, several studies have explored traditional, hybrid, and intelligent indexing techniques. These methods establish the foundation upon which the proposed solution is built. A variety of indexing strategies have been developed in response to the challenges of large-scale databases [13], ranging from classical ordered structures such as B+ trees [14] to more recent hybrid and machine learning-based approaches [15]. These solutions have been crucial to speeding up query execution time and reducing storage costs. Each has its limitations, which require even more engineering work to rectify. B+ trees are the most popular structure-optimized index used in relational databases due to their hierarchical structure and balanced search properties. These trees provide the benefit of all leaf nodes being at the same distance from the root node, thus keeping a query running time balanced at different levels [16]. However, carrying this structure comes with storage overhead and higher complexity in frequent updates, especially in high-transaction scenarios [17]. For categorical attributes, bitmap indexes offer an alternative, allowing fast bitwise operations to quickly prune the search space. Although bitmap indexes excel for analytical workloads, they are inefficient in environments with high update rates due to their expensive recomputation upon insertion or deletion of tuples [18].

Several studies have proposed improvements and optimizations to traditional indexing structures. In [19], B+ trees and dynamic hashing techniques were analyzed in a comparative study, focusing on how these structures have trade-offs between query execution speed, storage space, and updates. Although hash-based approaches are very effective for point queries, they are not optimal for range-based queries, as there is no ordering mechanism built in. In [20], cache-optimized indexing techniques were developed for RAM-based database systems to minimize memory access latency and optimize query performance when processing entirely in memory [21]. Hybrid indexing approaches have been proposed to overcome the limitations of classic techniques [22], combining the advantages of different indexing methods. In [23], a combined index was proposed by mixing B+ trees with hashing techniques so that databases can dynamically switch between different indexing mechanisms according to query workloads. Although hybrid indexing reduces query latency and increases adaptability, it adds maintenance complexity and increases computation costs in write-heavy use cases.

Parallel and distributed indexing approaches have been explored to achieve more scalable index maintenance solutions on cloud and big data scales. In [24], a distributed multidimensional data index strategy was designed based on cloud computing environments to improve query performance on multiple nodes. This study showed the potential of such distributed indexing schemes to improve query response time and fault tolerance in large systems. However, issues such as efficient partitioning and load balancing remain important challenges in their implementations. More recently, new techniques have involved machine learning at the heart of the indexing system to design index structures dynamically. In [25], the idea of learned index structures was introduced, which substitute conventional indexing methods with neural network models trained to predict the location of data. This method showed a considerable performance gain in terms of queries for read-heavy jobs but came with some challenges, including high training costs, adaptability to evolving datasets, and model interpretability.

This study proposes a new primary indexing mechanism that is built on a modified B+ tree, removes duplicate fragmented heap pointers, and increases querying performance. To do so, where classical dense primary indexes determine copies by simply referring to each other's records or by sequential search, this approach introduces an extra pointer pointing to duplicates at a sibling level. This improvement retains dense primary indexing's similar query efficiency with a fraction of the storage overhead and update complexity. The proposed approach builds on traditional, hybrid, and distributed indexing strategies that use B+ tree structures to facilitate sibling navigation in linked-list style by leveraging their effectiveness in rendering query efficiency. This method provides rapid access to retrieve duplicate records while maintaining the benefits of structured indices, ensuring fast retrieval without compromising the advantages of structured indexing. Furthermore, by addressing the limitations of existing indexing methods, this study contributes to the development of efficient and scalable database indexing strategies capable of supporting modern high-performance database systems.

II. COMPARATIVE ANALYSIS

Indexing is an invaluable aspect that significantly affects various factors, such as the speed at which queries are executed and the space, portability, and scalability of DBMSs. This section compares several indexing methods and their performance on different dimensions.

A. Performance Comparison of Indexing Techniques

To systematically evaluate the strengths and weaknesses of different indexing methods, Table I presents a comparative analysis based on query execution speed, storage efficiency, adaptability, scalability, update overhead, and suitability of the use case.

