Eötvös Loránd University

Faculty of Informatics

Dept. of Programming Languages and Compilers

DATABASE WITH PRIME NUMBER HASH

Supervisor:
Dr. Zsók Viktória
Lecturer

Author:
Liao Runkang
Computer Science BSc

Budapest, 2025

EÖTVÖS LORÁND UNIVERSITY
FACULTY OF INFORMATICS

Thesis Topic Registration Form

Student's Data:
Student's Name: Liao Runkang
Student's Neptun code: FKEP8U

Educational Information:
Training programme: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: *Dr. Zsók Viktória*
Supervisor's Home Institution: ***Department of Programming Languages and Compilers***
Address of Supervisor's Home Institution: ***1117, Budapest, Pázmány Péter sétány 1/C.***
Supervisor's Position and Degree: *Lecturer*

Thesis Title: A basic database with prime number hash and other new features

Topic of the Thesis:
(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

Any query related to complex data like string and object (for OOP feature supported database like Postgre SQL) is very time consuming.

However, the prime factorization can solve this problem. The database will give a unique prime number key for all types of data. So that, the records of the database will be a composition of prime numbers. The hash function from any type of data to a unique prime will be given in this database. The project will have a UI interface with QT framework, a memory pool, a thread pool, a storage unit, a basic SQL interpreter and a simple FTP connector.

The memory pool will be implemented by 4 byte step, which will provide effect way to load and release data from file. The thread pool will allocate numbers of threads based on the maximum thread number of machine installed, which will provide an efficient way to get and release a thread. The storage unit will contain a B+ tree and a file cache, which will load file data from disk and release file data to disk. The SQL interpreter will only support basic SQL statements like SELECT, UPDATE, WHERE, CREATE, DELETE, INSERT, DROP, FROM and ALTER, which will have a syntax and semantic tree inside the interpreter. The FTP connector will use socket API provided by operating system and use an FTP protocol to send and receive data. So that, there will be two types of connector, sender and receiver.

This database can be used in some time limited circumstances as the complexity of comparison for all types of data is O(1).

Budapest, 2025. 05. 15.

Contents

Chapter 1

Introduction

This chapter briefly introduces the **motivation** and the **task description** of the **Primed DB** project, including what problem the project solves and why. Meanwhile, this chapter lists the main target that the project should reach, such as the memory pool for the data blocks. Lastly, the benefits and changes brought about by this research will be given in the end.

1.1 Motivation

Firstly, **the primed Database(PrimedDB)** is a relational database that *converts all types of data into prime numbers and identifies a record mathematically.*

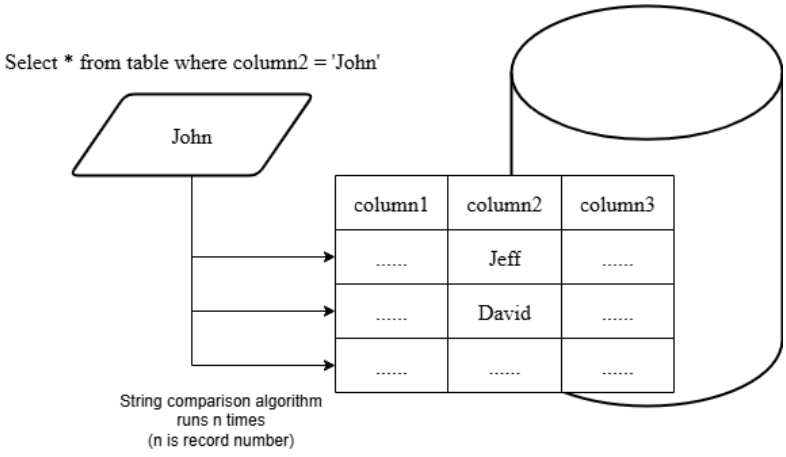

Figure 1.1: How do traditional relational databases query data

Thus, **Primed DB** explores a new solution, which optimizes the relational database by increasing the speed of data query and encryption. **PrimedDB** is designed to ensure that all data are encrypted at the time of insertion.

1.2 Task description

1.2.1 Save and Safe

Before **PrimedDB**, traditional relational database systems typically stored data directly in files, applying compression and encryption only when necessary [1]. However, this implementation requires additional calculation for encryption because raw data do not possess mathematical properties necessary to encrypt data [2].

In contrast, **PrimedDB** introduces a lightweight approach to derive prime-number-based hash keys from various data types at insert time.

Meanwhile, traditional relational databases exhibit variable time complexity when evaluating selection predicates over different data types. For example, unlike numeric comparisons, which are O(1), string comparisons require O(n) time (where n denotes string length), as shown in Figure 1.1 [3].

1.2.2 Unify columns

In **PrimedDB**, all types of data are stored in prime number format for different columns. Figure 1.2 illustrate the process of converting input data from their native representation into prime number based hash keys. **PrimedDB** erases the original data types and maps each input value to a unique prime number through a deterministic encoding function (explained in Convert any data into prime).

Meanwhile, there is a virtual column, called record key, which represents the product of all prime numbers format data from different columns.

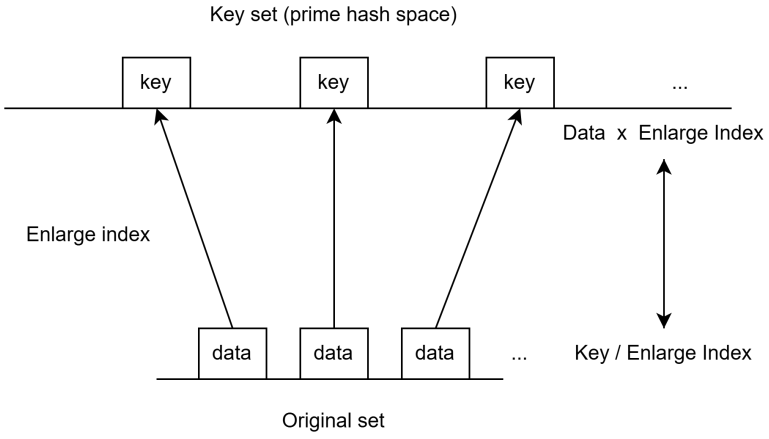

Figure 1.2: Data conversion to prime number hash key

The record key identifies duplicate records and supports fast query of target data (see What is PrimeDB). Table 1.1 compares the native input data format with their hashing form in **PrimedDB**, where values are mapped to prime number hash keys.

record number	name	phone number	GPA
Original data	Jeff	123456	4.3
11445867332.....	2399497119	3018604160	15802373

Table 1.1: This example demonstrates the data organization strategy employed by PrimedDB.

1.2.3 Components for manage data

In **PrimedDB**, the standard components of a relational database are implemented, including a SQL compiler, a block manager and a TCP server.

The TCP server handles multiple concurrent client connections, receiving SQL queries and returning resulting data. The block manager maps logical table structures to on-disk file representations and provides interfaces for querying and modifying data. Also, basic SQL statements, like `CREATE, INSERT, UPDATE, DELETE` and `SELECT`, are recognized by the SQL compiler which checks the syntax and the semantic validity during the processing period.

1.3 Thesis structure

The remainder of this paper is organized as follows:

The chapter 2 illustrates the installation procedure of both **PrimedDB** server, providing all functionalities of the database, and **PrimedDB** client, a user interface interacts with the **PrimedDB** server.

The chapter 3 provides all the necessary mathematical background knowledge in order to understand the new hashing function introduced.

chapter 4 describes the design of important **PrimedDB** components—including the transmission protocol, data storage layer, and SQL compiler—and explains how these components interact.

Finally, chapter 4.11 summarizes the thesis and outlines a potential future.

Chapter 2

User documentation

This chapter provides an **installation** and **user guide** for both **PrimedDB** server and client applications. Additionally, a list of prohibited operations is included, as the current version is a **demo** and does not yet offer the full functionality of a commercial database.

2.1 Installation

First, **Microsoft Edge** gives a **security warning** after the server and client installers are downloaded (see Figure 2.1), as **PrimedDB** is identified as a potentially suspicious application by Microsoft [4].

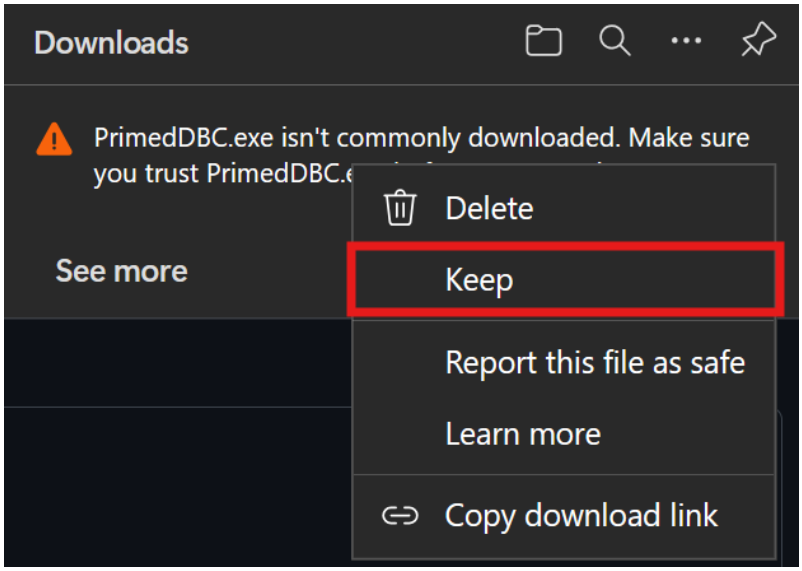

Figure 2.1: The error message sent by Microsoft Edge

Instead, users can either use the **Chrome** browser or **ignore** the warning and choose to **keep** the installer file. **PrimedDB** is an open-source application that contains no malicious code or viruses—its safety can be verified by inspecting the source code.

2.1.1 Server

The installer and source code are available on **GitHub** [5]. Users can choose whether to download and install the **Windows x64 installer** or the **source code archive** to compile it for their target platform (see Figure 2.2).

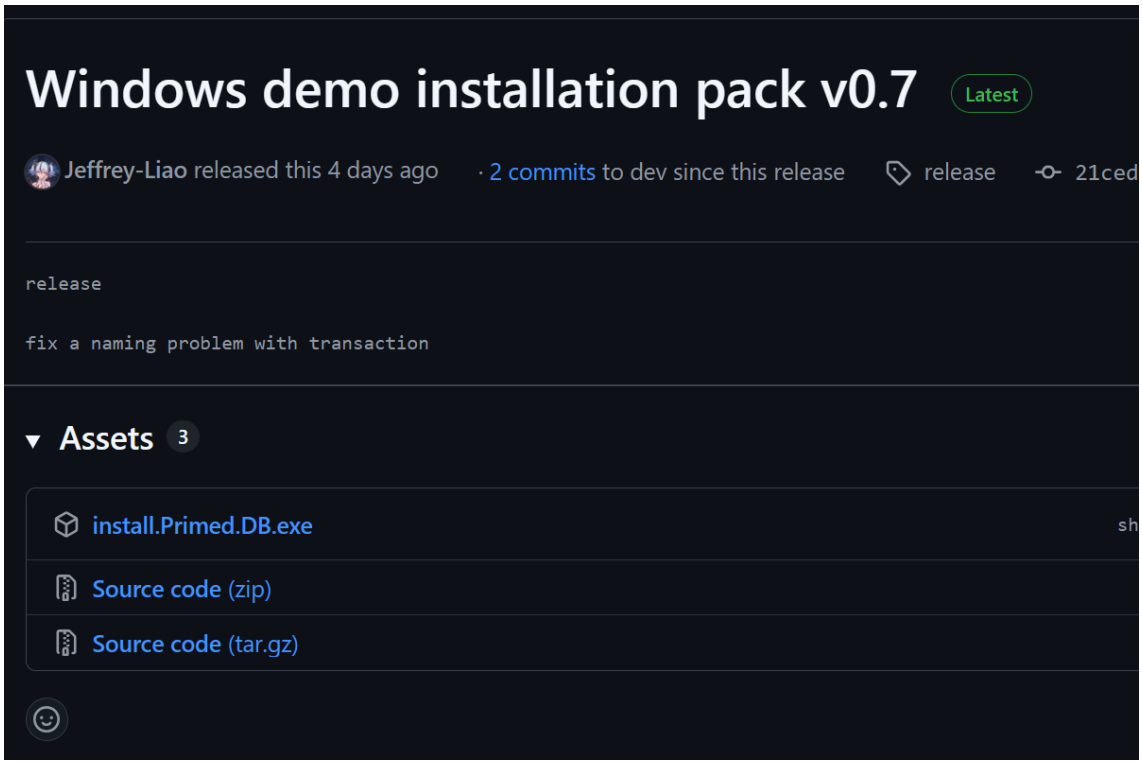

Figure 2.2: The download page of PrimedDB

The installation package is saved to the user-specified download folder. Upon double-clicking the installer, the user can **select a language for the software installer** (like Figure 2.3), either English or Hungarian.

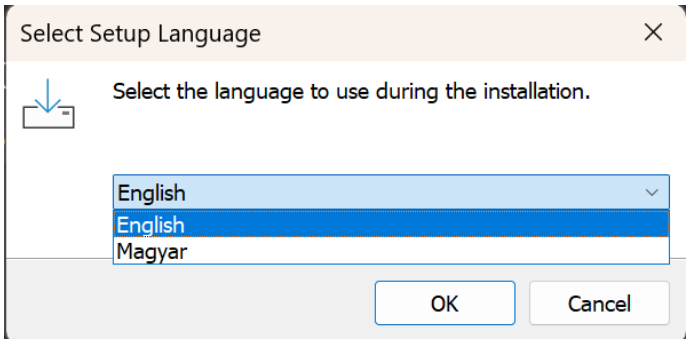

Figure 2.3: The language selection UI for installer

After that, the user selects an appropriate installation directory (see Figure 2.4). **PrimedDB** then creates a dedicated subdirectory, named `"/Primed DB"`, for the executable and associated files.

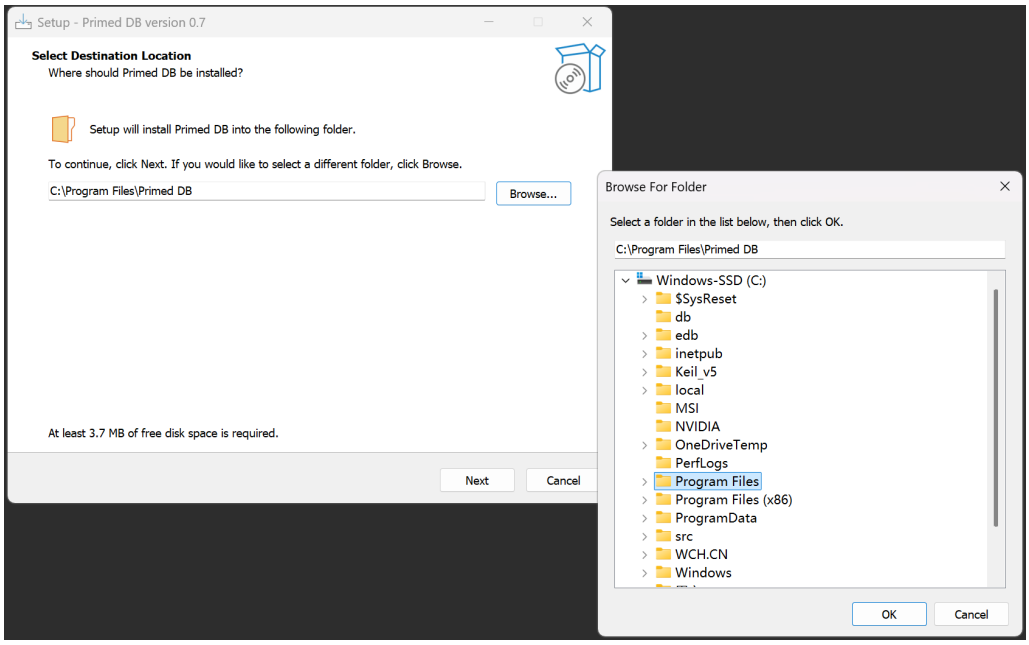

Figure 2.4: The interface of select an installation directory

In the end, the user can choose to **have a shortcut icon on the desktop** if the option of *creating a shortcut* has been chosen (see Figure 2.5).

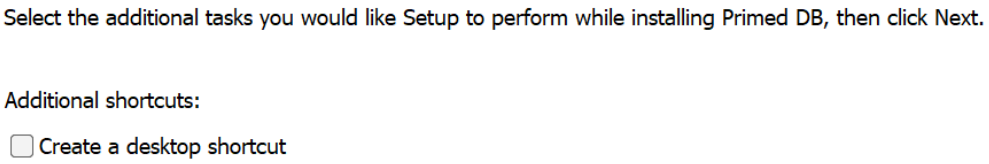

Figure 2.5: Choose whether create a shortcut on desktop or not

The installer displays Figure 2.6 once the **application has been successfully installed** on the user's device. If the user opted to create a desktop shortcut during installation, then it will be available on the desktop. On the other hand, user can choose to launch the application immediately or launch it later via a desktop shortcut.

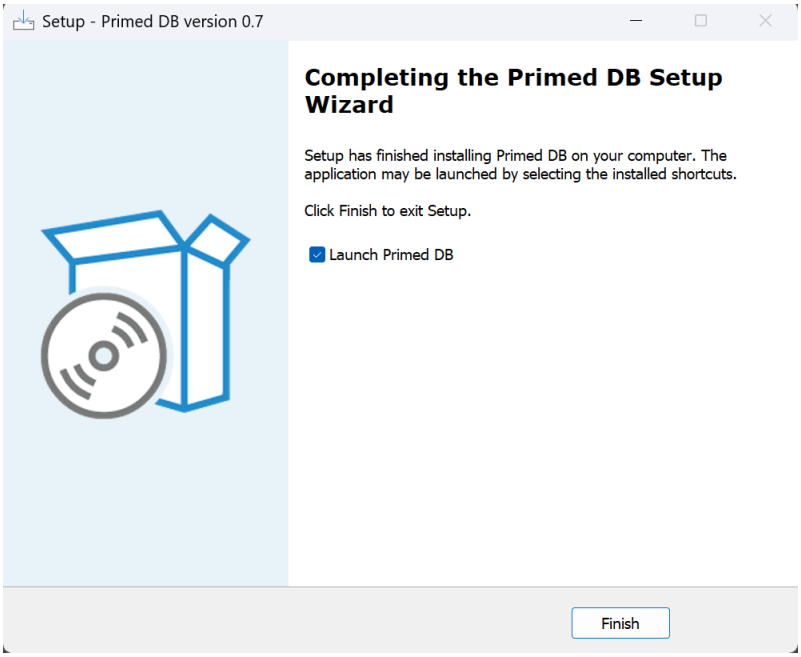

Figure 2.6: The figure indicate that PrimedDB is already installed on user's device

The program is **correctly installed** and runnable on the user's device when the user clicks the shortcut and **sees the console interface** of Figure 2.7.

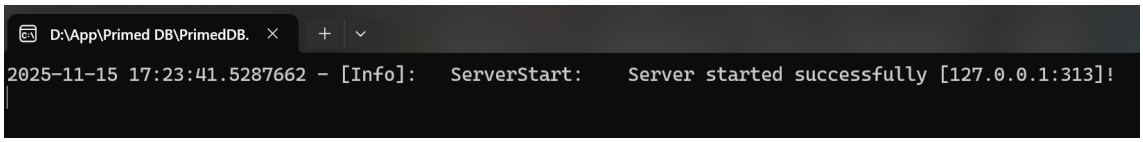

Figure 2.7: The console interface of PrimedDB

On the other hand, the executable file could be found in the installation directory (see Figure 2.8) if the user does not create a shortcut on the desktop.

Figure 2.8: The installation directory of PrimedDB

The shortcut exists on user's desktop when *"create a desktop shortcut"* (shown in Figure 2.5) has been chosen (see Figure 2.9).

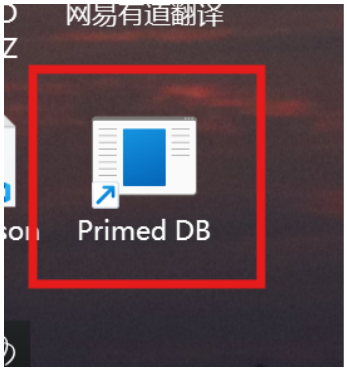

Figure 2.9: The desktop shortcut of PrimedDB

Meanwhile, the user can uninstall **PrimedDB** either in Windows *"setting/application"* or double click the uninstaller under the installation directory.

2.1.2 Client

The installer and the source code are available on **GitHub** [6]. Users can choose whether to download the **Windows x64 installer** to install the application or the **source code archive** to compile it for a specific platform (see Figure 2.10).

The installation package exists in the folder selected by the user before downloading. After a double click, the user can **choose a language for the installer software** (see Figure 2.3), English or Hungarian.

