Allegheny College
Computer And Information Science Department

Senior Thesis

QrypticDB – A Cryptography-Inspired Database Leveraging Symmetric Encryption

by

Miles Franck

ALLEGHENY COLLEGE

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Project Supervisor: **Gregory Kapfhammer**
Co-Supervisor: **Douglas Luman**

Abstract

This project presents a secure database solution designed for individuals and small organizations that uses PostgreSQL and encryption. The system implements full database encryption that protects data at rest while maintaining performance during active use, addressing the need for accessible security solutions for non-enterprise users. A key innovation is the platform-agnostic approach to key management, which leverages native operating system security mechanisms to store encryption keys separately from the protected data. Performance analysis through doubling experiments demonstrates the system's scalability across various dataset sizes. The architecture's modular design allows users to customize security levels based on their specific requirements, providing flexibility without compromising core protection. This work contributes to the field by offering an open-source implementation that bridges the gap between enterprise-grade security and the practical needs of small-scale database users, making strong cryptographic protection accessible without specialized security expertise.

Table of contents

1 Introduction **6**
- 1.1 Terms and Definitions . . . 6
- 1.2 Real-world Meaning . . . 7
- 1.3 Statement of the Problem . . . 8
- 1.4 Motivation . . . 9
- 1.5 Current State of the Art . . . 12
  - 1.5.1 Databases . . . 12
  - 1.5.2 PostgreSQL . . . 14
  - 1.5.3 BitLocker . . . 14
  - 1.5.4 SQLCipher . . . 15
- 1.6 Defining My Project . . . 15
- 1.7 Assumptions and Limitations and Delimitations . . . 15
  - 1.7.1 Assumptions . . . 15
  - 1.7.2 Limitations . . . 16
  - 1.7.3 Delimitations . . . 16
- 1.8 Ethical Implications . . . 16

2 Related Work **18**
- 2.1 Databases . . . 18
  - 2.1.1 PostgreSQL . . . 18
  - 2.1.2 MySQL . . . 19
- 2.2 Database Security Methods . . . 19
  - 2.2.1 Access Control Method . . . 20
  - 2.2.2 Database Encryption . . . 22
  - 2.2.3 Defense Against SQL Injections . . . 23
- 2.3 Similar Software . . . 26
- 2.4 Database Testing . . . 27

3 Method of Approach **29**
- 3.1 System Architecture . . . 32
  - 3.1.1 Security Implementation . . . 35
- 3.2 Security Implementation Overview . . . 35
- 3.3 Full Database encryption for comprehensive data protection: . . . 37
- 3.4 Secure key management system: . . . 37
- 3.5 Plan to Measure My Experiment . . . 38

4 Experiment **40**
- 4.1 Experimental Design Background . . . 40
- 4.2 Evaluation . . . 43
- 4.3 Exploring The Raw Data . . . 43
- 4.4 Exploring The Graph Data . . . 45
- 4.5 Summary of Results . . . 48
- 4.6 Validation Methodology . . . 48

5 Conclusion **50**
5.1 Takeaways 50
5.2 Ethical Considerations 50
5.3 Protection Against Malicious Use 51
5.4 Future Work 52
5.4.1 Different Encryption Methods 52
5.4.2 Built in Encryption Methods 52
5.4.3 Backups 52
5.4.4 Logging 52
5.4.5 Misuse Prevention 53
5.4.6 Total Encryption 53
5.5 Final Thoughts 53

List of Figures

1 Comparison 11
2 ACID Graph 23
3 database decrypt 46
4 database encrypt 46
5 database 3 db encrypt 47
6 database 1 db encrypt 47

List of Tables

1 Introduction

My thesis focuses on developing a secure, efficient, and easy-to-use encrypted database system designed for individuals and small groups to manage their data safely and effectively. By leveraging Python and PostgreSQL as its core technologies, this project addresses critical challenges in database encryption, such as optimizing performance, ensuring seamless integration with existing workflows, and securely managing encryption keys to prevent unauthorized access.

A key objective of this thesis was to strike a balance between security and usability. Security was defined using a modified version of the CIA (Confidentiality, Integrity, Availability) triad, which I adapted to CCR: Confidentiality, Control, Resilience. The tools used in my program inherently addressed integrity and availability, allowing me to focus on confidentiality, control, and resilience.

As for usability, my goal with this project was to make it so easy to use, the user does not have to think about anything. A lot of other software requires setup or even having some coding knowledge, my goal was to make sure that isn't required.

By providing a practical and accessible solution for users who require strong data protection without the complexity often associated with encrypted database systems, I created a system that safeguards sensitive information while remaining efficient and user-friendly for individuals and small teams in diverse environments.

1.1 Terms and Definitions

- Data breach: A data breach involves the unauthorized access and exfiltration of sensitive personal information from organizations.[24] Data breaches usually occur because of human error, security vulnerabilities and insider threats and even physical theft. Human error is when a worker or employee accidentally discloses sensitive financial data, they fall for a phishing email, a salesperson unnecessarily carrying sensitive customer data that leaks. A security vulnerability is when there is an exploit in software systems. Cybercriminals often exploit these weaknesses to gain unauthorized access to sensitive information. An insider threat can be something like a disgruntled employee selling sensitive data to an adversary, lastly theft of physical property can be the cause of a data breach. If a laptop containing important information is stolen, that can lead to a data breach [19].

- Database: A database is an organized collection of data that is stored and managed to enable easy access, retrieval, and updating. It serves as a central system for storing information that can be structured, such as tables and records, or unstructured, like documents and multimedia files. Databases are used in various applications to support decision-making, operations, and analytics, and they are managed using database man-

agement systems (DBMS). These systems ensure data is securely stored, efficiently queried, and consistently maintained [12].

1.2 Real-world Meaning

In an increasingly digital world, data has become one of the most valuable resources—and one of the most vulnerable. From personal identities to corporate trade secrets, sensitive information now lives on devices and servers, where it is often inadequately protected.

Data breaches can happen in many ways, but it only takes one for irreparable damage to occur.

Data breaches are a serious threat to enterprise operations, such as corporations and government agencies. The loss of sensitive information can lead to significant reputational damage and financial losses, and can even be detrimental to the long-term stability of an organization. Common types of leaked information range from employee/customer data, intellectual property, to medical records. According to IBM's 2016 Cost of Data Breach Study, the average consolidated cost of a data breach has reached $4 million. Juniper Research's forecast suggests that the global annual cost of data breaches will be over $2.1 trillion globally by 2019, due to the rapid digitization of consumers' lives and enterprise records [13]. Over the past few years, there have been many notable data loss incidents that cost companies millions of dollars. Cybercriminals breached the Target Corporation's network in 2013, stealing 40 million payment cards information and 70 million customers' personally identifiable information, which has incurred $248 million losses to date reported by Target3. In 2016, Yahoo reported that at least 500 million accounts in 2014 had been stolen in an apparent 'state sponsored' data breach. Since data volume is growing exponentially in the digital era and data leaks happen more frequently than ever before, preventing sensitive information from being leaked to unauthorized parties becomes one of the most pressing security concerns for enterprises[3].

Big companies are not the only ones affected by data breaches; small businesses are too. While large corporations often have dedicated cybersecurity teams and the resources to implement robust security measures, small businesses are more vulnerable due to their limited financial and technological resources. A data breach can be devastating for small businesses, leading to substantial financial losses, especially as many small firms cannot absorb the costs of a breach. Additionally, small businesses are heavily reliant on their reputation to maintain customer trust, and a breach of sensitive information can lead to a significant loss of customer loyalty and future revenue [23].

The impact of a data breach on small businesses extends beyond just financial loss. Legal risks are also a major concern, as small businesses may not have the expertise or resources to navigate complex data protection laws. Cyberattacks can also disrupt operations, causing interruptions that small businesses may struggle to recover from. With the rise of cyber threats, it is essential that small businesses implement stronger cybersecurity frameworks to safeguard their data and protect their ongoing viability in the face of these risks.

Customers trust companies to keep their data safe, but this trust comes with a trade-off: they relinquish control over their personal information. When a company fails to protect that data, the consequences can be devastating, from financial losses to the erosion of trust. Once trust is lost, it's incredibly hard to rebuild. That's why organizations must adhere to strict principles of data management.

First, proper acquisition and retention ensure that companies only collect and retain information that is necessary for their operations, reducing the risk of unnecessary exposure. They must also maintain data integrity, ensuring that information is accurate, complete, and timely. Any errors, such as outdated or incomplete data, can have serious consequences for individuals and the company's reputation. Processes like aggregation and derivation where data is combined or new information is deduced add another layer of complexity. These derived datasets often contain sensitive insights that require the same level of protection as the original data.

Organizations must also handle information sharing responsibly, limiting disclosure to authorized parties and ensuring it is done for specific, well-defined purposes. Finally, proper access control is vital, restricting data to individuals who have a clear, demonstrable need for it. This principle prevents unnecessary exposure and protects against both external and internal threats. By adhering to these practices, organizations can minimize the risk of breaches and uphold the trust customers place in them [11].

The very act of entrusting companies with sensitive data means that customers lose their ability to manage how their information is stored, used, and shared. They rely entirely on these organizations to make decisions on their behalf, often with limited insight into or influence over those choices. This lack of control leaves customers vulnerable, as they must depend on the company's commitment to safeguarding their data. If those companies make mistakes or fail to uphold best practices, it's the customers who suffer the consequences.

1.3 Statement of the Problem

Small businesses and individuals that rely on large corporations like AWS for data storage face considerable risks. If a hosting provider suffers a data breach, their sensitive information is exposed, often with severe financial and operational consequences. While outsourcing storage offers convenience, it also introduces inherent security vulnerabilities including the risk of malicious attacks and human error, which remains one of the leading causes of data breaches, even within organizations that implement rigorous security training [1].

To mitigate these risks, some turn to local solutions like BitLocker, VeraCrypt, or SQLCipher, which offer strong encryption for disks or databases. However, these tools often come with steeper setup requirements, limited flexibility, or require users to have technical knowledge to properly configure and manage encryption. For example, BitLocker, while it's a great tool for protecting things on your computer, it often requires users to navigate complex setup requirements that go far beyond a simple toggle. For example, enabling

BitLocker on devices with TPM 2.0 demands native UEFI mode, with Legacy BIOS and Compatibility Support Module (CSM) explicitly disabled. Additionally, the system must have two properly partitioned drives with specific formats: one for the OS (NTFS) and another boot drive (FAT32 or NTFS depending on firmware). Microsoft documentation outlines these prerequisites, including the need for TCG compliant firmware and UEFI Secure Boot support technical details that may be overwhelming for average users [16].

While solutions like BitLocker offer full-disk encryption, they may not be flexible enough for users who only want to encrypt specific files or parts of a system. Other tools, especially encrypted databases, provide more targeted security but often come with trade-offs.

For example, SQLCipher is a powerful open-source encrypted database engine based on SQLite. While it offers a free community edition, some features, such as official support, enhanced performance builds, or integrations are only available through a commercial license. This means that for individuals or small teams without funding, access to enterprise-grade encryption may come at a significant cost [26].

As a result, users often face a difficult decision: choose between limited, free tools or pay for more advanced options. This gap in accessibility is one of the core problems my project aims to address providing flexible encryption tools that are both free and user-controlled, without compromising on functionality or security

What if individuals and small businesses could maintain full control over their own data and security without relying on third-party providers or navigating complex, technical solutions? By eliminating dependence on external platforms, users could significantly reduce their exposure to data breaches caused by another company's mistakes. At the same time, avoiding overly technical tools like full-disk encryption setups or paid encrypted databases lowers the barrier to entry.

A system that allows users to manage their own encrypted database—easily and independently—ensures that sensitive information stays under their control. It gives them the freedom to implement security measures tailored to their specific needs without requiring deep technical expertise or costly software. Empowering users in this way is a critical step toward addressing the risks of both outsourced storage and inaccessible security solutions.

1.4 Motivation

In today's digital environment, small businesses and individuals often rely on third-party providers like AWS and Google Cloud to store their sensitive information. While convenient, this dependency introduces serious risks: data breaches caused by human error, malicious attacks, or even physical device theft can expose users to significant harm. Even tools designed to secure data, such as BitLocker or SQLCipher, come with their own barriers. BitLocker requires specific hardware configurations, TPM support, and often involves BIOS or UEFI setup that can be confusing or inaccessible to non-technical users. SQLCipher,

while powerful, may lock some features behind a paid license and assumes a level of technical fluency that many individuals or small teams simply don't have.

This raised a key question for me:
What if individuals and small businesses could maintain full control over their own data and security without relying on third-party services or complex, technical tools?

While working on my thesis, my professor introduced me to the Zettelkasten method, a system for personal knowledge management that completely changed how I thought about organizing and controlling information. Zettelkasten, which translates to "slip box" in German, was developed by sociologist Niklas Luhmann to organize his ideas as a network of interlinked, standalone notes. This decentralized structure allows users to tailor their information system to their needs and grow it organically over time.

