www.editada.org

International Journal of Combinatorial Optimization Problems and Informatics, 15(1), Jan-April 2024, 51-61. ISSN: 2007-1558.
https://doi.org/10.61467/2007.1558.2024.v15i1.435

An Empirical Hybrid Strategy for NoSQL Database Distribution

José Abiel Mendoza-Labastida[1], Mireya Clavel-Maqueda[1], Orlando García-Pérez[1] and Eduardo Cornejo-Velázquez[1]*

[1] Área Académica de Computación y Electrónica, Universidad Autónoma del Estado de Hidalgo, 42184, Pachuca, Hidalgo, México.
me453716@uaeh.edu.mx, mclavel@uaeh.edu.mx, orlando_garcia@uaeh.edu.mx, ecornejo@uaeh.edu.mx

Abstract. A case study of a pharmaceutical distribution company with a relational database that is migrated to a document-oriented NoSQL database for which a hybrid distribution strategy based on fragmentation and replication is implemented. The legacy relational database schema was thoroughly analyzed and evaluated to determine the collections and documents needed to move all the data and leverage the capabilities of the document-based database. In addition, a hybrid data distribution strategy was designed to effectively balance the performance, scalability, and fault tolerance of the system. Finally, integrated operational tests were performed on the distributed system nodes to verify that they function properly and ensure their availability for the company's operations.
Keywords: NoSQL, MongoDB, Replication, Sharding, Migration.

Article Info
Received Jan 30, 2024
Accepted April 11, 2024

1 Introduction

The role of a Pharmaceutical Distribution Company (PDC) is integral to the lives of small and medium-sized pharmacies and clinics. Given that pharmaceutical companies only provide large quantities of medicines whenever making a sale, it is logistically and financially impossible for these small and medium-sized entities to purchase and store these without incurring significant losses. Here, pharmaceutical distribution companies come into work; by purchasing these large quantities of medicines and then selling only a fraction of them to several small entities, they provide these establishments with the resources they need to operate without running the risk of overstocking. While this system supports the overstocking risk for pharmacies and clinics, it introduces data management challenges for PDCs. The challenges PDCs face in managing their data are multifaceted, ranging from handling large volumes of data to ensuring its integrity and availability. Additionally, PDCs often require real-time transaction processing capabilities to manage their inventory and sales effectively, emphasizing the importance of Online Transaction Processing (OLTP) systems.

For the last half-century, relational databases have been the principal choice for data storage and management (Silberschatz, Korth & Sudarshan, 2011). The rise of countless web services, coupled with the rising number of people with access to internet services has exponentially grown the amount of data being generated (Mukherjee, Ghatak, & Ray, 2021). This influx of data has put a massive strain on the databases that support these services. Furthermore, the complexity and speed with which this data is produced is increasing (Niu et al., 2021). However, traditional database systems often struggle to handle the large volume of data, leading to bottlenecks, data inconsistencies, and increased operational costs.

MongoDB is a NoSQL database that has outperformed a traditional SQL database in notable aspects, specifically when dealing with CRUD (Create, Read, Update, Delete) operations when dealing with Big Data (Bradshaw & Chodorow, 2019).

NoSQL databases are designed to perform read and write operations quickly and efficiently and provide horizontal scalability for storing large amounts of data (Rajaram, Sharma & Selvakumar, 2023).

One of the features that MongoDB offers is the ability to implement hybrid data distribution models, such as replicating shards in a cluster. This approach combines the benefits of sharding, which reduces load and improves performance, with replication, which ensures data durability and high availability.

© Editorial Académica Dragón Azteca S. de R.L. de C.V. (EDITADA.ORG), All rights reserved.

By replicating shards across a cluster, PDCs can achieve a balance between data distribution and redundancy, optimizing both performance and reliability. Moreover, MongoDB's support for OLTP ensures that PDCs can process transactions in real time, facilitating efficient inventory management and sales tracking.

1.1 Relational to Document-Based

As more companies seek to provide better services to their users, the question many are asking is not whether they should change their Database Management System (DBMS), but rather whether or not they need to restructure their existing data schemas once they do.