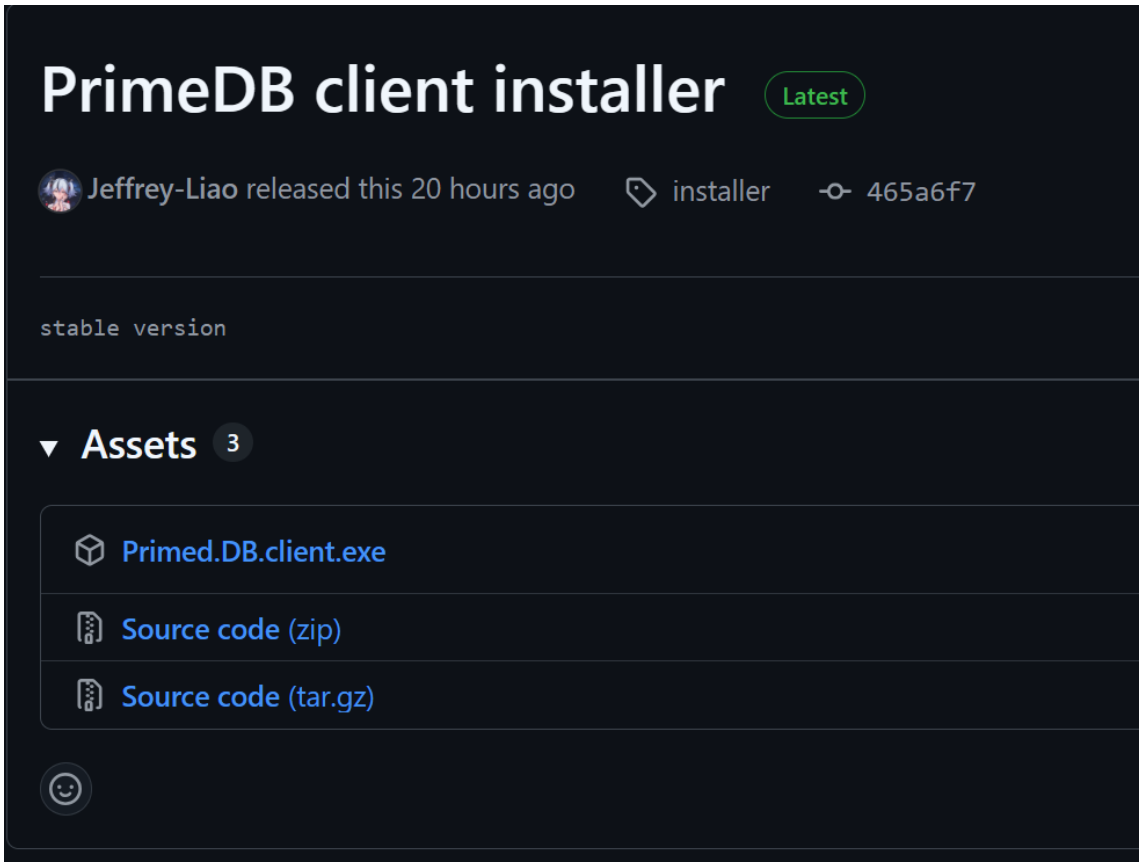

Figure 2.10: The web page for download Primed DB Client

After that, user could select an appropriate directory to install the application (see Figure 2.11). Next, **PrimedDB creates a new directory for the executable** file and necessary files, which is named `/PrimeDB Client`.

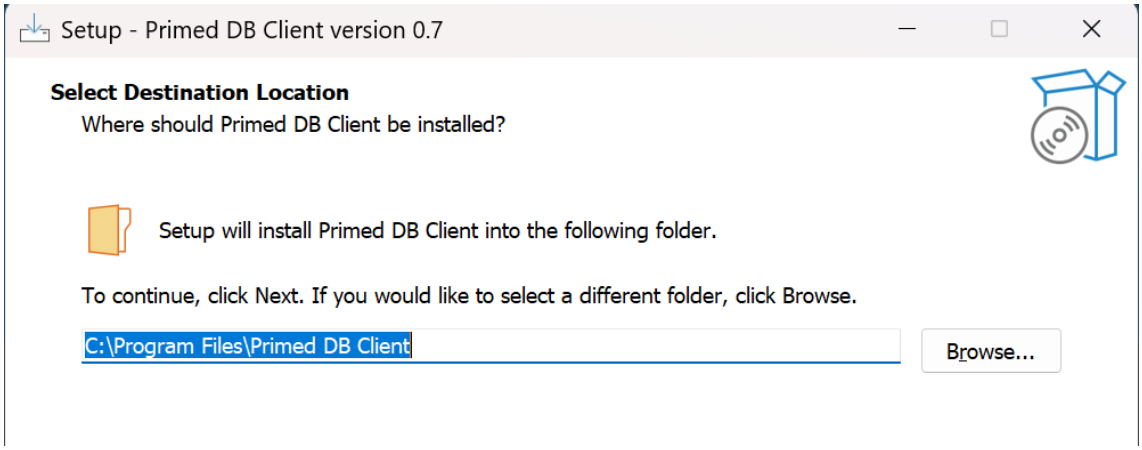

Figure 2.11: Directory selection interface for client

In the end, the user can choose to **have a shortcut icon on the desktop** if the option of *"creating a shortcut"* has been chosen (see Figure 2.5).

The installer is shown in Figure 2.12 when the **application is successfully installed** on the user's device.

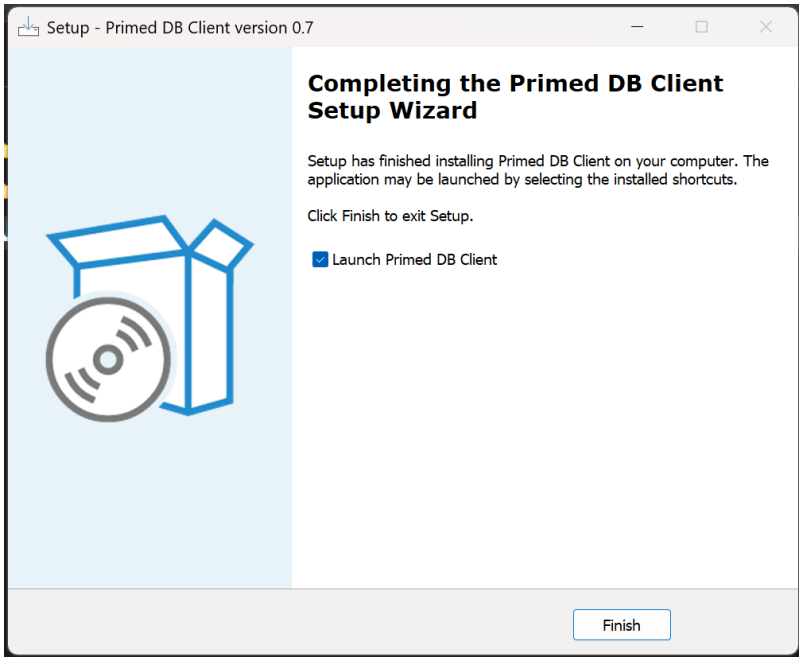

Figure 2.12: The interface which indicate client installed successfully on user's device

Meanwhile, the user can find a shortcut on the desktop if the user decides to have a shortcut during the installation process.

The program is **correctly installed** and runnable on the user's device when the user clicks the shortcut and sees the user interface Figure 2.13.

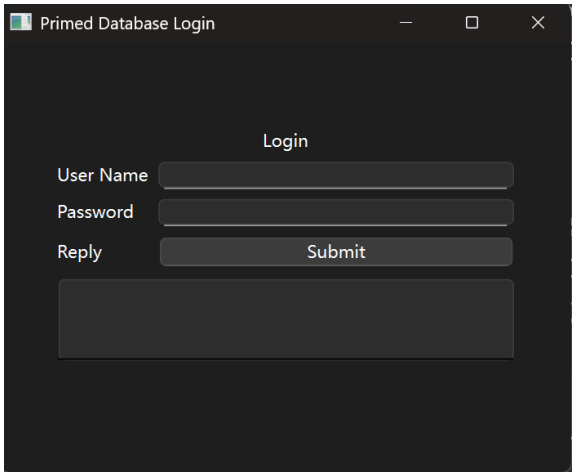

Figure 2.13: The running interface of PrimeDB Client

Meanwhile, the user can uninstall **PrimedDB** either in Windows *"setting/application"* or double click the uninstaller under the installation directory.

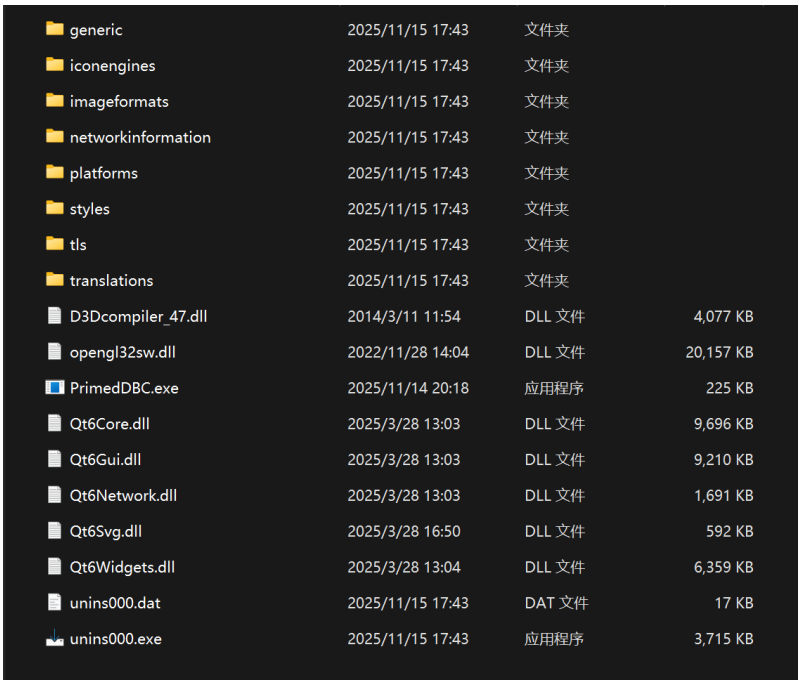

Figure 2.14: The installation directory of PrimedDB Client

On the other hand, the executable file could be found in the installation directory (see Figure 2.14) if the user does not create a shortcut on the desktop.

The shortcut exists on the user's desktop when *"create a desktop shortcut"* (shown in Figure 2.5) has been chosen (see Figure 2.15).

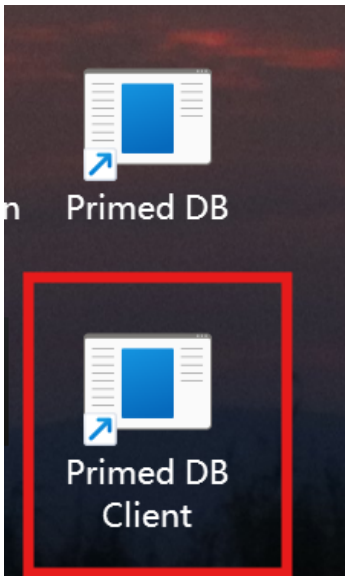

Figure 2.15: The shortcut of Primed DB Client

2.2 Application Usage

2.2.1 Start the application

The server must be **started before the client application** as it provides user authentication and SQL query execution. However, the **client can run without the server application** but fails to log in and displays only the interface, as shown in Figure 2.16.

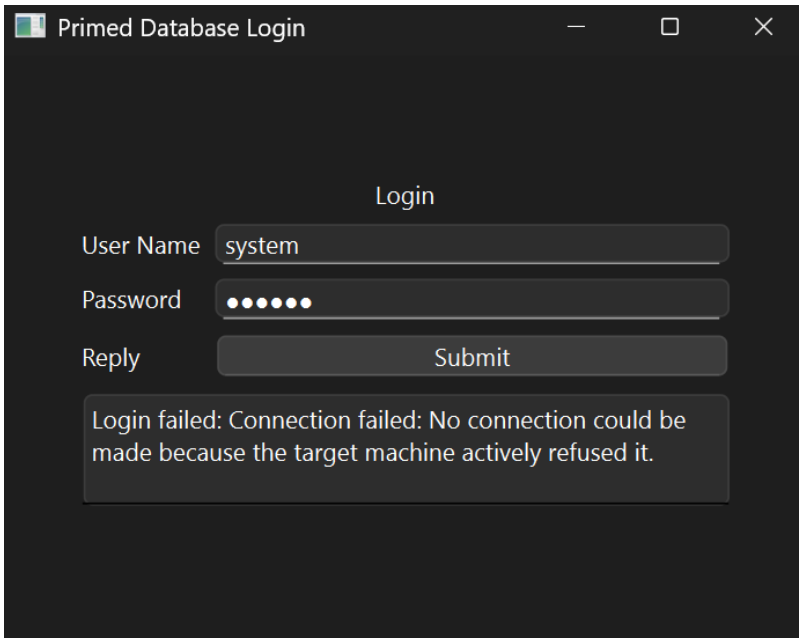

Figure 2.16: If the client run without server

Only one instance of the server application can run at any given time. Therefore, the message (see Figure 2.17) is shown in the terminal when more than one server application is started.

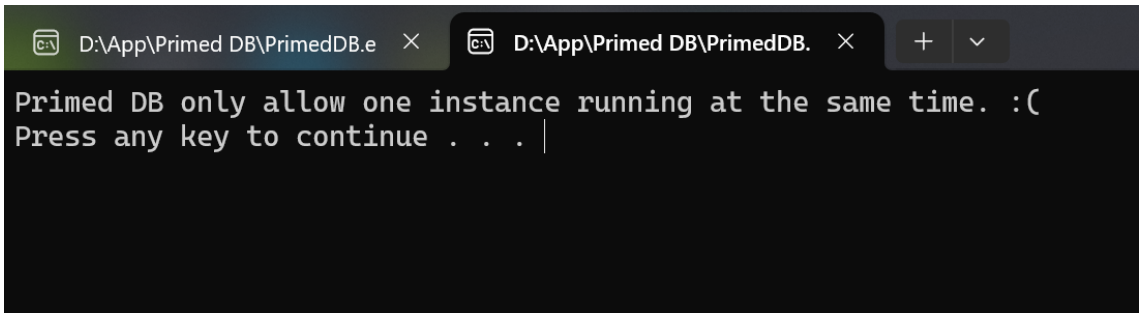

Figure 2.17: The error message when starting more than one server

2.2.2 During the application running

The client shows the **main interface** (shown in Figure 2.18) after the server application verified the identity of the user.

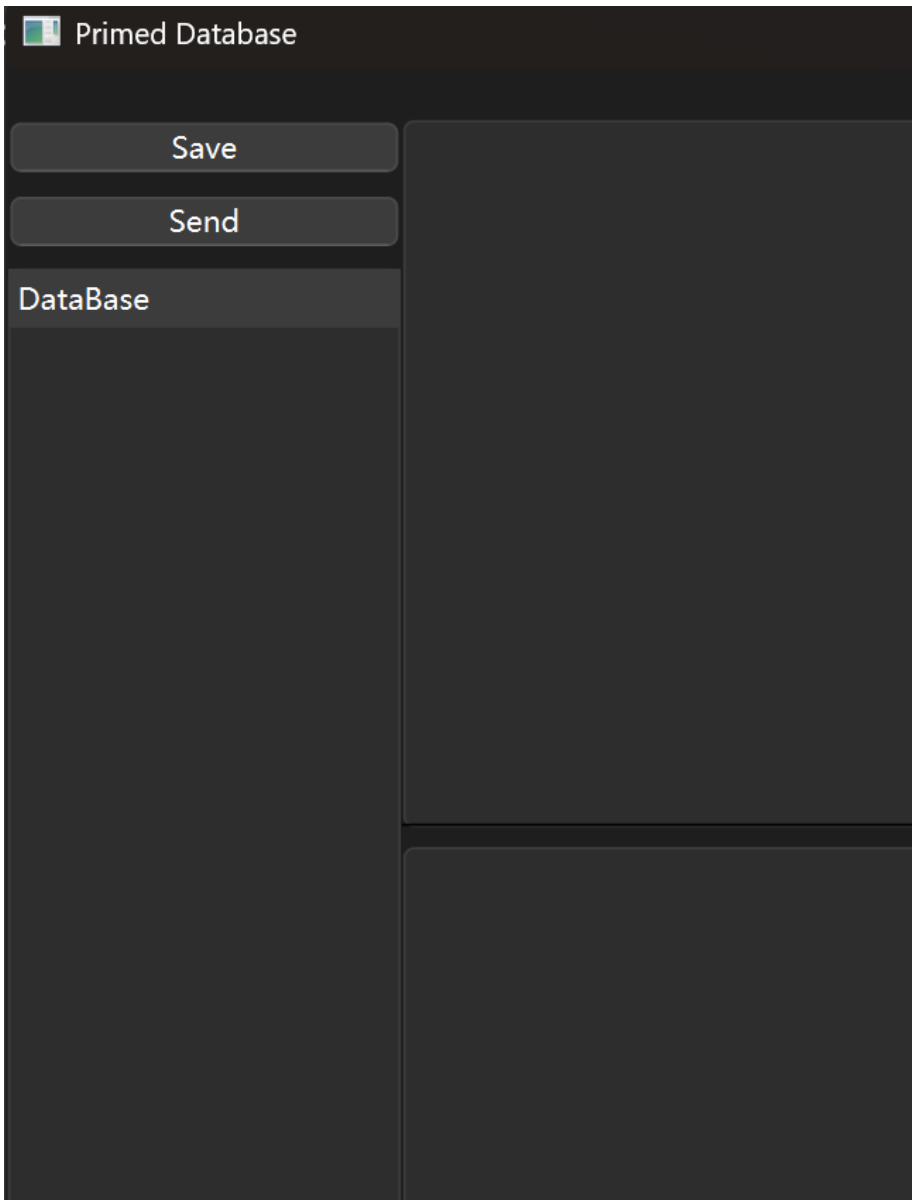

Figure 2.18: The main UI of client after successful log in

The **left** side of the client's main UI (see Figure 2.19) contains **two buttons** and a **schema renderer**. The schema renderer displays the table structure sent from the server, including the table name, column names, and column data types.

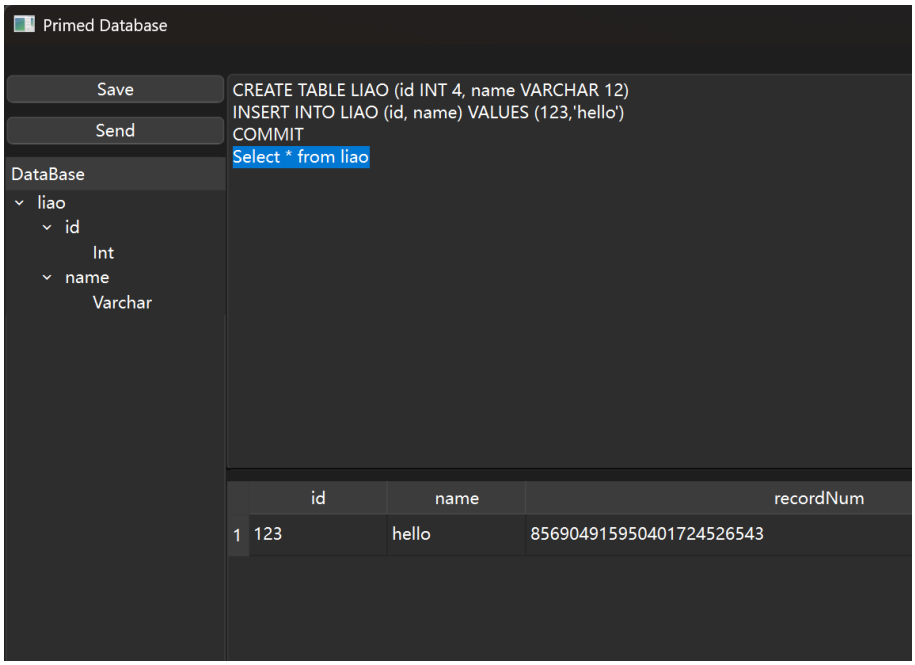

Figure 2.19: The main client UI with table structure.

On the right side, the upper zone contains the **editor**, where users can compose SQL statements and save them to a `".sql"` file using the *Save* button or send them to the server using the *Send* button. The lower zone is the **table view**, which renders the **data received from the database** (see Figure 2.20).

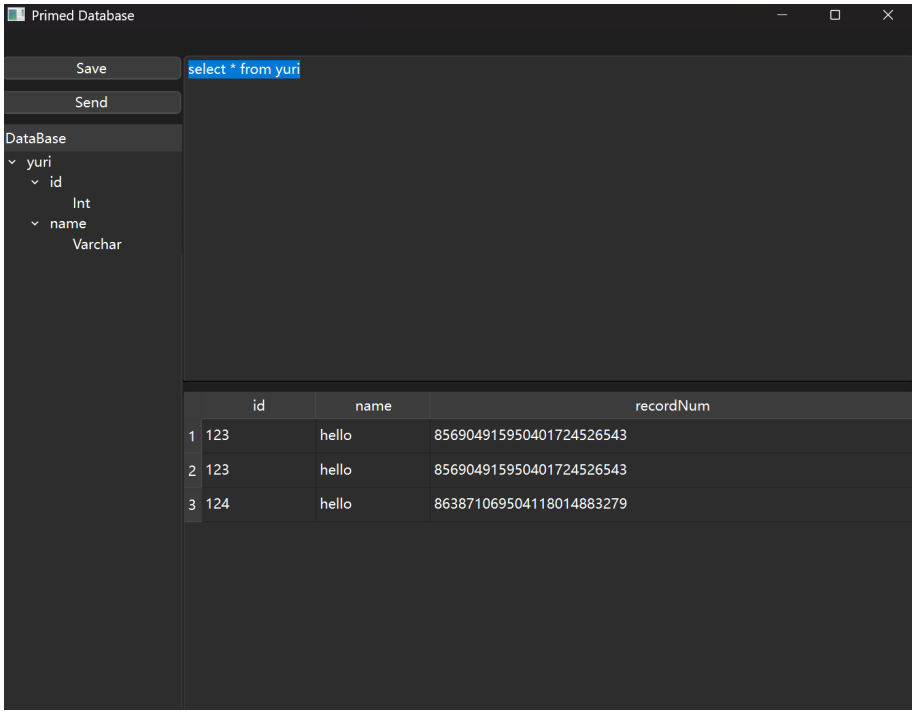

Figure 2.20: The client main UI with actual schema data and query result

The *Send* button sends either all SQL statements in the editor or only the portion selected by the user to the server.

