HYBRID JSON DATABASE (TEXT BASED DATABASE)

By

Samuel Jarai

H150346E

HIT400 Capstone project Submitted in Partial Fulfillment of the

Requirements of the degree of

Bachelor of Technology

In

Software Engineering

In the

School of Information Sciences and Technology

Harare Institute of Technology

Zimbabwe

Supervisor

……………………………MR…MANJORO………………………………….

May/2023

ITEM	TOTAL MARK /%	ACQUIRED/%
PRESENTATION- Format-Times Roman 12 for ordinary text, Main headings Times Roman 14, spacing 1.5. Chapters and sub-chapters, tables and diagrams should be numbered. Document should be in report form. Range of document pages. Between 50 and 100.Work should be clear and neat	**5**	
Pre-Chapter Section Abstract, Preface, Acknowledgements, Dedication & Declaration	**5**	
Chapter One-Introduction Background, Problem Statement, Objectives – smart, clearly measurable from your system. Always start with a TO… Hypothesis, Justification, Proposed Tools Feasibility study: Technical, Economic & Operational Project plan –Time plan, Gantt chart	**10**	
Chapter Two-Literature Review Introduction, Related work & Conclusion	**10**	
Chapter Three –Analysis Information Gathering Tools, Description of system Data analysis –Using UML context diagrams, DFD of existing system Evaluation of Alternatives Systems, Functional Analysis of Proposed System-Functional and Non-functional Requirements, User Case Diagrams	**15**	
Chapter Four –Design Systems Diagrams –Using UML Context diagrams, DFD, Activity diagrams Architectural Design-hardware, networking Database Design –ER diagrams, Normalized Databases Program Design-Class diagrams, Sequence diagrams, Package diagrams, Pseudo code Interface Design-Screenshots of user interface	**20**	
Chapter Five-Implementation & Testing Pseudo code of major modules /Sample of real code can be written here Software Testing-Unit, Module, Integration, System, Database & Acceptance	**20**	
Chapter Six –Conclusions and Recommendations Results and summary, Recommendations & Future Works	**10**	
Bibliography –Proper numbering should be used Appendices –templates of data collection tools, user manual of the working system, sample code, research papers	**5**	
	100	**/100**

HIT 400 /200 Project Documentation Marking Guide

Certificate of Declaration

This is to certify that work entitled "HIT400 Research Topic " *is submitted in partial fulfillment of the requirements for the award of Bachelor of Technology (Hons) in Software Engineering ,Harare Institute of Technology .It is further certified that no part of research has been submitted to any university for the award of any other degree .*

(Supervisor) Signature……………………………..
Date……………………….

(Mentor) Signature……………………………..
Date………………………

(Chairman) Signature………………………………..
Date………………………..

Abstract

This capstone project explores the design and implementation of a JSON based hybrid database that combines the features of client-server and server-less databases. A client-server database requires a separate server process to manage the data access requests from the client, while a server-less database runs as an embedded library or a cloud service that scales automatically. The hybrid database aims to offer advantages such as performance, reliability, flexibility and cost-efficiency for different types of applications. The project describes the architecture, data model, query language and API of the hybrid database, as well as evaluates its functionality and performance using various benchmarks and use cases

Preface

This document presents the design and implementation of a JSON-based hybrid database system that combines the features of client-server and serverless databases. The system is designed to provide performance, reliability, flexibility, and cost-efficiency for different types of applications. The system architecture, data model, query language, and API were developed as part of a software engineering capstone project under the supervision of Mr. Manjoro. The project involved a thorough investigation of existing database management systems and their features, as well as an in-depth analysis of the requirements and constraints of the target application domain. The implementation of the system was carried out using modern software development practices and tools, and the functionality and performance of the system were evaluated using various benchmarks and use cases.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Mr. Manjoro, for his invaluable guidance, support, and encouragement throughout this project. His expertise, insights, and feedback were critical in shaping the design and implementation of the system and in ensuring its functionality and performance. I am also grateful to the faculty and staff of the computer science department for their assistance and resources, and to my colleagues for their collaboration and feedback. Finally, I would like to thank my family and friends for their love and support throughout my academic journey.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Introduction

The rapid growth of data generation and consumption in the modern world has led to an increased demand for efficient and flexible database management systems. Databases are essential tools for storing, organizing, and accessing data in various applications. However, different applications may have different requirements and preferences for database features, such as data types, query languages, scalability, and security. Therefore, it is desirable to have a database management system that can adapt to the needs of the users without compromising the performance of the system.The aim of this software engineering capstone project is to design and implement a JSON-based hybrid database system that combines the features of client-server and serverless databases. The system is designed to provide performance, reliability, flexibility, and cost-efficiency for different types of applications. The system architecture, data model, query language, and API will be developed with the goal of creating a database management system that is highly scalable, easily maintainable, and interoperable with other systems and services.The project will involve a thorough investigation of existing database management systems and their features, as well as an in-depth analysis of the requirements and constraints of the target application domain. The implementation of the system will be carried out using modern software development practices and tools, and the functionality and performance of the system will be evaluated using various benchmarks and use cases.The outcome of this project will be a functional and efficient JSON-based hybrid database system that can be used in various applications, from small-scale personal projects to large-scale enterprise systems. The system will provide a flexible and scalable solution for managing complex data structures and processing complex queries, while maintaining low hardware requirements and ensuring high availability and fault tolerance. The project will also contribute to the knowledge and understanding of database management systems and their design and implementation.

Background

The rapid growth of data generation and consumption in the modern world poses new challenges and opportunities for database management systems. Databases are essential tools for storing, organizing and accessing data in various applications. However, different applications may have different requirements and preferences for database features, such as data types, query languages, scalability and security. Therefore, it is desirable to have a database management system that can adapt to the needs of the users without compromising the performance of the system. This project aims to create such a database management system from scratch using JSON as the data format. JSON is a popular and flexible data format that can represent various types of data structures and support different query languages. The project will also explore how to combine the advantages of client-server and server-less databases in a hybrid database architecture that can offer performance, reliability, flexibility and cost-efficiency for different types of applications.

Problem Statement

Database management systems vary in their features and performance depending on their architecture and design. Client server databases, such as PostgreSQL and MySQL, offer more features such as remote access, concurrency control and advanced query languages, but they also incur more performance overhead and require more hardware resources. Server-less databases, such as SQLite, offer less features but have higher performance and lower hardware requirements. However, SQLite does not support remote access or concurrency well, as it is essentially a file-based database with no server process. This limits its usability for applications that need to access data from multiple devices or users simultaneously. Therefore, there is a trade-off between the features and performance of database management systems that depends on the application needs and the hardware capabilities. There is a need for a database management system that can bridge the gap between complex and simple databases by providing some of the essential features of client server databases while maintaining high performance and low hardware requirements. Such a database management system would be suitable for applications that run on low-end hardware devices such as Raspberry Pi but also need to access data remotely or concurrently

Aims and Objectives

- To create a storage engine.
- To allow for basic authentication in order to access the database system.
- To create a user interface for creating and managing databases and tables.
- To allow remote access of the data via API calls.

Hypothesis

The system that will be developed will have performance closer to lightweight databases whilst incorporating features from complex databases like remote access.

Justification

Many users rely on low-end hardware like Raspberry pi for their projects and applications. However, these devices cannot handle complex databases that offer advanced features and functionality. They either run into performance issues or fail to run them at all. The only option they have is to use stripped-down databases like SQLite that only offer basic reading and writing operations. This limits their ability to manipulate and analyse their data effectively and remotely.

Proposed Tools

Hardware

- Laptop 8 Gb Ram ,Core i5 2Ghz 8th Gen , 256GB SSD

Software

- Python
- Visual studio code
- VMware
- GitHub version control
- Postman API Client

Feasibility study

Technical feasibility

To ensure successful development, it is crucial to assess the technical feasibility of the project. This involves checking whether the current technology can support the development process and determining the availability of necessary tools and technical expertise. In this case, the required tools for the project are readily available, and some are even open source. The IDEs for the languages to be used are also available, and the hardware and software requirements for development are well-defined.

Economic feasibility

Economic feasibility is a critical aspect of any project, and it is essential to determine whether the cost of developing the system will exceed the proposed budget. A cost-benefit analysis is performed to determine the feasibility of the project. In this case, the analysis has shown that it is feasible to develop the application within the budget. The software required for development is mostly open source and readily available, making it easy and cheap to acquire. Additionally, the risks associated with the project's development are minimal compared to the benefits it will provide.

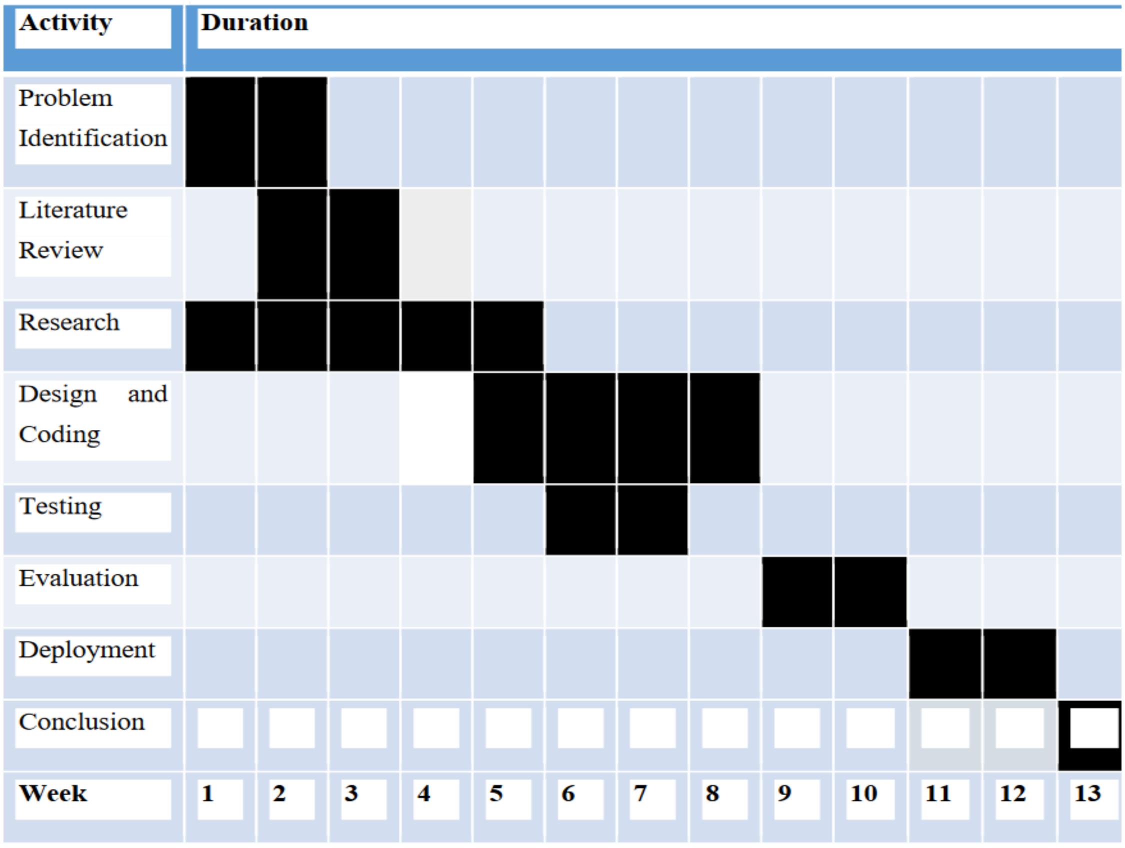
Operational feasibility

Operational feasibility is a measure of how well a proposed system solves the problems, and takes advantage of the opportunities identified during scope definition and how it satisfies the requirements identified in the requirements analysis phase of system development. It answers question “will the system work?” and how well the system would fit into the current business structures and once implemented how useable would it be. Operational feasibility is dependent
on human resources available for the project and involves projecting whether the system will be used if it is developed and implemented. After analyzing these operational requirements, we
see that the proposed system is operationally feasible.

Project plan

A project plan is a formal document designed to guide the control and execution of a project. A project plan is the key to a successful project and is the most important document that needs to be created when starting any business project

Schedule of activities and Gantt chart



Activity	**Duration**												
Problem Identification	■	■											
Literature Review		■	■										
Research	■	■	■	■	■								
Design and Coding					■	■	■	■					
Testing						■	■						
Evaluation									■	■			
Deployment											■	■	
Conclusion													■
Week	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**

Figure 1: Fig 1.1 Gnannt Chart

Chapter 2 Literature Review

Database Speed Comparison by SQLite Team

The relative performance of SQLite 2.7.6, PostgreSQL 7.1.3, and MySQL 3.23.41 was evaluated through a set of tests. The general conclusions from these trials are as follows:

For the majority of typical tasks, SQLite 2.7.6 performs noticeably better than the default PostgreSQL 7.1.3 installation on RedHat 7.2 (often up to 10 or 20 times faster).For the majority of typical operations, SQLite 2.7.6 frequently outperforms MySQL 3.23.41 (in some cases by more than twice as much).The other databases perform CREATE INDEX and DROP TABLE operations more quickly than SQLite. However, given the rarity of certain operations, this is not thought to be an issue. The best results with SQLite come from combining several operations within a single transaction.[1]

The following limitations apply to the findings provided here:

These tests did not attempt to evaluate the performance of many users or the optimization of large queries with numerous joins and subqueries . These tests use a database that is just about 14 megabytes in size. They don't assess how effectively database engines scale to more complex issues.

Testing Conditions:

The testing system consists of an IDE disk drive, a 1.6GHz Athlon processor, and 1GB of RAM. RedHat Linux 7.2 is the operating system, and a stock kernel is used. The RedHat 7.2 default PostgreSQL and MySQL servers were utilized. Versions 7.1.3 and 3.23.41 of PostgreSQL and MySQL, respectively. These engines received no tuning attention. Particularly take note that RedHat 7.2's default MySQL configuration does not support transactions. SQLite can still compete on the majority of tests, however MySQL has a significant speed advantage due to not having to support transactions.[1]

Performance comparison for different types of databases by Sofiia-Valeriia KHOLOD

They made the decision to gauge reaction time and throughput. The tests' applicability to problems encountered in real life was considered when selecting the type. They've chosen to run experiments that mirror social network posts or news management because of this—when there are lots of single reads and few deletions.[2]

Tests

They have chosen to run the following straightforward CRUD tests:

• CREATE - basic setup: truncate table, count the number of records created in a predetermined amount of time (or average requests per second).

• READ - initial setup: truncate table and generate one record; count the number of records read in a predetermined amount of time.

• UPDATE - basic setup: truncate table and generate one record; count the number of records that were updated in a defined amount of time (boolean field was being updated for simplicity).

