Optimizing Data Availability and Scalability with RP*-SD2DS Architecture for Distributed Systems

Mohammed Maabed
Mathematics and Applications - MIA Laboratory, Hassiba Benbouali University, Chlef, Algeria
m.maabed@univ-chlef.dz (corresponding author)

Nassim Dennouni
Higher School of Management, Tlemcen, Algeria, | Computer Science Department - LIA Laboratory, Hassiba Benbouali University, Chlef, Algeria
n.dennouni@univ-chlef.dz

Mohamed Aridj
Computer Science Department - LIA Laboratory, Hassiba Benbouali University, Chlef, Algeria
m.aridj@univ-chlef.dz

Received: 19 June 2024 | Revised: 6 July 2024 | Accepted: 13 July 2024

Licensed under a CC-BY 4.0 license | Copyright (c) by the authors | *DOI: https://doi.org/10.48084/etasr.8176*

ABSTRACT

This work introduces Range Partitioning Scalable Distributed Two-Layer Data Structures (RP*-SD2DS), an innovative data storage architecture with the objective of enhancing data availability and scalability in distributed systems. By employing SD2DS and preorder-preserving RP*, this design avoids the need for a router or coordinator, ensuring dynamic adaptability. The main goal is to minimize system downtime by efficiently distributing data across two layers and increasing availability during partitioning operations in traditional SDDSs, thereby avoiding the bottlenecks associated with master- or coordinator-based systems. The proposed solution offers significant improvements over MongoDB, a well-known and robust system, with a single or three Mongos instances.

Keywords-range partioning; multi-computers; large files; Scalable Distributed Two-Layer Data Structures (SD2DS); Non - Structured Query Language (NoSQL)

I. INTRODUCTION

It is becoming increasingly clear that dependable data storage systems are essential for maintaining the development of Internet of Things (IoT) infrastructures, considering the exponential growth of data both processed and stored online [1]. Delays in data transmission can result in substantial financial losses and reduced productivity, while the permanent loss of data can have severe and far-reaching consequences [2]. It is therefore evident that dependable and effective data storage solutions are of great importance for a multitude of applications, including those pertaining to business, social networking platforms, multimedia and cloud services [3, 4]. In order to enhance scalability and efficiency, a considerable number of applications are migrating their data to Non-Structured Query Language (NoSQL) systems [5].

However, these systems are often optimized for large datasets, leaving substantial amounts of data in flat file formats [6]. The management of large volumes of files necessitates the implementation of distributed storage solutions [7]. Typically, in NoSQL systems, data and metadata are stored separately, which poses difficulties in data aggregation and increases maintenance complexity [8]. Nevertheless, multi-computer systems remain the most cost-effective solution for high-performance applications [9].

Scalable Distributed Data Structures (SDDS), designed to improve performance within a client-server architecture, represent a significant innovation in this field [10]. However, SDDSs encounter difficulties with large files, particularly during the process of splitting them. Recently, research has indicated that it may be more efficient to store only the metadata on a dynamic partition, while the actual data are stored in a fixed partition on the outside [11]. In order to efficiently manage large files, a distributed data storage system based on Scalable Distributed Two-Layer Data Structures (SD2DS) [11], which supports simple and scalable key-value access is proposed. This system is particularly effective for medium and large files, as it employs Linear Hashing (LH*) SDDS to manage the initial layer [12]. Although SDDS LH*

employs a linear hash function to facilitate efficient access, it encounters difficulties in handling range requests and necessitates multicasting, which is a resource-intensive process. Furthermore, the process of sorting results is costly, and the necessity of a coordinator for splitting introduces a single point of failure. The SD2DS LH* employs a linear hash function for accessing the first layer, which, although effective, encounters difficulties with range queries and necessitates multicast for their processing. Furthermore, the process of sorting the results is also costly, and the system is dependent on a coordinator to oversee the splitting process, which presents a significant challenge if the coordinator is unable to perform its duties. Considering that many NoSQL systems, such as MongoDB, employ a multitude of access methodologies, including hashing and sharding, Range Partitioning (RP*) has been integrated into the initial layer [14]. This enhancement addresses the previous issues within a system that does not require a coordinator.