However, migration from one DBMS to another with different characteristics is a challenge for data administrators. Different strategies have been designed in the literature to address the migration from relational to NoSQL.

NoSQLayer is a framework that is integrated by two modules that handle data migration and data mapping (Rocha et al., 2015) Model to migrate relational database instance at a given time (snapshot) and in real time in a parallel way (Namdeo & Suman, 2021). Migration based on transforming the relational database to a NoSQL database and a cleansing process to enhance and improve data quality (Erraji et al., 2022).

It is possible to migrate data directly from any relational database to MongoDB, it is a mistake to use a document database the same way one would use a relational one (MongoDB, 2022).

Therefore, this paper aims to efficiently migrate the data from a relational database to a document-based database and data distribution strategy to deploy a distributed database MongoDB.

2 Data Modeling and Migration

Typically, data modeling occurs during the analysis and design phases of a project to ensure that the requirements for the new database are fully met before it is created (Hoberman, 2014). However, there are other uses for modeling, such as understanding the business model and rules. To take full advantage of the benefits, extending beyond conventional applications, we reorganized the data from one oriented toward applications to one that revolves around essential aspects of the business, but before one begins to look into how to change the database schema. It is necessary to understand how the business works so the applications supporting it will also work after the migration.

2.1 Restructuring Data

For restructuring data, we used a bottom-up approach to implement the physical integration of the data into a new NoSQL document-based database.

By following the legacy system’s main tables, attributes, and relationships in the relational model presented in Fig. 1, we got a sense of how the business operates. Simply put, a purchase to a provider is made by an employee; it is then stored in the company’s inventory to be later sold to a client by another employee. The next thing was to determine how to restructure the data to extract maximum value from the data. The begin task was to evaluate every relationship in the relational model in Figure 1 to decide if one should embed or reference the entities involved.

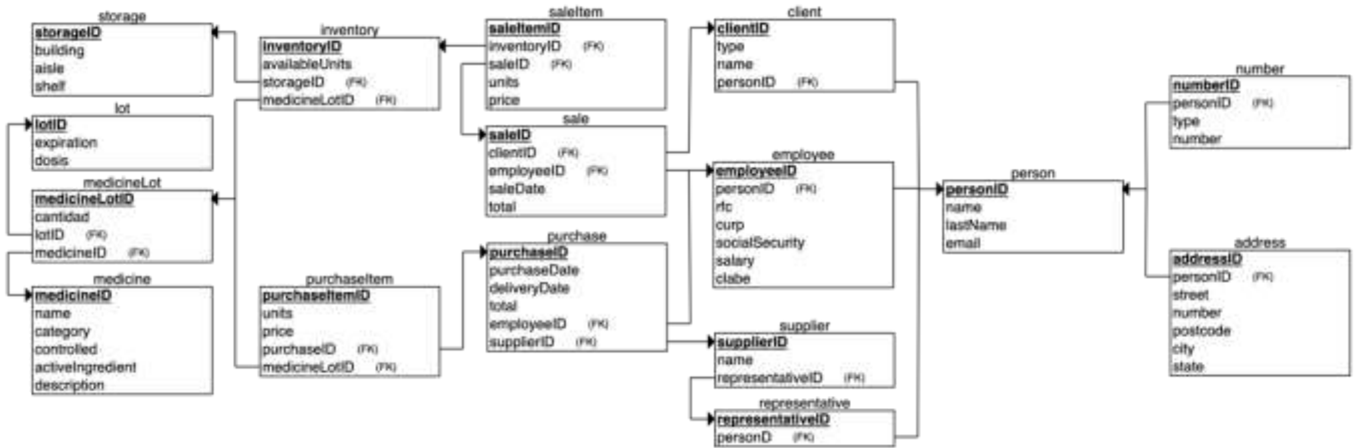

Fig. 1. Legacy relational database model.

MongoDB's guide (MoDB, 2024), offers a practical point-based system where we ask eleven yes/no questions about the data. For all questions, an affirmative answer accumulates points to embedding, while a negative answer contributes to referencing.

