University of South Carolina
Scholar Commons

Theses and Dissertations

Fall 2023

Enhancing Relation Database Security with Shuffling

Tieming Geng

Follow this and additional works at: https://scholarcommons.sc.edu/etd

Recommended Citation
Geng, T.(2023). *Enhancing Relation Database Security with Shuffling.* (Doctoral dissertation). Retrieved from https://scholarcommons.sc.edu/etd/7609

This Open Access Dissertation is brought to you by Scholar Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact digres@mailbox.sc.edu.

Enhancing Relation Database Security with Shuffling

by

Tieming Geng

Bachelor of Engineering
Nanjing University of Science and Technology Zijin College 2011

Master of Science
University of South Carolina 2018

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2023

Accepted by:

Chin-Tser Huang, Major Professor

Csilla Farkas, Committee Member

Marco Valtorta, Committee Member

Jianjun Hu, Committee Member

S Mo Jones-Jang, Committee Member

Ann Vail, Dean of the Graduate School

© Copyright by Tieming Geng, 2023
All Rights Reserved.

Dedication

I dedicate this dissertation to my loving wife, Lu Chen, and our beautiful daughter, Gloria. Your unwavering support, encouragement, and understanding throughout this long journey have been my greatest source of strength. Your sacrifices and patience have allowed me to pursue my academic goals. Thank you for being the pillars of my life.

Acknowledgments

I would like to express my heartfelt gratitude to my advisor, Dr. Chin-Tser Huang, for his guidance, patience, and expertise throughout the entire program. His mentorship to my academic growth have been invaluable. I am truly fortunate to have had the opportunity to learn from him.

I also would like to thank the rest of my dissertation committee members, Dr. Csilla Farkas, Dr. Marco Valtorta, Dr. Jianjun Hu, and Dr. S Mo Jones-Jang. Thanks for their kindness to serve on my dissertation committee and their assistance on my research.

I extend my deepest appreciation to my parents, for their endless love, encouragement, and support. Their belief in me and their sacrifices have been the driving force behind my academic achievements.

Abstract

Database security holds paramount importance as it safeguards an organization's most valuable assets: its data. In an age marked by escalating cyber threats, protecting sensitive information stored in databases is essential to preserve trust, prevent financial losses, and maintain legal compliance. In this dissertation, an exploration into the realm of relation database security is undertaken. The research introduces a cryptographic secure shuffling algorithm designed to fortify database security. Additionally, the dissertation presents a series of innovative solutions aimed at bolstering both the security and efficiency of the shuffling algorithm.

Encryption algorithms have long served as a mean of safeguarding sensitive and proprietary data. However, the proposed shuffling algorithm offers distinct advantages over the encryption methods: Firstly, the shuffling algorithm preserves the original data, minimizing the likelihood of arousing suspicion. Secondly, it can complement encryption, offering an additional layer of data protection. Lastly, the shuffling algorithm can introduce the deception, thereby rendering certain adversary techniques such as honeypot is more effective.

Nonetheless, whether employing encryption or shuffling techniques, data security preservation technologies impose a substantial volume of I/O requests on the system. To enhance efficiency, various endeavors have been made, encompassing the introduction of novel storage data structures, the adoption of column-oriented storage schemes, and the integration of computational storage devices. Empirical findings from experiments reveal that these design modifications notably mitigate the associated overhead.

Simultaneously, efforts have been made towards enhancing the security aspects of the research, incorporating strategies such as attribute-aware bundle shuffling, multi-level shuffling, and matrix shuffling. All these efforts collectively contribute to concealing the traces of shuffling and render the shuffling process irreversible in the absence of the shuffling sequence. A comprehensive security analysis has demonstrated the cryptographic robustness of the proposed shuffling algorithm.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In today's era of the Internet and big data, database security and the protection of database data play a particularly crucial role in preventing the leakage of sensitive information. Databases, serving as the repository for sensitive and proprietary data, have become one of the primary target scenarios for cybercriminals seeking valuable information. The escalating frequency of data leaks and the increasing complexity and sophistication of attacks pose significant threats to organizations and communities, resulting in substantial financial losses, legal liabilities, and damage to their reputation [1]. These threats ultimately have repercussions on individuals as well. In August 2023, the aftermath of the MOVEit security incident is still ongoing, with the ransomware group CL0P launching more attacks against government agencies [2], including the Colorado Department of Health Care Policy & Financing and many government contractors. Sensitive data and personal information, including SSNs and health records, of millions of people have been compromised, and more data is still being exposed.

Currently, cryptographic encryption stands as one of the most widely adopted protection methods to address database attacks and data breaches. Various encryption techniques, such as transparent data encryption, token-based systems, and fully homomorphic encryption, have been employed for this purpose. In Transparent data encryption, the pages in the database are encrypted before being written to disk and decrypted when read into memory. This approach ensures that even if the database server is breached, the malicious attacker gains no valuable information. However, encryption in the database can reveal file importance, attracting attackers. Token-based systems like CryptDB [3] and searchable encryption [4] allow searching over encrypted data but face criticism from Grubbs et al. [5] for potential issues. It is worth noting that achieving semantic security [6] becomes challenging if even a single token value is known to attackers. Fully homomorphic encryption [7] enables algebraic operations on ciphertext without decryption but may have efficiency concerns due to large encryption keys in practical use [8].

In this dissertation, a novel shuffling approach to enhance the security of relational database is proposed. This cryptographically strong algorithm is based on permutation to shuffle the data values in each field of the database table, in order to assure the privacy and reduce the risk of data compromise and aforementioned identity theft attacks. Randomization is applied in our algorithms to make sure the different fields would follow totally distinct shuffling parameters. Semantic check and statistical test are also performed to identify highly associated attributes which will be shuffled together as a bundle. There are two remarkable advantages of this bundle shuffling design. First, even if the attacker is able to defeat the encryption and hack into the shuffled database, he will not be able to detect that the database has been shuffled because the shuffling is attribute association aware. On the other hand, legitimate users of the database will not feel the existence of shuffling or restoring, because these procedures are transparent to them. Second, the tuples extracted from the shuffled database cannot be used for identity theft attacks because they contain mismatching attribute values; if the attacker tries to use the shuffled tuples, it may even touch the previously established honeypot and trigger an alert.

The design also integrates the shuffling algorithm with a column-oriented storage scheme and the computational storage device (CSD) to enhance the security and efficiency of the relational database management systems. A collection of two-column association tables is leveraged to record the disk locations of the actual data and their corresponding indexes. With the assistance of these association tables, shuffling can be accomplished without physically relocating the data, resulting in significant savings in time and system resources.

A computational storage drive (CSD) is a “in storage computing”. The conception of “in storage computing” or “near storage computation” was proposed as early as the 1990s, where the disk was designed to execute most processing tasks to processors inside the disk [9], [10]. However, this conception began to gain attention and development around the early 2010s [11], [12]. By deploying the in storage computing devices in database servers, certain tasks like shuffling and encryption can be offloaded to the storage devices. When shuffling or encryption requests are received, the host forwards these requests to the storage. The processor within the storage retrieves the necessary data from storage to its internal DRAM, where it handles the processing requests. Subsequently, the outcomes are written back to the storage. It also eliminates the need for such data to traverse the lengthy path through the system bus, spanning storage, RAM, cache, and processors. It reduces data transfer latency and enhances data processing efficiency by introducing processing capabilities inside the storage. CSD is able to accelerate the shuffling algorithm, including software and hardware architectures, ensuring that the performance overhead of shuffling algorithms does not affect the execution of other tasks in the database system.

There are five chapters in the following part of this dissertation. Chapter 2 describes the general knowledge and works that are related to this research. Chapter 3 discusses the database security problem and proposed the shuffling algorithm with attribute association aware design. Chapter 4 discusses the shuffling in column-oriented storage database and the novel data structure for efficient data shuffling. Chapter 5 introduces how computational storage devices are involved with database shuffling for better performance and describes the improvement on the shuffling algorithm. At last, chapter 6 concludes this dissertation and lists the major conclusion as well as some future work.

Chapter 2

Background and Brief Literature Review

2.1 Background

2.1.1 Column-oriented Database

Column-oriented database is not a new technique. It can be traced back to a storage system called TAXIR focused on biology information retrieval in 1969 [13]. However, due to the hardware limitations and application scenarios of early databases, the mainstream OLTP (Online Transactional Processing) databases adopted row-oriented storage. Along with the emerging development of OLAP (Online Analytical Processing), column-oriented storage becomes prevalent again.

In traditional OLTP databases, all the attributes are stored as a concatenated tuple, which serves as the base unit of storage. When combining with B-tree [14], B+ tree [15], or SS-table [16] for indexing, the corresponding data of the tuple can be efficiently retrieved using the primary key. The employment of row-oriented storage is well-suited for OLTP because most operations in OLTP revolve around entities, making CRUD (Create, Read, Update, and Delete) operations on a tuple the norm. Therefore, consolidating all attributes of a tuple together is a practical choice. However, in OLAP, typical operations involve traversing the entire relation and following grouping, sorting, and aggregation actions. Thus, the advantages of row-oriented storage diminish. Even worse, the SQL statements in OLAP often only require a few specific attributes, but those unrelated attributes still have to be included in the processing.

In addition to the analytical query, column-oriented storage is superior in data encoding and compression. No matter for in-disk databases or in-memory databases, typically IO is the system performance bottleneck, and reasonable compression could decrease the disk usage, but also reduce the IO throughput and increase the load performance.

Taking C-Store [17] as an example, based on two factors: 1) if the data is sorted, and 2) the number of distinct values, the data encoding can be divided into the following 4 situations:

- Sorted and not many distinct values: A series of tuples (v, f, n) will be used to represent the data, which means value v appears starting from line f and the total number is n. In other words, the values of line f to $f + n - 1$ are all v. For example, value 4 are seen in line 12 to line 18 can be written as $(4, 12, 7)$.

- Not sorted and not many distinct values: Build one binary sequence b for each value v to indicate the position bitmap of v. For instance, one column of attributes in a relation is $0, 0, 1, 1, 2, 1, 0, 2, 1$, then the encoding could be $(0, 110000100), (1, 001101001)$, and $(2, 000010010)$. Because of the sparse feature of bitmap, they can be coded for one more step using run-length coding.

- Sorted and many distinct values: For this case, one *delta* will be add to each value except the first one of course. For example $1, 4, 7, 7, 8, 12$ could be modified to $1, 3, 3, 0, 1, 4$. Clearly, coded data is easily to be a densepack [17], and the compression ratio is higher.

- Not sorted and many distinct values: That is not properly conducted in C-Store.

Data Compression in Column-oriented Database

Data size in the storage of database becomes increasingly sensitive due to the cost of network bandwidth and storage media along with the business development. Thus data compression is critical in the database management system. Column-oriented database has an overwhelming advantage on the compression ratio comparing to the row-oriented database because the unit of storage is one column and all the data in this column is of the same data type. Since all compression algorithms can be applied on column-oriented databases, we will not discuss their details due to space constraint. Various compression schemes for column-oriented database are briefly described in the followings.

Lempel-Ziv Codes. Lempel-Ziv [18], often written as LZ77 or LZ78 depending on which kind of variant is used, is one of the oldest, most simplistic, and most widespread lossless compression scheme in now days. This compression scheme replaces variable-sized pattern with fixed length codes. The Lempel-Ziv method is an incremental parsing strategy in which the coding process is interlaced with a learning process for varying source characteristics. The primary difference between LZ77 and LZ78 is the encoding of the phrases that LZ77 uses direct pointers to the preceding text while LZ78 uses pointers to a separate dictionary.

Dictionary Codes. In dictionary compression [19], variable length substrings are replaced by short and possibly fixed length codes. The dictionary $\mathcal{D}$ here is a collection of phrases, and the text T which is going to be compressed will be factorized into a sequence of phrases so that each phrase can be replaced by a code that acts as a pointer to the certain place in the dictionary.

Run Length Encoding. Run length encoding [20] is based the idea of combining the same value occurring many consecutive times and storing only a singe value with its count. For example, a sequence of same value can be represented as the following form: (v, s, l) in which v means the value, s means the start position and l means the lasting length. In row-oriented database, run length encoding is very suitable for the text fields that have many blanks or repeated characters. But in column-oriented database, sorting is typically applied before the compression, so run length encoding is useful in more places such as the columns that have less distinct values.