This study developed an SD2DS-based architectural framework with the specific objective of addressing the challenges posed by range requests. The header is managed within the scalable section, thereby streamlining the process. The storage of large files is facilitated by the use of fixed bucket partitions at each node, thereby enhancing scalability and minimizing the amount of data transmitted during the splitting process. This process involves the transfer of half of the stored data in a node to a new one. This design enables the system to efficiently manage large files with minimal transmission costs. A comprehensive description of the principal elements of the global system architecture is provided, followed by an overview of both recent and established research in the field. Subsequently, a detailed account of the RP*-SD2DS system for the management of large files is given, succeeded by a comparison of the simulated results with those of MongoDB.

II. RELATED WORK

The system consists of three main categories of nodes: servers, clients, and a split coordinator, as outlined in the SDDS. The servers are responsible for storing data in discrete units, which are collectively referred to as "buckets." The combination of these buckets constitutes the storage file. A bucket may be used to store both headers and bodies either simultaneously or separately, with the option of storing them on the same node or on different ones. The responsibility for data processing is assigned to the clients, while the split coordinator is tasked with monitoring the status of the two-layer system. The coordinator plays a significant role in maintaining the scalability of the system. SDDS is divided into categories, including SDDS LH* [15], SDDS RP*, and trees/tries. The LH* variant, which is particularly suitable for data access, employs a splitting mechanism that is coordinated. In contrast, the RP* variant employs range queries, thereby capitalizing on the benefits of pre-ordered data. In order to address records, SDDS LH* employs a hashing function, such as a straightforward modulo division:

$$h(C) = C \bmod 2^i \quad (1)$$

where C is a key, and i is the file level, which increases as the file grows.

It should be noted that the file does not require the use of buckets whose numbers are powers of two. Consequently, at most, two successive levels (i and $i + 1$) can be utilized simultaneously. However, SDDS is inadequately equipped to process large files (exceeding 1 MB), particularly in the context of data transfer. The RP* classification [16], employs Binary Trees (B-Trees) to maintain order. This category comprises the RP*N, RP*C, and RP*S structures, all of which have been designed for preserving order in SDDSs that employ range partitioning. RP*N deploys multicast messages across local, ATM, and wireless networks to implement range partitioning without the necessity for an index. The RP*C, enhancement to the RP*N structure, incorporates indexes, thereby facilitating point-to-point messaging for key searches, inserts, and deletes. Furthermore, RP*S enhances throughput by constructing indexes on servers.

Authors in [17] introduced Scalable Distributed Compact Trie Hashing (CTH*), a novel distributed data structure designed for multicomputer systems. The CTH* data structure is designed to maintain record order, operate without multicast, and use only three bytes to address a server, making it suitable for dynamic workloads. It was applied in the context of distributed databases, file systems and caching systems. In [18], the MapReduce (MR2P*) framework is introduced, which incorporates a distributed range partitioning algorithm into the MapReduce framework [19]. This approach is focused on the dynamic scheduling of data, with the objective of optimizing the shuffle phase between mapping and reducing operations. These enhancements are achieved through the utilization of a partitioning function based on the SDDS-RP framework. Authors in [20] introduced (TH*), a client/server architectural model. It is the responsibility of each client to maintain a partial trie that represents its own view of the distributed file. The system may be initiated with an empty trie, wherein each server is assigned a bucket containing the file's records, the trie itself, and an interval min, max. Initially, only server 0 is present, with an empty bucket and a trie. The number of servers is regarded as infinite, with each server being either determined statically or dynamically. In the event of an addressing error, a portion of the server's trie is transferred to the client for its trie to be updated.

MongoDB [21] is a NoSQL document-based data store with extensive capabilities that bear resemblance to those of traditional databases. The database utilizes Binary JavaScript Object Notation (BSON) for storage and supports atomic access for individual documents. MongoDB employs a sharding technique to distribute data across a cluster of servers, with the configuration server overseeing the management of all shards. MongoDB's facilitates communication between clients and instances, abstracting shard status. The data are stored on reliable media, such as disk drives, and access is facilitated by the use of journals and cache mechanisms. MongoDB facilitates efficient data storage via Grid File System (GridFS) [22], which manages data in two collections: fs.files and fs.chunks. The SD2DS data structure, which has recently been proposed, addresses the limitations of SDDS by implementing

a two-layer architectural model comprising a file layer and a storage layer. The file layer is responsible for the management of data keys, including the assignment of locators that point to the specific data locations. The storage layer, in contrast, is tasked with the storage of the complete data set, comprising not only the data itself but also the associated keys and attributes. The two layers are managed independently. The SD2DS employs the SDDS LH* to manage its first layer, utilizing headers as records. This design strategy aims to minimize the amount of data transferred during the expansion of the first layer. Table I provides a summary of the principal differences between the LH*-SD2DS, MongoDB and the introduced system design (RP*-SD2DS).