For example, the result for the relationship *inventory-storage* was ten points to embedding and one point to referencing; as result the Inventory document is then defined with the relationship data embedded (*building, aisle, and shelf*).

2.2 Data migration

Once the new data model with collections and documents has been designed as show in Fig. 2. We extracted the data from the relational database using relational algebra expressions and then define SQL queries with join and group to facilitate data processing. For example, the following relational algebra expression was used to extract all data of three tables in MySQL to populate the Client collection in MongoDB server:

$$\Pi_{\substack{client.clientID, client.name, client.type, \\ person.name+person.lastname, person.email, \\ address.number, address.street, address.number, \\ address.postcode, address.city, address.state}} (client \bowtie person \bowtie address)$$

As a general rule of thumb, we used join queries for the embedded tables and exported them to a comma-separated text file (*.csv), to later be processed through Python 3.9.6. Once the data was properly sorted, we used for-loops and print statements to produce a JavaScript file (*.js) with JSON data for each collection, so that they match the document model already established.

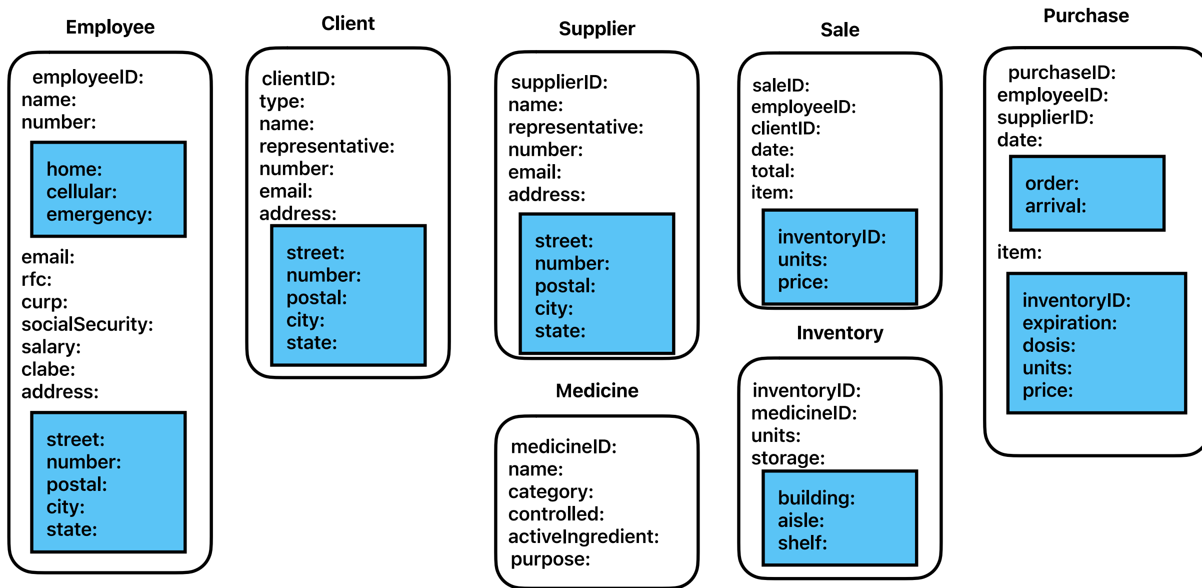

Fig. 2. NoSQL database model.

With the data processed and ready to be imported to MongoDB 7.0.2, we configured the database server in an Ubuntu 22.04 (Jammy Jellyfish) virtual machine with all its supporting components. First, starting the server and creating the user with the appropriate privileges that permitted it to manage the database and the replica set. We then populated the database using a simple load function to import the data in batches from Mongo Shell 2.0.2 to reduce load times.il.

3 Data Distribution Strategy

Database Administrators (DBA) have contended with the fact that their data is growing beyond their current database system for many years and have come up with two main ways of tackling this problem: vertical scaling (scaling up) and horizontal scaling - scaling out (Özsu & Valduriez, 2011).

