TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Shamchi Hoque Kaify 195667IASM

Analysis of Using Different Databases and integration of SQL and NOSQL database

Master's thesis

Supervisor: Vladimir Viies
Associate Professor

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Shamchi Hoque Kaify 195667IASM

Erinevate andmebaaside kasutamise võimaluste analüüs ja SQL ning NOSQL ühildamine

Magistritöö

Juhendaja: Vladimir Viies
Dotsent

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Shamchi Hoque Kaify

20.12.2021

MASTER`S THESIS TASK SPECIFICATION

Date: 25.11.2021

Student name: Shamchi Hoque Kaify
Student code: 195667IASM

Topic: Analysis of using different databases and integration of SQL and NOSQL database

Topic background: With the fast advancement of technology, interaction between application and user increased. Also, the need for faster interaction between the user and application increased as well. For this increasing demand and trend of object-oriented programming database connection and selection of appropriate database has become one of the important research topics. In this thesis, we will analyse different types of databases, different methods of database connection and integration. We will also develop a Smart Prescription System on top of two different database integration methods: polyglot persistence and multi-model database and evaluate their performance.

Supervisor: Vladimir Viies

Additional specifications: By the end of this project, it is expected that an application would be developed, and we can see the performance of each kind of database integration.

Issues to be resolved according to the following learning outcomes:
A) System aspects:
   i. Analysis of different types of databases and methods of database connection.
   ii. Analysis of different methods of database integration.
   iii. Implementation of two different methods and compare their performance
B) Software aspects:
   i. Web application

Student’s signature:
Shamchi Hoque Kaify

Abstract

The uprising of the Internet of Things (IoT) has led to an exponential increase in volume, velocity, and variety of data. This big data which needs to be handled is affecting the performance of databases. Until recently, the relational database was the dominant database for working with structured data, but sometimes it seems inefficient to handle large unstructured data. Nowadays, NoSQL databases are becoming a more popular solution for handling unstructured data. As some core features like ACID properties are compromised in NoSQL databases, relational databases cannot be discarded, and they still occupy a large part of the industry. To satisfy the business needs, it was needed to use the best features of RDBMS (Relational Database Management System) and NoSQL, and at that point, the idea of joining both the data models came. There are two mainstream methods for integrating RDBMS and NoSQL databases: Polyglot Persistence and Multi-model Database. This research aims to provide the reader with an up-to-date analysis of different kinds of databases, how to connect them with an application, integration methods of Relational and NoSQL databases, and introducing some such models. A simple web application of Smart Prescription System was developed to evaluate the performance of using database integration methods of both kinds, and future research directions are discussed.

This thesis is written in English and is 71 pages long, including 3 chapters, 25 figures and 9 tables.

List of abbreviations and terms

IoT	Internet of Things
NoSQL	Not only SQL
ACID	Atomicity, Consistency, Isolation, and Durability
RDBMS	Relational Database Management System
ML	Machine Learning
IBM	International Business Machines Corporation
ACM	Association for Computing Machinery
BASE	Basically Available, Soft State and Eventual Consistency
DBMS	Database Management System
SQL	Structured Query Language
JSON	JavaScript Object Notation
XML	Extensible Markup Language
YAML	Yet Another Markup Language
BSON	Binary JSON
JDBC	Java Database Connectivity
DCP	Database connection pool
CGI	Common Gateway Interface
ASP	Active Server Page
MySQL	My Structured Query Language
API	Application programming interfaces
HTTP	Hypertext Transfer Protocol
JDBC	Java Database Connectivity
ODBC	Open Database Connectivity
ORM	Object Relational Mapping
JTA	Java Transaction API
JNDI	Java Naming Directory
JPA	Java persistence API
ORDBMS	Object relational Database Management System
REST API	Representational State Transfer Application programming interfaces
N1QL	Non-first Normal Form Query Language
DSE	Dataset Enterprise
GUI	Graphical User Interface

CLI	Command Line Interface
UML	Unified Modeling Language
RAM	Random access memory
GPU	Graphics processing unit

Table of contents

List of figures

List of tables

Introduction

In recent times, with fast advancement in technology, rapid industrialization, accessibility of devices brought us to the time when we observe a sudden growth in mobile and compact devices usage. Accommodation and streamlining of these devices are followed by the evaluation of the Internet of Things (IoT), Cloud computing, etc., which eventually generate an immense amount of data, i.e., Big Data, and most of them are unstructured data. In general, the characteristics of Big Data are defined by the 3 V's, which are Volume, Velocity, and Variety. It is an evolving term that describes a large volume of structured, semi-structured, and unstructured data, and these data are beneficial for future ML (Machine Learning) and analytics projects. The generation of these enormous amounts of data introduced new challenges, and one of the biggest challenges is the storage of these data. While facing the Data storage challenges, we have to consider scalability, performance, and interoperability.

For data storage, the relational database is being used for almost 40 years, and after decades of experimentation and development of RDBMS (Relational Database Management System), we are now using the fully-fledged model, and it has been the industry standard till now. The theoretical concept of RDBMS was first introduced in June 1970 by Edgar Frank Codd, a researcher of the company IBM. He published the famous paper, i.e., A Relational Model of Data for Large Shared Data Banks, in Communications of ACM [1]. According to Paradaens et al., a database system is a collection of programs that run on a computer and help the user to collect, change, protect, and manage information [2]. RDBMS are undoubtedly useful for storing tabular data that can fit in a predefined schema and using SQL queries. They also support complex queries. The property of RDBMS is defined by ACID, which stands for atomicity, consistency, isolation, and durability. Nevertheless, while handling a large amount of highly connected data, these databases show performance issues such as more extended query time return and complex SQL queries. SQL seems to be less suitable for storing big data and interconnected data, resulting in the NoSQL movement. NoSQL is schema-less, and for better performance and scalability, they have discarded ACID compliance. The properties of NoSQL databases are defined by the BASE, which stands for Basically Available, Soft State, and Eventual Consistency. These types of databases are easy to design, have horizontal scalability, are highly available and open source. NoSQL databases provide less assurance than RDBMS, but they react quickly to rapid data changes and scale very well.

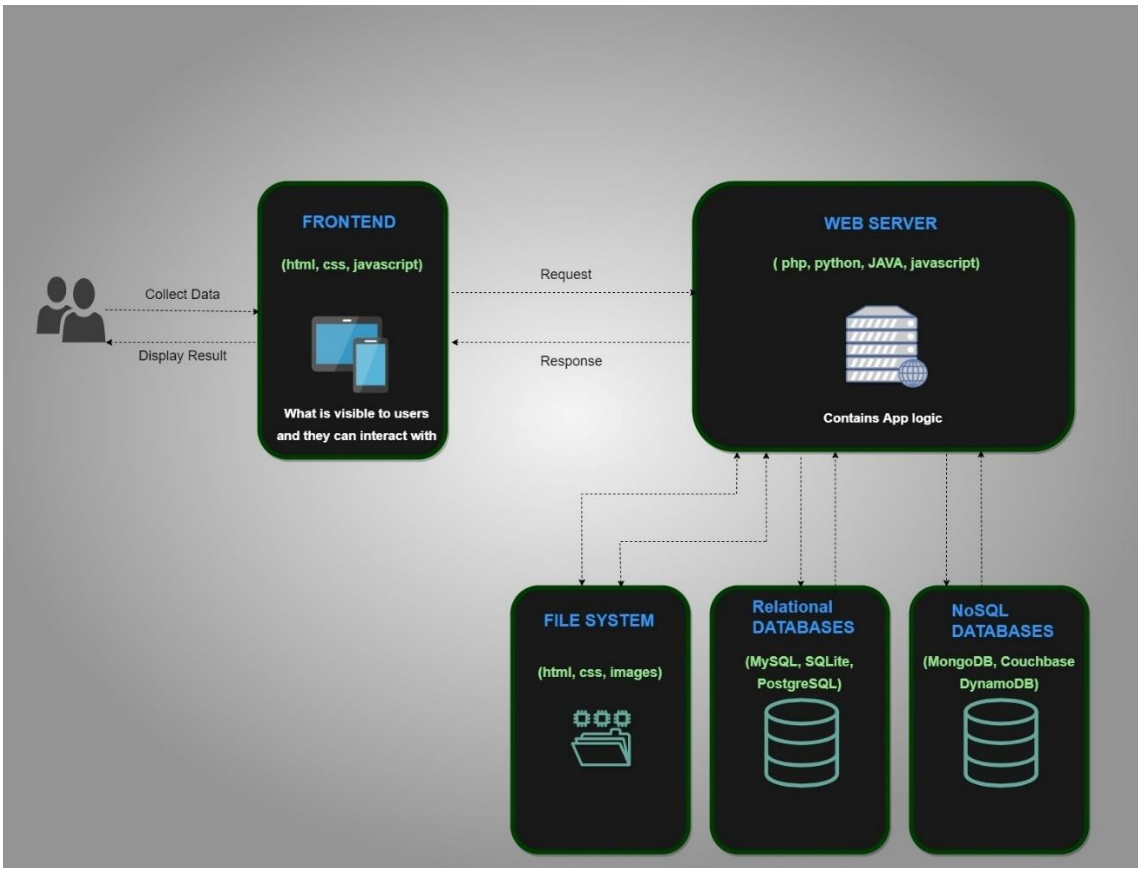


Figure 1: Web application architecture using SQL and NoSQL Data

Also, using a NoSQL database in many cases brings difficulties while storing ideal relational data. Both have their pros and cons. To handle the ever-increasing needs of users and enterprises, relying on only one kind of storage solution is not efficient. For solution, lots of research performed in recent years to utilize or combine both kinds of databases and data models, which proved that integration of RDBMS for structured data and NoSQL databases for unstructured data makes development, data management, and maintenance easier. Figure 1 shows such a scheme where the use of both SQL and NoSQL databases is required to provide optimal performance utilizing the best features of each type of database for optimal performance.

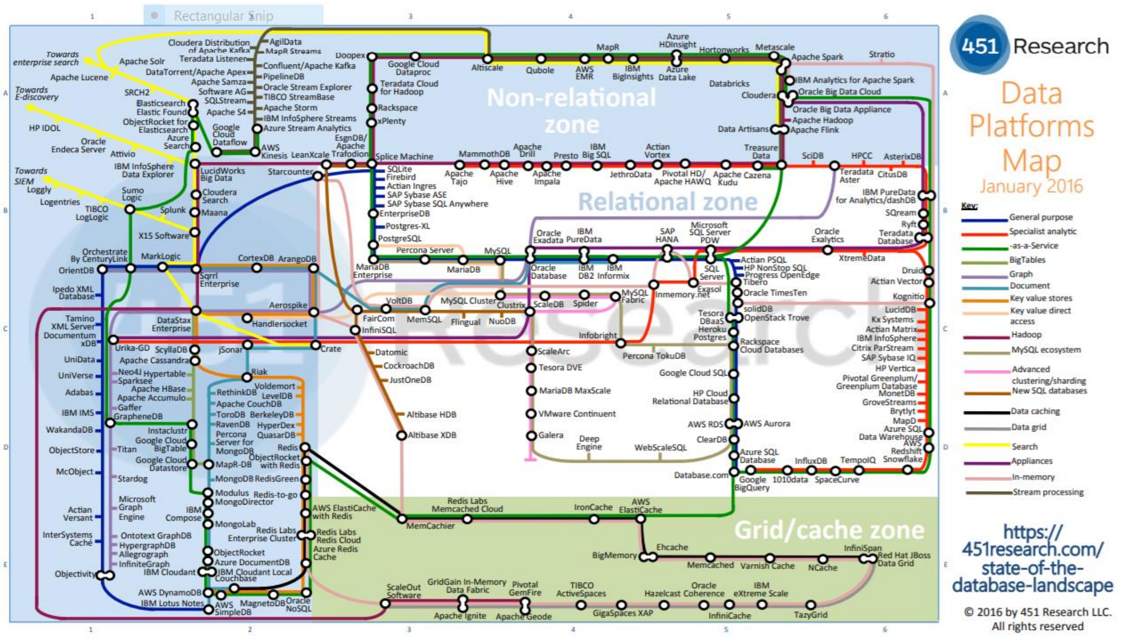
As the number of interactions increases, the load on the application increases as well, which leads to a rise in response time. Object-oriented design is the current trend in software design patterns, and the database connection pool is an essential research issue and how the application is connected with the database can impact this response time. In this study, we analyzed different databases, discussed different methods of connecting the application with a database, database connection pool, and different methods of database integration, e.g., multi-model database and polystores. Finally, a web application of Smart Prescription System is developed to compare the 2 types of database integration models based on different factors. Ongoing research works, finished research, and database and application development skills helped perform the research.

1 Analysis of different databases and connection of database with application

Every digital computing device generates data in today’s fast-paced environment, and these data can be monitored and analyzed by different users and enterprises for different purposes and the system that serves these services are called databases. These data can appear in structured or unstructured form, contain various data types, and need to be analyzed in real-time. Depending on this nature, the selection of a database is crucial. Also, how the database is connected to the application can affect response time. In this section, different types of databases are analyzed, and several methods of connecting databases with the application are discussed.

1.1 Analysis of Different Databases

The software that extracts helpful information and stores them from a collection of electronic and digital records is called Database Management Systems or DBMS. DBMS can save and retrieve user data while taking adequate security actions, including manipulating the database by several programs. The DBMS accepts the request from the applications and directs the OS (operating system) to supply the data. Data can be saved and retrieved on DBMS through users and other third-party applications, and through DBMS, businesses, and enterprises can get valuable insights and deep analytics. It serves as a link between the data and the software. This part will discuss the most popular types of DBMS in detail. There are more than 343 database engines available, and it is a huge challenge to choose the proper database engine to fit the business application. Figure 2 shows the database technology available on the market till January 2016 to help navigate the complex array of data platform providers [3].



451 Research
Data Platforms Map
January 2016
Non-relational zone
Relational zone
Grid/cache zone
https://451research.com/state-of-the-database-landscape
© 2016 by 451 Research LLC. All rights reserved

Figure 2: State of Database Landscape [3]

1.1.1 Relational Databases

These are the traditional database systems that use standardized Structured Query Language (SQL) for queries, and for this reason, they are also called SQL databases. They manage data using a relational model [2]. The relational model is designed based on the mathematical term relation, i.e., a subset of Cartesian products. The data are logically represented as tuples that describe types of entities forming relations and follow strict schema, preventing unnecessary data from entering the database. The property of Relational Database Management System (RDBMS) is defined by ACID, ensuring data validity in case of system failure and data validity. The relational data can be defined and queried using Structured Query Language (SQL). There are some basic rules for relational databases [1]. They are:

- Tables containing rows and columns must be used to store data and information.
- The name of the table, column, and primary key must be stated in order to access the content of a column.
- Cases of absence and improper entry must be treated differently from expected entries and must not be based on the type of data inserted.
- An active online index should be supported by the Database Management System.

- The Database Management System must support at least one language, which may be used independently or in conjunction with other applications.
- Views must be able to be updated by the Database Management System.
- The Database Management System must support basic operations such as insertion, updating, and deletion.
- Changes to the logical structure, such as adding or deleting columns from tables, must not have an impact on user perspective.
- Physical changes, such as storage, must not have an impact on the overall application.
- Integrity constraints should be separated from the application.
- The user must experience the impact of database distribution in a distributed system.

1.1.2 NOSQL Model

With the advancement of technology and usage of big unstructured data, RDBMS failed to show required performance, scalability, and flexibility when dealing with unstructured and semi-structured data. As analysis of these data became very important for research and innovation and RDBMS was not suitable for this task, NoSQL database systems were developed. NoSQL, which is also referred to as "Not Only SQL," is a non-relational Database Management System (DBMS) capable of accommodating a wide variety of data models. NoSQL is flexible, scalable, schema-agnostic, and data can be clustered in this kind of database. NoSQL databases do not guarantee ACID properties such as RDBMS; instead, they have Base properties that stand for Basically Available, Soft State, Eventual consistency. The meaning of basically available is the system will be available even though some parts are not. Soft state indicates the system can be used at the time of inconsistent states, and the system can tolerate these states without application interaction. Eventually, consistency indicates that the system will be consistent after the application input because data is replicated in different nodes, leading to a consistent state. Table-1 shows some key distinctions between RDBMS and NoSQL databases.

