PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH
HASSIBA BENBOUALI UNIVERSITY OF CHLEF (UHBC), Algeria
Faculty of Exact and Computer Sciences
Department of Computer Science

THESIS

Submitted for the diploma of

DOCTORAT

Field: **Computer Science**
Speciality: **Information System**

By: **MAABED Mohammed**

Title:

Towards a Multidimensional NOSQL DBMS Based on Dynamic and Scalable Data Distribution

.	TAHAR ABBES Mounir	Professor	UHBC-Chlef	Chairman
.	DENNOUNI Nassim	M.C.A	Higher school of management-tlemcen	Supervisor
.	ARIDJ Mohamed	M.C.A	UHBC-Chlef	Co-Supervisor
.	BECHAR Rachid	M.C.A	UHBC-Chlef	Examiner
.	LOUAZANI Ahmed	M.C.A	University Saad Dahlab Blida1	Examiner
.	SABRI Mohamed	M.C.A	ENPO-Oran	Examiner

UHBC Algeria 2024-2025

For thesis, just do it for the most precious people in your life. Especially, "FAMILLY", and make them proud of you.

Mohammed MAABED

To my beautiful mom, my role model, Dad, my dear wife, my children who resemble me, my brothers and sisters at first and last, I did it for you all. I will never forget your support and sacrifices for me. I love you more than the world. Please be proud of your:
select (son, husband, dad, brother) as me;

Mohammed MAABED

Acknowledgement

First, I praise Allah, the Almighty, whose grace and wisdom have made this work possible.

Second, I would especially like to express my gratitude to my supervisors, Mr. DENNOUNI Nassim and Mr. ARIDJ Mohamed, for their competent guidance, patience, humility, concern, and encouragement. Their critical insights were invaluable in helping me refine my contributions. I am also thankful to the laboratory director and his successor for their support. I would like to extend my sincere thanks to the jury members, Mr. TAHAR ABBES Mounir, Mr. BECHAR Rachid, Mr. LOUAZANI Ahmed, and Mr. SABRI Mohamed, for the honor they have bestowed upon me by dedicating their time to read, evaluate, and provide invaluable advice on this work. Their contributions have been significant not only to me but to all present on this Scientific Advisory Board.

I would also like to thank the doctoral students: Chenaoui Ali, Bettach Djalloul, Chaib Mostapha, Lotfi Bendiaf, and Bouhdjeur Brahim, for their encouragement, support, and assistance. Additionally, my dear friends Djalal, Rabi3, Marwan, Abdelhaq, and Mohamed have been a source of strength and motivation.

I am grateful to the teaching and administrative staff of the UHBC University of Chlef for their dedication in providing us with an excellent training program.

Finally, I would like to thank everyone who has contributed in any way to the realization of this work.

Abstract

This work introduces a new paradigm for Multi-Dimensional NoSQL, namely MD-NOS, which targets efficient data management in IoT, Fog, and cloud platforms. It proposes an overall scalable and fault-tolerant architecture that includes routing in IoT clients. The fog layer namely KV-MDSS, and a cloud layer, namely SD-PGSQL. Dynamic partitioning techniques are integrated into the system for load balance and query performance optimization; for single-key data, RP*-SD2DS is adopted, and ZK-RP* for multi-dimensional data.

It gives an overview of the basics of SDDS and then goes further to show some realizations of these efficiently using MD-NOS. It presents two specialized architectures: RP*-SD2DS, optimized for one-dimensional data, and ZK-RP*, which extends it further for multi-dimensional data. Experimental results prove the supremacy in performance and scalability of the system when compared to traditional NoSQL databases like MongoDB with large files and complex queries.

It gives reason to believe that MD-NOS can enable a revolution in the management of data.

Résumé

Ce travail introduit un nouveau paradigme pour le NoSQL multidimensionnel, nommé MD-NOS, qui cible la gestion efficace des données dans les plateformes Fog et cloud. Il propose une architecture globale évolutive et tolérante aux pannes qui inclut des clients IoT avec une couche Fog, nommé KV-MDSS, et une couche cloud, nommé SD-PGSQL. Des techniques de partitionnement dynamique sont intégrées dans le système pour équilibrer la charge et optimiser les performances des requêtes; pour les données à clé unique, RP*-SD2DS est adopté, et ZK-RP* pour l'indexation multidimensionnelle.

Il donne un aperçu des principes de base du SDDS et va plus loin en montrant quelques réalisations de ces principes à l'aide de MD-NOS. Il présente deux architectures spécialisées : RP*-SD2DS, optimisée pour les données unidimensionnelles, et ZK-RP*, qui l'étend aux données multidimensionnelles. Les résultats expérimentaux prouvent la suprématie du système en termes de performances et d'évolutivité par rapport aux bases de données NoSQL traditionnelles, telles que MongoDB, avec des fichiers volumineux et des requêtes complexes.

Contents

List of Figures

List of Tables

Table of Abbreviations (Sigles)

Abbreviation	**Definition**
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
B+ Tree	Balanced Tree Data Structure
CAP	Consistency, Availability, Partition Tolerance
CTH	Compact Trie Hashing
DDH	Distributed Dynamic Hashing
DRT	Dynamic Range Tree
EH	Extendible Hashing
IAM	Image Adjustment Message
IoT	Internet of Things
K-D Tree	K-Dimensional Tree
KRP	Key-Range Partitioning Metadata Table
LH	Linear Hashing
MDI	Multi-Dimensional Index
MD-NOS	Multi-Dimensional NoSQL based On SDDS
MR2P	MapReduce with RP Integration
NoSQL	Not Only SQL
PUT	Insertion Operation
RDBMS	Relational Database Management System
RP	Range Partitioning
RP-SD2DS	Range Partitioning Scalable Distributed Two-Layer Data Structure
SD2DS	Scalable Distributed Two-layer Data Structure
SD-PGSQL	Scalable Distributed PostgreSQL
SDDS	Scalable Distributed Data Structures
SDI	Storage Data Index Layer
TH	Trie Hashing
ZK-RP	Z-Order K-Dimensional Range Partitioning

List of Publications

1. **M. Maabed**, N.Dennouni, M.Aridj, (2024). Optimizing Data Availability and Scalability with RP*-SD2DS Architecture for Distributed Systems. Engineering, Technology & Applied Science Research, 14(5), 16178-16184.

2. **M. Maabed**, M. Aridj, N. Dennouni, (2023). Two-Layer-based Datastore Split Process. National Conference on Artificial Intelligence, Smart Technologies and Communications (AISTC'23), 1. ISSN: 1112-9778 – EISSN: 2437-0312.

Chapter 1

General Introduction

1.1 Motivations

Nowadays, the growth of data due to technological advancement, digitization, and global connectivity has drastically changed the face of data management. Modern applications generate data at a volume, variety, and velocity that conventional RDBMS are not able to cope with. Limitations of RDBMS in managing unstructured and semi-structured data coupled with increasing demand for real-time analytics and high availability have given birth to NoSQL databases.

NoSQL can extend flexibility and scalability beyond those of traditional systems. On the other hand, the variety present in the category of NoSQL from pure key-value stores through document stores and column-family databases to graph databases makes selecting an appropriate NoSQL solution significantly complicated for organizations that want effectively to deploy it.

Also, the advances in IoT, Fog computing, and cloud computing introduced new challenges concerning data handling. These computing environments require the development of systems that can integrate several layers, offering scalability, fault tolerance, and efficient data processing to meet modern applications characterized by large volumes of data.

1.2 Contributions

This thesis proposes an enhanced multi-dimensional NoSQL framework that will be able to integrate the IoT, Fog, and cloud layers seamlessly within one logical architecture. The main contributions of this work are:

Hierarchical Architecture: The proposed framework harmonizes the IoT clients with the decentralized Fog layer, called KV-MDSS, and the cloud layer, now referred to as SD-PGSQL, for optimal trade-offs between scalability, availability, and consistency. This hierarchical architecture ensures the efficient distribution of data and its processing for querying, especially in big data analytics.

Dynamic Partitioning The framework embraces novel dynamic partitioning techniques, where RP*-SD2DS is used for single-key data, while ZK-RP* is applied in multi-dimensional indexing. This encourages new heights of efficiency in the handling of large datasets and complex queries.

Advanced Architectures RP*-SD2DS and ZK-RP* architectures introduced for one-dimensional and multi-dimensional data respectively considerably enhanced the capability of the system to handle complex queries along with huge volumes. These architectures assure better performance in distributed settings.

Superior Performance: Experimental results show the proposed system performs better than traditional NoSQL databases in performance, particularly for large-sized file storage and high-velocity data streams. The scalability and efficiency of the system prove it as a robust solution to modern data management challenges.

Intelligent Query Routing: IoT clients, in the system, maintain partial images of data distribution and support consistency-aware intelligent query routing for the optimization of consistent and non-consistent query performance to effectively retrieve and process data.

1.3 Outline

The present thesis provides a Multi-Dimensional NoSQL system, MD-NOS, to solve major challenges in the modern management of data between IoT, Fog, and cloud

computing. Further, this thesis initiates an introduction in Chapter 1, presenting the background of the problem and motivation for the conducted research. This is followed by Chapter 2, focusing on NoSQL systems, reviewing types, classes, and most importantly, the CAP theorem.

It ranges from unidimensional and multi-dimensional SDDS (including hash-based and order-preserving techniques) to more advanced SDDS variants, such as SD2DS. Chapter 3 covers the core principles, families, and applications of SDDS. Chapter 4 introduces the MD-NOS global architecture; it describes how to integrate IoT, Fog, and cloud layers through dynamic partitioning techniques like RP*-SD2DS and ZK-RP*.

Chapter 5 presents KV-MDSS, an architecture of the key-value store for multi-dimensional data. This chapter also provides two specialized architectures: RP*-SD2DS for one-dimensional and ZK-RP* for multi-dimensional data. This chapter also presents high availability and fault tolerance for the system. Chapter 6 depicts performance and scalability studies of MD-NOS by using simulations and a comparison with traditional NoSQL databases to show its superior efficiency.

Chapter 2

NoSQL Systems

2.1 Introduction

Data grows very fast. This growth changes how information is stored. It also changes how information is processed and analyzed. Traditional relational database management systems (RDBMS) were widely used in the past. Now, these systems cannot meet modern needs. Modern applications handle large amounts of data. They require high speed. They also work in different formats.

NoSQL databases have become a solution. NoSQL means "Not Only SQL." These databases provide flexibility. They also provide scalability and performance. NoSQL databases work well in situations where traditional RDBMSs fail.

NoSQL systems support many uses that include real-time analytics, IoT applications, e-commerce platforms and big data processing. NoSQL databases use non-relational data models such that key-value stores, document stores, column-family databases, and graph databases. This approach allows flexible schema design. It also allows horizontal scalability. As a result, NoSQL systems are important in distributed environments. They are also important in high-performance environments.

2.1.1.1 CAP Theorem

One important idea is the CAP theorem. Distributed database systems are affected (see figure 2.1). Three qualities are the main emphasis of the theorem. These are partition tolerance, availability, and consistency. These characteristics are significant. Non-relational database design is made possible by them. A limitation is stated by the theorem. All three properties cannot be guaranteed simultaneously in any distributed system. Only two of these three characteristics can be attained by a system. As a result, sacrifices are required. The application's requirements determine the compromises. This restriction affects distributed databases in the current era. It is especially crucial for systems. These systems must be extremely fault-tolerant and scalable [84].

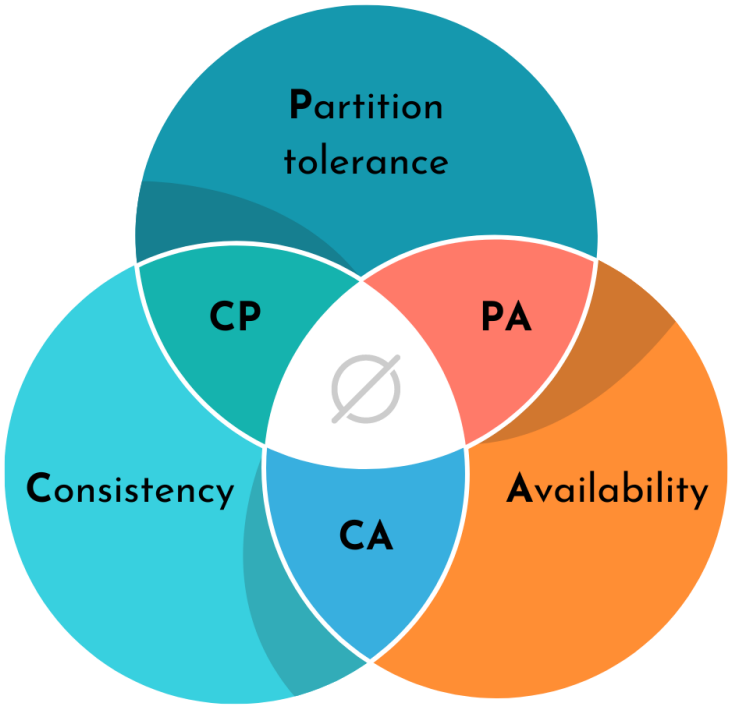

Figure 2.1: CAP theorem

a) **Consistency:** This characteristic ensures specific behavior. Each read operation reflects the most recent write operation. In practice, changes must be sent to all data nodes to maintain consistency. This ensures all nodes always have the same updated data. Users get the same results no matter which node

they access. However, achieving consistency can increase latency. It can also add complexity. This is especially true in systems with high transaction rates or nodes that are far apart [84].

b) **Availability:** The capacity of the system to continue being usable and accessible is the major focus of availability. Even if nodes malfunction or are interrupted, they still function. A service node's continued functionality is guaranteed by this attribute. User queries are processed by this node. Redundancy and failover techniques are employed by the system. These systems ensure continuous service. Prioritizing availability, however, may menace consistency. To stay responsive in the case of an interruption, nodes may serve outdated data [62], [93].

c) **Partition Tolerance:** Partition tolerance ensures that the system works correctly. It even works in the event of a network failure or partition. For example, it works when nodes lose connection for a short period. This feature is important for distributed systems. It is very important in unreliable networks. It keeps the system running. It also maintains access to data in the event of a communication failure. However, achieving partition tolerance often requires compromises. These compromises have an impact on consistency and availability. The system must make design choices. These choices enable data synchronization and error recovery to be properly managed [68], [91].

The CAP Theorem shows trade-offs in distributed system design. It helps architects and developers prioritize properties. These properties depend on the needs and limits of their applications. The principles of the CAP Theorem provide a framework. This framework helps understand challenges. It also helps understand complexities. These challenges and complexities relate to building reliable, scalable, and efficient database solutions.

2.1.1.2 BASE Properties

BASE's database model stands for Basically Available, Soft State, Eventual Consistency. It provides a scalable and adaptable substitute. The conventional ACID

(Atomicity, Consistency, Isolation, and Durability)[72] model is replaced by this one. Strict consistency and transaction guarantees are the main goals of the ACID paradigm. Scalability and availability are given top priority in the BASE model. BASE is appropriate for distributed systems because of this. There must be a high level of fault tolerance in distributed systems. The BASE paradigm has been widely used in contemporary NoSQL databases. Processing massive amounts of data is the main goal of these databases. Additionally, they ensure system responsiveness in various scenarios. Long-term consistency is sacrificed for immediate consistency in the BASE model. Reliability and system performance are balanced in this trade-off. It performs well in settings with a lot of dispersion [31].

a) **Basically Available:** The availability is the main focus of the BASE model. It guarantees that the system will continue to function and be available. Even if some of it fails, it still functions. Strict consistency is what the ACID model strives for. Temporary discrepancies are permitted by the BASE model. It makes it possible for the system to stay responsive. Data is replicated over numerous nodes in the design. In the case of a local failure, it permits the system to function to some extent. Preventing complete system failure is the primary goal. This results in a robust and fault-tolerant solution. Availability is given precedence above instantaneous consistency [34], [96].

b) **Soft State:** Soft state refers to the transient nature of the data in a BASE system. In this model, data is allowed to change or evolve over time without requiring immediate consistency across all nodes. This flexibility enables the system to store and retrieve volatile data efficiently, ensuring that operations are minimally affected by temporary inconsistencies. Soft state also allows data to be regenerated or restored in case of failures, leveraging mechanisms such as replication and eventual synchronization. This property reduces the overhead associated with strict consistency enforcement, making it ideal for systems that prioritize performance and scalability [30], [72].

c) **Eventual Consistency:** Eventual consistency is a key feature of the BASE model. It ensures that all nodes reach a consistent state over time. Temporary inconsistencies are allowed during synchronization. This enables the system to

better handle important data updates and network problems. Predefined rules and mechanisms ensure data consistency between nodes. This approach can lead to delays in consistency. However, it balances performance and reliability. It works particularly well in distributed systems requiring high availability [57], [96].

The BASE model changes how databases are designed. It trades consistency for availability. This fits the needs of modern distributed systems. BASE focuses on availability, soft state, and eventual consistency. It provides a framework for scalable and fault-tolerant databases. These databases meet the demands of large, dynamic applications.

2.1.1.3 NoSQL Systems

NoSQL is a non-relational database system. It has a flexible schema. It handles distributed data efficiently. It works well for applications with large volumes of data or real-time data. Logs and web applications are examples. Unlike relational databases, NoSQL supports structured, semi-structured and unstructured data. It overcomes many of the limitations of relational databases. It is particularly useful for large, variable data sets. NoSQL databases are horizontally extensible. They adapt well to hierarchical data models[63], [76].

NoSQL offers a number of key advantages. These characteristics improve ease of use and efficiency. NoSQL databases can run on devices with low specifications [50], [83]. They do not require a high-speed Internet connection or significant computing resources. They are therefore easy to use and deploy in a variety of environments. They offer high performance. They support flexible, elastic scaling. They are uncomplicated and guarantee minimal latency. NoSQL systems are fault-tolerant. They are highly available. They are reliable, even in distributed environments [49], [71].

NoSQL databases are effective in some situations. Traditional relational database characteristics like ACID compliance are not necessary for them. Applications with flexible schemas are the perfect fit for them [55], [63]. Strict database constraints or validations are not necessary for these applications. NoSQL databases

are good at handling transient or ephemeral data. They incorporate a variety of data kinds with ease. They are excellent at handling and preserving vast amounts of data. They are therefore an excellent option for real-time analysis and big data applications [35], [83].

2.1.1.4 Types and Classifications of NoSQL Databases:

The rise of database technology reflects a major evolution in data management. This evolution is driven by the need to handle more data, faster data and diverse data types in modern applications. Traditional relational databases use rigid schemas. NoSQL databases, on the other hand, offer flexible models. These models are adapted to the dynamic nature of today's data. There are four main types of NoSQL database: key-value databases, document databases, columnar databases and graph databases. Each type has unique characteristics, advantages and use cases. These characteristics make them worth exploring for analysis purposes [51], [91].

2.1.1.5 Key-Value Stores

Key-value stores are simple NoSQL databases. Data is stored as key-value pairs. Each key is unique. Keys help retrieve the corresponding value. This allows fast data access. The simplicity improves performance. It works well for reading and writing operations. Figure 2.2 represents an example of how the data are represented [42], [87].

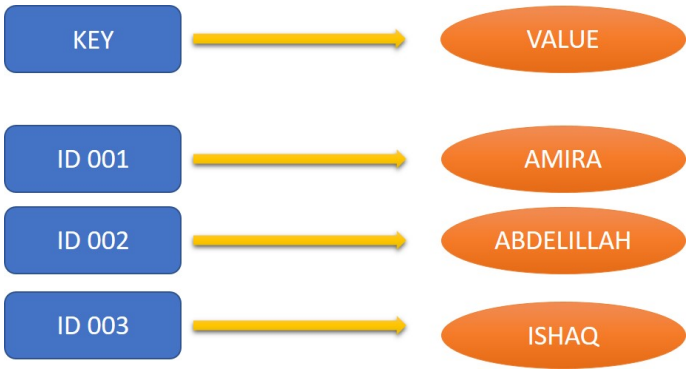

Figure 2.2: Key-Value Store.

The advantages of this model include horizontal scalability and the ability to handle massive amounts of data easily. These features make key-value stores suitable for applications such as cache storage, session management, and user preferences, where quick access to data is essential. Popular systems such as Redis and Amazon DynamoDB exemplify successful implementations of this concept [58], [73].

The following is a detailed explanation of some key-value store systems:

1. **Redis:** Redis is an open-source key-value store. It works in memory. It is known for high performance and versatility. It supports advanced data types like lists, sets, and hashes. This makes it suitable for caching, real-time analytics, and session storage. Its in-memory design ensures low-latency operations[26], [73].

2. **Amazon DynamoDB:** Amazon DynamoDB is a managed key-value store. It is cloud-based. It offers automatic scaling and data replication. It ensures high availability and fault tolerance. It is ideal for applications needing low-latency access and global scalability. Examples include gaming and IoT systems [43], [73].

3. **Riak:** Riak is a distributed NoSQL key-value database. It is designed for high availability and fault tolerance. Its peer-to-peer architecture handles large datasets efficiently. It provides eventual consistency. It is commonly used in applications needing strong scalability. Examples include messaging systems[53], [73].

4. **Aerospike:** Aerospike is a fast and scalable key-value database. It operates in memory. It uses hybrid memory architectures. It supports real-time data processing. It is perfect for fraud detection, financial transactions, and advertising technology platforms [59], [73].

2.1.1.6 Document Stores

Document stores expand key-value stores. They store semi-structured data as documents. Documents are often in JSON or XML format. Each document is self-

describing. It can have variable fields and data types. This improves data modeling flexibility [36], [66]. Figure 2.3 shows how the data are organized,

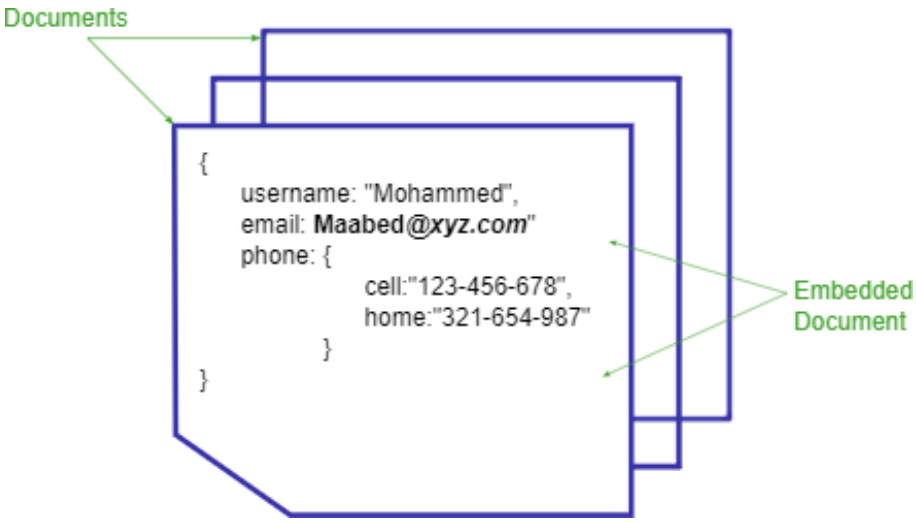

Figure 2.3: Document Store.

This feature helps developers adapt to changing needs. They avoid expensive migrations. Document stores excel in content management systems, e-commerce platforms, and real-time analysis. MongoDB is a popular example. They offer strong query capabilities. Developers prefer them for more complex needs than key-value stores provide [32], [94].

The following is a list of some document-oriented based systems:

1. **CouchDB:** CouchDB is a NoSQL open-source database. It is document-oriented. It is known for its robustness and simplicity. It uses JSON format to store data. It provides robust replication functions. It treats data using RESTful HTTP and JavaScript. It manages offline applications well. It is suitable for mobile and web applications [46], [101].

2. **MongoDB:** MongoDB is a popular cross-platform, document-oriented database that uses JSON-like documents with dynamic schemas. It offers advanced features such as ad hoc queries, indexing, replication, load balancing, and transactions. This flexibility and scalability make it a preferred choice for real-time analytics, e-commerce, and IoT applications [41], [105].

3. **Terrastore:** Terrastore is a distributed, document-oriented database designed for scalability and consistency. It ensures per-document consistency while

enabling distributed storage and processing. Its architecture is well-suited for large-scale applications requiring data synchronization and consistent querying [28], [66].

2.1.1.7 Column-Family Stores

Column family stores, such as Apache Cassandra and Hbase, use an optimized structure to organize data in column families with high writing and reading performance. This design allows you to recover data subsets efficiently, which is particularly beneficial when it comes to large data sets and analytical work sets [29], [80]. Figure 2.4 illustrates the scheme of the representation of row and column.

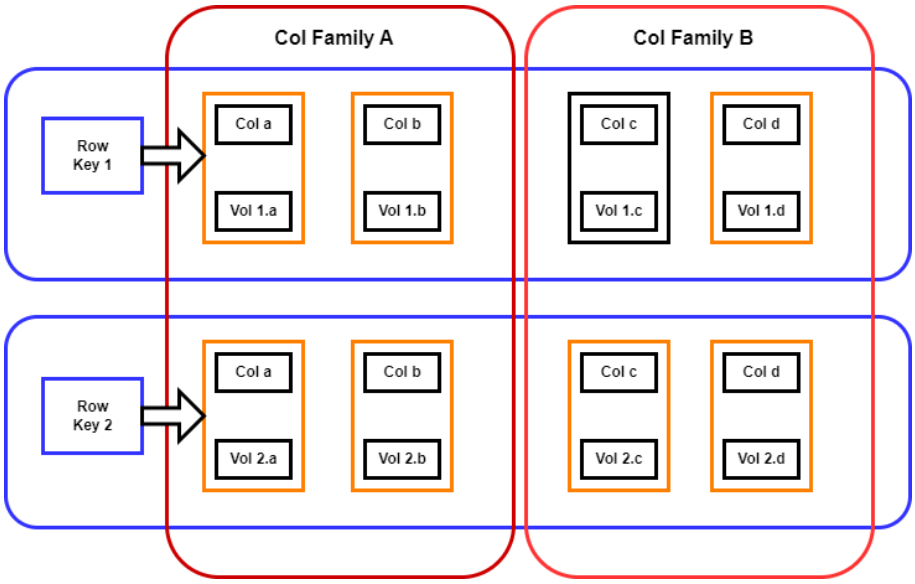

Figure 2.4: Column-Family Store.

The main advantage of column family stores lies in their ability to store and recover large volumes of dispersed data with variable structures while allowing the horizontal scale. Their suitability for applications that involve time series data, registration data and processing data of high speed positions them as essential tools in the analysis domain. In addition, the architecture of column family stores supports complex consultations in multiple data dimensions [39], [94], improving their versatility in various environments.

In what follow some of the systems based on the column-family stores:

1. **HBase:** this is an open-source database system based on HDFS, with low la-

tency because it allows random access and supports real-time data processing. Hbase is designed to scale linearly for high-volume data. It offers powerful features such as compression and in-memory processing, with the main objective of this technology being to provide a table-like architecture.

2. **Hypertable :** a database management system open-source database management system founded by Google, and modelled on Google Big Table, which is scalable and compatible with multiple distributed storage systems. Compatible with multiple storage systems distributed storage systems. It offers high performance and scalability by dividing tables into several key-value pairs, which are distributed between different storage units (shards) [23], [33].

3. **Cassandra :** a column-oriented open source distributed database, founded by Facebook, providing a scalable database capable of handling huge amounts of data with adjustable consistency, making Cassandra a consistent database [48], [105].

