e-ISSN: 3165-1453 https://doi.org/10.36782/gitcoin.v1i1.693

Journal of Digital Transformation, Computing, and Intelligence (GITCOIN)
Vol. 1, No. 1, March 2026

Performance Evaluation of a Hybrid Trie and Levenshtein Distance Search Architecture Compared to Various Search Algorithms

Nurhakim As'ad Wicaksono[1)], Nurul Asiah[2)*], Guson Prasamuarso Kuntarto[1)], Irwan Prasetya Gunawan[1)]

[1)] Department of Informatics, Faculty of Enginering and Computer Science, Universitas Bakrie, Jakarta, Indonesia
nurhakim.wicaksono@bakrie.ac.id, guson.kuntarto@bakrie.ac.id, irwan.gunawan@bakrie.ac.id

[2)] Universitas Bakrie Press, Universitas Bakrie, Jakarta, Indonesia
*nurul.asiah@bakrie.ac.id

ABSTRACT

Information retrieval in search systems frequently struggles to achieve both rapid query response times and robust error tolerance simultaneously. While numerous search algorithms are available, there is a lack of comparative studies evaluating their performance in balancing these two critical aspects. Among the available options, an opportunity exists to combine the Trie data structure and the Levenshtein Distance algorithm. Trie excels in computational speed, whereas Levenshtein Distance provides superior error tolerance. Leveraging these respective strengths creates an opportunity to merge both approaches to enhance overall search performance. Therefore, this study aims to propose and evaluate a hybrid search architecture that integrates Trie with Levenshtein Distance. An experimental quantitative methodology was conducted to evaluate eight different search algorithms. Performance was measured based on runtime execution and robustness metrics using a dataset of 3,000 book titles to simulate various typographical error scenarios. The evaluation process was automated using Python scripts, executing 30 iterations per query to ensure statistical validity. The results demonstrate that the proposed hybrid Trie and Levenshtein Distance achieves an average execution time of (4.324 ms). This performance is approximately 17.6 times faster than the native Levenshtein Distance (76.25 ms) and 34.6 times faster than the Damerau-Levenshtein Distance (149.73 ms). However, in terms of robustness, the hybrid approach did not show a significant improvement over the native fuzzy algorithms, as it successfully maintained the exact same level of error tolerance. The Trie structure effectively accelerates the search by pruning the search space, while Levenshtein Distance maintains flexibility in handling user input errors. In conclusion, this hybrid architecture provides a highly efficient and robust solution for autocomplete search features, successfully balancing computational speed with user-friendly error tolerance.

Keywords: Levenshtein Distance, Runtime Performance, Robust, Search Algorithm, Trie

Article history: *Received* 4 July 2026, *revised* 25 July 2026, *accepted* 7 August 2026

1. INTRODUCTION

The development of current information retrieval systems demands a strict balance between fast query response times and the reliability of providing relevant results [1]. Maintaining this rapid response time is a major challenge because as the number of data records to be evaluated increases, the execution time can spike sharply, triggering computational queues (bottlenecks) that burden server performance [2]. In specific implementations, such as search engines for library catalogs, this technical challenge is further complicated by the most fundamental problem frequently encountered in the field: user typographical errors (typos) when inputting keywords [13], [14], [18]. If the system solely relies on exact search algorithms like sequential search, the computational process runs very quickly. However, this approach is extremely rigid; the system will fail to find relevant results simply due to one or two mistyped letters. On the other hand, the use of fuzzy string matching algorithms, such as Levenshtein Distance, offers a flexible solution to recognize deviated spellings [16]. Unfortunately, calculating the distance matrix in this algorithm consumes significant computational resources, which inevitably exacerbates the aforementioned server bottlenecks [15].

To overcome the burden on server performance while handling typographical errors, various studies have implemented different search algorithms. However, an exploration of the literature shows that these algorithms are generally still utilized separately as standalone solutions, depending on the specific priority of the system requirements. In scenarios prioritizing exact search speed, a study by Sjofjan et al. [1] applied the Sequential Search algorithm combined with an Autocomplete feature in a digital archive system. This implementation recorded an average execution time of 0.00496 seconds for 250 user records with a 100% effectiveness rate in

© 2026 Journal of Digital Transformation, Computing, and Intelligence (GIT-COIN)
This is an open access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY-NC 4.0).

word suggestions. Its main strength is the extremely fast execution for exact matches, but its significant weakness is its absolute rigidity, as it completely fails to return results if a user makes a typographical error. Conversely, addressing the flexibility aspect, research by Daniati et al. [2] developed a book data search feature in the SMP N 2 Sooko library using the Levenshtein Distance algorithm. By calculating the shortest string distance, this system proved successful in improving accuracy and helping users find books more efficiently compared to manual recording. The primary advantage of the Levenshtein Distance lies in its robust flexibility and high tolerance for spelling errors; however, its major drawback is the heavy computational cost, making it significantly slower when processing large datasets.

