

Evaluating search key distribution impact on searching performance in large data streams

Bowonsak Srisungsittisunti¹, Jirawat Duangkaew¹, Nakarin Chaikaew²

¹Department of Computer Engineering, School of Information and Communication Technology, University of Phayao, Phayao, Thailand

²Department of Geographic Information Science, School of Information and Communication Technology, University of Phayao, Phayao, Thailand

Article Info

Article history:

Received Oct 9, 2024

Revised Jan 29, 2025

Accepted Mar 15, 2025

Keywords:

Binary search tree

Dense indexing

Indexing efficiency

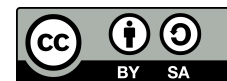
Large-scale dataset

Sparse indexing

ABSTRACT

The distribution pattern of search keys is assessed in this study by contrasting four methods of index searching on large-scale JSON files with data streams. The Adelson-Velskii and Landis (AVL) tree, binary search tree (BST), linear search (LS), and binary search (BS) are among the search strategies. We look at the normal distribution, left-skewed distribution, and right-skewed distribution of search-key distributions. According to the results, LS performs the slowest, averaging 653.166 milliseconds, whereas AVL tree performs better than the others in dense index, with an average search time of 0.005 milliseconds. With 0.011 milliseconds per keyword for sparse index, BS outperforms LS, which averages 1007.848 milliseconds. For dense indexing, an AVL tree works best; for sparse indexing, BS is recommended.

This is an open access article under the [CC BY-SA](#) license.



Corresponding Author:

Jirawat Duangkaew

Department of Computer Engineering, School of Information and Communication Technology

University of Phayao

19 Area 2 Maeka, Phayao, Thailand

Email: jirawat.du@outlook.com

1. INTRODUCTION

In our previous research [1], we introduced a method to enhance data retrieval efficiency in large-scale JavaScript Object Notation (JSON) datasets by using indexing techniques. Our approach applies dense indexing techniques for one-to-one dataset relationships and sparse indexing for one-to-many relationships, in comparison with non-indexed cases. The research was conducted in two main segments. The first involved experimenting with accessing positions within the index, and the second involved data retrieval within segments using both indexing types, showcasing the ability to skip segments to access large-scale Bigdata.json file sets. In terms of time efficiency for position access with a search key, dense indexing averaged 59.175 milliseconds, whereas non-dense indexing averaged 15,635.232 milliseconds. Sparse indexing averaged 36.387 milliseconds, while non-sparse indexing took an average of 15,635.232 milliseconds. Accessing data positions by search key from previous research [1] still follows a linear search (LS) pattern, which is efficient only when there is an index order. For this research, evaluating the impact of search key distribution on searching technique performance in large-scale JSON files that contain data streams is proposed. This evaluation compares the time efficiency of accessing data locations by search key: dense index for one-to-one data and sparse index for one-to-many data. Section 2.2 explains the approaches to applying LS, binary search (BS), binary search tree (BST), and Adelson-Velskii Landis tree (AVL) techniques. Experiments are conducted on the characteristics

of search keys in dense and sparse indexes, which have three different distribution patterns as described in section 2.1, consisting of search keys for normal distribution, left-skew distribution, and right-skew distribution. The aim is to study which technique has the highest and lowest time efficiency in accessing data locations in the large-scale JSON file set using search keys from dense and sparse indexes that have different search key distribution patterns.

In addition, we surveyed to offer a comprehensive understanding of current research in this field. Previous research involved improving index efficiency, such as through parallel indexing, hybrid methodologies, and adaptive indexing, which have the potential to significantly enhance indexing performance. They can reduce search durations, decrease storage overhead, and manage specialized data types or intricate data structures [2]–[8]. The deployment of these techniques is particularly beneficial for databases containing uncertain data, graph databases, large-scale geospatial data derived from the internet of things (IoT), and extensive time series datasets. The research section was related to databases using LS techniques. Such innovative methods as path-based index structures, online similarity searches, inverted index optimizations, dynamic compressed learned indexes, small-space algorithmic analysis, learned structures for spatial data, and distance-computation-free search schemes are emerging as innovative methods to enhance database performance. These techniques can streamline query processing, optimize storage utilization, and proficiently manage specialized data formats or complex data architectures [9]–[16]. The application of these methodologies can be especially advantageous for extensible markup language (XML) databases, uncertain time series data, relational database management systems, spatial data, and binary code databases.

