Design and Implementation of a ROLAP Cube in Scalable Distributed Data Structure

Amel Mechri
LEPCI Laboratory, Ferhat Abbas Setif 1 University, Algeria
amel.mechri@univ-setif.dz (corresponding author)

Bilal Bouaita
Salah Boubnider Constantine 3 University, Algeria | LEPCI Laboratory, Setif 1 University, Algeria
bilal.bouaita@univ-constantine3.dz

Djamel Eddine Zegour
LCSI Laboratory, High School of Computer Science (ESI), Algiers, Algeria
d_zegour@esi.dz

Walid Khaled Hidouci
LCSI Laboratory, High School of Computer Science (ESI), Algiers, Algeria
w_hidouci@esi.dz

Received: 21 November 2024 | Revised: 14 December 2024 | Accepted: 4 January 2025

Licensed under a CC-BY 4.0 license | Copyright (c) by the authors | DOI: https://doi.org/10.48084/etasr.9648

ABSTRACT

The Scalable Distributed Data Structure (SDDS) is a data model specifically designed for distributed environments. An SDDS file comprises records that are dynamically distributed across servers using an SDDS algorithm. A notable feature of SDDS is the removal of a centralized addressing component, simplifying client-server communication and reducing both the message count and data access time in distributed systems. This work also explores a Data Warehouse (DW) within a decision support system, where multidimensional data are represented as a cube and managed through Relational Online Analytical Processing (ROLAP). Although extensive research has been conducted in both the data warehousing and SDDS fields, no prior studies have combined these two areas. This paper introduces a novel approach to implementing a ROLAP cube within an SDDS using the Linear Hashing algorithm (LH*), which eliminates centralized addressing, enabling direct client-server communication and improving performance by reducing inter-site message exchanges. This work demonstrates the feasibility of this method and its positive impact on data processing efficiency in distributed systems.

Keywords-SDDS; dynamic linear hashing; linear hashing algorithm LH*; data warehouse; ROLAP cube

I. INTRODUCTION

The Scalable Distributed Data Structure (SDDS) is a concept that emerged in 1993, designed for multicomputers, particularly networks of interconnected workstations [1]. This type of data structure opened an important research area, as transparent data management is fundamental in computer networks [2]. SDDS is intended to be dynamic, with the number of nodes to which the file is distributed adapting to the amount of data stored. It is also scalable, meaning that the data can expand tremendously in size while maintaining usability and performance [3]. This requirement reflects the expectation that the size of application data will grow (or shrink) substantially over time [4].

An SDDS file consists of records distributed across the servers of a multicomputer and stored in buckets. Each record is identified by a unique key and each server handles a unique bucket with a fixed capacity. When a bucket reaches its capacity and a new record is assigned to it, it becomes overloaded. To address this, the server initiates a split operation, transferring half of the keys to a new server to make the file scalable. The server sites are accessed by autonomous clients, each having its own file image (information about the distribution of records in buckets) and being unaware of the existence of the other clients.

The main performance measure of a given operation in the SDDS paradigm is the number of point-to-point messages exchanged by the sites of the network to perform the operation [5]. The client uses its own addressing scheme to calculate the record's address and access to servers. For this reason, a centralized addressing component is not needed. This provides high performance by reducing the number of exchanged messages and the access time to distributed data, and evidently avoiding a bottleneck. Another measure of performance for a particular operation in the SDDS paradigm is the access time to the data. SDDSs are stored in the distributed main memory of the servers, which helps to reduce the access time compared to data stored in local disks.

To distribute data well on servers, three main techniques have been used to implement different SDDS approaches, including hashing-based using the LH* algorithm [1, 6-8] and its different variants, such as LH*LH [9, 10], LH*RS [11, 12], LH*RS in P2P [13, 14], EH*RS [15], IH* [16], LH*TH [17], LH* TT [18], interval-based [19-20], and digital-hashing-based [2] approaches. Linear hashing is an efficient and widely used version of extensible hashing [6]. The first SDDSs were primarily based on dynamic hashing with the work of W. Litwin, who proposed a generalization of his well-known and popular method (LH) for a distributed environment, more precisely for multicomputers [2]. LH* is its distributed version that stores key-value pairs on up to hundreds of thousands of sites in a distributed system. LH* implements the dictionary data structure efficiently by not using a centralized component and allows key-based operations such as insert, delete, update, and retrieve, as well as the scan operation [6]. Thereafter, several variants of LH* have been proposed, each improving a certain aspect of the method [2].