4. **Accumulo :** a big table open source distributed database, which can run on top of Hadoop and Zookeeper [63], and provides additional features such as compression and cell-level access management. It is commonly used when cell-level access is required [45], [101].

2.1.1.8 Graph Databases

Graph databases, such as Neo4J, add another layer of complexity by modeling data as interconnected nodes and edges. This structure allows for the representation of relationships and makes it incredibly efficient to traverse connections in datasets [40], [85].

Figure 2.5 represents a graph database, illustrating the significance of arcs and nodes.

By leveraging graph theory, these databases enable efficient queries involving relationships between data points, addressing challenges that traditional relational databases face, such as handling multiple levels of relationships. The primary advantage of graph databases is their ability to perform real-time queries in highly

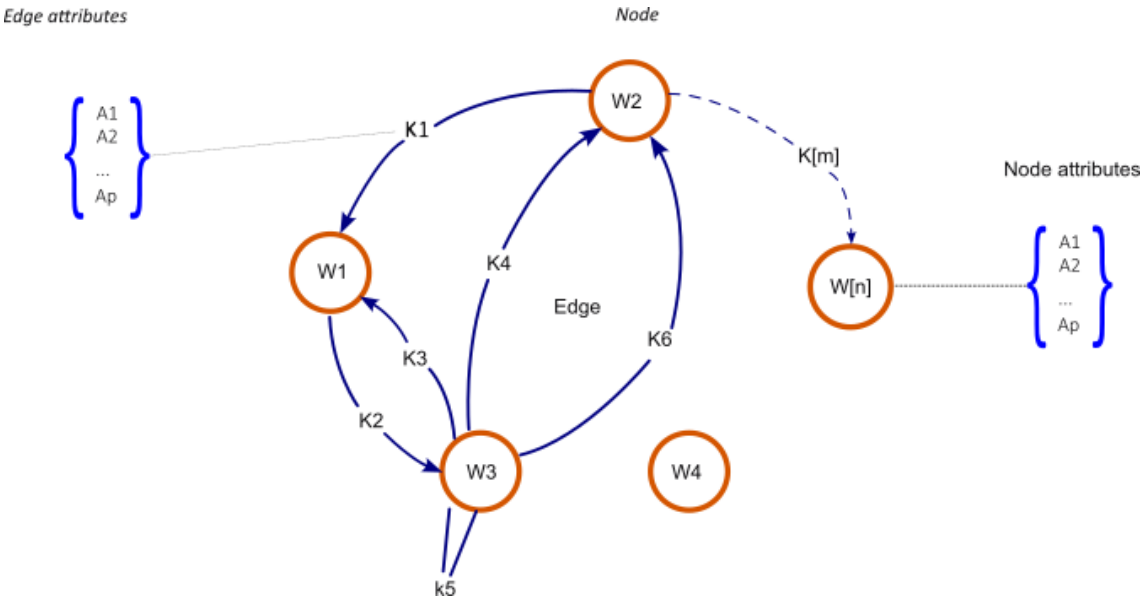

Figure 2.5: Graph Database.

interconnected datasets. This makes them ideal for applications like social networks, recommendation engines, and fraud detection systems [38], [66], the dynamic capabilities of graph databases provide critical insights into relational aspects of data that are increasingly relevant in modern complex environments.

Below is a detailed explanation of some graph-oriented database systems:

1. **Neo4J:** Neo4J is an ACID-compliant transactional graph database and the most popular in its category. It provides advanced features for graph traversal. It is widely used for social networking and recommendation systems [47], [81].

2. **InfiniteGraph:** InfiniteGraph is a distributed enterprise graph database designed for large-scale applications. It supports concurrency, consistency, multithreaded processing, and cloud-based deployments, making it ideal for high-performance use cases [37], [103].

3. **InfoGrid:** InfoGrid is a web-based graph database that facilitates the development of RESTful applications. It provides powerful APIs and tools for creating graph-centric web applications, suitable for dynamic, data-driven environments [19], [86].

4. **HypergraphDB:** HypergraphDB is an open-source graph database based on a directed hypergraph structure. It excels in complex data modeling, knowl-

edge representation, and relational queries. Its features include customizable indexing, non-blocking concurrent operations, and multithreaded transactional processing [16], [90].

5. **AllegroGraph:** AllegroGraph is a closed-source graph database optimized for document-oriented information retrieval. It is widely used in commercial projects, offering robust support for RDF data and semantic web technologies [12], [97].

2.1.1.9 Object-Oriented Databases

Object-oriented databases store data as objects, mirroring the paradigm of object-oriented programming. This design allows developers to model real-world entities more naturally, as the database structure aligns closely with the principles of object-oriented design. These databases are particularly well-suited for applications requiring the representation of complex data, such as multimedia systems, computer-aided design (CAD), and scenarios involving intricate data relationships [54], [104].

Figure 2.6 illustrates the organization of data within an object-oriented database. These databases are tightly integrated with programming languages, enabling developers to seamlessly store and retrieve data as native objects without additional mapping layers. This feature reduces the overhead associated with object-relational mapping (ORM) and accelerates application development [44], [95].

Object-oriented databases are tightly coupled with programming languages, allowing developers to store and retrieve data as native objects. Examples include db4o and ZODB, which integrate seamlessly with programming environments[52], [92].

1. **db4o:** db4o is an embedded, open-source object-oriented database tailored for Java and .NET developers. It enables objects to be stored directly without the need for conversion or mapping, significantly reducing development effort. Its lightweight design and ability to integrate with native queries make it ideal for mobile and embedded applications [95].

2. **ZODB:** ZODB is a Python-based database that provides seamless object per-

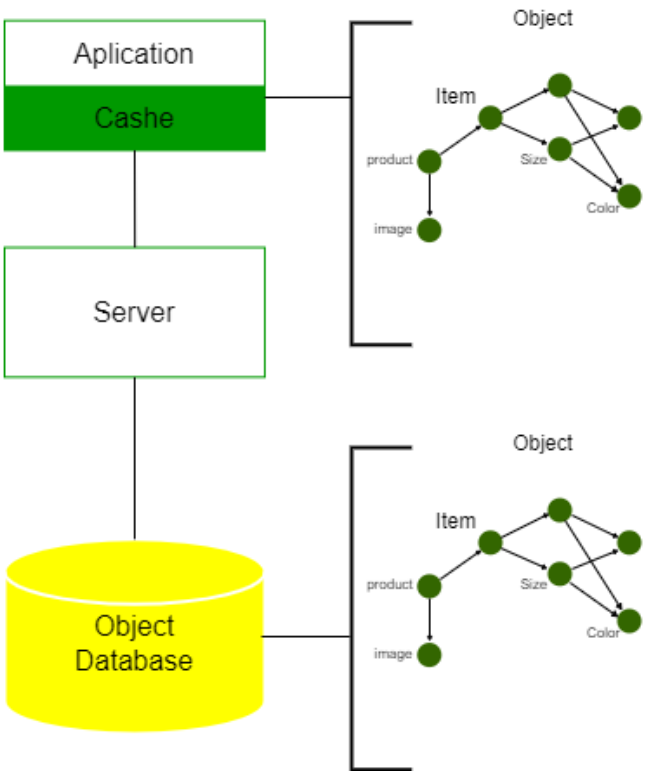

Figure 2.6: Object-Oriented databases.

sistence for Python applications. It allows developers to store, retrieve, and manipulate Python objects natively, eliminating the need for schema definitions. Its transactional support and scalability make it suitable for complex, hierarchical data structures [92].

3. **Versant:** Versant is a commercial object-oriented database designed for industries with demanding performance requirements, such as telecommunications, finance, and aerospace. It supports rich object modeling and provides efficient querying for large-scale object datasets. Its scalability and optimization make it a preferred choice for enterprise applications[106].

2.1.1.10 Multi-Model Databases

Multi-model databases represent an innovative approach, combining the features of various types of NoSQL databases in a single architecture. This feature allows users to manage different types of data, including key-value, document, column-family, and even graph representations, through a unified interface. The flexibility

of databases from various models meets the various needs of modern applications, making them relevant in scenarios where a hybrid approach is required [75]. Figure 2.7 represents how the multi-model database can hold many types of databases.

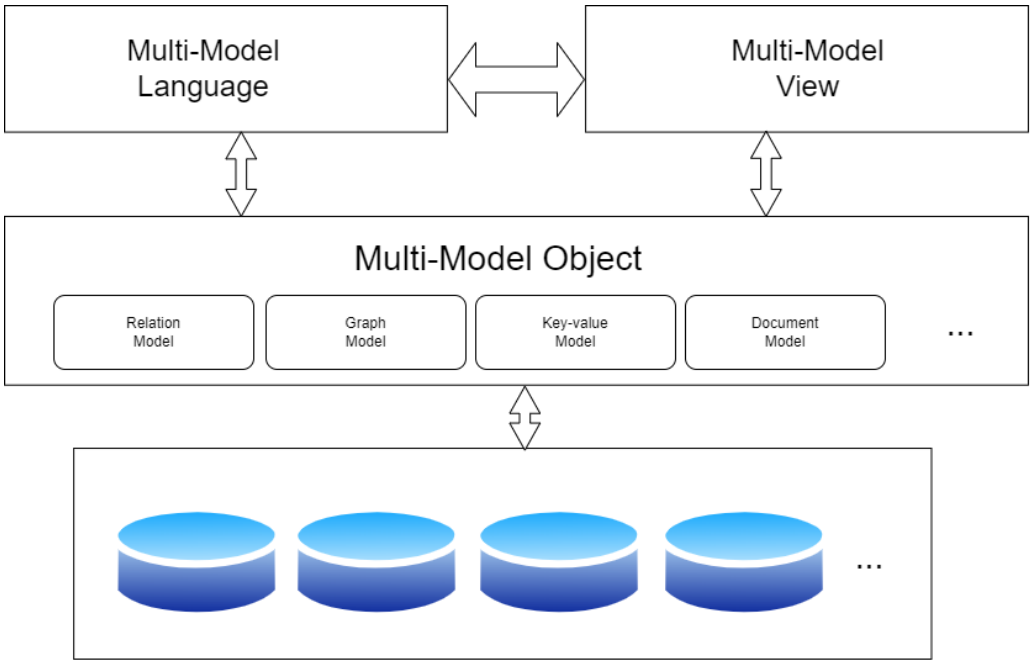

Figure 2.7: Multi-Model Database.

This versatility simplifies development and reduces the overload of interacting with various database systems as it allows comprehensive data handling in environments where various data models coexist [82].

Each of these types of NoSQL databases offers different advantages. Key value stores stand out for their simplicity and speed; Document stores for their flexibility to accommodate the dynamic scheme; Column-family stores for their efficiency in handling large volumes of data relevant to analytical tasks; Graph databases for their effectiveness in interconnected data management; and databases from various models for their ability to integrate multiple formats into a single structure. Understanding the strengths of each technology allows organizations to archive their data in alignment with specific operational demands and business objectives [83].

The following types of NoSQL databases key-value stores, document stores, column-family stores, graph databases, and multi-model databases each have unique characteristics, use cases, and advantages that allow them to meet different requirements in various data management domains. The evolution of these databases re-

flects the growing complexity of data interactions and the pressing need for flexible and scalable data solutions in a constantly changing technological scenario. Careful consideration of the appropriate NoSQL database is essential, as organizations seek to leverage data as a fundamental asset for innovation and competitive advantage [82]. Thus, continuous exploitation of NoSQL technologies and their applications will continue to shape the future of data management practices and systems.

Some of the multi-model-based systems are listed below:

1. **ArangoDB:** ArangoDB is a multi-model database. It supports graph, document, and key-value models. It uses a declarative query language called AQL. It integrates different data models seamlessly. Its versatility makes it ideal for hybrid data storage and complex queries [99].

2. **OrientDB:** OrientDB incorporates document, graph, and value-key models. It provides a high-performance database. It works best with horizontal scalability. It handles ACID transactions. It makes it ideal for flexible use cases such as financial systems and fraud detection [98].

3. **RavenDB:** RavenDB supports documents, graphs and time series. It is a transactional NoSQL database. It has advanced indexing and full-text search functions. It is often used for real-time monitoring and business intelligence applications [100].

2.2.1 Characteristics of NoSQL Models

The key metrics for comparing NoSQL models are listed below. These focus on their capabilities, trade-offs, and unique features. These factors make them suitable for different applications [86].

1. **Persistence:** Persistence keeps data safe. It makes data recoverable. It works even if the system fails. It is important for reliable applications. It is important for long-term storage. Techniques write data to non-volatile storage using distributed file systems to add redundancy at the database level. Many

NoSQL databases handle persistence. They may optimize it differently. Some focus on speed. Some focus less on durability in specific cases [67].

2. **Replication:** Replication keeps multiple copies of data. It stores data across different nodes. It improves fault tolerance. It increases availability. It ensures data stays accessible. It works even during node or network failures [63].

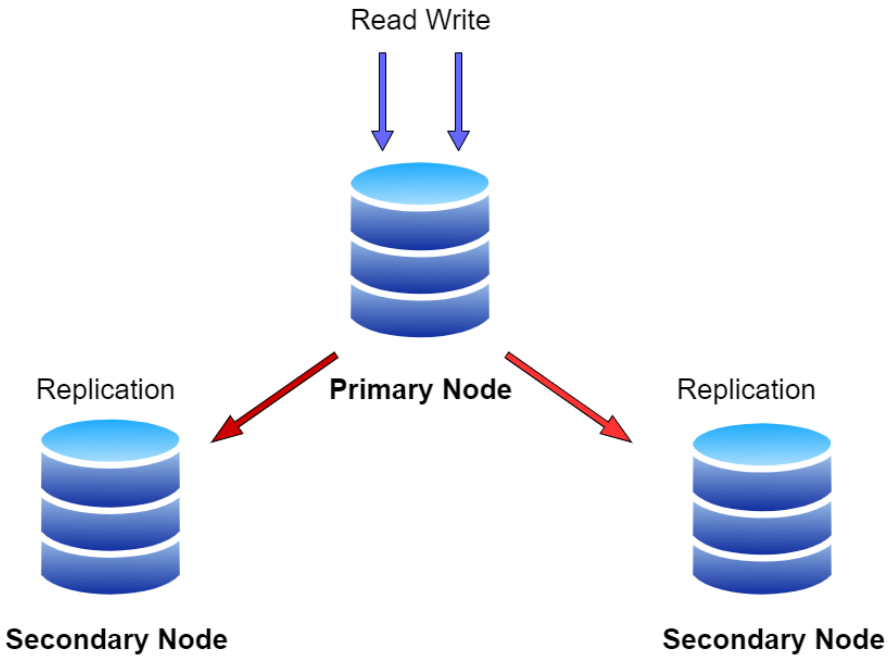

Figure 2.8: Replication.

Figure 2.8 shows a replication setup. Replication strategies differ. Synchronous replication focuses on consistency. Asynchronous replication focuses on availability. Challenges like write conflicts and eventual consistency exist. These challenges need proper management. Proper management ensures data integrity [83].

3. **Sharding:** Sharding splits a database into smaller parts. These parts are called shards. Figure 2.9 shows this. Each shard is stored on a separate node. Sharding allows horizontal scaling. It also enables parallel data processing [49].

   This approach works well for large datasets. It also handles high-speed data streams in distributed systems. Sharding algorithms decide how data is distributed. Examples include hash-based and range-based partitioning. Shard-

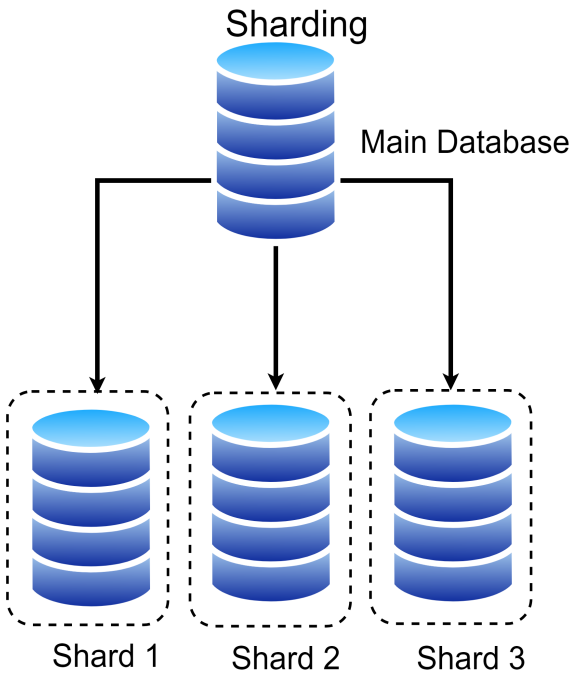

Figure 2.9: sharding.

ing needs careful design. Careful design avoids data hotspots. It also ensures efficient query processing [65].

4. **Consistency:** Consistency ensures all database replicas show the same data state. In NoSQL systems, consistency models vary. Strict consistency updates all replicas immediately. Eventual consistency updates replicas over time. Eventual consistency improves availability. However, it creates challenges for real-time data integrity. Understanding consistency guarantees is important. It helps when choosing a system for applications like financial transactions or inventory management [72].

5. **Query Method:** NoSQL databases offer different query methods. These methods correspond to their data models. Key-value shops use simple GET/PUT operations. Document-oriented databases support advanced queries. They use JSON or XML. Graphical databases support cross-database queries. These queries analyse relationships efficiently. Column-based databases use structured queries. These queries enable large-scale analysis. These methods offer great flexibility. However, developers need to adapt to database-specific query languages and APIs [93].

6. **Scalability:** NoSQL databases are designed to deliver high performance on a large scale. They use distributed architectures. They optimise read/write operations. They use in-memory caching. Performance measures include latency, throughput and query efficiency. These measures can be used to evaluate NoSQL systems. Scalability is ensured by horizontal scaling and data partitioning. Scalability enables the database to handle increasing workloads. It also helps to avoid major drops in performance [91].

7. **Fault Tolerance and Availability:** Fault tolerance keeps NoSQL databases running. It works even during hardware or network failures. Features like replication and automated failover improve availability. These features are essential for mission-critical applications. They help minimize downtime [96].

8. **Schema Flexibility:** NoSQL systems offer schema flexibility. They store semi-structured or unstructured data. This supports fast application development. It adapts to changing data needs. It is useful in content management systems. Data formats can change over time in such systems [96].

2.2.2 NoSQL Systems Based on CAP Theorem

NoSQL systems balance CAP theorem properties. The properties are Consistency, Availability, and Partition Tolerance. Each system prioritizes specific properties. The prioritization depends on its use case. The table below compares several NoSQL systems. It highlights their CAP theorem trade-offs.

NoSQL System	**Consistency (C)**	**Availability (A)**	**Partition Tolerance (P)**	**Features**
Redis [73]	Strong	Strong	Weak	Suitable for caching and session management.
MongoDB [105]	Eventual	Strong	Strong	Flexible for semi-structured data and horizontal scaling.

Cassandra [80]	Eventual	Strong	Strong	High performance for large-scale distributed data.
Neo4j [81]	Strong	Weak	Moderate	Focused on relationships in graph structures.
CouchDB [101]	Eventual	Strong	Strong	Optimized for distributed environments.
HBase [80]	Strong	Moderate	Strong	Designed for large-scale analytics and random access.
ArangoDB [99]	Eventual	Strong	Strong	Supports multi-model operations efficiently.

Table 2.1: NoSQL Systems Based on CAP Theorem

2.2.3 Comparison of NoSQL Database Types

NoSQL databases are divided into many types, which depend on their functionalities and applications. Each type solves specific challenges and fits specific use cases. They offer flexibility and scalability. The table below compares these types. It lists example systems and common use cases.

Type	**Example Systems**	**Use Cases**
Key-Value Store	Redis [73], Amazon DynamoDB [73], Riak [73], Aerospike [73]	Caching, session management, configuration storage.

Document Store	MongoDB [105], CouchDB [101], Terrastore [66], RavenDB [100]	Content management, e-commerce platforms, real-time analytics.
Column-Family Store	Cassandra [80], HBase [80], Hypertable [23], Accumulo [101]	Time-series data, big data analytics, log management.
Graph Database	Neo4j [81], HypergraphDB [90], AllegroGraph [97], InfiniteGraph [103]	Social networks, recommendation systems, fraud detection.
Multi-Model Database	ArangoDB [99], OrientDB [98], RavenDB [100]	Applications requiring multiple data representations.
Object-Oriented DB	db4o [95], ZODB [92], Versant [106]	Multimedia, CAD, tightly coupled object-oriented applications.

Table 2.2: Comparison of NoSQL Database Types

2.3 Conclusion

This chapter introduces NoSQL databases. It explains their role in overcoming the limitations of traditional RDBMSs. It analyzes key concepts such as the CAP theorem and BASE properties. These concepts show the trade-offs for scalability, availability, and consistency in distributed systems. Also, it classifies and compares NoSQL database types with different features and describes them. It shows the importance of choosing the right type of NoSQL database. The choices depend upon the application requirements of performance, data flexibility, and scalability goals. Insights into the CAP theorem ensure better understanding, hence helping in bringing forth the system design that will meet the desired outcomes such as availability, consistency, or partition tolerance.

The next chapter introduces the main mechanism which is called SDDS. SDDS is the core concept in our system, which is mainly used in developing our NoSQL system.

Chapter 3

Scalable Distributed Data Structures

3.1 Introduction

SDDS is a class of distributed systems. They deal with data in distributed environments. Traditional systems have drawbacks because they rely on centralization. New systems use scalability as a base. They support fault tolerance and very fast data access. SDDS like LH* and RP* are techniques for enhanced data distribution. They allow dynamic extension and offer highly scalability as well.

SDDS was first developed in the early 1990s. There was a need for scalability and fault tolerance in handling data as a distributed system. These systems have evolved over the years. They now serve many purposes, from cloud computing to realtime analytics. This chapter introduces the necessary concepts and motives related to SDDS. It starts with their definition and main features. Then, it covers different SDDS classes, the LH* family, the RP* family, and alternative techniques in Compact Trie Hashing, like CTH*. Each section describes the characteristics of these approaches. It also provides an overview of their usage and effectiveness in current computing environments.

3.2 Core Principles of SDDS

Scalable Distributed Data Structures (SDDS) are based on several essential ideas. The lack of a centralized directory for data access is the first fundamental. System access performance is hampered by a bottleneck created by a centralized directory. However, to boost performance in specific situations, some SDDS versions include lightweight centralized metadata management. SDDS enhances scalability and fault tolerance in distributed systems by doing away with the requirement for a centralized directory [4].

The second rule focuses on the incremental and transparent extension of the data file during insertions. An SDDS usually starts with a single storage site. It gradually expands to multiple sites as data insertions grow. These sites are called SDDS servers. The number of servers is theoretically unlimited. When a server becomes overloaded, it performs a split operation. It transfers about half of its data to a new server. This dynamic redistribution of data supports scalability. It also ensures balanced workloads across the servers [10].

A server in the redirection process sends an Image Adjustment Message (IAM) to the client. This helps the client update its local image of the data structure. IAMs are very important. They maintain consistency in distributed environments. They ensure clients have an accurate view of the data distribution.

SDDS servers are accessed by independent client sites. These sites are called SDDS clients. They manage their local image of the data structure. Changes in the structure, like server splits, are not immediately communicated to clients. This can make their local images outdated. Outdated images can cause addressing errors. A client might send a query or update request to the wrong server. However, SDDS servers can detect such errors. When an error is detected, the server redirects the request to another server. This server may hold the correct data. Sometimes, additional redirections are needed. SDDS are designed to minimize these occurrences. To prevent repeated errors, the server sends an IAM to the client. This allows the client to update its local image of the data structure [6].

Finally, SDDS is designed to support parallel processing efficiently. A parallel

query is an operation sent by a client to multiple SDDS servers. These requests can be targeted (point-to-point or unicast). They can also be broadcast to all servers in the network (multicast/broadcast). Each server processes the query independently. This parallel processing capability improves the performance of SDDS. It also enhances their scalability. These features make SDDS ideal for modern distributed systems [6].

3.3 Importance of Scalability

Scalability is a key element of modern data management systems. It is particularly important in distributed environments. Data is growing rapidly in areas such as big data analytics, cloud computing and real-time processing. Effective data management and recovery is crucial. SDDS offers a solution by allowing systems to scale dynamically as data volumes increase. This ensures optimum use of resources. They also maintain performance without compromise [10].

Traditional centralised systems suffer from bottlenecks and latency. This arises when data sets become too large for single server or database systems. SDDS addresses these problems[56]. It shares data across multiple nodes. It provides parallel processing and efficient load balancing. This distributed design minimizes reliability. even if a node fails, the operations continue smoothly. Efficiency is also enhanced and the network communication during queries and updating is minimised. This objective is achieved using various methods such as linear hashing (LH*) and range partitioning (RP*). These methods optimise data access and reduce overall costs [6].

Scalability is very important in industries like e-commerce. E-commerce processes large volumes of transactions in real-time. It is also crucial in scientific research. Scientific research generates and analyzes massive datasets. These features make SDDS essential. They are needed for applications that require scalable, high-speed, and reliable data management. Such applications are common in modern computing environments.

3.4 SDDS Families

To address the requirements of distributed systems, SDDS was developed. They make fault-tolerant, scalable, and effective data management possible. There are three primary families of SDDS. Their guiding concepts and areas of use form the basis of these families. The LH* (Hash-based) is the first family. They use a linear hash to organize data collections. The preorder (range-based) is the second family. It manages the distribution of the data keys by their intervals, and the third is the rest of the other techniques.

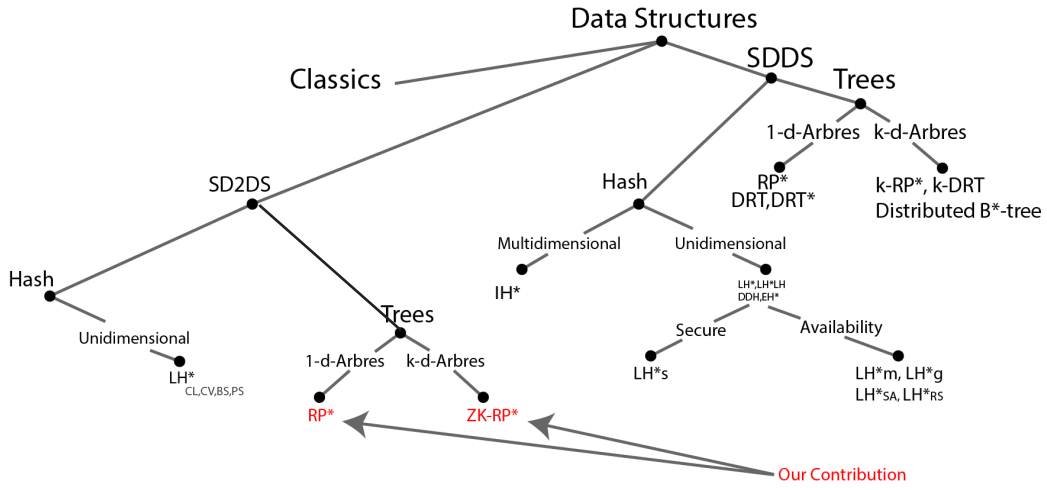

Figure 3.1: SDDS Families

3.4.1 Unidimensional SDDS

SDDS represent a major advance in distributed data management. They enable the efficient processing of large-scale data sets on interconnected nodes. One-dimensional SDDSs are one of the main classifications. They specialize in managing datasets with a single attribute or key. These structures use innovative techniques for dynamic data distribution and access. They guarantee high scalability, fault tolerance and adaptability. One-dimensional SDDS are divided into two main types. The first type is hash-based SDDS. The second type is range-based SDDS. Each

type offers unique advantages. These advantages address specific data management needs.

