Spring 5-22-2019

Schema Migration from Relational Databases to NoSQL Databases with Graph Transformation and Selective Denormalization

Krishna Chaitanya Mullapudi
San Jose State University

Schema Migration from Relational Databases to NoSQL Databases with Graph Transformation and Selective Denormalization

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

By

Krishna Chaitanya Mullapudi

May 2019

The Designated Project Committee Approves the Project Titled

Schema Migration from Relational Databases to NoSQL Databases with Graph Transformation and Selective Denormalization

by

Krishna Chaitanya Mullapudi

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2019

ABSTRACT

Schema Migration from Relational Databases to NoSQL Databases with Graph Transformation and Selective Denormalization

By

Krishna Chaitanya Mullapudi

We witnessed a dramatic increase in the volume, variety and velocity of data leading to the era of big data. The structure of data has become highly flexible leading to the development of many storage systems that are different from the traditional structured relational databases where data is stored in “tables,” with columns representing the lowest granularity of data. Although relational databases are still predominant in the industry, there has been a major drift towards alternative database systems that support unstructured data with better scalability leading to the popularity of “Not Only SQL.”

Migration from relational databases to NoSQL databases has become a significant area of interest when it involves enormous volumes of data with a large number of concurrent users. Many migration methodologies have been proposed each focusing a specific NoSQL family. This paper proposes a heuristics based graph transformation method to migrate a relational database to MongoDB called Graph Transformation with Selective Denormalization and compares the migration with a table level denormalization method. Although this paper focuses on MongoDB, the heuristics algorithm is generalized enough to be applied to other NoSQL families. Experimental evaluation with TPC-H shows that Graph Transformation with Selective Denormalization migration method has lower query execution times with lesser hardware footprint like lower space requirement, disk I/O, CPU utilization compared to that of table level denormalization.

ACKNOWLEDGMENTS

I want to thank my mentor, Dr. Suneuy Kim for supporting me in learning new concepts and continuously helping me throughout the research. I want to thank Dr. Robert Chun and Mr. Pradeep Roy for their guidance.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

Relational databases have been the traditional backend for most of the software applications for many years since their inception. They served the purpose well within a conventional web application architecture having an RDBMS backend hosted on a server which is vertically scalable without any network partitioning [1]. But at the current rate of data explosion with a large user population, it is necessary to be able to store large volumes for big-data. Platforms like Teradata and Netezza, which are created on relational semantics, have been in the market for a while and are capable of handling terabyte-scale analytical applications [2]. However, there are limitations to these platforms when it comes to elasticity, scalability and fault tolerance in a distributed environment [3]. These solutions, which are architecturally similar to a relational data model, are not very flexible when it involves semi-structured or unstructured data which is often the data collected by most of the big data systems today.

With the advent of cloud computing and horizontal scalability as a storage paradigm which uses less expensive commodity hardware, storage capacity has dramatically increased leading to the widespread use of alternate data stores called ‘Not Only SQL’ (NoSQL) that are partition tolerant, unlike relational databases. Several NoSQL systems are developed with the intent of high scalability that can handle thousands of concurrent users by readily deploying it on the cloud [1]. Many types of NoSQL storage systems have been developed to cater to the application and use case requirements. A majority of them can be placed under five categories, namely key-value stores, document stores, graph stores, wide-column stores and multi-model data stores.

Although NoSQL systems do not require the developers to specify a rigid schema like Relational Database Management Systems (RDBMS) do, modeling the data for better performance is highly imperative [1]. This has sparked interest in the area of data modeling in NoSQL for performance optimization. The data model of the NoSQL system should be in such a way that they take advantage of the features that they are explicitly developed for like scalability, elasticity and high availability. NoSQL database modeling is query driven, i.e., the modeling is optimized around access patterns while relational database modeling is data-driven with emphasis on data integrity and redundancy removal.

This research focuses on developing a heuristics approach for transforming a relational schema represented by an Entity-Relationship (ER) paradigm to a NoSQL schema using graph transformations. This graph transformation involves steps that convert an ER diagram into a graphical representation, formulating a Directed Acyclic Graph (DAG) from the graphical notation, identifying the root node for the order of embedding, a heuristics based graph traversal algorithm from the root node for denormalization and creation of views and indexes. This research illustrates and validates the effectiveness of Graph Transformation with Selective Denormalization (GTSD) by comparing with another heuristics based transformation algorithm – BFS [4] proposed by G Karnitis et al. Experimental comparison is conducted using the TPC-H benchmark [5] to evaluate the effectiveness of migration methodology.

The rest of the paper is organized as follows. Chapter 2 introduces the core concepts and differences between relational and NoSQL data modeling. Chapter 3 introduces the previous approaches proposed in various literature for relational to NoSQL schema transformation. Chapter 4 explains the TPC-H benchmark and determining the access patterns through query graphs that are used in the migration algorithm. Chapter 5 describes the novel schema migration algorithm introduced in this research called Graph Transformation with Selective Denormalization (GTSD) followed by data migration methodology in Chapter 6. Chapter 7 presents the experimental results followed by conclusion remarks and scope for future work in Chapter 8.

CHAPTER 2

Background

2.1 Relational Databases Systems

A relational model is an approach of grouping data, known as tuples that are grouped into relations represented using tables. A Relational Database Management System (RDBMS) is a database management system that adheres to the relational semantics and set theory. RDBMS handles user queries based on a predefined storage model. Developers should explicitly specify what information the database should store and how it has to be stored through a schema definition along with the constraints to access related data. Once the schema definition has been specified, the database engine will determine the underlying data structures for storing the data and procedures to retrieve it [6].

This rigidity of storing the relational semantics of the data into rows and tables with constraints is a limitation when it comes to storing unstructured data. It is not always possible to organize data into rows and tables especially unstructured data like JSON. Schema changes like adding a new row to an existing table are costly operations in a relational database as the changes should be propagated to the entire table while it is not necessary for a NoSQL database like MongoDB where each document can have a different structure with different fields. Relational databases are designed to be steady data retention stores with a rigid schema.

Relational databases are designed originally to be vertically scalable, i.e., as single-node systems with means to add more disk space and memory with the growing requirement. Vertical scalability has its limitations concerning the cost-effectiveness and the amount of resources that can be added to the node, whereas there is practically no limit to the number to nodes that can be added to a system built on the paradigm of horizontal scalability like the NoSQL databases.

Similar to the join operation in relational algebra, relational databases use joins to fetch data from one or more related tables. Joins are operations performed on the common values present in the participating tables often defined as the foreign keys. Distributed implementation of the relational databases to handle large volumes of data has been an active research area in recent times. Tables are sharded and spread across multiple nodes instead of a single node. While this might solve the storage size limitation, joins over such sharded tables involves a lot of data fetching and movement over the network especially if it involves a large number of rows. Distributed joins are resource intensive and usually don't scale well owing to more data movement and communication overhead of participating nodes in the cluster.

Relational databases are designed to be ACID compliant, i.e., the database transactions in a relational model adhere to the properties of Atomicity, Consistency, Isolation, and Durability. These properties are explained below [7]:

Atomicity: A database transaction consists of multiple operations. Atomicity guarantees such operations are treated separately as individual units which either complete or fail.

Consistency: Consistency ensures that the database is not left in an invalid state and hence prevents database corruption.

Isolation: Isolation ensures that the concurrent execution of multiple transactions does not leave the database in an inconsistent state, i.e., it provides isolation to multiple concurrent transactions as if they were executing sequentially one after the other.

Durability: Durability ensures the state of a transaction as complete even in the event of a system failure.

Integrity constraints are used to ensure that the data is ACID compliant along with measures like resource locking. Relational databases have catered the requirements very well until the big-data era. Guaranteeing ACID properties in a distributed implementation of relational models has complications that need to be solved like network failures and high bandwidth required for communication. ACID properties can be enforced even in distributed databases by implementing methods like two-phase locking to ensure global serialization [8]. But locking has the limitation of holding the resources until the transactions are completed which could be a problem in an environment with limited resources.

2.2 Not Only SQL (NoSQL) Database Systems

Not Only SQL paradigm came into existence with a promise of better scalability, availability and query performance of data-intensive applications [9]. Organizations are shifting towards NoSQL databases to overcome the limitations of relational modeling. For example, migration from Oracle to MongoDB by a company called Telefonica has improved query performance to a great extent with a 50% reduction in development costs and a 65% reduction in storage costs [10]. The decline in storage costs is attributed to the use of cheaper commodity hardware for scaling the NoSQL database termed as horizontal scalability.

With many NoSQL database systems emerging into the market, the choice of choosing the right database entirely depends upon the task at hand. Most of NoSQL databases can be categorized into document stores, key-value stores, wide-column stores and graph databases. NoSQL systems follow an alternative design principle to that of relational databases to account for fault tolerance and horizontal scalability. NoSQL databases transactions follow the CAP paradigm in contrast to the traditional ACID semantics of RDBMS [11].

The choice of picking a NoSQL system to be used can be assisted with the help of CAP theorem. CAP theorem [12] states that it is not possible for a distributed data store to guarantee more than two of the following features simultaneously:

Consistency: read requests receive the most recent value or an error when read from any of the nodes in the distributed environment.
Availability: every request receives a response at all-times irrespective of the correctness or consistency of data.
Partition: system continues to work regardless of failures caused by network partitioning or packet loss in the network.

After analyzing the CAP theorem and comparing it with the requirements of the application, the suitable NoSQL database for the use case under consideration can be narrowed down. The application access patterns, the frequency of reads/writes the structure of the data and the complexity of the queries are some of the primary factors that determine the choice of NoSQL for the application. Once the decision of a NoSQL database is made, efficient data modeling has to be done to ensure performance.

Document stores like MongoDB are suitable for the use cases that require flexible storage formats like JSON/BSON. MongoDB supports complex queries through aggregation pipeline and composite indexes. Figure 1 [28] classifies databases in the context of CAP theorem with MongoDB suitable for the use cases that require consistency and partition tolerance.

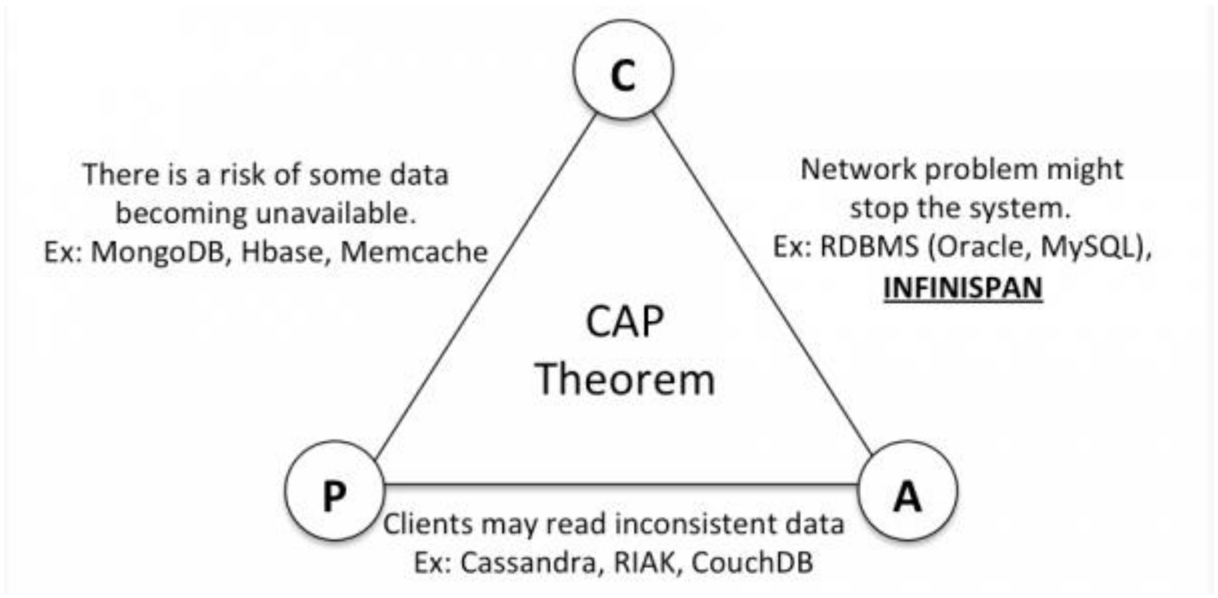

C
There is a risk of some data becoming unavailable.
Ex: MongoDB, Hbase, Memcache
Network problem might stop the system.
Ex: RDBMS (Oracle, MySQL), INFINISPAN
CAP Theorem
P
A
Clients may read inconsistent data
Ex: Cassandra, RIAK, CouchDB

Figure 1: CAP theorem (Shertil, 2016).

CHAPTER 3

Related Work

Model transformation from relational databases to NoSQL databases or data modeling of NoSQL, in general, has become an important research topic with the growing adoption of NoSQL databases. Due to the lack of migration tools and the differences in the design principles and features of different NoSQL databases, model transformation and data migration are often done manually and left to the expertise of the database administrators [10].

Different migration methodologies are proposed over time for various NoSQL systems. Most straightforward strategies involve migrating a relational schema into a NoSQL schema as it is, i.e., a one to one correspondence with the relational schema. An example would be creating a separate collection in MongoDB for every table in the relational schema. This migration results in poor query performance as joins are not always supported in NoSQL databases. Application level joins can be performed to fetch related data in such a scenario, but it is much more expensive to handle joins at the application layer especially if it involves a large amount of data.