TABLE I. A COMPARISON BETWEEN LH*-SD2DS, MONGODB, AND RP*-SD2DS

Approaches	Need special nodes	Max blob size	scale out (splitting)	Access method
LH*-SD2DS	Splitting coordinator	Not-fixed	Auto	Linear hashing
MongoDB	Router (mongos)	Fixed	Manual	Hash/range
RP*-SD2DS (proposed)	N/A	Not-fixed	Auto	Rrange

III. THE PROPOSED ARCHITECTURE

The implemented system comprises a cluster of multi-computers, which includes several servers and clients. Applications establish a direct connection with the clients, providing an Application Programming Interface (API) that enables the execution of a range of queries, including get, put, and delete. A server is constituted of two components, known as buckets, which are the first and second layers of buckets. The principal operational phases of the system are illustrated in Figure 1 where: (1) the application sends requests (put, set, update, delete, scan(range)) towards the client, (2) the client gets the target server from its image, (3) the server answers with an Image Adjustment Message (IAM), (4) the server forwards the request to the right server, (5) forwarding of the body is performed if it does not have enough capacity, (6) refers to the sending of the locator to the first layer, (7) is the split process in case the first layer bucket is full, (8) responds to the client with confirmation of insertion and a Capacity Adjustment Message (IAM_c), and (9) the client responds to the application. The proposed method employs RP*-SD2DS to enhance data availability, ensuring the efficient management of both the file and storage layers. The file layer is responsible for the management and storage of data keys, along with locators that point to the data. In contrast, the storage layer is tasked with retaining the complete data set. The following subsections provide a detailed account of each layer.

A. *First Layer*

The initial layer comprises a receptacle with a header that includes the bucket ID, the maximum number of records, the actual number of records, the range, a pointer to the root index (B-tree), and the index size. The data are composed of key-locator pairs. This layer incorporates algorithms for insertion, splitting and the processing of both search and range queries, ensuring efficient data management and retrieval.

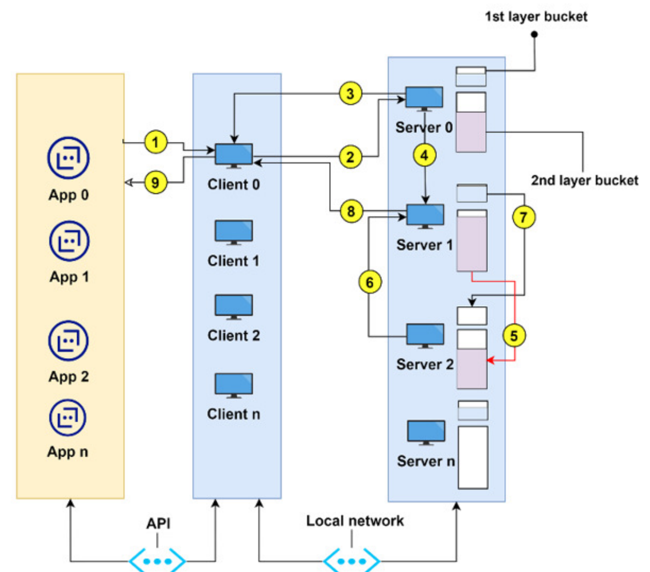

Fig. 1. Our proposed system architecture.

a) *Insertion*

Upon receipt of a request from a client, the system proceeds to differentiate the request into two distinct components: the header and the body. Subsequently, the server saves the file. If the second layer is unable to accommodate the request due to insufficient available space, the server initiates a transfer to an alternative server. Once the locator address is received, the key along with the locator is stored in the first layer. Subsequently, the client is furnished with confirmation of the successful insertion via an IAM. The primary stages involved in the processing of insertion requests are outlined in Algorithm (1). During the process of insertion, the client utilizes its localized image to transmit requests to the initial layer. In the case of an outdated client image, the server will issue an IAM and redirect the request to the relevant server.

```
Algorithm 1: Insert Query
Input:
1 key: record’s key
2 file: data in MiB
3 nb: current number of records
4 nbMax: Max number of records by bucket
5 if (key ∈ localInterval) then
6 locator ← secondLayer.storeFile(key,
file);
7 bucket.insert(key, locator);
8 sendClientSucceed();
9 if (nb > nbMax) then
10 split();
11 end if
12 else
13 S ← getImageServer(key);
14 forward(key, f ile, S );
15 sendClientIAM();
16 end if
```