Table 1. Difference between Relational and NoSQL Databases

Properties	**Relational Database**	**NoSQL Database**
Data Model	▪ Relational ▪ Stored in rows and Columns	▪ Domain Specific ▪ Includes wide range of databases and each type has different data models
Database Schema	▪ Strictly follows schema ▪ Less flexible due to following schema	▪ Shema agnostic ▪ More flexible data model
Integrity	Database level	Application level
Scalability	Centralized (vertical scaling) and distributed (sharding) which is time sometimes time consuming and expensive	Horizontally scaled and replication features
Properties	ACID properties guarantee transections are processed reliably to ensure data integrity	BASE properties does not guarantee data integrity
Normalization	Yes	Some NoSQL database does not support normalization
Querying	SQL	Simple API. Some of them supports SQL

There are many categories of NoSQL Databases. The basic four categories [5] are:

Key/Value Model

The key/value store is the simplest data model among the NoSQL databases. The goal of this kind of database is quick access to data. It corresponds to associative arrays, dictionaries, or hashes. Each record in this type of database consists of an arbitrary value and its unique key. This key is required to store, retrieve, and modify the value. The Key-Value model has no data schema, and the records are arranged into collections. The keys can only retrieve the values as they do not contain any reference. These values are independent of one another, stored as bytes. The data stored in these databases can be any kind of binary object which may include JSON documents, text, video, etc. These data are accessed by key, and it is the application's responsibility to understand what is stored inside each record. While a large number of small read and write operation needs to be performed on singular values or continuous streaming, these types of databases are very much efficient [6]. Key-value databases generally show excellent performance and have easy scalability [7]. Popular Key-value stores are Redis, Memcached, etcd, Hazelcast, etc.

Wide Column Database

A wide-column database or column family store is usually a NoSQL database that organizes storage data into different flexible columns, and these columns can be spread across multiple database nodes or servers. This type of database uses a concept called keyspace, similar to RDBMS. Keyspace contains all the column families, which is like tables in RDBMS, and these column families contain rows, and these rows contain columns. The number of column families is strictly defined. The values of some column families are located together, and the rest of the row resides elsewhere [8]. Multi-dimensional mapping is used to reference columns, rows, and timestamps data. This kind of database is easily scalable; data can be partitioned easily and have a flexible data model. Also, queries for a particular column are swift. These databases are very effective for the application where data needs to be the number of attributes in the dataset varies hugely. Some popular wide-column databases are Cassandra, HBase, Microsoft Azure Table Storage, etc.

Document Stores

Document storage is the storage unit of a document database, and it is the most general NoSQL data model. The idea of this semi-structured model is to represent the data without an explicit and separate definition of its schema. Instead, the particular pieces of information are interleaved with structural/semantic tags that define their structure, nesting, etc., and it ensures that processing and exchanging data is becoming more flexible. This type of database is more similar to Key-value stores, and for this case, value is a whole document related to a specific key. The document can be in different formats such as XML (Extensible Markup Language), YAML (Yet Another Markup Language), JSON (JavaScript Object Notation), BSON (Binary JSON), etc. and these documents can contain different datatypes even in nested form. Document stores are very efficient when the application handles structured data with some variance in the number of columns or data types. They structure the data when necessary as well as provide flexibility.

Graph Model

The graph data model is built based on the mathematical definition of a graph which is a collection of vertices (nodes)V and edges E that correspond to pairs of vertices from V. These kinds of database stores data in node and edge object, which refers to the relationship. Graph stores are devoted to efficient storage, mapping, and management of the graph data. Graphs stores are not an extension of the Key-value pair. They are very much efficient in handling relational querying and storing interconnected data. Though RDBMS also handles relationships efficiently when the data is densely interconnected, the performance issue arises queries become more and more complex. Graph databases can perform these tasks in a constant time, for which they are more effective for social networking scenarios and dependency analysis. Graphs stores can be distinguished into 2 types [9]. The first type is Transactional databases which deal with a large set of smaller graphs, for example, a set of linguistic trees or chemical compounds. Generally, the operations search for super graphs, subgraphs, or similar graphs. On the other hand, the second type, non-transactional databases, target a single large graph (e.g., a Covid contact tracing), which may contain several components. This kind of operation includes searching for the (shortest) path, locations (i.e., subgraphs with specific features), and so on. Table 2, 3, 4, and 5 shows different features of 3 most accepted representatives of Relational, wide column, key-value, and Document databases.

Table 2. Features of different Relational Databases

DBMS Name	Offered database models	Supported Programming Languages	Partitioning method	Query languages	Supported Data formats (for reading)	Access Methods
SQLite	Relational	C, C++, Ruby, Ada, Forth, Delphi, Lua, D, OCaml, Ruby, Haskell, Java, JavaScript, C#, Lisp, Lua, Perl, PHP, Python, R etc.	None	SQL	CSV, TSV	JDBC, ODBC, ADO.NET
MySQL	Relational, Document Store, Spatial DBMS	Ada, C, C#, C++, D, Delphi, Haskell, Java, JavaScript (Node.js), OCaml, Perl, PHP, Python, Ruby etc.	Sharding with MySQL Cluster or MySQL Fabric, horizontal partitioning	SQL, Memcached API	CSV, XML, Native JSON, Avro, text, Parquet	ADO.NET, JDBC, ODBC, Proprietary native API
Microsoft SQL Server	Relational, document store, Graph DBMS, Spatial DBMS	C#, C++, Delphi, Go, Java, JavaScript (Node.js), PHP, Python, R, Ruby etc.	Horizontal Partitioning, Sharding through federation	T-SQL	XML, text, JSON Binary	ADO.NET, JDBC, ODBC, OLE DB, Tabular Data Stream (TDS)

Table 3. Features of different Wide-Column Databases

DBMS Name	**Offered database models**	**Supported Programming Languages**	**Partitioning method**	**Query languages**	**Supported Data formats (for reading)**	**Access Methods**
Cassandra	Wide column stores	C#, C++, Clojure, Erlang, Go, Haskell, Java, JavaScript (Node.js), Perl, PHP, Python, Ruby, Scala	Sharding	SQL like CQL	JSON, CSV, Parquet using spark, TSV	Proprietary protocol, Thrift
HBase	Wide column store	C, C#, C++, Groovy, Java, PHP, Python, Scala	Sharding	Does not support SQL. Can query data using Drill.	Avro, JSON, CSV, TSV	Java API, Restful http API, Thrift
Crate DB	Relational DBMS (Columnar store based on Lucene and Elasticsearch), Document Store, Time Series DBMS	Erlang, Go, Perl, Java, JavaScript (Node.js), PHP, Python, Ruby, Scala	Sharding	SQL (without trigger and constraints)	JSON, CSV	ADO.NET, JDBC, MQTT, Postgre wire protocol, Prometheus Remote Read/Write, Restful HTTP API

Table 4. Features of different Key-Value Databases

DBMS Name	**Offered database models**	**Supported Programming Languages**	**Partitioning method**	**Query languages**	**Supported Data formats (for reading)**	**Access Methods**
Redis	Key-Value Store, Document Store, Graph DBMS, Spatial DBMS, Search Engine, Time Series DBMS	C, C#, C++, Clojure, Crystal, D, Dart, Elixir, Fancy, Go, Haskell, Haxe, Java, JavaScript (Node.js), Lisp, Lua, Perl, PHP, Prolog, Pure Data, Python, R, Rebol, Ruby, Rust, Smalltalk etc.	Automatic hash based sharding	Using Dataframe API with the help of Spark-Redis Library	CSV, JSON as key or hash structure	Proprietary protocol
Memcached	Key-Value Store	.Net, C, C++, ColdFusion, Erlang, Java, Lisp, Lua, OCaml, Perl, PHP, Python, Ruby	Data can be stored in different nodes	Using telnet (get, set, add, replace etc. commands can be executed)	CSV, JSON as hash structure	Proprietary protocol
etcd	Key-Value Store	.Net, C, C++, Clojure, Erlang, Haskell, Java, Go, PHP, JavaScript (Node.js), Perl, Python, R, Ruby, Scala etc,	Network partition	Using etcdctl, gRPC APIs or client library API (get, watch etc. command used)	CSV, JSON as hash structure, binary	gRPC JSON over http

Table 5. Features of different Document Databases

DBMS Name	Offered database models	Supported Programming Languages	Partitioning method	Query languages	Supported Data formats (for reading)	Access Methods
Mongo DB	Document Store, Spatial DBMS, Search engine, Time Series DBMS	Actionscript, C, ColdFusion, D, Dart, Erlang, Go Haskell, R, Java, Lisp, C#, JavaScript, Lua, MatLab, Perl, PHP, Prolog, Python, Ruby, Rust, Smalltalk, Swift, C++ etc.	Sharding (Hashed, range or zoned Sharding Keys)	JSON-based MongoDB query language(MQL)	Avro, Parquet, ORC, JSON, Mongo DB Extended JSON, BSON, CSV.	Proprietary protocol using JSON
Couchbase	Document Store, Key-value store, Spatial DBMS	.Net, C, Clojure, Erlang, Go, Java, JavaScript (Node.js), Perl, PHP, Python, Ruby, Scala etc.	Sharding	N1QL	AVRO, JSON, CSV, TSV	Native language bindings for CRUD, Query, Search and Analytics APIs
Firebase Real-time Database	Document Store	Java, JavaScript, Objective-C	Sharding	Cloud Firestore can be queried using SQL syntex	JSON	Android, IOS, Javascript API, RESTful HTTP API

1.2 Connecting Database with Application

In database access, the database connection plays an important role. The first step of database access is to establish a connection with the database, and the last step is to disengage from the database [11]. Each connection must perform user authentication, establish transactional context, security context configuration, and other aspects of sessions required for successful database usage. To perform all these tasks requires a certain amount of communication and memory. As a result, database access is frequently the most time-consuming and expensive process. Moreover, if the connections are left open after completion of the session, it may result in more severe issues such as memory leakage, which may lead the application to crushing [12]. That is why the best database connection option should be chosen, significantly increasing the system's database performance.

To solve the problems that comes with the traditional method of accessing a database, the database connection pool is used. A database connection pool is a collection of live connections from application to database systems. This is done by establishing a buffer pool where a certain number of connection objects can be stored. The applications maintain the database connection pool because they frequently need to establish and terminate connections with the database. When a connection request is made, a connection is pulled from the buffer pool if the demand is high enough, and if there are unused connection objects, the connection pool allocates that unused connection to the system. This improves application speed, reduces system overhead, and allows for more efficient concurrency and scalability. It significantly decreases or eliminates the number of times programs must wait for a connection. Additionally, the usage of connection pooling is transparent; it has no negative impact on business applications as no modification is needed to communicate with the database pools. Application administrators can tweak and tune the connection pool at any time, and for that, they do not need extensive knowledge about the application. After finishing operation through that connection, the database connection pool releases the connection to avoid waste of resources and reduce excessive overhead. It is possible to direct the connections to various types and vendor's databases if the applications use generic JDBC connections, and for that, no code is required.

It is very much flexible to create, close, and configure connection pools. For the database connection pool, there is no need to create and close the database connection repeatedly; rather same connection is reused, which enhances the performance and stability of the system.

The primary and most crucial requirement of database connection pooling is to predetermine the number of connections offered by the database system during the application system's initialization stage. This predetermined number of connections is accommodated within the memory, called the database connection pool. These pools are to be organized using container objects such as vectors, stacks, etc. [12] This way, the apps only incur the overhead of creating these connections at the start and closing them at the end of the tasks. Through these processes of acquiring and terminating connections using a database connection pool, massive quantities of system resource usage are reduced, and execution speed is enhanced.

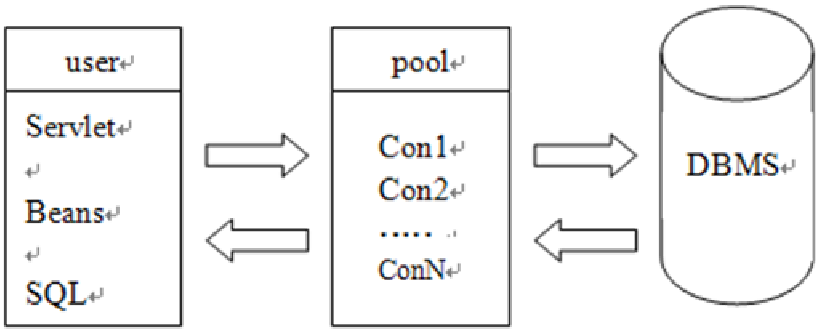


Figure 3: The principle working diagram of the database connection pool [13]

The database access operation primarily includes establishing connections, sending objects to connections, read and write databases, and closing connections [13]. The database connection pool is one of the most efficient solutions that will enhance the run-time performance of IoT/web/database transactional activities. The basic model of the database connection pool is a "resource pool" or buffer pool for the connections. To solve the optimization problem of connection, increase the speed of the system, and close issues of the database, this is used. A collection of connections that are established in advance and placed in-memory object, and are ready for the database operation are a database connection pool. When a program needs to access a database, first it needs to establish a database connection, and after using close the connection, which requires time and generates more overhead. With DCP (Database connection Pool) application does not create a new connection; rather, it just picks up an idle connection from memory and puts the connection back into the memory after use. Thus, it saves time because there is no need to create and close a new connection. Fig 3 shows the principle working diagram of the database connection pool [13]. In this diagram, the connection pool has N number of connection objects for database access, and each of these connection objects is commonly regarded as a heavy-weight data structure. Considering some of the connections are already in use and others are in an idle state if a client through the Servlet accesses the database, the connection pool management first looks for the idle connections, and when it detects Con1 is idle, the connection pool management allows the application to use Con1 connection to access the database. After completion of access operation, the connection pool management restores the idle state of Con1 and automatically puts it back to the connection pool.

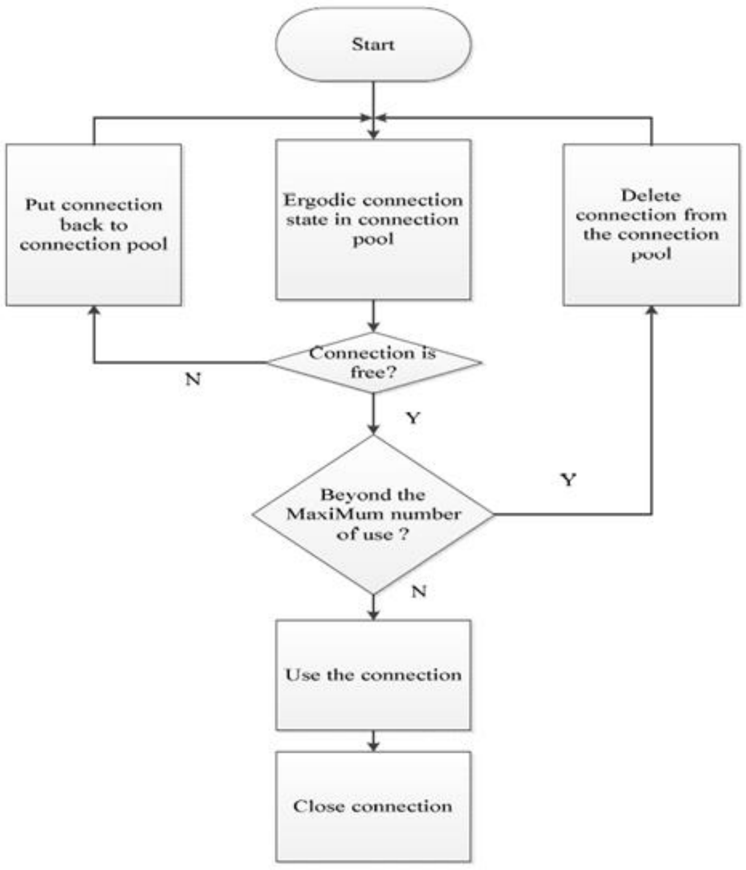

Start
Put connection back to connection pool
Ergodic connection state in connection pool
Delete connection from the connection pool
Connection is free?
N
Y
Beyond the MaxiMum number of use ?
Y
N
Use the connection
Close connection

Figure 4: Connection pool workflow diagram [13]

The connection pool uses its management mechanism, which can realize and decide about establishing connections, disconnecting, and managing state judgment. The connection pool workflow diagram is shown in figure 4 [13]. The connection pool maintains the usage counters for each of the connections, and there is a minimum and a maximum limit of usage for each connection. When a connection’s usage counter exceeds the maximum limit, the connection is removed from the connection pool by the connection pool management system to ensure the stability of the system. When an application needs to connect with the database, the connection pool management looks for the idle connections from the ergodic connection state from the connection pool as well as the number of usages of the idle connections. Based on its state judgment, connection pool management decides which connection to allocate for the operation.

