

Enhancing data retrieval efficiency in large-scale javascript object notation datasets by using indexing techniques

Bowonsak Srisungsittisunti¹, Jirawat Duangkaew¹, Sakorn Mekruksavanich¹, Nakarin Chaikaew²,
Pornthep Rojanavasu¹

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 Apr 18, 2023

Revised Sep 29, 2023

Accepted Oct 8, 2023

Keywords:

Dense indexing

Javascript object notation

Large scale dataset

Not only structured query

language

Sparse indexing

ABSTRACT

The use of javascript object notation (JSON) format as a not only structured query language (NoSQL) storage solution has grown in popularity, but has presented technical challenges, particularly in indexing large-scale JSON files. This has resulted in slow data retrieval, especially for larger datasets. In this study, we propose the use of JSON datasets to preserve data in resource survey processes. We conducted experiments on a 32-gigabyte dataset containing 1,000,000 transactions in JSON format and implemented two indexing methods, dense and sparse, to improve retrieval efficiency. Additionally, we determined the optimal range of segment sizes for the indexing methods. Our findings revealed that adopting dense indexing reduced data retrieval time from 15,635 milliseconds to 55 milliseconds in one-to-one data retrieval, and from 38,300 milliseconds to 1 millisecond in the absence of keywords. In contrast, using sparse indexing reduced data retrieval time from 33,726 milliseconds to 36 milliseconds in one-to-many data retrieval and from 47,203 milliseconds to 0.17 milliseconds when keywords were not found. Furthermore, we discovered that the optimal segment size range was between 20,000 and 200,000 transactions for both dense and sparse indexing.

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: 64024804@up.ac.th

1. INTRODUCTION

In the era of data explosion, the demand for javascript object notation (JSON) files as a not only structured query language (NoSQL) storage solution has surged. However, this growth introduces several technical challenges, especially regarding indexing techniques for large-scale JSON datasets. Since JSON files data often exceeds random access memory's (RAM) capacity, it is infeasible to query all data in a single access. Instead, an equal division of the data is necessary, with each segment queued in RAM for processing by the central processing unit (CPU), as illustrated in Figure 1. This research posits that retrieving for data by key eliminates the need to load all segments into RAM, thereby improving processing efficiency. Furthermore, the index file can help identify the segment containing data that matches the retrieve key, enabling the skipping of some segments when loading data into RAM and ultimately boosting performance.

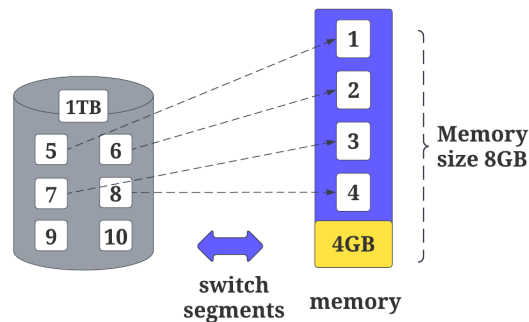


Figure 1. The case of no index being made for a large dataset

This study employs the JSON format to investigate indexing techniques for large-scale datasets in NoSQL systems. A review of related studies offers a comprehensive understanding of the state-of-the-art research in this area. Prior research on indexing has explored numerous applications, such as improving index efficiency. Chang *et al.* [1] proposed a keyword-index-based metadata search method for large-scale file systems, leveraging spatial locality and load characteristics. The proposed partitioning technique offers improved efficiency compared to non-partitioning methods. Chopade and Pachghare [2] proposed indexing on MongoDB's query performance and relevance in database forensics. The study highlights how indexing can significantly enhance efficiency by reducing document scan time, making it a crucial aspect of application development and forensic analysis. Yuan and Liu [3] proposed a two-level embedded k-Means algorithm for approximate nearest neighbor search, using multi-assignment operations and pruning strategies for efficiency. Experiments show improved performance, lower memory usage, and complexity compared to state-of-the-art approaches. Zineddine *et al.* [4] proposed a novel binary tree-based indexing structure to organize image properties (color, shape, and texture) for efficient large-scale image search. It introduces index construction and k-nearest neighbors (KNN) search algorithms, utilizing container concepts to enhance complexity performance evaluated on real data sets. Jin *et al.* [5] proposed adapting balanced tree plus (B+tree) and log-structured merge-tree (LSM tree) index structures for zoned namespaces (ZNS) solid state drives (SSDs), which have lower overhead and over-provisioning costs but only accept sequential writes.

Abdulkadhem and Assadi [6] proposed a geographic information system (GIS) based method for constructing important road landmarks using corner points and multimedia data. As a preprocessing step for roadmap discovery from video films, it aims to enable multimedia queries within the GIS environment. In addition, we have also studied research related to applying indexes. Alqatawneh [7] proposed a novel technique for orthogonal frequency-division multiplexing (OFDM) systems using zero-pilot symbols to transmit extra data bits without affecting channel estimation accuracy or system error performance. Minimum mean square error (MMSE) based detection is employed, and simulation results show improved system throughput and a lower error rate at high SNR. Mouneshachari and Pande [8] proposed an electroencephalogram (EEG) based emotion quantification index using KNN classification, enabling analysis of similarity and dissimilarity between individual signals and benchmark data. The results demonstrate the potential for understanding basic emotional feelings in individuals. Zeffora and Shobarani [9] proposed an adaptive attribute selection method for the random forest, considering structural changes in datasets. Applied to myocardial infarction data, it improves accuracy and avoids under/over-fitting. Tan and Lim [10] proposed a wireless fidelity (WiFi) sniffer-enabled surveillance camera system with a 3-stage WiFi frame inspection filter, using WiFi signal strength for filtering and tagging media access control (MAC) address to video frames. This technique leverages metadata (smartphone MAC addresses) to prioritize video frame retrieval, reducing manual search efforts in public safety surveillance. Abdulsada *et al.* [11] proposed a practical multi-keyword similarity scheme for searchable encryption using compressed trapdoors generated via key-based hash functions. The scheme computes similarity scores through hamming distances and offers improved search efficiency and performance compared to existing methods while maintaining robust security. Finally, we also studied the relevant research on large-scale data [12]-[29], exploring various techniques and applications in data retrieval, mining, and processing. Essential techniques include in-memory distributed indexing, internet of things (IoT), indexing and discovery, text mining, data clustering, and region-of-interest-based image retrieval. These studies improve data management, information extraction, and knowledge discovery in diverse domains such as media, internet of things (IoT), health, energy, and supply chain management, enabling more efficient and adequate decision-making in large-scale data environments.