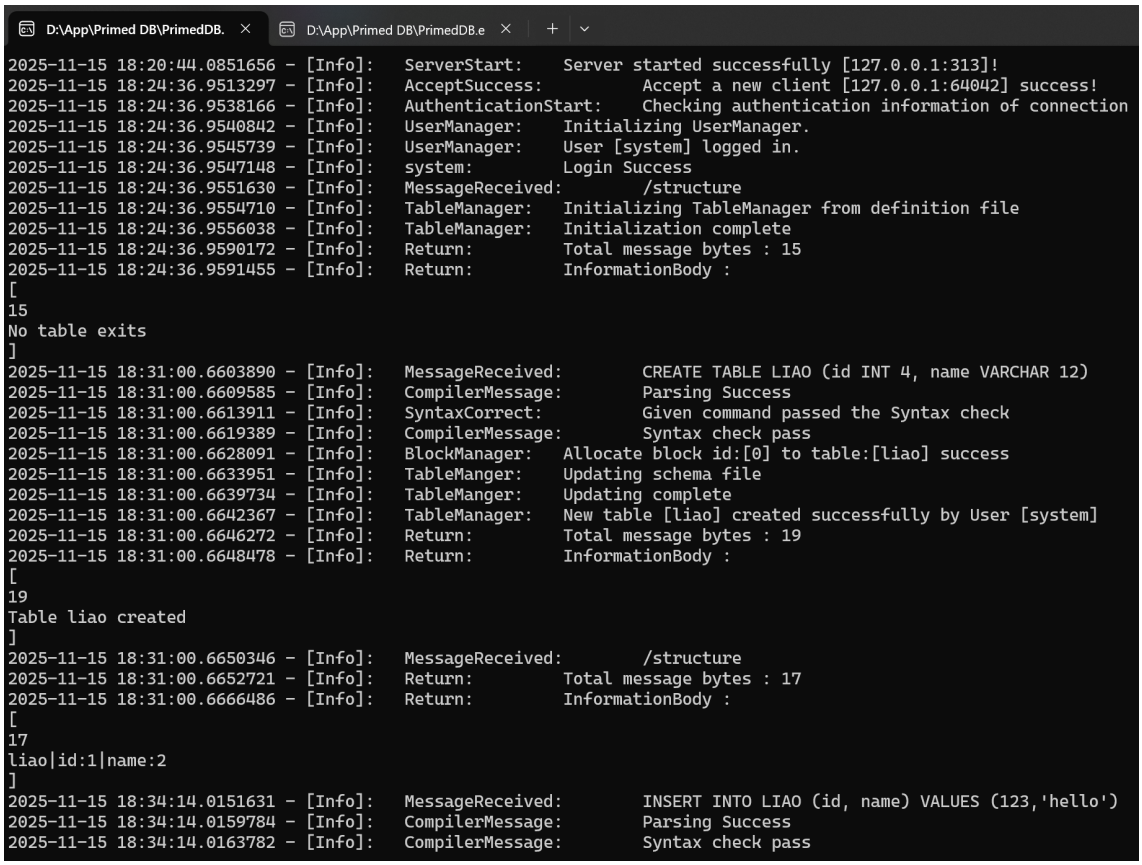

Figure 2.21: The server interface of PrimedDB

The server console UI (see Figure 2.21) prints all important messages in format: `"time-[message type(Info, Warning, Error)]: message category - message"`.

Meanwhile, all error messages are written to the file `/log/error[date].log` for later analysis. Additionally, the `Return` information category corresponds to network messages sent from the server to the client. The number before the reply indicates the message size in bytes.

Most importantly, the messages with `primize` and `deprimize` categories carry the results of the corresponding `primize()` and `deprimize()` functions (described in chapter 4).

The server will terminate and send the **terminate signal** to all active clients when the server receives an `"exit"` from an administrator via the server console. The Figure 2.22 illustrates a successful exit of the **PrimedDB** server.

Figure 2.22: The termination message of PrimedDB

2.3 Prohibited operations

2.3.1 Client

The client is built with **QT6 framework**. Thus, the installation directory contains numerous dependencies; the user **must not delete any of the folders or files shown in Figure 2.23**.

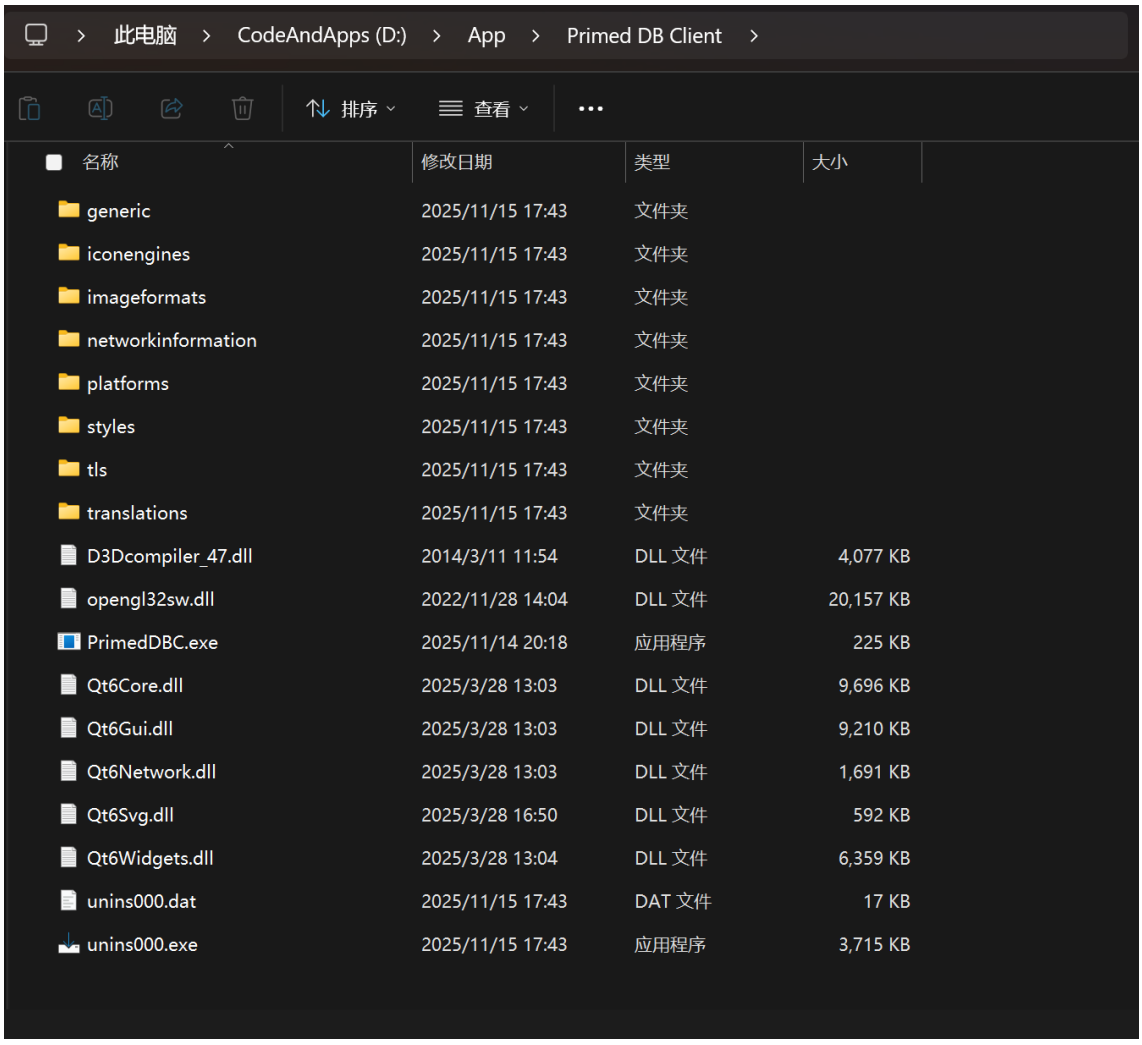

Figure 2.23: A figure illustrate all necessary files and directories for PrimedDB Client

2.3.2 Server

The server **must not be shut down until the `"exit"` command is executed**, which terminates all client connections before the server shuts down.

The table schema definitions and user configuration are stored in the `/global` directory as *`tables.def`* and *`user.def`*, respectively. Therefore, users must not delete or modify the `/global` directory or its contents.

Meanwhile, the `/data` directory stores all data in binary format in files named *`[table_name].dat`*. Therefore, users **must not modify or delete this directory to prevent accidental data loss** (see Figure 2.24).

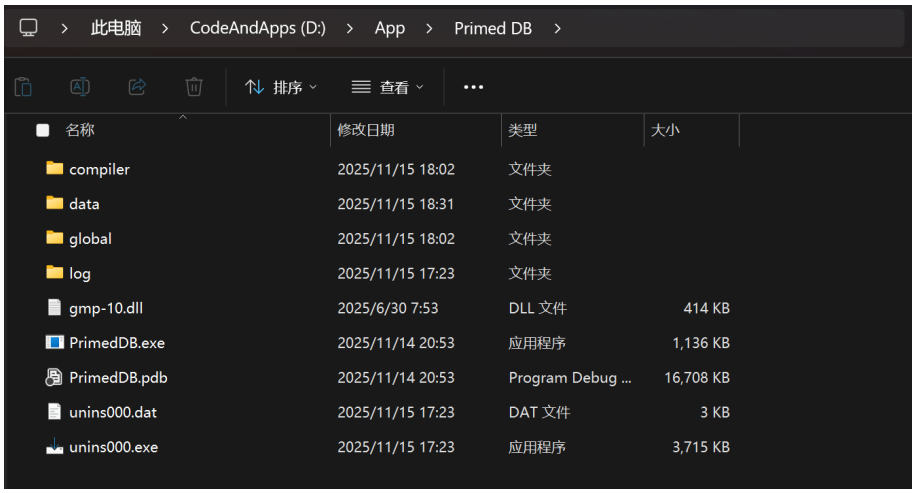

Figure 2.24: A figure illustrate all necessary directories for PrimedDB Server

This chapter outlines the installation process and usage of the **PrimedDB** server and client; chapter 3, which follows, presents the mathematical foundations underlying **PrimedDB**.

Chapter 3

Background knowledge

This chapter provides an introduction to the mathematical foundations of **PrimedDB**, including Euclid's theorem on the infinitude of primes and the application of the Fundamental Theorem of Arithmetic.

3.1 Mathematical base of PrimedDB

3.1.1 Infinity prime numbers

First, a fundamental function for **PrimedDB** is the prime-counting function $\pi(x)$, which estimates the number of prime numbers in domain ($[0, x]$).

Theorem 1 $(\pi(x) function)$**.**

$$\pi(x) = \{p \,|\, p \leq x \wedge is\_prime(p)\} \tag{3.1}$$

$\pi(x)$ *also can be written like:*

$$\pi(x) \approx \frac{x}{ln(x)} \tag{3.2}$$

when x is large enough [7].

Also, another very important mathematical theorem is:

Theorem 2 (Number of Primes)**.** ***There are infinitely many primes*** *[8].*

This indicates that **PrimedDB** has to **limit the input set of the hashing function**.

3.1.2 Map all data into prime numbers

Definition 1 (Enlarge Index)**.** The *enlarge index*, denoted ei, is a scaling factor that maps each original number to a corresponding value in an expanded index set.

$$ei = 2^n,\ (n > 0) \tag{3.3}$$

Most importantly, the new set must contain **as many primes as the size of the original numbers**. Thus, a valid enlarge index should be chosen with the following rule:

Definition 2 (Verification rule for enlarge index)**.** The verification rule for enlarge index checks **whether the total count of prime number in new domain is more than original domain** or not. The valid space represents the total amount of prime number in the current space is bigger than the count of numbers in original set.

$$x \leq \pi(x * ei)(x = max(\{n|n \in [x, y]\}) \tag{3.4}$$

Definition 3 (Prime Number Hash Hashing Function)**.** The prime number hashing function is the **key for PrimedDB**, which uses **definition 1** to map a given number to a new set of values.

Also, the **Prime Number Hashing Function** is a bijection that maps an original number to a prime number hash key using the formula below and recovers the original number from the prime hash key by dividing by ei.

$$phash(x) = next\_prime(x \cdot ei),\ (\{x|0 \leq x \leq 2^n\} \wedge 2^n \leq \pi(2^n * ei)) \tag{3.5}$$

Definition 4 (Prime Hash Space)**.** The **Prime Hash Space** (PHS) is the set which is generated by the **Prime Number Hashing Function**, which can be defined as:

$$phs(x) = \{p \,|\, (x \cdot ei \geq p \geq ei \vee p = 2) \wedge is\_prime(p), (\{x|0 \leq x \leq 2^n\} \wedge 2^n \leq \pi(2^n * ei)) \tag{3.6}$$

There are some special values:

1. The lower bound is 2 because the next prime number for 0 is 2.
2. The upper bound is smaller than $2^n * ei$.

Figure 3.1 visualizes the **Prime Hash Space** and its relationship to the original data set.

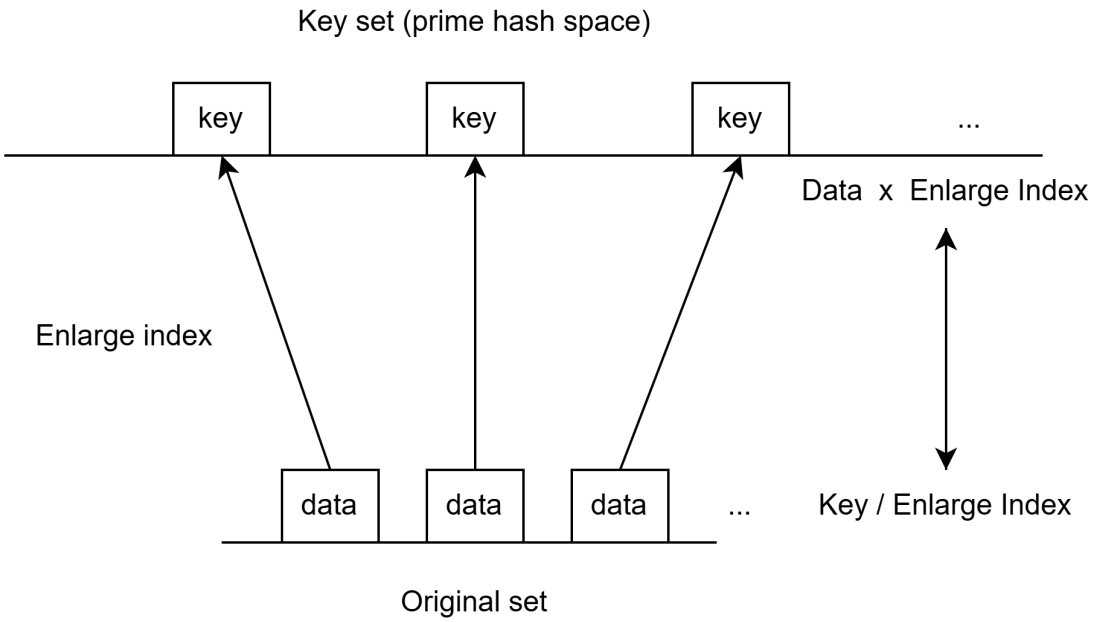

Figure 3.1: The relation between original data space and PHS

Name	In total	For each record	Space complexity
Traditional DB	$n_r * O(n_s)$	$O(n_s)$	$O(1)$
prime DB	$O(1) + n_r * O(1)$	$O(1)$	$O(n_r)$

n_s:Length of string. n_r:Number of records

Table 3.1: The comparison of time and space complexity between traditional and PrimedDB query with string type.

3.1.3 What is PrimeDB

Theorem 3 (Fundamental Theorem of Arithmetic)**.** *$\exists n$ is a natural number. n can be expressed as a product of prime numbers in only one way [9].*

PrimedDB converts each column of a record into a prime number (illustrated in Table 3.2) and then multiplies all resulting primes together to form a record key that uniquely identifies the combination of values in that record.

Furthermore, the database query process can be described as follows: converts all condition values into prime-number keys (with *primized()* illustrated in **Primize Function**), multiply them together to a product, and then computes the remainder of dividing each record key in the table by this product. If the remainder is zero, it indicates that the given data are present in the current record (supported by Theorem 3).

record key	name	phone number	GPA
Original data	Liao Runkang	123456	4.3
1144586733202795253498899 20	2399497119	3018604160	15802373

Table 3.2: The table structure example of traditional relational database and PrimedDB

In conclusion, **PrimedDB** chooses an appropriate **Prime Hash Space** with Theorem 1. After that, **PrimedDB** generates a prime number hashing key and decodes it back to the original data with **enlarge index**. Furthermore, **PrimeDB** converts the data in each column of a record into prime numbers and takes their product as the record key. Meanwhile, **PrimedDB** uses the record key with Theorem 3, which enables **PrimedDB** to support fast queries for all data types.

The next chapter 4 shows how all definitions and theorems are applied in **PrimedDB** implementation. Moreover, the concurrent design and the SQL compilation feature are also illustrated.

Chapter 4

Developer documentation

This chapter explains the design and implementation of **PrimedDB**, including the storage unit, the SQL compiler, and the transaction handling system. Most importantly, **Primize Function and Deprimize Function**[1] illustrates how the two functions are implemented based on the definitions provided in chapter 3.

4.1 The goal and scope of PrimedDB

Firstly, the main task of **PrimedDB** is to **give a new idea** which optimizes relational database with faster query speed in theory and more efficient self-encryption for all inserted data.

Nevertheless, the prototype presented in this thesis lags behind commercial databases in query performance due to their extensive optimizations—such as pruning redundant operations and refining query execution plans [10].

Thus, this demo provides only the core functionality needed to **demonstrate that the `primize()` and `deprimize()` functions can successfully convert at least two distinct data types into prime-number representations** and to validate the underlying idea that **querying via prime factorization is feasible**.

[1] 'Pimize' and 'deprimize' are two artifact words that describe the processes of hashing a regular number into a prime number hash key (with an enlarge index) and reversing the operation to recover the original number from the prime hash key, respectively.

4.2 Naming and coding format

The **PrimedDB** source code follows the camelCase naming convention for variables, instances, and function names. In contrast, static variables, functions, and class-level constants begin with an uppercase letter. Also, the **constant variable and instances** use full uppercase letters for naming.

Moreover, the code presented in this thesis **omits certain irrelevant parts** and renames some original functions to improve clarity. However, the original function names are retained alongside the modified signatures for reference. Importantly, these simplifications affect only the function names—not their signatures.

```
1 void test() // [original function name.]
2 {
3   //some example codes
4     cout<<hello world;
5 }
```

Code 4.1: The format of coding part

4.3 Development environment

This section introduces the development environment and settings for **PrimedDB**, which is an **open source** project that is written under **the C++20 stand**. It allows programmers to improve the database or attribute new code.

4.3.1 Open source libraries usage

It is impossible that one project did not use any open source library in modern software development. Thus, **PrimedDB** includes several battle-tested libraries to provide implementations that are highly stable and fast.

GNU Multiple Precision Arithmetic Library (GMP) [11]

GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. There is no practical limit to the precision, except for the ones implied by the available memory in the machine GMP runs on.

In **PrimedDB**, the **GMP** library is the cornerstone of the `primize()` function, which provides the finding of the next prime function, the prime number checking function, and exponentiation.

On the other hand, the `primize()` function limited the input domain to $[0, 2^{32}]$ because the **GMP uses the assembly to implement some calculations** and the maximum size of the x64 architecture CPU register is 64 bits [12, 13, 14, 15]. Thus, the prime number hashing key for the data is 64 bits, which can maximize the ability of the **GMP** implementation.

Crypto++ [16]

Crypto++ is an open source library that is implemented in C++ and STL. The library provides efficient implementation to generate various types of hash key including MD5, SHA128 and SHA256.

In **PrimedDB**, the identifiers for tables, pending connections, transactions, and users are generated by the Crypto++ library.

Boost.Asio [17]

Boost.Asio is the subsection of a cross-platform C++ library for network and low-level I/O programming that provides developers with a consistent asynchronous model using a modern C++ approach. It is a component of the widely used Boost library. The network features of both the server and the client side of **PrimedDB** are implemented based on boost.asio.

4.3.2 IDE used

Visual Studio 2022 community edition (*VS2022C*, see Figure 4.1) is the IDE to develop **PrimedDB**. It was selected for its simple interface and cross-platform debugging support (via WSL2 on Linux). Moreover, the database adopts the Visual Studio project structure instead of CMake to improve development efficiency.