Meanwhile, research related to BS, such as next-generation database indexing, decision tree algorithms tailored for spatial data, nature-inspired optimization methods, dynamic approaches in BS spaces, and random BS methodologies, offers considerable promise to advance database performance. These advancements can streamline search operations, optimize storage capabilities, and efficiently handle specialized data formats or complex data infrastructures [17]–[21]. The application of these strategies proves especially effective for progressive database systems, spatial data analysis, association rule mining, BS optimizations, and nearest neighbor searches within binary domains. On the other hand, research related to BST, such as deterministic finite tree automata minimization, space-efficient indexing tailored for cyber-physical systems, data-driven cache optimization, and B+-Tree-based search methods, presents significant advancements in database performance and security. These methodologies have the potential to optimize search operations, ensure efficient space utilization, and deliver secure and effective data retrieval in cloud environments [22]–[26]. Applying these techniques is especially advantageous for systems focused on minimizing automata, optimizing cyber-physical databases, refining cache performance in B+-Trees, ensuring encrypted cloud data security, and efficiently retrieving product information on cloud platforms.

Finally, we studied research related to the AVL tree. Such research includes secure document retrieval in encrypted cloud systems, hybrid swarm optimization for network clustering, energy-efficient cluster head selection, and ultra-scalable blockchain platforms for asset tokenization, showcasing promising advancements in their respective fields. These methods ensure data security in cloud environments, optimize network clustering and energy utilization in wireless sensor networks, and pave the way for large-scale asset management on blockchain platforms [27]–[30]. The amalgamation of these strategies offers substantial potential for systems that underscore encrypted cloud data protection, streamlined clustering in wireless infrastructures, and extensible blockchain mechanisms for global assets.

2. METHOD

In previous research [1], we proposed dense indexing for one-to-one data access, as illustrated in Figure 1. The file set is called `Dense_index_position.json`, with a size of 33 MB and containing 1,000,000 transactions. The file set consists of search keys and specific positions within the `Bigdata.json` dataset. While the sparse index for accessing one-to-many data is illustrated in Figure 2, the set of files is called `Sparse_index_position.json`, with a file size of 8 MB and containing 5,146 transactions. This file set consists of search keys and multiple positions within the `Bigdata.json` dataset.

In searching to access data positions using the dense and sparse indexes, no issues arise when searching for a single search key. However, when searching for multiple search keys, the search key distribution must be considered. Therefore, this research proposes an approach to compare the efficiency of access times to data positions using dense and sparse indexes. The researcher selected main search keys from both index datasets

as described in section 2.1, which have three different search key distributions: normal distribution, left-skew distribution, and right-skew distribution. The objective is to compare the efficiency of accessing positions using the main search keys within both index files. The researcher applied four techniques as explained in section 2.2, including LS, BS, BST, and AVL tree. The time efficiency of each technique was considered to determine which provided the fastest and slowest results for accessing positions within both index datasets.

```
"Phillip.Headlon@smith.com": 0,
"Jonnie@pacificare.com": 1,
...
"Annalee@pinnaclewest.com": 1000000
```

Figure 1. Examples of transactions in the Dense_index_position.json with a size of 33 MB and 1,000,000 transactions

```
"Maryalice": [0, 547, 836317, 1040693, 1041786],
"Luciana": [1, 3639, 9282, 1041132, 1044174],
...
"Eulah": [2, 1216, 630028, 630264, 1042683]
```

Figure 2. Examples of transactions in the Sparse_index_position.json with a size of 8 MB and 5,146 transactions

2.1. The selection of search keys for dense and sparse distribution

This section explains the selection of 524 search keys to evaluate the effectiveness of the research. We obtained these search keys from the dense index position file, which is 33 MB and has 1,000,000 transactions, and the sparse index position file, which is 8 MB and has 5,146 transactions. For the dense index, we created five email index files, each containing 524 non-repeating search keys, and divided these index files into three main types according to position range. The first type is the dense index for the normal distribution of the index, which has 2,070 search keys in the position range of 300,000 to 800,000, with 550 search keys in the left-skewed distribution and right-skewed distribution, as illustrated in Figure 3.

