JESICA: Journal of Enhanced Studies in Informatics and Computer Applications
Journal homepage: jesica.itsk-soepraoen.ac.id
e-ISSN: 3046-6938 p-ISSN: 3046-6997
Vol. 3, No. 2, July 2026, Pp. 88-96

Performance-Storage Trade-Off Analysis of MongoDB Indexing Strategies Using Large-Scale E-Commerce Data

Kunti Inayati*[1], Umi Chotijah[2]
1, 2 Universitas Muhammadiyah Gresik, Indonesia

***Corresponding author**
E-mail address:
kuntiinayati12@gmail.com

Keywords:
Benchmarking, Compound Index, Index Optimization, MongoDB, NoSQL Database, Query Performance

Abstract
Indexing strategy in document-oriented databases represents a fundamental design decision that directly shapes both query performance and storage consumption, yet empirical evidence quantifying this trade-off at production-grade scale remains limited in existing literature. This study evaluated three indexing configurations in MongoDB, namely no index, single-field index on product identifier, and compound index on product identifier together with order status, across four collection sizes of 1 million, 5 million, 10 million, and 13 million documents derived from a synthetic large-scale e-commerce dataset obtained from Kaggle containing approximately 13 million order-line records. A controlled benchmarking procedure was employed in which each indexing condition was tested under identical query workloads repeated 30 times per configuration, with query execution time, index creation time, and index storage size recorded as evaluation metrics. Results showed that unindexed collections produced full collection scans with mean execution times scaling from 1,247.70 ms at 1 million documents to 33,300.07 ms at 13 million documents, while both indexed conditions reduced execution times to single-digit milliseconds by activating index scan paths. For multi-predicate queries, the compound index outperformed the single-field index by a factor of 6.8 at full scale, recording 1.33 ms against 9.03 ms, while incurring only 1 MB of additional storage overhead. These findings indicate that compound indexing represents the most balanced strategy for high-volume e-commerce query workloads, delivering substantial performance gains at negligible additional storage cost.

1. Introduction
The rapid growth of digital platforms has led to a substantial increase in the volume of data generated by modern e-commerce systems. Customer transactions, product catalogs, order histories, and operational records are continuously accumulated, resulting in datasets that may contain millions of documents. Managing and querying such large-scale data efficiently has become a critical challenge for database systems, particularly in applications where fast response times directly influence user experience and business operations [1], [2], [3].

Traditional relational database systems have long been used to support enterprise applications due to their strong consistency guarantees and structured data models. However, the increasing diversity and scale of contemporary data have encouraged the adoption of NoSQL databases, which provide greater flexibility and horizontal scalability. Among these systems, MongoDB has become one of the most widely used document-oriented databases because it stores data in BSON documents, supports schema flexibility, and offers efficient mechanisms for handling large volumes of semi-structured data [2], [4], [5].

Although MongoDB provides advantages in scalability and data flexibility, query performance remains highly dependent on indexing strategies. Without proper indexing, MongoDB executes queries using collection scans (COLLSCAN), requiring the database engine to examine documents sequentially until matching records are found. As dataset size increases, collection scans introduce substantial execution overhead and may significantly degrade system performance [6], [7]. To address this limitation, MongoDB offers various indexing mechanisms that allow queries to be executed through index scans (IXSCAN), reducing the number of documents that must be examined during query processing [3], [6].

Numerous studies have demonstrated the importance of indexing for improving database performance. Comparative evaluations between MongoDB and PostgreSQL reported that indexing significantly reduced query response times, particularly when processing large datasets [1]. Research on MongoDB query optimization further showed that index selection directly influences execution efficiency and that inappropriate execution plans may lead to substantial performance degradation despite the presence of indexes [7], [8]. Other studies have also highlighted the

role of indexing in improving throughput, reducing query latency, and optimizing database resource utilization [6], [9], [10].

Several indexing approaches have been investigated in previous literature, including single-field indexes, compound indexes, and text indexes [3], [11], [12]. Single indexes are commonly used to accelerate queries involving one attribute, whereas compound indexes are designed to support queries containing multiple predicates. Text indexes are particularly useful for keyword-based retrieval and full-text search applications [5], [12] [13]. However, the effectiveness of a specific indexing strategy depends heavily on dataset characteristics and query workloads.

