

Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Engineering (FEM)



Bachelor Thesis

Comparison between SQL and NoSQL Database Systems: a case study of online elections system.

Mohamad Othman

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

BACHELOR THESIS ASSIGNMENT

Mohamad Othman

Systems Engineering and Informatics
Informatics

Thesis title

Comparison between SQL and NoSQL Database Systems: a case study of online elections system.

Objectives of thesis

The aim of this bachelor thesis is to describe the key similarities and differences between the SQL-based and NoSQL database systems on a practical application of electronic elections. With enormous growth in data complexity and the higher demand for more flexible and scalable database systems such e-democracy applications, for example, there are many situations that the traditional RDMS is not the optimal solution. While the dynamic and scalable nature of NoSQL databases can make a better solution for web applications featuring complex and hierarchical data structure.

Methodology

The methodology of the thesis is based on the study of literature and resources, as well as creating sample applications for an elections system using Apache CouchDB, PHP, and MySQL in order to show how to use an object-based database and how it can simplify the storage and retrieval of hierarchical data.

The proposed extent of the thesis

30 – 60 pages

Keywords

SQL, NoSQL, RDMS, couchdb, mongodb, oracle, mssql, elections

Recommended information sources

CouchDB: The Definitive Guide

NoSQL and SQL Data Modeling: Bringing Together Data, Semantics, and Software

NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence

Expected date of thesis defence

2020/21 WS – FEM (February 2021)

The Bachelor Thesis Supervisor

doc. Ing. Vojtěch Merunka, Ph.D.

Supervising department

Department of Information Engineering

Electronic approval: 19. 2. 2020

Ing. Martin Pelikán, Ph.D.

Head of department

Electronic approval: 19. 2. 2020

Ing. Martin Pelikán, Ph.D.

Dean

Prague on 10. 03. 2021

Declaration

I declare that I have worked on my bachelor thesis titled "Comparison between SQL and NoSQL Database Systems: a case study of online elections system." by myself and I have used only the sources mentioned at the end of the thesis. As the author of the bachelor thesis, I declare that the thesis does not break the copyrights of any person.

In Prague on 14.03.2021

Acknowledgment

I would like to thank Doc. Ing. Vojtěch Merunka, Ph.D., for their advice and support during my work on this thesis. And all the guidance provided on all stages of writing this thesis.

Comparison between SQL and NoSQL Database Systems: a case study of online elections system.

Abstract

This bachelor thesis aims to describe the key similarities and differences between the SQL-based and NoSQL database systems on a practical application of electronic elections. With the enormous growth in data complexity and the higher demand for more flexible and scalable database systems such as e-democracy applications. There are many situations that the traditional RDMS is not the optimal solution. While the dynamic and scalable nature of NoSQL databases can make a better solution for web applications featuring complex and hierarchical data structure.

Keywords: SQL, NoSQL, RDMS, CouchDB, MongoDB, oracle, MSSQL, elections

Porovnání databázových systémů SQL a NoSQL: případová studie online volebním systému.

Abstrakt

Tato bakalářská práce si klade za cíl popsat klíčové podobnosti a rozdíly mezi databázovými systémy založenými na SQL a NoSQL. S nárůstem složitosti dat a vyšší poptávkou po flexibilnějších a škálovatelných databázových systémech tradiční RDMS nemusí být vždy optimálním řešením. Dynamická a škálovatelná povaha databází NoSQL je vhodnějším řešením pro webové aplikace s komplexními a hierarchickými datovými strukturami.

Klíčová slova: SQL, NoSQL, RDMS, CouchDB, MongoDB, oracle, MSSQL, volby

Table of content

1	Introduction	12
2	Objectives and Methodology	13
2.1	Objectives	13
2.2	Methodology	13
3	Intro to data structures and database models	14
3.1	Brief history and origins of the DBMS	14
3.2	The difference between data structures	15
3.2.1	Structured Data:	15
3.2.2	Unstructured Data:	15
3.2.3	Semi-Structured Data:	16
3.3	The most common database models	16
3.3.1	Hierarchical model	17
3.3.2	Network model	17
3.3.3	Relational model	18
3.3.4	Object-Oriented model	18
4	An Overview of NoSQL	19
4.1	The relationship between ACID, BASE, and CAP Theorem	20
4.2	NoSQL Storage Types	22
4.2.1	Key-Value Databases	22
4.2.2	Document Databases	23
4.2.3	Graph Databases	24
4.2.4	Column Databases	25
5	Advantages and Drawbacks	25
5.1	Scalability	25
5.2	Eventual consistency vs Strong consistency	27
5.3	Multi-Tenant Support	28
5.4	Normalization vs Denormalization	30
6	Practical Part	32
6.1	Description of the project	32
6.2	Technology stack choices	32
6.3	Database Design	33
6.3.1	The logical database designs	33
6.3.2	The physical database designs	34
6.4	Database implementation	36
6.4.1	Creating the relational database	36

6.4.1.1	Creating the NoSQL documents.....	38
6.4.2	Application implementation and layout.....	39
6.5	Inserting Data:.....	39
6.6	Data Retrieval.....	40
6.6.1	SQL SELECT statement.....	40
6.6.2	CouchDB Views	42
6.7	Performance Comparison.....	44
6.7.1	Testing Setup	44
6.7.2	Test Scenario.....	45
6.7.3	Benchmark Results	45
7	Results and Discussion.....	48
7.1	Getting Started	48
7.2	Design and Implementation Stage	48
7.3	Data Storage and Retrieval.....	48
7.4	Performance Benchmark	49
8	Conclusion.....	50
9	References	51
10	Appendix.....	53

List of pictures

Figure 1: Hierarchical model	17
Figure 2: Network model.....	17
Figure 3: Relational Model	18
Figure 4: Object-Oriented Model.	19
Figure 5: The relationship between ACID, BASE, and CAP Theorem.	22
Figure 6: Graph Model example used in Neo4j Graph Database.....	24
Figure 7: Windows server approach for SQL clustering.	26
Figure 8: Conceptual Depiction of Replication with Eventual Consistency.	27
Figure 9: Conceptual Depiction of Replication with Strong Consistency.....	28
Figure 10: Single Tenant vs Multi-Tenant Architecture.....	29
Figure 11: The logical database designs. Source: Author	33
Figure 12: The physical database designs Source: Author	34
Figure 13: CouchDB candidate document. Source: Author	35
Figure 14: CouchDB questions list document. Source: Author.....	35
Figure 15: Create Database SQL Statement Source: Author.....	36
Figure 16: Tables create SQL statement. Source: Author	38
Figure 17: Database creation page in Fauxton GUI tool. Source: Author.....	38
Figure 18: Creating CouchDB database using cURL tool. Source: Author	38
Figure 19: Inserting new document in CouchDB. Source: Author.....	38
Figure 20: Candidate Class Model in PHP & Yii Framdwork. Source: Author.....	39
Figure 21: The page in the application that displays Candidates information's.....	39
Figure 22: Candidate creation form in the application layout.	40
Figure 23: SQL insert statements.....	40
Figure 24: Sample SQL Query for data retrieval.....	41
Figure 25: Sample SQL Query with Joins for data retrieval.	41
Figure 26: Sample SQL Query with COUNT function for data aggregation.....	42
Figure 27: View creation in CouchDB.	42
Figure 28: View with reduce function creation in CouchDB.	43
Figure 29: Calling CouchDB view using windows 10 terminal and cURL tool.	43
Figure 30: SQL Query for counting candidates.....	45
Figure 31: CouchDB map function for counting candidates.	45
Figure 32: MySQL and CouchDB benchmark and average query execution time.	46

Figure 33: Comparison between MySQL execution time and Big O Complexity.47

List of tables

Table 1: Average count query benchmark results after 10 runs.46

Table 2: MySQL Query execution time vs Big O Notation.47

List of equations

Equation 1: Equation for calculation query execution time.46

List of abbreviations

- DB: Database
- RDBMS: Relational Database Management System
- DBMS: Database Management System
- SQL: Structured Query Language
- NoSQL: Not-Only SQL

1 Introduction

From the pre-digital time to the modern days, there has been many forms for storing data and keeping records, from paper journals to cloud-powered database management systems. And by the invention of the computer and the use of basic files to store data, the database management systems have evolved to meet the demand and the need of the new world.

With the start of 21st century, the internet growth and accessibility to social media, search engines, and many enterprises and personal sites and applications and many other sources has started generating information that is complex in nature and massive in amount every day. Which requires a more modern flexible and scalable database solutions to overcome the shortcomings of the widely used traditional database management systems known SQL.

This thesis is focusing on the latest evolution in the database systems which is known as NoSQL (Not only SQL) and how it compares to the traditional database management systems (SQL).

The theoretical part of this thesis outlines the different structures and forms of data and the common database models to handle these structures, it also focuses on the theoretical principles behind NoSQL class of databases and how it compares to SQL and their advantages and drawbacks, In the practical part, the thesis demonstrates a real-world usage of SQL and NoSQL databases in an example of an election application implemented using both engines.

Furthermore, In the practical part, there is a benchmark of both databases using datasets up to 1 million record.

2 Objectives and Methodology

2.1 Objectives

The objectives of this thesis are to build a web app that utilizes both types of database systems to achieve the same functionality. And then highlighting the pros and cons of each database type based on the sample app code.

2.2 Methodology

The methodology of the thesis is based on the study of literature and resources, to highlights the advantages and drawbacks of using NoSQL databases as well as creating sample application in the practical part of the thesis, for an elections system using Apache CouchDB, PHP language, and MySQL to show how to use NoSQL Document database and how it can simplify the storage and retrieval of hierarchical data. Also a performance benchmark will be made to highlight the scalability of both solutions.

3 Intro to data structures and database models

3.1 Brief history and origins of the DBMS

I would like to start my thesis by taking a historical look at databases from pre-digital time to the origin of the first database management systems to current days where new solutions are demanded by the widespread internet usage on all kinds of devices and the utilization of cloud computing.

Starting in ancient times, to not so long ago, many enterprises, governments, and businesses created their way to record the data related to their business. Be it the log of hotel visitors or the state and medical record of a patient in a hospital bed. These systems have varied and the way they recorded data were sometimes completely different, according to their need. The medium has mostly been books and paper but by the invention of the computer and the availability of direct-access storage, the term “database” was first introduced by developing the first database software.