What struck me most about Zettelkasten was its emphasis on user empowerment. Rather than forcing users into rigid, pre-designed categories or relying on external software to dictate how knowledge is managed, Zettelkasten puts the individual in full control. This felt directly applicable to the problem of secure data storage: if users could manage their information in a similar way intuitively, securely, and without technical overhead they could escape the limitations and risks of third-party solutions.

The parallels between Zettelkasten and secure data management became more apparent as I worked through the method. In Zettelkasten, users actively engage with their data, create their own structures, and build a network that reflects their own priorities. Applied to databases, this would mean designing a storage system that gives users the ability to encrypt, organize, and control their data based on their specific needs without having to trust or depend on external providers.

Inspired by these ideas, I developed a lightweight, user-friendly encrypted database system that brings these principles to data security. The tool empowers small teams or individuals to take responsibility for their sensitive data. It is simple enough for non-experts yet secure enough to serve as a viable alternative to cloud-based solutions or complex corporate systems.

This project addresses a real and growing need. Large companies have the resources to invest in advanced cybersecurity infrastructure, but smaller entities often lack both the budget and the expertise.

My project aims to bridge this gap by creating an intuitive, accessible encryption and data management system that doesn't require specialized knowledge. The focus is deliberately on simplicity without compromising security offering features that prioritize ease of use such as straightforward setup processes, clear visual interfaces, and guided security protocols that don't overwhelm users with technical jargon.

Drawing inspiration from the Zettelkasten method's user-friendly approach to knowledge management, this database system empowers users to organize their information in ways that make sense to them personally. Rather than forcing adaptation to complex enterprise security models, it provides core protec-

tion features that can be implemented with minimal technical overhead making strong encryption accessible even to those with no background in data security.

At its heart, this solution recognizes that effective security tools must work for real people in everyday contexts. By creating a system that feels approachable and manageable for individuals and small teams while also remaining flexible to change, the project offers practical data protection that people will actually use, rather than avoid due to complexity. This human centered approach to security technology addresses a significant gap in current offerings, where solutions tend to be either too simplistic to be secure or too complicated to be widely adopted.

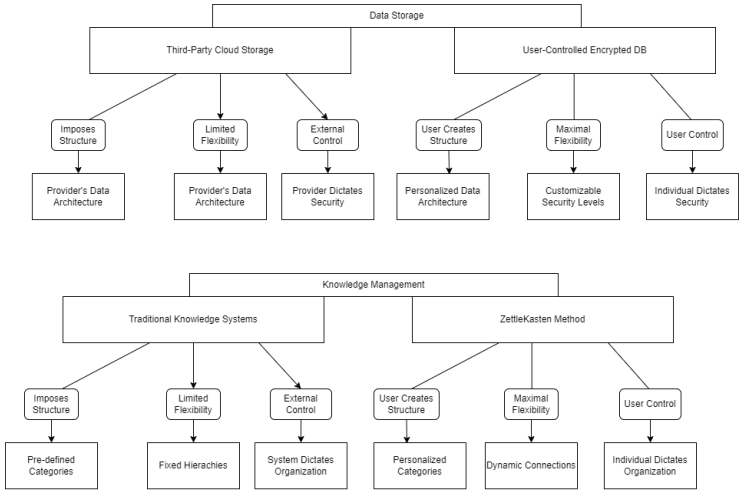


Figure 1: Comparison

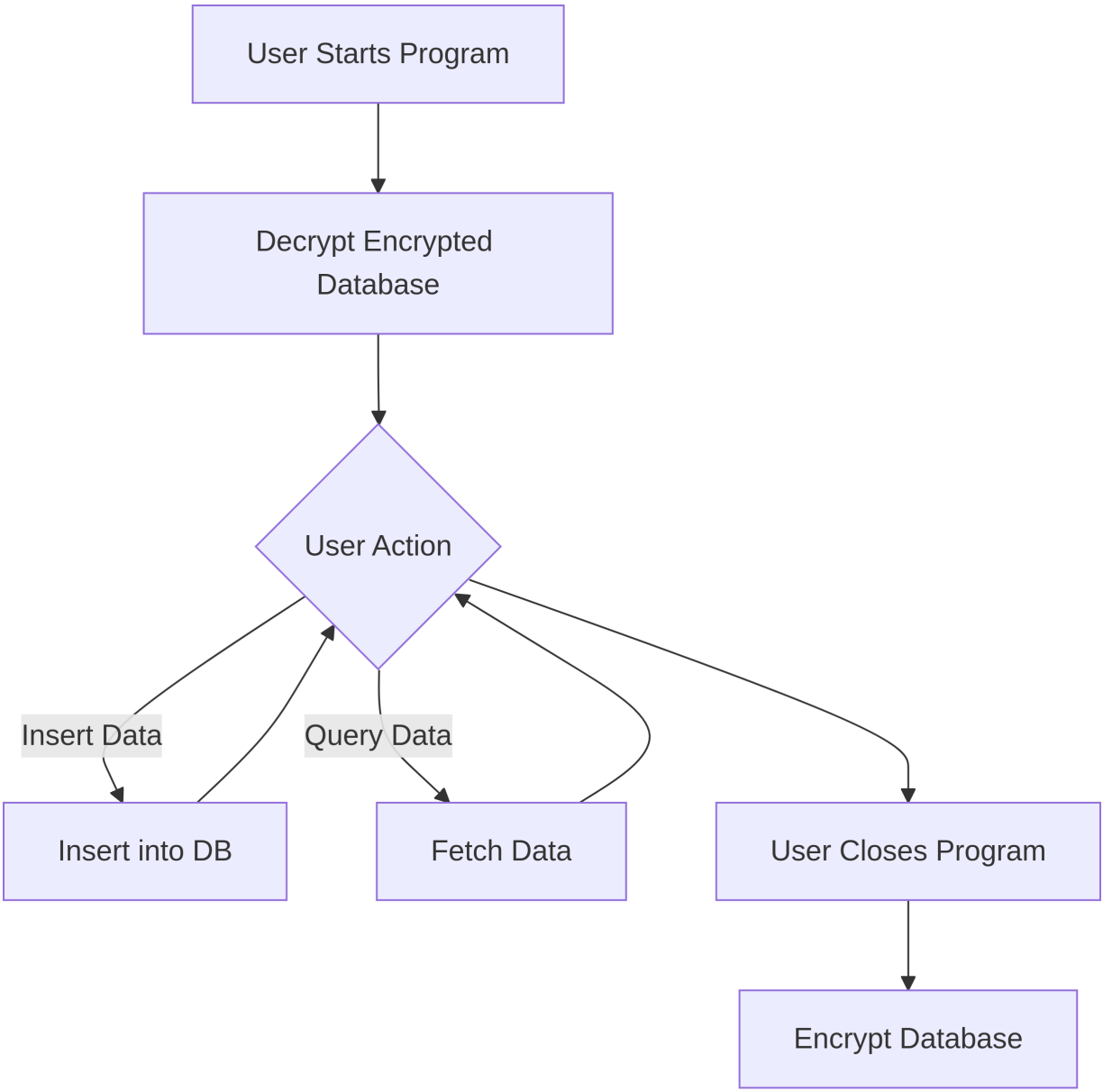
This figure illustrates the parallel between knowledge management systems and data storage approaches. On the left side, traditional knowledge systems impose rigid structures with predefined categories and fixed hierarchies, while the Zettelkasten method empowers users to create personalized frameworks with dynamic connections. Similarly, on the right side, third-party cloud storage imposes provider-defined data architectures and security measures, while a user-controlled encrypted database system would allow individuals to design their own data structures and security protocols. The dotted lines connecting these two domains highlight how the same principles of user empowerment that make the Zettelkasten method effective for knowledge management could be applied to data storage security. This comparison demonstrates that shifting control to the individual can lead to systems that are not only more secure but also better aligned with each user's specific needs and preferences.

1.5 Current State of the Art

1.5.1 Databases

While this will be further explored in the Related works section, to have a better understanding of my project this will be an overview of existing work. Today's world we use a database for almost everything. There are an important part of our life even if we don't know it. But what do the modern day systems do.

Modern databases use multiple methods to keep information safe. Before diving into these security mechanisms, it's helpful to understand how a database system operates. The diagram below illustrates a typical database system workflow, showing the interaction between the database management system (DBMS), applications, and users.



As shown in the diagram, applications communicate with the DBMS to send queries and retrieve data. The DBMS acts as a mediator, ensuring that data is stored, processed, and secured efficiently. This structure plays a crucial role in implementing security mechanisms, such as access control, query auditing, and inference control, which modern databases use to safeguard information.

Modern databases use multiple methods to keep information safe. They have access control mechanisms that decide who can see or change data, ensuring only

authorized users can access it. They also track where information comes from to prevent unauthorized sharing. Additionally, they consider the history of user queries to make sure new queries don't reveal too much information. Inference control helps stop users from figuring out new information by asking a series of questions. Finally, policies are in place to control what information can be shared in response to queries, preventing users from deducing exact details about individuals. These methods work together to enhance the security and privacy of modern databases [4].

While these security mechanisms form the foundation of modern database protection, practical implementation depends heavily on the capabilities of the underlying database system. One widely adopted and feature rich option is PostgreSQL, which combines industry-standard security practices with powerful extensibility and advanced encryption features.

1.5.2 PostgreSQL

PostgreSQL is a powerful, open-source relational database system that is widely regarded for its robustness, extensibility, and compliance with industry standards. It offers full ACID (Atomicity, Consistency, Isolation, Durability) compliance, making it suitable for applications requiring high reliability and transactional integrity. With support for advanced features such as JSON/JSONB, full-text search, and custom data types, PostgreSQL bridges the gap between relational and semi-structured data storage. Due to its strong performance, active community, and flexibility, PostgreSQL remains a leading choice for modern database-driven applications across various domains [8].

With this foundation in place, it's worth looking at how security is handled at different layers of the data stack. While some solutions secure the entire system drive, others focus on encrypting the database itself. To better understand the range of existing solutions, this section will briefly examine some widely used tools.

1.5.3 BitLocker

BitLocker is a full-disk encryption tool built into Microsoft Windows, designed to secure data at rest by encrypting the entire volume. It works at the operating system level and is typically used to protect devices in enterprise environments. BitLocker is integrated with the system's Trusted Platform Module (TPM), providing seamless encryption and decryption tied to the hardware.

While it offers strong protection against offline attacks and physical theft, BitLocker is largely invisible to the user once enabled; it encrypts everything, whether it's necessary or not. It's not particularly suited for more fine-grained control, such as encrypting just a single application's data or enabling selective sharing between users or systems [16].

1.5.4 SQLCipher

SQLCipher is a specialized encryption extension for SQLite. It encrypts the contents of a database at the application level, offering developers control over what gets encrypted and how keys are managed. Unlike BitLocker, SQLCipher is cross-platform and open-source, making it attractive for mobile apps and embedded systems where lightweight and embedded security is important [25].

1.6 Defining My Project

The core question driving this thesis was: How can a database system be designed to prioritize security and user control without compromising performance or usability? To tackle this challenge, I designed, developed, and evaluated a secure database system using Python and PostgreSQL, ensuring that individuals and small groups could effectively manage their own data with robust encryption measures in place.

To assess the effectiveness of my approach, I conducted thorough performance evaluations, analyzing key factors such as encryption overhead, response times, and computational complexity. These tests demonstrated how well the system maintained efficiency while ensuring data security. Additionally, I evaluated its resilience against potential security threats, ensuring that the encryption methods provided meaningful protection without adding unnecessary complexity for users.

Beyond its technical contributions, the significance of this project lies in its broader implications for digital independence. It advocates for a shift in data ownership, empowering users to manage their own information securely without relying on external platforms that may introduce security risks. This system is designed to provide strong encryption and security protections while maintaining ease of use, ensuring that individuals and small teams do not have to choose between privacy and practicality. Small businesses, independent professionals, researchers, and privacy-conscious individuals stand to benefit from a tool that allows them to easily store, organize, and manage sensitive data on their own terms free from the vulnerabilities and uncertainties that come with third party data storage solutions.

1.7 Assumptions and Limitations and Delimitations

1.7.1 Assumptions

The goal of this project is to develop a program that allows individuals and small groups to securely store and manage their data. For this study, "individuals or small groups" refers to single users or small teams, such as a professor and a group of students, who need a reliable and private database solution without relying on third-party services. This application's potential extends to small businesses, which will be explored in depth within the experiment section. While PostgreSQL is a powerful and scalable database management system capable of supporting large-scale operations—including unlimited database size, up to

4,294,950,911 databases, 1,431,650,303 relations per database, and relation sizes up to 32 TB with the default block size [7]—this project is deliberately designed with more modest, practical use cases in mind. The focus centers on moderate data storage needs typical of academic environments or small collaborative projects, rather than enterprise-scale systems. Small businesses implementing this solution should remain mindful of scaling constraints; if performance degradation occurs or storage requirements exceed capacity, transitioning to a commercial third-party solution would become necessary.

My last assumption is that users have a foundational understanding of PostgreSQL and can write basic SQL queries. This knowledge is essential for tasks like data storage, retrieval. The project is specifically tailored to users who are comfortable running PostgreSQL on their local machines and taking responsibility for managing their data independently. As such, it is not designed as a fully automated or beginner-friendly database solution but rather as a secure, customizable system for those who prefer direct control over their data.