b) The Search Algorithm

There are two categories of search requests: precise queries and range queries. The initial layer addresses the search request by identifying the key, monitoring the location, and subsequently retrieving the comprehensive data from the second layer. In instances where addressing errors have occurred, an IAM is obtained to prevent future occurrences. In the case of range requests, clients submit queries to the first layer buckets that fall within the specified interval, which they obtain from their local image. If an addressing error occurs, clients are furnished with IAMs, otherwise, they are provided with locators, which enable them to access their data from the second layer. Algorithms (2.1) and (2.2) respectively, outline the steps involved in the management of a search request at the client and server levels.

```
Algorithm 2.1: Range query (Client Side)
Input:
1 listFirstLayerServers: List of servers
from clientImage with a specified key
range.
2 keymin, keymax: Range of keys.
3 for each S in listFirstLayerServers do
4 sendServerRange(S, keymin, keymax,
S.serverInterval);
5 end for
6 // Reception Part:
7 Response.unionAll(Rep);
8 listSecondLayerServers ←
getListLocators(Rep);
9 for each Server in
listSecondLayerServers do
10 getFiles(S, keys);
11 end for
```

```
Algorithm 2.2: Range query (Server Side)
Input:
1 Req.serverInterval: Server interval from
the request.
2 B.interval: Interval of the bucket.
3 keymin, keymax: Range of keys.
4 if ( Req.serverInterval ≠ B.interval )
then
5 S.next ←
serverImage.getNextServer(keymin, keymax);
6 forwardServerRange(S.next, keymin,
keymax);
7 if (Req.keymin, Req.keymax) ∩ B.interval
≠ ∅  then
8 sendClient(B.keys);
9 end if
10 else if (Req.keymin, Req.keymax) ∩
B.interval ≠ ∅ then
11 sendClient(B.keys);
12 end if
```

c) Splitting Process

Once a bucket (B) has reached its maximum capacity for storing records, the splitting procedure is initiated by requesting the allocation of a new server. A request is then sent to a server bucket (M), which is assumed to be available. The destination server responds with an acknowledgement (ACK) if the split is approved, otherwise, the request is rejected. Subsequently, buckets B and M are designated as the left and right siblings, respectively. In the event of acceptance, half of the data from bucket B are transferred to the newly designated server, with the interval set to (λM, ΛM), where λM and ΛM represent the middle and maximum record keys of bucket B, respectively. If rejected, the splitting server continues to change server's ID until it identifies an available one. Algorithm (3) describes the steps of the splitting process.

B. Second Layer

The storage layer, or second layer, assumes the majority of server space and is a fixed partition, ensuring that data are permanently saved in this layer. Upon reaching its maximum capacity, the system requires to allocate additional storage. In this architectural configuration, the second layer maintains the capacity to process inserts and respond to specific GET requests in a manner that is independent of the first layer. This design optimizes server availability during the splitting process in the first layer. The storage algorithm commences with the current server verifying the availability of the requisite capacity. Once data and the associated key have been successfully inserted, the server updates its capacity and returns the locator. If the insertion cannot be accommodated due to insufficient capacity, the server denies the request and sends an IAM_c to prompt an update of the sender's capacity status. The system then proceeds to find another available server or allocate a new one to store the requested data, as depicted in the second layer and described in Algorithm (4).

```
Algorithm 3: Split
Input:
1 B: the overflowing bucket
2 M: a new bucket
3 Header: bucket header
4 keym: bucket middle key
5 Repeat
6 ack ← createNewBucket(M);
7 if (ack) then
8 λM ← keym(B);
9 ΛM ← ΛB;
10 Header(M, λM, ΛM);
11 copyRecords(B, M, λm, Λm);
12 removeRecords(B, λm, Λm);
13 else
14 M++;
15 end if
16 until ack
17 ΛB ← keym(B);
18 Header(B, λB, ΛB);
```

```
Algorithm 4: Storage
Input:
1 server.currentCapacity: Current capacity
of the server.
2 file.size: Size of the file to store.
```

```
3 if server.currentCapacity ≥  file.size
then
4 Store(file);
5 server.currentCapacity -= file.size;
6 return locator;
7 else
8 return IAM_c;
9 end if
```

Splitting data is restricted to the initial layer, whereas the expansion of servers, when necessary, is performed in the second layer. If the system's servers reach full capacity during a new insertion, the insertion will be directed to a new server or another server with sufficient capacity. The server that received the request will be given priority.