Scaling up can provide immediate performance improvements because one typically upgrades their server hardware to more powerful options. However, limitations when scaling up can become a problem in large-scale deployments and costs. Scaling out involves adding more servers to a database system. Although this approach allows for better scaling, it often requires architecture designs that support sharding data across multiple servers.

3.1 Scaling out techniques

The two main techniques for scaling out are replication and sharding. Replication keeps identical copies of one's data on multiple servers. This way, the system can continue to function even if one server fails, thus adding fault tolerance and accessibility to your database system. Sharding of a database allows data to be segmented into partitions that are distributed across multiple instances of the distributed system, essentially to speed up queries and scale the system (Abdelhafiz & Elhadef, 2021).

The hybrid data distribution model proposed for the case study is presented in Fig. 3. Since the inventory collection is the most consulted and the largest in size, it is the one chosen to be fragmented and distributed to the nodes of the system.

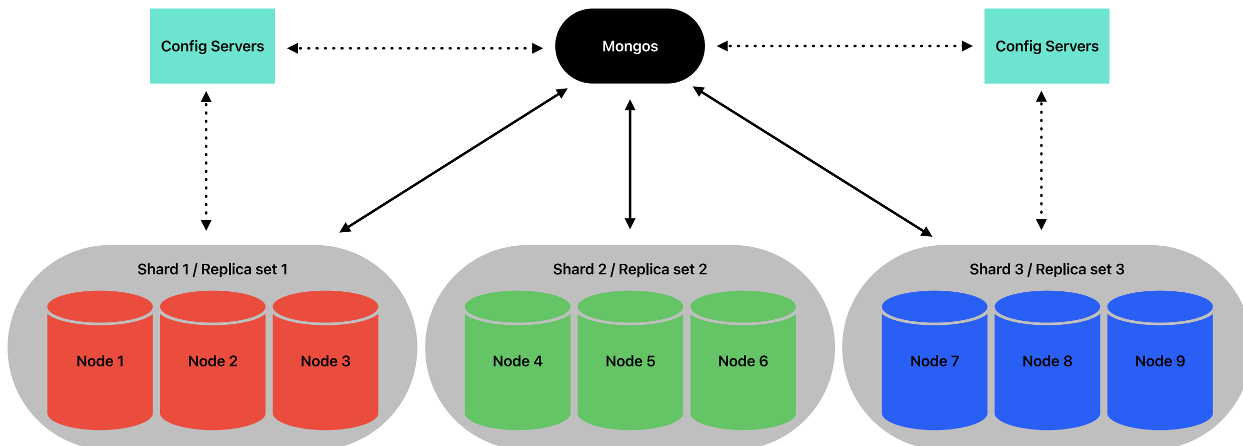

Fig. 3. Hybrid strategy for NoSQL data distribution.

To implement the proposed model, the combination of master-slave replication and sharding is performed in MongoDB. Three replica sets are used, each with three nodes (one master and two slaves), one replica set for the configuration server, and one router (Mongos).

3.2 Master - slave replication

In MongoDB the master node is called primary and receives the write operations. Slave nodes are called secondary nodes, which copy operations from the primary node.

The primary node can transmit data and operations to the secondary nodes in two ways: initial synchronization when a new (secondary) slave connects to the cluster, the primary node sends the complete set of data to the slave, and after complete synchronization, the primary node sends the ongoing changes to the new (and all) secondary nodes (Vágner & Al-Zaidi, 2023).

With the goal being sharded clusters, we must first create replica sets. It is necessary to be familiar with the format used; the prefix "#" will precede the code used in the system command line interface (Ubuntu terminal) and the prefix ">" is for the code used inside the MongoDB Shell.

The process implemented is as follows.