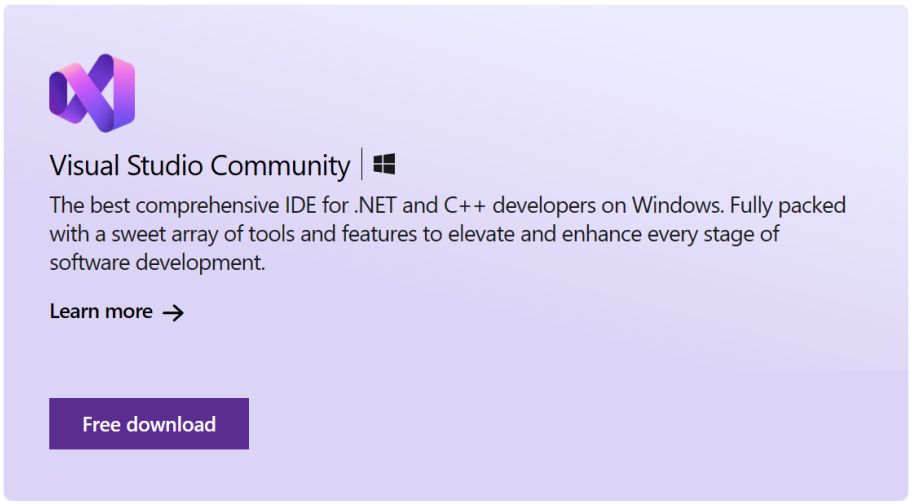

Figure 4.1: The Visual Studio Community Edition

Configuration

First, the C++ language standard must be set to ISO C++20 in the project property settings (see Figure 4.2) [18].

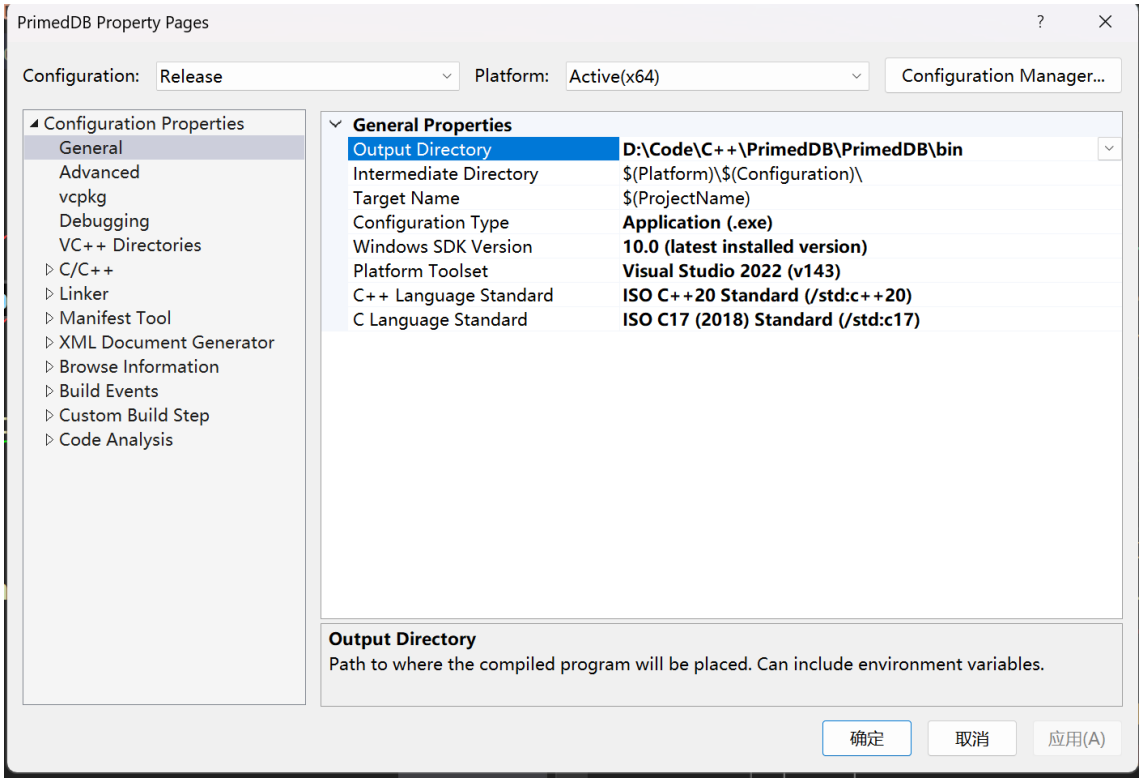

Figure 4.2: C++ Standard Configuration

After that, the open-source libraries (listed in **Open source libraries usage**) should be installed and integrated into the project structure by specifying the library

directory (see Figure 4.3).

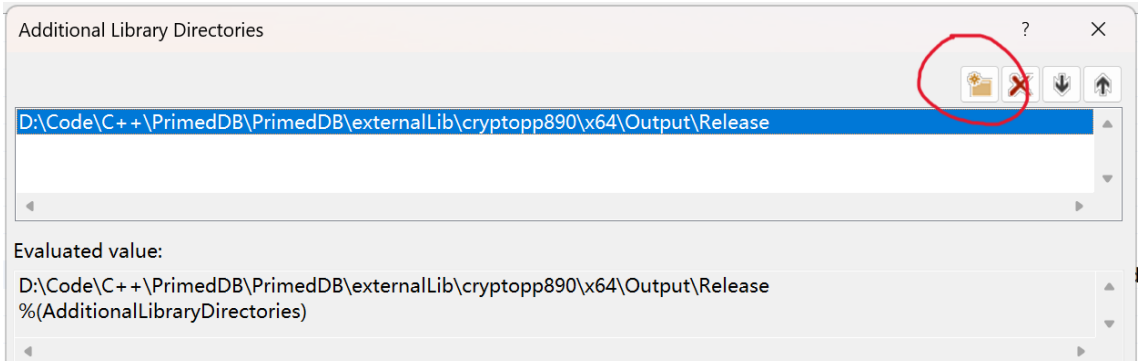

Figure 4.3: Import a downloaded library

Furthermore, the name of the library file should be specified in the *Linker* → *Input* → *Additional Dependencies* settings. (see Figure 4.4).

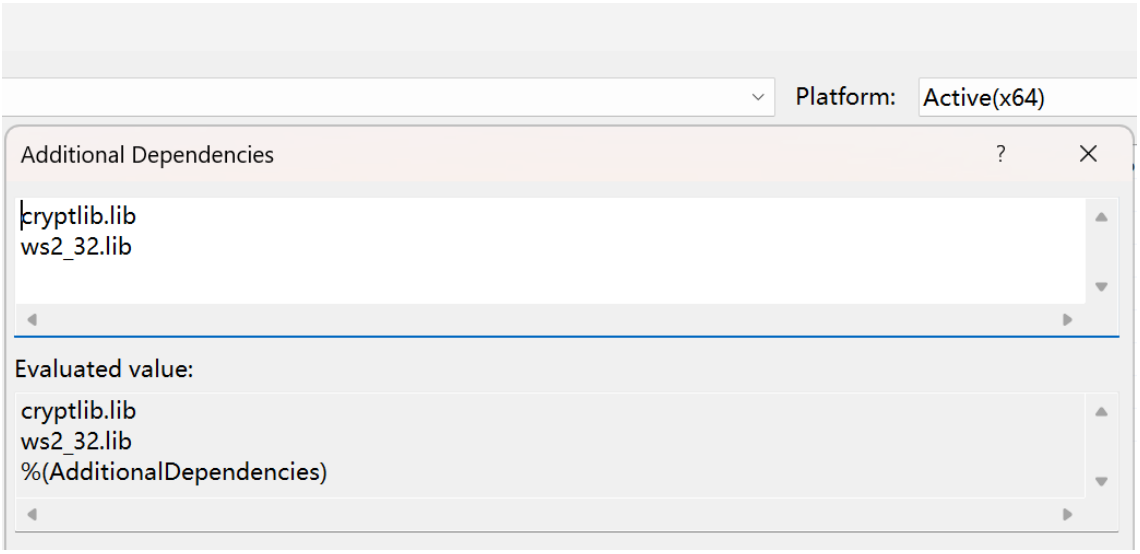

Figure 4.4: The name of .dll or .lib should be placed into additional dependencies blank

4.4 Multi-threads design

The database system handles query requests from multiple clients, which required all independent components to collaborate with each other asynchronously.

The Primed DB had **5 + n threads** (demonstrated in Figure 4.5):

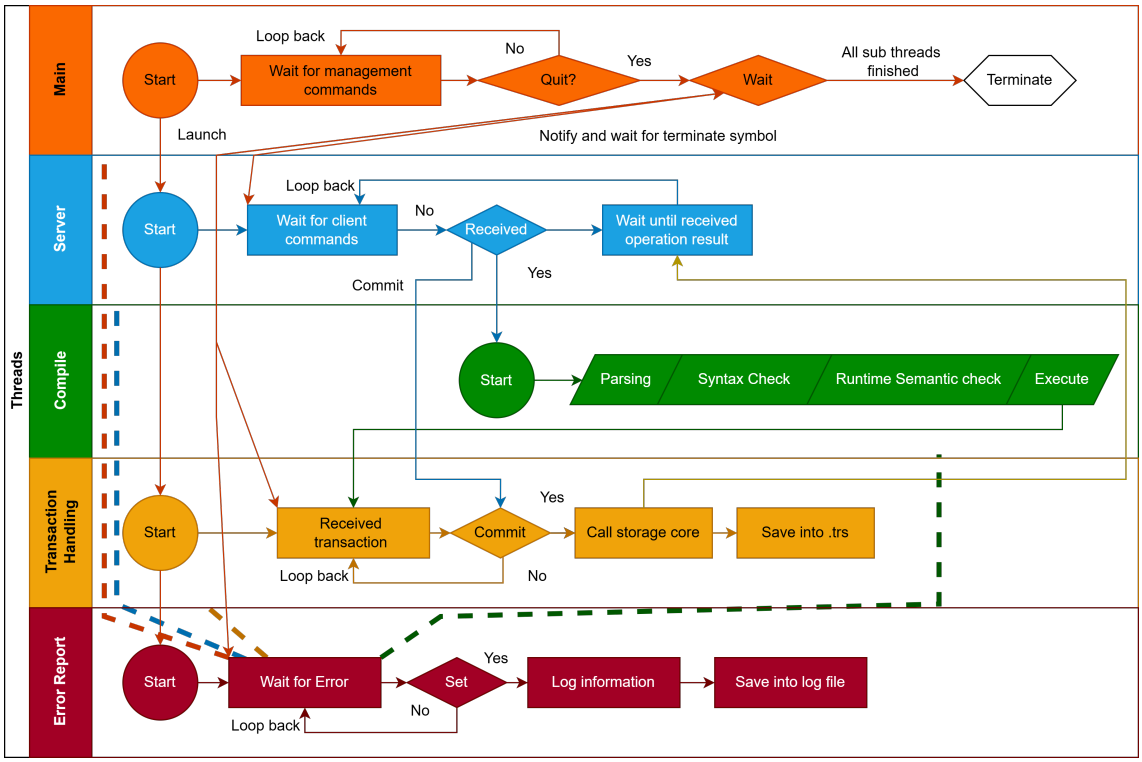

Figure 4.5: The multi-thread structure of Primed DB

1. **Main thread**: which runs the server `io_context` on another thread and waits for administrator commands.

2. **Client thread**: although it is out side from **PrimedDB.exe** but it is part of **PrimedDB**, which applies SQL operations on server application.

3. **Error handling thread**: which resolves all errors by pre-registered error handling functions when other threads submit a new error into the thread.

4. **Server thread**: this thread accepts new connections and communicates with established connections..

5. **Transaction handling thread**: it handles all submitted transactions asynchronously and converts all transactions into operations.

n. **Each compiling task** is a separate thread, which survives until the end of compiling task. Also, a future handle is left to the caller, and the thread terminates when the promised result is written into the future handle.

Moreover, error and transaction handling threads are not launched until the server thread receives the first client connection. Thus, **the first sql execution on PrimedDB might be slower than others**.

4.4.1 Main thread

The main thread serves as the **entry point for PrimedDB**, launching the server and **listening on port 313** in a separate thread (see **Server thread**). The port number can be changed in future versions via an administrator command.

After that, the main thread is waiting for the management command like `exit` to terminate the server or do other pre-defined actions. After the server receives an `exit` message, the main thread starts the closing session. The **closing session** closes all open sockets and **sends a termination message to all clients** that the server is terminated, since the administrator exits the sever application.

4.4.2 Client thread

Client threads do not necessarily run on the same device as the server application. It requires the user authentication service provided by the server and sends SQL requests to the server when needed (the transmission protocol explained in **Data transmission**). Moreover, the server thread executes DDL statements only after the user thread sends a `commit`; until then, all new transactions created by the compiler are temporarily stored in the user cache (see Figure 4.6).

The client thread is terminated under the following circumstances:

- the user explicitly closes the application;
- the server initiates its closing session;
- the client fails to connect to the server;
- the connection is refused by the server.

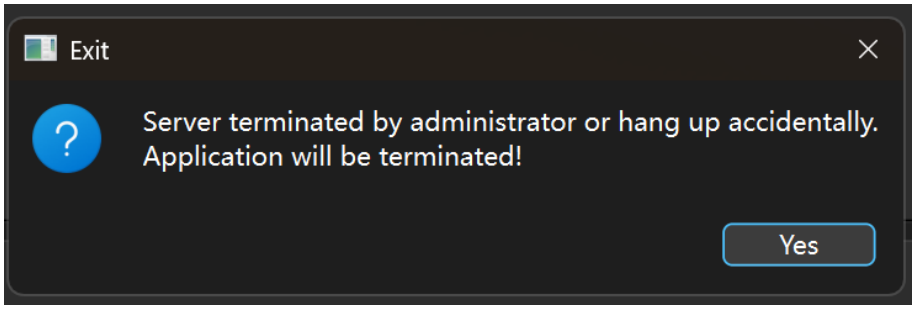

Figure 4.6: The interface of client received terminate signal or server terminated accidentally.

4.4.3 Error handling thread

The error handling thread receives error messages from all other threads and then prints and saves them. Meanwhile, any thread can register an error handler for a specific error name. The section 4.9 explains all operations related to error handling.

4.4.4 Server thread

The server thread accepts connections and reads SQL requests from different clients. It responds with messages in a specific format (see Data transmission).

The server thread launches a **compiling task** which creates a new transaction or error message if compilation fails. All pending transactions are submitted to transaction handling thread when the server thread receives a `commit` message from the client thread.

4.4.5 Transaction handling thread

The transaction handling thread handles all transactions submitted from the server thread. The section 4.8.4 explains how the transaction processing thread collaborates with the block manager to handle transactions submitted to the thread.

4.4.6 Compiling task

Furthermore, the compiler creates a separate task when a client submits SQL statements. After statements compiled, transaction generated by compiler task pushed into a pending queue, which stored in the user who submit the statement, then sends to transaction handling thread when user commits all changes.

4.5 Data delivery

4.5.1 Connection states

Figure 4.7 lists all possible connection states of the **PrimedDB** client, with each state corresponding to distinct operations performed by the client application and its associated server process.

States	Applications	Explanation
NOT_LOGIN	Client	A client is started but login failed or not login yet.
LOGIN	Client	A client successfully passed the authentication check.
LOGIN_FAIL	Client	Either username or password incorrect or server is in state OFFLINE.
WAIT_AUTH	Client	A client sent username and password then wait for server to reply 'success'
WAIT_RESULT	Client	The client will change to WAIT_RESULT state after client sent a query statement to server but not received the data yet.
CON_FAIL	Client	The client will be move to CON_FAIL state when client is LOGIN but server is OFFLINE. Client will notify user and terminate immediately.

Figure 4.7: All possible connection states of PrimedDB client application

Figure 4.8 lists all possible states of the **PrimedDB** server, with each state corresponding to specific operations and cleanup tasks performed by the server application and its associated client connections.

States	Applications	Explanation
CONNECT	Server	Server hold a TCP/IP connection with client and wait for commands or SQL statements.
DISCONNECT	Server	Server received '/quit' from client which indicates the client is terminated.
TERMINATE	Server	Send '9\n terminate\n' to all clients as the server is shutdown by administrator with 'quit' command.
COMP_ERR	Server	The server will return error message when the SQL statement given by client either had error or not supported by PrimedDB compiler.
ONLINE	Server	Server is running correctly on host device.
OFFLINE	Server	Server shutdown accidentally or closed by administrator.
AUTH	Server	When a new connection established with server but not verified by authentication test.
AUTH_PASS	Server	A client successfully passed the authentication check and server will reply 'success'.

Figure 4.8: All possible connection states of PrimedDB server application

4.5.2 Login and authentication

PrimedDB consists of a server and multiple clients, namely **PrimedDB.exe** and **PrimedDBC.exe**, respectively. The server waits for new connections and re-

sponds to SQL statements. Clients cannot access the server until the authentication check is passed (see Figure 4.9).

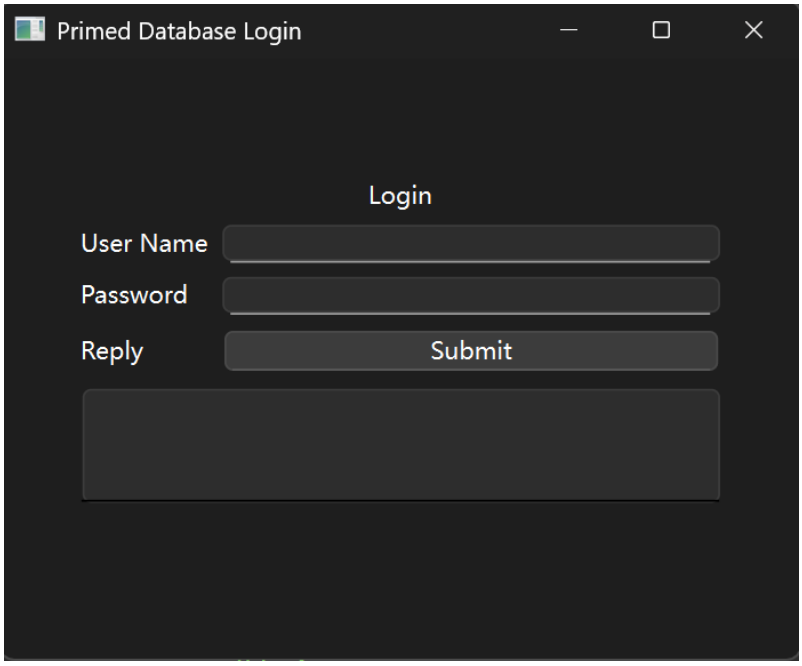

Figure 4.9: The login interface of Client

In the login process (illustrated in Figure 4.10), the server accepts requests sent by the client applications then puts them into a waiting queue of authentication (Code 4.2 illustrates how a client sends a request and receives the result).

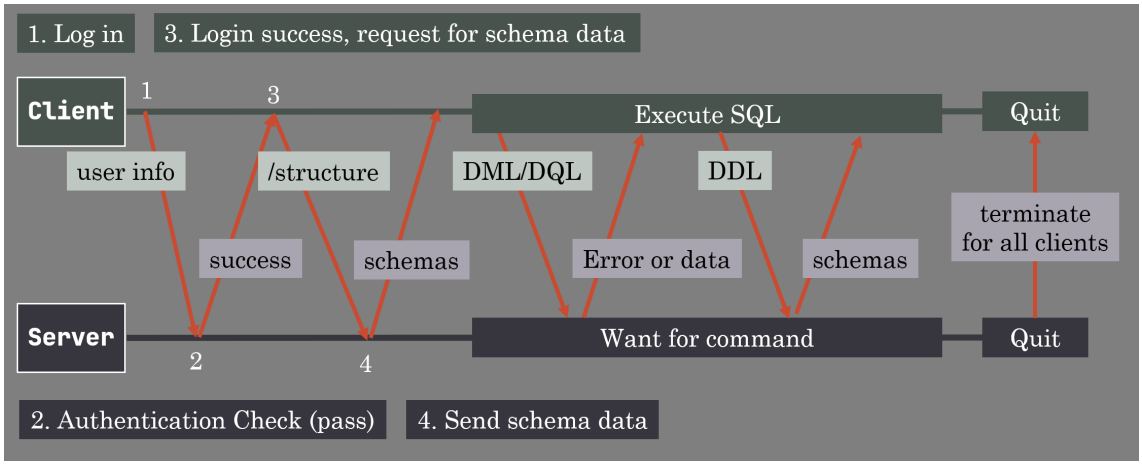

Figure 4.10: The visualization of login procedure and connection between server and client

```
void Client::login() // part of function [Login::onSubmitClicked]
{
  try {
    // Convert QString to std::string
    std::string username = m_userName.toStdString();
    std::string password = m_password.toStdString();

    //establish connection
    m_client = std::make_unique<Client>(localhost, 313, username,
        password);
    // Check connection and login status
    if (m_client->isOpen() && m_client->isLogin()) {
      m_loginSuccess = true;
      setServerReply(Login success);
      m_client->write(/structure);
      emit loginSuccess();
    } else {
      m_loginSuccess = false;
      // Get specific error message from server
      QString errorMsg = QString(Login failed: %1).arg(QString::
          fromStdString(m_client->getLastError()));
      setServerReply(errorMsg);
    }

  } catch (const std::exception& e) {
    m_loginSuccess = false;
    QString errorMsg = QString(Login failed: %1).arg(e.what());
    setServerReply(errorMsg);
  }
}
```

Code 4.2: The authentication and launching a session

The server checks the username and password (in hash format) sent by the client and waiting in the queue (shown in Code 4.3). In case the client receives `success` message from the server, it sends back a `\structure` message, and then it also receives the table schema from the server. In case of error, the client receives an error message.

```
void Server::validateUser(SocketPtr socket, const std::string&
    userName, const std::string& password, std::string& reply)
{
  auto& userManager = UserManager::Get();
  if (userManager.exist(userName))
  {
    auto ptr = PrimedDB::UserManager::Get().get(userName);
    if (m_user_clients.contains(ptr))
    {
      asio::write(*socket, asio::buffer(TERMINATE));
    }
    if (UserManager::Get().login(userName, password))
    {
      reply = Login Success;
      if (socket->is_open())
      {
        CallInfo(userName, reply);
        asio::write(*socket, asio::buffer(SUCCESS));
        auto ptr = PrimedDB::UserManager::Get().get(userName);
        this->m_user_clients[ptr] = socket;
        doRead(socket);
        return;
      }
    }
    else
    reply = Password incorrect.;
  }
  else
  reply = User not exist.;
}
```

Code 4.3: The authentication and launching a session

4.5.3 Data transmission

After authentication, the server sends table schemas (illustrated in Code 4.4), which are stored in `TableManager`, as formatted messages (like: `message size\n message body\n`). The client **decomposes the length of information** then **reads buffer from server until reach the size received**.

```
1  void Server::sendStructure(std::shared_ptr<asio::ip::tcp::socket>
       socket)
2  {
3    string structure = std::move(TableManager::Get().format());
4    if (structure.empty())
5    structure = No table exits\n;
6    writeMessage(socket,structure);
7  }
8  std::future<bool> Client::readPackage(std::shared_ptr<asio::
       streambuf> buf,int max)
9  {
10   auto result = std::make_shared<std::promise<bool>>();
11   int bytesLeft = max - buf->size();
12   if (m_socket.is_open())
13   {
14     asio::async_read(m_socket, *buf, asio::transfer_exactly(
         bytesLeft),
15     [this, buf, result](const asio::error_code& error, size_t
         bytes_transferred)
16     {
17       if (!error)
18       {
19         result->set_value(true);
20       }
21       else
22       {
23         result->set_value(false);
24       }
25     });
26   }
27   else
28   result->set_value(false);
29   return result->get_future();
30 }
```

Code 4.4: The send and receive schema information for all tables

In addition, the `readPackage()` function is used not only to receive the schema data but also to receive query and operation results. Furthermore, the return value of the function is passed by the handle of the future result because it invokes

`asio::async_read()` immediately when the outer function calls `future::get()`. Thus, the data arrived as expected if the operating system gives a lower priority to `asio::async_read()`. The data flow of **PrimedDB** is illustrated in Figure 4.12, which fully describes the end-to-end journey of the data from the moment it enters **PrimedDB** until it is returned to the client.