TABLE I. COMPARATIVE EVALUATION OF INDEXING TECHNIQUES

Technique	Speed	Storage	Scalability	Cost
B+ Trees	High	Moderate	Moderate	High
Bitmap	Very High	Low	Low	Very High
Hash Indexes	High	High	Moderate	Low
Hybrid Indexes (B+ Tree + Hashing)	High	Moderate	Moderate-High	Moderate
Distributed Indexing (DHT, Partitioned Indexes)	High	High	Moderate-High	Moderate
Machine Learning-Based Indexing	Very High	High	High	High

B. Key Insights from the Comparison

Based on the comparative analysis in Table I, several key insights can be drawn regarding the efficiency and limitations of various indexing approaches:

- B+ trees remain the most widely used general-purpose indexing structure due to their balanced query execution efficiency. However, they suffer from high storage overhead and update inefficiencies, making them less suitable for high-transaction environments.
- Bitmap indexes offer fast query execution for categorical filtering but are impractical for databases requiring frequent updates, as bitmap recompilation is resource-intensive.
- Hash indexes excel in exact-match queries, making them ideal for key-value stores but unsuitable for range queries, limiting their versatility in analytical workloads.
- Hybrid indexing methods attempt to optimize different query types dynamically but introduce added complexity in maintenance and workload balancing.
- Distributed indexing approaches are crucial for cloud-based and big-data environments, enabling parallel query execution. However, they require additional synchronization mechanisms to maintain consistency across multiple nodes.
- Machine learning-based indexing presents a revolutionary approach to indexing by dynamically predicting data locations, significantly reducing search overhead. However, high training costs and model adaptability issues remain challenges in real-world implementations.

C. Limitations of Existing Techniques and the Need for a New Approach

The comparisons make it clear that no single indexing method is best for all database environment types. Although B+trees work well for general-purpose databases, they do not scale well to distributed scenarios. Both bitmap and hash indexes are super-specialized and do not have flexibility across different workloads. In particular, hybrid and distributed indexing techniques can achieve better scalability, but they incur additional computation and maintenance costs. There is only a rudimentary implementation of machine learning-based indexing at the moment, and often such solutions require heavy computational resources for training and adaptation. To address these challenges, this study introduces a novel primary indexing technique that aims to:

- Enhance query execution speed while minimizing redundant lookups.
- Reduce storage overhead by eliminating unnecessary index duplications.
- Ensure efficient duplicate key management by introducing a sibling pointer structure that optimizes retrieval speed.

By incorporating these improvements, the proposed indexing method seeks to bridge the gap between traditional and modern indexing techniques, offering a scalable, storage-efficient, and high-performance solution for large-scale databases.

III. PROPOSED SOLUTION

Traditional indexing methods, such as dense and sparse indexing, have been widely used to speed up search queries. However, these techniques come with different limitations, especially in terms of storage overhead and query performance. The B + tree is one of the most popular indexing structures, as it provides a balanced search and is well-suited for sequential access. Transactional databases use it to index data. However, since it does not optimize to deal with duplicate key values properly, it is inefficient. Existing primary indexing techniques that deal with duplicate keys typically follow one of the following two approaches:

1. Key Duplication: Each duplicate key entry is stored separately along with its corresponding pointers, leading to increased storage consumption.
2. Single Entry with Sequential Search: A single entry is stored in the index, pointing to the first occurrence of the key, requiring sequential scans to locate additional duplicate records. This significantly slows down query performance, particularly for large-scale datasets.

Figure 1 shows a traditional dense index, where every record has a corresponding index entry, allowing faster searches at the cost of increased storage requirements. Figure 2 shows a sparse index, which improves storage efficiency by reducing the number of index entries but requires sequential searches when locating non-indexed records.

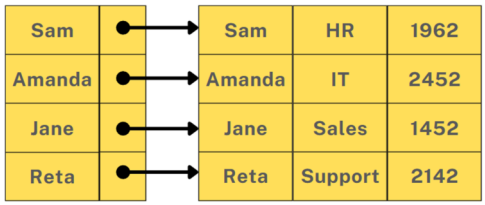

Fig. 1. Dense index.