When using a database connection pool, some factors should be considered for efficiency. Firstly, the number of connections should be predefined, and the connection pool can only establish a certain number of connections because the more connection, the more considerable time it takes to connect. So, the constraint management through the maximum and the minimum number of connections should be effective to serve all the use cases of the system. This also helps to manage the idle connections effectively. Another factor is allowing threads to access the connection pool synchronously when the connection pool is locked. This is an exclusive mechanism of the connection pool. Some of the approaches for connecting database with application are discussed below

1.2.1 CGI

CGI stands for Common Gateway Interface. It is an interface specification that allows web servers to execute external programs. It is trivial in design, and developing a CGI script can be simple. C, C++, Java, Perl, Visual Basic, PHP, etc., languages are used to develop CGI programs, and these programs are called CGI scripts and have the extension “.cgi”, and are placed under some particular directory typically called “cgi-bin”. CGI programs exchange data with web servers via special library files. These library files function as parsers and translate coding information into an understandable format for the web servers. In this way, the application can fetch information from the database in an understandable way for the users.

A CGI program must generate a separate process for each user request, and this process will be stopped once the data transmission is completed. This is CGI's huge performance problem because a separate program instance for each client request takes a lot of time and resources. The operating system has to load the program, allocate memory for the program, and then deallocate and unload the program from memory [14]. As the operating system is performing all these tasks, there is no room for other programs to run; as a result, CGI programs are not efficient in handling concurrent client requests. Therefore, CGI programs are appropriate for small applications but not suitable for applications where a large amount of data needs to be handled.

1.2.2 PHP3

PHP stands for recursive initialism PHP: Hypertext Pre-processor. PHP is a general-purpose server-side, cross-platform HTML embedded scripting language for web development. Rasmus Lerdorf, a Danish-Canadian programmer, developed it in 1994 to keep track of visitors to his online resume. The first public version consisted of a few special macros and several common utilities, which were known as PHP tools or Personal Home Page Tools. Later, this interpreter was revised and renamed as Professional Home Page and made available as open source. PHP3 has become a very attractive tool for web to database application development as it contains some excellent tools such as Stronghold web server and Redhat Linux. Additionally, PHP can be used in other contexts such as Graphical Applications and Control systems, etc.

PHP provides broad support for databases which offers it an advantage over VBSCripts [1]. PHP code can be directly placed in an HTML page or as binary image data and executed on the server. Object-oriented programming can be done using PHP, which helps to create a large project and can be built on Apache module or as binary to run on CGI.

PHP developed as an Apache module: Apache and PHP are an effective and dependable combination for websites that do not require a vast, robust web application. The web server Apache is responsible for processing requests and serving web assets and content through HTTP. Apache is a multiprocessing web server that uses a parent-child relationship for request handling. The parent process is used by the webserver to coordinate the handling of requests by its child processes. When a request arrives, the parent process passes it to one of its available child processes, and the request is handled by that child process. Following requests from the same client may be serviced by different child processes. When built as an Apache module, PHP offers excellent connectivity to various kinds of databases and persistent database connections [15]. By taking advantage of Apache’s microprocessing features, it is possible to improve the program efficiency and increase throughput. As there is no process creation overhead in there, the result is returned very quickly. This way, PHP built as an Apache module is very lightweight, and the processing speed is excellent. Database connections are maintained persistently in this process and are shared among the child processes.

PHP3 built as binary to run as CGI: In the CGI solution, no connection is stored after the script has finished. For each PHP3 page request to the web server, a new instance of the PHP3 interpreter is produced and discarded afterward. All assigned resources, such as a link to a server, are closed when the instance is destroyed after each request. So, it is not feasible if PHP3 is implemented as a CGI solution because this will fail to get the benefit of a persistent connection since no connection persists once the script has finished.

1.2.3 Python

Python is one of the most popular high-level, general-purpose programming languages. Essentially, it was developed with code readability in mind, and programmers may express their notions in fewer lines of code. Python DB-API is the python standard for database interfaces. This standard is followed by most Python database interfaces. Python Database API works with various database servers, including MySQL, PostgreSQL, Microsoft SQL Server, Oracle, SQLite, IBM DB2, etc. For a python connection with a database, first, a connection request is sent to the Database connector python, then it gets accepted by the database, and finally, the curser is executed with result data. For database connection, using python middleware should be considered. Middleware maintains multiple connections to several Database servers and relies on those connections being readily accessible. In the case of a MySQL database, establishing a MySQL connection using python is both resource-intensive and time-consuming, especially when the MySQL connector Python API is utilized in a middle-tier server context. To configure a connection pool using python, we should consider the following factors:

- The number of maximum connections a Database module can support.
- The application's size and type, as well as how database-intensive the application is.
- The size of the database connection pool.

The number of connection objects is determined by the factors mentioned above.

Handling a single HTTP request with one connection per thread is adequate in many circumstances. Alternatively, fewer may be required if not every HTTP request requires database access. It is possible to determine how to configure the connection pool by reviewing the past request history and analyzing the nature of the application. To create a connection pool using python, first, we need to create a connection pool the get connection from the connection pool. After the connection, perform some database operations on it and finally close the connection instance. Steps of using a connection pool in python are:

1.2.4 JAVA

Among the programming available today, Java is considered one of the mainstream programming languages. Java programming language can work with various databases such as SQL server, Oracle, Sybase, MySQL, SQLite, etc. Java uses Java Servlets in a server application and JDBC, JPA, etc., to work with a database. The following part discusses in detail the database connection using JAVA.

Server-Side JAVA Servlets

Java Servlets are pieces of server-side Java class executed in a server application to respond to client requests. Servlets are not connected to any particular client-server protocol; however, they are most typically used with HTTP. Thus, the term "Servlet" frequently refers to "HTTP Servlet." Essentially, the programmer selects which servlet will be used to execute which client request or kind of request. As a result, when the Web server receives a request, it finds the appropriate servlet for the request. Because servlets are built in the highly portable Java language and adhere to a standard architecture, they enable the creation of complex server extensions that are server and operating system independent [15].

When a client initiates a request to the webserver, the webserver receives it, passes it to the corresponding servlet. The first time a servlet program is called, it is loaded into memory. The request is then processed by the servlet, and a response is returned back to the server. The servlet remains in memory after the request is handled and will not be unloaded until the webserver is shut down. Once a servlet is activated, it runs in the background and lives as long as the webserver does because usually, a web server runs continuously without any interruption.

The advantage of implementing servlets as Java classes is that they may easily remain in memory and be called on repeatedly. To obtain a performance advantage, most web servers allow loading some commonly used servlet programs as the webserver boots up. In this manner, servlets may already be in memory for the initial client request, avoiding first-time access costs. Loading the servlets at startup ensures that response time for all requests is maintained to a minimum.

JDBC for interacting with Database

JDBC (Java Database Connectivity) is a Java API that manages connecting to a database, executing queries and commands, and dealing with the database's result set. It is considered a component of Java SE, or Java Standard Edition. JDBC was one of the initial components built for the Java persistence layer, and it was released as part of JDK 1.1 in 1997. This technology is still intensively used in numerous projects [16], despite the inherently close coupling that is required between the source code and the database schema.

JDBC was designed to be a client-side API that allowed a Java client to communicate with a data source. This changed with the introduction of JDCB 2.0, which included a package that permitted server-side JDBC connections as an option. Since then, every major JDBC release has included improvements to both the client-side (java.sql) and server-side (java.sql) packages (javax.sql). The most recent version, JDBC 4.3, was published in September 2017 as part of Java SE 9 [17]. In order to connect to the database, the JDBC API makes use of JDBC drivers. There are four types of JDBC drivers which mentioned below:

- JDBC-ODBC Bridge Driver
- Thin Driver
- Native Driver and
- Network Protocol Driver

JDBC is the standard API with which application code communicates. Underneath that is the JDBC-compliant driver for the database. To access the tabular data stored in any relational databases, JDBC API can be used. Usage of JDBC API provides the ability to save, update, delete, and fetch the data from the database. Before JDBC API, Open

Database Connectivity or ODBC API was used to connect and perform queries along with the database. ODBC API uses ODBC driver in C language, which is platform-dependent and unsecured. Considering all these Java

However, ODBC API makes use of ODBC drive in C language. Also, it is platform-dependent and, in addition, unsecured. Using JDBC drivers, Java built its own API in Java known as the JDBC API.Several operations may be carried out utilizing the JDBC API, which is necessary to interact with the database:

1. Connection to database
2. Execution of queries as well as update statements to the database
3. Retrieving results fetched from the database.

JDBC makes it simple to connect to an application or a data source, send queries, update statements, and process results. JDBC helps to establish a connection with a data source very easily; updating statements and sending queries has become accessible by using JDBC and simplified data fetching processing data from the data source.

JDBC driver that helps in the execution of the JDBC API. It makes it possible to submit the SQL statements and queries and retrieve the results by calling the JDBC classes and interfacing with the help of the Java Application. This driver is made up of a collection of classes that implement the JDBC interfaces, which helps in processing the JDBC calls and sending output/results to the Java application.

JDBC supports both two-layer and three-layer processing models to access databases. Figure 5 shows the architecture of JDBC. Below the architecture of two-layer architecture is discussed:

- JDBC API: The Java Database Connectivity (JDBC) API provides universal data access from the Java programming language. Using the JDBC API, you can access virtually any data source, from relational databases to spreadsheets and flat files [18]. This layer supports the application-to-JDBC Manager connection. To provide transparent connection to heterogeneous databases, It makes use of the driver management as well as database-specific drivers.

- JDBC Driver API: To use the JDBC API with a particular database management system, a JDBC technology-based driver is needed to mediate between JDBC technology and the database. This layer connects the JDBC Manager to the driver. This driver may be written in Java, a mixture of the Java programming language and Java Native Interface (JNI). This driver manager ensures that the correct driver is used when accessing each data source. It may also handle a large number of concurrent drivers that are linked to numerous heterogeneous databases.

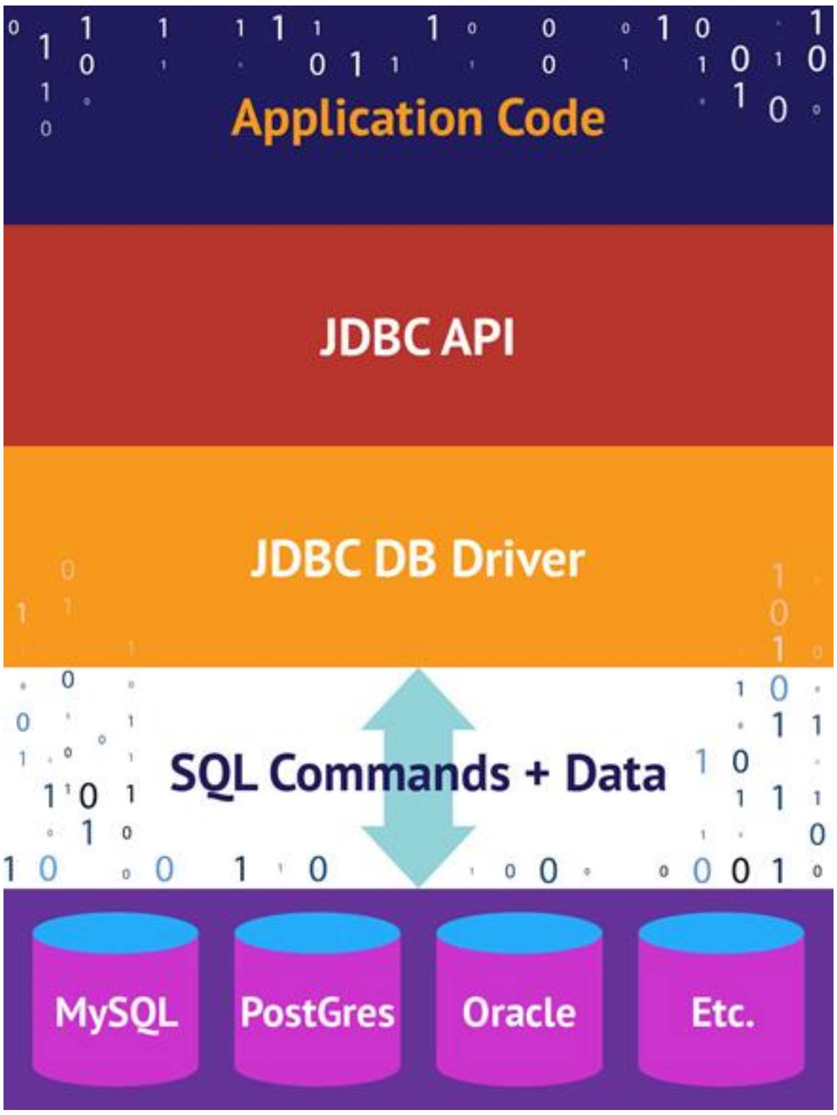


Figure 5: Architecture of JDBC [17]

JDBC consists of the following interfaces and classes:

- **Driver Manager:** Driver Manager is a static class in the Java™ 2 Platform, Standard Edition (J2SE) and Java SE Development Kit (JDK) [19]. The driver Manager class manages the list of drivers of the database and uses a sub-protocol of communication to match connection requests from the java application and the database driver. The first driver, which recognizes subprotocol under the JDBC, is used to establish a database connection [17].
- **Driver:** The Driver is the interface that manages interactions between the program and the database server. The odds of directly interacting with Driver objects are pretty low since most of the time, the objects of the Driver Manager are used to handle objects of this sort.
- **Connection:** For contacting the database a connection interface is used. The context of communication is represented by the object of connection; that is, all contact with the database is done only through the object of connection.
- **Statement:** The objects produced by this interface will allow you to send SQL queries to the database. While executing some stored procedures, some of the derived interfaces take arguments.
- **ResultSet:** ResultSet objects are used to retain data that has been fetched from the database. However, this occurs after a SQL query has been conducted using Statement objects. It also serves as an iterator, allowing us to traverse the data.
- **SQL Exception:** This class handles errors that occur in a database application.

The steps for connecting to a database with JDBC are as follows:

1. Installing or locating the database.
2. Loading drivers and including the JDBC library.
3. Registering JDBC driver using Driver Manager.
4. Establishing a connection to the database.
5. Using the connection to create SQL query.
6. Executing the query.
7. Closing the connection.

JAVA Hibernate

Java Hibernate is one of the most popular Java object-relational mapping (ORM) tools. ORM is the process that maps Java objects to database tables for web applications by providing a framework. It also transfers Java data types to SQL data types. The developer does not have to handle the database manually, and data can be inquired or retrieved using Hibernate.

Hibernate facilitates the interface between a database and the Java application under development. It is a JAVA Object-Relational Mapping (ORM) framework built-in 2001 by Gavin King. It is an open-source, high-performance, powerful, and lightweight ORM tool. Hibernate is a widely used implementation of the Java Persistence API specifications that provides a very sophisticated object-relational persistence and query service for Java applications.

Hibernate allows Java objects and database servers to communicate with one another.[20]. Hibernate will strive to persist Java objects based on the appropriate object-relational patterns and recognition techniques. Hibernate's design is built in such a way that for operating, the user does not need to know the underlying APIs. Hibernate is able to give persistence services and objects to the application by using the database and configuration data. Hibernate framework makes use of various objects such as session factory, session, transaction, and so on, as well as existing Java API such as JTA (Java Transaction API), JDBC (Java Database Connectivity), and JNDI (Java Naming Directory).
There four layers in Hibernate architecture are:

- Layer 1 – Java Application Layer
- Layer 2 – Hibernate Framework Layer
- Layer 3 – Backend API Layer
- Layer 4 – Database Layer

Java Hibernate is an ORM tool that helps ease issues such as writing the same line of code repeatedly, database switching, implementing object-oriented programming in JDBC, the association between database tables, etc. These problems are faced while using JDBC for database connectivity.

Hibernate connects itself with the database and uses HQL for the query. After executing queries, hibernate performs the mapping of the results to their respective application layer according to the configuration XML file in Hibernate. A session not only assists an application in establishing a connection with the database but also functions to save and retrieve the persistent object in Hibernate. The session allows to the construction of an instance of a session and utilizes it. However, there should only be one session factory per database.

JPA ORM

It is Java ORM standard to store, access, and manage Java objects in relational databases. JPA stands for the Java persistence API. It is not a tool or framework; rather it is a guideline that any tool or framework can enforce. The Java persistence API is a standard for persistence which is a method through which Java objects outlive their application process. Its definition allows you to specify which items must persist in Java programs and how they should persist. Although it was basically designed to be used with relational databases, certain JPA implementations with NoSQL datastores have been enhanced.