1.7.2 Limitations

A significant limitation of this project stems from my status as a solo developer with limited specialized expertise in cybersecurity. While I possess fundamental knowledge of database management and programming principles, I lacked extensive experience in advanced security implementations and threat modeling that would typically be available in commercial solutions developed by dedicated security teams. This constraint may impact the robustness of security features implemented in the final product.

1.7.3 Delimitations

This project intentionally establishes specific boundaries to maintain focus and feasibility. While the underlying PostgreSQL technology supports network connectivity through libraries like psycopg2 when provided with appropriate credentials, this implementation deliberately concentrates on local database instances accessed from a trusted local environment. I decided to do this to avoid the significant complexity involved in developing secure networked database solutions.

1.8 Ethical Implications

A database program designed for local, secure data management inherently carries potential for misuse. While intended for legitimate users such as individuals or small groups requiring data privacy, the system, like many secure technologies, could be exploited for unethical or illegal purposes. Potential misuses range from storing pirated content and unlawfully acquired sensitive information to data supporting criminal activities.

A particularly significant ethical concern is the possibility of the program being repurposed as a component in ransomware attacks. The core encryption

functionality that provides legitimate users with data security could, if modified by malicious actors, be weaponized to encrypt databases without authorization. This dual-use potential creates an ethical tension between developing robust security features for legitimate users while acknowledging that these same capabilities could be exploited.

Beyond illegal content, users might misuse the system by exceeding its intended capabilities. Attempting to manage important business operations on a personal use system could lead to system overloads, performance issues, or significant data loss. Responsibility for data loss presents another substantial ethical consideration. The project assumes users will understand the risks of relying on a single computer for critical information and maintain regular backups. When users experience hardware failures, security breaches, or accidental deletions without proper backups, severe consequences can occur. The core ethical question becomes, how much responsibility does the developer bear in such scenarios?

As the developer of this program, I recognize my responsibility in protecting user data. By encrypting the data and storing keys securely, I provide a fundamental layer of protection. Transparency about key management is crucial, as these keys are essential for data access. However, my responsibilities extend beyond encryption. While users ultimately decide how to use the program, I designed safeguards that discourage misuse and protect user interests. For example, I implemented built-in safeguards like warning prompts that appear when users attempt to store large volumes of sensitive data. These proactive measures help guide users toward responsible data management while maintaining the program's core functionality and user autonomy. This means implementing clear usage guidelines, providing warnings about potential risks, and creating mechanisms that prevent obvious misuse. While no system can eliminate all risks, thoughtful design and informing users ensures they can operate the program responsibly and securely.

2 Related Work

2.1 Databases

Databases are fundamental tools for storing, managing, and accessing information. Functioning as advanced virtual filing cabinets, they enable efficient retrieval and organization of data, from simple lists to vast, interconnected information systems.

Although databases and spreadsheets may appear similar, they serve distinct purposes. As Oracle explains, "Databases are great for a single user or small number of users who don't need to do a lot of incredibly complicated data manipulation. Databases, on the other hand, are designed to hold much larger collections of organized information—massive amounts, sometimes" [17].

A defining feature of modern databases is their structured approach to data management. Most databases store information in rows and columns across multiple tables, optimizing data processing and querying. They typically rely on Structured Query Language (SQL) for data access and manipulation [2].

Beyond simple storage, databases offer critical functionalities such as enforcing data consistency, managing access permissions, and ensuring data integrity. These capabilities are essential for systems that handle sensitive or large-scale information, including those in banking, healthcare, and social media.

2.1.1 PostgreSQL

For this project, I chose PostgreSQL, a robust and widely-used relational database management system, due to its unique capabilities in handling structured data efficiently. According to the PostgreSQL Global Development Group, it is an open source relational database management system (RDBMS) that provides a powerful, flexible way to store and manage data in tables.

PostgreSQL was selected over MySQL primarily because of how well it handles write-heavy workloads and manages large databases. PostgreSQL is known for its strong emphasis on data integrity, which was crucial for this application. It uses Multi-Version Concurrency Control (MVCC) to manage simultaneous reads and writes, making it excellent for applications that deal with numerous concurrent transactions.

While MySQL is also a solid relational database especially in read-heavy environments it prioritizes speed and simplicity, sometimes at the cost of strict data integrity, particularly when using storage engines other than InnoDB. PostgreSQL, conversely, is more feature-rich and standards-compliant, with built-in support for custom data types [6]

Comparative Database Approaches:

1. Relational Databases (e.g., PostgreSQL)

- Strengths: Strong data integrity, complex querying capabilities
- Best for: Structured data with complex relationships
- Example use cases: Financial systems, enterprise applications

2. NoSQL Databases (e.g., DynamoDB)

- Strengths: Horizontal scalability, flexible schema
- Best for: Large-scale, distributed applications
- Example use cases: Real-time web applications, IoT data [2]

In terms of performance and scalability, PostgreSQL excels at handling complex queries and large datasets. It supports sharding and replication for horizontal scaling. MySQL handles read-heavy loads efficiently and performs well with simple queries, though its replication can introduce data consistency challenges.

PostgreSQL particularly impressed me with its remarkable flexibility in handling data. One feature I found especially useful was its variety of specialized data types beyond standard SQL. For example, PostgreSQL offers array types that can properly store multiple values within a single column. Rather than storing a comma-separated list like 'Ann,Jack,John' as text (which would violate normalization principles), PostgreSQL lets you use proper arrays such as ARRAY['Ann','Jack','John'] that maintain data integrity while allowing individual element access and indexing. This versatility, combined with its open-source nature and reliability, explains why PostgreSQL is widely adopted across industries, serving organizations from small startups to large enterprises in applications ranging from e-commerce to data analytics [22]

Key advantages that made me choose PostgreSQL include:

- Ability to handle structured data efficiently
- Seamless integration with Python
- Support for complex queries
- Advanced security features
- Flexibility through robust extension support

The database serves as the backbone of my project, acting as the primary mechanism for storing, managing, and securing user data. Its role is central, not only enabling the functionality of storing encrypted information but also protecting user privacy by ensuring data is securely managed during operations.

2.1.2 MySQL

While I did not use MySQL for my project, it's still a widely-used and reliable relational database, especially popular for web applications and read-heavy workloads. MySQL is known for its speed and simplicity, which makes it a great choice for many developers. However, for projects that involve complex queries, frequent writes, or need strong data integrity, PostgreSQL often performs better and offers more advanced features

2.2 Database Security Methods

While this project is developed by a single researcher with limited specialized cybersecurity expertise, it incorporates established database security principles

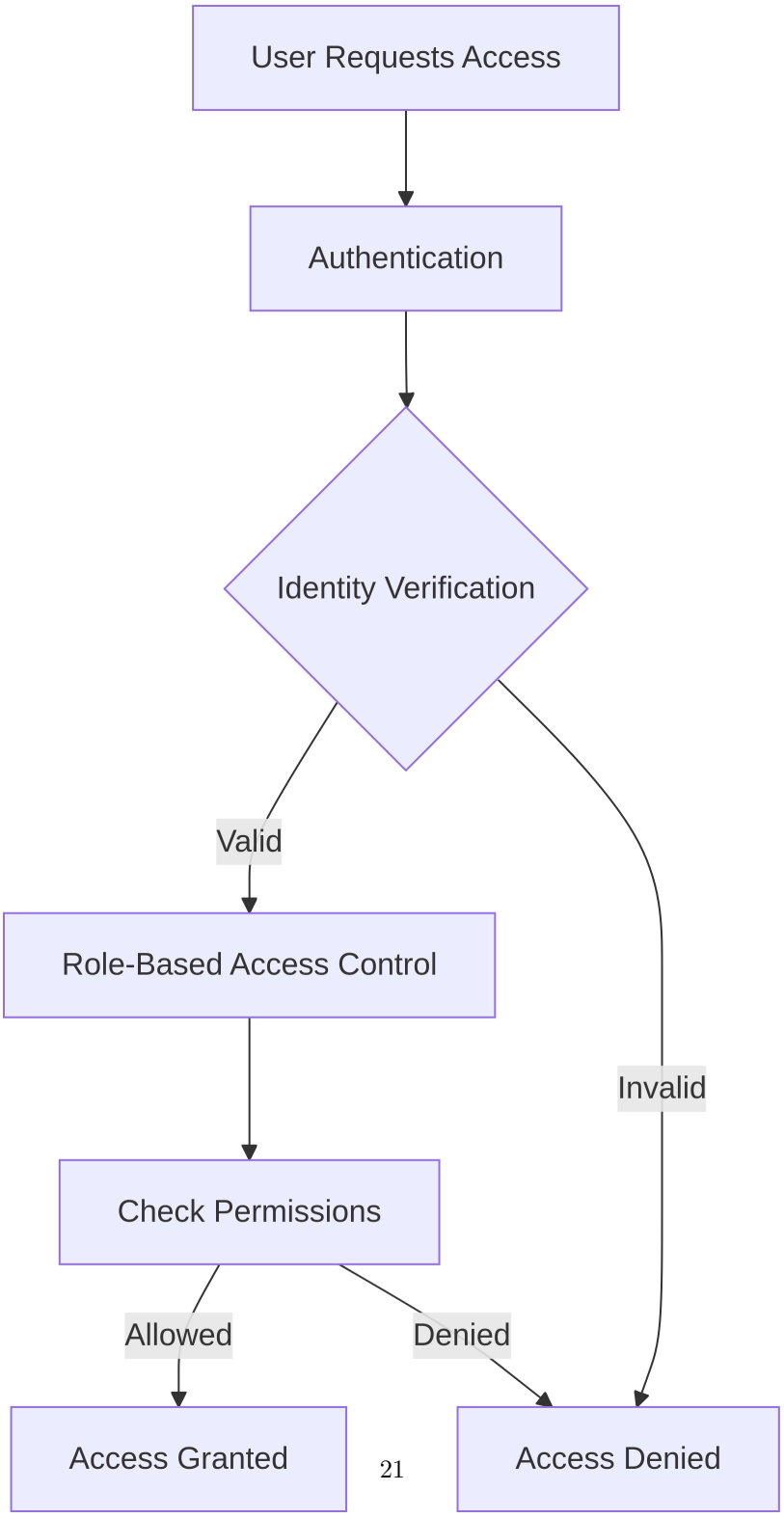
adapted to meet the needs of individual and small-group users. The following security methods represent foundational approaches that informed the development of this software. Although my implementation may not precisely replicate these enterprise-level techniques, I've adapted their core principles to create appropriate security measures within the project's scope and constraints.

2.2.1 Access Control Method

In recent years, various methods have been proposed to enhance database security through identity authentication and access control. One approach is the database security access control method based on identity authentication[15]. This method aims to establish trust relationships between users and databases through a trust transfer model and trust chains. According to the authors, "a trust chain consisting of guarantee nodes and recommendation nodes" is used to create direct trust relationships between users and databases through identity authentication.

The method leverages cloud service providers (CSPs) as trusted third parties to establish mutual trust relationships between different domains. The identity authentication gateway acts as a domain trust proxy, using gateway tokens as information credentials for trust transfer. This ensures secure transfer of user identities between different domains, thereby optimizing the identity authentication process. The authors state, "Gateway tokens are used as information credentials for trust transfer, thus achieving secure transfer of user identities between different domains".

Here is an example of an access control method:


User Requests Access
Authentication
Identity Verification
Valid
Invalid
Role-Based Access Control
Check Permissions
Allowed
Denied
Access Granted
Access Denied

By implementing this method, the system can accurately identify user identities and match them with the appropriate access permissions, thereby strengthening security control over database access. The authors highlight that this approach "can effectively ensure the security of database information" by maintaining a stable interception rate of over 98.0% for abnormal access. This method not only enhances security but also improves the efficiency of identity authentication for users accessing cloud services across domains.

While my program doesn't use an advanced identity authentication method like the one described, it follows a similar principle of restricting access to authorized users. Since my program is designed to run locally, the only way someone can access it is by physically accessing the laptop it is installed on. This requires having the device's password, ensuring that only the device's owner or someone they explicitly authorize can use the program.

2.2.2 Database Encryption

Ensuring database security is essential, especially in distributed and cloud-based environments. As cyber threats continue to evolve, modern database encryption techniques aim to protect sensitive data while maintaining performance and usability.

Researchers have proposed advanced encryption strategies that mitigate server-side data exposure by integrating SQL-based encryption modules and multi-layered encryption approaches. Their system ensures that only ciphertext is visible during potential breaches while maintaining query performance and application functionality. By incorporating robust access control mechanisms and developing language-specific encryption implementations, such as in Java, this method provides a practical solution for securing distributed databases. [27]

However, traditional encryption methods often focus solely on data-at-rest, leaving data-in-use and data-in-motion vulnerable to attacks. [21] tackle this issue by introducing a Transparent Data Encryption (TDE) model tailored for cloud-based Database-as-a-Service (DaaS) environments. Their approach encrypts data-at-rest, data-in-use, and, to some extent, data-in-motion using a proxy-based query transformation system. This system intercepts SQL queries, encrypts sensitive data before it reaches the database, and decrypts responses upon retrieval, ensuring the DBMS never directly accesses plaintext data.