Before that, the only way to store data was from unrelated files. Programmers had to go to great lengths to extract the data, and their programs had to perform complex parsing and relating.

In the 1970s the first relational database system based on SQL language was created and the term Relational Database Management System (RDBMS) was introduced.

While in the 1980s and by the introduction of the IBM personal computer, there has been a great demand for computer systems, and many companies adopted the SQL language as a world standard, and all the previous database models started disappearing. On the other hand, in 1985, the term object-oriented database system was introduced where the information inside the system is stored and structured similar to the objects in OOP programming languages. Which – in theory – could allow for storing complex and nested objects which could substitute the need for relational models.

In the 1990s, the widespread of the internet has brought exponential demand for more efficient and less complex computer systems which brought the first Object-Rational-Mapping (ORM) to bridge the gap between the RDBMS and the programming languages.

Also, “The Object-oriented Database System Manifesto” was written as an attempt to define an object-oriented database system features and characteristics

Another form of data storage was Extensible-Mark-up-Language (XML) was introduced. Which are structured object-like documents.

Early 21st Century, the term “NoSQL” was introduced to describe the databases that do not follow the ACID standards used in SQL engines but rather follow the CAP theorem to allow the data to be grouped naturally and logically and fit many more data models like unstructured and semi-structured models.

There are 4 major categories of NoSQL databases (document-oriented, graph-based, column-based, key-value, and hybrid) which we will look at in a later section of this thesis but first, we need to explore the common data structures and models.

3.2 The difference between data structures

The main challenge of the design of every real-world application is to classify and analyze the data relevant to the application domain. And this process affects the programming stack and database choices of that application. In an excellent article by “Tehreem Naeem”, he classifies different data structures into the following: (Naeem, 2020)

3.2.1 Structured Data:

When the raw data is formatted and can be transformed into a well-defined schema. For example addresses, credit card numbers, financial reports, etc... This type of data can easily be defined by rows and columns in and RDBMS can easily store it and retrieve it.

3.2.2 Unstructured Data:

When the data is difficult to process and organize due to its complex formatting and the loosely defined properties for each record. An example of this kind of data would be online survey questions and answers where even though all questions belong to the same collection, each one may contain different fields where a multiple-choice question will have different

attributes than text-based questions or another example can be the content of an instant messaging application where the same conversation may include text, videos, photos, and files. Storing such a structure in traditional SQL databases is very challenging. and that's why Object-Orient databases are preferred for such a job.

3.2.3 Semi-Structured Data:

Data does not always come as structured or unstructured but sometimes it is a mixture of both types at the same time. Semi-structured data does not have a rigid structure but has some defining and consistent characteristics. Consider e-mail service. The email content can be unstructured and may contain a mixture of objects. But it also has a well-defined attribute like send date, recipients, and subject.

NoSQL databases are considered as popular to handle semi-structured data. Because this type of data cannot be stored and queried efficiently in RDBMS.

3.3 The most common database models

Following up on the previous section, the process of building an application includes building the conceptual design of the database using the Entity-relationship model (ERD) and Semantic data models (SDM).

The next step is to identify the database models. To define what is a database model is, it's great to quote the book „Advanced Database Models“ by Dr.-Ing. Eike Schallehn

*“A **data model** is a model that describes in an abstract way how data is represented in an information system or a database management system. A **database model** is a data model for a database system. It provides a theory or specification describing how a database is structured and used. “*

(Schallehn, 2019).

The DBMS implements one or more database models. There are many kinds of data models. And Some of the most common ones include:

3.3.1 Hierarchical model

One of the first important database models, it was developed in the late 50s and used in IBM Information management system one of the first successful DDBMS in the 1960s.

The data is organized like an upside-down tree structure with the leaves are flat records, connected using links.

Each record has one parent and contains multiple fields and each field represents one value.

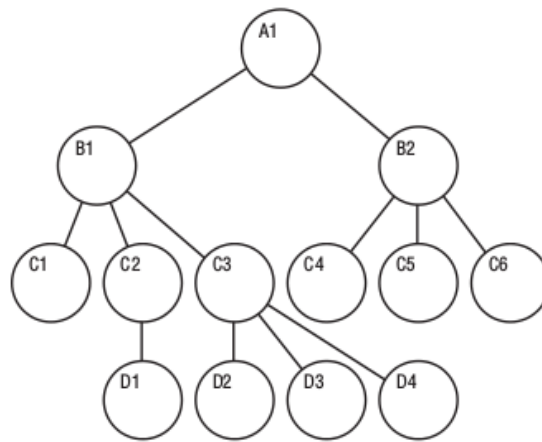


Figure 1: Hierarchical model

Source: <https://mariadb.com/kb/en/understanding-the-hierarchical-database-model/>

3.3.2 Network model

The network database model is an advancement over the hierarchical model. It was designed to solve the lack of flexibility in that model. Which allows having multiple parents per child instead of single-parent, which enabled mapping many to many relationships.

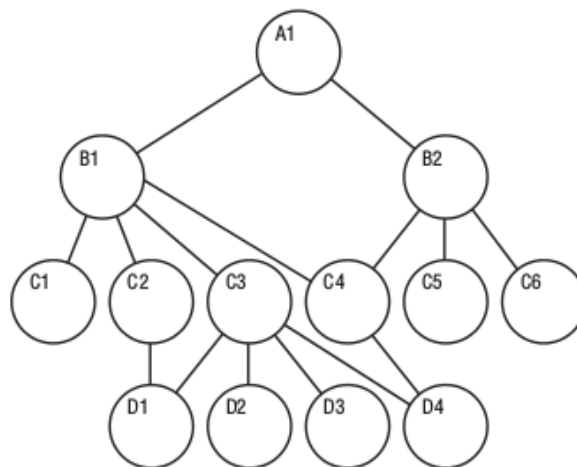


Figure 2: Network model.

Source: <https://mariadb.com/kb/en/understanding-the-network-database-model/>

3.3.3 Relational model

The relational model was a big step forward over the previous two models in terms of ease of use and capabilities. It enables the table to be related to one another by using a common field (primary key- foreign key).

The system was developed in 1970. and still the most used database model today.

It represented the database as a set of relations between tables. Every row in the tables represents a collection of values. That can be retrieved using SQL query that been later developed for the DBMS that supported this model.

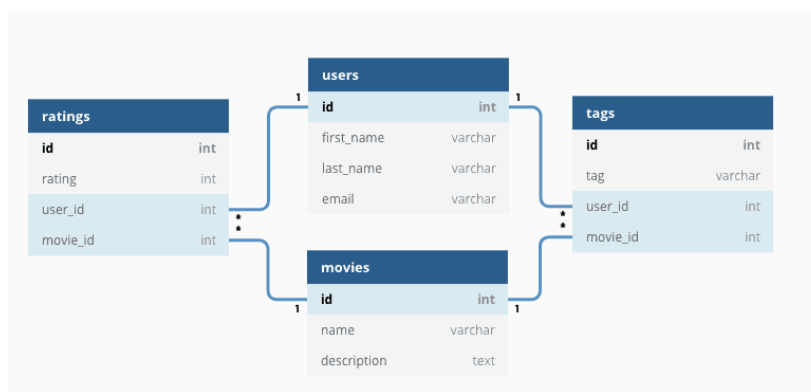


Figure 3: Relational Model

Source: <https://www.omnisci.com/technical-glossary/relational-database>

3.3.4 Object-Oriented model

When object-oriented programming started its rise in popularity, the object-oriented model was developed to overcome the mismatch between the relational model and object-oriented programming languages. The focus was on reducing the complexity mapping from in-memory data structures to relational tables.

The object-Oriented model adds database functionality to persist the OOP objects in the database engine. You could think of it as a persistent virtual memory system, allowing you to program with persistence yet without taking any notice of a database at all.

The issue with this model is that it is coupled the data-layer and application-layer together where it is not possible to access the data without the application-layer itself.

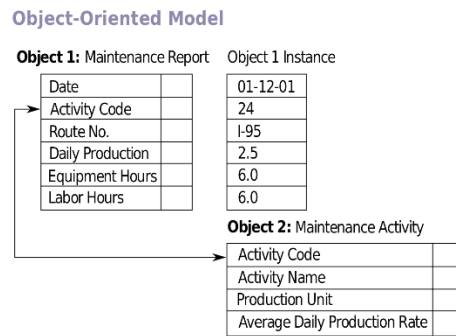


Figure 4: Object-Oriented Model.

Source: https://en.wikipedia.org/wiki/Object_database

4 An Overview of NoSQL

Since the 1970s and the relational database management systems have been the dominant database model and it has eclipsed all previous database models, and still to this day the most popular database model and it is not going away any time soon. RDBMS gain this popularity for many key factors, which has been described well in the book “NoSQL Data Models: Trends and Challenges” (Pivert, 2018). First, the solid theoretical and mathematical foundations of RDMS made it easier to calculate and predict the system’s performance, limits, and efficiency. Second; the use of tables made it easy for everyone to model and describe the data, it is a long-used system that is natural and common sense to everybody. And the third reason and perhaps the most important one, that made the RDBMS very popular is the existence of domain-specific query language (SQL). It was close to the natural language in its syntax and made retrieving data simple in comparison to low-level programming needed in older DBMSs. The last reason is the use of the ACID principles with guaranteed high reliability and consistency of the applications.

By the beginning of the 21st century. Driven by the accessibility of mobile networks and broadband internet; Social networks, search engines, email clients, stock markets, enterprises, and millions of other sources have started generating a massive amount of data every day. This massive expansion has demanded new requirements from the applications and DBMS to meet the new web-scale. Thus, expansion has also brought new concepts and terms, especially “Cloud Computing” and “Big Data”. In a simple term, “Cloud Computing” means, the data is available, distributed, and processed in multiple data centres. The term “Big Data” is described well in a quote from the same book:

“The umbrella term “Big Data” characterizes data sets with the so-called three “V”s: Volume, Variety, and Velocity. More precisely, “Big Data” data sets must be large (at least several terabytes), heterogeneous (containing both structured and unstructured textual data, as well as media files), and produced and processed at high speed.” (Pivert, 2018)

These new requirements of having the system scaled beyond not only vertically but also horizontally and being available even under massive traffic loads and consistent enough to be used in a production environment, have pushed the RDBMS beyond its limits and purpose that it was originally designed for. And NoSQL database systems have emerged to meet these new requirements.