Each JPA implementation provides some ORM layer that has differences during execution. The ORM layer is responsible for translating program objects into a relational database, which is a component of the framework architecture, and for that reason, developers do not have to map manually.

The ORM layer turns Java classes and objects into relational databases, which may then be stored and managed in Java. The name of the item kept by design becomes the name of the table, and fields are turned into columns. [21].

The ORM layer is an interface layer that converts Object Graph languages to SQL and relational tables. The ORM layer enables object-driven developers to design data-collection apps without leaving the object-driven paradigm.

Rather than how the objects are stored and retrieved, the mapping between the objects and the database is established, and the JPA is triggered to persist. When using a reference database, JDBC can handle a large portion of the connection between the application code and the server. Each JPA implementation has its own engine for JPA annotations for metadata to describe the mapping of objects to the database in a precise fashion.

To decrease the responsibilities of developing code for connection object maintenance, the JPA Provider structure is followed, allowing simple interaction with the database case. The architecture of JPA ORM is shown in Figure 6.

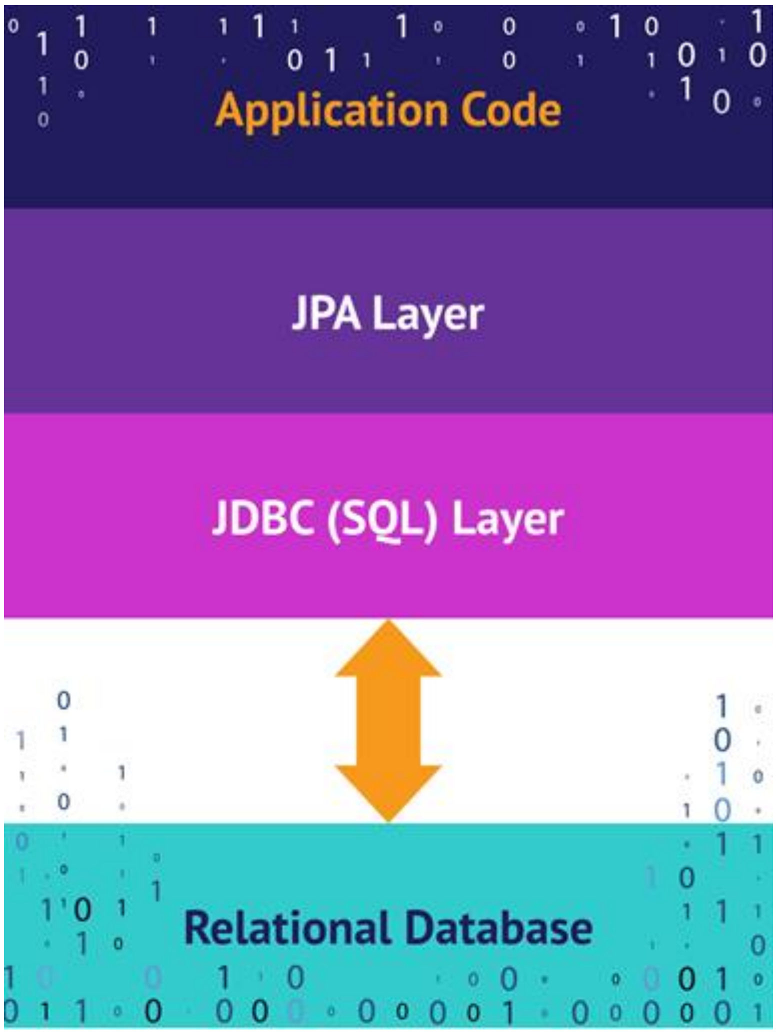


Figure 6: JPA ORM [21]

2 Database Integration

There are many research on how to integrate different kinds of databases. Following Section will discuss different methods for database integration.

2.1 Polyglot Persistence

With technological advancement, business needs and analysis of data guided us to develop different database management systems in the past decades. The majority of them are specialized to address particular problems. Relational database handles structured data exceptionally well where security is strictly ensured, and they have proven to be helpful for most business cases. On the other hand, NoSQL databases can handle a large amount of unstructured data and organize them effectively. As both of them have their distinct features and are effective in solving particular business cases, they cannot replace each other, and we can see that big enterprises using complex system relies on both kinds of databases. Doing so, they have to face problems such as integrating different kinds of databases, because there are several solutions out there and they have to choose the best one which suits their business needs such as Polyglot Persistence, Multi-model database, etc. Polyglot Persistence means that while storing data, it is better to employ several data storage technologies that are selected based on how data is used by individual apps or components of a single application, and the system which uses polyglot persistence is called polystore. Polystore works as a unit so that different systems under it can be managed easily and they can function effectively. It is basically picking the right tool for the proper use case, i.e., storing data to different data stores based on the nature and use of data and integrating them via a simple application, and this can be done using multiple languages. This is a distributed system focused on combining the capabilities of different database engines to ensure application-wide consistency and integrity. It's the same concept as Polyglot Programming, which holds that applications should be built in a variety of languages to take advantage of the fact that different languages are better suited to address different challenges. For implementing this type of system, middleware is designed which works on top of a database. This middleware handles queries by sending them to the relevant databases and integrating the partial results, and this middleware is accessed by the users to receive data in one place.

Polystores can fuel data from different storage engines, and these storage engines can be accessed through their own interface. While designing a polystore, multiple use cases are considered and combined effectively so that each data has a specific purpose regarding data. Building a polyglot persistence system is a complex task, so it is crucial to be concerned about some factors while designing the system. Some of the things that should be considered are data types, volume, and velocity of data, selection of right data storage technology, data storage type, access pattern, organizational needs, etc.

Polyglot persistence helps businesses to deal more effectively with scalability by using the capabilities of several data stores. An organization usually goes for polystores when a large amount of structured and unstructured data needs to be dealt with in a unified system, analysis of these data is needed, there is a need to provide transparency to the system, etc. Before moving to polyglot persistence, we must have to comprehend the nature of data and how those should be manipulated. Though Polyglot persistence makes it easy to store a large amount of data having different natures, it comes with complexity, higher cost, and there is always an issue with consistency. In the following part, we will discuss several polystore implementations and discuss multi-model databases. Polystores can fuel data from different storage engines, and these storage engines can be accessed through their own interface. While designing a polystore, multiple use cases are considered and combined effectively so that each data has a specific purpose regarding data. Building a polyglot persistence system is a complex task, so it is vital to be concerned about some factors while designing the system. Some of the things that should be considered are data types, volume, and velocity of data, selection of suitable data storage technology, data storage type, access pattern, organizational needs, etc.

Polyglot persistence helps businesses to deal more effectively with scalability by using the capabilities of several data stores. An organization usually goes for polystores when a large amount of structured and unstructured data needs to be dealt with in a unified system, analysis of these data is needed, there is a need to provide transparency to the system, etc. Before moving to polyglot persistence, we must have to comprehend the nature of data and how those should be manipulated. Though Polyglot persistence makes it easy to store a large amount of data having different natures, it comes with complexity, higher cost, and there is always an issue with consistency. In this section, we will discuss several polystore implementations and discuss multi-model databases.

2.1.1 BigDAWG

One of the common polystore implementation is BigDAWG which provides an array data model for three different database engines. For example, their D4M island provides common interface for relational database PostgreSQL, array database SciDB and text database Accumulo [22]. It is an opensource solution for polystore which is a single back end to support location transparency for the storage of objects and ensure semantic completeness which ensures that the user will not lose capabilities over the storage engines.

The BigDAWG architecture consists of four distinct layers as described in Figure 7: database and storage engines; islands; middleware and API; and applications [22].

The BigDAWG middleware design is built on a high-level common API. Applications use the Common API to interface with BigDAWG. Islands are components that give general-purpose interfaces to a specific data model. They may be queried using a particular query language or set of actions. Currently, BigDAWG supports three Islands. SQL is used by Relational Island to query PostgreSQL databases. Array Island is used to query SciDB databases using AFL, the query language of SciDB. Text Island queries Accumulo using SQL or a D4M language. BigDAWG Common API offers an interface across the Islands. It is made up of four parts: the Planner, the Executor, the Monitor, and the Migrator. The API also connects with a separate component known as Catalog, which stores all database and object schema information. Shims connect Islands to database engines by converting queries from the Island query specification into native queries for each database engine. Casts are used to transport data between database engines when using multi-model queries.

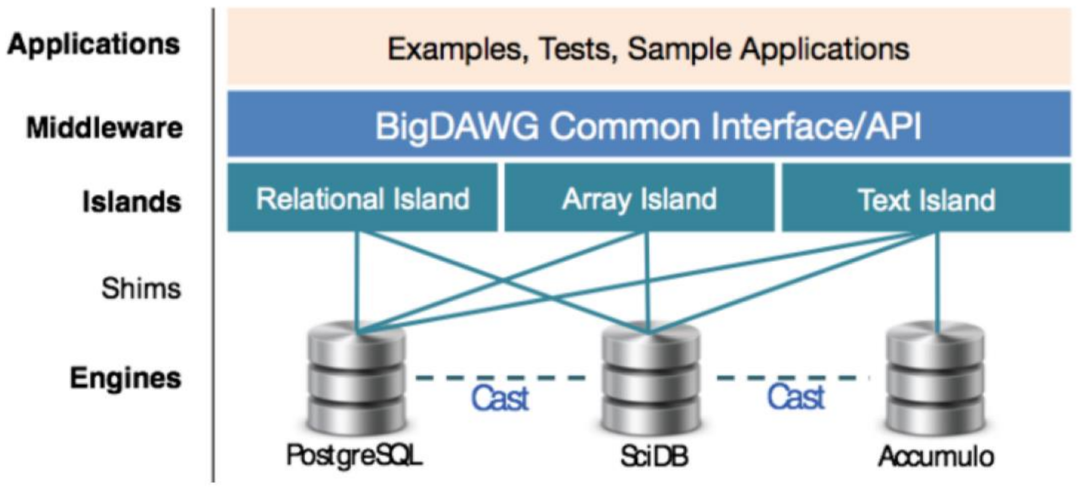


Figure 7: BigDAWG Architecture [22]

The Catalog is a component that includes two Postgres databases: a primary catalog database and a schema database. The catalog database stores information about the engine, database, object, shim, and cast, whereas the schema database keeps information about the object schema. Engine connection characteristics such as hostnames and ports are stored in the engines table. The database table includes information on the databases that are operating in each engine, as well as their credentials. The Objects table provides the names of the objects in each database as well as their field names. The schema database holds object schema globally, allowing casting from one database type to another while without breaking restrictions.

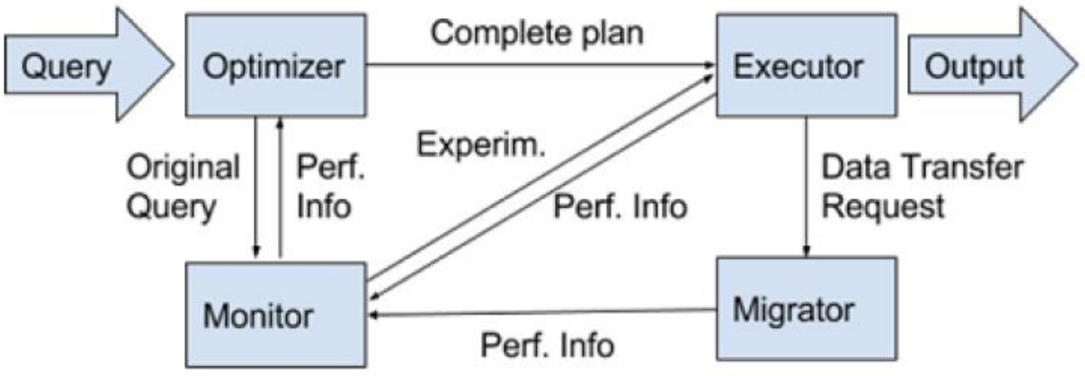


Figure 8 BigDAWG Middleware components and workflow [15]

Figure 8 depicts the components and operation of middleware. The Planner is in charge of coordinating query execution. It parses queries, plans them, and optimizes them. When optimizing queries, performance data from the Monitor is used. It communicates execution plans to the Executor. It also has a separate Training mode for collecting execution plan statistics prior to launching into production. While in Training mode, the Planner examines all alternative execution plans that give the same outcome and sends them to the Monitor to collect all conceivable metric data. The quickest plan is then picked from among them. The correct production mode just evaluates the query's characteristics and asks the Monitor for the optimal plan.

The Executor executes queries that have been created by the Planner and Monitor. It moves through execution plans, issuing sub-queries to the various islands. When data must be transferred across islands, it invokes the Migrator. The Migrator is in charge of migrating data across databases. When casting data from one island to another, it searches the Catalog for the schema information required. The Monitor keeps track of the execution timings of query plans. It executes training queries to learn result metric execution time characteristics in order to predict execution times for future queries.

2.1.2 Polybase

PolyBase is a technology designed to enhance Microsoft SQL Server applications by allowing T-SQL queries to read and process data from external data sources [23] such as SQL Server, Oracle, Teradata, MongoDB, Hadoop clusters, Cosmos DB, and others. PolyBase supports the polyglot persistence concept by enabling queries to simultaneously target a SQL server instance and an external data source. Polybase targets external data sources works with external tables or external data sources without the requirement to establish the target system. Parallel computing with several SQL Server instances is also possible with PolyBase. PolyBase additionally supports query computation pushdown by connecting additional providers to other systems through generic ODBC connectors and constructing SQL Server clusters for concurrent query processing. It is possible to avoid relocating data from its native location and format while using Polybase for data virtualization. Using Polybase connections, external data may be virtualized through the SQL Server instance, reducing the ETL process of data migration.

Apart from linking data to external data sources, the following are the reasons for adopting PolyBase: Transfer half of the data so that all the data is in one place and query both data sources, then create custom query logic at the client level to combine and integrate the data. PolyBase different kinds of data formatssuch as: structured data, semi-structured document data, as well as unstructured data. Some examples of such data are: delimited text data and Hadoop HDFS files. Previously to integrate or join data in a client application, data needed to be moved to a single location, but polyglot solved it efficiently only by using T-SQL.

PolyBase implements computation pushdown in Hadoop. It means that the query optimizer can make the decision to either perform query computations in Hadoop or in the main SQL Server system. The decision is made by estimating the performance cost. Pushing down computation can utilize Hadoop's optimized distributed computing with MapReduce jobs, for example. This can also be forced on or off in queries. PolyBase also allows parallel computing with multiple SQL Server instances. The query optimizer may choose whether to do query calculations in Hadoop or in the main SQL Server system In Hadoop, this is referred to as compute pushdown. Pushing computation down can take use of Hadoop's efficient distributed computing, such as MapReduce tasks.

2.1.3 RHEEM

With the philosophy of "one size does not fit all", REEM was developed. This system allows a large variety of applications to achieve processing platform independence and multi-platform task execution. RHEEM Architecture is a 3-layer data processing system that lies between applications and data processing systems like Hadoop and Spark: (i) an application layer that models all application-specific logic; (ii) a core layer that acts as an intermediary between applications and processing platforms; and (iii) a platform layer that incorporates the underlying processing platforms [24]. The architecture is shown in figure 9.

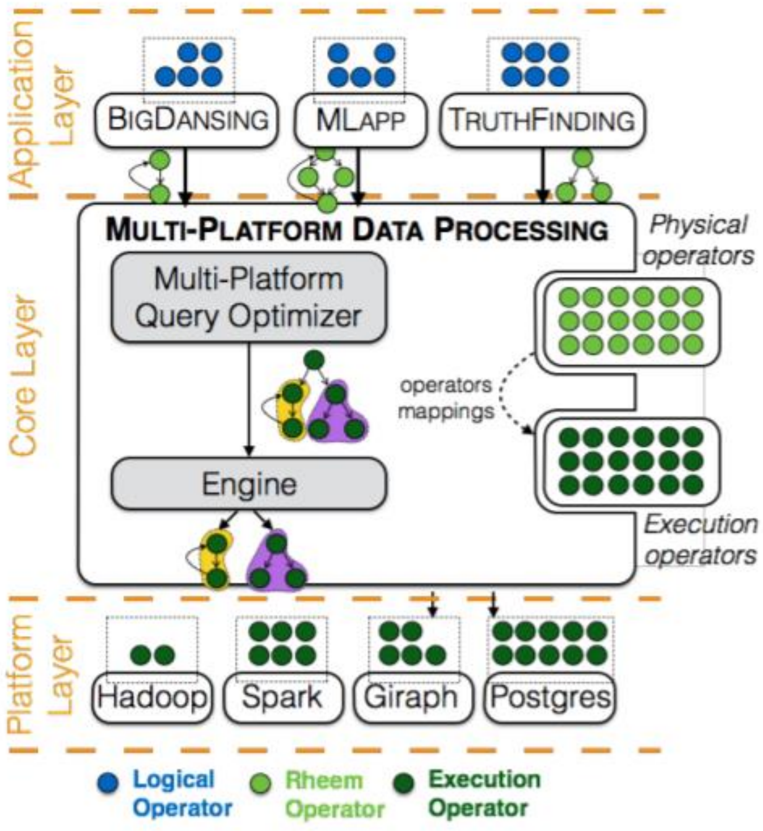


Figure 9: RHEEM Architecture [24]

Operators defined as user defined functions enable communication between these three levels or UDFs. At each tier, Rheem provides a collection of logical operators, Rheem operators, and execution operators. The application layer generates a collection of feasible optimal Rheem plans using user-supplied implementations of the logical operators unique to the application. An application sends these Rheem plans, together with the cost functions, to the core layer to assist the optimizer in selecting the optimum plan. Applications receive these cost functions and provide them to the core layer. Rheem conducts many multi-platform optimizations and generates an execution plan at the core layer.

