

Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Technologies



Diploma Thesis

Comparison of Graph and Relational Databases

Bo Bunmeng

© 2022 CULS Prague

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

DIPLOMA THESIS ASSIGNMENT

Bunmeng Bo

Systems Engineering and Informatics
Informatics

Thesis title

Comparison of Graph and Relational databases

Objectives of thesis

The goal of the thesis is to design, create, and test a small database application for parts of the car renting business model (car, client, staff, rent ...) in a graph database environment and also a relational database.

Methodology

The first part of the thesis contains the description of the theoretical tools used. The second part of the work will be a personal project in the Dgraph environment for the Graph Database and in MySQL for Relational Database. For documentation, UML standard will be used. Both applications (graph and relational) will contain the same data and the same queries. At the conclusion of the thesis, the differences, the advantages and disadvantages of both used technologies will be evaluated.

The proposed extent of the thesis

60-80 pages

Keywords

graph database; Dgraph; relational database; MySQL

Recommended information sources

Meier, A. a Kaufmann, M., 2019. SQL & NoSQL databases: models, languages, consistency options and architectures for Big data management. Wiesbaden: Springer. ISBN 978-3-658-24548-1.
Merunka, V., 2002. Objektový přístup v databázových systémech. Praha: Credit. ISBN 80-213-0882-6.
Robinson, I., Webber, J. a Eifrem, E., 2015. Graph databases. Second edition. Sebastopol: O'Reilly Media. ISBN 978-1-491-93089-2.

Expected date of thesis defence

2022/23 SS – FEM

The Diploma Thesis Supervisor

Ing. Marek Pícka, Ph.D.

Supervising department

Department of Information Engineering

Electronic approval: 4. 11. 2022

Ing. Martin Pelikán, Ph.D.

Head of department

Electronic approval: 28. 11. 2022

doc. Ing. Tomáš Šubrt, Ph.D.

Dean

Prague on 30. 11. 2022

Declaration

I declare that I have worked on my diploma thesis titled "Comparison of Graph and Relational Databases" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the diploma thesis, I declare that the thesis does not break copyrights of any person.

In Prague on 30.11.2022

Acknowledgement

I would like to thank my supervisor Marek Pícka and professor Vojtěch Merunka for their advice and support during my work on this thesis. I would like to take this opportunity to show my deep gratitude to my family, especially my parents namely Bo Kunthearith and Heng Kimchry, for their emotional support and unconditional love throughout the whole process. Although they are not here with me in the Czech Republic, their love from afar remains my biggest inspiration. Also, I would not forget to mention my manager, Robert Varga, for his effort to provide all the support I could get at the workplace.

Comparison of Graph and Relational Databases

Abstract

In this thesis, relational and graph databases have been selected as the two main databases to be evaluated and discussed. Databases are one of the most important entities in any type of application for a long run. While relational databases have been around for many decades, the growth of other types of databases is also remarkable, especially Graph database. Each type of databases has their own benefits and drawbacks. The main focus of the study is to find out the features provided by Relational database with MySQL and the ones by Graph database with GraphQL in Dgraph environment. The databases are designed to store the data of a small part of the car renting business model, which are used to display to users in a simple iOS application that has also been built during the process. These databases are evaluated based on their complexities and compatibilities with the client-side application, an iOS mobile application. The results discussed are focused on time saving for an application development and developer-friendliness of each database. Hence, those results are meant to help designers, project managers, developers, and people related to the field make decision towards the selection of the databases for their future projects.

Keywords: graph database; dgraph; graphql; relational database; mysql

Porovnání grafové a relační databáze

Abstrakt

V této práci byly vybrány relační a grafové databáze jako dvě hlavní databáze, které budou hodnoceny a diskutovány. Databáze jsou dlouhodobě jednou z nejdůležitějších entit v jakémkoli typu aplikací. Zatímco relační databáze existují již mnoho desetiletí, pozoruhodný je i růst jiných typů databází, zejména databáze Graph. Každý typ databází má své výhody a nevýhody. Hlavním zaměřením studie je zjištění funkcí, které poskytuje Relační databáze s MySQL a databáze Graph s GraphQL v prostředí Dgraph. Databáze jsou navrženy tak, aby ukládaly data malé části obchodního modelu pronájmu aut, která se používají k zobrazení uživatelům v jednoduché aplikaci pro iOS, která byla také vytvořena během procesu. Tyto databáze jsou hodnoceny na základě jejich složitosti a kompatibility s klientskou aplikací, mobilní aplikací pro iOS. Diskutované výsledky jsou zaměřeny na úsporu času při vývoji aplikace a vývojářskou přívětivost každé databáze. Tyto výsledky mají tedy pomoci návrhářům, projektovým manažerům, vývojářům a lidem souvisejícím s oborem při rozhodování o výběru databází pro jejich budoucí projekty.

Klíčová slova: grafová databáze; dgraph; graphql; relační databáze; mysql

Table of content

1.	<i>Introduction</i>	12
2.	<i>Objectives and Methodology</i>	13
2.1	Objectives	13
2.2	Methodology	13
3.	<i>Literature Review</i>	14
3.1	Relational Database	14
3.1.1	What is a Relational Database?	14
3.1.2	History of Relational Database	15
3.1.3	ACID Transaction Goal	16
3.1.4	Primary Key and Foreign Key	16
3.1.5	Relationships.....	17
3.1.6	Database Normalisation.....	18
3.2	Graph Database	20
3.2.1	What is a Graph Database?	20
3.2.2	NoSQL	21
3.2.3	Graph Database Types	23
3.3	iOS Mobile Application	28
3.3.1	About iOS	28
3.3.2	Guides to iOS Development	30
3.4	Car Rental Business	31
4.	<i>Practical Part</i>	33
4.1	Data Dictionary	33
4.2	Database with MySQL	34
4.2.1	UML	34
4.2.2	MySQL Database Preparation	35
4.3	Database with Dgraph GraphQL	38
4.3.1	UML	38
4.3.2	Dgraph GraphQL Database Preparation	38

4.4	Client-Side Application	41
4.4.1	Vapor API	41
4.4.2	Dgraph API	49
4.4.3	iOS Application	52
5.	<i>Results and Discussion</i>	65
6.	<i>Conclusion</i>	68
7.	<i>References</i>	70

List of pictures

Figure 1:	Relational and Graph databases	12
Figure 2:	Entity Relational Diagram Example of a Database Schema	17
Figure 3:	Database Normalisation Process	20
Figure 4:	Graph Database Example	21
Figure 5:	NoSQL Databases	23
Figure 6:	Property Graph	24
Figure 7:	RDF Triple	26
Figure 8:	iOS Supporting Devices	29
Figure 9:	Revenue history for Apple (Source: companiesmarketcap.com)	29
Figure 10:	Code Comparison between Objective-C and Swift	31
Figure 11:	Projected Global Car Rental Market Revenue (Source: Zippia.com)	32
Figure 12:	UML of Car Rental Relational Database Schema	34
Figure 13:	SQL Create Table identity	35
Figure 14:	SQL Create Table person	36
Figure 15:	SQL Create Table client	36
Figure 16:	SQL Create Table staff	36
Figure 17:	SQL Create Table car	37
Figure 18:	SQL Create Table price_type	37
Figure 19:	SQL Create Table rent	37
Figure 20:	UML of Car Rental Graph Database Schema	38
Figure 21:	GraphQL Enums (IDType, StaffGrade, CarCondition, GearType)	39
Figure 22:	GraphQL interface Person	39
Figure 23:	GraphQL Types (Client, Identity, Staff)	40

Figure 24: GraphQL Types (PriceType, Car, Rent)	40
Figure 25: Architecture of Application.....	41
Figure 26: Vapor API Workflow	42
Figure 27: MySQLKit MySQL Configuration	43
Figure 28: SQL query for all Clients	44
Figure 29: SQL query for all Staff.....	44
Figure 30: SQL query for all Cars	45
Figure 31: SQL query for all Rents.....	45
Figure 32: Controllers extend RouteCollection protocol.....	46
Figure 33: Routes Registration	47
Figure 34: Sample of struct Rent implements Codable protocol	47
Figure 35: getAll Method	48
Figure 36: Sample Result of Cars	48
Figure 37: GraphQL for Clients and Results	50
Figure 38: GraphQL for Staff and Results.....	50
Figure 39: GraphQL for Cars and Results	51
Figure 40: GraphQL for Rents.....	51
Figure 41: Results for GraphQL Rent.....	52
Figure 42: Swift Programming Language v5 and XCode IDE v14.....	53
Figure 43: iOS Application Workflow	53
Figure 44: Car Rent (Admin View) Launch Screen	54
Figure 45: Car Rent (Admin View) Main Screen.....	55
Figure 46: Display Screen for MySQL and GraphQL.....	55
Figure 47: Car Rent (Admin View) Menu Clicked	56
Figure 48: Car Rent (Admin View) Display All Clients	61
Figure 49: Car Rent (Admin View) Display All Staff.....	62
Figure 50: Car Rent (Admin View) Display All Cars	63
Figure 51: Car Rent (Admin View) Display All Rents.....	64

List of tables

Table 1: Relational Database Example	14
Table 2: Relational Model Terminology.....	15
Table 3: Relational Database Top Vendors and the Products.....	15
Table 4: Data Redundancy Example.....	19

Table 5: RDF subject-predicate-object26
Table 6: Data Dictionary34
Table 7: MySQL and GraphQL Evaluation67

1. Introduction

In today modern society, data is part of human’s daily life. Everyone generates and receives data in their own way. People use it to make sense of things around them. Data can be used in various ways from solving a small math problem to sending a rocket to the space. Just in 2022 alone, the prediction of data generated at the end of the year would be about 94 zettabytes which equals to 94,000,000,000,000 GB (Techjury, 2022).

With the exponential growth of data, we can also see the changes in data storing management. Up until now, there are Relational Database, Object-Oriented Database, NoSQL Database, Graph Database, and many more. Relational Database has been well-known over the years and been selected as a subject for teaching database by many institutions. Meanwhile, there is also a remarkable growth of Graph Database usage.

The information in a Relational Database is stored structured about other data and this type of database is often used when the integrity of the data is concerned. While in Graph Database, the data and the connections between them equally share the same value (Indeed.com, 2021).

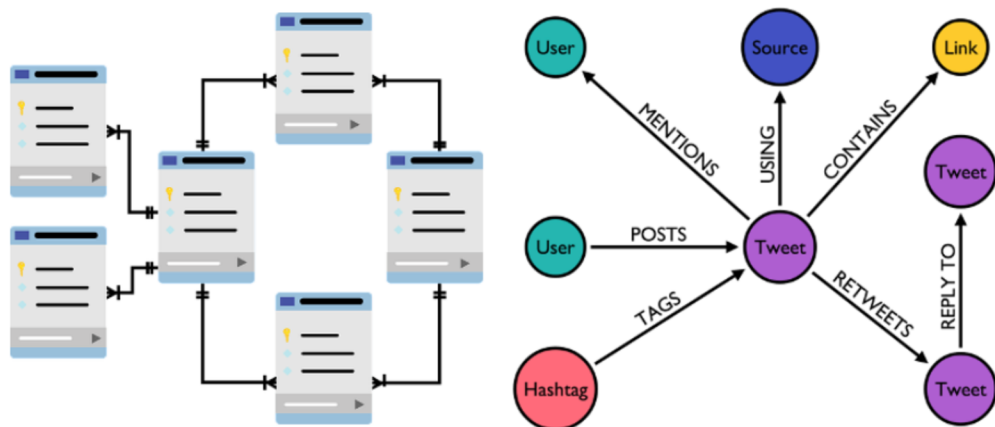


Figure 1: Relational and Graph databases

As each databases have its own strengths and weaknesses and along with the rise in popularity of Graph Database, this research is introduced to experiment Relational Database and Graph Database to understand better between these two.

2. Objectives and Methodology

2.1 Objectives

The main objective of the thesis is to design, create, and test a small database application for some parts of the car renting business model (car, client, staff, rent, ...) in a Graph Database environment as well as in Relational Database.

2.2 Methodology

The first part of the thesis contains the description of the theoretical tools used. The second part of the work will be a personal project in the Dgraph environment for the Graph Database and in MySQL for the Relational Database. For documentation, UML standard will be used. Both applications (Graph and Relational) will contain the same data and the same queries. At the conclusion of the thesis, the differences, the advantages and disadvantages of these two technologies will be evaluated.

3. Literature Review

3.1 Relational Database

3.1.1 What is a Relational Database?

A relational database is a type of database that stores and provides data as a set of tables with columns and rows. Each table has its own pre-defined relationship and is used to store information of a certain object. The rows in a table of this type of database are called records, each of which has a unique ID called the key. The columns, on the other hand, are the attributes of the data/table with its value being store in each record (Oracle, 2021).

first_name	last_name	sex	date_of_birth	identity_id
Michael	Lim	M	14.05.1994	4
Steven	Okrud	M	20.03.1996	9
Hailey	Williams	F	31.01.1992	13
John	Tucker	M	28.02.1994	14
Amy	Stark	F	09.09.1995	20
Anna	Jeiserova	F	08.12.1994	22
Shelly	Rower	F	04.07.1995	23

Table 1: Relational Database Example

In the example, the table represents table **Person** with five attributes “first_name”, “last_name”, “sex”, “date_of_birth”, and “identity_id”. Each row is a record that has a value corresponding to each attribute.

As the relational model provides a single way to represent data: as a two-dimensional table called a relation, there are some important terminologies regarding relations (Garcia-Molina, Jeffrey & Jennifer, 2014). Table 2 displays some important relational model terminologies with their explanations.

Terminology	Explanation
Attributes	The names of the columns of a relation (first_name, last_name, sex, etc.)
Schemas	The name of a relation and the set of attributes for a relation (Person, Book, etc.)
Tuples	The rows of a relation other than the header row containing the attribute names (Michael, 14.05.1994, M, etc.)
Relation	Table in a database
Entity	Names of a table

Table 2: Relational Model Terminology

3.1.2 History of Relational Database

The first introduction of the relational database was in 1970 after Edgar F. Codd, a computer scientist from IBM, had published an academic paper containing his proposed ideas regarding a new way to model data, which was called a relational model. The way that people thought about databased was changed since then and this relational model became continuously dominant in the entire database market in the 1980s and '90s. SQL, Structured Query Language, was chosen as the standard query language by the American National Standards Institute in 1986 and the International Organisation for Standardisation in 1987 (Quickbase, 2022).

Presently, many giant tech companies have developed their own Relational Database Management System (RDBMS) with different capabilities and cost. Some top vendors and their products are listed in the table below:

Vendor	Product
Microsoft Corporation	Microsoft SQL Server
Oracle	Oracle Database
Amazon Web Services	Amazon Relational Database Service (RDS)
Oracle Corporation	MySQL
IBM	IBM Db2
PostgreSQL Global Development Group	PostgreSQL
Microsoft Corporation	Azure SQL Database

Table 3: Relational Database Top Vendors and the Products

3.1.3 ACID Transaction Goal

A transaction is a piece of work that a user submits in one go to a database that might be made up of a single interactive command or of several commands sent from an application program (Harrington, 2009). In 1983, a collection of properties, including Atomicity, Consistency, Isolation, and Durability (ACID), was developed in order to enhance transaction reliability (Haerder & Andreas, 1983).

- Atomicity

Every transaction should be atomic. A DBMS with atomic transactions never leaves a transaction unfinished (data consistency problems, power failures, and so on). That means if any portion of the transaction fails, the entire transaction should fail without affecting the databases in any way.

- Consistency

After every transaction, the database should remain consistent. In other words, each piece of data shall conform to all constraints and whether the transactions are corrects or not, they must maintain the consistency of the database.

- Isolation

Each ongoing transaction should be independent of other concurrent transactions up until it has been properly completed and committed, according to the isolation property. As a result, the outcome of two transactions running simultaneously should be the same as if one transaction had run completely first, then the other.

- Durability

Durability refers to when changes made by a transaction should persist in the database even in the event of a system failure, power outage, or error. In other words, a transaction is permanent once it is finalised.

3.1.4 Primary Key and Foreign Key

A Primary Key (PK), as Davidson (2021) mentioned in his book, refers to one single attribute or a composite of multiple attributes in an entity that is used to identify each record. The value or the combined values of attribute(s) selected as a primary key must be unique in the whole table and can never have NULL value. In an entity, there is only one primary key. Attributes that can be considered and set as a primary key are *person_identification_number*, *passport_id*, *student_number*, *car_license_number*, etc. Database management system

normally requires this value to be set to ensure the uniqueness in order to reduce the redundancy in the table.

A Foreign Key (FK) in an entity is the primary key of another entity (Davidson, 2021). An entity can have one or multiple foreign keys and each foreign key represents the entity with which the current entity has a relationship.