IV. SIMULATION RESULTS AND DISCUSSION

In order to demonstrate the scalability of the system, simulations were conducted using large files of varying sizes (1, 2, 5, and 10 MiB) and different client counts (1, 2, 4, 8, 16, and 32). The system configuration was established using four client personal computers (Dell 5 i5, 8 GB RAM) and multiple servers (i5, 16 GB RAM) to form a cluster. A comparison was carried out between the proposed method and MongoDB, with the former being tested against the latter using one and three Mongos, respectively. The split process remains a critical operation, consuming significant time and resources in data storage systems.

A. Split-Process Results

An evaluation of the system was performed using a cluster comprising four client PCs and three server PCs. The consistent splitting time indicates that the procedure for splitting the data consistently takes the same amount of time, regardless of whether data are inserted simultaneously, based on the amount of data stored. In the simulated scenarios, the duration of the split period ranged from 133 milliseconds to 217 milliseconds. The efficacy of the method was evaluated using 512 records of varying file sizes (1 MiB, 2 MiB, 5 MiB, and 10 MiB) and different workloads, represented by the number of clients (1, 8, and 32 clients) working simultaneously. In essence, the splitting procedure is applicable to the entire data collection, whereby the data are divided into two portions and transferred to a distinct server.

The time required for the splitting process, depends on the configuration of the network and the volume of data being transported. The time required for this process can be estimated by considering the amount of data to be transported and the available throughput. The primary criterion was the amount of data sent. The storage space was divided into two sections: the first section contains the complete set of data, while the second section holds the locator, which serves as a reference to the data key. Figure 2 presents three distinct scenarios, each characterized by a varying number of clients (1, 8, and 32) and record sizes. The results demonstrate that the proposed strategy accurately predicts the splitting time for each scenario, with a few milliseconds of precision, while maintaining stability.

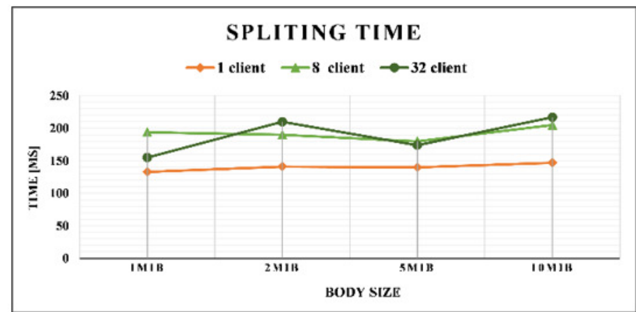

Fig. 2. A scenario with 512 records inserted.

Tables II and III show the insertion of 512 records, with and without accounting for the average splitting time, respectively. Upon reaching its maximum capacity, a node will undergo a split. Therefore, the insertion of the 513th record will be postponed until the completion of the split and insertion processes, as nodes undergo a split at the 512th insertion. If the split time is 133 ms and the insertion time is 44.75 ms, the total time would be 177.75 ms, as demonstrated in (2):

$$T_i = \frac{\sum_{i=1}^{R} t_i}{R} \quad (2)$$

where T_i is the insertion time, t_i is the insertion time by record and R the number of inserted records.

TABLE II. INSERT TIME FOR DIFFERENT BODY SIZE WITH SPLIT TIME

Body Size	1 MiB	2 MiB	5 MiB	10 MiB
1 client	106.95	212.103	531.199	1056.25
2 clients	173.54	273.3	653.66	1291.83
4 clients	255.109	385.015	840.57	1617.25
8 clients	374.95	627.21	1660.03	3165.53
16 clients	821.34	1378.93	3678.4	7174.093
32 clients	1902.25	2911.93	9095.062	12953.000

TABLE III. INSERT TIME FOR DIFFERENT BODY SIZES WITHOUT SPLIT TIME

Body Size	1 MiB	2 MiB	5 MiB	10 MiB
1 client	106.4305	211.5522	530.6521	1055.676
2 clients	172.9619	272.7688	653.0194	1291.256
4 clients	254.4957	384.4173	840.0388	1616.707
8 clients	374.1922	626.4678	1659.327	3164.729
16 clients	820.6408	1378.094	3677.822	7173.515
32 clients	1901.645	2911.11	9094.382	12952.15

Time lost, due to inserting a single record is equal to the sum of the insertion time and the split time, when the latter is divided among the 512 insertions as evidenced in (3). Therefore, the system will pause for approximately 0.25 ms for each record, which is roughly 133 ms divided by 512 insertions.

$$S_i(R,S) = \frac{\sum_{i=1}^{R} t_i - (t_s * S)}{R} \quad (3)$$

where t_s is a split time record and S is the number of split operations. The mean split time is taken into account in the insertion times for a set of records (R), as outlined in (2) and (3). Table IV demonstrates that this discrepancy remains approximately constant.

