MongoDB Schema Design and Evaluation

by

Vidhi Arvindbhai Rajvir
202011058

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF TECHNOLOGY
in
INFORMATION AND COMMUNICATION TECHNOLOGY
to

DHIRUBHAI AMBANI INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGY

May, 2022

Declaration

I hereby declare that

i) the thesis comprises of my original work towards the degree of Master of Technology in Information and Communication Technology at Dhirubhai Ambani Institute of Information and Communication Technology and has not been submitted elsewhere for a degree,

ii) due acknowledgment has been made in the text to all the reference material used.

[illegible]

Vidhi Arvindbhai Rajvir

Certificate

This is to certify that the thesis work entitled MongoDB Schema Design and Evaluation has been carried out by Vidhi Arvindbhai Rajvir for the degree of Master of Technology in Information and Communication Technology at *Dhirubhai Ambani Institute of Information and Communication Technology* under my/our supervision.

[illegible]

P M Jat
Thesis Supervisor

Acknowledgments

Prof. P M Jat, my thesis supervisor, deserves my heartfelt gratitude for his unwavering support and patience throughout the thesis process. He kept encouraging me that I would be able to finish my thesis even when I had given up hope. Thank you for encouraging me to get the best out of me.

I am thankful to the Resource center and all the resources of DAIICT for giving me the best research environment. I am also grateful to my friends for their motivation. Further, I want to acknowledge DAIICT for providing me with the most valuable learning resources. I was given numerous opportunities to express myself and develop new skills such as teaching assistantship, time management, and campus placement. I want to give a special thanks to all the faculty members of DAIICT.

I would like to thank God for giving me the strength needed to complete this thesis. At last, I very thankful to my parents and my sister for their constant support and faith in me. Without their encouragement and belief, it would not have been accomplished.

Contents

Abstract

Relational Databases have always been the only choice for several decades. However, modern times they have started falling short in terms of scalability. Also, current applications do not fit well in the strict schema approach of relational databases. Today NoSQL databases have become an essential alternative for web-scale applications. Many users of such applications are moving to No-SQL databases. In such a scenario, there is a requirement to convert from RDB to NoSQL.However, this field of research is not explored extensively. Moreover, expert knowledge is required to determine a suitable NoSQL schema for a given RDB and application.

The work is divided into two parts. The first part takes the algorithm to convert RDB to a NoSQL schema design outline. The algorithm accepts RDB as input. Here Brute force is applied to choose whether the collection in MongoDB schema needs to be embedded or referenced for each relationship between two tables. The second part uses a query-based metrics tool to determine the best schema from the generated schema outlines. The queries and schema outline are taken as input to the tool. The queries here define the application workflow. Through the various parameters and functions, the tool decides the number of collections required to fetch the query and the path of collection traversed by the query. Each schema outline gets a schema score and query score. The schema score is the sum of the query scores for the specific schema. The optimized algorithm is defined based on the results obtained, which provides the most appropriate schema for a given RDB and application query load.

List of Principal Symbols and Acronyms

ACID	Atomicity, Consistency, Isolation, and Durability
BFS	Breadth First Search
CAP	Consistency, Availability and Partition Tolerance
CPU	Central Processing Unit
DBMS	Database Management System
ORM	Object Relational Mapping
RAM	Random Access Memory
RDB	Relational Database
SQL	Structured Query Language
SSD	Solid State Drives

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1 Relational Database and NoSQL Database

Relational Databases have been the dominant technology for data storage and management purposes. The relational model works on a fixed schema that requires to be defined. Each table is a precisely defined collection of rows and columns. Relational databases use SQL query language to query and manipulate data. However, Programming languages, architectures, platforms, and methods have changed dramatically in recent decades in enterprise computing, causing the emergence of a new system for storing and managing the data. This new technique allows storing data in a non-relational and distributed manner known as 'NoSQL' databases.

1.2 Why NoSQL?

A relational model can be inefficient when storing data that cannot fit into tables. The advantage of NoSQL Databases over Relational Database is that they can store different types of data like videos, audio, documents, social media, and e-mail. Although these data types can be stored in the relational database, they impact performance. A NoSQL or Not Only SQL databases have a flexible schema to store these data types.

The object-relational impedance mismatch is a severe issue for developers working with relational databases. The difference between the relational model and in-memory data structures is an impedance mismatch. SQL queries are not well adapted to the object-oriented data structures used in most applications. Some application operations require multiple and very complex queries. Data mapping and query generation complexity raises too much and becomes challenging to maintain on the application side [1]. Though most translations (Object to Relations and Relations to Objects) are automated using Object Relational Mapping (ORM) solutions such as Hibernate or similar, these add complexity, maintenance, and computational expense.

In most situations, SQL databases are vertically scalable by putting extra CPU, RAM, or SSD capacity on a single server to increase the load. Relational databases do not fit onto “cluster computing” and hence do not scale well. In comparison, NoSQL databases can scale horizontally. Sharding is possible in a relational database, but we lose querying, referential integrity, transactions, and consistency across the shards. Sharding or adding more servers to the NoSQL database can handle more load effectively. Because horizontal scaling has a higher total capacity than vertical scaling, NoSQL databases are chosen for huge, constantly changing data sets.

NoSQL databases can perform queries faster than SQL databases in certain circumstances. Because SQL database data is usually normalized, searching for a single object or entity must join data from multiple tables. Joins might get costly as the tables expand in size. Because queries in NoSQL rarely use joins, they are relatively fast.

A flexible schema allows one to modify a database as an application needs change. Developers may quickly develop and integrate new application features to deliver more value to users under varying conditions. The most significant applications for NoSQL Databases are flexible data, distributed storage, cache memory, web applications, and social media. Compared to RDB, NoSQL databases are more compatible with changes based on business requirements, queries, or data patterns. There is an ever-changing schema design process throughout the application’s development and after that.

1.3 NoSQL Design

The significant distinction between relational and NoSQL databases is handling data relationships. NoSQL is primarily designed for high data volumes and sharding across numerous servers; distributed joins are difficult and expensive in this context, and data relationships must be handled differently.

The design of the NoSQL schema depends on the application and query load.

NoSQL databases' fundamental feature is optimizing data access based on query patterns and business needs. The access pattern provides which data is accessed frequently based on the use of the application—the goal of the design process is to plan a fast and effective structure for application queries. Schema design for NoSQL usually includes designing Keys, Indexes, and Denormalization of attributes. These things are inter-dependent on the application queries and application workflows. Since schema design in NoSQL is query-driven, and there may be query change based on the change in requirements, NoSQL design must be revisited and modified as iterative as needed.

NoSQL is "Aggregation Oriented." Aggregation-oriented databases are more cluster friendly. If related data (or data queried together) are stored, querying them in the distributed environment becomes efficient. In this case, we do not have to perform joins. These also lead to a de-normalized design, meaning related entities are embedded within an entity.