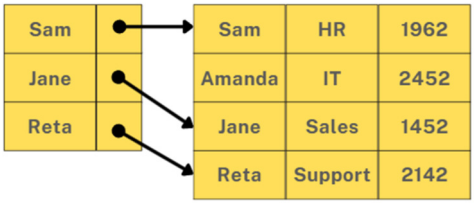

Fig. 2. Sparse index.

To address these inefficiencies, the proposed primary indexing technique uses pointers to duplicate records, called sibling pointers (s-pointers), to form a secondary linked list of duplicate records in the index, inspired by structures from B+ trees. This avoids duplicating keys, allowing direct traversal of duplicate entries, and significantly improving query execution time and storage. This method avoids the need to store multiple entries of the same key, which can also necessitate sequential searches, by linking duplicate records via sibling pointers to facilitate more efficient lookups. The proposed method can still leverage the fast retrieval performance of dense indexing while alleviating redundancy in the index structure. Figure 3 shows a B+ tree structure, which is the basis of the indexing solution that provides an efficient search with balanced trees and ordered data.

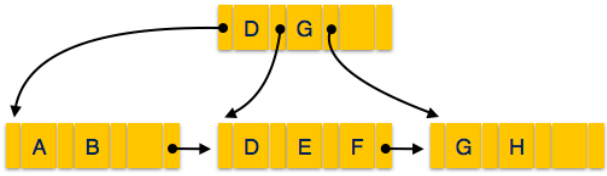

Fig. 3. B+ tree structure.

A. *Proposed Indexing Approach*

The proposed indexing approach introduces modifications to the DBMS indexing process, incorporating new functions to handle index setup, duplicate management, and optimized query execution.

The proposed indexing approach provides additional functions to set up the index, manage duplicates, and execute queries efficiently. In traditional dense indexing, as shown in Figure 4, if there are duplicate keys, it suffers from either redundant index entries or the need for a sequential search to find all occurrences of a key. Figure 5 shows the proposed sibling pointer mechanism, where duplicate records are linked together, reducing storage redundancy and improving query efficiency.

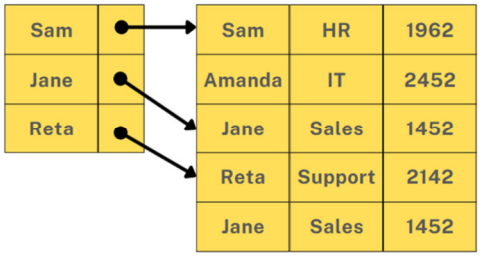

Fig. 4. Dense indexing with a duplicate key.

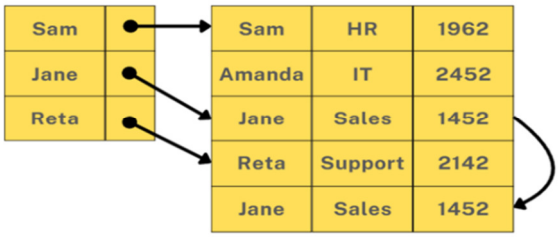

Fig. 5. Proposed indexing solution with sibling pointers.

B. *Insertion Mechanism*

During record insertion, the system follows these steps:

1. Check for Key Existence: When inserting a new record, the index first verifies whether the key already exists in the structure.
2. Handling Unique Keys: If the key is unique, the record is inserted normally into the index.
3. Handling Duplicate Keys: If the key already exists, the system links the new record to the existing one using a sibling pointer, avoiding duplication of index entries.

This mechanism ensures that duplicate records are connected in an optimal structure, eliminating redundant searches. The following pseudocode describes the insert process.

```
FUNCTION InsertRecord(record):
  key_exists = FindKey(record.key)
  IF key_exists:
    AttachSiblingPointer(key_exists,
      record)
  ELSE:
    InsertNewKey(record)
```

C. *Query Execution Process*

The query execution mechanism is optimized to leverage sibling pointers for efficient duplicate retrieval. The steps in query execution are:

1. Locate the first occurrence of the search key.
2. Retrieve all records connected via sibling pointers.
3. Return the full set of matching results efficiently, avoiding sequential scans.