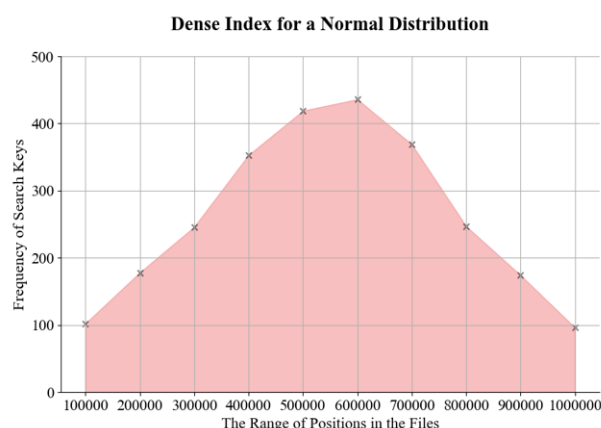


Figure 3. Dense index for normal distribution

The second type of index is a dense index for the left-skewed distribution. As illustrated in Figure 4, the index is distributed within the range of 100,000 to 500,000 with 1,806 search keys. Moreover, the distribution on the right side holds 814 search keys.

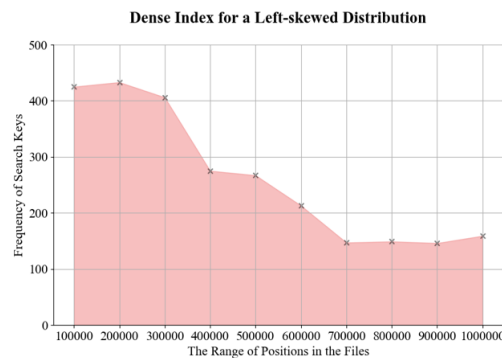


Figure 4. Dense index for a left skewed distribution

The third type of index is a dense index for a right-skewed distribution. As illustrated in Figure 5, the index is distributed within the range of 500,000 to 1,000,000, encompassing 1,999 search keys. The distribution on the left side contains 621 search keys.

For the sparse index, we created five email index files, each containing 524 non-repeating search keys. These index files were divided into three main types according to position range. The first type is the sparse index for normal distribution of the index, which has 2,600 search keys in the 2,000 to 4,000 position range, with 20 search keys in the left- and right-skewed distribution, as illustrated in Figure 6.

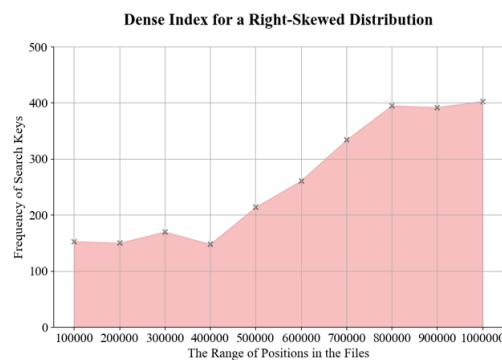


Figure 5. Dense index for a right skewed distribution

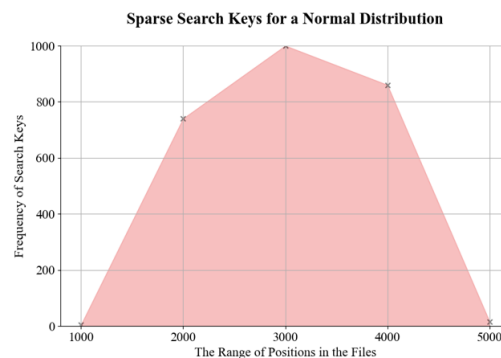


Figure 6. Sparse index for normal distribution

The second type of index is a sparse index for the left-skewed distribution. As illustrated in Figure 7, the index is distributed within the range of 1,000 to 3,000 with 1,885 search keys. Moreover, the distribution on the right side holds 735 search keys.

The third type of index is a sparse index for a right-skewed distribution. As illustrated in Figure 8, the index is distributed within the range of 3,000 to 5,000, encompassing 1,916 search keys. Moreover, the distribution on the left side contains 704 search keys.

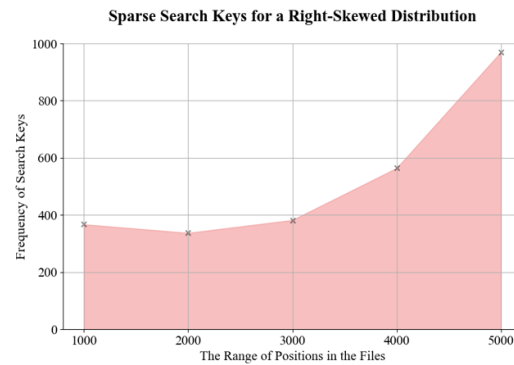
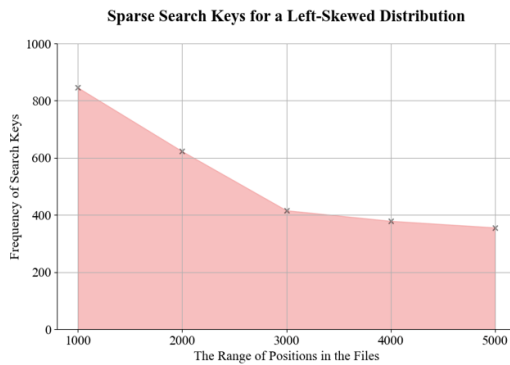