Denormalization is often chosen to avoid join operations where related data is duplicated and stored together to improve query performance. Denormalization might improve the query performance of read operations, but might slow down updates and compromise the data integrity [13]. Care has to be taken while choosing the level of denormalization and identifying related data to be put together. The following describes some of the significant research efforts in this area.

Li et al. made one of the early attempts to define a set of rules to transform a relational schema to a NoSQL schema in his research on HBase [14]. A three-step process has been proposed to convert tables in RDBMS to HBase along with key mapping and denormalization of related data into column families. To the best of our knowledge, this is the first work that takes the cardinality of the relations into consideration. Li used the conceptual terms ‘main tables’ and ‘attached tables’ to identify what tables go together in the resultant HBase schema. These conceptual mapping of main and attached tables are left to the knowledge of the developer.

D Serrano et al. have made an extension to the work by Li [14] in mapping a relational database to HBase schema using Entity Relation diagrams [1]. This work evaluates the role of Row Keys in designing effective HBase schema. A Graph Transformation Algorithm is proposed by Zhao [15], wherein all the tables of the relational databases are represented as vertices of a Directed Acyclic Graph (DAG). This Graph Transformation Algorithm only considers table merging and does not consider referencing the related tables and produces high redundancy by duplicating the entire table through multiple levels of embedding.

An extension of the work by Zhao has been made by Sutedi et al. to reduce the data redundancy by removing the transitive dependency among the edges [16]. The transitive edges are identified and are removed, thereby reducing the number of table merges in the target schema and reducing the space requirement.

All the above approaches are based on the concept of table level denormalization, i.e., the entire tables are duplicated and merged with the related tables based on the primary-foreign key relationship. Duplicating the table as a whole may sometimes lead to excessive redundancy. A novel method of duplicating only the necessary columns based on access patterns is introduced in this work J Yoo et al. [13]. They called their process ‘Column Level Denormalization’ in contrast to the traditional table level merging. Although this approach reduces the space requirement significantly than naïve table level denormalization, this method still requires joins when querying from multiple collections and its performance is limited to a pre-defined set of queries. The performance of the queries in column level denormalization is solely dependent on the choice of columns to be denormalized and this method chose to duplicate only non-primary foreign key predicates.

In our research, the process of migration of a relational database to a target NoSQL database has been streamlined through a graph transformation. The idea of denormalization is identified as the most critical step of data modeling of NoSQL databases. The proposed schema migration algorithm outlines a systematic way of denormalization through various stages like mapping the input entity-relation schema into an intermittent graph model, selective denormalization based on access patterns which include redundancy removal. The unique aspect of this algorithm is that it takes the rapid growth of data into consideration and evaluates its effect on schema migration and query performance. The use of views and indexes and their impact on the migration are also investigated. Most schema migration techniques produce multiple intermittent target physical models and compare them based on certain evaluation criteria based on access logs. The migration algorithm in this project considers the access patterns from the very first stage of schema migration for reducing redundancy.

CHAPTER 4

Transaction Processing Control Benchmark

The Transaction Processing Control (TPC) specifies database benchmarks for performance evaluation [17]. TPC Benchmark- H consists of a suite of ad-hoc analytical queries. TPC-H supports a set of complex queries with varying scale factors. A scale factor = N indicates a database size of N*1 GB. The TPC-H benchmark suite [5] is based on an e-commerce use case consisting of 8 tables depicted in Figure 2.

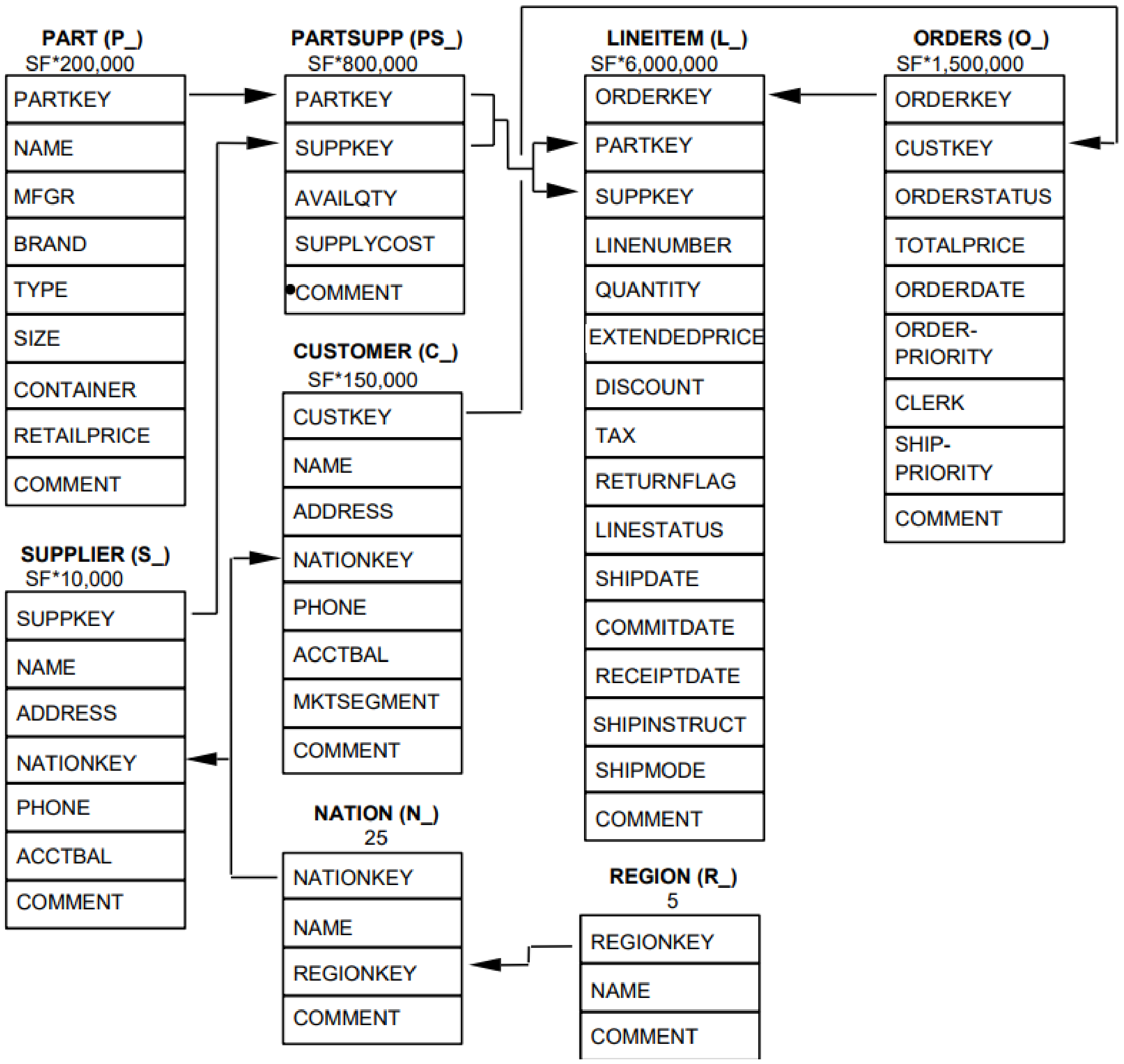


Figure 2: The TPC-H schema

Figure 2 shows the TPC-H schema with the arrows representing the relationship between two entities. The head of the arrow points to the foreign key in the entity while the tail points to the primary key to which it references. The Lineitem and Orders tables are the central fact tables where Lineitem references Orders. Lineitem has a composite foreign key (partkey, suppkey) with PartSupp. Every part can be supplied by multiple suppliers and every supplier can supply multiple parts. Customer places orders and every customer belongs to a nation. Every nation comes under a specific region. Every supplier has a nation which comes under a region.

4.1 Use Case and Workload Selection

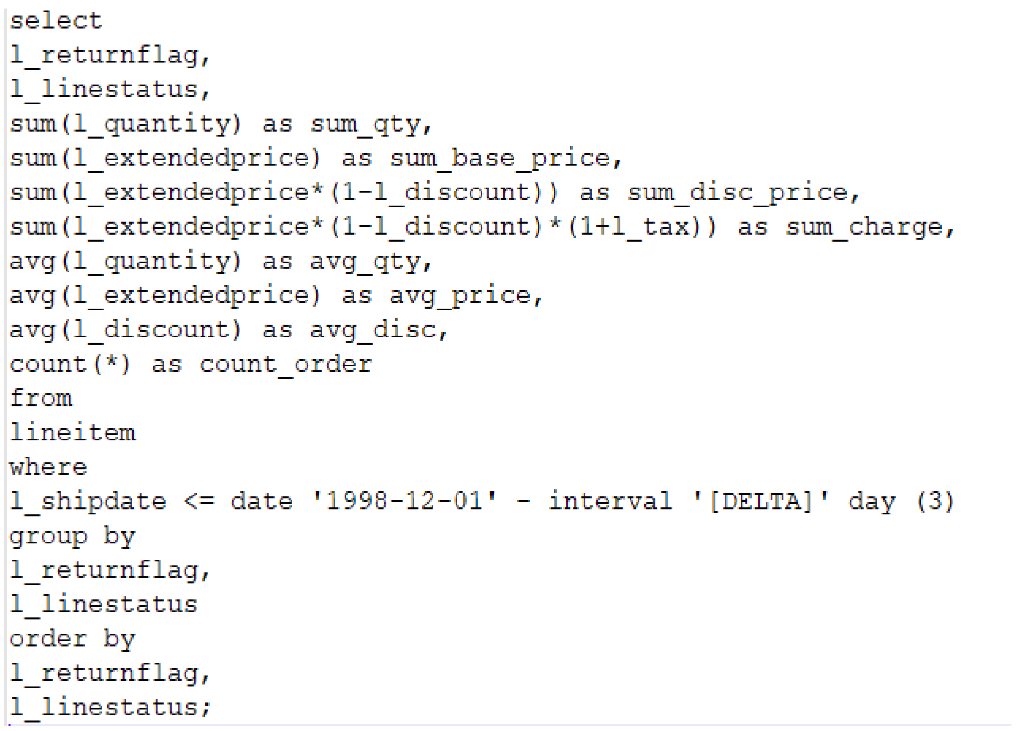
TPC is the most widely accepted decision support benchmark to evaluate database performance under varying workloads. Each query in the TPC – H benchmark aims at a specific use case and can be used to assess the query performance. TPC – H benchmark is chosen because it encompasses all the requirements to evaluate the migration algorithm like cardinality, a rapidly growing entity similar to that of a central fact table in a data warehouse and complex candidate keys representing the relationships. All the relationships in TPC – H are one to many. Methods to handle a one to one relationship has been studied in some of the researches mentioned in Chapter 3, with embedding it into the parent document in case of MongoDB [10] and including it into the column family of the referring entity in case of HBase [1]. Similarly, one to many relationships have been embedded into the parent document in [10], and many to many relations in a Relational world are modeled using references.

To the best of our knowledge, no research work has considered the velocity of data growth over a period while modeling the database. This is a significant factor in determining the performance of the queries as well as the size of the resultant database after migration. The rate of growth should be estimated in advance by database architects to design a scalable application. In case of a data warehouse, it is obvious that the Fact tables grow rapidly, while the dimension tables are rarely updated. TPC-H follows a Star Schema [18] with the Lineitem and Orders as the central fact tables that grow rapidly and the rest are dimension tables that are rarely updated. TPC – H is chosen as a candidate database to evaluate the migration methodology in this research as it covers all the cases under evaluation with varying workloads that helps in identifying the limitations of the migration algorithm with different data vs cache sizes.

4.2 TPC – H Queries

Every TPC – H query has a business use case associated with it and a rationale explaining where and why the query should be used [19]. These queries have a functional definition expressed in ANSI SQL standard. The queries simulate an actual access pattern which is one of the major deciding factors in the NoSQL data modeling. The choice of queries is made in such a way that they validate the effectiveness of the migration methodology compared with other migration techniques.