Zero Suppression. Zero suppression can actually be one kind of trimming before storing the data since it means the removal of redundant zeroes from a number [21]. For example, the first several zeros in one integer can be removed, and last several zeros in the decimal part of a number. After the general zero suppression, the number can be compressed further more with other compression schemes. The other way to utilize this technique is to omit the zeros in a sparse array that most of elements are zero. For instance the array $(0, 1, 0, 0, 0, 2, 5, 0, 4, 0, 0, 1)$ can be compressed into $(\{2, 1\}, \{6, 2\}, \{7, 5\}, \{9, 4\}, \{12, 1\})$. Of course, this example doesn't seem efficient on the compression ratio, but it will save much more space when dealing with a large sparse array.

Hybrid Columnar Compression. Hybrid columnar compression is one method from Oracle for organizing data within a database block which utilizes a combination of both row and columnar methods [22]. Similar to many other compression schemes, this compression scheme replaces the value from a row with a much smaller symbol, so the length of the entire row reduces. When data is loaded, column data from a set of rows are grouped together and compressed, and then stored in a compression unit. Actually, this compression scheme is invented for the row-oriented database for the purpose of increasing the compression ratio while avoiding the performance shortfalls of a pure columnar format.

2.1.2 SmartSSD

The SmartSSD CSD possesses the capability to perform computations directly within the storage device itself. As illustrated in Fig. 2.1, this SmartSSD CSD comprises a high-performance Samsung Enterprise SSD and a specialized Kintex Ultrascale+ FPGA from Xilinx (now part of AMD). These components are interconnected by a fast private peer-to-peer (P2P link, facilitating seamless data transfer between the

SSD controller and the FPGA [23].

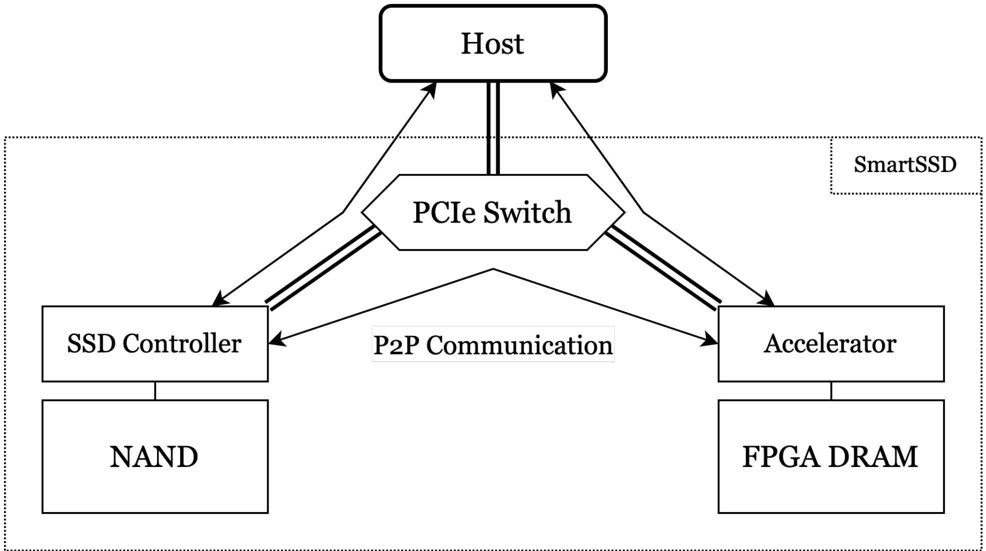

Figure 2.1 SmartSSD architecture

Xilinx provides a sophisticated development target shell characterized by high-performance capabilities, thereby facilitating the creation of customized acceleration applications tailored specifically for the SmartSSD CSD platform [24]. Within the architectural framework of the SmartSSD CSD, a pivotal component is the PCIe switch, which exhibits three distinct ports. The upstream port establishes a connection to the host system, while the downstream port establishes a link with the SSD controller. Intriguingly, an internal end port is dedicated to interact with the FPGA. This dedicated internal port serves as the conduit for enabling acceleration functions within the SmartSSD CSD.

One of the salient advantages offered by this architectural design is the ability to offload user application code either partially or in its entirety to the FPGA's DRAM. Additionally, data essential for FPGA-based operations, residing within the SSD storage, can be efficiently transmitted via the P2P link. This innovative approach significantly mitigates the need for the conventional round-trip data traffic between the storage and the host CPU.

This architecture refinement translates into real benefits in terms of computational efficiency, particularly when dealing with resource-intensive operations such as data shuffling and encryption, both of which involve substantial I/O interactions. The reduction in I/O latency and data transfer overhead is instrumental in optimizing the overall performance.

2.2 Database Security

2.2.1 Searchable Encryption

The management of the data in database is conducted by the database management system (DBMS), and the DBMS is consisted of several layers responsible for SQL statement parsing, index management, data reading and writing, ensuring data consistency and integrity, backup, etc. The engineering development of such a system is instructed by accumulative experience. In this context, the searchable encryption [25] solutions for the database are typically implemented not within but atop the DBMS, functioning as middleware. This middleware serves to translate encrypted queries to the DBMS while safeguarding plaintext data, subsequently decrypting the outcomes.

CryptDB

CryptDB is a system that provides practical and provable confidentiality for online applications backed by SQL databases [3]. It works by executing SQL queries over encrypted data using a collection of efficient SQL-aware encryption schemes. CryptDB can also chain encryption keys to user passwords, so that a data item can be decrypted only by using the password of one of the users with access to that data. As a result, a database administrator never gets access to decrypted data, and even if all servers are compromised, an adversary cannot decrypt the data of any user who is not logged in.

CryptDB works by intercepting all SQL queries in a database proxy, which rewrites queries to execute on encrypted data. The proxy encrypts and decrypts all data, and changes some query operators, while preserving the semantics of the query. The DBMS server never receives decryption keys to the plaintext so it never sees sensitive data, ensuring that a curious DBA cannot gain access to private information. To guard against application, proxy, and DBMS server compromises, developers annotate their SQL schema to define different principals, whose keys will allow decrypting different parts of the database.

SQL-aware encryption is a collection of efficient encryption schemes that CryptDB uses to execute SQL queries over encrypted data. These schemes are designed to preserve the semantics of SQL queries while providing confidentiality for the data. CryptDB adjusts the encryption scheme dynamically based on the query being executed, so that the DBMS server can compute certain functions over the data items based on encrypted data. For example, if the DBMS needs to perform a `GROUP BY` on column $\mathbb{C}$, the DBMS server should be able to determine which items in that column are equal to each other, but not the actual content of each item. Therefore, the proxy needs to enable the DBMS server to determine relationships among data necessary to process a query. By using SQL-aware encryption, CryptDB is careful about what relations it reveals between tuples to the server.

CryptDB addresses two threats. The first threat is a curious database administrator (DBA) who tries to learn private data (e.g., health records, financial statements, personal information) by snooping on the DBMS server; here, CryptDB prevents the DBA from learning private data. The second threat is an adversary that gains complete control of application and DBMS servers. In this case, CryptDB cannot provide any guarantees for users that are logged into the application during an attack, but can still ensure the confidentiality of logged-out users' data.

MONOMI

MONOMI [26] is a system designed to securely execute analytical workloads over sensitive data on an untrusted database server. It encrypts the entire database and runs queries over the encrypted data. The system employs split client/server query execution, which enables the execution of arbitrarily complex queries over encrypted data. Additionally, MONOMI utilizes several techniques to enhance performance, such as per-row precomputation, space-efficient encryption, grouped homomorphic addition, and pre-filtering. To optimize for different types of queries, MONOMI includes a designer for selecting an efficient physical design for a given workload, and a planner for determining an efficient execution plan for a specific query at runtime. A prototype of MONOMI built on top of Postgres can execute most of the queries with minimal overhead compared to an unencrypted Postgres database.

Arx

Arx is a database system that encrypts data with strong encryption schemes and computes on the encrypted data [27]. It is the first practical and functionally rich database system that provides the same level of security as regular AES-based encryption without sacrificing functionality. Arx uses a set of new mechanisms that embed computation into data structures built on top of traditional encryption schemes, such as AES. This enables strong security guarantees and helps performance due to hardware implementations of AES. Arx also introduces two new database indices, `Arx-RANGE` and `Arx-EQ`, which are data structures built on top of AES and provide rich computation with strong security.

`Arx-RANGE` and `Arx-EQ` are two new database indices introduced by Arx. `Arx-RANGE` is used for range and order-by-limit queries, while `Arx-EQ` is used for equality queries. Both indices are data structures built on top of AES. `Arx-RANGE` uses a cryptographic tool for one-time obfuscation at each node in an index tree to provide rich computation with strong security. This allows running an obfuscated program at each index node that implements the desired comparisons without leaking the data it compares. `Arx-EQ` works by embedding a counter into each repeating value, ensuring that the encryption of two equal values is different and the server does not learn frequency information. This enables building a regular database index over the encryptions. When searching for a value v, the client can provide a small token to the server, which the server can expand into many search tokens for all the occurrences of v.

Arx works by encrypting data with strong encryption schemes and computing on the encrypted data. It embeds computation into data structures built on top of traditional encryption schemes, such as AES, instead of embedding the computation into special encryption schemes as in Fully Homomorphic Encryption and CryptDB. Arx provides strong security guarantees and helps performance due to hardware implementations of AES.

Seabed

Seadbed was proposed by Papadimitriou et al. [28] for the purpose of efficient analytic over large encrypted databases. It uses a novel, additively symmetric homomorphic encryption scheme (ASHE) to perform large-scale aggregations efficiently. Seabed also introduces a randomized encryption scheme called Splayed ASHE (SPLASHE) that can prevent frequency attacks based on auxiliary data. The system is designed to hide all cryptographic operations from users, allowing them to interact with the system in the same way as they would with a standard Spark system. Seabed supports a wide range of big data analytics applications and can perform operations such as computing sums, averages, counts, and minimum values. It also supports more complex analytics such as anomaly detection, linear regression, and decision trees. Seabed provides secure and efficient analytics over encrypted datasets, allowing enterprises to keep their data confidential and hidden from cloud operators.

An additively symmetric homomorphic encryption scheme (ASHE) is a type of encryption scheme that allows for the addition of two ciphertexts to obtain a ciphertext that decrypts to the sum of the two encrypted values. In other words, it enables computations on encrypted data such that the computed result, when decrypted, matches the result of the equivalent computation performed on unencrypted data. ASHE is designed to be efficient and fast, making it a practical alternative for performing operations on large data sets. It is up to three orders of magnitude faster than other encryption schemes like Paillier [29].

SPLASHE is an encryption scheme introduced in the document. It stands for Splayed ASHE and is designed to protect against attacks by splaying sensitive columns into multiple columns. Each new column corresponds to data for each unique element in the original column. SPLASHE combines splaying and deterministic encryption to defeat frequency attacks while limiting storage and computational overhead. It is used in the Seabed system to provide efficient analytics over large encrypted datasets.

Other Searching Over Encrypted Data

The exploration of searching encrypted data has yielded significant advancements, ranging from symmetric to asymmetric ciphers, as documented in seminal works by Song and Wagner [30] and Boneh and Franklin [31], respectively. Subsequent research has delved into executing intricate search queries on encrypted databases, encompassing conjunctive searches [32], rich and range queries [33]–[36], and dynamic queries accommodating data modifications subsequent to encryption [37], [38].

2.2.2 Database Security with Auxiliary Hardware

Trusted Execution Environment (TEE) is an isolated environment that safeguards processed data by encrypting the incoming and outgoing data. It provides mechanisms to ensure the computation is correctly executed with an integrity guarantee.

Additionally, TEE protects the data and computation against any potentially malicious entity residing in the system (including the kernel, hypervisor, etc.). Current TEEs can be classified into two categories: TEE supports simple stateless functions and TEE supports complex stateful functions. Many TEE designs focus on the latter, including TPM/vTPM, Intel TXT, Intel SGX, ARM's TrustZone, Sanctum, KeyStone, and AMD SEV. TEE exhibits a broad range of security features to achieve confidentiality and integrity of the computation, including secure boot, attestation, and isolated execution.

The use of TEE in secure database queries aims to implement data query tasks with optimal performance while ensuring confidentiality and integrity. Encrypted databases provide a hardware and software infrastructure for securely storing sensitive data and offering data query services to authorized users. TEE-based secure computation enables isolated execution of user-defined programs within enclaves, maintaining sensitive data in protected memory areas. This isolation helps prevent unauthorized access to the data during query processing. Additionally, TEE offers security mechanisms like secure boot, attestation, and sealing to further strengthen the protection of the database and its queries. The promising avenue of leveraging auxiliary hardware as secure coprocessing platforms has been a subject of exploration, offering security assurance for specific operations.