4.6 Convert any data into prime

The core functions of **PrimedDB** are `primize()` and `deprimize()`, which converts the given memory into a $2n$-byte record (where n is the size of the original data in bytes) by encoding it with a prime number, and subsequently restores the data from this prime number hashing key format back to its original form.

On the other hand, `primize()` generates a record key for queries by multiplying all field values in the record. Thus, the query operation is reduced to a modulo operation between the record key and the product of the prime-encoded target conditions (see Figure 4.11).

Category	Record key	Name	GPA	Phone number
Original	N/A	Liao Runkang	4.3	123456
Prime number	114458673320279525349889920	2399497119	3018604160	15802373

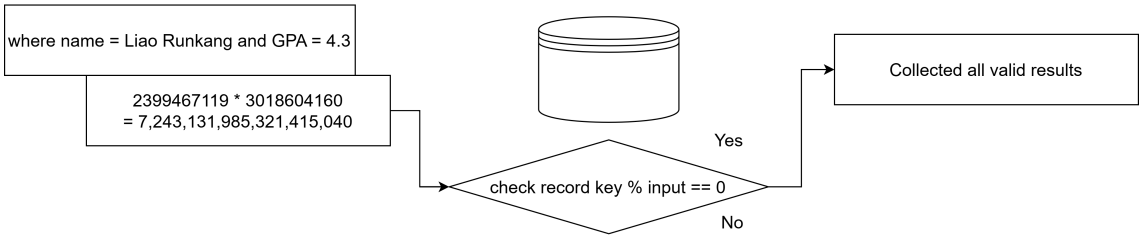

Figure 4.11: The query procedure of PrimedDB

Most importantly, the prime number generated by `primize()` is a prime number hash key, which produces the **same result for the same input** (see Figure 4.13). Meanwhile, **the hash key itself is also the value**, which can be divided by the enlarge index to restore the original data.

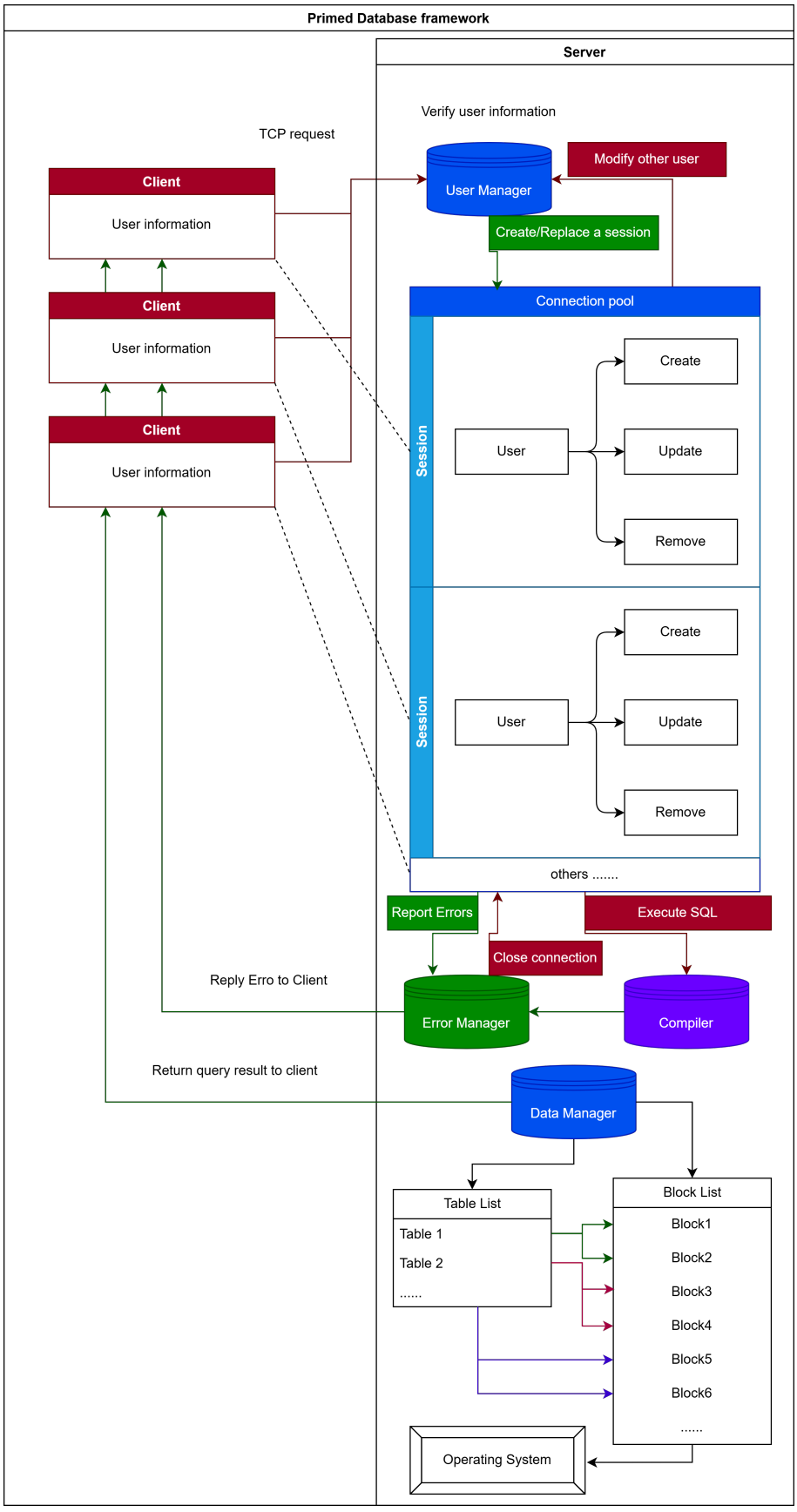

Figure 4.12: The data flow framework of PrimedDB

	id	name	recordNum
1	123	hello	856904915950401724526543
2	123	hello	856904915950401724526543
3	124	hello	863871069504118014883279

Figure 4.13: The query result which shows same input having same prime number hash key

4.6.1 Four bytes

Definition 1 and Theorem 2 imply that the hashing function must restrict its input to a suitable domain—specifically $[0, 2^{32}]$, since the general-purpose registers of modern x86-64 CPUs are 8 bytes (64 bits) wide [12, 13, 14, 15]. Thus, the GMP library achieves optimal performance through assembly code that exploits 64-bit (8-byte) CPU registers. This enables efficient handling of calculations up to 2^{32} bytes inputs and generating a 2^{41} bytes prime number hash key.

The enlarge index is 512 (2^9) because the largest gap between two consecutive prime numbers in the interval $[0, 2^{32}]$ is **292**, which can be verified by Theorem 1 and Definition 2($\frac{2^{32}*2^9}{ln(2^{32}*2^9)} = 77,378,535,258 > 4,294,967,296(2^{32})$). This result can be given by a straightforward implementation (see Code 4.5), which finds the longest distance between two consecutive prime numbers for domain ($[0, 2^{32}]$) with the **GMP** library.

```
1  void count()
2  {
3    for (; current < std::numeric_limits<int>::max();)
4    {
5      mpz_nextprime(current.get_mpz_t(), current.get_mpz_t());
6      liao::Math::GmpBigNumber temp = current - last;
7      if (distance < temp )
8      distance = current - last;
9      last = current;
10   }
11 }
```

Code 4.5: Find maximum distance of prime number in chosen domain

Furthermore, the enlarge index could be **replaced by a password** instead of 2^n. Thus, `primize()` can achieve that the data are encrypted when it is inserted into the database.

4.6.2 Primize Function

The `primize()` function takes a 4 bytes memory until the end of memory or the size left less than 4 bytes (see Code A.19). The function fills a 4 bytes memory with given data, then enlarges it with pre-defined index (512 for current version).

After that, `primize()` calls `mpz_nextprime()` to find the next prime number greater than the enlarged value [19]. Figure 4.14 shows an example of how a given data item is converted into a prime-based hash key.

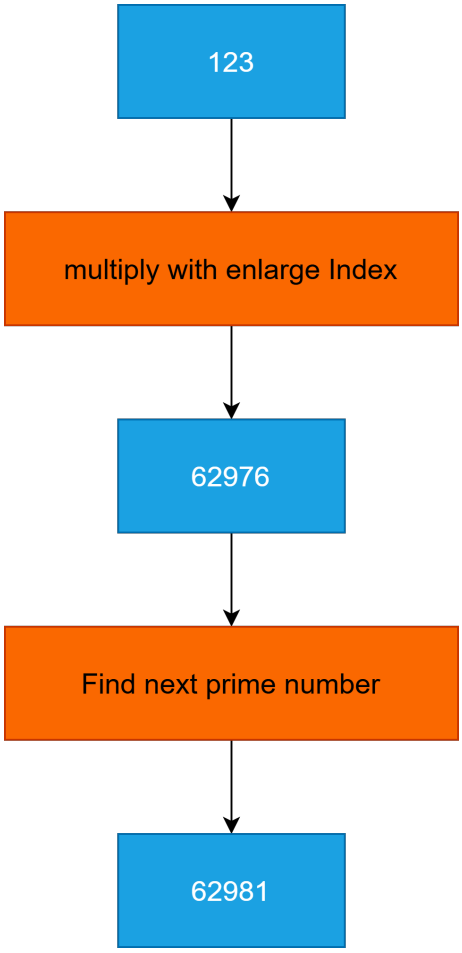

Figure 4.14: The picture illustrated how input data converted into prime number hash key

After commitment, the converted memory is written into a block (16kb). Then the block writes itself into a data file related to the table. The record number is stored

at the end of the record's memory layout, which is crucial for query processing.

The function requires three arguments:

- a pointer to the original data buffer,
- the size (in bytes) of the original value,
- the target size (in bytes) of the hash key.

Furthermore, both the encoded data and the record key each require $2n$ bytes of storage, where n is the size (in bytes) of the original data. Thus, **PrimedDB** stores a total of $4n$ bytes per record.

4.6.3 Deprimize Function

The `deprimize()` function converts only the data part of the original data but not the record key. Most importantly, the original data can be **easily recovered by dividing the prime-hash key by the enlarge index**. Code A.20 shows the implementation of the `deprimize()` function.

The function reads 8 bytes from the record buffer and divides this value by the enlarge index to recover the original data. For example, the prime hash key of 123 is 62981. The original value can be recovered by dividing the hash key by 512 (the enlarge index).

4.7 Working with SQL

The SQL compiler is mandatory for all types of database. For example, Postgres SQL, MySQL and Oracle are having SQL dialects for itself. Thus, **PrimedDB** also constructed a basic SQL compiler.

4.7.1 Compiler class

The `Compiler` (Code A.2) is a **singleton** (Code A.16) factory **class** which provides only one public interface `compile`.

The `compile()` function does not return a result immediately; instead, it returns a *future*, allowing the calling thread to perform other tasks without being blocked by the compilation process. The Code A.21 illustrates the implementation of the

`compile()` function, which takes the input SQL statements through parsing, syntax checking, and semantic analysis to ultimately produce an `ExecutionBody` (declared in Code A.1) then converts the `ExecutionBody` instance into a transaction and pushes the transaction into the pending list.

The server reports an error in the console interface (illustrated in Figure 4.15) and sends an error message to the client when the compile task failed due to syntax or semantic error. The server sends the query results, or a success notification, back to the client if the SQL statement is valid and executable.

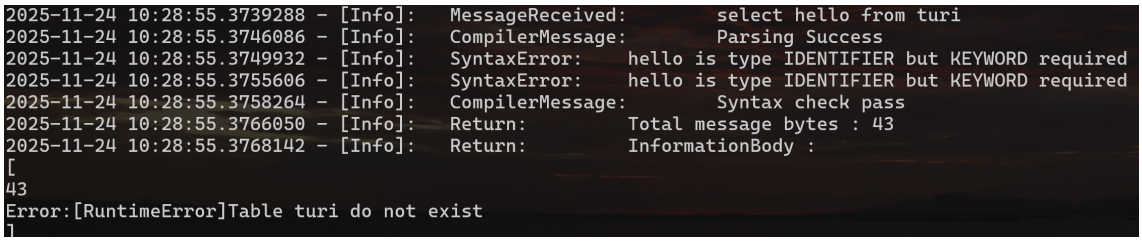

Figure 4.15: The error messages show in the console interface of PrimedDB server

4.7.2 Parsing SQL

The **PrimedDB** compiler parses the received SQL statement into a *token list* during the parsing stage (defined in Code A.22). There are 9 types of token:

- keywords (select, where, from, insert, into, etc.)
- identifiers
- operator (+,-,*,/,=,==,<=,>=)
- logical operator (and, or, not)
- left/right bracket ((,))
- delimiter (,)
- data type (INT, VARCHAR)
- number
- string literal.

However, the demo version of **PrimedDB** supports only a limited set of SQL keywords (e.g., `SELECT`, `INSERT`, and `DELETE`). On the other hand, the token type (declared in Code A.17) of **PrimedDB** is a structure which contains only two members, token type and command.

4.7.3 Syntax checking

Following parsing, the compiler validates the syntax of the token list (defined in Code A.23). First, the command is rejected if it is empty or does not begin with a SQL keyword. Second, the compiler verifies that the rule set contains a rule that begins with the given keyword. Additionally, the **PrimedDB** compiler discards all unsupported keywords during syntax checking. Lastly, the compiler iterates through the rule set until it finds a rule that matches the current keyword. If a match is found, the token list is accepted; otherwise, it is rejected.

Additionally, there are some combination rules in the rule set such as *expression, repetition, identifier list, value (the clause of values in insert keyword) and assign*. **PrimedDB** can recognize certain complex tokens due to the presence of subexpressions.

After the compiler accepts a command, the compiler generates an instance of `ExecutionBody` (declared in Code A.1) that can generate a transaction or an error message after the semantic check and code execution stage.

4.7.4 Runtime semantic check

The executor checks both the existence of the target table and its type information at runtime. Meanwhile, the type consistency between the input data and the corresponding column types is verified.

4.7.5 Code execution

After semantic checks, the instance of the `ExecutionBody` is classified into categories based on query type: DML, DDL, and DQL. For example, DQL statements do not generate transactions, whereas DML statements—such as `INSERT`—create a new transaction and add it to the pending transaction list.(implemented in Code A.24).

4.8 Storage management Unit

The storage unit is a critical component of any database system, as it directly influences data consistency, durability, and I/O efficiency.

4.8.1 Load settings

The `Setting` (Code A.13) is a **singleton** class which stores all setting attributes. Those attributes can be changed in the future version through SQL statements or administrator commands. The storage management unit uses some attributes to set up the environment, such as block size and maximum count of blocks.

4.8.2 Schema management

All definitions of the table are stored in *global/table.def*, which contains all schema information (see Code 4.6). There is a **singleton** class `TableManager` (declared in Code A.18) to manage all tables defined in this file, which load all tables from the file at the first user attempt to log into **PrimedDB**.

```
1
2 name: bit vector: record count: owner id:byte: primed byte: record
    number byte: column information(including name, type, byte)
3
4 jeff:111:8:3:54B53072540EEEB8F8E9343E71F28176:18:36:36:id|1|4|4:
    name|2|10|10:gender|1|4|4
5 bi::8:0:54B53072540EEEB8F8E9343E71F28176:18:36:36:id|1|4|4:name
    |2|10|10:gender|1|4|4
```

Code 4.6: How table schema stored on the disk

Additionally, the schema file is modified for certain DML operations—such as INSERT and DELETE—because these commands alter the bit vector stored within the schema file. After reading the schema information, **PrimedDB** assigns a lock to ensure atomicity and consistency of the definition file, which allows multiple reads but unique write at the same time.

The actual data are stored in the binary data file correlated to a specific table under path `/data/`. Each table is persisted in a data file whose name matches the table name; for example, the table `jeff` is stored in `/data/jeff.dat`.

Every instance of `Table`(declared in Code A.9) managed by `TableManager` stores the block IDs assigned to it, which are used to access the corresponding block. Consequently, the affected block is modified in accordance with the DML operation applied to the current table. Furthermore, the table also submits a transaction to the transaction processing unit because it can check the validity of given data and provide converted size to the block easily as the table contains all byte information of columns.

4.8.3 User management

Like tables, all users of **PrimedDB** are managed by `UserManager` (declared in Code A.12) which stores all user information in *`global/user.def`* (see Code 4.7). Also, the `UserManager` loads all users into memory on the first user attempt to log into the system. Furthermore, the `UserManager` supports adding new users to the system and removing users that already exist. However, the **PrimedDB** compiler cannot support related commands yet, but in the future.

```
1 system:54B53072540EEEB8F8E9343E71F28176:
    FCDBAA5CA2824053533B974FCFE5B6EEC891C68B637A7953BA7AD88987239A9D
    :8|
2 liao:65722F5A10FE90F70BF7B64E9B23A1AC:
    C5D77DEF77846F663EE2BEC3C1D3E5F675C3C46815EDAB6F540B52FF829A7BDB
    :7|
3 user name: user id: password: authorized level
```

Code 4.7: The definition file of user information

The `UserManager` provides validation functionality for **PrimedDB**, which validates the identity of all supplied information and subsequently grants an accept or reject verdict (defined in Code A.25).

4.8.4 Block management and transaction handling

In **PrimedDB**, transaction handling is integrated into `BlockManager` which allocates a new block to the table and writes given data to specified position in the block.

Block management

The `BlockManager` is a memory pool, which manages 256 blocks and each block is 16KB (the size of pool and block can be changed through command in the future). The manager retrieves a new block from the active block queue if the table requires one, either due to block depletion or initial creation (defined in Code A.26). On the other hand, the active queue moves recently used blocks to the back of the queue and pops the front block for allocation.

The data stored in **PrimedDB** are not immediately removed from the disk and memory when a `DELETE` command is executed. The table calls Code A.27 to remove all unavailable records when the table detects that the usage of all its owned blocks is below a specified threshold (percentage). The rearrangement function releases all unused blocks back to the pool and refreshes the table's data file.

Transaction handling

The function `manager()` (defined in Code A.28) waits for all transactions submitted from other threads.

After a transaction is committed, the function `manager()` calls the function `handle()` (defined in Code A.29) to process all pending changes submitted by users. The `handle()` function notifies the corresponding block when the transaction is processed and modifies it accordingly.

4.9 Report and manage errors

Properly handling errors in a multithreaded system is a complex task, especially to propagate an exception through different threads in C++. Thus, **PrimedDB** implements an `ErrorManager` (declared in Code A.15) that allows any thread to set an error and register a corresponding handler.

The `set()` function (implemented in Code A.30) can submit an error by specifying its type, name, message, and optional contextual information. On the other hand, an error handler could subscribe (defined in Code A.31) for a specific error name or type with a `functional<T>` object (a callable instance).

The error handler handles an error with pre-registered function when a subscribed error is published. Furthermore, an error can be registered by multiple dif-

ferent handlers. Three predefined functions (implemented in Code A.32) log error messages and, if needed, write error messages to a file.

The code information is an instance of `ClassInfor` (declared in Code A.3), which carries the name of the function and the line number of the file where the error was raised. The instance analyzes the compiler macros `__LINE__`, `__FILE__`, and `__FUNCSIG__` to extract contextual details, including the class name, function name, source line number, and function parameter list.