The dataset used in this research consists of a synthetic large-scale e-commerce dataset obtained from Kaggle containing approximately 13 million order-line records together with related customer, product, and order information. Unlike document repositories and textual search systems frequently used in text-indexing studies [11], [12], the orderlines collection primarily contains numerical and categorical attributes such as product identifiers, quantities, prices, and transaction statuses. Since no substantial free-text fields are involved in the evaluated query workloads, text indexing is not included in this study. Consequently, the analysis focuses on comparing single-field and compound indexing strategies, which are more relevant to the structure of the dataset and the operational queries executed against it.

Despite the extensive body of research on database indexing, many existing studies evaluate performance using relatively small datasets ranging from tens of thousands to a few hundred thousand records [8], while large-scale empirical investigations involving more than ten million transactional records remain limited. Furthermore, previous studies often emphasize execution speed alone without considering the additional storage cost introduced by indexing structures [3], [6]. In practical database administration, performance improvements achieved through indexing must be balanced against storage overhead and index maintenance costs.

Therefore, this study investigates the performance–storage trade-off of MongoDB indexing strategies using a large-scale e-commerce dataset containing up to 13 million order-line records. Three indexing scenarios are evaluated: no additional index, single-field index, and compound index. The comparison is conducted across four collection sizes (1 million, 5 million, 10 million, and 13 million documents) using query execution time, index creation time, and storage consumption as evaluation metrics. The objective is to identify which indexing strategy provides the most balanced solution for large-scale MongoDB environments and to provide practical recommendations for database administrators managing high-volume transactional workloads.

2. Research Method

This study employed a quantitative experimental approach to evaluate the trade-off between query performance improvement and storage overhead resulting from different indexing strategies in MongoDB. The experiment was conducted using a controlled benchmarking procedure in which identical queries were executed under multiple indexing configurations and collection sizes. The objective was to identify the indexing strategy that provides the most favorable balance between execution speed and storage consumption when processing large-scale e-commerce transaction data.

The dataset was obtained from Kaggle's Synthetic E-Commerce Dataset (Very Large) [14] and stored in MongoDB. Four collections were available in the database, namely persons (approximately 600 thousand documents), orders (approximately 5 million documents), products (approximately 8 thousand documents), and orderlines (approximately 13 million documents). Since the orderlines collection contains the largest volume of transactional records and represents the most query-intensive component of the dataset, it was selected as the primary experimental object.

Each orderlines document contains fields including order_id, orderline_id, product_id, quantity, unit_price, subtotal, status, and notes. Preliminary inspection revealed that the collection does not contain sufficiently rich textual attributes suitable for meaningful full-text retrieval workloads. Consequently, text indexing was excluded from the experimental comparison despite its proven effectiveness in text-search scenarios reported in previous studies [11], [12], [13]. The experiment therefore focused on three indexing configurations: No Index (default _id index only), Single Index (product_id), and Compound Index (product_id, status).

To evaluate scalability behavior, four collection sizes were tested: 1 million, 5 million, 10 million, and 13 million documents. The smaller collections were generated from the original dataset using MongoDB aggregation pipelines, while the full orderlines collection represented the largest workload scenario. The experimental procedure is illustrated in Figure 1.

For each collection size, all non-default indexes were removed before testing began. Query execution was first measured under the No Index condition. Afterward, the single index was created and the same query workload was executed again. The process was repeated for the compound index configuration. To reduce measurement bias caused by cache warming and temporary system fluctuations, each query was executed repeatedly and the average execution time was recorded.