Unlike conventional TDE solutions, which store encryption keys on the same system as the database, this method enhances security by ensuring that even malicious cloud administrators cannot access decrypted data. Additionally, it supports ACID-compliant transactions, allowing for efficient execution of complex queries—including joins and relational operations—without significantly impacting performance.

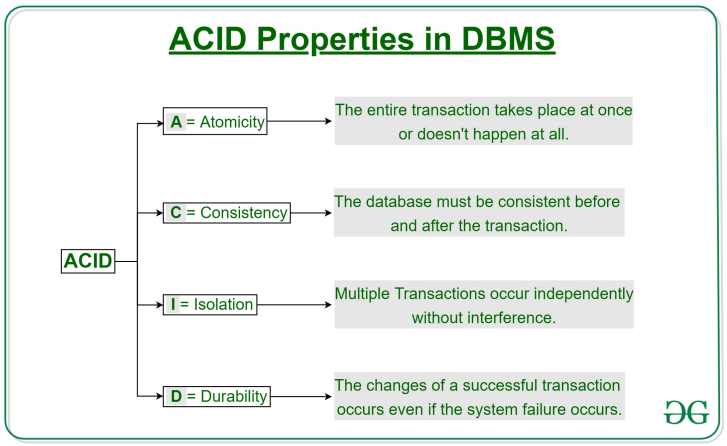


Figure 2: ACID Graph

[10]

The ACID properties maintain database integrity by enforcing atomicity, consistency, isolation, and durability [10]

By combining multi-layered encryption techniques [27]with proxy-based query encryption models [21], modern encryption strategies provide a comprehensive defense against database security threats, ensuring data protection across all operational states in distributed and cloud environments.

While my program does not utilize cloud services or complex encryption techniques like those described above, it still prioritizes data security. My program uses PyFernet to encrypt the entire database, ensuring that all sensitive information is protected. By doing so, it provides a practical and effective solution for users to secure their data, even in local environments.

2.2.3 Defense Against SQL Injections

SQL injection (SQLi), a prevalent and potent attack vector, targets the very foundation of database security by exploiting vulnerabilities in the way SQL queries are constructed and executed. Attackers insert malicious SQL commands, which can lead to unauthorized access to databases, data breaches, or even system control. SQLi attacks are significant security threats and have been a persistent issue for over twenty years, remaining in the OWASP Top 10 vulnerabilities. [14]

2.2.3.1 How SQL Injection Works: At its core, SQLi exploits improper handling of user input in database queries. Consider a simple SQL query used for user authentication:

```
SELECT * FROM users WHERE username = 'admin' AND password = 'password123';
```

If an attacker enters ' OR '1'='1 as the password, the query becomes:

```
SELECT * FROM users WHERE username = 'admin' AND password = '' OR '1'='1';
```

Since '1'='1' always evaluates to true, the attacker gains access without knowing the actual password.

2.2.3.2 Types of SQL Injection

- Classic SQL Injection (In-band SQLi) – The attacker directly manipulates SQL queries to retrieve, alter, or delete data. This is the most common and easiest to exploit.
- Blind SQL Injection – The attacker cannot see the database's response but can infer information based on application behavior (e.g., true/false responses or error messages).
- Boolean-based SQLi – Attackers use logical conditions (AND 1=1, AND 1=2) to determine if a query is valid, allowing them to extract data without visible outputs.
- Time-based SQLi – Attackers use database functions like SLEEP() to cause delays, allowing them to deduce whether a SQL statement is valid based on response time.
- Out-of-band SQLi – Attackers use alternate channels (like sending data via DNS requests) when direct database responses aren't available.

Real-World SQL Injection Attacks

- Yahoo SQLi Attack (2012) – Hackers exploited an SQLi vulnerability to steal 450,000 email addresses and passwords.
- Sony PlayStation Network (2011) – Attackers used SQLi to access 77 million user accounts, leading to one of the largest data breaches in gaming history.
- Heartland Payment Systems (2008) – A massive SQLi breach exposed 134 million credit card numbers, leading to significant financial losses.

2.2.3.3 The Impact of SQL Injection SQLi remains one of the most dangerous cyber threats due to its potential consequences:

Data Breaches – Attackers can exfiltrate personal information, credentials, and financial data.

Data Manipulation – Unauthorized modifications to records can disrupt business operations.

Privilege Escalation – Attackers may gain administrator-level control over a system.

Complete System Compromise – In severe cases, SQLi can be used to execute arbitrary system commands, leading to full server takeover.

The adaptable nature of SQL syntax presents a big challenge in covering all potential SQL injection (SQLi) scenarios, leading to diminished detection efficacy. One primary cause of failure in SQLi detection models is the absence of thorough testing before deployment, which prevents developers from fully understanding the weaknesses of their solutions. Also, malicious actors continuously evolve their techniques to create more sophisticated and evasive SQLi samples, exploiting weaknesses in machine learning (ML)-driven or signature-based Web Application Firewalls (WAFs). They use methods such as obfuscation and mutation to alter the structure and appearance of attacks, bypassing signature detection. Furthermore, attackers employ adversarial machine learning to craft inputs specifically designed to evade ML models, exploiting model overfitting and iteratively refining their techniques based on WAF responses. [14]

While malicious actors are developing ways to find new ways to SQLi, researchers have come up with ways to prevent the attacks. According to the research paper. Researchers have proposed SQLStateGuard. "SQLStateGuard is a custom SQL middleware based on the concept of Runtime Application Self-Protection (RASP) to capture and analyze raw SQL statements. The system employs SQLSG-Net, a database-oriented detection network based on gated linear units, to identify and block malicious SQL statements." [14]

SQLStateGuard protects against SQL injection (SQLi) attacks by combining custom middleware and a detection system called SQLSG-Net. The middleware intercepts SQL commands before they reach the database, acting as a go-between for the application and the database. SQLSG-Net is a deep learning-based system that analyzes these intercepted SQL commands using Natural Language Processing (NLP) to understand their structure and meaning. It has three main parts. A layer that simplifies the structure of SQL commands, another that looks at their meaning and context, and a final layer that uses advanced techniques to detect and classify SQLi attacks. SQLStateGuard can work in two ways, Blocking Mode, which stops harmful SQL commands in real-time, and Bypass Mode, which allows commands to proceed to the database while checking them and alerting security if anything harmful is found. This system improves accuracy, reduces false alarms, and provides strong protection against SQLi threats. [14]

Although my program does not include advanced SQLi prevention systems like SQLStateGuard, it still considers the threat of SQL injection in its overall design. Because my program is built for use in a local, personal environment

rather than a publicly exposed web server, the risk of external SQLi attacks is significantly reduced. However, if the tool is adapted for broader use or connected to external input sources, SQLi becomes a more relevant concern. For this reason, I structured the database interactions in a way that can be easily modified to include security best practices like input validation and query parameterization. This ensures that users who choose to expand or adapt the project can build on a secure foundation.

2.3 Similar Software

When designing this project, I looked at existing tools and systems that aim to provide encryption or secure data storage. While many of them are useful, they often come with limitations that this project tries to address more flexibly or simply.

2.3.0.1 Similar Programs

- BitLocker: While BitLocker is useful for full-disk encryption, it often ends up encrypting much more than you actually need like system files, temporary files, or unused data. It's not designed for selectively encrypting user data at the database level.
- VeraCrypt: VeraCrypt is great for encrypting containers or external drives, but it's not well-suited for use in a dynamic database environment. It's more for static or file-based encryption, not real-time insert/select operations like mine.
- SQLCipher: SQLCipher is a full-database encryption extension for SQLite. It offers strong AES encryption, but the setup process can be complicated, especially for new users. Also, some features and support fall under paid or commercial licensing options.

2.3.0.2 Database Engines with Encryption Features

- MySQL (InnoDB Encryption): InnoDB supports tablespace encryption, which protects data at rest. However, encryption keys are managed at the server level and not on a per-row or per-column basis, which limits flexibility. It's also more useful for general database security rather than granular, user-specific protection.
- PostgreSQL (pgcrypto extension): PostgreSQL supports column-level encryption through the pgcrypto extension. While it's a solid option, it can be somewhat manual to implement and requires careful key management from the developer.
- Microsoft SQL Server (TDE): Transparent Data Encryption (TDE) in SQL Server provides automatic encryption at the file level. Like BitLocker, it secures the entire database rather than specific pieces of data. That

makes it easy to implement but not ideal when more selective encryption is needed.

2.3.0.3 Limitations in Database-Level Encryption Tools

- Berkeley DB: While it offers encryption, Berkeley DB doesn't allow databases to be converted into an encrypted format after creation. To secure an existing database, you have to dump and recreate it, which can be a hassle and time-consuming.

2.4 Database Testing

It's important to set up strong security, but it's just as important to test that it actually works. Database testing isn't just about checking security features; it also involves making sure the database works well, is reliable, and performs properly in different situations. Just like protecting data is key, ensuring the database runs smoothly and keeps data safe under all conditions is equally important.

Key aspects of database testing, including functional validation, performance benchmarking, security assessments, and concurrency testing. Functional testing ensures that basic database operations, such as inserting, retrieving, updating, and deleting data, work as expected. It also checks that constraints, triggers, and transactions maintain data integrity. Without this type of testing, errors and inconsistencies could arise, leading to serious problems in applications that rely on the database.

Performance benchmarking is another crucial aspect of database testing. A study by Salunke and Ouda [20] compared the performance of PostgreSQL and MySQL using real-world test scenarios. Their research found that PostgreSQL was significantly faster in executing select queries, making it a more efficient choice for applications that require quick data retrieval. For example, when retrieving one million records, PostgreSQL completed the query in approximately 0.6 to 0.8 milliseconds, whereas MySQL took between 9 and 12 milliseconds. Similarly, select queries with conditions showed PostgreSQL to be about nine times more efficient than MySQL [20]. This highlights the importance of testing a database's speed and responsiveness before deployment.

Beyond performance, security testing ensures databases are protected from potential threats such as SQL injection, unauthorized access, and data breaches. Proper authentication mechanisms, encryption methods, and role-based access controls help keep sensitive information secure. However, security must be tested under various conditions to ensure that vulnerabilities do not emerge as the database scales.

Finally, concurrency and load testing evaluate how well a database handles multiple simultaneous requests. In real-world applications, databases often experience heavy traffic, requiring them to process numerous queries at once. Salunke and Ouda's [20] research demonstrated that MySQL's performance degraded significantly when handling concurrent operations, while PostgreSQL

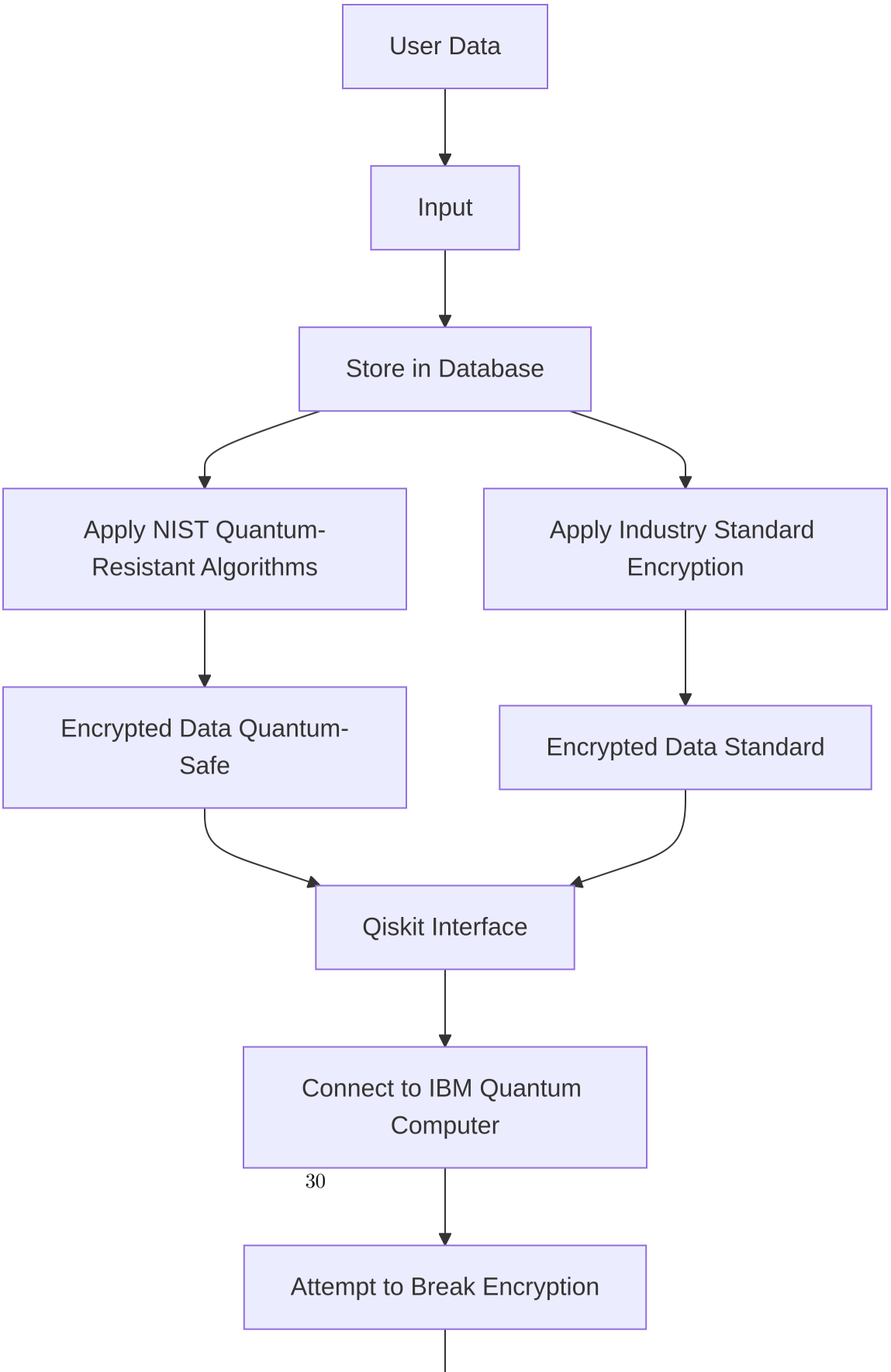
maintained stable query execution times. This suggests that PostgreSQL is better suited for environments where many users access and modify data at the same time.