Figure 7. Sparse index for left skewed distribution Figure 8. Sparse index for right skewed distribution

2.2. Algorithms for comparing the efficiency of dense and sparse indices

This section explains the methods for searching and accessing dense and sparse index position files, which comprise LS, BS, BST, and AVL tree. The details of these algorithms are provided as follows.

2.2.1. Dense and sparse index algorithms using linear search

For the method of searching the position of search key terms within dense and sparse index files using the LS algorithm, three different distributions of test search key terms were used, as detailed in section 2.1. The search starts by checking the search key term in each transaction entry within the dense index file. If the searched search key term matches the search key term appearing in the file, the position of that search key term can be retrieved. If not, the search continues until the desired key is found or until all entries are exhausted.

Algorithm 1 Dense and sparse index algorithms using LS

Require: Dense and Sparse Index position File, search keys test with distribution as described in section 2.1

Ensure: Position in Bigdata.json

```

1: function FINDSEARCHKEY(index.positions, desired.search.keys)
2:   result ← {}
3:   for search.key ∈ desired.search.keys do
4:     position.found ← False
5:     for index.entry, index.entry ∈ Enumerate(index.positions) do
6:       if index.entry["search.key"] == search.key then
7:         result[search.key] ← index.entry["position"]
8:         position.found ← True break
9:       end if
10:    end for
11:    if not position.found then
12:      result[search.key] ← "Search key not found."
13:    end if
14:  end for
15:  return result
16: end function

```

For the utilization of dense and sparse index algorithms using LS to access the position of search keys within dense and sparse index position files, import test search keys with all distribution patterns as described in section 2.1. Import one search key at a time into the search verification process. This process uses the LS technique as explained in section 2.2.1.

2.2.2. Dense and sparse index algorithms using binary search

When searching for the position of search key terms in dense and sparse index files using the BS technique, three search key term distribution patterns were tested, as described in section 2.1. The search process begins by sorting the search key terms in the dense and sparse index files. Test search key terms with different distributions are then used to find the midpoint of the search key term sequence. The midpoint is compared to the search value. If the search key term is found, the position of the search key term is returned. If the search key term is not found, the data sequence is divided into two parts, and the search continues in the part with search key term values closest to the desired search key term.

Algorithm 2 Dense and sparse index algorithms using binary search

```

1: Input: Dense and Sparse Index position File, search keys test with distribution as described in section 2.1
2: Output: Position in Bigdata.json
3: function BINARY_SEARCH(index_file, desired_search_key)
4:   low  $\leftarrow$  0
5:   high  $\leftarrow$  len(index_file) - 1
6:   while low  $\leq$  high do
7:     mid  $\leftarrow$  (high + low)//2
8:     mid_search_key  $\leftarrow$  index_file[mid][0]
9:     if mid_search_key == desired_search_key then
10:      return index_file[mid][1]
11:    else if mid_search_key < desired_search_key then
12:      low  $\leftarrow$  mid + 1
13:    else
14:      high  $\leftarrow$  mid - 1
15:    end if
16:  end while
17:  return "Search key not found."
18: end function

```

For the utilization of dense and sparse index algorithms using BS to access the position of search keys within dense and sparse index position files, import test search keys with six different distribution patterns as described in section 2.1. Each search key is processed through the search verification process using the BS technique, as explained in section 2.2.2. The efficiency of accessing the search key position based on the average or worst-case data range is $O(\log n)$, but the best-case scenario is $O(1)$ when the test search key matches the index position files, returning the position of Bigdata.json.

2.2.3. Dense and sparse index algorithms using BST and AVL Tree

This section explains the method of searching and accessing positions within dense and sparse index files using both the BST and the AVL tree techniques. The algorithms for these methods are illustrated in Algorithm 3, which integrates both BST and AVL tree operations for efficiency. Six different test search key distribution patterns are used, as described in section 2.1. The search process begins by arranging the main words in the index file. Test search keys are then used to search at the root node of either the BST or the AVL tree, depending on the desired efficiency. If the root node is empty, the process returns none. For both tree structures, the search proceeds by comparing the desired search key with the current node's search keys in the dense index position file. If the search key matches the current node, the position of the main word is returned, and the search for the next search key continues. Otherwise, the direction of traversal (left or right) is determined based on whether the search key is less or greater than the current node's search key.

