

Systematic Approach to NoSQL Indexing: Validating the ESR (Equality, Sort, Range) Heuristic for Query Performance

Narut Butploy¹, Kanokwan Khiewwan², Jindaporn Ongate³,
Thep Kueathaweekun⁴, Pakin Maneechot⁵

^{1,2}Department of Computer Technology, Kamphaeng Phet Rajabhat University, Thailand

³Department of Information Technology, Kamphaeng Phet Rajabhat University, Thailand

^{4,5}Department of Smart Grid Technology, Kamphaeng Phet Rajabhat University, Thailand

Article Info

Article history:

Received Dec 22, 2025

Revised Feb 25, 2026

Accepted Mar 9, 2026

Keywords:

NoSQL Database

Composite Index

Query Optimization

ESR Principle

Latency Reduction

ABSTRACT

In this paper we describe the ESR heuristic, a simple and repeatable way to build compound indexes in MongoDB. The method prioritizes the query requirements in order of equality, sort, and range. The ESR heuristic can boost the query performance without the need for a complicated machine learning model. The experiment was conducted on four real-world datasets and two kinds of mixed queries on Equality, Sort, and Range (ESR). The results show that the systematic technique for compound indexing based on the ESR principle reduces the mean query latency in most cases. For example, the Online Retail dataset had a clear improvement with p-values less than 0.01 in both Welch's t-test and the Mann-Whitney U-test. For the NYC Taxi dataset, the non-parametric test indicated a significant change in the latency distribution ($p=0.032$), implying that many queries have decreased tail latency. The Abalone dataset indicated a favorable trend, but it was not very significant statistically. The filter condition had low selectivity, hence only a little improvement was observed in the Bank Marketing dataset. The experiment also highlighted the COLLSCAN contradiction where MongoDB still scanned the entire collection with a proper compound index. However, the ESR heuristic decreased variance and tail latency in most circumstances.

Copyright © 2026 Institute of Advanced Engineering and Science.
All rights reserved.

Corresponding Author:

Kanokwan Khiewwan,
Department of Computer Technology,
Kamphaeng Phet Rajabhat University,
69 Moo 1, Nakornchum, Mueang, Kamphaeng Phet 62000, Thailand.
Email: kanokwan_kh@kpru.ac.th

1. INTRODUCTION

Document-oriented NoSQL databases have become popular due to their flexible schema design and ability to handle high volume transactional workloads [1, 2]. But query performance is still very dependent on index design. The absence of a proper index will make a query run as a full collection scan (COLLSCAN). This leads to high sorting cost as the database has to sort data in memory [3]. In practice, even a poorly designed index will still cause COLLSCAN and high resource usage. This is a major drawback of the NoSQL approach. It still lacks a clear and systematic way to construct effective compound indices [4].

Traditional relational database management systems or RDBMS have been optimized over the years compared to NoSQL. Examples include automatic index advisors and cost-based optimizers [5, 6]. Thus, query optimization needs efficient access paths, such as compound indexes, to filter, sort, and group data together.

MongoDB is a NoSQL database that stores data as documents. MongoDB still has major issues with index design. Sometimes queries still run as COLLSCAN and the system doesn't give many guidance as to the best key order for compound indexes. This often requires trial and error on users' part. It takes time and does not always give a clear answer. Another way of constructing compound indexes in MongoDB is proposed in this paper called ESR heuristic. The idea is based on the B-tree traversal and on the database index prefix rule.

In the ESR heuristic, Equality (E) conditions must precede Sort (S) and Range (R) conditions. This order is expected to increase query performance. From the experiment, we found the ESR heuristic does not guarantee the use of IXSCAN by MongoDB. The reason appears to be a mix of factors including the optimizer's internal rules, incorrect selectivity estimates, and the difference in results on the cloud platform. However, the primary goal of the ESR heuristic is not to minimize the total query latency but to provide more stable performance over different workloads. Even if IXSCAN is not selected, the ESR heuristic reduces tail latency. This helps to reduce the effect of slowest queries on high concurrency workloads. This paper evaluates the ESR heuristic for compound indexing in MongoDB. We analyze mean latency and tail latency, in real workloads with various characteristics. The experiment was run on four real-world datasets. The main contribution is the empirical evidence on the behavior and difficulty of query optimization in NoSQL databases. The following sections present related work, the ESR heuristic model, experimental design, results, and statistical analysis.