4.10 Code testing

PrimedDB relies primarily on system testing because multithread architecture makes unit testing impractical for many components.

All tests were conducted using a system with an **Intel Core Ultra 9 185H processor, 64 GB of RAM, and Windows 11 x64 Education version 25H2**.

4.10.1 Component test

PrimedDB tests all classes' non-concurrent functions with assertion test which checks that the applied modifications have indeed taken effect. Code 4.8 shows an example of a test subroutine that exercises self-contained operations that components are not depend on other multithreading components—and verifies that the expected changes have been applied.

```
void testTable()
{
  string columnName = name;
  Column column(columnName, 4, hello, DataType::Int);
  assert(column.getName(), name);
  assert(column.getOwner(), hello);
  assert(column.size(), 4);
  assert(column.getType(), DataType::Int);
  assert(column.primed(), column.size() * 2);
  column.resize(10);
  assert(column.size(), 10);
  column.setType(DataType::Null);
    assert(column.getType(), DataType::Null);
  deque<Column> columns = { column };
  string tableName = hello;
```

```
16   Table table(system, tableName,UserLevel::Administrator,columns);
17   assert(table.getAvailable().size(), 0);
18   assert(table.getOwner(), system);
19   assert(table.getName(), hello);
20     assert(table.getPermission(), UserLevel::Administrator);
21     assert(table.columnSize(), 1);
22   assert(table.existColumn(hello), true);
23     assert(table.existColumn(world), false);
24   assert(table.isEmpty(), true);
25 }
```

Code 4.8: A example testing function which tested Column and Table class

`UserManager` and `TableManager` are also tested with component tests. This test creates a new user or table, performs the function `exists()`, the function `(get())` and modifies data (using functions already validated in the component tests for `Table` and `User`), and finally deletes the selected user.

Nevertheless, an additional test is conducted in a concurrent environment for these `Manager` classes, as they are designed to be used concurrently. Code 4.9 shows an example test of the `UserManager` class for its `create()` function, which executes the operation in multiple threads and verifies the result after completion.

```
1  void testUserCreate()
2  {
3    string hello = hello, name = name;
4    string world = world, password = password;
5    thread create1([&]()
6      {
7        UserManager::Get().create(hello, world, UserLevel::
             Administrator);
8      });
9    thread create2([&]()
10   {
11       UserManager::Get().create(name, password, UserLevel::
             Administrator);
12   });
13   assert(UserManager::Get().userCount(), 2);
14 }
```

Code 4.9: A example testing function which tests the creation of a new user in concurrent environment

Data loss, race conditions, or even deadlocks occur when verifying the results after the operation is completed if the function under test does not properly acquire and release locks for its operations. Similar tests are also applied to the remaining functions of this class and to other `Manager` classes that are frequently used in multithreaded scenarios.

On the other hand, the client application provides debug mode (see Figure 4.16) for testing.

```cpp
#include "mainwindow.h"
#include "Login.h"
#include <QtWidgets/QApplication>

int main(int argc, char* argv[])
{
    QApplication a(argc, argv);

    // Check for debug mode (e.g., command line argument)
    bool debugMode = false;
    for (int i = 1; i < argc; ++i) {
        if (QString(argv[i]) == "--debug" || QString(argv[i]) == "-d") {
            debugMode = true;
            break;
        }
    }


    // Create login window
    Login login;
    // Set debug mode if enabled
    login.setDebugMode(debugMode);
    std::vector<std::string> tables;
    if (debugMode)
    {
```

Figure 4.16: The code definition of client main function which allows debug mode for components testing

In debug mode, the system bypasses authentication, allowing access with any username and password (see Figure 4.17), and verifies that the core components function correctly. The test verifies the response to all button interactions and provides sample input for key rendering areas, such as the schema structure view and the table view.

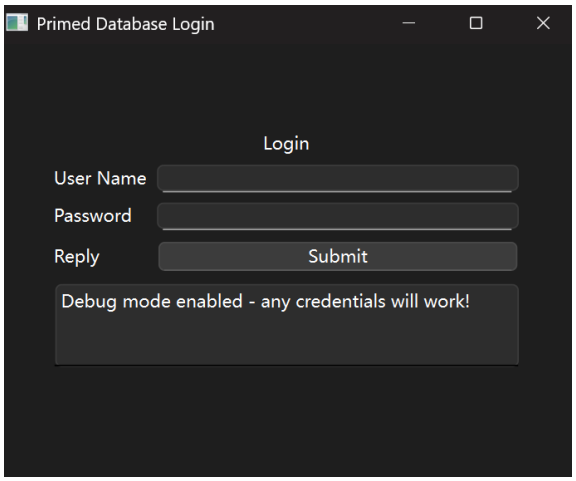

Figure 4.17: The client login interface when run with debug mode

4.10.2 System testing

The testing for multi-thread service in **PrimedDB** is using console logging information to verify the correction of operation. For example, a message indicating that the transaction is being processed must appear when the user commits all operations to the transaction handler (shown in Figure 4.18).

```
2025-11-24 18:01:21.0846664 - [Info]:   MessageReceived:        INSERT INTO YURI (id,name) VALUES (1234,'world')
2025-11-24 18:01:21.0861364 - [Info]:   CompilerMessage:        Parsing Success
2025-11-24 18:01:21.0867886 - [Info]:   CompilerMessage:        Syntax check pass
2025-11-24 18:01:21.0876682 - [Info]:   primize:        From 1234 to 631817
2025-11-24 18:01:21.0881850 - [Info]:   primize:        From 1819438967 to 931552751117
2025-11-24 18:01:21.0887680 - [Info]:   primize:        From 100 to 51203
2025-11-24 18:01:21.0892755 - [Info]:   primize:        From 0 to 257
2025-11-24 18:01:21.0897227 - [Info]:   yuri:   Received request from system to write data into block
2025-11-24 18:01:21.0906503 - [Info]:   Return:         Total message bytes : 14
2025-11-24 18:01:21.0910116 - [Info]:   Return:         InformationBody :
[
14
Insert success
]
2025-11-24 18:01:34.6975219 - [Info]:   MessageReceived:        commit
2025-11-24 18:01:34.6981694 - [Info]:   Transaction:    Handling Transaction [1D8F2AE513C24AAC00D1749D6DBBF9F5]
2025-11-24 18:01:34.6987234 - [Info]:   BlockInsert:    Write all data into block memory position 3 success
2025-11-24 18:01:34.6990195 - [Info]:   BlockInsert:    The occupied size of block is: 3
2025-11-24 18:01:34.6983062 - [Info]:   CompilerMessage:        Commit executed
2025-11-24 18:01:34.7002945 - [Info]:   Return:         Total message bytes : 14
2025-11-24 18:01:34.7006263 - [Info]:   Return:         InformationBody :
[
14
Commit success
]
2025-11-24 18:01:34.7016212 - [Info]:   TableManger:    Updating schema file
2025-11-24 18:01:34.7024852 - [Info]:   TableManger:    Updating complete
```

Figure 4.18: The correct message of insertion a new data

On the other hand, the client application sends sample input to the server application to test that network components are working properly for both the server and the client. For example, an error occurs if the client application fails to connect to the server application, even when the server socket is running correctly.

Moreover, the compiler rejects invalid inputs (see Figure 4.19 and Figure 4.20 demonstrating that the syntax-checking component is working correctly), such as empty strings and SQL statements with misspelled keywords.

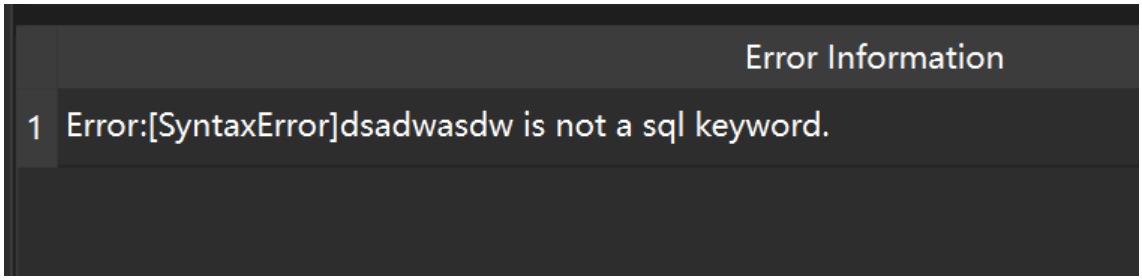

Figure 4.19: The error message indicate given SQL is an invalid statement

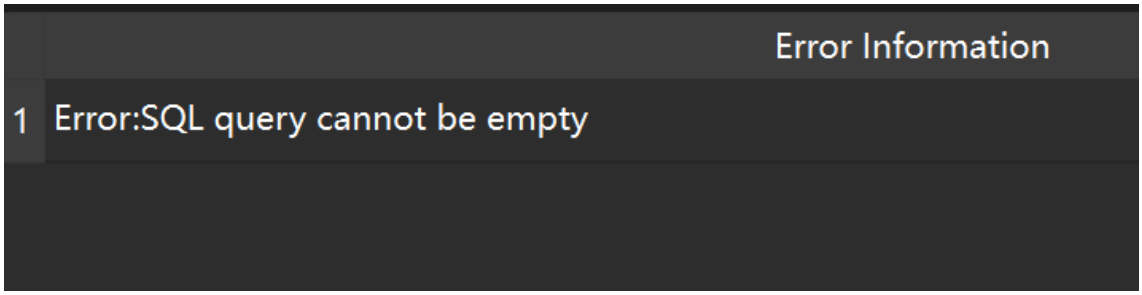

Figure 4.20: Compiler rejected empty SQL statement

Meanwhile, the log messages are asynchronously printed by `ErrorManager` with pre-defined handler functions (see section 4.9). Thus, Figure 4.21 demonstrates that the `ErrorManager` and its associated handler functions are operating correctly.

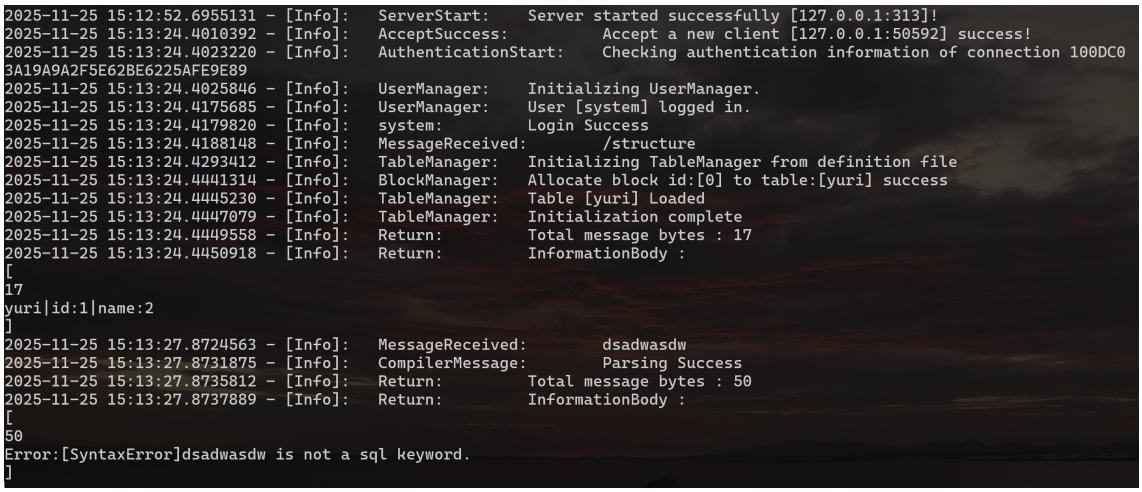

Figure 4.21: The output provided by ErrorManager

4.11 Future updates

This version is a beginning for **PrimedDB**. **PrimedDB** will strive to secure a foothold in performance-constrained or high-security scenarios in the future.

Firstly, **PrimedDB** will fully support all critical SQL statements in the upcoming versions, including JOIN, DROP, and indexing. Meanwhile, the current compiler will be enhanced with a better syntax analyzer and a compile time semantic checker. On the other hand, all operations related to user authorization will be accessed by SQL statements.

Secondly, the database will implement a new transaction system that supports *undo* and *redo* operations in **PrimedDB**.

Lastly, the administrator user can terminate the session of suspicious users with the server side command or SQL command in the future version.

In the future versions the client will reconnect to the server or re-login to the server in case is kicked by administrator. Also, a better user interface will be implemented which can load existing SQL files and provide code completion feature.

Conclusion

In conclusion, **PrimedDB** gives a new solution to optimize the relational database instead of a new data storage structure because relational databases have proven their validity and importance through 55 years of use and development [20].

The idea is to give a new hash function (shown in subsection 4.6.2) that generates a hash key in prime number format where the key itself is also the data. Meanwhile, all prime hash keys in the same record can be combined into a record key that uses the *Fundamental Theorem of Arithmetic* to determine whether a specific column or a set of columns is present in the current record or not (shown in subsection 3.1.3). Thus, **PrimedDB** demonstrates that using prime number hash keys for data storage and querying is feasible.

However, **PrimedDB** covers the majority of the key topics outlined in the ELTE computer science curriculum, including compilers, operating systems, concurrent programming, and network communication—making it an excellent practical application, despite some imperfections in certain components.

In the future, imperfections of **PrimedDB** will be fixed, its current functionalities will be extended, and it will be used in some scenarios that require strong security and performance. The features introduced by **PrimedDB**, including insertion and encryption, are particularly well-suited for certain scenarios (see section 4.6).

Overall, the concepts introduced by **PrimedDB** are promising; even though the current version is not yet perfect, it represents an excellent starting point.

Acknowledgments

The acknowledgment is mandatory thesis requirements for graduation at Chinese universities. Thus, I would like to express my gratitude to the following people who have helped, inspired, and motivated me.

Firstly, my parents provided me with sufficient financial and emotional support that makes me turn my ideas into reality.

Secondly, Dr. Zsók Viktória accepted my idea and allowed me to implement the database freely, then thoroughly and professionally reviewed my paper and provided expert feedback.

Third, Iurii Bulanov helped me prove that the prime hash key can get back to the original data by dividing the enlarge index.

Lastly, I express my gratitude to all the people I met in ELTE, who made my university life colorful and fun.

In the future, I will explore the edges of technology and bring more positive changes to human society.

Appendix A

Technical Details

A.1 Class declarations

```
enum class DataType :char
{
  Null,
  Int,
  Varchar,
  Number,
  RecordNum,
  Table,
  Column,
  User,
  Session,
  DataBase,
  String,
  Type,
  Operator,
  All,
  Not,
};
struct Entity
{
  PrimedDB::DataType m_type;
    std::string content;
public :
  Entity() = default;
  Entity(PrimedDB::DataType, std::string&);
  Entity(const Entity& copy);
```

```
27   void operator=(Entity&);
28   Entity(Entity&&) noexcept;
29
30   void operator=(Entity&&)noexcept;
31 };
32 class ExecuteBody
33 {
34   PrimedDB::UserPtr m_user;
35   PrimedDB::SQLType m_task;
36   std::unordered_map<unsigned, std::vector<Entity>> m_operations;
37   std::string m_message;
38   bool m_valid = false;
39   static PrimedDB::SQLType getTask(const std::string&);
40   void selectConstructor(std::deque<Token>&);
41   void insertConstructor(std::deque<Token>&);
42   void deleteConstructor(std::deque<Token>&);
43   void updateConstructor(std::deque<Token>&);
44   void createConstructor(std::deque<Token>&);
45   void analyzer(std::deque<Token>&);
46 public:
47     ExecuteBody()=default;
48   ExecuteBody(const std::string&);
49 ExecuteBody(std::deque<Token>&);
50   void construct(std::deque<Token>&);
51   bool isValid()const;
52   void setValid(bool);
53     const std::string& getMessage()const;
54   void setMessage(const std::string&);
55   std::unordered_map<unsigned, std::vector<Entity>>& get();
56   PrimedDB::SQLType getTask();
57 };
```

Code A.1: The declaration of ExecutionBody

```
class Compiler : public Singleton<Compiler>
{
  INVITESINGLETON
private:
  static TokenType getTokenType(std::string& token);
  static bool isKeyword(std::string& token);
  static bool isDataType(std::string& token);
  static bool isIdentifier(std::string& token);
  static bool isNumber(std::string& token);
  static bool isStringLiteral(std::string& token);

  static int syntaxChecker(const std::vector<TokenType>&, std::
     deque<Token>& tokens, unsigned& start);
  static std::deque<Token> parseSQL(const std::string& sql, std::
     string& error);

  //Special token type checker


  //check rule like:
  // (column name - byte size,
  // column name - byte size,
  // .....)
  static bool columnsCheck(std::deque<Token>& tokens, unsigned&
     start);
  //check expressions like:
  // identifier = value
    // identifier = 'literal'
  static bool expressionCheck(std::deque<Token>& tokens,unsigned&
     start);
  // check identifier identifier identifier ...
  static bool identifiersCheck(std::deque<Token>& tokens,unsigned&
     start);
  // check list like:
  //(value, value,...)
  static bool valueCheck( std::deque<Token>& tokens, unsigned&
     start);
  //check repetition like value, value,(the last token can be
     ignored)
  static int repeatCheck(const std::vector<TokenType>& types,
```

```
        unsigned& typeStart,std::deque<Token>& tokens, unsigned&
        tokenIter);
34    //convert token type to string
35    static std::string type2string(TokenType);
36
37    static ExecuteBody ruleMatch(const std::vector<TokenType>&,std::
        deque<Token>&);
38    static ExecuteBody syntaxFailed(std::string&, const std::string&)
        ;
39    static ExecuteBody syntaxCheck(PrimedDB::UserPtr,std::deque<Token
        >& tokens,std::string&);
40
41
42    static PrimedDB::Record execute(PrimedDB::UserPtr, ExecuteBody&,
        std::string&);
43
44
45    static void toLower(std::string& str);
46    static std::string trim(const std::string& str);
47
48    static std::unordered_map<std::string, std::vector<std::vector<
        TokenType>>> m_rules;
49    void constructRules();
50    Compiler();
51  public:
52
53    // compile function
54    std::future<Result> compile(PrimedDB::UserPtr user, const std::
        string& sql);
55
56  };
```

Code A.2: The declaration of Compiler

```
class ClassInfor
{
  string complete;
  //Dereference function name information from compiler Macro
  string raw2FunctionName(const string& functionSig);

  //Dereference class name information from compiler Macro
  string raw2ClassName(const string& functionSig);

  //Dereference parameter list information from compiler Macro
  string raw2ParameterList(const string& functionSig);

  //Get function return type from compiler macro
  string raw2ReturnType(const string& functionSig);

  //return compelete class, function information for the line
     constructed this instance
  string combineToComplete() const;
public:
  /*( compiler macro which carry function and class information,
    compiler macro which indicatte file directory information *
       OPTIONAL*,
    compiler macro which indicate the line which construct current
       instance *OPTIONAL* )*/
  ClassInfor()=default;
  ClassInfor(const ClassInfor&) = default;
  ClassInfor(const std::string& functionsig, const std::string&
     fileDirectory = "", long long lineNumber = -1);
  ClassInfor(ClassInfor&&)noexcept ;
  void init(const std::string& functionsig, const std::string&
     fileDirectory = "", long long lineNumber = -1);
  ClassInfor& operator=(ClassInfor&);
  //The complete information at the line which construct current
     instance
  const std::string& CompleteInfor();

  //The name of function call at the line which construct current
     instance
  std::string FunctionName;

```

```
34   //The class information of current method call at the line which
        construct current instance
35   std::string ClassName;
36
37   //The parameter list of function call at the line which construct
         current instance
38   std::string ParameterList;
39
40   //The return type of function call at the line which construct
        current instance
41   std::string ReturnType;
42
43   //The line number of the line which construct current instance
44   long long LineNumber;
45
46   //The file path of current source file.
47   std::string FileDirectory;
48
49   operator string();
50
51   //Check the string pos is -infinity or not
52   static bool SubStrNotFound(size_t pos);
53 };
```

Code A.3: The declaration of ClassInfor

```
class Server
{
    using acceptor = asio::ip::tcp::acceptor;
    using Socket = asio::ip::tcp::socket;
    using tcp = asio::ip::tcp;
    using MessageHandler = std::function<void(const std::string&,
       std::shared_ptr<Socket>)>;
    using ConnectionHandler = std::function<void(std::shared_ptr<
       Socket>)>;
    using ErrorHandler = std::function<void(const std::error_code&,
        std::shared_ptr<Socket>)>;
    using UserPtr = PrimedDB::UserPtr;
    using SocketPtr = std::shared_ptr<Socket>;
    acceptor m_acceptor;
    std::unordered_map<UserPtr, SocketPtr> m_user_clients;
std::unordered_map<std::string, std::pair<SocketPtr, std::
   shared_ptr<asio::streambuf>>> m_unauthorized;
    std::atomic<bool> m_stop;
    bool m_working;
    asio::io_context& m_io;
std::shared_ptr<asio::steady_timer> m_timer;
    std::mutex m_mutex;
    std::shared_mutex m_hashMutex;
    void doAccept();
    void doRead(SocketPtr socket);
    void handle_read(SocketPtr socket,
        std::shared_ptr<asio::streambuf> buffer,
        const std::error_code& error);
    auto getip(const std::string&);
    void stream2string(std::shared_ptr<asio::streambuf> buffer, std
       ::string& message);
    void exitAct(SocketPtr);
    bool validateUser(SocketPtr socket,const std::string& userName,
       const std::string& password, std::string& reply);
    void authenticate(const std::string& id);

    void sendStructure(std::shared_ptr<asio::ip::tcp::socket>
       socket);
    void writeMessage(std::shared_ptr<asio::ip::tcp::socket> socket
       ,std::string& message);
```

```
33 public:
34     Server(asio::io_context& io,const std::string& ip, unsigned
           short port);
35     ~Server();
36 
37     //Start server on given ip address
38     void start();
39 
40     //Terminate current server
41     void terminate();
42 
43     void setStop(bool value);
44 
45     bool isStop();
46 
47     UserPtr getUser(SocketPtr);
48 
49     void remove(SocketPtr);
50 
51     //Send message to all clients
52     void broadcast(const std::string& message);
53 
54     //Send message to specific clients or send message to all
           clients except specific clients(set except to true)
55     void broadcastTo(std::vector<UserPtr>&, const std::string&,
           bool except = false);
56 
57     //Send message to a specific client
58     void sendToClient(const UserPtr&, const std::string&);
59 
60     //Check if a client exists
61     bool exist(const UserPtr&);
62 
63     //Stop a client(Executor, User)
64     bool stop(const UserPtr&, const UserPtr&);
65     int size();
66 };
```