3 Method of Approach

The goal of this project is to make a program that enables individuals or small groups to securely store and manage their data, free from the vulnerabilities inherent in third-party data storage solutions. My first thought was "How can I protect the users data?" This led me to looking into encryption. Encryption is a crucial aspect of data security; it is a strong method to safeguard sensitive information from unauthorized access. By converting data into an unreadable coded format, encryption ensures that only those with the correct decryption key can access and understand the information. In addition to individual privacy, encryption plays a significant role in maintaining the integrity and security of data. By adopting strong encryption practices, organizations and individuals can significantly enhance their overall protection.

With encryption being so important to my goal and project, I looked for the best encryption available. That's when I learned about quantum encryption. What drew me towards quantum encryption is the fact that modern day encryption will eventually be able to be cracked by quantum computers. This is because quantum computers have the potential to break many standard encryption methods because they can efficiently solve certain mathematical problems that classical computers find extremely difficult. While quantum computers are still new and developing, the threat of them cracking regular encryption is still real. That is why I wanted to find a way to use quantum resistant encryption methods for my project. That way I would be able to theoretically future proof my project. Ensuring that the users data would be able to be shielded against quantum attacks. To implement the quantum resistant algorithms, I looked at algorithms recommended by the National Institute of Standards and Technology (NIST).

With these algorithms I planned to use them to encrypt the data inside the users database. Once I had the data encrypted, I would then use a programming language called qiskit created by IBM to connect to their super computer and try to use it to see if the computer could break the encryption and how long it would take. As a comparison, I would then encrypt the same data inside the database with modern day industry standard encryption and again use IBM's quantum computer to try and crack the encryption.


User Data
Input
Store in Database
Apply NIST Quantum-Resistant Algorithms
Apply Industry Standard Encryption
Encrypted Data Quantum-Safe
Encrypted Data Standard
Qiskit Interface
Connect to IBM Quantum Computer
Attempt to Break Encryption

There are multiple reasons why I had to give up on this approach. The first reason I gave up on this approach is because of the complexity. Quantum computing is still a relatively new field. While there is work being done with and about quantum computing, there is not a lot compared to other topics in the computer science field. If I were to go with this approach, I would have to really dig for valuable sources. Another roadblock is understanding the math behind quantum computing. I would have really struggled to understand the mathematical concepts on a low level, this would've caused me some trouble with my related works section. If I didn't understand what was going on, there would be no possible way I could write about it. The second issue is that I would not have help if I were to go this route. While the professors would've done their best to help me, they don't know much about quantum computing either so I wouldn't be able to go to them for help. The third reason I had to change was because I would heavily rely on IBM's quantum computer and its programming language. If there was an issue with the quantum computer I would not be able to run my project. The other issue is the programming language qiskit. I would have to learn qiskit which would've been complicated because the language was constantly being updated. Some of the old guides I found were outdated and did not work anymore.

While I really wanted to focus on quantum encryption I could not in the end. I was still very interested in encryption so I decided to pivot.

Instead of exploring quantum encryption, I focused on implementing standard encryption techniques to protect data within database systems. This approach still fulfills my original goal of helping individuals and small groups securely store and manage their information independently. With this new approach decided on, I then had to figure out how exactly I was going to encrypt the database. My original plan was to encrypt the data as it was being entered. For example, a insert query might look like

```
INSERT INTO employees (
  employee_id,
  first_name,
  last_name,
  email,
  hire_date,
  job_id,
  salary
) VALUES (
  101,
  'John',
  'Doe',
  'john.doe@example.com',
  '2025-02-21',
  'IT_PROG',
  75000
```

```
);
```

my program would then change the data to look something like this

```
VALUES (
  101,
  'J3hnD0e@eX4mpl#',
  'D03@eX4mpl3',
  'j0hn.d0e@3x4mple.c0m',
  '2O2%5-02-2!',
  '1T_PR0G!',
  '75kZ@123'
);
```

While this exact example isn't accurate, it shows what the result would look like. The issue I ran into was conflicting data types. At the time I did not realize that encrypting the data would change the datas data type. So a string, int would be turned into a byteA. This caused issues if a certain column was expecting a certain kind of datatype. Like the date column would be expecting a date but instead it got bytea. This would result in a confusing error that would take me hours to figure out what was causing it. Eventually I decided to switch how I would do the encryption. Instead of encrypting when the data is inserted. I would instead encrypt the entire database when the user is done with the program. That way I won't run into type issues, and when the user wants to access the database, it will be decrypted for them.

Throughout this project, I prioritized finding the optimal balance between security and performance. Because encryption and decryption operations can be computationally expensive, my central challenge became developing a solution that provides strong security while maintaining speed and usability for everyday applications. In order to achieve this goal, I had to plan out how I wanted to structure my program

3.1 System Architecture

My system was designed with three primary components:

- Database Layer: PostgreSQL database system for data storage and retrieval

I wanted to use PostgreSQL because PostgreSQL is good at handling datasets. Since the tool I'm making is for small groups of people like students, small businesses, or individuals to securely store their data in. I figured that PostgreSQL would be the better suited database program. Another reason why I used PostgreSQL is because it has a Python module named Psycopg. Psycopg is a Python module that allows Python code to interact with a PostgreSQL database, letting me execute SQL queries, fetch results,

and manage interactions seamlessly. Psycopg made building the database layer simple and easy. With its easy to understand documentation and my knowledge of Python, I was able to create the database layer.

PostgreSQL also offers many advantages that made it the right choice for my project. It has strong ACID compliance (Atomicity, Consistency, Isolation, Durability), which ensures data reliability even if system failures occur [9]. This was really important for a security-focused application where data integrity cannot be compromised. Additionally, PostgreSQL's ability to handle complex queries and its support for JSON data types gives my application room to grow if users need to store more complex data structures in the future. The active community around PostgreSQL means there are plenty of resources available if I run into problems during development or when adding new features later on.

I also considered alternatives like MySQL and SQLite before deciding on PostgreSQL. While MySQL is popular and fast, I found PostgreSQL's stronger emphasis on standards compliance and data integrity better aligned with my project goals. SQLite would have been simpler to set up, but its limitations with concurrent access would potentially create problems as the user base grows. For these reasons, PostgreSQL emerged as the most balanced choice that offered the right mix of features, reliability, and ease of integration with Python.

- Encryption Layer: Custom encryption implementation for data security flexibility

The encryption layer utilizes PyFernet, which implements AES (Advanced Encryption Standard) encryption, widely regarded as one of the most secure encryption methods available.

Fernet was specifically chosen because it's an industry standard implementation that provides symmetric encryption. The symmetric nature of Fernet means it uses the same key for both encryption and decryption, making it particularly suitable for

programs where the same entity needs to both protect and access the data. Fernet was a good choice for me because it gave me an easy and convenient way to encrypt the data while ensuring proper implementation of cryptographic standards.

This encryption method ensures that data remains secure, using a 256-bit encryption key. This is a big key size but it guarantees that the data will be protected against brute force attacks and other common cryptographic threats.

Even with the rise of quantum computing, a key sizeof 256 bits provides future-proofing. Quantum computers will be able to break it but with a key this size there will be enough time for more quantum resistant encryption algorithms to be made.

PyFernet might not offer the fastest encryption speeds,because it has to go through a few steps before the actual encryption takes place [5] and with the big built-in key size compared to simpler alternatives. If I were to just use raw AES from libraries like cryptography or pycryptodome to implement my I would be able to make my own encryption scheme [18]. I am not yet confident in my

ability to make a strong encryption scheme so I went with Pyfernet because it provided a simple implementation.

My work will be open source, which means if someone wanted to change the encryption method they would easily be able to implement their own method. This flexibility allows users to adapt the security level to their specific needs, whether they require faster performance with slightly reduced security or even stronger encryption at the cost of speed. The modular design means that switching encryption methods won't require rebuilding the entire system, just modifying the encryption layer while maintaining the same interface with the rest of the application.

The implementation also includes proper key management practices, ensuring that encryption keys are handled securely throughout their lifecycle. This was another advantage of using Fernet, as it includes built-in key rotation capabilities and secure key generation functions, reducing the risk of key-related vulnerabilities that often plague custom encryption implementations.

- Interface Layer: Python-based management system for database operations

Python was chosen as the programming language for this project due to its versatility and ease of use. The interface layer is a very simple terminal like a python program. Python is the backbone of the interface layer, handling database operations and query execution. Its simplicity and extensive library support made it an ideal choice for managing interactions with the database efficiently. The interface includes essential features such as:

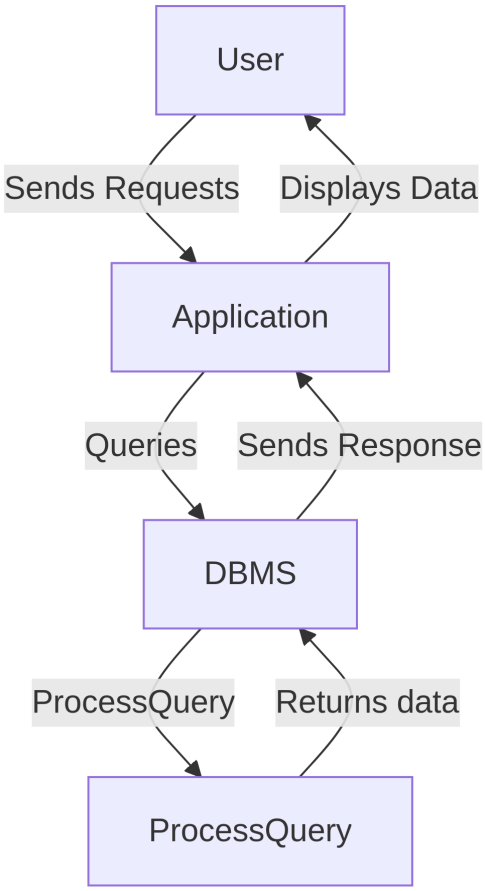
- User authentication and connection management
- Automated query processing and execution
- Error handling and validation
- Performance monitoring and timing measurements
- Encryption/decryption operation management

The user interface follows a command-line approach which keeps things simple while still providing all necessary functionality. I decided against creating a graphical interface because command-line tools are often preferred by technical users who value efficiency and scriptability. The commands are designed to be intuitive and similar to standard SQL syntax, which reduces the learning curve for users already familiar with database operations. For example, users can type commands like "CREATE TABLE users (name TEXT, age INTEGER)" which gets processed by the interface layer, encrypted if necessary, and then passed to the database.

Error handling was a major focus during interface development, as cryptographic operations can fail in various ways that might be confusing to users. The interface provides clear, actionable error messages rather than exposing

the raw technical details. For instance, instead of showing a cryptographic error like "Invalid padding," the system would display "Unable to decrypt data - password may be incorrect." This human-readable approach helps users troubleshoot problems without needing to understand the underlying encryption mechanisms.

Here is a graph visualizing an example of my system architecture



3.1.1 Security Implementation

3.2 Security Implementation Overview

The security system I've built operates on two fundamental principles: comprehensive database protection through full encryption and secure key management using platform-native security mechanisms.

When a user finishes their session, the entire PostgreSQL database gets encrypted as a single unit rather than encrypting individual tables or fields. This approach protects not just the data itself, but also the database structure and

access patterns that might otherwise reveal sensitive information to attackers. By encrypting only when the user is done working, the system maintains good performance during active use while ensuring complete protection at rest.