• DELETE - count the number of records that were removed in a predetermined amount of time; initial setup: truncate table and create sufficient records

Results

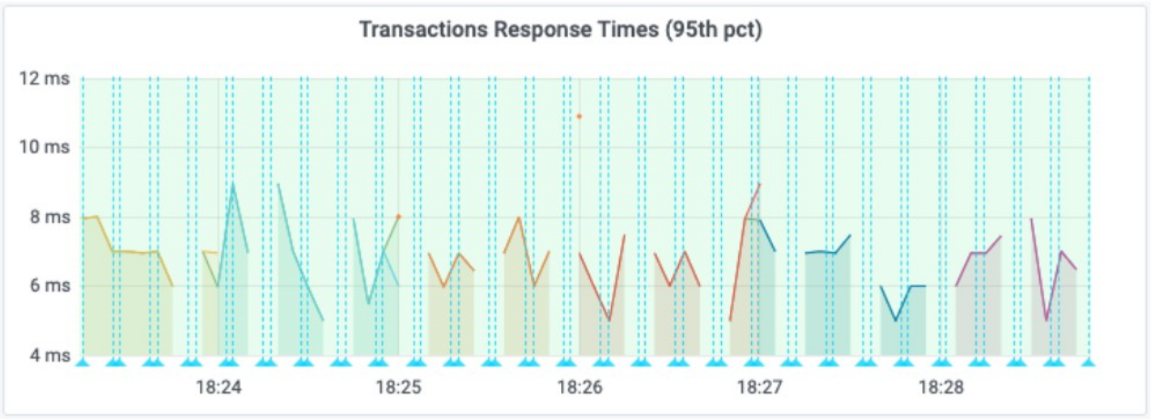


Figure 2: Fig 2.1 PostgreSQL CREATE test response time metric

This graph shows in milliseconds the average time it took to execute a create query on the PostgreSQL DBMS.

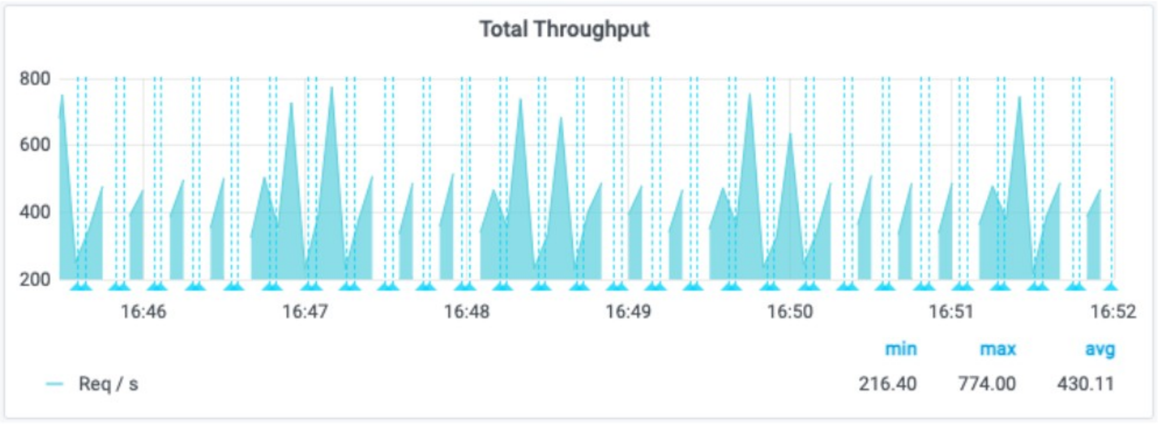


FIGURE 6.11: MongoDB CREATE test, response throughput metric

Figure 3: Fig 2.2 MongoDB CREATE test, response throughput metric

Fig 2.2 Show the amount of create queries that could be parsed per unit time.

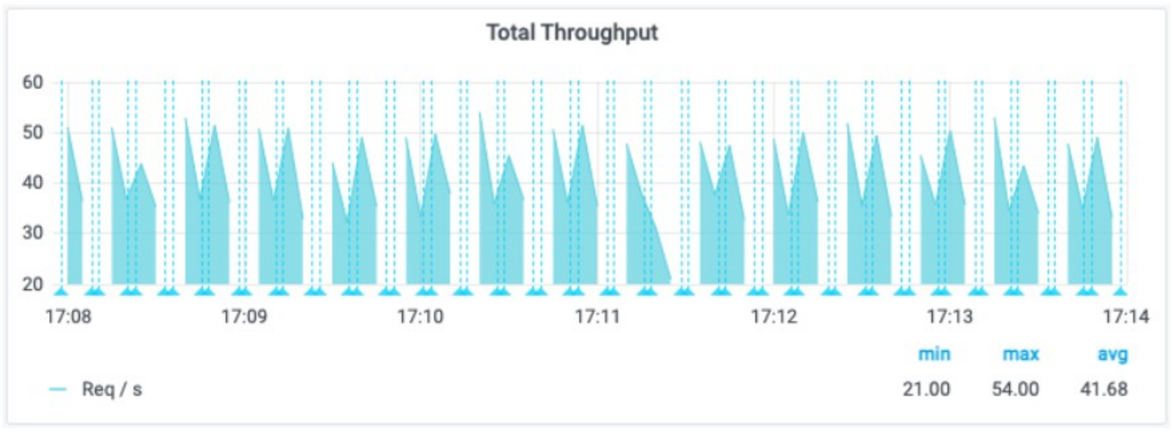


FIGURE 6.12: MySQL CREATE test, throughput metric

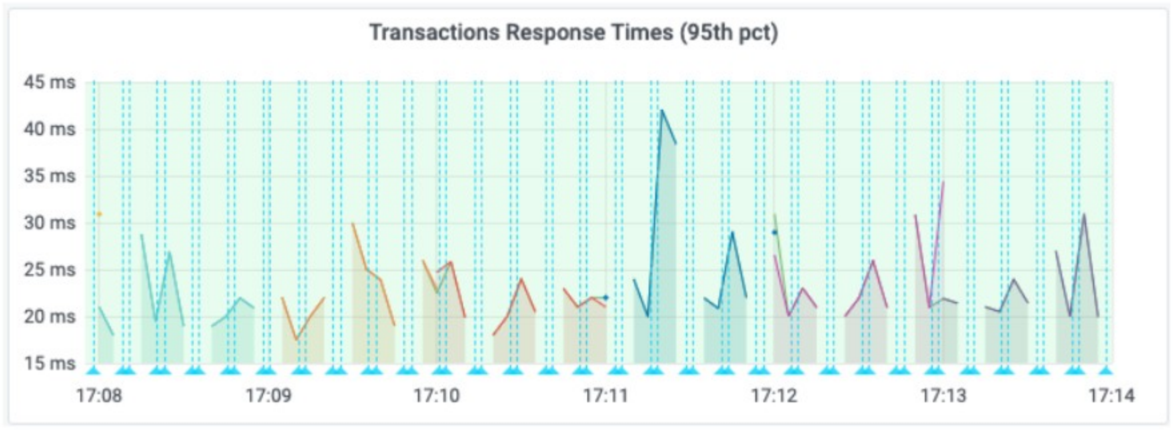


Figure 4: Fig 2.3 MySQL CREATE test, response time metric

Fig 2.3 Show the the performance of the MySQL database for create queries.

Non-relational databases show a clear performance advantage over relation databases.

Comparative Analysis of MySQL and SQLite Relational Database Management Systems by Jayesh Umre and Others

This paper investigates the comparative performance of the MySQL and SQLite relational database management systems (RDBMSs) in a Windows10 environment. The authors conducted extensive tests by varying the number of operations performed, the data size, and the number of clients to find the most efficient RDBMS system. The results showed that while SQLite performed better than MySQL in low-intensity tasks and with small databases, MySQL was better suited for high-intensity tasks and with larger databases. Furthermore, the authors concluded that MySQL was better at managing network traffic in a multi-client environment. This paper provides valuable insight into the relative strengths and weaknesses of MySQL and SQLite in a Windows10 environment.

In terms of query performance SQLite had a 300% performance advantage in the select query but only varied by 20-30% in all other queries.[3] The diagrams below indicate the query performance disparity.

Select Query

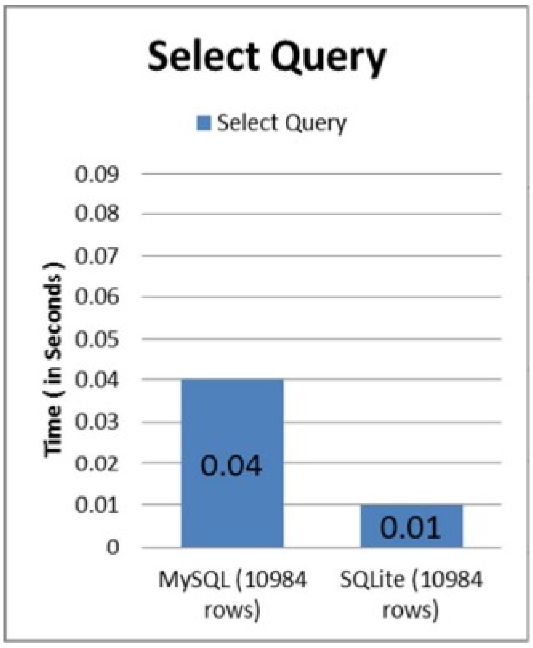


Figure 5: Fig 2.3 MySQL vs SQLite Select Query Comparison

This test measure how fast these 2 databases could perform the select query on 10984 records.

Insert Query

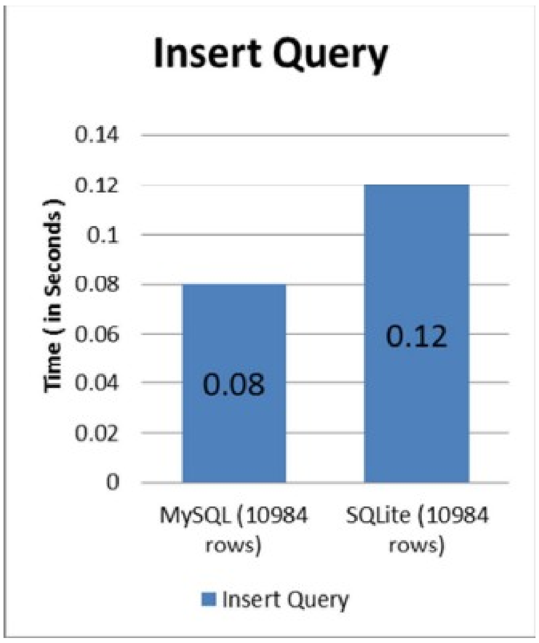


Figure 6: Fig 2.4 MySQL vs SQLite Insert Query Comparison

Update Query

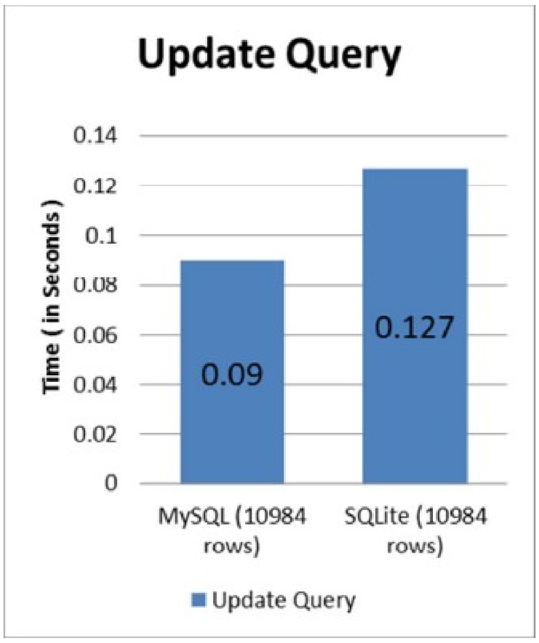


Figure 7: Fig 2.5 MySQL vs SQLite Update Query Comparison

Delete Query

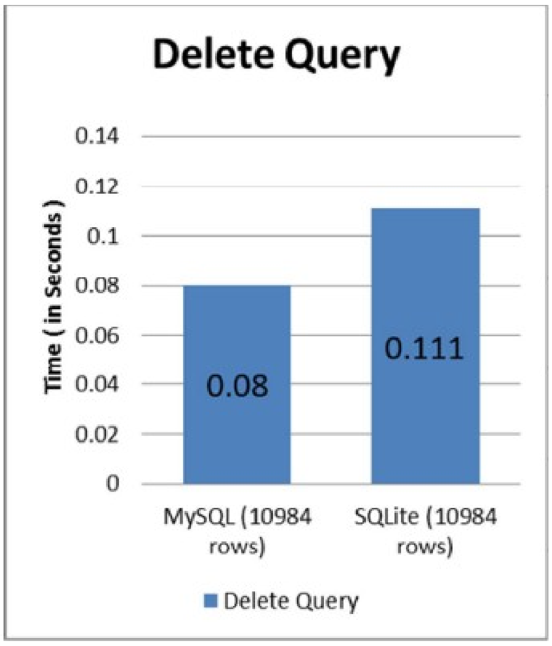


Figure 8: Fig 2.6 MySQL vs SQLite Delete Query Comparison

Fig 2.6 Show the amount of time it takes to delete 10984 records in SQLite and MySQL.

Concurrency Control in Distributed Database Systems by Philip A. Bernstein and Nathan Goodman

This paper surveys, consolidates, and presents the state of the art in distributed database concurrency control. The paper decomposes the concurrency control problem into two major subproblems: read-write and write-write synchronization.[8] It describes a series of synchronization techniques for solving each subproblem and shows how to combine these techniques into algorithms for solving the entire concurrency control problem. Such algorithms are called "concurrency control methods." The paper describes 48 principal methods, including all practical algorithms that have appeared in the literature plus several new ones. The paper concentrates on the structure and correctness of concurrency control algorithms. Issues of performance are given only secondary treatment.

The paper is relevant to this topic because it addresses the trade-off between the features and performance of database management systems that depends on the application needs and the hardware capabilities. It also discusses how to provide some of the essential features of client server databases such as remote access and concurrency control while maintaining high performance and low hardware requirements.

Database Management Systems (DBMS) Comparison: MySQL, PostgreSQL, MSSQL Server, MongoDB, Elasticsearch, and others by AltexSoft.

This paper compares the 10 most commonly used database management systems: MySQL, MariaDB, Oracle, PostgreSQL, MSSQL, MongoDB, Redis, Cassandra, Elasticsearch, and Firebase. [9]The paper provides an overview of each DBMS and its features, such as data model, query language, scalability, performance, security, and licensing. The paper also discusses the pros and cons of each DBMS and provides some use cases and recommendations for choosing the best DBMS for different scenarios.

The paper concludes that there is no one-size-fits-all solution when it comes to choosing a DBMS. Different DBMSs have different strengths and weaknesses, and the best choice depends on the specific requirements and preferences of the application and the developer. The paper also suggests some general guidelines for choosing a DBMS, such as considering the data model, the query language, the scalability, the performance, the security, and the licensing of each DBMS.

The paper is relevant to this topic because it provides a comprehensive comparison of different database management systems and their features and performance. It also helps to evaluate the trade-off between the features and performance of database management systems that depends on the application needs and the hardware capabilities.

Database Management System Performance Comparisons: A Systematic Survey by Toni Taipalus2.

This paper systematically synthesizes the results and approaches of studies that compare DBMS performance and provides recommendations for industry and research.[10]The paper shows that performance is usually tested in a way that does not reflect real-world use cases, and that tests are typically reported in insufficient detail for replication or for drawing conclusions from the stated results. The paper also identifies some challenges and gaps in the current state of DBMS performance comparison studies and suggests some directions for future research.