2. HEURISTIC MODEL OF ESR

A heuristic is a practical rule for finding a good solution without testing every possible option [7]. In this study, we employ a heuristic as an indexing rule for the MongoDB queries. The heuristic is not a replacement for the Query Optimizer. It is a hint to the optimizer when picking a compound index. Thus, the ESR heuristic reduces the trial-and-error effort in tuning the index. It is also a lighter alternative to a machine-learning-based index tuner. The ESR heuristic uses the same B-tree traversal logic that MongoDB uses for indexes. The rule is to put Equality fields first, followed by Sort fields and Range fields. There is a good reason for the order being E to S to R. Equality conditions reduce the number of documents in the early stage of query execution. S then has also a technical reason. Including the sort key in the index allows MongoDB to rely on the natural order of the B-tree and not have to sort in memory. Place R fields last, because a range scan works best after the query has filtered some documents. E to S to R order does not guarantee anything about IXSCAN access. There are queries for which MongoDB will not pick the ESR index even if it is created. It goes back to COLLSCAN. Thus, the ESR heuristic is a useful rule of thumb. In a compound index it only tells what is the field order to test. A complete index tuning process should also test other possible index orders. Sometimes the ESR order fails to find the best index. But it gives a direct and practical starting point. Easy to understand and easy to apply. This helps reduce the time spent on index tuning. The impact of the ESR heuristic is data distribution and selectivity of indexed fields dependent. Hence, this part describes the systematic approach for compound indexing from two views. The first viewpoint is traversal logic. The second view is a reduction in complexity.

2.1 Walking the B-tree and Pruning

The IndexB-tree indexing is one of the key building blocks of modern database management systems. It reduces the number of disk I/O operations by keeping the tree height low and by using an appropriate branching factor. The order of the ESR field is meant to help a query quickly discard a large amount of data from the search. This is similar to pruning the search tree. This is important for range queries that involve traversing the tree nodes from the root to the leaves. The ESR heuristic places Equality conditions at the front of the index field list. This helps the query to move to the required subtree faster without visiting nodes that are not related. This decreases disk I/O operations and improves the query processing time [8]. Another form of optimization involves the placement of Sort fields. For queries that contain a Sort operation, the default MongoDB query planner needs to sort the data in memory. This is why the ESR heuristic sorts Sort fields before Range fields. This order exploits the data order stored in the leaves of the B-tree. It also avoids extra in-memory sorting for queries with Range conditions.

2.2 Algorithm Complexity (Big O)

The ESR heuristic reduces the query time from $O(N)$ (naïve full collection scan) to $O(\log N + K)$ where K is the number of documents returned. The same index order as used for Range conditions helps the ESR heuristic to avoid the extra $O(K \log K)$ work to sort the results when sorting is not already specified in the query. That makes performance on small and large queries more predictable.

3. RESEARCH METHODOLOGIES

In this section we describe the evaluation of the ESR heuristic. IXSCAN using compound index in E to S to R order is not considered. The main goal is to analyze the performance of the ESR heuristic as a rule-based optimizer for queries on various datasets and mixed workloads. The ESR heuristic was tested on four datasets under real use conditions. To isolate the effect of the ESR heuristic from other factors, each workload was run 10 times with 100 queries per round per database. The design results were more robust and corresponded better to the true performance behavior of the ESR heuristic.

3.1. Document Data Model and Indexing Anti-Patterns Principles

Some of the most popular types of NoSQL database management systems are document-oriented NoSQL database models like MongoDB. The systems store data as semi-structured documents using the Binary JSON, or BSON, format. In the context of the CAP theory [9, 10], they provide horizontal scalability, high availability and flexible schema design. The indexing is one of the key parts of query performance in MongoDB [11]. MongoDB supports compound indexes for complex queries. A compound index is created on multiple fields, according to the index prefix rule. The order of the fields in the compound index is very important to the performance of queries that both filter and sort, and for range queries. COLLSCAN happens when there is no index or when an index exists but with the wrong field order and it scans the whole collection [12].

