School of Information Technology and Engineering at the ADA University

School of Engineering and Applied Science at the George Washington University

Transformation of SQL Databases models to NoSQL

A Thesis
Presented to the Graduate Program of Computer Science and Data Analytics
of the School of Information Technology and Engineering
ADA University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science and Data Analytics
ADA University

By
Turqay Umudzade

April 24th, 2023

THESIS ACCEPTANCE

This Thesis by: Turgay Umudzade
Entitled: *Transformation of SQL Databases models to NoSQL*

has been approved as meeting the requirement for the Degree of Master of Science in Computer Science and Data Analytics of the School of Information Technology and Engineering, ADA University.

Approved:

(Adviser)	(Date)
(Program Director)	(Date)
(Dean)	(Date)

ACADEMIC INTEGRITY STATEMENT

“I affirm that this is my own work, I attributed where I used the work of others, I did not facilitate academic dishonesty for myself or others, and I used only authorized resources for my thesis, per the ADA University Academic Integrity requirements. If I failed to comply with this statement, I understand consequences will follow my actions. Consequences may range from failing the course to expulsion from the program/university and may include a transcript notation.”

Turgay Umudzade		24.04.2023
(Full Name)	(Signature)	(Date: DD.MM.YY)

ABSTRACT

The rapid growth of data-driven applications and the increasing need for flexibility and scalability in database systems have led to a surge in interest in NoSQL databases. As organizations transition from traditional relational databases to NoSQL databases, schema conversion becomes a critical challenge. This research presents a novel graph-based algorithm for converting relational database schemas to NoSQL formats, specifically MongoDB, while preserving data integrity and relationships.

The proposed algorithm utilizes graphs to represent and analyze relationships between tables in a relational database schema. By calculating foreign key sequences, the algorithm guides the schema conversion process, ensuring that the relationships between tables are maintained during the transformation. The conversion process involves referencing and embedding techniques to construct a hierarchical data representation in the target NoSQL format. This research also explores multiple approaches to schema conversion and identifies the most suitable methods based on various factors.

Although performance optimization was not the primary goal of this research, the proposed algorithm demonstrates promising results in terms of efficiency. The research highlights the algorithm's versatility and usefulness in various scenarios, aiding organizations in migrating from relational databases to modern NoSQL databases, such as MongoDB.

Future work includes refining the algorithm to handle edge cases, supporting different NoSQL database paradigms, optimizing performance, and testing scalability for large-scale databases. By addressing these areas, the algorithm can be further developed and tailored to cater to a broader range of database structures and systems, paving the way for new advancements in database schema conversion.

TABLE OF CONTENTS

Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Explanation
RDBMS	Relational Database Management System
NoSQL	Not only SQL (refers to non-relational databases)
FK	Foreign Key
PK	Primary Key
JSON	JavaScript Object Notation
DB	Database
API	Application Programming Interface
CRUD	Create, Read, Update, Delete
SQL	Structured Query Language
BSON	Binary JSON
GUI	Graphical User Interface

1.1 *Definition of the Problem*

In the modern era, data is used in all parts of our lives, including statistics, business applications, social media, and much more. None of these could be managed properly without a reliable database management system.

The need for database management systems started in the 1900s when people began to use the data we produce to our benefit. Even today the general idea of a DBMS is to have a convenient way of storing, retrieving, and updating data, including taking precautions in order not to lose the data or its confidentiality.

Since 1979 relational or SQL databases have dominated the market due to the need of that time, but today with the rapid growth in technology some problems have started to arise. One of them is billions of people having access to modern computers and interacting with the world using the internet, creating, updating, and deleting an enormous amount of data each day started to cause performance and scalability issues with relational databases. Furthermore, with the rising popularity of different portable computers like mobile phones, smartwatches, et cetera, people are producing an even larger amount of unstructured data which is not suitable for relational databases at all.

With these problems in mind, NoSQL databases were proposed as a solution, due to the fact that they were easier to scale and some of them could support unstructured data. They are now commonly used in many industries, including but not limited to social networks, machine learning applications, and cloud computing.

However, despite their origins in a long-forgotten technology cycle, relational SQL databases are by no means ‘legacy’ technology. Some SQL databases, notably PostgreSQL and MySQL, have experienced a recent resurgence in popularity. A new generation of NewSQL databases, notably Google Spanner and Cockroach DB, leverage SQL as a query language and offer a distributed architecture similar to that of NoSQL databases yet provide full transactional support.

As more and more companies look to scale their operations and deal with ever-increasing amounts of data, many are considering the benefits of using NoSQL databases over traditional SQL databases. While NoSQL databases offer advantages in terms of scalability, flexibility, and handling unstructured data, many organizations face significant challenges when it comes to migrating their existing SQL databases to NoSQL.

One of the main challenges is the amount of manual labor required to migrate from SQL to NoSQL. Moving data between different types of databases can be a time-consuming and complex process, requiring significant expertise and resources. In many cases, companies need to hire dedicated teams to handle the migration, and even then, the process can take months or even years to complete.
However, recent research has focused on developing more efficient and automated methods for migrating from SQL to NoSQL. By leveraging advanced data migration tools and algorithms, researchers aim to streamline the process and make it more accessible to organizations of all sizes.

Additionally, researchers are exploring ways to optimize NoSQL databases for specific use cases, such as machine learning or real-time analytics, to improve performance and scalability even further.
By optimizing the migration process, organizations can reduce costs and improve efficiency while still enjoying the benefits of using NoSQL databases. While the challenges of migrating from SQL to NoSQL are significant, the rewards can be substantial in terms of improved performance, flexibility,

and scalability. As research continues to develop new tools and techniques for database migration and optimization, we can expect to see even more organizations adopt NoSQL databases in the years to come.

1.2 *Objective of the Study*

Our research will examine the different ways of transferring data from a relational database (in our case PostgreSQL) to a destination NoSQL database (such as MongoDB). We aim to find the most effective and efficient methods for transforming the data model and relationships between tables in the source database, to adapt them to the structure and requirements of the target NoSQL database. We achieve this starting with a transformation for a simple table, including the data and the type of the data, later moving on to transforming the relations of a SQL database to a Mongo collection using different methods (Referencing and Embedding).

Our study will focus on ensuring that the migrated data is accurate and the integrity of the information is preserved. We will conduct performance tests to evaluate the transformation's impact on the target database's efficiency and responsiveness. Additionally, we will analyze the benefits and challenges of such a migration, providing valuable insights for organizations considering a switch to NoSQL databases.

Our research will contribute to developing practical guidelines and models for effectively transforming SQL databases to NoSQL databases. We aim to provide organizations with a better understanding of the benefits and challenges associated with such a migration and assist them in leveraging the benefits of scalability, flexibility, and handling unstructured data.

1.3 *Significance of the Problem*

Traditional SQL databases face scalability challenges in today's data-driven world, where the most prominent companies are technology-based and rely heavily on processing and storing vast amounts of data. Established companies that adopted SQL databases in the late 1900s and early 2000s are now struggling to cope with the rapid growth of data. As a result, they are investing millions of dollars and countless hours of labor in migrating their databases to more scalable models, primarily NoSQL databases.

NoSQL databases come in various types, including document stores, key-value stores, and graph databases. This research will focus on document-based NoSQL databases, which are currently the industry's most popular choice, specifically MongoDB.

No single database performs well in all applications, as different applications have unique data storage, analysis, and processing requirements. This is even true for different SQL databases and more when considering what NoSQL databases are capable of. Several researchers, including John Ambler [11], state that organizations should use NoSQL databases for specific applications that offer a more efficient and scalable solution than relational databases.
However, as of today, no primary tools are available for transforming relational databases to NoSQL databases. This presents a significant challenge for companies seeking to migrate their databases and optimize their data storage and processing capabilities.

There needs to be more research regarding database transformations and tools from relational databases to NoSQL databases, especially on relational to document, and existing models are highly restricted. By doing so, this study will contribute to developing practical solutions for businesses facing scalability issues with their SQL databases, ultimately allowing them to harness the full potential of NoSQL databases for specific applications.

1.4 Review of Significant Research

In recent years, several research studies have attempted to address the issue of transforming relational databases to NoSQL databases. While there has not been a comprehensive solution discovered, some studies have partially tackled the problem by proposing methods for specific aspects of the transformation process.

The present study by Ghotiya, Mandal, and Kandasamy (2017) [1] had investigated multiples ways of transformations, that are described in the following table:

Table 1: Comparison of data migration approaches

S. No.	Author Name	Technique Used	Advantages	Efficiency
1	Liang, D., Lin, Y., & Ding, G. [1]	Mid-Model Design	Integrity of data Uniform for Any NoSQL databases	Requires metadata information
2	Rocha, L., Vale, F., Cirilo, E., Barbosa, D., & Mourão, F. [2]	NoSQL Layer	No changes required in the application code. Quantitative and qualitative evaluation ensures efficiency.	Only useful for MongoDB.
3	Chao-Hsien Lee, and Yu- Lin Zheng [4]	Correlation-aware Technique	Transform analyzed data into the same Hadoop datanode.	Cardinality should be the combination of all primary keys with the longest. chained length.
4	Chao-Hsien Lee and Yu-Lin Zheng[3]	Traditional web-based content management systems (CMS)	Flexibility while scaling up Greater computing power.	May limit thescalability
5	Liao, Y. T. et al.,[5]	Data Adapter system, MapReduce	Support hybrid database architecture Can handle database transformation.	Data inconsistency problem may occur.
6	Stanescu, L., Brezovan, M., & Burdescu, D. D. [6]	Automatic mapping framework.	Maintain the features of relational database. Data access easier than MySQL.	Works only for MongoDB. Causes overhead. Metadata required.
7	Xu, J. et al. [7]	Transparent query engine, ZQL	Hide the specific details ofboth NoSQL databases andRDBMS	Replication problem may occur

A fundamental motivation behind using NoSQL databases is to address issues related to replication and scalability, which are common in SQL databases. The table above provides an overview of various research studies that have explored different techniques for transforming data between SQL and NoSQL databases. These techniques address various challenges, such as maintaining data integrity, providing flexibility during scaling, and supporting hybrid database architectures.

However, some of these methods may have limitations, such as applying only to specific NoSQL databases or requiring additional metadata information. Additionally, specific techniques may still encounter replication problems. Overall, the table offers a comprehensive insight into the current state of research on data transformation techniques between SQL and NoSQL databases, emphasizing the need to overcome scalability and replication challenges.

The research conducted by Abo Dabowsa et al. (2021) [2] introduces a comprehensive method for migrating data from an existing relational database (RDB) to a MongoDB NoSQL database, ensuring data integrity and minimizing redundancy. The method focuses on preserving relationships and constraints from the source RDB to the target NoSQL database, using a tree-like structure with parent references to represent relationships.

The approach consists of three main phases: Metadata Extraction, Schema Translation, and Data Conversion. In the Metadata Extraction phase, the source RDB is analyzed to gather information about its schemas, such as tables, attributes, relationships, and keys. This metadata provides the foundation for the subsequent phases.
.
During the Schema Translation phase, the method translates the extracted metadata into a target structure (TS) that closely resembles the original RDB schema while compatible with MongoDB. This target structure is defined as a set of collections, each containing a set of documents with fields representing the original data.

Finally, in the Data Conversion phase, the method utilizes a data conversion algorithm to migrate the data from the RDB into the target MongoDB database. The data is transformed from rows in the RDB into documents in MongoDB, maintaining the original properties and values from the source RDB.

Compared to other research conducted in this area, the proposed method maintains data integrity and consistency during the migration process. Its focus on preserving relationships and constraints ensures a smooth transition from the source RDB to the target NoSQL database. The algorithms implemented in the system and the experimental validation demonstrate the accuracy of the conversion from input schema to output structure.

In summary, the approach presented in this paper significantly contributes to the field of data migration from relational to NoSQL databases. Its focus on data integrity, preservation of relationships and constraints, and the effective implementation of the three-phase approach make it a valuable resource.

Zhao, Lin, Li, and Li (2014) proposed a schema conversion model of SQL database to NoSQL in their publication [3], a model designed to convert schemas from SQL databases to NoSQL databases, emphasizing the table nesting process. They introduced a graph model that describes general database schemas, including relational and NoSQL schemas. Through this graph model, the authors outlined the procedure of schema conversion as a sequence of extension operations that transform the original schema into the final schema.

The authors proposed a general migration program and related algorithms based on the nested method to improve the query speed of NoSQL databases. They implemented an experimental MySQL-MongoDB migration system to demonstrate the practicality of their approach. Real-world data was used to verify the correctness of the migration and compare the query performance.

The authors rigorously proved the conversion algorithm's correctness. However, they acknowledged that the model has a downside – it increases space consumption in exchange for improved query efficiency. As a result, the authors suggested that future work should focus on minimizing spatial redundancy while maintaining the benefits of the proposed method.

This paper presents an innovative schema conversion model that leverages a nested approach to enhance query speed in NoSQL databases. The authors demonstrated the effectiveness of their model in migrating data from SQL to NoSQL databases and highlighted the need for further research to minimize spatial redundancy.

In summary, the existing body of research on this subject is relatively scarce, and the methodologies employed thus far exhibit certain shortcomings. We aim to develop a more flexible and extensible model that mitigates these drawbacks, primarily emphasizing data accuracy and preserving NoSQL's inherent advantages, such as high scalability. By doing so, we aim to contribute to the ongoing academic discourse and expand the knowledge base surrounding this crucial study area.