This study aimed to implement a ROLAP cube [21-25] in a distributed context based on the multidimensional SDDS LH* model:

- The logical data model is represented using a star scheme, characterized by a very large central fact table linked to smaller dimension tables.
- The physical implementation of data is organized into a ROLAP cube utilizing a relational model (relational databases).
- The data distribution across servers is managed using the LH*.

II. RELATED WORKS

The SDDS provides essential frameworks for efficiently managing data in distributed systems, ensuring scalability and performance. The first and most widely used SDDS algorithm was LH*. Since then, several models have been proposed reflecting the evolution and growing application of SDDS in various fields. For example, the LH*LH structure was introduced to improve performance on switched multicomputers [10], while linear hashing LH*LH was optimized on Windows NT [9]. SDDS usage was expanded to the SD-SQL Server database system [1]. LH*RS P2P enabled SDDS applications in peer-to-peer systems [13], and was later reinforced for churn resistance [14]. In [20], the efficiency of SDDS-based hierarchical HDHT structures was demonstrated for resource discovery in data grids, which is essential for scientific computing.

LH*TH [17] offers a fast and scalable structure, ideal for applications requiring high-throughput data access. In [5], SDDS was enhanced with an order-preserving model for efficient data management, known as ADST. Furthermore, the study in [18] contributed to the LH*TT model for temporal transaction scalability, while in [4] data integrity was emphasized by maintaining and verifying parity within SDDS.

SDDSs have also prioritized fault tolerance and high availability, as shown in the LH*RS and EH*RS models [11, 12, 15]. EH*RS ensures high availability by organizing data into record groups and applying Reed-Solomon error-correcting codes. In [7, 8], fault tolerance was strengthened with a decentralized record placement for LH*-based SDDS. Scalable Distributed Compact Trie Hashing (CTH*) [2] combines trie and hashing structures for efficient hierarchical data management. In [16], IH* was introduced, which is a new multidimensional SDDS based on hashing.

Three main techniques have been used to implement various SDDS approaches and distribute records across different servers. The first technique is distribution by hashing (linear hashing), where a hashing function is used to distribute data, with algorithms such as LH*, DDH*, EH*, and IH* [14]. The second technique is distribution by interval (range partitioning), where addresses are represented in a tree structure, and interval partitioning is performed in an ordered way according to the primary key. The algorithms used in this approach include RP*n, RP*c, RP*s, DRT, BDST, K-RP* [19], and LDT. The third technique, digital hashing, combines the simplicity and efficiency of hashing with the ordered record distribution characteristic of interval partitioning, using algorithms such as TH, CTH, and CTH* [2, 19].

In the context of data storage, access is managed by a Distributed Database Management System (DDBMS) [22]. Fact and dimension tables are managed as distributed databases, with inter-site communication facilitated by an intermediate central site known as the coordinator. This coordinator handles all communications between distributed sites and operates under a high workload to maintain these connections.

However, the centralization of inter-site coordination presents several major drawbacks in a distributed system. First, the coordinator can become a bottleneck, as all site-to-site communications must pass through it, leading to increased latency and reduced overall system throughput. This centralized approach also creates a single point of failure: if the coordinator experiences downtime, communication across the network is disrupted, compromising the system's availability and reliability. Moreover, the coordinator's central role in message routing and coordination can pose scalability issues, as the system may struggle to handle an increasing volume of messages or queries as the number of sites grows. As the coordinator's workload intensifies, its performance degrades, directly impacting the system's overall efficiency and response time.

III. THE PROPOSED APPROACH: SDDS_DW

This section presents Data Warehouse-based Scalable Distributed Data Structures (SDDS_DW) for the dynamic distribution of a file over servers over time and with increasing size (i.e. a record can change location and server). The proposed system is built on the same architecture as in classical data warehouses (based on MySQL): client, server(s), and coordinator (see Figure 1). However, there are major differences in the role of the coordinator.