2.2 Multi-modal Database

One of the most difficult problems for study and practice in data management systems is the variety of data. Structured data, semi-structured data, and unstructured data are all naturally arranged in different formats and models. Despite the fact that multi-model databases are a relatively new domain, we have seen numerous database systems embrace this category in recent years. We see a significant growth in demand to analyze and handle multi-model data, including structured, semi-structured, and unstructured data, since data of many sorts and forms is critical for optimum business choices. Structured data, in particular, encompasses relational, key/value, and graph data. XML and JSON documents are examples of semi-structured data. Either the data is kept in multiple database management systems (DBMSs) according to the four data models, or the data is translated into a single format.

Multi-model database management systems integrate many database systems into a single database system. It is only possible due to the use of a single back-end. Multi-model databases provide a single engine for numerous database types, eliminating the need to work with multiple models and figure out how to combine them. It enables developers to program in an agile and flexible manner. Another significant benefit of this type of database is that it decreases redundancy. Multi-model databases are capable of storing, querying, and indexing data from many models. The modeling benefits provided by a multi-model database eliminate the need to discover a way to merge disparate models, and data may be stored in a variety of ways. To use a single multi-model DBMS to get the benefits of both SQL and NoSQL solutions: (1) The data are stored in the most efficient manner for the specific models, and (2) just a single DBMS is used to query across all models. Users benefit from a single data platform for multi-model data by offering not just an uniform query interface, but also a single database platform to simplify query operations, decrease integration concerns, and remove migration issues.

Many vendors are offering different multi-model DBMS products, such as:

- OrientDB is a multimodal database management system that supports geographic, graph, full-text, and key–value data formats. It is essentially a distributed unique graph database of the second generation that provides document flexibility. On typical hardware, it is 220,000 records per second [25]. It supports schema-agnostic, full, and mixed SQL modes. In OrientDB, OO ideas are employed for user domain modeling. A record is the most fundamental (and smallest) data unit. There are four different types of records: byte record (BLOB), document, node, and edge.
- ArangoDB is a native multimodel database that supports key–value, document, and graph data models. A document can have an unlimited number of attributes

with simple and complex values. ArongoDB can employ more than a declarative model for safety. ArangoDB has two types of collections: document (node) collections and edges collections. Edge is a type of document that has two distinct characteristics: from and to are used to indicate the relationship between documents. As a result, documents are arranged in a directed graph.

- Couchbase Server is compatible with both key–value and document techniques. This database is relatively simple to set up and operate. This geographically distributed database offers unrivaled developer agility and management. It also has unrivaled performance. For key–value operations, Couchbase Server employs the memcached binary protocol, whereas N1QL and view queries are handled via REST APIs. N1QL is a sophisticated declarative language that can query, alter, and manipulate JSON data. This type of database has replication as a standard feature.
- MarkLogic allows you to save and search JSON, XML, and RDF triples. It supports JavaScript and JSON on the server. It includes several strong enterprise features, such as semantics and bitemporal, that assist businesses reduce risk. It can also store photographs, movies, and other data. Not only is the database scalable, but it also provides corporate security and ACID transactions.

3 Application Development and Performance Evaluation

3.1 Proposed Application

For experimentation, we developed a Smart Prescription System web application. It is a simple web application where doctors, patients, and medicine shops are connected to the system. This application intends to bring doctors, patients, and medicine shops to a single platform, and if the prescribed medicine by the doctor is not in the inventory of the medicine shop, they can offer alternative medicine of the same genre. The system is a client-server system where the front-end or client-end will be used by the doctors, patients, and pharmacists, and the system uses the back-end for accessing necessary information. For experimentation, we developed the system with 2 separate back-ends. In the first case, we used Ploystore built on SQLite and MongoDB, and for the latter case, we used a multi-model database (OrientDB) commercially used. The system will handle different kinds of structured data such as patient, doctor's, medicine, medicine store, appointment, transaction, etc. The system also receives data from patients' smartwatches, transections, account credentials, and unstructured logs, which have higher volume, velocity, and a lot of variety. The architecture of the system with polystore database is shown in figure 10, and the architecture of the system with a multi-model database is shown in figure 11.

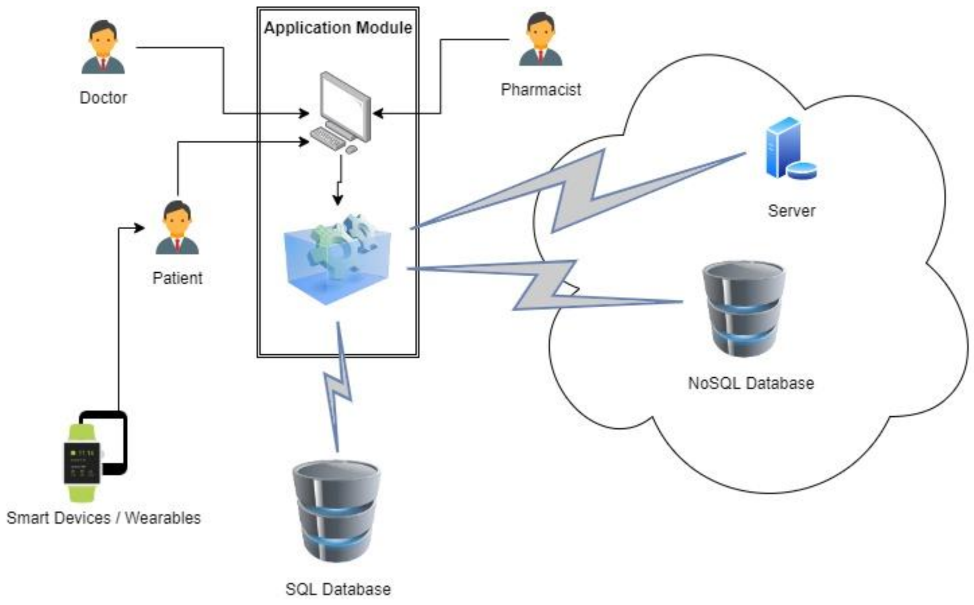


Figure 10: Proposed system architecture with polystore database of the Smart Prescription System

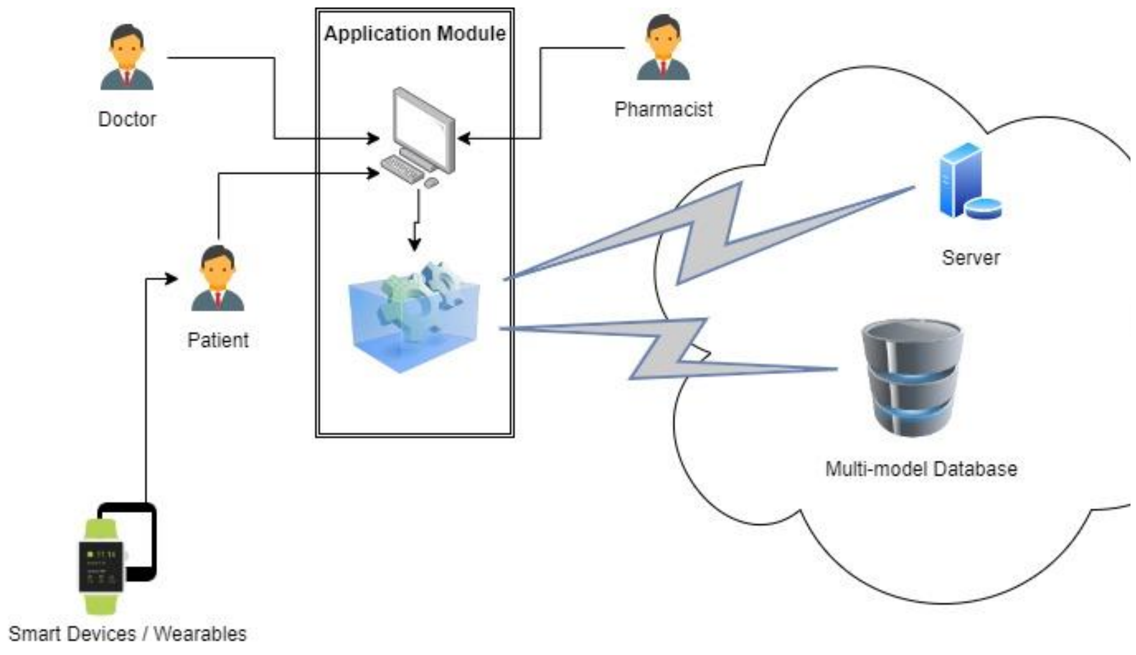


Figure 11: Proposed system architecture with multi-model database of the Smart Prescription System

The following part describes the modules of the system architecture.

3.1.1 Front-end

The front-end is the interactive part of a website visible to users, such as Graphical User Interface (GUI) and Command Line Interface (CLI). In our system, the front-end consists of the application software and computer or laptop, which belongs to either doctor, patient, or pharmacist. The front-end is developed using Next.js Through the application software, the doctors can search in the dashboard, add patients, lookup for appointments and accept them, lookup for patients, their records, including their smart device data, inquire about performing a test, and prescribe medicine. The patient can make and modify appointments, view prescriptions. The medicine stores or pharmacists can search for a patient, view their health record and prescription, and offer alternative medicine of the same generic name if the prescribed medicine is not available. When either of the users attempts to sign in to the system, the application software authenticates and verifies the user credentials.

3.1.2 Backend

The back-end is the code that runs on the webserver and which is not visible to the users. Back-end may include webserver, application software, and databases. A website is a collection of raw materials such as images, database contents, images, etc. and a webserver is a software that serves the webpages of that website. When a user requests for webpage access, the webserver puts all those raw materials in a webpage on its own or fetches from the database and sends it back to the user via a web browser. The purpose of back-end is to generate dynamic web pages. In our system, as mentioned earlier, we have 2 separate back-ends. For the first one with polystore built on MongoDB and SQLite, all the medicine or drug information and prescription which comes in a structured way will be stored in SQLite database, and all the other information such as user data, user health data, wearable device data of the user, diagnosis, appointments, previous appointments, etc. are stored in MongoDB. For the second case, we selected OrientDB as our multi-model database. We stored all the data in this database as it can handle both structured and unstructured data.

3.1.3 System Modelling

At first, the system was modeled using Unified Modelling Language (UML). Figure 12 shows the system's activity diagram, which shows the possible interaction between the system and the users.

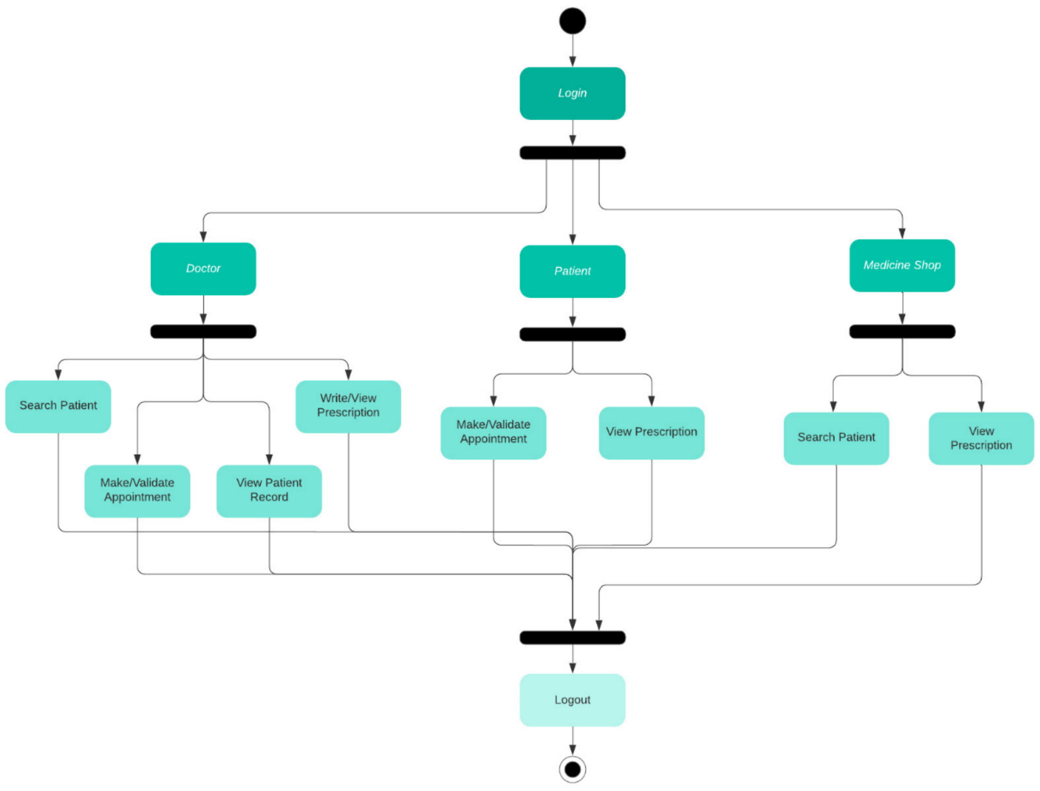


Figure 12: Activity diagram of Smart Medicine Prescription System

Figure 13 shows the use case diagram of the system where actors of the system are patients, doctors, and pharmacists. This figure also shows how the roles of these actors within the system.

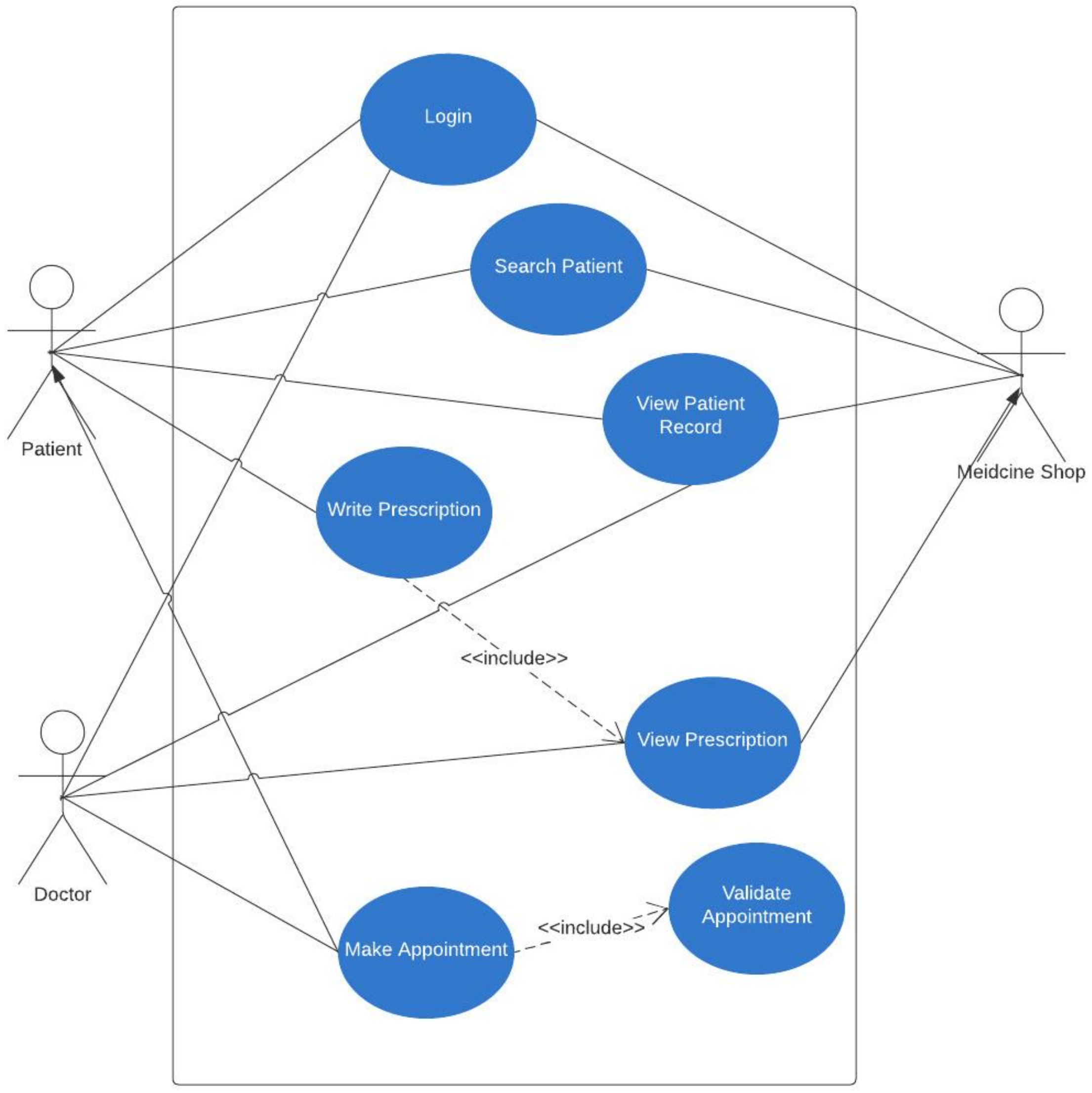


Figure 13: Use case diagram of Smart Medicine Prescription System

The purpose of developing two separate back-ends keeping the front-end same so that we could compare the performance of polystore and multi-model database. The interaction between application and polystore is shown in Figure 14.