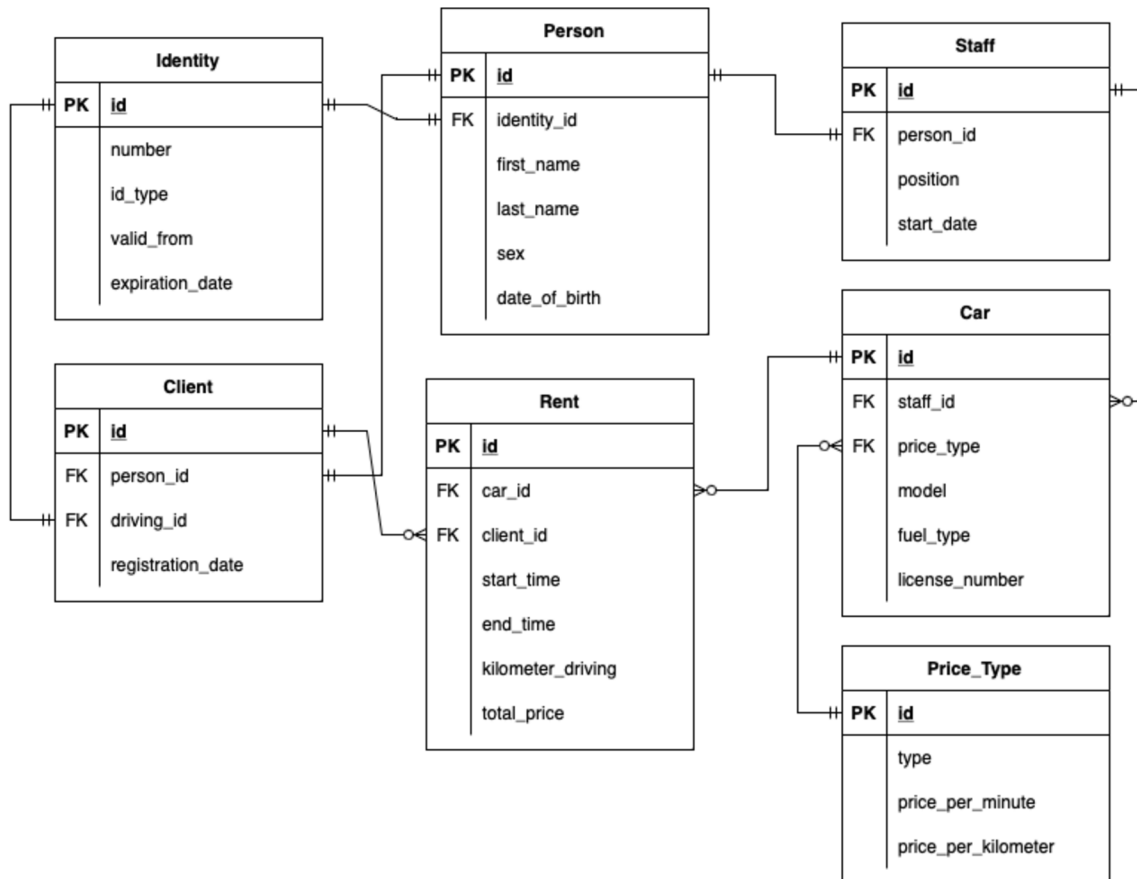


Figure 2: Entity Relational Diagram Example of a Database Schema

Figure 2 displays an entity relational diagram (ERD) of a database schema. There are 7 entities in this figure such as Identity, Person, Client, Staff, PriceType, Car and Rent. Each of these entities have an attribute called “id” as their Primary Key. The attributes “person_id”, “driving_id”, “identity_id”, “car_id”, “client_id”, “staff_id”, and “price_type” (Car) are the Foreign Keys, which originally the Primary Keys of other entities.

3.1.5 Relationships

Harrington (2009) stated in his book that entities in relational database generally have relationships between them. Relationship refers to the association of a to one or more tables

that is connected by a foreign key referencing the primary key of those tables. There are 3 types of relationship in relational database – one-to-one (1:1), one-to-many (1:M), and many-to-many (M:N or M:M).

- One-to-One (1:1) relationship refers to a relationship between two tables that only share one record on both side, which means a foreign key of a referenced table in the current table also has unique value as its own primary key.

Figure 2: entity **Person** and **Identity** have a one-to-one relationship. This can be described in a simple term as: “one person can only have one identity (card)” or “one identity (card) is owned by one single person”

- One-to-Many (1:M) relationship refers to a relationship of which one record in an entity may have one or more records in the other table. This can be explained by the duplicated value of Foreign Key found in one table.

Figure 2: entity **Client** and **Rent** have a one-to-many relationship. In simple explanation, “a client can rent (a car) multiple times; however, during one car rental, there is only one client.”

- Many-to-Many (M:N or M:M) relationship refers to the fact that one record of table A can be found multiple times in table B and one record in table B can also be found multiple times in table B. This kind of relationship is very uncommon and when there is such a relationship, it will be transformed into 1:M relationship via some database techniques.

3.1.6 Database Normalisation

Database normalisation is a process of replacing duplicate attributes, which are normally called redundant data, in a table with a reference to the original one in order to improve storage efficiency, data integrity, and scalability (Date, 2012).

Data redundancy refers to a piece of data that appears in multiple places. Data inconsistency, on the other hand, occurs when the same data have different formats in multiple tables.

In the table 4, it shows the data redundancy example of a table which a part of the car renting database schema. Looking at the column “CarLicenseNumber” and “TotalPrice”, there are a lot of values there in one box, which is understandable in a real-world scenario as a person

might drive or rent several cars and each rental costs differently. However, the values of the column “CarLicenseNumber” such as “CZ 0198”, “CZ 1829” and “CZ 2014” appear several times in the table. These values are called data redundancy.

Table 4: Data Redundancy Example

ID	FirstName	LastName	IdentityNumber	CarLicenseNumber	TotalPrice
1	John	Tucker	N012345678	CZ 0198	450
				CZ 1829	320
2	Melissa	Rower	N012345679	CZ 2014	230
3	Jessica	Miller	N012345680	CZ 1829	350
				CZ 2014	560
4	Amy	Stark	N012345681	CZ 1829	220
				CZ 2014	330
				CZ 0198	110

Data inconsistency can be caused by data redundancy because the values of the same attribute existed in different table have not been updated consistently, which leads to same attribute but different value (Yaowen, 2016). Additionally, data redundancy would also unnecessarily increase the size of the storage

Edgar F. Codd (1970) was the first person to propose, in his paper, the process of normalisation which he called a very simple elimination procedure of the nonsimple domains and replaced by domains that have atomic (undecomposable) values. Furthermore, he established three normal forms which are called First Normal Form (1NF), Second Normal Form (2NF), and Third Normal Form (3NF) (Codd, 1972). Up until today, there are now other NFs, however it is widely considered acceptable if the tables in a database reach 3NF.

Database normalisation is going through below processes:

- **First Normal Form (1NF)**

First Normal Form is used to deal with removal of redundancy of data in the records/rows. Tables in a database is considered in 1NF when each field in the table conveys unique information and the attributes in that table are single valued. Additionally, there must not be any repeating groups of attributes.

- **Second Normal Form (2NF)**

Second Normal Form, similarly to the First Normal Form, but instead of dealing with data redundancy across a horizontal row, it deals in vertical columns. 2NF tables are those tables that do not have any column data that exists in a table with which they have a relationship, instead those data can be fetched by relying on its primary key.

- **Third Normal Form (3NF)**

Third Normal Form is intended to minimise data duplication and ensure referential integrity. Every attribute in the table must solely depend on the primary key and not on any non-prime attributes for a relation to be in 3NF, which is only possible if it is in 2NF.

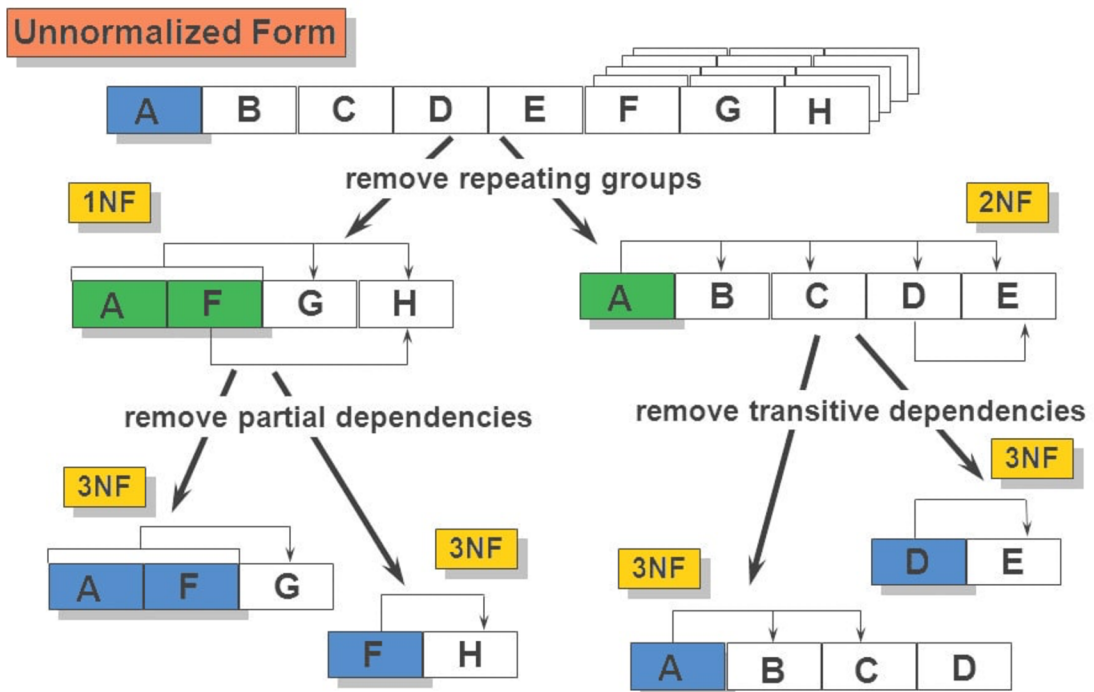


Figure 3: Database Normalisation Process

3.2 Graph Database

3.2.1 What is a Graph Database?

A graph is simply a set of vertices and edges, or, in less daunting terms, a collection of nodes and the connections between them (Robinson, Emil & Jim, 2015). Entities are shown as nodes in graphs, and relationships are the connection between those entities related to the world.

Harrison (2015) stated in his book that there are three major components of a graph such as:

- **Vertices**, or “nodes” represent distinct objects
- **Edges**, or “relationships” or “arcs” connect these objects
- **Properties** are the attributes of vertices or edges

Figure 4 represents a small social data in graph. Each node has User as the label. The “FOLLOWS” relationships connect these nodes together, which further establish the semantic context as Harry and Jasper follow each other; Harry and Jess also follow each other; however, Jasper follows Jess, yet Jess has not followed Jasper.

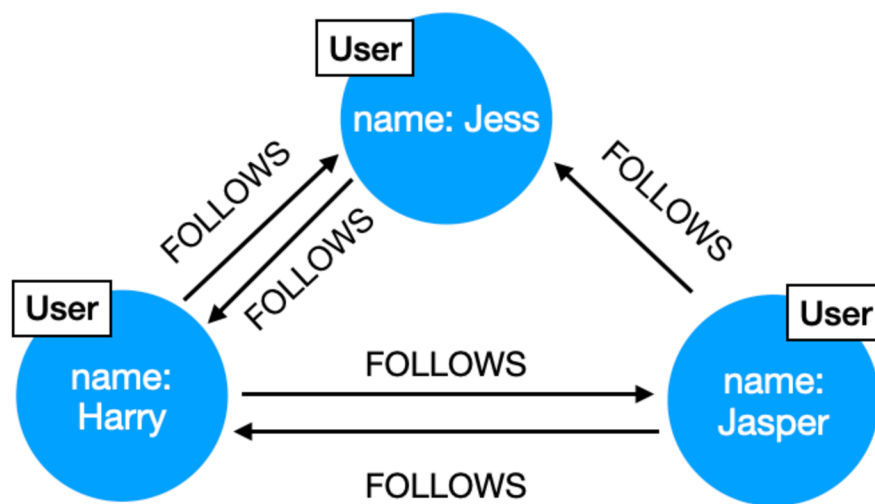


Figure 4: Graph Database Example

In closing, a graph database refers to a database that stores data using graphs, a type of highly interconnected data structure. Since it is simple to represent social actors as nodes, edges as relationships between users, and properties as social data of each user, it is very helpful for social networking applications.

3.2.2 NoSQL

Graph database started its fame after NoSQL database became effective. Kristi (2012) on History of Databases stated that in 1998, Carlo Strozzi used the term “NoSQL” to describe an open-source, lightweight relational database that did not provide the traditional SQL interface.

However, it was not until 2009 that the term NoSQL came into effect when an event organised by Johan Oskarsson to discuss open-source distributed databases.

In the last decade, NoSQL databases have been increasingly popular due to the growth of cloud computing and large-scale web application (Jing, Haihong, Guan & Jian, 2011). NoSQL is a large category of database management systems that differs from the popular relational database management model by not being primarily constructed on tables (Vaish, 2013). In other words, NoSQL databases typically do not use SQL for data processing and are used in attempt to solve the problems of scalability and availability against that of atomicity or consistency.

NoSQL databases have four major types: column-oriented, document store, key value store, and graph. Figure 5 shows these four types of NoSQL graphically.

- **Column-oriented**

Column-oriented database systems (sometimes known as “column stores”) store each database table column independently with attribute values relating to the same column stored concurrently as opposed to the way the RDBMS stores data. In simplified term, this type of database stores use columns to store data of the same attribute instead of rows (Daniel, Boncz & Harizopoulos, 2009).

- **Document store**

A document store allows the semi-structured data to be inserted, retrieved from, and modified. The majority of the databases in this category employ XML, JSON, BSON, YAML, and data access is often made through HTTP utilising a RESTful API or the Apache Thrift protocol for compatibility between languages. Compared to RDBMS, the documents themselves function as records (or rows).

- **Key-value store**

A key-value store, which enables the storage of a value against a key, is very similar to a document store. In other words, the data in this category is stored as a typical hash table in a seamless way. Since it has a straightforward design (key-value), this storage model provides high availability, scalability, and application-user friendliness, which are particularly beneficial in distributed environment.

- **Graph store**

Graph database or graph store, which is a unique subset of NoSQL database, displays relationships as graphs. In a graph, there may be several links between any two nodes, signifying the various connections that the two nodes have.

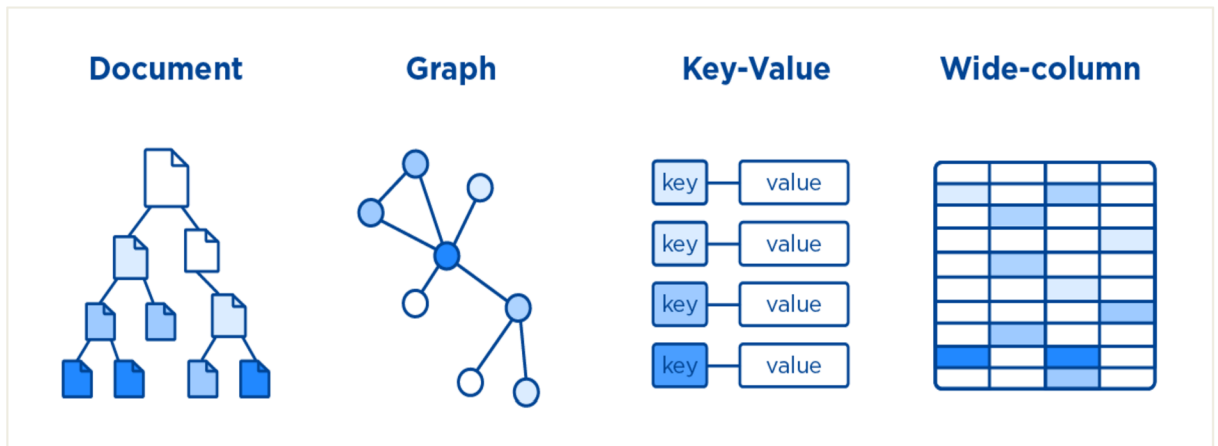


Figure 5: NoSQL Databases

3.2.3 Graph Database Types

Based on the underlying graph data structures, different graph databases have been categorised, including:

- Labeled Property Graphs
- RDF (Resource Description Framework) Triple Stores

Both RDF stores and property graphs are designed to store data that is graphically organised and provide a variety of ways to access it. But the implementation of these two graph databases and structural design differ greatly from one another.

3.2.3.1 Property Graphs or Labeled Property Graphs

A property graph or a labeled property graph (LPG) is made up of a set of nodes and a set of edges, with each node and edge effectively being a “struct” – a basic data structure made up of keys and values. Nowadays, JSON is the preferred method for encoding these structs; each node and edge is a JSON document, with edges having unique keys that represents a pointer to a node.