1.5 Assumptions and Limitations

Migrating from SQL to NoSQL is a complex challenge. This paper does not yield a perfect model at the end of the study. The study's primary goal is to find different models that can be later improved in other studies and a model which can at least work with different types of SQL databases having two to five tables and a data set that can fit in one shard of a database. Due to the limited hardware and workforce, the study will only focus on small databases and a single database server (e.g., no sharding or clustering of databases).

To begin this study, the technical literature relevant to various aspects of the study was reviewed to ensure an understanding of the relevant technologies that would be considered. This comprehensive review encompassed methodologies, techniques, and best practices in database schema conversion and NoSQL technologies. By examining the existing literature, this research aims to build upon and contribute to the ongoing discourse, ultimately fostering a deeper understanding of the challenges and opportunities in transitioning from relational databases to modern NoSQL database systems.

2.1 Relational Databases

Relational databases are built on the idea of creating relations between entities and dividing structured data using those relations. The data is stored in a combination of single columns that are linked via foreign keys. [4] They commonly consist of one to one, one to many, many to many, many to one relations and Structured Query Language (SQL) is used to interact with the database.

Normalization is crucial in designing relational databases to avoid issues. A set of normalization rules is used to maintain data consistency, and when updating data, it is only done in one place instead of multiple tables. The rules include:

- Creating a separate table for each set of related data designated by a primary key to eliminate groupings of repeating data.
- Moving a group of values to a new table and using a foreign key to connect the two tables if the group of values is the same for numerous entries.
- Removing fields that are not dependent on a table's primary key and relocating them to another table if required.

Additional normalization rules such as Boyce-Codd, 4NF, 5NF have been introduced, but the above-mentioned three are the main forms to consider a schema normalized. [4]

Most SQL databases share some common features and syntax, but they also have key differences. When comparing the most famous SQL databases such as Oracle, MSSQL Server, MySQL, and PostgreSQL, some differences in their compression capabilities can be observed [5]:

- PostgreSQL supports JSON.
- SQL Server uses Transact SQL and .NET for server-side scripting, whereas Oracle uses PL/SQL languages.
- PostgreSQL supports complex structures and a wide range of built-in and user-defined data types.
- MySQL supports a large number of storage engines, including InnoDB, MyISAM, and MEMORY.

2.2 Cap theorem

One of the most important factors when choosing a database is the need for its use and the trade-offs that need to be taken. CAP stands for Consistency, Availability, and Partition tolerance, and the theorem specifies that when choosing among those three factors, you can only choose two at a time. The term specifies that in a distributed systems environment, you must choose between availability or strong consistency. [6]

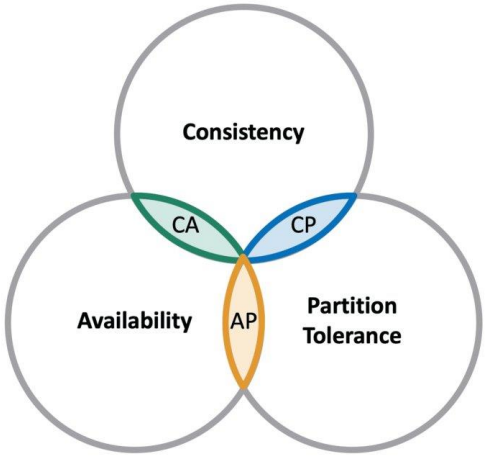


Figure 1:CAP Theorem Diagram [6]

2.3 Consistency

Consistency refers to getting consistent data for each request, it comes in three forms, weak consistency, eventual consistency, and strong consistency.

Weak consistency refers to when each server can make progress on its own, a good example of this would be a distributed web-cache service. [7]

Eventual consistency [8] is the best popular choice for most applications, it is a model that enables to have high availability while still maintaining useful levels of consistency. In a real-world use case of social media platforms, they run multiples nodes from different regions, making it a highly available system since there is no single point of failure. Making the data of different nodes highly available comes at a cost that they are not always in sync. So different users from different regions might get different data, but that is acceptable since the data is not critical, a good example of this is the number of likes on any social media platform, as time goes on the servers will sync with each other leading the data to eventually be consistent.

A study [8] on Amazon S3 storage investigated how fast ’eventual’ really is by testing nodes from 3 different regions, inconsistency window results are graphed in Fig. 2, furthermore, with 353,357,884 read operations. The result showed that 42,565,840 or about 12% of the requests violated monotonic read consistency, while the system only returned a single error resulting in a 99.9999997% available system.

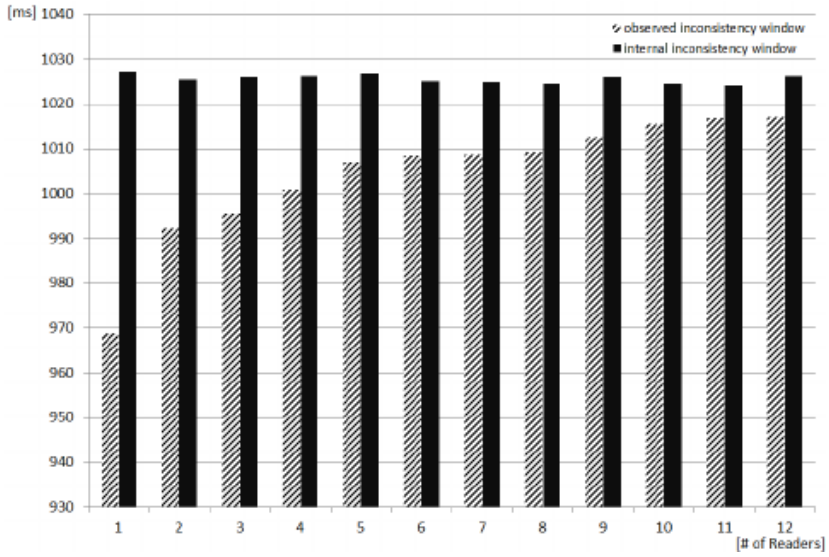


Figure 2:Inconsistency Windows [8]

A system is considered to have strong consistency when all the nodes across the world provide the same data, to achieve this, the only solution is to lock the nodes from being accessible while they are syncing with the other nodes, which in return makes the system not highly available.

2.4 Availability

Availability is usually a measure of what percentage of the year a system was up for to return a response, even though a late response is also considered available, in most cases a late response is as good as no response at all. When talking about an AP system, the main goal is to have high availability. High availability ensures that the service is up for almost 100% of the time, the main goal is to strive for 99.999% availability or the five 9s rule, this involves hardware or software fault tolerance. To achieve a system every system and database has its own mechanism of handling failures, which involves different methods of data replication to avoid data loss and have a node that returns the desired data.

2.5 Partition Tolerance

Partition tolerance is the third part of CAP. Partition tolerance, unlike the other two characteristics, is more of a statement about the underlying system than about the service: a connection between servers is unstable, and servers can be partitioned into various groups that cannot communicate with one another. A partition-prone system is modeled as one that struggles from improper communication, with messages being delayed and occasionally lost altogether. (A long-delayed message might as well be lost in practical terms.) [7]

2.6 ACID vs BASE

Ensuring performance and consistency in distributed computing environments is crucial, and transaction control plays a significant role in achieving this goal. There are two commonly used transaction control models, namely ACID (used in RDBMS) and BASE (found in many NoSQL systems). It is important to note that even if only a few database transactions require transactional integrity, both RDBMSs and NoSQL systems can implement these controls. The distinction between these models lies in the level of effort required by application developers and the location (tier) of the transactional controls.

In a distributed computing environment, transaction control is crucial for ensuring performance and consistency. When a website relies on computers located in various regions, the failure of one site may cause significant disruption to the overall system. Therefore, it is important to consider alternative transaction control models that do not rely solely on the ACID model.

In web applications such as shopping carts and checkout systems, the primary concern is not the consistency of reports but the ability to continue taking orders without interruption. This is where the BASE (Basic Availability, Soft-state, Eventual Consistency) model comes into play. Unlike the ACID model, which prioritizes consistency, BASE systems prioritize availability. They allow for temporary inconsistencies and data inaccuracies while still allowing transactions to occur. This relaxed approach simplifies the development process and allows for faster processing times.

Traditionally, ACID systems were seen as the only option for transactions in business, but with the rise of NoSQL systems, BASE models have become more widely used. NoSQL systems are highly decentralized and do not always require the guarantees provided by ACID transactions.

It is worth noting that ACID and BASE models are not mutually exclusive and can be used in conjunction with one another. Administrators and developers can determine which approach best suits the needs of their organization and implement them accordingly. Additionally, when managing large volumes of data, database sharding may be necessary to ensure systems remain operational.

In summary, when transitioning from centralized to distributed systems, transaction control becomes increasingly important. BASE systems offer a viable alternative to the traditional ACID model, prioritizing availability over consistency, and simplifying the development process. By considering the needs of the business and the specific requirements of the system, administrators and developers can make informed decisions about which approach to implement. [9]

2.7 ACID

ACID properties are essential because they guarantee that data is stored correctly and securely. Transactions can be trusted as they are always processed correctly and never interrupted, allowing users to have confidence in the system. Atomicity is the guarantee that all operations within a transaction happen, or none. Consistency ensures that the database will only accept valid data, keeping it reliable. Isolation makes sure that transactions are processed separately, avoiding interference with other operations. Finally, Durability guarantees that once a transaction is completed, its effects will remain even in the event of a system failure. All these properties ensure a reliable and secure database system.

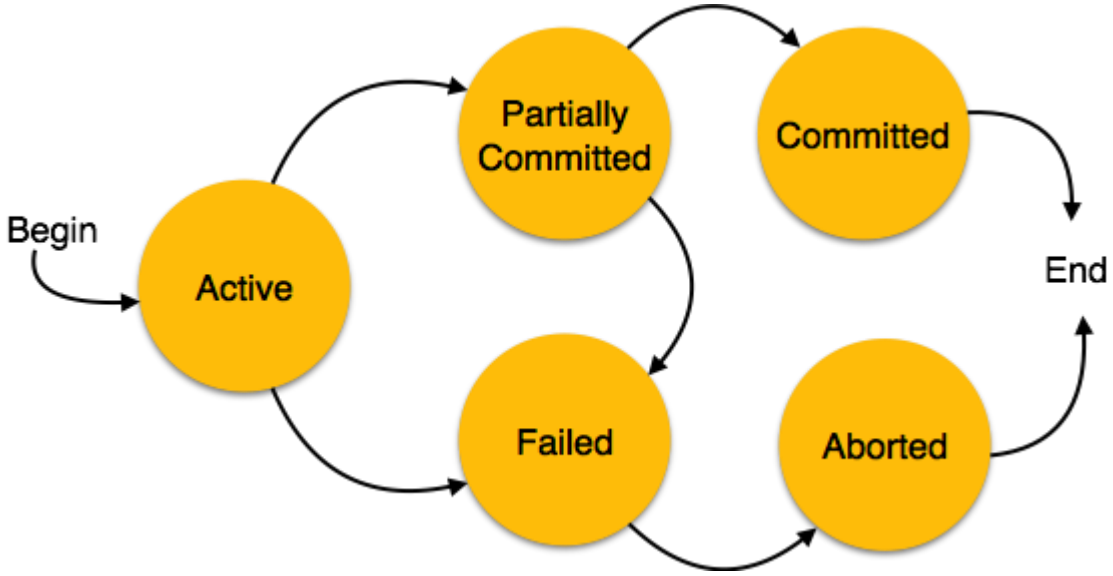


Figure 3 :States of Transactions Tutorialspoint.com. "DBMS - Transaction." [Online]. Available: https://www.tutorialspoint.com/dbms/dbms_transaction.htm. [Accessed: Apr. 17, 2023].

Atomicity: Atomicity means that all parts of a transaction must be completed, or the entire transaction is canceled. In the state of a transaction (Fig. 3), it can go two ways. [5]
Abort: If an operation fails a check, the database rolls back the operation, which is called aborting. After that, the system can either cancel the transaction or retry it.
Commit: If an operation is successful, it shall be committed to show visible changes.

Consistency: All database transactions can only transition the database from one valid state to another while maintaining database invariants: [5] Every write operation made to the database must be valid according to the pre-defined schema of the database. This prevents any chance of false data from entering the database and prevents an invalid state of the database but does not guarantee that the transaction is valid. Referential integrity assures the relation of the primary key to the foreign key.

Isolation: Within the realm of databases, the concept of isolation pertains to the idea that a transaction occurs independently from other transactions. This implies that the modifications made by one transaction remain hidden from other transactions until it is committed. Essentially, while a transaction is in progress, it appears as though it is the sole transaction taking place in the system.

Isolation plays a crucial role in ensuring that simultaneous transactions do not disrupt one another. Without isolation, conflicts and data inconsistencies may arise as multiple transactions attempt to modify the same data concurrently. Various isolation levels exist, each offering a distinct degree of separation between concurrent transactions. Some of these isolation levels include:

- ***Read Uncommitted***: This level permits transactions to access data that other transactions are currently modifying.
- ***Read Committed***: At this level, transactions can only read data that has been committed, excluding data currently being altered by other transactions.
- ***Repeatable Read:*** This level enables transactions to read identical data multiple times, without seeing the data being modified by other transactions.
- ***Serializable:*** As the highest isolation level, serializable allows transactions to operate as if they were executed sequentially, one after another.