3.4.1.1 Hash-Based SDDS

Hash-based SDDS uses hashing techniques. They dynamically distribute data across nodes. They adapt easily to changing workloads. The foundational model in this category is *LH** (Linear Hashing*) [10]. Litwin et al. introduced *LH**. It extends traditional Linear Hashing principles. It enables incremental data distribution across multiple nodes. This approach removes the need for a centralized directory. Clients maintain partial and updated images of the system. As a result, *LH** ensures scalability. It also ensures resilience. This makes it a strong solution for distributed data management.

The dynamic hash function in LH* adjusts as the dataset grows. It doubles the hash range. It redistributes records dynamically. This keeps the system scalable and efficient. It works well even as the dataset expands.

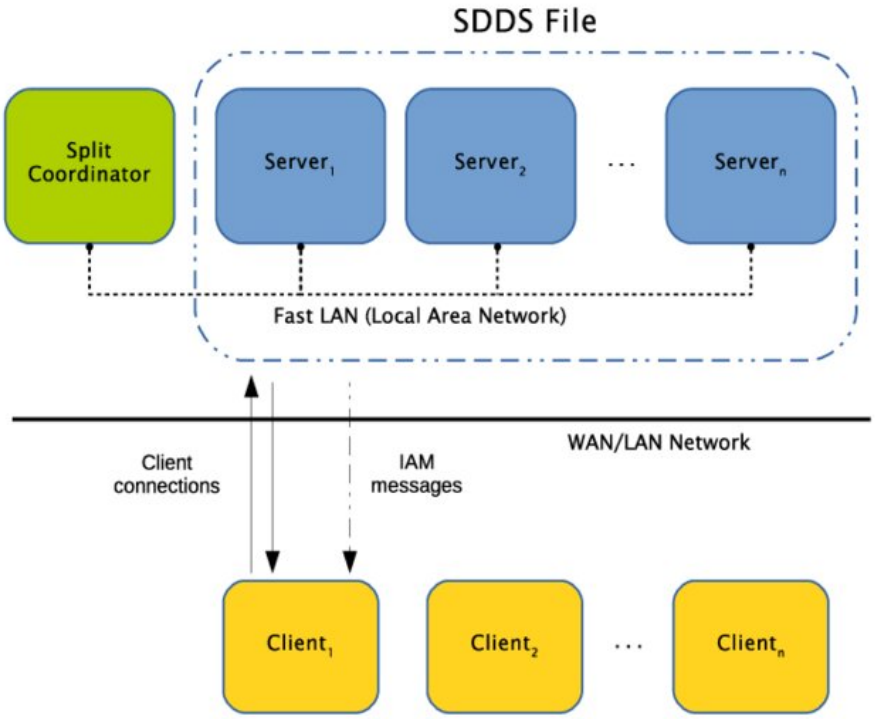

Figure 3.2: LH* SDDS.

1. **LH***

   Linear Hashing (LH*) is a foundational hash-based Scalable SDDS. Litwin et al [10]. introduced it. It is designed to manage dynamic datasets in distributed systems. (See figure 3.2).

   LH* extends traditional linear hashing. It allows data to be distributed incrementally across several servers. This ensures scalability. It prevents global reorganization. The heart of LH* is its dynamic hash function. The hash function adapts as the data set grows. The hash function is described as follows:

$$h_i(k) = k \bmod (2^i \cdot N) \tag{3.1}$$

   Here, k is the key. i is the current hash level. N is the initial number of buckets. As the data set grows, i increases. This doubles the hash range. It dynamically redistributes records. Insert operations use the hash function $h_i(k)$. This function associates a key k with its corresponding bucket. When a bucket exceeds its load threshold, a splitting operation takes place. This redistributes around half the bucket's data into a new bucket. Search queries also use the hash function (3.1). It identifies the target directory.

   If a client's directory image is obsolete due to splits, redirection mechanisms are useful. They ensure that the request reaches the right directory. The system makes dynamic adjustments via IAM. These messages enable clients to update their local directory images. These features make the LH* highly efficient. They guarantee balanced load distribution, minimal latency and scalability of distributed systems [4], [10].

   The variants of the LH*:

   The LH* family has several variants. These variants are designed for different use cases.

   (a) **LH*s (Secure Linear Hashing):** Linear hash (LH*) has a version called LH*s. It includes security functions. In distributed systems, these functions ensure data integrity and confidentiality. LH*s handles dynamic data distribution, just like LH*. In addition, it uses encryption techniques and secure hash

functions to guarantee data security throughout processing. These enhancements make LH* ideal for sensitive sites. Finance and healthcare are excellent examples. Data security is crucial in these fields. Hash and split operations incorporate security guarantees. They preserve the efficiency and scalability of the original LH*. They also offer solid protection against unwanted access [14].

(b) **LH*LH (Two-Level Linear Hashing):** A hierarchical structure is implemented. This structure provides greater flexibility and scalability. This, in turn can handle vast amounts of data. LH*LH uses a two-level linear hashing technique. Data is shared across servers globally. On individual servers, data is kept on a local basis. To handle this, local linear hashing is implemented. This forms a two-tier system in which complex distribution of large sets of data can be maintained effectively. This breaks down operations. The distribution of data between the nodes is balanced globally. This architecture optimizes access within each server [7], [17].

(c) **LH*m (Linear Hashing with Mirroring):** LH*m is a type of LH*. It uses data mirroring. This improves fault tolerance. Distributed systems benefit from this. LH* supports dynamic data distribution. LH*m adds data replication. It copies data to multiple servers. This keeps data accessible. Data remains available even if a server fails. During insertion, the system applies a hash function. It identifies the primary bucket. Mirrored copies are stored in backup buckets. For lookups, the system uses the hash function. It fetches data from the primary or mirrored bucket. This depends on availability. Redundancy increases system reliability. It maintains the scalability and keeps the efficiency of LH*. LH*m suits applications requiring high availability. It also suits applications needing fault tolerance [8].

(d) **LH*sa (Distributed Linear Hashing with Scalable Availability):** LH*sa is a type of LH*. It is designed for distributed systems. It optimizes operations for time-critical needs. LH*sa uses synchronization mechanisms. These ensure consistency. They also reduce latency. This happens during concurrent data access and updates. LH*sa introduces synchronized splitting. It also uses coordinated query handling. These minimize conflicts. They en-

sure accurate data for clients. Clients access the system at the same time. The synchronization features are part of the hashing process. They keep the scalability of LH*. They also keep the efficiency of LH*. [15].

(e) **LH*rs (Linear Hashing with Reed-Solomon Codes):** LH*rs is a sort of LH*. It is fault-tolerant. It uses Reed-Solomon coding. This improves data reliability in distributed systems. LH*rs ensures data recovery. This happens if a server fails. It encodes data using Reed-Solomon codes. During insertion, data is distributed across multiple servers. Redundant parity information is also stored. If a server fails, the lost data can be recovered. This uses the parity and data from other servers. This method improves fault tolerance. It keeps the scalability of the hashing mechanism. It also keeps the efficiency of the hashing mechanism. LH*rs works well for applications needing high data availability. It also suits applications needing robustness [18], [20], [108].

2. **DDH**

   Distributed Dynamic Hashing (DDH) is a type of hash-based SDDS. It improves adaptability. It also improves load balancing. It works in dynamic distributed environments. DDH uses dynamic bucket splitting. It also uses dynamic bucket merging. These manage data distribution well. They work across a distributed system. This helps the system adapt to changing workloads. It keeps performance stable [3].

3. **Extendible Hashing (EH*)**

   Extendible Hashing (EH*) is a type of hash-based SDDS. It provides efficient scalable data access. It works in distributed environments. EH* uses the principles of traditional Extendible Hashing. It adds distributed storage. It also adds dynamic bucket management. This helps the system handle data growth well. It adapts to different access patterns. It keeps workloads balanced across servers [13].

3.4.1.2 Order-Preserving SDDS

1. **Range Partitioning (RP*) [6]:** Range Partitioning (RP*) is the main SDDS pre-ordered method. It manages ordered datasets. It organizes records

using pre-ordered keys. It uses an index like a B+ tree, as shown in figure 3.3. The RP* splitting mechanism redistributes records dynamically. This maintains balanced storage. It also ensures scalability. When a bucket overflows, it splits into two buckets. The original bucket holds the range $]-\infty, c_m]$. The new bucket manages the range $]c_m, +\infty[$. This process happens dynamically. It continues as data grows. It keeps the system scalable. It also keeps the system efficient.

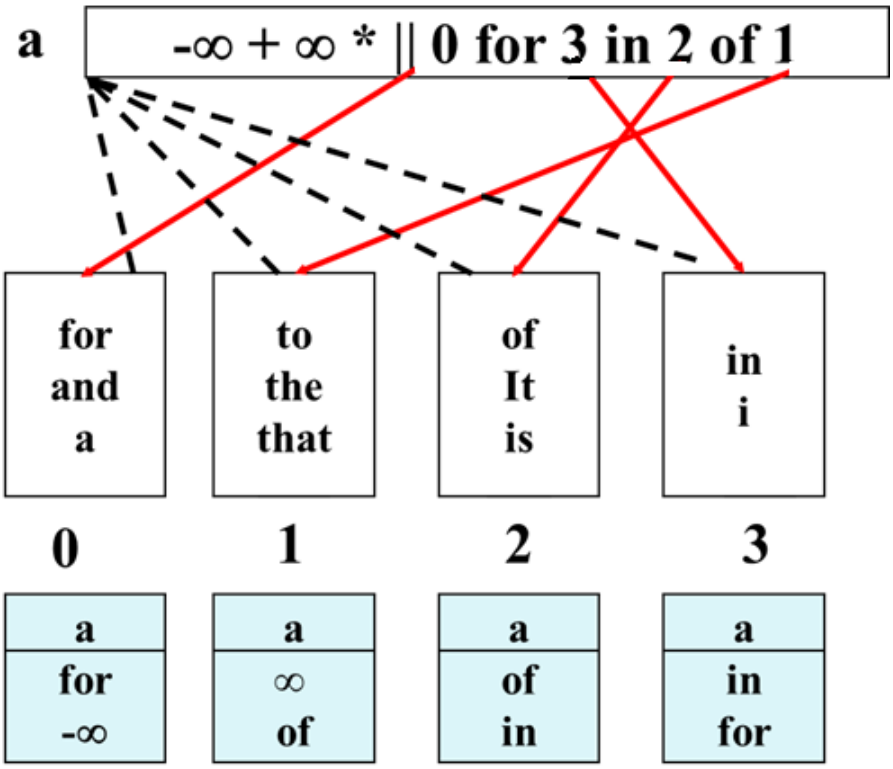

Figure 3.3: Index RP* with one level.

Each record is placed in the right bucket. This enables efficient processing of interval queries. It also enables efficient browsing of ordered data. RP* clients can request individual records. They can also request range of records. These ranges are organized into intervals. The RP* system starts with a bucket containing all the records in the interval $]-\infty, +\infty[$. If a bucket overflows, it splits into two buckets. The original bucket keeps the interval $]-\infty, c_m]$. The new bucket handles the interval $]c_m, +\infty[$. Here c_m is the key to the midpoint of the division. This process runs dynamically as the data grows. It redistributes records to maintain storage balance. It also guarantees

scalability. Each bucket has an interval $]\lambda, \Lambda]$. Here, λ is the minimum key. Λ is the maximum key.

At the client level, requests use the directory. The directory maps key ranges to buckets. It updates dynamically. This reflects bucket splits. It ensures accurate mapping. Sometimes, addressing errors occurs. These happen due to outdated directories. Servers redirect queries to the correct bucket. They send an IAM to clients. This updates the client's local view of the directory. Queries are sent as unicast messages. Server redirections often use multicast. This reduces communication overhead. The dynamic adjustment mechanism ensures efficient range handling. It also minimizes latency. RP* uses a structured approach. This works well for databases that need ordered data access. They also need precise range queries.

The variants of RP*:

The RP* family has several key variants. Each variant is tailored to different use cases. Each variant improves the core functionality of range partitioning [6].

- **RP*N:** This variant uses only multicast messages for client-server communication. because the client has no index or partial image of the system.
- **RP*C:** RP*C extends RP*N. It adds a partial client-side index. This index is created using IAMs. The partial index lets clients keep local information. This information is about accessed buckets. It reduces the need for multicast communication. It also improves query efficiency. This works well for frequently accessed data ranges.
- **RP*S:** RP*S introduces a distributed server-side index. Each bucket is indexed by its range $]\lambda, \Lambda]$. This variant splits overloaded buckets dynamically. It adjusts the index as needed. This ensures balanced load distribution. It also optimizes query handling.

2. **Dynamic Range Tree (DRT*):** DRT* is an advanced structure in the Range-Based SDDS family. It handles range queries. It works in distributed

systems. DRT* extends traditional range trees. It adds dynamic adjustments. These adjustments partition ranges as the dataset changes. The structure organizes data into hierarchical tree nodes. Each node represents a range of keys. It splits nodes dynamically when their load exceeds a threshold.

The splitting operation redistributes records. It creates new nodes. It updates the directory to show the new ranges. Unlike static range trees, DRT* adapts to changing data distributions. It ensures balanced load across nodes. It also ensures efficient query performance. Lookup operations traverse the tree hierarchy. They locate the correct range node for a query. This enables precise handling of single-key requests. It also handles range-based requests. [69].

3.4.2 Multi-dimensional SDDS

There are many SDDS support multi-dimensional or multi-attribute records. Some methods are hash-based. Others use pre-ordered indexing. Pre-ordered indexing often uses k-d trees.

1. **Multi-Attribute Range Partitioning (k-RP*):** k-RP* is an extension of the RP* family. It is designed for managing multidimensional datasets. It works in distributed systems. Unlike unidimensional RP*, which partitions data along one key, k-RP* supports multi-attribute data partitioning. Each record gets a composite key which is derived from its attributes. The system partitions data into multidimensional ranges. These ranges are based on the composite keys. The directory in k-RP* maps these ranges to servers (see figure 3.4). This ensures efficient data access and efficient query handling [9].

2. **Multi-Attribute Dynamic Range Tree (k-DRT*):** k-DRT* is an advanced extension of DRT*. It handles multidimensional datasets. It works in distributed environments. Unlike the unidimensional DRT*, k-DRT* organizes data into hierarchical tree structures. These structures span multiple attributes. This enables efficient resolution of complex multidimensional range queries. Each node in the k-DRT* tree corresponds to a specific range. This range covers multiple dimensions. Internal nodes partition data hierarchi-

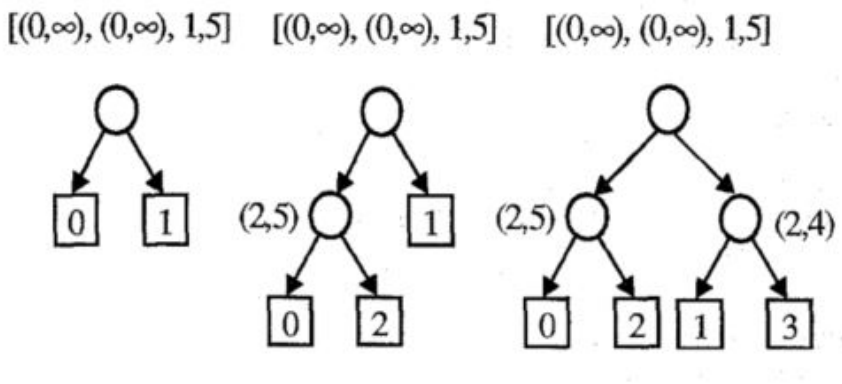

Figure 3.4: Index K-RP*'s evolution [9].

cally. Leaf nodes hold the actual records. The system adjusts dynamically to changing data distributions. It splits overloaded nodes into subranges. It redistributes records across new nodes. This ensures balanced workloads [9].

3. **Distributed B-Tree:**The distributed B-tree is a multidimensional data structure. It is designed for distributed systems. It extends the traditional B-tree. It operates efficiently across multiple nodes. The structure organizes data hierarchically. Each level of the tree represents a different dimension. This enables complex multidimensional queries to be processed. Each node corresponds to a data partition. Partitions are stored on separate servers. Internal nodes act as routing points. They direct queries to the right partitions. The system dynamically adapts to changes in data distribution. It redistributes data between nodes in the event of overload. This maintains a balanced tree structure.

   Insertions and deletions are handled locally. They occur in the corresponding leaf nodes. Global updates are propagated throughout the tree. This ensures consistency. Hierarchical relationships are maintained. Query operations are highly efficient. Hierarchical structure enables rapid navigation [27].

4. **Interpolation Hashing (IH*):** IH* is a multidimensional variant of hash-based SDDS. It handles datasets with multiple attributes. Unlike single-attribute hashing systems, IH* uses a multidimensional interpolation function. This function maps data records to buckets. It works in a distributed environment. This approach manages spatial and multidimensional datasets

efficiently. Each bucket in IH* corresponds to a specific multidimensional range. The system adjusts dynamically as data distributions change.

When a bucket becomes overloaded, IH* performs a split operation. It redistributes records across new buckets. This uses updated interpolation parameters. IH* also includes dynamic scaling mechanisms. It supports fault tolerance, which ensures robust performance. It works well with large and evolving datasets [107].

3.4.3 Others SDDS:

This section focuses on alternative SDDS methods. These methods are commonly known as trie-based techniques.

- **Trie Hashing (TH*)** TH* is a Scalable Distributed Data Structure (SDDS). It extends the principles of traditional trie structures. It works in distributed environments. It enables efficient management of ordered datasets. These datasets are spread across multiple servers. In TH*, data records are indexed using a hierarchical trie structure. Each level of the trie represents a segment of the key called a "digit" (see figure 3.5).

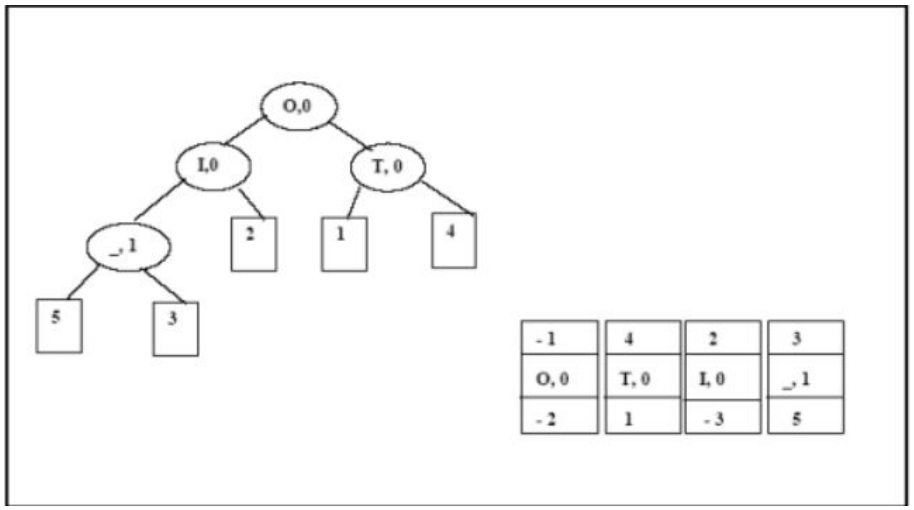

Figure 3.5: Trie Hashing.

This organization ensures efficient retrieval. It also ensures ordered traversal. This makes TH* effective for range queries. It also works well for sequential

data access. The trie is distributed across nodes. Each node handles a subset of the trie's branches. It also handles the leaf nodes. Insertions in TH* involve decomposing a key into segments. They navigate the trie structure level by level. This continues until the correct leaf node is found. The record is stored there.

If a node becomes overloaded, it triggers a split operation. This redistributes part of its data to a new node. It updates the parent nodes to reflect the change. This dynamic splitting ensures balanced load distribution. It prevents performance bottlenecks. For lookups, the trie structure allows rapid navigation. It follows the key's segments through the hierarchical levels. This minimizes search latency. Range queries are efficient. The hierarchical organization enables traversal of contiguous ranges [1].

- **Two-Level Trie Hashing (TH*TH)** TH*TH is an advanced extension of Trie Hashing (TH*). It introduces a two-level structure. This improves scalability. It also improves efficiency in distributed systems. TH*TH organizes the trie into two levels. The global level distributes trie branches across servers. The local level manages subtrees on each server. This two-level structure handles large datasets well. It separates operations across servers. It keeps the benefits of trie-based indexing.

  During insertion, a key goes to the correct global branch. This depends on its prefix. It is then processed locally within the subtree. If a local node becomes overloaded, it starts a split operation. This moves data to new nodes. Updates go to the global level for consistency. Lookups work similarly. The key moves through the global trie first. It then goes into the correct local trie. This two-level design balances load distribution. It reduces communication between nodes. It improves range and point queries [2].

- **Compact Trie Hashing (CTH*):** CTH* is an optimized variant of Trie Hashing (TH*). It is designed to improve memory efficiency. It also improves performance in distributed environments. It reduces redundancy in the trie structure. Traditional trie structures store each key segment independently. CTH* compresses common prefixes of keys. This greatly reduces memory

usage. It keeps the hierarchical organization. This organization is needed for efficient range queries. It also supports ordered data traversal. The compact representation allows CTH* to store larger datasets. It avoids overloading system resources.

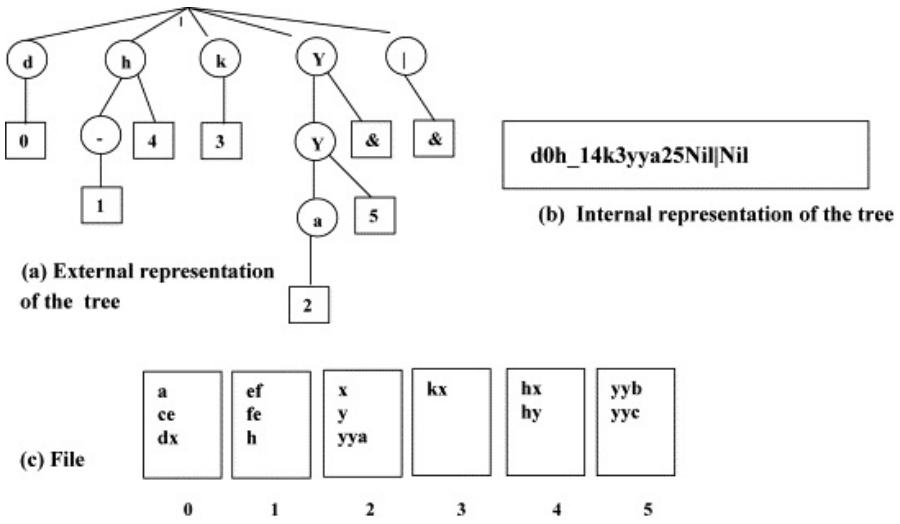

Figure 3.6: CTH*.

As shown in Figure 3.6, insertions in CTH* follow the hierarchical decomposition of keys. This is similar to TH*. They use compressed trie nodes. This avoids redundant storage of overlapping segments. When a node becomes overloaded, it triggers a split operation. This redistributes part of its data to a new node. It adjusts the compressed prefixes. This maintains the trie's structure.

Lookup operations hence benefit from the reduced size of the trie, traversal through compact nodes ensures faster access to the desired record. Similarly, range queries in CTH* are optimized because the compressed structure can efficiently traverse contiguous ranges without the need for exhaustive searches [21], [22].

Table 3.1: Summary of SDDS Methods

Method	Technique	Features
LH*	Dynamic hashing for distributed systems	Supports incremental splitting to balance load dynamically and uses efficient hash-based lookup to locate data quickly.
LH*s	Secure hashing with encryption	Integrates encryption mechanisms into the hashing process to protect data during storage and transmission without sacrificing performance.
LH*LH	Two-level hashing for scalability	Employs global hashing for distributing data across servers and local hashing for managing data within individual servers, enhancing flexibility.
LH*m	Fault tolerance through mirroring	Ensures redundancy by mirroring buckets. Queries can access either the primary or backup bucket, providing high availability.
LH*sa	Synchronization for concurrent access	Implements synchronized splitting and coordinated query handling to ensure consistent updates and minimize conflicts.
LH*rs	Fault tolerance with Reed-Solomon codes	Uses Reed-Solomon coding to store redundant parity information, allowing data recovery during server failures.
RP*	Range partitioning for ordered data	Divides datasets into dynamic ranges, enabling efficient range-based queries and maintaining ordered data traversal.
RP*N	Multicast communication for range partitioning	Uses multicast communication to interact with servers, minimizing overhead and improving performance for range-based queries.
RP*C	Partial client-side indexing	Builds a partial client-side index using Image Adjustment Messages (IAMs) to reduce inter-server communication during frequent queries.
RP*S	Server-side distributed index	Creates a distributed index on servers for dynamic range splitting, balancing loads during continuous data growth.
DRT*	Hierarchical tree for range-based queries	Organizes data hierarchically using tree structures, dynamically splitting overloaded nodes to maintain balance.
k-RP*	Multi-attribute range partitioning	Utilizes a global index for multidimensional data and dynamically splits ranges to adapt to complex queries and growing datasets.
k-DRT*	Multidimensional dynamic range tree	Handles multidimensional queries by dynamically redistributing records through tree-based partitioning and splitting mechanisms.
Distributed B-Tree	Hierarchical B-tree for distributed systems	Balances partitions dynamically and supports efficient range and point queries with low inter-node communication overhead.
TH*	Trie-based hashing for ordered datasets	Organizes data hierarchically into trie nodes, enabling fast lookup and efficient range queries through compact trie traversal.
TH*TH	Two-level trie for hierarchical scalability	Combines global and local trie structures, supporting large-scale hierarchical indexing and dynamic splitting for scalability.
CTH*	Compact trie hashing for efficient memory usage	Compresses common prefixes to reduce memory consumption, ensuring balanced workloads through dynamic splitting.

3.5 SDDS Applications:

Some applications use SDDS in their core functionality. This section lists a few examples.

3.5.1 MR2P (MapReduce with RP* Integration):

MR2P [64] is a new approach. It optimizes distributed data processing in MapReduce environments. It uses SDDS-RP* (Scalable Distributed Data Structures based on Range Partitioning). Traditional MapReduce frameworks face challenges in the shuffle phase. This phase involves large data transfers between mappers and reducers. It causes high bandwidth usage. It also causes network congestion. MR2P solves these issues. It replaces the default partitioning function. It uses an advanced range-partitioning mechanism. This approach organizes intermediate data into a distributed RP* file. Each mapper acts as an RP* client. Each reducer acts as an RP* server. The shuffle phase becomes a structured operation. It uses RP*'s efficient range-based data distribution. This reduces inter-node communication. It improves scalability. Figure 3.7 shows this.

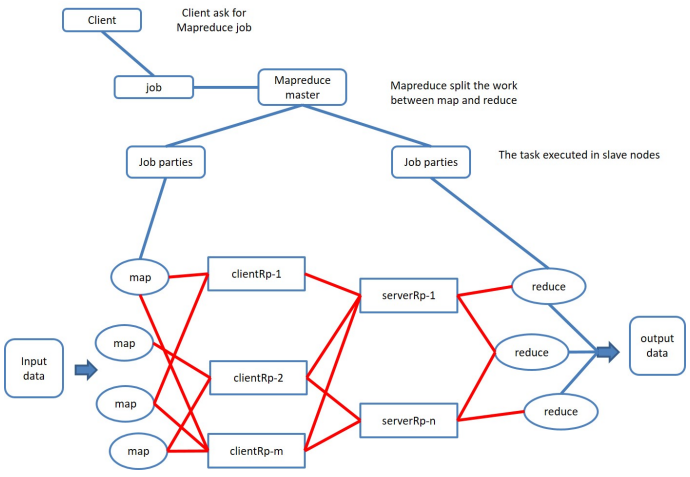

Figure 3.7: MR2P Architecture.