A property graph has the following elements (Yaowen, 2016):

- A set of vertices:
 1. Each vertex has its unique identifier
 2. Each vertex has several incoming edges
 3. Each vertex has several outgoing edges
 4. Each vertex has several properties associated with it, defined by a map from key to value

- A set of edges:
 1. Each edge has its unique identifier
 2. Each edge has an incoming head vertex
 3. Each edge has an outgoing tail vertex
 4. Each edge has a number of properties associated with it, defined by a map from key to value
 5. Each edge has a label to denote that relationship between the incoming vertex and outgoing vertex

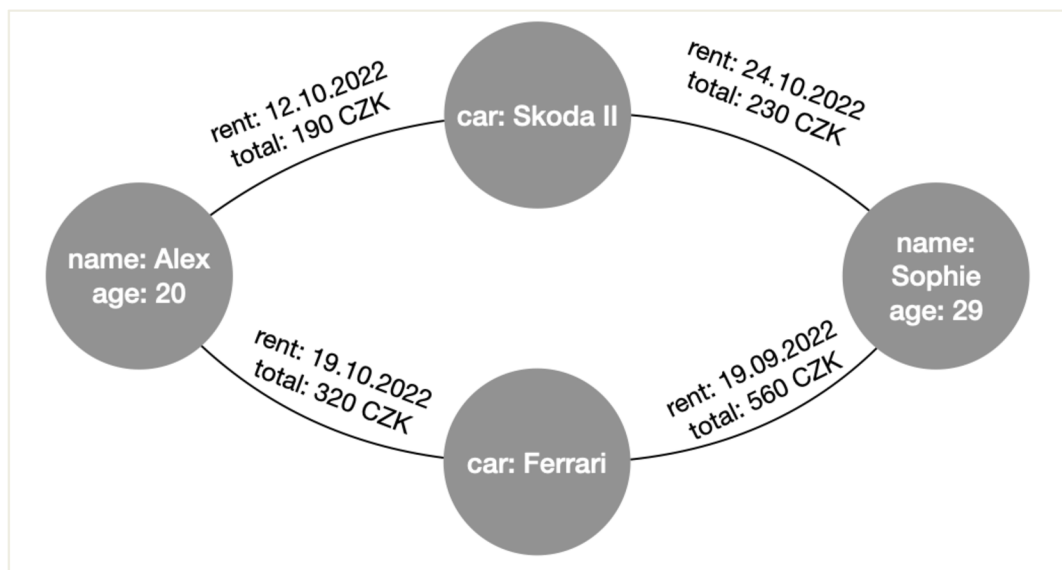


Figure 6: Property Graph

Figure 6 represents an example of a labeled property graph. It consists of 4 actors: “Alex”, “Sophie”, “Skoda II”, and “Ferrari”. They are nodes in the graph and their information is stored as the properties of nodes. At the same time, they are connected by edges and the information about the relationships is stored as the properties of edges.

Property graphs provide advantages such as:

- **Simplicity:** Property graphs are easy to use and set up quickly.
- **Easy Navigation:** Property graphs are simpler to navigate because they do not have constraints or predefined query languages.
- **Detailed:** Without having to add additional nodes for each detail, properties associated with relationship in property graphs provide more information about the data entries and their relationships. The user is in charge of how to interpret the data.

However, there are also disadvantages with property graphs:

- Lack of Interoperability: It is challenging to share or exchange data with multiple data storage since property graphs are not standardised. Because they are specific to the property graphs, the unique identifiers are meaningless to any other database.
- Vendor Lock-in: Business utilising graphs based on property graphs are unable to connect their data between various tools or systems. There is a very high likelihood of becoming trapped into a single vendor of property graphs.

3.2.3.2 RDF Triples

The Resource Description Framework (RDF) is web-born technology that was developed in 1990s at Netscap by Tim Bray as a meta-data framework for characterising objects. The fundamental concept is straightforward; RDF files are made up of a triple (subject, predicate, and object) of logical statements (Luke, 2022).

Using a range of syntax notations and data serialisation formats, RDF has been used as a broad way for conceptual description or modeling of information that is realised in web resources. Additionally, it breaks down all kinds of knowledge into manageable chunks while maintaining some standards for the semantics, or meaning, of those chunks. There are some facts about the RDF format (Powers, 2003):

- It is a data model where an RDF triple serves as the fundamental piece of data.
- RDF represents information based on the concept of subject-predicate-object expressions. It could alternatively be regarded as a name for an attribute or property, its value, or a resource identifier.
- A triple's subject and predicate must be URIs in order to make the information it states clear and unambiguous.

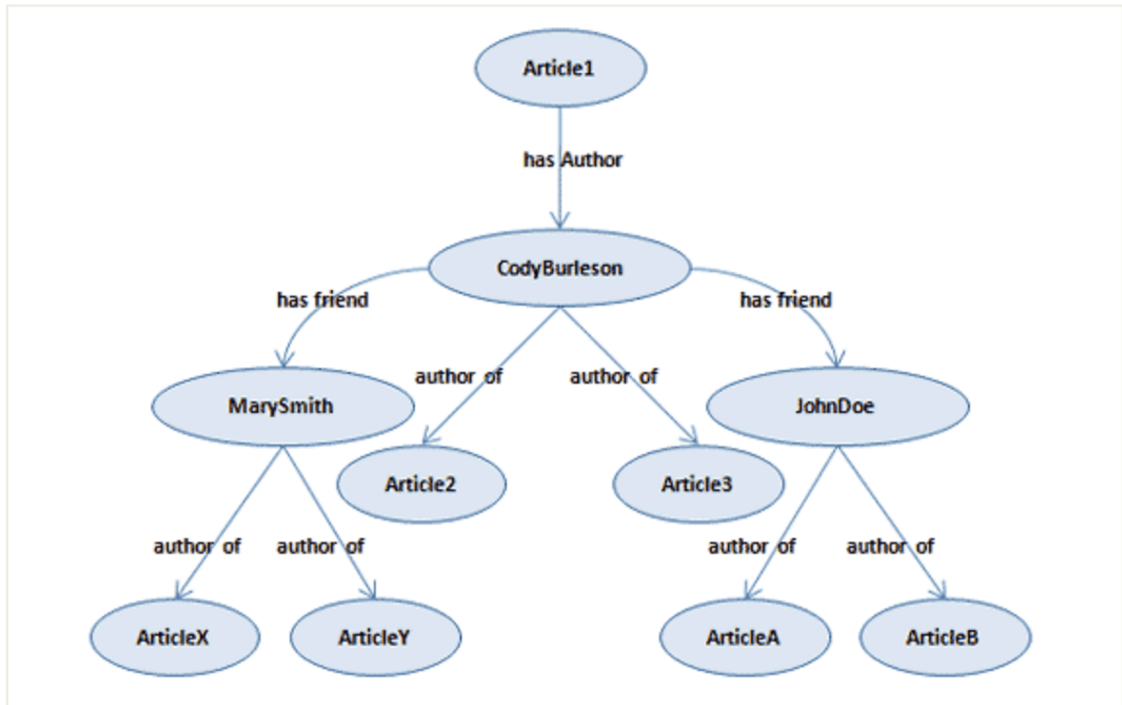


Figure 7: RDF Triple

The graph of the RDF triple in the figure 7 can be converted into the “subject-predicate-object” as below:

Subject	Predicate	Object
Article1	has Author	CodyBurleson
CodyBurleson	has friend	MarySmith
CodyBurleson	has friend	JohnDoe
CodyBurleson	author of	Article2
CodyBurleson	author of	Article3
MarySmith	author of	ArticleX
MarySmith	author of	ArticleY
JohnDoe	author of	ArticleA
JohnDoe	author of	ArticleB

Table 5: RDF subject-predicate-object

The Semantic Web now uses RDF extensively as a graph database and as one of the three core Semantic Web technologies. On the Semantic Web, there might not be enough data to tell whether two nodes are identical or not. RDF utilises the idea of the URI to address the identity issue. The WWW works very well to represent identity via URIs, therefore adopting

the URI as a standard for global identifiers enables a reference for any symbol to be used globally (Berners-Lee, James & Ora, 2001). It implies that individuals may determine whether any two users, located anywhere in the world, are making the identical reference. As standards body can easily declare the definition of particular phrases using this attribute of URIs. For usage with Web technologies, the World Wide Web Consortium (W3C) has established a number of standard namespaces, such as xsd for XML schema definition and xmlns for XML namespaces.

XML below represents a typical RDF file and shows how it is written in a car renting:

```
<?xml version="1.0"?>
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:company="http://company/Person#">
<rdf:Description
rdf:about="http://company/Person">
  <Person:firstName>Mary</Person:firstName>
  <Person:lastName>Owen</Person:lastName>
  <Person:sex>F</Person:sex>
</rdf:Description>
<rdf:Description
rdf:about="http://company/Person">
  <Person:firstName>John</Person:firstName>
  <Person:lastName>Hallo</Person:lastName>
  <Person:sex>M</Person:sex>
</rdf:Description>
</rdf:RDF>
```

Using RDF graphs has its own advantages and disadvantages (Vettrivel, 2022).

RDF graphs provide advantages such as:

- **Standardisation:** All RDF-based graphs have a common formal semantics, framework, and querying language for storing and representing data. Thanks to the web-native syntax of RDF, data sharing between RDF data stores on the web is made simpler.
- **Interoperability:** RDF Triple Stores adhere to a W3C-endorsed standard that enables communication between graphs. RDF-based graphs can interact and share information because of this interoperability.

- **Extensibility:** Users of RDF Graphs can add new nodes, relationships, or even substructures without having to recreate the database.

The disadvantages provided by RDF graphs are:

- **Deep Search Complexity:** A deep search in a big RDF network needs navigating every relationship, which is a challenging task.
- **Strict Adherence to Standards:** Only two objects can be linked at a time, which can be restricting for many use cases, as all data saved in RDF should be in the form of triples.

3.3 iOS Mobile Application

Techopedia (2020) defines a mobile application, more usually abbreviated as “an app,” as a category of application software created specifically to run on mobile devices like smartphones and tablets. Similar services to those accessed on PCs are routinely made available to consumers through mobile applications. Apps are often small, discrete software modules with constrained functionality. When mobile applications are discussed, in general two most famous mobile operating systems are focused: iOS and Android. However, this section will only discuss iOS mobile application as the client-side application of this project is built for iOS operating system.

3.3.1 About iOS

iOS is an operating system developed by Apple company (AAPL). It was first designed for the iPhone; however, it can now support iPod touch, iPad, and Apple TV. Despite being inspired from MacOS X, iOS offers features that are exclusive to it, like the multi-Touch interface and accelerometer support, which make the iPhone easier to use.



Figure 8: iOS Supporting Devices

When iPhone first hit the market in June 2007, the initial version of iOS was also introduced. All of mobile devices of Apple run iOS or iPhone Operating System, a Unix-based operating system. It was not until 2008 that Apple launched the iPhone software development kit (SDK), allowing anybody to create apps for the platform, that the term iOS was formally given to the program (Kenton, 2022). Up until now, the latest version of iOS is iOS 16.

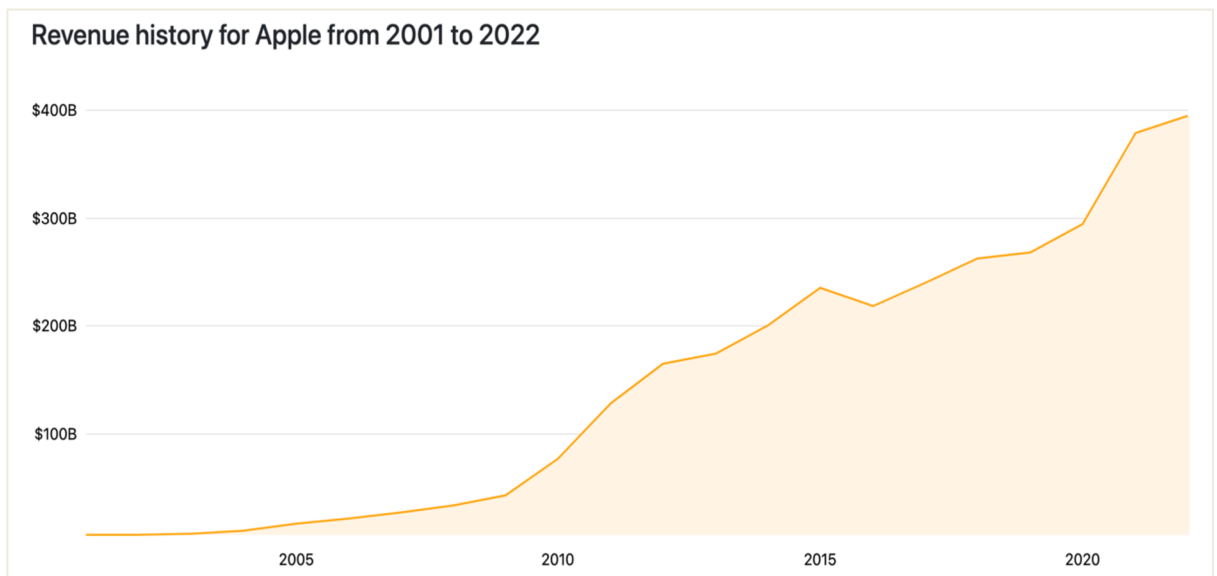


Figure 9: Revenue history for Apple (Source: companiesmarketcap.com)

Figure 9 represents the revenue history for Apple from 2001 to 2022. The graph shows how high the revenue of this company is. It is undeniable that iPhone alone makes up around

50% of the whole revenue, based on the Apple Statistics shown by Business of Apps (2022), and there are currently more than 1 billion Apple users.

3.3.2 Guides to iOS Development

In order to be able to develop iOS, there are some requirements that need to be fulfilled. IBM Cloud Education (2020) mentioned that developers are required to:

- Have a running Apple Mac computer with the latest version of macOS.
- Have an XCode, which is the sole IDE (Integrated Development Environment) for developing iOS app.
- Hold an active Apple Developer account. There is, although very limited, also a free Apple Developer account that can be registered to explore some basic iOS development, but without the possibility to push app into AppStore.

iOS application, at this moment, can be programmed with two programming languages:

- **Objective-C:** Objective-C served as the main programming language for all Apple devices for many years. The object-oriented programming language Objective-C, which is derived from C, is focused on conveying messages to various processes (as opposed to invoking a process in traditional C programming). Instead of converting their older Objective-C applications to the 2014-introduced Swift framework, many developers want to preserve them.
- **Swift:** Swift programming language is the new “official” language of iOS. Although Swift and Objective-C are quite similar, Swift is intended to have a simpler syntax and is more security-focused than its predecessor. Because it and Objective-C share a run time, updating apps with legacy code is simple. Even for those who are just learning to program, Swift is simple to learn. Unless one has a compelling reason to continue with Objective-C, one should aim to utilise Swift to develop the iOS app since it is faster, more secure, and simpler to use than Objective-C.

Objective-C	Swift
1 #import <Cocoa/Cocoa.h>	1 import Cocoa
2	2
3 @implementation aClass	3 class aClass {
4	4
5 - (void)aFunc:(int)a {	5 func aFunc(a: Int) {
6 NSString* s = @"Hi";	6 var s: String = "Hi"
7 NSArray* a = @[@"1",@"2"];	7 var a: Array = ["1","2"]
8 NSString* i;	8 var i: String
9	9
10 for ((i in a)	10 for i in a {
11 {	11 NSLog(@"i = %@", i)
12 NSLog(@"i = %@", i);	12 }
13 }	13 }
14 }	14 }
15	15
16 @end	16 }
	17
	18

Figure 10: Code Comparison between Objective-C and Swift

3.4 Car Rental Business

Car Rental is very well-known at the moment. Car Rental business has many services. It can be a service that allows a client to rent a car to use for days or months. This type of service is mostly served at the car rental company or branches, which means a client must go there and complete some forms before being able to use the service. The other service is the currently most popular one, car sharing. This service allows a client to use the distributed cars available at any moment. The available cars can be found parking somewhere specified in their respective application.

Salon (2022) shared a short history of Car Rental on the LinkedIn webpage that the oldest records of car rentals date back to 1904, when a Minneapolis bicycle shop began offering car rentals. Then, in 1912, around eight years later, a German corporate by the name of Sixt began renting out automobiles. They initially only offered three automobiles. But it did not take long for the business to start growing.

Up until today, as demand expanded, car rental business popped up everywhere. With the world becoming increasingly globally interconnected, people are travelling at a never-before-seen rate. They now hire cars, vans, and other vehicles more frequently than they did in the past.

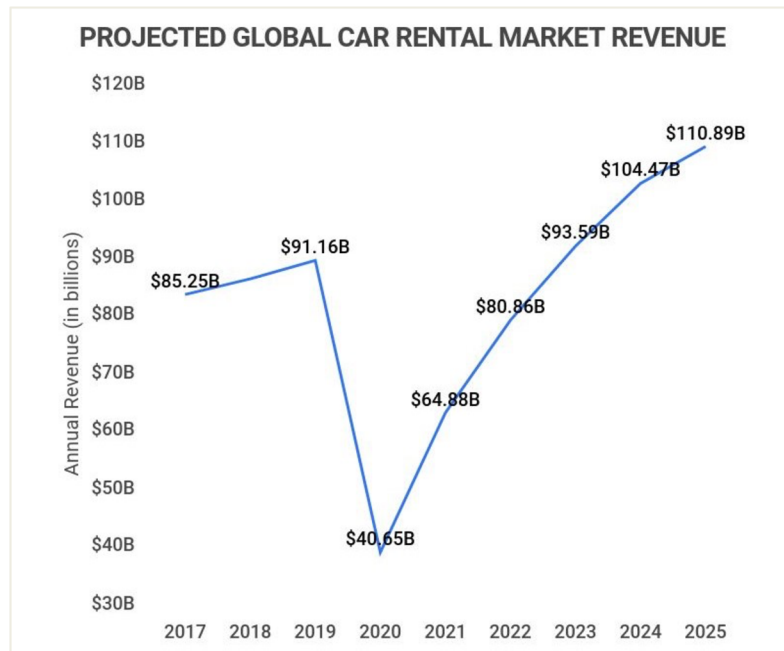


Figure 11: Projected Global Car Rental Market Revenue (Source: Zippia.com)

Although the car rental markets appear to be profitable over the years, it requires a lot of perseverance and good strategy plans. According to GrowThink (2022), there are several steps to be realised such as:

- Determine what type of car rental company to open
- Choose the name for the business and make sure the name is available, simple, but appealing and meaningful
- Develop the business plan
- Choose the legal structure for the business
- Secure startup funding and location
- Register the car rental business
- Open a business bank account and get a business credit card
- Get required business licenses and permits
- Get business insurance
- Buy or lease the right car rental business equipment
- Develop the marketing materials
- Purchase and setup the software needed to run the business
- Hire the team
- OPEN FOR BUSINESS

4. Practical Part

Car Rental business logic differs amongst business owners, each of whom have various tactics to tackle their market, which makes the data and its type that the business accumulates to store and to analyse are different between companies.