The paper concludes that DBMS performance comparison studies are often conducted in a way that does not reflect real-world use cases, and that they are typically reported in insufficient detail for replication or for drawing conclusions from the stated results. The paper also identifies some challenges and gaps in the current state of DBMS performance comparison studies, such as the lack of standardized benchmarks, the diversity of performance metrics, the variability of hardware and software configurations, and the influence of human factors. The paper suggests some directions for future research, such as developing more realistic and representative benchmarks, using more comprehensive and consistent performance metrics, reporting more details and metadata of the experiments, and applying more rigorous and transparent methods for data analysis and synthesis.

The paper is relevant to this topic because it provides a critical analysis of the existing DBMS performance comparison studies and their limitations. It also highlights some of the factors that affect the performance of different database management systems in various contexts.

Conclusion

Hybrid databases combine the benefits of serverless and client-server architectures, such as scalability, flexibility, security, and performance. Serverless databases enable on-demand provisioning and execution of data processing functions, while client-server databases provide persistent storage and query capabilities. Hybrid databases can run on lightweight hardware such as IoT devices, which reduces latency, bandwidth consumption, and operational costs.

However, the literature review has also identified some challenges and gaps in the current research on hybrid databases. These include:

- The lack of a unified framework and methodology for designing, developing, deploying, and evaluating hybrid databases for IoT applications.

- The trade-offs and optimization techniques for balancing the conflicting requirements of serverless and client-server components, such as availability, consistency, reliability, and efficiency.

- The integration and interoperability issues between hybrid databases and other cloud services and platforms, such as streaming, analytics, and machine learning.

- The security and privacy risks and solutions for hybrid databases in IoT contexts, such as data encryption, access control, and auditing.

Therefore, the literature review confirms the necessity of hybrid databases for IoT applications, but also suggests the need for further research and development in this area. Future work should address the challenges and gaps identified in the literature review, and propose novel solutions and best practices for hybrid database design and implementation for IoT scenarios.

Chapter 3 Analysis

Introduction

Database management systems (DBMS) are software applications that allow users to create, store, manipulate and retrieve data in a structured and efficient way. There are many types of DBMS available in the market, each with its own advantages and disadvantages. Some of the most popular ones are MySQL, PostgreSQL, MongoDB, Oracle and Sqlite. In this section, we will analyse these existing DBMS solutions based on their features, performance, scalability and suitability for low-end hardware devices. We will also identify the gaps and challenges that they pose for users who need a simple yet powerful DBMS for their projects and application

Description of Existing Systems

A database management system (DBMS) is a software that manages data stored in a database. A client-server DBMS is a distributed structure where the DBMS resides on a server and client applications access the database over a network. Examples of client-server DBMS are MySQL, Oracle, SQL Server, etc An in-process DBMS is a monolithic structure where the DBMS and the application run in the same process. An embedded database is a type of in-process DBMS that is tightly integrated with an application software and does not require installation or configuration. Examples of embedded databases are SQLite, SQL Compact, Berkeley DB, etc.

Description of Proposed System

I propose developing an innovative text based database system that will enable users to access and manipulate data locally via a simple python library of my design. The system will store the data in JSON files with objects that define the tables, their schemas and the records in said tables. This will allow for easy data serialization and de-serialization, as well as flexibility and scalability. The system will also provide remote access to the data via a simple python server that will accept and share data through rest API. The server will also offer basic authentication to ensure data security and integrity. In addition, I will create a user-friendly web GUI for managing your databases, such as creating, deleting, updating and querying tables and records. This system will offer several benefits, such as low overhead, high performance, portability and compatibility.

Information Gathering Tools

Information gathering techniques used:

Interviews

To elicit user requirements for the project, the developer conducted interviews with three Engineers who represented different use cases and pain points of the current databases. The developer aimed to understand the functionality, usability, reliability and performance of the existing solutions, as well as the desired features and enhancements for the new system. The interviews provided rich qualitative data that helped the developer identify and prioritize user needs and preferences.

Data Analysis

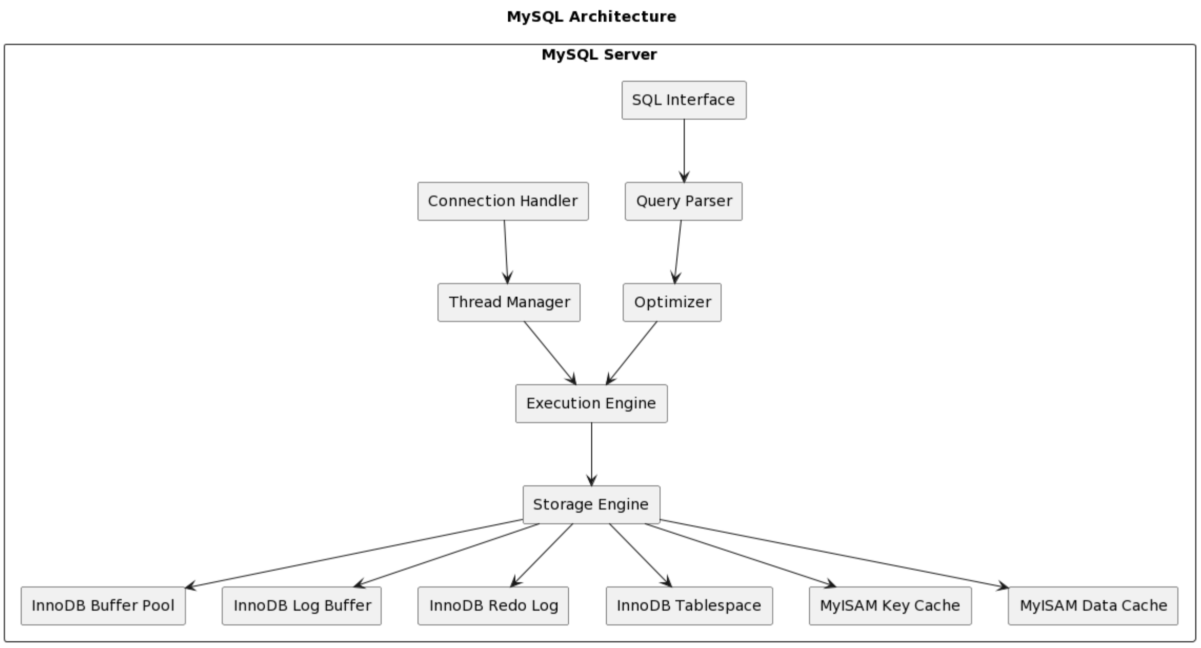


Figure 9: Fig3.1 MySQL Structure Diagram

Fig 3.1 Shows the structure diagram of the MySQL Database Management System

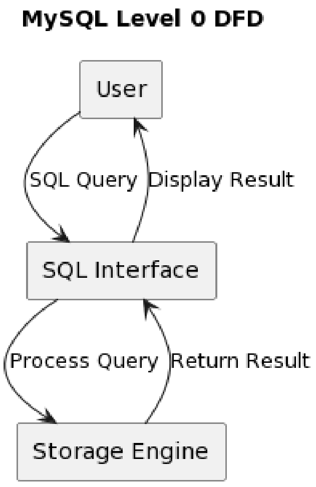


Figure 10: Fig3.2 MySQL DFD LEVEL 0

Fig 3.2 Shows the Level 0 Data Flow Diagram for the MySQL Database Management System

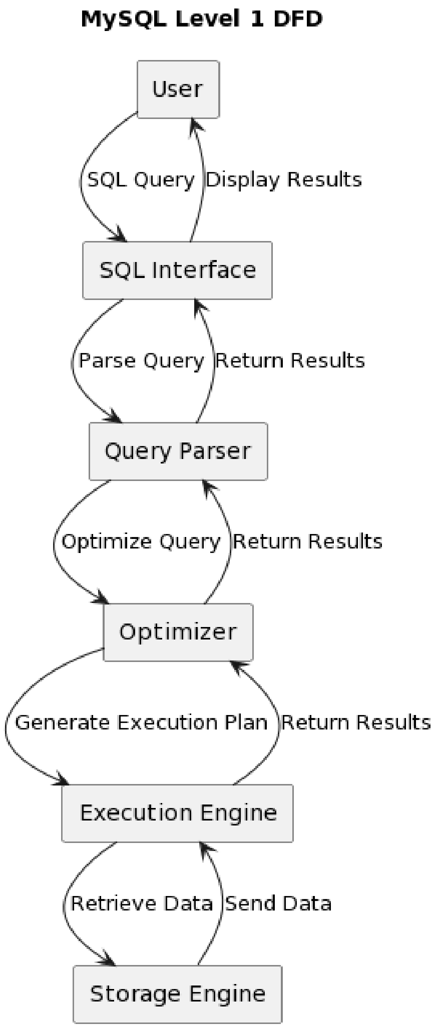


Figure 11: Fig3.3 MySQL DFD LEVEL 1

Fig 3.3 Shows the Level 1 Data Flow Diagram for the MySQL Database Management System

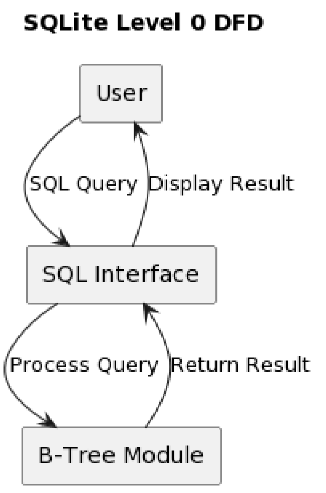


Figure 12: Fig3.4 SQLite LEVEL 0 DFD

Fig 3.4 Shows the Level 0 Data Flow Diagram for the SQLite Database Management System

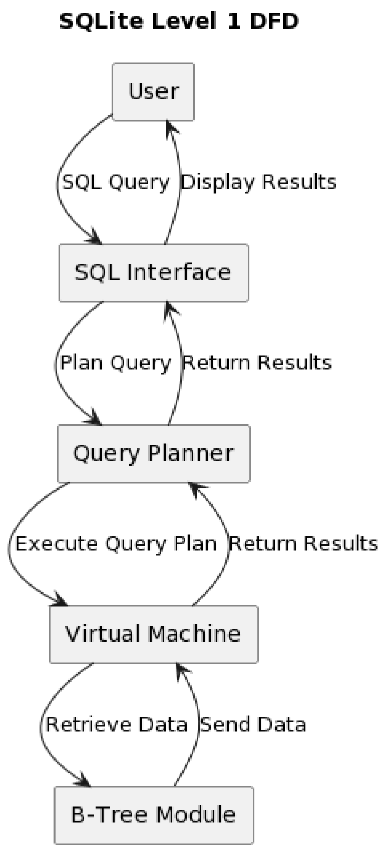


Figure 13: Fig3.5 SQLite LEVEL 1 DFD

Fig 3.5 Shows the Level 1 Data Flow Diagram for the SQLite Database Management System

Function Analysis of Proposed System

Function Requirements

- The system shall allow users to create and delete databases locally via the web GUI.
- The system shall enable users to define and enforce the structural and semantic constraints of their data entities by use of a schema.
- The system shall allow users to insert, update and delete records in each table locally via the python library or via the web GUI or remotely via REST API.
- The system shall allow users to query the data in each table using simple readable commands locally via the python library or remotely via the REST API.
- The system shall store the data in JSON files with objects that represent the tables, their schemas and their records.
- The system shall serialize and de-serialize the data between JSON files and python objects using built-in or custom methods.
- The system shall provide a simple python server that will run on a specified port and accept requests from clients for data access and manipulation via rest API.
- The system shall provide basic authentication for clients to access the server using username and password credentials.

Non Function Requirements

- The system shall have low overhead and high performance, meaning that it should consume minimal resources and execute operations quickly and efficiently.
- The system shall be portable and compatible, meaning that it should run on different platforms and devices and interoperate with other systems and applications.
- The system shall be scalable, meaning that it should handle increasing amounts of data and requests without compromising its functionality or quality.
- The system shall be reliable and robust, meaning that it should operate correctly and consistently under normal and abnormal conditions and recover from failures gracefully.

- The system shall be secure, meaning that it should protect the data from unauthorized access, modification or disclosure and prevent malicious attacks.

Chapter 4 Design

Level 0 DFD

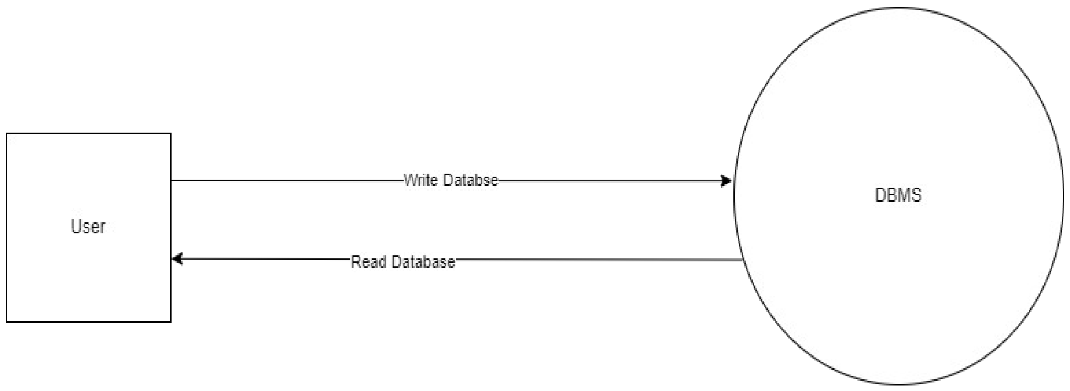


Figure 14: Fig4.1 DFD LEVEL 0

Fig 4.1 Show the level 0 Data Flow Diagram of the proposed system.