NoSQL is not a database or a database type by itself rather an umbrella term that is well described in the book “Getting Started with NoSQL”:

„NoSQL is a generic term used to refer to any data store that does not follow the traditional RDBMS model—specifically, the data is non-relational and it does not use SQL as the query language. It is used to refer to the databases that attempt to solve the problems of scalability and availability against that of atomicity or consistency.“ (Vaish, 2013).

We will look into the types of NoSQL databases in a later section.

4.1 The relationship between ACID, BASE, and CAP Theorem

The cloud distributed systems are the norm of the web-scale era. Massive applications are thriving to serve billions of connected devices and guaranteeing a great user experience at the same time.

This has defined a new set of capabilities that is expected from any database engine used in distributed systems. In reality, it's not possible to meet all of these expectations. That is what is known as the CAP theorem. Which is described in the book „A Deep Dive into NoSQL Databases The Use Cases and Applications“ as follows:

„The CAP theorem states that it is impossible for any distributed system to simultaneously provide all the three capabilities“ (Pethuru Raj, Ganesh Chandra Deka, 2018)

These capabilities are:

Consistency (a read operation will return the result of the most recent write to any node of the entire system or an error.)

Availability (a guarantee that every request receives a response that is not an error, without the guarantee that it contains the most recent write.)

Partition Tolerance (the system continues to operate despite arbitrary message loss or failure of a part of the system).

The NoSQL database systems favor availability over consistency. And the rationale behind this is that these systems need to be as fast as possible and can't wait for system-wide data synchronization before returning a response to the user. This approach implies that after data updates, the data stores will be inconsistent and out of sync for a short amount of time. But it will be consistent eventually. These systems follow the **BASE** principles as described in the book „Getting started with NoSQL“ :

- **Basic Availability:** Each request is guaranteed a response—successful or failed execution.
- **Soft state:** The state of the system may change over time, at times without any input.
- **Eventual consistency:** The database may be momentarily inconsistent but will be consistent eventually.

While on the other hand, the traditional RDMBS focuses on **ACID** principles that are summarized in the same book as follows:

- **Atomicity:** Everything in a transaction succeeds or is rolled back.
- **Consistency:** A transaction cannot leave the database in an inconsistent state.
- **Isolation:** One transaction cannot interfere with another.
- **Durability:** A completed transaction persists, even after applications restart.

It can be noted here that the **BASE** principles are much looser than **ACID** guarantees, but each has its uses and benefits. In real-world. A website like Facebook can afford to show stale data for some users for some time before the most up to date data is shown while an e-commerce website like amazon cannot show a certain product to be available while the last piece has been sold but the system state is not consistent yet. That might cause actual trouble.

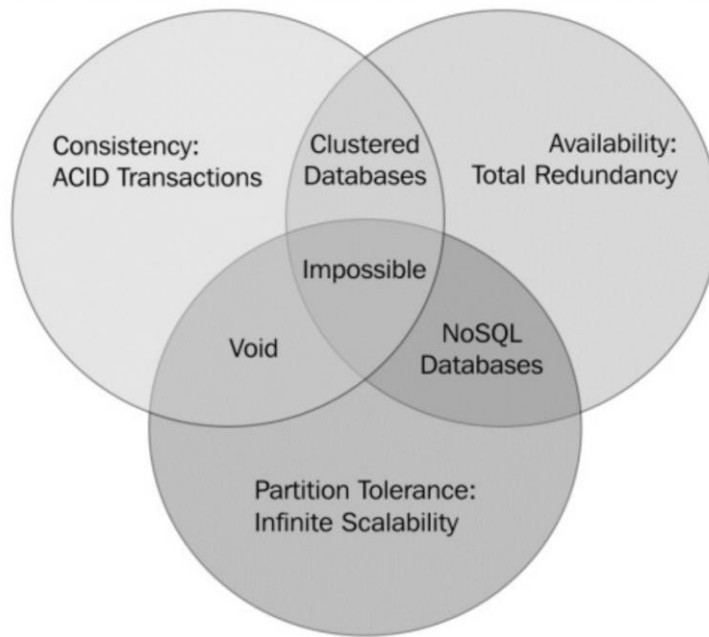


Figure 5: The relationship between ACID, BASE, and CAP Theorem.

Source: "Getting Started with NoSQL" (Vaish, 2013)

4.2 NoSQL Storage Types

This section of this thesis will describe the four main NoSQL database types according to the way they store data by following the excellent description provided by the book „NoSQL and SQL Data Modeling Bringing Together Data, Semantics, and Software (Hills, 2016)“

4.2.1 Key-Value Databases

It manages and organizes a data model that consists of a set of key-value pairs, the key is unique, and the values can be of simple types like numbers and strings or of a complex nature, like lists and hashes.

Having a unique primary-key for each pair enables extremely fast retrieve times which is the main goal of this database types in addition to advanced key-search operations such as searching for exact key values and ranges of key values, searching based on hashes of the key values, searching based on scores associated with keys, etc...

However, the database system will not create the key by itself for the pairs nor it has a built-in way to list all keys, thus, it requires the consumer application to provide the key upon data write and use the key again upon data retrieval.

Common Use Cases: The majority of the key-value database focused on being persistent in-memory storage which enables extremely fast read/write times. Some of the popular

examples are Redis, Memcached, Amazon DynamoDB. And they can be used in many roles such as:

Storing user-session information: where every session has a unique ID that can be used as a key in the DB and this will enable much higher access speeds than storing the session information on Disks or RDBMS.

Caching: as a highly available in-memory cache to decrease data access latency, increase throughput, and ease the load off your relational or NoSQL database and application.

Shopping Cart Data: can be saved where the key is the user ID, and the shopping cart's information will be available across browsers, devices, and sessions.

4.2.2 Document Databases

It enables storing and retrieving of documents where each document represents a record and contains semi-structured data. Which can be XML, JSON, YAML, etc...

Document-orient databases are do not enforce any kind of schema validation on the content of the document. The documents can be different in structure, and it may be deeply nested and contains other collections or even media files. The database system will assign a unique ID for each document. Which can be used for indexing.

Common Use Cases: According to the book “NoSQL Distilled A Brief Guide to the Emerging World of Polyglot Persistence”, the most common use cases are:

- **Event Logging and Web Analytics:** Document databases can be used to store information about event logging, error tracking, and usage analytics. Given, all the data being captured are nested objects.
- **Content Management Systems:** The flexible document structure is perfect for content management systems to store articles, tags, and manage comments without the complexity that RDBMS might bring.
- **E-Commerce Applications:** The Document Database systems are great to store different product information and descriptions, and helps reduce migrations issues as the applications grow.

4.2.3 Graph Databases

The graph database model is built around the idea of connections. And it's better described in the book “NoSQL Distilled A Brief Guide to the Emerging World of Polyglot Persistence”:

“Graph databases allow you to store entities and relationships between these entities. Entities are also known as nodes, which have properties. Think of a node as an instance of an object in the application. Relations are known as edges that can have properties. Edges have directional significance; nodes are organized by relationships which allow you to find interesting patterns between the nodes. The organization of the graph lets the data to be stored once and then interpreted in different ways based on relationships.” (Prمود Sadalage, Martin Fowler, 2012)

Common Use Cases:

- Social Networks and Connected Data: it can be used to represent the social graph and the connections between friends or the employment history of employees in different organizations
- Location-Based Services: where each location can be a node with attributes and connections between locations

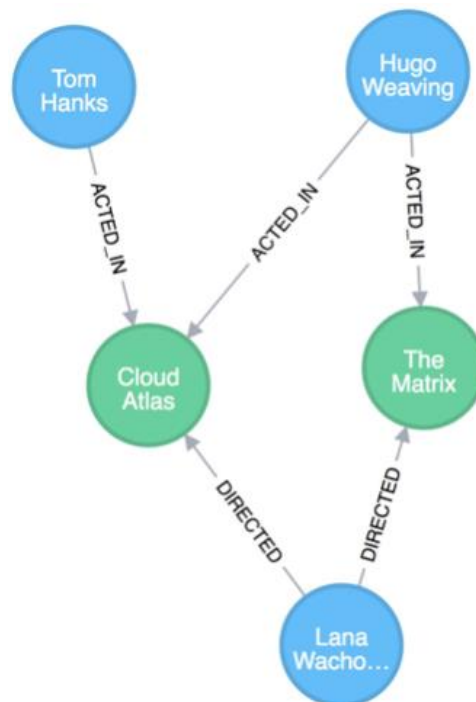


Figure 6: Graph Model example used in Neo4j Graph Database.

Source: <https://neo4j.com/developer/guide-data-modeling/>

4.2.4 Column Databases

Traditional database systems store the data in tabular form where the columns are known and new records are inserts as new rows. In the column-based databases, the new data is stored as new columns. This way the engine can more easily access the data it needs instead of searching and discarding unwanted data in rows.

The advantage of this approach is that it allows adding new columns in the future more easily without worrying about filling the default values. Which enables easier growth for the applications and more flexibility in fixing any missing data field.

Common Use Cases:

- Event Logging: This flexibility of creating dynamic columns is very useful in tracking events or creating reports where each new field in the report can be an actual new column in the Column-Based Database.
- Content Management Systems: It enables storing the entire post information, categories, tags, etc.. as new columns. And the post comments and content can be new column-families.

5 Advantages and Drawbacks

In this chapter of the thesis, I will focus on comparing SQL database engines and NoSQL databases and how they perform in different real-world scenarios.

5.1 Scalability

Rarely any new application takes into account how it will scale to serve millions of users from the beginning of the project. Many projects surges in popularity and the problem of scalability arises.

The basic solution is to scale the server hardware performance or aka “Vertical Scaling” by replacing the server with a faster one. This solution works well for both SQL and NoSQL database engines. But the traffic might reach a certain point where no single server can handle all the load. Then multiple connected physical or virtual servers are needed to handle the load, aka” Horizontal Scaling”. In this case, SQL and NoSQL handle this situation differently.

As shown in the previous chapter, SQL database engines follow ACID principles and use transactions to write data to the database, which makes it harder to synchronize the data across multiple servers while guaranteeing the ACID principles, the consistency in particular.