Code A.4: The declaration of Server

```
class Block
{
  std::string m_id;
  TablePtr m_owner;
  //memory
  char* m_memory = nullptr;
  //size of how many record in memory
  //the position of the first record in table
  unsigned m_start;
  int m_count;
  unsigned m_pointer = 0;
  mutable ShareMutex m_mutex;
  std::vector<char*> m_records;
  void allocate(std::shared_ptr<char> source = nullptr, unsigned
     size = 0);
  void deallocate();
public:
  Block();
  Block(Block&)=delete;
  Block(Block&&) noexcept;
  void assign(TablePtr, unsigned);
  //check two Blocks are same object or not
  bool same(const Block&) const;
  //compare content in memory
  bool equal(const Block&);
  unsigned max() const;
  void writeLine(UCharPtr, unsigned pos);
  void flush();
  void update(unsigned location,std::shared_ptr<char[]> memory);
  bool empty();
  unsigned int size();
  unsigned byte() const;
  TablePtr getOwner();
  //build the link of records
  void build();
  //get the memory space of entire block
  char* reference();
  void remove(unsigned index);
  std::pair<unsigned, std::shared_ptr<char[]>> insert(std::
     shared_ptr<char[]> memory,unsigned number);
```

```
39      char* get_noLock(unsigned index);
40      void drop(unsigned index);
41      double percentage();
42      ShareMutex& getMutex();
43      std::vector<char*>& getRecords();
44      //==same
45      bool operator==(const Block&)  const;
46      Block& operator=(Block&&) noexcept;
47      ~Block();
48  };
```

Code A.5: The declaration of Block

```
class BlockManager:public Singleton<BlockManager>
{
  INVITESINGLETON;

  std::deque<unsigned> m_active;

  //store all blocks, which will only enlarge the size.
  std::vector<Block> m_blocks;
  std::future<void> m_terminate;
  std::atomic<bool> m_notify = false, m_end = false;
  std::condition_variable m_cv;
  std::deque<Transaction> m_pendingOperations;
  mutable ShareMutex m_mutex;
  mutable Mutex m_cvMutex;

  std::promise<bool> m_finished;
  BlockManager();

  unsigned allocate();
  void manager();
  void rearrange(std::deque<int>&, std::vector<bool>&);
  void handle(Transaction&);
  void promote(unsigned pos);
public:
  static unsigned totalRecord(unsigned bytes);
  void operate(Transaction&);
  //table pointer and line number indicate the start of the block
  unsigned allocate(TablePtr, int pos = -1);
  unsigned allocateWithOutRead(TablePtr, int pos = -1);
  void drop(unsigned pos);
  Block& get_noLock(unsigned);
  ShareMutex& getMutex();

  std::future<bool> wait();

  ~BlockManager();
};
```

Code A.6: The declaration of BlockManager

```
class Column
{
  unsigned int m_byteSize;
  std::string m_columnName;
  unsigned int m_primed=0;
  std::string m_owner;
  DataType m_type;
  mutable std::shared_mutex m_mutex;

  void constructFromFileLine(const std::string& line);
public:
  Column(std::string& name, unsigned size, const std::string& owner
      , DataType);
  Column(const std::string&, const std::string&);
  Column(const Column&);
  Column(Column&&) noexcept;
  void rename(std::string&);
  void resize(unsigned);
  void setType(DataType);
  DataType getType() const;
  const std::string& getName() const;
    unsigned size() const;
  unsigned primed() const;
  const std::string& getOwner() const;
  const std::string& getId() const;
  std::string toString() const;
  bool operator<(const Column&);
  bool operator<(const Column&) const ;
  void operator=(const Column&&);
    ~Column() = default;
};
```

Code A.7: The declaration of Column

```
1  class Record
2  {
3    using RecordData = std::deque<std::deque<std::shared_ptr<std::
       string>>>;
4    using ColumnView = std::deque<std::shared_ptr<std::string >>;
5    using RowLine = std::deque<std::shared_ptr<std::string >>;
6    std::unordered_map<std::string, ColumnView> m_header;
7    std::deque<std::string> m_title;
8    std::vector<bool> m_available;
9    RecordData m_values;
10   unsigned m_byte;
11   bool m_valid = false;
12   std::vector<bool> m_include;
13   void constructColumnView(std::unordered_map<std::string, int>&)
       ;
14 public:
15   Record() = default;
16   Record(bool);
17   Record(std::unordered_map<std::string,int>&, RecordData&);
18   Record(Record&&)noexcept;
19   void set(std::unordered_map<std::string, int>&, RecordData&);
20   void include(std::string&);
21   Record& all();
22   Record& where(std::vector<bool>& drop);
23   //void where(std::string&, std::function<bool>);
24   ColumnView& at(std::string columnName);
25   RowLine& at(unsigned index);
26   bool isValid() const;
27   std::string format() const;
28   void operator=(Record&&)noexcept;
29 };
```

Code A.8: The declaration of Record

```
class Table : public std::enable_shared_from_this<Table>
{
  std::string m_name;
  std::vector<bool> m_available;
  std::deque<int> m_owned;
  UserLevel m_permission;
  unsigned m_size = 0;
  std::string m_ownerId;
  std::vector<Column> m_columns;
  mutable std::shared_mutex m_mutex;
  //byte size of record before primize
  unsigned m_byteSize;
  //byte size of record after primize
  unsigned m_primedSize;
  //the byte size of record number after all data
  unsigned m_recordByte;

  void construct(std::vector<std::string>&, const std::string&
     fileLine);
  std::pair<int, int> convertBlockPos(unsigned position);
  void intoString(int index, std::deque<std::deque<std::shared_ptr<
     std::string>>>&, std::vector<char*>&,bool);

public:
  Table(const Table&) = delete;
  void operator=(const Table&) = delete;
  Table(const std::string& fileLine);
  Table(const std::string& fileLine, std::vector<std::string>&);
  Table& operator=(Table&&) noexcept;
  Table(Table&&) noexcept;
  Table(std::string ownerId,std::string& name, UserLevel permission
     , std::deque<Column>&);
  //get_noLock name of the table
  const std::string& getName() const;
  //convert a record memory into primed format
  static void primize(UCharPtr& memory, unsigned byteSize, unsigned
      primedSize, unsigned recordSize);
  //convert a record memory into normal format
  static void deprimize(UCharPtr& memory, unsigned byteSize,
     unsigned primedSize, unsigned recordSize);
```

```
36  //return the iterator of column
37  auto findColumn(const std::string& name);
38  auto findColumn(const std::string& name) const;
39  //rename a column
40  void renameColumn(const std::string& name, std::string& newName);
41  //check is given name belongs to a column in the table
42  bool existColumn(const std::string& name) const;
43  //Get the states of all records, 1 is active, 0 is inactive
44  std::vector<bool>& getAvailable();
45  std::deque<int>&  getOwned();
46  //return permission of the table
47  UserLevel getPermission() const;
48  Column& getColumn(const std::string& name);
49  //remove a record at given position
50  void setUnavailable(unsigned pos);
51  //recover a record at given position
52  void setAvailable(unsigned pos);
53  void clear();
54  bool read();
55  unsigned incrementSize();
56  unsigned decrementSize(unsigned pos);
57  unsigned byte()const;
58  unsigned primedByte()const;
59  bool addColumn(std::string& name, unsigned byteSize,DataType);
60  void dropColumn(const std::string& name);
61    bool rename(std::string& name);
62  unsigned columnSize() const;
63  unsigned byteSize()const;
64  unsigned totalByte()const;
65  unsigned recordByte()const;
66  size_t size() const;
67  const std::string& getOwnerId() const;
68  std::string toString();
69  bool isEmpty()const;
70  void insert(const std::string&,UCharPtr memory, unsigned size =
       1);
71  void update(const std::string&, unsigned position, UCharPtr
       memory, unsigned size = 1);
72  void remove(const std::string&, unsigned position);
73  void rollback(const std::string& name);
74  void addBlock(unsigned pos);
```

```
75   void dropBlock(unsigned pos);
76   void dropBlockAt(unsigned pos);
77   std::vector<bool> where(std::unordered_map<std::string,std::
         string>&);
78   std::vector<Column>& getColumns();
79   std::string format();
80   void setSize(unsigned newSize);
81   void setOwner(const std::string& );
82   Record select(bool raw = false);
83   Record select(std::deque<std::string>& list);
84   ShareMutex& getMutex();
85   unsigned blockSize()const;
86   ~Table();
87 };
```

Code A.9: The declaration of Table

```
class Transaction
{
  std::string m_id, m_table;
  SQLType m_operation;
  std::string m_operator;
  int m_blockId,m_location;
  bool m_valid;
  unsigned m_size;
  SCharPtr m_memory;
public:
  Transaction();
  Transaction(const std::string& oprtor,const std::string&, SQLType
      operation, int blockId, int location, unsigned size = 0,
     UCharPtr memory = nullptr);
  Transaction(const std::string& oprtor, const std::string& name,
     const std::string& str, bool neg = false);
  Transaction(const std::string& oprtor, const std::string& name,
     bool neg = false);
  Transaction(Transaction&&) noexcept;
  void operator=(Transaction&&) noexcept;
  SQLType getType() const;
  const std::string& getId() const;
  const std::string& getOperator()const;
  const std::string& getTable() const;
  int getBlockId() const;
  void setBlockId(int);
  int getLocation() const;
  unsigned size() const;
  bool isValid() const;
  SCharPtr getMemory();
  void setValid();
  void setInvalid();
  std::string toString() const;
  void fromString(const std::string& str);
  void negFromString(const std::string& str);
    ~Transaction();
};
```

Code A.10: The declaration of Transaction

```
class User
{
  mutable ShareMutex m_mutex;
  std::string m_id;
  std::string m_name;
  std::string m_password;
  UserLevel m_level;
  std::unordered_map<std::string, TablePtr> m_tables;
  std::deque<Transaction> m_pendingOperations;
private:
  User(const User&) = delete;
  void constructFromFile(const std::string&);
public:
  User(const std::string& fileLine);
  User(std::string& id, std::string& name, std::string& password,
      UserLevel level);
  User(std::string& name,std::string& password, UserLevel level);
  User(User&&) noexcept;
  ~User();
  const std::string& getName()const;
      const std::string& getPassword()const;
      UserLevel getLevel()const;
      const std::string& getId()const;
  //1 GREATER, 0 EQUAL, -1 LESS
  int compare(const User& other) const;
  bool qualified(UserLevel level) const;
  bool validate(const std::string& password) const;
  bool validateWithHash(const std::string& password) const;
  bool rename(std::string& name);

      const std::unordered_map<std::string,TablePtr>& getTables()
         const;
  TablePtr getTable(const std::string& name)const;
  unsigned tableCount() const;
  static std::string PassWordHash(const std::string& rawText);
  TablePtr createTable(std::string name,UserLevel permission, std
     ::deque<Column>&);
      void dropTable(const std::string& name);
  void renameTable(const std::string& name, std::string& newName)
     ;
```

```
37      void addTable(TablePtr table);
38      void changePassword(const std::string& rawText);
39      void setPassword(std::string& hash);
40      void setLevel(UserLevel level);
41      void submit(Transaction&& operation);
42      void commit();
43      std::string toString() const;
44      TablePtr operator[](const std::string& name);
45      User& operator=(User&);
46      bool operator==(const User&) const;
47      static User createUser(std::string name,std::string password,
            UserLevel level, std::string id = "");
48  };
```

Code A.11: The declaration of User

```
class UserManager:public Singleton<UserManager>
{
  INVITESINGLETON;
  std::unordered_map<std::string,UserPtr> m_allUsers;
  static UserPtr System;
  static ShareMutex SystemMutex;
  mutable ShareMutex m_mutex;
  //check given name and password
  bool allowLogin(const std::string& name,const std::string&
     password);
  //check given level is higher than Manager or not
  bool levelQualified(UserLevel level, const std::string& name);
  static bool InvalidName(std::string& name);
  static bool isSafePassword(std::string& password);
  //update entire user file
  void save();

  UserPtr constructSystem(std::string& fileLine);
  UserManager();
public:

  bool exist(const std::string& name) const;
  UserPtr create(std::string& name,std::string& password, UserLevel
      userLevel);
  bool remove(const User& executor,const std::string& who);
  bool login(const std::string& name,const std::string& password);
  bool logout(const std::string& name);
  int userCount() const;
  const std::unordered_map<std::string, UserPtr> all()const;
  UserPtr get(const std::string& name);
    const UserPtr get(const std::string& name) const;
  static User& GetSystemUser();
};
```

Code A.12: The declaration of UserManager

```
class Setting:public Singleton<Setting>
{
  INVITESINGLETON;
  using path = std::filesystem::path;
  const unsigned int m_enlargeSize = 512;//the enlarge number for
     Primize Algorithm
  unsigned int m_blockNumber = 256;//the number of blocks
  unsigned int m_blockSize = 16;//the size of a block
  const unsigned m_kbConvert = 1024;
  unsigned int m_sessionNumber = 10;//the number of sessions
  unsigned int m_port = 1010;
  const unsigned int m_nameSize = 50;
  const unsigned int m_enlargePower = 9;
  const Math::HashType m_userIDHashType = Math::HashType::MD5;

  path m_configurationFile = "conf.cfg";

  path m_dataDirectory = "data";
  path m_compilerDirectory = "compiler";
  path m_globalDirectory = "global";
  path m_logDirectory = "log";
  path m_tableFile = "table.def";
  path m_userFile = "users.def";
  path m_tokenFile = "tokens.file";




  mutable ShareMutex m_mutex;
  //check existence of all directories
  //check existence of critical files
  //if one of them is missing, terminate the system.
  Setting();
public:
  std::string m_systemPassword;
  unsigned int getBlockNumber()const;
  unsigned int getBlockSize()const;
  unsigned int getEnlargeSize()const;
  unsigned int getSessionNumber()const;
  unsigned int getPort()const;
```

```
40   unsigned int getNameSize() const;
41   unsigned getEnlargePower() const;
42   void save(const std::string& key, const std::string& value);
43   Math::HashType getUserIDHashType() const;
44   unsigned int setBlockSize(unsigned int blockSize);
45   unsigned int setBlockNumber(unsigned int blockNumber);
46   //unsigned int setEnlargeSize(unsigned int enlargeSize);
47   unsigned int setSessionNumber(unsigned int sessionNumber);
48   unsigned int setPort(unsigned int port);
49
50   const std::string& getSystemPassword() const;
51
52   const path& getDataDirectory() const;
53   const path& getCompilerDirectory() const;
54   const path& getGlobalDirectory() const;
55   const path& getLogDirectory() const;
56   const path& getUserFile()const;
57   const path& getTableFile()const;
58 };
```

Code A.13: The declaration of Setting

```
struct Error
{
    ErrorLevel m_level;
    std::string m_name;
    std::string m_message;
    TimeStamp m_errorTime;
    Infor::ClassInfor m_info;
    Error() = default;
    Error(const Error&) = default;
    Error(Error&&) noexcept;
    Error(ErrorLevel, std::string& , std::string&, const Infor::
        ClassInfor&);
    Error(ErrorLevel, std::string_view, std::string_view,const
        Infor::ClassInfor&);
    bool operator==(Error& error);
    Error& operator=(Error&& error);
};
```

Code A.14: The declaration of Error

```
class ErrorManager final:public Singleton<ErrorManager>
{
    friend class Singleton;
    using ErrorHandler = std::function<void(Error&)>;
    std::deque<Error> m_errors;
    std::condition_variable m_conditionVar;
    std::unordered_map <std::string, std::vector<ErrorHandler>>
        m_serviceByName;
    std::unordered_map <ErrorLevel, std::vector<ErrorHandler>>
        m_serviceByLevel;
    std::atomic<bool> end = false;
    mutable ShareMutex m_mutex;
    mutable ShareMutex m_handleMutex;
    mutable std::mutex m_cvmutex;
    std::atomic<bool> m_reported = false;
    std::future<void> m_asyncTerminate;
    void publish();
    static void send(std::vector<ErrorHandler>& services, Error&
        error);
    static void send(ErrorHandler& service, Error& error);
protected:
    ErrorManager();
public:
    void set(Error& error);
    void set(ErrorLevel level, std::string& error,std::string&
        m_message,const Infor::ClassInfor& info);
    void set(ErrorLevel level, std::string_view error, std::
        string_view m_message,const Infor::ClassInfor& info);
    void set(ErrorLevel level, std::string_view error, std::
        string_view m_message);
    template<class F, class... Args>
    void subscribe(ErrorLevel level, F&& func, Args&&... args)
    {
        WriteLock lock(m_handleMutex);
        m_serviceByLevel[level].emplace_back(
            [func = std::forward<F>(func), args_tuple = std::
                make_tuple(std::forward<Args>(args)...)](Error&
                error) mutable
            {
                if constexpr (sizeof...(args) == 0) {
```

```
33                      std::invoke(func, error);
34                  }
35                  else {
36                      apply_custom(func, args_tuple, error);
37                  }
38              });
39      }
40      template<class F, class... Args>
41      void subscribe(std::string_view error, F&& func, Args&&... args
            )
42      {
43          WriteLock lock(m_handleMutex);
44          m_serviceByName[error.data()].emplace_back(
45              [func = std::forward<F>(func), args_tuple = std::
                    make_tuple(std::forward<Args>(args)...)]() mutable
46              {
47                  std::apply(func, args_tuple);
48              });
49      }
50      ~ErrorManager();
51  };
```

Code A.15: The declaration of ErrorManager

```
#define DYNAMIC template<class T>
DYNAMIC
class Singleton
{
  Singleton(const Singleton&) = delete;
  Singleton& operator=(const Singleton&) = delete;
  Singleton(Singleton&&) = delete;
  Singleton& operator=(Singleton&&) = delete;
protected:
  Singleton() = default;

public:
  static T& Get()
  {
    static T instance;

    return instance;
  }
};
#define INVITESINGLETON friend class Singleton;
```

Code A.16: The declaration of Singleton

```
struct Token
{
  TokenType type;
  std::string value;
  size_t position;//depreciated!!!!!!!!

  Token(TokenType t, std::string& v, size_t pos = 0);
};
```

Code A.17: The declaration of Token

```
class TableManager :public Singleton<TableManager>
{
  INVITESINGLETON;
  std::unordered_map<std::string, TablePtr> m_nameIndex;
  std::vector<TablePtr> m_tables;
  mutable ShareMutex m_listMutex;
  std::future<bool> m_isBlockTerminate;
  //read all table information from tables.struct
  void constructFromDataDefFile();
  void constructTableFromDataFile();
  int selectFromTables(std::string,const std::unordered_map<std::
     string, TablePtr>&);
  TableManager();
public:
  TablePtr add(User&, std::string& name, std::deque<Column>&,
     UserLevel level = UserLevel::None);
  bool exist(const std::string& name)const;
  bool drop(User&, const std::string& name);
  std::optional<unsigned> at(const std::string&);
  TablePtr get_noLock(const std::string&);
  ShareMutex& getMutex();
  TablePtr get(unsigned pos);
  bool existColumn(const std::string& name)const;
  void update();
  static void transfer(User&,TablePtr,User&);
  unsigned size()const;
  std::string toString() const;
  std::string format() const;
  void clear();
  ~TableManager();
};
```

Code A.18: The declaration of TableManager

A.2 Important function definitions

```
void Table::primize(UCharPtr& memory,unsigned byteSize,unsigned
    primedSize,unsigned recordSize)
{

  char* origin = memory.release();
  char* ptr = new char[primedSize + recordSize];
  memset(ptr, 0, primedSize + recordSize);
  Math::GmpBigNumber convert,recordNumber=1;
  unsigned m = 0;
  size_t count = 0;
  for (unsigned n = 0; n < byteSize;m+=8)
  {
    unsigned size = byteSize - n;
    if (size<4)
    {
      if (size == 1)
      convert = static_cast<unsigned char>(*(origin + n));
      else if (size == 2)
      {
        convert = *(unsigned short*)(origin + n);
      }
      else
      {
        convert = (static_cast<uint32_t>(*(origin + n)) << 16) |
        (static_cast<uint32_t>(*(origin + n + 1 )) << 8) |
        (static_cast<uint32_t>(*(origin + n + 2)) << 0);
      }
    }
    else
    convert = convert = *(unsigned int*)(origin + n);

    string before = convert.get_str();
    Math::PrimeNumberConvert::generate_big(convert, Util::Setting::
       Get().getEnlargePower());
    StaticFunc::WriteInfo("primize", std::format("From {} to {}",
       before, convert.get_str()));
    recordNumber *= convert;
    mpz_export(ptr + m, &count, Math::isBigEndian() ? 1 : -1, 1, 0,
        0, convert.get_mpz_t());
```

```
36      n += 4;
37    }
38    mpz_export(ptr + m , nullptr, Math::isBigEndian() ? 1 : -1, 1, 0,
          0, recordNumber.get_mpz_t());
39    count = 0;
40    memory.reset(ptr);
41    delete[] origin;
42  }
```