For the utilization of dense and sparse index algorithms using BST or AVL trees to access the position of search keys within dense and sparse index position files, the test search keys are processed with both techniques, depending on efficiency requirements. The BST technique, as described in section 2.2.3, allows searching with average or worst time complexity equal to $O(n)$. On the other hand, the AVL tree technique, as described in section 2.2.4, offers an improved average and worst-case time complexity of $O(\log n)$. The best case for both is $O(1)$ when the search key directly matches an index in the file, returning the position in Bigdata.json.

Algorithm 3 Dense and sparse index algorithms using BST and AVL Tree

```

1: Input: Dense and Sparse Index position File, search keys test with distribution as described in section 2.1
2: Output: Position in Bigdata.json
3: function INDEX_SEARCH(index_file, search_keys, tree_type)
4:   if tree_type == "BST" then
5:     tree ← new BinarySearchTree()
6:   else if tree_type == "AVL" then
7:     tree ← new AVLTree()
8:   else
9:     raise error: Unsupported tree type.
10:  end if
11:  for (search_key, position) ∈ index_file do
12:    tree.insert(search_key, position)
13:  end for
14:  result ← {}
15:  for desired_search_key ∈ search_keys do
16:    position ← tree.search(desired_search_key)
17:    if position ≠ NULL then
18:      result[desired_search_key] ← position
19:    else
20:      result[desired_search_key] ← "Search key not found."
21:    end if
22:  end for
23:  return result
24: end function

```

3. RESULTS

This section presents the evaluation of search time efficiency for accessing data positions in two datasets: dense_index_position.json and sparse_index_position.json. The search algorithms employed for the analysis include LS, BS, BST, and AVL tree. The experiments were conducted with three distinct search key distribution patterns: normal, left-skewed, and right-skewed distributions, as detailed in section 2.1. The comparative experiments focused on two types of indexes: dense index as shown in Table 1 and sparse index as shown in Table 2. For both index types, the experiments measured two primary characteristics: 'Average time of 15 rounds,' which assessed the total time required to access positions for 524 search keys over fifteen rounds, and 'Average time per search key,' which evaluated the search time required to access each search key individually. The results showed that sorting and indexing are necessary for BS, BST, and AVL Tree to enhance performance. For LS, there was no need to sort the index beforehand, as LS sequentially evaluates all transaction items, regardless of search key distribution. The results for the dense index (Table 1) demonstrate that BS had an average indexing time of 1,795.708 milliseconds, while BST completed indexing in 773.149 milliseconds, and AVL Tree achieved the fastest time of 635.262 milliseconds.

Table 1. Comparison of dense index searching efficiency with search key distribution

Techniques	Retrieve time by dense index (ms)	Avg. time of 15 rounds	Avg. per search key	Index file sorting time
Linear search	Normal distribution	342259.155	653.166	No index file sorting
	Left-skewed distribution	337220.625	643.551	
	Right-skewed distribution	349457.130	666.903	
Binary search	Normal distribution	9.811	0.019	1795.708
	Left-skewed distribution	10.199	0.019	
	Right-skewed distribution	11.572	0.022	
Binary search tree	Normal distribution	179.970	0.343	773.149
	Left-skewed distribution	173.583	0.331	
	Right-skewed distribution	155.317	0.296	
AVL Tree	Normal distribution	2.703	0.005	635.262
	Left-skewed distribution	2.346	0.004	
	Right-skewed distribution	2.433	0.005	

The first experiment evaluated the efficiency of accessing data positions in a dense index. Each test round included 524 search keys, and the results were averaged over 15 rounds. When tested with search keys following a normal distribution, the AVL tree was the fastest, with an average access time of 2.703 milliseconds.

On the other hand, LS was the slowest, taking 342,259.155 milliseconds. For search keys with a left-skewed distribution, the AVL tree performed the quickest, with an average time of 2.346 milliseconds, while LS was again the slowest, taking 337,220.625 milliseconds. In right-skewed search key distributions, the AVL tree was the fastest, with an average time of 2.433 milliseconds, whereas LS remained the slowest at 349,457.13 milliseconds.