One solution has been followed to overcome this limitation, it is called “SQL Clusters” as described in this great article “Always On Failover Cluster Instances - SQL Server” (Ray, 2017). is to connect multiple physical servers via a network, each server hosts an instance of SQL server and all have the same access to shared storage. Even though this solution increases the availability it does not provide enough performance benefits and has limitations to how much it can be scaled.

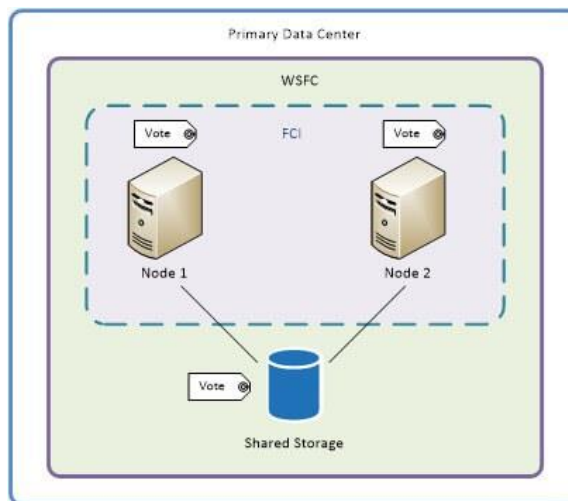


Figure 7: Windows server approach for SQL clustering.

Source: <https://www.mssqltips.com/sqlservertip/4717/what-is-sql-server-always-on/>

While NoSQL on the other hand is built to scale out since consistency is not a guarantee in the first place. As described in the book “A Deep Dive into NoSQL Databases The Use Cases and Applications” (Pethuru Raj, Ganesh Chandra Deka, 2018). The NoSQL characteristics that enable this ability are:

- Auto-Sharding: is a method for distributing data across multiple machines. Where servers can be added or removed without a downtime
- Distributed Queries: The database engine will take care of aggregating the result to a request from multiple nodes.
- Integrated Caching: NoSQL database engines automatically caches the results into the server memory to reduce the latency and increase the throughput without the need to explicitly adding this functionality as in SQL engines.

5.2 Eventual consistency vs Strong consistency

In the previous section, I have described the scaling approaches in SQL vs NoSQL databases, and how server nodes can be clustered or scaled horizontally. Another aspect to having multiple connected nodes serving the same data instead of a single node is the delay or the time needed for one node to receive a write query and replicate across the remaining nodes within the same data centre or across multiple data centres. This may lead to a write operation on one of the nodes, at the same time, a read operation is going on another node while the data is still being replicated. This may lead the system to behave in one of two main ways, first, is to block the read query until all replications are finished, which is known as “Strong Consistency” and it’s very common in ACID based database systems where consistency is one of the guarantees. Another way is to respond to the read query as soon as possible by returning stale data to the user until the newly written value is replicated across all the nodes, and eventually returning the new value, which is known as “Eventual Consistency”.

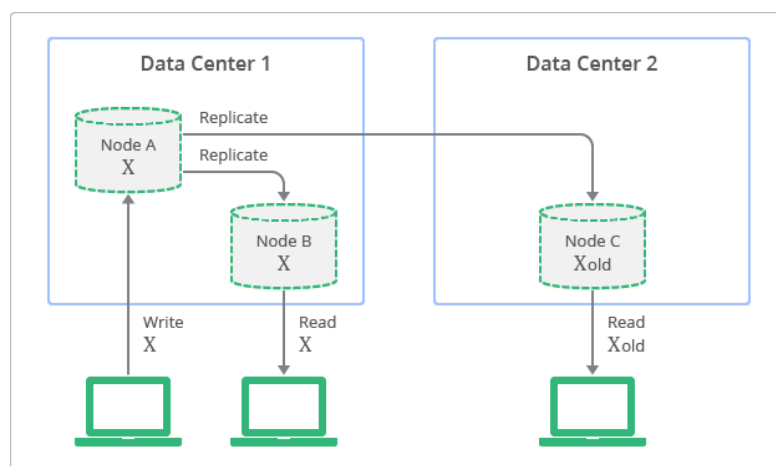


Figure 8: Conceptual Depiction of Replication with Eventual Consistency.

Source: <https://bit.ly/3opfo8i>

Based on a great article written by Amazon’s CTO “Werner Vogels” titled “Eventually Consistent – Revisited” (Vogels, 2008), each approach has its advantages and drawbacks and comes down to the requirements of the application. In a banking environment, it’s important that all transactions between nodes are consistent and no stale data is ever returned. That may come at the expense of speed or availability. But that is an acceptable price to be paid in exchange for accurate data every for every single request. While in a project similar to Facebook, the user will tolerate seeing relatively old feed until all the nodes have been updated. Since waiting for all the writes to finish writing and to for data to be synchronized

across nodes is not a realistic option. And the fast response time to serve billion of requests and high availability are the main goals of the system.

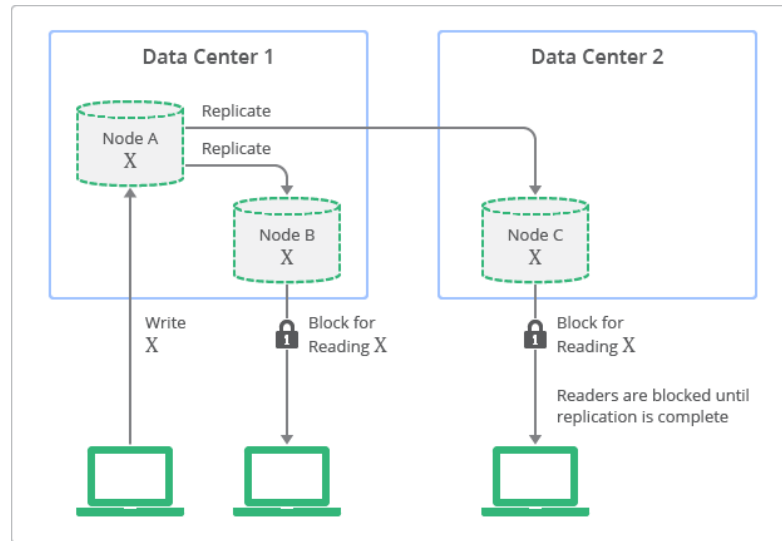


Figure 9: Conceptual Depiction of Replication with Strong Consistency.

Source: <https://bit.ly/3opfo8i>

However, in the same article, (Vogels, 2008), the author describes how many modern relational database management systems have implemented ways to improve the system behavior and responsiveness while the data is being replicated across the nodes, by supporting synchronous and asynchronous replication modes. In synchronous mode, the replication updates are part of the transactions. Where the system waits for the data to be replicated before finishing the transaction, blocking all the read responses in the meantime. While in the asynchronous mode the updates replicated to the rest of the nodes in a delayed manner. If the first node has failed, due to a network issue for example. The remaining nodes will return stale and inconsistent data. Which is an implementation of eventual consistency, which increases the ability to scale the read performance of the RDBMS.

5.3 Multi-Tenant Support

The rise of cloud computing has changed the way applications are built and distributed. From mostly desktop applications in the past era to mostly cloud-based applications distributed to the clients as a software as a service (SaaS) model.

There are two main approaches for building software as a service application, as explained in the article “SaaS: Single Tenant vs Multi-Tenant - What's the Difference?” (Brook, 2020). The first approach is “single-tenant”, a single instance of the application and its database

will serve a single customer, while on the other hand “multi-tenant” approach or “multi-tenancy” means:

“a single instance of the software and its supporting infrastructure serves multiple customers. Each customer shares the software application and shares a single database. Each tenant’s data is isolated and remains invisible to other tenants. (Brook, 2020)

Which reduces the operational costs of the application and simplify maintenance, even though it may increase the complexity of the up and the pushing updates and raises concerns in terms of data security and confidentiality. Therefore, it is essential to choose a suitable database management system powering the application.

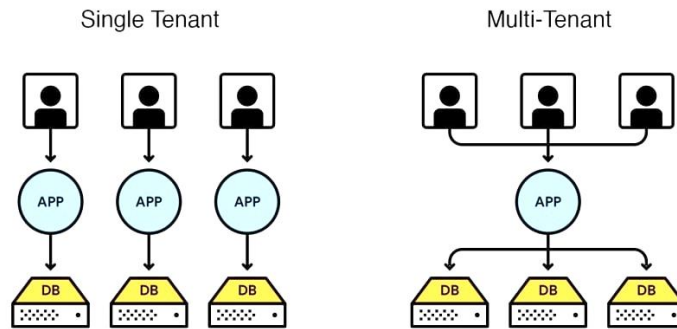


Figure 10: Single Tenant vs Multi-Tenant Architecture.

Source: <https://dev.to/sciencebae/multi-tenant-architecture-and-it-s-issues-h06>

NoSQL brings lots of benefits for multi-tenant applications, the schema flexibility enables easier application schema updates and more flexibility working with different data encryptions policies. It also provides easier scaling to handle any increase or peak usage caused by serving multiple enterprises or client by the same system. NoSQL databases security capabilities allow the isolation between tenants by creating collections isolated from each other, and even allocation more system resources to serve such collections faster. This data security approach is explained well in the article “Scalable Database Solutions for Multi-Tenant Applications”:

“Ensuring security at the application level by adding a tenant property inside the document for tenant identification, using filters to retrieve data belonging to a particular sub-set of tenants and utilizing authorization keys to isolate tenant data and restrict access” (labs, 2015)

5.4 Normalization vs Denormalization

Data normalization is one of the commonly used approaches in database design and data storage in SQL database management systems. The benefit of following the normal forms is very well highlighted in the book “Joe Celko's SQL for Smarties: Advanced SQL Programming” (Celko, 2010):

“Normal forms are an attempt to make sure that you do not destroy true data or create false data in your database. One of the ways of avoiding errors is to represent a fact only once in the database, since if a fact appears more than once, one of the instances of it is likely to be in error” (Celko, 2010)

In practice, Normalization is an incremental process of removing repetition and redundancy from the data by following the “normal forms” which will lead to and ensure each table has unique values assigned a primary key and each cell contains a single value, and any repeated related values are moved into a separate related table connected to the original table with a foreign key. This process helps to ensure data integrity and eliminating anomalies in data insert, updates, and delete by having one copy of each information. It also reduces the physical storage needed.