COLLSCAN is a problem because it is latency and resource-intensive. This is not what NoSQL systems are for. RDBMS systems employ a number of optimization techniques. Automatic index advisor, dynamic index usage, and secondary sorting for complex workloads are provided by relational databases [13]. This is not the pattern that is typical of NoSQL systems. NoSQL systems are not meant to do complex joins of tables as RDBMS systems are. This is why most NoSQL query optimizers are simpler. They don't always pick the best index for compound queries, especially on complex workloads. There exist some studies presenting useful results on NoSQL optimization using machine learning methods [14, 15]. The ML algorithms will analyze the workloads, identify the access patterns, and predict the appropriate index designs. These approaches usually require expensive training and regular maintenance. ML methods might not be suitable for NoSQL usage, because in this case the index has to be constructed at runtime when the query is being executed. There's also a need for structure in other NoSQL work. They are schema analysis [16] and analytical query optimization [17].

3.2. Workload and Data Set Definitions

The ESR heuristic was applied on four real datasets namely Online Retail, NYC Taxi Trips, Abalone and Bank Marketing. These datasets are from the performance test set of the Kaggle database. The four datasets have different schema structure, field types, cardinality and selectiveness. This difference enabled to test the ESR heuristic on different conditions of data. The records were then converted to JSON format to be processed in MongoDB Atlas. We defined three types of queries: Q1, Q2 and Q3. Query types are divided into three workload groups. Q1, 60%: This is a compound query with an equality condition, a sort operation and a range condition. It is the most important type of ESR query because it covers all three parts of the ESR heuristic. For this query type, the compound index was defined in the order E to S to R. Q2, 30 percent: This is a secondary level analytical query with range condition. It is a simpler query type. This experiment was designed to test the performance of the ESR heuristic on queries from outside the core target group. Q3, 10% This is a simple equality check. It was used to verify that the ESR heuristic incurred only a small overhead for simple queries. This query mix is typical for query patterns in document databases. The three types of query also hint at different kinds of complexity. The experiment combined analytical and transactional processing workloads in NoSQL databases to produce a stochastic workload like them. This provided a way to estimate the mean effect of the ESR heuristic for more realistic conditions.

3.3. Experimental Apparatus

The experiment was performed using Python and PyMongo driver to connect with MongoDB Atlas. We ran 10 iterations of 100 queries each workload. This step eliminated random errors from network instability and cache effects. For the Baseline and Optimal scenarios, each database was run for 1,000 query executions. The experiment was conducted in two steps (see Figure 1).

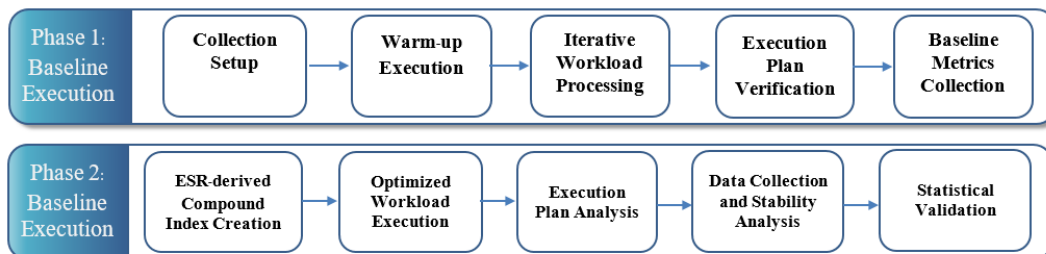


Figure 1. Two-Phase Experimental Procedure for ESR Heuristic Validation

Figure 1 shows the experimental procedure conducted to validate the ESR heuristic. The process was consisted of two phases as below:

Phase 1: Evaluate baseline performance. In this phase, the query performance was tested with MongoDB default settings. This result was used as the baseline for comparison to the ESR heuristic. The database was tested in a raw, un-optimised form. The goal was to see how full collection scans and in-memory sorting affect high latency and performance variability in a cloud environment. The steps are:

- Database setup: The JSON datasets were loaded into MongoDB Atlas to setup the testing environment. At this step, the database was checked to ensure that no compound index existed. Any compound indexes that existed were dropped. The only index that was kept in the beginning of the experiment was the default `_id` index.
- Warm-up run: The proper experiment was preceded by a warm-up run. This resulted in fewer stochastic factors such as network instability and caching of cloud services.
- Iterative query execution: The core of the experiment was executing 1000 queries for each workload. The 10 repeated rounds with 100 queries each were used to reduce random errors. The queries were chosen from a random distribution of workload. The workload consisted of three types of queries, Q1, Q2, and Q3, in the ratio of 60%, 30%, and 10%, respectively.
- Evaluation of execution plan: The `explain()` method was used to analyze the execution plan of each query. The query plan showed the type of scan used, COLLSCAN or IXSCAN. It also told you if it had to be sorted in memory.
- Baseline performance metrics: After completion and processing of the experiment, we measured query latency or query response time as descriptive statistics in Table 1. The Baseline results were taken as reference for the comparison with the ESR heuristic results

Phase 2: In this phase, we created a compound index based on the ESR order and then ran the same workload again. The purpose was to compare the new results with the Baseline results and check whether the ESR index helped reduce latency.