The second experiment assessed search efficiency in a sparse index. Each round also consisted of 524 search keys, and the results were averaged over 15 rounds. With a normal search key distribution, BS was the fastest, with an average time of 5.647 milliseconds, while LS was the slowest, taking 528,112.194 milliseconds. For left-skewed search key distributions, BS was the quickest, averaging 3.868 milliseconds, while LS was the slowest, taking 528,112.194 milliseconds. With right-skewed search key distributions, BS was the fastest, averaging 5.592 milliseconds, while LS remained the slowest at 528,179.767 milliseconds.

Table 2. Comparison of sparse index searching efficiency with search key distribution

Techniques	Retrieve time by sparse index (ms)	Avg. time of 15 rounds	Avg. per search key	Index file sorting time
Linear search	Normal distribution	528112.194	1007.848	No index file sorting
	Left-skewed distribution	528253.244	1008.117	
	Right-skewed distribution	528179.767	1007.977	
Binary search	Normal distribution	5.647	0.011	136.711
	Left-skewed distribution	3.868	0.007	
	Right-skewed distribution	5.592	0.011	
Binary search tree	Normal distribution	161.007	0.307	No index file sorting
	Left-skewed distribution	166.118	0.317	
	Right-skewed distribution	158.378	0.302	
AVL tree	Normal distribution	30.903	0.059	113.538
	Left-skewed distribution	35.200	0.067	
	Right-skewed distribution	35.223	0.067	

4. CONCLUSION

This research compares the time efficiency of accessing positions using both dense and sparse indexes. The study applied four search techniques: LS, BS, BST, and AVL tree, across three search key distribution patterns: normal, left-skewed, and right-skewed. The experimental findings indicate that for dense indexes with a normal distribution of search keys, the AVL Tree was the fastest, averaging 0.005 milliseconds per search key. Conversely, LS was the slowest, with an average of 653.166 milliseconds per search key. For left-skewed distributions, the AVL Tree was again the quickest, averaging 0.004 milliseconds per search key, while LS was the slowest at 643.551 milliseconds. The pattern was consistent for right-skewed distributions, with the AVL Tree being the fastest at 0.005 milliseconds and LS being the slowest at 666.903 milliseconds. For sparse indexes, the results were slightly different. When search keys were normally distributed, BS was the most efficient, averaging 0.011 milliseconds per search key, while LS was the slowest, taking 1007.848 milliseconds. For left-skewed distributions, BS remained the fastest, averaging 0.007 milliseconds per search key, with LS being the slowest at 1008.117 milliseconds. Similarly, for right-skewed distributions, BS was the quickest at 0.011 milliseconds per search key, and LS was the slowest, averaging 1007.977 milliseconds. In conclusion, the AVL Tree is highly efficient for dense indexes, particularly when handling large volumes of search keys, such as 1,000,000 transactions. On the other hand, BS is more suitable for sparse indexes with a smaller number of search keys, such as 5,000 transactions. LS consistently showed the least efficiency in both dense and sparse index scenarios. Future research can explore secondary indexing techniques to enhance scalability and efficiency for more complex data types and datasets.

ACKNOWLEDGMENTS

The authors thank Research Unit for Development of Intelligent Systems and Autonomous Robots (ISAR) and School of Information Technology and Communication, University of Phayao, for facility support.

FUNDING INFORMATION

This work was supported by the Super KPI project for international publications, under the School of Information and Communication Technology, University of Phayao.

AUTHOR CONTRIBUTIONS STATEMENT

This journal uses the Contributor Roles Taxonomy (CRediT) to recognize individual author contributions, reduce authorship disputes, and facilitate collaboration.

Name of Author	C	M	So	Va	Fo	I	R	D	O	E	Vi	Su	P	Fu
Bowonsak Srisungsittisunti	✓	✓	✓	✓	✓	✓		✓	✓	✓			✓	
Jirawat Duangkaew		✓	✓			✓	✓	✓	✓	✓	✓	✓		
Nakarin Chaikaew	✓			✓		✓			✓		✓		✓	✓

C : Conceptualization

M : Methodology

So : Software

Va : Validation

Fo : Formal Analysis

I : Investigation

R : Resources

D : Data Curation

O : Writing - Original Draft

E : Writing - Review & Editing

Vi : Visualization

Su : Supervision

P : Project Administration

Fu : Funding Acquisition

CONFLICT OF INTEREST STATEMENT

The authors state no conflict of interest.

DATA AVAILABILITY

Derived data supporting the findings of this study are available from the corresponding author, [JD], on request.




