DATA MIGRATION:

RELATIONAL RDBMS TO NON-RELATIONAL NOSQL

by

Feroz Alam

Bachelor of Science

in Computer Science

Ryerson University, 2010

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2015

Author's Declaration

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

DATA MIGRATION:

RELATIONAL RDBMS TO NON-RELATIONAL NOSQL

Feroz Alam

M. Sc. in Computer Science, 2015

Ryerson University, Toronto, Canada

Abstract

As a part of achieving specific targets, business decision making involves processing and analyzing large volumes of data that leads to growing enterprise databases day by day. Considering the size and complexity of the databases used in today's enterprises, it is a major challenge for enterprises to re-engineering their applications that can handle large amounts of data. Compared to traditional relational databases, non-relational NoSQL databases are better suited for dynamic provisioning, horizontal scaling, significant performance, distributed architecture and developer agility benefits. Based on the concept of Object Relational Mapping (ORM) and traditional ETL data migration technique this thesis proposes a methodology for migrating data from RDBMS to NoSQL. The performance of the proposed solution is evaluated through a comparative analysis of RDBMS and NoSQL implementations based on query performance evaluation, query structure and developmental agility.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Vojislav B. Misic for his valuable support and guidance in helping me to go through all the difficulties in my work. His suggestions have greatly enhanced my knowledge and skills in research and have significantly contributed to the completion of this thesis.

In addition, I would like to thank who have reviewed my thesis and have given me valuable comments to improve my thesis. Also, I would like to acknowledge the support of the Computer Science Department of Ryerson University.

Finally, I would like to express my deep appreciations to my family, relatives, and friends who have motivated and supported me during these years of study.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACID	Atomicity, Consistency, Isolation and Durability
ANSI	American National Standards Institute
API	Application Programming Interface
BASE	Basically Available, Soft state and Eventual consistency
BI	Business Intelligence
CAP	Consistency, Availability and Partition tolerance
DML	Data Manipulation Language
FK	Foreign Key
IaaS	Infrastructure as a Service
NoSQL	Not Only SQL
OLAP	Online Analytical Processing
ORM	Object Relational Mapping
PaaS	Platform as a Service
PK	Primary Key
RDBMS	Relational Database Management System)
SaaS	Software as a Service
SQL	Structured Query Language

Chapter 1

1. Introduction

1.1 Motivation

Large businesses have non-trivial requirements esp. with respect to data analysis solutions. In particular, businesses with large numbers of customers and massive amounts of transactional data are sure to experience the Big Data Performance Problem [41]. When businesses need to run analysis queries to respond quickly to a successful marketing campaign, their systems tend to slow down tremendously.

Databases are among the most important pieces of enterprise applications. Meaningful information is required for business decision making. As a part of achieving specific targets, business decision making involves processing and analyzing large volumes of data, which leads to growth of enterprise databases day by day [2]. Nowadays, big companies and social network organizations need to handle huge amounts of data in their every-day operations. As for examples, CERN (European Organization for Nuclear Research) needs to produce 15 petabytes of data annually [7] and Walmart needs to handle more than 1 million customer transactions hourly that requires more than 2.5 petabytes of data [2].

Large businesses like Amazon, eBay, Target, Sears have been wrestling with this issue and coming up with their own approaches. The recently emerged paradigm of Cloud Computing [6, 10, 11] can provide a flexible cost-efficient platform for business owners to host data. According to [6] a new concept is being introduced which is termed as Database as a Service or Storage as a Service, in addition to other popular paradigms of cloud computing like Infrastructure as a Service (IaaS), Software as a Service (SaaS) and Platform as a Service (PaaS).

According to the trend of recent years, cloud computing provides novel business potentials to business providers and their users by offering cost saving features which are the significant part of requirements for setting up a cloud based enterprise solution [10]. An important part of such solutions is an advanced database management system which, in order to meet the increased requirements of large group of users in terms of reliability and availability [8], requires horizontal scalability with the technology which is highly capable of managing and handling massive amounts of data distributed over many servers [2].

If we examine these approaches more carefully we notice that NoSQL (often interpreted as Not Only SQL) is often used for storing Big Data. This is a new type of database which is becoming more and more popular among large companies today. NoSQL provides simple scalability and improved performance relative to traditional relational databases. NoSQL solutions take advantage of 'Cloud Computing' meaning that database servers are delivered as a service, typically over the Internet [41]. NoSQL databases support dynamic provisioning, horizontal scaling, significant database performance, distributed architecture and developer agility benefits [3, 8]. This approach excels at storing data structures which are changing rapidly as the business needs of the company change, unlike traditional SQL database management systems which require a rigid, carefully designed data structure which does not change over time, and when it does, most (if not all) applications accessing it need to be redone. Moreover, NoSQL systems allow data to be easily partitioned to hundreds of servers and the queries can be automatically broken down to multiple parallel processes running on multiple servers. Whenever possible, data is held 'in-memory' to make access to it faster [41].

NoSQL can be the ideal solution to meet the users' demands [5]. According to [2] the significant challenges for making up a reliable database may include scalability, high availability

and fault tolerance, heterogeneous environment, data consistency and integrity, simplified query interface, database security and privacy, data portability and interoperability (as shown in Fig.1.1). As the infrastructure of cloud and NoSQL databases are integrally elastic [9] it would be a great fit for the demand. NoSQL databases mainly emphasize OLAP (Online Analytical Processing) [4, 9] that include data mining, analytics, decision making which are the parts of business intelligence. OLAP allows getting more access to the extensive possible views of information by consolidation, drill-down, and the slicing and dicing of data.

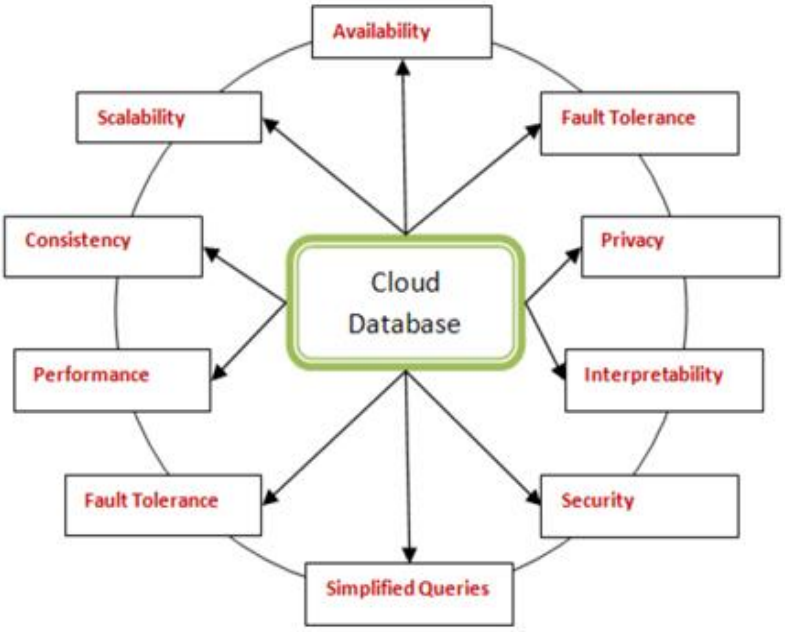


Fig.1.1: Possible Criteria for Cloud Database Formation [3]

Internet based businesses need to change their requirements continuously [25] and the most important challenge with a RDBMS (Relational Database Management System) based system may include development agility in terms of responding rapid users' change request. NoSQL databases offer simple data access features that ultimately provide development agility by reducing the development time [5].

However, most of the existing mission critical databases are, or at least have been, implemented using traditional relational database technology, often collectively referred to as RDMBS-based databases. The problem is, then, how to migrate the data from an RDBMS-based database to the next-generation NoSQL database. This problem is the focus of this thesis.

1.2 Problem Statement

In order to make data accessible, available and reliable to the large number of users with higher performance, the data storage requirements must include data partitioned over various servers [8] also known as Database Sharding, horizontal scalability and non-relational distributed data stores [9]. So that on the fly, a complex query can be broken down to smaller queries running on multiple server and the results can be consolidated into one result in order to respond users' request quickly. However, these features conflict with the main characteristics of RDBMS.

In addition, one of the requirements for quick access to unstructured or semi structured data is a loosely defined schema, which traditional RDBMSs do not support, especially in terms of handling huge volumes of unstructured data. Furthermore, RDBMS would be very costly to accommodate this type of requirements.

NoSQL data model that supports the requirements for above mentioned features, are different from relational model according to their structures and the way they store the information. Compared with NoSQL databases, relational databases do not support loosely defined schema or implicit schema as a part of processing unstructured or semi-structured data. Therefore the migration process of existing enterprise application data to the NoSQL database is

not a simple and easy task. The main challenge for business is to have a framework and methodology for the migration of existing traditional relational databases to NoSQL databases.

1.3 Contributions of This Thesis

This thesis has the following key contributions:

- A methodology has been proposed for the data migration process from relational databases to non-relational NoSQL databases. The methodology describes the details of the data migration steps where the Object Relational Mapping (ORM) concept is used for mapping relational data to different objects associated with the specific NoSQL data store. The approach taken in the methodology is different from existing data migration approaches as it uses a class diagram to depict a scenario of how relational data defined in the database schema to be mapped to the related objects according to the implicit schema of NoSQL data store.
- This thesis implements the proposed methodology using a C# Console Application. The implementation includes .NET as the framework, C# language, MySQL as the source relational database and MongoDB as the target database from NoSQL document database group. The proposed methodology and subsequent implementation can be served as a guideline to implement a generalized relational to NoSQL data migration tool.

1.4 Organization of This Thesis

The thesis is organized as follows:

As a part of data migration process from traditional RDBMS to NoSQL data store, Chapter 2 provides a detailed conceptual idea about RDBMS basics along with basic details of NoSQL data store followed by their comparative analysis. This chapter also consists of a literature review that includes the existing related works.

Chapter 3 provides a detailed overview of the proposed solution for the data migration process. The data migration process includes choosing source and target databases (RDBMS and NoSQL), technology for implementation, selection and overview of a test case, proposed methodology for the test case, and finally implementation of the test case.

Chapter 4 provides a detailed overview on the evaluation of the migration process based on some different measures. These measures include verification of migrated NoSQL MongoDB data with original relational MySQL data, comparison of data loading performance, comparative analysis on development agility in response to change request and comparison of the complexity vs simplicity of the structure of data query technique.

Chapter 5 is the concluding section that summarizes this thesis works followed by a brief guideline for the future scope of works in this area.

Chapter 2

2. Background and Literature Review

2.1 Background

The prime parts of the IT Enterprises are the databases that support the back end of their applications and over the last 40 years RDBMSs have played a leading role to dominate the IT enterprises [3] with their relational modeling, data integrity and powerful query techniques. Fig.2.1 shows the exponential growth of data in response to various types of transactions, user interactions and observations required for different ranges of IT solutions from ERP to Big Data applications. Nowadays enterprises require managing and handling of huge amounts of data for their business analysis in order to compete with the rapidly growing business market and meet the increasing users' requirements. Generated from various types of users' interactions, the large amounts of data are mostly unstructured as they include large varieties of data. This new generation of data is termed 'Big Data'. Big Data can be described as the massive amounts of data which is collected over time and unable to perform business analysis using common database management tools within the tolerable time because of the size [2]. It requires de-normalization of data set and loosely defined schema for processing unstructured big volumes of data which is opposed to the characteristics of RDBMS. RDBMS is restricted to a well-defined schema in order to store data following normalized model that can be joined to perform complex query.

Non-relational databases nowadays named NoSQL databases are being considered as the alternative to RDBMSs for cloud scale solution as they are capable of handling large amounts of data with high operational availability, scalability [12], consistent performance and rapid application development [13]. This chapter provides conceptual ideas about the basic details of

Relational Databases and Non-relational Databases with their comparative analysis followed by a literature review that includes the existing related works.

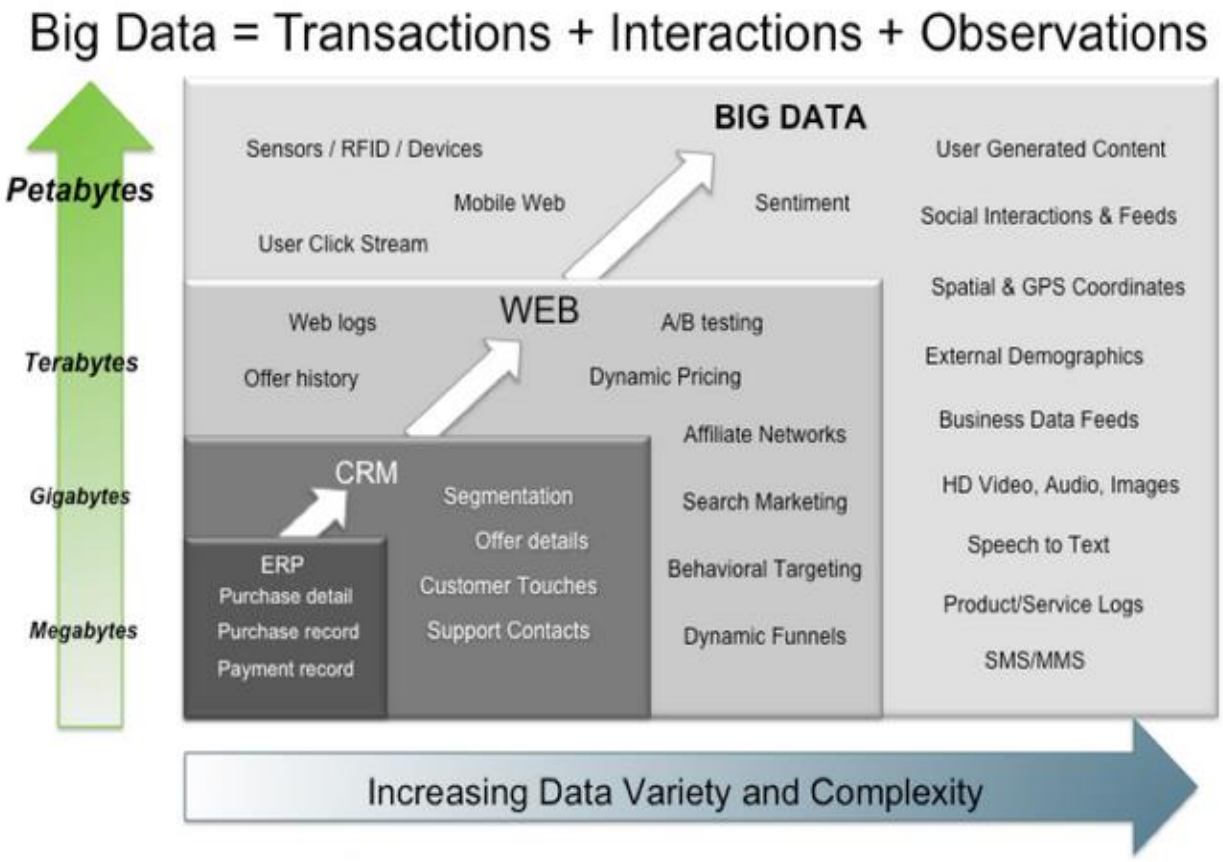


Fig.2.1: Exponential growth of data leads to evolve large volume data termed as Big Data. Source: http://hortonworks.com/blog/7-key-drivers-for-the-big-data-market/

2.1.1 Overview of Relational Database

Database management system is a methodology used by different organizations for managing and handling their data in efficient way with the help of different data model [5]. Among different data models including hierarchical model, network mode, object-oriented model, relational model and associative model, relational is highly efficient, most successful and

adoptable by professionals [5]. The idea of the relational model was introduced and presented by E.F. Codd in 1970 [22].

In a relational model data is stored in different tables through relationships which can be then accessible using these relationships. This relationship is called 'Entity Relationship' [23] which can be logically in the form of one-to-one, one-to-many or many-to-many. **Fig.2.2** shows the sales and service activities among different entities by an Entity-Relationship diagram.

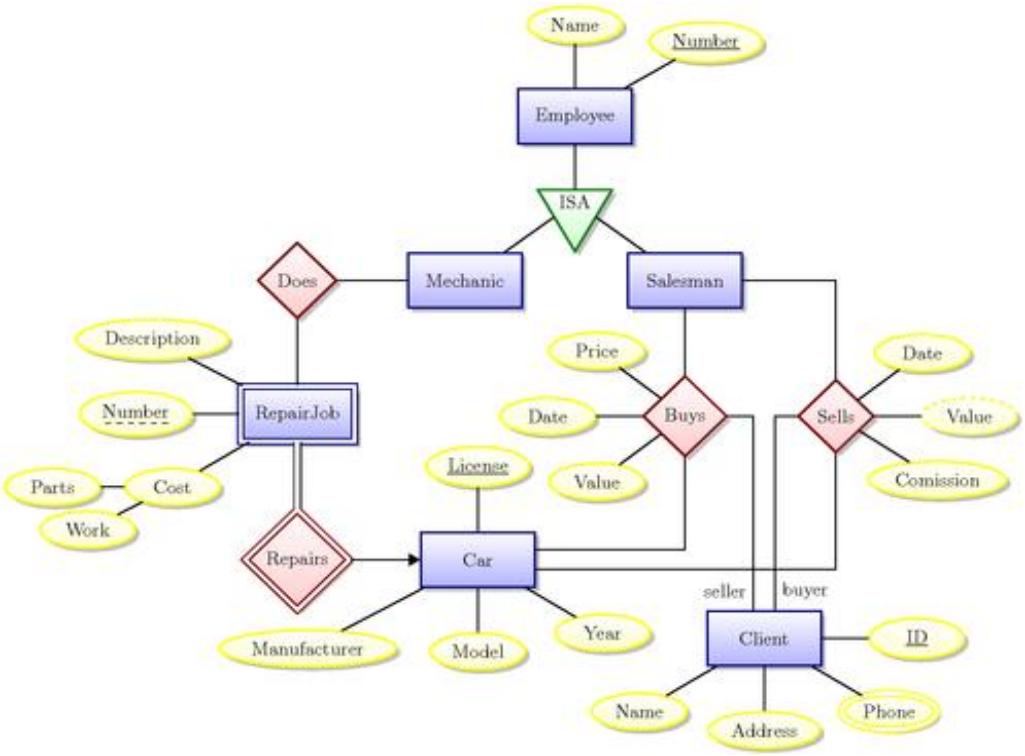


Fig.2.2: Entity Relationship (ER) Diagram of a Car Dealer
Source: http://www.texample.net/tikz/examples/entity-relationship-diagram/

The data structure in the relational model is defined by some of the components which are known as entity or table, row or tuple, column or domain, attribute or field and relation. E.F. Codd also defined relational view of data by some of the following properties [22, 29]:

1. Each row in a table represents a record or tuple.
2. All the values of attributes recorded in a table are atomic
3. All rows are unique.
4. The ordering of rows is not significant.
5. The ordering of columns is not significant.

Following are some of the concepts which are specified for RDBMSs:

2.1.1.1 Keys Concept

In RDBMS keys concept are very important as they are used to identify records and make relationship between tables. One of the important properties of relational model is uniqueness of rows or records which are also stablished using the concept of ‘Key’. There are mainly three types of keys. They are as follows:

- **Candidate Key** - A candidate key [28] is a key which can be used to uniquely identify any record in a table without referring to any other data. A candidate key can represent a single column or a combination of multiple columns. A table can have more than one candidate key. As for example in a supplier table, one candidate key can represent only the 'SupplierID' column and another candidate key can be the combination of 'SupplierID' and 'SupplierName'.
- **Primary Key** - A primary key [22, 28] is a key by which any record can uniquely be identified from a table. A table can have multiple candidate keys. But only one from those keys can be considered or chosen as a primary key. Primary keys ensure the uniqueness of record in a table and reduce data redundancy. It can be defined as a single column or a combination of multiple columns. **Fig.2.3** shows ‘order_id’ and ‘cust_id’ as the primary keys of the ‘orders’ table and ‘cutomer’ table respectively.

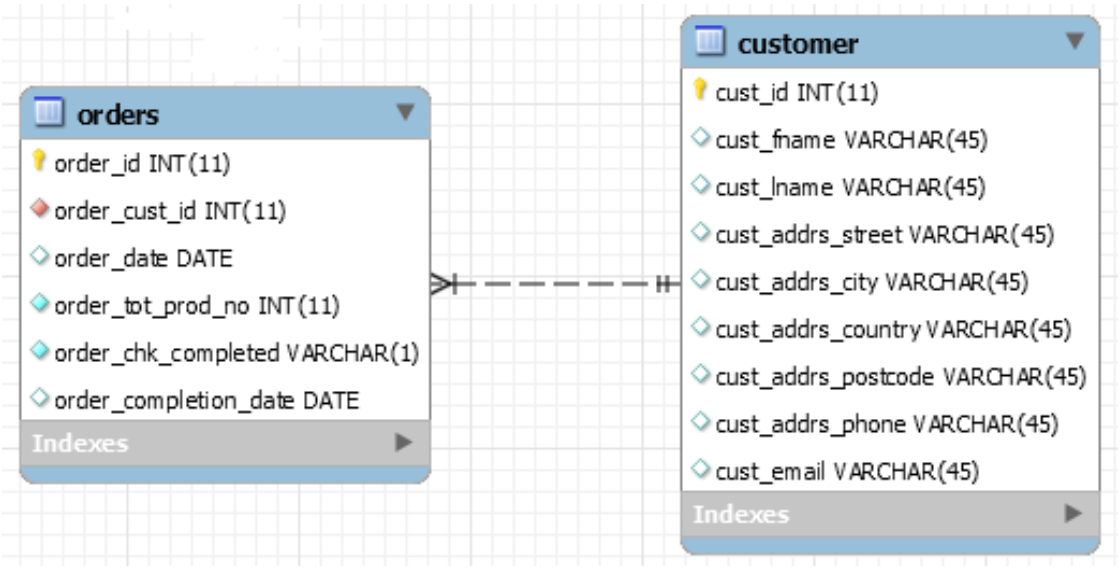


Fig.2.3: Relationship using Primary Key and Foreign Key

- **Foreign Key** - A foreign key represents a column or set of columns in a table which refers to the primary key of another table in order to uniquely identify a record of that table. In **Fig.2.3**, 'order_cust_id' in the 'orders' table represents the foreign key that refers to the primary key 'cust_id' which is defined in the 'customer' table. Using this foreign key, customer details for any order can be retrieved from the 'customer' table.

2.1.1.2 Database Normalization

Normalization is a set of rules by which tables or entities in the relational model are designed to be connected through relationships using Primary Keys and Foreign Keys in order to overcome the problem of complex domains described by E.F. Codd [22]. Normalization refers to the process of dividing tables into sub tables as a part of making database management operations easy and simple. The benefits of normalization are as follows:

- Avoid anomalies from updating.
- Optimize .queries.
- Provide data integrity.
- Increase speed and flexibility of queries, sorts and summaries.