Pricing Summary Report Query (Q1):
The query Q1 of TPC-H specification [5] reports the amount of business reported. This query aggregates on RETURNFLAG and LINESTATUS. The functional query definition of Q1 according to the TPC-H documentation:

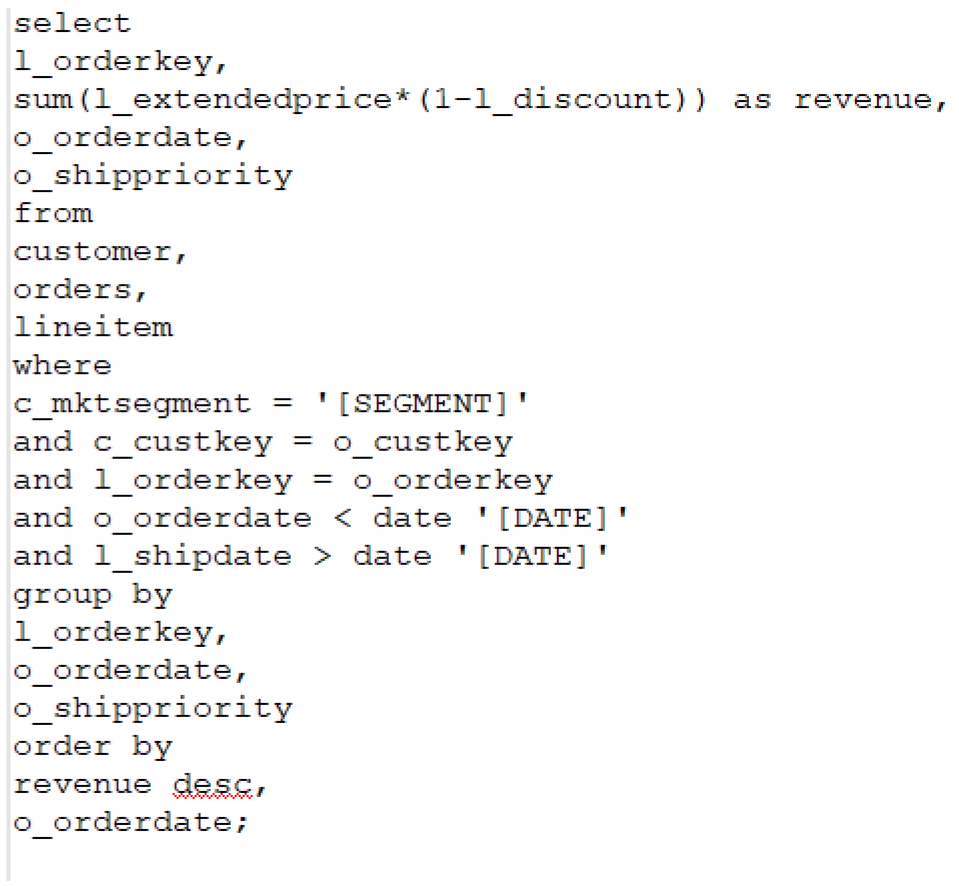


```
select
l_returnflag,
l_linestatus,
sum(l_quantity) as sum_qty,
sum(l_extendedprice) as sum_base_price,
sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
avg(l_quantity) as avg_qty,
avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_disc,
count(*) as count_order
from
lineitem
where
l_shipdate <= date '1998-12-01' - interval '[DELTA]' day (3)
group by
l_returnflag,
l_linestatus
order by
l_returnflag,
l_linestatus;
```

Figure 3: TPC – H Query 1 (TPC-H specification)

Shipping Priority Query (Q3)

The query Q3 of TPC-H specification [5] returns the top 10 unshipped orders with the highest value. This query involves three entities Lineitem, Orders and Customer, out of which Lineitem and Orders are rapidly growing or Fact tables. The functional query definition of Q3 according to the TPC-H documentation:

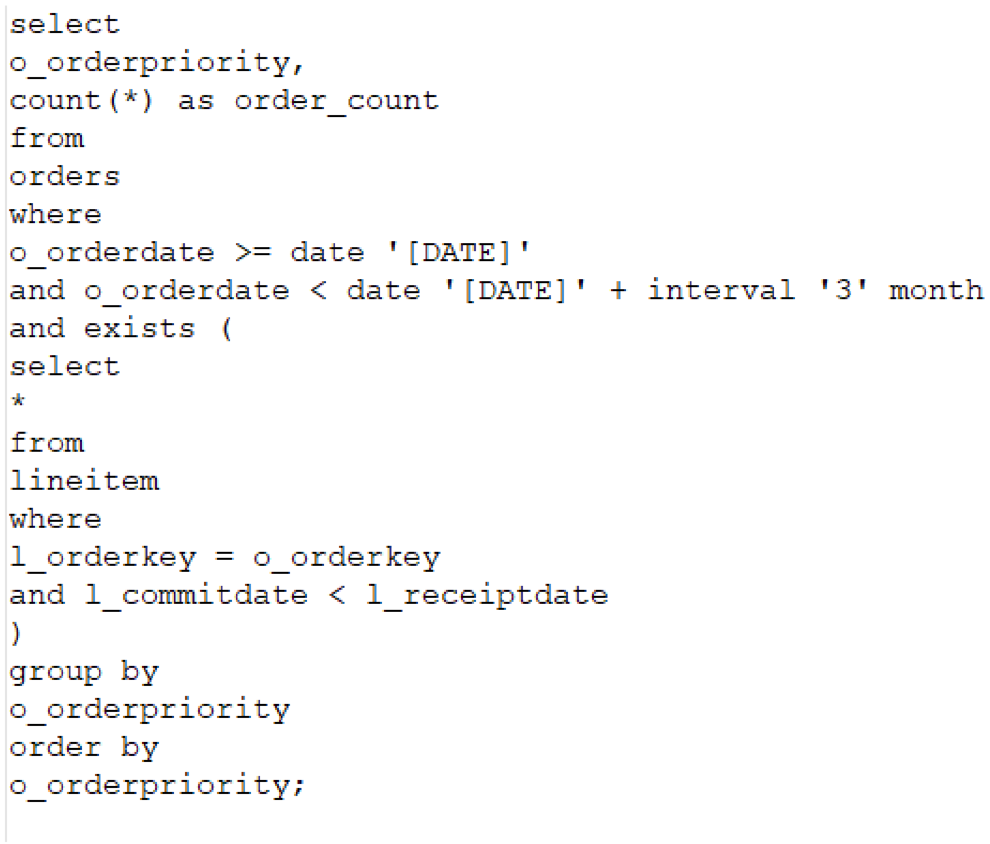


```
select
l_orderkey,
sum(l_extendedprice*(1-l_discount)) as revenue,
o_orderdate,
o_shippriority
from
customer,
orders,
lineitem
where
c_mktsegment = '[SEGMENT]'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate < date '[DATE]'
and l_shipdate > date '[DATE]'
group by
l_orderkey,
o_orderdate,
o_shippriority
order by
revenue desc,
o_orderdate;
```

Figure 4: TPC – H Query 3 (TPC-H specification)

Order Priority Checking Query (Q4)

The query Q4 of TPC-H specification [5] focuses on customer satisfaction wherein the customer receive the ordered items with the commit date which is the expected committed date by the supplier. This query tests the effectiveness of denormalization on the Orders table after migration. The functional query definition of Q4 according to the TPC-H documentation:

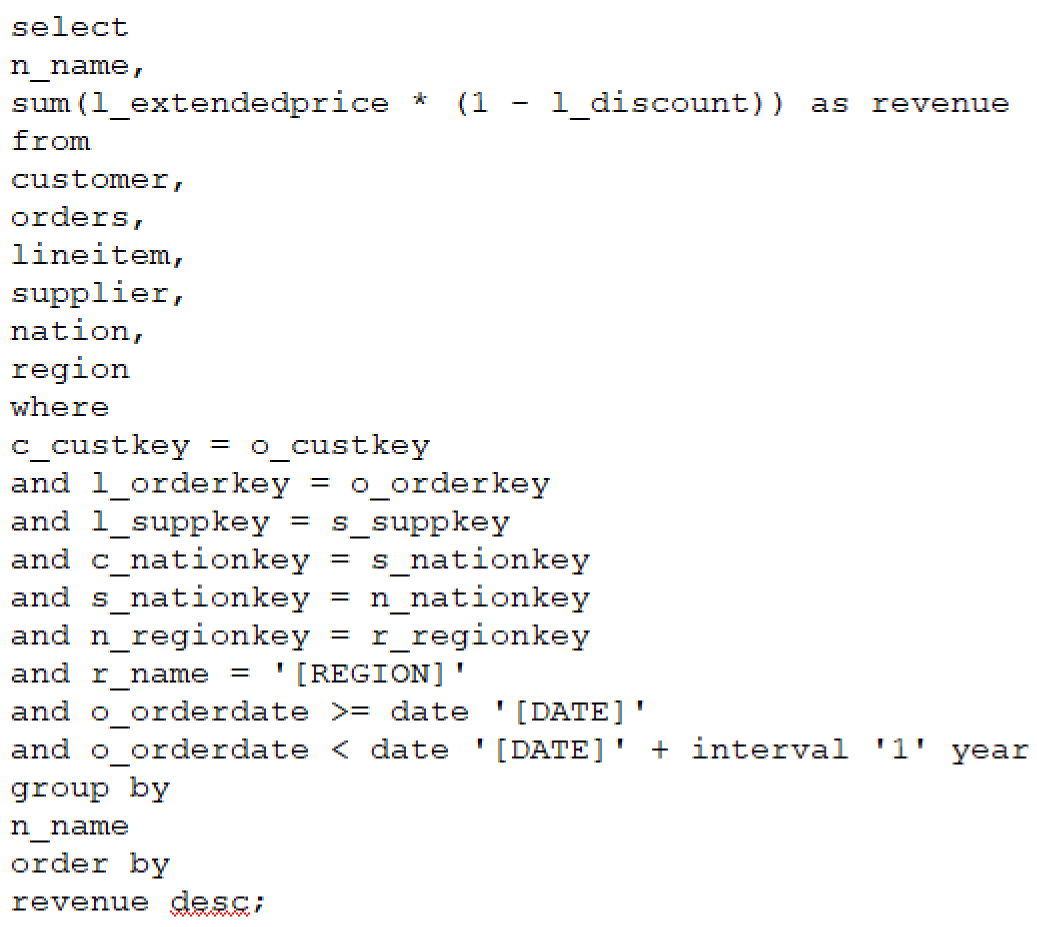


```
select
o_orderpriority,
count(*) as order_count
from
orders
where
o_orderdate >= date '[DATE]'
and o_orderdate < date '[DATE]' + interval '3' month
and exists (
select
*
from
lineitem
where
l_orderkey = o_orderkey
and l_commitdate < l_receiptdate
)
group by
o_orderpriority
order by
o_orderpriority;
```

Figure 5: TPC – H Query 4 (TPC-H specification)

Local Supplier Volume Query (Q5)

The query Q5 of TPC-H specification [5] calculates the revenue for the orders where the customers that ordered the parts and suppliers who supplied them to the customers belong to the same region. This is a highly complex query that requires fetching data from all the entities in the TPC-H schema. The functional query definition of Q5 according to the TPC-H documentation:



```
select
n_name,
sum(l_extendedprice * (1 - l_discount)) as revenue
from
customer,
orders,
lineitem,
supplier,
nation,
region
where
c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and r_name = '[REGION]'
and o_orderdate >= date '[DATE]'
and o_orderdate < date '[DATE]' + interval '1' year
group by
n_name
order by
revenue desc;
```

Figure 6: TPC – H Query 5 (TPC-H specification)

4.3 Identifying Access Patterns

The most important factor in data modeling of MongoDB is balancing the needs of the application, the data retrieval patterns and the performance characteristics of the database engine [20]. This holds true for any other NoSQL data modeling in general. While designing the data models in MongoDB, the usage characteristics like queries and processing of the data play a vital role in performance evaluation. The advantage of using a flexible data store like MongoDB is that it allows related data to be put together through embedding or referencing.

The migration algorithm developed in this research is modeled around the above TPC-H transactions, assuming that these queries form the access patterns of the target application for which data migration is to be done. Migrating data to a NoSQL database should be done by considering the queries and data access patterns of the target application. Data access patterns can be depicted using a directed dependency graph on the involved entities connected through a foreign key relationship. The direction of the arrow points from the foreign key in one entity to the primary key it references to in the other entity. Figures 7 and 8 show the query graphs for TPC-H Q3 and Q4.

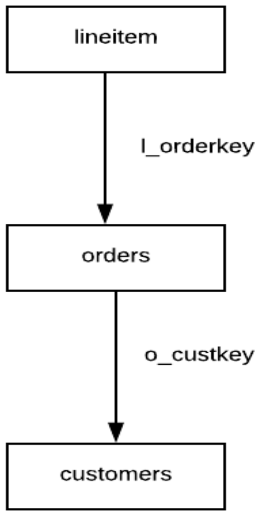


Figure 7: query graph of TPC-H Q3

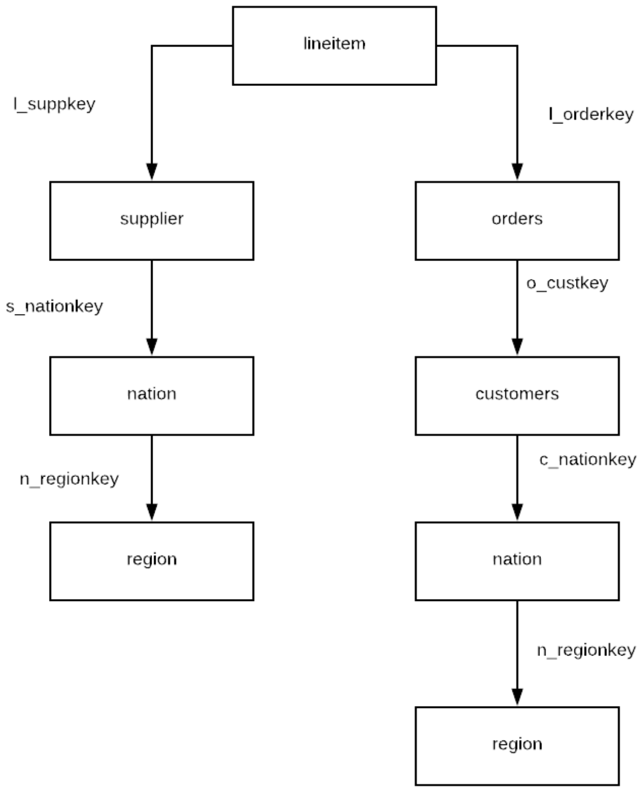


Figure 8: Query graph of TPC-H Q5

In a relational database, when multiple tables are involved in a query, join operations are performed on the foreign keys to fetch the relevant data from these tables. But most of the NoSQL databases do not support joins because joining data over multiple nodes in a distributed cluster will be expensive. In a document store like MongoDB, to overcome the limitation, data from various entities is denormalized together into a single document either by embedding or referencing. The choice of embedding or referencing a given relationship is a crucial step that determines the performance of the query and size of the resultant denormalized data. The migration algorithm developed in this project defines a systematic approach to make this choice of denormalization of related data based on the rate of growth of data. Denormalization usually results in data redundancy. This study uses non-primary foreign key predicates presented in column level denormalization [13] as a redundancy removal technique to be used along with the graph transformation algorithm.