The downside of such a system is that they are query load specific. One design is good for one set of queries while poor for another set of queries. It is a general problem issue with the de-normalized design and data redundancy. The purely normalized design does not suffer from this problem. Therefore, finding a trade-off between normalization and de-normalization remains the main challenge in designing NoSQL systems. Having appropriate aggregate is key in No SQL database designs.

1.4 Thesis Problem Area

There is a desire for relational databases to be converted to NoSQL. Because the data magnitude for storage is enormous and ongoing applications cannot meet scalability and availability, there have been attempts to shift from traditional RDB to NoSQL. There are several reasons for data transfer from RDB to NoSQL, including server and equipment upgrades, database updates to a new version, inefficient databases, and organization policy changes. Converting RDB to NoSQL data models is time-consuming and can take days or weeks to complete. Companies want an application-specific solution for the conversion.

Many people would want to shift from relational databases to NoSQL databases for the said reasons. For moving from RDB to NoSQL, we typically require the following tasks:

1. Determine the Target schema for the given relational schema
2. Create Target Schema
3. Move data from relational tables to target collections

It is highly desirable to automate these tasks to the fullest extent.In this thesis, we attempt to work on the first task.
While figuring out the target schema, we have quite a few challenges –

1. Figure out whether an entity should be embedded or not
2. How is the hierarchy of embedding?
3. If embedding is not considered then figure out what references are stored in the entity

In No-SQL systems, query load dictates how entities are to be embedded or stored externally. When not embedded and stored externally, their relationship is represented through references. Determining the "aggregations" and figuring out embedding and referencing remains the primary outcome of NoSQL design.

Here we use PostgreSQL as RDB and MongoDB as a NoSQL database. As is well known, data modeling methodologies for relational and non-relational databases are vastly different. There are several challenges in converting a relational database to a non-relational database. The structure and format diversity of the numerous data sources is the fundamental problem in data transformation.

Thus, there is no proper model for automatically converting RDB to a NoSQL database. This work uses an algorithm intuition to generate candidate schemas from a given RDB and query the application's load. Brute force is applied to determine the nature of schemas for a given relationship in RDB. After that, a query-based tool evaluates each candidate schema. This tool helps decide the most suitable schema outline based on different parameters.

1.5 Thesis Objective

Here, our thesis aims to find out an algorithmic solution that produces an optimal schema for a given relational schema.

1.6 Thesis Outline

The remaining thesis has the following structure: Chapter 2 discusses MongoDB Schema Design , the Data Model design considerations for MongoDB. Chapter 3 shows the literature survey, which inspires the algorithm intuition and the query-based tool to determine the most suitable schema. Chapter 4 is about the experiment conducted on the TPC-H dataset and the result discussion. The thesis work concluded with Chapter5.

CHAPTER 2
MongoDB Schema Design

This chapter gives an introduction to the MongoDB database. It also discusses the database design of the NoSQL Databases. Furthermore, Design Considerations of Document-based databases and Related Work are studied, leading to the intuition for the algorithm.

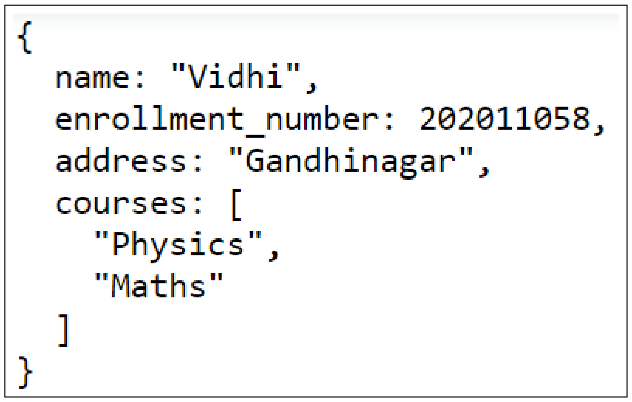
2.1 MongoDB Introduction

A document database must ensure enough data storage, good query speed, and high performance read and write concurrently. MongoDB is an open-source document-based database released in 2009.MongoDB is used to store an immense amount of data. The storage of data and retrieval in MongoDB is different from the traditional Relational Database. Many companies like Google, eBay, Adobe, and Facebook use MongoDB to store data.

MongoDB's document-based structure helps represent complex hierarchical relationships by embedding document objects or an array of objects into a single document. MongoDB also offers a range of benefits over relational databases, such as supporting an "aggregation pipeline" to optimize the database. It also allows indexing for faster querying.

2.2 MongoDB Data Model

A document is a data structure of field and value pairs in MongoDB. JSON objects and MongoDB documents are similar. Other documents, arrays, and Object arrays could also be field values. The way a document is represented varies by programming language, but most have a data structure that fits the bill, such as a map, hash, or dictionary. Duplicate keys are not allowed in MongoDB documents. Keys must not contain the null character.



```
{
  name: "Vidhi",
  enrollment_number: 202011058,
  address: "Gandhinagar",
  courses: [
    "Physics",
    "Maths"
  ]
}
```

Figure 2.1: Document Example

Field-and-value pairs make up MongoDB documents, which have the following structure [7]:

```
{
field1:value1,
field2:value2,
.
.
}
```

The value may be of any one of the following:

- JSON Primitive type : string,Boolean,number
- BSON Primitive type : datetime,objectId
- value Array
- Objects
- Object Array
- null

A group of documents is referred to as a collection. If a document in MongoDB is the equivalent of a row in a relational database, then a collection is the equivalent of a table. MongoDB combines collections into databases in addition to organizing documents by collection. A single MongoDB instance can host several databases containing zero or more collections. The table below shows the mapping of various components of RDB with MongoDB. Every relational database has a design schema that depicts the tables and their relationships. There is no definition of a relationship in MongoDB.

RDBMS	**MongoDB**
Table	Collection
Column	Field
Row/Tuple	Document
Index	Index
Joins	Embedding/Referencing
Primary Key	_id Field

Table 2.1: Mapping of RDB components with MongoDB components

2.3 Normalization vs. Denormalization

There are various ways to represent data, and one of the most significant considerations is how much data should be normalized. Normalization is separating data into several collections and linking them together through references. Each data has its collection, even if referenced in many documents. As a result, just one document needs to be updated to change the data. However, because MongoDB lacks join functionality, aggregating documents from various collections will require multiple queries.

The contrary of normalization is denormalization, which involves embedding all data into a single document. Instead of referencing a single definitive copy of the data, several documents may contain copies of the data. Multiple documents must be updated if information changes, but all associated data may be retrieved with a single query.

The critical decision is whether referencing (normalization) or embedding (denormalization) is the appropriate choice for a given context. When changing the schema, we must evaluate how to make trade-offs between performance and data redundancy.