TrustedDB [39] is an outsourced database prototype that allows clients to execute SQL queries with privacy and under regulatory compliance constraints without having to trust the service provider. It leverages server-hosted tamper-proof trusted hardware in critical query processing stages. Despite the cost overhead and performance limitations of trusted hardware, the costs per query are orders of magnitude lower than any (existing or) potential future software-only mechanisms. It is built and runs on actual hardware, and its performance and costs have been evaluated.

Arasu et al. [40] describes a novel secure FPGA-based query coprocessor that can be tightly integrated with a commercial database system such as SQL Server to provide strong security guarantees and a flexible, cost-effective way of ensuring the efficient migration of database applications to the cloud. The coprocessor enables the execution of SQL queries on encrypted data entirely in the cloud without handling plaintext data or cryptographic keys in a conventional server where a user with root privileges could gain access. The paper discusses different alternatives for securing cloud database processing and presents the design of an FPGA-based specialized database stack machine with a modest footprint, only needing to implement expression evaluation in the trusted processor. The paper also reports initial results demonstrating the effectiveness of the system for real-world cloud database applications.

EncDBDB [41] is a high-performance, encrypted cloud database that supports analytic queries on large datasets. EncDBDB utilizes Intel Software Guard Extensions (SGX) to create a secure environment called an enclave, where sensitive data is protected from any other code, including application code, other enclaves, and the operating system. EncDBDB works by storing data in a column-oriented, dictionary-encoded format in memory. This allows for efficient processing of analytic workloads and reduces the storage space overhead of large encrypted datasets. The database offers nine different encrypted dictionaries that provide varying levels of security, performance, and storage efficiency. Users can choose the appropriate dictionary for each column based on the sensitivity of the data and their specific requirements.

EnclaveDB [42] is a database system designed to protect sensitive data and ensure confidentiality, integrity, and freshness for both queries and data. It is built on top of Intel's Software Guard Extensions, a hardware-based security feature that enables the creation of protected regions of memory called enclaves. EnclaveDB assumes that table definitions and stored procedures are pre-defined and known to all users, and that only authorized users can execute stored procedures. When a user creates an instance of EnclaveDB, they compile all table definitions and stored procedures in their own trusted environment and embed the public keys of all authorized users in a well-known section of the trusted kernel binary. The database administrator repeats this process and deploys their version of the package on an untrusted server. Once the database is initialized, any user can use remote attestation to check that the database enclave has been correctly initialized and initiate the creation of a secure channel. The enclave authenticates requests for creating a secure channel using public keys embedded in the trusted kernel binary. Once a secure channel is established, the user can send requests to execute any of the pre-compiled stored procedures. EnclaveDB ensures that users learn nothing more than the response of the transactions they execute.

ObliDB [43] is an enclave-based oblivious database engine that efficiently runs general relational read workloads over multiple access methods. It stores tables, authenticated and encrypted, in unprotected memory and obliviously accesses them as needed by the various supported operators. The encryption key for data stored in unprotected memory always resides inside the enclave, encrypting/decrypting blocks of data as they are written or read from unprotected memory. ObliDB can store data via two methods: flat and indexed. The indexed method utilizes a B+ tree, whereas the flat method requires scanning the whole table on each query to ensure obliviousness. ObliDB supports oblivious versions of the operators SELECT, INSERT, UPDATE, DELETE, GROUP BY and JOIN as well as the aggregates COUNT, SUM, MIN, MAX, and AVG. It also includes a query planner that chooses between operator implementations for selection and join queries.

StealthDB [44] is an encrypted database system built using Intel SGX technology. It provides a way to protect sensitive data in untrusted infrastructures. StealthDB has a small trusted computing base, scales well to large transactional workloads, requires minimal changes to the underlying database management system, and offers strong security guarantees during both steady-state operation and query execution. The system uses AES-CTR encryption [45] to encrypt all data items in the database. When a client submits a query, it is first encrypted and sent to the server. An enclave-based query parser within StealthDB then decrypts the ciphertext and parses the query, converting it to a form where all constants are encrypted before it is executed on the database.

2.2.3 Transparent Data Encryption

Furthermore, the adoption of transparent data encryption (TDE) [46], [47] by prominent industry stakeholders, including Microsoft, IBM, and Oracle, constitutes a noteworthy development. Transparent Data Encryption works by encrypting the data stored in a database on the hard drive and on any backup media. This means that even if an attacker gains access to the physical hard drive or backup media, they would not be able to read the data without the decryption key. The process of using TDE typically involves creating a master key, which is used to create a certificate that is then used to protect the database master key. The database master key is then used to encrypt the data stored in the database. When a user attempts to access the encrypted data, the database management system uses the appropriate decryption key to decrypt the data before presenting it to the user. This process is transparent to the user, meaning that they do not need to take any special actions to access the encrypted data.

2.3 Shuffling

In the realm of data privacy, Dalenius and Reiss introduced the concept of "data-swapping" in the 1980s [48], a technique designed to facilitate secure data sharing and analysis while minimizing the risk of sensitive information disclosure.

The paper proposed by Muralidhar et al. [49] discusses several methods for masking numerical data, including data swapping, rank-based proximity swap, differential privacy, and data shuffling. Data shuffling protects data by minimizing the risk of disclosure while providing a high level of data utility. This is achieved through a procedure that involves generating observations from a conditional distribution and replacing original values with these generated values. The resulting data preserves the desirable properties of perturbation methods and performs better than other masking techniques in terms of both data utility and disclosure risk. Data shuffling is a non-parametric method for masking and can be implemented using only rank-order data. It is particularly useful when dealing with small or large datasets. Additionally, data shuffling ensures conditional independence between the original data and the masked data given the sensitive attribute, thereby providing strong security against both value and identity disclosures.

UberShuffle [50] is a coded shuffling system designed to optimize the communication cost of data shuffling in modern large-scale learning algorithms. The shuffling process involves redistributing data among compute instances after each epoch to improve the statistical performance of the algorithm. The UberShuffle algorithm works by constructing a directed acyclic graph (DAG) between the columns of the encoding tables. In this DAG, a directed edge from column i of table A to column j of table B exists if and only if $i = j$ and $A \in B$. Then, the algorithm finds a flow on the DAG corresponding to packet reallocations to reduce the total number of encoded packets.

The shuffling mechanism used in the paper proposed by Meehan et al. [51] involves a trusted shuffler that mediates upon the noisy responses received from the data owners. The data owners first randomize their inputs. The shuffler then receives these noisy responses and applies a permutation to them, resulting in the final output sequence which is sent to the untrusted data analyst. The permutation is determined by the shuffling mechanism, which can be implemented using a trusted execution environment similar to Google's Prochlo [52]. The shuffling provides anonymity to the data owners, reducing the amount of noise required to achieve the same level of privacy compared to the local model.

Multi-media Format Shuffling

In the paper proposed by Gao et al. [53], an image total shuffling matrix is used to shuffle the positions of image pixels. This matrix is generated using the logistic map equation: $x_{n+1} = 4x_n(1 - x_n)$, starting with a given initial value x_0. After several iterations, a new x0 is obtained, which is used to determine the position of each row in the matrix. This process is repeated for each row in the matrix, resulting in a new position matrix. To shuffle the columns of the matrix, the current value of x_0 is used in the logistic map equation again, followed by a similar process of determining the position of each column based on the resulting values. This process is repeated for each row in the matrix, resulting in a new image total shuffling matrix. The time complexity of this algorithm depends on the size of the image, with larger images requiring more iterations to complete the shuffling process.

In the paper proposed by Ali et al. [54], the 3D content is encrypted through a process that involves several steps. First, the 3D mesh model is shuffled using a 3D Lorenz chaotic map and an encrypted texture map. The vertices of the model are then encrypted using a 2D logistic chaotic map and a 2D key, resulting in an encrypted 3D mesh model. Finally, the encrypted 3D mesh model is further encrypted using a 1D tent chaotic map and a 1D key, resulting in the final encrypted 3D content.

In the audio shuffle algorithm designed by Tamimi et al. [55], the audio file is encrypted or shuffled using a novel algorithm that employs a shuffling procedure and the stream cipher method. The algorithm uses a private key to perform key dependent and data dependent encryption. It takes an audio file and a key as input and regards the audio file as a stream. Then, for each iteration, a single bit called fixBit is chosen by a hash function based on the key and iteration number. A shuffle vector is constructed by listing the numbers of bytes that have the value of bit number fixBit equal to zero, followed by the numbers of bytes that have the value of bit number fixBit equal to one. This vector gives a mapping that specifies the new location of each byte in the partially encrypted stream. This step is repeated for several iterations. Each iteration uses a different fixBit and applies the same steps to the new partially encrypted stream that resulted from the preceding iteration. Finally, the bytes of the current stream are substituted so that the new location of byte (j) is byte (Shuffle[j]).

2.4 Column-oriented Database

C-Store is one of the first OLTP database management systems which are optimized for data reading. There is no big difference on the logic view of C-Store with the logic view of traditional relational database management systems. However, the logic views will be divided into several projections inside C-Store, and each projection contains one ore more columns. Of course, each column must exist in at least one projection. This feature of projection brings the following two advantages because of the redundancy of projection since one column can be included by multiple projections: 1) reduce of query cost by choosing the optimized projection, and 2) improvement of availability due to the existing of redundancy.

MonetDB [56] and ClickHouse [57] implemented vectorized query execution in order to make the most use of hardware especially CPU and improve the efficiency of querying. ClickHouse is based on the SIMD and SSE instruction set, so that in the level of CPU registers, one instruction can parallelly operate multiple pieces of data.

Apache Parquet [58] is a column-oriented data storage format instead of a entire database management system. Parquet was inspired by the record shredding and assembly algorithm described in the Dremel paper [59]. Parquet is independent of platform and programming language, and it is compatible with many query engines such as Hive, many computation frameworks such as MapReduce and Spark, and many object models such as Avro, Thrift. The biggest difference between Parquet storage format with the storage formats in other column-oriented database is that Parquet supports complex nested data structure such as json.

2.5 Computational Storage Device

One of the original conceptions of computational storage device comes from [10]. Computational storage refers to the use of processing power within storage devices to perform more complex processing and optimizations. This trend towards "excess compute power in storage systems" has been driven by the increasing performance and decreasing cost of processors and memory, which has caused system intelligence to move from the central processing unit (CPU) to peripherals like storage devices. Storage system designers have taken advantage of this trend by incorporating processing power into storage devices, allowing them to perform tasks such as data compression, encryption, and caching at the storage level. This can significantly improve overall system performance and efficiency by reducing the amount of data that needs to be moved between the storage device and the CPU.

In the paper proposed by Kang et al. [12], one computational storage device called Smart SSD is used in conjunction with a Hadoop MapReduce framework and an in-storage processing engine to perform in-storage processing tasks. The in-store processing engine is an event-driven processing framework that executes tasklets, which are C programs cross-compiled for the ARM processors in the device. These tasklets are loaded onto the Smart SSD and memory space for input and output objects is reserved. By performing some of the processing tasks within the storage device itself, the Smart SSD can help save host CPU, I/O bandwidth, and memory resources.

Wang et al. [60] explored offloading list intersection tasks, which is crucial in search engine and analytic, onto the Samsung SmartSSD CSD research prototype.

They developed an analytical model to understand key performance factors, and their experiments indicated that SmartSSD CSD can enhance list intersection processing while reducing energy consumption.

Torabzadehkashi et al. [61] designed a computational storage device platform featuring a Linux operating system running on a quad-core ARM processor.

The paper proposed by Lee et al. [62] proposes an SSD with an onboard FPGA, which allows for computation to be offloaded within the SSD. This addresses the issue of the disparity between the aggregate SSD throughput and CPU-based computing capabilities, which limits the scalability of modern systems. The authors perform a detailed model-based evaluation to assess the end-to-end performance and energy benefits of SmartSSD for representative data analytic workloads using Spark SQL and Parquet columnar data format. They find that SmartSSD has the potential to significantly improve performance (up to 3.04x) and reduce energy consumption (down to 45.8% of the conventional CPU-based approach).

NASCENT [63] is Near-Storage Acceleration of Database Sort on Smart SSD. It is a method that improves the performance and energy efficiency of sorting large amounts of data on Solid State Drives. According to the provided figures, NASCENT outperforms CPU baselines in terms of execution time and energy consumption when data is stored in DRAM or FPGA. Additionally, it is shown that NASCENT's dictionary decoder kernel has a higher input and total bandwidth compared to CPU inputs for different ranges of input and output bit widths.

SmartSSD CSD are not limited to data analytics; they are also employed to accelerate neural network model training [64], [65].