TABLE IV. DIFFERENCES BETWEEN EACH INSERTION WITH AND WITHOUT SPLIT TIME

Body Size	1 MiB	2 MiB	5 MiB	10 MiB
1 client	0.51	0.55	0.54	0.57
2 clients	0.57	0.53	0.64	0.57
4 clients	0.61	0.59	0.53	0.54
8 clients	0.75	0.74	0.70	0.80
16 clients	0.69	0.83	0.57	0.57
32 clients	0.60	0.82	0.67	0.84

The results of the simulation exhibit that the proposed approach's split process is reliable and efficient for large-file storage systems. Furthermore, the split process ceases only upon the insertion of the initial layer of the split node, while the second layer continues to operate uninterrupted. To ascertain the efficacy of the solution introduced in comparison to alternative systems, this study conducted an experiment where the average time spent was measured by inserting 512 records, each containing a 1 MiB blob. Figure 3, illustrates the reduction in insertion time for MongoDB with one and three Mongos, respectively.

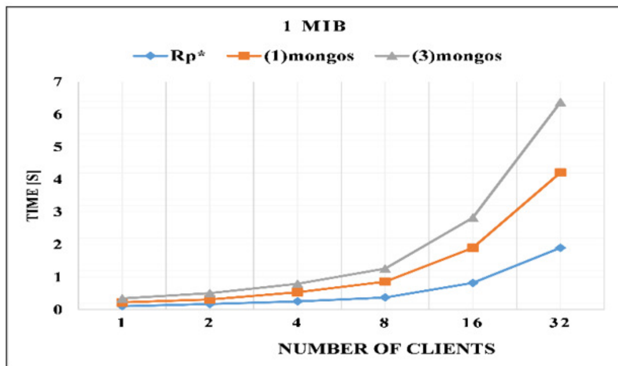

Fig. 3. 512 records of 1 MiB file size insertions.

Figures 4-6 depict the same simulations as those of Figure 3, but with files of varying sizes (1, 2, 5, and 10 MiB). They present the efficiency of the system in processing file insertions rapidly, whether using one or three Mongos, in comparison to MongoDB.

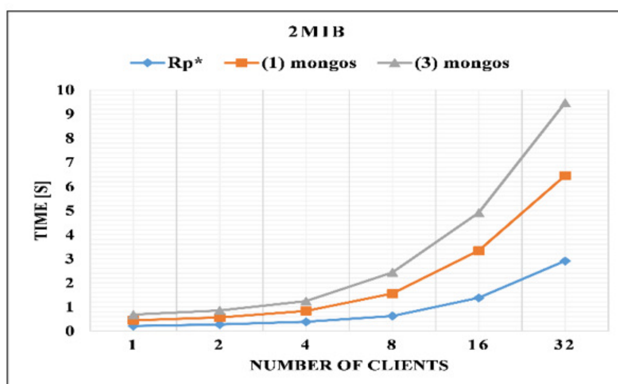

Fig. 4. 512 records of 2 MiB file size insertions.

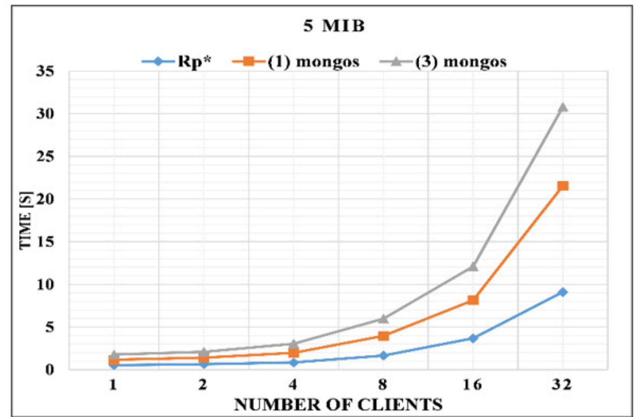

Fig. 5. 512 records of 5 MiB file size insertions.