Furthermore, research by Goenawan et al. [3] successfully applied the Trie algorithm as a data structure to store words in a dictionary, combined with depth-first search (DFS) to traverse the Trie in searching for words that match user input. Implemented on a website built with HTML, CSS, and JavaScript, the algorithm successfully displayed word suggestions quickly based on user input, accelerated word searches, and improved efficiency compared to manual search methods. The Trie algorithm offers a massive advantage in accelerating prefix-based searches and optimizing computational speed. Nevertheless, its critical limitation is the total lack of error tolerance; the system cannot handle typos and will fail if the input prefix is misspelled.

Lastly, an investigation by Arsyta et al. [4] attempted to overcome keyword mismatch issues in the archive information system of Jerukseger Village using the Damerau-Levenshtein Distance algorithm. This algorithm's strength is its highly comprehensive error handling proving effective in handling various types of typographical errors such as character deletion, insertion, substitution, and transposition with 100% accuracy. However, this capability comes with a severe drawback: it is computationally exhaustive, as the search process required an average execution time of 298.1452 ms, rendering it inefficient for scalable systems. The majority of literature chooses only one side: either relying independently on fuzzy algorithms that are accurate but consume longer processing times, or opting for sequential searches/trie that are fast but intolerant to typos. Studies exploring the potential of combining two algorithm architectures with different characteristics to mutually compensate for their respective weaknesses are rarely found. This is where the research gap and urgency of this study lie. There is a need for a search architecture that does not operate merely on a single axis, but rather a scalable system capable of delivering time efficiency while maintaining tolerance for typographical errors.

This series of previous studies prove that each algorithm possesses significant advantages in its respective domain; however, the approaches remain confined to singular implementations. The majority of literature chooses only one side: either relying independently on fuzzy algorithms that are accurate but consume longer processing times, or opting for sequential searches/trie that are fast but intolerant to typos. Studies exploring the potential of combining two algorithm architectures with different characteristics to mutually compensate for their respective weaknesses are rarely found. This is where the research gap and urgency of this study lie. There is a need for a search architecture that does not operate merely on a single axis, but rather a scalable system capable of delivering time efficiency while maintaining tolerance for typographical errors [20].

As a solution to this gap, this study proposes the implementation of a hybrid architecture that combines the strengths of the Trie and Levenshtein Distance algorithm. By merging these two algorithms, the system has the possibility to achieve computational efficiency and search flexibility. The core concept is to utilize the Trie data structure as an initial filtering framework (search space pruning) based on word prefixes. Through this mechanism, the computationally heavy Levenshtein calculations are no longer blindly executed across the entire dataset, but are instead computed only on specific nodes that have passed the initial filtering and are proven to have a high probability of relevance. To validate the success of this algorithmic combination, its performance must be systematically evaluated by comparing it against seven other search algorithms across two primary aspects, including response time and robustness testing.

To ensure a focused evaluation, the scope of this research is specifically centered on computational performance and robustness. The discussion emphasizes the comparative testing of execution time and error tolerance between the hybrid Trie + Levenshtein model compared to exact algorithms and single native fuzzy algorithms, using a dataset of 3,000 book titles.

2. RESEARCH METHOD

This study evaluates the performance of the search architecture by integrating a quantitative method for response time testing and a qualitative approach for robust testing. To facilitate this integrated evaluation, the study employs an experimental method in computational testing, which is a structured approach commonly utilized to design, execute, and compare the performance of various search algorithms objectively on a large-scale database [5]. According to a comprehensive literature review on text search structure testing and comparative algorithm evaluation [6], this experimental methodology relies on several sequential main stages. These stages include algorithm selection, determining testing metrics, algorithm testing, and comparison of test results as can be seen in Figure 1.

This multi-approach comparison was selected because the performance analysis of information retrieval systems requires concrete empirical evidence and cannot solely rely on theoretical assumptions of time

complexity. Consequently, the testing scenarios are specifically designed to compare the efficiency of various algorithm variations by capturing both their quantitative execution speed (runtime) and their qualitative reliability in handling input errors. The entire sequence of these processes is structured into an automated workflow to ensure the consistency of the evaluation results.