In this section, such Car Rental data is compromised into a generic one, which is considered as a necessary data for a Car Rental business to function.

4.1 Data Dictionary

Class	Description	Attribute
Identity	A class which stores the identity information of a person and the driving license information of a client	idNumber, idType, validFrom, expirationDate
Person	A superclass which holds a general information of a relevant person in the business (Client, Staff)	firstName, lastName, sex, dateOfBirth, nationality, phoneNumber, email, permanentAddress
Client	A class which holds the information of a client required to be able to use the service	registrationDate, currentAddress, drivingLicenseId
Staff	A class which stores the information of those who work in the service	currentAddress, typeOfContract, startDate, grade
PriceType	A class which saves the details of the price of the renting service set by the business	priceType, pricePerMinute, pricePerKilometer
Car	A class which keeps the details of the cars used in the service	model, color, fuelType, licenseNumber, gearType, entryDate, fuelAmount, currentLocation, kilometerCounter, carCondition

Rent	A class which stores the information of the renting services used by the client	startTime, endTime, kilometerDriving, totalPrice
------	---	--

Table 6: Data Dictionary

4.2 Database with MySQL

4.2.1 UML

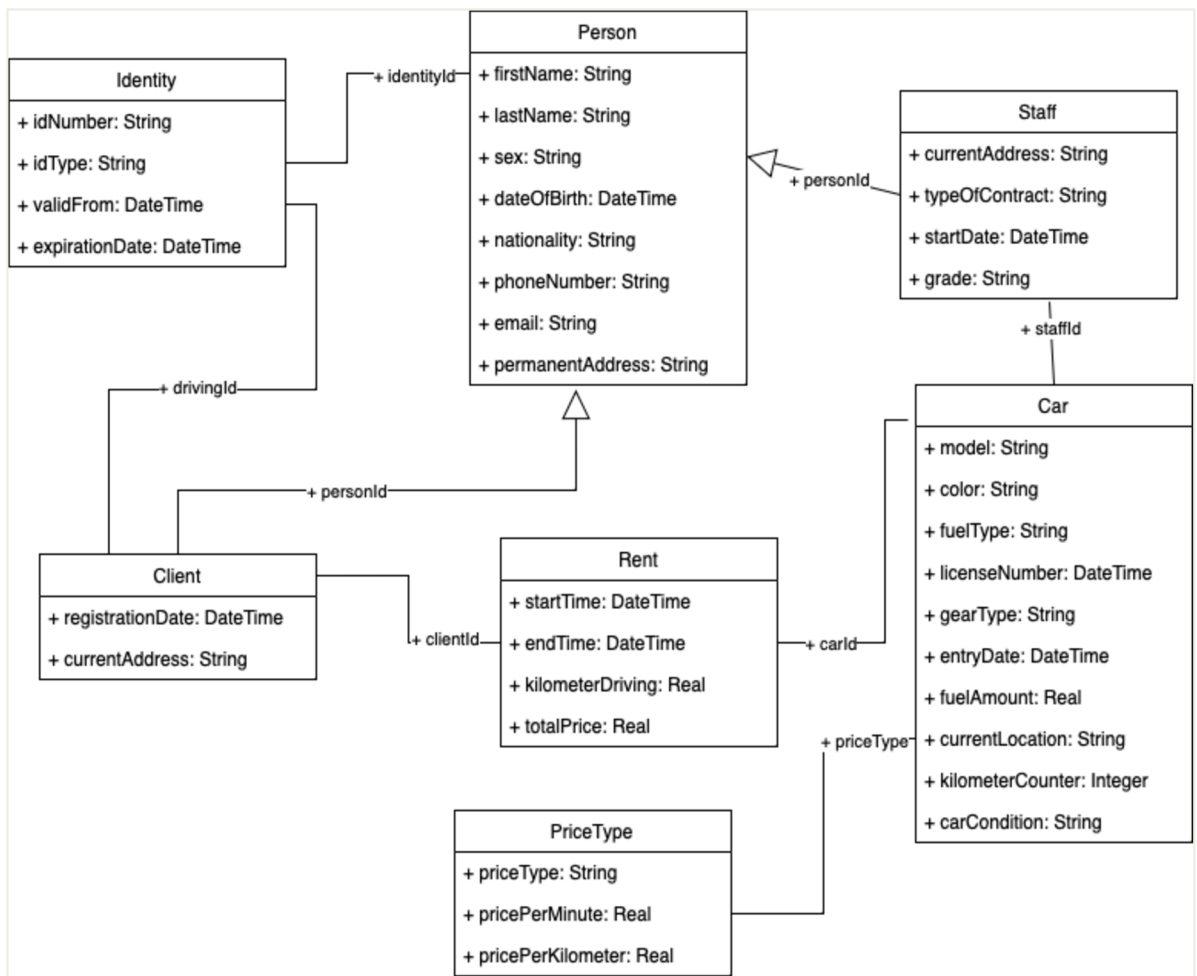


Figure 12: UML of Car Rental Relational Database Schema

- Table **Identity** is an independent one, storing all the information relating to a person's identification document, which can be a passport or an ID as well as a driving license of a client with the ID number and its validity.
- Table **Person** acts as a parent class for table **Client** and **Staff**. This table holds all basic but necessary information about a person who is related to the Car Renting service. Each

person needs to be verified as a legal one to be in this service, therefore the **Person** table holds and connects to the **Identity** table via the foreign key **identityId**.

- Table **Staff**, while extending its basic information from table **Person**, holds other information relating to the working environment.
- Table **Client** also extends the information from table **Person** while stores other information that such car renting service requires. This table also has a relationship with table **Identity** via its foreign key, **drivingId**, which is one of the most important information to verify that a person can legally drive.
- Table **PriceType** is a stand-alone table which has the information regarding the price of the car renting service usage set by the business owner.
- Table **Car** holds various information about cars being owned and used within the business. It has two relationships: the first one is with table **Staff** via **staffId** in order to provide information about who has worked with the car; the second one is with table **PriceType** via **priceType** to specify to which price category a car belongs.
- Table **Rent** is used to store information about the service being used. It has a relationship with table **Client** via **clientId** in order to provide information about who use the service; and it has another relationship with table **Car** via **carId** to specify which car is rented.

4.2.2 MySQL Database Preparation

Figures below are the SQL for creating all the tables.

```
CREATE TABLE identity (  
    id int NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    id_number varchar(30) NOT NULL,  
    id_type varchar(30) NOT NULL,  
    valid_from timestamp NOT NULL,  
    expiration_date timestamp NOT NULL  
);
```

Figure 13: SQL Create Table identity

```

CREATE TABLE person (
  id int NOT NULL AUTO_INCREMENT PRIMARY KEY,
  identity_id int NOT NULL,
  first_name varchar(50) NOT NULL,
  last_name varchar(50) NOT NULL,
  sex varchar(10),
  date_of_birth timestamp NOT NULL,
  nationality varchar(50),
  phone_number varchar(20) NOT NULL,
  email varchar(100),
  permanent_address text,
  FOREIGN KEY (identity_id) REFERENCES identity(id) ON UPDATE CASCADE ON DELETE CASCADE
);

```

Figure 14: SQL Create Table person

```

CREATE TABLE client (
  id int NOT NULL AUTO_INCREMENT PRIMARY KEY,
  person_id int NOT NULL,
  driving_id int NOT NULL,
  registration_date timestamp,
  current_address text,
  FOREIGN KEY (person_id) REFERENCES person(id) ON UPDATE CASCADE ON DELETE CASCADE,
  FOREIGN KEY (driving_id) REFERENCES identity(id) ON UPDATE CASCADE ON DELETE CASCADE
);

```

Figure 15: SQL Create Table client

```

CREATE TABLE staff (
  id int NOT NULL AUTO_INCREMENT PRIMARY KEY,
  person_id int NOT NULL,
  position varchar(100),
  current_address text,
  type_of_contract varchar(100),
  start_date timestamp,
  grade varchar(10),
  FOREIGN KEY (person_id) REFERENCES person(id) ON UPDATE CASCADE ON DELETE CASCADE
);

```

Figure 16: SQL Create Table staff

```

• ○ CREATE TABLE car (
    id int NOT NULL AUTO_INCREMENT PRIMARY KEY,
    model varchar(100) NOT NULL,
    color varchar(50),
    fuel_type varchar(10),
    license_number varchar(50),
    gear_type varchar(10),
    entry_date timestamp,
    staff_id int NOT NULL,
    price_type int NOT NULL,
    fuel_amount float,
    current_location text,
    kilometer_counter float,
    car_condition varchar(20),
    FOREIGN KEY (staff_id) REFERENCES staff(id),
    FOREIGN KEY (price_type) REFERENCES price_type(id)
);

```

Figure 17: SQL Create Table car

```

• ○ CREATE TABLE price_type (
    id int NOT NULL AUTO_INCREMENT PRIMARY KEY,
    price_type varchar(50) NOT NULL,
    price_per_minute float NOT NULL,
    price_per_kilometer float NOT NULL
);

```

Figure 18: SQL Create Table price_type

```

• ○ CREATE TABLE rent (
    id int NOT NULL AUTO_INCREMENT PRIMARY KEY,
    car_id int,
    client_id int,
    start_time timestamp,
    end_time timestamp,
    kilometer_driving float,
    total_price float
);

```

Figure 19: SQL Create Table rent

4.3 Database with Dgraph GraphQL

4.3.1 UML

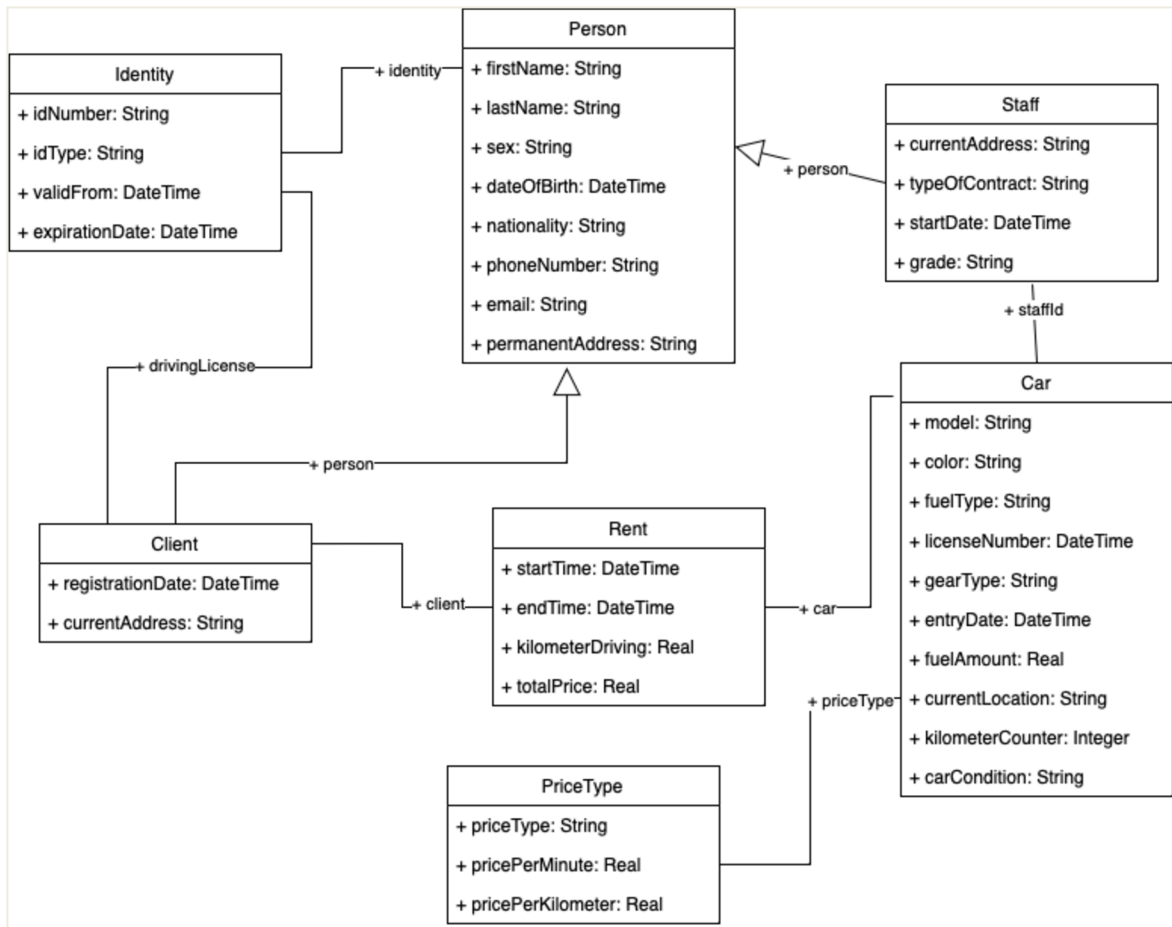


Figure 20: UML of Car Rental Graph Database Schema

Like the UML classes description in Database with MySQL section, the classes for GraphQL also store all the necessary information required for the car renting service. However, the relationship between each class in GraphQL is denoted by an object (**identity person client car priceType**) rather than by a foreign key.

4.3.2 Dgraph GraphQL Database Preparation

4.3.2.1 Schema Preparation

In GraphQL, everything and every class, which is called **type** in GraphQL, is input into its schema with the correct syntax and validation rules.

Taking advantage of the **enumeration types** of the GraphQL, IDType, StaffGrade, GearType, and CarCondition are created as an **enum** type, so that their values are restricted to a particular set.

```
enum IDType {
  NATIONALID
  PASSPORT
  DRIVINGID
}

enum StaffGrade {
  JUNIOR
  SEMI
  SENIOR
  DIRECTOR
}

enum CarCondition {
  FUNCTIONING
  NEEDCLEAN
  NEEDREPAIR
}

enum GearType {
  AUTO
  MANUAL
}
```

Figure 21: GraphQL Enums (IDType, StaffGrade, CarCondition, GearType)

Person is created as an interface, so that any type that implements this shall share the same basic information of a person.

```
interface Person {
  id: ID!
  identity: Identity
  firstName: String!
  lastName: String!
  sex: String
  dateOfBirth: DateTime!
  nationality: String
  phoneNumber: String!
  email: String
  permanentAddress: String
}
```

Figure 22: GraphQL interface Person

The other types such Identity, Client, Staff, PriceType, Car, and Rent are implemented as shown in the figures below.

```

type Client implements Person {
  id: ID!
  identity: Identity @id
  firstName: String!
  lastName: String!
  sex: String
  dateOfBirth: DateTime!
  nationality: String
  phoneNumber: String!
  email: String
  permanentAddress: String
  drivingID: Identity!
  registrationDate: DateTime
  currentAddress: String
}

type Identity {
  id: ID!
  idNumber: String! @id @search
  idType: IDType!
  validFrom: DateTime!
  expirationDate: DateTime!
}

type Staff implements Person {
  id: ID!
  identity: Identity @id
  firstName: String!
  lastName: String!
  sex: String
  dateOfBirth: DateTime!
  nationality: String
  phoneNumber: String!
  email: String
  permanentAddress: String
  currentAddress: String
  typeOfContract: String
  startDate: String
  grade: StaffGrade
}

```

Figure 23: GraphQL Types (Client, Identity, Staff)

```

type PriceType {
  id: ID!
  priceType: String! @id
  pricePerMinute: Float
  pricePerKilometer: Float
}

type Car {
  id: ID!
  model: String!
  color: String
  fuelType: String
  licenseNumber: String! @id
  gearType: GearType
  entryDate: DateTime
  priceType: PriceType!
  fuelAmount: Float
  currentLocation: String
  kilometerCounter: Float
  carCondition: CarCondition
  staff: Staff!
}

type Rent {
  id: ID!
  car: Car!
  client: Client!
  startTime: DateTime
  endTime: DateTime
  kilometerDriving: Float
  totalPrice: Float
}

```

Figure 24: GraphQL Types (PriceType, Car, Rent)

From these types above, a GraphQL file can be created and named as **schema.graphql**.

4.3.2.2 Schema Migration

As this research is developed on a MacBook, this Dgraph software is run using the standalone Docker image. Therefore, Docker installation and a few commands are needed to start the service and install the schema. However, Docker and Dgraph image installation will not be discussed in this section.

In order to start the Dgraph GraphQL, run the command line below: **docker run -it -p 8080:8080 dgraph/standalone:%VERSION_HERE**. The line of command will start the graphql service at localhost:8080/graphql.

After, the command line: `curl -X POST localhost:8080/admin/schema --data-binary '@schema.graphql'` is used to add and update the GraphQL schema.