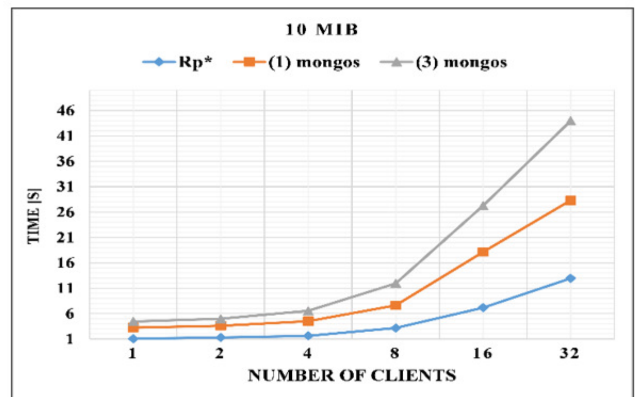

Fig. 6. 512 records of 10 MiB file size insertions.

Figure 7 portrays the insertion of 512 entries for a single client with one mebibyte (MiB) file size, demonstrating the rapid response to insertion requests. The graphical representations manifest that the insertion occurred at the point of splitting.

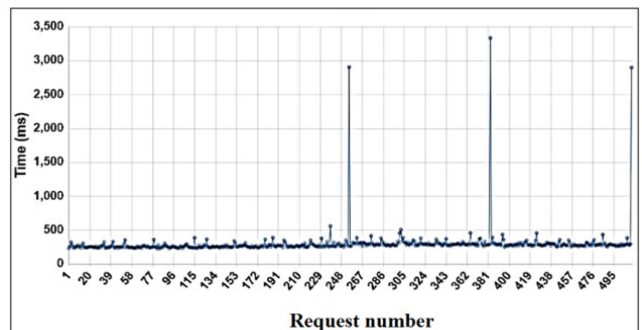

Fig. 7. 512 records of 10 MiB file size insertions for one client.

The configuration of clients allows for the input of multiple records, each of which is composed of 1 MiB. It is imperative that all insertions occur on the same server, specifically server 0. To achieve this, a scenario is established for each client, and

the triggers for these scenarios are identified in order for the waiting period and the split process to be determined. The system is designed to accommodate multiple insertions from a range of client counts with optimal efficiency. Upon reaching its capacity, a server node undergoes a rapid splitting process. In case an insertion request coincides with the splitting time, the system requires only a brief additional interval of a few milliseconds to complete the insertion. This constitutes a significant advantage over traditional systems, which require for operations to be halted until the splitting process is complete. The latter typically takes a considerably longer time.

V. CONCLUSION AND FUTRURE WORK

Range Partitioning Scalable Distributed Two-Layer Data Structures (RP* SD2DS) data store is scalable, distributed and supports range queries, improving both availability and scalability. It uses the RP* access technique within an SD2DS structure, optimizing the system for efficient data handling. The system consists of two tiers: one that stores keys with locators pointing to data in the second tier, improving availability during the partitioning process. It also expands dynamically without the need for a coordinator or master node. The RP*-SD2DS achieved superior performance over MongoDB in terms of record insertion for different numbers of clients (1, 2, 4, 8, 16 and 32), regardless of whether one or three Mongo instances are used. Future work must focus on extending RP*-SD2DS to handle large and complex multidimensional data.

REFERENCES

[1] A. Albugmi, "Digital Forensics Readiness Framework (DFRF) to Secure Database Systems," *Engineering, Technology & Applied Science Research*, vol. 14, no. 2, pp. 13732–13740, Apr. 2024, https://doi.org/10.48084/etasr.7116.

[2] C. Gomes, M. N. de O. Junior, B. Nogueira, P. Maciel, and E. Tavares, "NoSQL-based storage systems: influence of consistency on performance, availability and energy consumption," *The Journal of Supercomputing*, vol. 79, no. 18, pp. 21424–21448, Dec. 2023, https://doi.org/10.1007/s11227-023-05488-6.

[3] M. M. Sadeeq, N. M. Abdulkareem, S. R. M. Zeebaree, D. M. Ahmed, A. S. Sami, and R. R. Zebari, "IoT and Cloud Computing Issues, Challenges and Opportunities: A Review," *Qubahan Academic Journal*, vol. 1, no. 2, pp. 1–7, Mar. 2021, https://doi.org/10.48161/qaj.v1n2a36.

[4] G. Mahmood, N. Hassoon, H. N. Abed, and B. Jalil, "An Efficient and Secure Auditing System of Cloud Storage Based on BLS Signature," *International Journal of Computing and Digital System*, vol. 12, no. 01, pp. 1491–1501, Jul. 2021, https://doi.org/10.12785/ijcds/1201120.