E.F. Codd introduced three types of normalization process [29] which are termed as First Normal Form (1NF), Second Normal Form (2NF) and Third Normal Form (3NF). From more extended research [30, 31] on normalization process, two other normal forms are specified which are known as Fourth Normal Form (4NF) and Fifth Normal Form (5NF). The first three form of normalization are described as:

- **First Normal Form (1NF)** - In the First Norma Form there should not be any repeating groups. If different columns of a table contain same types of information like Item1, Item2, Item3 then it is not in 1NF. Data must be broken up into smallest possible unit. According to E.F. Codd a table is 1NF if and only if each of the values is atomic. Fig.2.4 shows that the Order table is normalized to the First Normal Form.

Order Table

OrderID	OrderDate	Item1	Item2	Item3	…
1	1970-01-01	Xxx	Yyy	Zzz	….
….	…….	…….	……	…..	…..
….	…….	…….	……	…..	…..

Order Table in 1NF

OrderID	OrderDate	Item
1	1970-01-01	Xxx
1	1970-01-01	Yyy
1	1970-01-01	Zzz
….	…….	…….

Fig 2.4: Order table normalized to First Normal Form (1NF)

- **Second Normal Form (2NF)** - Again according to E.F. Codd a table is 2NF if and only if it first fulfills all of the requirements of 1NF. Then in the Second Normal Form all non-key columns must entirely be functionally dependent on the primary key. Fig.2.5 shows the Second Normal Form of Order table.

Order Table

OrderID	OrderDate	ItemID	Item
1	1970-01-01	1001	Xxx
1	1970-01-01	1002	Yyy
1	1970-01-01	1003	Zzz
….	…….	…….	…….

Order Table in 2NF

OrderID	OrderDate	ItemID
1	1970-01-01	1001
1	1970-01-01	1002
1	1970-01-01	1003
….	…….	…….

Product Table

ItemID	Item
1001	Xxx
1002	Yyy
1003	Zzz
….	…….

Fig.2.5: Order table normalized to Second Normal Form (2NF)

- **Third Normal Form (3NF) -** According to E.F. Codd a table is 3NF if it fulfills all the requirements of 2NF. In the third normal form non-primary key fields should be dependent on the primary key rather than non-key field. Fig.2.6 represents the Third Normal Form of Order table.

Order Table

OrderID	CustomerID	CustomerName	ItemID
1	101	XYZ	1001
1	101	XYZ	1002
2	101	XYZ	1001
….	…….		…….

Order Table in 3NF

OrderID	CustomerID	ItemID
1	101	1001
1	101	1002
2	101	1001
….	…….	…….

Customer Table

CustomerID	CustomerName
101	XYZ
….	…….

Fig.2.6: Order table normalized to Third Normal Form (3NF)

2.1.1.3 Database ACID Properties

An important concept for database that ensures safe and reliable transaction process is known as ACID properties [28]. A single logical unit of work which is performed through a sequence of operations refers to as a transaction. In order to qualify a safe, reliable and consistent transaction, the logical unit of work must follows four properties – Atomicity, Consistency, Isolation and Durability (ACID). ACID properties are defined as follows:

- **Atomicity -** "Transactions are atomic (all or nothing)" [28]. In a transaction either all of its operations must be accomplished or nothing will be changed.
- **Consistency** - After the completion of a transaction all data must be in consistent state which means database will be updated only with valid data. This property ensures that any transaction will transform the database from one consistent state to another consistent state following the databases consistency rules.
- **Isolation** - "Transactions are isolated from one another" [28]. The intermediate state of a transaction cannot be accessed or seen by any other transactions. Isolation is required to maintain consistency between transactions. Modifications done by concurrent transactions must be isolated from one another.
- **Durability** - After the completion of a successful transaction the modified works or final state must permanently be saved in the system even in the case of system failure.

2.1.1.4 Data Query Language

Relational databases use a well-structured language to get access to the database and subsequently retrieve the information which is popularly known as Structured Query Language (SQL). During the 1970s Donald D. Chamberlin and Raymond F. Boyce developed a data manipulation language called Structured English Query Language (SEQL) for getting access to integrated relational databases [32]. Later SEQL was changed to SQL as it was already patented by another company.

SQL gained popularity within a very short period of time and major RDBMS vendors integrated SQL with their systems. The prime reason behind the increased popularity was

development facility. SQL reduces developers' involvement for their coding as they do not need to write additional code for data query which ultimately reduces the development cost. Considering the popularity and wide spread implementation of SQL, the American National Standards Institute (ANSI) developed a standard for SQL which is known as ANSI SQL. Based on the industry standard ANSI SQL there are different forms of SQL used by different relational database vendors. Table 2.1 shows different forms of SQL used by different relational databases.

RDBMS	Forms of SQL
IBM DB2, INFORMIX	SQL
Oracle	Procedural Language/Structured Query Language (PL/SQL)
Microsoft SQL Server	Transact SQL
MySQL	SQL
Paradox	SQL
PostgreSQL	SQL

Table 2.1: Different Forms of SQL used by Various RDBMSs

2.1.2 Overview of NoSQL Databases

Designed for distributed data store NoSQL is termed as 'Not Only SQL'. NoSQL databases have been formed in response to limitations of RDBMSs in storing and processing cloud big data particularly for large scale and high concurrency applications [14]. NoSQL databases are mainly designed to comply with the requirements of Web 2.0 applications where they need large data storage with flexible schema for storing different kinds of attributes like picture, videos, text, comments and other information [17]. They are apt with cloud scale

solutions where distributed data stores can meet the requirements of large user base in terms of reliability and availability. According to [15], key features of NoSQL databases include:

1. Ability to scale horizontally.
2. Ability to partition or distribute over many servers.
3. Comparably weaker concurrency than ACID.
4. Compared to SQL binding, simple call level protocol.
5. Ability to add new attributes to data records dynamically.
6. Capable use of RAM and distributed indexes for data storage.

Horizontal scaling, replication and distribution of data over various servers make data reading and writing operations more quickly. But at the same time NoSQL does not support ACID properties which are required for data consistency from concurrent transactions. The web based applications mainly run on the distributed environment where the main requirement is system scalability. And for a distributed system it is not possible to ensure simultaneous Consistency, Availability and Partition tolerance at the same which is stated as the CAP theorem that articulates two of them can be achieved [18]. A weaker model BASE (**B**asically **A**vailable, **S**oft state, **E**ventual consistency) replaces ACID in order to keep NoSQL data consistent and reliable. Invented by Eric Brewer and according to [18] BASE properties are described as:

- **Basic availability**: Any request will be responded with successful or failed execution.
- **Soft state**: The state of the system is 'soft' which may change over time. So due to eventual consistency changes even may be going on without any input.
- **Eventual consistency**: Eventually the database will be consistent even though it could be inconsistent momentarily.

The data structures of NoSQL databases are in different form compared to Relational data model. The following section classifies the different types of data models for NoSQL databases.

2.1.2.1 Data Models of NoSQL Databases

In this section we will categorize the different types of NoSQL data models. According to their data storing techniques, NoSQL databases are classified into following major data models:

- **Column-oriented Data Store -**Though data organization for both relational databases and column-oriented data stores are done based on rows and columns concept, but it is not necessary for column data stores to define their columns [19]. In contrast with the RDBMS where data sets are stored as rows in a table, column-oriented data store provides provisions for storing data sets as columns. Following Fig.2.7 represents an example of data sets and subsequent storing comparison between RDBMS and Column-oriented Data Store which has been taken from an online book [18]:

Data To Be Stored in	
RDBMS	**Column-Oriented Data Store**
SM1,Anuj,Sharma,45,10000000 MM2,Anand,,34,5000000 T3,Vikas,Gupta,39,7500000 E4,Dinesh,Verma,32,2000000	SM1,MM2,T3,E4 Anuj,Anand,Vikas,Dinesh Sharma,,Gupta,Verma, 45,34,39,32 10000000,5000000,7500000,2000000

Fig.2.7: Data Storing Way of RDBMS and Column-oriented Data Store

Fig.2.7 shows that each of the rows in RDBMS provides a complete data set that includes information about employee ID, first name, last name, age and salary. Whereas same data sets are organized in different columns of the column-oriented data store.

As shown in Fig.2.8, multiple attributes can be grouped in a single column in a Column-Oriented data structure which is also referred Column Family or Wide Column store [16] in order to facilitate with more complex query. According to [16] the primary uses of Column-Oriented data store include:

- Distributed data storage.
- Batch oriented large scale data processing that includes sorting, parsing, conversion etc.
- Investigative and prognostic analytics for programmers and statisticians.

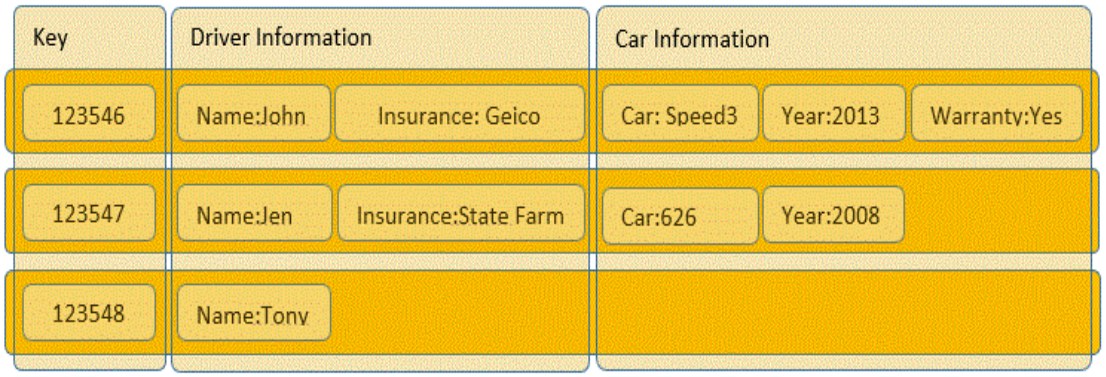


Fig.2.8: Example for the structure of Column-Family Data Store [19].

Many popular NoSQL databases like Apache Cassandra (Facebook, Twitter, Digg), HBase, Bigtable (Google Datastore), Hypertable SimpleDB (Amazon), DynamoDB are implemented by Column-Oriented data store.

- **Document Data Store -** Document-oriented data store is designed for managing and storing data in the form of documents that includes inserting, retrieving and manipulating of semi-structured data [18]. In order to make it convenient for the developer's work, the several different documents accommodated in the document data store are independent and free from a defined schema. The following example taken from [18] shows two different documents stored in a document data store to get a picture about the document data store:

Document 1	Document 2

{
"EmployeeID": "SM1",
"FirstName" : "Islam",
"LastName" : "Shamima",
"Age" : 40,
"Salary" : 10000000
}
	{
	"EmployeeID": "MM2",
	"FirstName" : "Amar",
	"LastName" : "Prem",
	"Age" : 34,
	"Salary" : 5000000,
	"Address" : {
	"Street" : "123, Park Street",
	"City" : "Toronto",
	"Province" : "Ontario"
	},
	"Projects" : [
	"nosql-migration",
	"top-secret-007"
	]
	}

Fig.2.9: Example for the structure of Document Data Store

In Document data store XML, JSON, BSON (Binary JSON) [16] formats are used to store data in each and every document. Fig.2.9 shows two JSON-format documents where 'Document 1' is a simple structured document and 'Document 2' is nested with another sub document 'Address'. 'Document 2' also contains a collection shown as 'Projects'. But none of them represents document ID which is required with the URL in order to get access to the document databases. In Document-Oriented data store a system

generated or developer defined identifier is used which is uniquely allocated for each of the documents to identify them [21]. Fig.2.10 shows how four records from a Relational data model are stored as four separate documents in a Document-Oriented data store.

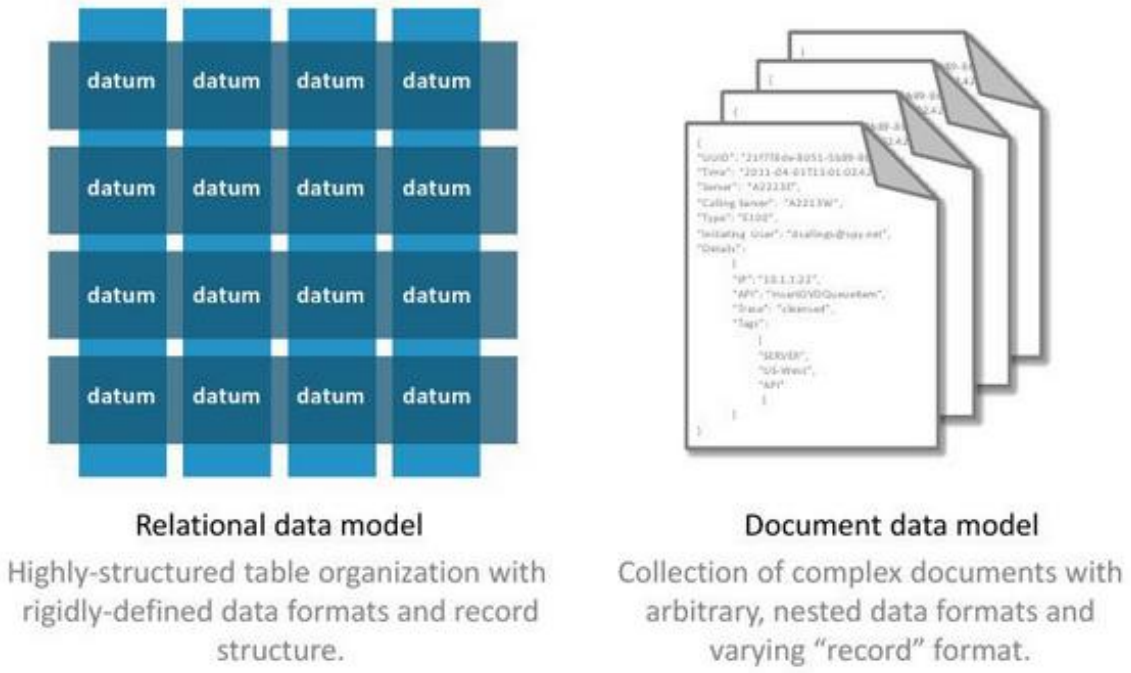


Fig.2.10: Records from Relational Model Documented in Document Data Model [43].

According to [16] document data model is mainly useful for web based applications as a part of managing and processing large scale data distributed in a network including text documents, email messages and XML documents. MongoDB, CouchDB, Jackrabbit, Lotus Notes, Apache Cassandra, Terrastore, BaseX are the popular examples of Document Oriented data store [18].

- **Key Value Data Store -** Key Value data store provides provisioning for storing data in a standalone schema free table which is also referred as a typical Hash Table against an identifier. The identifiers or keys are alphanumeric which can be system generated or developer defined [21] like document ID of Document data model. Fig.2.11 shows data

that represent Cars' attributes are stored against respective numeric keys in a Key Value store model.

Car	
Key	**Attributes**
1	Make: Nissan Model: Pathfinder Color: Green Year: 2010
2	Make: Honda Model: Odyssey Color: Grey Year: 2012

Fig.2.11: Data Stored against a Key in Key Value Store [16]

Key Value data stores are primarily useful for in-memory distributed cache [18] to facilitate retrieving data quickly. As "Key-value stores are optimized for querying against keys" [18], they are used for retrieving data from user profiles, look-up information for shopping cart system etc. Examples of Most popular Key Value data stores include **Memcached** (in-memory), **MemcacheDB** (built on Memcached), **Redis** (in-memory, with dump or command-log persistence), **Berkley DB, Voldemort** (LinkedIn, open source implementation of Amazon Dynamo), Riak [16, 18].

- **Graph Databases -** The graph databases store and represent data using graphical structures that include nodes, edges and properties as shown in Fig.2.12. Nodes represent conceptual objects those are connected by lines called edges. Edges are also used to make connections among nodes and properties. Like relational model graph databases handle relationships by traversing through edges. Using a graph algorithm Graph Databases store

data scalable over several servers with nodes and edges. Nodes and relationships are the basic parts of the graph databases where nodes are organized by properties associated with relationships and related data is stored in the nodes those also have properties.

The graph databases are primarily useful where relationships to data are more important [16]. Social networking web sites like Facebook and Twitter can be referred as the best example in this scenario as they need to store graph data as a part of making relationships among their users. FlockDB (used by Twitter), AllegroGraph, InfiniteGraph, Sones GraphDB are the examples of some of the Graph databases.

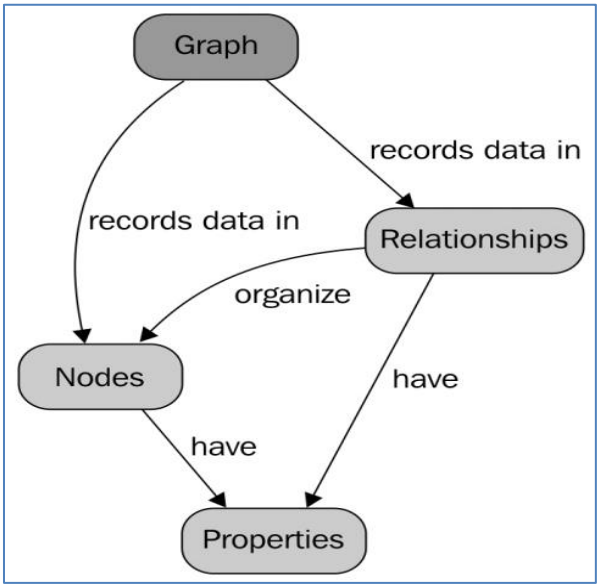


Fig.2.12: Graphical Representation of Graph Database [18].

2.1.2.2 Map Reduce Framework

In 2004 Google introduced a software framework known as MapReduce in order to process huge amounts of data distributed in a clustered environment [2]. As a programming model, MapReduce uses two functions Map and Reduce to facilitate parallel implementation that processes terabytes or petabytes of data distributed across several servers [33] within the desired

amount of time. Map function generates intermediate key-value pair data by processing key-value pair input data and then finally all intermediate value are combined against respective intermediate key using Reduce function. For getting a clear conception the MapReduce process has been represented in the Fig.2.13 where two-step functions 'Map' and 'Reduce' is used.

In the 'Map' step, input data is distributed over different worker nodes (nod1, nod2, node3) from the master node where it is divided into smaller sub-problems. The worker nodes work on the sub problem and get back with the answers to the master node. After collecting all of the answers from different worker nodes, master node then merges all of the sub-problems answers in order to form output in 'Reduce' step using reduce function.

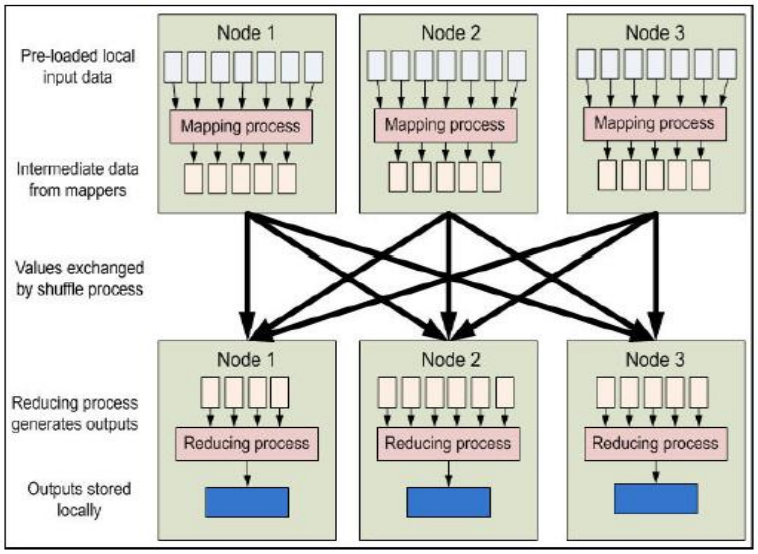


Fig.2.13: MapReduce Process using Two-Step Function [2].

2.1.2.3 The CAP Theorem

The CAP Theorem was introduced by Eric Brewer in 2000. The idea of the CAP stated as "there is a fundamental trade-off between consistency, availability, and partition tolerance" [35]. It is most necessary for every system to achieve all of the three components of the CAP Theorem

but it is impossible to achieve Consistency, Availability and Partitioning Tolerance at the same [7, 34, 35]. The three components of the CAP Theorem can be explained as:

- Consistency: A consistent systems guarantees same data is available to all of the servers in a clustered environment even at the event of any concurrent modification.
- Availability: Some version of data in a cluster must be accessible to all database clients even at the event of shutdown of a node in the cluster.
- Partition Tolerance: Even at the event of network and machine failure the system must keep working fine.

Data consistency is easily achievable in relational database systems as it supports ACID properties. At the same time horizontal scalability is a great challenge for RDBMS system. On the other hand though it is easier for NoSQL data store to achieve horizontal scalability but it can ensure lesser data consistency level due to its weaker BASE properties compare to ACID. Web based application require horizontal scalability as it deals with data distributed in many servers. It is not easy to achieve all of the three properties of the CAP Theorem. The distributed web based applications mainly ensures higher availability and partition tolerance at the cost of data consistency eventually.

2.1.2.4 Evaluation of NoSQL Databases

According to [16] a list of characteristics of NoSQL databases from four major groups with their evaluation is presented in this section. Table 2.2 shows the evaluation of several NoSQL data stores based on their Design and Features, Data Integrity, Indexing, Distribution and System.