The hypothesis of this research involves creating a dense index for one-to-one retrieves and a sparse index for one-to-many retrieves. This experiment aims to investigate how these indexing techniques can reduce

retrieving time. Additionally, we will explore the best segment size for efficient retrieving. The design process incorporated elements of relational dataset techniques, and we developed the research commands using Python. The results proved that the JSON files indexing technique for large scale datasets significantly improved data retrieval time.

2. METHOD

2.1. Simulating large-scale dataset in JavaScript object notation files

In this study, we generated a dataset named Bigdata.json, which consists of 32 GB and 1,000,000 transactions. The dataset is formatted as a JSON file and includes fields such as First_name, Last_name, email, and hex data. These fields originate from transforming picture files into hexadecimal representations. The dataset's flexibility enables customization to meet specific research requirements. For instance, Figure 2 illustrates some sample transactions from the dataset.

```
{
  "position": 1,
  "First_name": "Luciana",
  "Last_name": "Rodden",
  "email": "Luciana.Rod@example.com",
  "hex": "ffd8ffe000104a4649460001010100480048000..." (32MB)
}, ... ,
{
  "position": 1000000,
  "First_name": "Maryalice",
  "Last_name": "Kosareff",
  "email": "Maryalice.Kos@example.com",
  "hex": "ffd8ffe000104a4649460001010100480048000..." (32MB)
}
```

Figure 2. Example transactions in JSON with a size of 32 GB and 1,000,000 transactions

2.2. Algorithm for retrieving without an index

When handling large datasets in the form of NoSQL JSON files without indexing, the dataset, called Bigdata.json, is 32 GB and contains 1,000,000 transactions, as detailed in subsection 2.1 of this study. Importing all data into RAM simultaneously is impossible and challenging, making accessing and finding information problematic. This study presents a technique for retrieving and accessing information without indexing to address this issue. The pseudocode for retrieving without an index is displayed in Figure 3 and illustrated by a flowchart in Figure 4. This technique involves dividing the data from Bigdata.json into ten segments, each containing 3.2 GB and 100,000 transactions. It is important to note that during the process of segmenting Bigdata.json, the procedure does not involve splitting the file into ten separate files. Instead, it involves loading individually divided segments, one at a time, into the RAM. Afterward, the system retrieves information by retrieving each segment within the RAM. If the data matches the specified keyword, the system extracts and displays it after retrieving all segments. The retrieve covers all Bigdata.json segments. Although this technique helps retrieve and access data, it has significant drawbacks, such as consuming considerable RAM resources and processing power, leading to substantial delays.

1. Import Bigdata.json and specify the keyword to search for.
2. Divide Bigdata.json (size of 32 GB) into 10 segments.
3. For each segment (i = 1 to 10):
 - A. Load the segment i into RAM.
 - B. Search for specified keyword in segment i.
 - C. Retrieve the relevant data from segment i.
4. Display final search result

Figure 3. Pseudocode of retrieving without an index

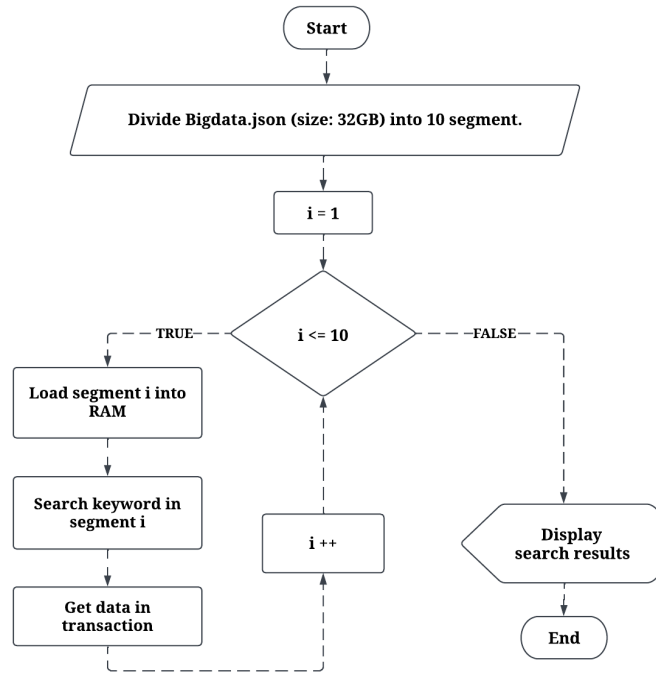


Figure 4. Flowchart of retrieving without an index

2.3. Algorithm for retrieving by index

This research presents an algorithm that enhances the time efficiency of retrieving and accessing data in large-scale NoSQL datasets by applying dense and sparse indices from relational databases to the Bigdata.json file. Figure 5 displays the pseudocode for retrieving by the index and Figure 6 illustrates the flowchart. The algorithm, optimized and divided into dense and sparse indexing for NoSQL datasets sized at 32 GB, includes keywords and positions to enable efficient referencing from RAM to the data in the Bigdata.json file during a query using the retrieve-by-keyword technique. The command imports the large-scale dataset, Bigdata.json, which is 32 GB and contains 1,000,000 transactions. The Bigdata.json file is divided into ten segments, each containing 100,000 transactions. We designate the first segment to correspond to positions of transactions 1 to 100,000, the second segment to positions of transactions 100,001 to 200,000, until the tenth segment represents positions of 900,001 to 1,000,000. Next, the algorithm checks the position of the transaction in each segment against the index file. If the position falls within the segment, the algorithm loads the segment into RAM and retrieves it using the keyword to find data matching the specified keyword. Upon finding a match, the algorithm retrieves and displays the data within the segment as the retrieved result.