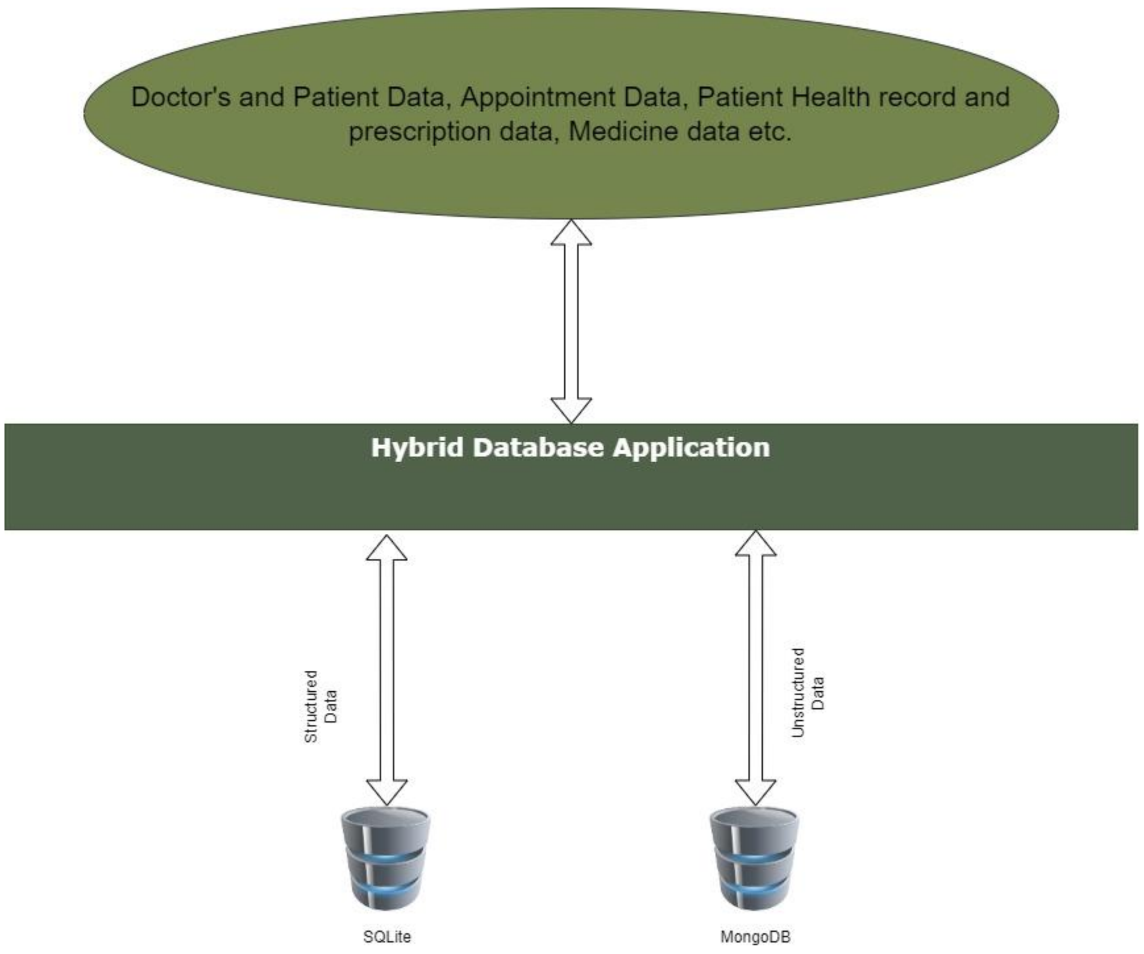

Doctor's and Patient Data, Appointment Data, Patient Health record and prescription data, Medicine data etc.
Hybrid Database Application
Structured Data
Unstructured Data
SQLite
MongoDB

Figure 14: System using SQLite and MongoDB integration (Polystore)

For the second case we used a multi-model database which is OrientDB. The frontend remained same and the interaction between the application and multi-model database is shown in Figure 15.

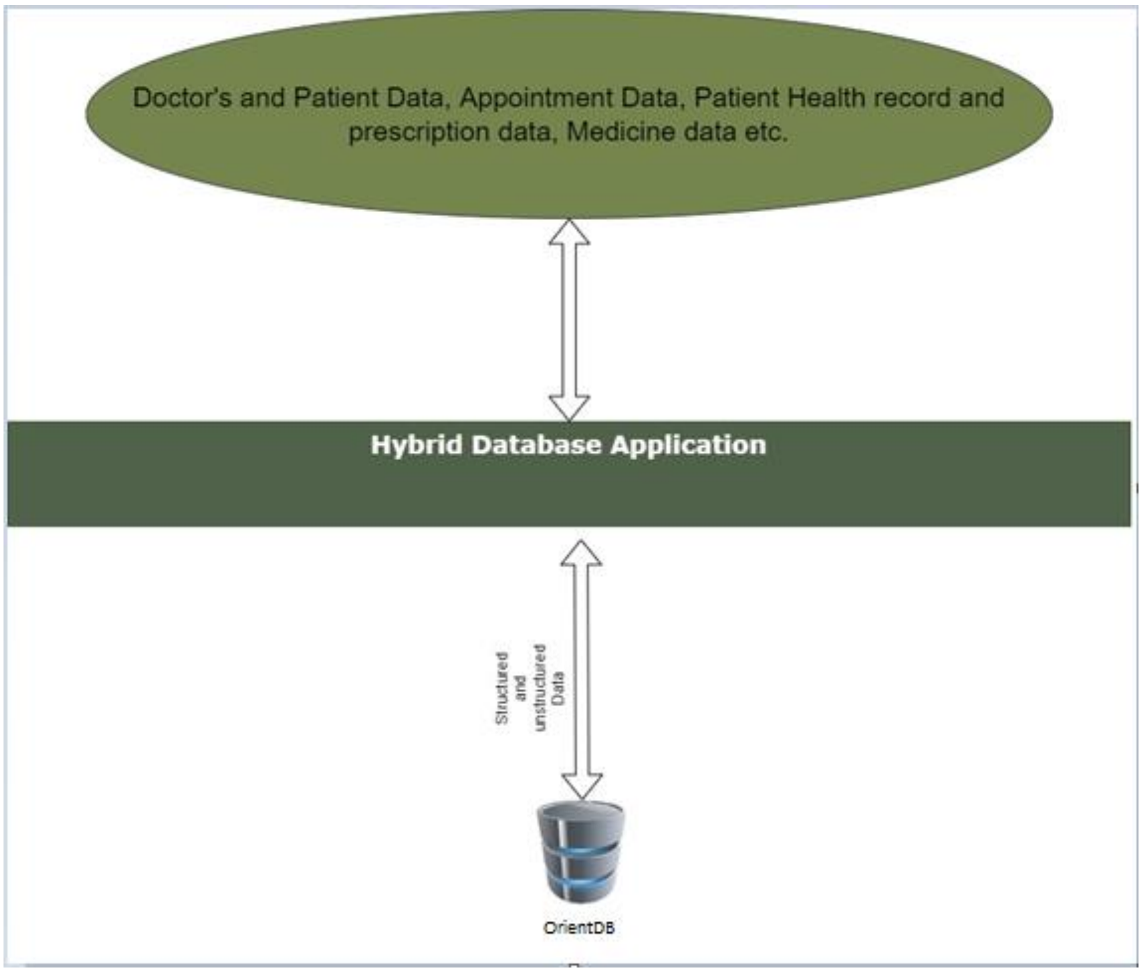
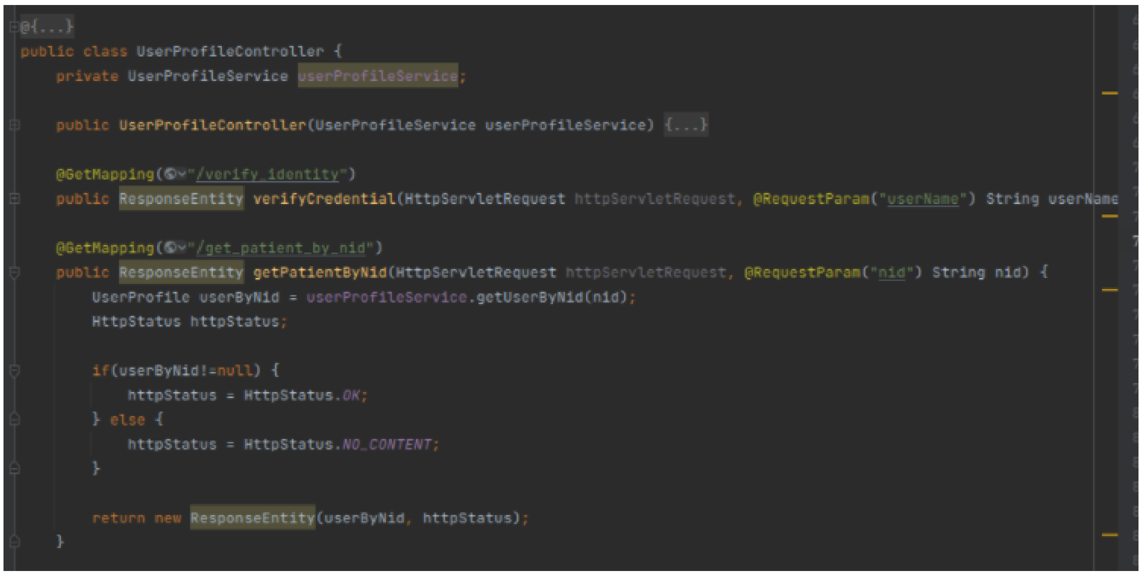

Doctor's and Patient Data, Appointment Data, Patient Health record and prescription data, Medicine data etc.
Hybrid Database Application
Structured and unstructured Data
OrientDB

Figure 15: System using OrientDB (Multi-model Database)

3.1.4 System Development:

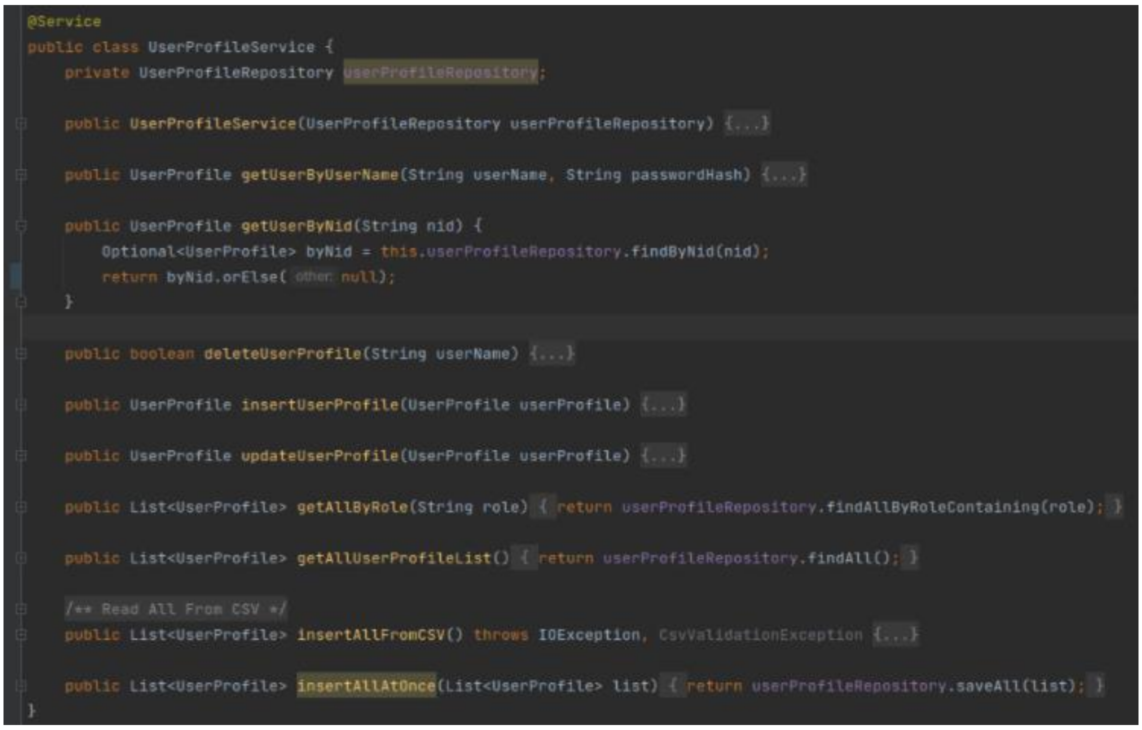
For the front-end, we developed an authentication-based dashboard user interface to maintain user profiles, prescriptions, medical histories, medicine information, etc. It was developed using NextJS. The back-end is developed using used Spring Boot on top of Java. Spring boot is an extension of the Spring Framework. The Relational Database SQLite is connected with the application using JDBC, and MongoDB is connected using proprietary protocol using JSON or Mongo API. For the other back-end with multi-model database, the application is connected using JDBC. Below, some screenshots of the code, interface along with their description, are given.



```java
@{...}
public class UserProfileController {
    private UserProfileService userProfileService;

    public UserProfileController(UserProfileService userProfileService) {...}

    @GetMapping("/verify_identity")
    public ResponseEntity verifyCredential(HttpServletRequest httpServletRequest, @RequestParam("userName") String userName

    @GetMapping("/get_patient_by_nid")
    public ResponseEntity getPatientByNid(HttpServletRequest httpServletRequest, @RequestParam("nid") String nid) {
        UserProfile userByNid = userProfileService.getUserByNid(nid);
        HttpStatus httpStatus;

        if(userByNid!=null) {
            httpStatus = HttpStatus.OK;
        } else {
            httpStatus = HttpStatus.NO_CONTENT;
        }

        return new ResponseEntity(userByNid, httpStatus);
    }
```

Figure 16 : User Profile Controller (API)