Attributes		NoSQL Databases								
	Database Model	**Document Store**		**Wide Column Store**				**Key Value Store**		**Graph**
	Features	MongoDB	CouchDB	DynamoDB	HBase	Cassandra	Accumulo	Redis	Riak	Neo4j
Design and Features	Data Storage	Volatile Memory File System	Volatile Memory File System	SSD	HDFS		Hadoop	Volatile Memory File System	Bitcask LevelDB Volatile Memory	File System Volatile Memory
	Query Language	Volatile Memory File System	JavaScript Memcached-protocol	API Calls	API Calls, REST, XML, Thrift	API Calls, CQL, Thrift		API Calls	HTTP, JavaScript, REST, Erlang	API Calls, REST, SparQL, Cypher, Tinkerpop, Gremlin
	Protocol	Custom, Binary (BSON)	HTTP, REST		HTTP/REST, Thrift	Thrift & Custom Binary CQL3	Thrift	Telnet-like	HTTP, REST	HTTP/ REST Embedded in Java
	Conditional Entry Updates	Yes	Yes	Yes	Yes	No	Yes	No	No	
	MapReduce	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No
	Unicode	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	TTL for Entries	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	
	Compression	Yes	Yes	-	Yes	Yes	Yes	Yes	Yes	
Integrity	Integrity Model	BASE	MVCC	ACID	Log Replication	BASE	MVCC	-	BASE	ACID
	Atomicity	Conditional	Yes	Yes	Yes	Yes	Conditional	Yes	No	Yes
	Consistency	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
	Isolation	No	Yes	Yes	No	No	-	Yes	No	Yes
	Durability	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-	Yes
	Transactions	No	No	No	Yes	No	Yes	Yes	No	Yes
	Referential Integrity	No	No	No	No	No	No	Yes	No	Yes
	Revision Control	No	Yes	Yes	Yes	No	Yes	No	Yes	No
Indexing	Secondary Indexes	Yes	Yes	No	Yes	Yes	Yes	-	Yes	-
	Composite Keys	Yes	Yes	Yes	Yes	Yes	Yes	-	Yes	-
	Full Text Search	No	No	No	No	No	Yes	No	Yes	Yes
	Geospatial Indexes	Yes	No	No	No	No	Yes	-	-	Yes
	Graph Support	No	No	No	No	No	Yes	No	Yes	Yes
Distribution	Horizontal Scalable	Yes	Yes	Yes	Yes	Yes	Yes		Yes	No
	Replication	Yes	Yes	Yes	Yes	Yes	Yes		Yes	Yes
	Replication Mode	Master-Slave Replica Replication	Master-Slave Replication	-	Master-Slave Replication	Master-Slave Replication	-	Master-Slave Replication	Master-Slave Replication	-
	Sharding	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
	Shared Nothing Architecture	Yes	Yes	Yes	Yes	Yes	-	-	Yes	-
System	Value Size Max.	16MB	20MB	64KB	2TB	2GB	2GB	1EB	-	64MB
	Operating System	Cross-Platform	Ubuntu, Red Hat, Windows, Mac OS X	Cross-Platform	Cross-Platform	Cross-Platform	NIX 32 Entries Operating System	Linux, *NIX, Windows, Mac OS X	Cross-Platform	Cross-Platform
	Programming language	C++	Erlang, C++, C, Python	Java	Java	Java	Java	C, C++	Erlang	Java

Table 2.2: Evaluation of Several NoSQL Data Stores from Four Major Categories [16].

.

2.1.3 OLAP

Online Analytical Processing (OLAP) is a category of data analysis that facilitates rapid response to the multi-dimensional queries [42]. As a part of wider group of Business Intelligence (BI), this approach is used for business reporting including the area of sales, marketing, and especially in business decision making that includes budgeting and forecasting. OLAP allows performing analytical operations that include consolidation, drill-down, and slicing and dicing. Consolidation refers to the roll-up the information as a part of aggregating data in order to analyze it in multi-dimensional way [42]. Then by drilling down, the extensive possible views of aggregated data can be accessed according to the consolidation paths. And finally users can get their specific set of data with the help of slicing and dicing features of OLAP.

2.1.4 Comparative Analysis on RDBMS vs NoSQL

Following points have been summarized from [12] as a part of providing a comparative analysis on Relational databases and NoSQL data bases:

- **Transaction reliability:** RDBMS support ACID properties to provide transaction reliability whereas NoSQL databases are not reliable like RDBMSs because of its weaker BASE properties compared to ACID.
- **Data Model:** Relational Databases are based on relational model where tables that contain set of rows represent the relation. On the other hand NoSQL databases take many modelling techniques like key value stores, document data store, column data store and graph data model (refer to the section 2.2.1)**.**

- **Scalability:** The internet based web applications require horizontal scalability as it spread over several servers in a distributed environment. NoSQL data store support horizontal scalability whereas it is a great challenge for the relational model.
- **Cloud:** The relational databases cannot handle schema less unstructured data as it can work only with well-defined schema. But it is one of the requirements for handling cloud databases. However NoSQL databases are fit for the cloud scale solution as it fulfills all of the characteristics which are desirable for cloud databases.
- **Big data handling:** Because of their issues with scalability and data distribution in a clustered environment, it is not an easy task for relational database to handle big data. On the other hand NoSQL databases designed to handle the big data distributed in the clustered environment.
- **Complexity:** Day by day complexity in relational databases rises because of the continuous rapidly changed requirements. If the data for the changed requirements does not fit in the existing RDBMS schema, then it would make a complex situation in terms of changing schema and related programming code. On the other hand there is no significant effect on NoSQL databases as they can store unstructured, semi-structured or structured data.
- **Crash Recovery:** Recovery manager ensures crash recovery for RDBMS data. On the other hand crash recovery depends on data replication for NoSQL databases. MongoDB uses Journal file as recovery mechanism.

- **Security:** Very secure mechanisms are adopted by RDBMSs to secure their data. NoSQL databases are designed for storing and handling big data, and subsequently providing higher performance at the cost of security. Security of information is a big concern of the newly evolving cloud environment which is being considered as the next generation architecture for enterprises [1]. Based on security services another comparison is shown in Table 2.3 which has been taken from [12].

Category	Relational Databases	NoSQL Databases
Authentication	Come with authentication mechanism.	Does not for many NoSQL databases. But options available for external method.
Data Integrity	Ensure data integrity using ACID properties.	Not achieved or weaker integrity using BASE properties
Confidentiality	Often achieved using encryption technique	Not achieved
Auditing	Provide auditing mechanism	Does not provide. Some of the NoSQL databases store user name and password in the log file as a part of auditing

Table 2.3: Security services in Relational and NoSQL Databases [2]

2.2 Related Works: Literature Review

Most of the literatures available talked about different types of NoSQL databases, their structures, their data storage techniques and their performances. Though quite a few of them provided some approaches related to data migration including comparative performance analysis based on the evaluation result set derived from their approached models. But they did not present

any steps that can help to get a clue for migrating data from relational model to cloud databases. And their model was also not evaluated for distributed environment.

Based on the analysis of the data structures of Relational databases and NoSQL databases, the thesis paper [21] implemented a GUI (Graphical User Interface) tool facilitating data migration Relational model to NoSQL data store. This paper presented a data migration scenario from MySQL relation database to CouchDB NoSQL document database. The work included some performance comparisons between MySQL and CouchDB based on different database operations. As the comparative analysis was accomplished with small amount of data set CouchDB got the negative impression compared with the MySQL performance. But at the same time it was indicating that CouchDB getting better performance with the increase of data volume which implies that NoSQL databases are fit for Big Data solution.

An optimal solution has been proposed in [2] for managing and handling large volume of data distributed over thousands of servers using Apache Hadoop Cluster with Hadoop Distributed File System (HDFS) as data storage. The solution also included the approach of Map Reduce programming framework for processing and analyzing large distributed data sets across cluster of computers. Their experiment showed (as shown in Fig.2.14) how the processing time can be reduced by increasing the number of nodes of the clusters. This approach can be combined with [21] to provide a methodology for migrating data from RDBMS to NoSQL data store for distributed environment in order to mitigate the limitations stated in the paper [21].

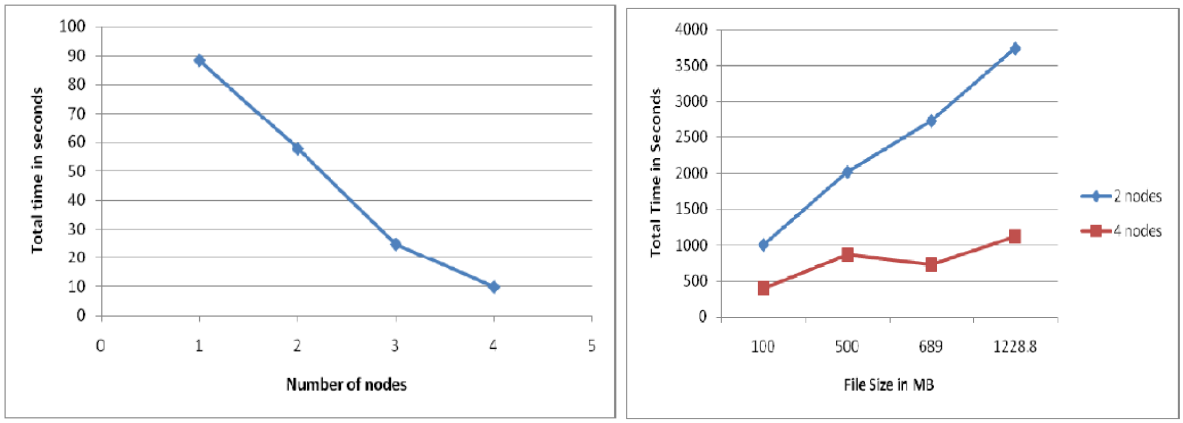


Fig.2.14: Execution time with varying number of nodes and datasets [6].

In [23], the authors presented some informative use cases based on the performance evaluation of NoSQL database Cassandra used with the Hadoop MapReduce engine that can meet the cloud application developers' decision making requirements in terms of performance issues.

A simulation platform was developed and evaluated in the paper [4] to support a case study regarding the migration of a telecom application to NoSQL environment. From Relational model PostgreSQL and Cassandra from NoSQL family were chosen for this case study. In order to support concurrent transaction with the NoSQL data model some sort of isolation design approach was used for shared transactions. But the case study could not overcome the limitation of non-supporting transactional operation. The approach was not implemented for distributed environment and also did not present any data migration steps.

Chapter 3

3. Data Migration: Problems and Solutions

Enterprise applications use relational data model that does not support improved performance relative to NoSQL in terms of analyzing large volumes of data. Data migration is required as a part of performing enterprises' statistical data analysis. With the reference of section 2 where a comparative analysis is discussed between RDBMSs and NoSQL databases, we can conclude that NoSQL data model are different from Relational model according to their structure and the way they store the information. Compared with the NoSQL databases the structure of the relational databases is more complex in terms of their concept of normalization. According to the rules of normalization they split their information into different tables with join relationship. On the other hand NoSQL databases store their information de-normalized way which is unstructured or semi structured. Therefor the successful migration with data accuracy and liability from Relational to NoSQL would not be an easy trip. This chapter proposes a methodology for the solution of data migration process followed by an implementation.

3.1 Choosing Databases

3.1.1 Choosing NoSQL Database

According to the Google Trends as shown in Fig.3.1**,** the search-term MongoDB has been entered more often than some other NoSQL databases like CouchDB, Cassandra, Redis and HBase. This search trends reflects how MogoDB is getting more popularity day by day. Considering the characteristics that include simplicity, agility and developer-friendly features available with the MongoDB, it would be good selection for meeting the purposes of the thesis.

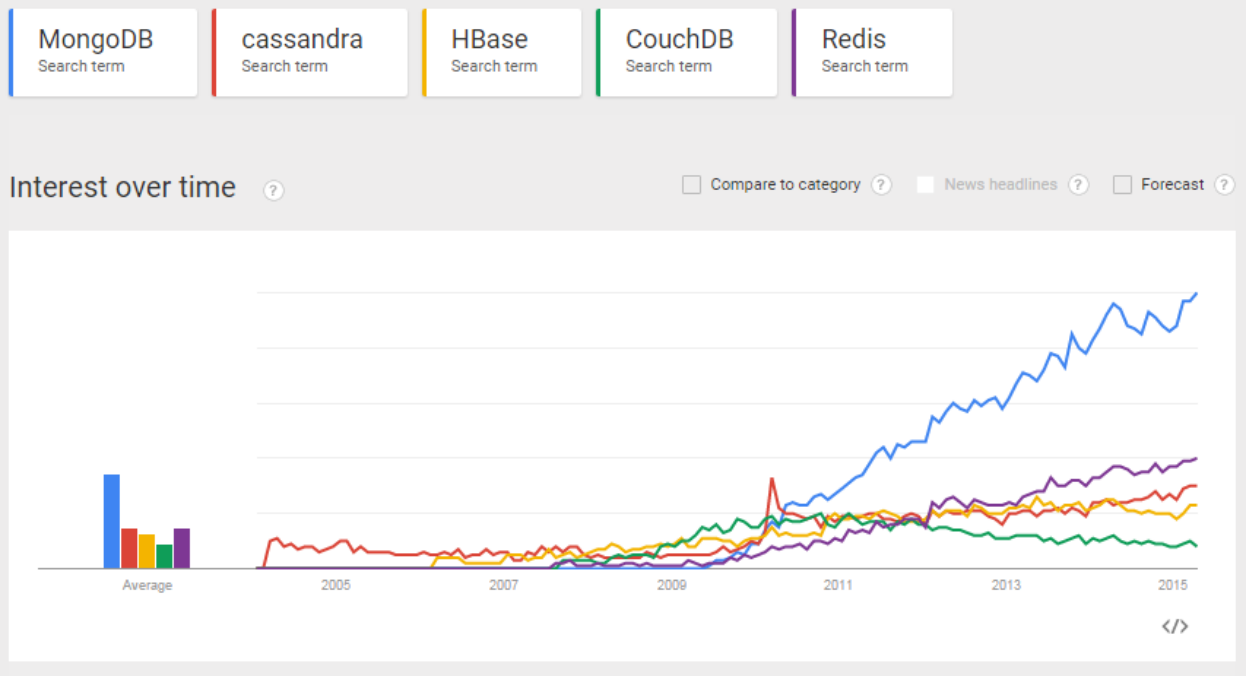


Fig.3.1: Popularity Comparison among different NoSQL Databases based on Google Search Trends.

3.1.1.1 Why MongoDB?

Written in C++ which is doing things fast and the open source JSON based document database MongoDB is popular NoSQL database among the NoSQL options. Available in many platforms it leverages standards which supports most of the popular languages like C#, Python, Ruby or Java either on Windows, Mac or Linux. The features of MongoDB include JSON based documents for storing data, flexibility, replication that leads to high availability, support indexing, auto sharding for horizontal scalability, data query and MapReduce.

The way MongoDB implemented is using memory mapped file where it uses as much memory as possible to put its indexes and collections in the RAM as a part of optimizing its performance. MongoDB supports distribution of data over multiple machines which is called ‘Sharding’ and which is also the part of scaling out data. Each of the machines where data is distributed can be replicated in order to avoid losing data. Query processing done by MongoDB

is very simple way that include choosing indexes, finding documents and finally sending output as BSON (Binary JSON) document to the socket.

The attractive features that include easy data model and data query with high performance make it more popular to the developer [37]. The way it implements it uses memory mapped files. It uses as much memory as possible to optimize the performance. MongoDB supports distribution of data over multiple machines which is called 'Sharding' which is also the part of scaling out data. And each of the machines where data is distributed can be replicated in order to avoid losing data.

3.1.2 Choosing Relational Database

From the group of relational model MySQL is chosen as the source database. MySQL is an open source database which has all the features of relational data models. According to the Oracle Corporation "MySQL is the world's most popular open source database, enabling the cost-effective delivery of reliable, high-performance and scalable web-based and embedded database applications" [38]. MySQL is very popular to the developer as it is freely available from Oracle Corporation as an open source database.

3.1.3 MySQL vs MongoDB: Syntax Comparison

For data manipulation MySQL database uses SQL language that provides functionalities like INSERT, UPDATE, DELETE and SELECT statement. On the other hand MongoDB uses functions available in JavaScript APIs (Application Programming Interfaces) for its data manipulation. This section represents some syntax differences between MySQL and

MongoDB databases for the same operation. Table 3.1 includes some of query commands used by MySQL and MongoDB for same operation.

Operations	MySQL Syntax	MongoDB Syntax
Creating table/collection	CREATE TABLE `customer` (`cust_id` int(11) NOT NULL, `first_name` varchar(45) DEFAULT NULL, `last_name` varchar(45) DEFAULT NULL);	Collection is created at the event of first insertion.
Dropping table/collection	DROP TABLE customer;	db.customer.drop();
Inserting New record	INSERT INTO customer(cust_id, first_name,last_name) values(1, 'John', 'Andrew');	db.customer.save({'cust_id': 1, 'first_name': 'John', 'last_name': 'Andrew'});
Updating Record	UPDATE customer SET first_name = 'Saint' where last_name='Andrew';	db.customer.update({'last_name': 'Andrew'},{'$set': {'first_name': 'Saint'}});
Deleting Record	DELETE from customer where cust_id > 50;	db.customer.remove({'cust_id': {'$gt': 50}});
Selecting Record	Select * from customer where first_name = 'Saint';	db.customer.find({'first_name': 'Saint'});
Order by/sort Selection	Select * from customer order by cust_id;	db.customer.find().sort({'cust_id': 1});

Table 3.1: Basic Syntaxes used by MySQL and MongoDB

3.2 Choosing Technology

C# has very good driver for MongoDB. Instead of explicit schema MongoDB can maintain implicit schema according to the application needs and respective classes can be defined using C# language according to that implicit schema. Officially .NET provides completely asynchronous driver for MongoDB to interact with MongoDB [39] using C# language. The driver is powered by Core library and BSON library. Alternative or high level of APIs can be built using Core library. BSON library facilitates handling BOSN documents stored as MongoDB data. Considering the availability of .NET driver for MongoDB and at the same

time .NET data provider for MySQL, the Data Migration Process for this thesis picks .NET platform and C# language with MySQL and MongoDB databases that makes a very good combination.

3.3 Data Migration Process

Considering the data structure and storage technique, NoSQL databases are different from RDBMSs. Relational models are highly structured and their data are normalized into different tables according to their relations whereas NoSQL data stores are semi structured or unstructured and store the data in de-normalized way. Therefore the data migration process would not be an easy trip. The Fig.3.2 illustrates how data to be migrated from relational SQL database to NoSQL document database. This figure shows data in the SQL model are normalized into different tables through relationship and same data set to be stored into JSON-style document nested with different other related documents through a migration process.

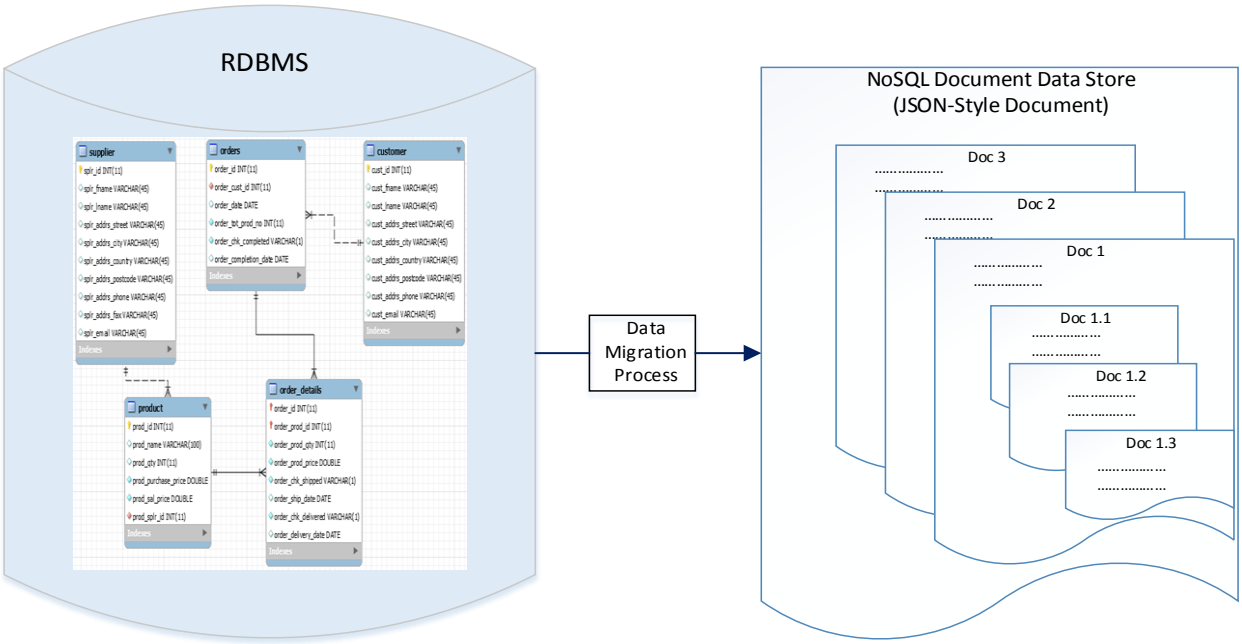


Fig.3.2: Basic Scenario for Data to be Migrated from RDBMS to NoSQL

For the migration process as shown in the above Fig.3.2, this thesis proposes an approach mainly based on traditional data migration procedure which is called ETL (Extraction, Transformation and Loading). Here extraction process includes retrieval of data from MySQL tables, then convert these data into objects using object relational mapping (ORM) and finally load them to JSON-style MongoDB documents. Fig.3.3 represents proposed conceptual flow diagram that shows the steps for data migration.

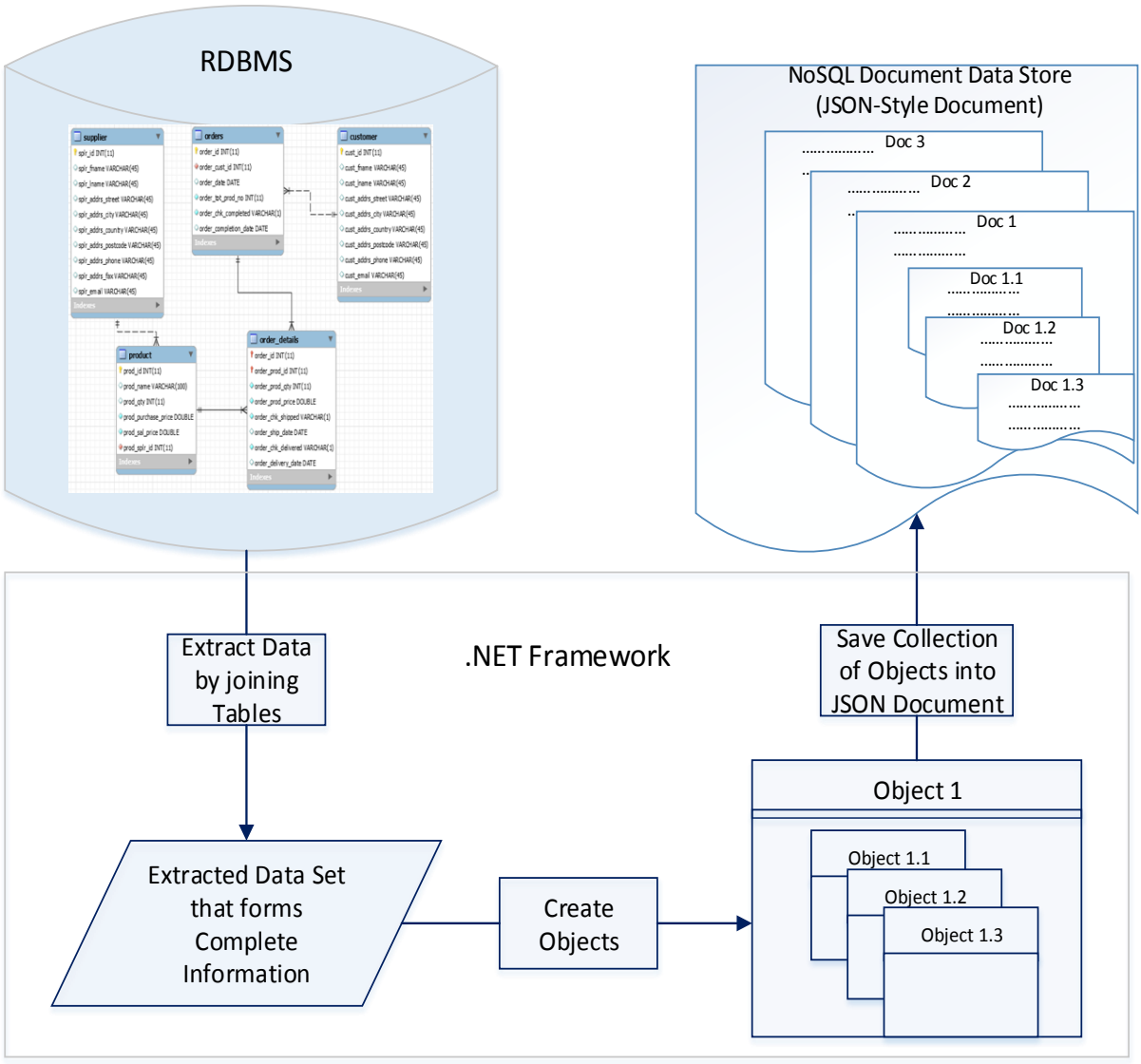


Fig.3.3: Proposed Conceptual Flow Diagram for Data Migration

Based on the data migration flow diagram shown in the Fig.3.3, following are the steps are considered for migration processes:

Step1: Analyze data with detail relationship defined in the database schema, and subsequently design and develop join criterion according to the relationship in order to get complete information.