CHAPTER 5

Graph Transformation with Selective Denormalization (GTSD)

This research adopts a heuristics based schema migration method using graphs based on the Entity-Relation (ER) diagram of the TPC-H benchmark presented in Figure 9 [29]. This migration algorithm preserves the relational semantics between entities as represented by the ER diagram with foreign key relationships. The logical flow of steps involved in GTSD schema migration is shown in Figure 10 followed by a detailed explanation.

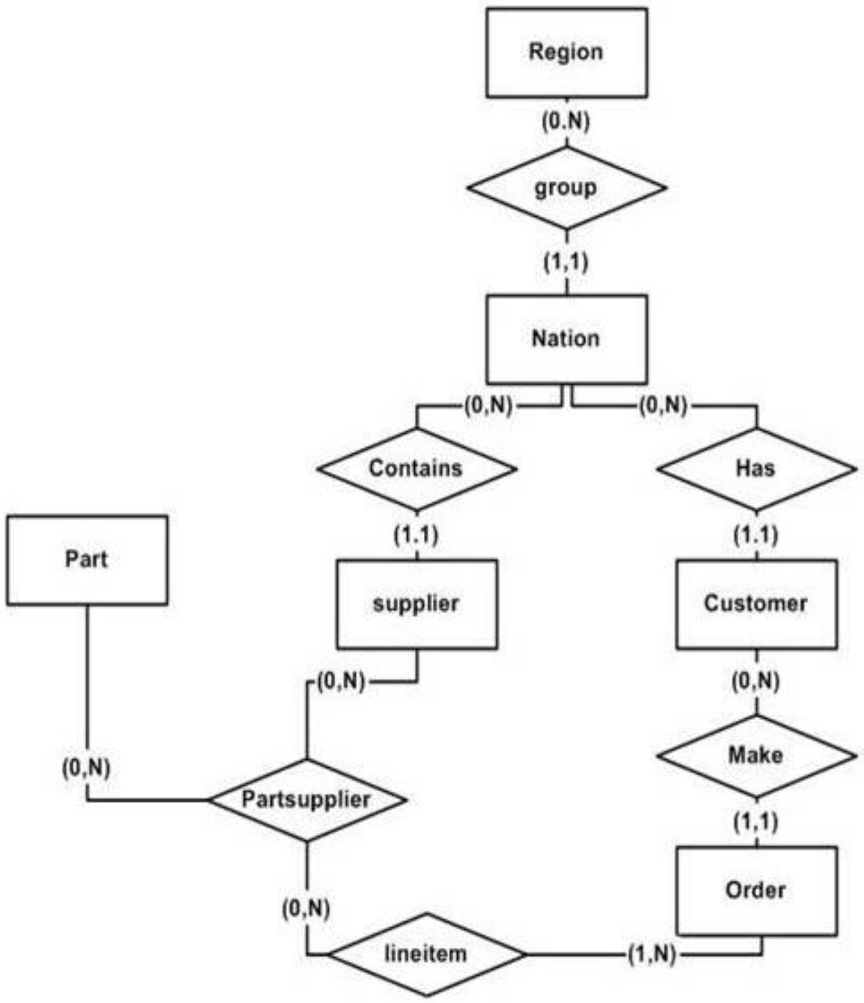


Figure 9: TPC-H Entity-Relationship diagram (Database Research Group, 2010)

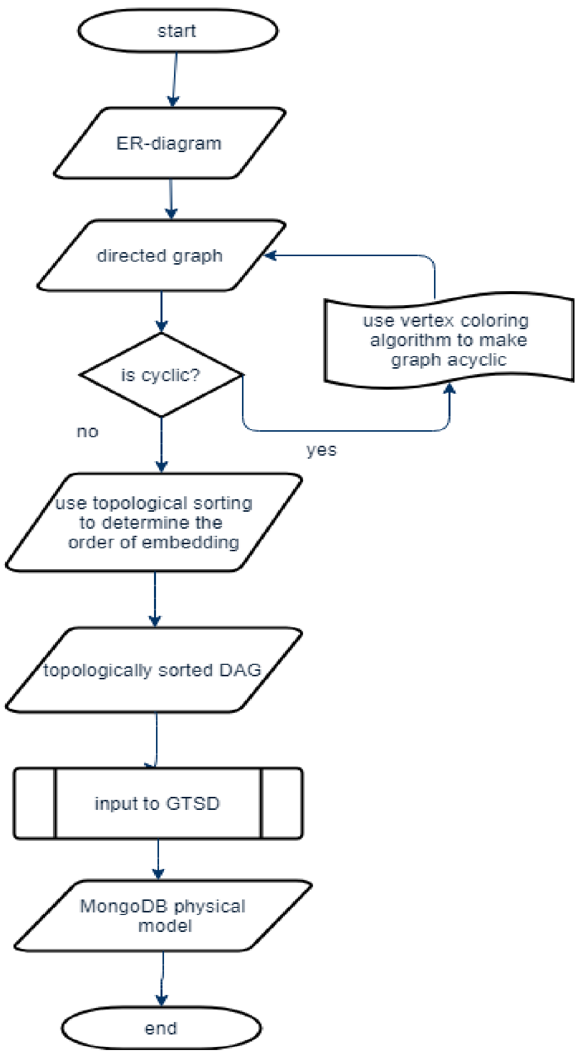

start
ER-diagram
directed graph
is cyclic?
use vertex coloring algorithm to make graph acyclic
no
yes
use topological sorting to determine the order of embedding
topologically sorted DAG
input to GTSD
MongoDB physical model
end

Figure 10: Flowchart of GTSD schema migration

The GTSD migration algorithm takes topologically sorted DAG representing the entity relationships as the inputs. The ER diagram is converted to a directed graph programmatically using an adjacency matrix [21] that describes the relationships between the vertices of the graph. The direction of the edge is always from foreign key to the primary key which represents the order of embedding in the final MongoDB model. This research chooses embedding as the default denormalization technique from referencing unless it involves a rapidly growing entity.

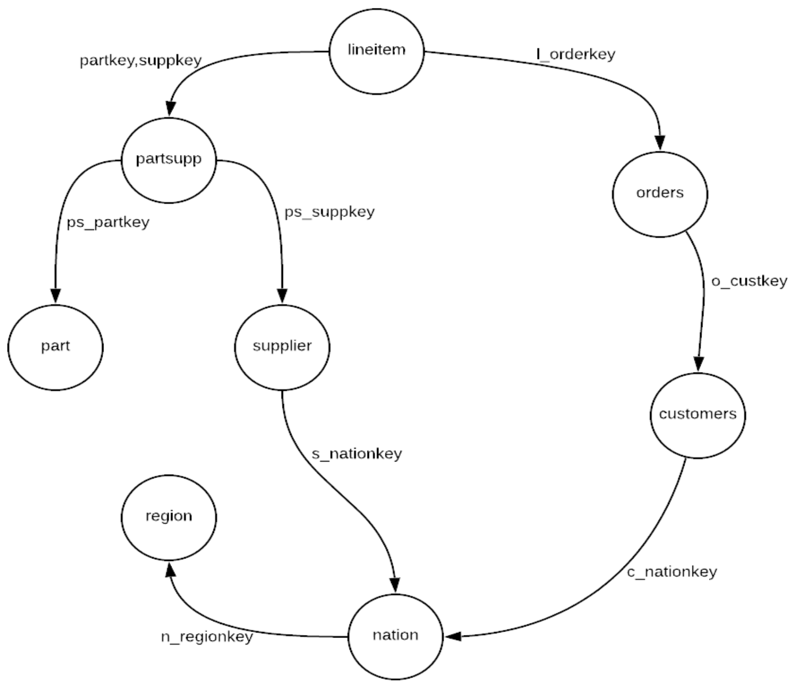


Figure 11: ER diagram to a directed graph

Figure 11 shows the directed graph from the TPC-H ER-diagram which does not have a cycle between its vertices, making it readily available for topological sorting. However, in a more complex use case, there is a high probability of the presence of a cycle in the graph as shown in Figure 12. This research addresses such an edge case and ways to handle the presence of a cycle using Vertex Coloring algorithm [23], a greedy algorithm that divides the vertices of a graph into colored sets based on some criteria. Here it is identifying the group of the vertices that form a cycle in a directed graph.

From the graph in Figure 11, every supplier belongs to a nation and every nation comes under a region. Let us assume a scenario where a particular region, for example, 'ASIA' accepts supplies only from a limited set of selectively chosen suppliers. In this case, there would be a relationship from region to supplier which is represented using an edge 'r_suppkey' as shown in Figure 12. When the vertex coloring algorithm is used on the graph in Figure 12, it returns a set of vertices that form the cycle, here it is Supplier- Nation - Region - Supplier. The graph has to be made acyclic by removing one of the edges. The choice of the edge to be removed depends upon the access pattern that can be identified through queries. For example, if no query accesses Suppliers from Region through the ‘r_suppkey,’ then this edge can be removed.

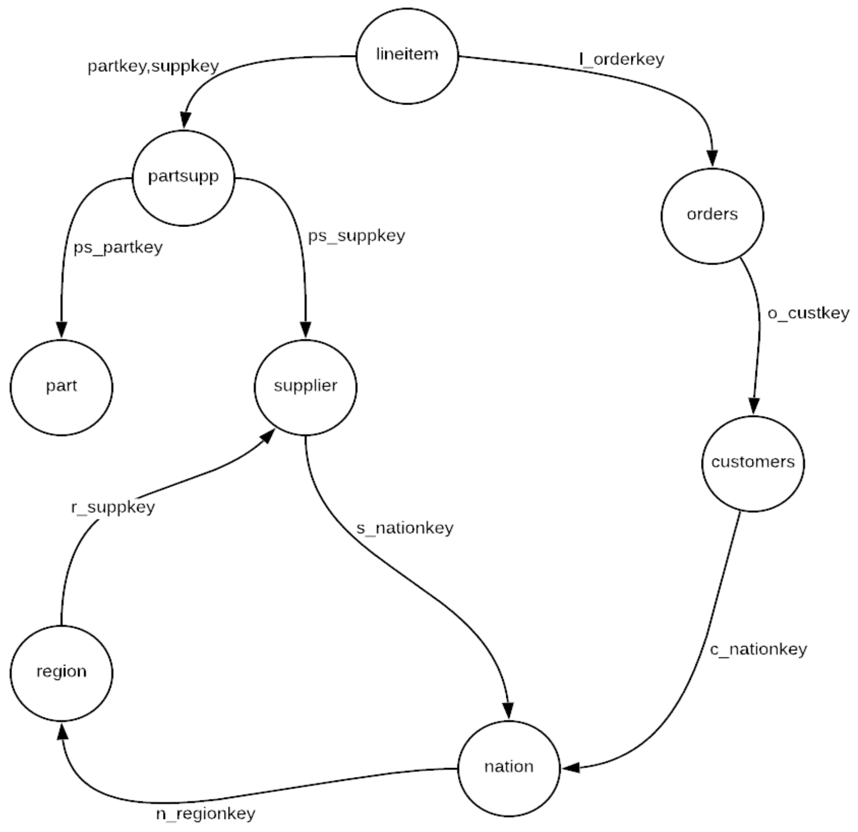


Figure 12: TPC-H directed graph with a cycle

The next step in GTSD is to traverse the DAG to selectively denormalization and decide on the type of denormalization, i.e., to embed or reference. Graph traversal can be done either using Breadth First Search (BFS) or Depth First Search (DFS) by defining a starting point for traversal. To identify the starting point, i.e., the root node of the graph, topological sorting is used. Topological sorting gives the linear ordering of the vertices in a graph [22]. Topological sort requires a Directed Acyclic Graph (DAG) to avoid infinite looping over a vertex. Topological sorting of the graph represented by Figure 11 gives Lineitem as the first vertex in the linear ordering. With Lineitem as the root, the graph traversal algorithm can be applied to identify the final physical MongoDB model.

Algorithm: Graph Transformation with selective denormalization

Input: Graph G (N,E) - A topologically sorted directed acyclic graph (DAG) where 'N' is the set of nodes represented by an adjacency matrix and E is the set of edges between any two nodes as represented by a foreign key- primary key relationship.