The size of segments utilized in the algorithm may be altered based on their appropriateness for a given dataset. The presented pseudocode serves as an illustrative example of one such configuration, assuming a segment size of 100,000 transactions. However, the experimentally determined optimal segment size will be reported in result and discussion section. To ensure the validity of our findings, we have provided detailed methodology and references supporting our approach.

1. Import files index_position.json, Bigdata.json and Keyword
2. Divide Bigdata.json (size of 32 GB) into 10 segments.
3. Search for key in index position.json to get position of Bigdata.json
4. For segment $i = 1$ to 10
 - A. If position is in segment i .
 - A1. Load segment i into RAM
 - A2. Search position in segment i
 - A3. Retrieve the relevant data from segment i .
 - B. Else, skip segment i .
5. Display search result

Figure 5. Pseudocode of retrieving by index

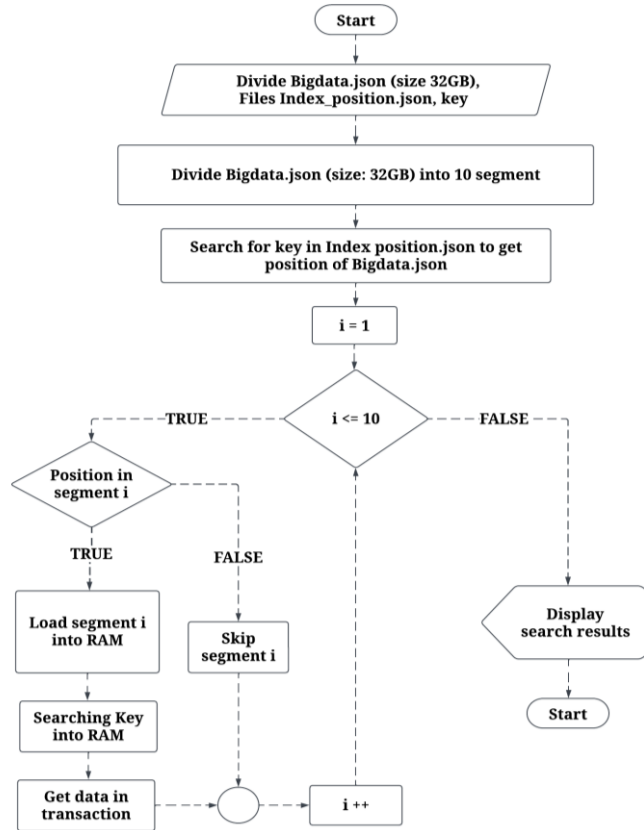


Figure 6. Flowchart of retrieving in a large-scale dataset by index

2.3.1. Dense index in a large-scale dataset

Researchers have developed a dense index algorithm specifically designed for large-scale NoSQL datasets stored in JSON format to improve the efficiency of data retrieval for one-to-one data groups. This algorithm creates a dense index of positions for the dataset and utilizes the dense index file to ensure fast and effective data retrieval in large-scale datasets. The following sections offer a comprehensive explanation of the algorithm and its implementation, detailing the generation of dense indexing for the dataset and how the dense index file is employed to retrieve data from large-scale datasets.

a. Generate dense indexing of position for dataset

In this study, the term large-scale dataset refers to a simulation file set named Bigdata.json created by this study in subsection 2.1. The dense index is a file called Dense_index_position.json generated by scanning the Bigdata.json file once to extract non-repeating data in the field that choose to be key. This study contains the email and its corresponding position in the Bigdata.json file. This dense index file, which has a structure shown in Figure 7, is designed for efficient one-to-one referencing and data access. It contains email and position data, where position enables quick access to data from the Bigdata.json file set in RAM. The present study incorporates a dense index file, which measures 20 megabytes in size and contains 1,000,000 transactions.

```

{
  "Phillip.Headlon@smith.com": 1,
  "Renetta.Marmon@simon.com": 2,
  "Annalee@pinnaclewest.com": 3,
  ...
  "Jonnie@pacificare.com": 1,000,000
}
    
```

Figure 7. Example transactions in dense index position.json

b. The use of `dense_index_position.json` for retrieving in a large-scale dataset

`Dense_index_position.json`, which facilitates querying and accessing data in a large-scale NoSQL dataset stored as JSON. As described in (a. Generate dense indexing of position for dataset), the process is initiated by importing the `Dense_index_position.json` index file and the `Bigdata.json` data set, followed by setting email as the keyword for retrieving and accessing information, in accordance with Figure 7. Upon executing a retrieve operation in the `Dense_index_position.json` file, a position that corresponds to the given keyword is obtained. The file's size of 20 megabytes renders it highly efficient for prompt access. However, each keyword results in a single position in the dense index file, thus requiring the loading of only a segment of the entire index into the RAM for data retrieval. Conversely, when a keyword does not match any transactions in the dense index file, no position is retrieved, and therefore, no segment of the index is loaded into the RAM.