Durability: Durability pertains to the characteristic that once a transaction has been committed, its effects are irreversible and remain intact despite any subsequent system failures. This implies that the modifications made by a committed transaction are saved to a durable storage medium, like a disk, ensuring their persistence even in cases of power outages, system crashes, or other types of failures.

Durability plays a critical role in guaranteeing the permanence of transaction changes, preventing data loss during system failures. Without durability, a committed transaction's changes might not be stored on disk and could be lost if a failure occurs.

Databases employ various methods to achieve durability, such as writing data to disk, maintaining transaction logs, and utilizing RAID (redundant array of independent disks) storage systems.

2.8 BASE

The BASE properties, an alternative to ACID properties in the context of NoSQL databases, encompass three fundamental aspects:

Basic Availability: This property ensures that data always remains accessible, even in partial system failures. As a result, NoSQL databases prioritize the continuous availability of data, allowing applications to function without interruption despite potential system disruptions.

Soft State: According to this property, individual data items function independently and do not necessitate consistency. This notion of independence permits NoSQL databases to accommodate data inconsistencies in the short term, allowing for greater flexibility in data management and system operations.

Eventual Consistency: This property posits that data may initially exhibit inconsistencies but eventually achieve consistency at an unspecified time. Consequently, NoSQL databases sacrifice a degree of immediate consistency to enhance availability and adaptability.

In summary, NoSQL databases prioritize eventual consistency over strong consistency, enabling them to manage data more effectively in distributed environments. Data stores adhering to BASE guarantees may occasionally fail to return the result of the most recent write operation, thereby providing differing responses to applications making requests. As a result, developers constructing applications on top of eventually consistent data stores often incorporate consistency checks within their application code, mitigating potential issues arising from temporary data inconsistencies. [9]

2.9 Referencing And Embedding

In the process of developing a new application, it is crucial to first establish a well-structured data model. For relational databases, such as MySQL, this is accomplished via normalization, which is a systematic approach to eliminate redundancy across multiple tables. However, MongoDB diverges from relational databases by storing data in structured documents rather than adhering to the constraints of fixed tables. For instance, while a relational table necessitates that each intersection of rows and columns contains a singular scalar value, MongoDB BSON documents enable more intricate structures by accommodating arrays of values, which may, in turn, consist of numerous subdocuments.

This chapter investigates one of the options presented by MongoDB's sophisticated document model: the decision to either embed related objects within one another or reference them through an identifier (ID). Throughout this discussion, readers will learn how to assess and balance factors such as performance, flexibility, and complexity when making this critical determination.[10]

Prior to delving into MongoDB's perspective on the matter of embedding documents versus linking documents, it is beneficial to briefly examine the modeling of specific types of relationships in relational (SQL) databases. In these databases, data modeling generally entails representing data as a collection of tables comprised of rows and columns, which together delineate the data schema. Relational database theory has established various methodologies for organizing application data into tables, known as normal forms. While an exhaustive discussion of relational modeling falls outside the purview of this thesis, two forms warrant particular attention in this context: first normal form and third normal form. [10]

2.10 Denormalization for Performance Enhancement

A somewhat concealed aspect of relational databases is that after completing the data modeling process to produce an optimized nth normal form data model, it frequently becomes necessary to denormalize the model to minimize the number of JOIN operations required for commonly executed queries.

In such situations, one might revert to storing the name and contact_id redundantly in the row. However, this reintroduces the redundancy initially sought to be eliminated, leading to increased application complexity, as updating data in all redundant locations becomes essential.

MongoDB enters the scene by challenging the notion that data must always be tabular, effectively discarding much of the traditional database normalization principles, beginning with the first normal form. In MongoDB, data is preserved in documents, enabling storage of an array of values, contrary to the first normal form in relational databases, which mandates that each row-column intersection contains a single value.

This flexibility presents new opportunities for schema design, as MongoDB's native ability to encode multivalued properties allows for performance improvements akin to those achieved through denormalization without the associated challenges of updating redundant data. However, this also complicates the schema design process, as there is no longer a straightforward path to normalized database design. Consequently, addressing general schema design problems in MongoDB often leads to the response, "it depends."

Before delving into the specifics of when and why to employ MongoDB's array types, it is essential to understand the nature of a MongoDB document. MongoDB documents are modeled on the JSON (JavaScript Object Notation) format but are stored in BSON (Binary JSON). In essence, this signifies that a MongoDB document comprises a dictionary containing key-value pairs, where the value can be one of several types:

- Primitive JSON types (e.g., number, string, Boolean)
- Primitive BSON types (e.g., datetime, ObjectId, UUID, regex)
- Arrays of values

- Objects consist of key-value pairs.
- Null

One possible motivation for embedding one-to-many relationships is data locality. As mentioned previously, spinning disks excel at sequential data transfer but perform poorly with random seeking. Since MongoDB stores documents adjacently on disk, incorporating all necessary data into a single document ensures that the required information is never more than one seek away.

MongoDB also imposes a limitation (stemming from the objective of effortless database partitioning) by not providing JOIN operations. For example, if referencing were used in a phone book application, the application might execute a process similar to the following:

```
contact_info = db.contacts.find_one({'_id': 3})
number_info = list(db.numbers.find({'contact_id': 3})
```

Nonetheless, adopting this approach results in a predicament that is arguably more problematic than a relational 'JOIN' operation. The database must still perform multiple seeks to locate the data, and additional latency is introduced into the lookup, as two roundtrips to the database are now required to obtain the information. Consequently, if an application frequently accesses contact details along with all associated phone numbers, embedding the numbers within the contact record would likely be the optimal strategy.[10]

2.11 Referencing for Enhanced Flexibility

While embedding often yields superior performance and data consistency assurances, certain cases may benefit from a more normalized model in MongoDB. One rationale for normalizing your data model across multiple collections is the heightened flexibility afforded in executing queries.

Consider a blogging application containing posts and comments. One potential strategy involves employing an embedded schema:

```
{
  "_id": "First Post",
  "author": "Rick",
  "text": "This is my first post",
  "comments": [
     { "author": "Stuart", "text": "Nice post!" },
     ...
     ]
}
```

This schema is effective for creating and displaying comments and posts. However, if a feature is introduced that allows users to search for all comments by a specific individual, the query (utilizing this embedded schema) would resemble the following:

```
// db.posts schema
{
  "_id": "First Post",
  "author": "Rick",
  "text": "This is my first post"
}


// db.comments schema
{
  "_id": ObjectId(...),
  "post_id": "First Post",
  "author": "Stuart",
  "text": "Nice post!"
}
```

Generally, if an application's query pattern is well-established and data is predominantly accessed in a singular manner, an embedded approach is suitable. Conversely, if data is subject to multiple query variations or if it is difficult to predict query patterns, a more "normalized" approach might be more appropriate. For example, in the "linked" schema, it is possible to sort the comments of interest or limit the number of comments returned from a query using limit () and skip () operators. In contrast, the embedded scenario necessitates retrieving all comments in the order that they are stored within the post.[10]

Numerous businesses are adopting NoSQL databases for data storage and management, necessitating the transformation of existing relational databases to NoSQL. Due to significant differences in data schemas between these two types of databases, users face a steep learning curve. Moreover, NoSQL databases do not support join operations, which can lead to multiple separate reads and reduced performance. Therefore, schema conversion and efficient data import from relational to NoSQL databases are crucial.

While there are various relational databases like Oracle, SQL Server, and MySQL with similar relational schemas, each NoSQL database has its unique data schema. NoSQL databases can be categorized into four main types: key-value, document, columnar, and graph databases. This paper focuses on the first three types, excluding graph databases that are designed for specific use cases like highly interconnected social networking data.

We present a system that implements their proposed solution, offering a general schema conversion model for transitioning from relational to NoSQL databases, which aids migration and enhances reading efficiency. They introduced the concept of table nesting to improve cross-table query performance. By treating references between tables as relationships between parent and child layers of semi-structured data in NoSQL databases, they store referred tables as child tags in the referring table's data item. This nesting approach guarantees a one-to-one mapping between relational tables and NoSQL datasets.

Using MongoDB as an example, if a relational database has 'n' tables, there must be 'n' corresponding collections in MongoDB. The paper [3] also proposes a graph transformation algorithm to generate a nesting sequence among relational tables, ensuring accurate schema conversion. By representing tables as vertices and table references as edges, they construct a graph based on the relationships between tables in a relational database. The elimination sequence, determined by the transformation algorithm, corresponds to the nesting sequence.

After schema conversion, a single access to a table is needed for any query statement, enabling efficient reading on NoSQL databases. Although this conversion model requires more storage space, it remains feasible due to lower storage costs.

3.1 Installation

During our research, we opted to utilize PostgreSQL and MongoDB as the primary databases for our experiments, primarily due to their widespread adoption and popularity within the industry. The former is a well-established relational database management system, while the latter is a widely used NoSQL document database. Both databases were deployed using Docker containers and orchestrated with Docker Compose, allowing seamless installation and management. Specifically, our experimental setup employed PostgreSQL version 15.2 and MongoDB version 6.0.0.

Our implementation was developed using JavaScript, a versatile and ubiquitous programming language, in conjunction with Node.js. This runtime environment facilitates the execution of JavaScript code outside the confines of a web browser. Similar outcomes could be achieved by deploying these databases on local machines instead of utilizing Docker containers. However, the containerized approach offers a more streamlined and consistent environment, which is particularly advantageous in collaborative research endeavors.

3.2 SQL to Graph

The process of converting database information into a graph representation consists of two primary steps: (1) retrieving the data structure information (e.g., table names, columns, data types, etc.) and (2) obtaining the relationships between tables to establish the graph's connections. Explanations for the queries involved are provided at each step.

3.3 Step 1: Retrieve data structure information.

The initial phase of the schema conversion process involves obtaining data structure information from the source database. This crucial step enables the algorithm to accurately represent and analyze the relationships between tables, which is vital for successfully converting the schema from a relational database to a NoSQL format. By thoroughly extracting the data structure information, the algorithm can effectively identify foreign key relationships and other crucial elements to be considered during the transformation process.

```
getAllTablesQuery:
SELECT *
FROM information_schema.tables
WHERE table_schema = 'public';
```

This query retrieves all the tables in the 'public' schema of the database. The information_schema.tables table holds metadata about all the tables in the database. By filtering the result set using “table_schema = 'public' ”, the query returns only the tables within the public schema.

```
getData function:
const getData = (table) => {
return `SELECT column_name, data_type
FROM information_schema.columns
WHERE table_name = '${table}'; `; };
```

This JavaScript function generates a query that retrieves column information (name and data type) for a specific table. The information_schema.columns table contains metadata about all columns in the database. By filtering the result set using table_name = '${table}', the query returns only the columns associated with the given table name.

3.4 Step 2: Obtain relationships between tables.

The second step in the schema conversion process entails identifying and capturing the relationships between the various tables in the source database. This crucial stage allows the algorithm to understand how tables are interconnected, which is essential for preserving data integrity and maintaining relationships during the conversion to a NoSQL format. The algorithm can better guide the transformation process by accurately mapping the relationships between tables and ensuring a seamless transition from the relational database to the target NoSQL database system.

```
SELECT tc.table_name, kcu.column_name, ccu.table_name
AS foreign_table_name, ccu.column_name
AS foreign_column_name
FROM information_schema.table_constraints tc
JOIN information_schema.key_column_usage kcu
ON tc.constraint_name = kcu.constraint_name
AND tc.table_schema = kcu.table_schema
JOIN information_schema.constraint_column_usage ccu
ON ccu.constraint_name = tc.constraint_name
AND ccu.table_schema = tc.table_schema
WHERE tc.constraint_type = 'FOREIGN KEY'
ORDER BY tc.table_name, kcu.ordinal_position;
```

This SQL query retrieves information about foreign key constraints in a PostgreSQL database. The purpose of the query is to list all the foreign key relationships between tables, including the columns involved in these relationships. The query is written using a multi-line template literal in JavaScript.

The query can be broken down into the following components:

- **SELECT** statement: This part of the query specifies the columns to be retrieved in the result set, including:
- **tc.table_name**: The name of the table containing the foreign key.
- **kcu.column_name**: The name of the column that is a foreign key in the table.
- **ccu.table_name AS foreign_table_name**: The name of the referenced table (the table being referred to by the foreign key), aliased as "foreign_table_name".
- **ccu.column_name AS foreign_column_name**: The name of the column in the referenced table, aliased as "foreign_column_name".
- **FROM** clause: This part defines the primary table for the query, which is the information_schema.table_constraints table (aliased as tc). This table contains metadata about constraints in the database.
- The first **JOIN** clause: This part joins the **information_schema.key_column_usage** table (aliased as **kcu**) to the primary table, based on matching constraint names and table schema. This table provides information about columns that are part of constraints.
- The second JOIN clause: This part joins the **information_schema.constraint_column_usage** table (aliased as **ccu**) to the primary table, again based on matching constraint names and table schema. This table contains information about columns that are referenced by constraints.
- **WHERE** clause: This part filters the result set, only including rows where the constraint type is 'FOREIGN KEY'. This ensures that only foreign key relationships are considered.
- ORDER BY clause: This part sorts the result set by table name and ordinal position (the position of the column in the table) to present the relationships in a more organized and readable manner.