- ESR Heuristic Execution Plan: We created a compound index by ordering the fields in the above mentioned E to S to R order. The index was designed to support the main ESR query pattern for Q1 queries.
- Optimized Query Execution: The same query workload was run again with the index in place. 1000 queries per data set, repeated 10 times.
- Query execution plan analysis: The execution plan was re-examined in the same manner as in the Baseline phase. This was to test if MongoDB was picking IXSCAN or still falling back to COLLSCAN in some cases.
- Performance data collection and stability analysis: The evaluation was not only based on average latency. We also took into consideration median and tail latency since some slow queries can affect the practical response time of the system.
- Statistical evaluation: The results of the Baseline and ESR were statistically compared using Welch's t-test and the Mann-Whitney U-test (Table 2).

3.4. Experimental Environment

Cloud Native MongoDB Atlas For this experiment, the database was hosted on a 3-node cluster on MongoDB Atlas 8.0.19. It was launched in ap-southeast-1, the AWS Singapore region. The setup was M0 Sandbox plan and WiredTiger storage engine. The server was using shared CPU and shared RAM, as stated in the test environment. All queries and evaluations were run in Python. The database client, PyMongo, ran on a Windows 11 machine. We ran each query 1,000 times to reduce the effect of network noise and cloud service instability on the query results.

3.5. Measurement Metrics

The main metric we measured was the average query latency, in milliseconds. The mean value was not used in isolation, as in some rounds a few slow queries affected the average. The latency distribution was therefore also analyzed with the median, interquartile range and whisker spread. Stability was evaluated by standard deviation and number of outliers. Execution behavior was explored via COLLSCAN, IXSCAN, FETCH and SORT stages. We measured the effect of the ESR index with two values: the mean latency and the run-to-run variance. IXSCAN alone was not used as the definition of success. Even when the ESR index was already present, MongoDB still selected COLLSCAN for some rounds at the plan check. This was within the planner's normal variation.

4. RESULTS AND DISCUSSION

First we ran the workload without ESR indexes. This run served as baseline. Table 1 lists the latency values and the execution-plan results.

4.1. Baseline Performance Analysis

ESR indexes were not part of the baseline test collections. Table 1 shows the latency values and the execution plan observed in this condition.

Table 1. Baseline Performance Analysis Summary

Dataset	Mean Latency			Median Latency			Execution Plan Key Findings
	Q1	Q2	Q3	Q1	Q2	Q3	
Online Retailing	60.46	52.84	65.42	49.69	52.12	53.21	COLLSCAN: True
NYC Taxi	60.38	159.25	500.89	52.46	147.42	543.94	COLLSCAN: True
Abalones	113.04	145.45	79.10	90.28	124.71	81.72	COLLSCAN: True
Bank	109.96	151.76	152.42	98.36	135.97	147.89	COLLSCAN: True

In the baseline run, all datasets employed COLLSCAN (see Table 1). This means MongoDB scanned the collections without a compound index for these queries. Thus the baseline result was used as the reference point for checking the effect of the ESR index.

4.2. Optimal Performance Analysis

Once the ESR index is built, the same workload was tested again. Table 2 shows the comparison between the baseline and the ESR runs.

Table 2. Comparison of mean and maximum latency reduction with ESR model

Dataset	Baseline Mean (ms)	Optimal Mean (ms)	Mean Reduc. (%)	Baseline Max (ms)	Optimal Max (ms)	Max Reduc. (%)
Online Retailing	53.97	41.49	31.40%	419.33	45.11	89.24%
NYC Taxi	60.38	46.53	22.93%	491.49	216.49	55.95%
Abalones	113.04	104.74	7.34%	255.58	158.71	37.90%
Bank	109.96	105.52	4.04%	185.98	190.09	-2.21%

Table 2. Latency comparison before and after applying the ESR index. The most evident improvement was on the Online Retail dataset. The mean latency was reduced from 60.46 ms to 41.49 ms, and the maximum latency was reduced from 419.33 ms to 45.11 ms. This result shows that the ESR index helped to reduce both average latency and slow query cases in this dataset. NYC Taxi also improved, but by less. The mean Q1 latency was reduced from 60.38 ms to 46.53 ms, and the maximum latency was reduced by 55.95%. Abalone decreased only slightly, from 113.04 ms to 104.74 ms. Bank Marketing had the smallest reduction in mean latency, at 4.04%, and its maximum latency increased slightly. The cost of write operations should also be considered. A compound index adds overhead to insert and update operations because MongoDB has to update the index in addition to the documents. For this read focused test the ESR index was small, less than 1MB and the overhead was acceptable. This overhead should be re-measured before using the ESR index for write-heavy workloads.