- Embedding for Locality: Because MongoDB stores documents contiguously on disc, putting all of the data embedded into one document means it will never be more than one disc seek away. If we use reference, this case will work worse than the 'JOIN' operation in SQL. The database still has to perform several seeks to locate the data.

- Cardinality: To some extent, embedding should not be done if more data is generated. If the embedded fields or the number of embedded fields is expected to extend indefinitely, they should be referenced rather than embedded. Comment trees and activity lists should be stored as separate documents rather than embedded.
- Flexibility: An embedded technique works well if the application's query pattern is well-known and data is accessible in just one way. If the application can query data in various ways or cannot predict the patterns in which data will be requested, a more "normalized" or Referencing approach may be preferable.
- Arity of the Relationship: Embedding has several drawbacks:
  - The larger a document is, the more RAM it consumes.
  - Documents that grow in size must eventually be copied to more prominent places.
  - MongoDB documents are limited to 16 MB in size.

  The difficulty with using too much RAM on a MongoDB server is that RAM is usually the most valuable resource. A MongoDB database, in particular, caches frequently requested documents in RAM, with the more oversized items taking up more space. It is possible to run out of space in a relationship if the arity is unbounded.
- Many-to-Many Relationships: The scenario of many-to-many or M: N relationships is another circumstance favouring employing document references upon embedding.
- Data locality within a document and MongoDB's ability to make atomic updates to a document (but not between two documents) are the two most significant advantages of embedding sub documents.

On the one hand, embedding all linked tables may improve speed while also causing data redundancy. On the other hand, MongoDB will send numerous queries when reading related documents if there are references for each table.

The option for embedding or referencing depends on the application's needs. The table below shows the circumstances under which embedding or referencing must be considered.

When to use Embedding...	**When to use Referencing..**
Sub-documents are of small size	Sub-documents are of large size
Rare changes in data	labile data
Eventual consistency considered	Immediate consistency is needed
Quick Reads	Quick Writes
Documents growing insignificantly	Documents growing extensively

Table 2.2: Embedding vs Referencing

2.4 Design Consideration of MongoDB Database

Designing a data model is the first stage in creating a new application. In RDBMS, a data model is designed by normalizing the data to remove redundancy from the tables. In document-based databases, data is stored in the form of documents. MongoDB allows keeping an array of values inside the document. In MongoDB, we can locally encode multi-valued properties of the data to get the performance benefits of a denormalized form of a relational database without the problems of updating redundant data. However, it also complicates our schema design process.

It is challenging to know when to normalize or denormalize: normalizing speeds up writes, and denormalizing speeds up reads. As a result, choosing which trade-offs are appropriate for a particular application is essential.

CHAPTER 3
Literature Survey

Document-based databases are frequently used for data storage in applications. Among Document-based Database, MongoDB is the most popular database. Working with data in documents is often more straightforward for developers than working with data in tables. In most programming languages, documents map to data structures. When storing or retrieving data in documents, developers do not have to worry about manually separating it across many tables or joining it back together. Developers also do not need to depend on an ORM to manage data manipulation for MongoDB.

Therefore, a requirement to move the relational database to Mongo DB. In migration, determining the target schema is one of the challenging tasks. This thesis focuses on defining the target schema for the migration problem.

Here we present a few related works from the literature.

3.1 Automatic Mapping of MySQL database to MongoDB

This approach [14] presents an algorithm for automatic mapping of MySQL database to MongoDB. The algorithm applies the metadata stored in the MySQL system tables. It maps the relational database concept with the MongoDB collections. The entities of the relational schema and the relationships between them are mapped to specific actions to model into the MongoDB schema. The algorithm verifies for each table what relationships are involved. Based on the relationship, the design consideration of whether to embed or reference is taken. Thus, it tries to automate the relational database mapping to the NoSQL database.

3.2 R2NoSQL Approach to convert Relational database into NoSQL databases

Another approach by Freitas et al. [8] describes the R2NoSQL method for converting data between the mentioned models, considering the structural heterogeneity between Relational and NoSQL models. It compares the Relational model's data structures to the four main NoSQL techniques (key-value, columns, documents, and graphs), resulting in a collection of probable conceptual mappings between RDB and a NoSQL. It also includes a tool prototype that implements a case study using a relational database and a NoSQL system based on documents. By comparing the results of the same set of queries run on both platforms, experiments were undertaken to assess the consistency of the created mappings.

3.3 General schema conversion Model for converting relational database to NoSQL database

Gansen Zhao et al. [15] offer a general schema conversion approach for converting relational databases to NoSQL databases, which aids migration and increases reading efficiency. To boost the efficiency of cross-table queries, Zhao designed the schema conversion technique using the idea of table nesting. They consider references as a relationship between the parent entity and the child entity of data in a NoSQL database and store the structured data. They present an algorithm that transforms the graph of the relational database and generates nested sequences among the tables in a relational database to assure the validity of change in the schema by ensuring that the table contains to fetch the given query content.

3.4 Model Transformation from Relational Database to MongoDB

Zhao and Wang et al. [10]. present a novel method for transforming models and migrating data from relational databases to MongoDB to address data migration issues. Their technique considers the query characteristics and the data features of relational databases. The tags and relationships in the relational database are the sole basis for their new model transformation procedure. The tags are obtained by the characteristics of the query from the query log. With the different description tags, they can customize different NoSQL model strategies.

3.5 The Approach

The data from the relational database is migrated to the NoSQL database to achieve better performance. As a result, we must investigate the current relational database's constraints. To convert the ER model to a MongoDB physical model, we must first map the ideas between them. While creating the MongoDB schema, the biggest concern is which table can be referenced and which must be embedded. Embedding means data redundancy. When designing the algorithm, we must have a trade-off between performance efficiency and data redundancy. The concept mapping of the relational table to MongoDB is shown in Table 2.1.

All relational database components may be easily transferred to document-based databases except table relationships. The most challenging element of converting RDB to MongoDB is mapping the relationships between the tables. There is no condition for joining two tables in MongoDB; instead, one table can be referred from another table. As a result, we could use MongoDB's Reference to show the relationship in an RDB.

On the other hand, client-side applications must execute numerous queries to perform a read that requires joining two collections under this MongoDB model design. It may also slow down the reading pace. Embedded documents in MongoDB allow reading data in a single query, which will improve performance. However, we cannot just embed all relationships because one entity can have several relationships with others.

To address the above mentioned issues, we formed a brute force approach inspired by [10] that checks for the possible document schema generated for a given RBD with the help of the tags. The algorithm takes inputs from the metadata of the relational database, which is the conceptual model. These tags are derived from the query log of the database. There are two types of the tags:

- Description Tags
- Action Tags

The log keeps track of user actions and the time it took to complete them. These description tags were extracted from a relational database's log. The number of thresholds in the system will be predetermined. Four tags characterize relational databases' query characteristics and data attributes.