MR2P's architecture adapts dynamically to workload changes. It offers optimized data locality. This happens during the map phase. It also happens during the reduce phase.

3.5.2 SD-SQL Server:

SD-SQL Server is a scalable distributed database system (SD-DBS). It is built on the principles of scalable distributed data structures (SDDS). It provides a new architecture for relational databases. Figure 3.8 shows this. It enables dynamic partitioning of scalable tables. It also enables distributed partitioning of scalable tables[25].

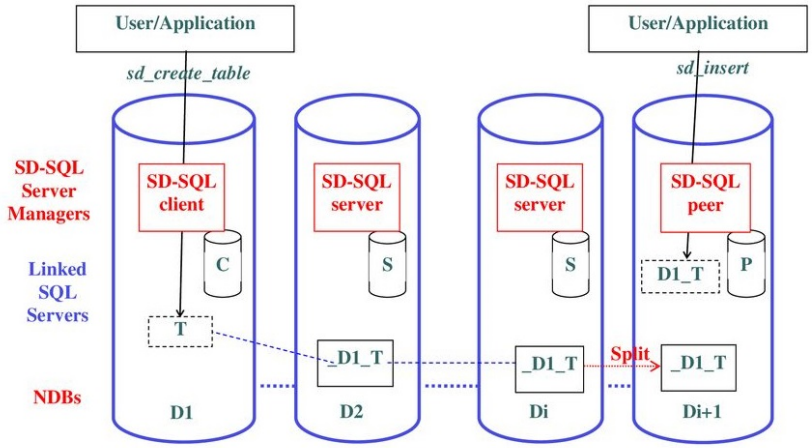

Figure 3.8: SD-SQL Server Architecture.

Traditional database management systems use static partitioning. They need periodic reorganization. SD-SQL Server is different. It allows dynamique table expansion. This happens through a process called splitting. Each table is divided into segments. These segments are distributed across storage nodes. They split dynamically when an insertion causes overflow. These splits are transparent to users. They are also transparent to applications.

Users and applications interact with partitioned views. These views are called images. They adjust dynamically to the changing table structure. This hides the partitioning from end-users. SD-SQL Server uses SQL Server's features. It integrates advanced commands which manage scalable nodes, databases, tables, and images. This ensures ease of use. It also ensures performance optimization. The system's design minimizes overhead. It uses efficient image adjustments. It also manages concurrent processing [24].

3.6 Advanced SDDS: SD2DS

Scalable Distributed Two-layer Data Structures (SD2DS) are an advanced evolution of Scalable Distributed Data Structures (SDDS). They are designed to improve scalability. They also improve efficiency and performance in distributed data environments. SD2DS organizes data into two layers. The first layer is the header layer. It contains metadata. The second layer is the body layer. It contains actual data. This separation optimizes the management of large datasets. It ensures metadata and data are handled independently. It reduces the overhead of data transfers. This happens during operations like bucket splits [60].

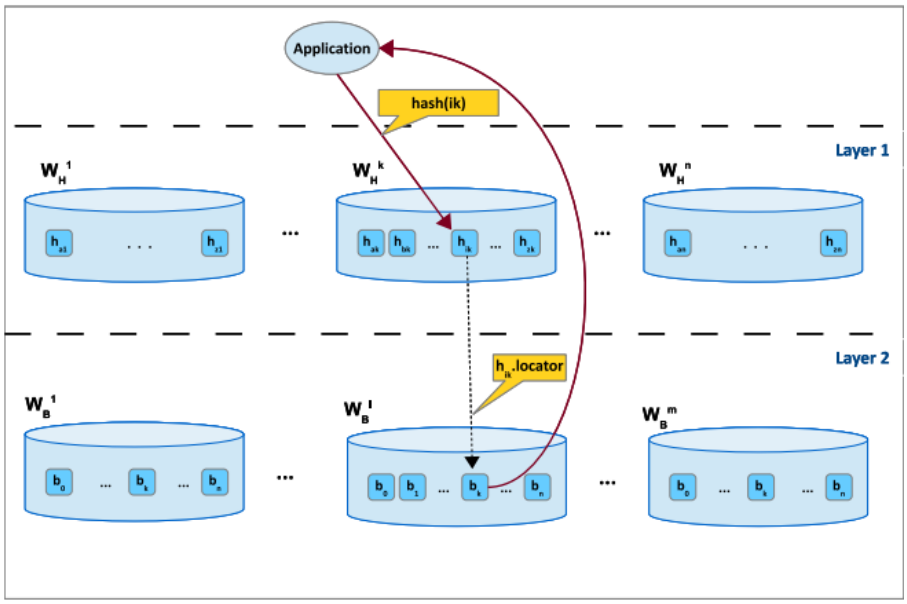

Figure 3.9: SD2DS Two-Layer Architecture

3.6.1 Architecture of SD2DS

The Scalable Distributed Two-layer Data Structure (SD2DS) manages large-scale distributed data efficiently. It separates metadata and actual data into two layers. The first layer is the **Header Layer**. The second layer is the **Body Layer**. This separation optimizes data management. It reduces overhead. It also improves scalability in distributed environments.

The **Header Layer** manages metadata. Metadata includes keys and locators. Each key identifies a data record uniquely. The locator points to the physical loca-

tion of the data body. The data body is in the storage layer. The header layer uses distributed indexing methods. An example is Linear Hashing (LH*). This ensures efficient key management. It allows dynamic scaling. It also balances metadata distribution across the system. This works even as the dataset grows [10].

The **Body Layer** stores the actual data. This data is called data bodies. This layer handles large data records efficiently. The header layer may change frequently. For example, changes happen during bucket splits. The body layer is different. It keeps data stationary. This design minimizes data movement. It reduces overhead. It also improves performance. Data bodies are accessed using locators which are provided by the header layer. This ensures efficient retrieval with an efficient storage operations [77].

3.6.2 The Operations in SD2DS

SD2DS supports several fundamental operations. These operations enable efficient data management. They work in distributed environments. The operations include **PUT**, **GET**, and **Bucket Splitting**. Each operation plays a critical role. They maintain scalability and ensure performance with reliability.

The **PUT** operation inserts the data by separating the header from the body, then the body goes to the storage layer, it provides the locator which goes into the header layer. When a new data record is added, its key is hashed. This determines the correct bucket in the header layer. If the target bucket is full, a split operation is triggered. This redistributes metadata across additional buckets. This ensures the system scales dynamically. It also maintains balanced load distribution [88].

The **GET** operation retrieves the data from the body layer with the help of the locator in the header layer. When receiving a query for a certain key, the system hashes the key to find the appropriate bucket in the header layer. It retrieves the locator from there, which is then used to access the data body in the storage layer [11].

Bucket Splitting This is a very important operation on which header layer scalability depends. When a bucket becomes full with new inserts of key-locator

pairs, the bucket will split dynamically into more buckets. Bucket splitting is one of the key mechanisms toward ensuring balanced load distribution and making it handle increasing datasets with efficiency [5].

3.6.3 SD2DS algorithms

To explain SD2DS functionality, pseudocode for the **PUT** and **GET** operations is provided below. These algorithms show the step-by-step processes. They explain how data is inserted. They also explain how data is retrieved. This happens within the SD2DS architecture.

Algorithm 1 PUT Operation in the Header Layer

Require: Key k, Locator loc

$i \leftarrow \text{Hash}(k)$ index
if $k \in \text{Bucket}_i$ **then**
    Stop ▷ Key already exists; no action needed
else
    Insert (k, loc) into Bucket_i ▷ Add new key-locator pair
end if
if Bucket_i is full **then**
    Split Bucket_i ▷ Trigger bucket split to redistribute metadata
end if

Algorithm 2 GET Operation in the Storage Layer

Require: Key k

1: $i \leftarrow \text{Hash}(k)$ ▷ Compute hash index for the key
2: **if** $k \in \text{Bucket}_i$ **then**
3: $loc \leftarrow \text{Locator}(k)$ ▷ Retrieve locator for the key
4: Retrieve data from Body_{loc} ▷ Access data body using the locator
5: **return** data ▷ Return the retrieved data
6: **else**
7: **return** “Key Not Found” ▷ Handle missing key scenario
8: **end if**

3.6.4 SD2DS Variants

SD2DS includes several variants. These variants extend its functionality. They address specific challenges in distributed data management. They improve fault tolerance. They support persistent storage which makes SD2DS adaptable for a wide range of applications.

The **SD2DS$_{CL}$ (Component Locator)** variant improves data retrieval efficiency. It uses precise locators. These locators map metadata to data bodies. It reduces latency. It supports fault tolerance through replication. This ensures high availability. It also ensures reliability in distributed environments[89].

The SD2DS$_{CV}$ (Component Versioning) enables versioning at a component level, which will trace the history of changes in data over time. The key applications include those needing auditability, rollbacks, and regulatory compliance [78].

The **SD2DS$_{BS}$ (Bucket Splitting)** variant of SD2DS addresses efficient bucket splitting so that the load is well distributed and the system can scale. It dynamically rebalances metadata with every split, which minimizes the overhead of splitting and thus generally keeps performance efficient [74].

The **SD2DS$_{PS}$ (Persistent Storage)** variant ensures long-term durability by storing data bodies on persistent media such as SSDs or hard disks. It provides fault tolerance through replication and backup mechanisms, making it suitable for applications requiring high data reliability [61].

3.7 Conclusion

This chapter presented the development, basic concepts, and applications of SDDS. It also included advanced variants like SD2DS. SDDS solves the limitations of traditional centralized systems. They provide scalability, fault tolerance, and efficient data search. They rely on decentralized architectures and adaptive data distribution methods.

We discussed the main classes of SDDS: unidimensional, multidimensional,

and trie-based structures. The discussion also involved advanced variants like LH* and RP*. Each variant addresses specific issues in data management, which offers scalability and performance features.

In the next chapter, we focus on our architecture, which is inspired by previous techniques like SD-SQL server, the RP*, K-RP* with the SD2DS.

Chapter 4

The global architecture of MDNOS

4.1 Introduction

This chapter describes the architecture and design of our system called MD-NOS (Multi-Dimensional NoSQL based on SDDS). MD-NOS is a hierarchical system. It meets modern demands in IoT, Fog, and cloud contexts. The system has three layers.The first layer is IoT clients. They generate data (get it from application) . The second layer is KV-MDSS (Key-Value Multi-Dimensional Storage System). It is located at the Fog layer for low-latency processing. The third layer is SD-PGSQL (Scalable Distributed PostgreSQL). It is in the cloud for centralized storage and advanced analytics.MD-NOS uses dynamic partitioning techniques. These include RP* for single-key data and K-RP* for multi-dimensional indexing. These techniques ensure scalability and fault tolerance.The system automatically splits segments. This happens when metadata thresholds are triggered. This optimizes data distribution and query performance.

Also, explains the system's architecture, metadata management, and operational workflows. It also describes how the system balances scalability, availability, and consistency.

4.2 The Architecture of Our System

MD-NOS has an advanced architecture. It has a hierarchy with several layers. It combines cloud computing, fog, and IoT technology. It tackles the difficulties associated with contemporary data administration. The system is shown in Figure 4.2.

There are three main layers in our system. Each layer is unique and they are related to each other. IoT clients make up the first layer. The fundamental data sources are IoT clients. KV-MDSS is the second layer. The fog layer is where KV-MDSS functions. Decentralized data handling is made possible by it. It guarantees effective data processing.

The SD-PGSQL layer is the third one. The cloud layer is where SD-PGSQL functions. It offers reliable storage. Advanced query processing is supported. Every layer performs specific functions. Scalability is ensured by the system. It guarantees processing with minimal latency and a high degree of consistency, and persistent storage.

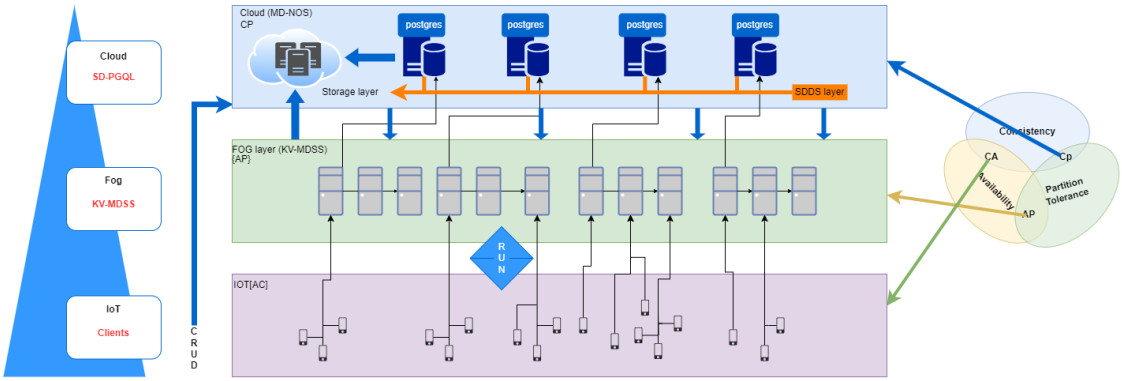

Figure 4.1: The system global architecture (MD-NOS).

4.3 SD-PGSQL Layer (Cloud Layer)

The **SD-PGSQL** (Scalable Distributed PostgreSQL) layer is the cloud-based component of the MD-NOS architecture, designed to provide centralized storage, strong consistency, and advanced query processing capabilities. It extends PostgreSQL with dynamic partitioning, scalable table management, and fault tolerance mechanisms,

making it suitable for handling large-scale, multi-dimensional datasets.

This section explores the design, functionalities, and workflows of the SD-PGSQL layer, with a focus on metadata management, database initialization, stored procedures, triggers, and the role of views.

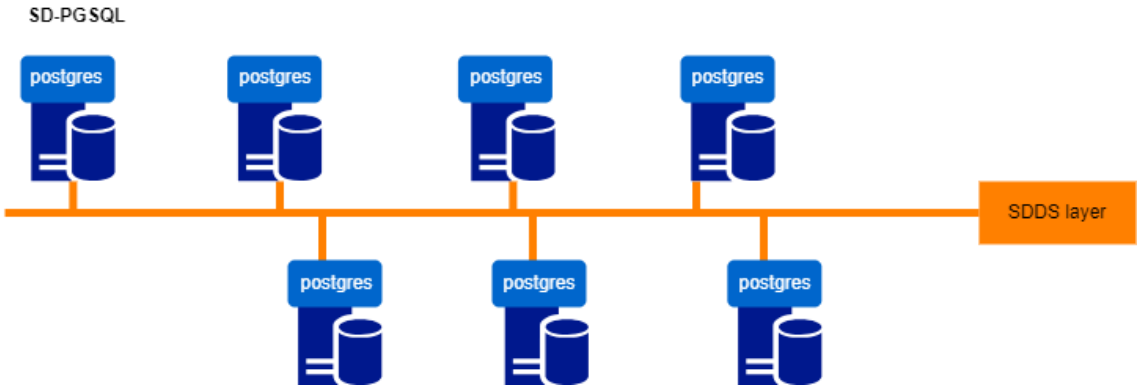

Figure 4.2: SD-PGSQL scalable distributed PostgreSQL.

4.3.1 Overview of SD-PGSQL

PostgreSQL is used in the SD-PGSQL layer. By using distributed instances of PostgreSQL. Additionally, they are able to tolerate error. This guarantees the system can manage increasing data requirements. Even when it malfunctions, it remains dependable. Two metadata tables are utilized by the layer. The tables are called `RP` and `KRP`. Single-key tables are maintained using the `RP` metatable. Multi-dimensional tables are controlled by the `KRP` metatable.

The metatables contains tables segments' names, IP addresses, Interval boundaries, sizes, and the thresholds for the maximum size.

The features of the SD-PGSQL layer are:

- **Dynamic Partitioning**: Automatically splits data segments when they exceed a predefined size threshold, ensuring balanced data distribution and efficient query performance.
- **Strong Consistency**: Adheres to the Consistency and Partition Tolerance (CP) properties of the CAP theorem, ensuring data integrity across distributed nodes.

- **Advanced Query Processing**: Supports complex SQL queries, including range queries and aggregations.
- **Fault Tolerance**: Implements replication and recovery mechanisms to maintain data availability and durability during system failures.

4.3.2 Metadata Tables: `RP` and `KRP`

The SD-Postgres system's metadata is made up of the `RP` and `KRP` tables. They make it possible to manage single-key tables effectively. They also make it possible to organize multi-dimensional tables. Data is arranged using this system. During the database initialization stage, the system generates these tables. They hold important data information as described in Table 4.1.

4.3.2.1 Structure of `RP` and `KRP` Tables

Field	Description
`table_name`	Name of the table.
`segment_name`	Name of the segment.
`ip_address`	IP address of the instance hosting the segment.
`interval_start`	Start of the key interval for the segment (text for `RP`, JSON for `KRP`).
`interval_end`	End of the key interval for the segment (text for `RP`, JSON for `KRP`).
`size`	Current size of the segment.
`max_size`	Maximum size threshold for the segment. Splitting occurs when `size > max_size`.
`dimension`	Current dimension used for partitioning (only for `KRP`).

Table 4.1: Structure of `RP` and `KRP` Metadata Tables

4.3.2.2 Database Initialization

In the database initialization phase, SD-PGSQL creates the RP and KRP metatables. The RP table manages single-key tables. The KRP table manages multi-dimensional tables. They ensure structured data organization. They enable efficient data retrieval.

Below is the SQL code that outlines the database initialization process. It includes the creation of these metadata tables which includes their configuration.

```
1  CREATE OR REPLACE FUNCTION create_database(input_database_name VARCHAR) RETURNS
   ↪  VOID AS $$
2  BEGIN
3      EXECUTE format('CREATE DATABASE %I', input_database_name);
4      EXECUTE format('\c %I', input_table_name);
5      EXECUTE 'CREATE TABLE RP(table_name VARCHAR(100), segment_name VARCHAR(100),
6                               ip_address VARCHAR(15), interval_start TEXT,
7                               interval_end TEXT, size INT DEFAULT 0, max_size INT
   ↪  DEFAULT 5)';
8      EXECUTE 'CREATE TABLE KRP (table_name VARCHAR(100), segment_name
       ↪  VARCHAR(100),
9                                 ip_address VARCHAR(15), interval_start JSON,
10                                interval_end JSON, size INT DEFAULT 0, max_size INT
   ↪  DEFAULT 5,
11                                dimension INT DEFAULT 0)';
12 END;
13 $$ LANGUAGE plpgsql;
```

4.3.3 Handling One-Dimensional Tables

One-dimensional tables in SD-Postgres use the RP metadata table. The RP table organizes single-key data. It accesses single-key data. Each record has a unique key. This ensures streamlined data retrieval. It also ensures streamlined data manipulation.

Below are the stored procedures and triggers that manage one-dimensional tables and automate data operations. They maintain consistency and enhance the

system's performance.

4.3.3.1 Stored Procedures and Triggers

- `create_table`: Creates a new one-dimensional table and initializes its metadata in the `RP` table. It also creates an initial segment and a corresponding view for querying.

```
1  CREATE OR REPLACE FUNCTION create_table(input_table_name VARCHAR, columns
   ↪  TEXT) RETURNS VOID AS $$
2  DECLARE
3      initial_segment_name VARCHAR := input_table_name || '_0';
4  BEGIN
5      -- Create the initial table segment with custom columns
6      EXECUTE format('CREATE TABLE %I (%s)', initial_segment_name, columns);
7
8      -- Insert initial metadata for the segment
9      EXECUTE format('INSERT INTO RP(table_name, segment_name, ip_address,
       ↪  interval_start, interval_end, size, max_size)
10                     VALUES (%L, %L, %L, %L, %L, 0, 5)',
11                     input_table_name, initial_segment_name, '10.2.1.1',
                       ↪  '-INFINITY', '+INFINITY');
12
13     -- Create the view for the table
14     EXECUTE format('CREATE VIEW %I AS SELECT * FROM %I', input_table_name,
       ↪  initial_segment_name);
15 END;
16 $$ LANGUAGE plpgsql;
```

- `insert_into_table`: Inserts data into a one-dimensional table. It identifies the appropriate segment based on the key interval and updates the `RP` metadata. If the segment size exceeds the threshold (`size > max_size`), the `split_segment` procedure is triggered.

- `update_table`: Updates a record in a one-dimensional table. It locates the segment containing the record, performs the update, and ensures the `RP` metadata remains consistent.

- `delete_from_table`: Deletes a record from a one-dimensional table. It locates the segment containing the record, performs the deletion, and updates the `RP` metadata to reflect the change in segment size.

- `split_segment`: Splits a one-dimensional segment when it exceeds the size threshold (`size > max_size`). It creates a new segment, redistributes the data, and updates the `RP` metadata to reflect the new intervals and sizes. The code below shows a part of the split function routin

```
1  -- Create the new segment table on the target instance
2     PERFORM dblink_exec(target_conn, format('CREATE TABLE %I AS SELECT *
      ↪  FROM %I WHERE name >= %L',
3                                             next_segment_name,
                                              ↪  input_segment_name,
                                              ↪  segment_interval_start));
4
5     -- Update size and metadata for the original segment
6     EXECUTE format('UPDATE rp SET size = size / 2, interval_end = %L WHERE
      ↪  segment_name = %L',
7                    segment_interval_start, input_segment_name);
```

4.3.3.2 Triggers for `RP` Tables (One-Dimensional)

- `trigger_split_segment`: Automatically triggers the `split_segment` procedure when the size of a segment exceeds the predefined threshold (`size > max_size`). This ensures balanced data distribution and efficient query performance.

4.3.3.3 Example: Family Table

As showing in the code below, the creation of the one-dimensional table "`family`". It stores the information about families. It includes a unique key for identification, family name , address ,and includes their specific situation which may classify them as orphans or widows. This table provides a structured way to organize data.

```
-- Create the family table
SELECT create_table('family', 'key TEXT PRIMARY KEY, name TEXT, address TEXT,
↪  situation TEXT');

-- Insert data into the family table
SELECT insert_into_table('family', 'F1', 'ahmed ali', 'Elkarimia ', 'Widow');
SELECT insert_into_table('family', 'F2', 'samir omar', 'Harchoun', 'Orphan');
```

After insertion, the RP table is updated as follows:

table_name	segment_name	ip_address	interval_start	interval_end	size	max_size
family	family_0	10.2.1.1	$-\infty$	$+\infty$	2	5

Table 4.2: State of RP Table for family

4.3.3.4 Segment Splitting in the family Table

The system sets a predefined threshold, as an example, 5 records. When a segment exceeds this threshold, the system splits the segment. It divides the segment into two parts and transfers half of its records into a new instance. This ensures efficient data organization. It also ensures data accessibility. Below is an example that shows the insertion of additional data that triggers the split operation. The example illustrates the system's ability to adapt the growing data volumes, maintain optimal performance, and preserve the data structure.

```
-- Insert more data into the family table
SELECT insert_into_table('family', 'F3', 'salah khaled', 'oued fodda', 'Orphan');
SELECT insert_into_table('family', 'F4', 'khalil jamal', 'chlef', 'Widow');
SELECT insert_into_table('family', 'F5', 'fares nabil', 'tenes', 'Orphan');
```

After insertion, the RP table is updated as follows:

table name	segment name	ip address	interval start	interval end	size	max size
family	family_0	10.2.1.1	$-\infty$	$+\infty$	5	5

Table 4.3: State of RP Table Before Split

The split_segment function is triggered automatically, and the RP table is updated as follows:

table name	segment name	ip address	interval start	interval end	size	max size
family	family_0	10.2.1.1	$-\infty$	F3	2	5
family	family_1	10.2.1.2	F3	$+\infty$	3	5

Table 4.4: State of RP Table After Split

4.3.4 Handling Multi-Dimensional Tables

Multi-dimensional tables in SD-Postgres use the KRP metadata table. The KRP table organizes data with multiple dimensions as keys. The table ensures a robust approach. It ensures a scalable approach to handling complex data structures. Below are the stored procedures and the triggers that manage multi-dimensional tables, automate operations, maintain data integrity, and optimize performance.

4.3.4.1 Stored Procedures and Triggers

- create_md_table: Creates a new multi-dimensional table and initializes its metadata in the KRP table. It also creates an initial segment and a corresponding view for querying.

```
1 CREATE OR REPLACE FUNCTION create_md_table(input_table_name VARCHAR,
  ↪ columns TEXT) RETURNS VOID AS $$
2 DECLARE
```

```
    initial_segment_name VARCHAR := input_table_name || '_0';
BEGIN
    -- Create the initial table segment with custom columns
    EXECUTE format('CREATE TABLE %I (%s)', initial_segment_name, columns);

    -- Insert initial metadata for the segment
    EXECUTE format('INSERT INTO KRP (table_name, segment_name, ip_address,
    ↪  interval_start, interval_end, size, max_size, dimension)
                    VALUES (%L, %L, %L, %L, %L, 0, 5, 0)',
                    input_table_name, initial_segment_name, '10.2.1.1',
                    '{"x": "-INFINITY", "y": "-INFINITY"}', '{"x":
                    ↪  "+INFINITY", "y": "+INFINITY"}');

    -- Create the view for the table
    EXECUTE format('CREATE VIEW %I AS SELECT * FROM %I', input_table_name,
    ↪  initial_segment_name);
END;
$$ LANGUAGE plpgsql;
```

- `insert_into_md_table`: Inserts data into a multi-dimensional table. It identifies the appropriate segment based on the multi-dimensional intervals and updates the `KRP` metadata. If the segment size exceeds the threshold (`size > max_size`), the `split_KRP_segment` procedure is triggered.

- `update_md_table`: Updates a record in a multi-dimensional table. It locates the segment containing the record, performs the update, and ensures the `KRP` metadata remains consistent.

- `delete_from_md_table`: Deletes a record from a multi-dimensional table. It locates the segment containing the record, performs the deletion, and updates the `KRP` metadata to reflect the change in segment size.

- `split_KRP_segment`: Splits a multi-dimensional segment when it exceeds the size threshold (`size > max_size`). It creates a new segment, redistributes the data, and updates the `KRP` metadata to reflect the new intervals, sizes, and split dimension.

4.3.4.2 Triggers for KRP Tables (Multi-Dimensional)

- The `trigger_split_KRP_segment` is a specialized trigger. It activates the `split_KRP_segment` procedure automatically. This happens when a segment's size exceeds the predefined threshold. The condition is `size > max_size`. This ensures even data distribution. It also optimizes query performance for multi-dimensional datasets. The trigger splits segments dynamically. It maintains an efficient structure. It ensures a scalable structure. This is crucial for handling the complexities of multi-dimensional data management.

4.3.4.3 Example: Kafil Table

The `kafil` table is a multi-dimensional table. It stores detailed information about sponsors. It includes spatial coordinates (`x`, `y`), the sponsor's name, and their phone number. This provides a comprehensive approach with a structured approach to manage complex data.

```
1 -- Create the kafil table
2 SELECT create_md_table('kafil', 'x TEXT, y TEXT, name TEXT, phone TEXT');
3
4 -- Insert data into the kafil table
5 SELECT insert_into_md_table('kafil', '{"x": "10", "y": "20"}', 'khalil',
  ↪  '123-456-7890');
6 SELECT insert_into_md_table('kafil', '{"x": "30", "y": "40"}', 'mohammed',
  ↪  '987-654-3210');
```

After insertion, the KRP table is updated as follows:

Table name	Segment name	IP adress	Start Interval	End Interval	Size	Max Size	Dim
kafil	kafil_0	10.2.1.1	{x: $-\infty$, y: $-\infty$}	{x: $+\infty$, y: $+\infty$}	2	5	0

Table 4.5: State of KRP Table for `kafil`

4.3.4.4 Segment Splitting in the `kafil` Table

The system sets a predefined threshold as example, 5 records. When a segment exceeds this threshold, the system splits the segment in two parts and transfers half of its recodes into the next available instance.