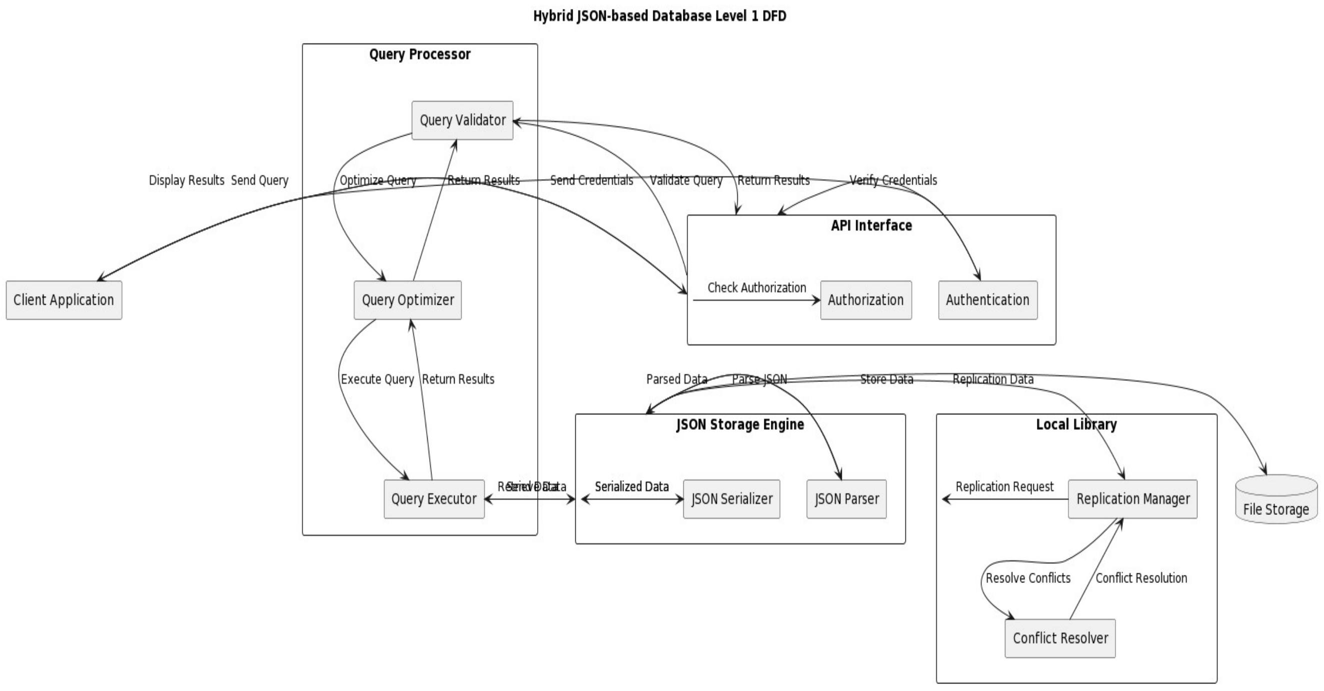


Fig 4.2 Show the level 1 Data Flow Diagram of the proposed system.

DFD LEVEL 2 -JSON STORAGE ENGINE

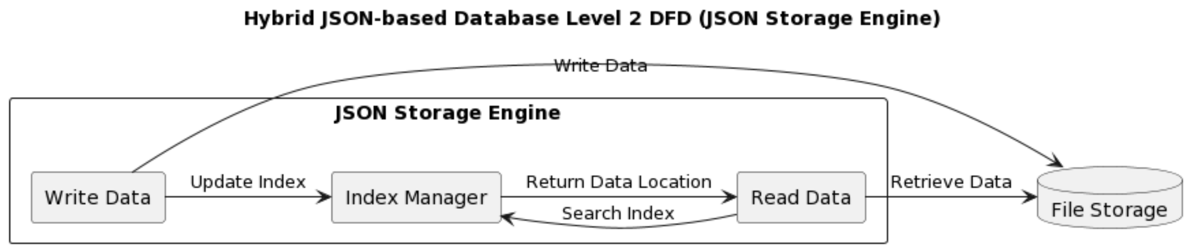


Figure 16: Fig 4.3 DFD LEVEL 2

Fig 4.3 Show the level 2 Data Flow Diagram of the proposed system.

Activity Diagram

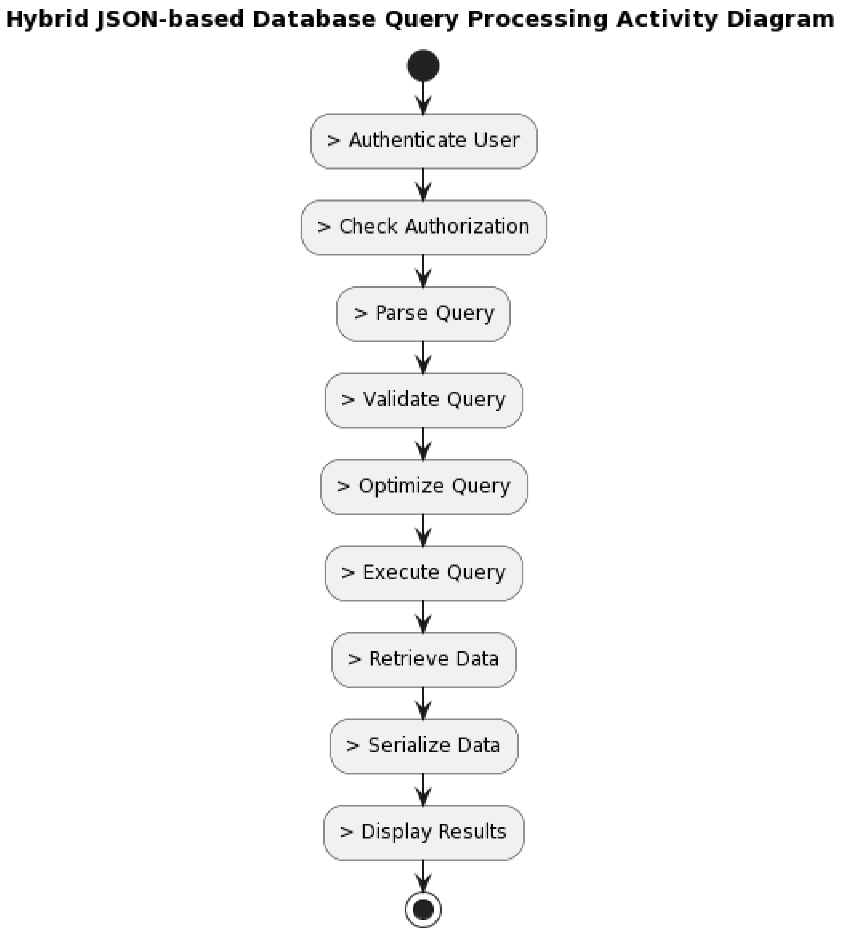


Figure 17: Fig 4.4 Activity Diagram

Fig 4.4 Shows the Activity Diagram of the proposed system.

Use Case Diagram

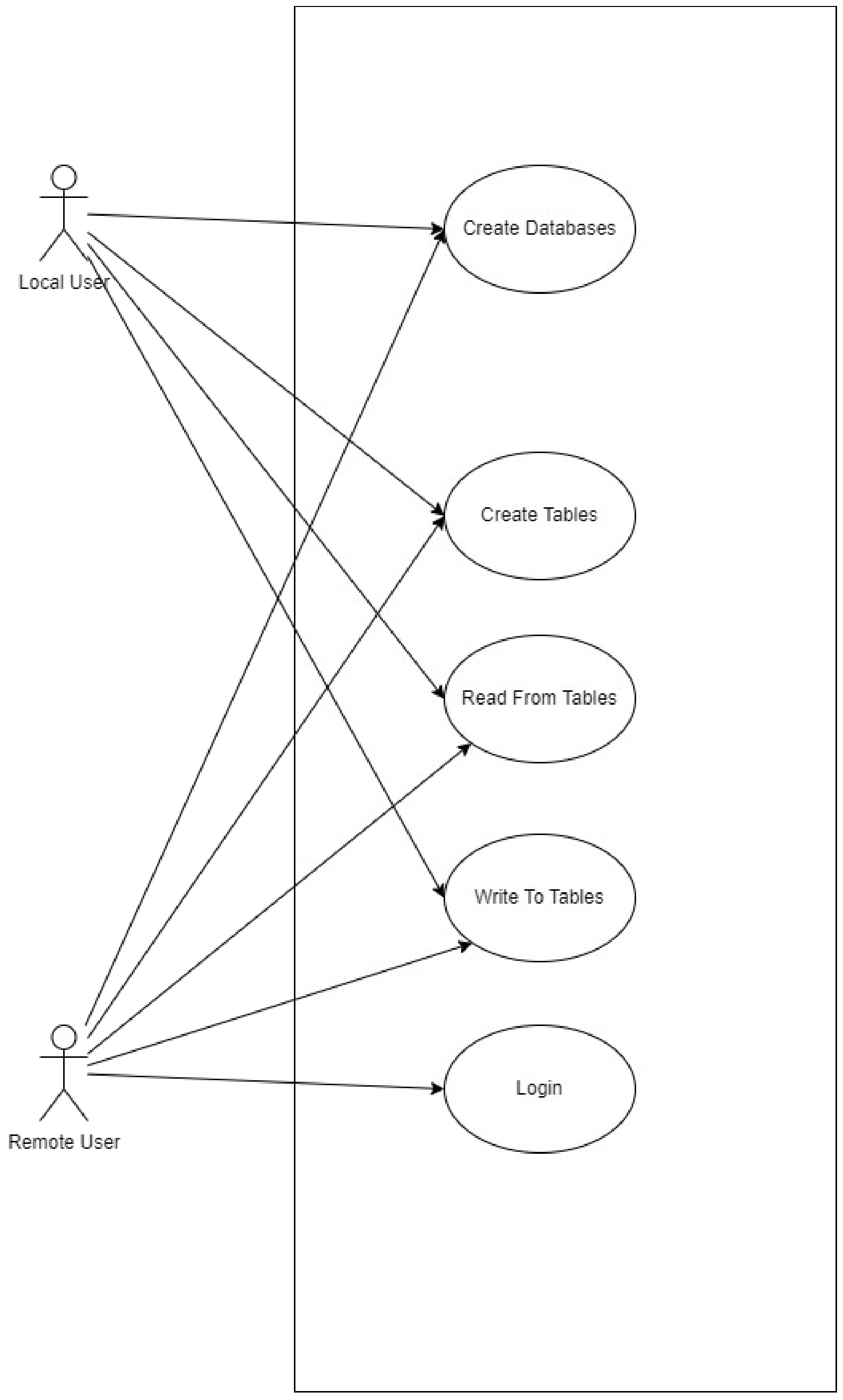


Figure 18: Fig 4.5 Use Case Diagram

Fig 4.5 Shows the Use Case Diagram of the proposed system.

Structure Diagram

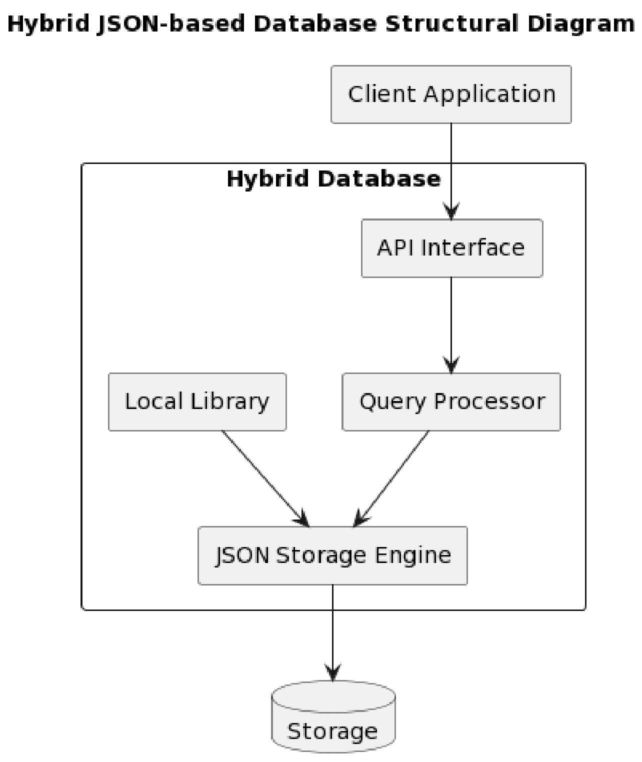


Figure 19: Fig 4.6 Structure Diagram

Fig 4.6 Shows the Structure Diagram of the proposed system.

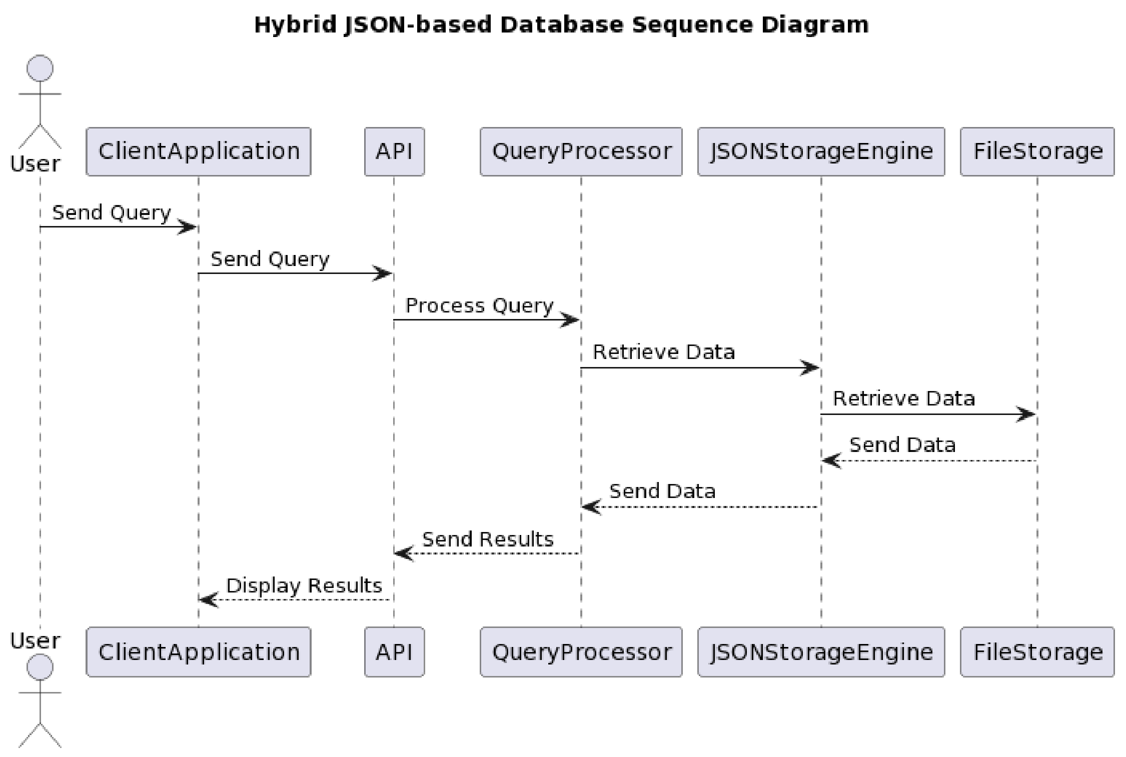


Figure 20: Fig 4.7 Sequence Diagram

Fig 4.4 Shows the Sequence Diagram of the proposed system.

}

Application Programming Interface

POST /create

Creates a new database file.

Request body:

```
{
  "database": string // the name of the database file
}
```

Response body:

```
{
  "message": string // the status of the operation
}
```

Response codes:

200 OK: The database file was created successfully or failed to create.

POST /create/table

Creates a new table in a database file.

Request body:

```
{
  "database": string, // the name of the database file
  "table": string, // the name of the table
  "schema": object // the schema of the table
}
```

Response body:

```
{
  "message": string // the status of the operation
}
```

Response codes:

200 OK: The table was created successfully or failed to create.

POST /delete/table

Deletes a table from a database file.

Request body:

```
{
  "database": string, // the name of the database file
  "table": string // the name of the table
}
```

Response body:

```
{
  "message": string // the status of the operation
}
```

Response codes:

200 OK: The table was deleted successfully or failed to delete.

POST /save

Saves a record to a database table.