REFERENCES

- [1] B. Srisungsittisunti, J. Duangkaew, S. Mekruksavanich, N. Chaikaew, and P. Rojanavas, "Enhancing data retrieval efficiency in large-scale JavaScript object notation datasets by using indexing techniques," *IAES International Journal of Artificial Intelligence (IJ-AI)*, vol. 13, no. 2, pp. 2342–2353, 2024, doi:10.11591/ijai.v13.i2.pp2342-2353.
- [2] R. Qiu, Y. Ming, Y. Hong, H. Li, and T. Yang, "Wind-bell index: towards ultra-fast edge query for graph databases," *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, Apr. 2023, pp. 2090–2098, doi: 10.1109/icde55515.2023.00162.
- [3] H. Wu *et al.*, "A performance-improved and storage-efficient secondary index for big data processing," *2017 IEEE International Conference on Smart Cloud (SmartCloud)*, Nov. 2017, pp. 161–167, doi: 10.1109/smartcloud.2017.32.
- [4] R. Ma, X. Zhu, and L. Yan, "A hybrid approach for clustering uncertain time series," *Journal of Computing and Information Technology*, vol. 28, no. 4, pp. 255–267, Oct. 2021, doi: 10.20532/cit.2020.1004802.
- [5] S. V. Limkar and R. K. Jha, "A novel method for parallel indexing of real time geospatial big data generated by IoT devices," *Future Generation Computer Systems*, vol. 97, pp. 433–452, Aug. 2019, doi: 10.1016/j.future.2018.09.061.
- [6] M. A. Qader, S. Cheng, and V. Hristidis, "A comparative study of secondary indexing techniques in LSM-based NoSQL databases," *Proceedings of the 2018 International Conference on Management of Data*, May 2018, pp. 551–566, doi: 10.1145/3183713.3196900.
- [7] Z. Wang, Q. Wang, P. Wang, T. Palpanas, and W. Wang, "Dumpy: A compact and adaptive index for large data series collections," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–27, May 2023, doi: 10.1145/3588965.
- [8] M. M. Lawal, H. Ibrahim, N. F. M. Sani, and R. Yaakob, "Analyses of indexing techniques on uncertain data with high dimensionality," *IEEE Access*, vol. 8, pp. 74101–74117, 2020, doi: 10.1109/access.2020.2988487.
- [9] D. Gopinathan and K. Asawa, "New path based index structure for processing CAS queries over XML database," *Journal of Computing and Information Technology*, vol. 25, no. 3, pp. 211–225, Oct. 2017, doi: 10.20532/cit.2017.1003557.
- [10] R. Ma, D. Zheng, and L. Yan, "Fast online similarity search for uncertain time series," *Journal of Computing and Information Technology*, vol. 28, no. 1, pp. 1–17, Jul. 2020, doi: 10.20532/cit.2020.1004574.
- [11] Y. Shin, J. Ahn, and D.-H. Im, "Join optimization for inverted index technique on relational database management systems," *Expert Systems with Applications*, vol. 198, Jul. 2022, doi: 10.1016/j.eswa.2022.116956.
- [12] P. Ferragina and G. Vinciguerra, "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1162–1175, 2020, doi: 10.14778/3389133.3389135.
- [13] D. Belazzougui *et al.*, "Linear-time string indexing and analysis in small space," *ACM Transactions on Algorithms (TALG)*, vol. 16, no. 2, pp. 1–54, 2020, doi: 10.1145/3381417.
- [14] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "LISA: A learned index structure for spatial data," *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, May 2020, pp. 2119–2133, doi: 10.1145/3318464.3389703.
- [15] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," *Proceedings of the 2018 International Conference on Management of Data*, May 2018, pp. 489–504, doi: 10.1145/3183713.3196909.
- [16] J. Song, H. T. Shen, J. Wang, Z. Huang, N. Sebe, and J. Wang, "A distance-computation-free search scheme for binary code databases," *IEEE Transactions on Multimedia*, vol. 18, no. 3, pp. 484–495, Mar. 2016, doi: 10.1109/tmm.2016.2515990.
- [17] J. Dittrich, J. Nix, and C. Schön, "The next 50 years in database indexing or: the case for automatically generated index structures," *Proceedings of the VLDB Endowment*, vol. 15, no. 3, pp. 527–540, Nov. 2021, doi: 10.14778/3494124.3494136.