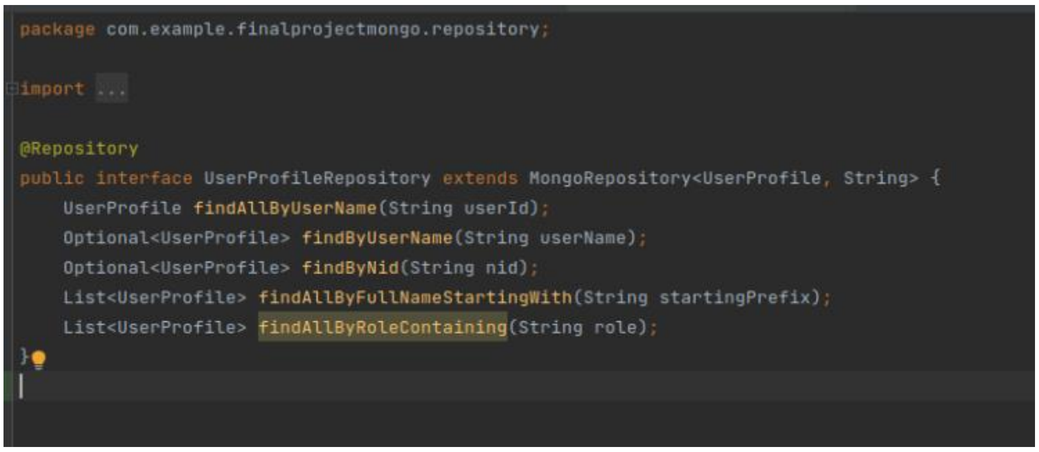
Figure 16 shows the code for the UserProfileController. This part of the code shows how the main APIs are called. We can see that inside the UserProfileController class, there are some APIs such as verify_identity, which is used while logging in to verify the user credentials, get_patient_identity, which is used for searching patients by id from doctor and pharmacist interface. These APIs are used to call different user profile services. Figure 17 shows how we defined the UserProfileService Class.



```java
@Service
public class UserProfileService {
    private UserProfileRepository userProfileRepository;

    public UserProfileService(UserProfileRepository userProfileRepository) {...}

    public UserProfile getUserByUserName(String userName, String passwordHash) {...}

    public UserProfile getUserByNid(String nid) {
        Optional<UserProfile> byNid = this.userProfileRepository.findByNid(nid);
        return byNid.orElse( other: null);
    }

    public boolean deleteUserProfile(String userName) {...}

    public UserProfile insertUserProfile(UserProfile userProfile) {...}

    public UserProfile updateUserProfile(UserProfile userProfile) {...}

    public List<UserProfile> getAllByRole(String role) { return userProfileRepository.findAllByRoleContaining(role); }

    public List<UserProfile> getAllUserProfileList() { return userProfileRepository.findAll(); }

    /** Read All From CSV */
    public List<UserProfile> insertAllFromCSV() throws IOException, CsvValidationException {...}

    public List<UserProfile> insertAllAtOnce(List<UserProfile> list) { return userProfileRepository.saveAll(list); }
}
```

Figure 17: User Profile Service

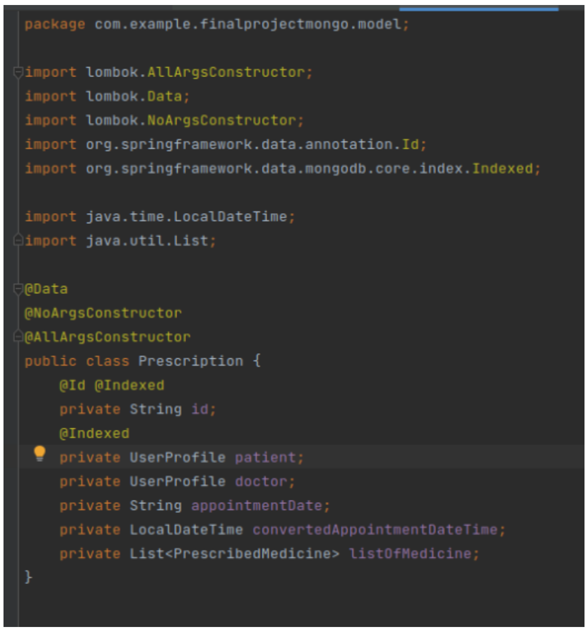
This service aims to get data from the database according to the service call. Some of such services are getUserByuserName used to collect user profile information by name and getUserByNid used to collect user profile information by id. Services get data from the database using UserProfileRepository. Figure 18 shows the interface and repository contents for UserProfileRepository using MongoRepository.



```java
package com.example.finalprojectmongo.repository;

import ...

@Repository
public interface UserProfileRepository extends MongoRepository<UserProfile, String> {
    UserProfile findAllByUserName(String userId);
    Optional<UserProfile> findByUserName(String userName);
    Optional<UserProfile> findByNid(String nid);
    List<UserProfile> findAllByFullNameStartingWith(String startingPrefix);
    List<UserProfile> findAllByRoleContaining(String role);
}
```

Figure 18: User Profile Repository

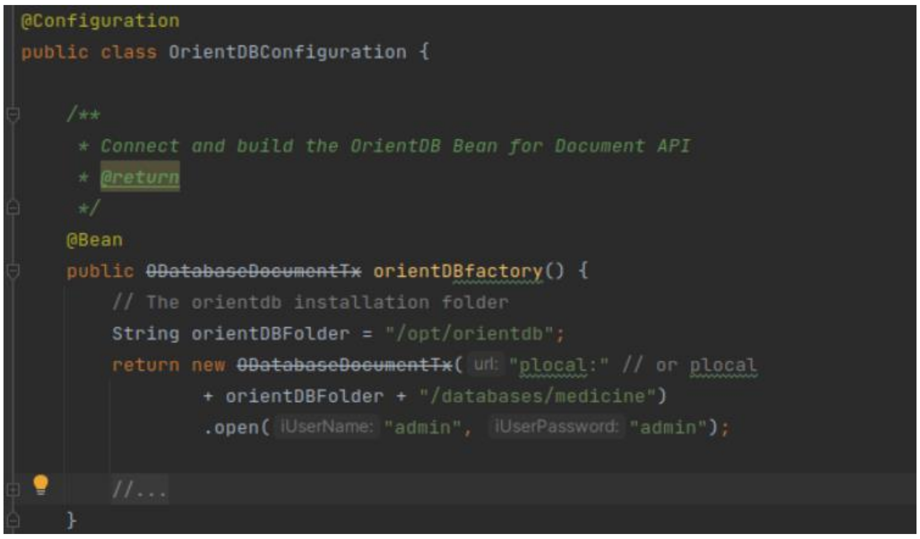
When UserProfileService requests access to data, UserProfileRepository helps to access the database. UserProfileRepository communicates with the database using queries.



```java
package com.example.finalprojectmongo.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.index.Indexed;

import java.time.LocalDateTime;
import java.util.List;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Prescription {
    @Id @Indexed
    private String id;
    @Indexed
    private UserProfile patient;
    private UserProfile doctor;
    private String appointmentDate;
    private LocalDateTime convertedAppointmentDateTime;
    private List<PrescribedMedicine> listOfMedicine;
}
```

Figure 19: Prescription Model class

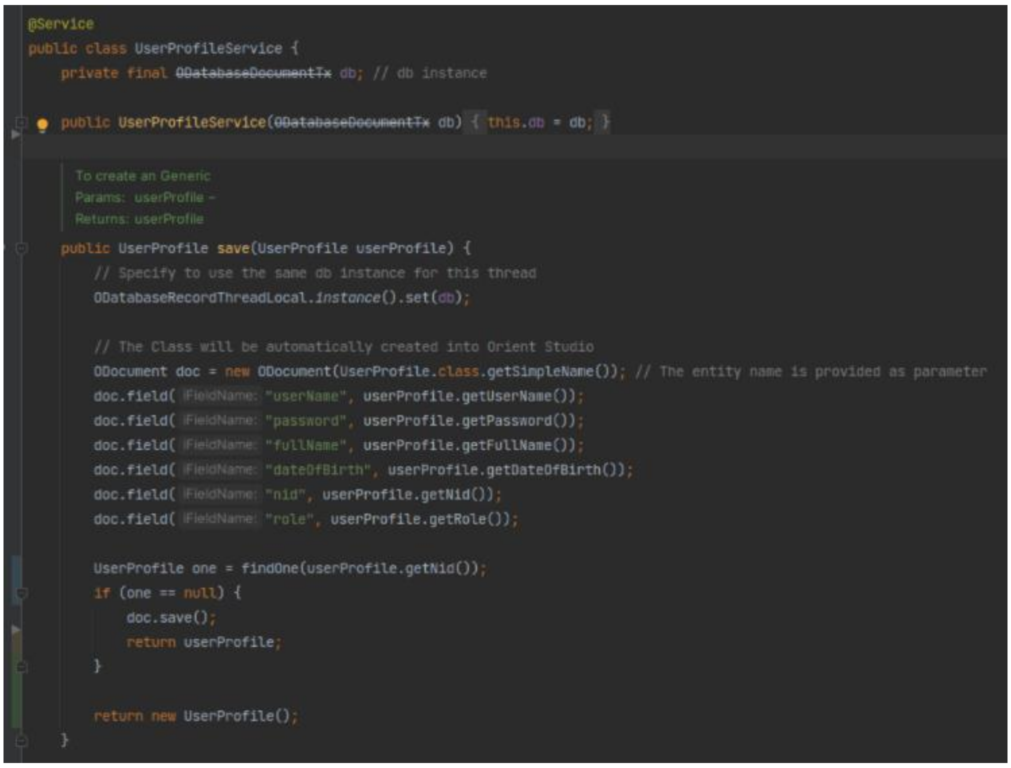
Figure 19 shows the creation of the Prescription class using MongoDB, the information needed for prescription, and libraries that are used for this. For prescription, we initially used information such as doctor's information, patient's information, appointment date, list of medicine, etc.



```
@Configuration
public class OrientDBConfiguration {

    /**
     * Connect and build the OrientDB Bean for Document API
     * @return
     */
    @Bean
    public ~~ODatabaseDocumentTx~~ orientDBfactory() {
        // The orientdb installation folder
        String orientDBFolder = "/opt/orientdb";
        return new ~~ODatabaseDocumentTx~~( url: "plocal:" // or plocal
                + orientDBFolder + "/databases/medicine")
                .open( iUserName: "admin",  iUserPassword: "admin");

        //...
    }
```

Figure 20: OrientDB Establishing Connection

Figure 20 shows how the connection with OrientDB is created. OrientDBConfiguration class was created inside which OrientDB Bean for Document API was connected and built.



```java
@Service
public class UserProfileService {
    private final ~~ODatabaseDocumentTx~~ db; // db instance

    public UserProfileService(~~ODatabaseDocumentTx~~ db) { this.db = db; }

    To create an Generic
    Params: userProfile -
    Returns: userProfile
    public UserProfile save(UserProfile userProfile) {
        // Specify to use the same db instance for this thread
        ODatabaseRecordThreadLocal.instance().set(db);

        // The Class will be automatically created into Orient Studio
        ODocument doc = new ODocument(UserProfile.class.getSimpleName()); // The entity name is provided as parameter
        doc.field( iFieldName: "userName", userProfile.getUserName());
        doc.field( iFieldName: "password", userProfile.getPassword());
        doc.field( iFieldName: "fullName", userProfile.getFullName());
        doc.field( iFieldName: "dateOfBirth", userProfile.getDateOfBirth());
        doc.field( iFieldName: "nid", userProfile.getNid());
        doc.field( iFieldName: "role", userProfile.getRole());

        UserProfile one = findOne(userProfile.getNid());
        if (one == null) {
            doc.save();
            return userProfile;
        }

        return new UserProfile();
    }
```

Figure 21: OrientDB UserProfile saving

Figure 21 shows how userProfile is saved using data such as username, password, full name, date of birth, role, etc. Once a user provides this information, a user profile can be created.

Figure 22 shows the login page of the website. Here the users provide their credentials; after verifying by the system, users are redirected to the home page based on their role. In case of wrong credentials, the system will show an error message.

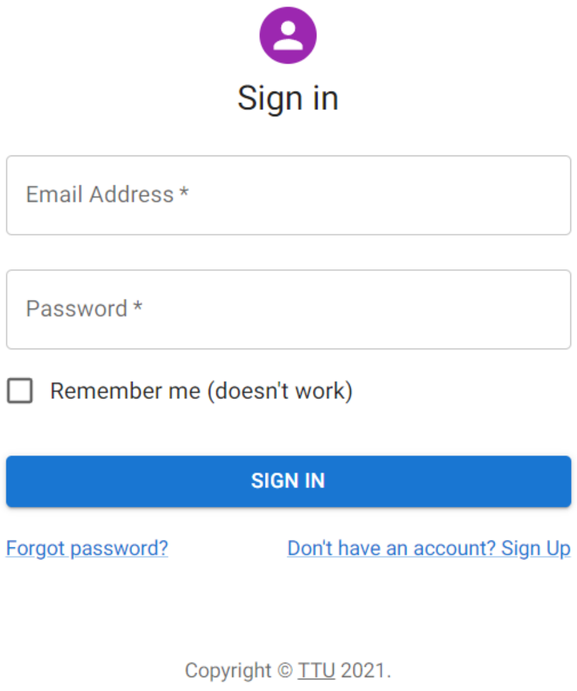
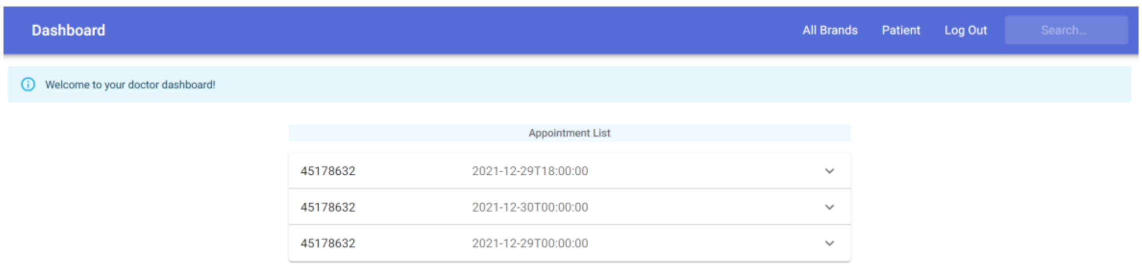


Figure 22: Sign in page of Smart Prescription System

When the doctor is signed in, the appointment of that particular day appears on the homepage. A doctor can click on an appointment and view patient information, health record, order test, view test report, prescribe medicine. The doctor can also add new patients, approve/cancel an appointment, view previous appointments, etc. Figure 23 shows the homepage when a doctor is signed in. Once a doctor generates a prescription, it is automatically saved in the SQLite database.



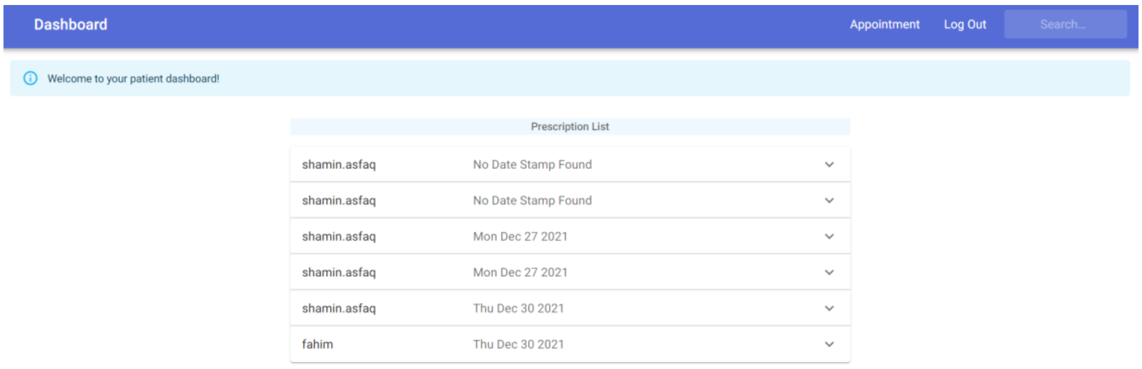
Welcome to your doctor dashboard!

Appointment List

45178632	2021-12-29T18:00:00
45178632	2021-12-30T00:00:00
45178632	2021-12-29T00:00:00

Figure 23: Doctor's Home Page

When the patient is signed in, he or she can only see his personal information, family doctor's information, previous reports, prescription, appointment history, upcoming appointments, etc. Figure 24 shows the patent homepage.