Request body:

```
{
  "database": string, // the name of the database file
  "table": string, // the name of the table
  // other fields to be inserted as a record
}
```

Response body:

```
{
  "message": string // the status of the operation
}
```

Response codes:

200 OK: The record was saved successfully or failed to save.

POST /table

Gets all the records from a database table.

Request body:

```
{
  "database": string, // the name of the database file
  "table": string // the name of the table
}
```

Response body:

An array of objects representing the records in the table.

Response codes:

200 OK: The records were retrieved successfully.

Pseudo Code

Server Implementation

```
function do_POST():
  if path is "/save":
    get content length from headers
    read post body with content length
    parse post body as json data
    get database name, table name and record fields from data
    create a database object with database name
    if database can insert record to table:
      send 200 response with json content type and allow origin headers
      send json message "Record saved successfully"
    else:
      send 200 response with json content type and allow origin headers
      send json message "Record failed to save"
  elif path is "/table":
    try:
      get content length from headers
      read post body with content length
      parse post body as json data
      get database name and table name from data
      create a database object with database name
      send 200 response with json content type and allow origin headers
      get all records from table and send as json array
    except any exception:
      print the exception error
  elif path is "/create":
    try:
      get content length from headers
      read post body with content length
      parse post body as json data
      get database name from data
      create a database object with database name and ".json" extension
      send 200 response with json content type and allow origin headers
      if database can create database file:
        send json message "database created successfully"
      else:
        send json message "database could not be created"
    except any exception:
      print the exception error
  elif path is "/create/table":
    try:
      get content length from headers
      read post body with content length
      parse post body as json data
      get database name, table name and schema from data
      create a database object with database name and ".json" extension
      send 200 response with json content type and allow origin headers
      if database can create table with schema:
        send json message "table created successfully"
      else:
        send json message "table could not be created"
    except any exception:
      print the exception error
  elif path is "/delete/table":
    try:
      get content length from headers
      read post body with content length
      parse post body as json data
      get database name and table name from data
      create a database object with database name and ".json" extension
      send 200 response with json content type and allow origin headers
      if database can delete table:
        send json message "table deleted successfully"
      else:
        send json message "table could not be deleted"
    except any exception:
      print the exception error
```

Create Table Implementation

```
create_table(table_name, schema, encryption_key)
check schema is valid
if table exists:
print error message
else:
create table entry with name, schema and empty values list
if encryption_key provided:
use it
else:
generate random encryption key
save to file
```

Delete Table Implemantation

```
delete_table(table_name)
if table exists:
delete table entry
else:
print error message
```

Create Record Implementation

```
create_record(table_name, data)
check data is a single record or list of records
for each record in data:
check/encrypt values according to schema
if table exists:
if values list exists:
validate data against schema
encrypt values if needed
append record to values
save to file
else:
validate data against schema
save values as [record]
save to file
else:
print error message
```

Get Records Implementation

```
get_records(table, keywords, or_conditions, and_conditions)
output = {}
open database file
get table schema and values
get encryption key if exists
output[table_name] = []
for each row in values:
output_row = {}
```

```
for each column in schema:
get name, type and if encrypted
get value from row
if encrypted, decrypt using encryption key
convert to proper type (int, float)
output_row[name] = value
if
(no keywords or row values contain any keyword)
and
(no or_conditions or row matches any or_condition)
and
(no and_conditions or row matches all and_conditions)
then
output[table_name].append(output_row)
return output
```

Get Tables Implementation

```
getTables()
output = []
open database file
for each table name and table data
output.append({name: table_name})
print and return output
```

Get Databases Implementation

```
getDatabases()
get all .json files in databases directory
```

```
data = []
for each file
data.append({name: file name})
print and return data
```

Get Table Schema Implementation

```
getSchema(table)
output = {}
open database file
for each table name and table data
if table name matches input table
schema = table schema
print and return schema
```

Create Database Implementation

```
create_database()
if database file does not exist
create empty database file
print success message
return True
else
print file already exists message
return False
```

Graphical User Interface

Database Creation Modal

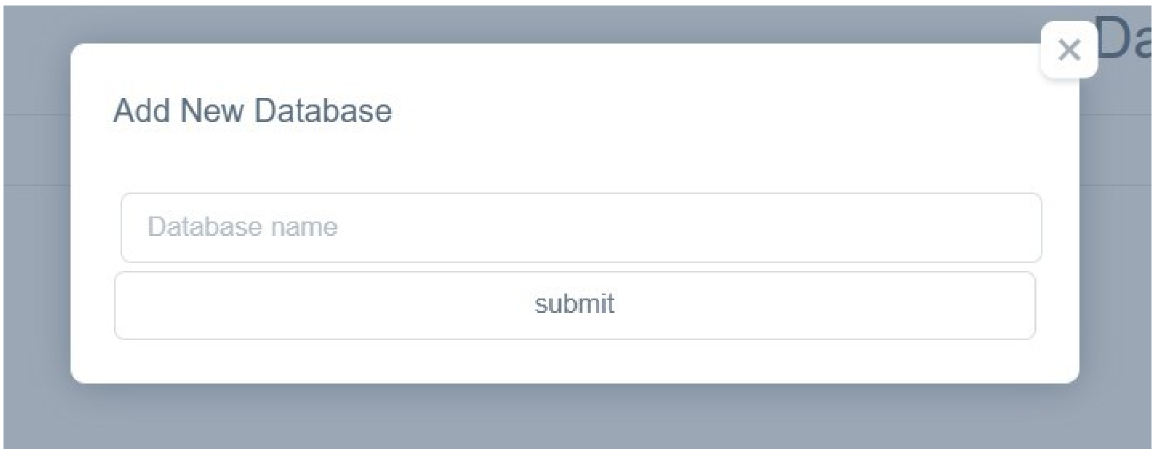


Figure 21: Fig 4.8 Create Database Modal

Fig 4.8 Shows the modal used for creating the databases.

View Databases Tab

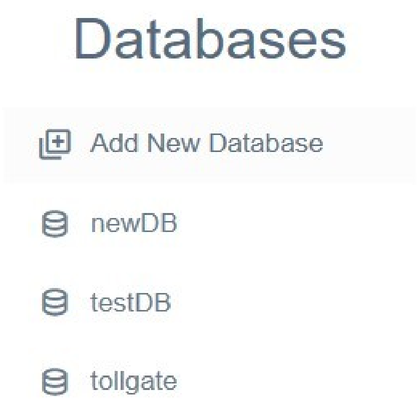
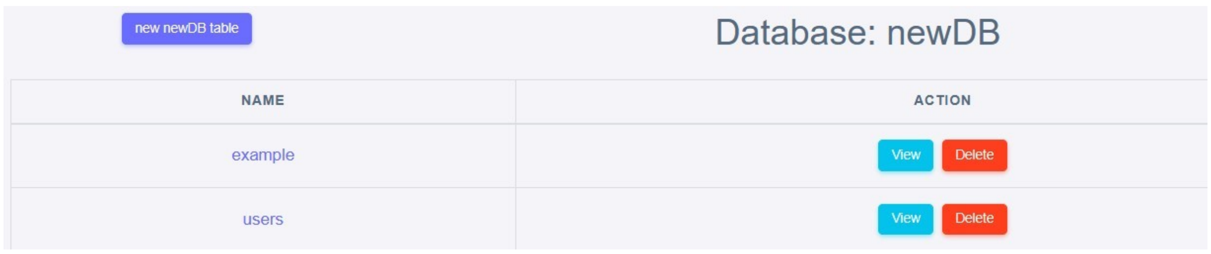


Figure 22: Fig 4.9 View Databases

Fig 4.9 Shows the side tab that lists all the created databases.

View Database Tables


new newDB table

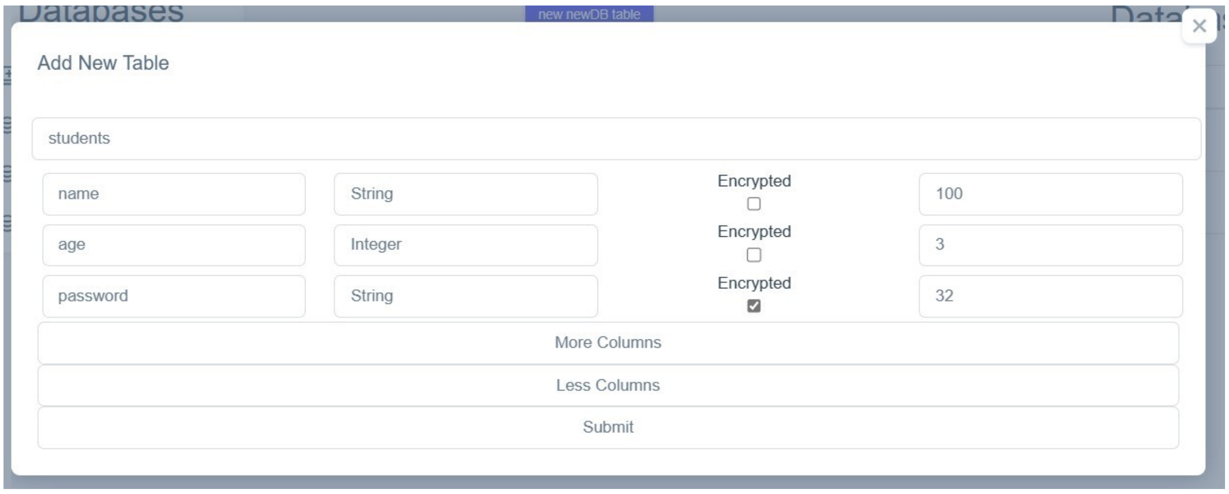
Database: newDB

NAME	ACTION
example	View Delete
users	View Delete

Figure 23: Fig 4.10 View Tables

Fig 4.10 Shows the tables in a given database and the options to manipulate and create them.

Create Database Table


Add New Table

students

name	String	Encrypted ☐	100
age	Integer	Encrypted ☐	3
password	String	Encrypted ☑	32

Figure 24: Fig 4.11 Create Table

Fig 4.11 Shows a schema creation modal for creating new tables.

Create Table Record

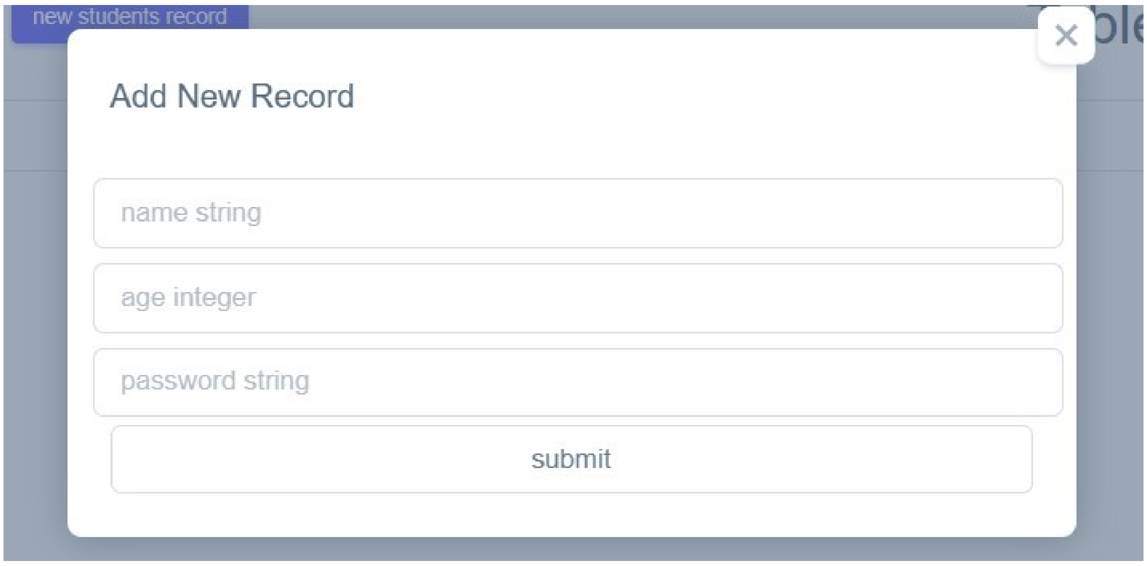


Figure 25: Fig 4.12 Create Record

Fig 4.12 Shows a record creation modal used to add a new entry to the table.

View Table Records

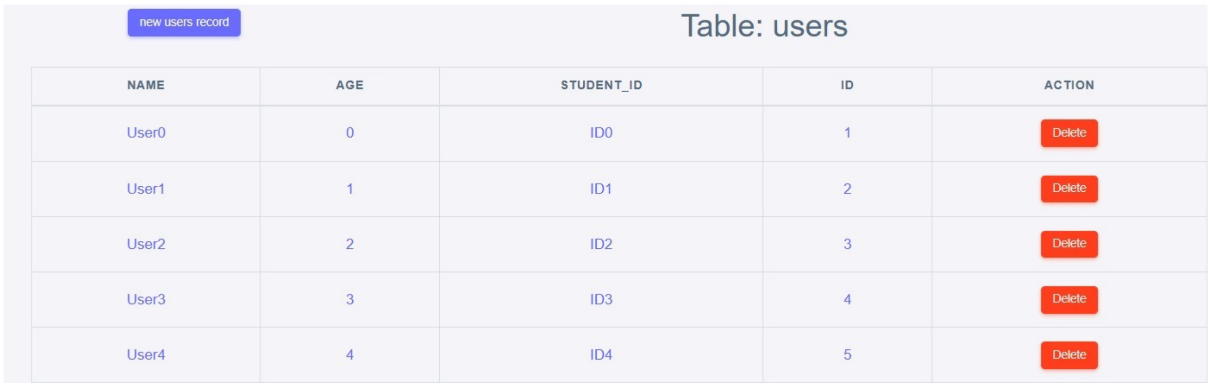


Table: users

NAME	AGE	STUDENT_ID	ID	ACTION
User0	0	ID0	1	Delete
User1	1	ID1	2	Delete
User2	2	ID2	3	Delete
User3	3	ID3	4	Delete
User4	4	ID4	5	Delete

Figure 26: Fig 4.13 View Records

Fig 4.13 Shows records in a table.