Below is the code that shows the insertion of additional records which provides the spliting operation and preserves the data structure in the next instance.

```
-- Insert more data into the kafil table
SELECT insert_into_md_table('kafil', '{"x": "50", "y": "60"}', 'ahmed',
↪  '555-123-4567');
SELECT insert_into_md_table('kafil', '{"x": "70", "y": "80"}', 'ali',
↪  '555-987-6543');
SELECT insert_into_md_table('kafil', '{"x": "90", "y": "100"}', 'salah',
↪  '555-456-7890');
```

After insertion, the `KRP` table is updated as follows:

Table name	**Segment name**	**IP adress**	**Start Interval**	**End Interval**	**Size**	**Max Size**	**Dim**
kafil	kafil_0	10.2.1.1	{"x": "−∞", "y": "−∞"}	{"x": "+∞", "y": "+∞"}	5	5	0

Table 4.6: State of `KRP` Table Before Split

The `split_KRP_segment` function is triggered automatically, and the `KRP` table is updated as follows:

Table name	**Segment name**	**IP adress**	**Start Interval**	**End Interval**	**Size**	**Max Size**	**Dim**
kafil	kafil_0	10.2.1.1	{"x": "−∞", "y": "−∞"}	{"x": "50", "y": "60"}	2	5	1
kafil	kafil_1	10.2.1.2	{"x": "50", "y": "60"}	{"x": "+∞", "y": "+∞"}	3	5	0

Table 4.7: State of `KRP` Table After Split

4.3.5 Role of Views in Querying and Aggregation

Views play a crucial role in the SD-PGSQL system. They provide a unified interface. The interface queries data across multiple segments. It aggregates data across multiple segments. When a table is created, the system creates a corresponding view which gathers all segments of the table. It forms a single logical entity. This allows clients to perform queries on the entire dataset. Clients do not need to know the underlying table distribution.

4.3.5.1 Creation of Views

For example, when the `family` table is created, a view named `family` is also created:

```
CREATE VIEW family AS SELECT * FROM family_0;
```

After the segment splits. A new segments are created. The view updates and includes all segments.

```
CREATE OR REPLACE VIEW family AS
SELECT * FROM family_0
UNION ALL
SELECT * FROM family_1;
```

4.3.5.2 Use of Views for Aggregation and Filtering

Views allow advanced querying and aggregation across all segments of a table. Users can perform operations like the following:

- **Aggregation**: Calculate the total number of families in the `family` table:

```
    SELECT COUNT(*) FROM family;
```

- **Filtering**: Retrieve all families with a specific situation (e.g., orphan):

```
    SELECT * FROM family WHERE situation = 'Orphan';
```

4.4 KV-MDSS (Fog Layer)

Our **KV-MDSS** (Key-Value Multi-Dimensional Storage System) operates **in memory (RAMs)** at the fog layer, acting as an intermediary between IoT clients and the cloud layer, providing low-latency responses, decentralized storage, and efficient query processing. Optimized for both one-dimensional and multi-dimensional data, **KV-MDSS** uses two architectures: **RP*-SD2DS** for one-dimensional data and **ZK-RP*** for multi-dimensional data. These ensure fault tolerance, scalability, and real-time analytics, making it ideal for different applications. Further details on the fog layer are discussed in the next chapter.

4.4.1 Component of KV-MDSS

KV-MDSS is a decentralized key-value storage system designed for large-scale, dynamically distributed files. It operates without a central coordinator, eliminating single points of failure and enhancing scalability. The system is built around two architectures:

- **RP*-SD2DS**: Optimized for one-dimensional data, using a two-layer approach for efficient range queries.
- **ZK-RP***: Extends capabilities to multi-dimensional data with a **Multi-Dimensional Index (MDI)** based on a K-D tree.

These architectures enable **KV-MDSS** to handle real-time data processing for IoT systems, location-based services, and real-time analytics.

4.4.2 RP*-SD2DS Architecture

The **RP*-SD2DS** (Range Partitioning Scalable Distributed Two-Layer Data Structure)[102] architecture is specifically designed to handle one-dimensional data efficiently. It employs a two-layer approach to separate metadata management from data storage, which reduces overhead and significantly improves performance, particularly for range queries.

The architecture is structured into two layers: the **File Layer** and the **Storage Layer**. The File Layer is responsible for managing data keys and locators, utilizing a **B+ tree index** to ensure efficient key management and quick access to data locations. This layer ensures scalability and balanced data distribution. On the other hand, the Storage Layer handles the storage of complete data, providing robust and scalable storage for large, dynamically distributed files. The separation of metadata and data storage minimizes the overhead associated with data retrieval and insertion, making the system highly efficient for range queries.

4.4.2.1 System Design

The **RP*-SD2DS** system is composed of multiple servers, forming a multi-computer cluster. Each server contains two primary components: the **First Layer Bucket** and the **Second Layer Bucket**. The First Layer Bucket manages data keys and locators, ensuring efficient key management and quick access to data locations. The Second Layer Bucket stores the complete data, providing robust storage for large and dynamically distributed files. The system dynamically redistributes data across servers during splitting operations, ensuring high availability and fault tolerance. This dynamic redistribution minimizes system downtime, addressing a common bottleneck in traditional NoSQL systems.

4.4.2.2 First Layer Algorithms

The **RP*-SD2DS** architecture relies on several key algorithms to ensure efficient data management. The **Insertion** algorithm processes client requests by separating them into headers and bodies. The header contains the key and locator, while the body contains the actual data. If the second layer is full, the body is sent to another server, and the first layer stores the key and locator. The **Split** algorithm is triggered when a bucket reaches its maximum capacity, transferring half of the data to a new server to maintain balanced data distribution.

The **Search** algorithm processes both exact and range requests. For exact requests, the system identifies the key and follows the locator to extract the data from the second layer. For range requests, it identifies relevant keys within the

specified range and retrieves the corresponding data. These algorithms ensure that the **RP*-SD2DS** architecture can handle large volumes of one-dimensional data efficiently.

4.4.3 ZK-RP* Architecture

The **ZK-RP*** architecture extends **RP*-SD2DS** to handle multi-dimensional data, introducing a **Multi-Dimensional Index (MDI)** based on a K-D tree for efficient query processing.

4.4.3.1 Z-Order (Morton Code)

The **linearization with Z-order curve** (Morton code) as shown in Figure 4.3 linearizes multi-dimensional data into a one-dimensional space by interleaving the bits of the coordinates[79]. ensuring that nearby points in multi-dimensional space remain close in the linearized space. For example, consider LBS (localization based-system) [70] coordinates (latitude, longitude):

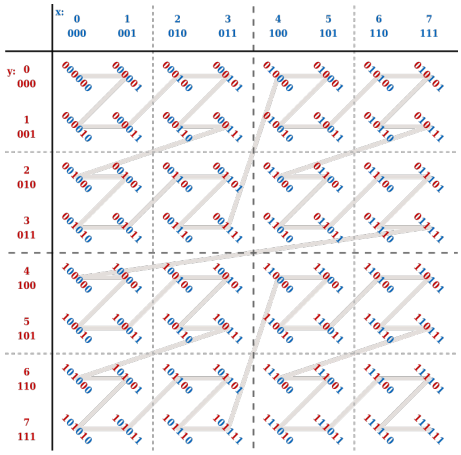

Figure 4.3: Z-order (Morton Code).

- Latitude: 37.7749 (binary: `01001010111101011100000100001`)
- Longitude: -122.4194 (binary: `11110110101011100000100001001`)

The Z-order value is computed by interleaving the bits of latitude and longitude:

$$\text{Z-value} = 01101101010111100100000100010001$$

We use this z-value in the KV-MDSS layer as a key (uni-key) , This linearization enables efficient range queries and nearest neighbor searches.

4.4.3.2 Node Architecture

Each node in the **ZK-RP*** architecture contains two main components: the **Multi-Dimensional Index Layer (MDI)** and the **Data Storage Index Layer (DSI)**. The MDI Layer is responsible for managing the global image of data distribution in the system. It uses a K-D tree to divide the space into subspaces, ensuring balanced data distribution and efficient query processing. The DSI Layer, on the other hand, manages buckets by their IDs, storing details such as capacities, intervals, and current split dimensions. This ensures that data is distributed evenly across nodes, maintaining high availability and fault tolerance.

4.4.3.3 Distributed Data Store

Data in the **ZK-RP*** architecture is distributed across nodes following a global addressing policy. This ensures load balancing and efficient data access. To maintain availability, the system uses **Image Adjustment Messages (IAM)** to synchronize changes to the data's addressing state during scaling up or shrinking. This synchronization ensures that all nodes have an up-to-date view of the data distribution, preventing addressing errors and maintaining system reliability.

4.4.3.4 Multi-Dimensional Index Layer (MDI)

The **Multi-Dimensional Index Layer (MDI)** is a critical component of the **ZK-RP*** architecture. It uses a K-D tree to split the space into subspaces, ensuring efficient indexing and querying of multi-dimensional data. The K-D tree divides the space using the median value of the current dimension, ensuring balanced data distribution.

Each subspace is mapped to a bucket, which stores the actual data. This approach allows the system to handle large-scale multi-dimensional datasets effectively, making it suitable for applications such as location-based services and real-time analytics. The MDI layer ensures that multi-dimensional queries, such as range queries and nearest neighbor searches, are processed efficiently, providing robust support for complex data management tasks.

4.4.3.5 Query Processing

The ZK-RP* supports many queries as mentioned below:

- **Subspace Lookup**: Finds the subspace for a given point.
- **Insertion**: Inserts new data points and redistributes data if necessary.
- **Range Query**: Processes multi-dimensional range queries by scanning relevant buckets.
- **Nearest Neighbor Query**: Finds the k-nearest neighbors of a given point.

4.4.4 Integration with SD-PGSQL (Cloud Layer)

KV-MDSS works in tandem with **SD-PGSQL**, the cloud-based component of the MD-NOS architecture. While KV-MDSS handles decentralized storage and real-time query processing at the Fog, SD-PGSQL provides centralized, strongly consistent data storage and complex analytical processing in the cloud see Figure 4.4.

- **Single-Key Tables (RP in SD-PGSQL)**: Represented in KV-MDSS using the **RP*-SD2DS** architecture. Each table in SD-PGSQL is represented as a view that assembles segments in KV-MDSS, ensuring efficient data distribution and query processing.
- **Multi-Dimensional Tables (KRP in SD-PGSQL)**: Represented in KV-MDSS using the **ZK-RP*** architecture. The multi-dimensional tables in SD-PGSQL are handled by the MDI layer in KV-MDSS, ensuring efficient query processing and data distribution.

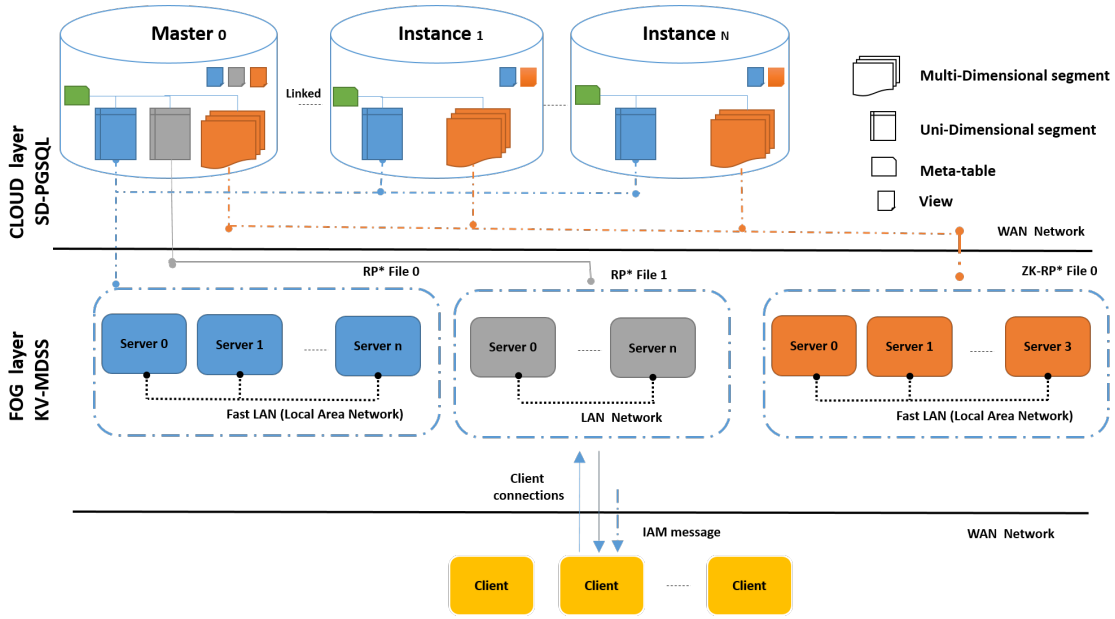

Figure 4.4: The components system of (MD-NOS).

The integration of KV-MDSS and SD-PGSQL ensures a seamless flow of data between the Fog and the cloud, providing a comprehensive solution for modern data-intensive applications.

4.4.5 Data representation

The data representation differs in each layer. In the fog layer, it is simplified into two forms: first, the data is represented by a single key, either in the RP* cluster for one-dimensional data or in the ZK-RP* cluster for multi-dimensional data. The main key for a uni-dimensional table in the SD-PGSQL is the primary key, while for a multi-dimensional table, it is represented by the Z-value

4.5 Clients (IoT layer)

The IoT clients in the system play a crucial role in interacting with both the cloud layer (SD-PGSQL) and the fog layer (KV-MDSS). Each IoT client maintains two partial images of the data distribution in the system, stored in SQLite tables. These images allow the client to make informed decisions about where to route queries based on the type of operation and the required consistency guarantees. This section

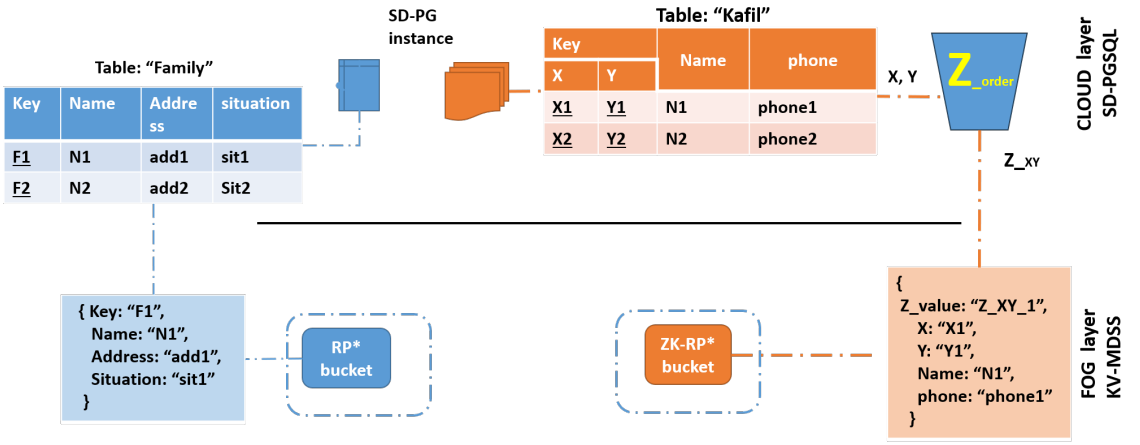

Figure 4.5: Data representation in each layer.

provides a detailed explanation of how IoT clients interact with the system, the structure of their partial images, and the decision-making process for query routing.

4.5.1 Partial Images in IoT Clients

Each IoT client maintains two SQLite tables that serve as partial images of the data distribution in the system. These tables contain metadata about the locations of data segments in both the cloud layer (SD-PGSQL) and the fog layer (KV-MDSS). The two tables are:

- **SD-PGSQL Image**: This table contains metadata about the data segments stored in the cloud layer (SD-PGSQL). It includes information such as segment names, IP addresses, interval ranges. This image is used for consistent queries that require strong consistency guarantees.
- **KV-MDSS Image**: This table contains metadata about the data segments stored in the fog layer (KV-MDSS). It includes information such as bucket IDs, node addresses, interval ranges. This image is used for non-consistent queries that prioritize availability and low latency.

The images allow the IoT client to have a partial view of the data distribution, enabling efficient query routing without requiring a global view of the system. This approach reduces the overhead associated with query processing and ensures that the client can make quick decisions based on the type of query.

4.5.2 Query Routing Based on Query Type

The IoT client has the flexibility to choose between the cloud layer (SD-PGSQL) and the fog layer (KV-MDSS) based on the type of query. The decision is made based on the consistency requirements of the query:

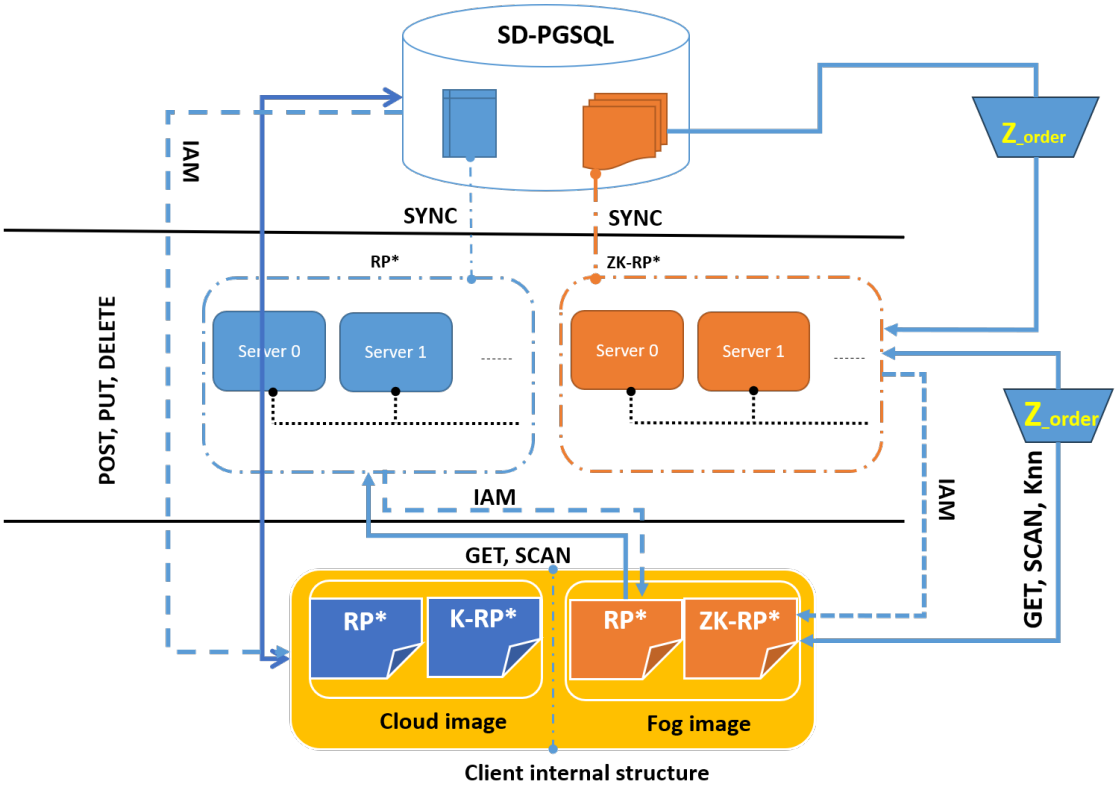

Figure 4.6: Query Routing .

- **Consistent Queries**: For queries that require strong consistency guarantees, the client routes the query to the cloud layer (SD-PGSQL). These queries include:
  - **POST (Insertion)**: Inserts a new record into the system.
  - **PUT (Update)**: Updates an existing record in the system.
  - **DELETE**: Removes a record from the system.

  These operations are routed to the appropriate table in SD-PGSQL, either **RP** (for single-key tables) or **KRP** (for multi-dimensional tables), depending on the data structure.

- **Non-Consistent Queries**: For queries that prioritize availability and low latency over strong consistency, the client routes the query to the fog layer (KV-MDSS). These queries include:
  - **GET**: Retrieves a record from the system.
  - **SCAN**: Performs a range query to retrieve multiple records within a specified range.

  These operations are routed to the appropriate architecture in KV-MDSS, either **RP*-SD2DS** (for single-key data) or **ZK-RP*** (for multi-dimensional data), depending on the data structure.

4.5.3 Advantages of Client Routing

The use of partial images in IoT clients provides several advantages:

- **Efficient Query Routing**: The partial images allow the client to quickly determine the location of the data, reducing the time required for query processing.
- **Reduced Overhead**: By maintaining a localized view of the data distribution, the client avoids the need for frequent communication with the central system, reducing network overhead.
- **Flexibility**: The client can dynamically choose between the cloud layer and the fog layer based on the query type, ensuring optimal performance for both consistent and non-consistent queries.

4.6 Conclusion

The MD-NOS architecture is a significant advancement in modern data management. It harmoniously integrates IoT, Fog, and cloud layers. The SD-PGSQL cloud layer provides centralized storage and advanced query processing. It ensures high consistency for complex analytical tasks.The KV-MDSS Fog layer handles high-velocity data streams. It offers low-latency processing and fault tolerance. This is

achieved through decentralized storage. RP-SD2DS* manages single-key data. ZK-RP* handles multi-dimensional indexing. IoT clients maintain partial snapshots of data distribution. They support smart routing of queries based on consistency requirements. Consistent queries, like updates and deletes, are sent to the cloud for high consistency. Non-consistent queries, such as reads and range scans, are served locally at the Fog for low latency. This smart routing capability provides optimal performance for various applications. These include smart cities, industrial IoT, and real-time analytics. MD-NOS offers a solid foundation for real-time data processing and analytics. It balances scalability, availability, and consistency.

The next chapter explains in detail the RP*SD2DS and also the ZK-RP* architecture, design, and its algorithms.

Chapter 5

KV-MDSS: Key-Value Store for Multidimensional Data

5.1 Introduction

In this chapter, we present **KV-MDSS**. It is a key-value storage system. It efficiently handles **one-dimensional** and **multi-dimensional** data. **KV-MDSS** solves the limitations of traditional NoSQL systems like MongoDB. It introduces two specialized architectures. **RP*-SD2DS**: A Range Partitioning Scalable Distributed Two-Layer Data Structure designed for handling one-dimensional data. It offers robust storage support for large and dynamically distributed files in multicomputer architectures, particularly excelling in range queries. **ZK-RP*SD2DS**: An extension of RP*-SD2DS designed for handling multi-dimensional data. It leverages a multi-dimensional index layer (MDI) to efficiently manage complex queries such as range queries and nearest neighbor searches.

KV-MDSS doesn't have a coordinator or master. High availability and fault tolerance are guaranteed. Data is dispersed across two distinct layers. When splitting operations happen, it minimizes system downtime, bypassing the bottleneck in conventional NoSQL systems.

5.2 The RP*-SD2DS Architecture

The **RP*-SD2DS** architecture is designed to handle **one-dimensional data** efficiently. It uses a two-layer buckets consisting of:

- **File Layer**: Manages and stores data keys with a locator pointing to the data. This layer ensures efficient key management and quick access to data locations. This layer is optimized for high availability and scalability.
- **Storage Layer**: Saves the complete data, ensuring robust storage for large and dynamically distributed files.

The two-layer approach divides the data into metadata and the data itself (body). Manage locators and keys in a single layer. The real data is kept on a different layer. As a result, the overheads related to data retrieval are decreased. Additionally, it lowers the overhead associated with data insertion. The design performs wonderfully when it comes to bottleneck issues. The keys and locators are swiftly located by the file layer. The storage layer retrieves the information (bodies).

Figure 5.1 shows the collaboration between the two layers, enabling efficient data storage and retrieval. This system is particularly effective for range queries. The architecture works on one-dimensional data, utilizing unique keys to efficiently manage range-based queries

5.2.1 The System Design

The **RP*-SD2DS** system consists of multiple servers and clients, forming a multicomputer cluster. Applications connect directly to the clients, which offer an API. This API supports get, put, delete, and scan methods, representing basic NoSQL queries. The scan method is specifically optimized for **range queries**, handling one-dimensional data. Each server comprises two components, referred to as buckets.

- **First Layer Bucket**: Manages data keys and locators. This bucket ensures efficient key management and quick access to data locations.

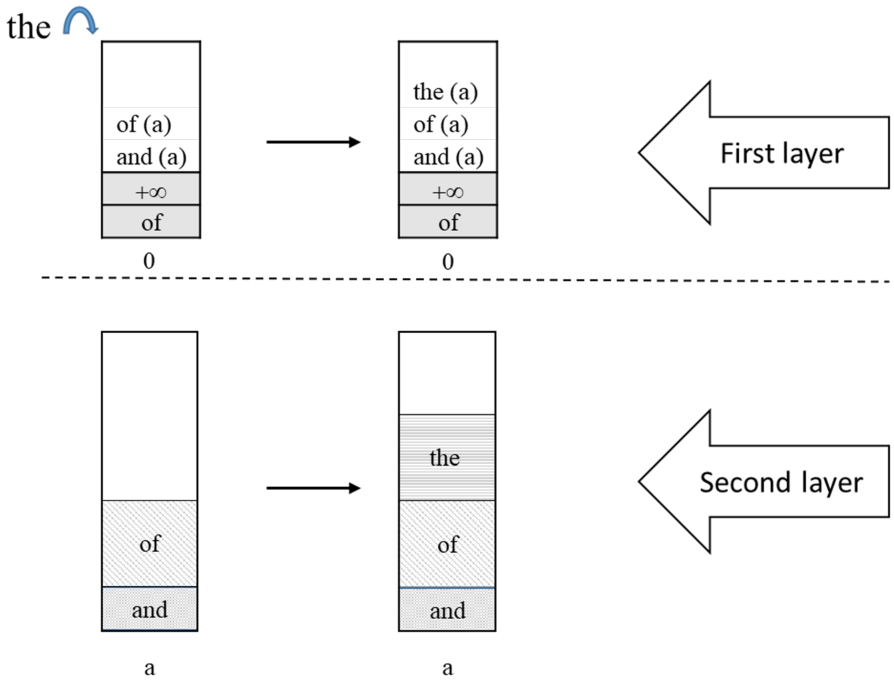

Figure 5.1: A simple SD2DS architecture.

- **Second Layer Bucket**: Stores the complete data, ensuring robust storage for large and dynamically distributed files.

The system handles scalability and fault tolerance. It redistributes data dynamically across servers during splitting operations. This keeps the system available, which keeps the system responsive as the data grows.

Figure 5.2 shows the main steps of the system's functioning. The steps include:

1. The client receives requests from the application. Put, set, update, remove, and scan(range) are among the requests.
2. The target server is recognized by the client based on its picture.
3. In response, the server sends an IAM.
4. The request is routed to the appropriate server by the server.
5. If there is not enough capacity on the server, the system transmits the body.
6. The locator is sent to the first layer by the system.