ALGORITHM 1: Database Schema Graph Conversion

```
Initialize graph: graph = {}
Query table relationships and store result in rows
For each row in rows, do:
    Extract table relationship data: tableName, columnName, foreignTableName,
    foreignColumnName
    Add table node to graph:
        If graph[tableName] does not exist, create it with empty edges list:
        graph[tableName] = { edges: [] }
    Add foreign table node to graph:
        If graph[foreignTableName] does not exist, create it with empty edges list:
        graph[foreignTableName] = { edges: [] }
    Add edge between table nodes in graph:
        Append the edge to the graph[tableName].edges: {table: foreignTableName,
        column: foreignColumnName, refColumn: columnName}
Query table data and store result in tableNames
Initialize empty list tables = []
For each table in tableNames, do:
    Query table columns and store result in rows
    Extract columns from rows: columns = rows.map(...)
    Create a table object with the name and columns: {name: table, columns}
    Append the table object to the tables list
Associate table data with graph nodes:
    For each key in graph, do:
    Assign table data (columns) to graph[key].data: graph[key].data = tables.find(t =>
    t.name === key).columns
Save graph data
```

This algorithm represents the process of constructing a graph that models a database schema. It outlines the steps to extract table relationships and data from the database, create nodes and edges in the graph, and associate the table data with the graph nodes. The resulting graph provides a comprehensive view of the database structure and relationships, facilitating schema analysis and data migration tasks.

3.5 Data visualization

Constructing the algorithm was facilitated by visualizing the data, which was achieved using D3.js, a prominent library for data visualization. The data and visualization code, as shown in APPENDIX B, involves the following methodology.

The data provided comprises information on tables and their interconnections, represented as a JSON object. The keys signify table names, and the values encompass edge and column details. The asynchronous function **getData** retrieves this JSON data and returns it as a JavaScript object.

Upon obtaining the data, a graph is constructed using D3.js. The graph consists of nodes and links, where nodes are derived from table names and links are established based on relationships between tables as specified in the input data's edges.

The graph's dimensions are set to 600x400 pixels, and a D3 force-directed layout is utilized to position nodes and links. This layout employs several forces: link force, which maintains specified distances between connected nodes; charge force, which repels nodes from each other; and center force, which ensures the graph's center is maintained at designated coordinates.

Subsequently, the code appends SVG elements to represent nodes and links, applying distinct classes and styling attributes. Nodes are depicted as circles with associated text labels, while links are portrayed as lines. Moreover, a tooltip is implemented to display the table name when hovering over a node.

Lastly, an event handler for the simulation's **tick** event updates node and link positions in accordance with the force-directed layout calculations. This event is triggered during each iteration of the simulation, ensuring the graph is accurately rendered and continuously updated.

Example of a few graphs:

Example 1: 2 Graphs (1 Circular, 1 LinkedList)


```
{
  "a": {"edges": [{"table": "b", "column": "a_id", "refColumn": "id"}], "data": [...]},
  "b": {"edges": [], "data": [...]},
  "c": {"edges": [{"table": "d", "column": "c_id", "refColumn": "id"}, {"table": "e", "column": "c_id", "refColumn": "id"}], "data": [...]},
  "d": {"edges": [{"table": "e", "column": "d_id", "refColumn": "id"}], "data": [...]},
  "e": {"edges": [], "data": [...]}
}
```


Figure 4: JSON of a graph (1 Circular, 1 LinkedList)

The provided data is a JSON object representing a schema of tables and their relationships in a database. Each key in the object corresponds to a table name (e.g., "a", "b", "c", "d", "e"), and the associated value is another object containing two properties: "edges" and "data".

"edges": This is an array of objects, each representing a relationship between tables. The object specifies the related table's name ("table"), the name of the column in the current table involved in the relationship ("column"), and the name of the referenced column in the related table ("refColumn").

For example, in table "b", there is an edge indicating a relationship with table "a", where the "id" column in table "b" references the "a_id" column in table "a".

"data": This is an array of objects, each representing a column in the table. Each object contains two properties: "column_name" and "data_type". The "column_name" indicates the name of the column, while the "data_type" specifies the type of data stored in that column (e.g., "integer" or "text").

In summary, the data represents a database schema with table structures and inter-table relationships. Each table is defined by its columns, and the relationships between tables are represented through edges, specifying the connecting columns between them.

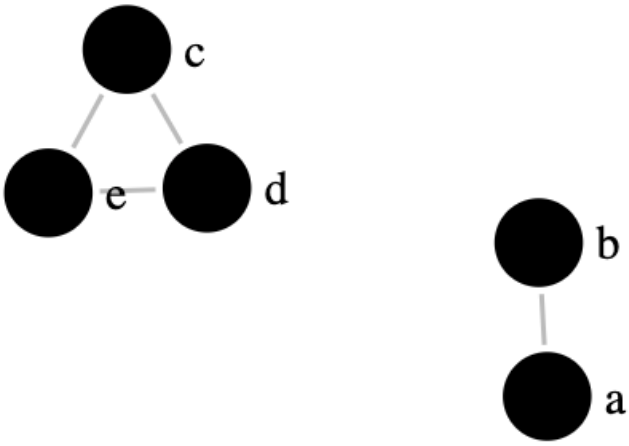


Figure 5: Graph (Figure 4) of a database in D3.js

Example 2: 2 Graphs (2 Linked Lists)


```
{
  "b": {"edges": [{"table": "a", "column": "id", "refColumn": "a_id"}], "data": [...]},
  "a": {"edges": [], "data": [...]},
  "c": {"edges": [{"table": "d", "column": "id", "refColumn": "d_id"}], "data": [...]},
  "d": {"edges": [{"table": "e", "column": "id", "refColumn": "e_id"}], "data": [...]},
  "e": {"edges": [], "data": [...]}
}
```


Figure 6: JSON of a graph

We would get the following graph:

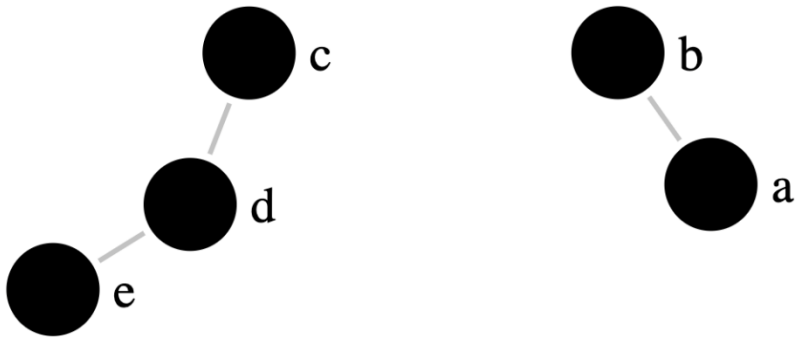


Figure 7: Graph (Figure 6) of a database in D3.js

3.6 Referrencing

To address the type conversion between PostgreSQL and JavaScript, we have implemented a dedicated conversion function that accommodates the most prevalent PostgreSQL data types and maps them to their corresponding JavaScript types. The following table presents a comprehensive illustration of the mapping between these data types:

Table 2: Type conversion between PostgreSQL and MongoDB

PostgreSQL Data Type	MongoDB Data Type
integer	Number
smallint	Number
bigint	Number
decimal	Number
numeric	Number
real	Number
double precision	Number
character varying	String
varchar	String
character	String
char	String
text	String
timestamp without time zone	Date
timestamp with time zone	Date

date	Date
time without time zone	Date
time with time zone	Date
interval	String
boolean	Boolean
bytea	Binary
json	Object
jsonb	Object

The implemented reference conversion algorithm establishes a connection to the designated MongoDB instance, iteratively processes the JavaScript object, and constructs corresponding collections while simultaneously performing type conversion. This approach is based on the aforementioned table, which maps PostgreSQL data types to their equivalent MongoDB counterparts. Given that each column's PostgreSQL data type is already stored, the algorithm facilitates seamless type conversion.

Furthermore, the algorithm generates an index on the primary column of the database, which is typically the identifier (ID) column. This indexing process not only enhances query performance but also ensures efficient data retrieval and management within the MongoDB database. In essence, the reference conversion algorithm serves as an essential tool for the seamless transformation and integration of data between PostgreSQL and MongoDB systems.

In addition to the type conversion and indexing processes, the reference conversion algorithm can be further refined by transforming the identifier (ID) column to an ObjectId data type. This adaptation ensures a more accurate representation of data within MongoDB, as ObjectId is the default identifier data type in MongoDB databases. Consequently, this modification not only improves compatibility between PostgreSQL and MongoDB data structures but also enhances the overall effectiveness of the data transformation process, fostering seamless data integration across these two distinct database systems.

ALGORITHM 2: Graph to Referenced Collection

```
function getColumnDataType(dataType):
  return corresponding MongoDB data type

async function main():
  connect to PostgreSQL and MongoDB servers
  initialize collections, schemas

  for each table in input graph:
    create MongoDB collection for table
    create document schema based on data types

    for each edge in table edges:
      create MongoDB document references
      add index for edge column
```

```
        update document schema with reference info

    write schema definitions to JSON file
    read schema definitions from JSON file

    for each table in schemaJson:
        transferTableData from PostgreSQL to MongoDB

    close connections to PostgreSQL and MongoDB

async function transferTableData(client, tableName, tableSchema):
    create Mongoose schema and model
    fetch data from PostgreSQL
    insert data into MongoDB with new ObjectIds

    for each item:
        replace foreign key values with new ObjectId values
        save item in MongoDB

invoke main() and handle errors
```

During the execution of our experiments, we observed that the use of ObjectId in MongoDB introduced unnecessary complexity to the data migration process. ObjectId is a 12-byte identifier typically used as a unique value for the _id field in MongoDB documents. However, in our case, retaining the original SQL primary key and the MongoDB _id provided a more straightforward approach to preserving the relationships between the entities. By keeping both identifiers, we ensured the data traceability between the two databases and simplified the data mapping process. Furthermore, this approach reduced the risk of data inconsistencies. It allowed for a more efficient data migration, as we did not need to perform additional operations to replace foreign keys with their corresponding ObjectId values. This decision streamlined the data transfer and facilitated a smoother transition between the PostgreSQL and MongoDB databases.

3.7 Embedding

The challenge of embedding documents proved to be complex. Our initial approach involved transforming the given graph into a traversable structure, as specific database scenarios may not have relationships that connect all tables comprehensively. To accomplish this, we developed an algorithm that takes an entry point key, representing a node in one of the graphs, and traverses the graph from that entry node. As it traverses, the algorithm embeds objects and constructs a hierarchical representation:

ALGORITHM 3: Graph to Embedded Schema

```
function makeBidirectionalGraph(graph):
  create bidirectional graph

  for each node in the graph:
    add node and edges to the bidirectional graph
    add node to the edges in the bidirectional graph

  return bidirectional graph

function getMongoSchema(startPoint):
  initialize schema, visited nodes, and queue

  while queue is not empty:
    process node, mark as visited, and add its edges to the queue
    update schema with node information

  return schema

function embedObjects(graph, currentNode):
  if current node has no edges:
    return empty object

  for each edge of the current node:
    recursively embed edge objects in the graph

  return embedded node object

function traverseObject(object, currentPath):
  for each key-value pair in the object:
    update object with data
    recursively traverse object with the updated path

invoke main():
  read data from file and parse into graph
  create bidirectional graph
  create Mongo schema from the graph
  create embedded graph from the schema
  traverse the embedded graph and update with data
  save the updated graph
```

Even with constructing the hierarchy, we encountered a challenge in differentiating between one-to-many and one-to-one relationships, as there is no direct method to ascertain this from a graph of relations. To address this issue, we first built the hierarchy and then employed a query, as shown below:

```
const isOneToOne = async (key, childKey) => {
const forKey =
allData[childKey].edges.find(e => e.table === key).refColumn;
const { rows } =
await pool.query(`select 1 from ${childKey} having count(${forKey}) > 1`);
return !rows.length;
};
```

The query uses a having clause to group the rows by the reference column and count the number of rows in each group. If any group has more than one row, it means that multiple rows in the childKey table reference the same row in the key table, which violates the one-to-one relationship.

This query assists in determining the relationship types and ensures the accurate traversal of the hierarchy. Subsequently, we traverse the hierarchy again, embedding objects as arrays or singular entities and maintaining an additional data column to store the associated data types.

3.8 Alternative Embedding

The approach of selecting entry points for traversal could lead to peculiarly structured or excessively nested objects. Therefore, an alternative method was proposed by leveraging an algorithm from a paper by G. Zhao et al.[3], while integrating our own logic into it. The steps involved in this approach include:

ALGORITHM 4: Graph to Embedded documents.

```
import required modules

define config object with connection strings

function fetchSchemaInfo(pgClient):
  fetch foreign key relationships from PostgreSQL
  extract vertices and edges
  return schemaInfo object

function fetchDataAndInsert(pgClient, mongoClient, schemaInfo):
  calculate foreign key sequence
  for each pair (source, target) in sequence:
    fetch data from source and target tables
    embed target data into source data
```

```
    insert embedded data into MongoDB

function calculateForeignKeySequence(vertices, edges):
    Calculate the order in which the tables should be processed
  Return the order as a sequence of source and target tables

main async function:
  connect to PostgreSQL and MongoDB clients
  fetch schemaInfo
  fetch data and insert into MongoDB
  log successful migration
  close clients and handle errors
```

This algorithm facilitates schema conversion by producing a sequence of foreign key relationships for the input relational database schema. The sequence represents the order in which the relationships should be processed during conversion. By adhering to the sequence generated by the algorithm, the relationships between tables in the relational database can be preserved during the schema conversion to a NoSQL database.

```
function calculateForeignKeySequence(vertices, edges):
  calculate out-degrees and f values for each vertex
  initialize P, Q, T, and S
  while P is not empty:
    remove vertex u from P
    add u to T
    for each edge in f[u]:
      add edge to S
      decrement out-degree of source vertex
      if out-degree of source vertex is 0:
        add source vertex to P and remove from Q
  return S
```

The algorithm processes the input schema by considering the in-degree and out-degree of the tables, representing the number of incoming and outgoing foreign key relationships, respectively. It iteratively selects tables with an out-degree of 0 (no outgoing foreign key relationships) and updates the corresponding in-degree and out-degree values for related tables. In doing so, the algorithm generates a sequence of foreign key relationships (S) that can be employed to guide the schema conversion process.