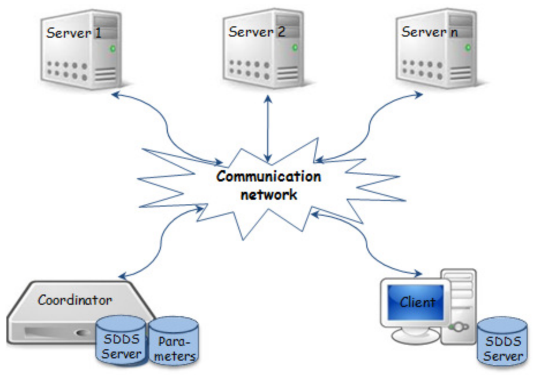

Fig. 1. The general architecture of the SDDS_DW system.

1) *Client*

The client sends insert or search queries to a server, which responds with a confirmation message for an insert query or a set of records for a search query. The client communicates directly with the server. It has its own image that collects information about the active server (address, communication port number).

2) *Server*

The rest of this article uses the term server instead of bucket. In the SDDS_DW system, there is initially a single-server site that can hold a limited number of records from the fact table. Records are stored in buckets (a bucket can contain *b* records). If the active server is overloaded (i.e. its storage capacity is exceeded when it manages *b* keys and a new record is assigned to it), new servers are added to host new records. The records are redistributed to the new servers by applying the LH* algorithm. Each server is identified by a logical number, an address, the port communication number, the total storage capacity, the storage space available for inserting new records, an indicator designating the active server, and another designating the server that will receive the new data records when the active server is exhausted.

3) *The Coordinator*

In an SDDS architecture, the coordinator only intervenes when an overflow occurs. It is the only one to have a correct image of the LH* file [8] and never intervenes in the query evaluation process [20]. The main functions of the coordinator are as follows [8]:

- Collects information about servers and implements file scaling strategy by maintaining an SDDSServers table containing the list of servers that store file records with their parameters.
- Maintains real LH* file parameters: The coordinator manages another table "Parameters". Grouping information (*i*, *n*, and *N*) is used to calculate record addresses during overflows to distribute them to the corresponding server.
- Controlls file splits and finds a new data server: Sends a split message to the server pointed with an *n* pointer. Receives split committing messages and updates file parameters.

B. Inserting a Query into the Fact Table

To insert or add one or more records, the client (which has its own image) extracts the parameters of the active server. Communication takes place directly between the client and the server without going through the coordinator. If there is no addressing error, i.e., the communicated server is actually the active server, there can be two cases: (i) the available storage space of the active server is greater than the number of records, and (ii) the available storage space of the active server is less than the number of records.

Note that an addressing error occurs if the information about the active server is not updated, so the actual values of the active server in the client's image do not match with the real values of the active server and, therefore, a query is sent to the wrong server. This information changes during overflows, where the active state passes from one server to another.

i. The available storage space of the active server is greater than the number of records: The client sends an insert query to add new records directly to the active server. The latter inserts them at his level and sends an insert confirmation message to the client.

ii. The available storage space of the active server is less than the number of records: Upon receiving the query, the server checks whether there is enough storage space to add all the new records. If the number of records is greater than the available storage space, an overflow occurs, and the active server contacts the coordinator by sending it an informative message. The coordinator extracts the address of the next server (server to be split) from its SDDS Servers table and sends it a split request message. The server in the overflow is not the server to be split, but the server with point n. Figure 2 illustrates the overflow scheme and server split.

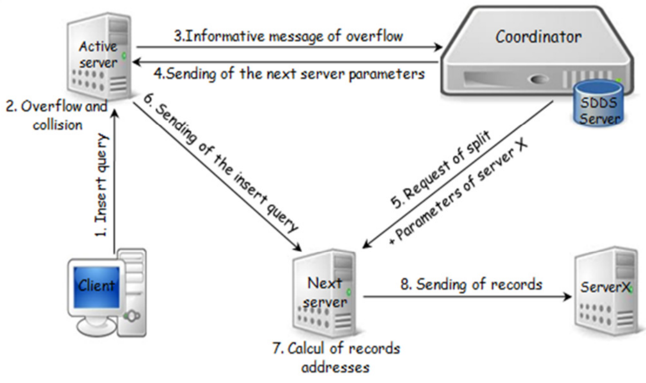
Active server
3.Informative message of overflow
4.Sending of the next server parameters
Coordinator
SDDS Server
2. Overflow and collision
6. Sending of the insert query
1. Insert query
5. Request of split + Parameters of server X
Client
Next server
8. Sending of records
ServerX
7. Calcul of records addresses

Fig. 2. Scheme of overflow and split of a server.