Code A.19: Primize Function

```
void Table::deprimize(UCharPtr& memory,unsigned byteSize,unsigned
    primedSize, unsigned recordSize)
{
  char* origin = memory.release();
  char* ptr = new char[byteSize];
  memory.reset(ptr);
  memset(ptr, 0, byteSize);
  Math::GmpBigNumber convert;
  unsigned m = 0;
  for (unsigned n =0 ;n< primedSize&&m< byteSize;n+=8)
  {
    unsigned size = byteSize - m;
    if (size < 4)
    {
      if (size == 1)
      convert = static_cast<unsigned char>(origin[n]);
      else if (size == 2)
      convert = *(unsigned short*)(origin + n);
      else
      convert = (static_cast<uint32_t>(origin[n]) << 16) |
      (static_cast<uint32_t>(origin[n + 1]) << 8) |
      (static_cast<uint32_t>(origin[n + 2]) << 0);
    }
    mpz_import(convert.get_mpz_t(), 8, Math::isBigEndian() ? 1 :
        -1, 1, 0, 0, origin+n);
    auto before = convert.get_str();
    convert /= Util::Setting::Get().getEnlargeSize();
    StaticFunc::WriteInfo("deprimize", std::format("From {} to {}",
        before, convert.get_str()));
    if (convert != 0)
    mpz_export(ptr + m, nullptr, Math::isBigEndian() ? 1 : -1, 1,
        0, 0, convert.get_mpz_t());
    m += 4;
  }
}
```

Code A.20: Deprimize Function

```
std::future<Result> Compiler::compile(UserPtr user,const std::
    string& sql)
{
  return std::async(std::launch::async, [user, sql]()
    {
      std::string inner = sql;
      Transaction transection;
      Record record;
      toLower(inner);
      if (inner == "commit"|| inner == "commit;")
      {
        user->commit();
      CallInfo("CompilerMessage", "Commit executed");
        return Result(true, "Commit success", 0);
      }
      std::string error;
      auto tokens = parseSQL(inner, error);
      if (error.empty()) {
      CallInfo("CompilerMessage", "Parsing Success");
        auto value = syntaxCheck(user, tokens, error);
        if (value.isValid())
        {
      CallInfo("CompilerMessage", "Syntax check pass");
          auto result = std::move(execute(user, value, error));
          if (result.isValid())
          {
            if (error.empty())
              return Result(result.isValid(), result.format(), 0);
            else
              return Result (result.isValid(), error, 0);
          }
        }
      }
      return Result(false, error, 0);
    });
}
```

Code A.21: The definition of compile function

```
std::deque<Token> Compiler::parseSQL(const std::string& sql, std::
    string& error)
{
  std::deque<Token> tokens;
  std::string currentToken;
  std::string internal = "Error:[ParsingError]";
  bool inQuotes = false;
  char quoteChar = '"';
  size_t position = 0;
  if (sql.empty())
  {
    internal += "Given command is empty (Parsing Error)";
    error = std::move(internal);
  }
  else
  {
    for (size_t i = 0; i < sql.length(); ++i) {
      char c = sql[i];
      position = i;
      if (!inQuotes) {
        if (c == '"' || c == '\'') {
          inQuotes = true;
          quoteChar = c;
          currentToken += c;
        }
        else if (c == ' ' || c == '\t' || c == '\n' || c == '\r') {
          if (!currentToken.empty()) {
            TokenType type = getTokenType(currentToken);
            if (type == TokenType::UNKNOWN)
            {
              internal += "Invalid token detected(Parsing Error)";
              error = std::move(internal);
              return tokens;
            }
            tokens.emplace_back(type, currentToken, position -
                currentToken.length());
            currentToken.clear();
          }
        }
        else if (c == ',' || c == '(' || c == ')' || c == ';' || c
```

```
           == '=' || c == '<' || c == '>' || c == '!') {
39          if (!currentToken.empty()) {
40            TokenType type = getTokenType(currentToken);
41            if (type == TokenType::UNKNOWN)
42            {
43              internal += "Invalid token detected (Parsing Error)";
44              error = std::move(internal);
45              return tokens;
46            }
47            tokens.emplace_back(type, currentToken, position -
                  currentToken.length());
48            currentToken.clear();
49          }
50          if (c != ';') {
51            std::string delim(1, c);
52            TokenType type = getTokenType(delim);
53            if (type == TokenType::UNKNOWN)
54            {
55              internal += "Invalid token detected (Parsing Error)";
56              error = std::move(internal);
57              return tokens;
58            }
59            tokens.emplace_back(type, delim, position); }}
60        else {
61          currentToken += c;
62        }
63      }
64      else {
65        currentToken += c;
66        if (c == quoteChar && (i == 0 || sql[i - 1] != '\\')) {
67          inQuotes = false;
68        }}}
69    if (!currentToken.empty()) {
70      TokenType type = getTokenType(currentToken);
71      tokens.emplace_back(type, currentToken, position -
           currentToken.length()); }}
72  return std::move(tokens);
73 }
```

Code A.22: The definition of PrimedDB parsing function

```
1  int Compiler::syntaxChecker(const std::vector<TokenType>& rule,
       std::deque<Token>& tokens, unsigned& tokenIter)
2  {
3    bool skip = false;
4    for (unsigned ruleIter = 0; ruleIter < rule.size()&& tokenIter<
       tokens.size(); ++ruleIter)
5    {
6      if (rule[ruleIter] == TokenType::REPEAT)
7      {
8        auto begin = tokenIter;
9        auto value = repeatCheck(rule, ruleIter, tokens, tokenIter)
           ;
10
11       if (value == -1)
12       {
13         ReportInfo("InvalidRepeat", "The format of repetition is
             incorrect.");
14         return tokenIter;
15       }
16       else if (value == 0)
17       {
18         ;
19       }
20       else
21       {
22         std::string size(tokenIter-begin, '.');
23         tokens.insert(tokens.begin() + begin,Token(TokenType::
             SIZE,size));
24
25         tokenIter++;
26       }
27       continue;
28     }
29     else if (rule[ruleIter] == TokenType::EXPRESSION)
30     {
31       if (expressionCheck(tokens, tokenIter))
32         continue;
33       ReportInfo("InvalidExpression", "Given expression is
           invalid.");
34       return tokenIter;
```

```
35      }
36      else if (rule[ruleIter] == TokenType::VALUE)
37      {
38        if (valueCheck(tokens, tokenIter))
39          continue;
40        ReportInfo("InvalidValueList", "Given value list is invalid
            .");
41        return tokenIter;
42      }
43      else if (rule[ruleIter] == TokenType::COLUMNS)
44      {
45        if (columnsCheck(tokens, tokenIter))
46          continue;
47        ReportInfo("InvalidColumnList", "Given column list is
            invalid.");
48        return tokenIter;
49      }
50      else if (rule[ruleIter] == TokenType::IDENTIFIERS)
51      {
52        if (identifiersCheck(tokens, tokenIter))
53          continue;
54        ReportInfo("InvalidIdentifierList", "Given IdentifierList
            is invalid.");
55        return tokenIter;
56      }
57      else if (rule[ruleIter] == TokenType::END && tokens.size()-
          tokenIter == 1)
58      {
59        ReportInfo("SyntaxCorrect", "Given command passed the
            Syntax check");
60        return -1;
61      }
62      else if (rule[ruleIter] == TokenType::OR)
63        continue;
64      if (tokenIter >= tokens.size()&& rule.size() - ruleIter <= 2)
65        return -1;
66      if (tokens[tokenIter].type != rule[ruleIter])
67      {
68        if (skip|| rule.size() - ruleIter == 1)
69        {
70          ReportInfo("SyntaxError", std::format("{} is type {} but
```

```
            {} required",
71            tokens[tokenIter].value,
72            type2string(tokens[tokenIter].type), type2string(rule[
                 ruleIter])));
73          return tokenIter;
74        }
75        skip = true;
76      }
77      else
78      {
79        if (skip)
80          skip = false;
81        ++tokenIter;
82      }
83    }
84    return -1;
85  }
```

Code A.23: The definition of PrimedDB sytax checking function

```
void Table::insert(const std::string& userName, UCharPtr memory,
    unsigned size)
{
  auto blockPos = convertBlockPos(this->m_size);
  WriteLock lock(m_mutex);
  if (m_owned.empty() || m_available.size()/Util::Setting::Get().
      getBlockSize()+1 > m_owned.size())
    blockPos.first = -1;
  primize(memory,m_byteSize,m_primedSize,m_recordByte);
  StaticFunc::WriteInfo(m_name, std::format("Received request from
      {} to write data into block", userName));
  const char* ptr = memory.get();
  auto user = UserManager::Get().get(userName);
  if (user != nullptr)
  {
    user->submit(Transaction(userName, m_name, SQLType::Insert,
        blockPos.first, blockPos.second, size, std::move(memory)));
  }
}
```

Code A.24: The definition of PrimedDB data insertion function

```
bool UserManager::allowLogin(const std::string& name, const std::
    string& password)
{
  shared_ptr<User> user;
  if (!exist(name))
    return false;
  {
        ReadLock lock(m_mutex);
    user = m_allUsers[name];
  }
  return user->validate(User::PassWordHash(password));
}
```

Code A.25: The definition of user validation

```
unsigned BlockManager::allocate(TablePtr table, int pos)
{
  auto newBlock = allocate();
  if (pos == -1)
  {
    table->addBlock(newBlock);
    m_blocks[newBlock].assign(table, table->getOwned().size()*table
       ->totalByte());

  }
  else
  {
    table->getOwned()[pos/StaticFunc::MaxSizeForBlock(table->
       totalByte())] = newBlock;
    m_blocks[newBlock].assign(table, pos);
  }
  StaticFunc::WriteInfo("BlockManager",
    std::format("Allocate block id:[{}] to table:[{}] success",
      std::to_string(newBlock), table->getName()));
  return newBlock;
}
```

Code A.26: The definition of allocating a new block to table

```
void BlockManager::rearrange(std::deque<int>& owned, std::vector<
    bool>& available)
{
  std::vector<char*> memory;
  unsigned byteSize = 0;
  std::vector<ReadLock> locks;
  for (unsigned n =0;n<owned.size();++n)
  {
    if (byteSize == 0)
    {
      byteSize = m_blocks[n].byte();
    }
    for (unsigned m = 0;m<m_blocks[n].size();++m)
    {
      locks.emplace_back(m_blocks[n].getMutex());
      memory.push_back(m_blocks[n].get_noLock(m));
    }
  }
  unsigned fast = 0, slow =0;
  for (;fast<available.size();++fast)
  {
    if (!available[fast])
    {
      memcpy_s(memory[slow],byteSize,memory[fast],byteSize);
      available[slow] = true;
      slow = fast;
    }
  }
}
```

Code A.27: The clear procedure when table have too much free memory

```
1  void BlockManager::manager()
2  {
3    Lock lock(m_cvMutex);
4    while (true)
5    {
6      m_cv.wait(lock, [this]()
7      {
8          return m_notify||m_end;
9      });
10
11     while (!m_pendingOperations.empty())
12     {
13       Transaction transaction;
14       {
15         WriteLock lock(m_mutex);
16         transaction = std::move(m_pendingOperations.front());
17         m_pendingOperations.pop_front();
18       }
19       StaticFunc::WriteInfo("Transaction", std::format("Handling
             Transaction [{}]", transaction.getId()));
20       handle(transaction);
21     }
22     m_notify = false;
23     if (m_end)
24       break;
25   }
26 }
```

Code A.28: A separate thread which handles all transactions

```
void BlockManager::handle(Transaction& transection)
{
  if (transection.isValid())
  {
    transection.setInvalid();
    auto memory = transection.getMemory();
    int position = transection.getBlockId();
    auto table = TableManager::Get().get_noLock(transection.
       getTable());
    unsigned location = transection.getLocation();
    if (transection.getType() == SQLType::Insert)
    {
      std::pair<unsigned, std::shared_ptr<char[]>> result = std::
         make_pair(transection.size(), std::move(memory));
      if (position == -1)
      {
        position = allocate();
        transection.setBlockId(position);
        {
          m_blocks[position].assign(table, table->size());
        }
        table->addBlock(position);
      }
      result = m_blocks[position].insert(result.second, result.
         first);
      table->incrementSize();
      TableManager::Get().update();
    }
    else if (transection.getType() == SQLType::Update)
    {
      if (position == -1)
      {
        Util::ErrorManager::Get().set(Util::ErrorLevel::Error, "
           BlockIdError", "Given block id is incorrect.", Infor::
           ClassInfor(THISFUNC, THISFILE));
        return;
      }
      m_blocks[position].update(location, memory);
    }
    else if (transection.getType() == SQLType::Delete)
```

```
36      {
37        if (position == -1)
38        {
39          Util::ErrorManager::Get().set(Util::ErrorLevel::Error, "
                BlockIdError", "Given block id is incorrect.", Infor::
                ClassInfor(THISFUNC, THISFILE));
40          return;
41        }
42        m_blocks[position].remove(location);
43        table->decrementSize(location);
44        if (m_blocks[position].percentage() < 0.5)
45        {
46          rearrange(table->getOwned(), table->getAvailable());
47        }
48        TableManager::Get().update();
49      }
50
51    }
52
53  }
```

Code A.29: The handler function which handling transactions by category on the handling thread

```
void ErrorManager::set(ErrorLevel level, std::string& error, std::
    string& m_message, const Infor::ClassInfor& info)
{
    Error errorObject(level, error, m_message, info);
    set(errorObject);
}
void ErrorManager::set(ErrorLevel level, std::string_view error,
    std::string_view m_message,const Infor::ClassInfor& infor)
{
    Error errorObject(level, error, m_message,infor);
    set(errorObject);
}
void ErrorManager::set(ErrorLevel level, std::string_view error,
    std::string_view m_message)
{
    Error errorObject(level, error, m_message, Infor::ClassInfor())
        ;
    WriteLock lock(m_mutex);
    m_errors.emplace_back(errorObject);
    m_reported = true;
    m_conditionVar.notify_one();
}

void ErrorManager::set(Error& error)
{
    //if (!contains(error.m_name))
    {
        WriteLock lock(m_mutex);
        m_errors.emplace_back(error);
        m_reported = true;
    }
    m_conditionVar.notify_one();
}
```

Code A.30: The four different overloads for set() which submit an error into ErrorManager

```cpp
template<class F, class... Args>
void subscribe(ErrorLevel level, F&& func, Args&&... args)
{
    WriteLock lock(m_handleMutex);
    m_serviceByLevel[level].emplace_back(
        [func = std::forward<F>(func), args_tuple = std::make_tuple
           (std::forward<Args>(args)...)](Error& error) mutable
        {
            if constexpr (sizeof...(args) == 0) {
                std::invoke(func, error);
            }
            else {
                apply_custom(func, args_tuple, error);
            }
        });
}
template<class F, class... Args>
void subscribe(std::string_view error, F&& func, Args&&... args)
{
    WriteLock lock(m_handleMutex);
    m_serviceByName[error.data()].emplace_back(
        [func = std::forward<F>(func), args_tuple = std::make_tuple
           (std::forward<Args>(args)...)]() mutable
        {
            std::apply(func, args_tuple);
        });
}
```

Code A.31: The two different overloads for subscribe() which respectively subscribe a specific error type or error message with a handler function

```
static void ErrorLog(Error& error)
{
    Infor::Log::Get()[Infor::LogType::Error].openToFile("error"+
       NOW.getDate()).split('-') << error.m_name << error.m_message
        << error.m_info << Infor::Log::LogEndl;
}
static void FatalLog(Error& error)
{
    Infor::Log::Get()[Infor::LogType::Fatal].openToFile(std::string
       ("fatal"+ NOW.getDate())).split('-') << error.m_name <<
       error.m_message << error.m_info << Infor::Log::LogEndl;
}
static void WarningLog(Error& error)
{
    Infor::Log::Get()[Infor::LogType::Warning].openToFile(std::
       string("warning"+ NOW.getDate())).split('-') << error.m_name
        << error.m_message << Infor::Log::LogEndl;
}
static void InfoLog(Error& error)
{
    Infor::Log::Get()[Infor::LogType::Info].split('-') << error.
       m_name  << error.m_message << Infor::Log::LogEndl;
}
```

Code A.32: Three predefined handler functions which log message for different error type

Bibliography

[1] Jennifer Widom Hector Garcia-Molina Jeffrey D. Ullman. *DATABASE SYSTEMS The Complete Book*. English. 2nd ed. Pearson Education, Inc, 2010. Chap. 13, p. 590. ISBN: 978-0-13-606701-6.

[2] Bruce Schneier. *Applied Cryptography-Protocols, Algorithms, and Source Code in C*. Chinese. Trans. English by Wu Shizhong, Zhu Shixiong, and Zhang Wenzhen. 2nd ed. John Wiley and Sons, Inc, 2014. Chap. 19, p. 330. ISBN: 978-7-111-44533-3.

[3] Thomas H. Cormen et al. *Introduction to Algorithms*. English. 4th ed. Cambridge, Massachusetts London, England: Massachusetts Institute of Technology, 2022. Chap. 32, p. 1257. ISBN: 9780262367509.

[4] Microsoft. *Microsoft Defender SmartScreen Q and A*. 2025. URL: `https://feedback.smartscreen.microsoft.com/smartscreenfaq.aspx` (visited on 11/21/2025).

[5] LIAO RUNKANG. *Download link of PrimedDB server*. 2025. URL: `https://github.com/Jeffrey-Liao/PrimedDB/releases/tag/release` (visited on 11/25/2025).

[6] LIAO RUNKANG. *Download link of PrimedDB client*. 2025. URL: `https://github.com/Jeffrey-Liao/PrimedDBC/releases/tag/installer` (visited on 11/25/2025).

[7] John H. Silverman. *A Friendly Introduction to Number Theory*. Chinese. Trans. English by Sun Zhiwei, Wu Kejian, and Cao Huiqin Lu Qinglin. 3rd ed. Pearson Education Inc., 2008. Chap. 13, p. 53. ISBN: 978-7-111-23911-6.

[8] John H. Silverman. *A Friendly Introduction to Number Theory*. English. 3rd ed. Pearson Education Inc., 2008. Chap. 12, p. 78. ISBN: 978-7-111-23911-6.

[9] E.M.Wright G.H.Hardy. *An Introduction to the Theory of Numbers*. Chinese. Trans. English by Zhang Fan Zhang Mingyao. 6th ed. Post and Telecom Press, 2010. Chap. 1, p. 3. ISBN: 978-7-115-23203-8.

[10] Jennifer Widom Hector Garcia-Molina Jeffrey D. Ullman. *DATABASE SYSTEMS The Complete Book*. English. 2nd ed. Pearson Education, Inc, 2010. Chap. 21, p. 1056. ISBN: 978-0-13-606701-6.

[11] The GMP Project. *The GNU Multiple Precision Arithmetic Library*. 2025. URL: https://gmplib.org// (visited on 11/10/2025).

[12] Intel Corporation. *13th Generation lntel Core, lntelCore 14th Generation, lntel Core Processor (Series 1) and (Series 2),lntel Xeon E 2400 Processor and lntel Xeon 6300 Processor Datasheet, VoIume 1 of 2*. Accessed: 2025-11-23. 2025. URL: https://cdrdv2.intel.com/v1/dl/getContent/743844/view.

[13] Intel Corporation. *Intel® Core™ Ultra Processor Datasheet Volume 1 of 2*. Accessed: 2025-11-23. 2025. URL: https://cdrdv2.intel.com/v1/dl/getContent/792044/view.

[14] Inc. Advanced Micro Devices. *Processor Specifications*. 2025. URL: https://www.amd.com/en/products/specifications/processors.html (visited on 11/23/2025).

[15] Arm Limited. *Learn the architecture - AArch64 memory management Guide*. 2025. URL: https://developer.arm.com/documentation/101811/0105/Address-spaces/Address-sizes?lang=en (visited on 11/23/2025).

[16] Crypto++. *Crypto++*. 2025. URL: https://www.cryptopp.com/ (visited on 11/21/2025).

[17] Boost library. *Boost Asio*. 2025. URL: https://www.boost.org/doc/libs/latest/doc/html/boost_asio.html (visited on 11/10/2025).

[18] International Organization of Standardization. *C++ standard*. 2025. URL: https://isocpp.org/std/standing-documents (visited on 11/23/2025).

[19] Torbjorn Granlund and the GMP development team. *The GNU Multiple Precision Arithmetic Library Manual*. Accessed: 2025-11-23, page 38. 2023. URL: https://gmplib.org/gmp-man-6.3.0.pdf.

[20] Jennifer Widom Hector Garcia-Molina Jeffrey D. Ullman. *DATABASE SYSTEMS The Complete Book*. English. 2nd ed. Pearson Education, Inc, 2010. Chap. 13, p. 3. ISBN: 978-0-13-606701-6.

List of Figures

List of Tables

List of Codes