2.3.2. Sparse indexing in large-scale dataset

The primary goal of this study was to improve query times and streamline access to one-to-many data groups in a large-scale dataset stored in NoSQL JSON format. Researchers developed and implemented a sparse index algorithm to achieve this objective. This approach significantly enhances the efficiency of data retrieval and provides a scalable solution for managing large datasets. The following sections will delve into the specifics of the sparse index algorithm, its creation process, and its impact on data retrieval performance.

a. Generate sparse indexing of position for dataset

The present study defines the term large-scale dataset as the `Bigdata.json` file described in section 2.1. To create the sparse index file, named `Sparse_index_position.json` the `Bigdata.json` file is scanned once to extract unique data, using the first name as the chosen key field. This sparse index file set, as depicted in Figure 8, is designed to facilitate one-to-many data retrieve and access. It includes both the first name and its corresponding position data, where a single name may have multiple positions. The position data enables efficient referencing from RAM to data within the `Bigdata.json` file set. The use of the sparse index file set enables efficient data access by including both the first name and position data, where the latter facilitates efficient referencing to data within the `Bigdata.json` file set. The present study incorporates a sparse index file, which measures 8 megabytes in size and contains 5146 transactions.

```
{
  "Bowonsak": [46163, 66867, 88916, 94633, 130606, 150253, 185197, 189917, 214931, 999999],
  ...
  "jirawat": [46153, 59781, 93151, 94460, 101100, 172271, 244064, 257080, 368397, 439620, 888888]
}
```

Figure 8. Example transactions in sparse index position.json

b. The use of `Sparse_index_position.json` for retrieving in a large-scale dataset

To query and access data within a large-scale NoSQL dataset in JSON format, one can utilize the sparse index file `Sparse_index_position.json`, as shown in Figure 8. This index file is particularly well-suited for querying and accessing one-to-many duplicate data within the sparse index files, beginning with the data import. The index file includes first names and their corresponding positions, which act as reference points to the data within the sparse index files stored in the RAM, leading to efficient query times. Subsection 2.1 and Figure 2 illustrate the process starting with the import of the `Bigdata.json` files. In this context, the first name is the primary keyword for retrieving and accessing information. When executing a retrieve operation using the `Sparse_index_position.json` file, the command retrieves a position related to the input keyword. The file is relatively small, at 8 megabytes, contributing to its remarkable efficiency in providing rapid access to the data. However, a keyword retrieve might produce multiple positions within the sparse index file. In such cases, the system must load several index segments into the RAM to extract the necessary data. Conversely, if a keyword fails to match any transactions in the sparse index file, the command retrieves no position, and as a result, the RAM does not load any index segment.

3. RESULTS AND DISCUSSION

This study evaluates the effectiveness of newly developed indexing techniques for NoSQL databases that store data in JSON files in retrieving data from large-scale datasets. Specifically, we compare the performance of dense and sparse indexes with and without indexing. We conducted the study using a personal

laptop with a 3.3 GHz AMD Ryzen™ processor, 16 GB of RAM, and a 500 GB solid-state drive. We imported the large dataset utilized in this study, as described in subsection 2.1, from the Bigdata.json file. To obtain a representative sample, we randomly selected 524 transactions from the email and First_name fields, accounting for 0.05% of the total simulated transactions, and repeated this process three times. The study aims to investigate the time efficiency of data retrieval with and without indexing in the presence or absence of the specified data within the Bigdata.json file. We present the experimental results below.

In the first experiment, we evaluated the efficiency of using dense indices for retrieving data from the Bigdata.json files by comparing the performance of one-to-one query and access times with and without a dense index. It is worth noting that all randomly selected keywords were present in the Bigdata.json file. The experiment aimed to examine the retrieve time efficiency of using dense indices compared to no indexing, focusing on the presence of keywords in the Bigdata.json file. Table 1 presents the experiment results that involved retrieving 524 keywords using a dense index. The study was conducted in three rounds, presenting the average retrieval time per keyword and round. These findings show that using density indexes can help reduce the time to retrieve data compared to not indexing. The average retrieval time per keyword with the dense index was 59.175 milliseconds; without indexing, it was 15,635.232 milliseconds. Across all three rounds, retrieval using a dense index had an average time of 31,008.210 milliseconds, while retrieval without indexing had an average time of 8,192,861.41 milliseconds. The dense index outperforms no indexing because it can skip unnecessary data and retrieve information only once. Moreover, it loads data into RAM once per keyword, reducing retrieval and processing time. In contrast, retrieval without indexing loads all keyword data, leading to significant memory space loss and substantially longer retrieval time based on the data size.

The second experiment assessed the performance of one-to-one query and access times in the dataset using a dense index, compared to not using a dense index, with data from the Bigdata.json files. It is essential to mention that the Bigdata.json file did not contain all the randomly selected keywords. The study yielded the results of this experiment, as shown in Table 2. In situations where a keyword cannot be found, the retrieval time is limited to the time spent retrieving in the index file and does not involve retrieving in RAM. This is because there is no corresponding segment to be loaded and no position in the index to retrieve. The experiment was conducted using the retrieving by dense index method with 524 keywords for retrieval. The results show that the average retrieval time per keyword was 1.097 milliseconds, with an average retrieval time for all three rounds of 574.767 milliseconds. In contrast, the retrieving by no index in case not found method was used with the exact 524 keywords for retrieval. The results indicated that the average retrieval time per keyword was 38,300.085 milliseconds, with an average retrieval time for all three rounds of 20,069,244.44 milliseconds. The results indicate that the retrieving by dense index in case of keyword not found was more efficient, as unnecessary data was skipped, and the data was retrieved only once per keyword, reducing the retrieval time and the average retrieval time per keyword to 1.097 milliseconds. On the other hand, retrieving by no index in case keyword not found resulted in a much longer retrieval time, with an average retrieval time per keyword of 38,300.0848 milliseconds, which caused significant RAM usage for loading all the keywords and retrieving the data.