With these two commands running successfully, schema preparation and migration for Dgraph can be considered ended.

4.4 Client-Side Application

Data inside the database is raw of which people find it difficult to make sense. The client-side application is developed in order to put those raw data together and combine them, so that they can provide a clear information to a client.

In this research, iOS mobile application is selected as the client-side application to fetch the data from both databases (MySQL, GraphQL) and display them. The application is responsible for displaying information about the staff, clients, cars, and rents.

In a security sense, the client-side application should never be authorised to directly communicate with the database. Usually, these applications communicate with each other via a middleman, the API. Therefore, in this section the API is also built along with the iOS Application.

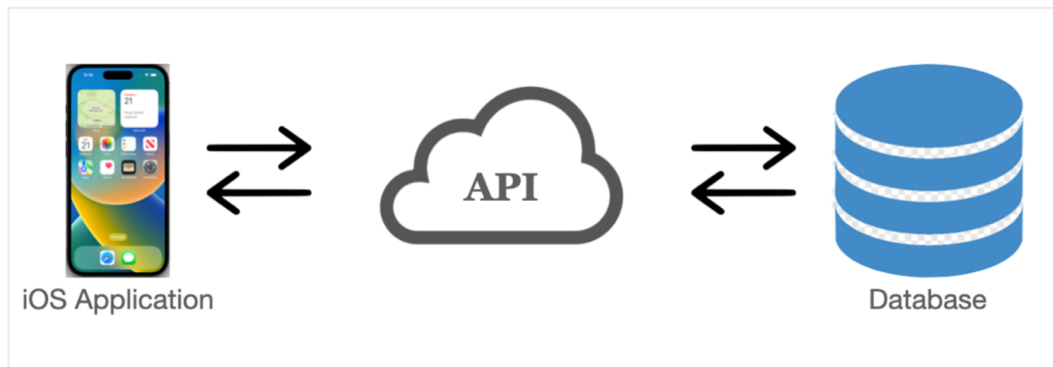


Figure 25: Architecture of Application

4.4.1 Vapor API

Vapor is a Swift web framework that enables user to create HTTP servers, backends, and APIs for web applications. Swift, a cutting-edge, powerful, and secure language that offers many advantages over many conventional server languages, is the language used in Vapor.

In this section, Vapor is used to create the API services for passing the data from MySQL database to the iOS client-side. As for Graph Database, Dgraph has already built-in

APIs that allows the client-side application to connect and fetch the data with a few URL request parameters and headers. Dgraph API will be explained in the next section 4.4.2.

There are various ways to install Vapor framework. However, this will not be discussed in this section. The installation process is elaborately described in the Vapor official website.

4.4.1.1 Workflow

There are 4 main and important tables in the current car renting project – Client, Staff, Car, and Rent. Client table stores all the information about clients using this service; Staff provides information about their employees; Car has the information about their distributed vehicles (cars); and Rent, the most important one, shows the details how this renting service is used by which client and on which car.

Figure 26 represents the expected API endpoints that this Vapor must provide to the applications. As presenting in the figure, there are 4 must-created APIs with the endpoints (*/clients*, */staff*, */cars*, and */rents*). Additionally, at the end of flow, each API shall return in a JSON format the data queried from the database.

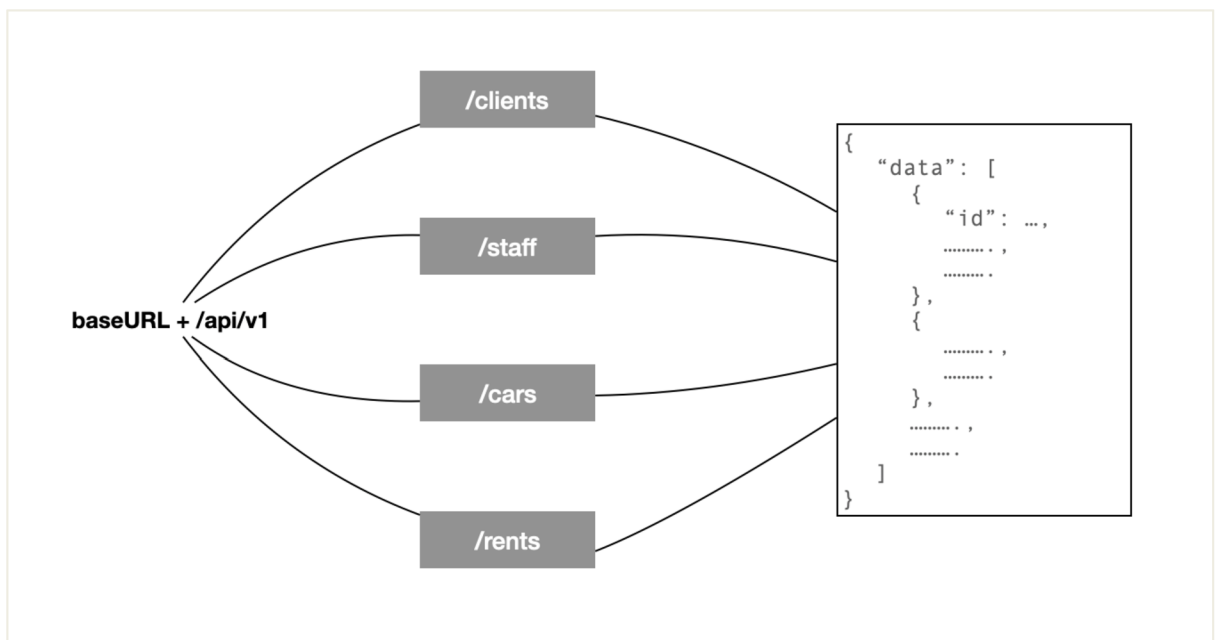


Figure 26: Vapor API Workflow

4.4.1.2 Vapor Implementation

Using **MySQLKit** dependency available for Vapor framework, it offers the possibility to configure itself to connect to MySQL database installed in the local machine or in the server. Figure 27 represents a piece of Swift code embedded with the MySQLKit library to create a configuration to connect to a local database named “car_renting_db” on port 3306 with “root” as username, “imaginar_password” as root password.

```
var tls = TLSConfiguration.makeClientConfiguration()
tls.certificateVerification = .none
let mysqlConf = MySQLConfiguration(
    hostname: "localhost",
    port: 3306,
    username: "root",           // MySQL username
    password: "imaginary_password", // Password of MySQL user
    database: "car_renting_db", // Database name
    tlsConfiguration: tls)
```

Figure 27: MySQLKit MySQL Configuration

4.4.1.2.1 MySQL Queries

Being as an API server, it provides the privilege to decide which data could be passed down to the client-side application. By limiting, it also helps the application to work faster and avoid unnecessary data to be sent. Therefore, before writing an API, the decision of which data should be passed to which endpoint must be made in advance.

- **Query for getting Clients**
 - **Information required:** Person (first name, last name, sex, date of birth, nationality, phone number, email, permanent address), Identity (number, type, valid from, expiration date), Client (id, registration date, current address), Diving License (number, valid from, expiration date)
 - SQL for getting all clients based on the decided information

```

select
  id.id_number, id.id_type, id.valid_from, id.expiration_date,
  p.first_name, p.last_name, p.sex, p.date_of_birth, p.nationality,
  p.phone_number, p.email, p.permanent_address,
  cl.id, cl.registration_date, cl.current_address,
  d.id_number as driving_license_number, d.valid_from as driving_license_valid_from,
  d.expiration_date as driving_license_expiration_date
from client cl
join person p on cl.person_id = p.id
join identity id on p.identity_id = id.id
join identity d on cl.driving_id = d.id

```

Figure 28: SQL query for all Clients

- **Query for getting Staff**

- **Information required:** Person (first name, last name, sex, date of birth, nationality, phone number, email, permanent address), Identity (number, type, valid from, expiration date), Staff (id, current address, type of contract, start date, grade)
- SQL for getting all staff based on the decided information

```

select
  id.id_number, id.id_type, id.valid_from, id.expiration_date,
  p.first_name, p.last_name, p.sex, p.date_of_birth, p.nationality,
  p.phone_number, p.email, p.permanent_address,
  st.id, st.current_address, st.type_of_contract, st.start_date, st.grade
from staff st
join person p on st.person_id = p.id
join identity id on p.identity_id = id.id

```

Figure 29: SQL query for all Staff

- **Query for getting Cars**

- **Information required:** Car (id, model, color, fuel type, license number, gear type, entry date, fuel amount, current location, kilometer counter, car condition), Price Type (price type, price per kilometer, price per minute), Staff (id, first name, last name)
- SQL for getting all staff based on the decided information

```

select
  car.id, car.model, car.color, car.fuel_type, car.license_number,
  car.gear_type, car.entry_date, car.fuel_amount, car.current_location,
  car.kilometer_counter, car.car_condition, car.staff_id,
  price_type.price_type, price_type.price_per_minute, price_type.price_per_kilometer,
  person.first_name as staff_first_name, person.last_name as staff_last_name
from car
join price_type on car.price_type = price_type.id
join staff on staff.id = car.staff_id
join person on person.id = staff.person_id

```

Figure 30: SQL query for all Cars

- **Query for getting Rents**

- **Information required:** Client (id, first name, last name), Car (id, model, color, license number), Price Type (price type, price per minute, price per kilometer), Rent (id, start time, end time, kilometer driving, total price)
- SQL for getting all staff based on the decided information

```

select
  cl.id as client_id,
  p.first_name as client_first_name, p.last_name as client_last_name,
  c.id as car_id, c.model as car_model, c.color as car_color,
  c.license_number as car_license_number,
  pt.price_type as car_price_type, pt.price_per_minute as car_price_per_minute,
  pt.price_per_kilometer as car_price_per_kilometer,
  r.id as rent_id, r.start_time, r.end_time, r.kilometer_driving, r.total_price
from rent r
join car c on r.car_id = c.id
join client cl on r.client_id = cl.id
join person p on cl.person_id = p.id
join price_type pt on c.price_type = pt.id

```

Figure 31: SQL query for all Rents

4.4.1.2.2 API Development

Logic of Vapor is written in its Controller. Based on the workflow, 4 controllers will be created (ClientController, StaffController, CarController, RentController). In order to create the endpoints such as `"/api/v1/clients"`, `"/api/v1/staff"`, `"/api/v1/cars"`, `"/api/v1/rents"`, there are

various ways; however, in this project, RouteCollection, the Vapor route protocol, will be used, therefore each controller will be extended from this protocol.

Figure 32 shows how each controller can extend the RouteCollection protocol. In the protocol (interface for other programming languages), there is a must-override function **boot(Routes: RoutesBuilder)** which tells the application which method to use in case a certain “endpoint” is passed. In this case, when an empty endpoint is passed to the route with GET HTTPMethod, a function named “getAll” will be utilised.

```
class ClientController: RouteCollection {
    func boot(routes: RoutesBuilder) throws {
        routes.get("", use: getAll)
    }
}

class StaffController: RouteCollection {
    func boot(routes: RoutesBuilder) throws {
        routes.get("", use: getAll)
    }
}

class CarController: RouteCollection {
    func boot(routes: RoutesBuilder) throws {
        routes.get("", use: getAll)
    }
}

class RentController: RouteCollection {
    func boot(routes: RoutesBuilder) throws {
        routes.get("", use: getAll)
    }
}
```

Figure 32: Controllers extend RouteCollection protocol

In order to allow these controllers to come into effect, it is a must to register these controllers in the “routes” file in the Vapor project. Figure 33 shows how to register the routes with the group “/api/v1”.

```
func routes(_ app: Application) throws {
    let apiRoutes = app.grouped("api", "v1")
    try apiRoutes.grouped("clients").register(collection: ClientController())
    try apiRoutes.grouped("staff").register(collection: StaffController())
    try apiRoutes.grouped("cars").register(collection: CarController())
    try apiRoutes.grouped("rents").register(collection: RentController())
}
```

Figure 33: Routes Registration

Before starting the query, the models of each class/table should be prepared in advance. Select query with MySQLKit will return an array of its built-in type **SQLRow** and has a decode method that allows the developer to map the value of SQLRow to the model that they want in just one line of code. However, it is required that the model is implemented the Swift protocol **Codable**, which actually is a type-alias of protocol **Encodable** and **Decodable**. Figure 34 below shows how a struct can implement the protocol Codable. One thing needs to be specified here is the **enum CodingKeys** which tells the protocol to know which struct property should be mapped with which “json_key”.

```
struct Rent: Codable {
    let clientId: String
    let clientFirstName: String
    let clientLastName: String

    let carId: String
    let carModel: String

    let rentId: String
    let startTime: Date
    let endTime: Date

    enum CodingKeys: String, CodingKey {
        case clientId = "client_id"
        case clientFirstName = "client_first_name"

        case carId = "car_id"
        case carModel = "car_model"

        case rentId = "rent_id"
        case startTime = "start_time"
        case endTime = "end_time"
    }
}
```

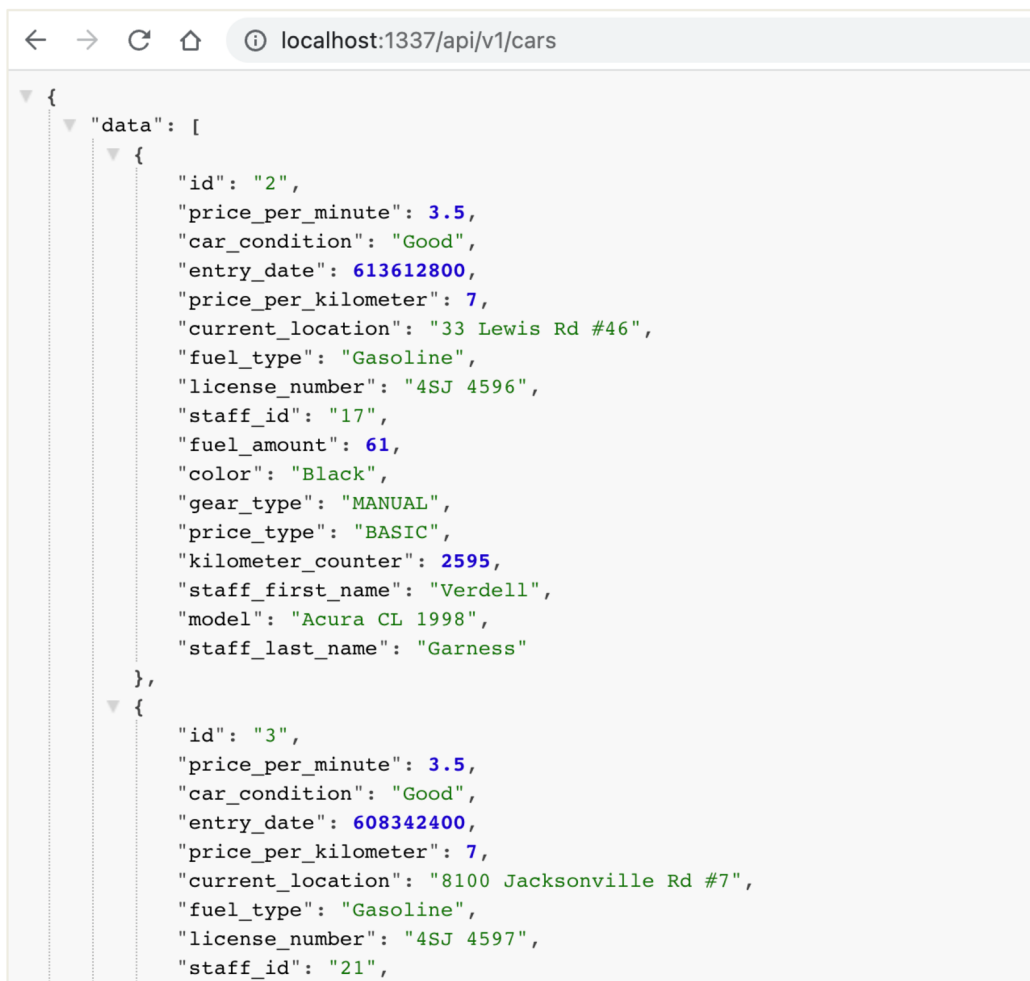
Figure 34: Sample of struct Rent implements Codable protocol

With the MySQL queries ready, the function “getAll” can be now. Figure 35 represents a piece of code that is used to fetch the data from the database with the prepared queries. “db” is an object of SQLiteDatabase getting from MySQLKit library. Figure 36 shows the result from this function.

```
// sql -> prepared query (carQuery, staffQuery, ...)
// MODEL_TYPE => Rent.self, Car.self, ...
func getAll(req: Request) async throws -> String {
    let queries = try db.raw("\(raw: sql)").all().wait()
    let models = try queries.map { try $0.decode(model: /*MODEL_TYPE*/) }
    return "{\"data\": [ \(try models.compactMap({ try convertToJson(val: $0) }).joined(separator: ",") )]}"
}

func convertToJson(val: String) throws -> String {
    let encoder = JSONEncoder()
    let data = try encoder.encode(val)
    guard let jsonString = String(data: data, encoding: .utf8) else {
        throw MySQLError.errorConvertJSON
    }
    return jsonString
}
```

Figure 35: getAll Method



```
localhost:1337/api/v1/cars
{
  "data": [
    {
      "id": "2",
      "price_per_minute": 3.5,
      "car_condition": "Good",
      "entry_date": 613612800,
      "price_per_kilometer": 7,
      "current_location": "33 Lewis Rd #46",
      "fuel_type": "Gasoline",
      "license_number": "4SJ 4596",
      "staff_id": "17",
      "fuel_amount": 61,
      "color": "Black",
      "gear_type": "MANUAL",
      "price_type": "BASIC",
      "kilometer_counter": 2595,
      "staff_first_name": "Verdell",
      "model": "Acura CL 1998",
      "staff_last_name": "Garness"
    },
    {
      "id": "3",
      "price_per_minute": 3.5,
      "car_condition": "Good",
      "entry_date": 608342400,
      "price_per_kilometer": 7,
      "current_location": "8100 Jacksonville Rd #7",
      "fuel_type": "Gasoline",
      "license_number": "4SJ 4597",
      "staff_id": "21",
    }
  ]
}
```

Figure 36: Sample Result of Cars

4.4.2 Dgraph API

Dgraph is a distributed GraphQL database with a graph backend that is horizontally scalable. It is designed for the intensive transactional workloads needed to run modern apps and websites, but it is not limited to only these types of applications. Since Dgraph is a native GraphQL database, sparse data set can be efficiently queried.