For example, imagine a large table contains users’ countries. Instead of having two columns contains the name and the country which may lead to thousands of instances of repeated values taking space and risking the update to being out of sync. Transforming the table into the normal form will result in two tables, the first contains the names and the second will contains a unique set of country names. This approach will use less storage and describes the relationship better and enables the reuse of the countries table in another place of the application but on the other hand, the queries containing joins are slower and requires more resources.

Most modern applications fancy speed above all, especially above storage space which in most cases much affordable and scalable than compute time and processing power, that is why an alternatives approach normalization called “Denormalization” is being followed, even though some RDBMS based applications follow it, but it is far more common in NoSQL database management systems.

As described by the official documentation of one of the most popular NoSQL databases “MongoDB”:

“Denormalization allows you to avoid some application-level joins, at the expense of having more complex and expensive updates. Deformalizing one or more fields makes sense if those fields are read much more often than they are updated.” (Zola, 2017)

In practice, denormalization is embedding the related data into the same table or document which enables faster access to this data, but the system needs to ensure updating every occurrence of this piece of information’s which can be a complex task. For example. Imagine a document in the NoSQL database containing information about a blog post, like title, content, category, and author name. single read of this document will return the entire post information, no joints or additional queries are needed. This is an optimal scenario for performance demanding system. But changing the author name or category name to something else is far more complicated than updating a single cell in a single table in normalized form but it is an update on every single document containing this category.

6 Practical Part

6.1 Description of the project

In this chapter, I will demonstrate the differences and advantages, and drawbacks of using NoSQL database systems through an application that matches a real-world scenario and highlights the benefits of using Document Database and how it can ease and speed up the development in certain project requirements.

The project is a resemblance of an online election system where the candidates' information is displayed to help the voters learn more about them, their plans, and their views. The user can filter the candidates by the area they belong to and then they can view the complete information of the candidate like their name, political party, photo, brief about them, etc... The application will not contain an actual voting mechanism as it is beyond the scope of demonstrating the Document Databases.

6.2 Technology stack choices

The technologies and programming languages used to build this application are as follows:

PHP version 7.4: It is one of the most popular programming languages in the world. Due to its simplicity and short learning curve and it is used in many popular open-source projects. This application will use version 7.4 of the language which has strict-types support unlike older versions of the language.

MySQL: It is considered one of the most popular RDBMS in the world, according to their official website (MySQL, 2021). It is open-source and free to use and has huge community resources to help developers get started and resolve any issue they might face. It is also extremely fast and highly optimized for high reads applications. Which makes it a perfect choice for our application.

Apache CouchDB: It is an open-source NoSQL database management system that according to the official documentation (CouchDB, 2021) stores the data as documents in JSON format. And provides RESTful HTTP API for updating database documents.

The use of documents allows us to store polymorphic and semi-structured data, which is the case that we are dealing with in this application. The HTTP interface makes it versatile and easy to connect to from any programming language or even from web clients directly.

Yii PHP Framework version 2: It is a fast PHP framework built using Model-View-Controller (MVC) architecture which enables us to separate data models from business logic and UI elements. It also has great support for many popular SQL and NoSQL database systems. The use of a framework will speed up the development and increase clarity and ease of maintenance.

6.3 Database Design

6.3.1 The logical database designs

It is a detailed version of the conceptual database design. The Logical ERD design for our application would look like this:

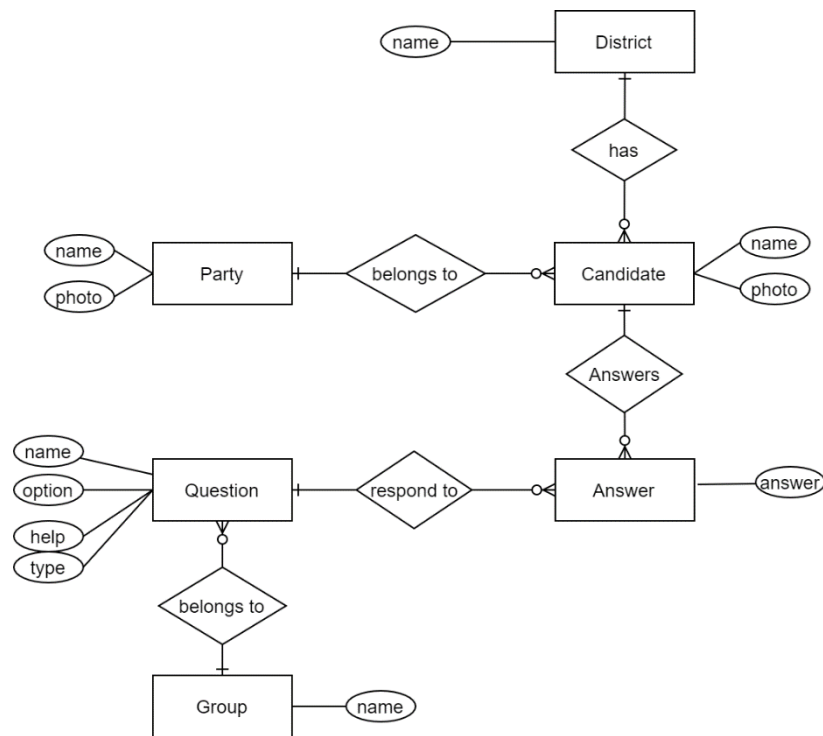


Figure 11: The logical database designs.
Source: Author

6.3.2 The physical database designs

In this stage of the application database design, we will have different designs based on the type of the database model. For the relational model, I will create an ERD based on the physical design of the tables and the actual data types. And it would look like this:

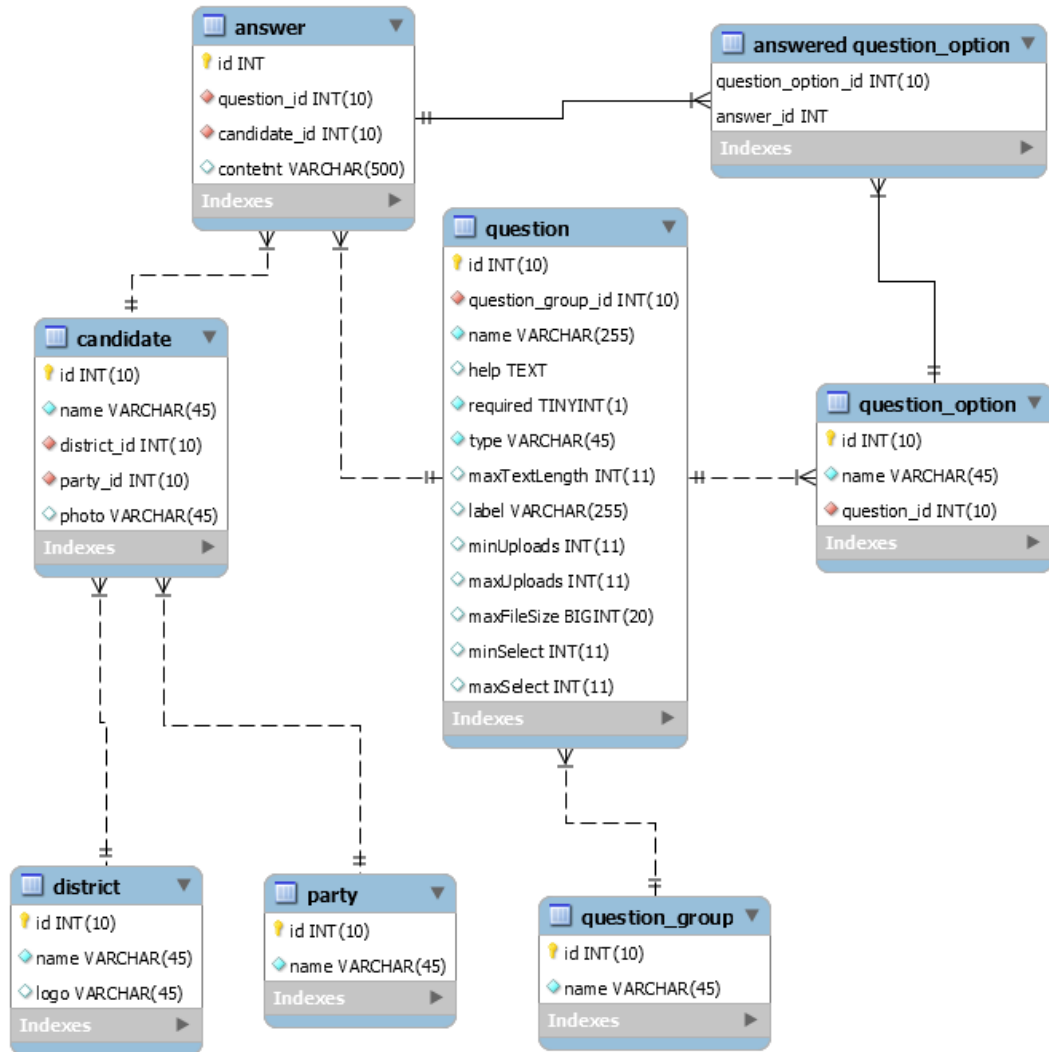


Figure 12: The physical database designs
Source: Author

While for the document database I will design a JSON document to represent the document schema. There are two main approaches in transforming ERD relationships into a document.

- Embed: embedding the relationship inside the document itself. For Example, the district can be a nested attribute inside the candidate document itself.
- Referencing: is by creating another document for the related entity and reference it in the current document using a unique ID.

In this application we have the districts and party names as a static list that is not going to change which enable us to embed the district and party information inside the document design for the candidate, this approach simplifies data retrievals and search. We can also embed the candidate document, the list of answers by indexing it by the question id. The questions groups should be in a separate document design.