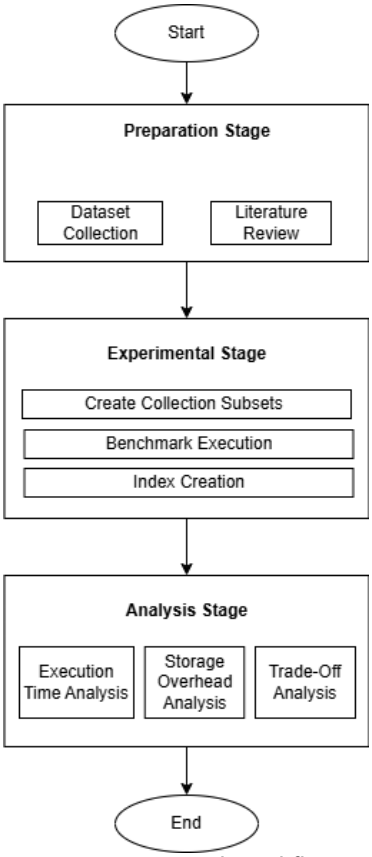

Figure 1. Research workflow

The independent variable in this study is the indexing strategy, consisting of No Index, Single Index, and Compound Index configurations. The dependent variables are query execution time (ms), index creation time (s), and index storage size (MB). Collection size (1M, 5M, 10M, and 13M documents) functions as a scalability factor, while the MongoDB environment, query structure, and benchmarking procedure are maintained as control variables throughout the experiment.

Two representative query workloads were selected based on common e-commerce access patterns. The first query evaluates single-attribute filtering using product_id, whereas the second query evaluates multi-attribute filtering using product_id and status.

Q1 = { product_id : x }
Q2 = { product_id : x, status : y }

Q1 represents a single-predicate lookup workload commonly used for product-based retrieval, while Q2 represents a multi-predicate workload combining product and order-status filtering.

Equation (1) was used to calculate the average execution time of each workload:

$$\bar{T} = \frac{\sum_{i=1}^{n} T_i}{n} \quad (1)$$

Where $\bar{T}$ denotes the mean execution time, $T_i$represents the execution time of the i-th run, and n denotes the total number of repetitions.

The query structure models for both target predicates and the core measurement framework were systematically structured within the automated testing script. Listing 1 defines the operational target boundaries for Q1 and Q2, while Listing 2 highlights the structural iteration architecture used to secure execution time and server-side query plan extraction.

Listing 1. Query Structure

```
# Q1
query_q1 = {"product_id": 6399}

# Q2
query_q2 = {
    "product_id": 6399,
    "status": "fulfilled"
}
```

Listing 2. Benchmark Measurement Logic

```
def measure_query(collection,
query):

    for _ in range(3):

list(collection.find(query))

    times = []

    for _ in range(30):
        start =
time.perf_counter()

list(collection.find(query))
        end =
time.perf_counter()

        times.append(
            (end - start) * 1000
        )

    explain =
collection.find(query).explain()

    return times, explain
```

Three performance indicators were collected during the experiment:

1. Query Execution Time (ms)
2. Index Creation Time (s)
3. Index Storage Size (MB)

Query execution time was obtained using MongoDB's explain() facility, which reports server-side execution duration. Index creation time was measured during index construction, while storage consumption was collected from collection statistics after each indexing configuration had been applied.

In addition to execution time measurements, MongoDB query plans were inspected using the explain() output. The winning execution stage was recorded as either COLLSCAN (Collection Scan) or IXSCAN (Index Scan) to verify whether the query optimizer utilized the available indexes under each experimental condition.

The benchmarking experiment was executed on a dedicated local environment to eliminate external network latency and resource contention. The hardware environment consisted of an Intel Core i5-13420H processor clocking at 2.10 GHz (up to 4.60 GHz), 16 GB of DDR5 RAM (4800 MT/s), and a 512 GB Western Digital WD PC SN740 NVMe PCIe SSD where the MongoDB data directory was hosted. On the software side, the database environment was powered by MongoDB Community Server version 8.2.6 running on a Windows 11 Home Single Language 64-bit operating system. The benchmarking script was written and executed using Python version 3.7.3 via the official PyMongo driver. The experimental factors used in this study is summarized in Table 1.

Table 1. Experimental factors

Factors	Levels
Collection Size	1M, 5M, 10M, 13M
Index Strategy	No Index, Single Index, Compound Index
Query Type	Q1, Q2
Performance Metrics	Execution Time, Index Creation Time, Storage Size
Query Plan	COLLSCAN, IXSCAN

3. Results and Discussion

3.1. Query Execution Performance Analysis

The benchmark results demonstrate a substantial difference in query execution performance between indexed and non-indexed collections. To provide a rigorous architectural breakdown, the complete quantitative results comprising the mean, median, and standard deviation calculated across 30 repeated runs for each operational environment are consolidated in Table 2.