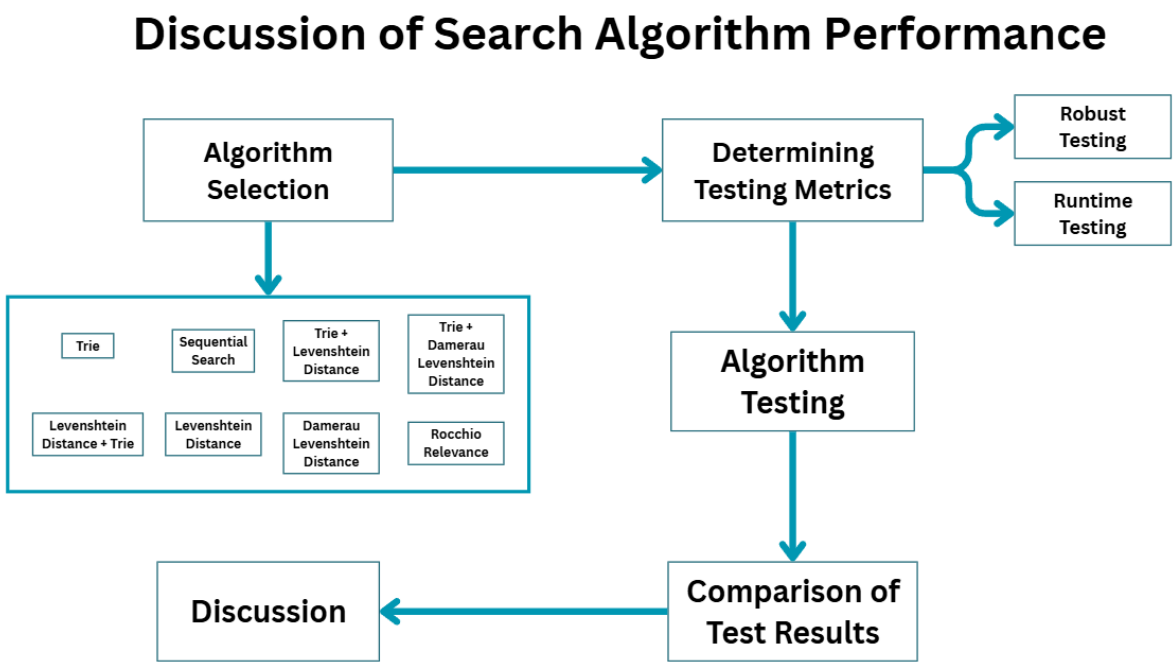


Figure 1. Stages of Search Algorithm Performance Testing

Within this overarching experimental framework, the core focus of the evaluation lies in testing the proposed hybrid architecture, which integrates the Trie and Levenshtein Distance algorithms. Because the performance of this specific hybrid algorithm is being evaluated, it is essential to first understand its operational workflow to provide a clear technical overview. Therefore, before detailing the testing scenarios, the specific operational workflow is illustrated in Figure 2. The flowchart demonstrates how the Trie structure acts as an initial filter to prune the search space based on prefixes, significantly reducing the dataset before the Levenshtein Distance calculation is selectively applied to the remaining candidate nodes to tolerate typographical errors.