Output: MongoDB physical model

```
for (node n in the set of  Nodes N of DAG G):
        if(n is marked as a Dimensional table):
              createCollection (n);

        else
           for(n is marked as a Fact table)
            createCollection (n);
               Depth First Search (with root as node n);
               if (any of n's child node represented as '$c_n$' is a Fact table)
                 Identify the non-primary-foreign-key-attributes of $c_n$ and add it to C(n);
                 remove $c_n$ and it's subtree from the graph;
                 continue;
               else:
                  identify all the non-primary-foreign-key-attributes of children of n and
                  add it to C(n);
                  SelectiveDenormalize(node n, a subset of nodes that can be reached from
                  n as root through directed edges e):
                         Identify attributes of the node that are accessed in any of the queries
                         apart from non-primary-foreign-key predicates and add them to
                         C(n);

 createCollection(node n):
               return collection on node n with the auto index on '_id' field;
```

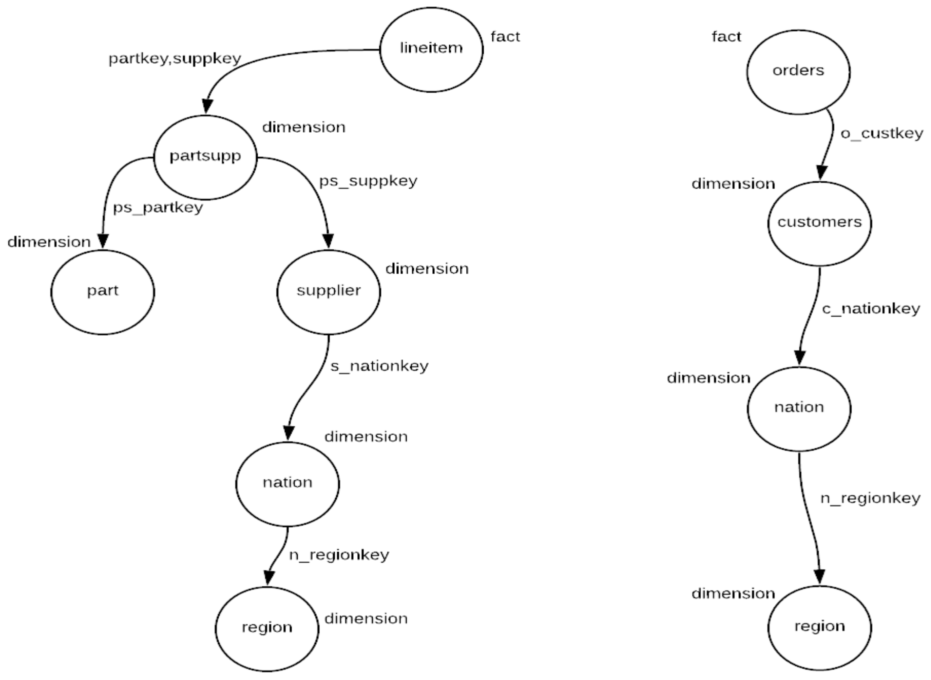


Figure 13: Sub-trees obtained from Graph Traversal with selective denormalization

Every node is attributed with metadata to be used while traversing the graph. This metadata includes the attributes of the entity, i.e., all the column names of the table, a Boolean value to represent whether an entity is a fact table or not. Being a fact table is analogous to a rapidly growing entity and a dimension table to a static or almost static entity that is rarely updated. This plays a vital role in deciding whether to embed or refer which dictates the performance of our migration. This research chooses references to model a rapidly growing entity instead of embedding owing to limitations of the MongoDB framework and performance issues explained in the experimental evaluation section.

CHAPTER 6

Data Migration

The migration algorithm in the previous section produces eight collections for eight different entities of the TPC – H schema, out of which six are normalized, which means they have a one to one correspondence with the tables in the relational schema. The two other collections, Lineitem, and Orders are denormalized. The emphasis of this research is on testing the efficiency of the migration algorithm and performance comparison with other migration strategies.

To the best of our knowledge, no tool currently available does automatic data migration into MongoDB from flat files based on a defined MongoDB physical model. TPC-H supports benchmark evaluation for various relational databases like Oracle, SQL Server, MySQL and DB2 along with support for data population. A decision support benchmark for NoSQL databases is an active research area and the need for such a tool that automatically populates data into a NoSQL with different input data models is essential.

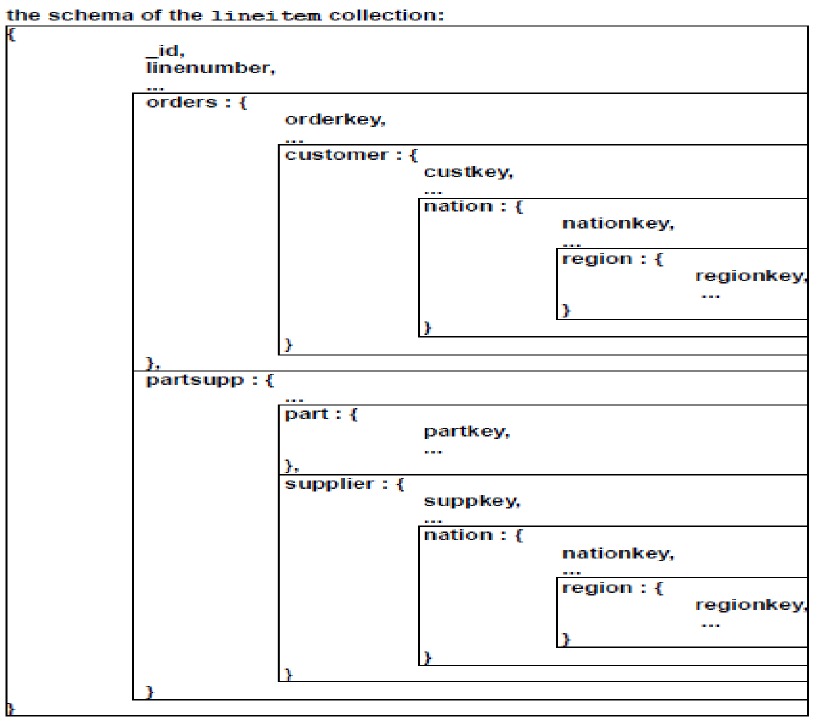
Data migration from flat files has been done manually based on a MongoDB object modeling tool called Mongoose [24]. The data has been programmatically parsed from the flat files and loaded into MongoDB based on the model generated from GTSD. The data for BFS is also parsed programmatically with multiple levels of embedding as defined in Figure 14.

CHAPTER 7

Experiments and Results

7.1 Overview of Experiments

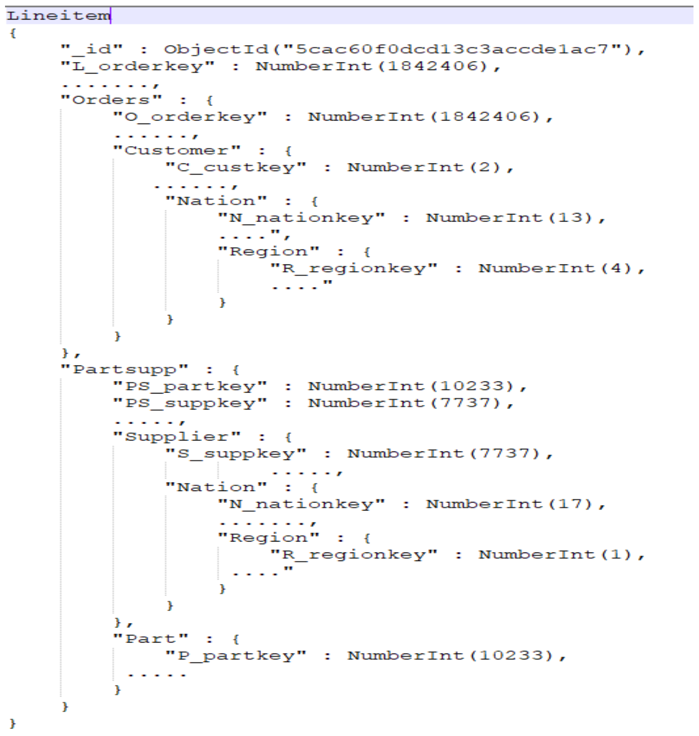
To prove that the migration using Graph Transformation with Selective Denormalization is effective, the MongoDB model obtained from GTSD algorithm as shown in Figure 16a and 16b is compared with a model derived from BFS [4], which is a variant of table level denormalization. We measured the average query execution time of the four queries shown in Chapter 4 with varying scale factors (1 and 16) of the TPC-H benchmark, and the sizes of the resultant MongoDB databases are compared. While implementing the BFS algorithm, we chose Lineitem as the root node to make an apple to apple comparison with the GTSD algorithm. The document structure obtained from BFS as shown in Figure 14 is the same as the one presented in the research by Yoo et al. [13] in their study.



```
the schema of the lineitem collection:
{
        _id,
        linenumber,
        ...
        orders : {
                orderkey,
                ...
                customer : {
                        custkey,
                        ...
                        nation : {
                                nationkey,
                                ...
                                region : {
                                        regionkey,
                                        ...
                                }
                        }
                }
        },
        partsupp : {
                ...
                part : {
                        partkey,
                        ...
                },
                supplier : {
                        suppkey,
                        ...
                        nation : {
                                nationkey,
                                ...
                                region : {
                                        regionkey,
                                        ...
                                }
                        }
                }
        }
}
```

Figure 14: Lineitem collection with BFS

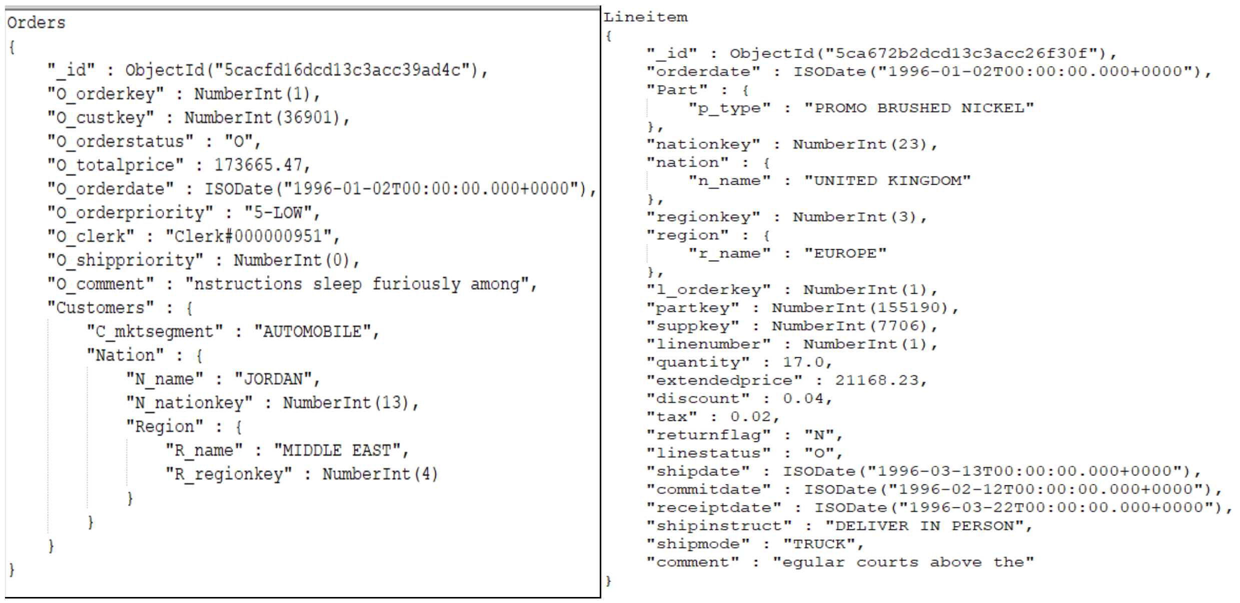
Figure 15 shows a document in the collection obtained from BFS corresponding to Figure14. The Lineitem collection embeds the orders and Partsupp collections which have other collections embedded into them.



```
Lineitem
{
    "_id" : ObjectId("5cac60f0dcd13c3accde1ac7"),
    "L_orderkey" : NumberInt(1842406),
    ........,
    "Orders" : {
        "O_orderkey" : NumberInt(1842406),
        .......,
        "Customer" : {
            "C_custkey" : NumberInt(2),
          .......,
            "Nation" : {
                "N_nationkey" : NumberInt(13),
                ....",
                "Region" : {
                    "R_regionkey" : NumberInt(4),
                    ...."
                }
            }
        }
    },
    "Partsupp" : {
        "PS_partkey" : NumberInt(10233),
        "PS_suppkey" : NumberInt(7737),
        ......,
        "Supplier" : {
            "S_suppkey" : NumberInt(7737),
                    ......,
            "Nation" : {
                "N_nationkey" : NumberInt(17),
                ........,
                "Region" : {
                    "R_regionkey" : NumberInt(1),
                  ...."
                }
            }
        },
        "Part" : {
            "P_partkey" : NumberInt(10233),
          .....
        }
    }
}
```