Additionally, Hedam et al. [66] introduced Delilah, a design for computational storage that uses a host-controller transport protocol implemented through a driver module in the host kernel and a controller on the device. The design and implementation of Delilah is the first publicly described system supporting BSD Packet Filter code offload on a real computational storage platform, OpenSSD Daisy.

Kim et al. [67] proposes a computational storage platform that can significantly boost the performance of a graph-based nearest neighbor search algorithm. The platform uses the SmartSSD which allows computing capabilities to be placed directly on the storage, enabling data to be processed in place with significantly less data movement. The paper also proposes a software/hardware co-design approach to make the target graph-based algorithm more suitable for the computational storage platform. The proposed platform achieves impressive results, outperforming conventional CPU-based and GPU-based systems in terms of speed and energy efficiency.

Chapter 3

Shuffling Algorithm and Attribute Association Aware Shuffling

This chapter describes the initial design of the shuffling algorithm and the attribute association aware shuffling. Some security analysis are also provided.

In this chapter, a novel shuffling approach to enhance the security of relational database storage is introduced. The contributions of the chapter are twofold. First, an attack that is capable of recovering the entire database with only the individual data files in the MySQL environment is identified and discussed. It is essentially a feature of the database management system, but malicious users can employ it to get access to the database data if they can break into the server's operating system. Second, a cryptographically strong algorithm based on permutation to shuffle the data values in each field of the database table is proposed, in order to assure the privacy and reduce the risk of data compromise and identity theft attacks. Randomization is applied in the proposed algorithms to make sure the different fields would follow totally distinct shuffling parameters. Statistical tests are performed to identify highly associated attributes which will be shuffled together as a bundle.

There are two remarkable advantages of the proposed shuffling algorithm: First, even if the attacker is able to defeat the cipher and hack into the shuffled database, he will not be able to detect that the database has been shuffled because the shuffling is attribute association aware. On the other hand, legitimate users of the database will not feel the existence of shuffling or restoring, because these procedures are transparent to them. Second, the tuples extracted from the shuffled database cannot be used for identity theft attacks because they contain mismatching attribute values; if the attacker tries to use the shuffled tuples, it may even trigger an alert and be reported to law enforcement. Therefore, we regard our approach as an extra layer of protection on top of existing encryption schemes.

3.1 The problems

This section identifies and delves into a potential attack capable of recovering the MySQL database using solely the individual data files within its environment. It is essentially a feature of the database management system, but it could be exploited by malicious actors in the event of a successful breach of the server's operating system.

3.1.1 Database Recovery Attack

The MySQL InnoDB storage engine is used here as an illustrative example to demonstrate a potential database recovery attack. As previously mentioned, this method of database recovery can be exploited as an attack technique, enabling unauthorized access to the entire database if an intruder gains possession of the essential data files, which include "db.opt" ".frm", and ".ibd". Importantly, this "attack" can occur even when the hacker does not possess the secret password of the database management system. Furthermore, this "attack" method is designed to be covert, leaving no discernible traces of the breach attempts.

The recovery attack proceeds with the following steps:

1. Establish an empty database on a separate machine. Extract the database name and encoding settings from the "db.opt" file.

2. Create an empty table. Utilize the "*.frm" file to define the table's format. Employ a tool like dbsake [68], to parse the ".frm" file and generate the necessary

DDL (Data Definition Language) statements for table creation.

3. Transfer the “*.ibd;; file to the MySQL data directory. Adjust file permissions in accordance with the operating system’s requirements.

4. Reconstruct the tablespace. By rebuilding the tablespace, the attacker gains unfettered access to the original database without any data loss.

3.1.2 Types of Attackers and Targets

Before delving into the threat model, it is imperative to identify the categories of attackers and the potential attack targets. In the context of compromising the data confidentiality and privacy of semi-trusted databases, there exist two primary categories of attackers [69]:

- **Outside attackers**: These individuals or entities lack the inherent authority to gain direct access to the database. However, they possess the capability to potentially access the database either directly or indirectly through malicious actions. For instance, an outside attacker may exploit vulnerabilities within the operating system to obtain administrator permissions, subsequently gaining access to the database.

- **Inside attackers**: An inside attacker is an individual who possesses the necessary privileges to execute malicious actions. What sets them apart is their potential advantage of being intimately familiar with the network architecture and system protocols. For instance, a maintenance engineer or other employees within a cloud service provider may misuse or abuse their legitimate rights to disrupt or exploit the database for nefarious purposes.

Additionally, three types of attack targets are categorized based on the extent of information known to the attacker:

- **The attacker knows little about the database.** The attacker possesses minimal information about the database. In this scenario, the attacker is unaware of whether the stored database has undergone shuffling. Their objective is to determine the presence or absence of shuffling. For instance, the attacker might manually inspect tuples for any anomalies, such as a female first name associated with a male gender.

- **The attacker already knows that the stored database is shuffled but does not know any correct combination.** Therefore, their goal is to identify some correct attribute combinations, aiming to discover clues that can help them break the shuffling scheme. Certain combinations of data, such as credit card information and names, can be verified.

- **The attacker knows not only the fact that the stored database is shuffled but also a few correct combinations such as the tuple about his personal data.** In this case, their objective aligns with the previous scenario: to identify additional combinations and uncover clues that can aid in breaking the shuffling.

3.1.3 Threat Model

In the provided threat model, the scope encompasses an application responsible for managing sensitive data within a relational database, hosted on a server situated either in a private or public cloud environment. The design primarily targets potential attackers focusing on the database server rather than the client side. It assumes that these attackers lack the permissions to read data on the client side and are restricted to accessing server-side components.

This model considers both types of server-side attackers, who can access almost all kinds of information: the data files and log files of the entire database, all the data stored in memory, and network communications. While certain databases may be safeguarded by user passwords, preventing unauthorized entry to the database server, the threat model acknowledges that all database data files remain accessible. Consequently, the potential exists for these attackers to retrieve sensitive data by reconstructing the database. Additionally, it is assumed that these attackers act passively, refraining from any unauthorized modification or deletion of tuples within the database.

This scenario mirrors real-world use cases in both private and public cloud environments. In the event of database table restoration, attackers could exploit unencrypted information, including details such as names, birthdays, credit card numbers, and even Social Security numbers. Such actions may result in financial losses or, in more severe instances, identity theft, underscoring the critical nature of this threat model.

3.2 Proposed Solution

In this section, the proposed solution is presented in detail, with the core objective of disrupting the relationship between data tuples in a database and the connections among the fields within individual data tuples. The primary goals are to thwart the attacker's ability to compromise the entire database through a single storage server breach and to hinder their capacity to retrieve comprehensive information about a specific individual or entity by tracking all the fields within the same data tuple. To illustrate this concept, we employ a straightforward example demonstrated in Table 3.1 and Table 3.2.

Table 3.1 depicts the conventional method of data storage. In this scenario, each tuple consists of three fields: name, credit card number, and Social Security number. When an attacker infiltrates the storage server and accesses the files stored within the DBMS, they can extract sensitive identification information related to each individual, potentially enabling identity theft attacks.

Table 3.1 Data Tuples Before Shuffling

id	name	credit_card_no	ssn
1	Sipes	4251546446736274	473-34-5897
2	Shank	4703038231755507	768-12-5834
3	Howell	5393619347816820	583-35-5681
4	Kohler	6011220281842771	981-34-6481
5	Ramona	4688678394067007	781-59-1384

Table 3.2 Data Tuples After Shuffling

id	name	credit_card_no	ssn
1	Sipes	6011220281842771	583-35-5681
2	Shank	5393619347816820	981-34-6481
3	Howell	4688678394067007	768-12-5834
4	Kohler	4251546446736274	781-59-1384
5	Ramona	4703038231755507	473-34-5897

Table 3.2 demonstrates the core concept of the approach being proposed. The method involves applying secure shuffling based on cryptographic-strength permutation to each field. This process relies on a deranged permutation sequence securely stored in an external key vault server. It is essential to emphasize that, to ensure the efficacy of shuffling, exclusively *deranged permutation* is considered. This type of permutation rearranges elements in a manner that ensures no element remains in its original position. Consequently, even if an attacker successfully breaches the storage system and acquires a copy of the file, deriving the corresponding credit card numbers and Social Security numbers for individuals becomes a formidable challenge. Any attempt by the attacker to use attribute values from the same tuple is likely to result in errors, potentially exposing their malicious activities and raising the risk of detection by law enforcement authorities.

Next, the main algorithms in the solution will be introduced, encompassing the shuffling algorithm, the algorithm for the discovery of associations between attributes, and the restoring algorithm.

3.2.1 Shuffling Algorithm

Let l represent the total number of tuples in the relational database table, which is partitioned into equal-sized blocks. We denote w as the number of elements contained within one block, and $(k^{[1]}, k^{[2]}, \ldots, k^{[n]})$ as the random deranged permutation sequences used as the shuffling parameters. The default value assigned to w is 20, except for the last block, which may contain fewer elements. The selection of the value 20 is based on a balance between ensuring a sufficiently large key space and managing the shuffling process with a reasonable level of overhead.

Multiple level shuffling. Multiple levels of shuffling can be applied based on the total number of tuples within the table. The shuffling parameters, represented as the set $(k^{[1]}, k^{[2]}, \ldots, k^{[n]})$, delineate the manner in which shuffling occurs across n levels. If the number of "elements" at a given level exceeds 20, an additional level is introduced. Notably, these 20 "elements" are treated as a single element in the newly added level, which prevents exponential growth in the value of n. For instance, a database table containing 3 million tuples can be effectively managed with just 5 levels of shuffling, as 20^5 exceeds 3 million.

Each deranged permutation sequence $k^{[i]}$ for level i in $(k^{[1]}, k^{[2]}, \ldots, k^{[n]})$ is randomly selected from the set of all possible deranged permutation sequences. To achieve this, we have designed and implemented a function denoted as $dp(w)$. This function is responsible for generating a random deranged permutation sequence of 0, 1, 2, ..., $w - 1$. The operation of the $dp(w)$ function involves the uniform random generation of a permutation sequence, followed by a check to ascertain if it qualifies as a deranged permutation. If it does, the function outputs this permutation; otherwise, it repeats the random generation process until a deranged permutation is obtained. Empirical testing has demonstrated the practicality of this function. Tests involving the generation of 10K, 50K, 100K, and 500K deranged permutations have consistently shown that, on average, only around 1.6 attempts of random permutation generation are needed to obtain a random deranged permutation.

To visualize the shuffling process, Figures 3.1 and 3.2 illustrate how shuffling is performed, considering the permutation sequence and the position of each element within the block.

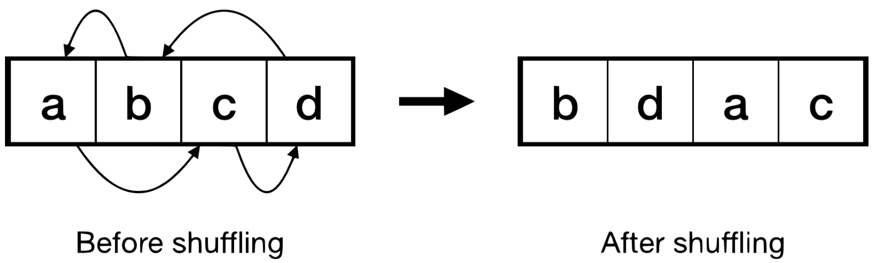

Figure 3.1 Example of shuffling at the first level.

Figure 3.1 provides a visual representation of the shuffling process at the first level. To elucidate the steps involved in shuffling, we use a block of 4 elements as an example. Given that $w = 4$, there exist a total of $!w = 9$ deranged permutations, where $!w$ denotes the sub-factorial and represents the count of deranged permutations.

In this process, the original block is shuffled in accordance with the chosen permutation sequence. The permutation sequence $k^{[i]}$ specifies that the element currently situated at the position indicated by the value of $k_j^{[i]}$ will be relocated to position j following the shuffling. For instance, consider a block containing elements (a, b, c, d) and a permutation sequence $k = (1, 3, 0, 2)$. According to this sequence, the element at position $k_0 = 1$, which is element b, will be relocated to position 0, while the element at position $k_1 = 3$, which is element d, will be moved to position 1. After shuffling has been performed for every position, the block transforms into (b, d, a, c). The same shuffling process is applied to all other blocks, with the difference that another sequence will be used for the last block if its number of elements does not align with the regular value of w.