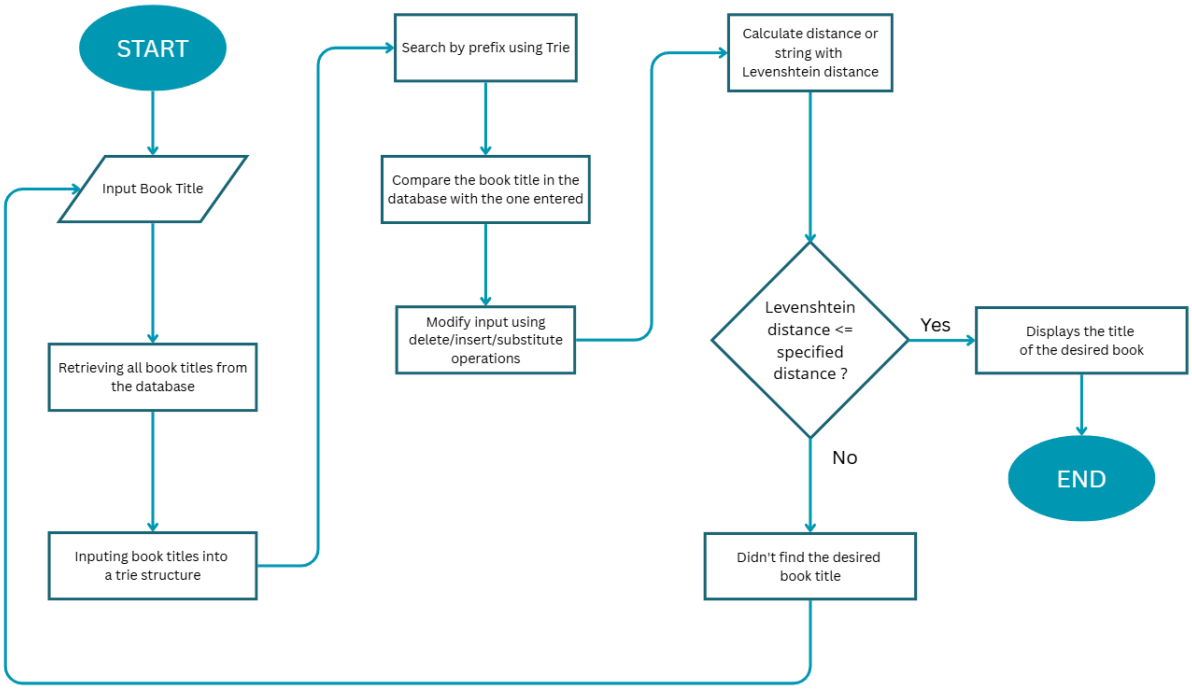


Figure 2. Flowchart of the Hybrid Trie and Levenshtein Distance Algorithm

2.1. Algorithm Selection

This evaluation stage compares eight variations of search algorithms representing various computational approaches. The eight evaluated algorithms consist of the hybrid Trie + Levenshtein Distance, Levenshtein Distance + Trie, native Trie, Sequential Search, Rocchio Relevance Feedback, native Levenshtein Distance, Damerau-Levenshtein Distance, and Trie + Damerau-Levenshtein Distance. These variations were deliberately selected to observe a comprehensive performance spectrum ranging from rigid exact search methods and dynamic fuzzy character distance evaluations to hybrid approaches [7], [19]. Through this multi-approach comparison, the position, advantages, and efficiency level of the proposed algorithm (Trie + Levenshtein Distance) can be objectively evaluated when compared against other conventional methods in database search scenarios [6].

2.2. Determining Testing Metrics

Performance testing focuses on two main metrics that represent modern search engine architectures. The first metric is runtime performance, which is measured using benchmark runtime testing methods and scalability testing. The objective is to track how quickly the algorithm resolves search queries, as well as to observe fluctuations in its response speed alongside the increase in data volume [5]. The second metric is robustness, which is designed to measure the system's reliability in handling imperfect query inputs, such as spelling errors (typos) or missing characters. This metric is highly essential to validate how accurately edit distance-based algorithms tolerate user errors and still return relevant search results [7].

2.3. Algorithm Testing Scenarios

To measure the runtime performance metric, the testing utilized a database consisting of 3,000 book titles from Kaggle. Specifically for the Kaggle dataset, preprocessing was conducted by restricting the title length to between 20 and 40 characters. This selection aims to maintain the consistency of the test data complexity to ensure it is representative. The testing was executed incrementally, starting from a baseline of 55 data entries, and then the load was increased to 300, 600, until reaching a maximum limit of 3,000 data entries. To minimize system bias and ensure statistical validity, each query was executed through 30 iterations [8]. Furthermore, to isolate the testing environment and mitigate external interferences that could cause runtime fluctuations such as background memory consumption or unpredictable processor management all non-essential applications were completely terminated during the testing phase. The evaluation was conducted in a strictly controlled environment where only the Python testing script was actively running. This isolation strategy guarantees that the variance recorded across the 30 iterations is kept to an absolute minimum, ensuring the measured execution times accurately and objectively reflect the algorithms' true computational performance.

The entire computational process of the execution time was automated using a Python script via Anaconda Navigator. To obtain pure time records without the intervention of interface interaction overhead (such as user typing delays), the testing directly evaluated the full character string of the titles. The output of this testing was then extracted into Comma Separated Values (CSV) format to calculate the average values. On the other hand, the robustness testing was purely focused on the 3,000 Kaggle data entries using the PHP programming language. The selection of this large-scale data volume is crucial to ensure that the variations in search results are more representative in testing error tolerance. This scenario simulated various forms of typographical errors on specific keywords, namely "life" and "book", and then tracked how each algorithm processed and tolerated these typos.

To provide a concise overview of the runtime performance algorithm testing scenario flow, it is illustrated in Figure 3.