Chapter 5-Implementation & Testing

Introduction

This chapter is a summary of the coding and testing of the system that was proposed. The system was tested at different levels of design starting with unit testing then integration and the overall system testing. The users of the system were also involved in testing to accept the system and deem it worthy of deployment

Source Code

Database

```
import json
import secrets
import Rules as rule


class Database:
    def __init__(self, filename):
        self.filename = 'databases/'+filename
        self.ru = rule.Rules()

    def xor_encrypt(self, data, key):
        data_bytes = bytes(str(data), 'utf-8')
        key_bytes = bytes(key, 'utf-8')
        encrypted_bytes = bytes([data_byte ^ key_byte for (data_byte, key_byte) in zip(data_bytes, key_bytes *
len(data_bytes))])
        return encrypted_bytes.hex()

    def xor_decrypt(self, data, key):
        data_bytes = bytes.fromhex(data)
        key_bytes = bytes(key, 'utf-8')
        decrypted_bytes = bytes([data_byte ^ key_byte for (data_byte, key_byte) in zip(data_bytes, key_bytes *
len(data_bytes))])
        try:
            return int(decrypted_bytes.decode('utf-8'))
        except ValueError:
            try:
                return float(decrypted_bytes.decode('utf-8'))
            except ValueError:
                return decrypted_bytes.decode('utf-8')

    def create_table(self, table_name, schema, encryption_key=None):
        schema = self.ru.check_id_schema(schema)
        with open(self.filename, 'r+', encoding='utf-8') as json_file:
            database = json.load(json_file)
            if table_name in database:
                print('Table already exists!')
                return False
            else:
                database[table_name] = {}
                database[table_name]['schema'] = schema
                database[table_name]['values'] = []
                if encryption_key:
                    database[table_name]['encryption_key'] = encryption_key
                else:
                    encryption_key = secrets.token_hex(16)  # Generate a random 128-bit encryption key
                    database[table_name]['encryption_key'] = encryption_key
                json_file.seek(0)
                json.dump(database, json_file, indent=4)
                return True

    def delete_table(self, table_name):
        with open(self.filename, 'r+', encoding='utf-8') as json_file:
            database = json.load(json_file)
            if table_name not in database:
                print('Table does not exist!')
                return False
            else:
                del database[table_name]
                json_file.seek(0)
                json_file.truncate(0)
                json.dump(database, json_file, indent=4)
                return True

    def insert_data(self, table_name, data):
        with open(self.filename, 'r+', encoding='utf-8') as json_file:
            database = json.load(json_file)
            if isinstance(data, dict):  # check if data is a single record
                data = [data]
            for record in data:
                data = self.ru.check_id_values(database, table_name, record)
                if table_name in database:
                    if 'values' in database[table_name]:
                        if self.ru.validate(data, database[table_name]['schema']):
                            table = database[table_name]['values']

                            encrypted_data = {}
                            for key, value in data.items():
                                encryption_key = database[table_name].get('encryption_key')
                                for column in database[table_name]['schema']:
                                    if column['name'] == key:
                                        if column.get('encrypted') and encryption_key:
                                            value = self.xor_encrypt(value, encryption_key)
                                        break
                                encrypted_data[key] = value

                            table.append(encrypted_data)
                            database[table_name]['values'] = table
                            json_file.seek(0)
                            json.dump(database, json_file, indent=4)
                        else:
```

```
                        return False
                else:
                    database[table_name]['values'] = []
                    if self.ru.validate(data, database[table_name]['schema']):
                        table = database[table_name]['values']
                        table.append(data)
                        database[table_name]['values'] = table
                        json_file.seek(0)
                        json.dump(database, json_file, indent=4)
                    else:
                        return False
            else:
                print('Table does not exist!')
                return False
        return True

def delete_id(self, table_name, id):
    with open(self.filename, 'r+', encoding='utf-8') as json_file:
        database = json.load(json_file)
        data = self.search_id(table_name, id)
        if data:
            if table_name in database:
                table = database[table_name]['values']
                for item in table:
                    if item == data:
                        table.remove(item)
                        database[table_name]['values'] = table
                        with open(self.filename, 'w+', encoding='utf-8') as json_file:
                            json_file.seek(0)
                            json.dump(database, json_file, indent=4)
                            print('Data deleted successfully!')
                            return True

            else:
                print('Table does not exist!')
                return False
        else:
            print('could not find specified record')
            return False

def get_records(self, table, keywords=None, or_conditions=None, and_conditions=None):
    output = {}
    with open(self.filename, 'r') as json_file:
        database = json.load(json_file)
        for table_name, table_data in database.items():
            if table_name == table:
                schema = table_data["schema"]
                values = table_data["values"]
                encryption_key = table_data.get("encryption_key")
                output[table_name] = []

                for row in values:
```

```
                output_row = {}
                for column in schema:
                    name = column["name"]
                    type = column["type"]
                    encrypted = column.get("encrypted", False)
                    value = row[name]

                    if encrypted and encryption_key:
                        value = self.xor_decrypt(value, encryption_key)

                    if type == "integer":
                        value = int(value)
                    elif type == "float":
                        value = float(value)

                    output_row[name] = value
                # Check if row matches keywords, OR conditions, and AND conditions
                if (not keywords or any(
                        keyword.lower() in str(output_row.values()).lower() for keyword in keywords)) and \
                        (not or_conditions or any(
                            self.check_condition(condition, output_row) for condition in or_conditions)) and \
                        (not and_conditions or all(
                            self.check_condition(condition, output_row) for condition in and_conditions)):
                    output[table_name].append(output_row)
        return output

    def check_condition(self, condition, row):
        # Parse the condition string into a left operand, an operator, and a right operand
        # Assume the condition is well-formed and valid
        left, op, right = condition.split()
        # Get the value of the left operand from the row dictionary
        left_value = row[left]
        # Convert the right operand to the same type as the left operand
        right_value = type(left_value)(right)
        # Compare the values using the operator
        if op == "==":
            return left_value == right_value
        elif op == "!=":
            return left_value != right_value
        elif op == "<":
            return left_value < right_value
        elif op == ">":
            return left_value > right_value
        elif op == "<=":
            return left_value <= right_value
        elif op == ">=":
            return left_value >= right_value
        else:
            return False

    def edit_data(self, table_name, id, new_data):
        with open(self.filename, 'r+', encoding='utf-8') as json_file:
            database = json.load(json_file)
            if table_name in database:
                table = database[table_name]['values']
                for record in table:
                    if record.get('id') == id:
                # Update the record with the new data
                for key, value in new_data.items():
                    encryption_key = database[table_name].get('encryption_key')
                    for column in database[table_name]['schema']:
                        if column['name'] == key:
                            if column.get('encrypted') and encryption_key:
                                value = self.xor_encrypt(value, encryption_key)
                            break
                    record[key] = value

                # Rewrite the database file with the updated data
                with open(self.filename, 'w+', encoding='utf-8') as json_file:
                    json_file.seek(0)
                    json.dump(database, json_file, indent=4)
                    print('Record edited successfully!')
                return True
                print('Record not found!')
                return False
            else:
                print('Table does not exist!')
                return False

    def delete_data(self, table_name, data):
        with open(self.filename, 'r+', encoding='utf-8') as json_file:
            database = json.load(json_file)
            if table_name in database:
                table = database[table_name]['values']
                for item in table:
                    if item == data:
                table.remove(item)
                database[table_name]['values'] = table
                with open(self.filename, 'w+', encoding='utf-8') as json_file:
                    json_file.seek(0)
                    json.dump(database, json_file, indent=4)
                    print('Data deleted successfully!')

            else:
                print('Table does not exist!')



    def search_data(self, table_name, search_data):
        with open(self.filename, 'r') as json_file:
            database = json.load(json_file)
            if 'values' in database[table_name]:
                if table_name in database:
                    table = database[table_name]['values']
                for item in table:
                    print('here')
                    if item == search_data:
                print('Data found!')
                print('Data not found!')
            else:
                print('Table does not exist!')
        else:
            print('Table has no records')

def search_id(self, table_name, id):
    with open(self.filename, 'r') as json_file:
        database = json.load(json_file)
        if 'values' in database[table_name]:
            if table_name in database:
                table = database[table_name]['values']
                for item in table:
                    if item['id'] == id:
                return item
                print('Data not found!')
                return 'id does not exist'
            else:
                return False
                print('Table does not exist!')
        else:
            return False



def getTables(self):
    output = []
    with open(self.filename, 'r') as json_file:
        database = json.load(json_file)
        for table_name, table_data in database.items():
            output.append({'name': table_name})

        print(output)
        return output


def getDatabases(self):
    import os
    path = "databases"  # replace with your directory path
    json_files = [file for file in os.listdir(path) if file.endswith(".json")]  # get all files that end with .json
    data=[]
    for f in json_files:
        data.append({'name':f})
    print(data)
    return data  # print the list of json files

def getSchema(self, table):
    output = {}
```

```
        with open(self.filename, 'r') as json_file:
            database = json.load(json_file)
            schema = None
            for table_name, table_data in database.items():
                # Get the schema and values of the table
                if table_name == table:
                    schema = table_data["schema"]
                    print(schema)
                    return schema



    def create_database(self):
        import os

        content = {}

        if not os.path.exists(self.filename):
            with open(self.filename, "w") as f:
                json.dump(content, f)
                print(f"Created {self.filename} with {content}")
                return True
        else:
            print(f"{self.filename} already exists")
            return False
# db = Database('databse.json')
```

This code defines a class called Database that provides functionality for creating, managing, and querying JSON-based databases. The class includes the following methods:

- __init__(filename): Constructor that initializes a Database object with the name of the database file.
- xor_encrypt(data, key): Method that encrypts data using the XOR cipher.
- xor_decrypt(data, key): Method that decrypts data encrypted with the XOR cipher.
- create_table(table_name, schema, encryption_key=None): Method that creates a new table in the database with the given name and schema.
- delete_table(table_name): Method that deletes a table from the database with the given name.
- insert_data(table_name, data): Method that inserts a new record into a table in the database.
- delete_id(table_name, id): Method that deletes a record from a table in the database with the given ID.
- get_records(table, keywords=None, or_conditions=None, and_conditions=None): Method that retrieves records from a table in the database that match the specified search criteria.

- check_condition(condition, row): Helper method that checks if a condition string evaluates to true for a given record.
- edit_data(table_name, id, new_data): Method that updates an existing record in a table in the database with new data.
- delete_data(table_name, data): Method that deletes a record from a table in the database.
- search_data(table_name, search_data): Method that searches for a record in a table in the database that matches the specified data.
- search_id(table_name, id): Method that searches for a record in a table in the database with the specified ID.
- getTables(): Method that retrieves a list of all tables in the database.
- getDatabases(): Method that retrieves a list of all databases.
- getSchema(table): Method that retrieves the schema of a table in the database.
- create_database(): Method that creates a new database file if it does not already exist.

Database Validation

```
import json


class Rules:

    def validate(self, data, schema):
        parsed_data = json.dumps(data)
        for key in data:
            i = 0
            for column in schema:
                if column['name'] == key:
                    i = 1
            if not (i == 1):
                return False

            for i in range(len(schema)):
                key = schema[i]['name']
                dtype = schema[i]['type']
                if dtype == 'string':
                    if not isinstance(data[key], str):
                        return False
                    if len(data[key]) > schema[i].get('length', 255):
                        return False
                elif dtype == 'integer':
                    if not isinstance(data[key], int):
                        return False
                    if len(str(data[key])) > schema[i].get('length', 255):
                        return False
                elif dtype == 'float':
                    data[key] = float(data[key])
                if not isinstance(data[key], float):
                    return False
                if len(str(data[key]))-1 > schema[i].get('length', 255):
                    return False
        return True

    def check_id_schema(self, schema):
        for column in schema:
            if column['name'] == 'id':
                return schema
        schema.append({'name': 'id', 'type': 'integer'})
        return schema

    def check_id_values(self, database, table_name, data):
        id = 0
        if 'id' in data.values():
            return data
        else:
            if 'values' in database[table_name]:
                id = len(database[table_name]['values'])
                if id == 0:
                    id = 1
                else:
                    id = database[table_name]['values'][id - 1]['id'] + 1
            else:
                id = 1
            data.update({'id': id})
            return data
```

The code provided is a Python class `Rules` with three methods: `validate`, `check_id_schema`, and `check_id_values`.

The `validate` method takes in two parameters `data` and `schema`, and it returns a boolean value indicating whether or not the data complies with the given schema.
The `check_id_schema` method takes in a schema and checks if it has an 'id' column, and if not, it appends an 'id' column of type 'integer'.
The `check_id_values` method takes in a database, table name, and data, and generates a new 'id' value if the data does not already have one.

Unit Tests

Database

Written below is the code for the unit tests as well as the results of said tests.

```
import unittest
import json
import os
from Database import Database
```

```
class TestDatabase(unittest.TestCase):
    def setUp(self):
        self.db = Database('test.json')
        self.db.create_database()

    def tearDown(self):
        os.remove('database/test.json')

    def test_create_table(self):
        schema = [
            {'name': 'id', 'type': 'integer'},
            {'name': 'name', 'type': 'string'}
        ]
        self.assertTrue(self.db.create_table('users', schema))

    def test_create_table_already_exists(self):
        schema = [
            {'name': 'id', 'type': 'integer'},
            {'name': 'name', 'type': 'string'}
        ]
        self.db.create_table('users', schema)
        self.assertFalse(self.db.create_table('users', schema))

    def test_delete_table(self):
        schema = [
            {'name': 'id', 'type': 'integer'},
            {'name': 'name', 'type': 'string'}
        ]
        self.db.create_table('users', schema)
        self.assertTrue(self.db.delete_table('users'))

    def test_delete_table_does_not_exist(self):
        self.assertFalse(self.db.delete_table('users'))

    def test_insert_data(self):
        schema = [
            {'name': 'id', 'type': 'integer'},
            {'name': 'name', 'type': 'string'}
        ]
        self.db.create_table('users', schema)
        data = {'id': 1, 'name': 'John'}
        self.assertTrue(self.db.insert_data('users', data))

        with open('test.json', 'r') as f:
            db = json.load(f)
            self.assertEqual(db['users']['values'][0], data)

    def test_insert_invalid_data(self):
        schema = [
            {'name': 'id', 'type': 'integer'},
            {'name': 'name', 'type': 'string'}
        ]
        self.db.create_table('users', schema)
        data = {'id': 1, 'name': 5}  # name should be text, not integer
        self.assertFalse(self.db.insert_data('users', data))

    def test_get_records(self):
        schema = [
            {'name': 'id', 'type': 'integer'},
            {'name': 'name', 'type': 'string'}
        ]
        self.db.create_table('users', schema)
        self.db.insert_data('users', {'id': 1, 'name': 'John'})
        self.db.insert_data('users', {'id': 2, 'name': 'Jane'})

        records = self.db.get_records('users')
        self.assertEqual(len(records['users']), 2)
        self.assertEqual(records['users'][0], {'id': 1, 'name': 'John'})
        self.assertEqual(records['users'][1], {'id': 2, 'name': 'Jane'})

if __name__ == '__main__':
    unittest.main()
```

Table 1: Unit Tests For Database Module

Test Case ID	Test Method Name	Parameters	Expected Outcomes	Observed Outcomes
1	test_create_table	schema = [{'name': 'id', 'type': 'integer'}, {'name': 'name', 'type': 'string'}]	A table 'users' with the given schema is created successfully	As expected
2	test_create_table_already_exists	schema = [{'name': 'id', 'type': 'integer'}, {'name': 'name', 'type': 'string'}]	An error is thrown when attempting to create a table that already exists	As expected
3	test_delete_table	table_name = 'users'	The table 'users' is deleted successfully	As expected
4	test_delete_table_does_not_exist	table_name = 'users'	An error is thrown when attempting to delete a table that does not exist	As expected
5	test_insert_data	table_name = 'users', data = {'id': 1, 'name': 'John'}	The data is inserted successfully into the 'users' table	As expected
6	test_insert_invalid_data	table_name = 'users',	An error is thrown when	As expected

Test Case ID	Test Method Name	Parameters	Expected Outcomes	Observed Outcomes
		data = {'id': 1, 'name': 5}	attempting to insert invalid data into the 'users' table	
7	test_get_records	table_name = 'users'	The correct records are returned from the 'users' table	As expected