Step 2: Design and develop an implicit schema for MongoDB data storing.

Step 3: Design and develop class diagrams based on the data analysis and implicit schema.

Step 4: Writing codes for classes defined in the class diagrams (refer to Step 1).

Step 5: Writing code for Data Migration.

3.3.1 Data Migration from Customer Order System: A Sample Case

This thesis considers Customer Order System as the test case for justifying and validating the data migration from relational data model to NoSQL data store. The Customer Order System tracks all of the orders status placed by different customers from different places by registering themselves with the system. The backend of the system is MySQL relational database. This thesis also uses MongoDB document database as the NoSQL data store which is to be considered performing all of the functionalities same as the existing MySQL database available in the Customer Order System. Fig.3.4 is the database schema that shows the relationships among different tables exist in the MySQL database of the Customer Order System.

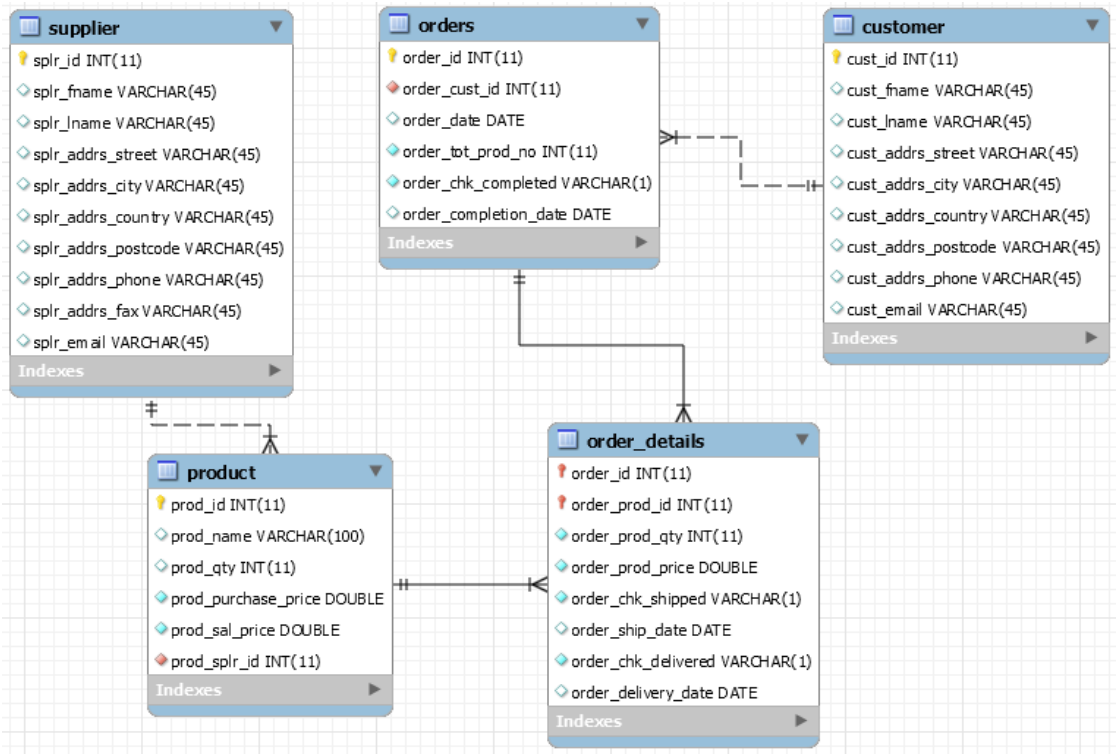


Fig.3.4: Database Schema for Customer Order System

Based on the above source database schema (Fig.3.4) following steps are designed and developed as a part of migration process.

Step 1: Analyze Source Data and Define Join Criterion

In the customer order system, tables are defined in a schema (**Fig.3.4**) using primary key (PK) and foreign key (FK) concept in order to make relationships among them. Each and every order is placed by customer and a customer can have several orders. Therefore customer has one-to-many relationship with order. Again an order can consists of one or more than one products which is/are stored in the 'Order Details' table by one-to-many relationship between order and order details. And every ordered product has a valid customer which relationship is stablished by introducing a 'SupplierID' field as a foreign key in the order details table refer to the primary key in the supplier table. The way data is recorded using relationship is shown in Fig.3.5.

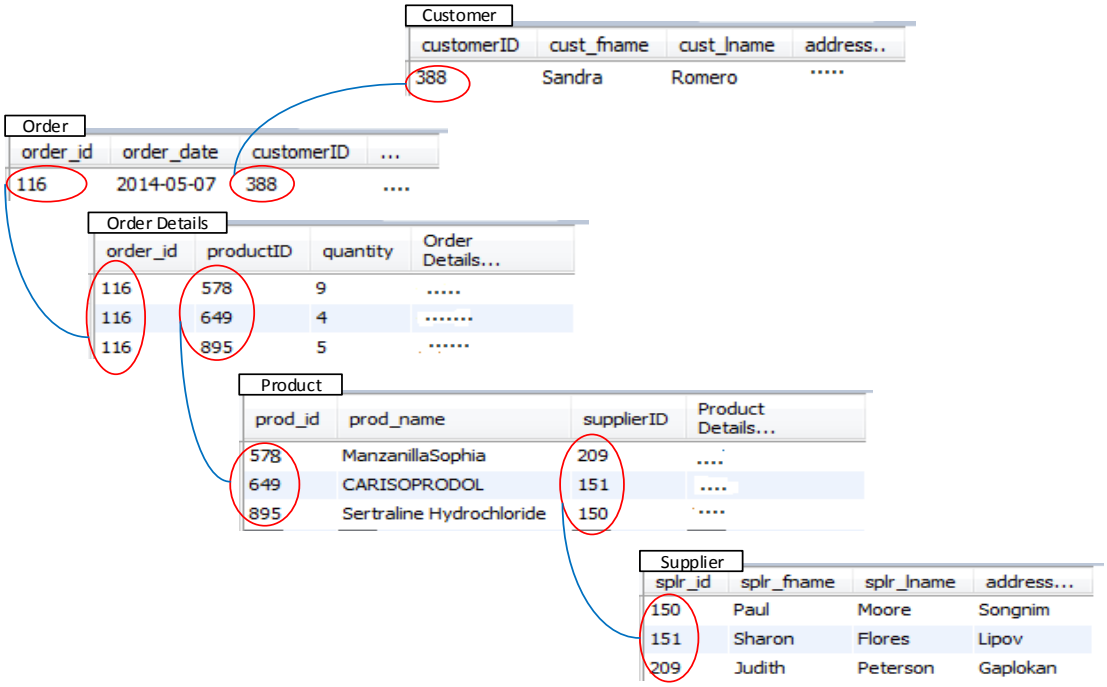


Fig.3.5: Composition Model for Storing Data through Relationship

In order to form complete information about an order, tables should be connected using different joining criterion (left join/right join/inner join/outer join). Based on the relationship of different tables shown in the schema (Fig.3.4) and the observation of data storing technique in different tables as shown in the Fig.3.5, following join structure is proposed to retrieve complete order information using data query (Fig.3.6):

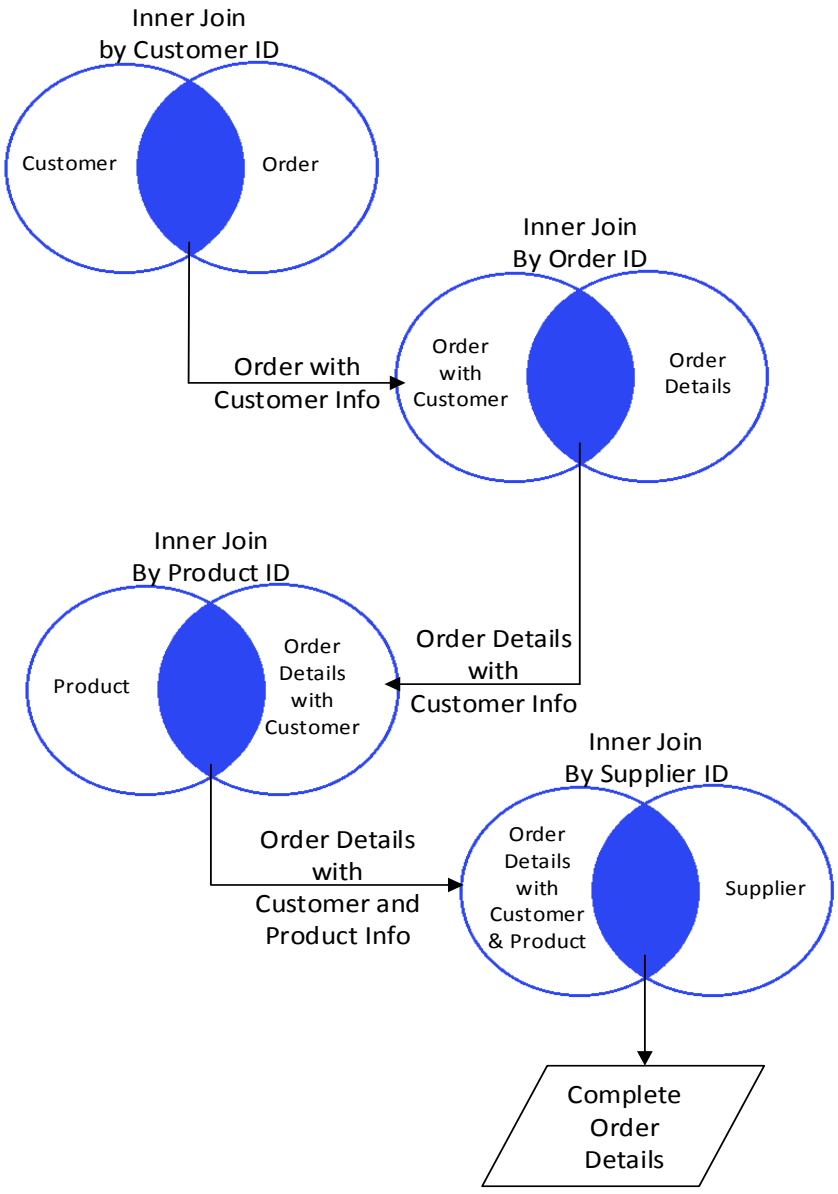


Fig.3.6: Structure of the Table Join to Extract Data as Complete Order Information

Step 2: Implicit Schema for MongoDB

As data stored in MongoDB is represented by a collection of JSON document, respective order objects will be created based on complete order information which will be saved with the collection of MongoDB as a JSON document. Every order object consists of subsets of object representing individual JSON document nested into order document that includes customer,

products and their suppliers. Based on the MySQL Database Schema for Customer Order System (Fig.3.4) and observation of data storing technique represented in the Step 1, a proposed Implicit Schema for storing MongoDB data is presented which is shown in Fig.3.7.

```
{
    _id: Object ID,
    Order_ID: ....,
    Order_Date: ......
    Customer: [
    {
        _id: Object ID,
        Customer_ID: ....,
        Name:......
        Address: [
        {
            <Address Details>
        }
        ]
    }],
    Product: [
    {
        _id: Object ID,
        Product_ID: ....,
        ................
        Supplier: [
        {
            _id: Object ID,
            Supplier_ID:.....,
            Name:.....
            Address: [
            {
              <Address Details>
            }]
        }]
        Shipping: [{ <Shipping Details> }]
    },
    {
        Another Product....
    }]
}
```

Fig.3.7: Proposed Implicit Schema for Migrated MongoDB Data Structure

Step 3: Design and Develop Class Diagram

According to Step 1, the source MySQL order data are stored in different tables with relationships. The relationships show that a customer can have an order where customers and orders have an association relationship. An order has one to many relationships with ordered

items as the customer can place an order with more than one item and it is required to have multiple records to complete a customer order. But in the NoSQL MongoDB database a complete order will be stored as a single JSON document nested with some other JSON documents associated with that order. With the consideration of this fact this thesis paper proposes a class diagram (Fig.3.8) as a part of data migration process which includes a class named as 'Orders' that instantiates an order object. The order object then includes its customer object, collection of product objects and their related supplier object.

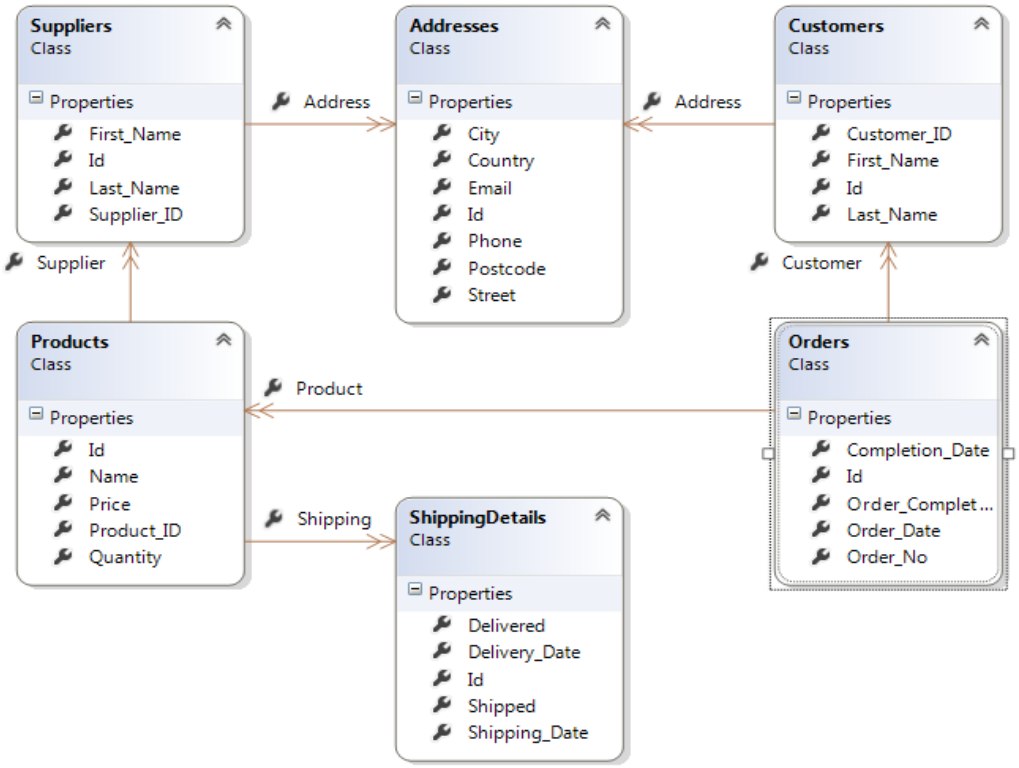


Fig.3.8: Proposed Class Diagram for Data Migration

Fig.3.8 illustrates the proposed class diagram. As the data structure of MongoDB is different from MySQL, the data migration class diagram may not be designed directly following MySQL database schema. The class diagram has been designed and developed based on the

Implicit Schema as shown in Fig.3.7. It is a composition model where the main ‘Orders’ class has other objects like ‘Customers’, ‘Products’ and its ‘Suppliers’ object. So the ‘Orders’ class is the aggregation of these objects. Based on the database schema and implicit schema, the class diagram includes ‘Products’ class by combining ‘order_details’ and ‘product’ tables. Another class ‘ShippingDetails’ is also derived from ‘order_detials’ table. The ‘Address’ class which is associated with both ‘Customer’ and ‘Supplier’ classes, is mainly derived from ‘customer’ and ‘supplier’ tables.

Step 4: Coding for Defining Classes

Based on the class diagram, this step represents how different classes are defined in the .NET platform using C# language. As a JSON document requires an Object ID, all of the classes include an Object ID that serves as the Document ID of that respective JSON document.

Code samples for some of the classes are given below:

```
// Defining Order Class
public class Orders
{
    public ObjectId Id { get; set;}
    public int Order_No { get; set;}
    public List<Customers> Customer {
get; set;}
    public List<Products> Product { get;
set;}
    -------------
}
```

```
// Defining Customer Class
public class Customers
{
    public ObjectId Id { get; set;}
    public int Customer_ID {get;set;}
    public string First_Name {get;set;}
    public string Last_Name {get; set;}
    public List<Addresses> Address {
get; set;}
}
```

```
// Defining Supplier Class
public class Suppliers
{
    public ObjectId Id { get; set;}
    public int Supplier_ID { get; set; }
    public string First_Name {get; set;}
    public string Last_Name {get; set;}
    public List<Addresses> Address {
get; set; }
}
```

```
// Defining Product Class
public class Products
{
    public ObjectId Id { get; set;}
    public int Product_ID { get; set;}
    public string Name { get; set;}
    -----------------
    public List<Suppliers> Supplier {
get; set;}
    public List<ShippingDetails>
Shipping { get; set; }
}
```

Step 5: Codes for Data Migration

This step represents some coding samples that include getting or creating MongoDB data collection, extracting data from different SQL tables in order to form complete order information using join criterion identified in Step 1, mapping the extracted data to the BSON objects instantiated from classes (refer to Step 4) and subsequently uploading these collection of objects to the MongoDB collections as BSON document. Coding samples are given as follows:

```
.................
.................
MongoClient client = new MongoClient();
var server = client.GetServer();
//Get the MongoDB database.
//If it doesn't exist MongoDB will create it for the first use
var db = server.GetDatabase("mydata");
//Get the Orders collection where the name of the class
//is used as the collection name.
//If it doesn't exist, MongoDB will create it for the first time use.
var collection = db.GetCollection<Orders>("CustomerOrders1");
try
{
    MySqlConnection conn = new MySql.Data.MySqlClient.MySqlConnection();
    conn.ConnectionString = myConnectionString;
    conn.Open();
    //Define SQL string following join criterion
    sqlstr = "SELECT orders.order_ID, orders.order_date, orders.order_cust_ID,
    ........... FROM customer INNER JOIN (((order_details INNER JOIN product ON
    order_details.order_prod_ID = product.prod_id) INNER JOIN supplier ON
    product.prod_splr_id = supplier.splr_id) INNER JOIN Orders ON
    order_Details.order_ID = orders.order_ID) ON customer.cust_id =
    orders.order_cust_ID;
    MySqlCommand cmd = new MySqlCommand(sqlstr, conn);
    MySqlDataReader myReader = cmd.ExecuteReader();
    //Instantiating Orders object
    Orders order = new Orders();
    //Define variable for contacting list of product objects for an order
    var prodList = new List<Products>();
    while (myReader.Read())
    {
        var orderID = myReader.GetInt16(0);
        // Checking for end of an order
        if (mprvordrNo != orderID)        {
            if (mchk>0) // for skipping the first instance
            {
                // include all of the product objects with the order
                order.Product = prodList;
                collection.Save(order); // Save an order to the MongoDB Collection
                order = new Orders();
```

```
                order = new Orders();
                prodList = new List<Products>();
                mchk = 0;
            }
            mchk++;
            ......................
            order.Order_No = myReader.GetInt16(0);
            Customers customer = new Customers(); // Instantiating Customer Object
            customer.Customer_ID = myReader.GetInt16(2);
            ......................
            var custList = new List<Customers>();
            custList.Add(customer);
            order.Customer = custList; // Include the customer object with an order

        }
        Products product = new Products(); // Instantiating product object
        product.Product_ID = myReader.GetInt16(5);
        ......................
        var splrList = new List<Suppliers>();
        var addrs = new List<Addresses>();
        Suppliers splr = new Suppliers(); // Instantiating supplier object
        splr.Supplier_ID = myReader.GetInt16(13);
        ......................
        splrList.Add(splr);
        //Include supplier object with the respective product
        product.Supplier = splrList;
        prodList.Add(product);
        ......................
    }
    order.Product = prodList; //Include list of product object with an order
    collection.Save(order); //Save order details with the MongoDB collection
    ......................
```

The above implementation is done only for a specific system. It is not generalized. A generalized data migration tool can be developed by following the methodology proposed in this thesis and the subsequent implementation.

Chapter 4

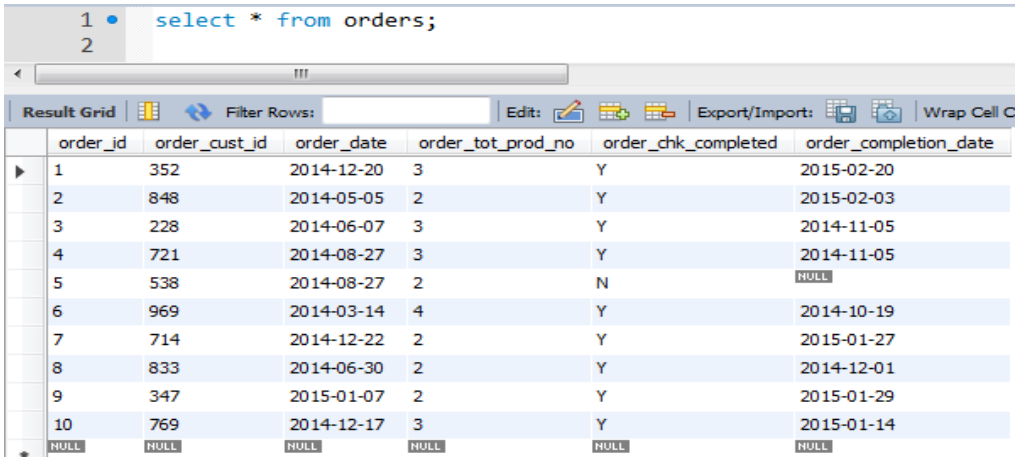
4. Evaluation

This chapter consists of the evaluation of the migration process based on the comparison of migrated NoSQL MongoDB data and original source RDBMS MySQL data with different measures. The evaluation process includes verification of data migration process with performance comparison based on identical operations between MySQL and MongoDB database. It also covers comparative analysis on some issues with developers' facilities for their related database application development works. The following measures will be considered as the evaluation goals:

- Verification of Data Migration
- Performance
- Development Agility
- Simplicity of Query

4.1 Verification of Data Migration

In order to verify whether data migration process is performed successfully or not, this section includes representation of source data and respective migrated data. This section also includes representation of some basic operations of MongoDB database which are identical with MySQL database like INSERT, UPDATE, DELETE and SELECT. For MySQL data representation, MySQL Workbench is used. A shell-centric MongoDB data management tool Robomongo is used for representing MongoDB data.