The documents design for the candidate would look like this:

```
{
  "_id": "cidde89e1d8251a59763e5cde1025762",
  "_rev": "4-13c8fc2275c2b31263462d810f1dfc3e",
  "name": "John Smith",
  "photo": null,
  "docType": "candidate",
  "district": {
    "_id": null,
    "name": "Praha 1",
    "logo": null,
    "docType": "district"
  },
  "party": {
    "_id": null,
    "name": "Political Party Name 1",
    "docType": "party"
  },
  "answers": {
    "questionId": {
      "docType": "answer",
      "content": "Improve medical services"
    }
  }
}
```

Figure 13: CouchDB candidate document.
Source: Author

The design for the question group provides great flexibility for storing question details. And it would look like this:

```
{
  "_id": "cidde89e1d8251a59763e5cde1031f70",
  "_rev": "3-5bf9ec2720831dd98f89d1c3b570d7a3",
  "docType": "questionGroup",
  "questions": [
    {
      "docType": "questionGroup",
      "name": "What is your plan for improving internet speed?",
      "help": "Please describe your plans for improving the internet infrastructure",
      "type": "text",
      "required": true,
      "maxLength": 200
    },
    {
      "docType": "questionGroup",
      "name": "Are you satisfied with internet medical services",
      "type": "radio",
      "required": true,
      "options": [
        {
          "id": 1,
          "text": "Yes"
        },
        {
          "id": 2,
          "text": "No"
        }
      ]
    }
  ]
},
  "name": "Infrastructure Improvement Plans"
}
```

Figure 14: CouchDB questions list document.
Source: Author

6.4 Database implementation

6.4.1 Creating the relational database

The next step in the project implementation is to transform the physical database design into SQL create statement and run it inside the MySQL server. There are many MySQL clients to help us execute SQL statements. I choose a powerful GUI tool called “HeidiSQL”.

I will start by creating the database, the create statements would look like this:

```
CREATE SCHEMA IF NOT EXISTS `thesis` DEFAULT CHARACTER SET latin1;
```

*Figure 15: Create Database SQL Statement
Source: Author.*

The next step would be creating the tables, the primary keys are auto-increment so that new id number will be automatically inserted even if it is not provided inside the SQL statement. And the foreign key constraints will reflect the changes on the related table whenever a row has been removed or the id has been changed. This improves the data integrity in the application.

```
-----  
-- Table `district`  
-----  
CREATE TABLE IF NOT EXISTS `district` (  
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,  
  `name` VARCHAR(45) NOT NULL,  
  `logo` VARCHAR(45) NULL DEFAULT NULL,  
  PRIMARY KEY (`id`))  
ENGINE = InnoDB  
AUTO_INCREMENT = 2  
DEFAULT CHARACTER SET = latin1;  
-----  
-- Table `party`  
-----  
CREATE TABLE IF NOT EXISTS `party` (  
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,  
  `name` VARCHAR(45) NOT NULL,  
  PRIMARY KEY (`id`))  
ENGINE = InnoDB  
AUTO_INCREMENT = 2  
DEFAULT CHARACTER SET = latin1;  
-----  
-- Table `candidate`  
-----  
CREATE TABLE IF NOT EXISTS `candidate` (  
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,  
  `name` VARCHAR(45) NOT NULL,  
  `district_id` INT(10) UNSIGNED NOT NULL,  
  `party_id` INT(10) UNSIGNED NOT NULL,  
  `photo` VARCHAR(45) NULL DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  INDEX `fk_candidate_district_idx` (`district_id` ASC) VISIBLE,  
  INDEX `fk_candidate_party1_idx` (`party_id` ASC) VISIBLE,  
  CONSTRAINT `fk_candidate_district`  
    FOREIGN KEY (`district_id`)  
    REFERENCES `district` (`id`)  
    ON DELETE CASCADE  
    ON UPDATE CASCADE,  
  CONSTRAINT `fk_candidate_party1`  
    FOREIGN KEY (`party_id`)  
    REFERENCES `party` (`id`)  
    ON DELETE CASCADE ON UPDATE CASCADE)  
ENGINE = InnoDB  
AUTO_INCREMENT = 3
```

```

DEFAULT CHARACTER SET = latin1;
-----
-- Table `question_group`
-----
CREATE TABLE IF NOT EXISTS `question_group` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`id`))
ENGINE = InnoDB
AUTO_INCREMENT = 2
DEFAULT CHARACTER SET = latin1;
-----
-- Table `question`
-----
CREATE TABLE IF NOT EXISTS `question` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `question_group_id` INT(10) UNSIGNED NOT NULL,
  `name` VARCHAR(255) NOT NULL,
  `help` TEXT NULL DEFAULT NULL,
  `required` TINYINT(1) UNSIGNED NOT NULL DEFAULT '0',
  `type` VARCHAR(45) NOT NULL,
  `maxTextLength` INT(11) NULL DEFAULT NULL,
  `label` VARCHAR(255) NULL DEFAULT NULL,
  `minUploads` INT(11) NULL DEFAULT NULL,
  `maxUploads` INT(11) NULL DEFAULT NULL,
  `maxFileSize` BIGINT(20) NULL DEFAULT NULL,
  `minSelect` INT(11) NULL DEFAULT NULL,
  `maxSelect` INT(11) NULL DEFAULT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_question_question_group1_idx` (`question_group_id` ASC) VISIBLE,
  CONSTRAINT `fk_question_question_group1`
    FOREIGN KEY (`question_group_id`)
      REFERENCES `question_group` (`id`)
      ON DELETE CASCADE
      ON UPDATE CASCADE)
ENGINE = InnoDB
AUTO_INCREMENT = 5
DEFAULT CHARACTER SET = latin1;
-----
-- Table `question_option`
-----
CREATE TABLE IF NOT EXISTS `question_option` (
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NOT NULL,
  `question_id` INT(10) UNSIGNED NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_question_option_question1_idx` (`question_id` ASC) VISIBLE,
  CONSTRAINT `fk_question_option_question1`
    FOREIGN KEY (`question_id`)
      REFERENCES `question` (`id`)
      ON DELETE CASCADE
      ON UPDATE CASCADE)
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;
-----
-- Table `answer`
-----
CREATE TABLE IF NOT EXISTS `answer` (
  `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `question_id` INT(10) UNSIGNED NOT NULL,
  `candidate_id` INT(10) UNSIGNED NOT NULL,
  `contetnt` VARCHAR(500) NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_answer_question1_idx` (`question_id` ASC) VISIBLE,
  INDEX `fk_answer_candidatel_idx` (`candidate_id` ASC) VISIBLE,
  CONSTRAINT `fk_answer_question1`
    FOREIGN KEY (`question_id`)
      REFERENCES `question` (`id`)
      ON DELETE CASCADE
      ON UPDATE CASCADE,
  CONSTRAINT `fk_answer_candidatel`
    FOREIGN KEY (`candidate_id`)
      REFERENCES `candidate` (`id`)
      ON DELETE CASCADE
      ON UPDATE CASCADE)
ENGINE = InnoDB;
-----
-- Table `answered question_option`
-----
CREATE TABLE IF NOT EXISTS `answered question_option` (
  `question_option_id` INT(10) UNSIGNED NOT NULL,
  `answer_id` INT UNSIGNED NOT NULL,
  PRIMARY KEY (`question_option_id`, `answer_id`),
  INDEX `fk_question_option_has_answer_answer1_idx` (`answer_id` ASC) VISIBLE,
  INDEX `fk_question_option_has_answer_question_option1_idx` (`question_option_id` ASC) VISIBLE,
  CONSTRAINT `fk_question_option_has_answer_question_option1`

```

```

FOREIGN KEY (`question_option_id`)
REFERENCES `question_option` (`id`)
ON DELETE CASCADE
ON UPDATE CASCADE,
CONSTRAINT `fk_question_option_has_answer_answer1`
FOREIGN KEY (`answer_id`)
REFERENCES `answer` (`id`)
ON DELETE CASCADE
ON UPDATE CASCADE)
ENGINE = InnoDB
DEFAULT CHARACTER SET = latin1;

```

Figure 16: Tables create SQL statement.
Source: Author

6.4.1.1 Creating the NoSQL documents

The CouchDB uses a RESTful HTTP API which allows us to connect to the database using a wide variety of tools. For this application, I will use the official web-based management tool for CouchDB. It is called “Fauxton Visual Guide”.

I will start by creating a new database:



Figure 17: Database creation page in Fauxton GUI tool.
Source: Author

We can alternatively use “cURL” to call the rest API directly:

```

C:\Users\Mohamad Othman>curl -X PUT http://administrator:zJcbxh9A@localhost:5984/thesis-db1
{"ok":true}

```

Figure 18: Creating CouchDB database using cURL tool.
Source: Author

The next step is to create the “physical database design” section of this thesis using the “Fauxton” tool, the ID will be provided automatically by the database, and the layout looks this:

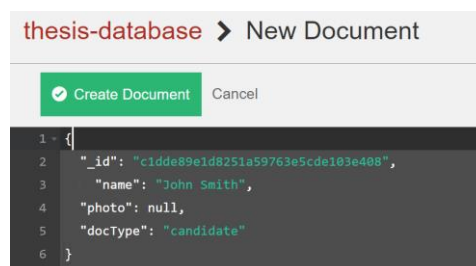


Figure 19: Inserting new document in CouchDB.
Source: Author

6.4.2 Application implementation and layout

The application will be programmed in PHP and using Yii Framework version 2 which enables us to map the database entities to Models in MVC architecture.

Since we are trying to achieve the same logical database design in both database types. The model should be common for both database types. And it would look like this for the Candidate entity:

```
class Candidate extends Model
{
    public string $id;
    public string $name;
    public string $photo;

    public District $district;

    public Party $party;
}
```

Figure 20: Candidate Class Model in PHP & Yii Framework.
Source: Author

And the UI element that will utilize this model is designed like this:

<p>Candidates List and their Answers (Count: 1)</p> <p>John Smith</p> <p>Party: Political Party Name 1 District: Praha 1</p>
--

Figure 21: The page in the application that displays Candidates information's.

Source: Author

6.5 Inserting Data:

The form where the data can be inserted follows this design and uses form validation based on the model attributes:

Figure 22: Candidate creation form in the application layout.

Source: Author

However, each database types uses different syntax and approach to insert data into the database:

The SQL approach is to use INSERT statements for new Party, District and Candidate. trying to insert a new candidate using a non-existing party or district IDs will throw an error because SQL preserves the data integrity using the integrity constraints.

```
-- Insert new Party
INSERT INTO `party` (`id`, `name`)
  VALUES ('2', 'Political Party Name 1');
-- Insert new District
INSERT INTO `district` (`id`, `name`)
  VALUES ('2', 'Praha 1');
-- Insert new Candidate using the previously inserted Party and District.
INSERT INTO `thesis`.`candidate` (`id`, `name`, `district_id`, `party_id`)
  VALUES ('3', 'John Smith', 2, 2);
```

Figure 23: SQL insert statements.