The encryption keys are never stored alongside the database but are instead managed through the operating system's secure keyring facility. This creates a crucial security boundary - even if someone obtains the encrypted database files, they remain useless without the keys. The system keyring (Windows Credential Manager, macOS Keychain, or Linux equivalents like GNOME Keyring) provides platform-specific security that leverages each operating system's authentication mechanisms and security updates.

This two-layer approach means an attacker would need both the encrypted database and physical access to the authenticated user's device to compromise the data. The architecture follows the security principle of separation - keeping the thing being protected (the database) separate from the means of access (the keys), with each secured by different mechanisms. Even in the event of a severe security breach where an attacker gains physical access to the device and user credentials, the encryption buys critical time for the breach to be detected and addressed.

The system works seamlessly across different operating systems without requiring users to understand the underlying security implementations, making strong security accessible to small businesses, students, and individuals who need to protect their data without specialized IT knowledge.

The security system operates on two fundamental principles: full-database encryption for comprehensive protection and secure key management to ensure that encryption keys remain inaccessible to unauthorized users.

When a user finishes their session, the entire PostgreSQL database is encrypted as a single unit rather than encrypting individual tables or fields. This ensures that all data remains protected, preventing attackers from inferring sensitive information based on database structure. By encrypting only at the end of a session, the system maintains optimal performance during active use while ensuring complete security at rest.

However, encryption is only as strong as its key management. To prevent unauthorized access, encryption keys are never stored alongside the database. Instead, they are secured using the operating system's native key management system, such as Windows Credential Manager, macOS Keychain, or Linux equivalents like GNOME Keyring. This approach enforces strict separation between encrypted data and the means to decrypt it, ensuring that even if an attacker obtains the database files, they remain useless without access to the keyring.

This layered security model follows the principle of separation—keeping the protected database independent from the means of accessing it. Even in a worst-case scenario where an attacker gains access to the database the encryption will prevent them from accessing it unless they know where to find the keys.

A key component of this system is full-database encryption, which, combined with strong key management, protects data from unauthorized access and forensic analysis.

3.3 Full Database encryption for comprehensive data protection:

While PostgreSQL does have built in encryption features like column level encryption and even network encryption

The system encrypts the entire PostgreSQL database once the user is finished using it. It avoids performance slowdowns during active usage. Unlike table-level encryption, which only protects specific data fields, full database encryption secures everything, preventing attackers from analyzing database structure and access patterns. This approach also simplifies security management by using a single encryption process rather than requiring encryption every time data is entered reducing the complexity and minimizing the risk of type errors. Additionally, encrypting the entire database makes it safer to transfer or back up, as the data remains protected even if intercepted or stored in a remote location. Without the decryption key, unauthorized users, including attackers with database credentials, cannot read or interact with the encrypted database, making this method effective in preventing unauthorized access. Furthermore, full encryption defends against forensic analysis, ensuring that even if an attacker extracts raw database files from disk, they cannot recover any meaningful data. By encrypting the entire database only when the user is done, this approach strikes a balance between security and efficiency, safeguarding data without compromising performance during active use.

While encryption tools already exist for PostgreSQL, they often require complex configuration, third-party extensions, or manual encryption logic for each database operation. This increases setup time and the chance of implementation errors. My solution is intentionally minimal and designed for ease of use, it requires no database configuration or extension setup. The encryption and decryption are handled externally and automatically, making it accessible even for users without deep PostgreSQL or cryptography experience. By encrypting the entire database only when the user is done, this approach strikes a balance between security and efficiency, safeguarding data without compromising performance during active use.

3.4 Secure key management system:

Key security is managed through the system keyring, a secure platform-specific storage solution that enhances protection against unauthorized access and key extraction attacks. By storing encryption keys separately from the database and application code, this approach ensures that even if the database is compromised, the encryption keys remain secure. The system keyring leverages the operating system's native security mechanisms, meaning access to stored keys is governed by system-level policies, permissions, and user authentication requirements. This adds an additional layer of protection, as accessing the keys requires both physical access to the machine and valid user credentials. Furthermore, the keyring automatically isolates encryption keys from application memory, reducing the risk of exposure through runtime attacks such as memory

scraping. By integrating directly with the OS's security framework, the keyring also benefits from regular security updates and built-in protections, minimizing vulnerabilities. This method not only enhances security but also ensures compliance with best practices for cryptographic key management, providing a reliable and scalable solution for safeguarding encrypted databases. The system keyring provides cross-platform compatibility, ensuring seamless key management across Windows, macOS, and Linux environments. This universal approach eliminates the need for users to understand platform-specific security implementations, as the application automatically leverages each operating system's native secure storage mechanism.

While this architecture offers strong protection for encryption keys, it's important to acknowledge its security boundary. If an attacker gains access to the system's secure storage, requiring both physical access to the machine and valid user credentials, they could potentially retrieve the keys and decrypt the database. This represents an inherent limitation in any client side encryption system rather than a weakness in the implementation itself.

By utilizing the operating system's secure storage, the application follows security best practices by delegating credential protection to specialized, well-maintained system components designed specifically for that purpose. This approach significantly raises the barrier for potential attackers, as they would need to compromise both the physical security of the device and the user's authentication credentials, a scenario that indicates a severe security breach beyond the scope of what application level protections can mitigate. In such extreme circumstances, the database encryption still provides critical time protection, giving the users an opportunity to detect the breach and take appropriate countermeasures.

3.5 Plan to Measure My Experiment

To assess the performance of my database encryption system, I conducted a doubling experiment. This methodology involved progressively doubling the dataset size in each iteration while measuring key performance metrics. By analyzing how computational resources scaled with growing workloads, I determined that my system exhibited linear growth patterns in terms of performance efficiency.

The experiment began with a baseline dataset size, a small but representative sample of database records. From there, I doubled the dataset size in each subsequent trial, encrypting and decrypting the database at each step. This process continued until the performance reached a practical limit.

The key performance metrics I measured included:

Encryption and decryption time – To assess how long the system takes to secure and retrieve data as the database grows.

Database operation throughput – To evaluate how encryption impacts the speed of typical database operations, such as inserts, updates, and queries.

Memory usage – To understand how system resource consumption scales with increasing data volumes.

By plotting these measurements against dataset size, I aimed to determine the computational complexity of the encryption process in practical scenarios. If performance scales linearly, it suggests that encryption time and resource usage grow predictably with database size. If it follows logarithmic or sublinear growth, the system may be optimized for handling larger datasets efficiently. However, if exponential or superlinear growth is observed, it could indicate potential bottlenecks that may hinder scalability.

Ultimately, the insights gained from this doubling experiment helped guide optimization strategies, ensuring that the encryption system remained effective across different usage scenarios from individual users with moderate data needs to small businesses handling larger datasets. The findings informed potential refinements, such as changing encryption methods or key size.

4 Experiment

4.1 Experimental Design Background

The experimental evaluation was designed to assess both the performance and scalability of my encrypted database system using a systematic doubling experiment.

When designing my experiment, I wanted it to be realistic while ensuring I properly tested my system. I considered three key factors: who would be using my program, what kind of data they would store, and how much data they would input. My goal is to create a database that allows individuals or small groups to manage their data securely and efficiently.

Initially, I planned to use an LLM to generate insert queries, which would have saved time and let me focus on testing. However, after generating a few hundred queries, I started noticing inconsistencies, some queries contained errors that caused my program to crash. To avoid this, I decided to write my own program to generate queries, ensuring they were accurate and error-free. My program would generate three types of queries.

The first type of query would be customer information, Its an insert statement with randomly generated data

```
f"""
INSERT INTO Customers (
  name,
  email,
  phone,
  address
) VALUES (
  {name},
  {email},
  {phone},
  {address}
);
```

The second type of query is the order queries. It is also an insert statement with randomly generated data

```
f"""
INSERT INTO Orders (
  order_id,
  total_amount,
  status
) VALUES (
  {id},
  {total_amount},
  {status}
);
```

Lastly, the third type of query is the product queries. It is also an insert statement with randomly generated data

```
f"""
INSERT INTO Products (
  name,
  price,
  stock
) VALUES (
  {product_name},
  {price},
  {stock}
);
```

Once I had a reliable way to generate queries, I focused on designing the experiment. In my related works section, I explored database benchmarking methods. Inspired by existing research, I decided to benchmark my system by measuring encryption and decryption speed. To do this, I modified my code to include timing functions that accurately track performance.

I designed a doubling experiment to stress test my program and analyze how encryption and decryption performance scales with increasing data sizes. This method effectively measures performance scaling by repeatedly doubling the amount of inserted data, making it possible to observe trends and identify potential bottlenecks. My testing process consists of two distinct benchmarking approaches: the Multiple Database Benchmark, which encrypts and decrypts multiple large databases, and the Single Database Benchmark, which performs the same operations on one large database. By comparing these two methods, I aim to determine whether my program is efficient enough to handle encrypting and decrypting large amounts of data at a time. By stress testing my program like this I will be able to figure out if my encryption methods are reliable. The reason why I decided to decrypt one large database and 3 databases of the same size is to simulate more probable use cases of my program. I can imagine that some people might just use one database while others might have multiple.

To conduct these experiments, I used Python's time module to measure encryption and decryption performance. The timer starts when the decryption process begins and stops when it completes, capturing the exact duration required for decryption. Similarly, encryption time is recorded once the data has been inserted into the database and the encryption process is triggered. My experiment for the single big database follows a simple procedure that ensures consistency and repeatability. First, I insert 1,250 queries into the database, representing a controlled initial dataset. Once the data is inserted, the encryption process begins, and I record the encryption time. Following encryption, I decrypt the database and measure the time taken to restore the original data. After completing this cycle, I drop the tables and create new ones to reset the database environment before proceeding to the next stage. The number of inserted queries is then doubled, meaning the next iteration involves 2,500

queries, followed by 5,000, 10,000, and finally 20,000 queries. This structured doubling approach allows me to track performance trends and observe whether the relationship between encryption/decryption time and data size is linear, exponential, or follows another pattern.

The procedure is similar for the three databases, but instead of inserting a total of 20,000 combined order, customer, and product queries. I instead made three separate tables for each type of queries and put 20,000 queries worth of data into each. Then I would follow the process and encrypt and decrypt all of them then record the times.

By using pre-generated queries, I maintain consistency across test runs, reducing variability caused by differing database operations. The structured approach of inserting data, encrypting, decrypting, and repeating the process with an increasing dataset provides a clear picture of how my encryption implementation scales. This experiment is critical in assessing whether my encryption system remains efficient as database size grows, helping determine if it is a viable solution for real-world applications that require secure storage and retrieval of encrypted data.

Once I had my experiment designed, I needed to determine the best way to present my results in a clear and meaningful way. At first, I considered using a Markdown table to display the data. Markdown tables are simple to create, easy to read, and widely supported across different platforms. There are even tools like tablesgenerator.com that can automatically format Markdown tables, making it a convenient option. I would simply need to input my data, generate the table, and copy it into my report. However, I quickly realized that this method had significant drawbacks. A table, while structured, would not be an effective way to highlight the trends in my data, especially when comparing encryption and decryption times across different setups. The number of decimal places in my dataset would also make the table difficult to read and align neatly, leading to potential confusion when analyzing the results. Given these limitations, I decided to explore a more visual approach to data presentation.

This semester, I took a course called Data Exploration, where I was introduced to Matplotlib, a powerful Python library for data visualization. This experience influenced my decision to use Matplotlib for my project, as it allows for clear, customizable graphs that make it easier to see patterns in data. To generate the graphs, I first needed to organize my raw data into CSV format, ensuring that each dataset was properly structured for easy processing. Once the data was formatted correctly, I used a Google Colab notebook to write a script that would generate my graphs. Google Colab provided a convenient environment for running Python code without needing to install anything locally, making the process smoother. After experimenting with different types of graphs, I decided on a line graph because it effectively shows how encryption and decryption times change as the dataset size doubles. A line graph allows for easy comparison between the different setups, making trends more apparent than a static table would. By visualizing my results in this way, I was able to present my findings in a way that is both intuitive and informative. In the next

section, I will discuss the insights gained from these visualizations.

while the absolute worst-case time complexity of my encryption and decryption system could theoretically be exponential — for example, if something caused recursive reprocessing or infinite loops in data parsing — this would likely only occur due to severe flaws or misuse. In realistic use cases, the time complexity is closer to O(n log n). This is because encryption operations scale linearly with the size of the data (O(n)), but additional overhead such as key management, logging, or data lookups introduces a logarithmic component. As a result, O(n log n) reflects a more accurate picture of how performance scales in practice, particularly when working with structured data and modern encryption libraries that are optimized for performance.

4.2 Evaluation

In this section, I analyze the performance results of the encryption and decryption operations across different database configurations. After collecting timing data from multiple experimental runs, I organized the results into a structured format to facilitate analysis.

The data was recorded in the following format:

```
experiment,run,double_size,operation,time,num_lines
```

This organization allowed me to easily import the data for visualization and further analysis. By structuring the data this way, I could examine how encryption and decryption times scale with increasing data size across different experimental configurations.