Table 1. Comparison of the efficiency in retrieval time with and without dense index and the existence of keywords in the data file named Bigdata.json

Type	524 keywords First round	524 keywords Second round	524 keywords Third round	Average 524 keywords	Average 1 keyword
Retrieve time by dense index. (milliseconds)	31272.417	30879.134	30873.079	31008.210	59.175
Retrieve time by no index (milliseconds)	8151247.800	9008687.670	7418648.760	8192861.410	15635.232

Table 2. Comparison of the efficiency in retrieval time with and without dense index and the non-existence of keywords in the data file named Bigdata.json

Type	524 keywords First round	524 keywords Second round	524 keywords Third round	Average 524 keywords	Average 1 keyword
Retrieving by dense index in case keyword not found.	569.960	572.780	581.560	574.767	1.097
Retrieving by no index in case keyword not found.	20528300.020	20871312.760	18808120.530	20069244.440	38300.085

The third experiment evaluated the performance of one-to-many query and access times in the dataset using a sparse index, compared to the absence of a sparse index, for data present within the Bigdata.json files.

It should be noted that all randomly selected keywords were present in the Bigdata.json file. The results of this experiment, as illustrated in Table 3, are the results of an experiment comparing the retrieval time efficiency with and without a sparse index, using 524 keywords from the Bigdata.json data file. The experiment involved three rounds of tests, and the study presents the average retrieval time per keyword and round. The study reveals that using a sparse index significantly improves retrieval time compared to no indexing. The average retrieval time per keyword with the sparse index was 36.387 milliseconds, whereas, without indexing, it was 33,726.653 milliseconds. Furthermore, retrieval with a sparse index had an average retrieval time of 19,067.037 milliseconds across all three rounds, while retrieval without indexing had an average retrieval time of 17,672,766.08 milliseconds. The sparse index outperforms no indexing because it can skip unnecessary data and retrieve information only once. Additionally, it loads data into RAM once per keyword, even if the keyword appears in multiple locations, reducing retrieval and processing time. In contrast, retrieval without indexing loads all keyword data, leading to a significant loss of memory space and a much longer retrieval time based on the data size.

The fourth experiment assessed the performance of one-to-many query and access times in the dataset using a sparse index, compared to not using a sparse index, with data from the Bigdata.json files. It is essential to mention that the Bigdata.json file did not contain all the randomly selected keywords. The study yielded the results of this experiment, as shown in Table 4. When a keyword is absent, the retrieval time is restricted to the duration required for retrieving the index file and does not entail retrieving in RAM. This is because no related segment needs to be loaded, and no position in the index necessitates retrieving. Nevertheless, it is worth noting that the retrieval time for a sparse index is typically shorter than that of a dense index since the size of a sparse index is smaller. In the experiment, data retrieval was conducted using the sparse index with 524 keywords, and retrieving by the sparse index was employed if a keyword was not found. The results indicate that the first retrieval round took 93.12 milliseconds, the second round took 90.2 milliseconds, and the third round took 97.31 milliseconds. The average retrieval time for all three rounds was 93.543 milliseconds, with an average retrieval time per keyword of 0.179 milliseconds. In contrast, when retrieving with no index in case keyword not found was used with the exact 524 keywords for retrieval, the first retrieval round took 26,873,520.89 milliseconds, the second round took 23,599,661.86 milliseconds, and the third round took 23,730,331.4 milliseconds. The average retrieval time for all three rounds was 24,734,504.72 milliseconds, with an average retrieval time per keyword of 47203.253 milliseconds. These findings demonstrate that retrieving by sparse index without a keyword is more efficient. By skipping unnecessary data and conducting retrieves only once per keyword, retrieval time and the average retrieval time per keyword were significantly reduced to 0.179 milliseconds. In contrast, retrieving by no index in case the keyword was not found resulted in a significantly longer retrieval time, with an average retrieval time per keyword of 47203.253 milliseconds. Furthermore, given the data size, this approach consumed significant RAM usage for loading all keywords, and the retrieval time was longer.

Table 3. Comparison of the efficiency in retrieval time with and without sparse index and the existence of keywords in the data file named Bigdata.json

Type	524 keywords First round	524 keywords Second round	524 keywords Third round	Average 524 keywords	Average 1 keyword
Retrieving by sparse index.	18966.679	19118.347	19116.085	19067.037	36.387
Retrieving by no index	18462275.870	19661980.150	14894042.220	17672766.080	33726.653

Table 4. Comparison of the efficiency in retrieval time with and without sparse index and the non-existence of keywords in the data file named Bigdata.json

Type	524 keywords First round	524 keywords Second round	524 keywords Third round	Average 524 keywords	Average 1 keyword
Retrieving by sparse index in case keyword not found	93.120	90.200	97.310	93.543	0.179
Retrieving by no index in case keyword not found.	26873520.890	23599661.860	23730331.400	24734504.720	47203.253

Another objective of this experiment was to determine the optimal segment sizes by conducting tests in three distinct ranges. The first range spanned from 100 to 1,000, increasing by increments of 100 for each segment size. The second range covered 1,000 to 10,000, with increments of 1,000 for each segment size. Lastly, the third range extended from 10,000 to 200,000, with increments of 10,000 for each segment size. We carried out each trial three times and calculated the average value. Furthermore, Tables 5 and 6 summarize the

appropriate segment size values for evaluating the dense and sparse index performance across the three specified ranges. The details of the experimental results follow below.