Source: Author

While on the other hand, inserting the data into CouchDB is simpler because we do not need to specify the target fields or table names, only the target database. And this process has been covered in the previous section “Creating the NoSQL documents”.

6.6 Data Retrieval

6.6.1 SQL SELECT statement

The main way of data retrieval in SQL is the select statement, the basic syntax is like this:


```

1 SELECT `id`, `name`, `district_id`, `party_id`, `photo`
2 FROM `candidate`

```

id	name	district_id	party_id	photo
1	Candidate 1	1	1	(NULL)
2	Candidate 2	1	1	(NULL)
3	John Smith	2	2	(NULL)

Figure 24: Sample SQL Query for data retrieval.

Source: Author

However, this query does not provide the entire information for the related tables. We can execute separate queries to fetch the related data but this does not comply with the SQL way and has many drawbacks. To load the candidate information and all the related entities' information, we need to use Joins or subqueries. However, joins are simpler to write and maintain. Such query would look like this:

```

1 SELECT c.`id`, c.`name`, c.`photo`,
2       c.`district_id`, d.`name` AS `district_name`,
3       c.`party_id`, p.`name` AS `party_name`
4 FROM `candidate` AS `c`
5 JOIN `district` AS `d` ON `c`.`district_id` = `d`.`id`
6 JOIN `party` AS `p` ON `c`.`party_id` = `p`.`id`

```

id	name	photo	district_id	district_name	party_id	party_name
1	Candidate 1	(NULL)	1	District 1	1	Party 1
2	Candidate 2	(NULL)	1	District 1	1	Party 1
3	John Smith	(NULL)	2	Praha 1	2	Political Party Name 1

Figure 25: Sample SQL Query with Joins for data retrieval.

Source: Author

In this query, I used the default join type in MySQL, which is “Inner Join” which only returns a result if both tables have the common key. This approach will not return any candidate that does not belong to a party or district.

The select statement is very flexible and powerful in helping us select exactly the data we want. For example, we can get the number of candidates per party using the following statement that utilizes the data aggregation function “COUNT”

```

1 SELECT p.name as PartyName , COUNT(c.`id`) AS candidatesCount
2 FROM `candidate` AS `c`
3 JOIN party AS `p` ON `c`.party_id = `p`.id
4 GROUP BY p.id

```

PartyName	candidatesCount
Party 1	2
Political Party Name 1	1

Figure 26: Sample SQL Query with COUNT function for data aggregation.

Source: Author

6.6.2 CouchDB Views

The main difference between MySQL and CouchDB is that the latter does not use SQL for data manipulation and retrieval. CouchDB uses a special kind of documents called “views” as a method of aggregating and reporting the documents’ data. It uses “map” and “reduce” functionality and JavaScript language as a syntax. Each view is indexed after creation to increase the performance.

In order to return the candidate data along with the party and district information we need to write a view that loops over all the documents with the type “candidate” and check if this document has both party and district attributes have been set:

Edit View

Design Document ?

Index name ?

Map function ?

```

1 function (doc) {
2   if(doc.name && doc.party.name && doc.district.name) {
3     emit(doc._id, doc);
4   }
5 }

```

Reduce (optional) ?

Save Document and then Build Index

Figure 27: View creation in CouchDB.

Source: Author

The view has the main map function that emits part of or the entire document when a condition is met. We also provide a name and index name for the view to help us call it or edit it.

It is also possible to aggregate the data similar to what GROUP BY does in SQL SELECT statements. For example, a query that returns the number of candidates that has both party name and district name set would look like this:

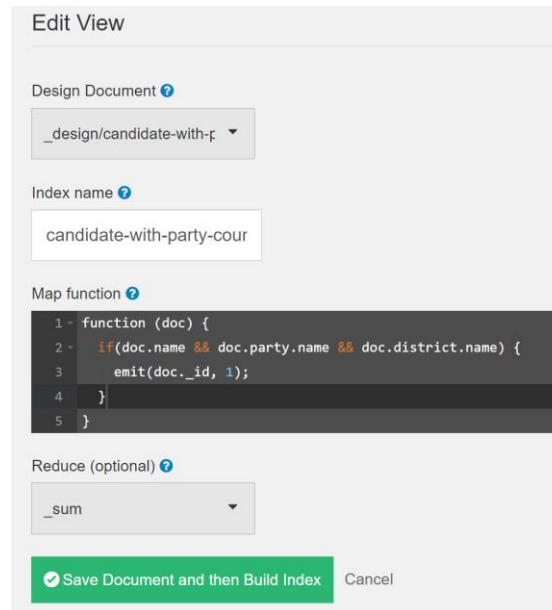


Figure 28: View with reduce function creation in CouchDB.

Source: Author

Notice the use of “_sum” reduce function that accumulates the emitted value “1” from the map function. Other built-in “reduce” functions are “count, stats, approximate count distinct” or you can even write custom aggregation functionality using JavaScript. Which is very powerful.

Upon calling the last view, using any web browser, or using “cURL” tool we get the following output:

```
C:\Users\Mohamad Othman>curl "http://administrator:zJcbxh9A@localhost:5984/thesis/_design/candidate-with-party-count/_view/candidate-with-party-count?include_docs=false&%20skip=0&limit=21&reduce=true"
{"rows": [
{"key": null, "value": 2}
]}
```

Figure 29: Calling CouchDB view using windows 10 terminal and cURL tool.

Source: Author

The value “2” in the output is representative of the current count of candidates matching the provided conditions.

6.7 Performance Comparison

There are many theoretical models for calculation query complexity in SQL that reflects directly on the execution time. In the book “Database Systems - A Practical Approach to Design, Implementation, and Management, 4th Edition” (Connolly, Begg, 2005) the author explains how to calculate the query complexity using “relational algebra and relational calculus” conceptual query languages which tries to take all the query parts and come up with an equation representing the query complexity in relation to the data size in the targeted tables. This approach takes into account the statements used, the number of tables, the number, and type of joins and the grouping and ordering operations, etc...

We can simply use the “Big O Notation” to represent the query time complexity formula. In this chapter, I will insert a large dataset into the application and evaluate **a single count query performance with no concurrency** in both database systems and compare the results to each other’s and to the mathematical model representing the query time complexity.

6.7.1 Testing Setup

I will be using the latest versions of both database engines (MySQL v8.0.23 and CouchDB v3.1.1) installed on a high-end server to eliminate any hardware bottlenecks. The server specifications are as follows:

- Processor: AMD Ryzen 2700x (8 cores)
- Memory: 32GB DDR4
- Storage: NVME SSD
- Operation System: Windows 10 Professional.

To benchmark the queries, I will be using the “SHOW PROFILE;” command in SQL which returns detailed information about the query execution duration and details.

Unfortunately, no similar built-in tool is available in CouchDB, so I will be using the “cURL” total connection time as an indicator of the execution duration. Unfortunately, this number is not a real representative of the time the DB takes to execute the view but the entire time from connecting to the server and parsing the request and returning and displaying the response, which also includes authentication and the network latency.

However, we will be focusing on the value changes between the runs instead of the absolute value itself.

6.7.2 Test Scenario

I will be using a script written in PHP to generate random data and insert it into each database engine. The test will be done in 3 stages and each stage and different amount of data. In the first one, the MySQL database will contain 3 rows in candidates tables and 3 rows in the party table. While the CouchDB we will have 3 documents contains the candidate information and the party details.

In the second stage, MySQL will contain 10,000 records in both candidate and party tables. CouchDB will contain 10,000 as well. In the latest stage, we will have 1 million records on the candidate table and 10,000 on the party table. As of the CouchDB that will contain 1 million documents containing the candidate and party information.

I will run a query that will count all the records/documents of candidates that belong to a party. In SQL, the query will look like this:

```
SELECT COUNT(c.id) AS candidatesCount
FROM candidate AS c
JOIN party AS p ON p.id = c.party_id
```

Figure 30: SQL Query for counting candidates.

Source: Author

And the CouchDB view will look like this:

```
function (doc) {
  if(doc.name && doc.party.name) {
    emit(doc._id, 1);
  }
}
```

Figure 31: CouchDB map function for counting candidates.

Source: Author

6.7.3 Benchmark Results

Each one of the queries has been executed 10 times in each scenario and the result is averaged out and look like this:

	First Phase	Second Phase	Third Phase
Rows/Documents Count	3	10,000	1,000,000
MySQL Average Time (ms)	225.1	228.6	229.5
CouchDB Average Time (ms)	0.3085	3.7988	486.7575

Table 1: Average count query benchmark results after 10 runs.

Source: Author.

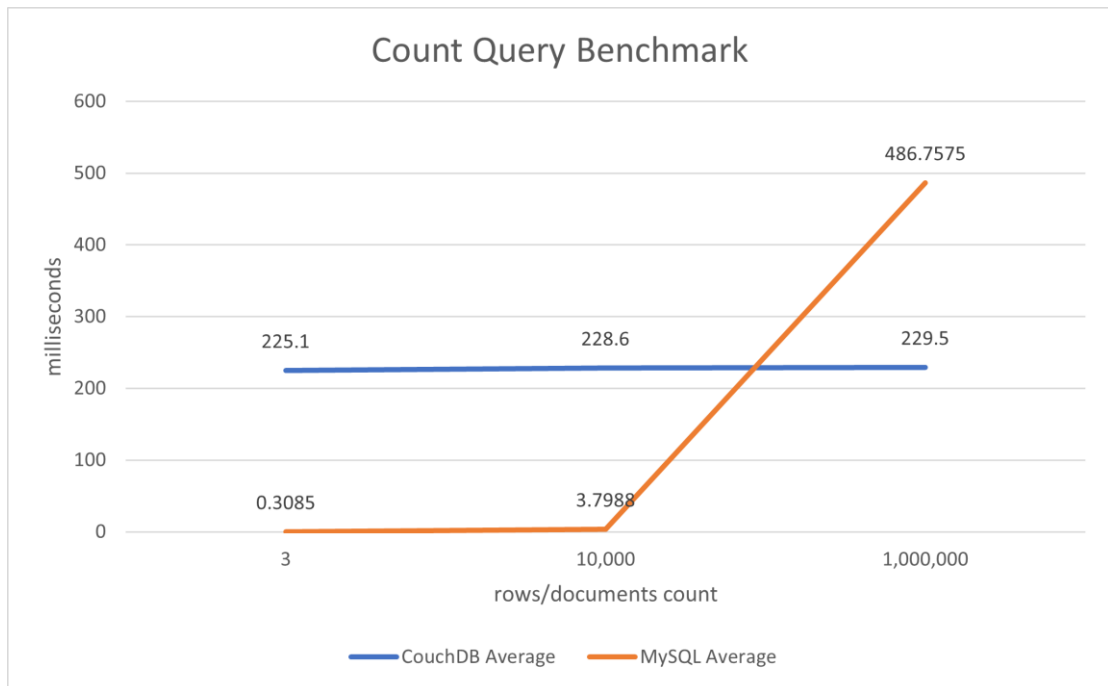


Figure 32: MySQL and CouchDB benchmark and average query execution time.