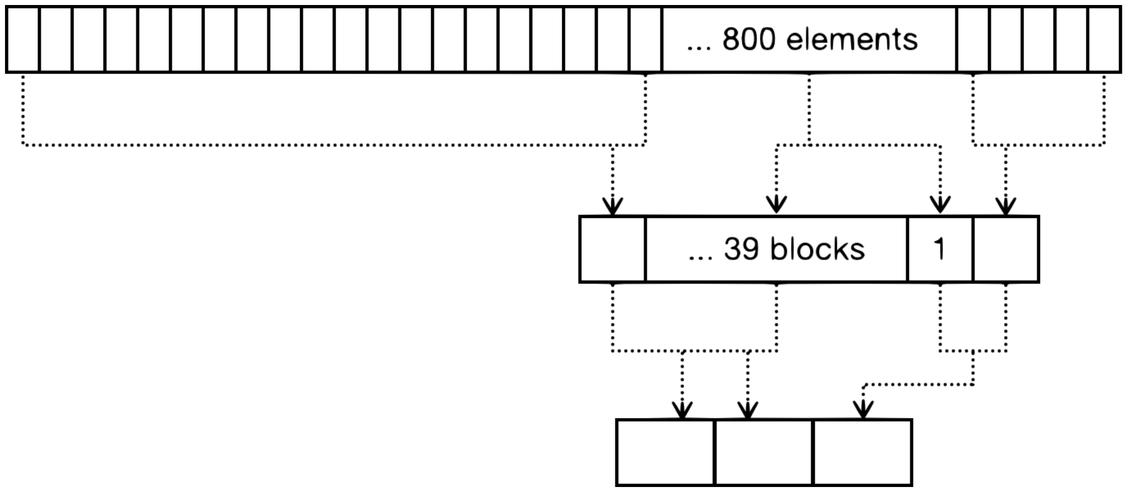

Figure 3.2 Example of shuffling the blocks in multiple levels.

Figure 3.2 provides an illustration of the shuffling process across multiple levels, using an example with 825 elements in the field (i.e., $l = 825$). In this scenario, there are 41 full blocks, each containing 20 elements, and 1 block containing the remaining 5 elements.

After shuffling within each block at the first level, we progress to level 2. The shuffling process at this level resembles that of the first level in Figure 3.1, but the unit of shuffling here comprises the blocks, each consisting of several blocks from the previous level. The 42 blocks from the first level are considered as 3 blocks at level 3.

Regarding the permutation sequences, the sequence $k^{[1]}$ for the first level is randomly chosen from the range of !20 deranged permutations, while the range of the last block's permutation sequence is !5. For level 2, sequence $k^{[2]}$ is one random sequence from !20 deranged permutations, and the range of permutation sequences for the last 2 remaining blocks is !2. In level 3, shuffling can be directly applied to the combined three "elements" without requiring further levels.

For each field (attribute) within a given table, the algorithm selects a distinct set of random permutation sequences $(k^{[1]}, k^{[2]}, \ldots, k^{[n]})$. This ensures that each attribute undergoes independent shuffling, effectively breaking the original association between attribute values belonging to a tuple. For instance, let's consider a tuple from a database that stores user details for an online merchant's website: **User("Mike", "Johnson", "01011965", 4012888888881881, "04/17", ...)**. This tuple comprises fields such as first name, last name, birthday, credit card number, card expiration date, and other fields in their respective order. After shuffling, the credit card number field could now contain Tina's credit card number, while the expiration date field might reflect Fisher's card expiration date. Consequently, the shuffled table appears garbled but remains indistinguishable from a genuine table to potential attackers.

3.2.2 Discovery of Attribute Associations and Bundle Shuffling

A notable challenge with shuffling arises when dealing with highly related attributes within a tuple. Shuffling these attributes independently can sever their inherent relationships. For instance, consider email addresses chosen by users, which are often linked to their first and last names or initials. Similarly, gender is highly associated with the first name. In a healthcare provider's database, diagnosis codes related to obstetrics and gynecology must be inherently associated with female patients.

When such highly correlated attributes are shuffled individually and independently, the resulting discordant attribute values within the same tuple can be easily discerned by an attacker. This exposure may indicate the presence of shuffling. To mitigate this issue, bundling these closely associated attributes together and shuffling them as a group can create a more deceptive appearance for the shuffled table. This approach makes it less likely for an attacker to recognize the table as valuable data, enhancing the security of the shuffling process.

Association-aware bundle shuffling is achieved through a combination of two methods. Firstly, attributes that are semantically related, such as those mentioned in the previous paragraph, are manually identified. These related attributes are then shuffled together using the same set of permutation sequences. Secondly, to avoid over-looking any hidden statistical correlations between attributes that may not appear to be semantically related, we introduce an algorithm designed to discover associations between attributes by quantifying the statistical strength of their relationship.

The statistical strength between two attributes can be measured using various techniques, depending on the data type of each attribute. For instance, when both attributes are categorical, the χ^2 Chi-squared test can be employed [70]. If both attributes are numerical, the Pearson Correlation Coefficient (PCC), also known as Pearson's r, is a suitable test [71]. When one attribute is categorical and the other is numerical, the analysis of variance (ANOVA) test can be utilized [72].

The pseudocode of the association discovery algorithm is presented in Algorithm 1.

In the algorithm, the χ^2 test, Pearson correlation coefficient test, and ANOVA test were utilized to discover associations among the database attributes. Each attribute in the database was modeled as a vertex in a graph denoted as G, initially lacking any edges. The statistical test involved the analysis of two complete attributes, producing a statistical strength value, commonly referred to as the P-value. If the P-value fell below the threshold of 0.05, a standard significance level, the algorithm established an edge connecting the corresponding vertices in graph G for those attributes. This process was systematically applied to all possible pairs of attributes. Upon the algorithm's completion, the final graph G was obtained. Within this graph, each strongly connected component was designated as a bundle, and the attributes contained within the same bundle were grouped together for shuffling. These associated attributes were then shuffled using the same set of permutation sequences $(k^{[1]}, k^{[2]}, \ldots, k^{[n]})$. This approach ensured that attributes displaying statistical associations were collectively processed during the shuffling procedure.

Algorithm 1 Discovery of association between attributes

Input: $R(A_1, \ldots, A_n)$ tuple R with attributes $A_1, \ldots, A_n$

Output: $S = \{S_1, \ldots, S_k\}$ where S_i is a set of attributes such that $S_i \in \{A_1, \ldots, A_n\}$

```
def connect_vertices(A1, A2, w, E):
  if w ≤ 0.05 then
    E.append((A1, A2))
  end if

V = {A1, ..., An} // vertices
E = { } //edges
G = (V, E) // draw an undirected graph
for i in 1 to n do
  for j in 1 to n do
    if i ≠ j then
      if Ai and Aj are categorical then
        x = measure Ai and Aj using Chi-squared test
        connect_vertices(Ai, Aj, x, E)
      else if Ai and Aj are numerical then
        y = measure Ai and Aj using PCC test
        connect_vertices(Ai, Aj, y, E)
      else
        z = measure Ai and Aj using ANOVA test
        connect_vertices(Ai, Aj, z, E)
      end if
    end if
  end for
end for
S = strongly connected components sets in G
return S
```

3.2.3 Restoring Algorithm

An integral component of our approach is the restoring algorithm, which reverts the shuffled database to its original state, enabling it to be presented to authorized users. The inverse permutation sequence, denoted as $-k^{[i]}$, contains the same elements as the permutation sequence $k^{[i]}$ used in the shuffling process but with opposite implications. In other words, the element situated at position j will be relocated to the position indicated by the value of $k_j^{[i]}$.

3.2.4 Key Vault

The randomly generated permutation sequences used for shuffling the fields can be encoded into a wallet file in plaintext and subsequently encrypted using symmetric encryption ciphers like Triple DES and AES. Once the shuffling process is applied, the security of the wallet file becomes paramount within the system. To address this concern, we have incorporated an existing application known as the key vault.

The key vault server serves as a protective mechanism for cryptographic keys and secrets, encompassing elements such as authentication keys, encryption keys, and even credential files, by centrally managing them. By segregating the wallet file and the database into distinct cloud environments, we significantly enhance data confidentiality. Even in scenarios where malicious employees are present, their ability to decrypt the wallet file is greatly restricted.

Without access to the correct secret key required for decrypting the shuffling parameters, an attacker cannot retrieve genuine data from the database. Furthermore, the implementation of attribute association-aware bundle shuffling enhances the appearance of the shuffled database, making it deceptively "genuine". For instance, with fields such as full name, credit card number, Social Security number, address, birthday, and cell phone number, any combination of values from these fields may appear plausible, rendering it exceedingly challenging to discern that shuffling has taken place.

3.3 Analysis

In this section, an analysis of the security of the proposed shuffling algorithm is presented, encompassing key space, key sensitivity, guess countering, and performance evaluation.

3.3.1 Key space analysis

In cryptography, one cryptosystem's key space means the set of all possible distinct keys based on key length. In the shuffling algorithm, the key space analysis is to show the range of all possible permutations of the shuffling parameters and to show the difficulty to successfully conduct the brute-force attack on shuffling algorithm. Theoretically, the key space should be large enough to make the brute-force attack infeasible, and the key space of the shuffling algorithm can be assessed from the following two perspectives: 1) arbitrary key space, and 2) global key space. Arbitrary key space stands for the key space of one single attribute's shuffling parameter. If the brute-force attack succeeds, the adversary can restore the original order of all tuples in this attribute. This restoring doesn't violate confidentiality (if the data are encrypted), integrity, and availability since the only difference is the order of those tuples. However, this may bring some concerns on the privacy preservation due to the revealing of data order. For example, in the relation *Salary* in which all attributes are ordered by ascending salary amount, shuffling on the attribute could remove the underlying relationship that Bob's salary is lower than Alice's because the *id* of Alice is bigger. As to the global key space, it indicates the maximum number of shuffling parameters to try for restoring the entire relation.

For one relation with m tuples and n attributes, the size of the global key space is at least

$$\left(m!\sum_{i=0}^{m}\frac{(-1)^i}{i!}\right)^n \tag{3.1}$$

which is expressed as the $(!m)^n$ in combinatorics. For the sake of simplicity, the factor of multi-level shuffling is not considered, so the actual key space is even larger for multi-level shuffling. The arbitrary key space for any attribute with m tuples would be $!m$ which is unfolded as

$$m!\sum_{i=0}^{m}\frac{(-1)^i}{i!} \tag{3.2}$$

The above expression clearly indicates that the key space is highly dependent on the scale of relation. Because of the factorial in the global key size function, a small increase in the number of tuples will lead to huge growth. For instance, the key space of the relation with 5 tuples and 2 attributes is 1936, and when the number of tuples rises to 20 tuples and if there are 5 attributes, the size of the global key space dramatically grows to $5.74e+89$. This number is even larger than the estimated total number of particle in the observable universe [73]. This key space is sufficiently large to protect the shuffling from brute-force attacks trying to guess the correct shuffling parameters.

3.3.2 Key Sensitivity Analysis

Effective data masking method, like the shuffling algorithm we employ, should exhibit sensitivity to even minor alterations in the secret keys. Confusion and diffusion were identifier by Claude Shannon in his paper [74]. In Shannon's original definitions, confusion refers to making the relationship between the key and the ciphertext as complex as involved as possible. It means that a good encryption or data protection technique should be sensitive to a tiny change on the secret key. As for the shuffling algorithm, it's necessary to make sure that all the tuples move after the shuffling.

In most instances, any modification to the shuffling parameters $(k^{[1]}, k^{[2]}, \ldots, k^{[n]})$ results in a rearrangement of tuples. To illustrate this sensitivity, we define a "coincidence" as a tuple in which one or more attributes remain in the same position after shuffling. Furthermore, a "strong coincidence" is a shuffled tuple in which either two or more of its attributes originate from the same tuple before shuffling or the tuple qualifies as a "coincidence".

If the number of tuples is small enough to carry out the shuffling with one level, it is straightforward that there is no *coincidence* or *strong coincidence* in the shuffled data since the definition of deranged permutation requires that no element appears in its original position. However, multiple levels of shuffling could possibly move the tuple back to its original position theoretically. To assess this sensitivity, experiments were conducted on four distinct data tables, each containing a different number of tuples: 5,000; 10,000; 50,000; and 100,000, all with 10 attributes. Each dataset will be shuffled for 1000 rounds with different randomly generated shuffling parameters and then each shuffled dataset will be compared to the original dataset for counting the number of *coincidence* and the number of *strong coincidence* accumulatively. The results revealed that, across all cases, the average occurrence probabilities of both *coincidences* and *strong coincidences* remained at zero. Therefore, it can be confidently concluded that the proposed shuffling algorithm is highly sensitive to modifications in the shuffling parameters, with any changes resulting in the generation of an entirely distinct shuffled table.