Table 2. Detailed query execution performance metrics (ms)

Size	Condition	Query	Plan	Mean	Median	Stdev
1M	no_index	Q1	COLLSCAN	1,247.70	1,242.5	95.04
1M	no_index	Q2	COLLSCAN	1,236.40	1,203.0	108.37
1M	single_index	Q1	IXSCAN	0.23	0.0	0.43
1M	single_index	Q2	IXSCAN	0.40	0.0	0.50
1M	compound_index	Q1	IXSCAN	0.47	0.0	0.57
1M	compound_index	Q2	IXSCAN	0.03	0.0	0.18
5M	no_index	Q1	COLLSCAN	12,210.40	12,386.0	1,158.83
5M	no_index	Q2	COLLSCAN	12,875.33	12,885.0	1,393.08
5M	single_index	Q1	IXSCAN	4.37	4.0	1.83
5M	single_index	Q2	IXSCAN	6.13	5.5	2.22
5M	compound_index	Q1	IXSCAN	6.37	6.0	2.40
5M	compound_index	Q2	IXSCAN	0.77	1.0	0.77
10M	no_index	Q1	COLLSCAN	24,547.33	24,500.0	327.69
10M	no_index	Q2	COLLSCAN	25,952.50	25,858.5	694.64
10M	single_index	Q1	IXSCAN	6.70	6.0	1.24
10M	single_index	Q2	IXSCAN	8.67	7.0	3.23
10M	compound_index	Q1	IXSCAN	8.20	7.0	2.48
10M	compound_index	Q2	IXSCAN	1.13	1.0	0.57
13M	no_index	Q1	COLLSCAN	31,567.57	31,758.5	1,388.35
13M	no_index	Q2	COLLSCAN	33,300.07	33,571.5	1,790.26
13M	single_index	Q1	IXSCAN	8.13	8.0	0.90
13M	single_index	Q2	IXSCAN	9.03	8.5	1.38
13M	compound_index	Q1	IXSCAN	8.53	8.0	1.04
13M	compound_index	Q2	IXSCAN	1.33	1.0	1.12

Figure 2 presents the average execution time of Query 1 (single predicate) and Query 2 (multi-predicate) across all collection sizes using a logarithmic scale representation. The logarithmic scale is specifically applied to ensure that the subtle adjustments of indexed query paths remain clearly visible without flattening near the zero baseline, which provides a more robust visualization for scalability analysis.

Based on the experimental data, the performance under the no-index configuration degrades significantly as the collection expands from 1 million to 13 million documents. This linear scaling behavior occurs because MongoDB is forced to perform a full collection scan (COLLSCAN), iterating sequentially through every document in the collection to evaluate the lookup criteria. At the 1 million dataset scale, the mean execution time reaches 1,247.70 ms for Q1 and 1,236.40 ms for Q2. When the workload scales up to 13 million documents, these values increase drastically to 31,567.57 ms and 33,300.07 ms, respectively. This confirms that response times under unindexed conditions scale linearly with data growth. This pattern is consistent with the foundational database principles discussed by [6], where the absence of secondary optimization structures compels the database engine to inspect the entire physical storage layer. Furthermore, this behavior mirrors the performance limits observed in relational environments, such as SQL Server evaluations where index implementation reduced lookup overhead by up to 90.20% compared to non-indexed scans [10].