Source: Author

The CouchDB has sustained near identical performance between the first and small dataset and the large dataset, and that is due to 2 main factors

- The View does not contain referencing other documents, but rather, the related data is embedded inside the document itself.
- The Indexing that is applied on the view after every data update in CouchDB increase the response time significantly.

The SQL query that includes 1 inner join has shown a dramatic increase in execution time the more data we have available in the tables. And this time increase can be represented using Big O Notation as follows:

$$\text{Query Time Complexity} = O(N + M)$$

Equation 1: Equation for calculation query execution time.

Source: Author.

Where:

- N = The main table row count (Candidate Table)
- M = The joined table rows count (Party Table)

This function will give us the following figures:

	First Phase	Second Phase	Third Phase
MySQL Query Execution Time	0.3085ms	3.7988ms	486.7575ms
Big O Notation	9	20,000	1,010,000

Table 2: MySQL Query execution time vs Big O Notation.

Source: Author

And upon plotting these figures in a graph we will notice that both lines taking a similar shape, however not identical and that due to the optimizations that have been applied in MySQL query optimizer over the years.

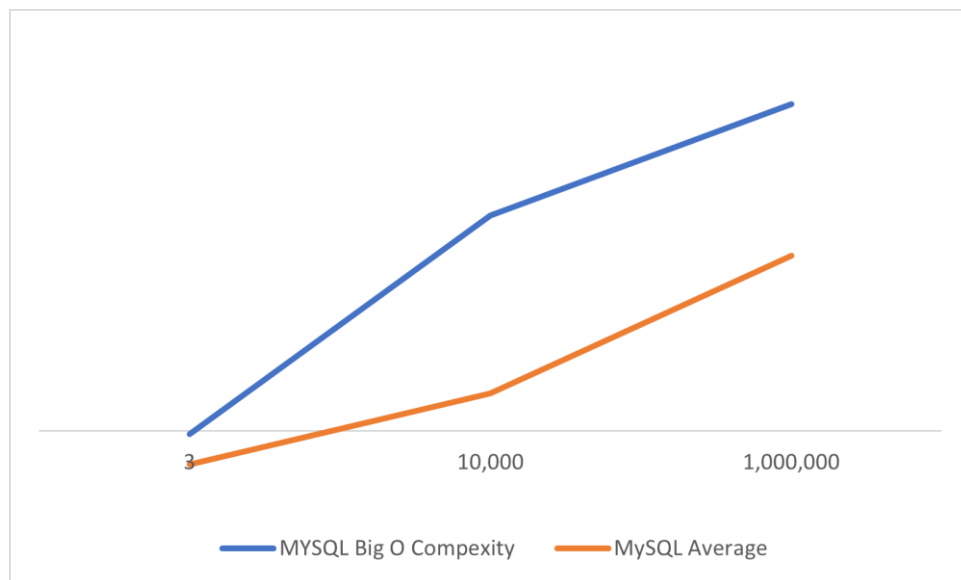


Figure 33: Comparison between MySQL execution time and Big O Complexity.

Source: Author

7 Results and Discussion

7.1 Getting Started

In the previous chapter, both MySQL and CouchDb are used to implement the same requirements. This demonstrates that either system is capable of storing and retrieving complex data structures if need be. However, the NoSQL database has shown many advantages over the SQL database and can be seen in all the implementation stages up to the application performance using large datasets.

7.2 Design and Implementation Stage

In the design stage, after building the common logical database design, the physical design of the MySQL database was more defined and rigid which requires much more work to alter it in case of any new requirements or any additional modifications are needed.

On the other hand, the NoSQL transformation from ERD was shorter and easier. Basically in defining the document we choose whether to embed or reference the related models. And it also leaves big room for adjustment and future modifications as you only need to add new properties or remove the unused ones or even use both models at the same time without the need for a complex migration from the old structure to the new.

7.3 Data Storage and Retrieval

Both database systems provide easy ways to store data, either by inserting a single record at a time or by patch inserting multiple records at a time. However, MySQL supports constrains checks which guarantee the integrity of the inserted data, NoSQL does not provide such functionality, however, it is less an issue since most related data stored in the document itself and does not need to be referenced.

Retrieving data, on the other hand, is completely different. Since each database has it is own way of querying data, MySQL uses SELECT statements and JOIN and Sub-Queries to retrieve related data and complex combinations. CouchDB uses build-in map-reduce functions to filter and aggregates the documents or custom JavaScript functions to implement even more complex queries.

7.4 Performance Benchmark

In the previous section, we have demonstrated the power of each database engine and how it can handle massive amounts of data, however, even though MySQL can still perform very well while dealing with millions of records, it has slowed significantly with the increased number of records in each table of in the select statement and joined tables.

This has not been an issue for CouchDB as the related data is part of the document itself and does not need to be fetched from another table. This has kept the query execution time nearly the same even with the increase of the data set size.

8 Conclusion

This thesis has introduced NoSQL databases, the logical reason that drove their invention such as the increasing demand for databases that can handle extreme amounts of data and flexible enough to meet the complex and agile development requirements, and their different types and the most common usage cases which they are tailored for, unlike the general-purpose SQL databases that they are commonly used for all kind of applications.

This thesis has also explained the reasons behind NoSQL databases surging popularity and why they are very common nowadays especially in cloud-based applications which is due to their ability to horizontally scale in a distributed system environment such as data centres. A feature that traditional SQL struggles to meet.

However, this thesis shows even though NoSQL databases have advantages in certain areas but this does not mean that they will replace the use of SQL database, due to many reasons, such as the maturity of current RDBMS and their wide-spread community and experienced developers who trust it and because of main features such the strong-consistency, and data integrity provided using a hard defined schema and integrity constrains which makes it suitable for transactional or data-sensitive applications, such as banking and e-commerce/warehouse platforms.

It has also shown in the practical part, that even though NoSQL database provides a better developer experience and more flexibility with working with complex data types. SQL databases still provide sufficient performance and easy to maintain queries to make them attractive for many developers looking for a database solution.

9 References

- Atkinson, Malcolm, DeWitt, Maier. (1992). The Object-oriented Database System Manifesto. *Morgan Kaufmann Publishers Inc.* Načteno z https://www.researchgate.net/publication/200034531_The_Object-oriented_Database_System_Manifesto
- Brook, C. (2020). SaaS: Single Tenant vs Multi-Tenant - What's the Difference? *Digital Guardian*. Načteno z <https://digitalguardian.com/blog/saas-single-tenant-vs-multi-tenant-whats-difference>
- Celko, J. (2010). *Joe Celko's SQL for Smarties : Advanced SQL Programming*. Morgan Kaufmann.
- Connolly, Begg. (2005). *Database Systems - A Practical Approach to Design, Implementation, and Management, 4th Edition*. UNIVERSITY OF PAISLEY.
- CouchDB. (2021). *Technical Overview*. Načteno z CouchDB Documentation: <https://docs.couchdb.org/en/stable/intro/overview.html>
- Dr. Tom Seymour, Kristi Berg. (2012). History Of Databases. *International Journal of Management & Information Systems (IJMIS)*. Retrieved from https://www.researchgate.net/publication/298332910_History_Of_Databases
- Hills, T. (2016). *NoSQL and SQL Data Modeling Bringing Together Data, Semantics, and Software*. Technics Publications.
- labs, s. (2015). Scalable Database Solutions for Multi Tenant Applications – The Rise of NoSQL. *sogeti labs*. Načteno z <https://labs.sogeti.com/scalable-database-solutions-for-multi-tenant-applications-the-rise-of-nosql/>
- MySQL. (2021). *Getting Started with MySQL*. Načteno z MySQL Documentation: <https://dev.mysql.com/doc/mysql-getting-started/en/>
- Naeem, T. (2020). Understanding Structured, Semi-Structured, and Unstructured Data. *Astera Blog*. Načteno z <https://www.astera.com/type/blog/structured-semi-structured-and-unstructured-data/>
- Pethuru Raj, Ganesh Chandra Deka. (2018). *A Deep Dive into NoSQL Databases The Use Cases and Applications*. Academic Press.
- PHP. (nedatováno). *Preface*. Načteno z PHP official documentation: <https://www.php.net/manual/en/preface.php>
- Pivert, O. (2018). *NoSQL Data Models : Trends and Challenges*. Wiley.
- Pramod Sadalage, Martin Fowler. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional.

- Ray. (2017). Always On Failover Cluster Instances (SQL Server). *Microsoft Developer Network*. Načteno z <https://docs.microsoft.com/en-us/sql/sql-server/failover-clusters/windows/always-on-failover-cluster-instances-sql-server?view=sql-server-ver15>
- Schallehn, D.-I. E. (2019). *Advanced Database Models*. Magdeburg - Germany: University of Magdeburg - Faculty of Computer Science.
- Vaish, G. (2013). *Getting Started with NoSQL*. Packt Publishing.
- Vogels, W. (2008). Eventually Consistent - Revisited. *All Things Distributed*. Načteno z https://www.allthingsdistributed.com/2008/12/eventually_consistent.html
- Zola, W. (2017). 6 Rules of Thumb for MongoDB Schema Design. *MongoDB*. Načteno z <https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-3#:~:text=Denormalization%20allows%20you%20to%20avoid,if%20you've%20missed%20them.>

10 Appendix