Upon obtaining the sequence of foreign key relationships, it can be used to design a conversion process that maintains the relationships between tables when converting the schema from a relational database to a NoSQL database. Utilizing sets to store vertices based on their out-degrees and in-degrees (P and Q) and a set to store vertices with both out-degree and in-degree equal to zero (T), the algorithm ensures

that the converted schema preserves the integrity of the relationships in the original schema while adapting it to the target NoSQL database format.

3.9 Dataflow

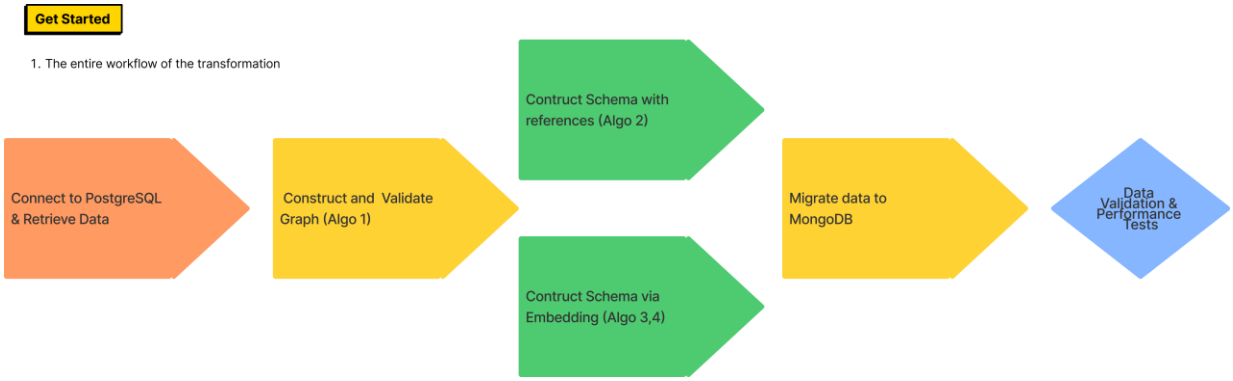


Figure 8: Dataflow of the transformation

Our methodology for transitioning from a PostgreSQL database to a MongoDB database involves the following steps:

1. Connect to the PostgreSQL database and retrieve the necessary data to construct a graph representation of the relationships among the data entities.
2. Validate the generated graph's accuracy by examining the corresponding JSON representation or visualizing it using the D3.js library.

Upon successful validation, the process can proceed in one of two ways:

Option 1: a. Use Algorithm 2 to transform the given graph into a MongoDB reference-based schema, enabling efficient referencing among collections. b. Migrate the data from the SQL tables to the corresponding MongoDB collections, preserving the relationships defined in the reference-based schema.

Option 2: a. Employ either Algorithm 3 or 4 to transform the given graph into a MongoDB embedded-style schema, promoting data locality and potentially enhancing query performance. b. Migrate the data from the SQL tables to the MongoDB collections, ensuring the embedded-style relationships are maintained.

By following these steps, we can effectively transition from a PostgreSQL database to a MongoDB database while preserving the integrity of the data and optimizing the schema for specific use cases.

4 RESEARCH RESULTS AND ANALYSIS OF RESULTS

In this section, we present the findings of our research, which focused on developing and implementing a graph-based algorithm for converting relational database schemas to NoSQL formats. The primary objective of our research was not to optimize performance but to ensure data accuracy and maintain relationships between tables during the conversion process. Nevertheless, we conducted performance tests to gain insights into the algorithm's efficiency.

Through various scenarios and test cases, we have successfully demonstrated the potential of our algorithm as a foundation for database conversion tools. The research results highlight the versatility and usefulness of the proposed algorithm in assisting organizations in transitioning from traditional relational databases to modern NoSQL databases. In the following subsections, we provide a detailed overview of our research results, including the progress made, benchmarks achieved, and an analysis of the results to showcase the algorithm's performance regarding data accuracy and its impact on schema conversion.

4.1 Experiment 1: Single table conversion

During our experiment, we generated a single table, designated as Table A, and populated it with a set of fields. The table creation was accomplished through SQL queries, each encompassing three unique columns. The following SQL statements were employed to create and populate Table A:

```
CREATE TABLE a (id SERIAL PRIMARY KEY, name TEXT, age INTEGER, email TEXT);
INSERT INTO a
VALUES
(DEFAULT, 'John Smith', 32, 'john.smith@example.com'),
(DEFAULT, 'Jane Doe', 27, 'jane.doe@example.com'),
(DEFAULT, 'Bob Johnson', 45, 'bob.johnson@example.com');
```

The generated table was visually represented in a graphical user interface (GUI), as depicted below:

id	name	age	email
1	John Smith	32	john.smith@example.com
2	Jane Doe	27	jane.doe@example.com
3	Bob Johnson	45	bob.johnson@example.com

Figure 9: Experiment 1 SQL Table

After the graph conversion process, the resulting JSON object was obtained:

Figure 10 Experiment 1 Graph in JSON

Following the completion of the graph conversion, we executed our referencing algorithm, which yielded the subsequent JSON object:

Figure 11 Experiment 1 Graph in Mongo Schema

In order to enforce schema consistency and take advantage of additional development tools, we employed the Mongoose library for MongoDB, as MongoDB does not enforce schemas by default. Given that our generated schema JSON file closely resembled the actual schema object, Mongoose was able to efficiently infer the schema structure and facilitate the data transfer process.

The final representation of the data in the MongoDB UI is displayed below:

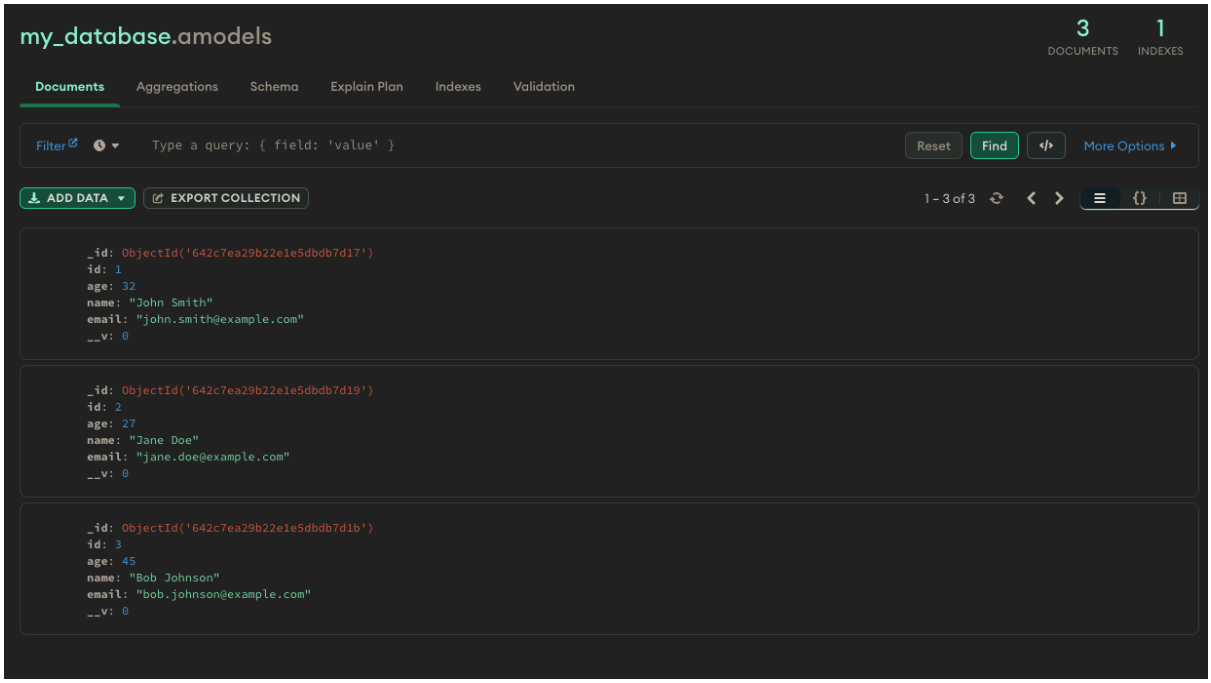


Figure 12 Experiment 1 Mongo collection

The preliminary single-table transformation and data migration demonstrated high accuracy with respect to data integrity and data type preservation, yielding favorable outcomes across all experimental parameters. Consequently, it is now feasible to progress towards more intricate structures encompassing multiple tables.

4.2 Expriment 2: Multi table transformation via referencing

In the subsequent phase, five tables were generated, and data was inserted into them utilizing SQL queries detailed in Appendix G. These tables exhibit interrelations among them:

Table from A-E:

id	name	age	email
1	John Smith	32	john.smith@example.com
2	Jane Doe	27	jane.doe@example.com
3	Bob Johnson	45	bob.johnson@example.com

id	title	a_id
1	Title 1	1
2	Title 2	2
3	Title 3	3

id	description	a_id
1	Description 1	1
2	Description 2	2
3	Description 3	3

id	value	b_id
1	100	1
2	200	2
3	300	3

id	comment	c_id
1	Comment 1	1
2	Comment 2	2
3	Comment 3	3

Figure 13:Generated Tables a, b, c, d, e and their data in top to bottom order

The graph transformation and type conversion were articulated in a more compact JSON format as follows:


```
{
  "b": { "edges": [{ "table": "a", "column": "id", "refColumn": "a_id" }], "data": [{...}, {...}, {...}] },
  "a": { "edges": [], "data": [{...}, {...}, {...}, {...}] },
  "c": { "edges": [{ "table": "a", "column": "id", "refColumn": "a_id" }], "data": [{...}, {...}, {...}] },
  "d": { "edges": [{ "table": "b", "column": "id", "refColumn": "b_id" }], "data": [{...}, {...}, {...}] },
  "e": { "edges": [{ "table": "c", "column": "id", "refColumn": "c_id" }], "data": [{...}, {...}, {...}] }
}
```


Figure 14: Experiment 2 Graph in JSON

Schema conversion:

Figure 15: Experiment 2 Mongo Schema

Upon the successful completion of data migration, the following results were obtained:

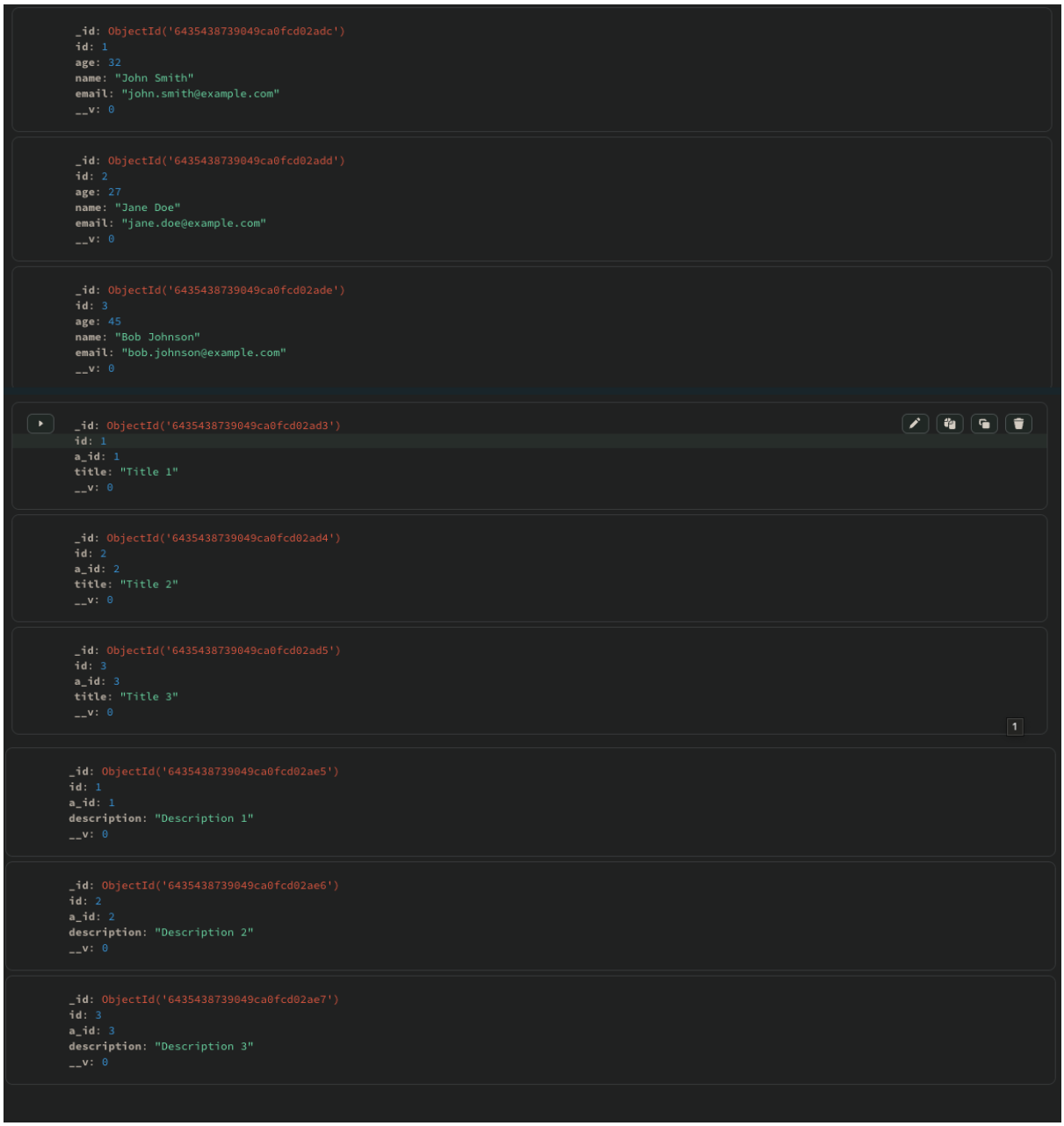


Figure 16: Generated collections a,b,c,d,e and their data in top to bottom order

The referenced collections functioned effectively, and a series of query tests were conducted, encompassing join operations, single cluster joins specific to MongoDB, and software-level joins for distributed systems. These tests were performed to evaluate the accuracy and benchmark performance, the details of which will be elucidated in subsequent sections.