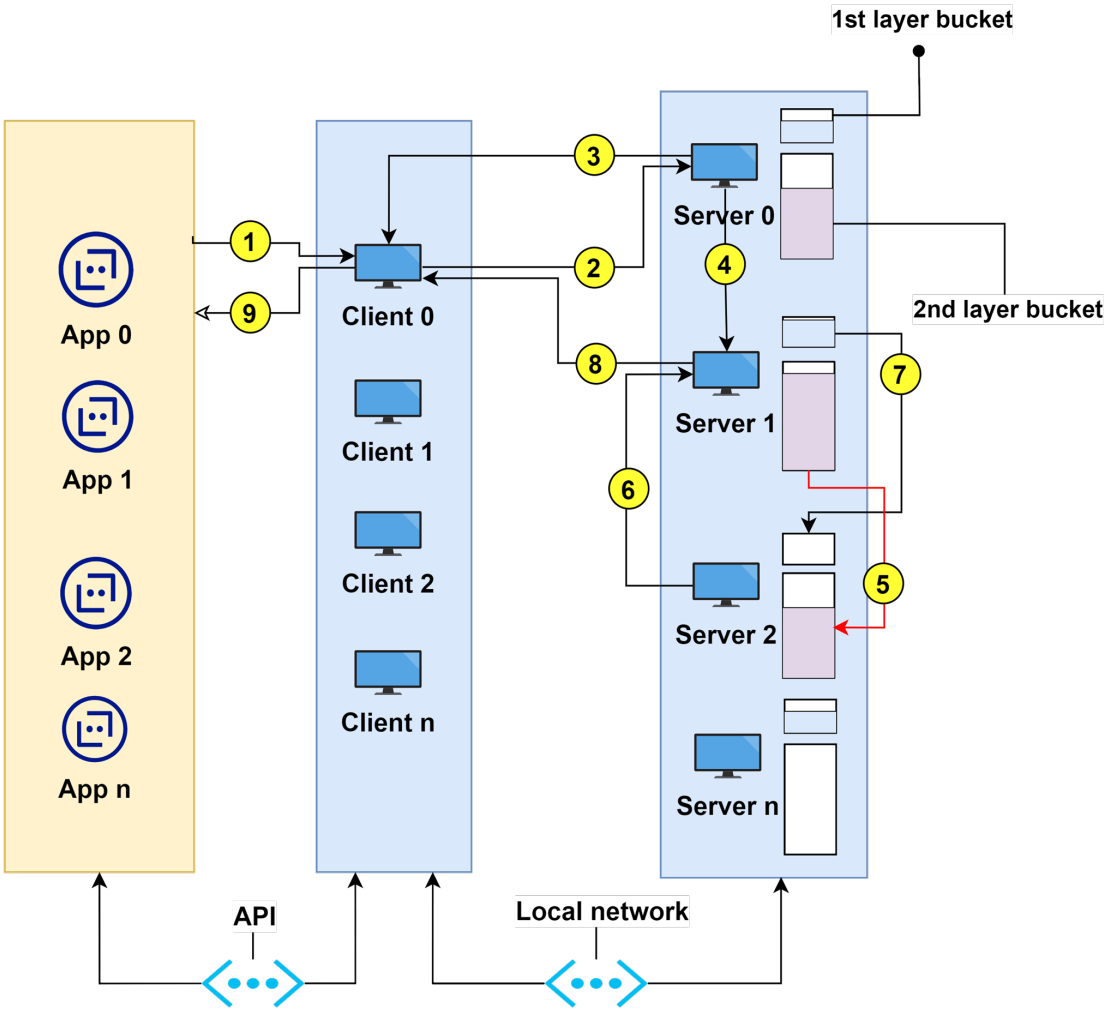

Figure 5.2: RP* SD2DS architecture.

7. If the first layer bucket is full, the system initiates the split process.

8. An IAM and insertion confirmation are sent to the customer.

9. The client answers the application.

5.2.1.1 The First Layer

The first layer, known as the file layer, holds data keys and locators, storing the records in a pre-ordered manner. A B+ tree index facilitates direct access to the stored data, ensuring efficient retrieval and scalability. This layer also handles system scalability by splitting the first-layer bucket upon reaching maximum capacity. As illustrated in Figure 5.3 half of the records are relocated to another bucket, maintaining balanced data distribution.

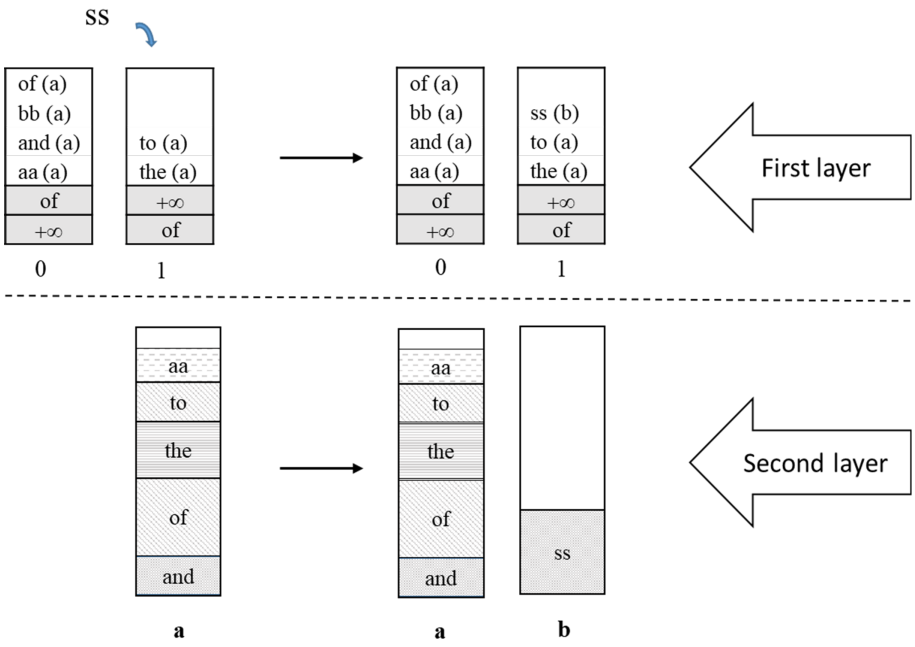

Figure 5.3: The SD2DS 2^{nd} layer after a new insertion with insufficient space.

The first layer bucket contains a header, which includes the bucket ID, the maximum number of records, the current number of records, and the bucket intervals with a pointer to the root index (B+ tree). Figure 5.4 shows the structure. The header contains the index size. and the data itself is composed of key-locator pairs.

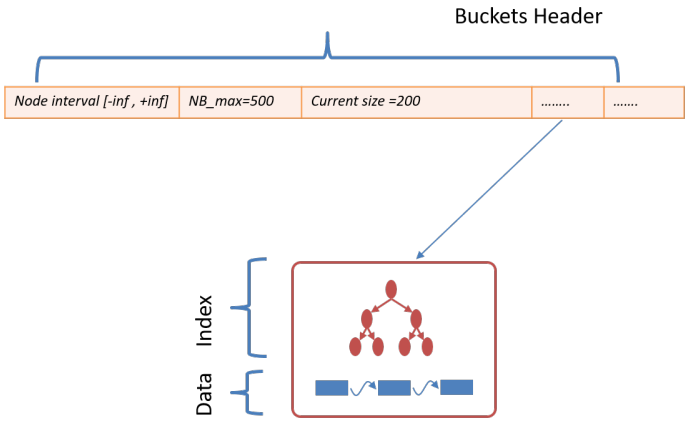

Figure 5.4: bucket structer.

5.2.1.2 First Layer Algorithms

1. **Insertion:** The system processes a client request by splitting it into a header and a body. The server saves the file and if the second layer is full, forwards

the body to another server. Upon obtaining the locator address, the server stores the key and locator in the first layer. The client is then notified of the successful insertion and receives an IAM.

Algorithm 3 Insert Query

```
1: Input:
2: key: Record's key.
3: file: Data in MiB.
4: nb: Current number of records.
5: nbMax: Maximum number of records per bucket.
6: if key ∈ localInterval then
7:     locator ← secondLayer.storeFile(key, file)
8:     bucket.insert(key, locator)
9:     sendClientSucceed()
10:    if nb > nbMax then
11:        split()
12:    end if
13: else
14:    S ← getImageServer(key)
15:    forward(key, file, S)
16:    sendClientIAM()
17: end if
```

The client directs requests to the first layer during every insert operation, relying on its local image. If the client's image is outdated, the chosen server reroutes the request to the correct server using its own image. The chosen server then sends an IAM back to the client. The correct server finalizes the insertion and responds to the client, confirming the successful insertion and including an Image Adjustment Message (IAM). The insertion process splits into two distinct operations:

- The system divided the request into a header and a body then transfers the body to the second layer. The server waits for the body locator.
- The system stores the header and its body locator in the first layer's

bucket. If the first layer reaches maximum capacity, it starts the splitting process (Algorithm 4).

Insertion-based two layers reduce transfer costs. The system splits only the first layer. The second layer acts as a fixed data store. Figure 5.5 shows this.

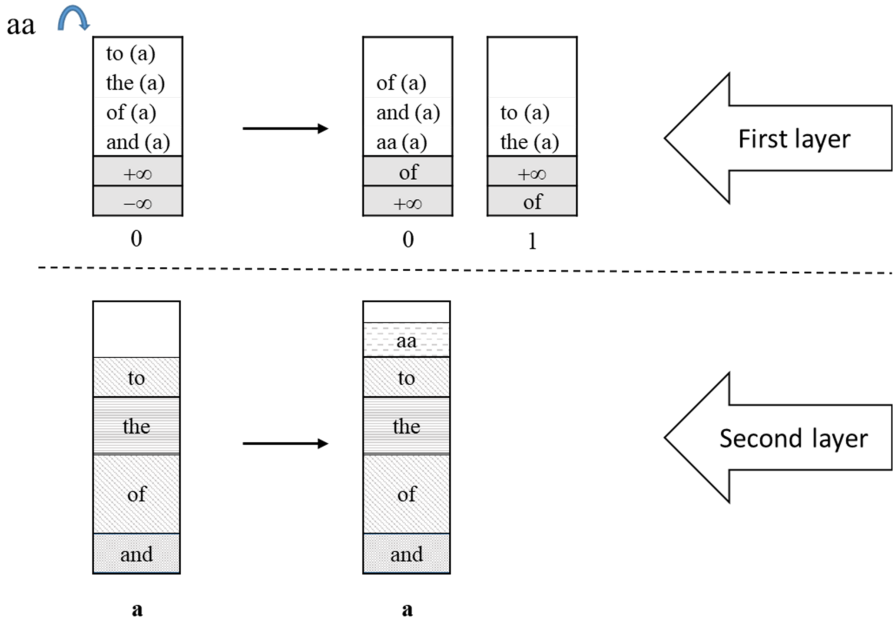

Figure 5.5: The SD2DS first layer after a splitting operation.

2. **Split:**

   Algorithm 4 shows the split process. Bucket B checks for overflow. It requests a new server for splitting. It sends a request to server M. Server M is considered available. The target server responds with an ack if it accepts the split. It denies the request if it refuses. Buckets B and M become left and right siblings.

   If the request is accepted, the system transfers half the data in the bucket of the first layer to the new server. The range of the new server becomes $]\lambda_M, \lambda_M]$. Here λ_M represents the central key of the bucket B, and λ_M represents the maximum key of the bucket B. If the request is rejected, the split server increments the server IDs. It continues this process until it finds an available server.

3. **Search Algorithm:** There are two types of search requests: exact requests and range requests. The first layer processes the search request. It identifies

Algorithm 4 Split

```
1: Input:
2: B: The overflowing bucket.
3: M: A new bucket.
4: Header: Bucket header.
5: keym: Middle key of the bucket.
6: repeat
7:     ack ← createNewBucket(M)
8:     if ack then
9:         λ_M ← keym(B)
10:        Λ_M ← Λ_B
11:        Header(M, λ_M, Λ_M)
12:        copyRecords(B, M, λ_M, Λ_M)
13:        removeRecords(B, λ_M, Λ_M)
14:    else
15:        M++
16:    end if
17: until ack
18: Λ_B ← keym(B)
19: Header(B, λ_B, Λ_B)
```

the key. It follows the locator to extract the data from the second layer. The client receives the requested data. In the event of an error, the client receives an IAM. This prevents future addressing errors.

For interval requests, clients obtain the interval from their image. They send requests to the buckets in the first layer in the interval. Clients receive IAMs in the event of an addressing error. If not, they receive locators. They use the locators to collect data from the second layers.

A server receives a range request. It checks its own interval included in the request. If the client's image is outdated, the server sends an IAM message. It checks the requested interval. It answers the client with existing data locators. It forwards the request to its sibling if needed. The client collects all IAMs.

Algorithm 5 Range Query (Client Side)

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

It updates its image. It uses the responses (locators) to gather data from the second layer.

Algorithm 6 Range Query (Server Side)

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

5.2.1.3 Storage Layer

The second layer, also known as the storage layer, occupies the majority of the server space and functions as a permanent partition. It ensures data persistence, scalability, and dynamic allocation of additional storage when the current layer reaches capacity. This design enables the system to handle large files efficiently, managing dynamically distributed data while guaranteeing high availability and fault tolerance.

By keeping the second layer independent of the first layer, the architecture allows uninterrupted data insertion and processing of specific GET requests. This separation enhances server availability, ensuring continuous operation even during the splitting process in the first layer.

Algorithm 7 Storage Algorithm

```
1: Input:
2: server.currentCapacity: Current capacity of the server.
3: file.size: Size of the file to store.
4: if server.currentCapacity ≥ file.size then
5:     store(file)
6:     server.currentCapacity − = file.size
7:     return locator
8: else
9:     return IAM_c
10: end if
```

Only the first layer is splittable by the system. It expands second-layer servers as required. During a new insertion, the system inserts data on a new server if the servers are fully loaded. Another server with enough space might be used as well. It gives the server that receives the request priority. The process is depicted in Figure 5.6.

The existing server determines whether the required capacity is available. It uses its key to insert the data. It modifies its existing capacity. The locator is its response. It rejects the insertion if capacity is not available. A Capacity Adjustment Message (IAM_c) is the result. The sender's capacity image is updated by the IAM_c.

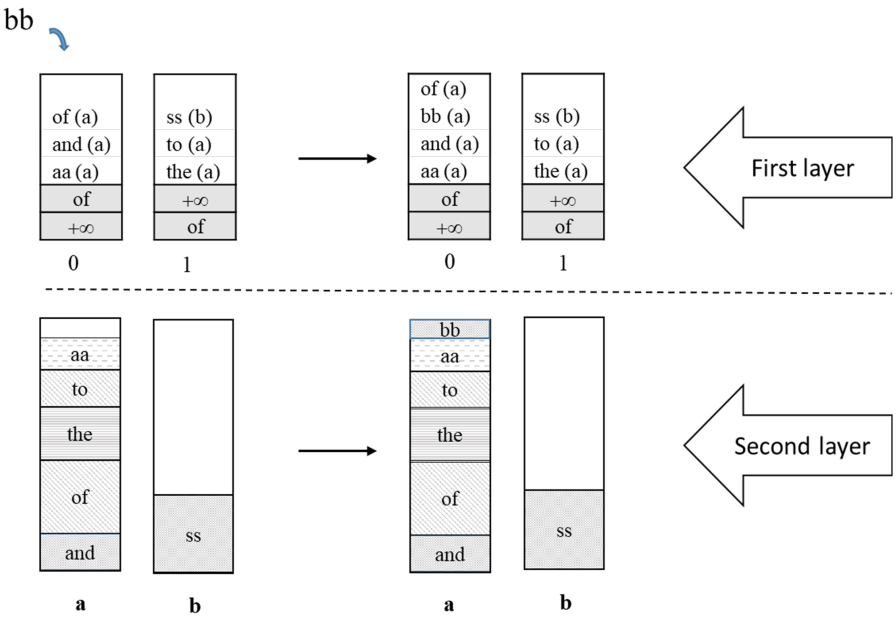

Figure 5.6: The 2^{nd} layer after a new insertion with no free space.

The system looks for a different server that is available. If necessary, it assigns a new server. The required data is stored on the new server.

5.3 ZK-RP* Architecture

The **ZK-RP*** architecture extends RP*-SD2DS, enabling it to handle **multi-dimensional data** through the use of a **Multi-Dimensional Index (MDI)**. Built on a K-D tree foundation, the MDI efficiently processes complex queries, including nearest neighbor and range searches. It ensures balanced data distribution and fast query execution, making it ideal for large-scale multi-dimensional datasets. This architecture is particularly well-suited for applications requiring advanced spatial or multi-dimensional queries, such as location-based services.

Data is stored in files, with each file containing multiple buckets distributed across nodes. Each bucket holds $b \gg 1$ records. When a bucket overflows, it splits, creating a new bucket and adding a corresponding entry to the MDI. A node accommodates a specific number of buckets, and if it overflows, half of its buckets are relocated to a new node.

5.3.1 ZK-RP*'s Node Architecture

A node contains two main components. The first component is the Multi-Dimensional Index Layer (MDI). The second component is the Data Storage Index Layer (DSI).

A. **Multi-Dimensional Index Layer (MDI):** This layer contains the global image of how data is distributed in the system. It selects the subspaces in which a query could be processed. Section 4 provides more details on the MDI index.

B. **Data Storage Index Layer (DSI):** This layer manages buckets by their IDs. It stores bucket details, such as capacities, intervals, and current split dimensions.

5.3.2 ZK-RP*'s Distributed Data Store

The system distributes data over nodes. Nodes follow a global addressing policy. This ensures load balancing. Each node has a global image. The global image

shows how data is distributed. Nodes respond to any client request. The system synchronizes changes to the data's addressing state. This happens during scaling up or shrinking. It uses a special message called an Image Adjustment Message (IAM). The synchronization occurs in real-time across all nodes.

5.3.3 ZK-RP* Framework

This network of interconnected nodes is known as the **ZK-RP* framework**. Each node has a large number of buckets. A node transfers half its buckets to another node when its capacity is reached. This ensures a balanced distribution of data. The master node is the initial node. It manages the administration and initialisation of the system. ZK-RP* clients communicate with the system via an API. The API enables programs to run complex multidimensional searches efficiently. See Figure 5.7.

A ZK-RP* client receives queries from applications. It transfers the queries to the correct node. It uses its own MDI-C for this. The MDI-C contains partial information about request execution. During a split operation, the MDI-C may become outdated. This causes addressing errors in future requests. The client receives an IAM to correct its MDI-C in such cases.

In our architecture, the splitting operation happens in two ways. These are inner splitting and outer splitting.

Our system starts with the first bucket. The first bucket has an ID equal to one. It is in the first node (node zero). When the first bucket reaches its capacity, it splits inside the node. It splits by the first dimension. In our case, this is the X dimension. The next split uses the Y dimension. This pattern continues. It splits using the modulo of the cardinality of the dimensions set. It stores half of its records in a new bucket. This process repeats until the node reaches the maximum number of buckets. Then, the node splits outward. It transfers half of its buckets to another node.

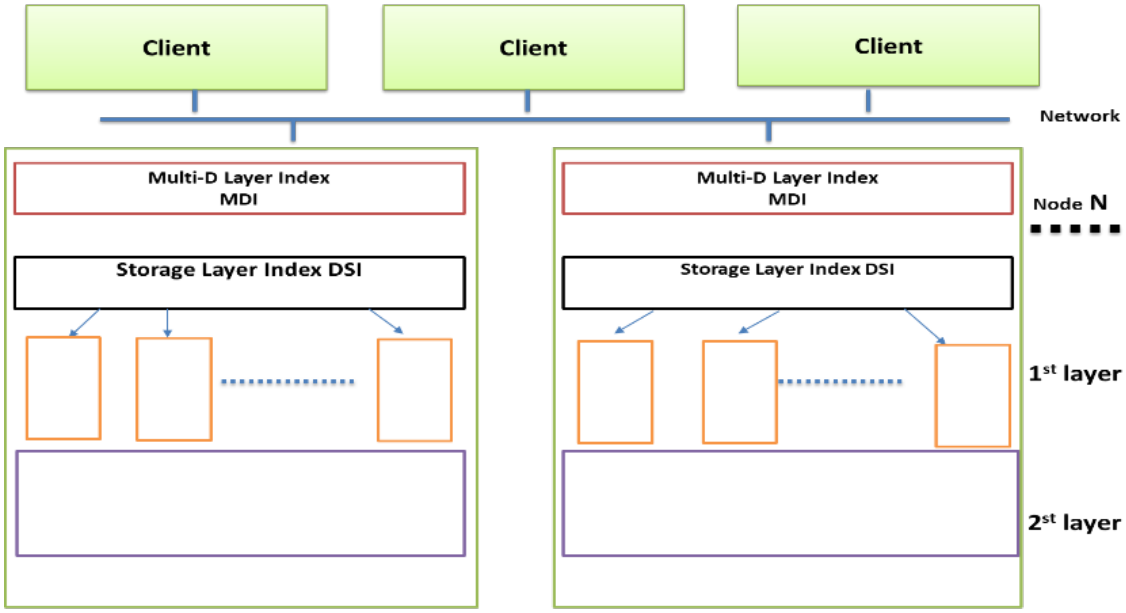

Figure 5.7: ZK-RP* General Architecture.

5.4 Multi-Dimensional Index Layer (MDI)

The ZK-RP*SD2DS architecture's multi-dimensional indexing (MDI) layer is an essential component. The K-D tree approach is used to split the space. This guarantees effective searching and indexing of multi-dimensional data. This approach ensures that multi-dimensional data is efficiently indexed and queried. It divides a space using the dimension D_{split}. As the division point, the median value is employed. The points in the space are categorized by the system using the dimension D_{split}. It is necessary to use the division point to guarantee that the points in the two subspaces are equal. The data will be distributed evenly thanks to this method. It also makes query processing more effective.

This approach assumes deletions are rare in LBSs. It assumes the distribution of location data stays consistent over time. The data concentrates around cities and roads. These assumptions justify the choice of this approach.

Conceptual sub-spaces are created by the MDI. These buckets are mapped to these subspaces. A predetermined maximum number of buckets is present in each node. These are known as *max buckets per node* (N). Half of the buckets are moved to a new node ($N/2$) when a node reaches the maximum number of buckets. There must be an even number for *max buckets per node*. In this way, the data is

distributed evenly between the nodes. The conceptual subspace of the MDI layer and the bucket of the data storage layer are mapped to each other.

This approach enables efficient processing of multi-dimensional queries. It creates size-balanced buckets. It ensures even data distribution across nodes. As a result, queries have almost the same response time over the scanned buckets. It allows a node to manage several buckets. This increases the number of buckets when necessary. It creates perfectly sized buckets for the scanning operation.

The MDI index structure follows the K-D tree scheme. Figure 5.8 shows the first part of the index. It represents the K-D tree. The second part includes a table. This table matches each bucket's ID with the corresponding node ID.

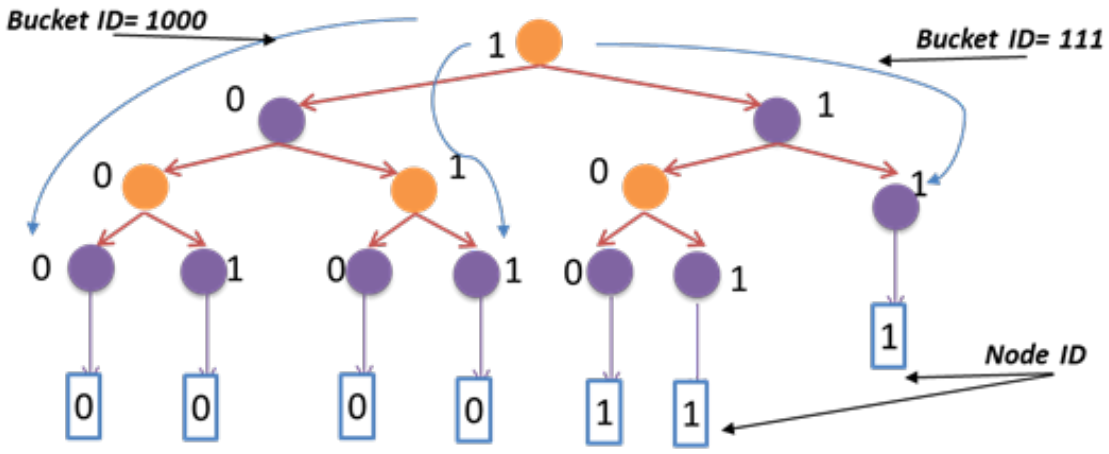

Figure 5.8: Multi-Dimensional Index Structure.

A bucket's ID shows the path in the K-D tree. The first bucket in the system has the ID '1'. When a bucket splits, two new buckets are created. The system gets the new IDs by shifting the original ID to the left in binary. It adds '0' for the lower bucket and '1' for the upper bucket.

5.4.1 Subspace Lookup and Point Queries

The subspace lookup algorithm finds the subspace for a point. It helps with efficient point queries. It also helps with data insertion. This is very important for multi-dimensional data. The K-D tree index makes sure queries are fast. It also makes sure queries are accurate.

Algorithm 8 Subspace Lookup

Input:

`q:` Query point.

`RN:` Root node of the K-D tree.

`1:` Starting bucket ID.

Ensure:

`BktId_q:` Bucket ID for the query point.

`NodeId_q:` Node ID for the bucket.

procedure FINDBUCKET(q, RN, 1)

```
    BktId_q ← findBucketId(RN, q, 1)
    NodeId_q ← getNode(BktId_q)
    if NodeId_q is localNode then
        Z_q ← calculateZvalue(q)
        return ScanBucket(BktId_q, Z_q)
    else
        return RequestScanBucket(NodeId_q, BktId_q, q)
    end if
```

end procedure

5.4.2 Insertion

The algorithm inserts a new data point. It works like the point query algorithm. It first finds the bucket for the subspace of the point. It then adds the data point to the bucket. Each bucket has a maximum limit for points. The algorithm checks the bucket size. It decides if a division is needed. It also checks if the node has the maximum number of buckets. If yes, it moves half of the buckets ($n/2$) to a new node.

5.4.3 Range Query

A multi-dimensional range query is common. It is also important for location-based applications. Algorithm 10 shows the steps for query processing.

Algorithm 9 Insert a New Location Data Point

Input:

`p:` The new data point.

`RN:` Root node of the K-D tree.

`1:` Starting bucket ID.

Ensure:

`BktId_p:` Bucket ID for the new data point.

`NodeId_p:` Node ID for the bucket.

procedure InsertLocation(p, RN, 1)

```
    BktId_p ← findBucketId(RN, p, 1)
    NodeId_p ← getNode(BktId_p)
    if NodeId_p is localNode then
        Z_p ← calculateZvalue(p)
        InsertIntoBucket(BktId_p, Z_p, p)
        if MaxBktSize is reached then
            SplitBucket(BktId_p)
            if MaxBktPerNode is reached then
                SplitNodeBuckets()
            end if
        end if
    else
        RequestInsertIntoBucket(NodeId_p, BktId_p, p)
    end if
```

end procedure

5.4.4 Nearest Neighbor Query

Nearest neighbor queries are important. They are a basic operation for many location-based applications. Each subspace has at least k points. This makes sense because the buckets are nearly the same size.