The pseudocode for query execution is as follows:

```
FUNCTION RetrieveRecords(search_key):
  result = []
  node= FindKey(search_key)//first record
  WHILE node IS NOT NULL:
   result.append(node.record)
   node=node.sibling_pointer
   //linked duplicates
  RETURN result
```

D. Deletion Mechanism

As shown in the following algorithm, the deletion process is also optimized to handle sibling-linked records efficiently. The mechanism follows three possible cases:

- Case 1: Deleting the First Record in a Sibling Chain: The system checks if a sibling pointer exists and updates the pointer reference to the next record, making it the new index entry.
- Case 2: Deleting a Middle Record: The previous sibling pointer is updated to the next record.
- Case 3: Deleting the Last Record The previous sibling pointer is removed, and the record is deleted, ensuring a seamless update of the structure.

```
FUNCTION DeleteRecord(record):
  IF record is First in Chain:
    Update
    IndexPointer(record.sibling_pointer)
  ELSE IF record is Middle:
    UpdatePreviousSibling(
      record.sibling_pointer)
  ELSE:
    RemoveLastSiblingPointer(record)
  DELETE record
```

E. Performance Improvements Over Traditional Indexing

Compared to existing primary indexing techniques, the proposed method demonstrates significant improvements in:

- Query Execution Speed: Direct sibling traversal eliminates redundant index lookups.
- Storage Efficiency: Reduces memory footprint by eliminating duplicate key entries.
- Update Efficiency: Minimizes reorganization overhead during record insertions and deletions.
- Scalability: Ensures consistent performance across high-volume databases.

By implementing a sibling pointer-based primary index, this solution optimizes query execution time by eliminating unnecessary sequential searches. Additionally, storage efficiency is improved by reducing redundant index entries, making it suitable for high-transaction databases with frequent insertions, deletions, and updates.

IV. EXPERIMENTAL RESULTS

To evaluate the proposed indexing solution, a synthetic dataset was created that was similar to real-world transaction records. The dataset was developed in a MySQL 8.0 database with 100,000, 1 million, and up to 10 million records. The synthetic record included an integer primary key (id), categorical attributes (category), a numerical value (amount), and a timestamp (created at). Duplicate key values were intentionally included to stress-test the sibling pointer mechanism. The synthetic dataset allowed for a controlled and scalable way to evaluate query performance, storage overhead, and scalability.

For comparisons, B+ tree and bitmap indexing were used in MySQL 8.0. B+ Trees were tested using InnoDB's clustered index on the primary key, which is the default indexing method in MySQL for transactional workloads. Bitmap indexing was simulated using indexed ENUM/categorical fields and bitwise filtering queries to simulate bitmap behavior, as MySQL does not provide bitmap indexes. All indexes were created on the same schema fields, for instance, id and category, and the queries were run on the same dataset and hardware conditions. The experimental evaluation was performed on a system with an Intel Core i7-11700, 16 GB RAM, and 512 GB SSD. Query caching was performed using MySQL 8.0.31 Community Edition. All queries were executed through the MySQL command line. The aim was to quantify the performance of query executions, storage overhead, and scalability compared to traditional indexing methods such as B+ trees and bitmap indexes. Three primary query types were tested for evaluation, which are commonly used in database applications:

- Range Queries: Queries that retrieve multiple records within a specified range of values (e.g., SELECT * FROM table WHERE value BETWEEN X AND Y;).
- Exact Match Queries: Queries that search for a specific value (e.g., SELECT * FROM table WHERE id = constant;).
- Aggregate Queries: Queries that compute functions such as COUNT, SUM, or AVERAGE over a dataset (e.g., SELECT COUNT(*) FROM table WHERE category = 'A';).

A. Results and Analysis

The experimental results show that the proposed indexing technique outperformed traditional indexing methods in terms of query execution time and storage efficiency. The first set of results compares the average query execution time for B+ tree indexing, bitmap indexing, and the proposed method. As shown in Figure 6, the proposed indexing method demonstrated a 26% improvement in query execution time over B+ trees and a 15.7% improvement over bitmap indexing.