Table 3. Statistical Significance of Q1 Latency Improvements After Applying the ESR Index

Dataset	The significance of statistics					Interpretation
	n (ERS)	t-statistic	p-value (t-test)	U-statistics	p-value (U-test)	
Online Retailing	58	2.72	0.0085	820	0.006	Significant in both tests
NYC Taxi	56	1.71	0.091	960	0.032	Significant in U-test, non-significant in t-test
Abalones	47	1.59	0.116	1030	0.084	Marginal trend, not significant
Bank	63	0.92	0.361	1200	0.274	Not significant in both tests

Table 3. Results of statistical test for the change of Q1 latency after applying the ESR index. Mean latency was compared using Welch's t-test and latency distribution was compared using Mann-Whitney U-test. Both tests showed significant results for Online Retail with $p = 0.0085$ for Welch's t-test and $p = 0.006$ for the Mann-Whitney U-test. NYC Taxi was only significant in Mann-Whitney U-test ($p=0.032$) and the result of its Welch's t-test was not significant. Abalone exhibited only a slight trend. Bank Marketing also did not prove to be significant in either test. The results show that the ESR index was the best performer in cases where the equality field could reduce the number of documents scanned, such as in Online Retail. For NYC Taxi the mean latency was not significantly improved, but the latency distribution improved. This implies lower tail latency in practice. Abalone and Bank Marketing show the limitation of the benefit of ESR when the equality field is of low selectivity.

4.3. Latency Distribution Analysis

Average latency isn't the full picture of query performance. Even if the average value looks good, some queries may take significantly longer than others. Thus, we also performed the box plots of the distribution of Q1 latency as shown in Figure 2(a)–(d).