4.3 Experiment 3: Multi table transformation via embedding

To facilitate more straightforward testing during the embedding process, we opted for improved naming conventions rather than using letters ‘a’ through ‘e’. The table generation queries are available in Appendix F, with the tables and their relationships as follows:

- Author
- Publisher
- Category
- Book (related to Author and Publisher)
- BookCategory (related to Book and Category)

The resulting tables in SQL:

id	name
1	John Doe
2	Jane Smith
3	Alice Johnson

id	name
1	Fiction
2	Non-fiction
3	Science Fiction

3626 rows

id	title	author_id	publisher_id
1	The Great Novel	1	1
2	Fact or Fiction	2	2
3	Mars Colony	3	3

id	book_id	category_id
1	1	1
2	2	2

id	name
1	Penguin Publishing
2	HarperCollins
3	Random House

Figure 17: tables listed in alphabetical order from left to right and top to bottom.

Our graph representation is as follows:

```
{
  "vertices": ["book", "author", "publisher", "bookcategory", "category"],
  "edges": [
  {"source": "book", "target": "author", "key": "author_id"},
  {"source": "book", "target": "publisher", "key": "publisher_id"},
  {"source": "bookcategory", "target": "book", "key": "book_id"},
  {"source": "bookcategory", "target": "category", "key": "category_id"}
]
}
```

Figure 18:Experiment 3 Graph in JSON

After executing the algorithm and the code provided in Appendix E, we obtained the following results:

Figure 19: Results of embedding algorithm 4

As previously stated, manual validation of data integrity was conducted, along with a series of test queries to evaluate both accuracy and performance.

4.4 Performance Benchmarks

After the initial data conversion and schema generation, we conducted a series of tests to compare the performance of PostgreSQL and MongoDB in terms of query execution. The test code, implemented using JavaScript, involved connecting to both PostgreSQL and MongoDB databases, defining the Mongoose schema, and executing a set of queries using the ***runQuery*** function. The queries were designed to cover various typical database operations, such as selection, sorting, and updating records.

The benchmark results were gathered by calculating the execution time of each query for both PostgreSQL and MongoDB. The tests were conducted multiple times to ensure the reliability and consistency of the results. The outcomes demonstrated varying performance characteristics across different query types. In some cases, PostgreSQL exhibited faster execution times, while in others, MongoDB demonstrated superior performance.

The following summary presents the key findings from the benchmark tests:

Select all records with age > 30:
PostgreSQL consistently outperformed MongoDB in this query, with faster execution times.

Select specific fields (name, email) with age > 30:
MongoDB generally performed better in this query, with shorter execution times compared to PostgreSQL.

Sort by age and select top 5 records:
Although the performance difference was minimal, MongoDB typically executed this query faster than PostgreSQL.

Update the email of a specific id:
MongoDB consistently demonstrated faster execution times when updating records, in comparison to PostgreSQL.

These results provide valuable insights into the comparative performance of PostgreSQL and MongoDB databases under specific query scenarios. By analyzing these findings, developers can make informed decisions when selecting the appropriate database system for their specific use case and requirements.

To get a chart of the graph we used a data generation query to have enough data to perform some tests.

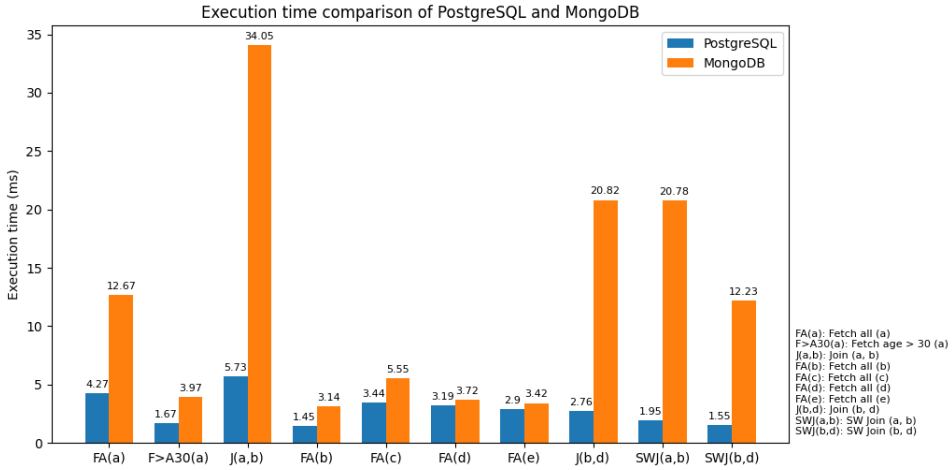
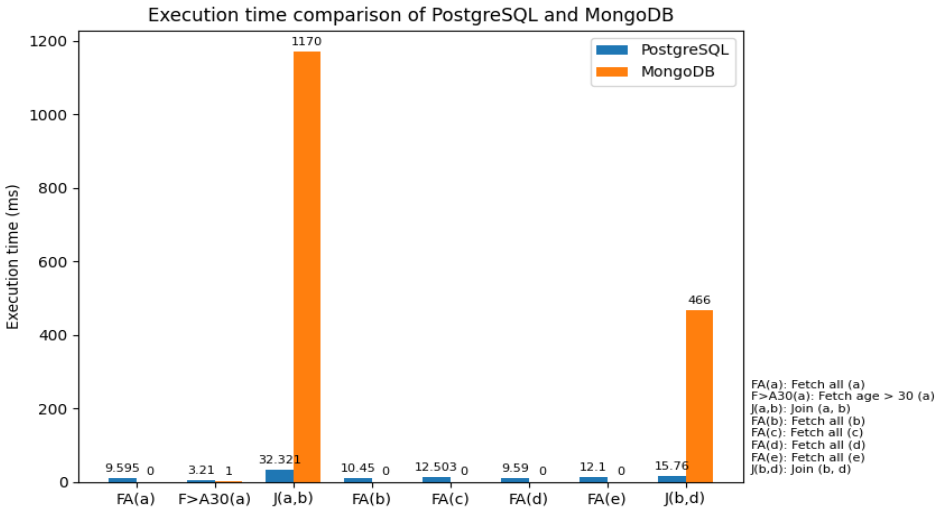


Figure 20: Performance Comparison via JavaScript drivers

The results on larger databases (10k) gave us the chart as seen as above on Figure 20.

However the results did not make much sense nor did they align with other research papers[12][13][14] where almost in all cases MongoDB outperformed the competting SQL database. We assume this due to the extra time gained over Node.JS and libraries used to generate the queries.

To obtain more accurate results, we decided to use database-level queries using the shell CLIs psql and mongosh. When we ran the queries using mongosh and retested them, we consistently saw 0ms in MongoDB and 3-22ms in PostgreSQL for datasets below 15k records. When we retested the queries with 10-15k records using the shell, we observed that MongoDB was more optimal for simple tasks and that PostgreSQL was slower for joins.



4.5 Testing for accuracy and performance

The following tests have been in 2 steps, one with fewer data to test for data accuracy and then we bulk inserted generated data to test for performance, since tesing below 10k records would result in 0ms on mongosh execution time.

4.5.1 Experiment 4:Query testing on single table conversion

The following queries find records from table/collection “a” where age is greater than 30:

```
SELECT * FROM a WHERE age>30
```

```
{age:{$gt:30}}
```

id	name	age	email
1	John Smith	32	john.smith@example.com
3	Bob Johnson	45	bob.johnson@example.com

Figure 21:Table a where age is greater than 30.

Figure 22:Collection a where age is greater than 30.

As expected the query logic on single table produces the same data.

4.5.2 Experiment 5:Query on multi table conversion via references

For multi table the main breaking point could be getting data from differet tables/collections, so we tested precisely for that. The following queries were used.

```
SELECT a.*, b.*
FROM a
JOIN b
ON a.id = b.a_id


Mongo aggregation:
[
  {
    $lookup: {
      from: "bmodels",
      localField: "id",
      foreignField: "a_id",
      as: "b_records",
    },
  },
  { $unwind: "$b_records" },
  {
    $project: {
      _id: 0,
      id: 1,
      name: 1,
      age: 1,
      email: 1,
      "b_records.id": 1,
      "b_records.title": 1,
      "b_records.a_id": 1,
    },
  },
```

```
];
```

The explanation of the following aggregate functions:

Stage 1: $lookup
This operator performs a left outer join on the specified collection, in this case, 'bmodels'. The localField parameter, 'id', is matched with the foreignField parameter, 'a_id', and the resulting joined documents are stored in a new array field, 'b_records'. This stage allows combining data from two separate collections, based on a specified matching condition.

Stage 2: $unwind
This operator is responsible for deconstructing the 'b_records' array field. The operator creates a separate document for each element in the 'b_records' array. The path parameter is set to '$b_records', signifying the array to deconstruct. This stage effectively normalizes the data, making it easier to work with and manipulate in subsequent stages.

Stage 3: $project

This operator is designed to either include or exclude specified fields from the output documents, effectively shaping the structure of the result. In this stage, the '_id' field is set to 0 to exclude it from the output, while the 'id', 'name', 'age', 'email', 'b_records.id', 'b_records.title', and 'b_records.a_id' fields are all set to 1, ensuring their inclusion in the output. This stage allows for the customization of the output data, making it easier to consume and analyze.

id	name	age	email	id	title	a_id
1	John Smith	32	john.smith@example.com	1	Title 1	1
2	Jane Doe	27	jane.doe@example.com	2	Title 2	2
3	Bob Johnson	45	bob.johnson@example.com	3	Title 3	3
1	John Smith	32	john.smith@example.com	4	Title 1	1
2	Jane Doe	27	jane.doe@example.com	5	Title 2	2
3	Bob Johnson	45	bob.johnson@example.com	6	Title 3	3

Figure 23: Join results of SQL database of

Figure 24: Result of aggregate joins in Mongo

4.5.3 Experiment 5: Query and performance testing on multi table conversion via embedding

In this experiment we already showed the accuracy results in Experiment 3, thus we decided to bulk insert and test for performance. This is the most common use of NoSQL databases. This was done using the following queries.

```
SELECT
b.id AS book_id,
b.title AS book_title,
b.author_id,
a.id AS author_id,
a.name AS author_name,
b.publisher_id,
p.id AS publisher_id,
p.name AS publisher_name
FROM book b
INNER JOIN author a
ON b.author_id = a.id
INNER JOIN publisher p O
N b.publisher_id = p.id;
```

Since MongoDB was embedded there is no need for extra aggregations, we can return the document. We bulk inserted around 7k documents since below this we would always get 0ms due to mongosh explain query does not show decimals like SQL's explain and analyze queries do.