- [18] S. Oujdi, H. Belbachir, and F. Boufares, "C4.5 decision tree algorithm for spatial data, alternatives and performances," *Journal of Computing and Information Technology*, vol. 27, no. 3, pp. 29–43, May 2020, doi: 10.20532/cit.2019.1004651.
- [19] Y. Gheraibia, A. Moussaoui, S. Kabir, and P. Y. Yin, "Penguins search optimisation algorithm for association rules mining," *Journal of Computing and Information Technology*, vol. 24, no. 2, pp. 165–179, Jun. 2016, doi: 10.20532/cit.2016.1002745.
- [20] A. Baykasoğlu and F. B. Ozsoydan, "Dynamic optimization in binary search spaces via weighted superposition attraction algorithm," *Expert Systems with Applications*, vol. 96, pp. 157–174, Apr. 2018, doi: 10.1016/j.eswa.2017.11.048.
- [21] M. Komorowski and T. Trzciński, "Random binary search trees for approximate nearest neighbour search in binary spaces," *Applied Soft Computing*, vol. 79, pp. 87–93, Jun. 2019, doi: 10.1016/j.asoc.2019.03.031.
- [22] Y. Guellouma and H. Cherroun, "Efficient implementation for deterministic finite tree automata minimization," *Journal of Computing and Information Technology*, vol. 24, no. 4, pp. 311–322, Dec. 2016, doi: 10.20532/cit.2016.1002867.
- [23] Y.-H. Kuan, Y.-H. Chang, T.-Y. Chen, P.-C. Huang, and K.-Y. Lam, "Space-efficient index scheme for PCM-based multiversion databases in cyber-physical systems," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 1, pp. 1–26, Oct. 2016, doi: 10.1145/2950060.
- [24] R. Kühn, D. Biebert, C. Hakert, J.-J. Chen, and J. Teubner, "Towards data-based cache optimization of B+-trees," *Proceedings of the 19th International Workshop on Data Management on New Hardware*, Jun. 2023, pp. 63–69, doi: 10.1145/3592980.3595316.
- [25] H. Shen, L. Xue, H. Wang, L. Zhang, and J. Zhang, "B+-tree based multi-keyword ranked similarity search scheme over encrypted cloud data," *IEEE Access*, vol. 9, pp. 150865–150877, 2021, doi: 10.1109/access.2021.3125729.
- [26] Y.-S. Zhao and Q.-A. Zeng, "Secure and efficient product information retrieval in cloud computing," *IEEE Access*, vol. 6, pp. 14747–14754, 2018, doi: 10.1109/access.2018.2816919.
- [27] J. Fu, N. Wang, B. Cui, and B. K. Bhargava, "A practical framework for secure document retrieval in encrypted cloud file systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 5, pp. 1246–1261, May 2022, doi: 10.1109/tpds.2021.3107752.
- [28] R. M. Alamelu and K. Prabu, "Hybridization of Pigeon inspired with glowworm' swarm optimization based clustering technique in wireless sensor networks," *Microprocessors and Microsystems*, vol. 91, Jun. 2022, doi: 10.1016/j.micpro.2022.104528.
- [29] R. R. Priyadarshini and N. Sivakumar, "Cluster head selection based on minimum connected dominating set and Bi-partite inspired methodology for energy conservation in WSNs," *Journal of King Saud University - Computer and Information Sciences*, vol. 33, no. 9, pp. 1132–1144, Nov. 2021, doi: 10.1016/j.jksuci.2018.08.009.
- [30] A. Buldas *et al.*, "An ultra-scalable blockchain platform for universal asset tokenization: design and implementation," *IEEE Access*, vol. 10, pp. 77284–77322, 2022, doi: 10.1109/access.2022.3192837.

BIOGRAPHIES OF AUTHORS






Dr. Bowonsak Srisungsittisunti    is Assistant Professor at Computer Engineering, School of Information and Communication technology, University of Phayao, Thailand. He holds a Ph.D. degree in computer engineering with specialization in data processing. His research areas are data processing, data analytic, data mining, and database system. He can be contacted at email: bowonsak.sr@up.ac.th.



Jirawat Duangkaew    received a Bachelor of Science degree in computer science from Rambhai Barni Rajabhat University, Thailand, in 2020, and completed his Master of Engineering in computer engineering at the University of Phayao, Thailand, in 2024. He is currently pursuing his Ph.D. in computer engineering at the University of Phayao, Thailand. His research interests include indexing techniques, non-relational databases, large databases, and incremental databases. He can be contacted at email: jirawat.du@outlook.com.



Nakarin Chaikaew    is Ph.D. in remote sensing and geographic information systems, Asian Institute of Technology (AIT). He is Assistant Professor in geographic information science, University of Phayao, Thailand. He can be contacted at email: nakarin.ch@up.ac.th.