3.3.3 Countering Guessing Attacks

Three types of attack targets were delineated in subsection 3.1.2 based on the extent of prior knowledge possessed by the attacker regarding the database. In this subsection, an examination is conducted to elucidate how the proposed algorithm can effectively counter these attacks.

In the case that the attacker doesn't know if the stored database table is shuffled or not, one method the attacker can apply is to open the database and find out all the attributes of a tuple, identify the attributes that should or may be associated, and scan through the database table to see if there is any tuple that contains non-matching attribute values. For instance, the attribute related to obstetrics and gynecology (OB-GYN) treatments should be associated with the attribute sex; otherwise, the attacker can find the inconsistency. Through the bundle shuffling, the proposed shuffling algorithm can resist this kind of guessing attack by bundling semantically related attributes, so that the attributes like "ob-gyn_treatment" and "sex" will be shuffled together. Moreover, manually bundling is also supported in the design. For example, it is required that each issuer entity's payment card number must follow the standard ISO/IEC 7812 [75], which allows an attacker to find mismatching attribute values such as the attribute "credit_card_no" has a card number belonging to VISA but the attribute "bank_name" has the value "Mastercard". Since the credit card number has the feature of uniqueness that the computed association between attribute "credit_card_no" and attribute "bank_name" is not that tight, manually bundling is a way to complement this.

If the attacker knows that the database is shuffled but does not know any correct tuple, the attacker's target will be to locate at least one correct tuple or more. The attacker has to randomly pick some information combinations such as {name, birthday, phone_no, credit_card_no, zip_code} to try on the merchant website. Thanks to the huge key space as given in subsection 3.3.1, it's extremely difficult to find out even one correct tuple.

If the attacker is aware of the presence of shuffling along with a few correct combinations such as his own data, then his target becomes to locate more combinations and find a chance to reveal the entire database. Because the default number of elements in one block as 20, the attacker needs to master at least 19 tuples in this block to ascertain the secret shuffling parameter for this level. Otherwise, he can only use the brute-force method and verify one by one. As discussed in subsection 3.3.1 that the size of the key space ascends dramatically along with the number of tuples, it is not feasible for the attacker to restore the original tuples. Even if the attacker happens to know all the tuples in the same block (which is highly unlikely, because if so the attacker must be very familiar with the target database and does not need to launch such guessing attack) and thus be able to compute the shuffling parameter for this level and restore all other blocks in this level, the attack still has no idea about the shuffling parameter for the next higher levels if the total number of tuples in the table is large enough for multiple levels. Based on the preceding analysis, it can be concluded that the approach outlined in this dissertation is robust in thwarting the attacker's attempts to achieve the three identified targets.

3.3.4 Performance Evaluation

One simple prototype of proposed shuffling algorithm was implemented on a PC with 4.0GHz Intel Core i7-6700K CPU and 32GB RAM running Ubuntu 18.04. The data structure used to store the data is B+ tree [76]. Five random datasets was generated with sizes of 10K, 50K, 100K, 500K, and 1 million items to build the primary key index, and each item contains an index of type integer and data of type string. For every dataset, the evaluation measured and compared the elapsed time of two cases: when shuffling is not applied on the dataset, and when shuffling is applied on the dataset (that is, shuffle the tuples before storing to hard drive and restore the tuples to normal order after loading to memory). The storing and loading of each dataset for 10 rounds were executed, during which the average elapsed time was recorded. Additionally, the ratio of the time expended on the shuffling and restoring procedures to the overall elapsed time was calculated, as summarized in Table 3.3. Notably, the outcomes of these experiments reveal that the shuffling and restoring procedures within our algorithm entail only a minimal time investment in comparison to the time required for the conventional storage and retrieval of data. This observation reaffirms the feasibility and practicality of our algorithm, underscoring its efficiency and suitability for real-world applications.

3.4 Contribution

In this chapter, the challenges and opportunities in utilizing a shuffling algorithm is studied to enhance the security and privacy of storing relational database. The proposed approach not only individually works to mask the data relationship, but

Table 3.3 Extra time of shuffling and restoring

Number of Items	**10K**	**50K**	**100K**	**500K**	**1 million**
Elapsed time without shuffling or restoring (ms)	50.51	284.27	586.44	3262.31	6673.19
Elapsed time with shuffling (ms)	54.22	292.21	598.72	3299.15	6727.93
Elapsed time with restoring (ms)	54.17	292.45	599.98	3298.60	6733.24
Ratio of time with shuffling	7.34%	2.79%	2.09%	1.13%	0.82%
Ratio of time with restoring	7.24%	2.88%	2.31%	1.11%	0.90%

also provides an additional layer of security along with encryption.

Chapter 4

Shuffling Column-Oriented Relational Database

In Chapter 3, a robust shuffling algorithm with cryptographic strength is introduced. The primary objective of the algorithm was to shuffle the data values within each attribute of the database relation, thereby safeguarding against data confidentiality violations that may arise from compromise of the database storage disk. However, considering the computational overhead associated with shuffling in the row-oriented storage scheme, our algorithm was originally designed for offline storage rather than the online running instance of the database management system.

In this chapter, a novel improvement mainly on the efficiency is described. The improvement combines the shuffling algorithm with a column-oriented storage strategy to augment the security and efficiency of relational database management systems. It also utilizes a set of two-column association tables to document the disk locations of the authentic data alongside their corresponding indexes. This table allows the database to perform shuffling operations without the need for physically relocating the data, thereby leading to substantial reductions in both time and system resource consumption.

Considering the requirements of supporting online shuffling in addition to offline shuffling, the following research challenges have been identified:

RC1: How can shuffling be performed while the database is running?

RC2: What measures can be taken to improve the efficiency of the shuffling algorithm?

RC3: How to strike a balance between the security requirements and the performance requirements?

4.1 Design

4.1.1 Threat Model

Attackers can be classified into three categories based on their access points and privileges: intruders, insiders, and administrators [77]. An **intruder** is an individual who gains unauthorized access to a computer system in order to extract valuable information. An **insider**, belonging to the trusted user group, seeks unauthorized access to restricted information. An **administrator**, possessing system privileges, abuses his rights to extract valuable information. These attackers can employ at least four attack strategies, as discussed below:

- Physical storage attacks directly target the storage media and hardware architectures used by database management systems and their backup systems. Examples include cloning the disk or disrupting the power supply.

- System storage attacks aim at compromising the storage of database management systems through methods other than attacking the hardware directly. This can involve exploiting administrator privileges or hacking into the database log server.

- Memory attacks focus on exploiting the memory where database cache data are stored. Attackers attempt to manipulate or gain unauthorized access to the data in the memory.

- Transmission attacks occur during the data transmission between the database server and the database clients. Attackers target this communication channel to intercept or manipulate the data being transmitted.

In the threat model, the following assumptions are made:

- *The database management system is trusted.*[78] We place our trust in the database management system to accurately carry out operations such as disk reading and writing, ensuring that no vulnerabilities are created for attackers to exploit. Additionally, we assume that the operating system housing the database management system is free from malicious software and that the memory space utilized by the database management system is adequately safeguarded. Hence, we do not need to focus on protecting against memory attacks.
- *The communications among the database management system clusters are trusted.* This aspect pertains to network security, and encryption during data transmission, such as SSL and IPSec, is extensively employed to safeguard communications. Consequently, our focus does not need to be directed towards protection against transmission attacks.
- *The hardware architectures or software systems where the database management system is located are vulnerable to compromise.*[78] Trusted database management systems commonly rely on the storage system offered by the underlying operating system. However, the vulnerability of the operating system creates a potential avenue for attackers to illicitly access the valuable information stored within the database.
- *The data in the database management system are beneficial from the attacker's perspective.* Network attacks can provide benefits to the attackers, which may include unauthorized access to valuable information, financial gain through theft or fraud, and competitive advantage through industrial espionage.

4.1.2 Deranged Permutation Generation

As mentioned in Chapter 3, shuffling relies on a deranged permutation sequence. In order to improve the efficiency of the shuffling, in which deranged permutation generation is a necessary step, a new design is proposed. The new design can generate a deranged permutation in constant time, based on the work of Korsh and LaFollette [79].

Previously, the method involved randomly generating a permutation and then testing if it was deranged. However, as the tuple size w increases, the hit ratio (the probability of generating a deranged permutation) decreases due to the ratio R between the number of permutations N_p and the number of deranged permutations N_{dp}, where $R = N_{dp}/N_p$. Consequently, generating a deranged permutation becomes increasingly time-consuming.

To enhance the efficiency of the shuffling process, an integration has been implemented, combining the constant-time deranged permutation algorithm with a memorization algorithm. This integrated approach optimizes the time required to obtain deranged permutation sequences. When a deranged permutation sequence for a tuple size of w_i is initially requested, all potential permutation sequences for w_i are calculated and stored in a sequence pool. Subsequent requests involve selecting a sequence from the pool, eliminating the necessity for re-computation. This enhancement notably boosts the efficiency of the shuffling algorithm itself and effectively addresses the challenge RC2.

4.1.3 Data Storage with *BAIT*

In disk-optimized database management systems, data is typically managed using a combination of block-level and file-level access to the disk, rather than direct control of disk blocks by the database engine or reliance on the operating system's file system for data storage [80]. Data files are treated as raw disk storage, with the operating system responsible for mapping logical blocks within data files to corresponding physical blocks on the disk. The block size may vary among different database management systems, resulting in the possibility of multiple logical blocks residing within a single data file, each assigned a unique block number for future reference. To minimize the need for multiple hard disk I/O operations, these blocks are first transferred to memory. Within memory, pages store the content of these blocks, and a component known as the "buffer manager" is tasked with the retrieval of blocks into the buffer cache and the writing of pages back to disk blocks in the event of any modifications [81]. Consequently, the terms "page" and 'block" are sometimes used interchangeably to denote the fundamental unit for operations on data files. Thus, data files are structured as collections of pages, with each page assigned a distinct identifier [82]. The prevailing layout scheme for a page is often referred to as a "slotted page", consisting of a header, a slot array, and values. The slot array serves as a mapping between the slot identifier (`slot_id`) and the starting position offset of the corresponding value.

The main difference between row-oriented storage and column-oriented storage is how the data are organized in data files [83]. In row-oriented storage, one tuple which consists of multiple attributes occupies one slot in the page file, so that the `slot_id` can be used for future addressing and each attribute can be traversed based on the attribute length predefined in relation scheme. In the design, the page files only store the values from one single attribute, and each value will be assigned one `slot_id`. Because of the fixed size of one page along with the growing number of tuples in the relation and the requirement of atomicity, the size of one page is usually a few kilobytes such as 4KB in SQLite and Oracle, thus one attribute may be stored in multiple pages. Given the `page_id` and `slot_id`, every value can be addressed.

For the purpose of improving security and preserving privacy, shuffling algorithm is applied to attributes. As mentioned earlier, the attributes in one relation may be located on different pages due to the size limit, thus directly moving the attributes according to the permutation generated by the algorithm is extremely inefficient for the following reasons: 1) high diffusion feature and deranged permutation of our algorithm will move every attribute to another location, which requires a mass of I/O operations on data files, and 2) the newly inserted tuples need the shuffling to provide the security and privacy on those new tuples, and frequent insertions will hugely increase the time of reading and writing accesses.

Instead of directly moving the values themselves, our approach is inspired by the physical data model of MonetDB [56]. MonetDB represents relational data using Binary Association Tables (BATs) consisting of two columns. The right column stores the actual value, while the left column contains the object identifier (`OID`) of the corresponding tuple. In a relation R with m attributes, there are m BATs, and attribute values within the same tuple share the same `OID`. The `OID` values are generated in an ascending sequence during tuple insertion, indicating the explicit position of the tuple. In this design, the Binary Association Index Tables (BAITs) is proposed in the physical data model. As illustrated in Fig. 4.1 which depicts the architecture, the BAIT is positioned between the Indexing module and Record module, responsible for managing querying indexes and unordered data records, respectively.

There are two main differences between MonetDB's BAT and our BAIT:

- In BAIT, the `OID` in the left column represents the iterator of either the indexing data structure or the record data structure. However, unlike the ascending sequence used in MonetDB, our design generates random and hashed `OID` values.
- The right column in BAIT does not store the actual data; instead, it holds the address of the storage location.