1. Copy the MongoDB configuration file to multiple nodes.
   # sudo cp /etc/mongodb.conf /etc/node{1..3}.conf

2. Create directories for the data of each node, config server, and the key file used to authenticate.
   # mkdir /data/node{1..3}
   # mkdir /data/csrs{1..3}
   # mkdir /data/keyfile

3. Generate a random key and set the appropriate permissions.
   # openssl rand -base64 756 > /data/keyfile/key

4. Edit the configuration file for each node.

```
# nano node1.conf
storage:
        dbpath: /data/node1
systemLog:
        path: /var/lg/node1.log
net:
        port: 27018
```

bindIp: 127.0.0.1, localhost
processManagement:
fork: true
security:
keyFile: /data/keyfile/key
replication:
replSetName: north

We will do the same with the other nodes, changing the storage dbpath, systemLog path, net port, and net bindIP with the data corresponding to each node.

3.3 Initialization and Configuration

The initialization and configuration of the nodes of the replica set were carried out with the following steps.

1. Start each MongoDB daemon.
   # mongod -f /etc/node1.conf

2. Authenticate and initialize the replica set.
   # mongod -f /etc/node1.conf
   > rs.initiate()

3. Connect to the MongoDB instance, and create a root user with a root role for authentication.
   > use admin
   > db.createUser({user:"root", pwd:"pwd", roles:["root"]})
   > db.auth("root","pwd")

4. Connect to the replica set and add secondary nodes to the replica set.
   > rs.add(127.0.0.1:27019)
   > rs.add(127.0.0.1:27020)

5. Check the status of the replica set before continuing.
   > rs.status()

Verify that the replica set was configured correctly and repeat the process twice. There should be a total of nine data nodes and three replica sets.

If the node primary (master) fails, one of the nodes secondaries (slaves) can be the new primary. The nodes in the replica set vote for the new master among the secondary nodes. Next to the primary and secondary nodes, there may be some referees in the replica set. The referee only participates in the voting if necessary. An arbitrator node does not store any data.

3.4 Sharding

In MongoDB the sharding unit is a document and uses shard keys to distribute the documents in a collection. The shard key is a field or more fields of the document and in most cases, it is not the same as the document key.

To shard the data, the collection must have an index in which the first part must be the sharding key. This means that the sharding configuration belongs to a collection.

After successfully configuring all the replica sets and data nodes, the next step is configuring the config servers and Mongos before configuring sharding.

To do this, we perform the following steps and execute the codes shown.

1. Configure the first configuration server
   # sudo nano csrs1.conf
   sharding:
   clusterRole: configsvr
   security:
   keyFile: /data/keyfile/key
   replication:
   replSetName: csrs-repl
   net:
   bindIp: 127.0.0.1
   port: 26011
   systemLog:
   destination: file
   path: /var/log/mongodb/csrs1.log
   processManagement:
   fork: true
   storage:
   dbPath: /data/csrs1

2. Copy the configuration file (modify the copied files to such that they contain the appropriate data)
   # sudo cp csrs1.conf csrs2.conf

3. Start the the configuration servers.
   # mongod -f csrs1.conf

4. Connect to the configuration server and initiate the replica set.
   # mongosh --port 26011
   > rs.initiate()

5. Add additional configuration servers to the replica set.
   > rs.add("127.0.0.1:26012")
   > rs.add("127.0.0.1:26013")

6. Check the status of the replica set before continuing.
   > rs.status()

3.5 Rolling Update

Rolling upgrade is important in a distributed system like MongoDB because it allows you to update or upgrade the system without causing downtime or service interruption.

Here is how to do it.

1. Connect to a secondary node, authenticate, shut down the server, and exit.
   # mongosh –port 27019
   > use admin
   > db.auth("root","root")
   > rs.stepdown()
   > db.shutdownServer()
   > exit

2. Update the secondary node configuration file by adding.
   # nano node2.conf
   sharding:
   clusterRole: shardsvr

3. Start the secondary node.
   # mongod -f node2.conf

4. Repeat for all the secondary nodes.

5. Connect to the primary node, authenticate, step down, and shut down the server.

6. Start the primary node.

3.6 Configure the Mongos

For the proposed model, the Mongos instance provides the interface between the client applications and the fragmented cluster. It will act like a router, routing queries, balancing chunks, and managing shards, among other things.

Once the replica set and config servers are configured, the next step was configuring the Mongos.

Here is how we configured it.