Each record has a key that is used to calculate its address. In the event of an overflow, the hash function h_i is used to calculate the addresses of all records (new and already stored on the next server), and about half of the records are transferred to a new server.

$$h_i(C) = C \bmod N * 2^i$$

where C is the record key, i is the file level (i = 0, 1, 2, 3, …), and N represents the number of buckets, initialized to 1.

The hashing function h_{i+1}, defined by $h_{i+1}(C) = C \bmod N * 2^{i+1}$, is applied to the records transferred to the new server as seen in Algorithm 1. After all records have been added to either the next server or the new server, a confirmation message is sent to the client. The coordinator updates the parameters: active and next server, i, n (n is the split pointer, indicating the next bucket to split and takes values 0, 1, 2, 3, …) and N. The new server will be the active server. The address of the next server is calculated by applying Algorithm 2.

```
Algorithm 1: Computation of the address of
the new data set
1:  $a = h_i(C)$  // $a$ is a bucket number
2:  If ($a < n$)
3:     $a = h_{i+1}(C)$
4:  Endif
```

The function h_i is applied to calculate the record's address a. If a is less than n, it means that the record of key C belongs to a bucket that has already been split, and therefore, the function h_{i+1} is applied to recalculate the record's address.

After each split, the counter n is incremented by 1 to indicate the address of the next bucket to split. The split of buckets in LH* occurs in a cyclical manner. n consecutively takes the values : $0 \dots N-1, 0 \dots 2*N-1$.

If $= 2^i * N$, then the total number of buckets has been exceeded, n is then reset to 0 (to point to the first bucket), and file level i is incremented. The new addressing function will be h_{i+1} instead of h_i.

```
Algorithm 2: Algorithm of updating the
file level $i$ and the split pointer $n$
1:  Create a new bucket
    // Recalculate the addresses of all
       keys of the bucket $n$
2:  $n = n + 1$
3:  If ($n >= 2^i * N$)
4:     $n = 0$
5:     $i = i + 1$
6:  Endif
```

C. Fact Search Query on the Fact Table

The client sends its search query to all the servers in multicast after it has extracted their parameters from its local table. Servers send their responses to the client, which recomposes them using a union operation. If the search query concerns a single record, the client calculates its address with the hash function to find its location (the hosting server). In this case, the request is sent only to the specified server. If there is an addressing error, the contacted server forwards the request to the correct server.

```
Algorithm 3: Search query algorithm
// Recalculate the addresses of all keys
   of the bucket $n$
1:  $a' = h_j(c)$
2:  If ($a' = a$)
3:    Accept $c$
4:  else
5:    $a'' = h_{j-1}(c)$
6:   EndIf
7:   If ($a < a''$) and ($a'' < a'$)
8:     $a' \leftarrow a''$
9:     Send $c$ to the bucket $a'$
10:  Endif
```

When a bucket receives a research request, it calculates the address using its parameters (the level j). If the address is the same, the bucket accepts the record, else it applies h_{j-1} to calculate the address.

IV. RESULTS AND DISCUSSION

This section presents the evaluation of the proposed approach using two well-known performance measures: the number of point-to-point messages exchanged by the sites of the network to perform the operation and the access time to the data. The advantage of SDDS_DW is that communication is performed directly between the client and the server without passing through an intermediate site, reducing the number of exchanged messages and the required time to execute a query.

A. Description of the Tested Data Model

The logical data model is represented according to the following star scheme.