Running a self-managed Dgraph via the installation mentioned above (section 4.3.2), Dgraph provides the GraphQL API at */graphql*. Therefore, there would not be any time needed to spend on building a GraphQL API.

Upon schema migration, Dgraph creates two root types which are Query and Mutation. Based on the types specified in the schema, these two root types self-create fields that are necessary. Query will create fields contained **aggregate-** (aggregateCar, aggregateClient), **get-** (getCar, getClient), and **query-** (queryCar, queryClient). Fields in Query are used to fetch data. On the other hand, Mutation type will create fields contained **add-** (addCar, addClient), **delete-** (deleteCar, deleteClient), and **update-** (updateCar, updateClient). These fields are used to make changes in the database.

In this section, to match the APIs created in Vapor, only field **query** in Query type will be discussed. In order to use query the data from the Graph Database with Dgraph, the endpoint “/graphql” is used and a complete URL would be “localhost:port_number/graphql”. Using the **POST** HTTPMethod, the body of the request will be sent in the GraphQL format.

- GraphQL for getting Clients

The screenshot shows a GraphQL client interface with a POST request to `http://localhost:8080/graphql`. The query is as follows:

```

1 query {
2   queryClient {
3     id
4     identity {
5       id
6       idNumber
7       idType
8       validFrom
9       expirationDate
10    }
11    firstName
12    lastName
13    sex
14    dateOfBirth
15    nationality
16    phoneNumber
17    email
18    permanentAddress
19    drivingID {
20      id
21      idNumber
22      idType
23      validFrom
24      expirationDate
25    }
26    registrationDate
27    currentAddress
28  }
29 }
30

```

The response in the Preview pane is:

```

1 {
2   "data": {
3     "queryClient": [
4       {
5         "id": "0x17",
6         "identity": {
7           "id": "0x573",
8           "idNumber": "003126015",
9           "idType": "NATIONALID",
10          "validFrom": "2022-06-02T00:00:00Z",
11          "expirationDate": "2032-06-01T00:00:00Z"
12        },
13        "firstName": "Carmen",
14        "lastName": "Sweigard",
15        "sex": "M",
16        "dateOfBirth": "2002-06-10T00:00:00Z",
17        "nationality": "English",
18        "phoneNumber": "7329412621",
19        "email": "csweigard@sweigard.com",
20        "permanentAddress": "61304 N French Rd",
21        "drivingID": {
22          "id": "0x645",
23          "idNumber": "EP01231519",
24          "idType": "DRIVINGID",
25          "validFrom": "2017-12-13T00:00:00Z",
26          "expirationDate": "2032-12-12T00:00:00Z"
27        },
28        "registrationDate": "2022-04-11T00:00:00Z",
29        "currentAddress": "42744 Hamann Industrial Pky #82"
30      },
31    ]
32  }
33 }
34
35
36

```

Figure 37: GraphQL for Clients and Results

- GraphQL for getting Staff

The screenshot shows a GraphQL client interface with a POST request to `http://localhost:8080/graphql`. The query is as follows:

```

1 query {
2   queryStaff {
3     id
4     identity {
5       id
6       idNumber
7       idType
8       validFrom
9       expirationDate
10    }
11    firstName
12    lastName
13    sex
14    dateOfBirth
15    nationality
16    phoneNumber
17    email
18    permanentAddress
19    currentAddress
20    typeOfContract
21    startDate
22    grade
23  }
24 }
25

```

The response in the Preview pane is:

```

1 {
2   "data": {
3     "queryStaff": [
4       {
5         "id": "0x2711",
6         "identity": {
7           "id": "0x2722",
8           "idNumber": "001431981",
9           "idType": "NATIONALID",
10          "validFrom": "2020-05-07T00:00:00Z",
11          "expirationDate": "2030-05-06T00:00:00Z"
12        },
13        "firstName": "Vilma",
14        "lastName": "Berlanga",
15        "sex": "M",
16        "dateOfBirth": "1998-04-25T00:00:00Z",
17        "nationality": "German",
18        "phoneNumber": "6167373085",
19        "email": "vberlanga@berlanga.com",
20        "permanentAddress": "79 S Howell Ave",
21        "currentAddress": "6305 Elstow St",
22        "typeOfContract": "Full Time",
23        "startDate": "2019-08-01",
24        "grade": "DIRECTOR"
25      },
26    ],
27    {
28      "id": "0x2713",
29      "identity": {
30        "id": "0x2743",
31        "idNumber": "003215630",
32        "idType": "NATIONALID",
33        "validFrom": "2019-07-25T00:00:00Z",
34        "expirationDate": "2029-07-24T00:00:00Z"
35      },
36      "firstName": "Von",
37      "lastName": "Aprigliano",
38      "sex": "M",
39      "dateOfBirth": "2004-04-06T00:00:00Z",
40    }
41  ]
42 }
43
44
45

```

Figure 38: GraphQL for Staff and Results

- GraphQL for getting Cars

The screenshot shows a GraphQL IDE interface with a POST request to `http://localhost:8080/graphql`. The query is as follows:

```

1 query {
2   queryCar {
3     id
4     model
5     color
6     fuelType
7     licenseNumber
8     gearType
9     entryDate
10    priceType {
11      priceType
12      pricePerMinute
13      pricePerKilometer
14    }
15    fuelAmount
16    currentLocation
17    kilometerCounter
18    carCondition
19    staff {
20      id
21      identity {
22        id
23        idNumber
24        idType
25        validFrom
26        expirationDate
27      }
28      firstName
29      lastName
30      sex
31      dateOfBirth
32      nationality
33      phoneNumber
34      email
35      permanentAddress
36      currentAddress
37      typeOfContract
38      startDate
39      grade
40    }
41  }
42 }

```

The response is a JSON object:

```

1 {
2   "data": {
3     "queryCar": [
4       {
5         "id": "0x31ff",
6         "model": "Skoda Fabia 1999",
7         "color": "White",
8         "fuelType": "Diesel Fuel",
9         "licenseNumber": "4B0 9600",
10        "gearType": "MANUAL",
11        "entryDate": "2020-06-08T00:00:00Z",
12        "priceType": {
13          "priceType": "BASIC",
14          "pricePerMinute": 3.500000,
15          "pricePerKilometer": 7.000000
16        },
17        "fuelAmount": 74.000000,
18        "currentLocation": "34 Saint George Ave #2",
19        "kilometerCounter": 9983.000000,
20        "carCondition": "NEEDREPAIR",
21        "staff": {
22          "id": "0x271c",
23          "identity": {
24            "id": "0x2776",
25            "idNumber": "J048795535",
26            "idType": "PASSPORT",
27            "validFrom": "2021-04-16T00:00:00Z",
28            "expirationDate": "2031-04-15T00:00:00Z"
29          },
30          "firstName": "Verona",
31          "lastName": "Jobst",
32          "sex": "M",
33          "dateOfBirth": "1991-05-30T00:00:00Z",
34          "nationality": "German",
35          "phoneNumber": "5148427487",
36          "email": "verona_jobst@jobst.org",
37          "permanentAddress": "9041 Grand Plaza Plac",
38          "currentAddress": "33108 S Yosemite Ct",
39          "typeOfContract": "Full Time",
40          "startDate": "2019-10-02",
41          "grade": "SENIOR"
42        }
43      },
44      {
45        "id": "0x3200",
46        "model": "Toyota Camry 1996",

```

Figure 39: GraphQL for Cars and Results

- GraphQL for getting Rents

The screenshot shows a GraphQL IDE interface with a POST request to `http://localhost:8080/graphql`. The query is as follows:

```

1 query {
2   queryRent {
3     id startTime endTime kilometerDriving totalPrice
4   }
5   car { id model color fuelType licenseNumber
6     gearType entryDate
7     priceType { priceType pricePerMinute pricePerKilometer }
8     fuelAmount currentLocation kilometerCounter carCondition
9   }
10  staff { id
11    identity { id idNumber idType validFrom expirationDate }
12    firstName lastName sex dateOfBirth nationality
13    phoneNumber email permanentAddress currentAddress
14    typeOfContract startDate grade
15  }
16 }
17 }
18 client {
19   id
20   identity { id idNumber idType validFrom expirationDate }
21   firstName lastName sex dateOfBirth nationality
22   phoneNumber email permanentAddress
23   drivingID { id idNumber idType validFrom expirationDate }
24   registrationDate currentAddress
25 }
26 }
27 }

```

Figure 40: GraphQL for Rents

```

Preview ▾ Headers 9 Cookies Timeline
1 {
2   "data": {
3     "queryRent": [
4       {
5         "id": "0x6313",
6         "startTime": "2022-01-01T17:50:00Z",
7         "endTime": "2022-01-01T19:19:00Z",
8         "kilometerDriving": 18.000000,
9         "totalPrice": 440.540000,
10        "car": {
11          "id": "0x321e",
12          "model": "Toyota Camry 1996",
13          "color": "Red",
14          "fuelType": "Diesel Fuel",
15          "licenseNumber": "4S5 0242",
16          "gearType": "MANUAL",
17          "entryDate": "2019-03-25T00:00:00Z",
18          "priceType": {
19            "priceType": "BASIC",
20            "pricePerMinute": 3.500000,
21            "pricePerKilometer": 7.000000
22          },
23          "fuelAmount": 76.000000,
24          "currentLocation": "57 Haven Ave #90",
25          "kilometerCounter": 11856.000000,
26          "carCondition": "FUNCTIONING",
27          "staff": {
28            "id": "0x2754",
29            "identity": {
30              "id": "0x272e",
31              "idNumber": "J048795523",
32              "idType": "PASSPORT",
33              "validFrom": "2018-11-14T00:00:00Z",
34              "expirationDate": "2028-11-13T00:00:00Z"
35            },
36            "firstName": "Viola",
37            "lastName": "Eddens",
38            "sex": "M",
39            "dateOfBirth": "1996-03-20T00:00:00Z",
40            "nationality": "Irish",
41            "phoneNumber": "5067723108",
42            "email": "veddens@eddens.org",
43            "permanentAddress": "51 S Hulén St",
44            "currentAddress": "9892 Hernando W",
45            "typeOfContract": "Full Time",
46            "startDate": "2019-07-28",
47            "grade": "DIRECTOR"
48          }
49        }
50      }
51    ]
52  }
53 }

```

Figure 41: Results for GraphQL Rent

4.4.3 iOS Application

As mentioned above, the client-side application is built on iOS mobile application, using Swift5 as the programming language and XCode as the application. And with the latest technology of iOS development, SwiftUI, which has been introduced as a declarative UI framework over the legacy Storyboard, will be used.



Figure 42: Swift5, IDE XCode14, and SwiftUI

4.4.3.1 Workflow

This mobile application is developed for the business owners, investors, authorised staff as intended users. As it is not intended for public use, the application is lack of functionalities such as login, registration, car availability, rent, etc.

Figure 43 represents the flow of the application focus on viewing the final results of the car renting services with a clear distinction of which database is being used for comparison.

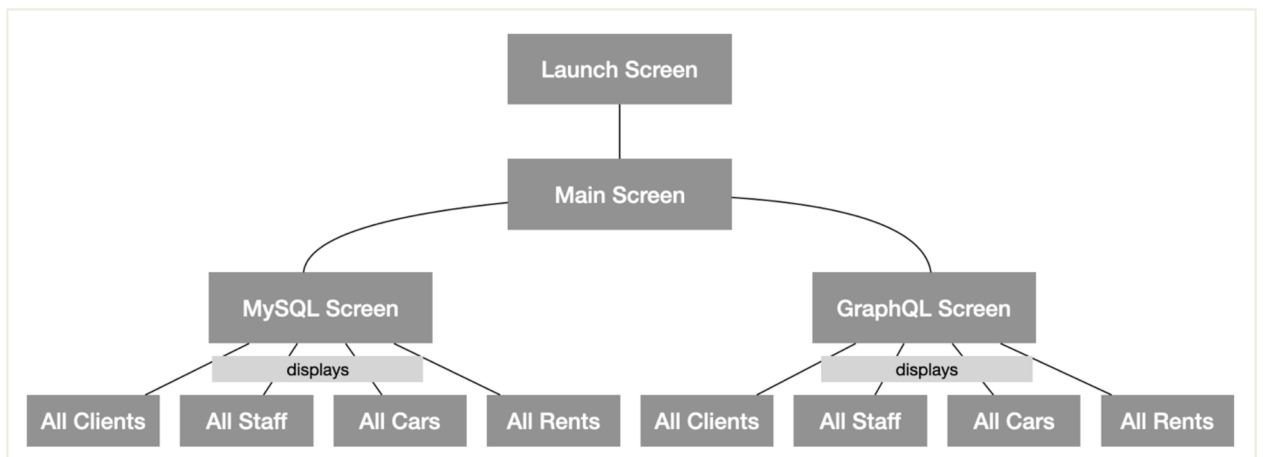


Figure 43: iOS Application Workflow

4.4.3.2 Screens

4.4.3.2.1 Launch Screen

A screen that appears shortly and then disappears at the start of almost all mobile applications. Most of the time, it shows the logo and the name of the application and/or with a little bit description about the application.

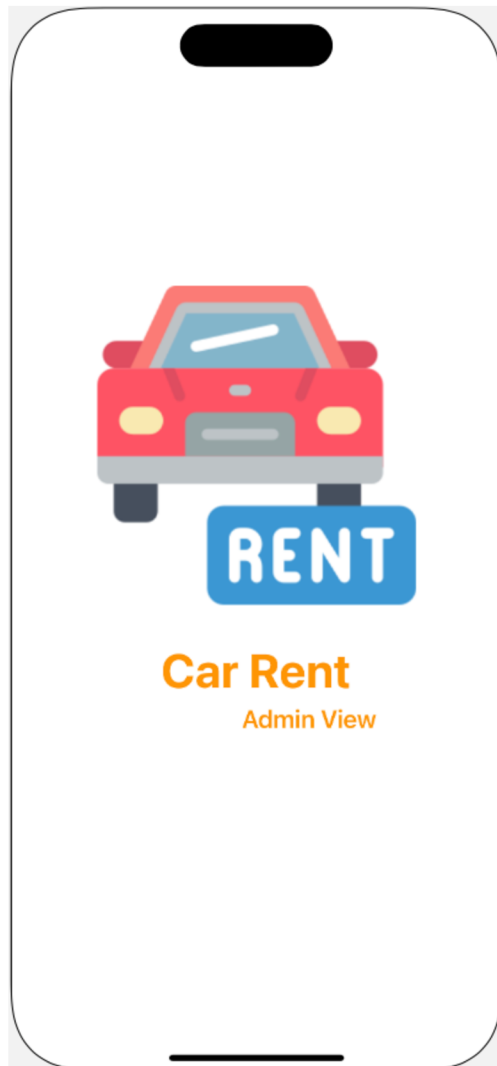


Figure 44: Car Rent (Admin View) Launch Screen

Figure 44 shows the Car Rent Launch Screen. The screen is embedded with an image of a car rent representing as the logo of the company or the service. Below the image, there is a title text specifying the name and

4.4.3.2.2 Main Screen

On Main Screen, there are two options for a user to choose.