The bulk insert results were the following.

id ▲	title	author_id	publisher_id
1	The Great Novel	1	1
2	Fact or Fiction	2	2
3	Mars Colony	3	3
4	Detective Stories	1	2
5	Love in the Time of AI	2	1
6	Book 1	2	2
7	Book 2	3	3
8	Book 3	4	4
9	Book 4	5	5
10	Book 5	6	6
11	Book 6	7	7
12	Book 7	8	8
13	Book 8	9	9
14	Book 9	10	10
15	Book 10	11	11
16	Book 11	12	12
17	Book 12	13	13
18	Book 13	14	14
19	Book 14	15	15
20	Book 15	16	16
21	Book 16	17	17
22	Book 17	18	18

Figure 25: Bulk insert table of SQL

The first few were inserted for accuracy testing and the rest were generated. The migration process took about 7-12 seconds to migrate and embed 7252 documents. The resulting mongo collection is shown in figure 26.

book

	_id ObjectId	id Int32	title String	author Object	publisher Object
1	ObjectId('643c1013c33095e8e…	1	"The Great Novel"	{} 2 fields	{} 2 fields
2	ObjectId('643c1013c33095e8e…	2	"Fact or Fiction"	{} 2 fields	{} 2 fields
3	ObjectId('643c1013c33095e8e…	3	"Mars Colony"	{} 2 fields	{} 2 fields
4	ObjectId('643c1013c33095e8e…	4	"Detective Stories"	{} 2 fields	{} 2 fields
5	ObjectId('643c1013c33095e8e…	5	"Love in the Time of AI"	{} 2 fields	{} 2 fields
6	ObjectId('643c1013c33095e8e…	6	"Book 1"	{} 2 fields	{} 2 fields
7	ObjectId('643c1013c33095e8e…	7	"Book 2"	{} 2 fields	{} 2 fields
8	ObjectId('643c1013c33095e8e…	8	"Book 3"	{} 2 fields	{} 2 fields
9	ObjectId('643c1013c33095e8e…	9	"Book 4"	{} 2 fields	{} 2 fields
10	ObjectId('643c1013c33095e8e…	10	"Book 5"	{} 2 fields	{} 2 fields
11	ObjectId('643c1013c33095e8e…	11	"Book 6"	{} 2 fields	{} 2 fields
12	ObjectId('643c1013c33095e8e…	12	"Book 7"	{} 2 fields	{} 2 fields
13	ObjectId('643c1013c33095e8e…	13	"Book 8"	{} 2 fields	{} 2 fields
14	ObjectId('643c1013c33095e8e…	14	"Book 9"	{} 2 fields	{} 2 fields
15	ObjectId('643c1013c33095e8e…	15	"Book 10"	{} 2 fields	{} 2 fields
16	ObjectId('643c1013c33095e8e…	16	"Book 11"	{} 2 fields	{} 2 fields
17	ObjectId('643c1013c33095e8e…	17	"Book 12"	{} 2 fields	{} 2 fields
18	ObjectId('643c1013c33095e8e…	18	"Book 13"	{} 2 fields	{} 2 fields
19	ObjectId('643c1013c33095e8e…	19	"Book 14"	{} 2 fields	{} 2 fields
20	ObjectId('643c1013c33095e8e…	20	"Book 15"	{} 2 fields	{} 2 fields

Figure 26: Mongo book collection first 20 entries

Removing cache in PostgreSQL:

```
sync;
sudo service postgresql stop;
echo 1 > /proc/sys/vm/drop_caches;
sudo service postgresql start
```

The performance result on SQL without caching:

Planning Time: 3.720 ms
Execution Time: 19.026 ms

The performance of a mongo query.

```
{
 “stage”: “COLLSCAN”,
 “nReturned”: 7252,
 “executionTimeMillisEstimate”: 1,
 “works”: 7254,
 “advanced”: 7252,
 “needTime”: 1,
 “needYield”: 0,
 “saveState”: 7,
 “restoreState”: 7,
 “isEOF”: 1,
 “direction”: “forward”,
 “docsExamined”: 7252
}
```

The field we need is “executionTimeMillisEstimate”.
As we can see now the results are in line with the other paper showing a large gap in read performance in favor of MongoDB.

5 SUMMARY AND FUTURE WORK

In conclusion, the proposed algorithm has demonstrated its potential in converting relational database schemas to NoSQL formats. Throughout the course of this research, several key achievements have been made, which include:

1. Developing a novel methodology that leverages graphs for schema conversion, enabling the algorithm to efficiently analyze relationships between database tables and create a structured representation of the schema in the target NoSQL format.
2. Successfully creating one of the first tools for database conversion using graph-based analysis, which has the potential to serve as a foundation for future research and development in this area.
3. Exploring various techniques for referencing and embedding data, which played a crucial role in maintaining data integrity and relationships during the conversion process.
4. Demonstrating the usefulness of the algorithm in various scenarios, highlighting its potential to aid organizations in migrating from traditional relational databases to modern NoSQL databases, which can offer advantages such as scalability, flexibility, and adaptability to changing data requirements.

While the developed algorithm has shown promising results, there are several areas of improvement and exploration that can be pursued in future research:

1. Edge cases: The algorithm can be fine-tuned to handle edge cases more effectively, such as self-referential relationships and circular dependencies. These adjustments would allow for more robust schema conversion in complex database structures. Research into these edge cases could lead to innovative solutions that further enhance the algorithm's capabilities.
2. Support for different database paradigms: The algorithm could be adapted and extended to work with other NoSQL databases, such as Cassandra and Neo4j. This would involve considering each database system's unique features and constraints and adjusting the algorithm accordingly. Developing support for additional NoSQL databases would broaden the algorithm's applicability and potential impact.
3. NoSQL to NoSQL transformations: Future research could also explore the possibility of converting schemas between different NoSQL databases. Such transformations would enable organizations to transition more easily between various NoSQL systems as their needs evolve.
4. Distributed systems: The current algorithm could be tested and optimized for distributed systems, where the data is partitioned and replicated across multiple nodes. This would involve addressing consistency, availability, and partition tolerance challenges in the schema conversion process. Research in this area could lead to new techniques for handling distributed data during schema conversion.
5. Performance optimization: While the current focus of the research was not on performance, there is potential for optimizing the algorithm to improve its efficiency and reduce execution time. This could be achieved through various techniques like parallel processing and caching. As the demand for large-scale data processing grows, performance optimization will become increasingly important.
6. Scalability: The algorithm could be tested and refined for large-scale databases to assess its performance and scalability in handling vast data and relationships. This would provide insights into the algorithm's limitations and potential bottlenecks, enabling further refinement to accommodate the growing needs of data-intensive applications.
7. Comprehensive testing: Rigorous testing of the algorithm with diverse datasets and varying database configurations would help identify any shortcomings or potential issues, enabling

further refinement of the algorithm and ensuring its reliability in different situations. Systematic testing and evaluation could provide valuable insights into the algorithm's performance and behavior in real-world scenarios.

8. Additional performance testing with varying numbers of documents would be beneficial in understanding the effect of keeping the document in memory instead of swapping in the segments of the table, as done by PostgreSQL. By conducting such tests, you could gain insights into how the performance of MongoDB and PostgreSQL changes with different data sizes and access patterns. This information could help optimize your database implementation for specific use cases and provide a deeper understanding of the underlying performance characteristics of these database systems.

By addressing these areas for improvement and exploration, the algorithm can be further refined and made more versatile, catering to a broader range of database structures and systems. This research has laid the groundwork for future advancements in the field of database schema conversion, with the potential to revolutionize how organizations manage and adapt their data infrastructure.

6 BIBLIOGRAPHY

1. Ghotiya, Sunita & Mandal, Juhi & Kandasamy, Saravanakumar. (2017). Migration from relational to NoSQL database. IOP Conference Series: Materials Science and Engineering. 263. 042055. 10.1088/1757-899X/263/4/042055.
2. N. I. Abo Dabowsa, A. M. Maatuk, S. M. Elakeili and M. Akhtar Ali, "Converting Relational Database to Document-Oriented NoSQL Cloud Database," 2021 IEEE 1st International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering MI-STA, Tripoli, Libya, 2021, pp. 381-386, doi: 10.1109/MI-STA52233.2021.9464488
3. G. Zhao, Q. Lin, L. Li and Z. Li, "Schema Conversion Model of SQL Database to NoSQL," 2014 Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, Guangdong, China, 2014, pp. 355-362, doi: 10.1109/3PGCIC.2014.137.
4. C. Hadjigeorgiou, "RDBMS vs NoSQL: Performance and Scaling Comparison," M.S. thesis, The University of Edinburgh, Edinburgh, UK, 2013
5. S. Palanisamy and P. SuvithaVani, "A survey on RDBMS and NoSQL Databases MySQL vs MongoDB," 2020 International Conference on Computer Communication and Informatics (ICCCI), 2020, pp. 1-7, doi: 10.1109/ICCCI48352.2020.9104047.
6. "CAP Theorem." A Dictionary of Computer Science, 7th ed., Oxford University Press, 2016.
7. S. Gilbert and N. Lynch, "Perspectives on the CAP Theorem," in Computer, vol. 45, no. 2, pp. 30-36, Feb. 2012, doi: 10.1109/MC.2011.389.
8. D. Bermbach and S. Tai, "Eventual Consistency: How Soon Is Eventual? An Evaluation of Amazon S3's Consistency Behavior," in Proceedings of the 6th Workshop on Middleware for Service Oriented Computing, ACM, 2011, pp. 1-6, doi: 10.1145/2093334.2093344.
9. McCreary, D. et al. (2014) Making sense of NoSQL : a guide for managers and the rest of us. 1st edition. Shelter Island, New York: Manning Publications.
10. R. Copeland, "MongoDB Applied Design Patterns," 1st ed., O'Reilly Media, 2013.
11. Ambler, J. (2012). NoSQL distilled: A brief guide to the emerging world of polyglot persistence. Addison-Wesley Professional.
12. A. Yassien and A. Desouky, "RDBMS, NoSQL, Hadoop: A Performance-Based Empirical Analysis," in Proceedings of the 2nd Africa and Middle East Conference on Software Engineering, ACM, 2016, pp. 52–59.
13. Y. Li and S. Manoharan, "A performance comparison of SQL and NoSQL databases," in 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), 2013, pp. 15–19, doi: 10.1109/PACRIM.2013.6625441.
14. S. Kontogiannis, C. Asiminidis, and G. Kokkonis, "Comparing Relational and NoSQL Databases for carrying IoT data," The Journal of Scientific and Engineering Research, vol. 6, no. 2, 2019, pp. 1–10.

7 APPENDIX A – SCHEMA TO GRAPH

```
const { Pool } = require('pg');
const util = require('util');
const fs = require('fs');
const path = require('path');
const connectionString =
'postgres://postgres:mysecretpassword@localhost:5432/postgres?sslmode=disable';
const pool = new Pool({ connectionString });

const getAllTablesQuery = `SELECT * FROM information_schema.tables WHERE
table_schema = 'public';`;

const getData = (table) => {
  return `SELECT column_name, data_type FROM information_schema.columns WHERE
table_name = '${table}';`;
};


const query = `
  SELECT tc.table_name, kcu.column_name, ccu.table_name AS foreign_table_name,
         ccu.column_name AS foreign_column_name
  FROM information_schema.table_constraints tc
  JOIN information_schema.key_column_usage kcu
    ON tc.constraint_name = kcu.constraint_name
    AND tc.table_schema = kcu.table_schema
  JOIN information_schema.constraint_column_usage ccu
    ON ccu.constraint_name = tc.constraint_name
    AND ccu.table_schema = tc.table_schema
  WHERE tc.constraint_type = 'FOREIGN KEY'
  ORDER BY tc.table_name, kcu.ordinal_position;
`;

async function saveData(data) {
  try {
    const jsonString = JSON.stringify(data);
    const filePath = path.join(__dirname, 'data.json');
    await fs.promises.writeFile(filePath, jsonString);
    console.log('Data saved to file.');
  } catch (err) {
    console.error(err);
  }
}
```

```
const main = async () => {
  try {
    const { rows } = await pool.query(query);
    const graph = {};
    rows.forEach((row) => {
      const tableName = row.table_name;
      const columnName = row.column_name;
      const foreignTableName = row.foreign_table_name;
      const foreignColumnName = row.foreign_column_name;

      if (!graph[tableName]) {
        graph[tableName] = { edges: [] };
      }

      if (!graph[foreignTableName]) {
        graph[foreignTableName] = { edges: [] };
      }

      graph[tableName].edges.push({
        table: foreignTableName,
        column: foreignColumnName,
        refColumn: columnName,
      });
    });

    const res = await pool.query(getAllTablesQuery);
    const tableNames = res.rows.map((row) => row.table_name);
    const tables = [];

    for (const table of tableNames) {
      const { rows } = await pool.query(getData(table));
      const columns = rows.map(({ column_name, data_type }) => ({
        column_name,
        data_type,
      }));
      tables.push({
        name: table,
        columns,
      });
    }

    pool.end();
```

```
    if (tables.length == 1) {
      const onlyTable = tables[0]
      saveData({
        [onlyTable.name]: {
          edges: [],
          data: onlyTable.columns
        }
      })
      return
    }

    for (const key in graph) {
      graph[key].data = tables.find(t => t.name === key).columns
    }

    saveData(graph)

  } catch (error) {
    console.log(error);
  }
};
```

8 APPENDIX B – GRAPH VISUALIZATION

```
async function getData() {
  try {
    const response = await fetch("data.json");
    const data = await response.json();
    return data;
  } catch (error) {
    console.error(error);
  }
}

getData().then(str => {
  const graph = str;
  const nodes = Object.keys(graph).map(tableName => ({ id: tableName }));
  const links = [];

  nodes.forEach(node => {
    graph[node.id].edges.forEach(edge => {
      links.push({
        source: node.id,
        target: edge.table,
        refColumn: edge.refColumn
      });
    });
  });

  const width = 600;
  const height = 400;
  const svg = d3.select("body").append("svg").attr("width",
width).attr("height", height);
  const simulation = d3.forceSimulation(nodes)
    .force("link", d3.forceLink(links).id(d => d.id))
    .force("charge", d3.forceManyBody())
    .force("center", d3.forceCenter(width / 2, height / 2));

  const node =
svg.selectAll(".node").data(nodes).enter().append("g").attr("class",
"node").call(drag(simulation));
```

```
  node.append("circle").attr("r", 10)
    .on("mouseover", function(event, d) {
      tooltip.html(`<strong>${d.id}</strong>`).style("opacity",
1).style("left", event.pageX + 10 + "px").style("top", event.pageY - 20 +
"px");
      d3.select(this).attr("r", 15);
    })
    .on("mouseout", function(event, d) {
      tooltip.style("opacity", 0);
      d3.select(this).attr("r", 10);
    });

  node.append("text").text(d => d.id).attr("dx", 12).attr("dy", ".35em");

  const tooltip = d3.select("body").append("div").attr("class", "tooltip");
});
```