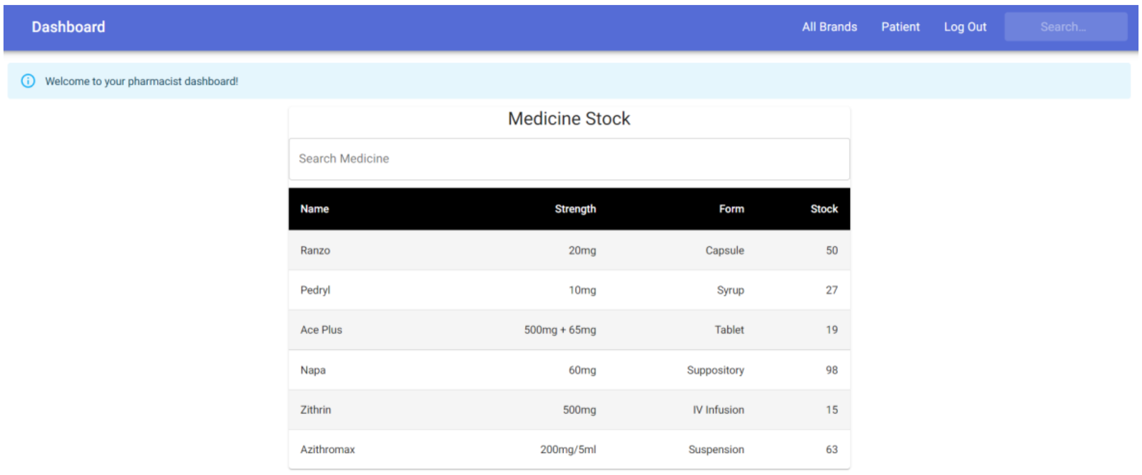
Welcome to your patient dashboard!

Prescription List

shamin.asfaq	No Date Stamp Found
shamin.asfaq	No Date Stamp Found
shamin.asfaq	Mon Dec 27 2021
shamin.asfaq	Mon Dec 27 2021
shamin.asfaq	Thu Dec 30 2021
fahim	Thu Dec 30 2021

Figure 24 : Patient's homepage

Figure 25 shows the homepage when the pharmacist is logged in. A pharmacist can search for a patient in the system using their id code. Once the patient is found in the database and selected, the system shows all the prescriptions about the patient, some information from health records such as allergy and drug immunology. In the prescription, if the pharmacist clicks on a specific drug, the system will show alternative medicines with the same generic name. This way, he can provide other medicine if the prescribed medicine is not in inventory.



Welcome to your pharmacist dashboard!

Medicine Stock

Name	Strength	Form	Stock
Ranzo	20mg	Capsule	50
Pedryl	10mg	Syrup	27
Ace Plus	500mg + 65mg	Tablet	19
Napa	60mg	Suppository	98
Zithrin	500mg	IV Infusion	15
Azithromax	200mg/5ml	Suspension	63

Figure 25: Pharmacist's Homepage

3.2 Description of Data

We collected the data from various sources. Medicine data is collected from the Github repository https://github.com/WSAyan/medicinedb/tree/main/csv where we accessed various data such as different manufacturing companies and id, generic name of medicine and id, the form of medicine, strength of medicine, prices, and packet size, etc. The total number of data accessed from this database is 17590 lines. For the user data, we used https://namso-gen.com/ to generate data of 5000 users, which offered us data such as: name, email, username, gender, age, birthdate, street, city, state, postal code, phone, cell. We also added some other information such as id code, blood group, email address, role, etc.

3.3 Experimentation and result

For evaluating the performance of both kinds of data models, we performed some read, write, query, and delete operations on the system separately for both kinds of data stores. We performed these operations 5 times for each of them and measured the average value. For evaluating the read-write performance of both models, we took three different-sized datasets of user data which have 1100, 2335 and 3671 items respectively. For sql datasets are 9542, 15769 and 17589 respectively.

For performance analysis, the specification of the machine which was used is:

Processor: i5-8400

RAM: 32 GiB

GPU: Zotac GTX 1050Ti

Table 6 Shows the read response time for both types of database solutions.

Table 6. Average Read response time for both type of data stores

Database Model	**Average read response time (s)**		
	First Dataset	**Second Dataset**	**Third dataset**

Polystore (SQLite and MongoDB)	MongoDB: 905 ms SQLite: 262 ms for	MongoDB: 6806 ms SQLite: 228 ms for	MongoDB: 10521 ms SQLite: 286 ms
Multi-model Database (OrientDB)	108 ms	201 ms	138 ms

Table 7. Shows write performance for both kind of database solutions

Database Model	**Average write response time (s)**		
	First Dataset	**Second Dataset**	**Third dataset**
Polystore (SQLite and MongoDB)	SQLite: 193 ms MongoDB: 2709 ms	SQLite: 265 ms MongoDB 3282 ms	SQLite 306 ms MongoDB: 5516 ms
Multi-model Database (OrientDB)	227 ms	378 ms	469 ms

We performed read and write operations on all the databases, where Medicine data in SQLite, prescription, appointment, and user data in MongoDB and all of them separately in OrientDB. From here, we can see that performance of SQLite and OrientDB is much better than MongoDB where we are reading and writing data. When the size of the dataset is increased, read-write performance decrease for all of them, but MongoDB's performance is significantly poor, which will ultimately affect the performance of our Polystore.

Some query operations were performed to fetch information of 1000 users and delete data of 1000 users. Table 5 Shows performance for both kinds of database solutions while performing the query and delete operation.

Table 8. Average Query response time for both type of data stores

Database Model	**Query operation**	**Delete Operation**
Polystore (SQLite and MongoDB)	MongoDB: 1612 ms	MongoDB: 248340 ms
Multi-model Database (OrientDB)	209 ms	30792

It is clearly visible that performance of OrientDB is much better than MongoDB while querying and as a result it will also affect the performance of the polystore. In this case we also can say that OrientDB is a better solution than Polystore.

Table 9. Shows Memory and RAM usage while performing read and write operation on both kind of database solutions

Dataset	Polystore (SQLite and MongoDB)		Multi-model Database (OrientDB)	
	Read Operation	Write Operation	Read Operation	Write Operation
First dataset	58%	48%	105%	98%

Though MongoDB has shown worse performance in all the criteria, when it comes to memory usage OrientDB has utilized much more memory than MongoDB which we can see from table 9. This performance test shows that with suitable optimization strategies, multi-model databases can be better than system developed using polyglot persistence.

Summery

Database selection should be based on the merit of data. Due to lack of flexibility, it is not possible to use a relational database for all business use cases, and for that reason, NoSQL databases evolved. But NoSQL databases cannot fully replace Relational databases because some features provided by Relational databases, such as security, consistency and strict schema etc. cannot be provided by NoSQL databases. So, both have their own use cases in the industry. But when it is about big data, which comes with its V properties, brings new and unique challenges, and to tackle these challenges, we cannot solely depend on a single type of storage solution. Businesses should eventually move to complex systems which use the capabilities of both Relational and NoSQL databases.

Integration of database is a topic of concern, and both multi-model and polyglot persistence is developing and dealing with new challenges. Nowadays many vendors are providing multi-model database services though none of them has proven to be efficient in handling all the challenges. On the other hand, though polyglot persistence database is in an early stage, but some existing models have proven to be promising. Both can serve as excellent beginning points for future research to handle big data.

In this thesis, our developed Smart prescription system was helpful to compare both database integration systems. The system can provide doctors and pharmacists with a well-informed prescription document by using patient health records, appointment records, and medicine knowledgebase. The pharmacist can offer alternative medicine based on the information provided. Though in this system, multi-model database seems to be more efficient but there is other polyglot persistence models that shows promising result. In the future, we can investigate other database integration methods with larger amount of data for better evaluation. Also, we can make the application supported for mobile platform.

References

[1] F. CODD, E. "A relational model of data for large shared data banks". Communications of the ACM. 1983, vol 26, núm. 1, p. 64–69.

[2] PAREDAENS, Jan, VAN GUCHT, Dirk, GYSSENS, Marc y BRA, Paul De. The Structure of the Relational Database Model. Springer Berlin Heidelberg, 2012.

[3] “State of the database access” [Online] Available: https://451research.com/state-of-the-database-landscape (Accessed on: 25.07.21)

[4] CODD, E. F.. The relational model for database management: version 2. Reading, Mass: Addison-Wesley, 1990.

[5] INDRAWAN-SANTIAGO, Maria. Database Research: Are We at a Crossroad? Reflection on NoSQL. IEEE. 2012.

[6] LAKHE, Bhushan. Re-Architecting for NoSQL: Design Principles, Models and Best Practices. Apress, 2016, 117–148.

[7] “NoSQL Database: An Overview” [Online] Available: https://www.thoughtworks.com/insights/blog/nosql-databases-overview (Accessed on 05.08.21)

[8] GESSERT, Felix, WOLFRAM WINGERATH, STEFFEN FRIEDRICH, y NORBERT RITTER. "NoSQL database systems: a survey and decision guidance". Computer Science - Research and Development. 2016, vol 32, núm. 3-4, p. 353–365.

[9] Graph Data Management. Sherif Sakr and Eric Pardede. IGI Global, 2012.

[10] LU, Jiaheng y IRENA HOLUBOVÁ. "Multi-model Databases". ACM Computing Surveys. 2019, vol 52, núm. 3, p. 1–38.

[11] Qu, Xiaona. "Application of Java Technology in Dynamic Web Database Technology." Journal of Physics: Conference Series. Vol. 1744. No. 4. IOP Publishing, 2021.

[12] Shaikh, Sohel S., and Vinod K. Pachghare. "A Comparative Study of Database Connection Pooling Strategy." (2017).

[13] SHUN JING, Xiang y YOU HUO, Meng. "The Application of Database Connection Pool Based on Java in Things Networking". Applied Mechanics and Materials. 2013, vol 432, p. 622–625.

[14] WILLIAMSON, Alan. Java servlets by example. Greenwich, CT: Manning, 1999.

[15] Wu, Amanda, Haibo Wang, and Dawn Wilkins. "Performance Comparison of Alternative Solutions For Web-To-Database Applications–." Proceedings of the Southern Conference on Computing. 2000.

[16] GOEMINNE, Mathieu y TOM MENS. Towards a survival analysis of database framework usage in Java projects. IEEE. 2015.

[17] “What is JDBC? Introduction to Java Database Connectivity” [Online] Available: https://www.infoworld.com/article/3388036/what-is-jdbc-introduction-to-java-database-connectivity.html (Accessed on 06.08.21)

[18] "Java JDBC API" [Online] Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/ (Accessed on 06.08.21)

[19] "Java DriverManager class" [Online] Available: https://www.ibm.com/docs/en/i/7.1?topic=connections-java-drivermanager-class (Accessed on: 06.08.21)

[20] "What is Java Hibernate" [Online] Available: https://www.educba.com/what-is-java-hibernate/ (Accessed on: 06.08.21)

[21] "What is JPA? Introduction to the Java Persistence API" [Online] Available: https://www.infoworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html (Accessed on: 06.08.21)

[22] GADEPALLY, Vijay, PEINAN CHEN, JENNIE DUGGAN, AARON ELMORE, BR, HAYNES, on, JEREMY KEPNER, SAMUEL MADDEN, TIM MATTSON, y MICHAEL STONEBRAKER. The BigDAWG polystore system and architecture. IEEE. 2016

[23] "Microsoft Polybase Documentation." [Online] Available: https://docs.microsoft.com/en-us/sql/relational-databases/polybase/polybase-guide?view=sql-server-ver15 (Accessed on: 14.08.21)

[24] AGRAWAL, Divy, LAMINE BA, LAURE BERTI-EQUILLE, SANJAY CHAWLA, AHMED ELMAGARMID, HOSSAM HAMMADY, YASSER IDRIS, ZOI KAOUDI, ZUHAIR KHAYYAT, SEBASTIAN KRUSE, MOURAD OUZZANI, PAOLO PAPOTTI, JORGE-ARNULFO QUIANE-RUIZ, NAN TANG, y J. ZAKI, Mohammed. Rheem. ACM. 2016.

[25] "Top 7 multi-model database" [Online] Available: https://www.predictiveanalyticstoday.com/top-multi-model-databases/ (Accessed on: 18.08.21)

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Shamchi Hoque Kaify

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Analysis of Using Different Databases and integration of SQL and NOSQL database”, supervised by Vladimir Viies, PhD, Associate Professor.
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

20.12.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Description and Manual

This section describes how our application interested. It contains a step-by-step explanation of how Smart Prescription System works and how to access the codes and demo.

1. The codes of the system can be found in the following Github Repository.

   SQL Backend: https://github.com/ShamchiKaify/final-project.git

   MongoDB Backend: https://github.com/ShamchiKaify/final-project-mongo.git

   FrontEnd: https://github.com/ShamchiKaify/final-project-next

   OrientDB Backend: https://github.com/ShamchiKaify/springboot-orientdb.git

2. The application can be accessed using a web-browser.

3. Video Demo of the project can be found in the following link:

   https://drive.google.com/drive/folders/1Dn_OIEOWT0faUR4dezQynCgLJQtxcuv4?usp=sharing

Appendix 3 – Source Code

Following is the SAVE / INSERT method to create a new user:

```
public UserProfile insertUserProfile(UserProfile userProfile) {
   Optional<UserProfile> byUserName =
userProfileRepository.findByUserName(userProfile.getUserName());
   if (!byUserName.isPresent()) {
       return userProfileRepository.save(userProfile);
   }
   return null;
}
```

It checks through the repository if any user already exists or not and then, conditionally, it saves the new user.

The below code is for fetching a user by NID:

```
public UserProfile getUserByNid(String nid) {
   Optional<UserProfile> byNid = this.userProfileRepository.findByNid(nid);
   return byNid.orElse(null);
}
```

It fetches a user through the repository and if found, it returns the user, else it returns NULL.
This is what the Repository looks like:

```
@Repository
public interface UserProfileRepository extends MongoRepository<UserProfile,
String> {
   UserProfile findAllByUserName(String userId);
   Optional<UserProfile> findByUserName(String userName);
   Optional<UserProfile> findByNid(String nid);
   List<UserProfile> findAllByFullNameStartingWith(String startingPrefix);
   List<UserProfile> findAllByRoleContaining(String role);
}
```

In case of an INSERT method with Generic Medicine, this is how it was done in my project using JPA Repository:

```
@Transactional
public Generic insertGeneric(Generic generic) {
   Generic genericById = getGenericById(generic.getGenericId());
   return genericRepository.save(genericById);
}
```

And this is what the JpaRespository looks like:

```
@Repository
public interface GenericRepository extends JpaRepository<Generic, Long> {
   Optional<Generic> findGenericByGenericId(Long id);
   Optional<Generic> findGenericByGenericNameContaining(String search);
   List<Generic> findAllByGenericId(String id);
}
```

```
This is the Class definition of Generic (Generic Brand Name):
```

```
@Data
@Entity
public class Generic {
   @Id
   private Long genericId;
   private String genericName;
   private String precaution;
   private String indication;
   private String contraIndication;
   private String dose;
   private String sideEffect;
   private String pregnancyCategoryId;
   private String modeOfAction;
   private String interaction;
}
```

Note that the @Data annotation is from lombok to implement the getter and setter automatically, to reduce redundant coding. And the @Entity is to tell our code that this model class is an Entity of the SQL Database.

This is what the Appointment model class looks like:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Appointment {
   @Id
   private String id;
   private String doctorUserName;
   private String patientNid;
   private UserProfile patientProfile;
   private LocalDateTime dateOfAppointment;
   private String dateString;
   private boolean appointmentStatus;
}
```

And this is the UserProfile model class:

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class UserProfile {
   @Id @Indexed
   private String id;
   @NotNull(message = "Username cannot be null")
   private String userName;
   private String password;
   private String fullName;
   private String dateOfBirth;
   @Indexed
   private String nid;
   @NotBlank(message = "Role cannot be null")
   private ROLE role;
}
```

The following is the API through which we can get an Patient with NID:

```
@GetMapping("/get_patient_by_nid")
public ResponseEntity getPatientByNid(HttpServletRequest httpServletRequest,
@RequestParam("nid") String nid) {
   UserProfile userByNid = userProfileService.getUserByNid(nid);
   HttpStatus httpStatus;

   if(userByNid!=null) {
       httpStatus = HttpStatus.OK;
   } else {
       httpStatus = HttpStatus.NO_CONTENT;
   }

   return new ResponseEntity(userByNid, httpStatus);
}
```

It asks the userProfileService to return the user of the given NID. If it is found, we are returning a response with HttpStatus.*OK* but if the user is not found, we are returning HttpStatus.*NO_CONTENT*.

Below is the source API code to get the Alternative Medicine of any given medicine:

```
@GetMapping("get_alternative_medicine/{searchMedicine}")
public ResponseEntity getAlternativeList(HttpServletRequest
httpServletRequest, @PathVariable String searchMedicine) {
   List<Brand> list = brandService.getAlternateList(searchMedicine);

   HttpStatus httpStatus;

   if(list!=null) {
       httpStatus = HttpStatus.OK;
   } else {
       httpStatus = HttpStatus.NO_CONTENT;
   }

   return new ResponseEntity(list, httpStatus);
}
```

It receives a medicine name as @PathVariable and through the **BrandService,** it fetches a list of alternate medicine list and returns it to the user.