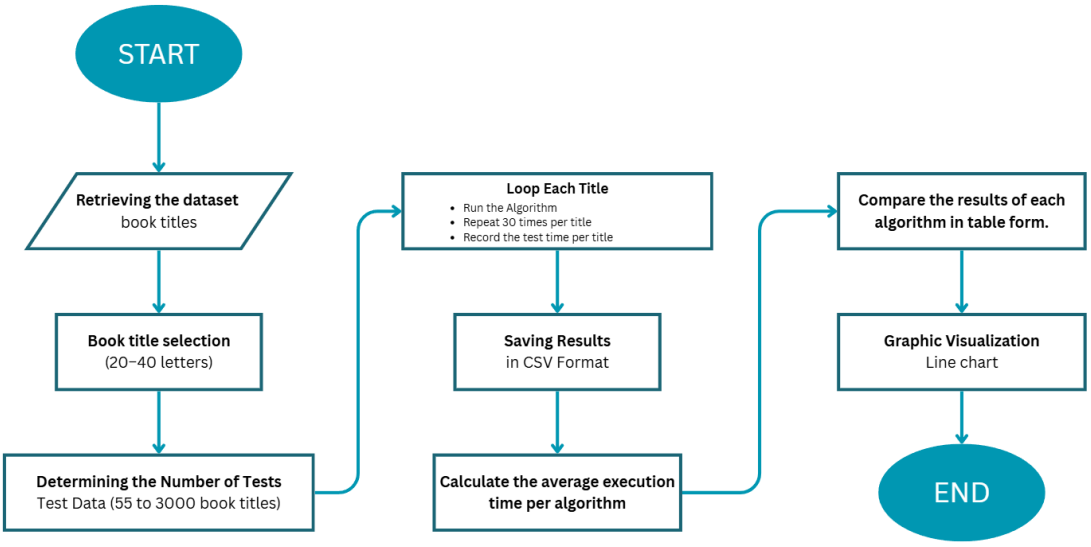


Figure 3. Algorithm Performance Testing Flowchart (Runtime Performance).

2.4. Search Algorithm Performance Results

Through the executed scenarios, the results of each algorithm were compiled to enable a balanced comparison. The runtime performance metric, which includes the average execution time of a total of 3,000 book title queries, is presented visually in the form of a scalability graph. Meanwhile, the analytical response of the algorithms regarding spelling error handling (robustness) is mapped into a comparative table, demonstrating the level of output relevance based on the simulated typo variations. These two evaluation metrics directly reflect the integrated methodology of this study. The runtime measurement represents the quantitative method, focusing on numerical data to objectively evaluate computational efficiency. Conversely, the robustness assessment embodies the qualitative method, which analyzes the contextual accuracy and practical quality of the search results when handling imperfect user inputs.

3. RESULTS AND DISCUSSION

3.1 Runtime Testing

The primary objective of the runtime testing is to evaluate the computational efficiency and scalability of each algorithm as the dataset grows. Based on the evaluation, the results demonstrate a clear general trend as the number of tested data entries increases scaling from the initial baseline of 55 records up to the maximum load of 3,000 entries, the computational time required to retrieve the search results inevitably increases for all algorithms, although at significantly different rates [10], [12]. The detailed performance trajectories illustrating this upward trend across all evaluated algorithms can be seen in the following Figure 3.

Furthermore, a deeper analysis of Figure 3 shows that execution times are clustered into three distinct performance patterns. The first pattern is speed-oriented, where the lowest execution time curves on the graph belong to the native Trie and Sequential Search. This pattern occurs because these algorithms focus strictly on exact search speed without any computational overhead for typo tolerance. The second pattern takes a balanced approach, represented by the middle-tier curves consisting of the hybrid algorithms, specifically Trie + Levenshtein Distance and Trie + Damerau-Levenshtein Distance. This cluster demonstrates an optimal compromise; they focus on both accelerating the search speed (via prefix filtering) while still maintaining the capability to tolerate typographical errors. The third pattern is tolerance-oriented, characterized by the highest execution time curves occupied by native fuzzy and vector-based algorithms, namely Levenshtein Distance, Damerau-Levenshtein Distance, Levenshtein Distance + Trie and Rocchio Relevance. This pattern emerges because these algorithms focus extensively on maximizing typo tolerance. While they successfully execute the search and correct various input errors, this robust flexibility requires heavy matrix calculations, explaining why their execution times are the highest on the chart.