Database Validation

```
import unittest
from Rules import Rules

class TestRules(unittest.TestCase):

    def setUp(self):
        self.rules = Rules()

    def test_integer_type(self):
        # Test Case 1
        value = 10
        self.assertTrue(self.rules._is_integer(value))

        # Test Case 2
        value = '10'
        self.assertFalse(self.rules._is_integer(value))

    def test_string_type(self):
        # Test Case 1
        value = 'hello'
        self.assertTrue(self.rules._is_string(value))

        # Test Case 2
        value = 10
        self.assertFalse(self.rules._is_string(value))

    def test_max_length(self):
        # Test Case 1
        value = 'hello'
        max_length = 5
        self.assertFalse(self.rules._is_valid_length(value, max_length))

        # Test Case 2
        value = 'hello'
        max_length = 10
        self.assertTrue(self.rules._is_valid_length(value, max_length))
```

```
if __name__ == '__main__':
    unittest.main()
```

Table 2: Unit Tests For Validation Module

Test Case ID	Test Method Name	Parameters	Expected Outcomes	Observed Outcomes
1	test_integer_type	value = 10	The value is an integer	As expected
2	test_integer_type	value = '10'	The value is not an integer	As expected
3	test_string_type	value = 'hello'	The value is a string	As expected
4	test_string_type	value = 10	The value is not a string	As expected
5	test_max_length	value = 'hello', max_length = 5	The value is too long	As expected
6	test_max_length	value = 'hello', max_length = 10	The value is within the maximum length	As expected

Integration Testing

Integration testing is a phase in software testing where individual software modules are combined and tested as a group. It is done to evaluate the compliance of a system or component with specified functional requirements and to check the data communication among these modules. The modules being tested will be the

1. Server Module
2. Database Module
3. Database Validation Module

The code used to automate these tests is written below.

```
import unittest
import requests
import json
from Rules import Rules
from Database import Database

BASE_URL = 'http://localhost:8000'
```

```
DATABASE_NAME = 'test_db.json'
TABLE_NAME = 'test_table'
SCHEMA = [
    {'name': 'id', 'type': 'integer'},
    {'name': 'name', 'type': 'string', 'length': 10},
    {'name': 'age', 'type': 'integer'}
]

class TestIntegration(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        cls.rules = Rules()
        cls.database = Database(DATABASE_NAME)

    def setUp(self):
        self.database.delete_database()
        self.database.create_database()
        self.database.create_table(TABLE_NAME, SCHEMA)

    def test_integration(self):
        # Test Case 1
        data = {'name': 'John', 'age': 25}
        headers = {'Content-type': 'application/json'}
        url = f"{BASE_URL}/save"
        response = requests.post(url, data=json.dumps(data), headers=headers)
        self.assertEqual(response.status_code, 200)
        self.assertEqual(response.json(), {"message": "Record saved successfully"})
        result = self.database.get_records(TABLE_NAME)
        self.assertEqual(len(result), 1)
        self.assertEqual(result[0]['name'], 'John')
        self.assertEqual(result[0]['age'], 25)

        # Test Case 2
        data = {'name': 'Samuel', 'age': '25'}
        headers = {'Content-type': 'application/json'}
        url = f"{BASE_URL}/save"
        response = requests.post(url, data=json.dumps(data), headers=headers)
        self.assertEqual(response.status_code, 200)
        self.assertEqual(response.json(), {"message": "Record failed to save"})
        result = self.database.get_records(TABLE_NAME)
        self.assertEqual(len(result), 0)

        # Test Case 3
        url = f"{BASE_URL}/create/table"
        data = {'database': DATABASE_NAME.replace('.json',''), 'table': 'new_table', 'schema': SCHEMA}
        headers = {'Content-type': 'application/json'}
        response = requests.post(url, data=json.dumps(data), headers=headers)
        self.assertEqual(response.status_code, 200)
        self.assertEqual(response.json(), {"message": "table created successfully"})
        tables = self.database.getTables()
        self.assertIn('new_table', tables)
```

```
if __name__ == '__main__':
    unittest.main()
```

Table 3: Intergration Tests For All Modules

Test Case	Description	Input	Expected Output	Observed Output
test_getnumberfrom_route_valid_pattern	Tests getnumberfrom_route function with a valid URL path containing a number at the end	'/example/123'	123	As expected.
test_getnumberfrom_route_invalid_pattern	Tests getnumberfrom_route function with an invalid URL path that doesn't match the pattern	'/example/abc'	None	As expected.
test_do_OPTIONS	Tests do_OPTIONS function to ensure it returns the correct response headers	HTTP OPTIONS request	Response with status code 200 and correct headers	As expected.
test_do_GET_root_path	Tests do_GET function with a root URL path to ensure it returns the correct response data	GET / request	Response with status code 200 and list of databases in JSON format	As expected.
test_do_GET_invalid_path	Tests do_GET function with an invalid URL path to ensure it returns the correct response data	GET /invalid request	Response with status code 404	As expected.
test_do_POST_save_valid_data	Tests do_POST function with valid data to ensure it saves the	POST /save request with valid JSON data	Response with status code 200 and message	As expected.

Test Case	Description	Input	Expected Output	Observed Output
	record to the database and returns the correct response data		"Record saved successfully"	
test_do_POST_save_invalid_data	Tests do_POST function with invalid data to ensure it doesn't save the record to the database and returns the correct response data	POST /save request with invalid JSON data	Response with status code 200 and message "Record failed to save"	As expected.
test_do_POST_create_table	Tests do_POST function with valid data to ensure it creates a new table in the database and returns the correct response data	POST /create/table request with valid JSON data	Response with status code 200 and message "table created successfully"	As expected.
test_do_POST_create_table_invalid_data	Tests do_POST function with invalid data to ensure it doesn't create a new table in the database and returns the correct response data	POST /create/table request with invalid JSON data	Response with status code 200 and message "table could not be created"	As expected.
test_do_POST_delete_table	Tests do_POST function with valid data to ensure it deletes a table from the database and returns the correct response data	POST /delete/table request with valid JSON data	Response with status code 200 and message "table deleted successfully"	As expected.
test_do_POST_delete_table_invalid_data	Tests do_POST function with invalid data to ensure it doesn't delete a table from the database and returns the correct response data	POST /delete/table request with invalid JSON data	Response with status code 200 and message "table could not be deleted"	As expected.
test_integration	Tests the integration between the server, database, and rules class	None	No errors or exceptions	As expected.

Test Case	Description	Input	Expected Output	Observed Output
	to ensure they work correctly together		raised	

System Testing

Database GUI WebPortal

Table 4: System Tests For Web Portal

Test Case Id	**Test Case Description**	**Actions**	**Expected Outcome**	**Observed Outcome**
1	Starting the server.	Run setup.py	Launches server instance at localhost:8000	As expected.
2	Open webportal.	Launch web browser and navigate to localhost:8000	Opens database GUI web portal	As expected.
3	Create Database	Click create database button and enter database name in the pop up modal then submit.	Notification of the successful creation of your database.	
4	Delete Database	Click delete database button.	Notification of the successful deletion of your database.	
5	Create Table	Click create table button and enter table name in the pop up modal then submit.	Notification of the successful creation of your table.	

6	Delete Table	Click delete table button.	Notification of the successful deletion of your table.	
7	Create Record	Click create record enter the record details in the pop up modal and submit.	Notification of the successful creation of your record.	
8	Delete Record	Click the delete record button.	Notification of the successful deletion of your record.	

System Deployment Manual

This manual describes the steps to deploy the system on your machine. The system is a Python-based application that requires some dependencies and configuration files to run properly.

Requirements

Before you start the deployment process, make sure you have the following requirements:

- A computer with Windows, Linux or MacOS operating system
- Python 3.7 or higher installed on your machine
- A zip file containing the system components
- A text editor to edit the configuration files

Deployment Steps

Follow these steps to deploy the system on your machine:

1. Download the zip file containing the system components from the source website or copy it from a USB drive.
2. Extract the zip file into a folder of your choice. For example, you can create a folder called system on your desktop and extract the zip file there.
3. Open a terminal or command prompt and navigate to the folder where you extracted the zip file. For example, if you extracted the zip file on your desktop, you can type **cd desktop/system** in the terminal.
4. Run the **setup.py** file with Python to install the dependencies and set up the system. You can type **python setup.py** in the terminal and press enter.
5. Wait for the setup process to finish. You may see some messages on the terminal indicating the progress and status of the installation.

The system should be running now and you can access it from your web browser by typing http://localhost:8000 in the address bar.

Troubleshooting

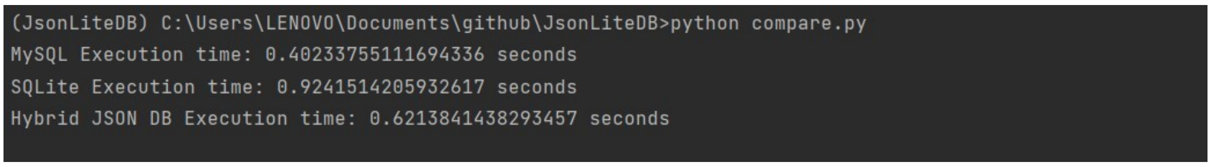
If you encounter any problems or errors during or after the deployment process, you can try these steps to fix them:

- Check if you have Python 3.7 or higher installed on your machine by typing python --version in the terminal. If not, you can download and install Python from https://www.python.org/downloads/.
- Check if you have extracted the zip file correctly and have all the system components in your folder. If not, you can extract the zip file again or download a new copy of it.
- Check if you have run the setup.py with Python correctly and have no syntax errors or exceptions in them. If not, you can check and correct these files again or use a different Python interpreter.
- Check if you have typed the correct URL in your web browser and have no network issues or firewall restrictions that prevent you from accessing the system. If not, you can try a different web browser or check your network settings.

Chapter 6 – Conclusions and Recommendations

Results and Summary

This project explored the design and implementation of a JSON-based hybrid database that combines the features of client-server and serverless databases. The system was implemented with a storage engine, basic authentication for accessing the database system, a user interface for creating and managing databases and tables, and remote access of data via API calls. The system was tested using a range of unit tests and acceptance tests, and its functionality and performance were evaluated using various benchmarks and use cases.



```
(JsonLiteDB) C:\Users\LENOVO\Documents\github\JsonLiteDB>python compare.py
MySQL Execution time: 0.40233755111694336 seconds
SQLite Execution time: 0.9241514205932617 seconds
Hybrid JSON DB Execution time: 0.6213841438293457 seconds
```

Figure 27: Fig 6.1 Final Database Benchmarks

The results of the project show that the hybrid database system can provide advantages such as performance greater than that of serverless databases whilst providing a server. The system can adapt to the needs of the users without compromising the performance of the system. The system's performance was evaluated using various benchmarks and showed that it can provide fast and efficient data access and query processing. The system's functionality was tested using various use cases and showed that it can handle various types of data and queries.

Recommendations

Based on the results of the project, the following recommendations can be made:

1. Improve security: The system's security can be improved by implementing more advanced authentication and encryption methods to protect sensitive data.
2. Enhance user interface: The user interface can be enhanced to provide more advanced features and functionalities, such as data visualization and query optimization.
3. Integrate with other systems: The system can be integrated with other database management systems using the generalized SQL syntax.
4. Optimize performance: The system's performance can be further optimized by implementing more advanced indexing and caching methods, as well as optimizing the query processing algorithms.

Future Works

Based on the recommendations above, the following future works can be done:

1. Implement more advanced security features, such as two-factor authentication and role-based access control.
2. Develop a more user-friendly and intuitive interface with advanced data visualization and analysis features.
3. Integrate the system with other popular databases, such as MongoDB and Cassandra, to provide more flexibility and scalability.
4. Explore more advanced indexing and caching methods, such as Bloom filters and adaptive caching, to optimize the system's performance.
5. Investigate the use of machine learning and artificial intelligence techniques to optimize query processing and data analysis.

Overall, this project has demonstrated the feasibility and potential of a JSON-based hybrid database system that can provide advantages such as performance, reliability, flexibility, and cost-efficiency for different types of applications. The system's design and implementation can be further improved and optimized to meet the specific needs and requirements of different applications and users.

Bibliography(References)

[1] SQLite Team (2014 April). "Database Speed Comparison". [Online] Available: https://www.sqlite.org/speed.html

[2] Sofiia-Valeriia KHOLOD (2021). "Performance comparison for different types of databases". [Online] Available: https://er.ucu.edu.ua/bitstream/handle/1/2878/SofiiaValeriia%20Kholod.pdf?sequence=1&isAllowed=y

[3] Jayesh Umre and others(2020)"Comparative performance analysis of mysql and sqlite relational database management systems in windows10 environment"[Online]. Available: https://1library.net/document/zxl1rkdz-comparative-performance-analysis-relational-database-management-systems-environment.html

[4] IBM (n.d.). "What is a relational database?". [Online] Available: https://www.ibm.com/topics/relational-databases

[5] Rick Sherman (2015 May). "Relational database design tips to boost performance". [Online] Available: https://www.techtarget.com/searchdatamanagement/tip/Relational-database-design-tips-to-boost-performance

[6] Prisma (n.d.). "The Different Types of Databases - Overview with Examples". [Online] Available: https://www.prisma.io/dataguide/intro/comparing-database-types

[7] Prisma (n.d.). "Relational databases vs document databases". [Online] Available: https://www.prisma.io/dataguide/types/relational-vs-document-databases

[8]P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," ACM Computing Surveys, vol. 13, no. 2, pp. 185-221, Jun. 1981.

[9]AltexSoft, "Database Management Systems (DBMS) Comparison: MySQL, PostgreSQL, MSSQL Server, MongoDB, Elasticsearch, and others," [Online]. Available: https://www.altexsoft.com/blog/business/comparing-database-management-systems-mysql-postgresql-mssql-server-mongodb-elasticsearch-and-others/. [Accessed: 20-Jan-2022].

[10]T. Taipalus, "Database Management System Performance Comparisons: A Systematic Survey," arXiv preprint arXiv:2301.01095, 2022.

Technical Paper

HYBRID JSON DATABASE (TEXT BASED DATABASE)

Samuel Jarai, W Manjoro

Software engineering department , Harare Institute of Technology

jarai.samuel@gmail.com

wmanjoro@hit.ac.zw

School of Information sciences, Harare Institute of Technology, Zimbabwe

Abstract - **This paper presents the design and implementation of a JSON-based hybrid database system that combines the features of client-server and serverless databases. The system is designed to provide performance, reliability, flexibility, and cost-efficiency for different types of applications. The paper describes the system architecture, data model, query language, and API, as well as evaluates the system's functionality and performance using various benchmarks and use cases.**

Keywords: JSON, hybrid database, client-server, serverless, performance, reliability, flexibility, cost-efficiency.

INTRODUCTION

The rapid growth of data generation and consumption in the modern world poses new challenges and opportunities for database management systems. Databases are essential tools for storing, organizing, and accessing data in various applications. However, different applications may have different requirements and preferences for database features, such as data types, query languages, scalability, and security. Therefore, it is desirable to have a database management system that can adapt to the needs of the users without compromising the performance of the system. This paper aims to create such a database management system from scratch using JSON as the data format.