1. Create and configure the first MongoS.
```
# sudo nano mongos.conf
sharding:
        configDB: icbi-csrs/127.0.0.1:26011, 127.0.0.1:26012, 127.0.0.1:26013
security:
        keyFile: /data/keyfile/key
net:
        bindIp: localhost, 127.0.0.1
        port: 26000
systemLog:
        destination: file
        path: /var/log/mongodb/mongos.log
processManagement:
        fork: true
```

2. Start and connect to MongoS.
```
# mongos -f mongos.conf
# mongosh --port 26000
```

3. Add the shards to the MongoS.
```
# sh.addShard("127.0.0.1:26020")
# sh.addShard("127.0.0.1:26023")
# sh.addShard("127.0.0.1:26026")
```

Adding a single node from each replica set is sufficient; the MongoS will detect the other nodes and add them automatically.

4. Check that all shards were added.
```
# sh.status()
```

If every MongoDB server, config server, and MongoS were configured correctly, sh.status() should show something similar to what is presented in Fig. 4. Here, one can see that the sharded cluster contains three shards (central, north, and south), each with three nodes.

```
[direct: mongos] test> sh.status()
shardingVersion
{ _id: 1, clusterId: ObjectId('661ca9661fad890890eb110c') }
---
shards
[
  {
    _id: 'central',
    host: 'central/127.0.0.1:27023,127.0.0.1:27024,127.0.0.1:27025',
    state: 1,
    topologyTime: Timestamp({ t: 1713155402, i: 2 })
  },
  {
    _id: 'north',
    host: 'north/127.0.0.1:27020,127.0.0.1:27021,127.0.0.1:27022',
    state: 1,
    topologyTime: Timestamp({ t: 1713155375, i: 2 })
  },
  {
    _id: 'south',
    host: 'south/127.0.0.1:27026,127.0.0.1:27027,127.0.0.1:27028',
    state: 1,
    topologyTime: Timestamp({ t: 1713155425, i: 2 })
  }
]
```

Fig. 4. Sharded cluster status.

3.7 Sharding

Once the shard clusters are configured, need to choose a shard key. Since we already determined that we want our database sharded (by storage building). The next step was to add a single index or a compound index where the shard key is a prefix.

MongoDB's advice to choose a high cardinality field works well in most scenarios. In any case, do what works best for you and your data needs. The *inventoryID* was used as our shard key.

This is how the collection was sharded.

1. Connect to the MongoS.
   # mongosh –port 26000

2. Enable sharding on the desired database.
   > sh.enableSharding("ICBI")

3. Create an index on the field that will be used as the shard key.
   > use ICBI
   > db.inventario.createIndex({"inventarioID":1})

4. Shard the collection.
   > sh.shardCollection("ICBI.inventario",{"inventarioID":1})

5. Check the status of the sharded cluster.
   > sh.status()

6. Target a query.
   > db.inventario.find({"inventarioID":80002}).explain()

The query *db.inventario.find({"inventarioID":80002})* will run on the current database "ICBI" and find a document in the *"inventario"* collection where the *"inventarioID"* matches the value 80002.

Appending *.explain()* to the query, one can see the winning plan and where the document was stored. In Fig. 5, one can see that the data come from the shard named *"south"* as well as all three replication nodes in that shard, as well as the host, port, version, and *gitVersion* of the Mongo server.

```
[direct: mongos] ICBI> db.inventario.find({"inventarioID":83329}).explain()
{
  queryPlanner: {
    mongosPlannerVersion: 1,
    winningPlan: {
      stage: 'SINGLE_SHARD',
      shards: [
        {
          shardName: 'south',
          connectionString: 'south/127.0.0.1:27026,127.0.0.1:27027,127.0.0.1:27028',
          serverInfo: {
            host: 'ubuntu-linux-22-04-desktop',
            port: 27026,
            version: '7.0.8',
            gitVersion: 'c5d33e55ba38d98e2f48765ec4e55338d67a4a64'
          },
```

Fig. 5. Shard "south" and their three replication nodes.