Left Column: The `OID` in the left column serves as the iterator for either the indexing data structure or the record data structure, depending on whether the column is indexed or not. To ensure the persistence of the iterator, each record or

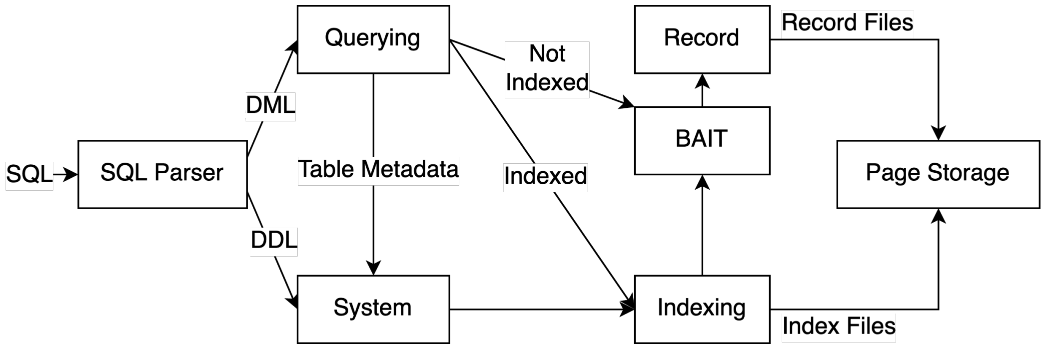

Figure 4.1 System architecture of the database with BAIT integrated.

indexing iterator is assigned a unique identifier. To generate the `OID`, a highly efficient hash algorithm called xxHash [84] is employed. This algorithm operates at RAM speed limits, enabling fast processing. The value range of the random `OID` generator does not need to be excessively large to maintain optimal performance. To ensure the uniqueness of the `OID`, generated random number and the current timestamp are concatenated. This concatenation is then passed through the xxHash algorithm, which guarantees both uniqueness and speed. The random number acts as a salt for the hash function, enhancing the security of the hash result and preventing attacks such as rainbow table attacks [85]. Since the hash result is not used for integrity verification, the randomness of the salt does not pose any issues for storage and reuse.

Right Column. The data stored in the right column are the concatenated address only, represented as `page_id + slot_id`, rather than the actual value. This unified indexing approach significantly reduces the frequency of I/O operations by operating solely on the BAIT files, instead of reading and writing all the page files during the shuffling process. The BAITs can be efficiently utilized in memory, allowing for faster shuffling operations. Moreover, if the size of the BAIT files exceeds the available memory, cache replacement policies such as LRU (Least Recently Used) [86] can be employed to manage memory usage effectively. This ensures optimal performance even when the BAIT files cannot be entirely held in memory.

One example of BAIT is shown in Fig. 4.2. With the perspective of the implementation of the database system, the BAIT is typically located between the Indexing Module and Record Management Module. Therefore, the left column of the BAIT is connected to the Indexing, while the right column of the BAIT holds the address of the record, which is the unique identifier of the record in Record Management Module. By only shuffling one column of the BAIT, the shuffling of the attributes of the relation can be achieved without actually moving the physical data in storage medium.

e2293b2f	page 5, slot 38
2b752fcc	page 5, slot 39
c5a92f8c	page 5, slot 40
4497a5e6	page 15, slot 2
1ed60055	page 15, slot 3
c4988d24	page 88, slot 9
cb685b43	page 88, slot 10
a31eed39	page 88, slot 10

Figure 4.2 Example Data Representation of BAIT

4.1.4 Shuffling and Restoring

Based on the BAIT storage format, the entire shuffling on the relation doesn't have to touch the actual data files; only the left column of BAIT files will be changed. Comparing to the actual data, BAIT files are much smaller, so it is feasible and efficient to put BAIT files in memory during the running of the database management system. In an extreme case, every data insertion or update will trigger the full shuffling to guarantee the data security and privacy. Another extreme case is to execute the shuffling only before the shutdown of the database management system, which only provides the protection on offline storage. To balance the performance and security, a time interval can be set as the countdown to trigger the shuffling.

The efficiency of attribute statistical association bundle shuffling has significantly improved due to the adoption of column-oriented storage. In the computation of association strength, all values within a column are involved. However, when using a row-oriented storage scheme, this computation incurs a substantial amount of file I/O operations. In contrast, the column-oriented storage scheme benefits from a larger page size, such as the 1MB page size in Amazon Redshift. Consequently, measuring statistical strength can be achieved with just a few I/O requests, as the related columns are stored together. By leveraging the advantages of column-oriented storage, bundle shuffling processing can be accelerated.

4.2 Implementation

This section describes the implementation of the database which combines the shuffling algorithm and the column-oriented database.

4.2.1 Parameter Settings

It is necessary to stay with consistent shuffling parameters for the transparent restoring (i.e., unshuffling) of the data. For the shuffling of each relation, $\mathbb{K}$, as a three-layer structure, records the shuffling parameters as shown in Fig. 4.3. In the first layer, the shuffling parameters are represented as $\mathbb{K} : (K0, K1, ..., Kn)$ in which $K0$ means the shuffling parameters for the first attribute and Kn means the shuffling parameters for the last attribute. Coming to the next layer, each $k^{[i]}$ indicates the shuffling parameters for each level. The sequence (k_0, k_1, ..., k_n) stands for the shuffling parameters inside each block. Among these parameters, only the sequence of (k_0, k_1, ..., k_n) is generated, and the sequence plays the role of leaf in the tree. Accordingly, the database engine randomly generates the shuffling sequence (k_0, k_1, ..., k_n), and constructs all the shuffling sequences together.

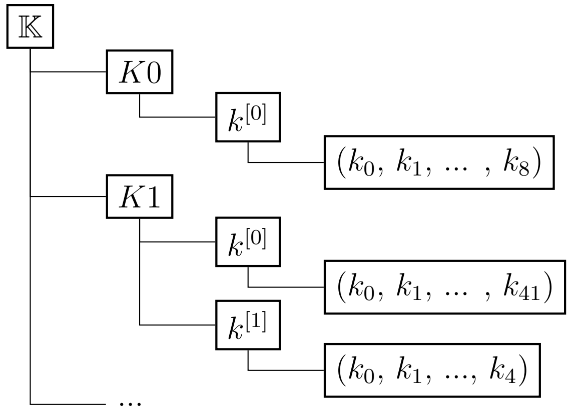

Figure 4.3 General view of the shuffling parameter collection for one relation. $\mathbb{K}$ stands for root, next level contains the shuffling parameters of each attribute, then next level contains the shuffling parameters for each level.

4.2.2 Datasets and Implementation

A portion of the IMDB dataset was utilized for evaluating the performance of our proposed design. The dataset comprises alphanumeric unique identifiers of individuals, along with essential information. It was obtained from [87] and has a size of approximately 400 MB. Initially, the dataset was in CSV format, but we converted it to SQL files for improved convenience during the analysis.

Both the column-oriented storage scheme and row-oriented storage scheme are implemented for the purpose of comparison. The experimental environment is equipped with an Intel i7-10700 CPU and 32 GB of memory, running Debian 11. The shuffling parameters are stored in a separate machine functioning as the security key vault. Both the dataset server and key vault are connected to the same switch. Therefore, the time required for reading and storing the shuffling parameters in the actual product may vary depending on the connection between the database server and the key vault server.

4.2.3 Shuffling Overhead

To simplify the performance evaluation process, the following assumptions are made: 1) The shuffling process is triggered based on the number of tuples that are inserted, deleted, or updated in the database; 2) All columns in the relation are indexed to optimize query performance. The system performance was evaluated using a collection of insertion and query statements denoted as $\mathbb{Q}$. The overall execution time without shuffling enabled is denoted as $T^{\mathbb{Q}}_{\text{OFF}}$, while the execution time with shuffling enabled is denoted as $T^{\mathbb{Q}}_{\text{ON}}$. To demonstrate the overhead of shuffling, the evaluation tests is conducted using 50,000 and 100,000 statements for INSERT and SELECT operations. The average results of 10 rounds are shown in Table 4.1.

Table 4.1 Running Times When Shuffling is Disabled and Enabled.

Operation	# of Statements	$T^{\mathbb{Q}}_{\text{OFF}}$	$T^{\mathbb{Q}}_{\text{ON}}$	Difference
INSERT	500000	5.46 s	5.98 s	0.52 s
INSERT	1000000	9.49 s	10.18 s	0.69 s
SELECT	500000	281.45 ms	286.36 ms	4.91 ms
SELECT	1000000	376.29 ms	377.87 ms	1.58 ms

Based on the results, the following observations can be made: 1) The time consumed by the shuffling process is small compared to the overall execution time. It indicates that the shuffling operation has minimal impact on the system's performance. 2) For INSERT operations, the time taken for shuffling increases proportionally with the number of tuples being inserted. This demonstrates that the shuffling process scales well with the size of the dataset. 3) Enabling shuffling for SELECT operations does not significantly affect the execution time. This is because no additional steps are performed during the querying process for SELECT operations, resulting in minimal overhead. These results confirm the practicality and scalability of the amelioration on column-oriented storage scheme.

4.3 Analysis

This section provides an analysis on the security of our the shuffling algorithm in terms of correlation coefficient and distribution.

4.3.1 Correlation Coefficient Analysis

The correlation coefficient analysis can be used to measure the strength of a linear association between two variables, which are an unshuffled table and a shuffled table in our case. The correlation value 1 means a perfect positive correlation, value 0 means no correlation, and the value -1 means a perfect negative correlation. The correlation coefficient can reflect to what extent the shuffling algorithm strongly resists the statistical attack [88], so it's a success if the correlation coefficient value of our algorithm ranges between 0 and -1 with high probability. Due to the randomness of shuffling parameters, the output of every new round is rarely the same as last round, so the correlation coefficient values are fluctuant. Experiments were conducted on data tables containing different number of tuples: 5K, 10K, 50K, and 100K respectively, with the number of attributes set as 10 for all four cases. The results of 1,000 rounds show that the probability of the correlation coefficient value falling between 0 and -1 is around 75%. This means that the shuffling algorithm can effectively resist the statistical attack.

4.3.2 Distribution analysis

In Shannon's original definitions, confusion and diffusion are two properties of a secure cipher. Confusion refers to making the relationship between the secret key and ciphertext as complex as possible, while diffusion refers to dissipating the statistical structure of plaintext over the bulk of ciphertext which basically means making the relationship between plaintext and ciphertext as complex as possible. In the key sensitivity analysis of Chapter 3, complex relationship between secret key and ciphertext has been described, which is the sensitivity between our shuffling parameters and permutations. To show that the shuffling algorithm achieves good diffusion, a distribution analysis is undertaken. Each tuple is put in one attribute as one dot into one axis system in which the X-axis represents the index of the tuple in the attribute, and Y-axis represents the attribute value. By comparing the locations of those dots before and after shuffling, the analysis can assess whether the relationship between **plaintext** and **ciphertext** is well diffused.

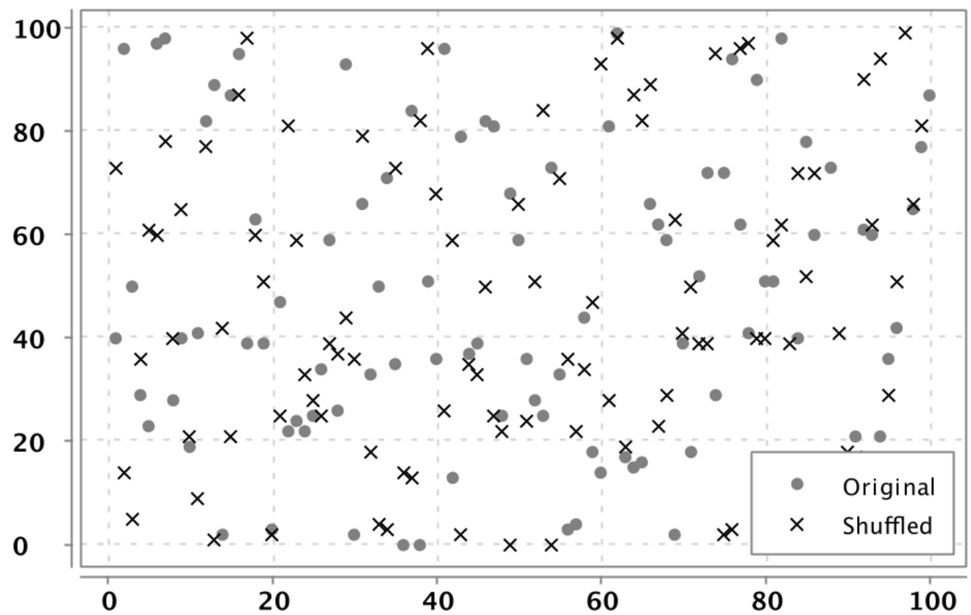

Figure 4.4 Distribution of 100 tuples in the simulation. Gray dots represent original attributes, and black crosses represent the shuffled attributes.