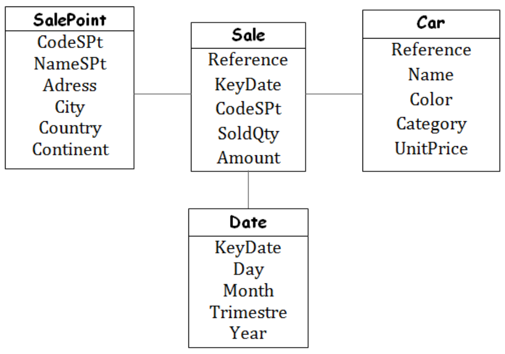

Fig. 3. A data model in star scheme.

The central fact table *Sale* relies on the dimension tables *SalePoint*, *Car*, and *Date* with foreign keys. All queries are directed to the fact table *Sale*, which undergoes new inserts or searches.

B. Comparison of the Number of Exchanged Messages

The number of exchanged messages is the first metric to demonstrate the performance of a system. Tables I and II show the number of exchanged messages between the client, servers, and the coordinator in the classic DW and the proposed approach.

TABLE I. NUMBER OF EXCHANGED MESSAGES IN A CLASSIC DW

	Client - Coordinator	Server - Coordinator	Total
Search query	2	2*nb	2+2*nb
Insert query without overflow	2	2	4
Insert query with overflow	2	2 or 4	4 or 6

TABLE II. NUMBER OF EXCHANGED MESSAGES IN SDDS_DW

	Server - Server	Client - Server	Server - Coordinator	Total
Search query	--	2*nb	--	2*nb
Insert query without overflow	--	2	--	2
Insert query with overflow	2	3	3	8

nb: Number of servers in the system that hold data.

The above tables show that the coordinator in a classic DW is overloaded because any communication or message exchanged between the client and a server necessarily passes through the server. If a significant amount of exchange traffic occurs, the coordinator becomes a bottleneck. Additionally, any failure of the coordinator will block communication between the client and servers. However, the number of exchanged messages between the client and servers in a classic DW is two times higher than that required in SDDS_DW for search or insert queries without overflow. In SDDS_DW, 2*nb and 2 messages are needed for a search or insert query without overflow, compared to 2 + 2*nb + 4 messages in a classic DW.

The issue in SDDS_DW arises with insert queries with overflow when records must be transferred between servers. In this case, messages are exchanged not only between the servers and the coordinator but also between servers, increasing the number of exchanged messages.

C. *Comparison of the Access Time to Data*

The second metric for demonstrating system performance is the time it takes to access data from sending a query to receiving the results. The curves below indicate the time taken to execute a search or insert queries with and without overflow.

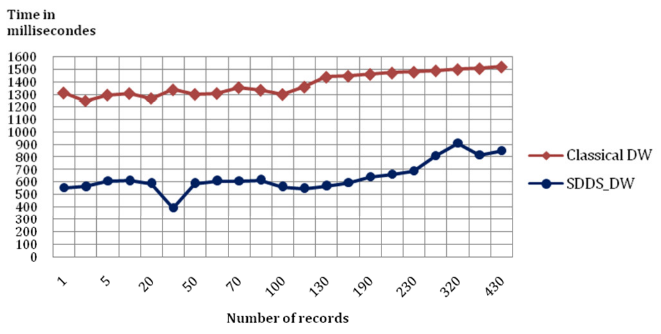

Fig. 4. Search in a DW.

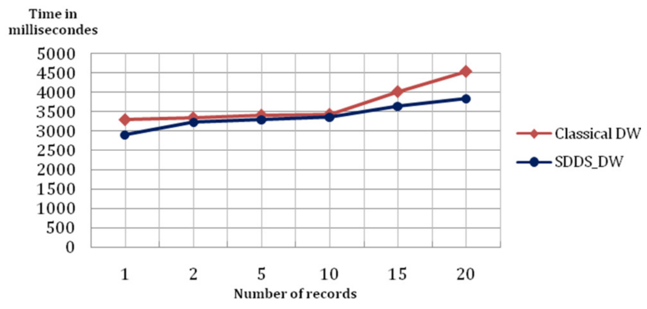

Fig. 5. Insertion in a DW without overflow.