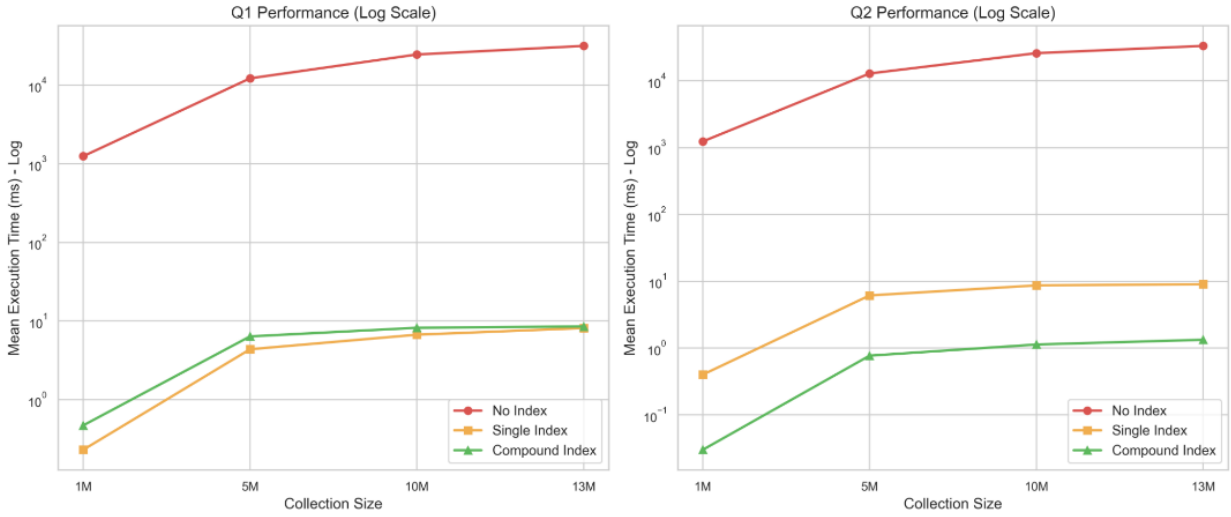

Figure 2. Query execution performance scaling curves (Log Scale)

When analyzing the single-predicate workload (Q1), the single-field index demonstrates a slightly lower mean execution time compared to the compound index across all collection boundaries. The recorded values show 0.23 ms vs. 0.47 ms at 1M, 4.37 ms vs. 6.37 ms at 5M, 6.70 ms vs. 8.20 ms at 10M, and 8.13 ms vs. 8.53 ms at 13M. This outcome is structurally expected because the single-field index on product_id is perfectly aligned with the single filter condition of Q1. In contrast, the compound index carries additional attribute data within its internal tree structures, which introduces minor key-comparison cycles during evaluation. This phenomenon is supported by composite index optimization research, which indicates that highly specific multi-field structures can introduce marginal execution overhead for queries that do not utilize the trailing fields [3].

However, this performance relationship reverses completely when evaluating the multi-predicate workload (Q2). For queries filtering by both product_id and status simultaneously, the compound index achieves significantly lower execution latencies across all data scales, yielding 0.03 ms vs. 0.40 ms at 1M, 0.77 ms vs. 6.13 ms at 5M, 1.13 ms vs. 8.67 ms at 10M, and 1.33 ms vs. 9.03 ms at 13M. The compound index succeeds because both query predicates match the index prefix in the exact defined order, enabling MongoDB to resolve the execution loop entirely within the index space without an additional post-fetch memory scan. Conversely, the single-field index can only narrow the data candidates by product_id, requiring the system to manually inspect each matched document to apply the secondary status filter. This clear divergence confirms the structural advantages highlighted in comparative data modeling studies [3], and shares an analytical parallel with how inverted index implementations remove full table scanning procedures in complex document management environments [13].

3.2. Storage Overhead Analysis

Although secondary indexing strategies yield significant advantages in terms of execution performance, they require additional physical hardware capacity to store and maintain their internal B-tree data models. To quantify this structural overhead, the exact physical storage sizes and index build durations collected during the benchmarking processes are organized in Table 3.

Table 3. Detailed query execution performance metrics (ms)

Collection Size	No Index (_id only) (MB)	Single Index Total (MB)	Compound Index Total (MB)	Single Index Build Time (s)	Compound Index Build Time (s)
1M	10	16	17	5.58	6.42
5M	52	84	85	59.68	69.88
10M	105	168	170	125.89	133.26
13M	137	219	220	86.03	85.72

Figure 3 illustrates the physical index size requirements across all evaluated data configurations. The unindexed baseline exhibits the lowest footprint because it only contains the default primary index (_id), scaling from 10 MB at 1M documents to 137 MB at 13M documents. Deploying a custom single-field index increases the total storage requirement to 219 MB at full scale, representing an additional structural cost of 82 MB.