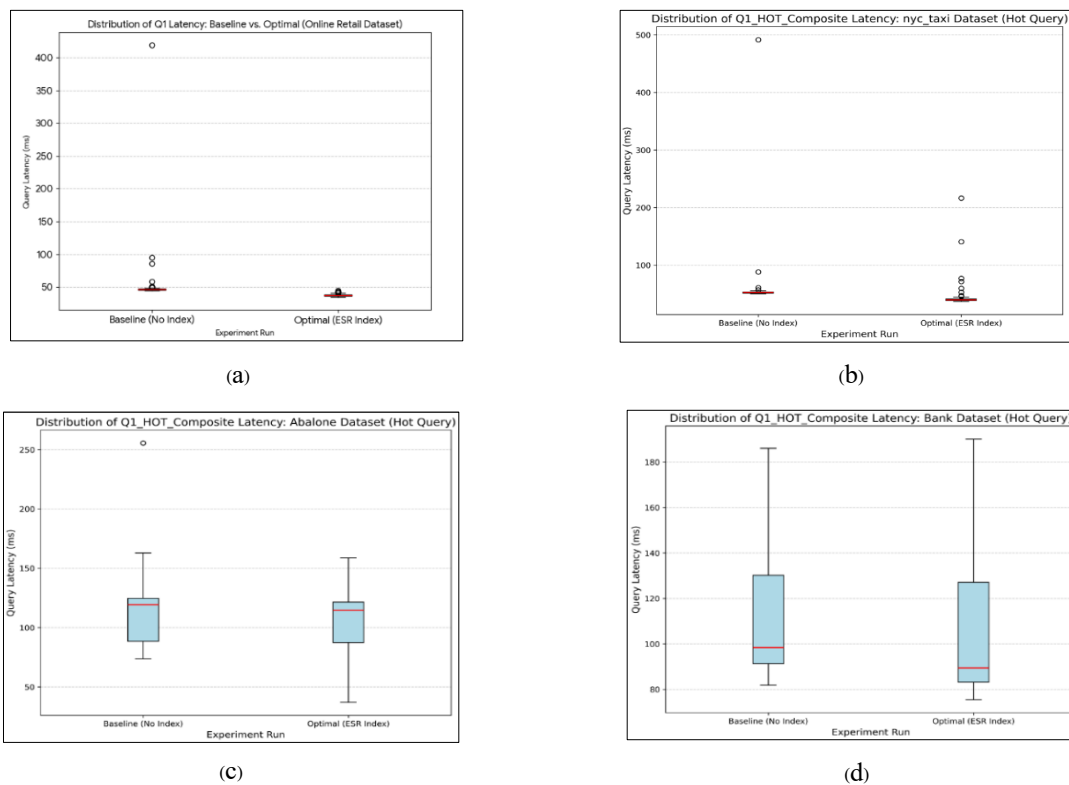


Figure 2. Q1 Latency Distribution Comparison

Figure 2 shows the change in Q1 latency after applying the ESR index. The most visible change was in the Online Retail dataset. After the application of the ESR index, the interquartile range was reduced and the highest outlier was eliminated. The lower tail latency of NYC Taxi is also consistent with the Mann-Whitney U test result [17]. Abalone provided only a slight improvement, mainly on the median value. Bank Marketing had a lower median, but it had a wider IQR. Some queries were faster, while other queries still suffered from unstable execution time. The result is related to the low selectivity of the equality field.

The ESR field ordering rule did not help for all data sets in an equivalent manner. Its effectiveness depended on how well the equality field separated the data into smaller groups. For instance, the NYC Taxi had the second best improvement. One possible reason is its equality field cut down on the number of documents MongoDB had to scan. COLLSCAN behavior was also different among datasets, hence, scan latency was not uniform in all the cases [18]. One important observation was that for some queries MongoDB did not use the ESR index even though it was available. The cause of the problem was not immediately clear.

This could be a result of a query planner cache, the equality field having less selectivity, performance differences due to multi-tenancy, or contention in Atlas. However, the ESR index was useful in most cases despite the above problems.

The result also pointed out the limitation of ESR rule. The primary value of the ESR rule is in cases where the equality field reduces the number of documents to scan. When the selectivity of a field is low, the value of the ESR index is limited; it may be better to ignore such fields when creating indexes. Thus, the benefit of ESR index needs to be evaluated based on the database's specific features and the query workload. The simplicity of the ESR heuristic makes it an attractive alternative to the model-driven approaches to index tuning. Unlike ML-based index recommenders that require a lot of computational resources to train and do not perform well on small training datasets [19], ESR does not introduce any conceptual complexity and overhead.

5. CONCLUSION

This paper tests the ESR field order rule on four data sets, using repeated tests in MongoDB Atlas . The results indicate that the rule is most effective for the Online Retail data set. The mean latency of Q1 reduced by 31.40% while the maximum latency reduced by 89.24%. NYC Taxi also improved, but the effect was not as big. Its mean latency was decreased by 22.93%. The Mann-Whitney U test also indicated a significant change in the distribution of latency ($p = 0.032$) [20, 21]. Abalone had improved a little. The main limitation of the ESR index was shown in Bank Marketing, since its equality field had low selectivity [22]. MongoDB compound indexes for equality-sort-range queries are a good start for the ESR order. After creating the index developers should run `explain()` and check the execution plan to see that IXSCAN is used and that COLLSCAN or SORT is no longer present.