Figure 15: Document corresponding to BFS schema

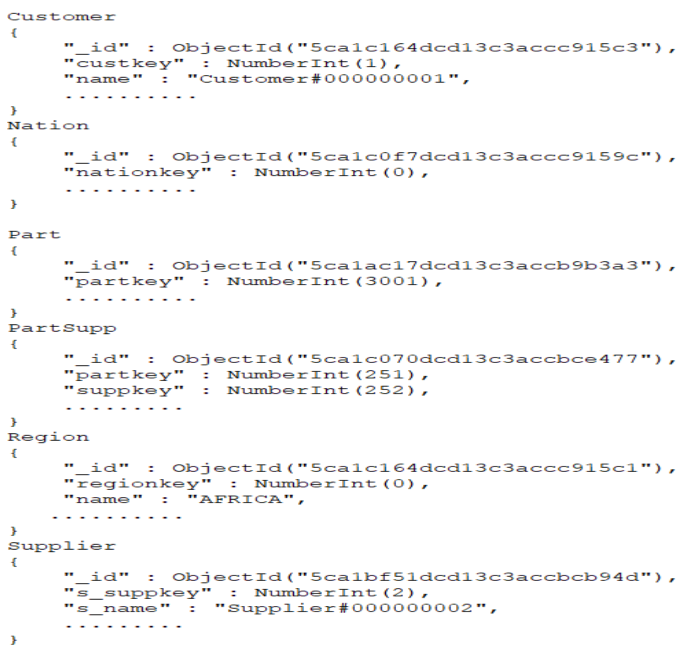
According to GTSD migration algorithm, Lineitem and Orders are two separate collections with data selectively denormalized into them. The graph traversal starts at Lineitem as the root and selectively denormalizes nonprimary foreign key predicates that can be added to Lineitem along with other attributes obtained from query graphs. When the traversal encounters Orders node, it excludes Orders along with all the child nodes of Orders node. References of Orders are created and included in the Lineitem collection. The graph traversal resumes with Orders as the root node and selectively denormalizes all the nonprimary foreign key predicates of the subtree with Orders as the root node. The collections obtained from GTSD migration algorithm are shown in Figures 16a and 16b.



```
Orders
{
    "_id" : ObjectId("5cacfd16dcd13c3acc39ad4c"),
    "O_orderkey" : NumberInt(1),
    "O_custkey" : NumberInt(36901),
    "O_orderstatus" : "O",
    "O_totalprice" : 173665.47,
    "O_orderdate" : ISODate("1996-01-02T00:00:00.000+0000"),
    "O_orderpriority" : "5-LOW",
    "O_clerk" : "Clerk#000000951",
    "O_shippriority" : NumberInt(0),
    "O_comment" : "nstructions sleep furiously among",
    "Customers" : {
        "C_mktsegment" : "AUTOMOBILE",
        "Nation" : {
            "N_name" : "JORDAN",
            "N_nationkey" : NumberInt(13),
            "Region" : {
                "R_name" : "MIDDLE EAST",
                "R_regionkey" : NumberInt(4)
            }
        }
    }
}
```

```
Lineitem
{
    "_id" : ObjectId("5ca672b2dcd13c3acc26f30f"),
    "orderdate" : ISODate("1996-01-02T00:00:00.000+0000"),
    "Part" : {
        "p_type" : "PROMO BRUSHED NICKEL"
    },
    "nationkey" : NumberInt(23),
    "nation" : {
        "n_name" : "UNITED KINGDOM"
    },
    "regionkey" : NumberInt(3),
    "region" : {
        "r_name" : "EUROPE"
    },
    "l_orderkey" : NumberInt(1),
    "partkey" : NumberInt(155190),
    "suppkey" : NumberInt(7706),
    "linenumber" : NumberInt(1),
    "quantity" : 17.0,
    "extendedprice" : 21168.23,
    "discount" : 0.04,
    "tax" : 0.02,
    "returnflag" : "N",
    "linestatus" : "O",
    "shipdate" : ISODate("1996-03-13T00:00:00.000+0000"),
    "commitdate" : ISODate("1996-02-12T00:00:00.000+0000"),
    "receiptdate" : ISODate("1996-03-22T00:00:00.000+0000"),
    "shipinstruct" : "DELIVER IN PERSON",
    "shipmode" : "TRUCK",
    "comment" : "egular courts above the"
}
```

Figure 16a: Document structure of Orders and Lineitem collections obtained from GTSD



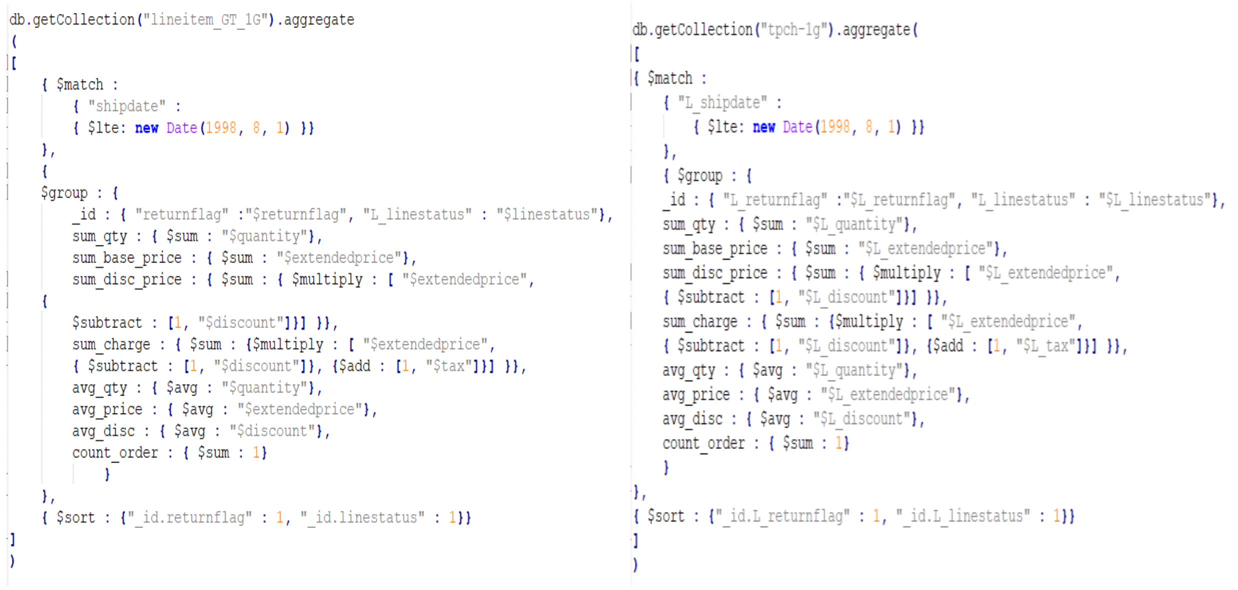
```
Customer
{
     "_id" : ObjectId("5ca1c164dcd13c3accc915c3"),
     "custkey" : NumberInt(1),
     "name" : "Customer#000000001",
     ..........
}
Nation
{
     "_id" : ObjectId("5ca1c0f7dcd13c3accc9159c"),
     "nationkey" : NumberInt(0),
     ..........
}

Part
{
     "_id" : ObjectId("5ca1ac17dcd13c3accb9b3a3"),
     "partkey" : NumberInt(3001),
     ..........
}
PartSupp
{
     "_id" : ObjectId("5ca1c070dcd13c3accbce477"),
     "partkey" : NumberInt(251),
     "suppkey" : NumberInt(252),
     .........
}
Region
{
     "_id" : ObjectId("5ca1c164dcd13c3accc915c1"),
     "regionkey" : NumberInt(0),
     "name" : "AFRICA",
    ..........
}
Supplier
{
     "_id" : ObjectId("5ca1bf51dcd13c3accbcb94d"),
     "s_suppkey" : NumberInt(2),
     "s_name" : "Supplier#000000002",
     .........
}
```

Figure 16b: Document structure of the rest of the collections obtained from GTSD

We chose scale factors 1 and 16 of TPC-H benchmark data to analyze the impact of data set size on the execution time. We used the default cache size of WiredTiger storage engine [23], which is 50% of (total memory available - 1) which comes to 7.5 GB on a machine with 16 GB memory. WiredTiger uses at most 80% of the cache allocated to it to fetch the data from disk leaving the rest of the cache to handle the process pool. So the final cache size available for WiredTiger comes to 6 GB. With scale factor 1 (data size of 1GB) the working set fits entirely in the cache. While with scale factor 16 (data size of 16GB) the working set does not fit entirely in the cache. The average execution time of the four TPC-H queries with a cache of 6 GB and the database sizes after the migration are compared.

WiredTiger is configured to use the file system cache along with its internal cache to reduce disk I/O. Collections are compressed with Snappy block compression technique in WiredTiger while indexes use prefix compression [23]. The default index of MongoDB uses B- trees as the internal data structure. Since Mongo does not support joins all the four queries in the workload are translated manually using the aggregation pipeline. For example, Figures 17 shows two versions of TPC-H query Q1, one for GTSD and other for BFS.

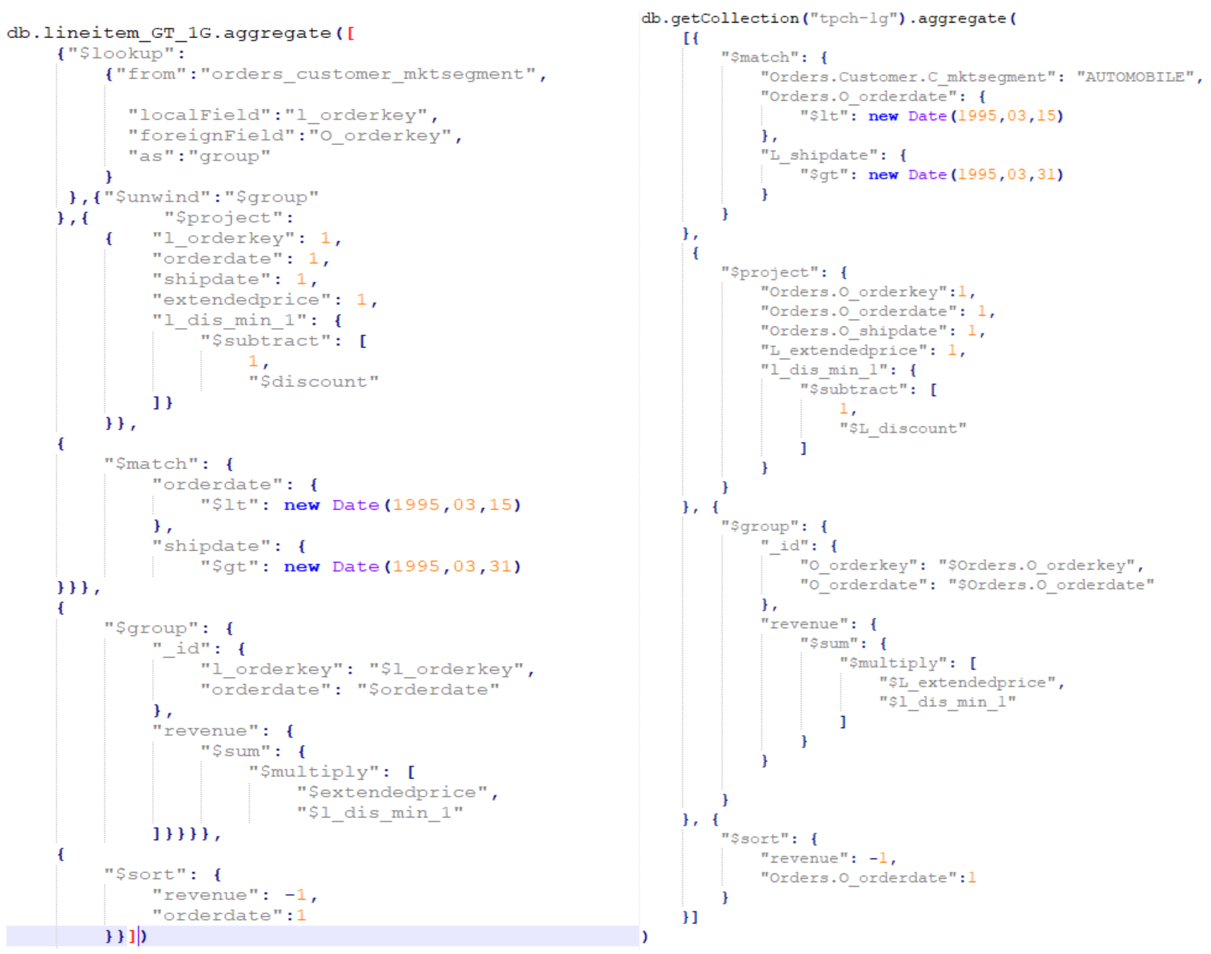


```
db.getCollection("lineitem_GT_1G").aggregate
(
[
    { $match :
        { "shipdate" :
        { $lte: new Date(1998, 8, 1) }}
    },
    {
    $group : {
        _id : { "returnflag" :"$returnflag", "L_linestatus" : "$linestatus"},
        sum_qty : { $sum : "$quantity"},
        sum_base_price : { $sum : "$extendedprice"},
        sum_disc_price : { $sum : { $multiply : [ "$extendedprice",
    {
        $subtract : [1, "$discount"]}] }},
        sum_charge : { $sum : {$multiply : [ "$extendedprice",
        { $subtract : [1, "$discount"]}, {$add : [1, "$tax"]}] }},
        avg_qty : { $avg : "$quantity"},
        avg_price : { $avg : "$extendedprice"},
        avg_disc : { $avg : "$discount"},
        count_order : { $sum : 1}
            }
    },
    { $sort : {"_id.returnflag" : 1, "_id.linestatus" : 1}}
]
)
```

```
db.getCollection("tpch-1g").aggregate(
[
{ $match :
    { "L_shipdate" :
        { $lte: new Date(1998, 8, 1) }}
    },
    { $group : {
    _id : { "L_returnflag" :"$L_returnflag", "L_linestatus" : "$L_linestatus"},
    sum_qty : { $sum : "$L_quantity"},
    sum_base_price : { $sum : "$L_extendedprice"},
    sum_disc_price : { $sum : { $multiply : [ "$L_extendedprice",
    { $subtract : [1, "$L_discount"]}] }},
    sum_charge : { $sum : {$multiply : [ "$L_extendedprice",
    { $subtract : [1, "$L_discount"]}, {$add : [1, "$L_tax"]}] }},
    avg_qty : { $avg : "$L_quantity"},
    avg_price : { $avg : "$L_extendedprice"},
    avg_disc : { $avg : "$L_discount"},
    count_order : { $sum : 1}
    }
},
{ $sort : {"_id.L_returnflag" : 1, "_id.L_linestatus" : 1}}
]
)
```