The algorithm works in two steps. First, it finds k nearest neighbors in the subspace of the point. Second, it improves the result by checking other possible subspaces, if they exist. Figure 5.9 shows an example. It finds the three nearest

Algorithm 10 Range Query

1: **Input:**
2: `(q_l, q_h)`: Range for the query.
3: `RN`: Root node of the K-D tree.
4: `1`: Starting bucket ID.
Ensure:
5: `R`: Resulting data points.
6: **procedure** RangeQuery(q_l, q_h, RN, 1)
7: `S` ← `findBuckets(RN, q_l, q_h, 1)` ▷ Identify candidate buckets
8: **for all** `BktId` in `S` **do**
9: `NodeId` ← `getNode(BktId)` ▷ Fetch the node for the bucket
10: **if** `NodeId` is localNode **then**
11: `Z_low` ← `calculateZvalue(q_l)`
12: `Z_high` ← `calculateZvalue(q_h)`
13: `R` ← `R` ∪ `ScanBucket(BktId, Z_low, Z_high)`
14: **else**
15: `R` ← `R` ∪ `RequestScanBucket(NodeId, BktId, q_l, q_h)`
16: **end if**
17: **end for**
18: **return** `R` ▷ Return the resulting data points
19: **end procedure**

neighbors of the red point q.

First, the algorithm finds the bucket for the point q. Here, the bucket is bucket 12. Next, it looks for the three nearest points in the same bucket. These points are a, b, and c. Next, the algorithm looks for better points in other subspaces. It starts with the nearest point, which is a. It creates an interval based on the distance between the point a and the point q (the black square). It then checks the subspaces of this interval. In this case, the interval lies entirely within bucket 12, which has already been scanned. The algorithm moves on to the next nearest point, which is b.

The algorithm repeats the same steps for the point b. The red square indi-

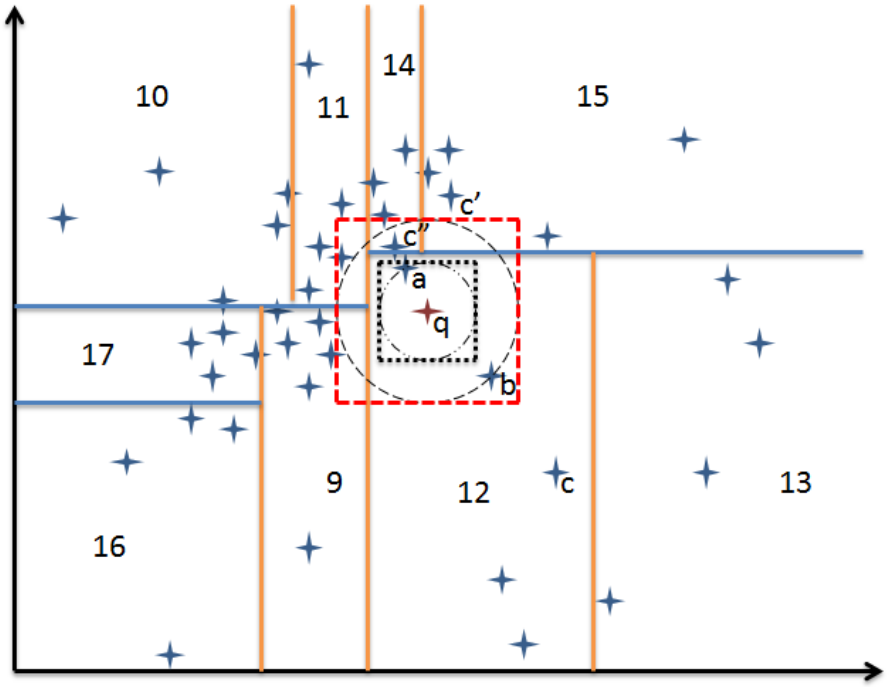

Figure 5.9: K-Nearest Neighbors Query.

cates the defined interval. The interval includes buckets 9, 11, 12, 14 and 15. The algorithm analyses well 15, replacing the point with the point. The best result is now a, b, c' . The position of the point b in the result remains the same (2). The algorithm continues to analyse the next bucket.

The algorithm analyzes bucket 14. The best result changes to a, c'', and b. This time, the position of the point b changes. It may even be removed. The algorithm stops analyzing the other buckets. It starts again from the nearest point in the result. It follows the same steps but skips the points already checked. The algorithm finds the final result when all the points have been checked. No new subspace is likely to have better points.

Algorithm 11 shows the steps. It processes k-nearest neighbors queries in ZK-RP*.

5.5 Storage Data Index Layer (SDI)

The Storage Data Index Layer (SDI) manages the physical storage of data in the ZK-RP*SD2DS system. Figure 5.10 shows the SDI. It contains the node's details. These

Algorithm 11 k-Nearest Neighbor Query

```
1: Input:
2: q: Point for the query.
3: RN: Root node of the K-D tree.
4: 1: Starting bucket ID.
5: R: Resulting data points.
6: S: Candidate buckets.
7: procedure kNearestNeighbor(q, RN, 1, k)
8:     R ← ∅                                          ▷ Initialize result set
9:     S_scanned ← ∅                                  ▷ Initialize scanned subspaces
10:    BktId_q ← findBucketId(RN, q, 1)
11:    NodeId ← getNode(BktId_q)
12:    R ← RequestKNearest(NodeId, BktId_q, q, R, k)
13:    S_scanned ← S_scanned ∪{BktId_q}
14:    EndLoop ← True
15:    repeat
16:        for all Point p in R starting from the closest do
17:            B_low ← (p.x - Distance(q,p), p.y - Distance(q,p))
18:            B_high ← (p.x + Distance(q,p), p.y + Distance(q,p))
19:            S ← findBuckets(RN, B_low, B_high, 1) − S_scanned
20:            while Position(p, R) is the same and S is not empty do
21:                BktId ← Dequeue(S)
22:                NodeId ← getNode(BktId)
23:                R ← RequestKNearest(NodeId, BktId, q, R, k)
24:                S_scanned ← S_scanned ∪{BktId}
25:                if S is not empty then
26:                    EndLoop ← False
27:                end if
28:            end while
29:            PointChecked(p)
30:        end for
31:    until EndLoop
32:    return R                                       ▷ Return the resulting data points
33: end procedure
```

details include the node's interval and the node's maximum number of buckets. It also includes the buckets' header information. Each header provides information about its bucket. This information includes the bucket ID, range, current bucket size (number of records), and index size.

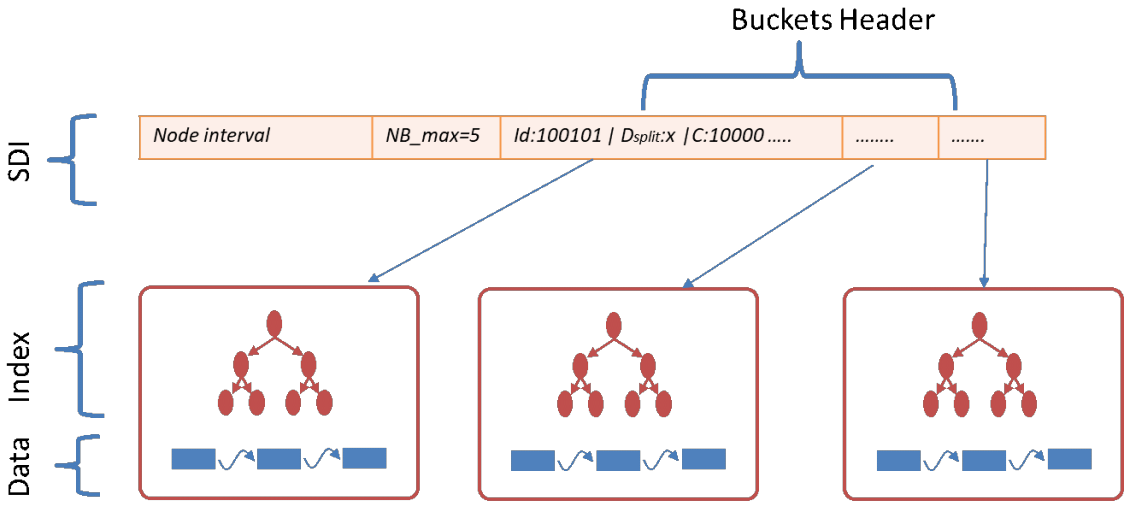

Figure 5.10: The bucket component.

The internal organization of a bucket has two areas: index and data. The index is a variant of a B+Tree. It has three levels and no leaves. It is hierarchical. It consists of nodes connected by pointers. The root is at the first level. It acts as an index for the next level. The second level acts as an index for the third level. The third level refers to lists of records. These records act as leaves. The leaves are linked together using pointers.

Each index node stores N index items. These items are key-pointer pairs. Each pointer points to the node index at the next level. A node must contain at least $N/2$ items. The root is an exception. It can store at least one element.

The next splitting must use the following dimension. After splitting with the last dimension, it starts again from the first dimension. Equation (5.1) represents this.

$$D_{\text{split}} \leftarrow D_{\text{next}} \mod (N_d) \tag{5.1}$$

Where:

- D_{split}: Represents the splitting dimension.
- D_{next}: The next dimension.
- N_d: Expresses the total number of dimensions.

The bucket records are sorted by their *z_value* (Morton value).

5.6 ZK-RP* Client

The system communicates with the ZK-RP* client. It makes use of an MDI-C, or multi-dimensional index client. A partial picture of the system is maintained by the MDI-C. Only the node interval is known to it. It contains references to the appropriate server node. This is seen in Figure 5.11. The MDI-C facilitates the client's effective data retrieval and location. Even in a system that is evolving dynamically, this is effective.

The information could occasionally be out of date. An Image Adjustment Message (IAM) is sent to the client in these situations. Together with the requested data, it obtains this. Each bucket ID is stored by server nodes along with node information. Range inquiries and k-Nearest Neighbor (KNN) processes make advantage of these specifics.

The splitting of each server node happens in two stages. First, it performs an inner split. Then, it performs an outer split. Inside each bucket, the data are linearized. They are stored using their z-value (Morton code).

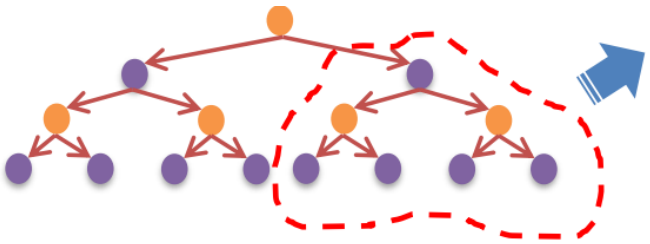

Figure 5.11: ZK-RP* client index.

5.7 Conclusion

In this chapter, we presented **KV-MDSS**, a robust and scalable key-value store system designed to handle both **one-dimensional** and **multi-dimensional data** efficiently. The system is composed of two main architectures: **RP*-SD2DS** and **ZK-RP*SD2DS**. The **RP*-SD2DS** architecture is optimized for one-dimensional data, leveraging a two-layer approach with a file layer for managing data keys and locators and a storage layer for storing complete data. Its use of a B+ tree index ensures efficient data retrieval and scalability, while the dynamic splitting mechanism minimizes system downtime during data redistribution. On the other hand, **ZK-RP*SD2DS** extends this architecture to handle multi-dimensional data by introducing a **Multi-Dimensional Index (MDI)** based on a K-D tree. This enables efficient processing of complex queries such as range queries and nearest neighbor searches, ensuring balanced data distribution and fast query performance.

Both architectures operate without a master or coordinator, ensuring high availability and fault tolerance. The two-layer approach in RP*-SD2DS and the multi-dimensional index in ZK-RP*SD2DS significantly reduce system downtime during splitting operations, addressing a common bottleneck in traditional NoSQL systems. Experimental results demonstrate that **KV-MDSS** outperforms MongoDB in terms of query performance and scalability, whether handling single-dimensional or multi-dimensional data. In summary, **KV-MDSS** provides a comprehensive solution for scalable and efficient data storage, making it a promising candidate for large-scale data management in multi-computer environments. Its ability to handle both one-dimensional and multi-dimensional data makes it particularly suitable for modern data-intensive applications, such as real-time analytics and location-based services.

The next chapter discusses the measures and performance of the realized parts of the global system.

Chapter 6

Measures and Performance

6.1 Introduction

This chapter explores critical performance metrics for Multi-Dimensional NoSQL based On SDDS (MD-NOS) , focusing on **scalability**, **split operation efficiency**, and **large files handling**. These metrics are vital for maintaining robust and adaptive distributed data systems. Scalability ensures consistent performance under increasing workloads, while efficient split operations enable dynamic resource allocation and storage management. The ability to handle large files ensures seamless data storage and retrieval without compromising system reliability.

We analyze the architecture of **RP*-SD2DS**, which addresses these challenges with innovative solutions. **RP*-SD2DS** employs a two-layer architecture with range partitioning, eliminating the need for a coordinator and minimizing system downtime during splits. **ZK-RP*** enhances performance through optimized range queries and pre-ordered data access, ensuring efficient data management in dynamic environments.

This chapter focuses only on the RP*-SD2DS performance, which is already published.

6.2 RP*-SD2DS Implementation Results

To demonstrate the efficiency of our system, we conducted simulations using large files of varying sizes (1 MiB, 2 MiB, 5 MiB, and 10 MiB) and a range of client configurations (1, 2, 4, 8, 16, and 32 clients). The system was deployed in a cluster comprising 4 client PCs (Dell i5, 8 GiB RAM) and multiple server nodes (i5 processors, 16 GiB RAM). Our approach was benchmarked against **MongoDB**, using configurations with 1 and 3 Mongos instances.

To validate the superiority of our datastore system, all simulations were performed under identical configurations. The **split process**, a critical and resource-intensive operation in data storage systems, was individually simulated and analyzed. We specifically focused on **insertion queries (PUT operations)** that trigger the splitting process, allowing us to evaluate the system's behavior and performance during this key operation.

6.2.1 Split Process Results

In this evaluation, we used a cluster setup. The cluster included four clients and three servers. The results appear in Figure 6.1. The figure shows the stability of the splitting process with our method. Even with data being inserted at the same time, the splitting time stays steady. This means the performance stays consistent no matter how much data is stored.

We tested the splitting time with 512 records with a bucket size of 256 records (with this configuration, the system, at least, splits two times). The records had different file sizes (1 MiB, 2 MiB, 5 MiB, and 10 MiB). We also tested under different workloads, represented by the number of clients (1, 8, and 32). The splitting time in the simulations ranged between 133 ms and 217 ms. This shows the splitting mechanism is robust.

The dataset is split in two by the split process. One part is moved to a different server. The network design and data volume determine how long the split takes. The volume of data and the available throughput are used in this time prediction.

The storage capacity divides into two sections in order to shorten the split time. The data key and a pointer to the second component are stored in the first section. The complete data records are kept in the second section. The volume of data transferred is decreased by this splitting. Consistent splitting time is guaranteed.

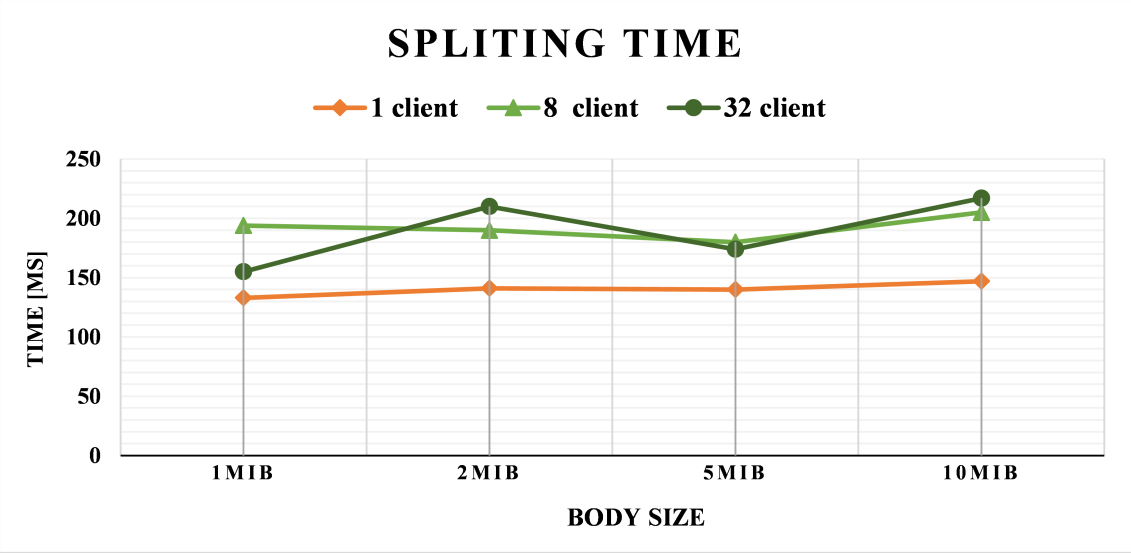

Figure 6.1: The average splitting time for 512 record insertions.

6.1 and 6.2 are tables that display the outcomes of adding 512 records. The table 6.1 displays the average split time. The average split time is absent from the table 6.2. In a distributed system, nodes split when they reach their limit. The 512th entry overloads the nodes in this example. Consequently, there was a split. After waiting for the insertion and split times, the 513th record was added.

For instance, if the split time was 133 ms and the insertion time was 44.75 ms, the 513th record's total insertion time was 177.75 ms. This relationship makes use of the equation 6.1.

$$T_i = \frac{\sum_{i=1}^{R} t_i}{R} \tag{6.1}$$

Where:

- T_i: Average insertion time per record.
- t_i: Insertion time for a single record.
- R: Total number of records inserted.

The results show how splitting times affect the overall insertion latency. They also emphasize the need to optimize the split process. This optimization helps maintain system responsiveness during high-insertion workloads.

Table 6.1: Insertion Time for Different Numbers of Clients with Split Time (in ms)

Clients	**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

The insertion times rise with larger body sizes. The insertion times also rise with a greater number of clients. This table format shows the relationship between client workload and body size. It highlights the scalability and performance challenges caused by increasing operational intensity.

6.2.1.1 Impact of Split Time on Insertion Performance

The split time is spread over 512 insertions. The extra time needed to insert a single record is the sum of the insertion time and the split time. The equation 6.2 describes this. In this case, the system adds about 0.25 milliseconds of delay per recording. This calculation divides 133 milliseconds (splitting time) by 512 insertions.

$$S_i(R, S) = \frac{\sum_{i=1}^{R} t_i - (t_s \cdot S)}{R} \tag{6.2}$$

Where:

- t_s: Split time per record.
- S: Number of split operations.
- R: Total number of records.

Table 6.2: Insertion Time for Different Body Sizes Without Split Time (in ms)

Clients	**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

- t_i: Insertion time per record.

Equations 6.1 and 6.2 describe the insertion time for a set of records (R). Equation 6.1 does not include the splitting time. Equation 6.2 includes the splitting time. The equation 6.3 calculates the difference between these times. This difference is denoted by D_{ts}. The table 6.4 shows that D_{ts} remains practically the same in all scenarios.

$$D_{ts} = T_i(R) - S_i(R) \tag{6.3}$$

Where:

- D_{ts}: Difference in insertion time with and without split time consideration.
- $T_i(R)$: Insertion time with split time consideration.
- $S_i(R)$: Insertion time without split time consideration.

This analysis shows the predictable effect of split time on overall insertion performance. The D_{ts} value stays consistent across different file sizes. It also stays consistent across different client workloads. This consistency proves the stability and efficiency of our approach.

6.2.2 Scalability performance

Our system demonstrates scalability in terms of handling a large number of insertions from multiple clients. Additionally, when a server node reaches its capacity, the system seamlessly splits to another server node. If an insertion request arrives during a split operation, the system incurs only a few milliseconds of additional delay to complete the insertion. This is a significant advantage over traditional systems, which typically require the system to halt operations until the split process is complete, resulting in prolonged downtime.

Figure 6.2 illustrates the insertion of 512 records from a view of one client with a file size of 2 MiB. The graph depicts the response times for insertion requests, with specific points highlighting insertions that occurred during a split operation.

The first server (server 0) undergoes three split operations. The first split occurs at the insertion of the 256th record, where half of the records are transferred to a new server (server 1). The second split takes place after inserting an additional 128 records, which is observed at the 384th record. Finally, the third split is triggered by the insertion of the 512th record, marking the completion of the insertion process. During these operations, all insertions are intentionally directed to the first server to demonstrate the splitting mechanism.

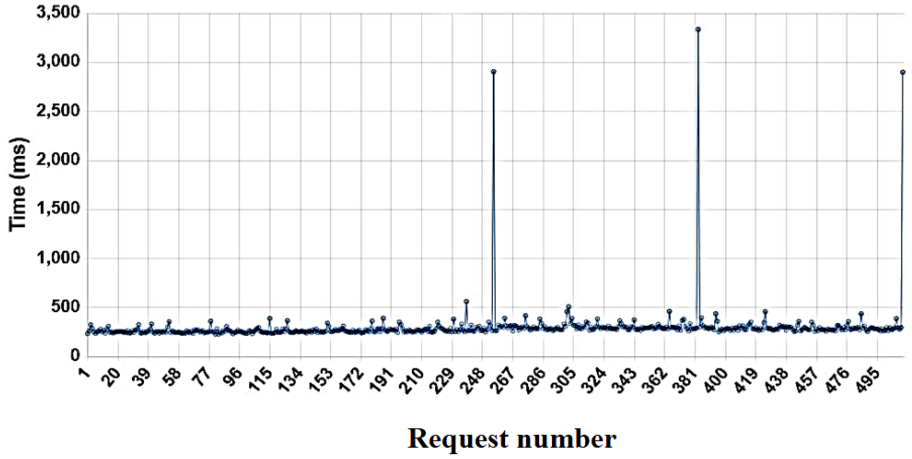

Figure 6.2: 512 records insertions with 2 MiB file size for one client.

To evaluate the system's performance, we configured clients to insert multiple records, each of size 2 MiB, targeting the same server (i.e., all insertions were directed to server 0). We created specific scenarios for each client, defining keys that would

trigger these scenarios to measure the waiting time and observe the split process.

As shown in Figure 6.2, all insertion operations completed in less than 500 ms. When an insertion coincides with a split process, it waits for the split to finish before being processed. This ensures minimal disruption and maintains system efficiency.

6.2.3 Availability performance

The RP*-SD2DS architecture improves data storage systems. It is better than the RP*-SDDS. Both systems use range partitioning to increase scalability and efficiency. However, RP*-SD2DS adds a two-layer structure. This structure improves data availability. RP*-SDDS uses a single layer for data management. RP*-SD2DS divides storage into two layers: a file layer and a storage layer.

The file layer manages data keys and has locators. These locators point to the actual data locations (meta data). The storage layer keeps the full data (the body). This separation makes data management and retrieval more efficient. The file layer can quickly find and access data. It does not store large data volumes. RP*-SD2DS also removes the need for a central coordinator or master node.

RP*-SD2DS distributes data and storage management across two layers. This ensures higher availability, especially during splitting operations. This feature is because of the simultanious insertion of the body in the second layer during the split process. When a server reaches its capacity and needs to split, the system continues processing requests. It handles data without major interruptions. In RP*-SDDS, such operations could cause temporary unavailability. RP*-SD2DS also integrates the RP* access technique into the SD2DS structure. This optimizes the system for efficient data handling. It allows dynamic scalability without the overhead of coordinator-based systems.

As a result, RP*-SD2DS offers better performance and reliability. It is a stronger solution for managing large and complex datasets in distributed environments.

6.2.4 Supporting large files

Table 6.3 summarizes the main differences between the LH*-SD2DS, MongoDB, and our system design (RP*-SD2DS).

Table 6.3: A comparison between LH*-SD2DS, MongoDB, and RP*-SD2DS

Approaches	Need special nodes	Max blob size	scale out (splitting)	Access method
LH*-SD2DS	a splitting coordinator	not-fixed	auto	linear hashing
MongoDB	router (mongos)	fixed	manual	hash/range
RP*-SD2DS (proposed)	N/A	not-fixed	auto	range

Table 6.4 shows the differences in insertion times for different body sizes. The body sizes are 1 MiB, 2 MiB, 5 MiB, and 10 MiB. The table also includes different client workloads. The client workloads are 1, 2, 4, 8, 16, and 32 clients. The values highlight the small impact of split time on insertion performance. The values demonstrate the efficiency of our system.

Table 6.4: Difference in Insertion Times with and Without Split Time (in ms)

Clients	**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

Our approach works well for large-file storage systems. Our method performs better than conventional approaches. The minimal insertion time differences prove this even under different workloads with different file sizes. A key advantage of our method is the temporary halt in the split process. This halt happens only during the insertion of the first layer in the split node. The second layer continues to work

without interruption.

We measured the average time for inserting 512 records to validate the efficiency of our system. Each record contains a 1 MiB Blob. Figure 6.3 shows these results. The figure compares our RP*-SD2DS results to MongoDB (configured with 1 Mongos and 3 Mongos). The results show that our system achieves the minimum insertion times with better performance than MongoDB in the two different configurations.

This analysis shows the effectiveness of our approach that maintains consistent performance and minimizes downtime during split operations. These qualities make our approach highly suitable for large-scale storage systems.

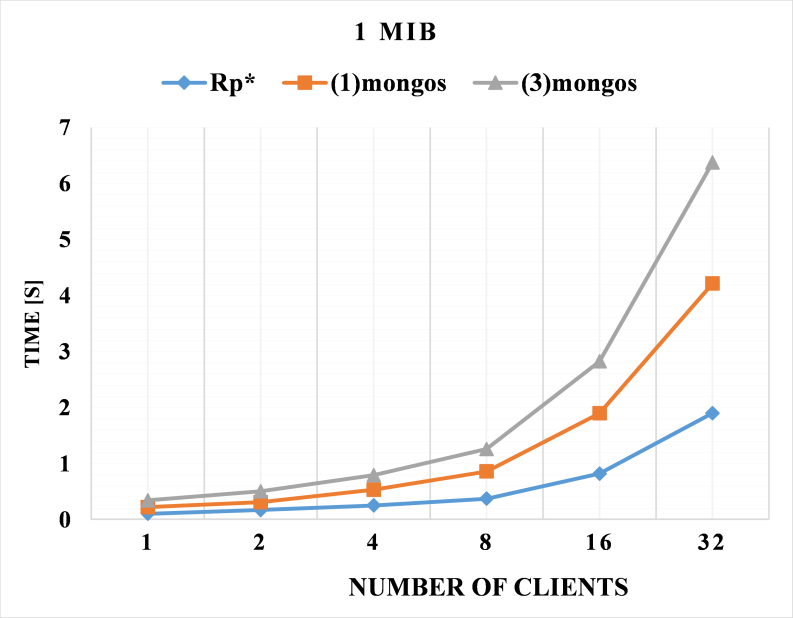

Figure 6.3: 512 records insertions with 1 MiB file size.

Figures 6.4, 6.5, and 6.6 use different large-file sizes: 1 MiB, 2 MiB, 5 MiB, and 10 MiB. They show that our system manages large file insertions successfully in a significantly shorter time compared to MongoDB configured by either with one Mongos or with three Mongos.

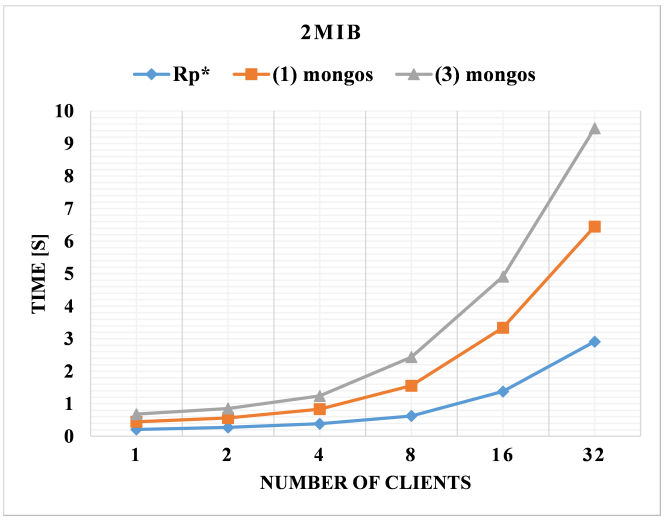

Figure 6.4: 512 records insertions with 2 MiB files.