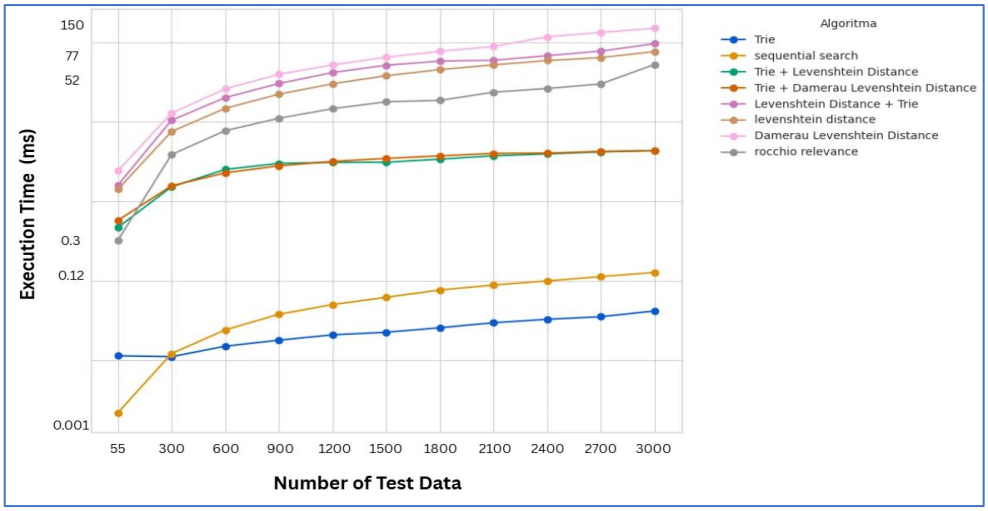


Figure 4. Performance Comparison of All Algorithms on Dataset

Based on the visualization of the runtime testing results in Figure 4, the implementation of the hybrid Trie and Levenshtein Distance architecture recorded the most optimal balanced performance. Although it is not the absolute fastest algorithm on the chart, being outperformed in raw speed by the native Trie and Sequential Search but those algorithms lack in error tolerance. In the full-scale testing involving 3,000 book title entries from Kaggle, this hybrid model was able to complete the search queries with an average execution time of 4.324 ms. The hybrid Trie and levenshtein Distance also demonstrates a highly drastic efficiency when compared to single native distance algorithms, where the native Levenshtein Distance required 76.25 ms and the Damerau-Levenshtein Distance took up to 149.73 ms. This empirical finding proves that the integration of the Trie with other algorithms can significantly contribute to pruning the search space and accelerating prefix-based computations [3]. Through this early-stage data filtering, the server load can be massively reduced, aligning with recent optimizations in real-time fuzzy matching systems [15]. Therefore, the hybrid model offers the most ideal trade-off by being exponentially faster than traditional fuzzy algorithms while successfully preserving the capability to handle user typographical errors. This flexibility becomes the weakness of the fastest algorithm.

3.2 Robust Testing

The testing results demonstrate the capability of each algorithm, ranging from those able to handle all typo cases to those entirely unable to return results. A more detailed explanation of these outcomes, along with their corresponding performance statuses, including No Recommendation Provided (NRP), Successfully Recommend (SR), and Inaccurate Recommendations (IR), can be categorized based on the algorithms' fundamental approaches.

The robust testing indicates that exact match algorithms, namely the native Trie and Sequential Search, result in No Recommendation Provided (NRP) when encountering typographical errors, as detailed in Table 1. Because these methods rely exclusively on identical prefix matching or explicit string comparisons against the database, inputs containing typos such as bok or lif3 consistently yield an NRP status. While Trie is exceptionally fast in searching, its shared inability with Sequential Search to tolerate spelling deviations makes both algorithms fundamentally unsuitable for search scenarios requiring robust error tolerance, which perfectly aligns with the prior research findings of Goenawan et al. [3] and Sjofjan et al. [1], as well as broader reviews on exact versus approximate pattern matching capabilities [17].

Conversely, the implementation of hybrid architectures demonstrated highly effective typo handling, consistently achieving a Successfully Recommend (SR) status. As shown in Table 1, the combination of Trie and Levenshtein Distance, as well as Trie coupled with Damerau Levenshtein Distance, achieved an SR outcome for diverse typo inputs such as bok, bo0k, lif3, and lofe, accurately mapping them to their intended words (book and life). The distinct advantage of these models is their ability to leverage Trie for rapid prefix based acceleration while maintaining the robust error tolerance obtained from edit distance calculations, corroborating the findings of Griffo [9]. Furthermore, executing the process in reverse by calculating the Levenshtein distance first before filtering the results using Trie yielded identical SR accuracy in correcting typos. The primary difference lies in the execution order, which makes the reverse process slightly slower than the initial combination as can be seen Figure 3., yet it remains fully reliable for error correction.

When evaluating the native fuzzy and vector based algorithms, Table 1 illustrates that the native Levenshtein Distance and Damerau Levenshtein Distance managed all typo scenarios perfectly, recording

an SR status across the board. Both algorithms ensured that misspelled words were mapped correctly as long as the edit operations did not exceed the predetermined threshold, with Damerau Levenshtein offering the added advantage of overcoming character transposition errors [10] [4]. On the other hand, the Rocchio Relevance method exhibited mixed results. While it managed to achieve an SR status for bok and bo0k based on TF-IDF vector similarity, it showed Inaccurate Recommendations (IR) for lif3 and lofe because the vector distance from the original word was too vast. As noted by Siswanto and Kanedi [11], this highlights to critical weakness where the strict reliance on word vector representation can lead to severe output discrepancies when typographical errors are extensive or rarely appear in the dataset.