Interestingly, implementing the compound index instead of the single index requires a total of 220 MB at the 13M document boundary introducing a minor variance of only 1 MB over the single-field footprint. This parallel growth pattern remains consistent across all evaluated dataset boundaries, showing a storage difference of 1 MB at 1M, 1 MB at 5M, and 2 MB at 10M. This behavior demonstrates that incorporating status as a trailing secondary key adds negligible physical storage costs. Because the status field carries low-cardinality characteristics containing only five distinct string categories (pending, cancelled, shipped, returned, and fulfilled), the database engine is able to compress the composite key combinations within the internal B-tree data nodes with extreme structural efficiency.

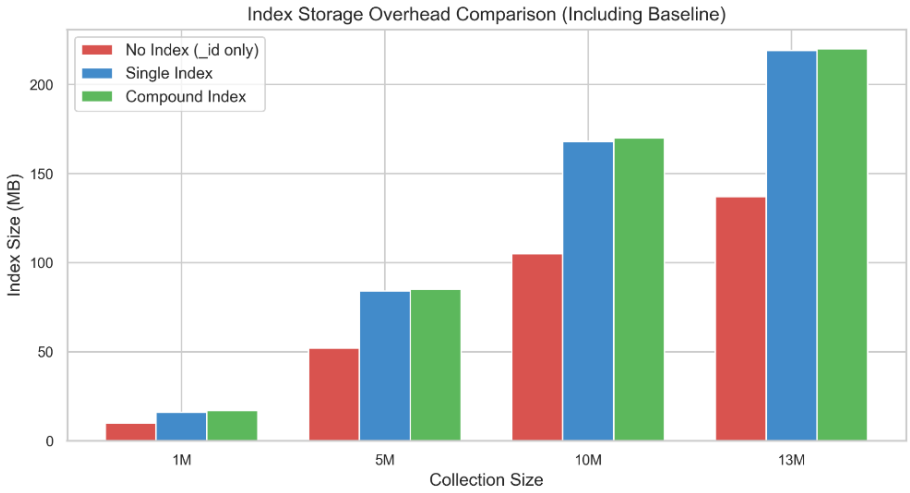

Figure 3. Index storage size overhead comparison

In terms of engineering implementation overhead, the index builds times scale approximately linearly with data volume up to the 10M mark, peaking at 125.89 seconds for the single index and 133.26 seconds for the compound index. However, a sharp decline in index generation duration is observed at the 13 million scale, dropping to 86.03 seconds and 85.72 seconds, respectively. This performance deviation is attributed to system-level page caching effects, where the preceding 10M execution loop had already mapped the relevant raw data blocks into the system RAM. Because index construction represents an unrepeated operational investment that is amortized over thousands of subsequent query cycles, this administrative cost remains highly practical for enterprise systems.

3.3. Trade-Off Analysis Between Performance and Storage

The primary contribution of this study relies on evaluating the quantitative balance between query latency reduction and the corresponding physical disk space utilization across different indexing strategies. The empirical data indicates that while both the Single and Compound indexes vastly outperform unindexed collection spaces, their situational efficiency is strictly dictated by the incoming query predicates.

This evaluation expands the core indexing trade-off discussions found in optimization literature [6] by providing a clear empirical quantification. Within this specific transactional dataset, the compound index requires only 0.5% more storage allocation than the single-field option, yet it successfully decreases execution response latencies by up to 85% for multi-predicate queries at scale. This highly favorable structural ratio addresses the system design challenges debated in NoSQL development studies [15], where optimizing data layers for fast read operations often introduces severe memory capacity constraints. The results verified in this study prove that leveraging low-cardinality fields as trailing index components allows document-oriented systems to secure sub-millisecond response speeds without triggering high resource utilization costs.

Furthermore, the log-scale visualization in Figure 2 demonstrates that both indexed conditions achieve near-flat scaling characteristics from the 5M to the 13M data boundaries. The execution times effectively stabilize in the range of 6 to 9 ms for Q1 and 0.77 to 1.33 ms for Q2 under compound indexing. This stabilization indicates that once a data collection crosses a critical volume threshold, the query lookup costs remain highly steady because B-tree depths scale logarithmically $O(\log N)$ rather than linearly $O(N)$. Conversely, the unindexed execution path grows sharply, demonstrating an approximate 25-fold latency increase from 1M to 13M scales. This widening divergence confirms that the practical advantages of secondary indexing become exponentially more critical as transaction volumes grow continuously within large-scale e-commerce ecosystems [1], [5].