The proposed method achieved lower storage requirements due to optimized pointer management and reduced index redundancy. The lower storage cost also resulted in better cache utilization and faster retrieval times, making it an optimal choice for high-transaction databases. The proposed method reduced storage overhead by 10% compared to B+ trees and 25% compared to bitmap indexes, as shown in Figure 7.

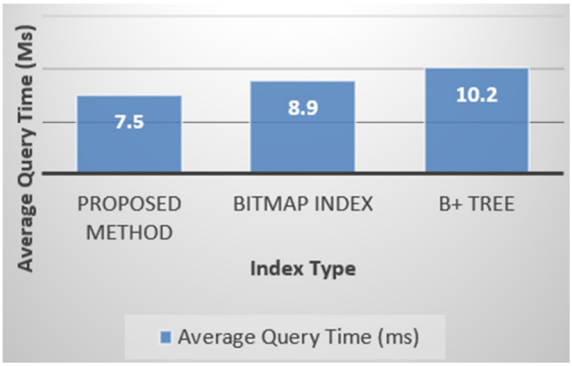

Fig. 6. Comparison of average query time across indexing techniques.

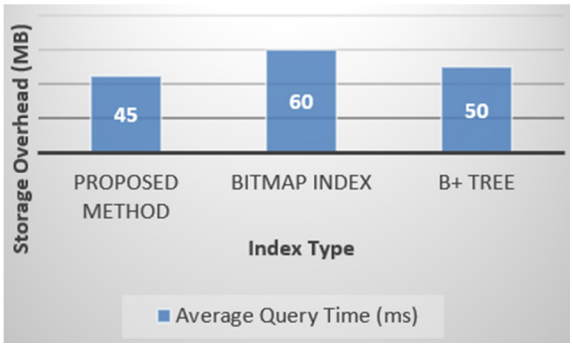

Fig. 7. Comparison of storage overhead across indexing techniques.

To analyze how indexing methods scale with increasing dataset sizes, query execution time was measured at different data volumes (100K, 1M, and 10M records). As shown in Figure 8, with the growth of dataset size, the query execution time of the proposed method remained more stable compared to B+ trees and bitmap indexes. Although B+ trees and bitmap index execution times rise sharply as the dataset grows to 10M records, the proposed approach exhibited a more desirable scaling characteristic, with a 33.5% reduction in query latency over B+ trees, and a 29.7% reduction over bitmap indexes. To further assess how different indexing methods handle different query types, execution times were analyzed for range queries, exact match queries, and aggregate queries.

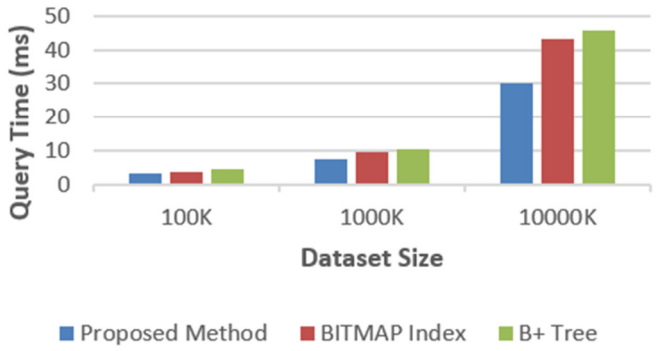

Fig. 8. Query performance as dataset size increased.

TABLE II. EXECUTION TIME COMPARISON BY QUERY TYPE

Query type	B+Tree	Bitmap Index	Proposed Method	Gain vs B+ tree (%)	Gain vs Bitmap (%)
Range query	11.8 ms	10.3 ms	8.2 ms	30.5%	20.4%
Exact match	7.4 ms	6.6 ms	5.6 ms	24.3%	15.2%
Aggregate query	9.9 ms	9.1 ms	6.8 ms	31.3%	25.3%

As shown in Table II, the proposed indexing method achieved a significant performance improvement compared to B+ tree and bitmap indexes, outperforming the other methods for all major query types, achieving a 31.3% reduction in query time over B + trees and 25.3% over bitmap indexing in aggregate queries, which was the greatest performance gain. This improvement is primarily due to the use of a sibling pointer mechanism that reduces redundant index lookups when reading grouped values.