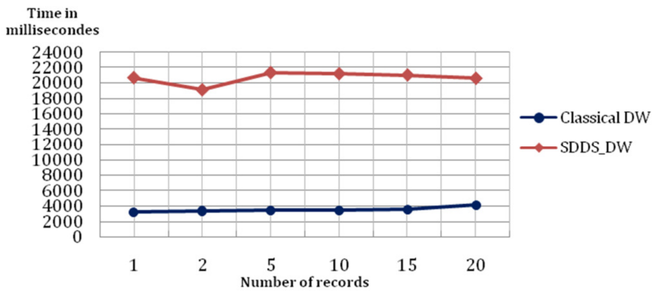

Fig. 6. Insertion in a DW with overflow.

As shown by the search query execution time curves, the time required for SDDS_DW is significantly shorter than that of a classic DW, with the time in SDDS_DW being reduced by more than half compared to a classic DW. This improvement is because, in SDDS_DW, communication occurs directly between the client and servers without the intervention of the coordinator. For an insert query without overflow, the execution time in SDDS_DW matches that of a classic DW. Furthermore, SDDS_DW enables direct client-server communication, thereby eliminating the need for a coordinator.

However, a primary drawback of SDDS_DW is the substantial execution time required for an insert query when a server becomes overloaded. In such cases, additional message exchanges occur between the client and the server, the server and the coordinator, and between the servers, significantly increasing the query execution time. This delay is further exacerbated by the need to recalculate the record addresses and redistribute them to a new server.

V. CONCLUSION

This study implemented a classic DW and a new approach using the SDDS_DW model with the LH* algorithm. A key advantage of SDDS is the absence of a centralized addressing component, allowing direct client-server communication that enhances performance by reducing message exchanges across multiple sites. The results show that for search or insert queries without overflow, SDDS_DW outperforms the classic DW in terms of fewer messages exchanged and faster query execution times. However, SDDS_DW faces a limitation when handling insert queries with overflow, as it requires more message exchanges and longer execution times due to the recalculation and redistribution of record addresses.

To address the overflow issue, a storage threshold setting is proposed for each server, triggering the redistribution of records before new insert queries are processed, thereby enabling periodic DW updates independent of client-initiated inserts. Future work will focus on resolving addressing errors and query redirection. In addition, alternative distribution methods, such as interval-based distribution or digital hashing, will be explored to implement the ROLAP cube and compare them to identify the most effective approach and optimal algorithm.

REFERENCES

[1] W. Litwin, S. Sahri, and T. Schwarz, "An Overview of a Scalable Distributed Database System SD-SQL Server," in *Flexible and Efficient Information Handling*, 2006, pp. 16–35, https://doi.org/10.1007/11788911_2.

[2] 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.

[3] E. Erturk and K. Jyoti, "Perspectives on a Big Data Application: What Database Engineers and IT Students Need to Know," *Engineering, Technology & Applied Science Research*, vol. 5, no. 5, pp. 850–853, Oct. 2015, https://doi.org/10.48084/etasr.592.

[4] D. Cieslicki, S. Schaeckeler, and T. Schwarz, "Maintaining and checking parity in highly available Scalable Distributed Data Structures," *Journal of Systems and Software*, vol. 83, no. 4, pp. 529–542, Apr. 2010, https://doi.org/10.1016/j.jss.2009.10.013.

[5] A. Di Pasquale and E. Nardelli, "A Very Efficient Order Preserving Scalable Distributed Data Structure," in *Database and Expert Systems Applications*, 2001, pp. 186–199, https://doi.org/10.1007/3-540-44759-8_20.

[6] J. Chabkinian and T. J. E. Schwarz SJ, "Fast LH*" *International Journal of Parallel Programming*, vol. 44, no. 4, pp. 709–734, Aug. 2016, https://doi.org/10.1007/s10766-015-0371-8.

[7] G. Łukawski and K. Sapiecha, "Fault Tolerant Record Placement for Decentralized SDDS LH*," in *Parallel Processing and Applied Mathematics*, 2008, pp. 312–320, https://doi.org/10.1007/978-3-540-68111-3_33.

[8] K. Sapiecha and G. Lukawski, "Fault-Tolerant Protocols for Scalable Distributed Data Structures," in *Parallel Processing and Applied Mathematics*, 2006, pp. 1018–1025, https://doi.org/10.1007/11752578_123.