1. Car Rent with MySQL: to display the list of all necessary information being stored in Relational Database using MySQL to query
2. Car Rent with GraphQL: to display the list of all necessary information that is being stored in the Graph Database using GraphQL to query

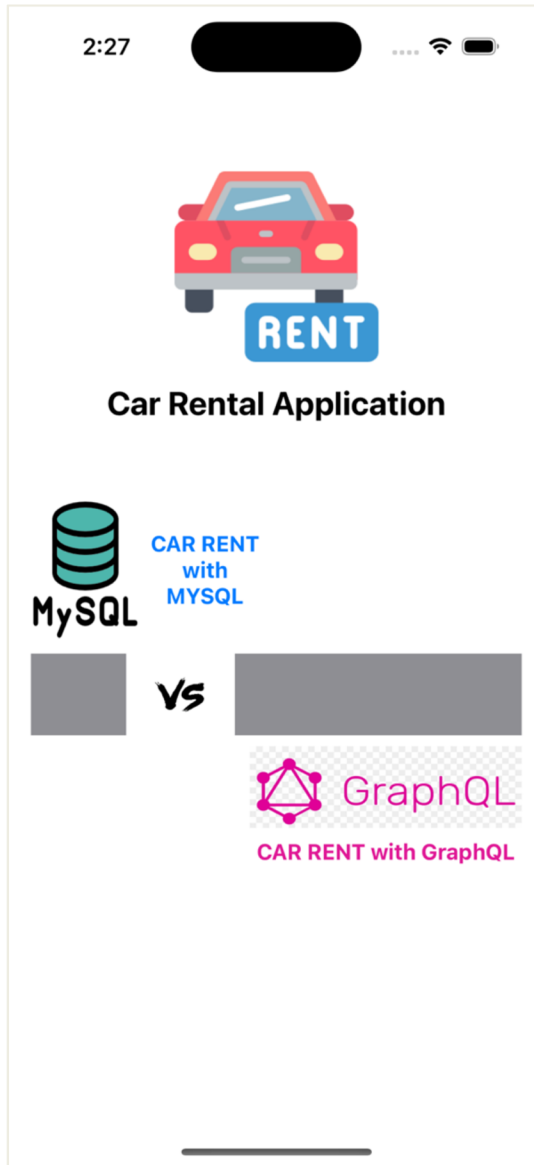


Figure 45: Car Rent (Admin View) Main Screen

4.4.3.2.3 Display Screen

From the Main Screen, although there are two options, there is only one Display Screen. The title of the display screen (MySQL or GraphQL) is differentiated based on the selection of the user on the Main Screen.

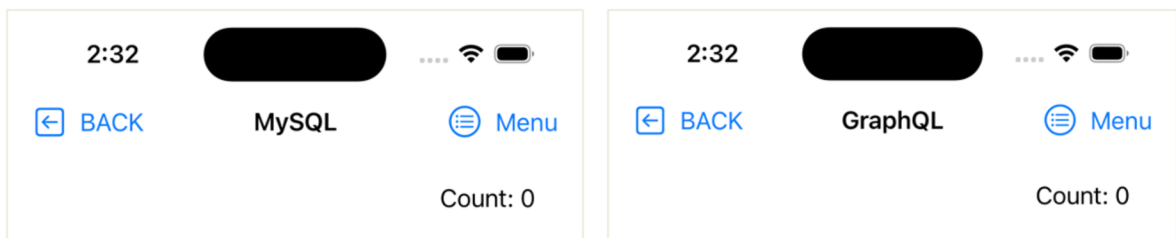


Figure 46: Display Screen for MySQL and GraphQL

On the top left of the Display Screen, the button “BACK” serves as an action to return to the Main Screen which is the previous screen. On the top right of the screen, the button “Menu” allows user to select one from the 4 existing menu items to display the data that they want to view.

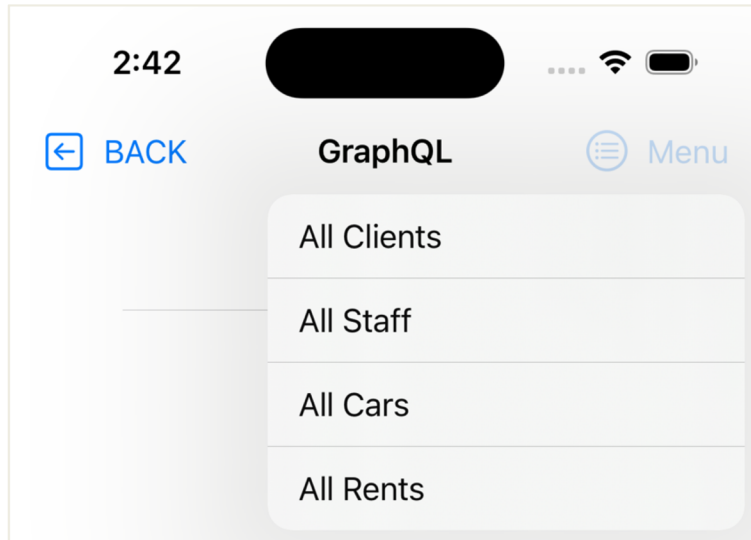


Figure 47: Car Rent (Admin View) Menu Clicked

In order to be able to the fetch data from the API, some codes are needed to do the request. The app will use pure Swift programming language without embedded any third-party library. Figures 48 and 49 below are two created functions in the **Network** class to be used to request the data API.

```
static func request(path: String, completion: @escaping ([[String:Any]]?, Error?) -> Void) {  
  
    let urlSession = URLSession(configuration: .default)  
    let urlString = "http://localhost:1337/api/v1/" + path  
  
    guard let url = URL(string: urlString) else {  
        print("Fail to parse URL string")  
        return  
    }  
  
    let dataTask = urlSession.dataTask(with: url) { (data, response, error) in  
        if let error = error {  
            print("Error: \(error.localizedDescription)")  
            completion(nil, error)  
            return  
        }  
  
        if let data = data {  
            do {  
                let json = try JSONSerialization.jsonObject(with: data, options: .allowFragments) as! [String: Any]  
                completion(json["data"] as? [[String:Any]], nil)  
            } catch {  
                print("Error parse: \(error.localizedDescription)")  
            }  
        }  
    }  
  
    dataTask.resume()  
}
```

Figure 48: Network Request for Relational Database

Network Request for Relational Database takes paths such as *clients*, *rents*, *cars*, and *staff* – which are the URL paths that are created in the Vapor API. The function is written using Swift closure that is typically used for asynchronous request. Upon completion, the function returns either a list of key-value of the result getting from the API or an error.

```

static func callGAPI(queryBody: String, completion: @escaping ([String:Any]?, Error?) -> Void) {
    let urlSession = URLSession(configuration: .default)
    guard let url = URL(string: "http://localhost:8080/graphql") else {
        print("Fail to parse URL string")
        return
    }

    var request = URLRequest(url: url)
    request.httpMethod = "POST"
    request.httpBody = "query { \(queryBody) }".data(using: .utf8)
    request.addValue("application/graphql", forHTTPHeaderField: "Content-Type")

    let dataTask = urlSession.dataTask(with: request) { (data, response, error) in
        if let error = error {
            print("Error: \(error.localizedDescription)")
            completion(nil, error)
            return
        }

        if let data = data {
            do {
                let json = try JSONSerialization.jsonObject(with: data, options: .allowFragments) as! [String: Any]
                let query = (json["data"] as? [String: Any])
                completion(query, nil)
            } catch {
                print("Error parse: \(error.localizedDescription)")
            }
        }
    }

    dataTask.resume()
}

```

Figure 49: Network Request for Graph Database

Request for Graph Database is more special. The method takes string queryBody in form of GraphQL as a parameter. The request is made with POST HTTP method and the queryBody as the HTTP body. The function also returns a list of key-value as the result upon completion.

```

struct GRent: GModel, Hashable {
    let id: String
    let car: GCar
    let client: GClient
    let startTime: String
    let endTime: String
    let kilometerDriving: CGFloat
    let totalPrice: CGFloat

    static var gQuery: String {
        return "queryRent { id startTime endTime kilometerDriving totalPrice car { id model color fuelType licenseNumber gearType entryDate priceType { priceType pricePerMinute pricePerKilometer } fuelAmount currentLocation kilometerCounter carCondition staff { id identity { id idNumber idType validFrom expirationDate } firstName lastName sex dateOfBirth nationality phoneNumber email permanentAddress currentAddress typeOfContract startDate grade } } client { id identity { id idNumber idType validFrom expirationDate } firstName lastName sex dateOfBirth nationality phoneNumber email permanentAddress drivingID { id idNumber idType validFrom expirationDate } registrationDate currentAddress } }"
    }
}

```

Figure 50: Client-Side Rent Struct for Graph Database

```

struct Rent: Model, Hashable {
  let clientId: String
  let clientFirstName: String
  let clientLastName: String

  let carId: String
  let carModel: String
  let carColor: String
  let carLicenseNumber: String

  let carPriceType: String
  let carPricePerMinute: CGFloat
  let carPricePerKilometer: CGFloat

  let rentId: String
  let startTime: Date
  let endTime: Date
  let kilometerDriving: CGFloat
  let totalPrice: CGFloat
}

extension Rent {
  enum CodingKeys: String, CodingKey {
    case clientId = "client_id"
    case clientFirstName = "client_first_name"
    case clientLastName = "client_last_name"

    case carId = "car_id"
    case carModel = "car_model"
    case carColor = "car_color"
    case carLicenseNumber = "car_license_number"

    case carPriceType = "car_price_type"
    case carPricePerMinute = "car_price_per_minute"
    case carPricePerKilometer = "car_price_per_kilometer"

    case rentId = "rent_id"
    case startTime = "start_time"
    case endTime = "end_time"
    case kilometerDriving = "kilometer_driving"
    case totalPrice = "total_price"
  }
}

```

Figure 51: Client-Side Rent Struct for Relational Database

Figures 50 and 51 are example of Rent structs for the APIs of Graph and Relational. For GRent, the fields of other structs (classes) related to it are included as an object. Since the fields in Graph schema are conformed to camel case convention, this helps in API data mapping in iOS without having needs to specify the key as shown in the Rent struct for Relational database.

```

private func fetchRents() {
    Network.callGAPI(queryBody: GRent.gQuery) { (dictArr, error) in
        if error != nil {
            print("Error fetching gRents")
            return
        }

        guard let dictArr = dictArr,
              let queries = dictArr["queryRent"] as? [[String: Any]] else { return }

        var models = [GRent]()
        queries.forEach {
            do {
                let data = try JSONSerialization.data(withJSONObject: $0, options: .prettyPrinted)
                let model = try JSONDecoder().decode(GRent.self, from: data)
                models.append(model)
            } catch {
                print("Error: cannot decode json")
            }
        }
        DispatchQueue.main.async {
            self.rents = models
            self.rowCounts = models.count
            self.showLoading = false
        }
    }
}

```

Figure 52: Client-Side Rent Struct for Graph Database

```

private func fetchRents() {
    Network.request(path: MySQLURLPath.rents) { (jsonArr, error) in
        if let error = error {
            print("Error fetch \((error.localizedDescription)")
            return
        }

        guard let jsonArr = jsonArr else { return }

        var models = [Rent]()
        jsonArr.forEach {
            do {
                let data = try JSONSerialization.data(withJSONObject: $0, options: .prettyPrinted)
                let model = try JSONDecoder().decode(Rent.self, from: data)
                models.append(model)
            } catch {
                print("Error: cannot decode json")
            }
        }
        DispatchQueue.main.async {
            self.rents = models
            self.rowCounts = models.count
            self.showLoading = false
        }
    }
}

```

Figure 53: Client-Side Rent Struct for Relational Database

Figure 52 and 53 shows a piece of code that is used to fetch the Rent data both API (Graph and Relational). From the key-value (JSON) results received, those will be converted to the Rent struct of each database respectively. The values will then be used to display in the UI to the user.

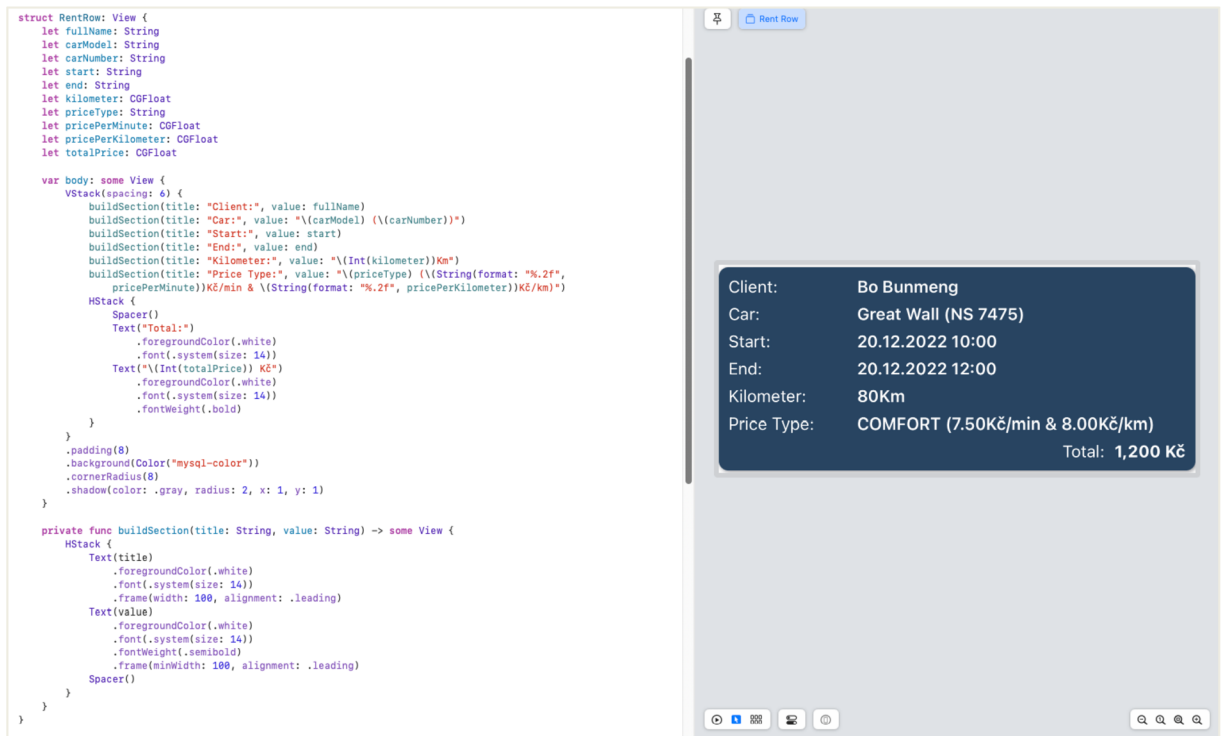


Figure 54: List item for displaying Rent information

Figure 54 shows the implementation in SwiftUI in order to create a sample list item for displaying Rent information. The view at the right side is called canvas, which makes the UI development in SwiftUI much easier. As being shown, the list item of Rent will take in information such as full name of client, car model with car license number, start time and end time of renting, kilometres driven, car price type with price per minute and price per kilometre, and the total cost of that rent.

```
private func buildRentListView() -> some View {
    List(viewModel.rents, id: \.self) { rent in
        RentRow(fullName: "\(\rent.client.firstName) \(\rent.client.lastName)", carModel:
            rent.car.model, carNumber: rent.car.licenseNumber, start:
            rent.displayGDateStr(rent.startTime), end: rent.displayGDateStr(rent.endTime),
            kilometer: rent.kilometerDriving, priceType: rent.car.priceType.priceType,
            pricePerMinute: rent.car.priceType.pricePerMinute, pricePerKilometer:
            rent.car.priceType.pricePerMinute, totalPrice: rent.totalPrice)
    }.listStyle(.plain)
}
```

Figure 55: SwiftUI build Rent list view

Figure 55 shows a piece of code that is used for building the list view to display Rent information upon user's selection on the menu item. This function returns the SwiftUI built-in List using the list of Rent objects that are obtained after the data fetching from the API. For

each Rent object received, the value is passed to the RentRow, which is the list item created specifically to serve as the UI for display Rent information (mentioned above).

Figures below (Figure 56, Figure 57, Figure 58, Figure 59) display the various data when user clicks on each menu. Above list, there is a title text specifying which tables/classes user is viewing with the number of items displayed at the top right of the list.