Table 1. Robust Testing Results.

No	Algoritma	book → Bok	book → bo0k	life → lif3	life → lofe	Remarks
1	Trie	NRP	NRP	NRP	NRP	Cannot handle typos as it only performs exact prefix matching.
2	Sequential Search	NRP	NRP	NRP	NRP	Cannot handle typos. No results are displayed for typo inputs because this method matches strings explicitly.
3	Trie + Levenshtein Distance	SR	SR	SR	SR	Successfully provides relevant recommendations such as "book" and "life" despite typographical errors.
4	Trie + Damerau Levenshtein Distance	SR	SR	SR	SR	Successfully handles typos, providing appropriate and relevant recommendations.
5	Levenshtein Distance + Trie	SR	SR	SR	SR	Successfully provides relevant recommendations such as "book" and "life" despite typographical errors.
6	Levenshtein Distance	SR	SR	SR	SR	Successfully corrects typos. Provides correct and relevant recommendations such as "book" and "life".
7	Damerau Levenshtein Distance	SR	SR	SR	SR	Successfully corrects typos. Provides correct and relevant recommendations such as "book" and "life".
8	Rochio Relevance	SR	SR	IR	IR	Can handle certain typos (e.g., "bok", "bo0k") but inconsistently. For cases like "lif3" and "lofe", results are inaccurate as the TF-IDF vector difference is too far from the original word.

Notes: NRP = No recommendation provided, SR = Successfully Recommend, IR = Inaccurate recommendations

Based on the results of the robust testing conducted, the combination of the Trie and Levenshtein Distance algorithms is proven to be quite effective in handling typographical error variations, such as character deletion, character addition, and character substitution. This algorithm is capable of calculating the edit distance between strings so that the system can still provide relevant search results even if the user input contains typos. Furthermore, in comparison, the native Levenshtein Distance also processes faster than the Damerau-Levenshtein Distance because it lacks the additional transposition operation. Therefore, the combination of the Trie and Levenshtein Distance algorithms serves as a more effective solution compared to using only the native Levenshtein Distance or the native Damerau-Levenshtein Distance alone. This is because Trie helps accelerate the search process before Levenshtein Distance calculates the typo error tolerance, allowing the system to remain responsive while simultaneously being tolerant of user input errors.

4. CONCLUSION

The results of the testing and comparative analysis in this study confirm that the implementation of the hybrid Trie and Levenshtein Distance architecture is a highly effective and scalable computational solution for search features. Testing on a scale of 3,000 database entries proves that this hybrid model is capable of significantly reducing execution time, achieving an average of 4.324 ms. In addition to this rapid performance, the robust testing validates the model's exceptional flexibility, as it consistently achieved a Successfully Recommend (SR) status in handling and correcting various complex typographical errors. This computational success relies on the functional synergy of both architectures, where the Trie data structure plays a crucial role in

accelerating prefix-based search computation, while Levenshtein Distance ensures the system maintains high robustness in handling and correcting user typographical errors (typos). Overall, the integration of these two algorithms empirically succeeds in resolving the deadlock of information retrieval systems by balancing server speed efficiency and user experience flexibility.

As a recommendation for further development, future research is advised to evaluate this hybrid model on a much more massive scale database (big data) to measure space complexity and optimize memory consumption, which remains a main challenge in the Trie tree architecture. Furthermore, the search accuracy level can also be improved by integrating phonetic algorithms (such as Soundex or Metaphone) to handle spelling errors caused by similarities in word pronunciation sounds, or by combining it with predictive text search methods based on Natural Language Processing (NLP) to enrich the relevance of word suggestion results.

DATA AVAILABILITY STATEMENT

The data are not publicly available due to confidentiality agreements with the participating company

ACKNOWLEDGMENTS AND AI USE DISCLOSURE

During the preparation of this manuscript, the author used Gemini AI to assist with language editing, paraphrasing, and refining the text to improve readability and structural flow. The authors remains fully responsible for the accuracy, originality, validity, and integrity of all content and data presented in this manuscript.

CONFLICT OF INTEREST

The author declares that there is no conflict of interest regarding the publication of this paper.

REFERENCES

[1] M. Sjofjan, F. Fauziah and R. Aldisa, "Kombinasi Algoritma Sequential Search dan Fitur Autocomplete Pada Aplikasi Arsip (E-Arsip) Perpustakaan Berbasis Web," *Journal of Information System Research (JOSH),* vol. 4, pp. 1262-1269 , 2023 .

[2] Y. Daniati, I. Zulkarnain and K. Nurfitri, "Penerapan Algoritma Levenshtein Distance Pada Sistem Pencarian Data Buku Berbasis Web," *KOMPUTEK,* vol. 6, p. 81, 4 2022.