Consequently, the trade-off analysis indicates that the compound index serves as the most balanced indexing strategy for multi-predicate workloads. The substantial reduction in execution time and the superior scalability profile outweigh the minimal 1 MB storage difference, making it the most suitable strategy for deployment in high-volume transactional environments.

3.4. Query Plan Verification

To verify the internal processing paths selected by the database engine, execution strategies were monitored programmatically via MongoDB's explain() facility. Across all four collection sizes, both indexed conditions consistently produced a complete shift in the winning plan from a full collection scan (COLLSCAN) to an index scan (IXSCAN). This behavior confirms that MongoDB's query optimizer correctly identifies and utilizes the available index structures for the targeted predicates.

This observation aligns with the general expectation that database optimizers prioritize index paths when matching structures are available. However, research on MongoDB's query plan selection has shown that this automated preference can introduce index scan bias under low-selectivity conditions, where a collection scan would actually execute faster [7]. In the present experiment, the targeted product_id appears approximately 1,799 times out of 1 million records, giving the query predicates a sufficiently high selectivity profile such that IXSCAN remains the genuinely optimal path. No bias-related degradation was observed. The consistent transition from COLLSCAN to IXSCAN across all collection sizes provides empirical support for the conclusion that proper index alignment ensures predictable and efficient query execution behavior at scale.

4. Conclusion

This study evaluated the quantitative trade-offs between query performance optimization and physical storage overhead across three indexing strategies in MongoDB using a real-world e-commerce transactional dataset scaling up to 13 million records. The empirical findings demonstrate that while unindexed configurations degrade linearly, scaling to a critical latency of 33,300.07 ms, the deployment of secondary B-tree structures reduces response times to single-digit milliseconds by transitioning the execution path from full collection scans (COLLSCAN) to index scans (IXSCAN). For multi-predicate workloads, the compound index yields a 6.8-fold performance improvement over the single-field index (1.33 ms versus 9.03 ms), while introducing a minimal storage differential of only 1 MB (220 MB versus 219 MB). This efficiency is driven by the low cardinality of the trailing secondary field, which allows the engine to compress composite keys without imposing disproportionate storage overhead, rendering its one-time index build cost justifiable for production deployment in high-volume e-commerce environments.

Nevertheless, certain architectural limitations remain to be addressed in future work. This benchmarking was conducted on a single-node deployment, which may not capture the distributed coordination or network replication latencies inherent to multi-node sharded production environments. Additionally, the experimental workload focused exclusively on read queries and did not measure the write latency trade-offs introduced by index maintenance during intensive write operations. Future research should expand this evaluation framework into multi-node MongoDB cluster deployments, encompass a broader operational mix including write-heavy transaction benchmarks and complex aggregation pipelines, and conduct comparative analyses across other document-oriented NoSQL systems to validate cross-system structured field indexing scalability at production scale.

References

[1] A. Makris, K. Tserpes, G. Spiliopoulos, D. Zissis, and D. Anagnostopoulos, “MongoDB Vs PostgreSQL: A comparative study on performance aspects,” *Geoinformatica*, vol. 25, no. 2, pp. 243–268, Apr. 2021, doi: 10.1007/s10707-020-00407-w.

[2] D. A. Saputra, Mohammad Adi Farich, Muhammad Naafi’an Anugerah, Imam Prayogo Pujiono, and Dicky Anggriawan Nugroho, “Perbandingan Kinerja MongoDB dan MySQL pada Aplikasi dengan Beban Data Besar,” *SAINSTECH: JURNAL PENELITIAN DAN PENGKAJIAN SAINS DAN TEKNOLOGI*, vol. 35, no. 4, pp. 71–78, Dec. 2025, doi: 10.37277/stch.v35i4.2543.

[3] A. Hamadou *et al.*, “A COMPARATIVE EXPERIMENTAL STUDY OF INDEX PERFORMANCE IN MONGODB AND POSTGRESQL,” *Int. J. Eng. Sci. Res. Technol.*, vol. 15, no. 4, 2026, [Online]. Available: http://www.ijesrt.com.