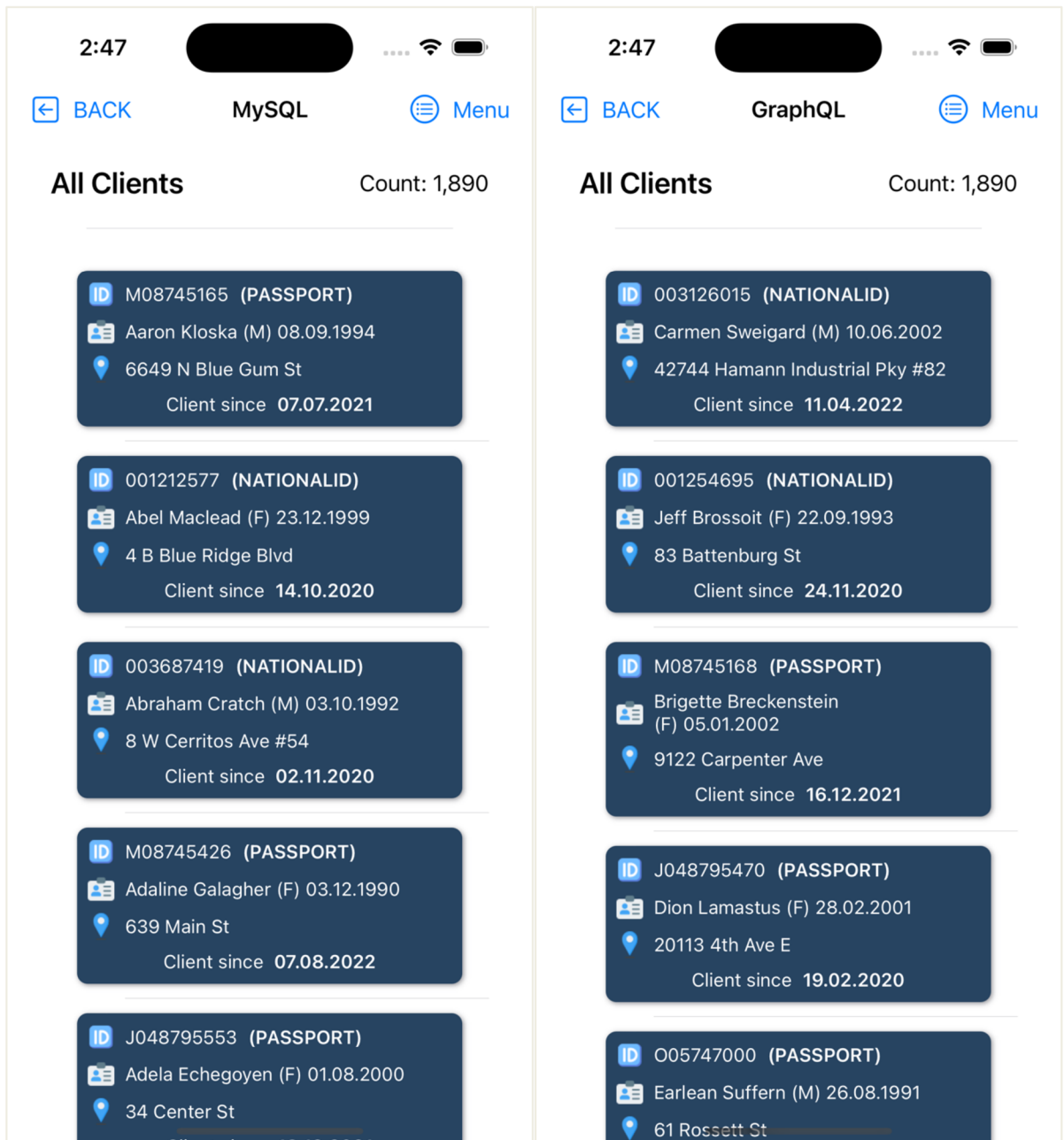


Figure 56: Car Rent (Admin View) Display All Clients



Figure 57: Car Rent (Admin View) Display All Staff



Figure 58: Car Rent (Admin View) Display All Cars

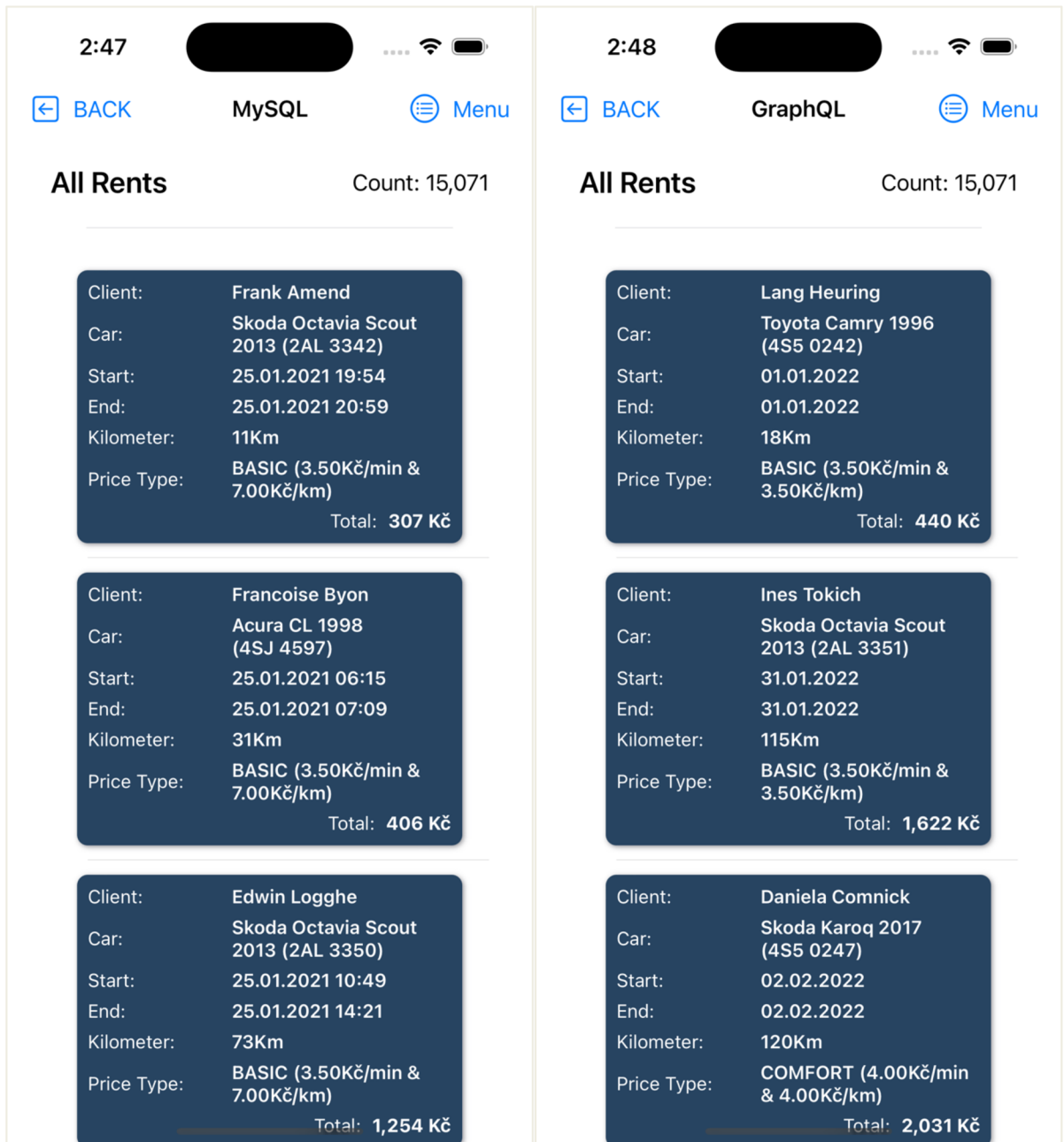


Figure 59: Car Rent (Admin View) Display All Rents

5. Results and Discussion

Based on the whole practical part from creating database to applications development (client-side, Vapor API), relational and graph database has its own strengths and weaknesses. The evaluations for comparing these two databases will be made based on source lines of code, complexity of queries, development time consumption, and database collaboration complexity. The evaluation will conclude between these two databases which one is more developer friendly.

- **Source Lines of Code**

Source lines of code (SLOC), also referred to as lines of code (LOC), is a software metric that counts the lines in the source code of a computer program to determine the size of the program. SLOC is often used to assess programming productivity or maintainability after the software is created, as well as to predict the amount of effort that will be needed to develop a program.

There are two major types of SLOC measures: physical source lines of code (LOC) and logical source lines of code (LLOC).

In this comparison, two projects are considered: the first one is the API application, and the last one is the mobile application.

- **Complexity of Queries**

In database, query is one of the most important parts. A query might ask a database for data results, a specific action to be taken with the data, or both. A query can be add, alter, or remove data from a database, conduct computations, integrate data from other databases, and answer simple questions (Indeed.com, 2021).

As Relational database uses SQL and Graph database uses GraphQL, these are two different query languages, which mean they both have different syntax. As the car renting project is intended for admin view, only the queries for fetching the data are used.

- **Collaboration Complexity**

The relational database using MySQL and the graph database with Dgraph GraphQL are two independent software. With Swift as a programming language at the client-side, it requires to find a way to connect the database to the client-side. The connection can be made

through API or direct connection. The comparison will be made based on how resourceful it is to make connection between MySQL and iOS and between GraphQL and iOS

- **Development Efficiency**

There are multiple criteria that can be used to consider a development is efficient. However, in this comparison, the efficiency of development will focus on how efficiently the application can receive the data and use them in the application.

Comparison

Criteria	MySQL	GraphQL
Source Lines of Code	API (Vapor) SLOC: 722 Application SLOC: 1,135 Total: 1,857 MySQL has more lines as code as it requires two applications: Vapor API and iOS.	API SLOC: <i>N/A</i> Application SLOC: 1,060 Total: 1,060
Complexity of Queries	In order to do queries with MySQL, relationships between tables need to be taken into consideration, the fields need to be clearly specified in order to tell the database to which table they belong, and the connection between foreign key of the main table to the primary key of the related tables must be done correctly. <pre> select <i>important_fields</i> from <i>table</i> join <i>table1</i> on (<i>foreign_key</i>) join <i>table2</i> on (<i>foreing_key</i>) </pre>	Query with GraphQL requires the body to be wrapped by the query at the front and the query[Type] after. <pre> query { queryClient { <i>body</i> } } </pre> “ <i>body</i> ” contains the field of the Graph Type that need to be retrieved

	
Collaboration Complexity	Mobile Application needs to have a developed API application to be able to fetch the data from MySQL database. Therefore, in order to allow the collaboration between MySQL and iOS app, a Vapor API application is created. Although it is required to have another project to allow the collaboration between these two, there are fortunately abundant frameworks for connecting an API application to MySQL.	Dgraph GraphQL provides sufficient built-in APIs that allow developers to use it in the mobile application to retrieve the data immediately right after the data has been inserted without having a need to create them from scratch. It also provides the possibility to create additional APIs to match the requirements of the application with a simple POST request.
Development Efficiency	Query in MySQL does the transaction line by line. The data returned from MySQL is a list of rows. Therefore, in order to get display this data, the mobile application needs to first read one row and then map the values to appropriate class/struct members. This process is repeated N times of rows.	Query in GraphQL returns a JSON string with all the values ready to be mapped to appropriate class/struct members and display at the client-side. Therefore, it is more efficient and less time consuming for the mobile application.

Table 7: MySQL and GraphQL Evaluation

Based on the evaluation made in the table, it can be seen that for the Car Rental application at Admin Site, the graph database with Dgraph GraphQL is more efficient and time saving, which helps reduce the workload of the developers involved in the project, in comparison to relational database with MySQL.

6. Conclusion

Relational Database and Graph Database have their advantages and disadvantages. Based on the project Car Rental alone, it cannot be used to determine which type of database is the best or better than which. It is advisable to take full consideration and weigh the positive and the negative sides of each database before choosing one.

Before selecting a database for any project, from physical databases to cloud solutions, Silnitsky (2021) mentioned about some criteria that people need to consider:

- **Query pattern:** How intricate are your search patterns? Do you require key retrieval alone or do you additionally need a variety of other parameters? Do the data also require fuzzy search?

If fetching data by key is required, then a key-value store is needed. (e.g. S3, Redis, DynamoDB). If the query is used to get many different fields, Relational Database (e.g. MySQL, PostgreSQL) or Document Database (e.g. MySQL, MongoDB, CouchDB) would be ideal. Lastly, if the fuzzy search query capabilities are the case, then it is advisable to use search engines like Elasticsearch and Solr.

- **Consistency:** Is strong consistency (read after write, especially when you transfer writes to a new data center) necessary, or is eventual consistency also acceptable?

A relational database, such as MySQL or PostgreSQL, is typically more suited for strong consistency requirements than a document database, such as MongoDB or CouchDB.

- **Storage Capacity:** How much storage is needed?

Most database systems struggle with performance as the number of Nodes and Shards increases into the hundreds (e.g. Elasticsearch) or are by the amount of disk space (such as MySQL). Therefore if infinite storage is the target, then cloud storage would be the best choice. Data can be store as much as you want using object storage services like S3 or GCS.

- **Performance:** What throughput and latency are required?

Solutions from cloud providers like Amazon's DynamoDB and Google's Bigtable may be the perfect fit if you need very low latency and huge traffic. However, the price is obviously a drawback.

- **Maturity and Stability:** How much experience does your DBA team have with self-hosted deployment, and how advanced is the technology?

It may be tempting to self-host the most popular, robust, and feature-rich database, but if your organisation lacks experience with it, you might come to regret it.

Database setup, configuration, and fine tuning is a time-consuming and dangerous process. When it comes to production consistency, sometimes going with the “old” organisation self-hosted workhorse will yield more long-term rewards.

- **Cost:** What are the costs if you choose a managed cloud solution? Which restrictions apply to it?

Typically, read/write traffic is inversely proportional to the cost of managed cloud solutions. Ensure that each managed solution is cost-effective for your unique read/write consumption patterns by carefully reading the fine print.

On the query language side, MySQL and GraphQL, on the other hand, are both important for their respective databases. It is undeniable for the fact that everybody must be feeling more comfortable with MySQL (SQL) as it has been on its fame for a long time and has been used for many legacy projects. Nevertheless, it is advisable to keep oneself up to date with latest technologies.

7. References

1. “How Much Data Is Created Every Day in 2022?” Techjury. Accessed November 20, 2022. <https://techjury.net/blog/how-much-data-is-created-every-day/>.
2. “What Are the Different Types of Databases? | Indeed.com.” Accessed November 20, 2022. <https://www.indeed.com/career-advice/career-development/types-of-databases>.
3. “What Is a Relational Database?” Oracle. Accessed November 21, 2022. <https://www.oracle.com/database/what-is-a-relational-database/>.
4. Codd, Edgar. F. “A Relational Model of Data for Large Shared Data Banks.” *Communications of the ACM* 13, no. 6 (1970): 377–87. <https://doi.org/10.1145/362384.362685>.
5. Quickbase. “A Timeline of Database History.” Quickbase. Accessed November 27, 2022. <https://www.quickbase.com/articles/timeline-of-database-history>.
6. Davidson, Louis. *Pro SQL Server Relational Database Design and Implementation: Best Practices for Scalability and Performance*. Berkeley, CA: Apress, 2021.
7. Harrington, Jan L. *Relational Database Design and Implementation Clearly Explained*. Amsterdam: Morgan Kaufmann/Elsevier, 2009.
8. Date, C.J. *Database Design and Relational Theory Normal Forms and All That Jazz*. Sebastopol, Calif: O'Reilly, 2012.
9. Yaowen, Chen. “Comparison of Graph Databases and Relational Databases When Handling Large-Scale Social Data,” 2016.
10. Codd, Edgar F. “Further Normalization of the Data Base Relational Model,” 1972.
11. Haerder, Theo, and Andreas Reuter. “Principles of Transaction-Oriented Database Recovery.” *ACM Computing Surveys* 15, no. 4 (1983): 287–317. <https://doi.org/10.1145/289.291>.
12. Robinson, Ian, Emil Eifrem, and Jim Webber. *Graph Databases: New Opportunities for Connected Data*. Sebastopol, CA: O' Reilly, 2015.
13. Harrison, Guy. *Next Generation Databases: Nosql, NewSQL, and Big Data*. New York: Apress, 2015.
14. Jing Han, Haihong E, Guan Le, and Jian Du. “Survey on NoSQL Database.” *2011 6th International Conference on Pervasive Computing and Applications*, 2011. <https://doi.org/10.1109/icpca.2011.6106531>.
15. Kristi, Berg, Seymour Tom, and Goel Richa. “History Of Databases,” December 31, 2012.

16. Vaish, Gaurav. *Getting Started with Nosql Your Guide to the World and Technology of Nosql*. Birmingham: Packt Publishing, 2013.
17. Garcia-Molina, Hector, Jeffrey David Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Harlow: Pearson, 2014.
18. Daniel, Abadi, Boncz Peter, and Harizopoulos Stavros. "Column Oriented Database Systems," August 1, 2009.
19. Luke. "Graph Database Fundamentals - RDF, Property Graph, Linked-Data." TerminusDB, October 14, 2022. <https://terminusdb.com/blog/graph-database-fundamentals/>.
20. Hayes, Jonathan. "A Graph Model for RDF," 2004.
21. Powers, Shelley. *Practical RDF: Solving Problems with the Resource Description Framework*. Sebastopol (California): O'Reilly, 2003.
22. Berners-Lee, Tim, James Hendler, and Ora Lassila. "The Semantic Web." *Scientific American* 284, no. 5 (2001): 34–43. <https://doi.org/10.1038/scientificamerican0501-34>.
23. Salon, Pamela. "A Brief History of the Car Rental Industry." LinkedIn, May 16, 2022. <https://www.linkedin.com/pulse/brief-history-car-rental-industry-pamela-salon/>.
24. "How to Start a Car Rental Business [Updated 2022]." Growththink, November 22, 2022. <https://www.growththink.com/businessplan/help-center/how-to-start-a-car-rental-business>.
25. "What Is a Mobile Application? - Definition from Techopedia." Techopedia.com. Accessed November 25, 2022. <https://www.techopedia.com/definition/2953/mobile-application-mobile-app>.
26. Kenton, Will. "Apple IOS." Investopedia. Investopedia, March 24, 2022. <https://www.investopedia.com/terms/a/apple-ios.asp>.
27. "Apple Statistics (2022)." Business of Apps, October 28, 2022. <https://www.businessofapps.com/data/apple-statistics/>.
28. By: IBM Cloud Education. "IOS App Development." IBM. Accessed November 27, 2022. <https://www.ibm.com/cloud/learn/ios-app-development-explained>.
29. Vettrivel, Vishnu, and Name *. "Knowledge Graphs: RDF or Property Graphs, Which One Should You Pick?" Wisecube AI. Accessed November 27, 2022. <https://www.wisecube.ai/blog/knowledge-graphs-rdf-or-property-graphs-which-one-should-you-pick>.
30. Community, Vapor. "Welcome." Vapor Docs. Accessed November 27, 2022. <https://docs.vapor.codes/>.

31. Silnitsky, Natan. "How to Choose the Right Database for Your Service." Medium. Wix Engineering, August 27, 2022. <https://medium.com/wix-engineering/how-to-choose-the-right-database-for-your-service-97b1670c5632>.