Table 5 presents the segment size values suitable for evaluating dense index performance, categorized into three ranges. The first segment size range, from 100 to 1,000, has an average time per keyword of 67.189 milliseconds. The second range, from 1,000 to 10,000, has an average time per keyword of 59.872 milliseconds. Finally, the third range, from 10,000 to 200,000, has an average time per keyword of 56.314 milliseconds. Figure 9 illustrates that smaller segment sizes, such as those in the 100 to 1,000 and 1,000 to 10,000 ranges, yield slower performance due to increased memory access frequency. The high memory access frequency causes the system to load only a portion of the segment into the RAM. Rather than examining the entire segment, the system investigates its index position to determine its presence, consequently increasing latency. In contrast, larger segment sizes, such as 10,000 to 200,000, provide enhanced stability and faster performance because they require fewer comparisons between segment levels and do not substantially affect retrieving speed. However, if the segment size exceeds the specified value, the system may run out of memory and fail. It is essential to emphasize that the optimal range of 20,000-200,000 depends on the machine's testing capabilities, and the ideal range may differ based on the system's available resources.

Table 6 presents the segment size values suitable for evaluating sparse index performance, categorized into three ranges. The first segment size range, from 100 to 1,000, has an average time per keyword of 42.776 milliseconds. The second range, from 1,000 to 10,000, has an average time per keyword of 36.464 milliseconds. Finally, the third range, from 10,000 to 200,000, has an average time per keyword of 32.989 milliseconds. Figure 10 illustrates that smaller segment sizes, such as those in the 100 to 1,000 and 1,000 to 10,000 ranges, yield slower performance due to increased memory access frequency. The high memory access frequency causes the system to load only a portion of the segment into the RAM. Rather than examining the entire segment, the system investigates its index position to determine its presence, consequently increasing latency. In contrast, larger segment sizes, such as 10,000 to 200,000, provide enhanced stability and faster performance because they require fewer comparisons between segment levels and do not substantially affect retrieving speed. However, if the segment size exceeds the specified value, the system may run out of memory and fail. It is essential to emphasize that the optimal range of 20,000 to 200,000 depends on the machine's testing capabilities, and the ideal range may differ based on the system's available resources.

Table 5. Dense index performance based on segment sizes and keyword frequency

Segment sizes	Average 1 keyword (ms)
100 to 1000	67.189
1000 to 10000	59.872
10000 to 200000	56.314

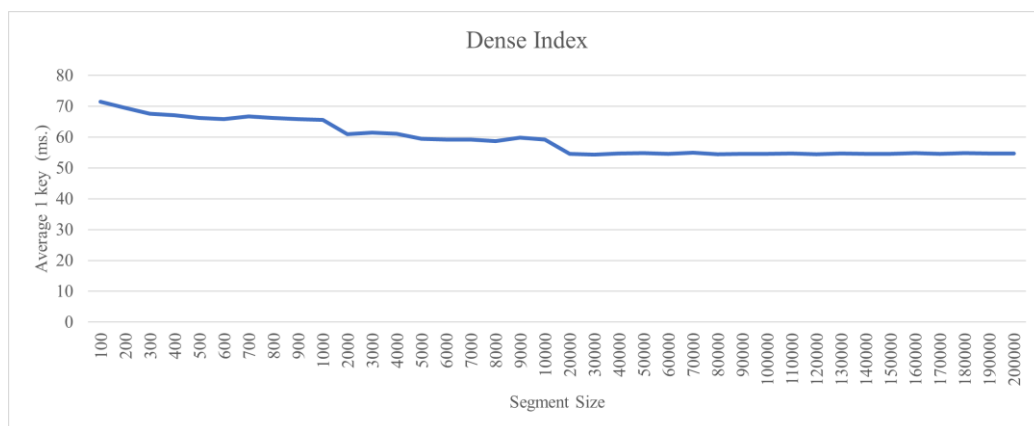


Figure 9. Dense index performance comparison

Table 6. Sparse index performance based on segment sizes and keyword frequency

Segment sizes	Average 1 keyword (ms.)
100 to 1000	42.776
1000 to 10000	36.464
10000 to 200000	32.989

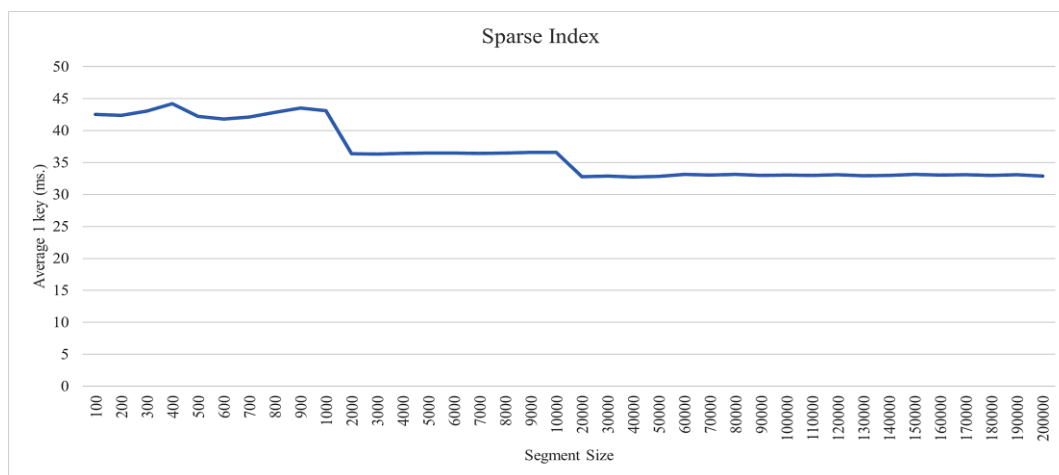


Figure 10. Sparse index performance comparison

Notice, the retrieval time using dense indexing was found to be longer than that using sparse indexing. This can be attributed to the longer keyword search time required to retrieve the position of data in the dense index file, as compared to the sparse index file. This difference is also evident when considering the file size and number of transactions, where the dense index file has a size of 20 megabytes and contains 1,000,000 transactions, while the sparse index file is only 8 megabytes in size and contains 5146 transactions.