Figure 17: TPC-H Q1 - Mongo queries for Graph Transformation Algorithm and BFS

The aggregation pipeline in both versions of queries is optimized for best performance on both the collections. For example, the order of operations, $project and $match determine the amount of data to be fetched as $project fetches only a few fields and then $match filters the fields based on a defined condition.



```
db.lineitem_GT_1G.aggregate([
    {"$lookup":
        {"from":"orders_customer_mktsegment",

          "localField":"l_orderkey",
          "foreignField":"O_orderkey",
          "as":"group"
        }
     },{"$unwind":"$group"
    },{       "$project":
        {    "l_orderkey": 1,
             "orderdate": 1,
             "shipdate": 1,
             "extendedprice": 1,
             "l_dis_min_1": {
                 "$subtract": [
                     1,
                     "$discount"
             ]}
        }},
    {
        "$match": {
            "orderdate": {
                "$lt": new Date(1995,03,15)
            },
            "shipdate": {
                "$gt": new Date(1995,03,31)
    }}},
    {
        "$group": {
            "_id": {
                "l_orderkey": "$l_orderkey",
                "orderdate": "$orderdate"
            },
            "revenue": {
                "$sum": {
                    "$multiply": [
                        "$extendedprice",
                        "$l_dis_min_1"
            ]}}}},
    {
        "$sort": {
            "revenue": -1,
            "orderdate":1
        }}])
```

```
db.getCollection("tpch-lg").aggregate(
    [{
        "$match": {
            "Orders.Customer.C_mktsegment": "AUTOMOBILE",
            "Orders.O_orderdate": {
                "$lt": new Date(1995,03,15)
            },
            "L_shipdate": {
                "$gt": new Date(1995,03,31)
            }
        }
    },
     {
        "$project": {
            "Orders.O_orderkey":1,
            "Orders.O_orderdate": 1,
            "Orders.O_shipdate": 1,
            "L_extendedprice": 1,
            "l_dis_min_l": {
                "$subtract": [
                    1,
                    "$L_discount"
                ]
            }
        }
    }, {
        "$group": {
            "_id": {
                "O_orderkey": "$Orders.O_orderkey",
                "O_orderdate": "$Orders.O_orderdate"
            },
            "revenue": {
                "$sum": {
                    "$multiply": [
                        "$L_extendedprice",
                        "$l_dis_min_l"
                    ]
                }
            }

        }
    }, {
        "$sort": {
            "revenue": -1,
            "Orders.O_orderdate":1
        }
    }]
)
```

Figure 18: TPC-H Q3 - Mongo queries for GTSD and BFS

The experiments are conducted on an ASUS machine with Intel i7, Quad-core 7th generation processor with 16 GB of DDR3 RAM. It is to be emphasized that the resultant data model with GTSD and BFS is catered to answer the four queries presented in Chapter 4.

7.2 Results and Analysis

The results defined in this section are obtained from running the four queries of the TPC-H benchmark translated through the aggregation pipeline into mongo queries of both GTSD and BFS databases. Since MongoDB uses system cache, the machine is rebooted before every experiment. Every query is executed five times and the average execution time is calculated.

7.2.1 Data Size

For a scale factor =1, which represents a data size of 1 GB, the database size after BFS transformation is 5.5 GB while it is 1.6 GB with GTSD. The database size of GTSD is almost 3.6 lesser than BFS for a scale factor of 1, while the difference is nearly 4.2 times with a scale factor of 16. The growth in the number of records with the scale factor is not linear in the TPC-H benchmark. For example, there are 1.5 million orders for linenumber=1 with scale factor 1 whereas there are 6 million orders for linenumber=1 with scale factor 16.

The rate of growth of database size is less with GTSD compared to BFS due to the selectivity applied while choosing the fields to denormalize. The output of GTSD limits denormalization only to Lineitem and Orders collections, while the rest are normalized collections having a one to one correspondence with the relational schema.

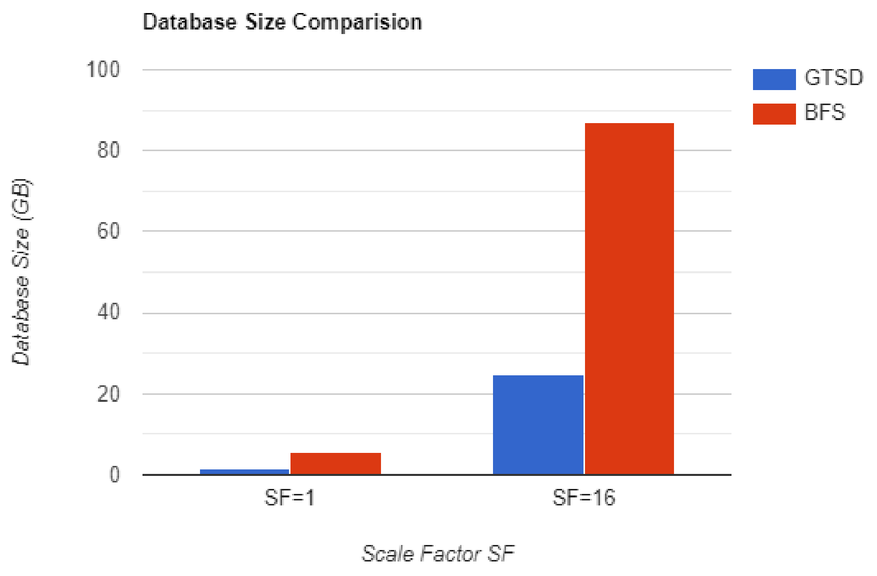


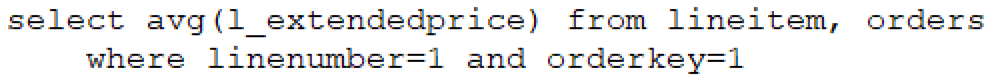
Figure 19: Database size comparison GTSD vs BFS

7.2.2 GridFS storage vs. modeling with references

This migration methodology chose the referencing method for rapidly growing entity rather than embedding it into another entity. The size of the rapidly growing entity easily exceeds the maximum BSON document size of MongoDB which is 16 GB and thus embedding is not suitable in such a use case. MongoDB imposes the 16 GB limitation on a document size to facilitate document distribution in a cluster. MongoDB has a specification to store large amounts of data exceeding the document size limitation using GridFS [26]. GridFS stores a large file in two collection, files and chunks, chunks store the actual data in 255KB blocks and metadata regarding the chunks are stored in the files collection.

Consider the Lineitem and Orders tables, the migration methodology in this research chose to refer Order references in Lineitem collections instead of embedding as these two entities are rapidly growing and will exceed the document size limitation of 16 MB. However, they can also be loaded into GridFS buckets in smaller chunks. An experiment is conducted to show that GridFS involved excessive overhead while storing the large embedded document. In this experiment, we created two

Data sets: one data set consists of one large collection where 150,000 records with Orders are embedded into LineItem and saved data through GridFS. The other data set consists of two collections LineItem and Orders where LineItem references Orders and saved data without using GridFS. We measured the execution time of a query that calculates the avg(l_extendedprice) from each data set based on the query shown in Figure 20.



```
select avg(l_extendedprice) from lineitem, orders
    where linenumber=1 and orderkey=1
```

Figure 20: Query to evaluate GridFS performance

As the results in Figure 21 shows, the execution time of querying two collections to fetch the avg(l_extendedprice) is 1.7 times faster than the denormalized data stored together in GridFS. The reason is that the 255KB chunks of data stored in GridFS have to be fetched entirely before doing an aggregation like avg(l_extendedprice) in the application logic. It can be concluded that referencing the data performs better than GridFS in a use case as described above.

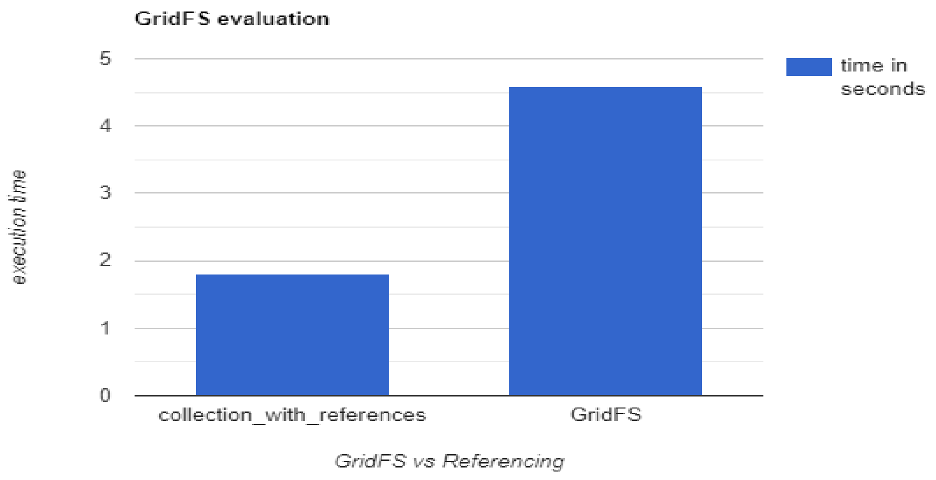


Figure 21: GridFS evaluation

However, if the use case requires only fetching the data stored in GridFS and no further operations are required, then GridFS is faster than performing joins on two separate collections. This experiment justifies that referencing is a better choice to model rapidly growing data even if it involves application level joins than using embedded documents stored in GridFS.

7.2.3 Execution Time

The average execution times of the four queries are depicted in Figures 22 and 23. For SF=1, the working set, which is the entire data required to answer a query fits in the cache. A typical read operation path in MongoDB is shown in Figure 24. When a read request is encountered by the database engine, the working set required to answer the read request is loaded into the WiredTiger cache from disk assuming that it is not already present. The data is stored on the disk using Snappy compression which has to be decompressed before processing in the RAM. The operating system caches frequently accessed files in the disk into the memory and WiredTiger accesses this OS cache before reading from the disk. Hence disk I/O is one of the major contributing factors in assessing database performance. However, for SF=1 disk I/O is not a major factor for performance evaluation since the working set fits in the memory and is already loaded in memory after the first execution.

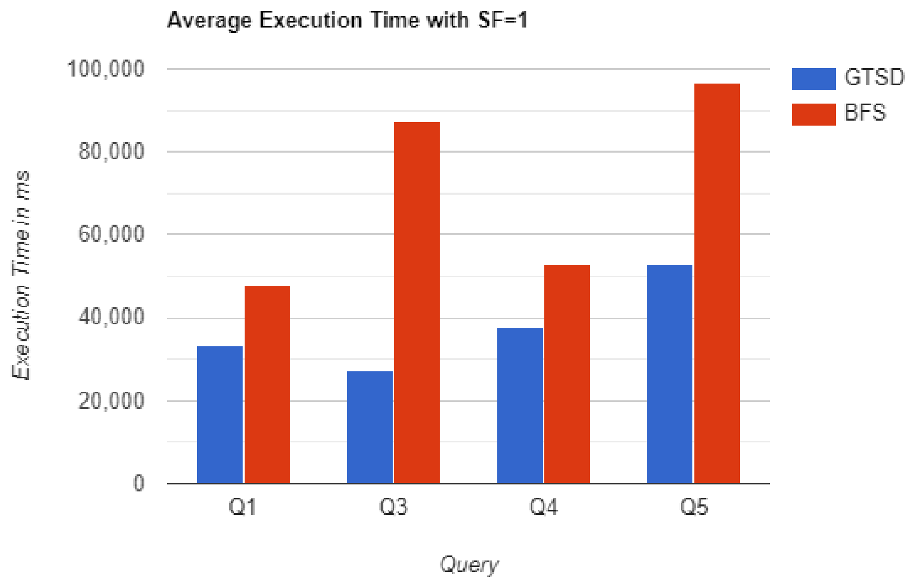


Figure 22: Average execution time for SF=1

GTSD has better average execution times for all the queries than BFS. The improvement is approximately around 1.3 times for Q1 and Q4 which are queries on a single table, but the queries Q3 and Q5 which involves queries on more than three tables have seen an improvement of around 1.9 times. Since I/O stats is not a major performance factor in this scenario, the CPU utilization dictates the performance in this case. BFS requires a lot of extract operations to fetch the data embedded deep into one single large collection. The CPU utilization is directly proportional to how deep the data is embedded. The deeper the data is embedded in the document, more unwrangling operations are needed to fetch it. In the case of SF = 16, where the working set does not fit in the memory both disk I/O and CPU utilization plays a crucial role in performance. The improvement ratio is more than two times for GTSD over BFS on queries Q3 and Q5 whereas it is around 1.7 times for Q1 and Q4. The reason between such higher improvement factors is that the size of working set in GTSD is less compared to that of BFS because the average object size in GTSD is 512 bytes whereas the average object size is around 2.5KB for BFS. The Smaller size of the working set increases cache hits and thus reducing disk access. Another factor regarding the

CPU utilization is that GTSD does not have as many unwrangling operations as BFS since the non-primary foreign key attributes are added to the topmost level of the document, whereas one has to do more number of unwrangling operations in BFS to reach for a data point embedded deep within the document. The maximum cursor timeout option has to be modified to avoid the timeout exception during execution.