Here an example of the raw data:

This data is the time it took for my program to encrypt and decrypt the queries. The first experiment is a single database. The second experiment is with 3 databases

Experiment Num, Run, double size, decrypt time, # of lines, encrypt time, # of lines

```
1,1,1,Encrypt,1.03,1250
1,1,2,Encrypt,2.30,2500
1,1,3,Encrypt,5.14,5000
1,1,4,Encrypt,10.73,10000
1,1,5,Encrypt,20.93,20000
```

4.3 Exploring The Raw Data

When just looking at the raw data you can see huge jumps in the time it takes for my program to run for each operation, especially around run 4 and 5. As the number of lines increases, processing times grow in an linear manner, with particularly dramatic time jumps occurring around 10,000 to 20,000 lines. This suggests that my program has exponential time complexity. It doesn't double every run but there is no exponential increase.

In order to figure out the time complexity I had to find the doubling ratio. A doubling ratio is a method for analyzing an algorithm's time complexity by looking at how the run time changes when the input size doubles. A doubling ratio is calculated by dividing the execution time of an algorithm with input size 2n by the execution time with input size n:

Doubling Ratio = T(2n) / T(n)

Where:

T(n) is the time taken to run the algorithm with input size n

T(2n) is the time taken to run the algorithm with input size 2n

To do this I had to calculate the doubling ratio of each run, I used a python function to do that.

```
def compute_doubling_ratios(rows, times):
    doubling_ratios = []

    # Calculate each doubling ratio by looping through the times
    for i in range(1, len(times)):
        # Divide current time by previous time to get the ratio
        ratio = times[i] / times[i-1]
        # adds it to the list
        doubling_ratios.append(ratio)

    # Print each doubling ratio
    for i in range(len(doubling_ratios)):
        print(f"Doubling {rows[i]} → {rows[i+1]}: {doubling_ratios[i]:.2f}x")

# Example data for a single run
rows = [1250, 2500, 5000, 10000, 20000]
# the time for each input double in a run
#
times = [1.188982, 2.215667, 4.566037, 8.473383, 21.453281]
```

2.215667/ 1.188982 = dr for run 1

The rows list represents the input sizes I tested: [1250, 2500, 5000, 10000, 20000]. The times list contains the time it took for each input size in a single run. I used these times to calculate the doubling ratios—for example, dividing the time at 2500 by the time at 1250 (2.215667/ 1.188982 = doubling ratio for run 1), and so on. After calculating the ratios for one run, I moved on to the next run and repeated the process. Once I had all the doubling ratios from each run, I grouped them by the input size range they came from (like all the ratios from 1250 → 2500). Then, I calculated the average of each group of ratios to see how the performance scaled as the input size doubled.

Here are the averages for each database

Doubling Step	Run 1	Run 2	Run 3	Run 4	Run 5	**Average**
1250 → 2500	1.86	2.61	2.18	2.52	1.84	**2.20**
2500 → 5000	2.06	1.85	1.70	2.15	1.98	**1.95**
5000 → 10000	1.86	1.87	2.15	1.88	1.53	**1.86**
10000 → 20000	2.53	2.18	2.05	1.71	2.58	**2.21**

One database (encryption times)

Doubling Step	Run 1	Run 2	Run 3	Run 4	Run 5	**Average**
1250 → 2500	2.22	2.27	2.27	1.34	1.98	**2.02**
2500 → 5000	2.23	2.21	1.77	2.53	1.95	**2.14**
5000 → 10000	1.91	2.00	2.12	2.09	1.42	**1.91**
10000 → 20000	2.14	1.92	1.86	1.81	2.60	**2.07**

Three databases (encryption times)

Doubling Step	Run 1	Run 2	Run 3	Run 4	Run 5	**Average**
1250 → 2500	2.12	3.14	2.13	1.69	2.02	**2.22**
2500 → 5000	2.11	1.47	1.51	2.05	1.93	**1.81**
5000 → 10000	1.87	2.36	1.78	2.04	1.82	**1.97**
10000 → 20000	1.91	2.11	2.67	1.77	2.22	**2.14**

Three databases (decryption times):

Doubling Step	Run 1	Run 2	Run 3	Run 4	Run 5	**Average**
1250 → 2500	2.10	2.21	2.30	1.98	2.45	**2.21**
2500 → 5000	1.70	1.76	1.93	1.95	2.05	**1.88**
5000 → 10000	2.09	3.00	1.62	1.88	1.38	**2.00**
10000 → 20000	2.07	2.12	2.65	2.02	2.61	**2.29**

If you were to round all the averages to the nearest whole number, you would get 2. That means that my program has a linear time complexity O(n)

4.4 Exploring The Graph Data

By using Matplotlib to visualize my data, I was able to clearly observe the rate of growth for each doubling step. I split my results into 4 graphs. One is the decrypt times and the other is encrypt times. The first graph is the decrypt graph.

The decrypt Graphs:

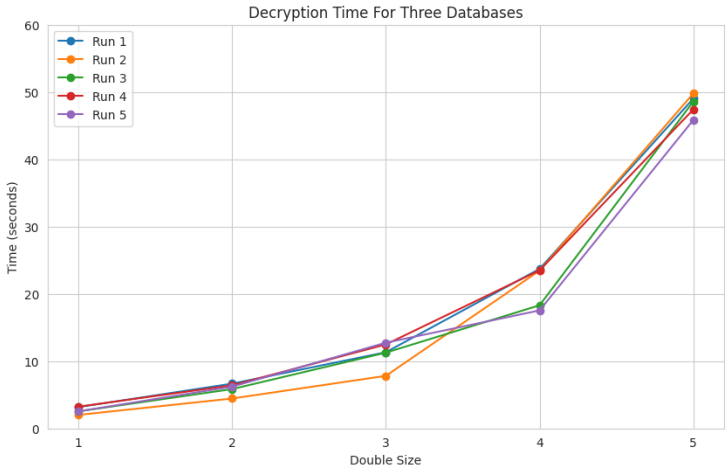


Figure 3: database decrypt

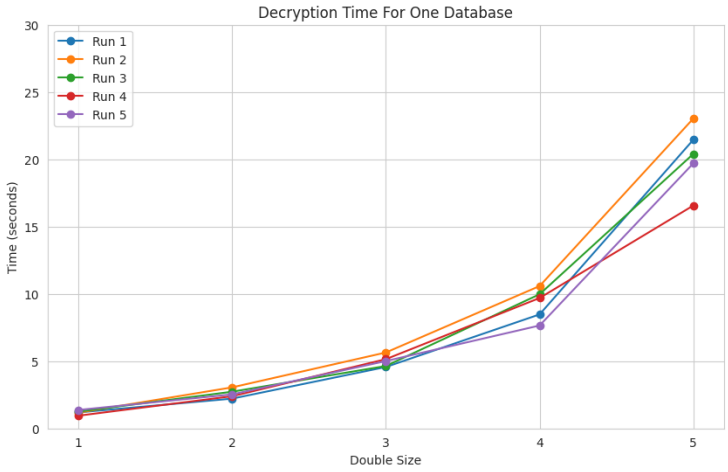


Figure 4: database encrypt

These graphs illustrate how decryption time scales as the dataset size increases. The x-axis represents the doubling run, where each step corresponds to the dataset size that has doubled from the previous run. For example, Run 1 starts with the baseline dataset of 1,250 rows. Run 2 represents twice that size, Run 3 is double Run 2, and so on. The y-axis measures decryption time in seconds, showing how the computational cost evolves as the dataset grows.

The way my program is designed, it decrypts the entire database every time the program starts up. This explains why decryption times at the beginning are relatively low. However, as more rows are inserted, the time required for

decryption increases. Just by looking at the lines, it's clear that the graph is showing a linear growth in the time it takes to complete decryption.

The encrypt Graphs:

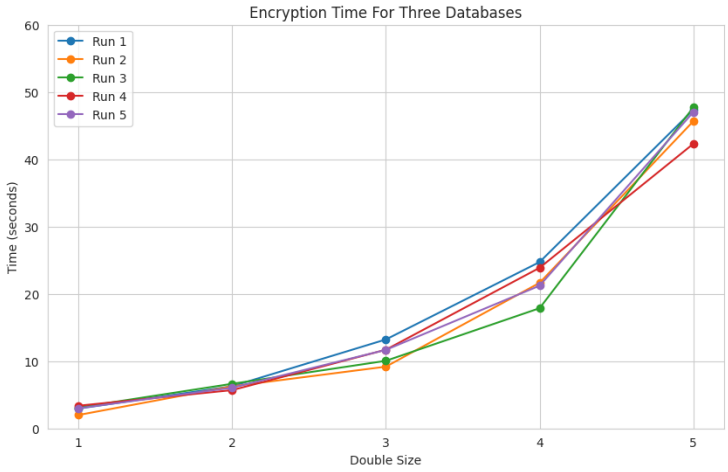


Figure 5: database 3 db encrypt

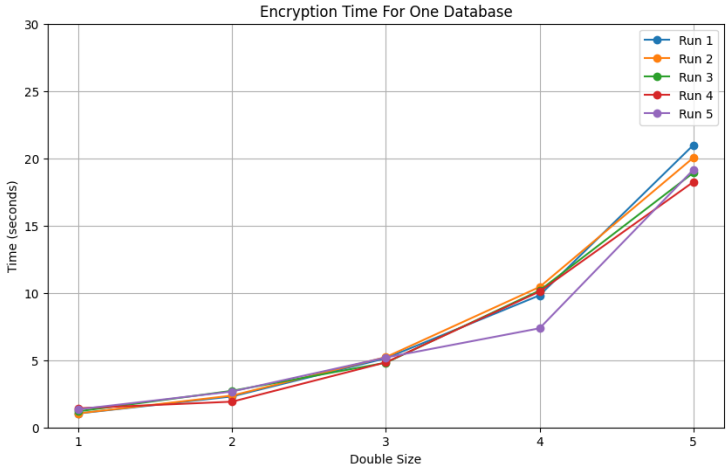


Figure 6: database 1 db encrypt

These graphs illustrate how encryption time scales as the dataset size increases. The x-axis represents the dataset doubling with each run, meaning Run 1 starts with a baseline dataset, Run 2 doubles that size, and so on. The y-axis measures encryption time in seconds, showing how the computational cost grows as the dataset expands.

Similarly to the decryption results, the encryption graphs display linear growth patterns. The consistent slope of each line confirms that encryption time increases proportionally with dataset size, reinforcing the O(n) complexity revealed by the doubling ratio analysis. The three-database configuration again shows higher overall processing times compared to the single-database setup. One notable observation is the relatively consistent performance across different runs of the same configuration, particularly in the single-database setup. This consistency suggests that my encryption algorithm delivers predictable performance

4.5 Summary of Results

My empirical analysis of the encryption and decryption algorithms reveals a consistent linear time complexity across all test configurations. By calculating doubling ratios—the ratio of execution times when input size doubles—I found that both encryption and decryption operations scale linearly with input size.

The average doubling ratios across five experimental runs consistently approximated 2 for both operations:

```
- For single database encryption: 2.02-2.14 (average ~2.04)
- For single database decryption: 1.86-2.21 (average ~2.06)
- For three database encryption: 1.81-2.22 (average ~2.04)
- For three database decryption: 1.88-2.29 (average ~2.10)
```

These values strongly indicate O(n) time complexity, confirming that processing time increases proportionally with input size. Visual analysis of the timing graphs corroborates these findings, though the steeper curves at larger input sizes (particularly for the three-database configuration) suggest some overhead that becomes more pronounced as data volume increases.

While the algorithm demonstrates linear complexity, there are notable performance differences between configurations. The three-database setup consistently requires approximately twice the processing time compared to the single-database implementation when handling equivalent data volumes, indicating potential optimization opportunities in the multi-database approach.

4.6 Validation Methodology

The most important part of my work was making sure my findings were accurate. A key component of my validation approach was systematically calculating doubling ratios across multiple independent runs. As shown in the doubling ratio tables, I conducted five complete experimental runs for each database configuration. For every doubling step (1250→2500, 2500→5000, etc.), I calculated the ratio of execution times.

This methodical approach provided several validation benefits:

- It allowed me to identify any anomalous runs that deviated significantly from expected patterns
- The consistency of doubling ratios across different runs (~2.0) strongly validated my conclusion of linear time complexity
- By averaging these ratios, I reduced the impact of any environmental variations or outliers on my final assessment

For example, in the single database decryption experiment, the doubling ratios across five runs for the 5000→10000 step were 1.86, 1.87, 2.15, 1.88, and 1.53, with an average of 1.86. The relative consistency of these values confirms that the observed performance scaling is reproducible and not a result of random fluctuations.

There are a lot of variables that can affect a doubling experiment. Things like how many tabs are open, programs running in the background, the age of the computer, or even what browser is being used can all impact how long the experiment takes. That's why I made sure to keep my testing conditions the same every time. This included:

- Running experiments at similar times of day to minimize system load variations
- Closing unnecessary background applications before each test
- Verifying database connection status before beginning each run
- Using the same hardware and system configuration throughout all tests

I also double checked my code to make sure that it was working properly, inserting the correct amount of data and decrypting it by adding print statements to my code. I even looked at the database itself through PostgreSQL to verify everything was working and it was. These steps helped confirm that my findings were real and could be repeated.