1. Big Size
2. Frequent Modify
3. Frequent Join
4. Frequent Insert

Description Tags	Interpretation	Values
Frequent Modify	The tables having update and alter more often	True\False
Frequent Insert	The tables having inserts more than the threshold value	True\False
Frequent Join	The tables having joins more than threshold value	True\False

Table 3.1: Description Tags and Interpretation

Action Tags	Interpretation	Values
Can _ embed _ leafentity	whether leaf vertex entity can be embedded into successor vertex	True \False
Can _ embed _ successorentity	whether succesor vertex can be embedded into leaf vertex	True\False

Table 3.2: Action Tags and Interpretation

To generate Action tags, we use the description tags and the nature of the relation of the tables. The action tags determine when to use embedding and when to use references in the database. Assume we have access to an existing relational database's metadata. By accessing the metadata of the relational database, we generate DAG for the table and the relationship between the tables in the relational database. We also have DAG for the queries in the Query Log developed.

The figure below shows the example of an RDB converted into a DAG. A DAG (Directed Acyclic Graph) takes the relational database tables as their vertices and the nature of the relation of the tables as the edges. The direction of the edges determines the transformation flow. Each DAG can be viewed as a tree, with the target entity at the root vertex. That means the edge points from leaf vertex to parent vertex. The metadata of each RDB table is stored in each vertex, including the table name, fields, and primary key. The relationship data between two tables is encapsulated by the edge between two vertices, including primary and foreign keys, determining which table has one side and many sides.

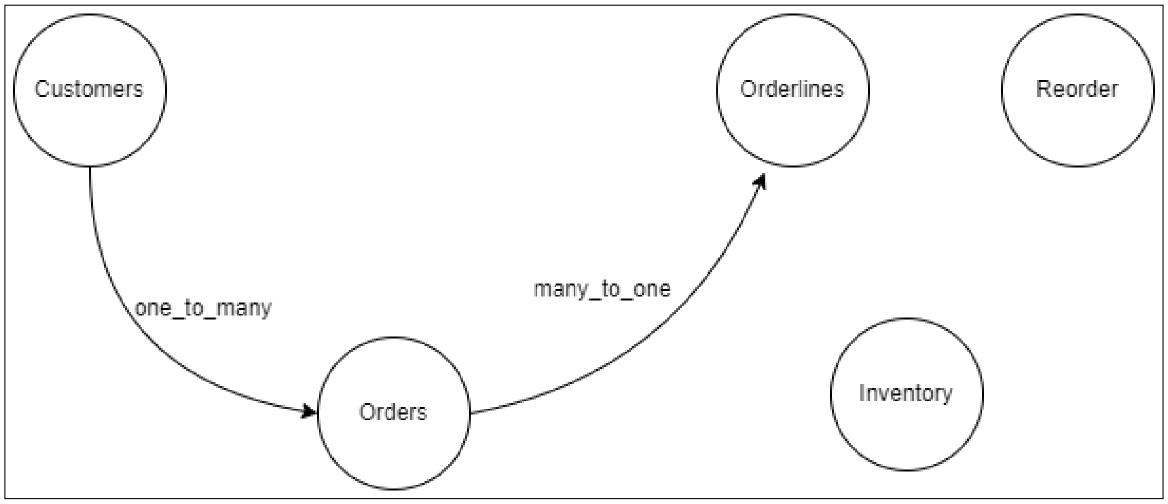


Figure 3.1: DAG Example

Breadth-First Search (BFS) is used to create the DAG [2] . For generating a DAG from RDB, the metadata of the RDB is taken. One table from the RDB is taken as input. Other tables are added to the queue based on the table's foreign keys until all the tables are generated. Based on the breadth-first search algorithm, all the exported keys of the input table are extracted.

Other tables are added to the queue to create a DAG-based on the exported keys. This process is continued till the queue is empty. While creating DAG, the description tag big size is attached to each table, showing whether the table size exceeds a given threshold.

First and foremost, the description tags must be added to the ER model. The last section discussed obtaining description tags from a relational database's log system. We employ description tags and the ER model's relationship to generate action tags. The algorithm for generating action tags is presented in Algorithm 2 based on the MongoDB design. Can_embed_leaf entity /Can _embed _successor entity is used to determine the embedding or reference the collections?

The DAG's input parameter for this function is an entity. The purpose of the function is to determine whether to embed or not, and it will return true for the former and false for the latter. Depending on the MongoDB document size limit and data consistency, our algorithm recommends against embedding this entity into other entities if it has the "Big size" tag, "Frequent edit" tag, or "Frequent insert" tag. We use the Topological sorting algorithm [4] on the DAG to determine the embedding order. Here Topological sorting is used to determine the ordering of the embedding of the entities.

Finally, the algorithm's output embeds all the entities based on the conditions and generates a MongoDB schema. It is advisable to embed the entity in the object form for one-to-one relationships. In contrast, for one-to-many relationships, entities should be embedded in the form of an Object array.

3.6 Schema Evaluation Related Work

Evandro [12] proposed six query-based metrics to evaluate the NoSQL document schema against the query load of the application. A standard structure is used in the form of DAGs to represent the schemas and the queries. The metrics are used to determine which schema has the access pattern that the application requires. These Query-based metrics tool helps choose the best appropriate NoSQL schemas [3].

There are different techniques to convert RDB schema to NoSQL schemas [8–11].In one of the previous works by the same authors [13], one of the approaches for transforming RDB to NoSQL is discussed. This approach, along with others, has been used to convert RDB to NoSQL models. As a collection of DAGs (Directed Acyclic Graphs) is applied to the source RDB, abstraction is used to represent a NoSQL schema. Each DAG has a list of RDB tables converted into a single NoSQL entity (document structure).DAGs are used to describe the conversion of RDB to NoSQL.

A DAG (Directed Acyclic Graph) is defined as a graph G=(V_i, E_i), where V is the vertex which means tables of the RDB, and E is an edge that shows the relationship among the tables. Each NoSQL schema can be represented by a collection of DAGs, where each DAG represents a collection in the schema.NoSQL schema is defined as S = $\{DAG_1, DAG_2, ..., DAG_n | DAG_i \in$C},where C is collections in the schema.

Query is also represented in form of DAG like q =(V_q, E_q),where V_q is vertex here as query tables and E_q represents the join in the query tables. Here q belongs to Query set.

Two rules are defined to convert SQL Select statement into a DAG. The tool does not support Sub-queries and full outer join clauses in SQL statements.

- **Rule 1**: if there is only one table in the SQL statement ,then DAG is formed with one table as the vertex
- **Rule 2**: if there is more than one table in the SQL statement,then DAG is created by defining the root vertex and the other tables are added based on the join condition.