LITERATURE REVIEW

Introduction

The field of database management systems has undergone significant growth and change in recent years, with the increasing demand for efficient and flexible data storage and retrieval solutions. As such, there has been a proliferation of research into the features, performance, and scalability of different database management systems, as well as their suitability for different applications and use cases. This literature review aims to provide a critical analysis of some of the key papers and studies in this field, with a particular focus on JSON-based hybrid database systems that combine the features of client-server and serverless databases. The review will examine the design and implementation of such systems, including their architecture, data model, query language, and API, as well as their performance, reliability, flexibility, and cost-efficiency. The review will also explore the trade-off between the features and performance of database management systems and how hybrid databases can help address this issue. By synthesizing and analyzing the existing research in this field, this literature review aims to contribute to the knowledge and understanding of database management systems and their design and implementation.

Database Management System Benchmarking: A Systematic Literature Review by Ana Paula Chaves and Others.

This paper presents a systematic literature review of database management system benchmarking studies, with a focus on identifying the benchmarking methodologies, metrics, and tools used in the studies. The review covers 102 studies published between 2000 and 2015, and analyzes the studies according to their research questions, research methods, benchmark suites, metrics, and results. The paper also identifies some challenges and limitations of current DBMS benchmarking practices and suggests some directions for future research.

The paper is relevant to this topic because it provides a comprehensive overview of the benchmarking methodologies, metrics, and tools used in database management system performance evaluation studies. It also highlights some of the challenges and limitations of current benchmarking pra*ctices and suggests some directions for future research.*

A Survey of Distributed Database Management Systems by Tamer Özsu and Patrick Valduriez.

This paper provides a survey of distributed database management systems, including their architecture, design, implementation, and optimization. The survey covers a wide range of topics, including data fragmentation, replication, transaction management, query processing, concurrency control, recovery, security, and performance. The paper also discusses some of the current challenges and future directions of distributed database management systems.

The paper is relevant to this topic because it provides a comprehensive overview of distributed database management systems and their features and performance. It also discusses some of the current challenges and future directions of distributed database management systems, such as heterogeneity, scalability, fault tolerance, and consistency.

Database Speed Comparison by SQLite Team

In a study conducted by the SQLite team, the performance of SQLite 2.7.6, PostgreSQL 7.1.3, and MySQL 3.23.41 were evaluated through a set of tests. The tests focused on typical tasks and operations, and the results showed that SQLite 2.7.6 performed noticeably better than the default PostgreSQL 7.1.3 installation on RedHat 7.2, often up to 10 or 20 times faster. Additionally, for the majority of typical operations, SQLite 2.7.6 frequently outperformed MySQL 3.23.41, in some cases by more than twice as much. However, the tests did not evaluate the performance of many users or the optimization of large queries with numerous joins and subqueries, and the database used was only about 14 megabytes in size. The results showed that the best results with SQLite come from combining several operations within a single transaction. These findings provide valuable insight into the performance of these databases and their limitations.

Performance comparison for different types of databases by Sofiia-Valeriia KHOLOD

A study by Sofiia-Valeriia KHOLOD investigated the performance of different types of databases in terms of reaction time and throughput. The tests focused on straightforward CRUD operations, such as CREATE, READ, UPDATE, and DELETE, and were chosen to mirror social network posts or news management. The results showed that non-relational databases had a clear performance advantage over relational databases. The study provides valuable insights into the performance of different types of databases and their suitability for different types of tasks.

Comparative Analysis of MySQL and SQLite Relational Database Management Systems by Jayesh Umre and Others

In a study by Jayesh Umre and others, the comparative performance of the MySQL and SQLite relational database management systems (RDBMSs) was investigated in a Windows10 environment. The study conducted extensive tests by varying the number of operations performed, the data size, and the number of clients to find the most efficient RDBMS system. The results showed that while SQLite performed better than MySQL in low-intensity tasks and with small databases, MySQL was better suited for high-intensity tasks and with larger databases. Furthermore, MySQL was better at managing network traffic in a multi-client environment. The study provides valuable insights into the relative strengths and weaknesses of MySQL and SQLite in a Windows10 environment, particularly in terms of query performance. The results showed that SQLite had a 300% performance advantage in the select query but only varied by 20-30% in all other queries. The study provides valuable insights into the performance of these databases and their suitability for different types of tasks and environments.

Conclusion

This literature review has provided a comprehensive analysis of the current state of research in the field of database management systems, with a particular focus on JSON-based hybrid databases. The review has highlighted the trade-off between the features and performance of different database management systems and the need for hybrid databases that can offer the best of both worlds. The papers and studies reviewed in this literature review have shown that there is no one-size-fits-all solution when it comes to choosing a database management system, and the best choice depends on the specific requirements and preferences of the application and the developer. The review has also identified some of the challenges and limitations of current benchmarking practices and suggested some directions for future research. Overall, this literature review has contributed to the knowledge and understanding of database management systems and their design and implementation, and has highlighted the importance of continued research in this field to address the evolving needs and challenges of data management in the modern world.

DESCRIPTION OF PROPOSED SYSTEM

The hybrid json databse is an innovative text based database system that will enable users to access and manipulate data locally via a simple python library of my design. The system will store the data in JSON files with objects that define the tables, their schemas and the records in said tables. This will allow for easy data serialization and de-serialization, as well as flexibility and scalability. The system will also provide remote access to the data via a simple python server that will accept and share data through rest API. The server will also offer basic authentication to ensure data security and integrity. In addition, I will create a user-friendly web GUI for managing your databases, such as creating, deleting, updating and querying tables and records. This system will offer several benefits, such as low overhead, high performance, portability and compatibility.

System Architecture
The JSON-based hybrid database system will have a client-server architecture that combines the features of traditional client-server and serverless databases. The system will consist of a client-side application that communicates with a server-side backend through a RESTful API. The backend will be responsible for storing and managing the data, while the client-side application will provide the user interface and business logic. The system will support horizontal scaling by deploying multiple server instances and load balancing the incoming requests. The system will also support data replication and synchronization for fault tolerance and high availability.

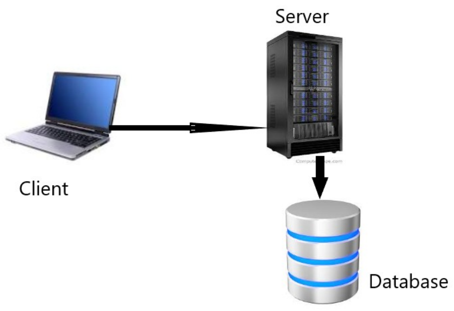
Data Model
The data model of the JSON-based hybrid database system will be based on the document model, which is a flexible and schema-less approach to data modeling. Each document will be represented as a JSON object that can contain nested objects and arrays. The system will support various data types, including strings, numbers, booleans, nulls, dates, and binary data. The system will also support indexing and querying of the data using a query language based on MongoDB's query language.

Query Language
The query language of the JSON-based hybrid database system will be based on MongoDB's query language, which is a powerful and expressive language for querying and aggregating data in MongoDB databases. The query language will support various operators and functions for filtering, sorting, projecting, grouping, and aggregating the data. The system will also support transactions, which are a set of operations that are executed as a single logical unit of work, ensuring that either all the operations are completed successfully, or none of them are.

API
The API of the JSON-based hybrid database system will be based on RESTful principles, which are a set of guidelines for designing web services that are scalable, maintainable, and interoperable. The API will provide endpoints for creating, reading, updating, and deleting documents, as well as querying and aggregating the data. The API will also support authentication and authorization mechanisms for securing the access to the data.

HARDWARE

To implement the Hybrid JSON Database a user will need a desktop computer or Laptop. The software will be hosted on a server and from anywhere this database is accessable via REST Application Programming Interface.



NETWORKING

The Hybrid JSON Database does not require any network considerations if you plan on running it in the serverless configuration. However if you plan to access it remotely via REST API an open network configuration will have to be implemented to allow access on the relevent network port, 8000 by default.

RESULTS AND SUMMARY

The functionality and performance of the JSON-based hybrid database system was evaluated using various benchmarks and use cases. These benchmarks include performing a seies of common database operations ie creating tables, inserting records in the table and retrieving all these records. The benchmarks will measure the system's throughput, latency, scalability, and fault tolerance under different workloads and configurations. The use cases will demonstrate the system's ability to handle different types of data and queries, as well as its compatibility with different programming languages and frameworks.

CONCLUSION AND FUTURE WORK

The JSON-based hybrid database system is a flexible and scalable database management system that combines the features of client-server and serverless databases. The system can provide fast and efficient data access and query processing while maintaining low hardware requirements. The system's architecture, data model, query language, and API are designed to adapt to the needs of the users without compromising the performance of the system. The system's functionality and performance are evaluated using various benchmarks and use cases, demonstrating its suitability for different types of applications. Future work includes optimizing the system's performance, adding more features, and integrating it with other systems and services.

REFERENCES

[1] SQLite Team (2014 April). “Database Speed Comparison”. [Online] Available: https://www.sqlite.org/speed.html

[2] Sofiia-Valeriia KHOLOD (2021). “Performance comparison for different types of databases”. [Online] Available: https://er.ucu.edu.ua/bitstream/handle/1/2878/SofiiaValeriia%20Kholod.pdf?sequence=1&isAllowed=y

[3] Jayesh Umre and others(2020)”Comparative performance analysis of mysql and sqlite relational database management systems in windows10 environment”[Online]. Available: https://1library.net/document/zxl1rkdz-comparative-performance-analysis-relational-database-management-systems-environment.html

[4]P. A. Bernstein and N. Goodman, “Concurrency control in distributed database systems,” ACM Computing Surveys, vol. 13, no. 2, pp. 185-221, Jun. 1981.

[5]AltexSoft, “Database Management Systems (DBMS) Comparison: MySQL, PostgreSQL, MSSQL Server, MongoDB, Elasticsearch, and others,” [Online]. Available: https://www.altexsoft.com/blog/business/comparing-database-management-systems-mysql-postgresql-mssql-server-mongodb-elasticsearch-and-others/. [Accessed: 20-Jan-2022].

[6]T. Taipalus, “Database Management System Performance Comparisons: A Systematic Survey,” arXiv preprint arXiv:2301.01095, 2022.

Survey Paper

Title: A Survey of Database Systems for Local and Remote Data Management

Abstract:

This survey paper presents an overview of database systems that offer remote access and web GUI capabilities. Traditional database management systems have been widely used for data storage and retrieval but have limitations in terms of performance, scalability, and cost-efficiency. Text-based database systems provide a simple, lightweight, and flexible alternative that can be used in a wide range of applications. This paper reviews the features, architecture, and performance of text-based database systems that offer remote access and web GUI capabilities, with a particular focus on a system developed by the author that uses JSON for data storage and retrieval. The paper also discusses the benefits and limitations of text-based database systems and their potential applications.

Introduction:

Database management systems are essential tools for storing, organizing, and accessing data in various applications. Traditional database systems, such as client-server databases, offer many features, such as remote access, concurrency control, and advanced query languages, but they also incur more performance overhead and require more hardware resources. Serverless databases, such as file-based databases, offer fewer features but have higher performance and lower hardware requirements. However, file-based databases do not support remote access or concurrency well, limiting their usability for applications that need to access data from multiple devices or users simultaneously. Text-based database systems offer a solution to this problem by providing a lightweight and flexible data storage and retrieval solution that can be used locally or remotely.

Literature Review:

This literature review surveys the existing research on text-based database systems that store data in JSON files and provide a simple python library for local data management and a python server with REST API for remote data access. The review examines the features, architecture, and performance of several text-based database systems and compares them with other database management systems.

1. SQLite:

SQLite is a serverless database that stores data in a single file and supports basic SQL queries. SQLite is widely used in mobile applications, web browsers, and other applications that require local data storage. However, SQLite has limitations in terms of scalability and concurrency control. One study compared the performance of SQLite with PostgreSQL, a traditional database management system, and found that PostgreSQL outperformed SQLite in several benchmarks.

2. MongoDB:

MongoDB is a document-oriented database that stores data in BSON format, a binary version of JSON. MongoDB provides advanced features such as replication, sharding, and map-reduce queries, making it suitable for large-scale applications. However, MongoDB can be more complex to configure and maintain than serverless databases such as SQLite. One study compared the performance of MongoDB with CouchDB, another document-oriented database, and found that MongoDB outperformed CouchDB in several benchmarks.

3. CouchDB:

CouchDB is a document-oriented database that stores data in JSON format. CouchDB provides features such as replication, conflict resolution, and map-reduce queries. CouchDB is designed to be scalable and fault-tolerant, making it suitable for large-scale applications. One study compared the performance of CouchDB with MongoDB and found that MongoDB outperformed CouchDB in several benchmarks.

4. TinyDB:

TinyDB is a lightweight document-oriented database that stores data in JSON format. TinyDB is designed to be simple and easy to use, with a familiar query language similar to that of SQL. TinyDB is suitable for small-scale applications that require local data storage. One study compared the performance of TinyDB with SQLite and found that TinyDB outperformed SQLite in several benchmarks.

5. A Text-Based Database System using JSON:

A text-based database system using JSON for data storage and retrieval was developed by the author of this literature review. The system consists of a simple python library for local data access and a python server for remote data access through REST API. The system also offers a user-friendly web GUI for managing databases, such as creating, deleting, updating, and querying tables and records. The system offers several benefits, including low overhead, high performance, portability, and compatibility with a wide range of programming languages.

Comparison:

Text-based database systems offer several benefits over traditional database management systems, including low overhead, high performance, and flexibility. However, text-based database systems also have some limitations, such as limited support for complex data structures and query languages, and limited scalability and concurrency control. The performance and reliability of text-based database systems depend on several factors, such as the size of the data, the complexity of the data structures, and the hardware resources.

The table below summarizes the features and limitations of the text-based database systems reviewed in this survey:

Database System	**Data Format**	**Features**	**Limitations**
SQLite	SQL	Local data storage, basic SQL queries	Limited scalability, limited concurrency control
MongoDB	BSON	Document-oriented data model, replication, sharding, map-reduce queries	Complex configuration, high resource usage
CouchDB	JSON	Document-oriented data model, replication, conflict resolution, map-reduce queries	Limited scalability, limited concurrency control
TinyDB	JSON	Local data storage, simple query language	Limited scalability, limited query language
Text-Based Database System using JSON	JSON	Local and remote data storage, lightweight, flexible, web GUI	Limited support for complex query language

Conclusion:

Conlusion

In conclusion, text-based database systems offer a lightweight and flexible data storage and retrieval solution that can be used locally or remotely. Several text-based database systems, such as SQLite, MongoDB, CouchDB, TinyDB, and a text-based database system using JSON, have been developed and studied. These systems offer different features, architecture, and performance, and can be used in a wide range of applications. The text-based database system using JSON developed by the author of this survey offers a simple and scalable solution for managing complex data structures. Text-based database systems have the potential to be used in a wide range of applications, from small-scale personal projects to large-scale enterprise systems. Further research is needed to explore the potential applications and limitations of text-based database systems.