[4] A. Juan Syahwali, J. Jenderal Ahmad Yani No, P. Palembang, and S. Selatan, “Pemanfaatan MongoDB dalam Sistem Informasi Akademik untuk Pengelolaan Data Mahasiswa Pada Universitas Bina Darma,” *Jurnal Sains Student Research*, vol. 3, no. 2, pp. 466–469, 2025, doi: 10.61722/jssr.v3i2.4335.

[5] A. A. Muhamad, D. Adhi Kusuma, I. Renaldi, and A. Wibowo, “Studi Literatur Komparasi SQL dan NoSQL dalam Pemilihan Basis Data Ideal untuk Skalabilitas Tinggi,” 2025. [Online]. Available: https://jurnal.unw.ac.id/index.php/IKN.

[6] M. Nuriev, R. Zaripova, O. Yanova, I. Koshkina, and A. Chupaev, “Enhancing MongoDB query performance through index optimization,” in *E3S Web of Conferences*, EDP Sciences, Jun. 2024, doi: 10.1051/e3sconf/202453103022.

[7] D. Tao, E. Liu, S. R. Kadupitige, M. Cahill, A. Fekete, and U. Röhm, “First Past the Post: Evaluating Query Optimization in MongoDB,” *Preprint*, Sep. 2024, [Online]. Available: http://arxiv.org/abs/2409.16544.

[8] R. Ivanov, “MongoDB Aggregation Pipeline Performance: Analysis of Query Plan Selection and Optimizer Behavior Across Versions and Collection Scales,” *Information (Switzerland)*, vol. 17, no. 5, May 2026, doi: 10.3390/info17050488.

[9] V. Sumalatha and S. Pabboju, "Optimal Index Selection using Optimized Deep Deterministic Policy Gradient for NoSQL Database," *Engineering, Technology and Applied Science Research*, vol. 14, no. 6, pp. 18125–18130, Dec. 2024, doi: 10.48084/etasr.8832.

[10] G. Raphaela Mei Lanny Br Aritonang, M. Arief Hasan, M. Khasbulla Ridwan, M. Nur Iman, R. Carlo Pratama Silalahi, and R. Suranta Sipayung, "Optimasi Query SQL Server dengan Teknik Indexing dan Performance Monitoring," *JATI : Jurnal Mahasiswa Teknik Informatika*, vol. 9 No. 2, 2025, doi: 10.36040/jati.v9i2.13179

[11] K. S. Lestari, F. A. Riansyah, A. H. Taqyudin, and J. I. Saputro, "Optimalisasi Kinerja Indexing Dalam Pencarian Data Di Database Mongodb," *Journal Sensi:Strategic Of Education in Information System* , vol. 12 No.01, 2026, doi:10.33050/sensi.v12i1.4351

[12] T. P. Avrylya and Y. A. Susetyo, "Perbandingan Response Time Pencarian Menggunakan Text Indexing Pada MongoDB dan ArangoDB Berbasis Web," *MALCOM: Indonesian Journal of Machine Learning and Computer Science*, vol. 4, no. 3, pp. 777–785, May 2024, doi: 10.57152/malcom.v4i3.1308.

[13] Imam Thoib, Beda Puspita Candra, Nafis Sururi, and Danang Satya Nugraha, "Perbandingan Performa Pencarian Data Berbasis Teks dengan Dan Tanpa Full-text Index pada Basis Data MySQL," *INSOLOGI: Jurnal Sains dan Teknologi*, vol. 3, no. 6, pp. 663–673, Dec. 2024, doi: 10.55123/insologi.v3i6.4596.

[14] Swain, "Synthetic E-Commerce Dataset (Very Large)," 2025. Accessed: Jun. 01, 2026. [Online]. Available: https://www.kaggle.com/datasets/swainproject/synthetic-e-commerce-dataset-very-large.

[15] A. Hodijah, "Experimental Evaluation of Indexing Between HBase and MongoDB," *Atlantis Press Proceedings of the 2nd International Seminar of Science and Applied Technology (ISSAT 2021)*, 2021, doi: 10.2991/aer.k.211106.002