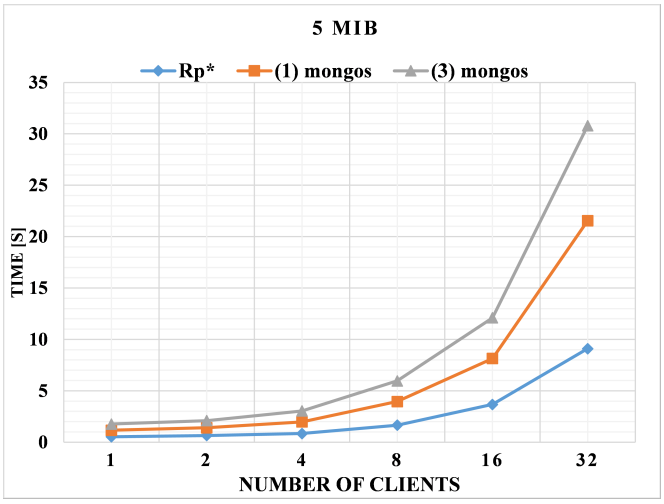

Figure 6.5: 512 records insertions with 5 MiB files.

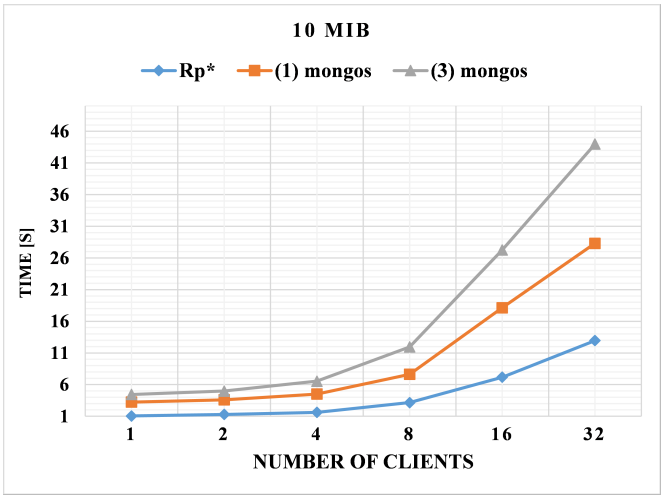

Figure 6.6: 512 records insertions with 10 MiB files.

6.2.4.1 Results Analysis and Discussion

The simulation results showed the high scalability and stability of the system's split process. The splitting process kept a constant time range of 133 ms to 217 ms. This highlights the reliability of the system which manages simultaneous data insertion and splitting operations well. The split operation transferred data to another server. It divided the dataset in half. This process depended on the amount of data to transfer. It also depended on the available network throughput. The simulation mainly focused on data volume. It revealed a better splitting time with consistent splitting time across all studied cases.

Two sets of tables were generated. The tables analyzed insertion times for 512 records. One set included average splitting times. The other set did not include average splitting times. The results validated the system's efficiency. The system was compared to MongoDB configured with 1 and 3 Mongos and performed better in record insertion times for Blobs of different sizes. It showed remarkable scalability. It handled a large number of insertions from multiple clients effectively. It distributed data to another server node seamlessly when a server node reached its capacity.

6.3 RP*-SD2DS vs RP*-SDDS

In distributed systems, the number of messages exchanged during operations is a critical performance metric. These operations include insertion, splitting, and querying. The number of messages directly affects the system's scalability. It also affects the system's latency. Additionally, it affects the system's network overhead. In this section, we analyze the **message complexity** of insertion operations in **RP*-SD2DS**. We compare it with **RP*-SDDS**.

6.3.1 Performance Metrics

To comprehensively evaluate the performance of **RP*-SD2DS** and **RP*-SDDS**, we introduce the following key performance metrics: **Throughput**, **Latency**, and **Network Overhead**.

Table 6.5: Performance: RP*-SD2DS vs. RP*-SDDS

Metric	**RP*-SD2DS**	**RP*-SDDS**
Data Transfer During Splits	Metadata only	Metadata + Data
Network Overhead	Low	High
Latency	Low (fast splits)	High (slow data transfer)
Throughput	High (T_{total} reduced)	Low (T_{total} increased)

Throughput refers to the number of insertion operations completed per unit time, where higher throughput indicates better performance. **RP*-SD2DS** demonstrates significantly higher throughput compared to **RP*-SDDS**, primarily due to the reduced data transfer during split operations. By transferring only metadata during splits, **RP*-SD2DS** can handle a larger number of insertion operations within the same time frame, maintaining high throughput even as the number of clients and file sizes increase. In contrast, **RP*-SDDS** experiences lower throughput because it transfers both metadata and data during splits, which increases the time required for each operation.

Latency measures the time taken to complete an insertion operation, including any split operations that may occur, with lower latency indicating better performance. **RP*-SD2DS** exhibits lower latency, especially for large files, due to its efficient handling of metadata during splits. By minimizing the amount of data transferred, **RP*-SD2DS** ensures faster completion of insertion operations. On the other hand, **RP*-SDDS** suffers from higher latency, particularly in scenarios with large files and high client counts, as it must transfer both metadata and data during splits, resulting in longer processing times.

Network Overhead refers to the total amount of data transferred during insertion and split operations, where lower network overhead indicates better performance. **RP*-SD2DS** significantly reduces network overhead by transferring only metadata during splits, whereas **RP*-SDDS** transfers both metadata and data. This reduction in network overhead not only improves the overall performance of **RP*-SD2DS** but also reduces the load on the network infrastructure, making it more scalable and efficient in distributed environments. In contrast, the higher net-

work overhead in **RP*-SDDS** limits its scalability and efficiency, particularly in high-throughput scenarios.

For **RP*-SD2DS**, the total time T_{total} is primarily determined by the time required for metadata transfer during splits. This efficient handling of metadata ensures that the system can maintain high throughput and low latency, even under heavy load. In contrast, for **RP*-SDDS**, the total time T_{total} is dominated by the time required for both metadata and data transfer during splits. This additional data transfer significantly increases the overall latency and reduces the system's throughput, particularly in scenarios with large files and high client counts.

In summary, **RP*-SD2DS** outperforms **RP*-SDDS** across all key performance metrics, demonstrating higher throughput, lower latency, and reduced network overhead. These advantages make **RP*-SD2DS** a more efficient and scalable solution for distributed data storage systems, particularly in environments where large files and high client counts are common.

6.4 Conclusion

this chapter, we proposed a large-file data store based on **RP*-SD2DS**, a distributed and scalable architecture. Unlike SD2DS, which is based on LH*, our proposed approach efficiently handles range queries. It leverages the RP access method and SD2DS structure to manage large files effectively.

The system architecture is composed of two layers: the first layer stores the keys and locators pointing to the data in the second layer, and it is indexed using a structure similar to a B+ tree. This design enhances availability during the split process and manages the workload dynamically, allowing the system to scale without requiring a coordinator or master node.

Compared to MongoDB, our **RP*-SD2DS** demonstrates superior efficiency in handling record insertions for varying file sizes (1 MiB, 2 MiB, 5 MiB, and 10 MiB) across different numbers of clients (1, 2, 4, 8, 16, and 32). These results highlight the robustness and scalability of the proposed architecture, making it a reliable solution for managing large-file storage in distributed environments.

Chapter 7

General Conclusion

The MD-NOS is a significant advancement in today's data management. It combines IoT, Fog, and cloud layers into a single system. The system uses a hierarchical model for modern data environment management. It employs dynamic partitioning methods and intelligent query routing algorithms. RP*-SD2DS and ZK-RP* architectures demonstrate the flexibility and potential of using SDDS as a foundation for scalable and efficient data storage systems.

Experimental tests have confirmed these claims. They show that MD-NOS outperforms conventional NoSQL databases, especially in performance and scalability. It is an ideal choice for data-intensive applications that require strong and stable data management. The system also serves as a solid foundation for real-time analytics, IoT-based applications, and big data processing.

In the short term, several key enhancements will strengthen the MD-NOS paradigm. These include security improvements with end-to-end data encryption for data at rest and in transit. This ensures data privacy and security in dispersed IoT and Fog environments.

The system will also integrate Distributed Data Processing systems like MapReduce and Apache Spark. This will enhance analytical capabilities for large in-memory data.

The system will also incorporate Spatial Joins and the Multi-Dimensional Index (MDI) Layer. This will improve geospatial indexing efficiency and handle

complex spatial queries. Other components in development include Query Optimization Techniques, Fault Recovery Mechanisms, and Load Balancing Strategies.

Performance testing will be conducted in real-world conditions. This will assess the system's fault tolerance, scalability, and query performance under varying workloads. The goal is to ensure all application-level demands.

In the long term, MD-NOS will involve creating energy-aware algorithms for data processing and storage. These algorithms will prioritize resource limitations in IoT and Fog nodes to ensure a sustainable and efficient system across different operating environments.

Interoperability is a major concern. MD-NOS is being developed to easily integrate with other data processing environments and database systems. This will be achieved through standardized APIs and data formats for easier adoption and integration into existing data management environments.

The architecture will continue to emphasize Scalability and Robustness. This will allow it to meet the evolving demands of data-intensive applications. This includes ongoing improvements to its architecture and performance to address future data management challenges.

Finally, the MD-NOS project will focus on Global Collaboration. This will foster innovation and accelerate the development cycle for next-generation data management technology. In partnership with other research institutions and industry stakeholders, the project will push the boundaries of what is possible in data management and drive innovative solutions for a more connected world.

Bibliography

[1] W. Litwin, "Trie hashing", in *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, 1981, pp. 19–29.

[2] W. Litwin, D. Zegour, and G. Lévy, "Multilevel trie hashing", in *Advances in Database Technology—EDBT'88: International Conference on Extending Database Technology Venice, Italy, March 14–18, 1988 Proceedings 1*, Springer, 1988, pp. 309–335.

[3] R. Devine, "Design and implementation of ddh: A distributed dynamic hashing algorithm", in *International Conference on Foundations of Data Organization and Algorithms*, Springer, 1993, pp. 101–114.

[4] W. Litwin, M.-A. Neimat, and D. A. Schneider, "Lh: Linear hashing for distributed files", in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, 1993, pp. 327–336.

[5] W. Litwin, M.-A. Neimat, and D. Schneider, "Rp*: A family of order preserving scalable distributed data structures", in *VLDB*, Citeseer, vol. 94, 1994, pp. 12–15.

[6] ——, "Rp*: A family of order preserving scalable distributed data structures", in *VLDB*, Citeseer, vol. 94, 1994, pp. 12–15.

[7] 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: 5th International Conference on Extending Database Technology Avignon, France, March 25–29, 1996 Proceedings 5*, Springer, 1996, pp. 573–591.

[8] W. Litwin and M.-A. Neimat, "High-availability lh* schemes with mirroring", in *Proceedings First IFCIS International Conference on Cooperative Information Systems*, IEEE, 1996, pp. 196–205.

[9] ——, "K-rp* s: A scalable distributed data structure for high-performance multi-attribute access", in *Fourth International Conference on Parallel and Distributed Information Systems*, IEEE, 1996, pp. 120–131.

[10] W. Litwin, M.-A. Neimat, and D. A. Schneider, "Lh*—a scalable, distributed data structure", *ACM Transactions on Database Systems (TODS)*, vol. 21, no. 4, pp. 480–525, 1996.

[11] ——, "Lh*: A scalable, distributed data structure", *ACM Transactions on Database Systems (TODS)*, vol. 21, no. 4, pp. 480–525, 1996.

[12] V. Corporation, "Versant: A high-performance object-oriented database", *Versant Corporation*, 1997.

[13] V. Hilford, F. B. Bastani, and B. Cukic, "Eh/sup*/-extendible hashing in a distributed environment", in *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, IEEE, 1997, pp. 217–222.

[14] W. Litwin, M.-A. Neimat, G. Lev, S. Ndiaye, and T. Seck, "Lh* s: A high-availability and high-security scalable distributed data structure", in *Proceedings Seventh International Workshop on Research Issues in Data Engineering. High Performance Database Management for Large-Scale Applications*, IEEE, 1997, pp. 141–150.

[15] ——, "Lh* s: A high-availability and high-security scalable distributed data structure", in *Proceedings Seventh International Workshop on Research Issues in Data Engineering. High Performance Database Management for Large-Scale Applications*, IEEE, 1997, pp. 141–150.

[16] Z. Corporation, "Zodb: A python object-oriented database", *Zope Corporation*, 2000.

[17] F. Bennour, "Performance of the sdds lh* lh under sdds-2000", *Distributed Data and*, 2002.

[18] R. Moussa and W. Litwin, "Experimental performance analysis of lh* rs parity management.", in *WDAS*, 2002, pp. 87–98.

[19] db4o, "Db4o: An object-oriented database for java and .net", *db4o*, 2004.

[20] W. Litwin, R. Moussa, and T. J. Schwarz, "Lh* rs: A highly available distributed data storage", in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, 2004, pp. 1289–1292.

[21] D. E. Zegour, "Scalable distributed compact trie hashing (cth*)", *Information and Software Technology*, vol. 46, no. 14, pp. 923–935, 2004.

[22] L. Chellouche and Z. Gharbi, "Gestion de transactions et reprise après pannes pour la sdds compact trie hashing (cth*)", *Mémoire d'ingénieur, Institut National d'Informatique–Alger*, 2006.

[23] A. Khetrapal and V. Ganesh, "Hbase and hypertable for large scale distributed storage systems", *Journal of Distributed Systems*, vol. 10, no. 3, pp. 137–150, 2006.

[24] W. Litwin, S. Sahri, and T. Schwarz, "An overview of a scalable distributed database system sd-sql server", in *British National Conference on Databases*, Springer, 2006, pp. 16–35.

[25] S.SAHRI, "Conception et implantation d'un système de bases de données distribuées & scalables : Sd-sql server", PhD thesis, Université Paris Dauphine, 2006.

[26] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Amazon dynamodb: A fast and scalable nosql database service", *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pp. 205–220, 2007.

[27] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed b-tree", *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 598–609, 2008.

[28] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Column-family stores: A survey", *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, 4:1–4:26, 2008.

[29] A. S. Foundation, "Hbase: A distributed, versioned, column-family store", *Apache Software Foundation*, 2008.

[30] P. Helland, "Soft state: The other cap", *Communications of the ACM*, vol. 51, no. 7, pp. 40–44, 2008.

[31] D. Pritchett, "Base: An acid alternative", *ACM Queue*, vol. 6, no. 3, pp. 48–55, 2008.

[32] J. C. Anderson, J. Lehnardt, and N. Slater, "Couchdb: The definitive guide", *O'Reilly Media, Inc.*, 2009.

[33] Greenplum, "Hypertable: A high-performance, scalable, distributed storage system", *Greenplum*, 2009.

[34] W. Vogels, "Eventual consistency: How to live with weak consistency", *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[35] R. Cattell, "Nosql databases for flexible schema design", *Communications of the ACM*, vol. 53, no. 6, pp. 48–55, 2010.

[36] K. Chodorow and M. Dirolf, "Mongodb: The definitive guide", *O'Reilly Media, Inc.*, 2010.

[37] F. Inc., "Allegrograph: A graph database for the semantic web", *Franz Inc.*, 2010.

[38] InfoGrid, "Infogrid: A web graph database", *InfoGrid*, 2010.

[39] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system", *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[40] I. Objectivity, "Infinitegraph: A distributed graph database", *Objectivity, Inc.*, 2010.

[41] RavenDB, "Ravendb: A transactional nosql database", *RavenDB*, 2010.

[42] D. Sullivan, "Key-value stores: A practical overview", *Communications of the ACM*, vol. 53, no. 10, pp. 12–14, 2010.

[43] B. Technologies, "Riak: A decentralized key-value store", *Basho Technologies*, 2010.

[44] O. Technologies, "Orientdb: A multi-model nosql database", *Orient Technologies*, 2010.

[45] N. Technology, "Neo4j: An open source graph database", *Neo Technology*, 2010.

[46] Terrastore, "Terrastore: A scalable and consistent document store", *Terrastore*, 2010.

[47] H. Boley, A. Damásio, and M. Kifer, "Hypergraphdb: A general-purpose, open-source graph database", *Proceedings of the 5th International Workshop on Graph-Based Technologies and Applications*, pp. 25–31, 2011.

[48] A. S. Foundation, "Accumulo: A bigtable-like nosql database", *Apache Software Foundation*, 2011.

[49] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database", in *2011 6th international conference on pervasive computing and applications*, IEEE, 2011, pp. 363–366.

[50] R. Hecht and S. Jablonski, "Nosql databases for real-time web applications", *Proceedings of the 2011 IEEE International Conference on Big Data*, pp. 1–6, 2011.

[51] ——, "Nosql databases: A new generation of database technologies", *Proceedings of the 2011 IEEE International Conference on Big Data*, pp. 1–6, 2011.

[52] N. Software, "Ravendb: A multi-model nosql database", *Northwoods Software*, 2011.

[53] Aerospike, "Aerospike: A high-performance nosql database", *Aerospike*, 2012.

[54] ArangoDB, "Arangodb: A multi-model database", *ArangoDB*, 2012.

[55] P. J. Sadalage and M. Fowler, "Nosql databases for big data processing", *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 1125–1128, 2012.

[56] M. ARIDJ, "Intégration des structures de données distribuées et scalables ordonnées dans les systèmes distribués.", PhD thesis, esi, 2013.

[57] P. Bailis and A. Ghodsi, "Eventual consistency revisited", *Communications of the ACM*, vol. 56, no. 9, pp. 82–89, 2013.

[58] J. L. Carlson, "Redis: The definitive guide", *O'Reilly Media, Inc.*, 2013.

[59] K. Chodorow, "Document stores: A survey", *O'Reilly Media, Inc.*, 2013.

[60] 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, 2013.

[61] ——, "Scalable distributed two-layer data structures (sd2ds)", *International Journal of Distributed Systems and Technologies (IJDST)*, vol. 4, no. 2, pp. 15–30, 2013.

[62] S. Gilbert and N. Lynch, "Availability in distributed systems", *ACM SIGACT News*, vol. 45, no. 2, pp. 53–76, 2014.

[63] R. Kumar, B. B. Parashar, S. Gupta, Y. Sharma, and N. Gupta, "Apache hadoop, nosql and newsql solutions of big data", *International Journal of Advance Foundation and Research in Science & Engineering (IJAFRSE)*, vol. 1, no. 6, pp. 28–36, 2014.

[64] M. Aridj, "Mr 2 p*: Un framework pour le traitement parallèle d'une très grande quantité de données", in *8ième édition du Séminaire National en Informatique (SNIB' 2015)*, Biskra, Algeria: SNIB' 2015, Jan. 2015.

[65] J. R. Lourenço, B. Cabral, P. Carreiro, M. Vieira, and J. Bernardino, "Choosing the right nosql database for the job: A quality attribute evaluation", *Journal of Big Data*, vol. 2, pp. 1–26, 2015.

[66] R. T. Mason, "Nosql databases and data modeling techniques for a document-oriented nosql database", in *Proceedings of informing science & IT education conference (InSITE)*, vol. 3, 2015, pp. 259–268.

[67] J. Piekos, "Sql vs, nosql vs. newsql: Finding the right solution", *Retrieved June*, vol. 11, p. 2018, 2015.

[68] D. Abadi, "Partition tolerance in distributed systems", *Communications of the ACM*, vol. 59, no. 9, pp. 30–33, 2016.

[69] D. Belayadi and W. Hidouci, "Dynamic range partitioning with asynchronous data balancing", in *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, IEEE, 2016, pp. 1214–1220.

[70] N. Dennouni, "Orchestration des activités d'apprentissage mobile", PhD thesis, Université Lille 1 and Université Djillali Liabès (Sidi Bel-Abbès, Algérie), 2016.

[71] D. Seybold, J. Domaschka, and S. Wesner, "Nosql databases for iot applications", *Proceedings of the 2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 337–342, 2016.

[72] A. Corbellini, C. Mateos, A. Zunino, D. Godoy, and S. Schiaffino, "Persisting big-data: The nosql landscape", *Information Systems*, vol. 63, pp. 1–23, 2017.

[73] A. Gupta, S. Tyagi, N. Panwar, S. Sachdeva, and U. Saxena, "Nosql databases: Critical analysis and comparison", in *2017 International conference on computing and communication technologies for smart nation (IC3TSN)*, IEEE, 2017, pp. 293–299.

[74] 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*, Springer, 2017, pp. 150–168.

[75] E. Płuciennik and K. Zgorzałek, "The multi-model databases–a review", in *Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation: 13th International Con-*

ference, BDAS 2017, Ustroń, Poland, May 30-June 2, 2017, Proceedings 13, Springer, 2017, pp. 141–152.

[76] C. Strauch, "Nosql databases: A survey and decision guidance", *Computer Science–Research and Development*, vol. 32, no. 3-4, pp. 353–365, 2017.

[77] A. Krechowicz and S. Deniziak, "Hierarchical clustering in scalable distributed two-layer datastore for big data as a service", in *2018 Sixth International Conference on Enterprise Systems (ES)*, IEEE, 2018, pp. 138–145.

[78] ——, "Hierarchical clustering in scalable distributed two-layer datastore for big data as a service", in *2018 Sixth International Conference on Enterprise Systems (ES)*, IEEE, 2018, pp. 138–145.

[79] K. Shahna and A. Mohamed, "An image encryption technique using logistic map and z-order curve", in *2018 International Conference on Emerging Trends and Innovations In Engineering And Technological Research (ICETI-ETR)*, IEEE, 2018, pp. 1–6.

[80] J.-K. Chen and W.-Z. Lee, "An introduction of nosql databases based on their categories and application industries", *algorithms*, vol. 12, no. 5, p. 106, 2019.

[81] Z. Fu, Z. Wu, H. Li, Y. Li, M. Wu, X. Chen, X. Ye, B. Yu, and X. Hu, "Geabase: A high-performance distributed graph database for industry-scale applications", *International Journal of High Performance Computing and Networking*, vol. 15, no. 1-2, pp. 12–21, 2019.

[82] J. Lu and I. Holubová, "Multi-model databases: A new journey to handle the variety of data", *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–38, 2019.

[83] B. Acharya, M. Pandey, and S. S. Rautaray, "Survey on nosql database classification: New era of databases for big data", *Journal of Big Data Analytics*, vol. 5, no. 2, pp. 123–145, 2020.

[84] E. A. Brewer, "Consistency in distributed databases", *ACM SIGACT News*, vol. 51, no. 1, pp. 58–65, 2020.

[85] K. A. ElDahshan, A. A. AlHabshy, and G. E Abutaleb, "A comparative study among the main categories of nosql databases", *Al-Azhar Bulletin of Science*, vol. 31, no. 2, pp. 51–60, 2020.

[86] T. N. Khasawneh, M. H. AL-Sahlee, and A. A. Safia, "Sql, nosql, and newsql databases: A comparative survey", in *2020 11th International Conference on Information and Communication Systems (ICICS)*, IEEE, 2020, pp. 013–021. DOI: 10.1109/ICICS49469.2020.239569.

[87] M. A. Kausar and M. Nasar, "Sql versus nosql databases to assess their appropriateness for big data application", *Recent Advances in Computer Science and Communications*, vol. 14, no. 4, pp. 1098–1108, 2021. DOI: 10.2174/2213275914666210615141935.

[88] A. Krechowicz, S. Deniziak, and G. Łukawski, "Highly scalable distributed architecture for nosql datastore supporting strong consistency", *IEEE Access*, vol. 9, pp. 69 027–69 043, 2021.

[89] ——, "Highly scalable distributed architecture for nosql datastore supporting strong consistency", *IEEE Access*, vol. 9, pp. 69 027–69 043, 2021.

[90] J. I. Lopez-Veyna, I. Castillo-Zuñiga, and M. Ortiz-Garcia, "A review of graph databases", in *International Conference on Software Process Improvement*, Springer, 2022, pp. 180–195.

[91] L. F. Da Silva and J. V. Lima, "An evaluation of relational and nosql distributed databases on a low-power cluster", *The Journal of Supercomputing*, vol. 79, no. 12, pp. 13 402–13 420, 2023.

[92] I. Dasoulas, D. Chaves-Fraga, D. Garijo, and A. Dimou, "Declarative rdf construction from in-memory data structures with rml", in *International Semantic Web Conference (ISWC)*, Springer, 2023, pp. 123–135.

[93] 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. 21 424–21 448, 2023.

[94] M. Kaufmann and A. Meier, "Nosql databases", in *SQL and NoSQL Databases: Modeling, Languages, Security and Architectures for Big Data Management*, Springer, 2023, pp. 223–244.

[95] M. Ahishakiye, "Online equivalence application management system: Case study of onecs (chad)", PhD thesis, University of Lay Adventists of Kigali (ULAK), 2024.

[96] Y. Demchenko, J. J. Cuadrado-Gallego, O. Chertov, and M. Aleksandrova, "Data structures for big data, modern big data sql and nosql databases", in *Big Data Infrastructure Technologies for Data Analytics: Scaling Data Science Applications for Continuous Growth*, Springer, 2024, pp. 237–275.

[97] S. A. Dyachkov, O. Adilbek, P. Y. Korotaev, V. Andrey, I. V. Morozov, A. Mikhail, and V. Y. Zitserman, "Database for properties of nuclear reactor materials based on the ontology and nosql data", *Journal of Nuclear Materials*, vol. 45, no. 3, pp. 456–470, 2024.

[98] M. Hulea, R. Miron, and A. Rusu, "Optimizing multi-modal transportation in smart cities: A graph-oriented database approach", in *2024 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, IEEE, 2024, pp. 1–4.

[99] R. Kesav Mohan, R. Rakshit Sukumar Kanmani, K. Anandan Ganesan, and N. Ramasubramanian, "Evaluating nosql databases for olap workloads: A benchmarking study of mongodb, redis, kudu and arangodb", *arXiv e-prints*, arXiv–2405, 2024.

[100] P. Khan, "Comparison of nosql databases for internet of things systems", in *2024 IEEE International Conference on Internet of Things (IoT)*, IEEE, 2024, pp. 1–6.

[101] D. Kpekpassi and D. Faye, "Preprint nosql databases: A survey", 2024.

[102] 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. 16 178–16 184, 2024.

[103] V. Santos and B. Cuconato, "Nosql graph databases: An overview", *arXiv preprint arXiv:2412.18143*, 2024.

[104] U. Serles and D. Fensel, "Databases", in *An Introduction to Knowledge Graphs*, Springer, 2024, pp. 69–73.

[105] H. Tu, "Cassandra vs. mongodb: A systematic review of two nosql data stores in their industry uses", in *2024 IEEE 7th International Conference on Big Data and Artificial Intelligence (BDAI)*, IEEE, 2024, pp. 81–86.

[106] A. S. Wälken, *Integrating a database into a real-time communication system: Optimizing performance and reliability in defense industry applications*, https://www.kth.se/en, 2024.

[107] D. Boukhelef and D.-E. Zegour, *Ih*: A new hash-based multidimensional sdds.*

[108] W. L. R. M. T. JE and S. Schwarz, "Lh* rs: A highly available distributed data storage", *Parity*, vol. 1, p. 0,