Depending on the types of queries used, the experimental results show that the proposed approach was 30.5% faster than B+ trees and 20.4% faster than bitmap indexes for processing a range of queries. The sibling pointer proves useful in cases where sequential records are read in large blocks, making traversal more efficient. On exact match queries, where bitmap indexes usually perform very well, the proposed method still performed better with a 15.2% gain. This shows the flexibility and robustness of the proposed approach in transactional and analytical workloads. Overall, the results validate that sibling pointer-enhanced indexing not only improves average query time but, unlike traditional indexing structures, provides consistent benefits across a wide series of query patterns.

V. CONCLUSION

This study introduced a novel primary indexing technique that enhances query execution efficiency while reducing storage overhead. By incorporating sibling pointers, the proposed approach optimizes duplicate key management, minimizing redundant lookups and improving retrieval performance. Experimental results conducted on a controlled MySQL database environment demonstrated that the proposed indexing method outperformed traditional techniques such as B+ trees and bitmap indexes across various query types. These results were obtained by direct implementation and evaluation of all techniques under identical conditions to ensure a fair comparison. The results showed that the proposed method achieved up to 33.5% faster query execution and 25% lower storage overhead compared to B+ trees and bitmap indexes. Additionally, the method maintained consistent performance across different query types and scaled efficiently with increasing dataset sizes. The findings highlight the practical benefits of the proposed indexing approach for large-scale and high-transaction database environments. Future research could explore its integration into distributed and cloud-based systems, as well as potential enhancements through adaptive indexing and machine learning-based optimizations. In conclusion, the proposed technique provides an efficient, scalable, and high-performance indexing solution that offers improvements in query speed, storage efficiency, and database scalability for modern data management applications.

REFERENCES

[1] A. Raman, K. Karatsenidis, S. Xie, M. Olma, S. Sarkar, and M. Athanassoulis, "QuIT your B+-tree for the Quick Insertion Tree." OpenProceedings.org, 2025, https://doi.org/10.48786/EDBT.2025.36.

[2] S. R. Jeong, Y. Kim, I. Ghani, and J. H. Kim, "A New Database Archiving Approach for Effective Storage and Data Management: A Case Study of Data Warehouse Project in a Korean Bank," *International Journal of Advance Soft Computing Application*, vol. 6, no. 3, pp. 31–46, 2014.

[3] S. Emanuilov and A. Dimov, "Billion-Scale Similarity Search Using a Hybrid Indexing Approach with Advanced Filtering," *Cybernetics and Information Technologies*, vol. 24, no. 4, pp. 45–58, Dec. 2024, https://doi.org/10.2478/cait-2024-0035.

[4] J. S. Hwang, S. Lee, Y. Lee, and S. Park, "A Selection Method of Database System in Bigdata Environment: A Case Study From Smart Education Service in Korea," *International Journal of Advance Soft Computing Application*, vol. 7, no. 1, pp. 9–21, 2015.

[5] S. H. Adil, M. Ebrahim, S. S. A. Ali, and K. Raza, "Performance Analysis of Duplicate Record Detection Techniques," *Engineering, Technology & Applied Science Research*, vol. 9, no. 5, pp. 4755–4758, Oct. 2019, https://doi.org/10.48084/etasr.3036.

[6] M. Al-Ani, Q. Al-Shayea, A. R. Alshehadeh, B. Annisa, and H. A. Al-khawaja, "Creating Visual Knowledge Representation Based on Data Mining in Educational Jordanian Databases," *International Journal of Advance Soft Computing Application*, vol. 16, no. 1, pp. 155–168, 2024.

[7] J. Wang and M. Athanassoulis, "CUBIT: Concurrent Updatable Bitmap Indexing," *Proceedings of the VLDB Endowment*, vol. 18, no. 2, pp. 399–412, Oct. 2024, https://doi.org/10.14778/3705829.3705854.