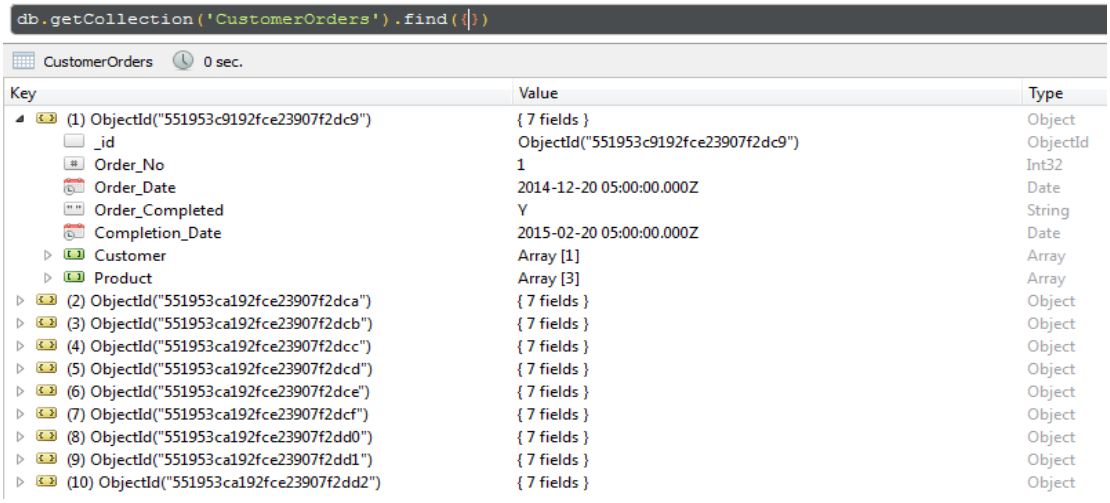


```
1 • select * from orders;
2
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell C

order_id	order_cust_id	order_date	order_tot_prod_no	order_chk_completed	order_completion_date
1	352	2014-12-20	3	Y	2015-02-20
2	848	2014-05-05	2	Y	2015-02-03
3	228	2014-06-07	3	Y	2014-11-05
4	721	2014-08-27	3	Y	2014-11-05
5	538	2014-08-27	2	N	NULL
6	969	2014-03-14	4	Y	2014-10-19
7	714	2014-12-22	2	Y	2015-01-27
8	833	2014-06-30	2	Y	2014-12-01
9	347	2015-01-07	2	Y	2015-01-29
10	769	2014-12-17	3	Y	2015-01-14
NULL	NULL	NULL	NULL	NULL	NULL

(a) Data Retrieved from MySQL using SELECT Statement



```
db.getCollection('CustomerOrders').find({})
```

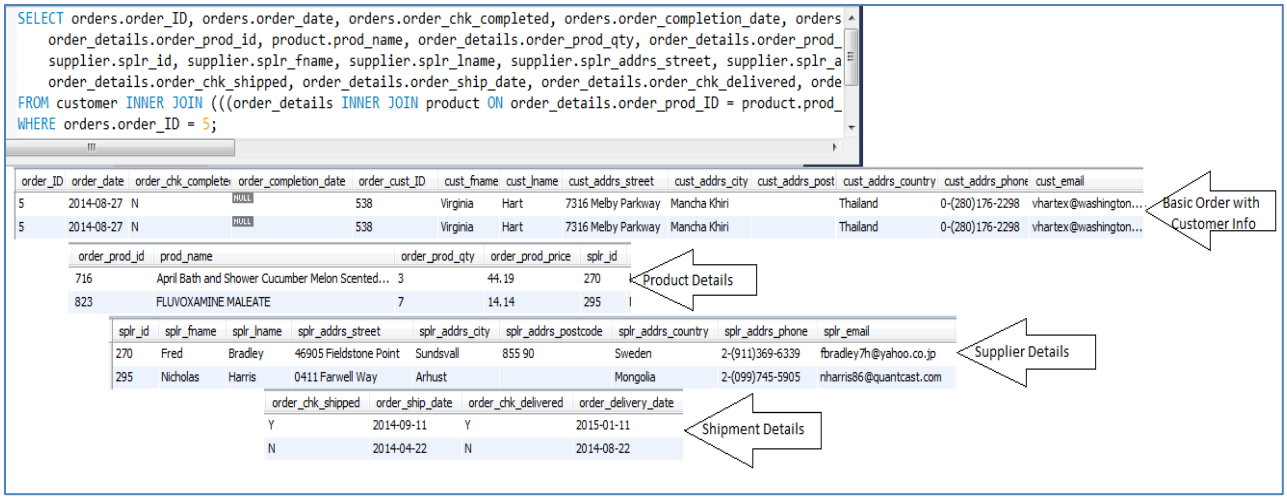
CustomerOrders 0 sec.

Key	Value	Type
(1) ObjectId("551953c9192fce23907f2dc9")	{ 7 fields }	Object
_id	ObjectId("551953c9192fce23907f2dc9")	ObjectId
Order_No	1	Int32
Order_Date	2014-12-20 05:00:00.000Z	Date
Order_Completed	Y	String
Completion_Date	2015-02-20 05:00:00.000Z	Date
Customer	Array [1]	Array
Product	Array [3]	Array
(2) ObjectId("551953ca192fce23907f2dca")	{ 7 fields }	Object
(3) ObjectId("551953ca192fce23907f2dcb")	{ 7 fields }	Object
(4) ObjectId("551953ca192fce23907f2dcc")	{ 7 fields }	Object
(5) ObjectId("551953ca192fce23907f2dcd")	{ 7 fields }	Object
(6) ObjectId("551953ca192fce23907f2dce")	{ 7 fields }	Object
(7) ObjectId("551953ca192fce23907f2dcf")	{ 7 fields }	Object
(8) ObjectId("551953ca192fce23907f2dd0")	{ 7 fields }	Object
(9) ObjectId("551953ca192fce23907f2dd1")	{ 7 fields }	Object
(10) ObjectId("551953ca192fce23907f2dd2")	{ 7 fields }	Object

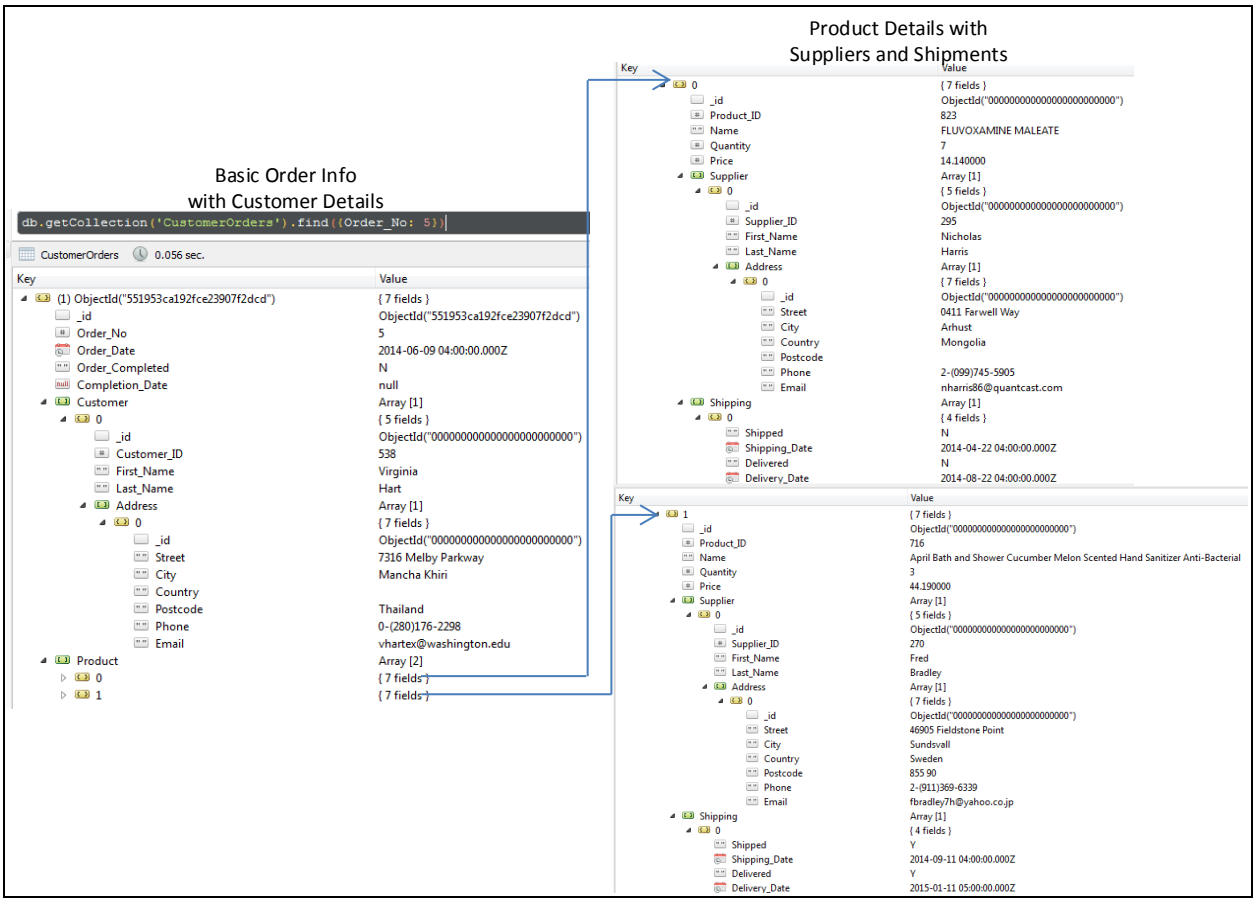
(b) Migrated Data, Retrieved from MongoDB using ‘find’ Syntax

Fig.4.1: Initial Data Verification by Comparing the Total Number of Records.

Fig.4.1 shows 10 MongoDB objects listed on the Robomongo interface. These objects are created from 10 related MySQL data which is listed on the MySQL Workbench interface. The objects are created using MongoDB ‘save’ query function which is identical to MySQL ‘INSERT’ statement. Data retrieval in MongoDB is done using ‘find’ query function which is identical to ‘SELECT’ statement. The following example represents retrieval and subsequent comparison details of a specific record.



(a) Details of a Specific Order (Order No. # 5) Retrieved from MySQL



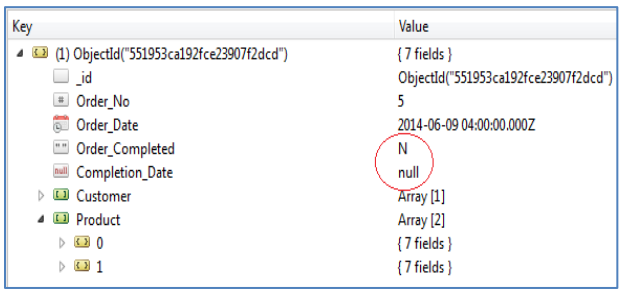
(b) Details of a Specific Order (Order No. # 5) Retrieved from Migrated MongoDB Data

Fig.4.2: Verification of a Specific Order Details (Order No. # 5)

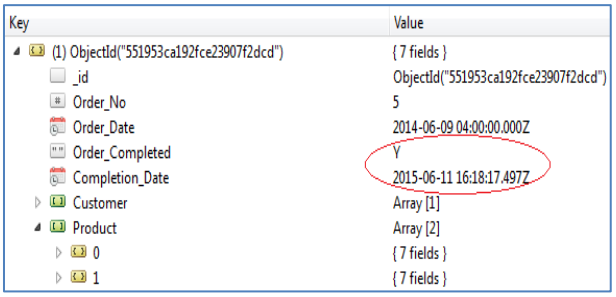
As a part of data verification, Fig.4.2 shows the details of a particular order (Order No. # 5) that includes basic order information, customer details and product details with respective

suppliers and shipment information. Parameter 'Order_No: 5' is used with 'find' function to retrieve the details of the order number 5 from migrated MongoDB data. Here the parameter serves as the 'WHERE' clause of MySQL database.

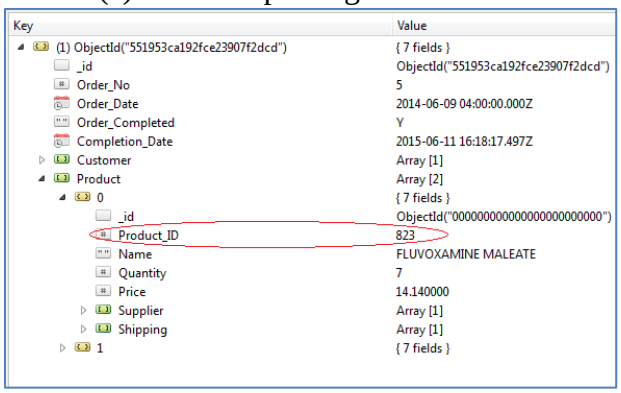
Following is an example for representing update operations in MongoDB database. In order to update or modify any information in MongoDB database, 'update' function is used which is similar to the 'UPDATE' statement of relational model. MongoDB 'update' function is also used to delete or remove any nested document from the main document.



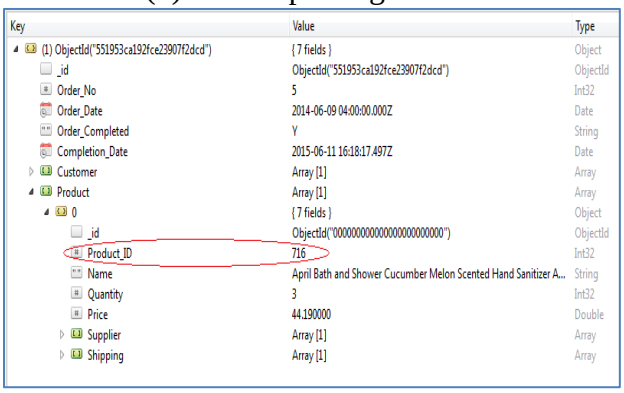
(a) Before Updating Information



(b) After Updating Information



(c) Before Updating/Deleting a Product

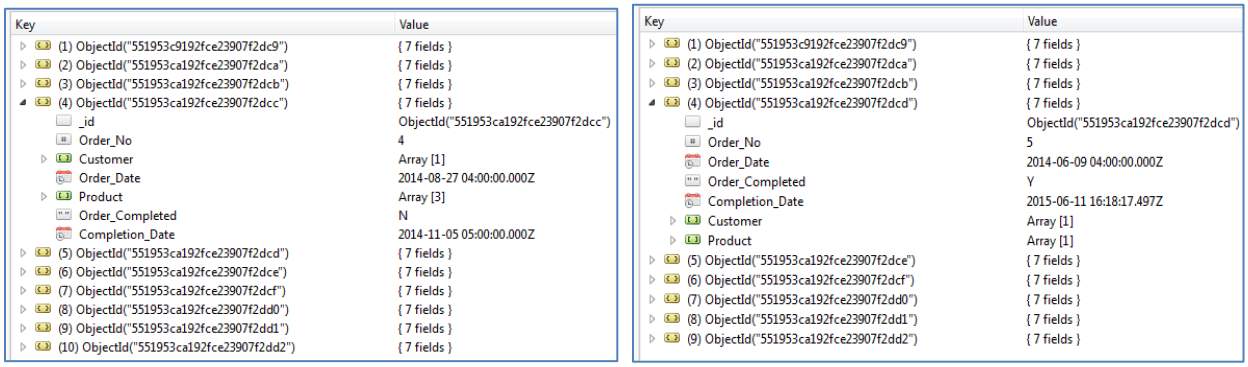


(d) After Updating/Deleting a Product

Fig.4.3: Example of Two Different Update Operations with MongoDB Data

Fig.4.3 presents two different update operations which are described by four different scenarios (a), (b), (c) and (d). Scenarios (a) and (b) describe the update operation that updates order completion status and date for a particular order. Scenarios (c) and (d) show the deletion of a product which is stored as a nested document with a particular order document (Order No. # 5).

Like a record in the relational model, any collection can be deleted or removed from the MongoDB database using 'remove' function. In Fig.4.4, (a) shows that MongoDB has ten collections where the 4th collection represents the Order No. 4. But (b) shows that it has total nine collections where the 4th collection represents the Order No. 5 instead of Order No. 4. This means the collection with Order No. 4 has been deleted.



(a) Before Removing Order No. 4 (b) After removing Order No. 4

Fig.4.4: Example of Delete Operation in MongoDB Database

4.2 Performance Assessment

MongoDB is a general purpose open source database which mainly focuses on high performance [37]. The performance analysis mainly done based on time comparison between MySQL and MongoDB required for basic database operations. Based on the type of operations, this section includes following two sub sections.

4.2.1 Data Storage Related Performance

This section includes performance comparison based on data storage operations. The performance analysis compares time required for both MySQL and MongoDB in order to

execute INSERT, UPDATE and DELETE operations. For each of the operations, 10 observations are recoded as shown in Table 4.1. The performance operations are done with different number of records ranging from 10 to 100 records. Fig.4.5, Fig.4.6 and Fig.4.7 show the graphical representation of the performance analysis performed by INSERT, UPDATE and DELETE operations respectively. From these observations we can see that MongoDB exhibits better and significant performances for data storage operation that include INSERT, UPDATE and DELETE.

Number of Records	INSERT (Time in Milliseconds)		Update (Time in Milliseconds)		DELETE (Time in Milliseconds)	
	MySQL	MongoDB	MySQL	MongoDB	MySQL	MongoDB
10	626	82	361	68	648	94
20	1219	121	705	105	1490	121
30	1780	147	1162	134	2135	143
40	2250	174	1532	165	2793	167
50	2718	204	1851	195	3581	196
60	3279	228	2207	227	4159	222
70	3914	250	2541	260	4768	246
80	4373	283	2882	289	5248	270
90	4816	313	3213	318	5795	294
100	5383	343	3549	346	6258	324

Table 4.1: Observations from Performance Comparison on INSERT, UPDATE and DELETE operations.

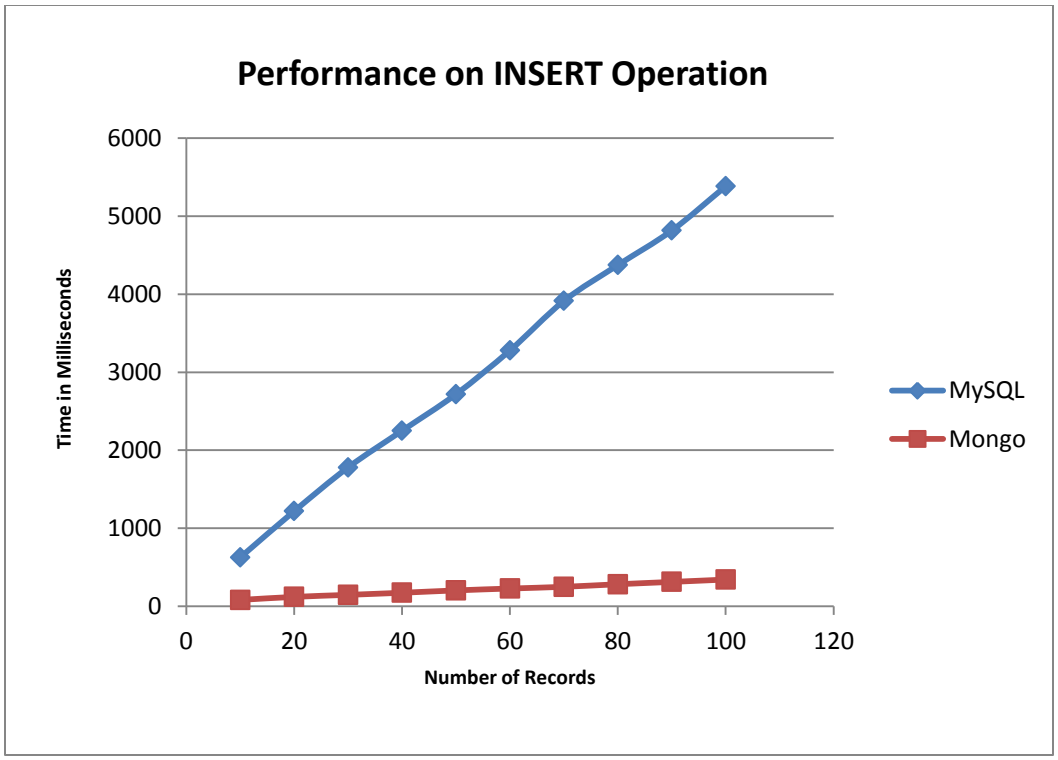


Fig.4.5: Performance Comparison for INSERT Operation

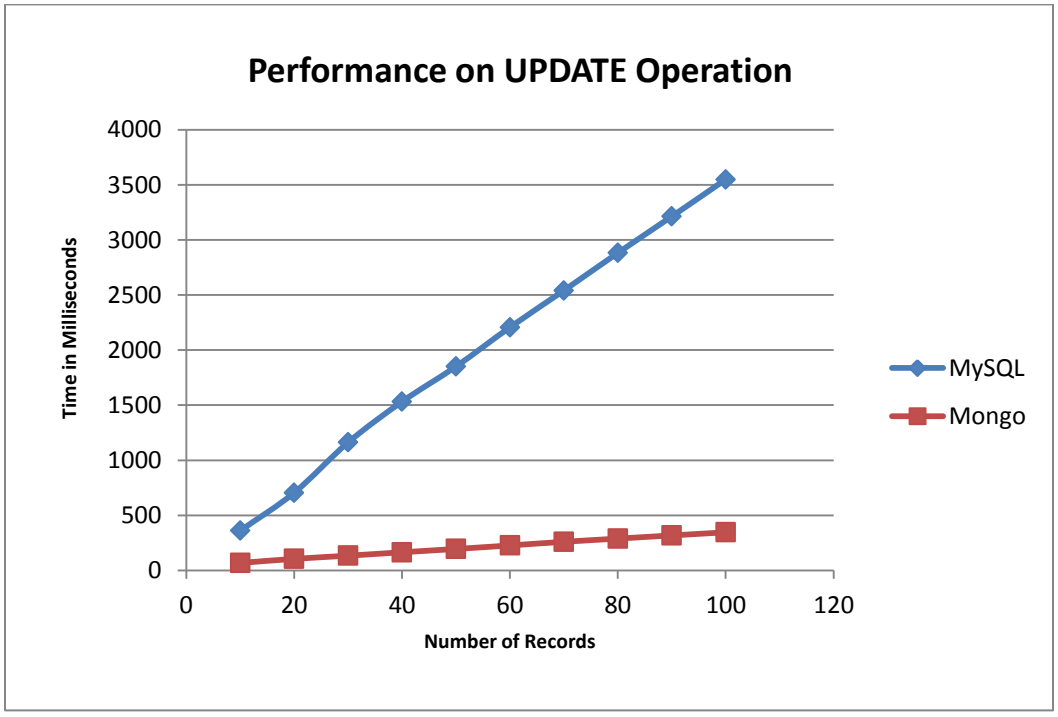


Fig.4.6: Performance Comparison for UPDATE Operation

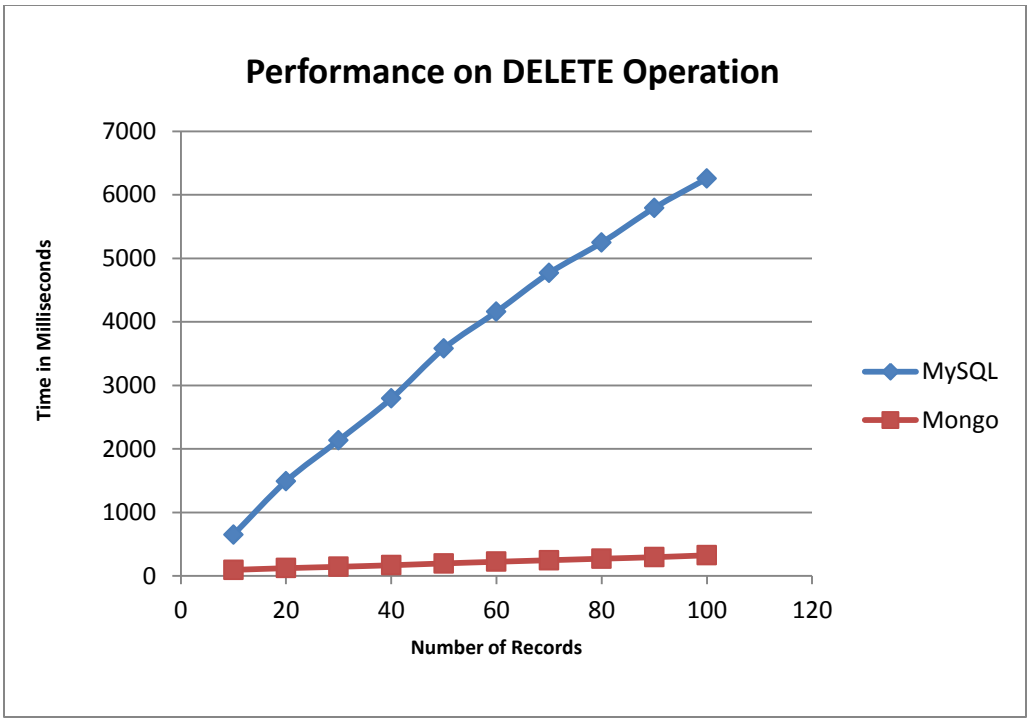


Fig.4.7: Performance Comparison for DELETE Operation

4.2.2 Data Loading Related Performance

This section includes performance analysis based on different data loading or data selection operations applied to MongoDB and MySQL which is in general popularly known as 'SELECT' SQL Data Manipulation Language (DML) statement for relational databases. For each of the operations here we observe and analyze that how the both databases take time to perform the same result for the same type of operations. Following four cases exhibit the performance results derived from different data selection criterion according to the data analysis requirements. For all of the four cases the operations are done with different number of data sets ranging from 1000 to 10000. Time taken by each and every test run is recorded in millisecond where each test run time is recorded as the average of ten different test runs for both MySQL and MongoDB databases. Initially the time for test run was recorded from ten consecutive test run using a loop. But it was observed that MongoDB only took time for the first test run where

remaining nine was showing as zero. Therefore in order to make the fair comparison, every test run result was derived from different individual execution.

Case 1: Simple Data Loading

In this case performance is observed based on simple selection criterion without applying any condition or features. The performance is measured by time taken to load or select all data with complete order information from MySQL different relational tables and MongoDB JSON document without applying any clause. Ten different observations are shown in the Table 4.2 and comparative performances are represented in the Fig.4.8. Following two queries are used for loading data from MySQL and MongoDB respectively:

MySQL Query:

```
SELECT      orders.order_ID, orders.order_date, orders.order_cust_ID,
        orders.order_chk_completed, orders.order_completion_date, order_details.order_prod_id,
        product.prod_name, order_details.order_prod_qty, order_details.order_prod_price,
        order_details.order_chk_shipped, order_details.order_ship_date,
        order_details.order_chk_delivered, order_details.order_delivery_date, supplier.splr_id,
        supplier.splr_fname, supplier.splr_lname, supplier.splr_addrs_street,
        supplier.splr_addrs_city, supplier.splr_addrs_postcode, supplier.splr_addrs_country,
        supplier.splr_addrs_phone, supplier.splr_email, customer.cust_fname,
        customer.cust_lname, customer.cust_addrs_street, customer.cust_addrs_city,
        customer.cust_addrs_postcode, customer.cust_addrs_country,
        customer.cust_addrs_phone, customer.cust_email
FROM customer INNER JOIN (((order_details INNER JOIN product ON
        order_details.order_prod_ID = product.prod_id) INNER JOIN supplier ON
        product.prod_splr_id = supplier.splr_id) INNER JOIN Orders ON
        order_Details.order_ID = orders.order_ID) ON customer.cust_id = orders.order_cust_ID;
```

MongoDB Query:

For MongoDB Shell

```
db.CustomerOrders.Find();
```

For C# Driver:

```
Collection.FindAll();
        where Collection = db.GetCollection<BsonDocument>("CustomerOrders");
```

Performance Test Results:

Number of Records	Data Loading Time (in Milliseconds)	
	MySQL	MongoDB
1000	3	28
2000	3	28
3000	3	30
4000	3	25
5000	3	27
6000	3	26
7000	3	28
8000	3	26
9000	3	28
10000	3	28

Table 4.2: Observations from Simple Data Loading Test Run

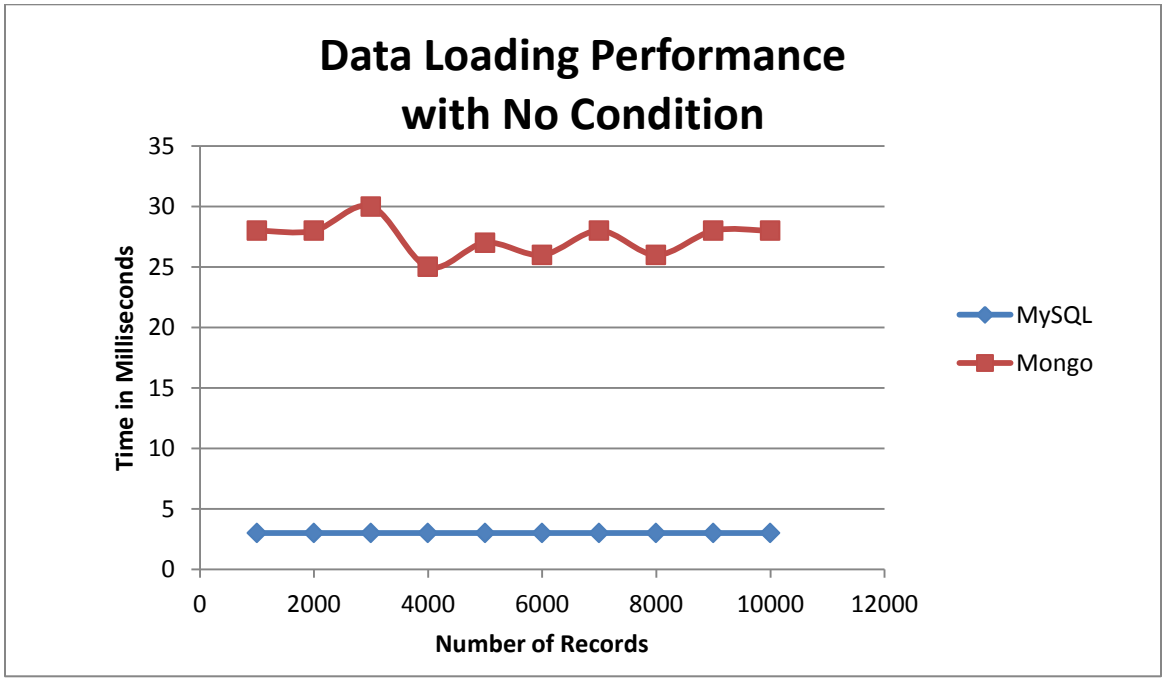


Fig.4.8: Performance Comparison for Simple Data Loading

The graph shown in Fig.4.8 has been plotted from the test results which are represented in Table 4.2. Fig.4.8 shows that MySQL performance is better compared to MongoDB. For every test run, MySQL database exhibits significant performance over MongoDB database which is steady and does not vary with increase of number of records. On the other hand, though MongoDB takes more time for simple data loading, but its performances get steady with the increase of number of records.

Number of Records	Standard Deviation (in Milliseconds)		Coefficient of Variation	
	MySQL	MongoDB	MySQL	MongoDB
1000	0.70	4.01	0.23	0.14
2000	0.74	7.50	0.25	0.27
3000	0.57	7.25	0.19	0.24
4000	0.57	8.00	0.19	0.32
5000	0.74	9.11	0.25	0.34
6000	0.32	7.69	0.11	0.30
7000	0.32	6.43	0.11	0.23
8000	0.32	6.35	0.11	0.24
9000	0.70	7.76	0.23	0.28
10000	0.32	6.76	0.11	0.24

Table 4.3: Standard Deviation and Coefficient of Variation Derived from Table 4.2 Data Sets

Fig.4.8 also shows that MongoDB performances fluctuate for different number of records compared to MySQL that exhibits steady performances. In order to investigate varying performances, more comparisons are done using standard deviation and coefficient of variation. Table 4.3 represents the respective standard deviations and coefficient of variations derived from the performance test results recorded in Table 4.2. Based on data sets in Table 4.3, comparison graphs for standard deviations and coefficient of variations are represented in Fig.4.9 and

Fig.4.10 respectively. These two figures show that both MySQL and MongoDB performances' fluctuate for different number of records. Fig.4.10 shows that variation trends are almost same.

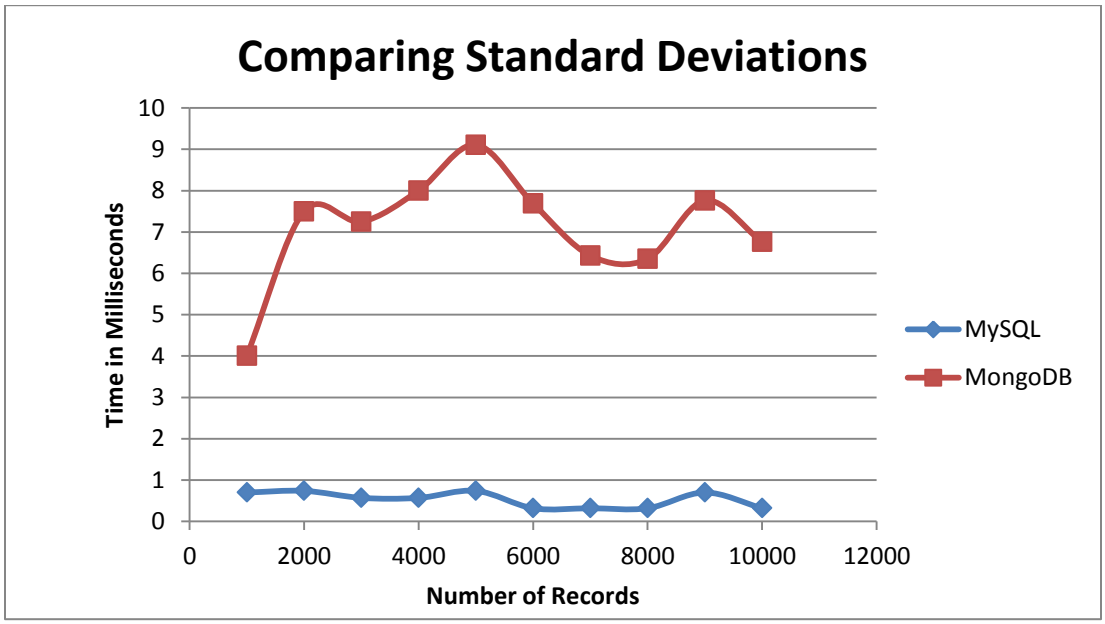


Fig.4.9: Comparison by Standard Deviations for Simple Data Loading

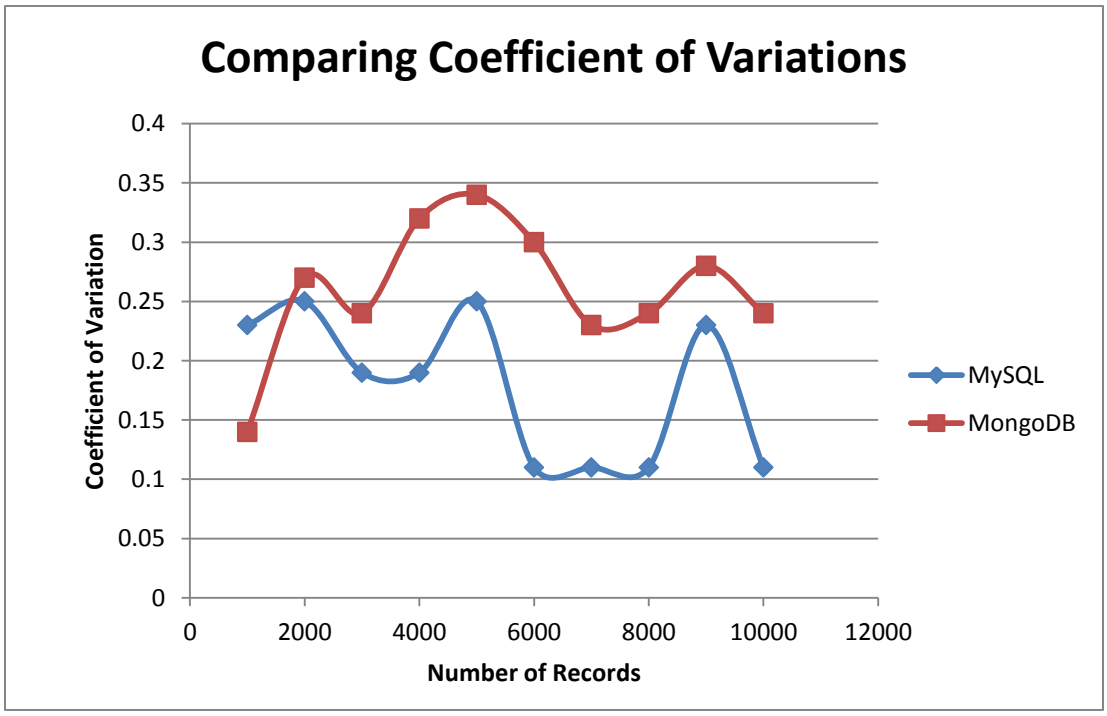


Fig.4.10: Comparison by Coefficient of Variations for Simple Data Loading

Case 2: Data Loading with ORDER BY Clause

This case exhibits comparably how much time is required for MySQL and MongoDB to load data with MySQL ORDER BY clause and respective MongoDB syntax. The performance is measured by time taken to load or select all data with complete order information from MySQL different relational tables and MongoDB JSON document by applying ORDER BY clause and identical syntax. The extracted data is organized according to the order of 'Order No'. The test run data recorded from observations, are represented in Table 4.4. Fig.4.11 is the respective graphical representation for the comparative analysis obtained from the observation data available on Table 4.4. Following two queries are used for loading data from MySQL and MongoDB respectively applying 'order by' clause:

MySQL Query:

```
SELECT orders.order_ID, orders.order_date, orders.order_cust_ID,
        orders.order_chk_completed, orders.order_completion_date, order_details.order_prod_id,
        product.prod_name, order_details.order_prod_qty, order_details.order_prod_price,
        order_details.order_chk_shipped, order_details.order_ship_date,
        order_details.order_chk_delivered, order_details.order_delivery_date, supplier.splr_id,
        supplier.splr_fname, supplier.splr_lname, supplier.splr_addrs_street,
        supplier.splr_addrs_city, supplier.splr_addrs_postcode, supplier.splr_addrs_country,
        supplier.splr_addrs_phone, supplier.splr_email, customer.cust_fname,
        customer.cust_lname, customer.cust_addrs_street, customer.cust_addrs_city,
        customer.cust_addrs_postcode, customer.cust_addrs_country,
        customer.cust_addrs_phone, customer.cust_email
FROM customer INNER JOIN (((order_details INNER JOIN product ON
        order_details.order_prod_ID = product.prod_id) INNER JOIN supplier ON
        product.prod_splr_id = supplier.splr_id) INNER JOIN Orders ON
        order_Details.order_ID = orders.order_ID) ON customer.cust_id = orders.order_cust_ID
ORDER BY orders.order_ID;
```

MongoDB Query:

```
For MongoDB Shell
        db.CustomerOrders.Find().sort({Order_No: 1});
```

For C# Driver:

```
Collection.FindAll().SetSortOrder(SortBy<Orders>.Ascending(o => o.Order_No));

        where Collection = db.GetCollection<BsonDocument>("CustomerOrders").
```

Performance Test Results:

Number of Records	Data Loading Time (in Milliseconds)	
	MySQL	MongoDB
1000	3	89
2000	4	91
3000	3	94
4000	3	86
5000	84	87
6000	107	91
7000	121	91
8000	127	90
9000	140	89
10000	293	93

Table 4.4: Observations from Ordered Way Data Loading Test Run

As shown in Fig.4.11, the comparative analysis derived from the test run using ORDER BY clause, is reflecting a significant performance of MongoDB database over MySQL database. According to the above graphical representation the time required for performing data extraction is almost unchanged and steady with the increase of data volume for NoSQL MongoDB database. On the other hand MySQL performance is better and steady up to 4000 records. But after 4000 records, MySQL suddenly starts taking more time and its performance gradually decreases with the increase of data volume.

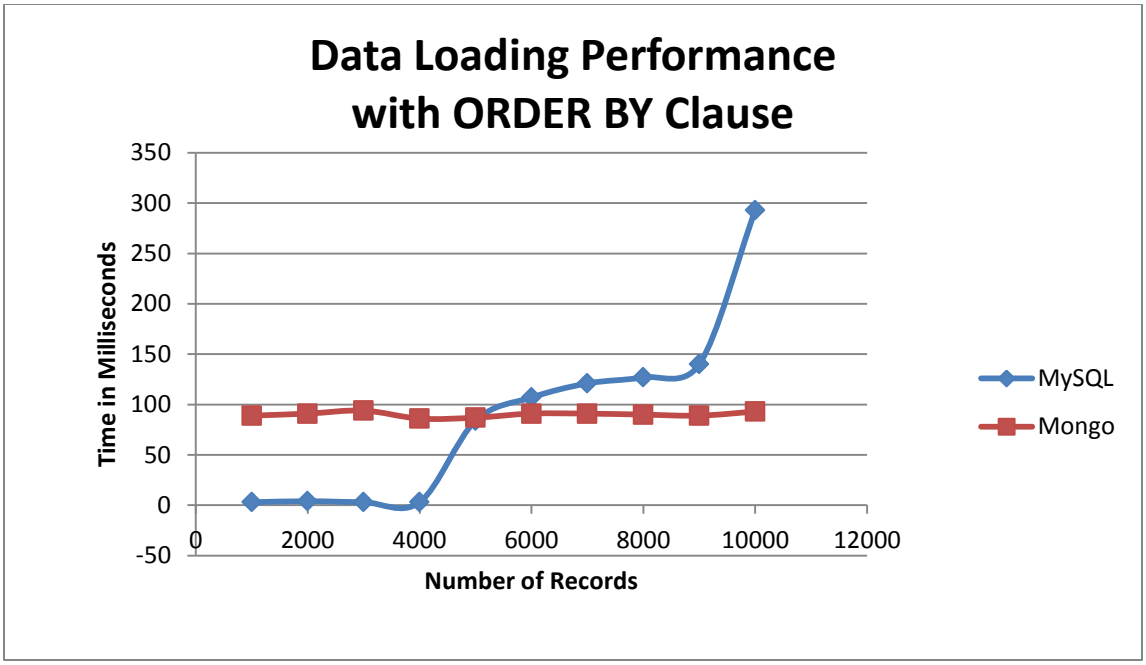


Fig.4.11: Performance Comparison for Ordered Way Data Loading

Case 3: Data Loading with WHERE Clause

The WHERE clause is a part of the DML of SQL which is used to retrieve records only that meet some specific criterion. Here in the Case 3, WHERE clause for relational MySQL and the similar query feature available in the NoSQL MongoDB has been used to retrieve complete order information with certain criterion and subsequently measure the respective query execution time. Data obtained from the observation of ten different test run is recorded and shown in Table 4.5. The comparative analysis based on this observation data is graphically represented in Fig.4.12. The identical relevant queries for MySQL and MongoDB used for this case are as follows:

MySQL Query:

```
SELECT orders.order_ID, orders.order_date, orders.order_cust_ID,
        orders.order_chk_completed, orders.order_completion_date, order_details.order_prod_id,
        product.prod_name, order_details.order_prod_qty, order_details.order_prod_price,
        order_details.order_chk_shipped, order_details.order_ship_date,
        order_details.order_chk_delivered, order_details.order_delivery_date, supplier.splr_id,
```

```
        supplier.splr_fname, supplier.splr_lname, supplier.splr_addrs_street,
        supplier.splr_addrs_city, supplier.splr_addrs_postcode, supplier.splr_addrs_country,
        supplier.splr_addrs_phone, supplier.splr_email, customer.cust_fname,
        customer.cust_lname, customer.cust_addrs_street, customer.cust_addrs_city,
        customer.cust_addrs_postcode, customer.cust_addrs_country,
        customer.cust_addrs_phone, customer.cust_email
FROM customer INNER JOIN (((order_details INNER JOIN product ON
        order_details.order_prod_ID = product.prod_id) INNER JOIN supplier ON
        product.prod_splr_id = supplier.splr_id) INNER JOIN Orders ON
        order_Details.order_ID = orders.order_ID) ON customer.cust_id = orders.order_cust_ID
WHERE orders.order_ID > 100;
```

MongoDB Query:

For MongoDB Shell

```
db.CustomerOrders.Find({Order_No: {$gt: 100}});
```

For C# Driver:

```
Collection.Find(searchQuery);
        where searchQuery = Query.GT("Order_No", 100);
```

Performance Test Result:

Number of Records	Data Loading Time (in Milliseconds)	
	MySQL	MongoDB
1000	3	24
2000	3	24
3000	3	22
4000	3	20
5000	3	22
6000	3	21
7000	4	23
8000	3	23
9000	3	25
10000	3	24

Table 4.5: Observations from Data Loading Test Run applying WHERE Clause

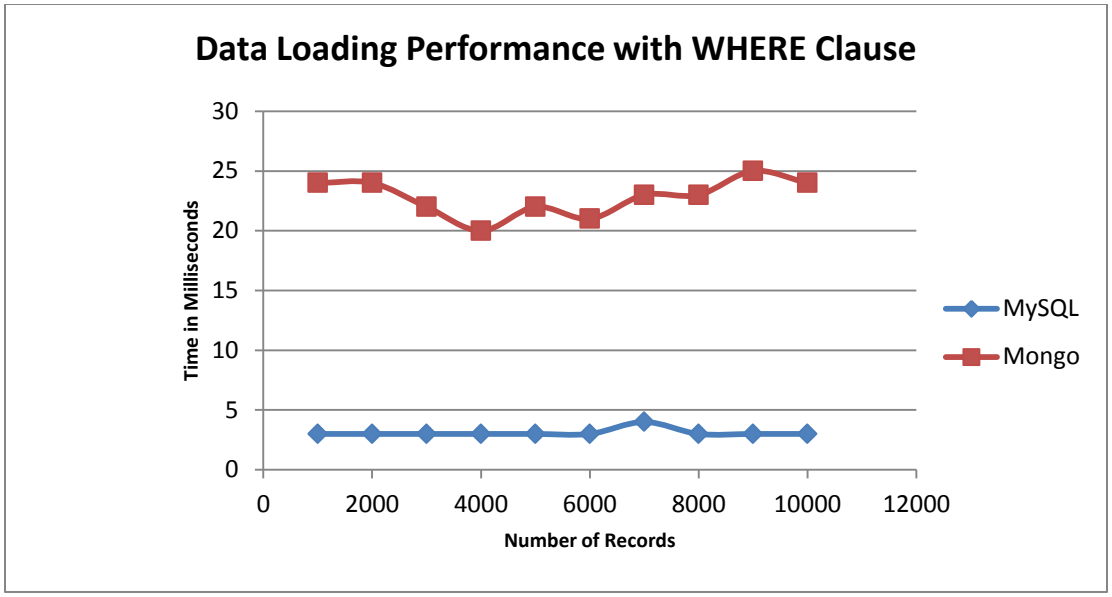


Fig.4.12: Performance Comparison for Data Loading with WHERE Clause

According to the performance comparison as represented in Fig.4.12, the performance trend is almost same as normal data loading. It is shown that MySQL database exhibits significant performance over MongoDB database which is steady and almost unchanged. On the other hand MongoDB takes more time compared to MySQL with unsteady performance which varies from 20 to 25 milliseconds. As MongoDB performances significantly more fluctuate than MySQL, standard deviations and coefficient of variations are used for more comparisons.

Table 4.6 represents the respective standard deviations and coefficient of variations derived from the performance test results recorded in Table 4.5. Based on data sets in Table 4.6, comparison graphs for standard deviations and coefficient of variations are plotted which are shown in Fig.4.13 and Fig.4.14 respectively. These two figures show that both MySQL and MongoDB performances' fluctuate for different number of records. According to Fig.4.14,

MongoDB exhibits smaller range of coefficient of variations compared to MySQL. MongoDB also has less fluctuation.

Number of Records	Standard Deviation (in Milliseconds)		Coefficient of Variation	
	MySQL	MongoDB	MySQL	MongoDB
1000	0.84	4.46	0.28	0.19
2000	0.42	5.72	0.14	0.24
3000	0.63	5.23	0.21	0.24
4000	0.67	5.02	0.22	0.25
5000	0.32	4.18	0.11	0.19
6000	0.32	4.20	0.11	0.20
7000	0.85	4.12	0.21	0.18
8000	0.42	5.32	0.14	0.23
9000	0.47	4.13	0.16	0.17
10000	0.42	5.25	0.14	0.22

Table 4.6: Standard Deviation and Coefficient of Variation Derived from Table 4.5 Data Sets

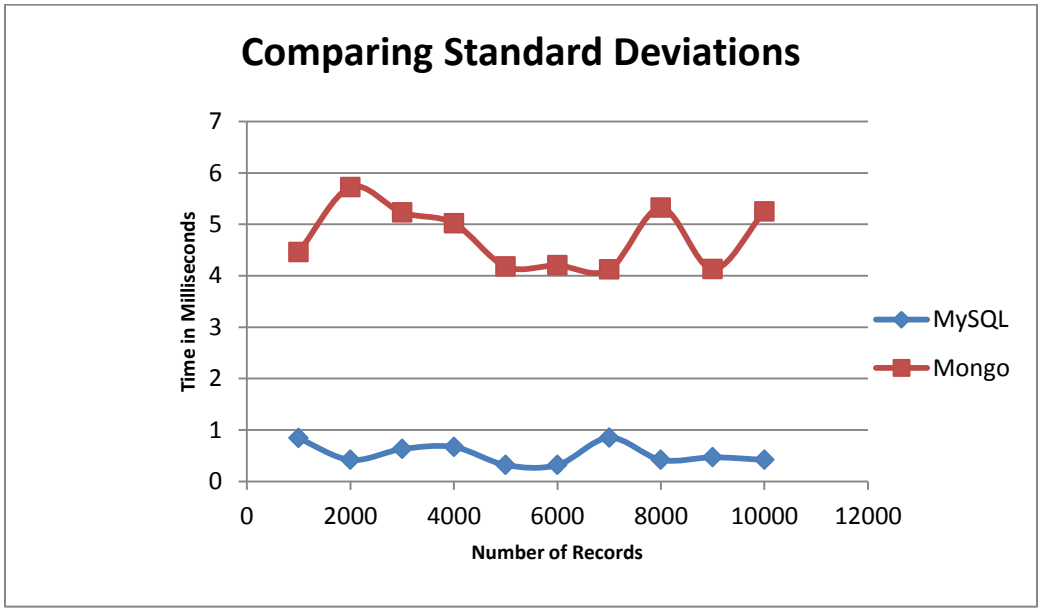


Fig.4.13: Comparison by Standard Deviations for WHERE Clause.

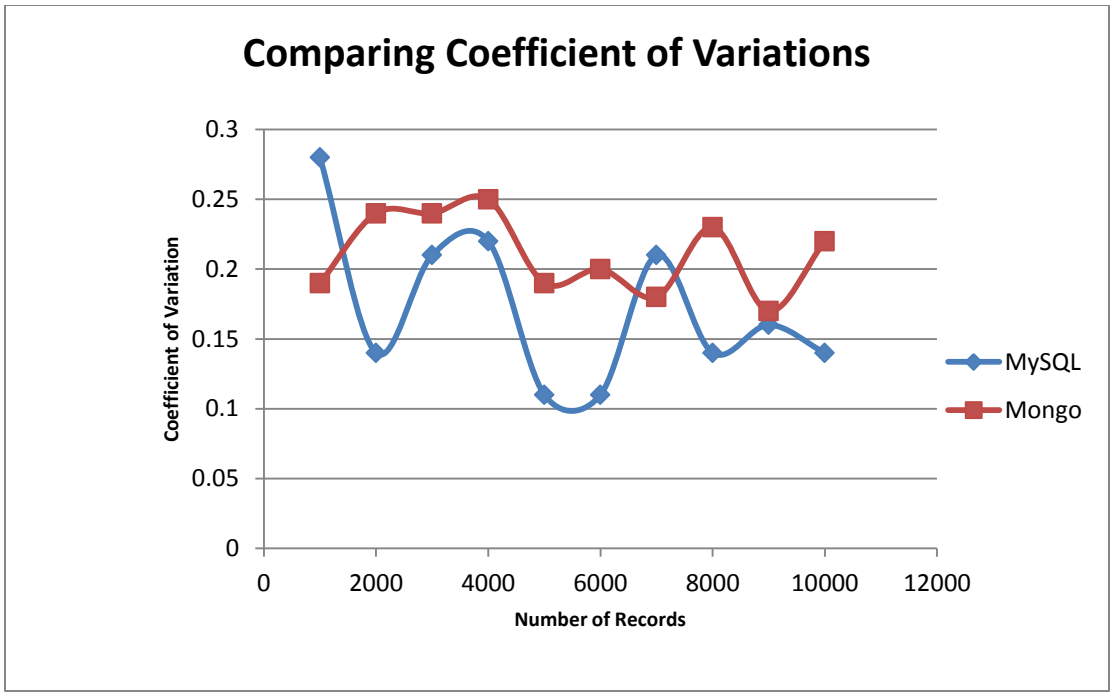


Fig.4.14: Comparison by Coefficient of Variations for WHERE Clause

Case 4: Data Loading Performance for Data Aggregation

Data aggregation is defined as "any process in which information is gathered and expressed in a summary form, for purposes such as statistical analysis" [36]. Data aggregation is one of the most important features for data mining as a part of getting desired compiled information from database. According to [40], as a part of Business Intelligence (BI) data mining plays an important role for expanding future business opportunities. Case 4 includes a comparative performance analysis based on data aggregation operations performed by both MySQL and MongoDB databases applying relevant AGGREGATE queries. In this case customer wise total sales are summed up from order details using GROUP BY clause for MySQL and MongoDB 'Aggregate' function which is available in JavaScript APIs. Observation data derived from ten different test runs for respective number of data sets ranging from 1000 to 10000 is recorded in the Table 4.7 and respective performance is plotted on a graph which is

shown in Fig.4.15. For MongoDB any complete order information is stored in a document with some other nested subdocuments. Therefore an additional unwind operation is done before main data aggregation. Following are the relevant data aggregation queries applicable for MySQL and MongoDB databases:

MySQL Query:

```
SELECT orders.order_cust_ID,customer.cust_fname,customer.cust_lname,
        SUM(order_details.order_prod_qty*order_details.order_prod_price)
FROM customer INNER JOIN (order_details INNER JOIN Orders ON order_Details.order_ID
        = orders.order_ID) ON customer.cust_id = orders.order_cust_ID
GROUP By orders.order_cust_ID,customer.cust_fname,customer.cust_lname
```

MongoDB Query:

For MongoDB Shell:

```
   db.CustomerOrders1.aggregate(
     [ {$unwind : "$Product"},
      {
       $group:
        {
          _id: {"Customer_ID": "$Customer.Customer_ID", "First_Name":
       "$Customer.First_Name", "Last_Name": "$Customer.Last_Name"},
          totalAmount: { $sum: { $multiply: [ "$Product.Price", "$Product.Quantity" ] } }
        }
      }
     ]
   )
```

For C# Driver:

```
       collection2.Aggregate(group);

       where,
            AggregateArgs group = new AggregateArgs()
            {
               Pipeline = new[]
                 { new BsonDocument{{"$unwind","$Product"}}, new BsonDocument
                 ("$group", new BsonDocument
                  {
                    {"_id", new BsonDocument
                     {

                     {"First_Name","$Customer.First_Name"},{"Last_Name","$Customer.Last_
                     Name"},
                      }
```

```
        },

        { "Total_Amount", new BsonDocument

            {
              {"$sum", new BsonDocument
                {
                  {"$multiply", new BsonArray {"$Product.Quantity",
 "$Product.Price"}}
                }
              }
            }
          }
        })
      }
};
```

Performance Test Result:

Number of Records	Data Loading Time (in Milliseconds)	
	MySQL	MongoDB
1000	7	2
2000	11	2
3000	13	2
4000	16	2
5000	37	2
6000	45	2
7000	52	2
8000	55	2
9000	62	2
10000	70	2

Table 4.7: Observations from Data Aggregation

Fig.4.15 exhibits a significant performance done by MongoDB database over MySQL database. According to the graph derived from data aggregation test run, the time required for retrieving data is almost unchanged and steady with the increase of data volume for NoSQL

MongoDB database whereas MySQL performance is gradually decreasing with the increase of data volume.

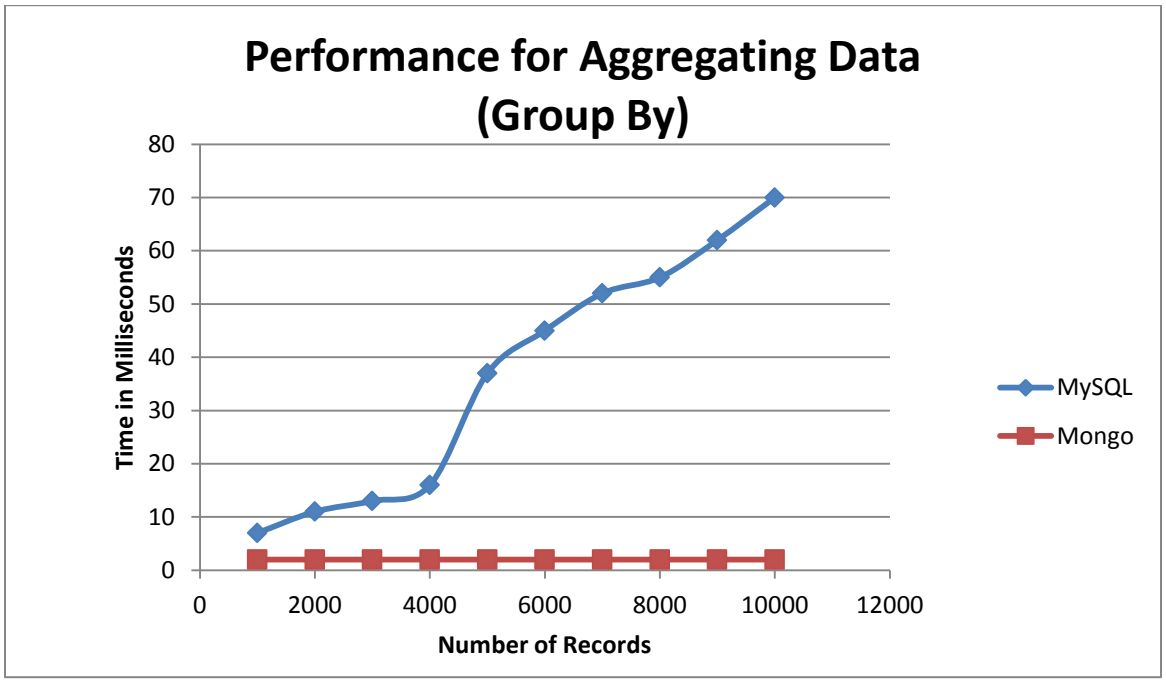


Fig.4.15: Performance Comparison for Data Aggregation

4.3 Development Agility

The development agility means how fast the development processes can response to meet the users' changed requirements. One of the twelve characteristics of agile methodologies' *continuous integration* is to integrate the new code including respective changes of the business requirements with the existing code after the completion of the changes [24]. The ultimate success of the software development process is to satisfy the end user with quality software and meet their changed requirements within the time constraint. In the area of Internet based businesses, the users need to change their requirements continuously and the adaptive methods of agile methodologies allow it to response and subsequently adapt quickly to those changing realities [25]. MongoDB provides development features that make developers' data modeling

and data querying jobs easy and handy [37]. This section includes some scenario based analysis that will provide some informative ideas about development performance in terms of complexity of development stages for fulfilling users' requirements.

Scenario 1: Customer Type and Sales Tax based on Province

There are mainly two kinds of sales can be performed like Personal Sales and Business to Business (B2B) sales based on which sales taxes are calculated. Basically sales tax is not primarily applicable for B2B sales. But exception also should be applicable when the B2B customer buys the product for their own consumption instead of trading. Therefore according to sales category there can be two types of customers – personal type customer and business type customer. Another type of customer should also be considered that may include religious or charitable organizations. Tax exemption will be applicable for this third type of customer.

At the same time sales tax rate differs from province to province. But the existing system does not have any provision for defining customer type or for keeping any information related to province wise tax rate. Here the database schema level will mainly be affected for incorporating these options with the existing system. Also new business rules should be introduced with the existing system to define customer type, customer type wise taxing rules and other relevant details according to the changed requirements. The Fig.4.16 shows the modified database scheme for incorporating the change request.

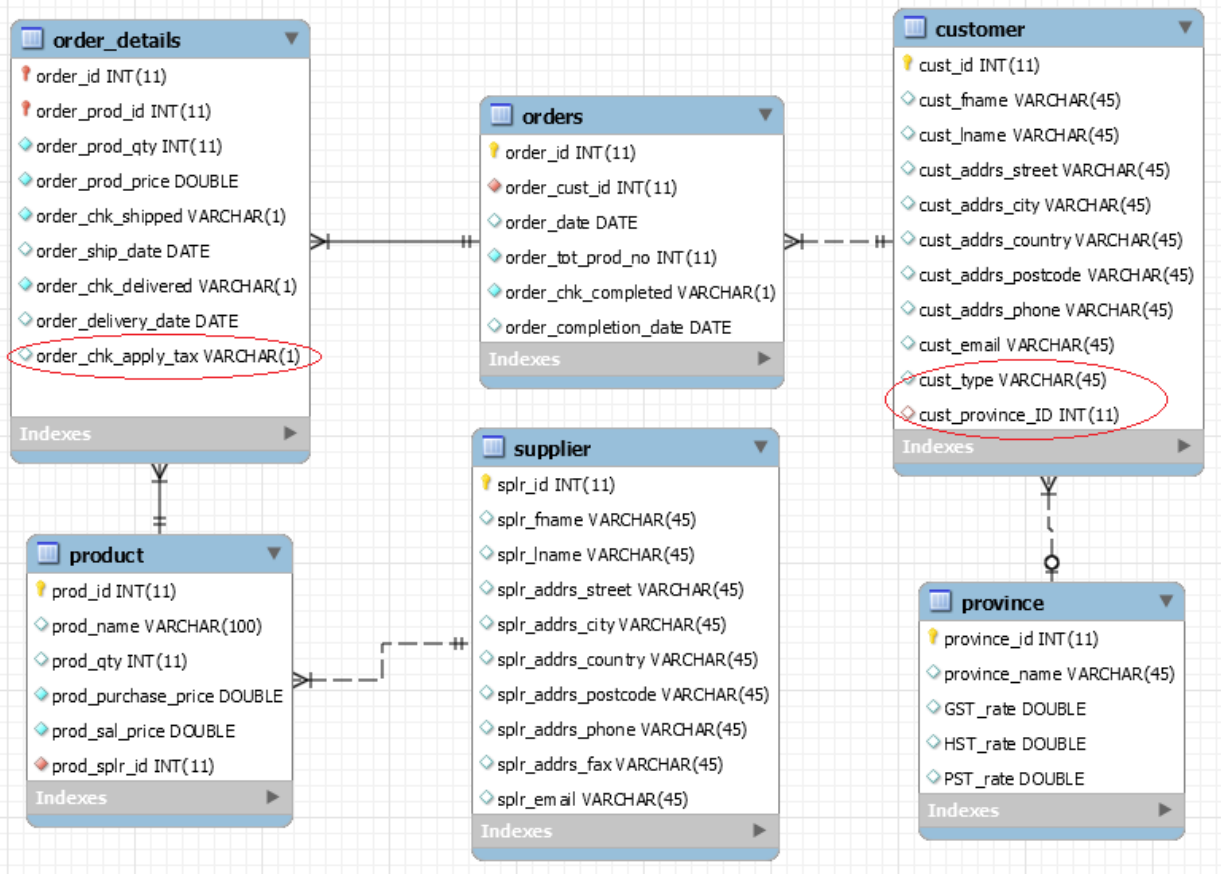


Fig.4.16 Modified schema for defining customer type and province wise sales tax calculation

From the above Fig.4.16**,** it is shown that a new table 'province' is introduced for keeping information about province wise GST, HST or PST rate. Two extra fields are added with Customer table where one of them is for defining customer type and the other one is for making relationship with new 'province' table. Order_Details table also include one extra field for checking whether a product is taxable or not. By default it will be 'Yes' for personal customers and 'No' for business and tax-exempt customers. Sometimes business customer can buy a product for their own consumption. In that case it will not be considered as B2B sales and they will change the default value to 'Yes' for that particular transaction. And then sales tax will be calculated based on value of this field and applicable provincial tax rate for a particular customer.

After getting this change request, the involvement of the possible tasks include identifying potential change with the system, analyzing and evaluating the change request to measure the workloads in different levels of system development, planning to distribute the task details in order to carry out the change request, and finally implementing, reviewing and closing the change request.

For relational model MySQL based system, this change request will effect on the following area:

- Schema Level Change – Database Administrator (DBA) need to be involved for changing existing schema. They need to go through some analysis in order to examine how it will impact on the databases.
- Application Development and Query Defining Level – New business rules should be introduced based on which design and development phase, and relevant query definition task specially related to table join criterion for the changes will need to be done.
- Reporting Level - Sales related new reports will be added with the system. Existing sales report will also need to be modified due to these changes.

On the other hand for its schema-less design approach MongoDB does not need extensive level of DBA involvement and it does not require any changes in the schema level for the above change request. Required changes can only be adjusted in the development level instead of database level. Therefor it requires less time to meet the change requirements compare with the relational MySQL based system.

Scenario 2: A product can have multiple suppliers

The current system allows having one-to-one relationship between product and supplier. But if the system needs to be changed like one product can have multiple suppliers, it would be massive involvement with MySQL based system. The changed requirement will mainly affect database schema. According to the existing schema the product and supplier have one-to-one relation. In order to incorporate the requirement the schema requires to include one more extra table that will allow product table to make one-to-many relationship with supplier table. On the other hand as MongoDB is schema less this change will not affect the MongoDB based system in terms of changing in schema. Fig.4.17 shows the changed effect on MySQL schema.

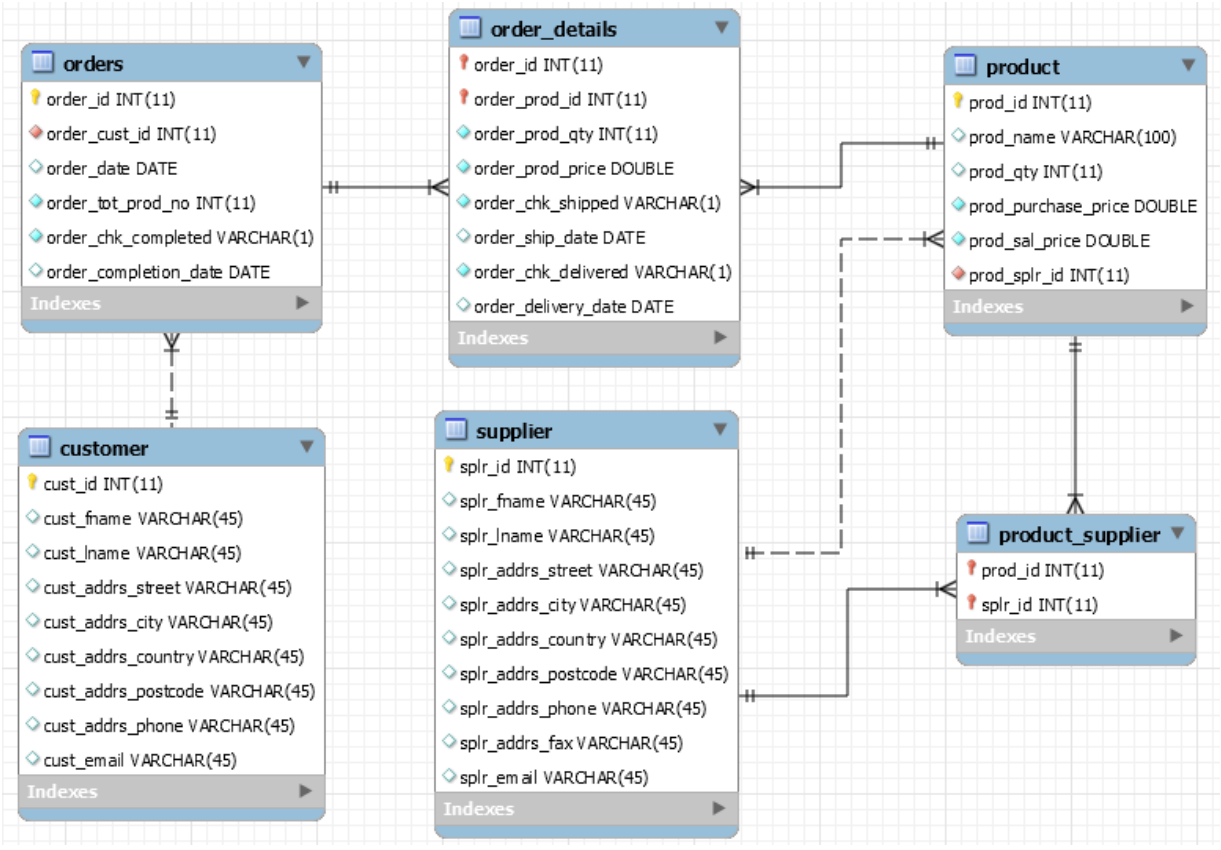


Fig.4.17 : Changed Schema for one-two-many product-supplier relationship.

The involvement for This change request also include all of the stages like identifying, analyzing, evaluating, planning and implanting the changes as described in the scenario 1 section.

According to the above scenario the impact of the request will affect in changing on query level and application development level. As the database schema needs to be changed, join conditions for the related queries must be changed. Therefore all of the queries associated with product and supplier needs to be redesigned and redeveloped. The effect of the application development level will mainly impact its inventory module as they need to maintain their product inventory by supplier. In this case the new product-supplier relational table will requires an additional field for their inventory module in order to maintain supplier wise product threshold quantity or re-order level. Therefore the system will require some new coding for incorporating these relationships and also require modifying existing code associated with updating supplier and product information. All of the reports associated with supplier and product also need to be modified and some additional reports may be introduced to accommodate these changes.

On the other hand as MongoDB collections do not have database schema like MySQL and this change will not affect that much on the MongoDB based system in terms of changing the schema. But at the same time MongoDB maintain implicit schema and migration code needs some changes in order to migrate persisted data according to the changed schema.

4.4 Simplicity of the Query

As a flexible database MongoDB offers rich features for data modeling and data query. Its developer oriented query features make the developers' life easy to write elegant queries. MongoDB query is performed using functions available from JavaScript APIs where queries are sent to MongoDB database as JSON objects. Usually the queries are sent to MongoDB by the database driver using 'find' method. This section includes comparative analysis on query structure of MySQL and MongoDB databases. Analysis will be done based on the following queries:

Case 1: Find out all order details for a particular customer:

MySQL Query:

The following query will return all of the order details for a particular customer from MySQL database:

```
SELECT orders.order_ID, orders.order_date, orders.order_cust_ID, customer.cust_fname,
    customer.cust_lname, customer.cust_addrs_street, customer.cust_addrs_city,
    customer.cust_addrs_postcode, customer.cust_addrs_country,
    customer.cust_addrs_phone, customer.cust_email, orders.order_chk_completed,
    orders.order_completion_date, order_details.order_prod_id, product.prod_name,
    order_details.order_prod_qty, order_details.order_prod_price,
    order_details.order_chk_shipped, order_details.order_ship_date,
    order_details.order_chk_delivered, order_details.order_delivery_date, supplier.splr_id,
    supplier.splr_fname, supplier.splr_lname, supplier.splr_addrs_street,
    supplier.splr_addrs_city, supplier.splr_addrs_postcode, supplier.splr_addrs_country,
    supplier.splr_addrs_phone, supplier.splr_email
FROM customer INNER JOIN (((order_details INNER JOIN product ON
    order_details.order_prod_ID = product.prod_id) INNER JOIN supplier ON
    product.prod_splr_id = supplier.splr_id) INNER JOIN Orders ON
    order_Details.order_ID = orders.order_ID) ON customer.cust_id = orders.order_cust_ID
    where customer.cust_fname="Wanda" and customer.cust_lname="Peterson";
```

MongoDB Query:

The following MongoDB query will retrieve all of the documents with order details for a particular customer from MongoDB database:

```
db.CustomerOrders.find(
    {
        $and: [{"Customer.First_Name":"Wanda"},{"Customer.Last_Name":"Peterson"}]
    })
```

Case 2: Order Details within a Date Range

The following MySQL and MongoDB query will return all of the order details for a particular date range:

MySQL Query:

```
SELECT orders.order_ID, orders.order_date, orders.order_cust_ID, customer.cust_fname,
        customer.cust_lname, customer.cust_addrs_street, customer.cust_addrs_city,
        customer.cust_addrs_postcode, customer.cust_addrs_country,
        customer.cust_addrs_phone, customer.cust_email, orders.order_chk_completed,
        orders.order_completion_date, order_details.order_prod_id, product.prod_name,
        order_details.order_prod_qty, order_details.order_prod_price,
        order_details.order_chk_shipped, order_details.order_ship_date,
        order_details.order_chk_delivered, order_details.order_delivery_date, supplier.splr_id,
        supplier.splr_fname, supplier.splr_lname, supplier.splr_addrs_street,
        supplier.splr_addrs_city, supplier.splr_addrs_postcode, supplier.splr_addrs_country,
        supplier.splr_addrs_phone, supplier.splr_email
FROM customer INNER JOIN (((order_details INNER JOIN product ON
        order_details.order_prod_ID = product.prod_id) INNER JOIN supplier ON
        product.prod_splr_id = supplier.splr_id) INNER JOIN Orders ON
        order_Details.order_ID = orders.order_ID) ON customer.cust_id = orders.order_cust_ID
        where orders.order_date between '2014-06-01' and '2014-06-30'
```

MongoDB Query:

```
db.CustomerOrders100.find(
        {"Order_Date": {$gte: ISODate("2014-06-01"),$lt: ISODate("2014-07-01")}
        })
```

Case 3: Order by Query

The following MySQL and MongoDB queries will sort the query result according to the customers:

MySQL Query:

```
SELECT orders.order_ID, orders.order_date, orders.order_cust_ID, customer.cust_fname,
        customer.cust_lname, customer.cust_addrs_street, customer.cust_addrs_city,
        customer.cust_addrs_postcode, customer.cust_addrs_country,
        customer.cust_addrs_phone, customer.cust_email, orders.order_chk_completed,
        orders.order_completion_date, order_details.order_prod_id, product.prod_name,
        order_details.order_prod_qty, order_details.order_prod_price,
        order_details.order_chk_shipped, order_details.order_ship_date,
        order_details.order_chk_delivered, order_details.order_delivery_date, supplier.splr_id,
        supplier.splr_fname, supplier.splr_lname, supplier.splr_addrs_street,
        supplier.splr_addrs_city, supplier.splr_addrs_postcode, supplier.splr_addrs_country,
        supplier.splr_addrs_phone, supplier.splr_email
FROM customer INNER JOIN (((order_details INNER JOIN product ON
        order_details.order_prod_ID = product.prod_id) INNER JOIN supplier ON
        product.prod_splr_id = supplier.splr_id) INNER JOIN Orders ON
        order_Details.order_ID = orders.order_ID) ON customer.cust_id = orders.order_cust_ID
ORDER BY orders.order_cust_ID;
```

MongoDB Query:

```
db.CustomerOrders.find().sort({"Customer.Customer_ID":1})
```

Case 4: Aggregate Query

MySQL Query:

```
SELECT orders.order_cust_ID,customer.cust_fname,customer.cust_lname,
        SUM(order_details.order_prod_qty*order_details.order_prod_price)
FROM customer INNER JOIN (order_details INNER JOIN Orders ON order_Details.order_ID
        = orders.order_ID) ON customer.cust_id = orders.order_cust_ID
GROUP By orders.order_cust_ID,customer.cust_fname,customer.cust_lname
```

MongoDB Query:

For MongoDB Shell:

```
   db.CustomerOrders1.aggregate(
     [ {$unwind : "$Product"},
       {
```

```
        $group:
         {
           _id: {"Customer_ID": "$Customer.Customer_ID", "First_Name":
         "$Customer.First_Name", "Last_Name": "$Customer.Last_Name"},
           totalAmount: { $sum: { $multiply: [ "$Product.Price", "$Product.Quantity" ] } }
         }
       }
     ]
  )
```

Through its BSON data structure and powerful query features MongoDB supports most of the query functions available in the relational model by provisioning high-speed data access to mass data [14]. From the above four cases we can see MySQL query is somewhat complex in compared to the structure of MongoDB query as MySQL query needs to join different relational tables according to their relationship to get the complete total information. On the other hand MongoDB query is straightforward as there is no relational schema. MySQL query requires too many lines to express the total query that make it somewhat complex for developmental purposes.

4.5 Findings

Based on the above measures for evaluating the thesis objective, a summary of the findings are as follows:

- The data verification shows that data migration process from MySQL relational database to NoSQL MongoDB database was performed successfully by applying the proposed methodology. MongoDB also performs all of the basic operations like INSERT, UPDATE, DELETE and SELECT, which are identical to MySQL.

- MongoDB performs significantly better than MySQL for data storage related operations that include INSERT, UPDATE and DELETE.
- MongoDB exhibits an outstanding performance for data aggregation and data sorting which indicates incredible opportunities with MongoDB in the area of statistical analysis which is a part of Business Intelligence (BI) aspect for analytical data management. Enterprises will be interested for their future business growth based on their business analysis gearing with the MongoDB performance.
- Based on some selective scenarios, it is shown that MongoDB provides development agility in terms of meeting the continuous change requirements.
- Using JavaScript APIs, MongoDB provides simple and straightforward query structures that facilitate development work for the developers.

Chapter 5

5. Conclusions and Future Works

The demand for NoSQL databases is increasing because of their diversified characteristics that offer rapid smooth scalability, great availability, distributed architecture, significant performance and rapid development agility. The main result of this thesis was to provide a methodology for migrating rapidly growing enterprise data from back end relational model to NoSQL data store. Specifically data migration facilitates enterprises' Online Analytical Processing (OLAP) which is the part of broader category of BI. The ways relational model and NoSQL databases store their data are totally different from relational databases. RDBMSs follow strictly a predefined schema and store their data in different tables through relationship according to the structure of the schema, whereas schema-less NoSQL have a different way for storing and retrieving their flexible, unstructured or semi-structured data available in the different format of databases that include document, key-value, columnar and graph data store group.

The structural differences between RDBMS and NoSQL databases makes the data migration process challenging. From different choices of databases this thesis selected open-source MySQL from relational databases group and MongoDB from the NoSQL document databases group as the test case. As the document database MongoDB is formed with collection of JSON objects and .NET C# provides completely asynchronous driver to interact with MongoDB, the object oriented approach was taken for migrating data utilizing the underlying technological advantages from C# language available in the .NET platform. This thesis accomplished a successful implementation of data migration process following the steps of traditional ETL process that includes data Extraction, Transformation and finally Loading where

data extraction was done using SQL query, then extracted data was transformed into different objects using ORM and finally loaded or saved it to MongoDB JSON-style document.

Some measures including Verification of Data Migration, Performance, Development Agility and Query Simplicity, were deliberated as a part of evaluating this thesis works. Based on these measures the major findings of this thesis represented that data migration using the proposed methodology was achieved successfully. The findings also indicated that MongoDB exhibited an outstanding performance on data aggregation and data sorting that can attract the enterprises for their BI reporting which is based on analytical data management. BI refers to OLAP – a simple type of data aggregation that facilitates enterprises for generating their particular group based reports [36] and MongoDB may have a great opportunity for OLAP. But this thesis did not contribute exploring any opportunities on the area of Online Transaction Processing (OLTP) that ensures data integrity for transaction-oriented application and can be considered as the future scope of works. As an expansion of this thesis works, some of the future scope of works is given below:

- Combinational Idea: OLTP has ACID properties in order to maintain data integrity. NoSQL does not support ACID properties for transactional database management system. According to the literature reviews done in this thesis no solution is still available to overcome this constraint though some of the papers discussed about a substitute BASE as a part of supporting their transactional requirements which is comparatively weaker than ACID. This thesis also did not explore anything on this area. Future scope of works can include finding out a true alternative for this issue. Considering the popularity and stability of RDBMS for years after years, a combinational approach is argued which is called ‘SomeSQL’ where NoSQL will be integrated with RDBMS as an additional tools

for providing large-data oriented applications [27]. The open-source Object-Relational Database Management System PostgreSQL introduces two fascinating NoSQL features within its relational environment those include HStore – a key-value store and JSONB – a binary version of JSON storage which is like BSON that MongoDB uses for its storage [20]. Future scope of works may include PostgreSQL for the combinational idea - a hybrid of the two which is to keep the transactional system tied to the relational environment, and make the data analysis and data mining activities tied to NoSQL database.

- Generalized Data Migration Tool: The future scope of work may include developing a generalized data migration tool. The current implementation is not generalized. In order to make it as a generalized data migration tool, an interface can be introduced where the interface will allow entering the query string for data extraction. Based on this data extraction a list of data fields will be generated which will be mapped into different objects and their subsets of objects (if required) by selecting and defining with the help of interface. The generalized implementation may needs to include a dynamic class which will create all of the respective objects. Then finally the data migration process can follow the proposed data migration flow diagram.

Bibliography

[1] Subashini, S., Kavitha, V. (2012), "A Metadata Based Storage Model for Securing Data in Cloud Environment", American Journal of Applied Sciences 9 (9): 1407-1414, 2012,

[2] Patel, A. B., Birla M., Nair, U. (2012), "Addressing Big Data Problem Using Hadoop and Map Reduce", Engineering (NUiCONE), 2012 Nirma University International Conference on Engineering, pp. 1-5.

[3] Arora, I., Gupta, A. (2012) "Cloud Databases: A Paradigm Shift in Databases", IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 4, No 3, July 2012

[4] Cruz, F., Gomes, P., Oliveira, R., Pereira, J. (2011), "Assessing NoSQL Databases for Telecom Applications", IEEE 13th Conference on Commerce and Enterprise Computing, pp. 267-270.

[5] Ganiee, N., Bhargava R., "NOSQL for Interactive Applications", International Journal of Allied Practice, Research and Review, Website: www.ijaprr.com (ISSN 2350-1294), retrieved on 2015.

[6] Agrawal, D., Das, S. Abbadi, A. E. (2011), "Big Data and Cloud Computing: Current State and Future Opportunities", Proceedings of the 14th International Conference on Extending Database Technology, pp. 530-533.

[7] Burtica, R., Mocanu,E. M., Andreica, M. I., Ţăpuş, N. (2012) "Practical application and evaluation of no-SQL databases in Cloud Computing", Systems Conference (SysCon), 2012 IEEE International, pp. 1-6.

[8] Han, J., Song, M., Song, J. (2011), "A Novel Solution of Distributed Memory NoSQL Database for Cloud Computing", 2011 10th IEEE/ACIS International Conference on Computer and Information Science, pp. 351-355.

[9] Konstantinou, I., Angelou E., Boumpouka, C., Tsoumakos, D., Koziris, N. (2011), "On the Elasticity of NoSQL Databases over Cloud Management Platforms", Proceedings of the 20th ACM international conference on Information and knowledge management, pp. 2385-2388 .

[10] Bhadauria, R., Sanyal, S. (2012), "Survey on Security Issues in Cloud Computing and Associated Mitigation Techniques", International Journal of Computer Applications, Volume 47- Number 18, June 2012, pp. 47-66.

[11] Zhang, Q., Cheng, L., Boutaba, R. (2010), "Cloud computing: state-of-the-art and research challenges", Journal of Internet Service Application, 1 (1) (2010), pp. 7–18.

[12] Mohamed A. M., Altrafi G. O., Ismail O. M. (2014), "Relational vs. NoSQL Databases: A Survey", International Journal of Computer and Information Technology (ISSN: 2279–0764) Volume 03 – Issue 03, May 2014.

[13] Avram, A. (2012), "Transitioning from RDBMS to NoSQL. Interview with Couchbase's Dipti Borkar", Online Article. retrieved from http://www.infoq.com/articles/Transition-RDBMS-NoSQL, posted on September 2012.

[14] Han, J,, Haihong, E., Le G., Du, J. (2011), "Survey on NoSQL Database", Pervasive Computing and Applications (ICPCA), 2011 6th International Conference, PP. 363 - 366.

[15] Cattell, R. (2010), "Scalable SQL and NoSQL Data Stores", Newsletter - ACM SIGMOD Record archive Volume 39 Issue 4, December 2010, PP. 12-27.

[16] Moniruzzaman, A B M, Hossain, s. A. (2013), "NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison", International Journal of Database Theory and Application Vol. 6, No. 4. 2013.

[17] Hecht, R., Jablonski, S. (2011), "NoSQL Evaluation A Use Case Oriented Survey", Cloud and Service Computing (CSC), 2011 International Conference on Cloud and Service Computing, pp. 336-341.

[18] Vaish, G. (2013), "Getting Started with NoSQL", Retrieved through Ryerson Library Catalogue from http://proquest.safaribooksonline.com/book/databases/9781849694988.

[19] "Rules of Engagement – NoSQL Column Data Stores", Online article retrieved from http://www.ingenioussql.com/2013/02/, posted on February 2013.

[20] Lerner, R. (2015), "PostgreSQL, the NoSQL Database", Online article retrieved from http://www.linuxjournal.com/content/postgresql-nosql-database, posted on January 2015.

[21] Mughees, M. (2013), "Data Migration FROM Standard SQL TO NoSQL", A thesis paper retrieved from http://ecommons.usask.ca/handle/10388/ETD-2013-11-1342.

[22] Codd, E. F. (1970), "A relational model of data for large shared data banks", Communications of the ACM, v.13 n.6, p.377-387, June 1970 [doi>10.1145/362384.362685].

[23] Dede, E., Sendir, B., Kuzlu, P., Hartog, J., Govindaraju, M. (2013), "An Evaluation of Cassandra for Hadoop", Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference, pp. 494-501.

[24] Ferreira, C., Cohen, J. (2008), "Agile Systems Development and Stakeholder Satisfaction: A South African Empirical Study", Proceedings of the 2008 annual research conference of

the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology, pp. 48-55.

[25] Livermore, J.A. (2007), "Factors that Impact Implementing an Agile Software Development Methodology", SoutheastCon, 2007. Proceedings. IEEE DOI: 10.1109/SECON.2007.342860.

[26] Tudorica, B.G., Bucur, C. (2011) "A comparison between several NoSQL databases with comments and notes", Roedunet International Conference (RoEduNet), 2011 10th DOI: 10.1109/RoEduNet.2011.5993686, pp. 1-5.

[27] Lerner, R. M. (2010), "At the Forge NoSQL? I'd Prefer SomeSQL", Online article retrieved from http://www.linuxjournal.com/article/10720, posted on April 2010.

[28] Date, C.J. (2003), "An Introduction to Database Systems"(8th Edition), United States of America, Pearson Education, Inc.

[29] Codd, E. F., "Extending the database relational model to capture more meaning", ACM Transactions on Database Systems (TODS), v.4 n.4, p.397-434, Dec. 1979 [doi>10.1145/320107.320109].

[30] Fagin R. (1977) , "Multivalued Dependencies and a New Normal Form for Relational Databases", ACM Transactions on Database Systems (TODS), v.2 n.3, p.262-278, Sept. 1977 [doi>10.1145/320557.320571].

[31] Kent W. (1983), "A Simple Guide to Five Normal Forms in Relational Database Theory", Communications of the ACM, v.26 n.2, p.120-125, Feb. 1983 [doi>10.1145/358024.358054].

[32] Chamberlin, D. D., Boyce, R. F. (1974), "SEQUEL: A Structured English Query Language", Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control, p.249-264, May 01-03, 1974, Ann Arbor, Michigan [doi>10.1145/800296.811515].

[33] Yang., H, Dasdan., A., Hsiao, R., Parker, D. S. (2007), "Map-Reduce-Merge: Simplified Relational Data Processing on Large clusters", Proceedings of the 2007 ACM SIGMOD international conference on Management of data, June 11-14, 2007, Beijing, China [doi>10.1145/1247480.1247602].

[34] Tharakan, R. (2010), "Brewers CAP Theorem on distributed systems", Online article retrieved from http://www.royans.net/wp/2010/02/14/brewers-cap-theorem-on-distributed-systems/, posted on February 14 2010.

[35] Gilbert, S., Lynch, N. A. (2012), “Perspectives on the CAP Theorem”, IEEE Computer, vol. 45, no. 2, DOI: 10.1109/MC.2011.389, pp. 30-36.

[36] Rouse, M., "Data Aggregation", Online article retrieved from http://searchsqlserver.techtarget.com/definition/data-aggregation.

[37] Tauro, C. J. M., Patil, B. R., Prashanth, K.R. (2013), "A Comparative Analysis of Different NoSQL Databases on Data Model, Query Model and Replication Model", In Proceedings of International Conference on ”Emerging Research in Computing, Information, Communication and Applications” ERCICA. Elsevier, 2013.

[38] "MySQL The World's Most Popular Open Source Database", ORACLE online article retrieved from http://www.oracle.com/us/products/mysql/overview/index.html, February 2015.

[39] "MongoDB .NET DriverThe next generation .NET driver for MongoDB", Online MongoDB resources retrived from http://mongodb.github.io/mongo-csharp-driver/?jmp=docs, February 2015.

[40] "Think Before You Dig: Privacy Implications of Data Mining & Aggregation", NASCIO Research Brief, September 2004, retrieved from http://www.nascio.org/publications/documents/nascio-datamining.pdf.

[41] Feinleib, D. (2012), “Big Data and NoSQL: Five Key Insights”, Forbes online article retrieved from http://www.forbes.com/sites/davefeinleib/2012/10/08/big-data-and-nosql-five-key-insights, May 2015.

[42] Codd, E.F., Codd, S.B., Salley, C.T., “Providing OLAP to User-Analysts: An IT Mandate”, Retrieved from http://www.minet.uni-jena.de/dbis/lehre/ss2005/sem_dwh/lit/Cod93.pdf, May 2015.

[43] Couchbase (2013), “Making the Shift from Relational to NoSQL”, Retrieved from http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/Couchbase_Whitepaper_Transitioning_Relational_to_NoSQL.pdf, May 2015.