5 Conclusion

5.1 Takeaways

Overall, I am pleased with the results. The goal of my experiment was to stress-test my program, and I'm happy with how it turned out. My intention is for the program to be used by small groups, small businesses, or individuals who need a simple and secure way to manage their data.

The encryption and decryption processes scaled linearly with the amount of data, which was a positive indicator—it means the system handles growth in a predictable and manageable way. For the kind of users I had in mind, this level of performance is more than enough. The results show that my program strikes a balance between simplicity, flexibility, and security without requiring complex setup or advanced technical knowledge.

5.2 Ethical Considerations

The ethical implications of designing an encrypted database system are significant, as encryption is fundamentally about securing sensitive information. At its core, encryption serves as a means of protecting personal and organizational data from unauthorized access, ensuring privacy, and preventing potential misuse. However, with great security comes ethical responsibility—both in how encryption is implemented and in how it impacts users and broader society.

A primary ethical goal in my design is to ensure that users have complete control over their data. Encryption ensures that only the rightful owner of the data, who possesses the correct decryption key, can access it. This guarantees that even if an attacker gains access to the database, the stored information remains unreadable without the proper credentials. By implementing encryption at the application level before data is stored in the database, security is enhanced beyond traditional database protection methods. Even database administrators, who typically have broad access privileges, cannot access plaintext data, preventing insider threats and unauthorized viewing of sensitive information.

This design choice reflects a strong commitment to user privacy. In an era where data breaches and unauthorized data collection are increasing concerns, encryption empowers individuals and organizations to safeguard their information. Users, not third parties, retain control over their own data, reducing the risks associated with centralized data collection and mass surveillance.

However, ethical considerations extend beyond just privacy. Encryption can sometimes create barriers to data recovery in cases where users lose their decryption keys, leading to irreversible data loss. To address this, my system could incorporate optional key recovery mechanisms or user-defined backup options that do not compromise overall security. Additionally, encryption raises questions about regulatory compliance. Different industries, such as healthcare and finance, must follow strict data protection laws, meaning encryption must be implemented in a way that aligns with legal standards while still maintaining

user security.

Another ethical aspect involves ensuring accessibility and ease of use. If encryption is too complex for non-technical users to manage, they might be forced to rely on weaker security practices or external services that diminish their control over their own data. Therefore, a balance must be struck between strong encryption and user-friendly implementation, ensuring that security does not come at the cost of usability.

Ultimately, ethical encryption is about more than just securing data—it is about ensuring that security measures empower users rather than restrict them. By prioritizing privacy, security, and responsible design choices, this encryption system supports the ethical goal of giving individuals and organizations control over their own data while maintaining the integrity of sensitive information

5.3 Protection Against Malicious Use

While encryption is primarily designed to protect users from unauthorized access, it is also important to consider how encryption might be misused for nefarious purposes. Encrypted databases, if not designed carefully, could be exploited to conceal harmful activity, such as storing illicit data without accountability. This has been a growing concern in cybersecurity, with debates surrounding the balance between user privacy and the ability to prevent criminal misuse of encryption.

To address this concern, my encryption system is structured in a way that does not provide complete anonymity. While the contents of the data remain encrypted, it is still possible to decrypt the data inside the database under the correct circumstances. The system ensures that while users maintain privacy, if legal action needs to be taken, the data can be recovered easily. This approach aligns with principles of responsible encryption ensuring strong security while avoiding a completely opaque system that could be exploited for harmful purposes.

Additionally, my design choice takes into account potential real-world applications where organizations may need to comply with data access regulations. In some industries, such as finance and healthcare, encrypted data must still be retrievable under legal orders. By allowing for controlled decryption, the system can be adapted to environments where compliance with such regulations is necessary.

However, this approach also raises important questions about who should have access to decryption capabilities. If a system allows any third party unrestricted access to encrypted data, it could undermine the very security that encryption is meant to provide. To mitigate this, the system ensures that only the data owner, or an authorized entity under specific legal conditions, can perform decryption.

By carefully balancing privacy and accountability, this encryption system aims to provide strong data security without inadvertently creating a tool for misuse. Future improvements could focus on refining access control mechanisms, ensuring that encryption remains both practical and ethically responsible.

5.4 Future Work

5.4.1 Different Encryption Methods

There is significant potential for improvement in this work, particularly in the encryption and decryption process. Currently, I utilized PyFernet to manage encryption, and while the results show acceptable performance, there is room for optimization. One area of enhancement could involve switching to a more efficient encryption method. PyFernet, while convenient, is relatively slow due to its lengthy process of encryption and subsequent decryption [5]. By developing a custom encryption method using AES, it could be possible to accelerate these operations significantly. This improvement might enable the program to process even larger databases and handle more rows, enhancing its overall efficiency and scalability.

5.4.2 Built in Encryption Methods

On the topic of different encryption methods, another future improvement could be to have different built in encryption methods. Since one of the goals of this project is ease of use, having multiple built in encryption methods and allowing the user to pick from one would grant them more control over their privacy and possibly allow the program to be even faster if they decide they want a weaker encryption method.

5.4.3 Backups

Another important feature that could be added is backups. Backups ensure that the system's data is not lost in case of unforeseen circumstances. Implementing an automated backup system that periodically saves snapshots of the database to a secure location would provide peace of mind and maintain data integrity. These backups could be stored locally with encryption applied to ensure the data remains secure even in its backup state. Furthermore, options for incremental backups storing only the changes since the last backup could improve efficiency and reduce storage requirements.

5.4.4 Logging

Logging is another feature that can enhance both the program's functionality and its security. Detailed logs of program activities, including database queries, encryption and decryption operations, and user actions, can help the user monitor the system's performance and identify potential issues. Logs also serve as a valuable tool for debugging, allowing developers to trace errors and optimize processes. From a security perspective, logs can help detect unauthorized access attempts or anomalous behavior. By incorporating backups and logging, alongside encryption optimizations, the program could achieve greater reliability, data protection, and ease of maintenance, making it more robust for real-world deployment.

5.4.5 Misuse Prevention

As of writing this, my program can be used as ransomware. If someone is able to get the proper credentials, they can connect to a database and encrypt the entire database. One of the few deterrents of using my program as ransomware is the time it takes to encrypt. As it was shown, my program can handle large databases but it will still take some time to encrypt all that data. Hopefully most databases nowadays have something put in place to detect and stop a massive change to the data inside it.

5.4.6 Total Encryption

As of writing this my program does not have total encryption. The reason for this is because of you cannot encrypt table column names normally in postgres. This is because column names are integral to SQL queries and database operations. When you encrypt a column name, it becomes unreadable to the database engine, which relies on these names to execute queries like SELECT, INSERT, UPDATE, and DELETE. Encrypted column names would break the syntax of SQL queries, as they would no longer match the original schema or adhere to the database's rules for valid identifiers (e.g., alphanumeric characters, no special symbols). Additionally, column names are part of the database's metadata, not the data itself, and are used to define the structure of tables. Encrypting them would require renaming the columns in the database, which could disrupt existing queries, indexes, and relationships. To handle this, you would need to maintain a mapping between the original and encrypted column names and dynamically translate them during query execution, adding significant complexity to your system.

5.5 Final Thoughts

Overall, this project gave me much to think about—not just in terms of performance and features, but also about the bigger picture: who controls data, and how we can make security tools more accessible. I set out to build something simple, usable, and secure, and I think I've made a strong start. There's still plenty of room to grow—better encryption methods, smarter logging, backups, and protections against misuse—but the foundation is there. I hope that with more development, this kind of system can help regular users and small teams take back control of their data without needing to be security experts.

References

[1] Katherine Amoresano and Benjamin Yankson. 2023. Human error-A critical contributing factor to the rise in data breaches: A case study of higher education. *Holistica Journal of Business and Public Administration* 14, 1 (2023), 110–132.

[2] Rick Cattell. 2011. Scalable SQL and NoSQL data stores. *SIGMOD Rec.* 39, 4 (May 2011), 12,27. https://doi.org/10.1145/1978915.1978919

[3] Long Cheng, Fang Liu, and Danfeng Yao. 2017. Enterprise data breach: causes, challenges, prevention, and future directions. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 7, 5 (2017), e1211.

[4] David Dobkin, Anita K. Jones, and Richard J. Lipton. 1979. Secure databases: protection against user influence. *ACM Trans. Database Syst.* 4, 1 (March 1979), 97106. https://doi.org/10.1145/320064.320068

[5] Fernet. n.d.. Fernet Specification. https://github.com/fernet/spec/blob/master/Spec.md.

[6] GeeksforGeeks. 2024. Difference between MySQL and PostgreSQL. https://www.geeksforgeeks.org/difference-between-mysql-and-postgresql/. Accessed: 2025-04-07.

[7] PostgreSQL Global Development Group. 2024. PostgreSQL: Documentation: Limits. https://www.postgresql.org/docs/current/limits.html. Accessed: 2024-12-04.

[8] PostgreSQL Global Development Group. n.d.. About PostgreSQL. https://www.postgresql.org/about/. Accessed: 2025-04-06.

[9] PostgreSQL Global Development Group. n.d.. PostgreSQL 7.4 Features. https://www.postgresql.org/docs/7.4/features.html.

[10] GUVI. 2024. DBMS ACID Properties for Data Integrity. https://www.guvi.com/blog/dbms-acid-properties-for-data-integrity/. Accessed: 2024-02-20.

[11] Sushil Jajodia. 1996. Database security and privacy. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 129–131.

[12] Matthew Kosinski. 2024. Database. https://www.ibm.com/think/topics/database. Accessed: 2024-12-06.

[13] Matthew Kosinski. 2024. Database. https://www.ibm.com/think/topics/database. Accessed: 2024-12-06.

[14] Daniel Leung, Omar Tsai, Kourosh Hashemi, Bardia Tayebi, and Mohammad A. Tayebi. 2024. XploitSQL: Advancing Adversarial SQL Injection Attack Generation with Language Models and Reinforcement Learning. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management* (Boise, ID, USA) *(CIKM '24)*. Association for Computing Machinery, New York, NY, USA, 4653–4660. https://doi.org/10.1145/3627673.3680102

[15] Jiwei Li, Xuewei Ding, Yuchen Jiang, Yongqi Ai, Zhen Jia, and Xinyi Zhang. 2024. A Database Security Access Control Method Based on Identity Authentication. In *Proceedings of the 2023 International Conference on Artificial Intelligence, Systems and Network Security* (Mianyang, China) *(AISNS '23)*. Association for Computing Machinery, New York, NY, USA, 380–385. https://doi.org/10.1145/3661638.3661710

[16] Microsoft. n.d.. BitLocker. https://learn.microsoft.com/en-us/windows/security/operating-system-security/data-protection/bitlocker/. Accessed: 2025-03-31.

[17] Oracle. 2024. What is a database? https://www.oracle.com/database/what-is-database/. Accessed: 2024-12-05.

[18] Python Cryptographic Authority. n.d.. Symmetric Encryption — Cryptography 41.0.3 documentation. https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/.

[19] Jukka Ruohonen, Kalle Hjerppe, and Maximilian Von Zastrow. 2024. An Exploratory Case Study on Data Breach Journalism. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*. 1–9.

[20] Sanket Vilas Salunke and Abdelkader Ouda. 2024. A Performance Benchmark for the PostgreSQL and MySQL Databases. *Future Internet* 16, 10 (2024), 382.

[21] Vasily Sidorov and Wee Keong Ng. 2015. Transparent data encryption for data-in-use and data-at-rest in a cloud-based database-as-a-service solution. In *2015 IEEE world congress on services*. IEEE, 221–228.

[22] Michael Stonebraker and Rick Cattell. 2011. 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM* 54, 6 (June 2011), 9 pages. https://doi.org/10.1145/1953122.1953144

[23] N. Udeshi. 2019. Saving small business from the big impact of data breach: A tiered federal approach to data protection law. *Brooklyn Journal of Corporate, Financial & Commercial Law* 14 (2019), 389.

[24] Spencer Wheatley, Thomas Maillart, and Didier Sornette. 2016. The extreme risk of personal data breaches and the erosion of privacy. *The European Physical Journal B* 89 (2016), 1–12.

[25] LLC Zetetic. n.d.. About SQLCipher - Information and History. https://www.zetetic.net/sqlcipher/about/. Accessed: 2025-03-31.

[26] LLC Zetetic. n.d.. Buy SQLCipher - Commercial Edition Purchasing and CipherCare Support. https://www.zetetic.net/sqlcipher/buy/. Accessed: 2025-03-31.

[27] Luwei Zhang, Tianyu Li, Huang Jun, Dongmin Li, Bei Zhu, and Jing Li. 2020. Design and implementation of database encryption system for cloud environment. In *Proceedings of the 1st International Conference on Advanced Information Science and System* (Singapore, Singapore) *(AISS '19)*. Association for Computing Machinery, New York, NY, USA, Article 25, 7 pages. https://doi.org/10.1145/3373477.3373502