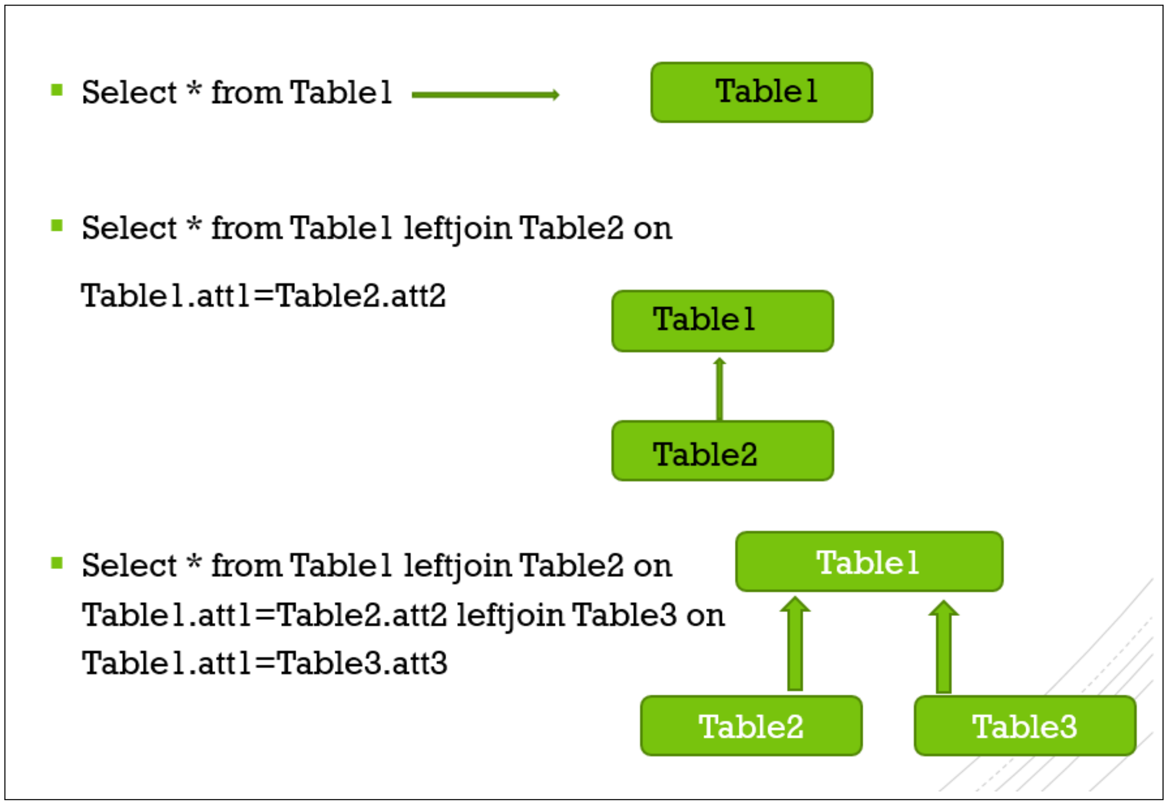

Select * from Table1
Table1
Select * from Table1 leftjoin Table2 on Table1.att1=Table2.att2
Table1
Table2
Select * from Table1 leftjoin Table2 on Table1.att1=Table2.att2 leftjoin Table3 on Table1.att1=Table3.att3
Table1
Table2
Table3

Figure 3.2: Converting query into DAG

There are six metrics for evaluating the best fit schema for converting from RDB to NoSQL schema.

- **Direct Edge Coverage** which gives the edge coverage of a query against the edges of the given collection.Here the direction of the edges is taken into consideration.
- **All Edge Coverage** which gives the edge coverage of a query against the edges of the collection .Here the direction is not taken into consideration.
- **Path Coverage** measures the coverage of paths of the query in relationship to the paths of the collection.

- **Sub Path Coverage** measures of there is a query path existing in the collection of subpath.
- **Indirect Path Coverage** to check if the query path exist in the collection in the form of indirect path.
- **Required Collection Coverage** measures the least number of collections to fetch a query.

These metrics are combined to calculate the QScore and SScore, respectively. Here QScore gives the score per metric or related metrics per query. SScore is for a set of queries on a given schema.

$$
\begin{aligned}
&QScore(DirEdge, q){=}DirEdge(q)\\
&QScore(AllEdge, q){=}AllEdge(q)\\
&QScore(ReqColls, q){=}ReqColls(q)\\
&path_v = Path(q_i) * w_p\\
&subpath_v = (SubPath(q_i) * w_s p)/depthSP(q_i)\\
&indpath_v = (IndPath(q_i) * w_i p)/depthIP(q_i)\\
&QScore(Paths, q_i){=}max(path_v, subpath_v, indpath_v)\\
&SScore(metric, Q){=}\sum_{n=1}^{|Q|} QScore(metric, q_i) * w_i
\end{aligned}
$$

The metric value is first weighted according to its path type, then split by the little depth in the schema where the path's root vertex is placed. A specific function determines each metric's depth. The weights are used to prioritize the metrics Path, Sub Path, and Indirect Path.

As per changes in the metrics, we have taken an additional key value in the JSON the object for input, which is the "type" variable. The "type" variable can be:

- Reference
- Embed Object Array
- Embed Object

Based on the variable, we have adjusted the path weights used to calculate the path metrics and SScore. If the variable is "Reference," then the path weight will be doubled, indicating that the query needs to be fetched from another collection in the path. Also, we have tried to make changes in the depth variable. If the type of variable is "Reference," the depth variable is doubled, indicating that the depth of the query fetched increases.

CHAPTER 4

Schema Generation and Evaluation

This section includes execution flow and data set, query set, and hardware and software used.

4.1 Experiment Execution Flow

Execution Flow is shown in the following steps:

Step1: TPC-H Dataset is loaded into PostgresSQL

Step2: Connection is established between PostgresSQL and Java Program

Step3: Algorithm is run, taking input from the query log and the metadata of the database

Step4: Generate candidate MongoDB schemas outline.

Step5: Take queries describing the query load of the application

Step6: Feed the candidate schema as well as queries in the form of DAG to the query-based metrics tool