[8] T. Taipalus, "The effects of database complexity on SQL query formulation," *Journal of Systems and Software*, vol. 165, Jul. 2020, Art. no. 110576, https://doi.org/10.1016/j.jss.2020.110576.

[9] S. AlZu'bi *et al.*, "Diabetes Monitoring System in Smart Health Cities Based on Big Data Intelligence," *Future Internet*, vol. 15, no. 2, Feb. 2023, Art. no. 85, https://doi.org/10.3390/fi15020085.

[10] M. Aumüller, E. Bernhardsson, and A. Faithfull, "ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms," in *Similarity Search and Applications*, vol. 10609, C. Beecks, F. Borutta, P. Kröger, and T. Seidl, Eds. Springer International Publishing, 2017, pp. 34–49.

[11] J. C. Corbett *et al.*, "Spanner: Google's Globally Distributed Database," *ACM Transactions on Computer Systems*, vol. 31, no. 3, pp. 1–22, Aug. 2013, https://doi.org/10.1145/2491245.

[12] G. DeCandia *et al.*, "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, Oct. 2007, pp. 205–220, https://doi.org/10.1145/1294261.1294281.

[13] C. McMillen, *Advancements in Database Management Systems: A Comprehensive Review and Future Directions*. 2024.

[14] H. V. Jagadish, B. C. Ooi, K. L. Tan, C. Yu, and R. Zhang, "iDistance: An adaptive B^+ -tree based indexing method for nearest neighbor search," *ACM Transactions on Database Systems*, vol. 30, no. 2, pp. 364–397, Jun. 2005, https://doi.org/10.1145/1071610.1071612.

[15] D. Deutch, N. Frost, A. Gilad, and T. Haimovich, "Explaining Missing Query Results in Natural Language." OpenProceedings.org, 2020, https://doi.org/10.5441/002/EDBT.2020.49.

[16] C. Zhong *et al.*, "IndeXY: A Framework for Constructing Indexes Larger than Memory," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, Utrecht, Netherlands, May 2024, pp. 516–529, https://doi.org/10.1109/ICDE60146.2024.00046.

[17] P. O'Neil and D. Quass, "Improved query performance with variant indexes," in *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, Mar. 1997, pp. 38–49, https://doi.org/10.1145/253260.253268.

[18] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Transactions on Database Systems*, vol. 31, no. 1, pp. 1–38, Mar. 2006, https://doi.org/10.1145/1132863.1132864.

[19] J. Rao and K. A. Ross, "Making B+- trees cache conscious in main memory," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, Dallas, TX, USA, May 2000, pp. 475–486, https://doi.org/10.1145/342009.335449.

[20] J. Ma and J. V. Nickerson, "Hands-on, simulated, and remote laboratories: A comparative literature review," *ACM Computing Surveys*, vol. 38, no. 3, Sep. 2006, Art. no. 7, https://doi.org/10.1145/1132960.1132961.

[21] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking Database Algorithms for Phase Change Memory," presented at the 5th Biennial Conference on Innovative Data Systems Research (CIDR '11), Jan. 2011.

[22] J. Dittrich, J. Nix, and C. Schön, "The next 50 years in database indexing or: the case for automatically generated index structures," *Proceedings of the VLDB Endowment*, vol. 15, no. 3, pp. 527–540, Nov. 2021, https://doi.org/10.14778/3494124.3494136.

[23] S. Nakazono, Y. Bessho, H. Kawashima, and T. Nakamori, "Griffin: Fast Transactional Database Index with Hash and B+-Tree." arXiv, Jul. 18, 2024, https://doi.org/10.48550/arXiv.2407.13294.

[24] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The Case for Learned Index Structures," in *Proceedings of the 2018 International Conference on Management of Data*, Feb. 2018, pp. 489–504, https://doi.org/10.1145/3183713.3196909.

[25] D. Gui and G. He, "Distributed Multi-Dimensional Data Index Strategy in Cloud Computing Environment," in *2021 5th International Conference on Electronics, Communication and Aerospace Technology (ICECA)*, Coimbatore, India, Dec. 2021, https://doi.org/10.1109/ICECA52323.2021.9675845.