4. CONCLUSION

This study presents and assesses a novel JSON file indexing approach for large datasets, integrating dense and distributed indexing techniques to enhance retrieval efficiency. The findings demonstrated that the new indexing method significantly decreased retrieval time across all three test rounds. Moreover, experimenting with optimal segment sizes ranging from 20,000 to 200,000 can boost stability and achieve quicker performance due to fewer segment-level comparisons. Dense Indexing yields an average fetch time per keyword of 59.175 ms., a 1.097 ms. Reduction in fetch time compared to a non-indexed one-to-one fetch without keywords. Sparse indexing optimizes the average fetch time per keyword to 36.387 ms. for one-to-many data, decreasing fetch time by 0.179 ms. When no keywords are present, these findings suggest that the proposed indexing approach enhances data retrieval in NoSQL datasets formatted in JSON, particularly when simulating large datasets. Future studies can extend this research by testing the performance of the indexing technique on different types of data and different sizes of datasets to determine its scalability and generalizability. In future work, improving the searching algorithm for retrieving data positions in dense index files could be a potential area of investigation. As dense index files often contain many transactions, the search process can be time-consuming and inefficient. Reducing the time required for this process could significantly improve the overall efficiency of the indexing approach. Hence, exploring methods to optimize the search algorithm for dense indexing could be an interesting avenue for future work.

ACKNOWLEDGEMENTS

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




REFERENCES

- [1] J. Chang *et al.*, "Optimization of index-based method of metadata search for large-scale file systems," in *2017 10th International Symposium on Computational Intelligence and Design (ISCID)*, Dec. 2017, pp. 476–481, doi: 10.1109/ISCID.2017.147.
- [2] R. Chopade and V. Pachghare, "MongoDB indexing for performance improvement," *Advances in Intelligent Systems and Computing*, vol. 1077, pp. 529–539, 2020, doi: 10.1007/978-981-15-0936-0_56.
- [3] J. Yuan and X. Liu, "A novel index structure for large scale image descriptor search," in *2012 19th IEEE International Conference on Image Processing*, Sep. 2012, pp. 1937–1940, doi: 10.1109/ICIP.2012.6467265.
- [4] K. Zineddine, F. M. Amine, and A. Adeel, "Indexing multimedia data with an extension of binary tree–image search by content–," *International Journal of Informatics and Applied Mathematics*, vol. 1, no. 1, pp. 47–55, 2021.
- [5] P. Jin, X. Zhuang, Y. Luo, and M. Lu, "Exploring index structures for zoned namespaces SSDs," *Proceedings - 2021 IEEE International Conference on Big Data, Big Data 2021*, pp. 5919–5922, 2021, doi: 10.1109/BigData52589.2021.9671606.