Once our hybrid data distribution model has been configured, it is important to monitor and maintain the set of nodes on a regular basis to ensure optimal performance. Some best practices for sharding MongoDB are determine the correct shard key, plan for data growth, use replica sets for shard servers, monitor shard performance, and plan for disaster recovery.

4 Discussion of the Results

Data redundancy can sometimes seem like a juggling act for DBAs, wanting to reduce storage and maintenance costs while keeping servers on standby, synchronized, and up to date in an emergency. The model implemented might not be budget friendly.

Servicing these servers might not be feasible for some, but it does have a proper amount of availability. The model allows servers to go offline without having downtime on the applications that depend on them.

Having the biggest resource-consuming data divided into several servers allows for better-performing queries. At the same time, if one of the shards fails, a replica can take its place at any given moment.

5 Conclusions

The successful implementation of our strategy for the distribution of a database NoSQL emphasizes the importance of leveraging solutions to meet the demands of a rapidly changing industry landscape. The NoSQL database modeling process based on the detailed analysis of the relational database allowed the successful migration of the data to the collections and documents created in MongoDB. The configuration of the nodes for the implementation of the data distribution allowed us to test the proposed hybrid strategy and verify the performance of the system. While enterprises as PDCs continue to navigate the challenges of Big Data, modern database technologies, and scalability, scalable architectures will be paramount to achieving sustainable growth and a competitive advantage.

References

Abdelhafiz, B. M., & Elhadef, M. (2021). Sharding database for fault tolerance and scalability of data. *2nd International Conference on Computation, Automation and Knowledge Management (ICCAKM)*. 17-24. IEEE.

Bradshaw, S., & Chodorow, K. (2019). *MongoDB: Powerful and Scalable Data Storage.* O'Reilly Media.

Erraji, A., Maizate, A., Ouzzif, M., & Batouta, Z. I. (2022). Migrating Data Semantic from Relational Database System to NOSQL Systems to Improve Data Quality for Big Data Analyt4cs System. *ECS Transactions*, 107(1).

Hoberman, S. (2014). *Data modeling for MongoDB: Building Well-Designed and Supportable MongoDB Databases*. Technics Publications.

MoDB. Courses and Trainings (10/04/2024). MongoDB University https://learn.mongodb.com/learn/course/modeling-data-relationships/files

MongoDB. (10/04/2023). *Embedding MongoDB documents for ease and performance*. MongoDB Basics https://www.mongodb.com/basics/embeddied-mongodb

Mukherjee, S. B., Ghatak, S. G. N., & Ray, N. (Eds.). (2021). *Digitization of Economy and Society: Emerging Paradigms.* CRC Press.

Namdeo, B., & Suman, U. (2021). A Model for Relational to NoSQL database Migration: Snapshot-Live Stream Db Migration Model. *7th International Conference on Advanced Computing and Communication Systems (ICACCS),* (1), 199-204.

Niu, Y., Ying, L., Yang, J., Bao, M., & Sivaparthipan, C. B. (2021). Organizational business intelligence and decision making using big data analytics. *Information Processing & Management*, 58(6).

Özsu, M. T., & Valduriez, P. (2011). *Principles of distributed database systems, Third edition*. Springer Science.

Rajaram, K., Sharma, P., & Selvakumar, S. (2023). DLoader: Migration of Data from SQL to NoSQL Databases. *Proceedings of the International Conference on Cognitive and Intelligent Computing: ICCIC 2021,* 2(1), 193-204.

Rocha, L., Vale, F., Cirilo, E., Barbosa, D., & Mourão, F. (2015). A Framework for Migrating Relational Datasets to NoSQL, *Procedia Computer Science,* (51), 2593-2602.

Silberschatz, A., Korth, H., & Sudarshan, S. (2011). *Database system concepts, Sixth edition.* McGraw Hill.

Vágner, A., & Al-Zaidi, M. (2023). Sharding and Master-Slave Replication of NoSQL Databases: Comparison of MongoDB and Redis. *Proceedings of the 12th International Conference on Data Science, Technology and Applications (DATA 2023).* 576-582. https://doi.org/10.5220/0012142700003541