[5] S. Amghar, S. Cherdal, and S. Mouline, "Which NoSQL database for IoT Applications?," in *2018 International Conference on Selected Topics in Mobile and Wireless Networking (MoWNeT)*, Jun. 2018, pp. 131–137, https://doi.org/10.1109/MoWNet.2018.8428922.

[6] F. Chang *et al.*, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 1–26, Jun. 2008, https://doi.org/10.1145/1365815.1365816.

[7] A. Ergüzen and M. Ünver, "Developing a File System Structure to Solve Healthy Big Data Storage and Archiving Problems Using a Distributed File System," *Applied Sciences*, vol. 8, no. 6, Jun. 2018, Art. no. 913, https://doi.org/10.3390/app8060913.

[8] A. Petrov, *Database Internals: A Deep Dive into How Distributed Data Systems Work*, 1st ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2019.

[9] M. B. Ahmad and S. Sagheer, "Issues and Algorithm of Distributed Shared Memory," in *2021 International Conference on Innovative Computing (ICIC)*, Lahore, Pakistan, Nov. 2021, pp. 1–9, https://doi.org/10.1109/ICIC53490.2021.9693062.

[10] W. Litwin, M.-A. Neimat, and D. A. Schneider, "LH: Linear Hashing for distributed files," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 327–336, Jun. 1993, https://doi.org/10.1145/170036.170084.

[11] K. Sapiecha and G. Lukawski, "Scalable Distributed Two-Layer Data Structures (SD2DS)," *International Journal of Distributed Systems and Technologies (IJDST)*, vol. 4, no. 2, pp. 15–30, Apr. 2013, https://doi.org/10.4018/jdst.2013040102.

[12] A. Krechowicz, A. Chrobot, S. Deniziak, and G. Łukawski, "SD2DS-Based Datastore for Large Files," in *Proceedings of the 2015 Federated Conference on Software Development and Object Technologies*, Cham, 2017, pp. 150–168, https://doi.org/10.1007/978-3-319-46535-7_13.

[13] A. Ali, S. Naeem, S. Anam, and M. M. Ahmed, "A State of Art Survey for Big Data Processing and NoSQL Database Architecture," *International Journal of Computing and Digital Systems*, vol. 14, no. 1, pp. 297–309, May 2023, https://doi.org/10.12785/ijcds/140124.

[14] W. Litwin, M.-A. Neimat, and D. A. Schneider, "RP*: A Family of Order Preserving Scalable Distributed Data Structures," in *Proceedings of the 20th International Conference on Very Large Data Bases*, San Francisco, CA, USA, Sep. 1994, pp. 342–353.

[15] W. Litwin, M.-A. Neimat, and D. A. Schneider, "LH*—a scalable, distributed data structure," *ACM Trans. Database Syst.*, vol. 21, no. 4, pp. 480–525, Dec. 1996, https://doi.org/10.1145/236711.236713.

[16] M. Bedla and K. Sapiecha, "Scalable Store of Java Objects Using Range Partitioning," in *Advances in Software Engineering Techniques*, Berlin, Heidelberg, 2012, pp. 84–93, https://doi.org/10.1007/978-3-642-28038-2_7.

[17] D. E. Zegour, "Scalable distributed compact trie hashing (CTH*)," *Information and Software Technology*, vol. 46, no. 14, pp. 923–935, Nov. 2004, https://doi.org/10.1016/j.infsof.2004.04.001.

[18] A. Mohammed, "Framework for Parallel Processing of Very Large Volumes of Data," *International Journal of Computing and Digital Systems*, vol. 08, no. 01, pp. 43–50, Jan. 2019, https://doi.org/10.12785/ijcds/080105.

[19] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008, https://doi.org/10.1145/1327452.1327492.

[20] A. Mohamed and D. Zegour, "TH*:Scalable Distributed Trie Hashing," *IJCSI International Journal of Computer Science Issues*, vol. 7, no. 6, pp. 109–115, Nov. 2010.

[21] S. Bradshaw, E. Brazil, and K. Chodorow, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*, 3rd ed. Beijing Boston Farnham: O'Reilly Media, 2019.

[22] S. Wang, G. Li, X. Yao, Y. Zeng, L. Pang, and L. Zhang, "A Distributed Storage and Access Approach for Massive Remote Sensing Data in MongoDB," *ISPRS International Journal of Geo-Information*, vol. 8, no. 12, Dec. 2019, Art. no. 533, https://doi.org/10.3390/ijgi8120533.