- [6] A. A. Abdulkadhem and T. A. A. -Assadi, "An important landmarks construction for a GIS-Map based on Indexing of Dolly Images," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 15, no. 1, pp. 451-459, 2019, doi: 10.11591/ijeecs.v15.i1.pp451-459.
- [7] A. Alqatawneh, "Orthogonal frequency division multiplexing system with an indexed-pilot channel estimation," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 26, no. 2, pp. 808-818, May 2022, doi: 10.11591/ijeecs.v26.i2.pp808-818.
- [8] Mouneshachari and M. S. Pande, "Indexing Intelligence using benchmark classifier," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 18, no. 1, pp. 179-187, Apr. 2019, doi: 10.11591/ijeecs.v18.i1.pp179-187.
- [9] J. Zeffora and S. Shobarani, "Optimizing random forest classifier with Jenesis-index on an imbalanced dataset," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 26, no. 1, pp. 505-511, Apr. 2022, doi: 10.11591/ijeecs.v26.i1.pp505-511.
- [10] K. L. Tan and K. C. Lim, "Fast surveillance video indexing & retrieval with WiFi MAC address tagging," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 16, no. 1, pp. 473-481, Oct. 2019, doi: 10.11591/ijeecs.v16.i1.pp473-481.
- [11] A. I. Abdulsada, D. G. Honi, and S. Al-Darraj, "Efficient multi-keyword similarity search over encrypted cloud documents," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 23, no. 1, pp. 510-518, 2021, doi: 10.11591/ijeecs.v23.i1.pp510-518.
- [12] Y. Ma, D. Liu, G. Scott, J. Uhlmann, and C. R. Shyu, "In-memory distributed indexing for large-scale media data retrieval," *Proceedings - 2017 IEEE International Symposium on Multimedia, ISM 2017*, vol. 2017-January, pp. 232-239, 2017, doi: 10.1109/ISM.2017.38.
- [13] Y. Fathy, P. Barnaghi, and R. Tafazolli, "Large-scale indexing, discovery, and ranking for the internet of things (IoT)," *ACM Computing Surveys*, vol. 51, no. 2, pp. 1-53, Mar. 2018, doi: 10.1145/3154525.
- [14] T. Silwattananusarn and P. Kulkanjanapiban, "A text mining and topic modeling based bibliometric exploration of information science research," *IAES International Journal of Artificial Intelligence*, vol. 11, no. 3, pp. 1057-1065, 2022, doi: 10.11591/ijai.v11.i3.pp1057-1065.
- [15] Bertalya, Prihandoko, L. Setyowati, F. I. Irawan, and S. R. Irlianti, "Formulation of city health development index using data mining," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 23, no. 1, pp. 362-369, 2021, doi: 10.11591/ijeecs.v23.i1.pp362-369.
- [16] Y. Asri, D. Kuswardani, E. Yosrita, and F. Hendrik Wullur, "Clusterization of customer energy usage to detect power shrinkage in an effort to increase the efficiency of electric energy consumption," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 22, no. 1, pp. 10-17, Apr. 2021, doi: 10.11591/ijeecs.v22.i1.pp10-17.
- [17] M. Jupri and R. Sarno, "Data mining, fuzzy AHP and TOPSIS for optimizing taxpayer supervision," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 18, no. 1, pp. 75-87, Apr. 2019, doi: 10.11591/ijeecs.v18.i1.pp75-87.
- [18] M. K. Yusof and M. Man, "Efficiency of JSON for data retrieval in big data," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 7, no. 1, pp. 250-262, Jul. 2017, doi: 10.11591/ijeecs.v7.i1.pp250-262.
- [19] D. Lee, A. Althoff, D. Richmond, and R. Kastner, "A streaming clustering approach using a heterogeneous system for big data analysis," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2017, pp. 699-706, doi: 10.1109/ICCAD.2017.8203845.
- [20] S. Gebeyehu, W. Wolde, and Z. S. Shibeshi, "Information extraction model from Ge'ez texts," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 30, no. 2, pp. 787-795, 2023, doi: 10.11591/ijeecs.v30.i2.pp787-795.
- [21] A. Althaf Ali and R. M. Shafi, "Test-retrieval framework: Performance profiling and testing web search engine on non factoid queries," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 14, no. 3, pp. 1373-1381, 2019, doi: 10.11591/ijeecs.v14.i3.pp1373-1381.
- [22] M. K. Zuhanda *et al.*, "Supply chain strategy during the COVID-19 terms: sentiment analysis and knowledge discovery through text mining," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 30, no. 2, pp. 1120-1127, 2023, doi: 10.11591/ijeecs.v30.i2.pp1120-1127.
- [23] S. W. Kareem, R. Z. Yousif, and S. M. J. Abdalwahid, "An approach for enhancing data confidentiality in hadoop," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 20, no. 3, pp. 1547-1555, Dec. 2020, doi: 10.11591/ijeecs.v20.i3.pp1547-1555.
- [24] S. Nagendra Prasad and S. S. Kulkarni, "Quality and energy optimized scheduling technique for executing scientific workload in cloud computing environment," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 21, no. 2, pp. 1039-1047, Feb. 2020, doi: 10.11591/ijeecs.v21.i2.pp1039-1047.
- [25] G. Panatula, K. S. Kumar, D. E. Geetha, and T. V. S. Kumar, "Performance evaluation of cloud service with hadoop for twitter data," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 13, no. 1, pp. 392-404, 2019, doi: 10.11591/ijeecs.v13.i1.pp392-404.
- [26] S. Wilson and R. Sivakumar, "Twitter data analysis using hadoop ecosystems and apache zeppelin," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 16, no. 3, pp. 1490-1498, Dec. 2019, doi: 10.11591/ijeecs.v16.i3.pp1490-1498.
- [27] N. R. Gayathiri, D. D. Jasper, and A. M. Natarajan, "Big Data retrieval techniques based on Hash Indexing and MapReduce approach with NoSQL Database," in *2019 International Conference on Advances in Computing and Communication Engineering (ICACCE)*, Apr. 2019, pp. 1-8, doi: 10.1109/ICACCE46606.2019.9079964.
- [28] R. S. A. Usmani, W. N. F. Binti Wan Azmi, A. M. Abdullahi, I. A. T. Hashem, and T. R. Pillai, "A novel feature engineering algorithm for air quality datasets," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 19, no. 3, pp. 1444-1451, Sep. 2020, doi: 10.11591/ijeecs.v19.i3.pp1444-1451.
- [29] E. M. Marouane and Z. Elhoussaine, "A fuzzy neighborhood rough set method for anomaly detection in large scale data," *IAES International Journal of Artificial Intelligence*, vol. 9, no. 1, pp. 1-10, 2020, doi: 10.11591/ijai.v9.i1.pp1-10.

BIOGRAPHIES OF AUTHORS






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. He is currently pursuing a master's degree 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: 64024804@up.ac.th.






Sakorn Mekruksavanich    received a Ph.D. in Computer Engineering from Chulalongkorn University in 2012. He also holds an M.S. in Computer Science from King Mongkut's Institute of Technology Ladkrabang in 2004 and a B.Eng. in Computer Engineering from Chiang Mai University in 1999. He is currently a faculty member of the Department of Computer Engineering, School of Information and Communication Technology at the University of Phayao, Phayao, Thailand. His research interests are deep learning, human activity recognition, neural network modeling, wearable sensors, and applying deep learning techniques in software engineering. He can be contacted at email: sakorn.me@up.ac.th.



Nakarin Chaikaew    Asst. Prof. Dr. Nakarin Chaikaew Ph.D. in Remote Sensing and Geographic Information Systems, Asian Institute of Technology (AIT). Assistant Professor in Geographic Information Science, University of Phayao, Thailand. He can be contacted at email: nakarin.ch@up.ac.th.



Pornthep Rojanavasu    received a Ph.D. in Electrical Engineering from King Mongkut's Institute of Technology Ladkrabang, Bangkok, Thailand, in 2010. He also received a B.Eng. degree in computer engineering from Chiang Mai University, Chiang Mai, Thailand, in 1999 and an M.Eng. degree in Computer Engineering from King Mongkut's Institute of Technology Ladkrabang, Bangkok, Thailand, in 2004. He is currently a faculty member of the Department of Computer Engineering, School of Information and Communication Technology at the University of Phayao, Phayao, Thailand. His research interests are large-scale data mining, distributed data mining, problem decomposition, learning classifier systems, and neural network. He can be contacted at email: pornthep.ro@up.ac.th.