9 APPENDIX C – REFERENCE SCHEMA CREATION

```
const { MongoClient } = require('mongodb');
const graph = require('./data.json');
const fs = require('fs');
const url = 'mongodb://localhost:27017';
const client = new MongoClient(url);

function getColumnDataType(dataType) {
  switch (dataType) {
    case 'integer':
      return 'Number';
    case 'numeric':
      return 'Number';
    case 'character varying':
      return 'String';
    case 'text':
      return 'String';
    case 'timestamp without time zone':
      return 'Date';
  }
}

async function main() {
  await client.connect();
  const db = client.db("thesis");
  const collections = {};
  const schemas = {};

  for (const [tableName, tableData] of Object.entries(graph)) {
    const edges = tableData.edges;
    const data = tableData.data;
    const collection = db.collection(tableName);
    collections[tableName] = collection;

    const schema = {};
    for (const column of data) schema[column.column_name] =
getColumnDataType(column.data_type);
    schemas[tableName] = schema;

    for (const edge of edges) {
      collection.createIndex({ [edge.column]: 1 });
      schema[edge.column] = { type: 'ObjectId', ref: edge.table };
    }
  }

  const schemaString = JSON.stringify(schemas, null, 2);
  fs.writeFileSync('schema.json', schemaString);
}
main().catch(console.error);
```

10 APPENDIX D – RERENCE SCHEMA TO MONGODB DATA TRANFER

```
const fs = require('fs');
const { Pool } = require('pg');
const mongoose = require('mongoose');
const { Schema } = mongoose;

const connectionString =
'postgres://postgres:mysecretpassword@localhost:5432/postgres?sslmode=disab
le';

const pool = new Pool({ connectionString });

const mongoConnectionString = 'mongodb://localhost:27017/my_database';
mongoose.connect(mongoConnectionString, { useNewUrlParser: true,
useUnifiedTopology: true });
mongoose.set('strictQuery', true);

const idMap = new Map();
fs.readFile('schema.json', 'utf8', async (err, data) => {
  if (err) {
    console.error(err);
    return;
  }

  const schemaJson = JSON.parse(data);

  try {
    const client = await pool.connect();
    await mongoose.connection.db.dropDatabase();

    for (const tableName in schemaJson) {
      await transferTableData(client, tableName, schemaJson[tableName]);
    }

    client.release();
  } catch (error) {
    console.error(error);
  } finally {
    mongoose.connection.close();
  }
});
```

```
async function transferTableData(client, tableName, tableSchema) {
  const schema = new Schema(tableSchema);
  const Model = mongoose.model(`${tableName}Model`, schema);

  const result = await client.query(`SELECT * FROM ${tableName}`);
  const rows = result.rows;
  const newItems = [];

  for (const row of rows) {
    const newItemData = Object.assign({}, row);
    const oldId = row.id;
    newItemData._id = mongoose.Types.ObjectId();
    if (!idMap.has(tableName)) idMap.set(tableName, new Map());
    idMap.get(tableName).set(oldId, newItemData._id);
    newItems.push(newItemData);
  }

  for (const newItemData of newItems) {
    const newItem = new Model(newItemData);
    await newItem.save();
  }
}
```

11 APPENDIX E – EMBEDDING ALGORITHM

```
const { Client } = require('pg');
const { MongoClient } = require('mongodb');

const config = {
  pgConnectionString:
'postgres://postgres:mysecretpassword@localhost:5432/postgres?sslmode=disab
le',
  mongoUri: 'mongodb://localhost:27017',
  mongoDbName: 'embed',
};

// Connect to PostgreSQL
const pgClient = new Client({ connectionString: config.pgConnectionString
});

// Connect to MongoDB
const mongoClient = new MongoClient(config.mongoUri);

async function fetchSchemaInfo(pgClient) {
  try {
    const res = await pgClient.query(`
      SELECT
        tc.table_schema,
        tc.constraint_name,
        tc.table_name,
        kcu.column_name,
        ccu.table_schema AS foreign_table_schema,
        ccu.table_name AS foreign_table_name,
        ccu.column_name AS foreign_column_name
      FROM
        information_schema.table_constraints AS tc
        JOIN information_schema.key_column_usage AS kcu ON
tc.constraint_name = kcu.constraint_name
        JOIN information_schema.constraint_column_usage AS ccu ON
ccu.constraint_name = tc.constraint_name
      WHERE constraint_type = 'FOREIGN KEY';
    `);
```

```
    const vertices = new Set();
    const edges = res.rows.map(row => {
      vertices.add(row.table_name);
      vertices.add(row.foreign_table_name);
      return {
        source: row.table_name,
        target: row.foreign_table_name,
        key: row.column_name,
      };
    });

    return { vertices: Array.from(vertices), edges };
  } catch (err) {
    console.error('Error fetching schema info', err);
    throw err;
  }
}
```

```
async function fetchDataAndInsert(pgClient, mongoClient, schemaInfo) {
  const foreignKeySequence =
calculateForeignKeySequence(schemaInfo.vertices, schemaInfo.edges);
  try {
    const db = mongoClient.db(config.mongoDbName);

    for (const { source } of foreignKeySequence) {
      const sourceData = await pgClient.query(`SELECT * FROM ${source}`);

      const embeddedDataPromises = sourceData.rows.map(async sourceRow => {
        const result = { ...sourceRow };

        // Iterate over each edge
        for (const edge of schemaInfo.edges) {
          // Check if the current edge's source matches the source table
          if (edge.source === source) {
            try {
              const targetData = await pgClient.query(`SELECT * FROM
${edge.target} WHERE id = $1`, [sourceRow[edge.key]]);
              if (targetData.rows.length > 0) {
                result[edge.target] = targetData.rows[0];
                delete result[edge.key];
              }
            } catch (err) {
              console.error(`Error fetching data from ${edge.target}`,
err);
            }
          }
        }

        return result;
      });

      // Wait for all promises to resolve before inserting the data
      const completedData = await Promise.all(embeddedDataPromises);
      await db.collection(source).insertMany(completedData);
    }
  } catch (err) {
    console.error('Error fetching data and inserting', err);
    throw err;
  }
}
```

```
function calculateForeignKeySequence(vertices, edges) {
  const outDegree = {};
  const f = {};
  vertices.forEach(vertex => {
    outDegree[vertex] = edges.filter(edge => edge.source ===
vertex).length;
    f[vertex] = edges.filter(edge => edge.target === vertex);
  });

  let P = vertices.filter(vertex => outDegree[vertex] === 0);
  let Q = vertices.filter(vertex => outDegree[vertex] !== 0);
  let T = new Set();
  let S = [];

  while (P.length !== 0) {
    const u = P.pop();
    T.add(u);
    f[u].forEach(edge => {
      S.push({ source: edge.source, target: edge.target });
      outDegree[edge.source]--;
      if (outDegree[edge.source] === 0) {
        P.push(edge.source);
        Q = Q.filter(vertex => vertex !== edge.source);
      }
    });
  }

  return S;
}

(async () => {
  try {
    await pgClient.connect();
    await mongoClient.connect();

    const schemaInfo = await fetchSchemaInfo(pgClient);
    await fetchDataAndInsert(pgClient, mongoClient, schemaInfo);

    console.log('Data successfully migrated from PostgreSQL to MongoDB');
  } catch (err) {
    console.error('Error during migration', err);
  } finally {
    await pgClient.end();
    await mongoClient.close();
  }
})();
```

12 APPENDIX F – SQL CREATE QUERIES FOR EMBEDDIG

```sql
CREATE TABLE Author (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) NOT NULL
);

CREATE TABLE Publisher (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) NOT NULL
);

CREATE TABLE Category (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) NOT NULL
);

CREATE TABLE Book (
  id SERIAL PRIMARY KEY,
  title VARCHAR(255) NOT NULL,
  author_id INTEGER REFERENCES Author(id),
  publisher_id INTEGER REFERENCES Publisher(id)
);

CREATE TABLE BookCategory (
  id SERIAL PRIMARY KEY,
  book_id INTEGER REFERENCES Book(id),
  category_id INTEGER REFERENCES Category(id)
);

INSERT INTO Author (name)
VALUES ('John Doe'), ('Jane Smith'), ('Alice Johnson');

INSERT INTO Publisher (name)
VALUES ('Penguin Publishing'), ('HarperCollins'), ('Random House');

INSERT INTO Category (name)
VALUES ('Fiction'), ('Non-fiction'), ('Science Fiction'), ('Mystery'),
('Romance');

INSERT INTO Book (title, author_id, publisher_id)
VALUES ('The Great Novel', 1, 1), ('Fact or Fiction', 2, 2), ('Mars
Colony', 3, 3), ('Detective Stories', 1, 2), ('Love in the Time of AI', 2,
1);

INSERT INTO BookCategory (book_id, category_id)
VALUES (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (1, 3), (4, 5);
```

13 APPENDIX G – SQL DATA GENERATION QUERIES FOR EMBEDDING

```sql
INSERT INTO Author (name)
SELECT CONCAT('Author ', n)
FROM (
  SELECT ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) AS n
  FROM Information_schema.tables LIMIT 10000
) AS t;

INSERT INTO Publisher (name)
SELECT CONCAT('Publisher ', n)
FROM (
  SELECT ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) AS n
  FROM Information_schema.tables LIMIT 10000
) AS t;

INSERT INTO Category (name)
SELECT CONCAT('Category ', n)
FROM (
  SELECT ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) AS n
  FROM Information_schema.tables LIMIT 10000
) AS t;

INSERT INTO Book (title, author_id, publisher_id)
SELECT CONCAT('Book ', n), (n % 10000) + 1, (n % 10000) + 1
FROM (
  SELECT ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) AS n
  FROM Information_schema.tables LIMIT 10000
) AS t;

INSERT INTO BookCategory (book_id, category_id)
SELECT ((n - 1) % 10000) + 1, ((n - 1) % 5000) + 1
FROM (
  SELECT ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) AS n
  FROM Information_schema.tables LIMIT 20000
) AS t;
```

14 APPENDIX L – SQL DATA GENERATION QUERIES FOR REFERENCING

```sql
CREATE TABLE a (
  id SERIAL PRIMARY KEY,
  name TEXT,
  age INTEGER,
  email TEXT
);

CREATE TABLE b (
  id SERIAL PRIMARY KEY,
  title TEXT,
  a_id INTEGER REFERENCES a(id)
);

CREATE TABLE c (
  id SERIAL PRIMARY KEY,
  description TEXT,
  a_id INTEGER REFERENCES a(id)
);

CREATE TABLE d (
  id SERIAL PRIMARY KEY,
  value INTEGER,
  b_id INTEGER REFERENCES b(id)
);

CREATE TABLE e (
  id SERIAL PRIMARY KEY,
  comment TEXT,
  c_id INTEGER REFERENCES c(id)
);

INSERT INTO a (name, age, email) VALUES ('John Smith', 32,
'john.smith@example.com'), ('Jane Doe', 27, 'jane.doe@example.com'), ('Bob
Johnson', 45, 'bob.johnson@example.com');

INSERT INTO b (title, a_id) VALUES ('Title 1', 1), ('Title 2', 2), ('Title
3', 3);

INSERT INTO c (description, a_id) VALUES ('Description 1', 1),
('Description 2', 2), ('Description 3', 3);

INSERT INTO d (value, b_id) VALUES (100, 1), (200, 2), (300, 3);

INSERT INTO e (comment, c_id) VALUES ('Comment 1', 1), ('Comment 2', 2),
('Comment 3', 3);
```

```
DO $$
DECLARE counter INT := 1;
BEGIN
  WHILE counter <= 97 LOOP
    INSERT INTO a (name, age, email) VALUES
      ('Name ' || counter, counter + 20, 'email' || counter ||
'@example.com');
    counter := counter + 1;
  END LOOP;
END $$;
DO $$
DECLARE counter INT := 1;
BEGIN
  WHILE counter <= 97 LOOP
    INSERT INTO b (title, a_id) VALUES
      ('Title ' || counter, counter);
    counter := counter + 1;
  END LOOP;
END $$;
DO $$
DECLARE counter INT := 1;
BEGIN
  WHILE counter <= 97 LOOP
    INSERT INTO c (description, a_id) VALUES
      ('Description ' || counter, counter);
    counter := counter + 1;
  END LOOP;
END $$;
DO $$
DECLARE counter INT := 1;
BEGIN
  WHILE counter <= 97 LOOP
    INSERT INTO d (value, b_id) VALUES
      (counter * 100, counter);
    counter := counter + 1;
  END LOOP;
END $$;
DO $$
DECLARE counter INT := 1;
BEGIN
  WHILE counter <= 97 LOOP
    INSERT INTO e (comment, c_id) VALUES
      ('Comment ' || counter, counter);
    counter := counter + 1;
  END LOOP; END $$;
```