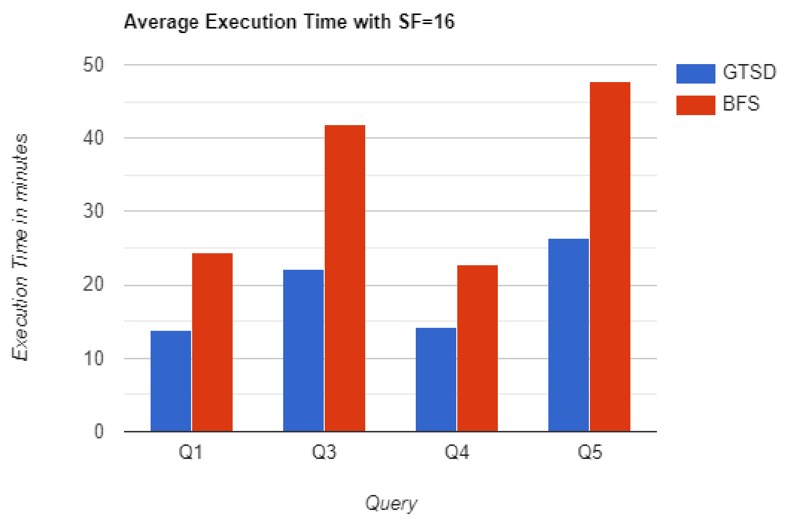


Figure 23: Average execution time for SF=16

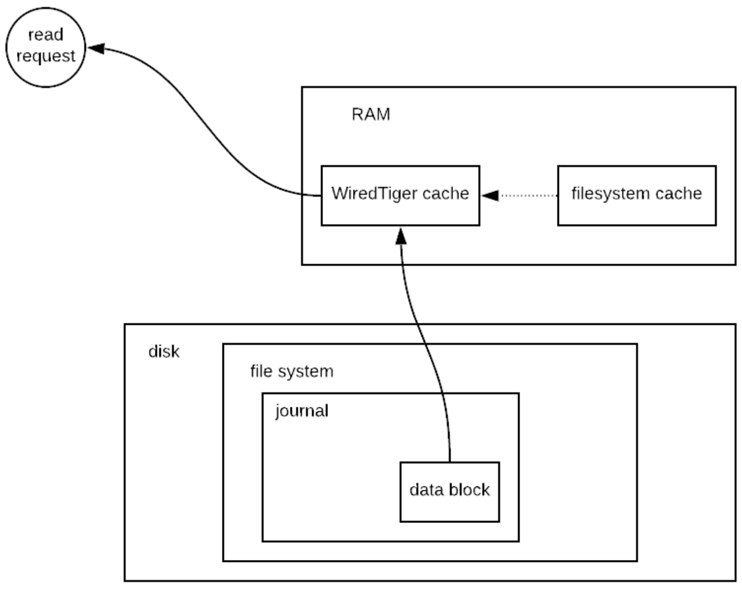
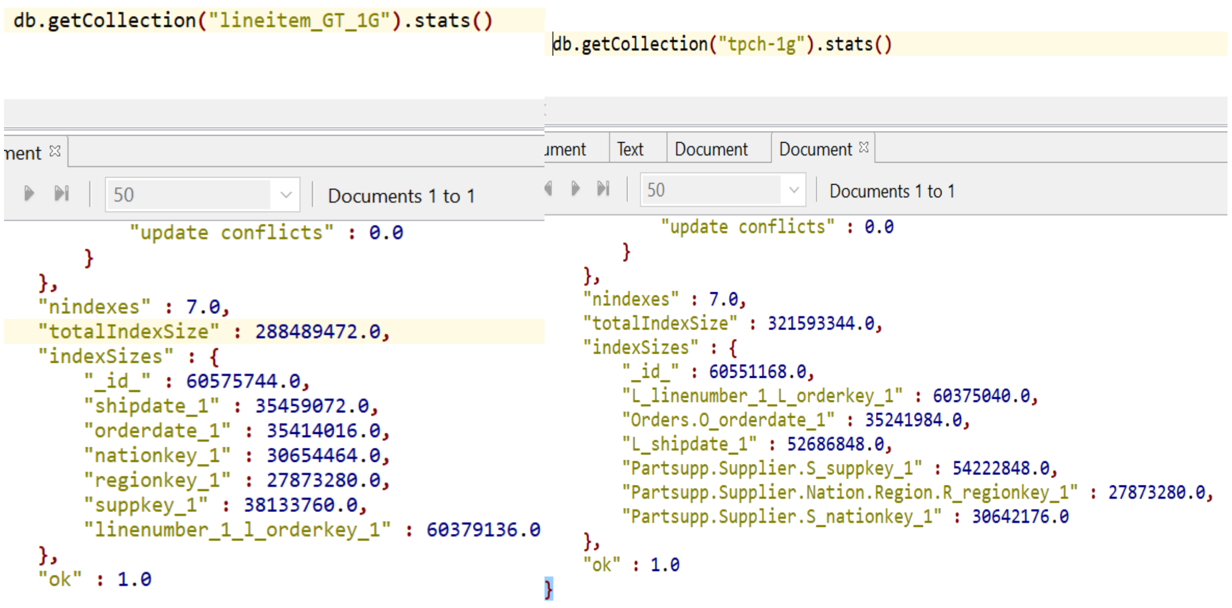


Figure 24: Read path in MongoDB

7.2.4 Size of Indexes

Indexes are created on the non-primary foreign key predicates apart from the _id attribute for every collection. The index size of GTSD is 1.2 times larger than the BFS model since BFS has only one huge collection with indexes on _id attribute and the non-primary foreign key predicates while GTSD has a total of eight collections with indexes on _id attributes and the non-primary foreign key predicates.

Collection stats shows the index sizes on individual collections of GTSD and BFS databases for a scale factor =1 are shown in Figure 25. The total index size of all the collections in the GTSD database is around 386 MB while it is around 321 MB for BFS.



```
db.getCollection("lineitem_GT_1G").stats()

nent ☒
50   Documents 1 to 1
            "update conflicts" : 0.0
        }
    },
    "nindexes" : 7.0,
    "totalIndexSize" : 288489472.0,
    "indexSizes" : {
        "_id_" : 60575744.0,
        "shipdate_1" : 35459072.0,
        "orderdate_1" : 35414016.0,
        "nationkey_1" : 30654464.0,
        "regionkey_1" : 27873280.0,
        "suppkey_1" : 38133760.0,
        "linenumber_1_l_orderkey_1" : 60379136.0
    },
    "ok" : 1.0
```

```
db.getCollection("tpch-1g").stats()

ument  Text  Document  Document ☒
50   Documents 1 to 1
            "update conflicts" : 0.0
        }
    },
    "nindexes" : 7.0,
    "totalIndexSize" : 321593344.0,
    "indexSizes" : {
        "_id_" : 60551168.0,
        "L_linenumber_1_L_orderkey_1" : 60375040.0,
        "Orders.O_orderdate_1" : 35241984.0,
        "L_shipdate_1" : 52686848.0,
        "Partsupp.Supplier.S_suppkey_1" : 54222848.0,
        "Partsupp.Supplier.Nation.Region.R_regionkey_1" : 27873280.0,
        "Partsupp.Supplier.S_nationkey_1" : 30642176.0
    },
    "ok" : 1.0
}
```

Figure 25: Index statistics from collection stats in GTSD and BFS databases

7.2.5 Impact of Views

The performance of the data model from the migration algorithm depends on how well the related data are identified and grouped so that application joins can be avoided. Sometimes it is inevitable to avoid joins even with the best modeling practices. Joins operations in the application layer are expensive, and thus it is a common practice to delegate this task to the database through materialized views. MongoDB does not support materialized views as of version 4.0.9, but the option to create materialized views is in the active pipeline. A simple expansion of the command for creating a view [27]. A new materialized: <boolean> option which defaults to false would allow the view to be created as a materialized view as shown below.

```
db.runCommand( { create: <view>, viewOn: <source>, pipeline:
<pipeline>, materialized: <boolean> } )
```

MongoDB 4.0.9 supports virtual views with disk access using the $lookup aggregation. The $lookup is essentially a left outer join on two collections. In Q3, there is a $lookup on the view created on the data from Lineitem and Orders, which are rapidly growing collections. Fetching data by joining such collections in the application layer is not an ideal practice as it requires intermittent storage on either the disk/cache. Materialized views fare well in such a scenario as they pre-fetch and store data that is needed on the disk as physical objects, decoupling the application and database layers. For Q3, a left outer join on the Lineitem and Orders collections are made to pre-fetch the customer market segment from the Orders collection and is stored in the view. Virtual views in MongoDB use the indexes defined on the underlying collections, here Lineitem and Orders while executing. Virtual views are not stored physically on the disk and require periodic execution of the aggregation pipeline to fetch updated data from the underlying collections.

CHAPTER 8
Conclusion/Future work

In this research, we proposed a heuristics based schema migration algorithm from a relational database to MongoDB called Graph Transformation with Selective Denormalization. Our method reduces join operations and disadvantages of BFS schema migration using the notion of selective denormalization that considers the data access patterns to denormalize. Furthermore, our approach addresses the rate of growth of data and its impact on denormalization and suggests guidelines to model rapidly growing data.

Experimental results show that our method significantly improves query performance with lesser hardware requirements compared with BFS schema migration. Future work can include an extension of GTSD to other NoSQL families like wide-column and graph stores. The creation of a benchmark suite that supports multiple NoSQL databases with automatic data population with different input schema could be another extension to the data migration methodology used in this research.

LIST OF REFERENCES

[1] D. Serrano, D. Han, "*From Relations to Multi-Dimensional Maps: Towards A SQL-HBase Transformation methodology*" – 2015 IEEE 8th International Conference on Cloud Computing

[2] S. Madden, "*From databases to big data*", IEEE Internet Computing, vol. 16, no. 3, May 2012

[3] A. D. Popescu, D. Dash, V. Kantere, and A. Ailamaki, "Adaptive query execution for data management in the cloud," in *Proceedings of the Second International Workshop on Cloud Data Management*, ser. CloudDB '10. New York, NY, USA: ACM, 2010, pp. 17–24.

[4] G. Karnitis and G. Arnicans, "*Migration of relational database to document-oriented database: structure denormalization and data transformation*," in Proceedings of the International Conference on Computational Intelligence, Communication Systems and Network, 2015, pp. 113-118.

[5] TPC-H - a Decision Support Benchmark "*http://www.tpc.org/tpch/*"

[6] Relational Model: "*https://en.wikipedia.org/wiki/Relational_model#Implementation*"

[7] Database Normalization: "https://en.wikipedia.org/wiki/Database_normalization"

[8] Distributed transactions: "*https://en.wikipedia.org/wiki/Distributed_transaction*"

[9] G Daniel, G Sunye, "*UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases*"

[10] T Jia, X Zhao, Z Wang, "*Model Transformation and Data Migration from Relational Databases to MongoDB*", 2016 IEEE Internation Conference on Big Data

[11] Eventual Consistency, "*https://en.wikipedia.org/wiki/Eventual_consistency*"

[12] CAP Theorem, "*https://en.wikipedia.org/wiki/CAP_theorem*"

[13] J Yoo, K H Lee, "*Migration from RDBMS to NoSQL Using Column-level denormalization with atomic aggregates*", Journal of Information Science and Engineering, 2016

[14] Li, "*Transforming Relational Database into HBase: A Case Study*", IEEE 2010

[15] Zhao, Lin, "*Schema Conversion Model of SQL Database to NoSQL*", IEEE 2014

[16] Sutedi, "*Enhanced Graph Transforming Algorithm to Solve Transitive Dependency between Vertices*", IEEE 2017

[17] Transaction Processing Control Benchmark, "http://www.tpc.org/*"*

[18] Star Schema Benchmark (SSB), 1P. O'Neil, E. O'Neil, X. Chen. "*http://www.cs.umb.edu/~poneil/StarSchemaB.PDF*"

[19] TPC BenchmarkTM H Standard Specification Revision 2.18.0, "*http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf"*

[20] Data Modelling in MongoDB, *https://docs.mongodb.com/manual/core/data-modeling -introduction/*

[21] Adjacency Matrix in Graph theory, "*https://en.wikipedia.org/wiki/Adjacency_matrix"*

[22] Topological Sorting in Graphs, "*https://en.wikipedia.org/wiki/Topological_sorting"*

[23] Vertex Coloring Algorithm, *"http://mathworld.wolfram.com/MinimumVertexColoring.html"*

[24] Mongoose ORM, "*https://mongoosejs.com/"*

[25] WiredTiger – MongoDB Storage, "*https://docs.mongodb.com/manual/faq/storage/"*

[26] GridFS, "*https://docs.mongodb.com/manual/core/gridfs/index.html"*

[27] Materialized views in MongoDB, *https://docs.mongodb.com/manual/release-notes/4.0/*

[28] Shertil M. (2016), Retrieved from "*https://www.researchgate.net/figure/The-CAP-Theorem-Many-of-the-NOSQL-databases-have-loosened-up-the-requirements-on_fig5_324922396*"

[29] Database Research Group (2010), Retrieved from "http://dbgroup.eecs.umich.edu/timber/ mct/er3.html"