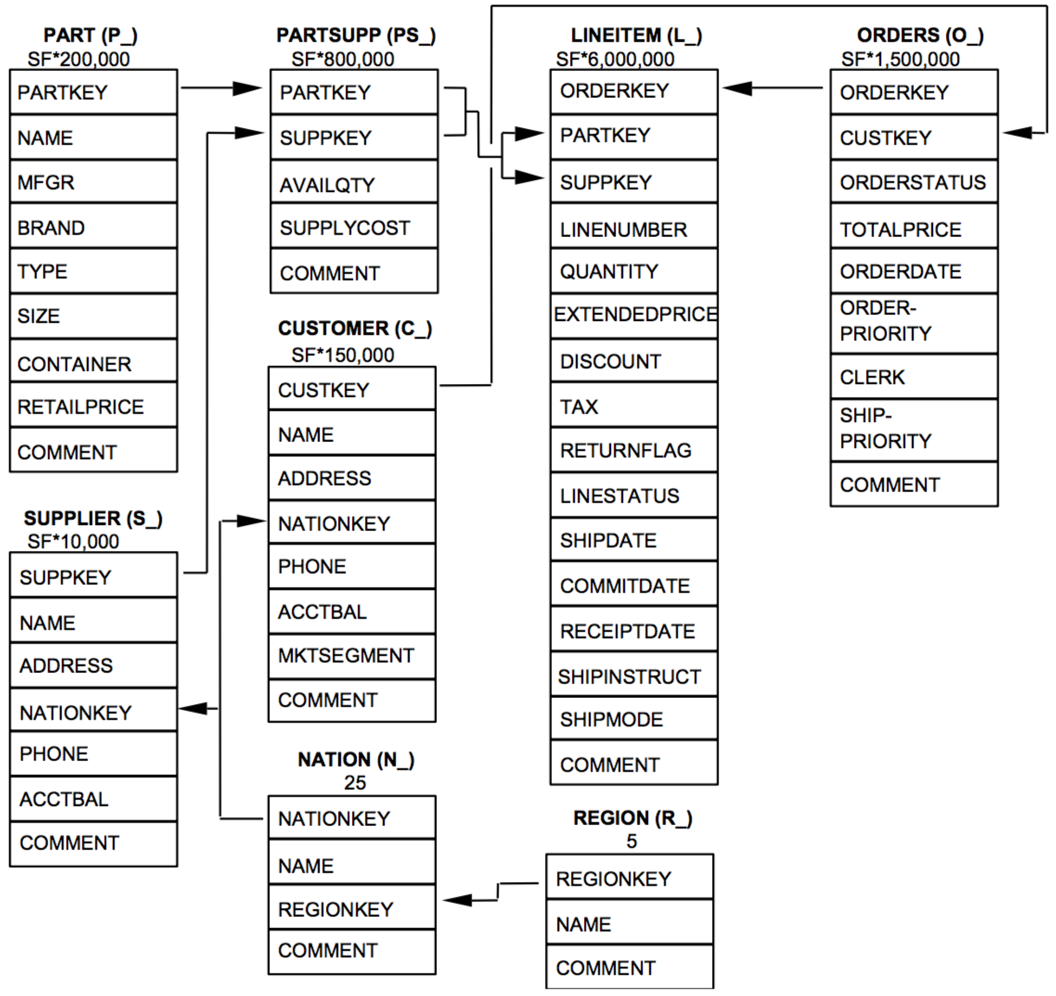
Step7: Compare the results generated

Step8: Determine an optimal algorithm that provides the best results based on the results.

4.2 DataSet

The TPC Benchmark-H (TPC-H) is a decision support benchmark. It consists of a set of business-oriented ad-hoc queries and concurrent data modifications. The queries and data used to populate the database were intended to be relevant to a wide range of industries. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions [6].The TPC-H database's components are divided into eight discrete and distinct tables (the Base Tables). Figure 4.1 depicts the relationships between the columns of these tables: The TPC-H Diagram.

Figure 2: The TPC-H Schema



Legend:

- The parentheses following each table name contain the prefix of the column names for that table;
- The arrows point in the direction of the one-to-many relationships between tables;
- The number/formula below each table name represents the cardinality (number of rows) of the table. Some are factored by SF, the Scale Factor, to obtain the chosen database size. The cardinality for the LINEITEM table is approximate (see Clause 4.2.5).

Figure 4.1: TPC-H DataSet [5]

4.3 QuerySet

The set of Queries has been taken. Here, the queries' purpose is to define the application workflow and are application-specific. The queries are simple queries without sub queries containing joins. The queries are converted into DAG based on the figure 3.2.

Below is the list of the queries and the DAG formed from those queries.
Queries:

1. select * from customer, orders where o_orderkey BETWEEN 30000 AND 70000
   DAG:customer\orders

2. select s_name,s_address from SUPPLIER left outer join NATION where s_nationkey=n_nationkey
   DAG: supplier\nation

3. select ps_partkey from PARTSUPP right outer join SUPPLIER where ps_suppkey=s_suppkey
   DAG: supplier\partsupp

4. select c_custkey, count(o_orderkey)
   as c_count from CUSTOMER ,ORDERS where o_comment not like '%pending%deposits%'group by c_custkey
   DAG: customer\orders

5. select lineitem.l_orderkey, sum(lineitem.l _extendedprice) , customer.o_orderdate, customer.o_shippriority from customer ,lineitem where customer.c_mktsegment = 'AUTOMOBILE'group by lineitem.l_orderkey, customer.o_orderdate, customer.o_shippriority group by l_orderkey, o_orderdate, o_shippriority
   DAG: customer\lineitem

6. select supplier.s_acctbal, supplier.s_name, nation.n_name, part.p_partkey, part.p_mfgr, supplier.s_address, supplier.s_phone, supplier.s_comment from part , supplier , nation , region where supplier.s_suppkey = part.ps_suppkey and part.p_size = 30
   DAG: part\supplier\nation

7. select $*$ from partsupp
   DAG: partsupp

8. select sum(l_extendedprice* (1 - l_discount)) as revenue from LINEITEM, PART where (p_partkey = l_partkey and p_brand = ’Brand#52’ and p_container in (’SM CASE’, ’SM BOX’, ’SM PACK’, ’SM PKG’) and l_quantity >= 4 and l_quantity <= 4 + 10
   DAG: lineitem\part

9. select c _custkey, c_name, sum(l_extendedprice * (1 - l_discount)) as revenue, c_acctbal, n_name, c_address, c_phone, c_comment from CUSTOMER, ORDERS, LINEITEM, NATION where c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate >= date ’1993-08-01’ and o_orderdate < date ’1993-08-01’ + interval ’3’ month and l_returnflag = ’R’ and c_nationkey = n_nationkey group by c_custkey, c_name, c_acctbal, c_phone, n_name, c_address, c_comment order by revenue desc limit 20
   DAG: customer\orders\lineitem \nation

10. select l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue, o_orderdate, o_shippriority from CUSTOMER, ORDERS, LINEITEM where c_mktsegment = ’AUTOMOBILE’ and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < date ’1995-03-13’ and l_shipdate > date ’1995-03-13’ group by l_orderkey, o_orderdate, o_shippriority order by revenue desc, o_orderdate limit 10
    DAG: customer\orders\lineitem

- Input: TPC _H Dataset
- Output: DAGs of candidate schemas in form of JSON Object

4.4 Experimental Setup

Here we generated dummy schema Outlines to check for the best schema outline. These schema outlines are generated by randomly assigning the tag as"Reference" or "Embed Object Array" to the relationships. This assignment is done by simple if and else statements in the algorithm. Schema D, however, is generated based on the Action Tags and Description Tags in the Algorithm.