REFERENCES

- [1] A. O. Thakare, O. W. Tembhurne, A. R. Thakare, and S. N. Reddy, "NoSQL Databases: Modern Data Systems for Big Data Analytics - Features, Categorization and Comparison," *International Journal of Electrical and Computer Engineering Systems*, vol. 14, no. 2, pp. 207-216, 2023.
- [2] B. Jose and S. Abraham, "Performance analysis of NoSQL and relational databases with MongoDB and MySQL," *Materials Today: Proceedings*, vol. 24, no. 3, pp. 2036-2043, 2020.
- [3] S. Ramzan, I. S. Bajwa, R. Kazmi, and Amna, "Challenges in NoSQL-Based Distributed Data Storage: A Systematic Literature Review," *Electronics*, vol. 8, no. 5, p. 488, 2019.
- [4] M. A. Qader, S. Cheng, and V. Hristidis, "A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases," in *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*, Houston, TX, USA, 2018, pp. 551-566.
- [5] D. Preuvenciers and W. Joosen, "Automated Configuration of NoSQL Performance and Scalability Tactics for Data-Intensive Applications," *Informatics*, vol. 7, no. 3, p. 29, 2020.
- [6] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic Database Management System Tuning Through Large-scale Machine Learning," in *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*, Chicago, IL, USA, 2017, pp. 1009-1024.
- [7] M. R. Hasanzadeh, F. Keynia, and M. Hashemipour, "A New Index-Based Hyper-Heuristic Algorithm for Global Optimisation Problems," *IET Software*, vol. 16, no. 5, pp. 493-515, 2022.
- [8] Y. Kwon, S. Lee, Y. Nam, J. C. Na, K. Park, S. K. Cha, and B. Moon, "DB+-tree: A new variant of B+-tree for main-memory database systems," *Information Systems*, vol. 119, p. 102287, 2023.
- [9] M. Diogo, B. Cabral, and J. Bernardino, "Consistency Models of NoSQL Databases," *Future Internet*, vol. 11, no. 2, p. 43, 2019.
- [10] Y. Yan, S. Yao, H. Wang, and M. Gao, "Index Selection for NoSQL Database with Deep Reinforcement Learning," *Information Sciences*, vol. 561, pp. 20-30, 2021.
- [11] F. Z. Belkadi and R. Esbai, "A Model-Driven Engineering: From Relational Database to Document-oriented Database in Big Data Context," in *Proceedings of the 16th International Conference on Software Technologies (ICSOFT 2021)*, 2021, pp. 653-659.
- [12] A. Rafique, D. Van Landuyt, B. Lagaisse, and W. Joosen, "On the Performance Impact of Data Access Middleware for NoSQL Data Stores: A Study of the Trade-Off between Performance and Migration Cost," *IEEE Transactions on Cloud Computing*, vol. 6, no. 3, pp. 843-858, 2018.
- [13] X. Yao, J. Li, Y. Tao, and S. Ji, "Relational Database Query Optimization Strategy Based on Industrial Internet Situation Awareness System," in *Proceedings of the 7th International Conference on Computer and Communication Systems (ICCCS)*, Wuhan, China, 2022, pp. 152-155.
- [14] M. Abbasi, M. V. Bernardo, P. Váz, J. Silva, and P. Martins, "Optimizing Database Performance in Complex Event Processing through Indexing Strategies," *Data*, vol. 9, no. 8, pp. 1-24, 2024.
- [15] A. Karras, C. Karras, D. Samoladas, K. C. Giotopoulos, and S. Sioutas, "Query Optimization in NoSQL Databases Using an Enhanced Localized R-tree Index," in *Proceedings of the 24th International Conference on Information Integration and Web Intelligence (iiWAS 2022)*, Springer LNCS 13635, 2022, pp. 391-398.

- [16] N. El Moukhi, I. El Azami, and S. Hajbi, "Towards a New Hybrid Approach for Building Document-Oriented Data Warehouses," *International Journal of Electrical and Computer Engineering*, vol. 12, no. 6, pp. 6423–6431, 2022.
- [17] C. Delimitrou and C. Kozyrakis, "Amdahl's Law for Tail Latency," *Communications of the ACM*, vol. 61, no. 8, pp. 65-73, 2018.
- [18] H. He, *et al.*, "Benchmarking Distributed Transactional Database Systems," in *Proceedings of the 2024 International Conference on Benchmarking, Measuring, and Optimizing (Bench 2024)*, LNCS 15519, 2025, pp. 37-53.
- [19] T. Siddiqui and W. Wu, "ML-powered Index Tuning: An Overview of Recent Progress and Open Challenges," *SIGMOD Record*, vol. 52, no. 4, pp. 19-30, 2024.
- [20] A. E. Topcu, *et al.*, "Evaluating the Performance of NoSQL Databases for Big Data in Cloud Computing Environments," *HighTech and Innovation Journal*, vol. 6, no. 3, pp. 808-830, 2025.
- [21] Z. Fan, *et al.*, "Understanding the Long Tail Latency of TCP in Large-scale Cloud Networks," in *Proceedings of the 2025 Asia-Pacific Workshop on Networking (APNET 2025)*, Shanghai, China, 2025, pp. 1-8.
- [22] M. Aytimur and A. Çakmak, "Estimating the Selectivity of LIKE Queries Using Pattern-based Histograms," *Turkish Journal of Electrical Engineering & Computer Sciences*, vol. 26, pp. 3319-3334, 2018.