[9] F. Bennour, A. Diène, Y. Ndiaye, and W. Litwin, "Scalable and distributed linear hashing LH∗ LH under Windows NT," in Proceeddings of the IEEE Fourth World Multiconference: Systems Cybernetics & Informatics and Information Systems Analysis & Synthesis, Orlando, FL, USA, 2000, pp. 23–26.

[10] J. S. Karlsson, W. Litwin, and T. Risch, "LH*lh: A scalable high performance data structure for switched multicomputers," in *Advances in Database Technology — EDBT '96*, 1996, pp. 573–591, https://doi.org/10.1007/BFb0014179.

[11] W. Litwin, R. Moussa, and T. J. E. Schwarz, "LH*$_{RS}$: a highly available distributed data storage," in *Proceedings of the Thirtieth International Conference on Very Large Databases - Volume 30*, Toronto, Canada, May 2004, pp. 1289–1292.

[12] W. Litwin, "LH*RS: A Highly Available Distributed Data Storage," in *Proceedings of the 30th VLDB Conference*, Toronto, Canada, Jan. 2004.

[13] W. Litwin, H. Yakouben, and T. Schwarz, "LH*RSP2P: a scalable distributed data structure for P2P environment," in *Proceedings of the 8th international conference on New technologies in distributed systems*, New York, NY, USA, Mar. 2008, pp. 1–6, https://doi.org/10.1145/1416729.1416731.

[14] H. Yakouben and S. Soror, "LH*RSP2P: a fast and high churn resistant scalable distributed data structure for P2P systems," *International Journal of Internet Technology and Secured Transactions*, vol. 2, no. 1–2, pp. 5–31, Jan. 2010, https://doi.org/10.1504/IJITST.2010.031470.

[15] X. Ren and X. Xu, "EH*RS: A High-Availability Scalable Distributed Data Structure," in *Algorithms and Architectures for Parallel Processing*, 2007, pp. 188–197, https://doi.org/10.1007/978-3-540-72905-1_17.

[16] D. Boukhelef and D. E. Zegour, "IH*: A New Hash-Based Multidimensional SDDS," presented at the WDAS 2002.

[17] M. Aridj, "LH* TH: New fast Scalable Distributed Data Structures (SDDSs)," *International Journal of Computer Science Issues (IJCSI)*, vol. 11, no. 6, pp. 123-128, 2014.

[18] M. N. Issaoui and R. Bouaziz, "SDDS LH* TT: Une solution pour la scalabilité d'une relation temporelle de transaction standard," presented at the 5th International Conference: Sciences of Electronic, Technologies of Information and Telecommunications, Mar. 2009.

[19] M. Maabed, N. Dennouni, and M. Aridj, "Optimizing Data Availability and Scalability with RP*-SD2DS Architecture for Distributed Systems," *Engineering, Technology & Applied Science Research*, vol. 14, no. 5, pp. 16178–16184, Oct. 2024, https://doi.org/10.48084/etasr.8176.

[20] R. Mokadem, F. Morvan, and A. Hameurlain, "SDDS Based Hierarchical DHT Systems for an Efficient Resource Discovery in Data Grid Systems," in *The Semantic Web: ESWC 2012 Satellite Events*, 2015, pp. 327–342, https://doi.org/10.1007/978-3-662-46641-4_25.

[21] K. Boukhalfa, Support de cours Entrepôts et fouille de données, Université des sciences et de la Technologie Houari Boumediene USTHB, Alger, Algeria, 2024.

[22] L. Chouder, "Entrepôt Distribué de Données," M.S. Thesis, Institut National d'Informatique, INI, Alger, Algeria, 2007.

[23] M. F. Masouleh, M. A. A. Kazemi, M. Alborzi, and A. T. Eshlaghy, "A Genetic-Firefly Hybrid Algorithm to Find the Best Data Location in a Data Cube," *Engineering, Technology & Applied Science Research*, vol. 6, no. 5, pp. 1187–1194, Oct. 2016, https://doi.org/10.48084/etasr.702.

[24] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*. Springer International Publishing, 2020.

[25] A. Vaisman and E. Zimányi, *Data Warehouse Systems: Design and Implementation*. Springer, 2022.