The figures below show the pictorial representation of the DAGs generated for different candidate schemas. The DAG of the schemas contains the entities. Each Entity has vertices and edges. Vertices contain the name, id, primary key, and fields of the tables. In comparison, the edges contain the foreign key of many side, many side entity, one side entity, the primary key of one side entity, the source of the edge, the target of the edge, and the type of the variable.

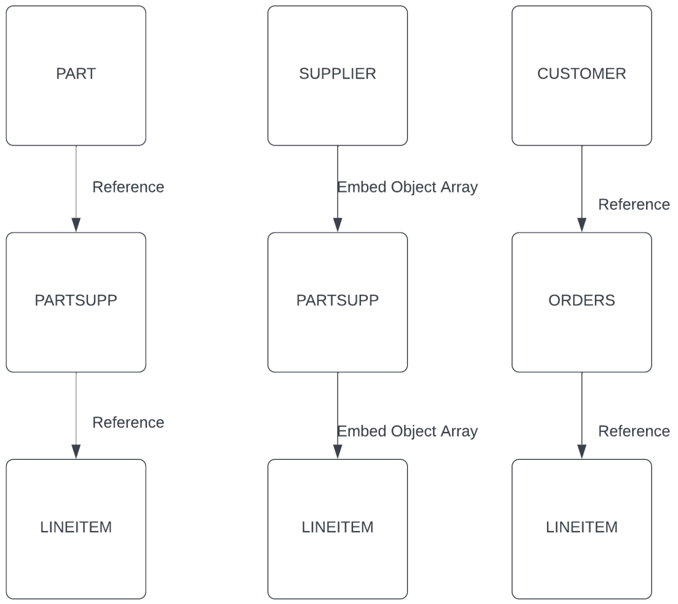


Figure 4.2: Schema A

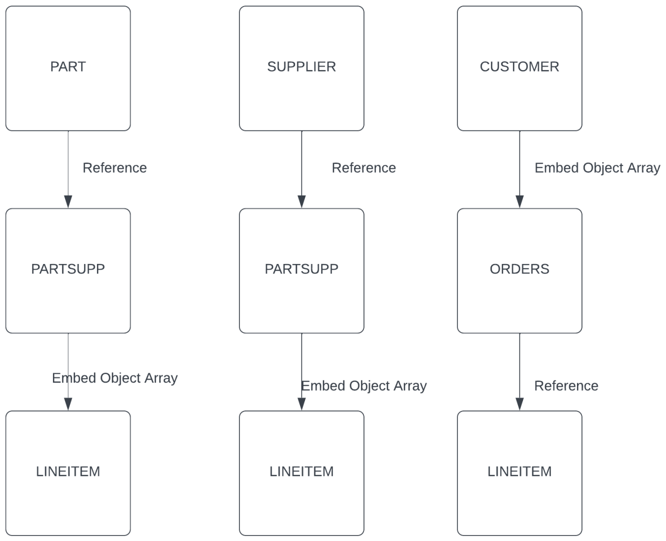


Figure 4.3: Schema B

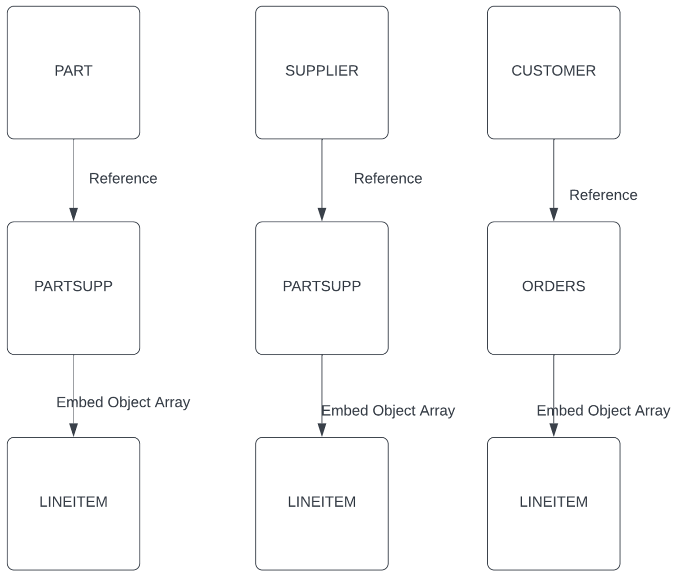


Figure 4.4: Schema C

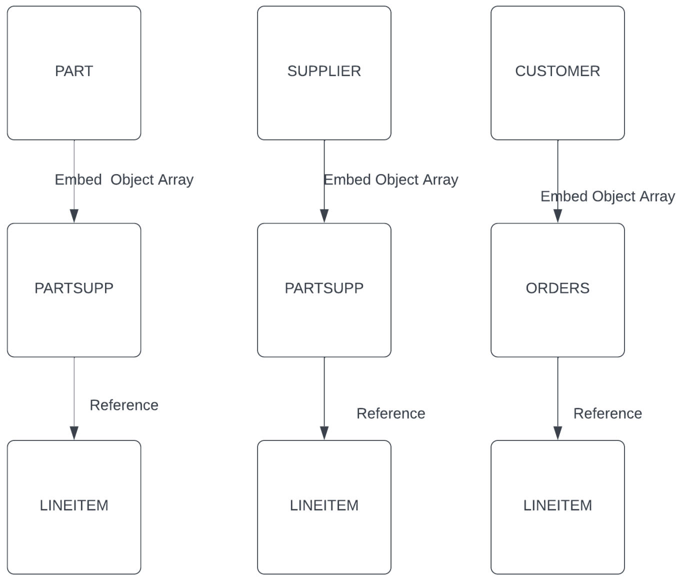
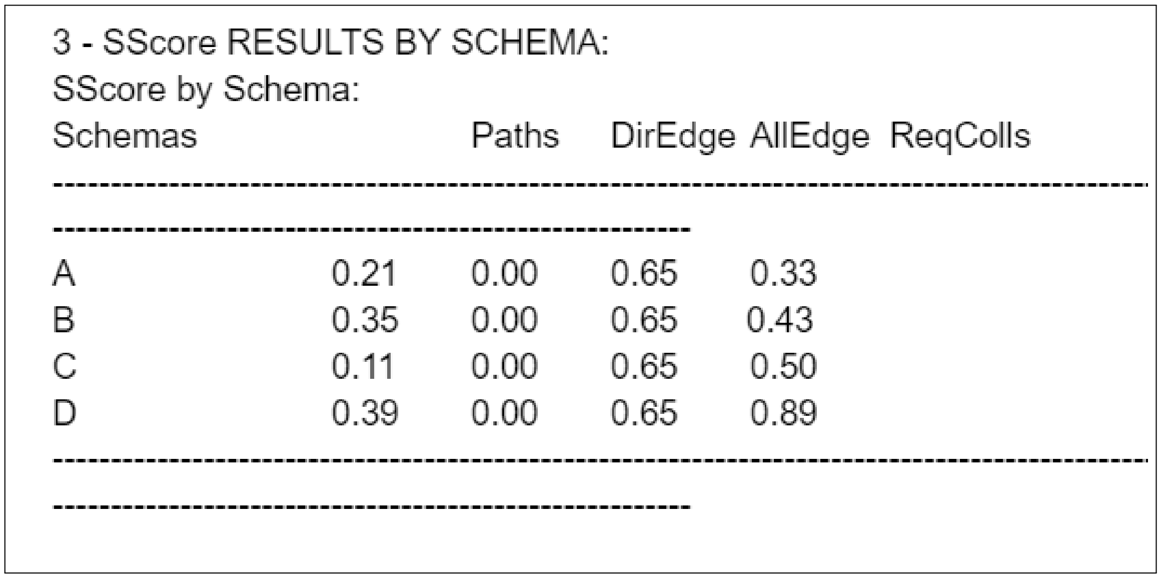