[3] A. D. Goenawan, A. Ammar, M. P. Pulungan and D. Komalasari, "Autocomplete Indonesian Dictionary with Trie and Depth-First Search Algorithm," *Jurnal Ilmiah Teknik Informatika dan Komunikasi,* vol. 2, pp. 99-103, 3 2022.

[4] F. W. Arsyta and R. E. Putra, "Penerapan Algoritma Damerau Levenshtein Distance Pada Pencarian Arsip Desa Jerukseger Pendukung ISO," *JINACS: Journal of Informatics and Computer Science,* vol. 4, no. 4, pp. 423-435 , 2023.

[5] N. Purnama, "Comparative Performance Study Of Search Algorithms On Large-Scale Data Structures," *JITK,* vol. 11, no. 1, pp. 99-109, 2025.

[6] S. I. Hakak, A. Kamsin, P. Shivakumara, G. A. Gilkar, W. Z. Khan and M. Imran, "Exact String Matching Algorithms: Survey, Issues, and Future Research Directions," *IEEE Access,* vol. 7, pp. 69614-69637, 2019.

[7] W. Suwarningsih and N. Nuryani, "Generate fuzzy string-matching to build self attention on Indonesian medical-chatbot," *International Journal of Electrical and Computer Engineering (IJECE),* vol. 14, p. 819, 2 2024.

[8] K. Buffardi, P. Valdivia and D. Rogers, "Measuring Unit Test Accuracy," *Proceedings of the 50th ACM Technical Symposium on Computer Science Education,* 2019.

[9] U. Griffo, "A Mixed Trie and Levenshtein distance implementation in Java for extremely fast prefix string searching and string similarity," GitHub, 2022.

[10] R. Adawiyah and N. E. Saragih, "Implementasi Algoritma Levenshtein Distance Dalam Mendeteksi Plagiarisme," *Journal Computer Science and Information Technology (JCoInT),* vol. 3, no. 2, pp. 54-63 , 2022.

[11] Siswanto and I. Kanedi, "Aplikasi Pencarian Barang Dengan Algoritma Relevance Feedback Di Toko Mega Komputer Bengkulu," *Jurnal Sains, Teknologi dan Industri,* vol. 18, p. 63, 2020.

[12] R. Zidan, K. N. U. Rehman and D. I. Khan, "Experimental Performance Benchmarking of Popular Search Algorithms in Java and Python," *International Journal of Computer Applications,* vol. 187 , no. 62 , pp. 31-38, 12 2025.

[13] A. A. Yaqin, S. E. Shamsi, K. Samadi and A. R. Rahimi, "Enhancing Text Search Accuracy in Low-Resource Languages Using N-gram and Fuzzy Matching: A Case Study on Dari," *Journal of Advanced Computer Knowledge and Algorithms,* vol. 3, no. 3, 2026.

[14] D. Primasari, F. Hakim, M. D. R. Saneistha and I. W. Cahya, "Implementation of Fuzzy Search and Audio Player for Web-Based Quranic Verse Retrieval," *Komputasi: Jurnal Ilmiah Ilmu Komputer dan Matematika,* vol. 22, no. 2, 2025.

[15] O. Rozinek, J. Marek, J. Panuš and J. Mareš, "Real-Time Fuzzy Record-Matching Similarity Metric and," *MDPI,* vol. 18, no. 3, 2025.

[16] A. R. Kaufman and A. Klevs, "Adaptive Fuzzy String Matching: How to Merge Datasets with Only One (Messy) Identifying Field," *Political Analysis,* vol. 30, pp. 590-596, 2021.

[17] Y. Zhou, F. Wang and Y. Tang, "Order-Preserving Pattern Matching: Review," *IEEE Transactions on Knowledge and Data Engineering,* vol. 38, no. 3, pp. 1885-1904, 2026.

[18] A. I. A. Gurning, Z. Zarnelly and A. Adawiyah, "Penerapan Fuzzy String Matching Pada Aplikasi Pencarian Tugas Akhir Mahasiswa Jurusan Sistem Informasi Berbasis Web (Studi Kasus: Fakultas Sains dan Teknologi UIN Suska Riau)," *Jurnal Ilmiah Rekayasa dan Manajemen Sistem Informasi,* vol. 2, no. 1, 2016.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, 4th ed., Cambridge, Massachusetts London, England: The MIT Press, 2022.

[20] V. Christanti, Rudy and D. S. Naga, "Fast and Accurate Spelling Correction Using Trie and Damerau-levenshtein Distance Bigram," *TELKOMNIKA Telecommunication, Computing, Electronics and Control,* vol. 16, no. 2, pp. 827-833, 2018.