Figure 4.5: Schema D

4.5 Results

The results shown here in the figure 4.6 are the Schema Score(SScore) generated for each schema against the ten queries. Paths are computed based on the metrics Path, SubPath, and IndPath.

Path metrics indicate which schema best covers the queries computed based on the metrics Path, SubPath, and IndPath. Path metrics indicate which schema best covers the queries. Based on the query load, the path from customer to order is frequently used, and schema A and schema C have a reference from customer to order. Therefore whenever there is a 'Reference' type in the relationship between the tables, the path weight increases. Here Schema D has the highest path score, which means it has better query access than other schemas.



```
3 - SScore RESULTS BY SCHEMA:
SScore by Schema:
Schemas                  Paths    DirEdge  AllEdge  ReqColls
-----------------------------------------------------------------------------------------------
-----------------------------------------------------
A                 0.21     0.00     0.65     0.33
B                 0.35     0.00     0.65     0.43
C                 0.11     0.00     0.65     0.50
D                 0.39     0.00     0.65     0.89
-----------------------------------------------------------------------------------------------
-----------------------------------------------------
```

Figure 4.6: Result metrics

We validate the degree to which the entities in the schema are related to one another using the metrics DirEdge and AllEdge, as required by the query access pattern. Here there is a query that has access from the supplier/partssup while all the schema has either reference or embedding of partssup in supplier. Therefore we do not get a direct edge metrics value. In this transformation, all the edges have the same direction; the difference is generated based on the depth and path weights, so the dirEdge and AllEdge values are the same for all the schema. The ReqCols gives the number of collections required to answer the queries.

Here schema D has the maximum ReqCols metrics indicating that it has the maximum alignment of the collections in the schema with the schema access in the query load.

4.6 Result Discussion

According to the Results generated, Schema D is a better schema outline for the given application and query load. Schema D is developed by considering the properties of when to use embedding or referencing for the documents. The primary considerations that will decide upon the schema outline are taken in the form of tags in the algorithm. These tags are formulated based on the characteristics of the tables and the relationships between them. Schema D has the path of the tables, which has a maximum alignment to the query set taken for the evaluation of the schemas. Therefore, it gives maximum value for path metrics.

Moreover, the query load fetches the customer and order table together, so keeping the order table embedded in the customer table increases the performance. Schema B and D have order tables embedded as object array in the customer table. One other advantage of schema D is that the lineitem table is enormous, and it is not feasible to embed lineitem in other tables; therefore, it is better to add a reference to the lineitem table. Schema D achieves this constraint too. Due to the above reasons, Schema D is proving to be a better schema.

CHAPTER 5
Conclusions

Finding the best document-based schema for a particular application has not been easy. There is a trade-off between redundancy and performance efficiency. When there is a case where the tables are embedded despite the conditions, the query efficiency decreases as the schema does not consider the size of the table or frequent changes in the table. But the disadvantage is that we have no choice of accessing the embedded collections individually. Whereas, if only reference is considered, the read query will take more time as the query needs to go from one document to another to fetch the data.

Here we suggest an algorithm that can be used for automatically generating MongoDB schema outlines. This algorithm is optimal while evaluating through the Query Based Metrics tool. Our technique is used as a reference for determining the most appropriate target document schema in an RDB to NoSQL document translation scenario. DAGs are used to represent the queries and the set of target schemas. The measurements show how well the destination schema outline matches the input query. The metrics can be examined individually or collectively using a score per schema (SScore), allowing for specialized analysis. We could only evaluate these diverse transformation methodologies because we embraced DAGs as a uniform unified structure. It also implies that the method is not dependent on any particular technology. Furthermore, if a specific output schema cannot be chosen, the metrics may be used to assist in refactoring current queries. Based on the metrics results, we optimized the algorithm and curated it based on the application's needs.

References

[1] Benefits of nosql database. https://www.devbridge.com/articles/benefits-of-nosql/.

[2] Breath first search algorithm. https://en.wikipedia.org/wiki/Breadth-first_search.

[3] Nosql query based tool. https://github.com/evandrokuszera/nosql-query-based-metrics.

[4] Topological sorting algorithm. https://en.wikipedia.org/wiki/Topological_sorting.

[5] Tpc-h. https://docs.snowflake.com/en/user-guide/sample-data-tpch.html.

[6] Tpc_h dataset. https://www.tpc.org/tpch/.

[7] S. Bradshaw, E. Brazil, and K. Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage*. O'Reilly Media, 2019.

[8] M. C. de Freitas, D. Y. Souza, A. C. Salgado, et al. Conceptual mappings to convert relational into nosql databases. In *ICEIS (1)*, pages 174–181, 2016.

[9] C. de Lima and R. dos Santos Mello. A workload-driven logical design approach for nosql document databases. In *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*, pages 1–10, 2015.

[10] T. Jia, X. Zhao, Z. Wang, D. Gong, and G. Ding. Model transformation and data migration from relational database to mongodb. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 60–67. IEEE, 2016.

[11] G. Karnitis and G. Arnicans. Migration of relational database to document-oriented database: Structure denormalization and data transformation. In

2015 7th International Conference on Computational Intelligence, Communication Systems and Networks, pages 113–118. IEEE, 2015.

[12] E. M. Kuszera, L. M. Peres, and M. D. Del Fabro. Query-based metrics for evaluating and comparing document schemas. In *International Conference on Advanced Information Systems Engineering*, pages 530–545. Springer, 2020.

[13] E. M. Kuszera, L. M. Peres, and M. D. D. Fabro. Toward rdb to nosql: transforming data with metamorfose framework. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 456–463, 2019.

[14] L. Stanescu, M. Brezovan, and D. D. Burdescu. Automatic mapping of mysql databases to nosql mongodb. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 837–840. IEEE, 2016.

[15] G. Zhao, Q. Lin, L. Li, and Z. Li. Schema conversion model of sql database to nosql. In *2014 Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 355–362. IEEE, 2014.