The distribution simulation of 100 tuples in one attribute using 100 randomly generated data pairs (x, y), where x stands for the index and y stands for the attribute value, and the result is shown as Fig.4.4. x values increase by 1 from 1 to 100, and y values scatter randomly within the range $(1, 100)$. The shuffling is only carried out on Y-axis and not on X-axis in order to use X-axis as the reference of comparison. The outcome can obviously conclude from the figure that there is no clear relationship between the two distributions, thus a good diffusion is achieved by our algorithm.

4.4 Contribution

In this chapter, SCORD is introduced to apply the shuffling in the running instance of the database system. Within SCORD, the column-oriented storage scheme and the novel data structure BAIT are adopted to sharply improve the efficiency of the shuffling algorithm.

Chapter 5

SmartSSD-Accelerated Shuffling

Chapter 4 explores one way to mitigate the performance burden of the data shuffling in database system, while there are still many methods to achieve the improvement on efficiency and security of the proposed shuffling algorithm. This chapter propose a design to further enhance the security of shuffling algorithm and improve the efficiency by employing SmartSSD computational storage device from Samsung and AMD.

Looking back at the era when mechanical hard drives were the absolute dominant storage medium, one significant bottleneck for database performance was the read and write speed of mechanical hard drives, especially for random read and write operations. This was due to the internal structure of mechanical hard drives. In comparison to mechanical hard drives, solid state drives (SSDs) are better suited for scenarios with high-speed read and write and high I/O demands. SSDs offer faster access speeds, lower latency, better random read and write capabilities, and lower power consumption [89]. However, the rapid development of SSDs has shifted the system bottleneck from storage medium to the time spent on data transfer [90]. For example, when a significant amount of data pages are required from storage for tasks such as shuffling and encryption, these data pages must be transferred from storage to memory through interfaces like PCIe, SATA, and SAS. Afterward, the CPU accesses and processes the data, with high-speed cache playing a supporting role when necessary.

The conception of “in storage computing” or “near storage computation” was proposed as early as the 1990s, where the disk was designed to execute most processing tasks to processors inside the disk [9], [10]. However, this conception began to gain attention and development around the early 2010s [11], [12]. By deploying the in storage computing devices in database servers, certain tasks like shuffling and encryption can be offloaded to the storage devices. When shuffling or encryption requests are received, the host forwards these requests to the storage. The processor within the storage retrieves the necessary data from storage to its internal DRAM, where it handles the processing requests. Subsequently, the outcomes are written back to the storage. This architecture avoids competition between data requiring shuffling or encryption and other data. It also eliminates the need for such data to traverse the lengthy path through the system bus, spanning storage, RAM, cache, and processors.

A computational storage drive (CSD) is a “in storage computing”. It reduces data transfer latency and enhances data processing efficiency by introducing processing capabilities inside the storage. There are several commercial products in the market from companies like NGD Systems, ScaleFlux, Pliops, SNIA, and Xilinx (acquired by AMD). The core of the SmartSSD CSD is the AMD FPGA programmable platform and Samsung’s SSD controller [23]. A dedicated high-speed P2P link connects the SSD controller to the FPGA for low-latency processing.

In this chapter, a revised design that combines SmartSSD CSD and shuffling algorithms is depicted for enhancing database data security with minimal additional performance overhead. The main contributions of the chapter are listed as follows:

- This chapter demonstrates how to use SmartSSD CSD to accelerate the shuffling algorithm, including software and hardware architectures, ensuring that the performance overhead of shuffling algorithms does not affect the execution of other tasks in the database system.
- This chapter proposes effectiveness and security enhancement for the shuffling algorithm.

- This chapter optimizes the shuffling algorithm further using the built-in FPGA accelerator of the SmartSSD CSD to improve efficiency.

5.1 Design and Implementation

This section introduces the design of the relational database with accelerated shuffling on CSD. To the best of our knowledge, this is the first public description of such a system that utilizing CSD on the database security.

At its essence, the revised database system comprises three integral components: 1) modules pertaining to relational databases, excluding storage considerations, 2) the BAIT and storage module, and 3) a module dedicated to accelerating data shuffling processes. Within the domain of relational databases modules, some modules have been developed, which encompasses an SQL parser, a system manager responsible for overseeing DDL (Data Definition Language) statement, a query manager responsible for handling DML (Data Manipulation Language) statements, and an indexing component. The BAIT and storage module have the primary responsibility of managing page I/O operations and shuffling processes. Consequently, this module exhibits a close association with SmartSSD CSD. It is noteworthy that while the source code of this module within the prototype may not exhibit distinctiveness from the relational database modules, it is conceptually delineated as a separate entity. Given that SmartSSD CSD incorporates embedded FPGA chips, this section also introduces an acceleration module to optimize the shuffling operation by harnessing the computational capabilities of the FPGA.

5.1.1 Matrix-Shuffling Algorithm

The multi-level shuffling described in Chapter 3 primarily focuses on account of performance and security considerations. The rationale to conduct the shuffling in the form of multiple level is grounded in the following key factors:

- **Efficiency of Computation**: When dealing with lengthy columns, it is more computationally efficient to partition them into multiple blocks since both generating a single large permutation sequence and verifying if this sequence remains deranged can be highly time-consuming processes.

- **Information Disclosure**: Another compelling reason for multi-level shuffling is to reduce the potential information leakage during the "segmented" shuffling process. By shuffling elements within the confines of their respective blocks, the degree of diffusion is restricted, which may unintentionally reveal the presence of the shuffling.

However, it is essential to acknowledge that even with the multi-level shuffling, certain patterns may persist in the distribution of the shuffled column. This observation holds particularly true for columns characterized by data that follows a monotonically increasing or decreasing pattern, such as timestamps or serial numbers. In such cases, the effectiveness of concealing the original data distribution may be limited, and certain underlying patterns may still be discernible in the shuffled outcome.

As depicted in Fig. 5.1, a straightforward shuffling experiment involving a column with parameters $l = 256$ and $w = 16$ reveals obvious patterns in multi-level shuffling. In this visualization, black dots signify monotonic data points with random increments ranging between 0 and 5, while the gray dots represent the positions of these data points following the shuffling process. The previous key sensitive analysis suggests that data points are unlikely to return to their original positions, an evaluation of the statistical distribution can still readily unveil the presence of shuffling. Consequently, potential attackers may focus their efforts on exploiting weaknesses in the shuffling process.

As a solution, an enhanced *Matrix-Shuffling* algorithm based on SCORD's shuf-

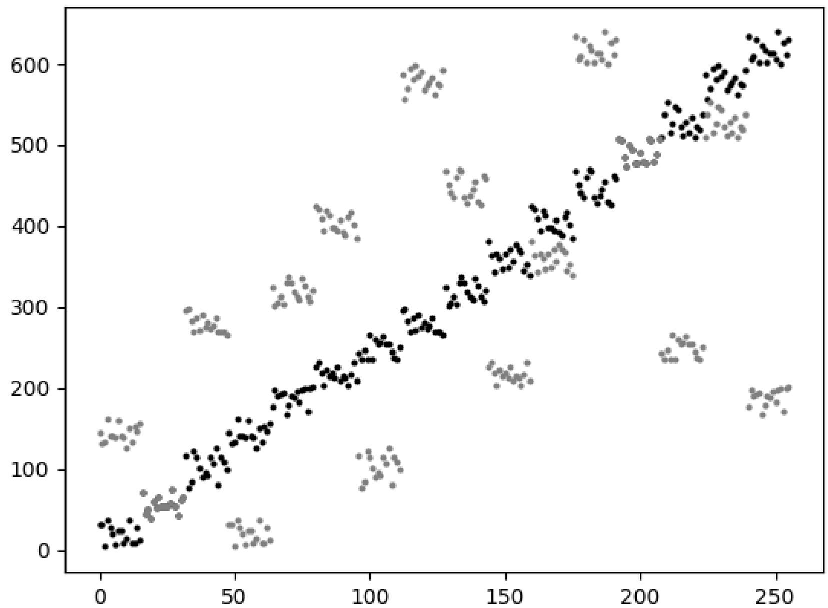

Figure 5.1 Distribution of monotonic data. Black dots represent the original data points before shuffling, while gray dots depict the shuffled data points. The horizontal axis represents the index of the data, and the vertical axis represents the corresponding values of the data points.

fling technique is proposed in this chapter. Instead of employing multiple levels of shuffling, our algorithm adopts a matrix-based approach. The tuples within a given column, denoted as $x_0, x_1, ..., x_{l-1}$, are organized into an $l/w \times w$ matrix denoted as $\mathbb{X}$

$$\mathbb{X} = (\mathbb{X}_{ij}) = \begin{pmatrix} x_0 & x_1 & \cdots & x_{w-1} \\ x_w & x_{w+1} & \cdots & x_{2w-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{(l/w-1)w} & x_{(l/w-1)w+1} & \cdots & x_{l-1} \end{pmatrix}$$

according with the unique representation of each integer in $0, 1, ..., l-1$ as

$$iw + j, \quad 0 \leq i \leq l/(w-1), 0 \leq j \leq w-1$$

so that $\mathbb{X}_{ij} = x_{iw+j}$.

The transpose $\mathbb{X}^{\mathbb{T}}$ is expressed as

$$\mathbb{Y} = (\mathbb{Y}_{i'j'}) = \begin{pmatrix} x_0 & x_w & \cdots & x_{(l/w-1)w} \\ x_1 & x_{w+1} & \cdots & x_{(l/w-1)w+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{w-1} & x_{2w-1} & \cdots & x_{l-1} \end{pmatrix} = \mathbb{X}^{\mathbb{T}}$$

Within the matrix framework, each row undergoes shuffling according to one deranged permutation sequences, followed by independent shuffling of each matrix column. Considering data structure conventions, a matrix is commonly represented as a two-dimensional vector or array. Therefore, to execute column-wise shuffling on the matrix, a matrix transpose operation becomes necessary. It is noteworthy that the column cannot be reorganized into a matrix all the time and the remainder $l\%w$ is suggested to do the shuffling in-situ.

The pseudocode of the *Matrix-Shuffling* is shown as follows:

```
num_rows = number of rows of the matrix
num_cols = number of columns of the matrix
matrix = initialize a num_rows by num_cols matrix

index = 0
for i from 0 to (num_rows - 1):
    for j  from 0 to (num_cols - 1):
        matrix[i][j] = tuple[index]
        index++

for i from 0 to (num_rows - 1):
    p = generate shuffling sequence
```

```
    shuffle the row i based on p

matrix = transpose matrix

for j from 0 to (num_cols - 1):
    p = generate shuffling sequence
    shuffle the column i based on p
```

5.1.2 Shuffling Offloading

In regular computer systems, the host processor engages in a sequence of communications with the storage devices, wherein data from the storage devices is read into the memory hierarchy, and subsequent computational operations are performed. Upon completion of these computations, the data is then written back to the storage devices from the memory.

When an SmartSSD CSD is installed in the system, the accelerated host computation may fall into one of the following two scenarios: 1) the host processor reads the data from the SmartSSD storage and transfers them into the memory dedicated to the SmartSSD CSD, and 2) the processor inside the SmartSSD CSD communicate directly with the storage inside the SmartSSD CSD via the fast and private P2P link, enabling it to access and retrieve the data directly. In the former case, data must traverse through the host memory before reaching the FPGA DRAM. Data stored in SmartSSD CSD follows a path that includes traversal through the SmartSSD storage controller, the SmartSSD PCIe switch, the host memory, the SmartSSD PCIe switch once again, and finally, the FPGA DRAM. Conversely, the latter scenario involves direct communication between the FPGA and the storage, and there is no reliance on host resources for data transfer. Therefore, the P2P communication approach can significantly reduce latency and is less resource-intensive on the host system.

Figure 5.2 shows the interaction between the SmartSSD CSD and the host system in the context of SQL querying. The host system initiates this interaction by dispatching instructions and, where necessary, pertinent data essential for executing SQL statements, such as data to be inserted or updated. The SmartSSD CSD then starts to load data from the SmartSSD storage to the FPGA DRAM via the P2P communication link, with data being loaded in page-sized units. Once all the relevant pages have been successfully loaded into the FPGA DRAM, the de-shuffling and subsequent processing procedures are initiated. Upon the completion of these operations, the pages are subjected to shuffling once again and then stored back into the SmartSSD storage. At the same time, the processing results will be put to the output buffer and then returned to the host side. This approach empowers the SmartSSD CSD to effectively offload a significant portion of the I/O-intensive tasks, thereby conserving computational resources on the host for other concurrent operations.

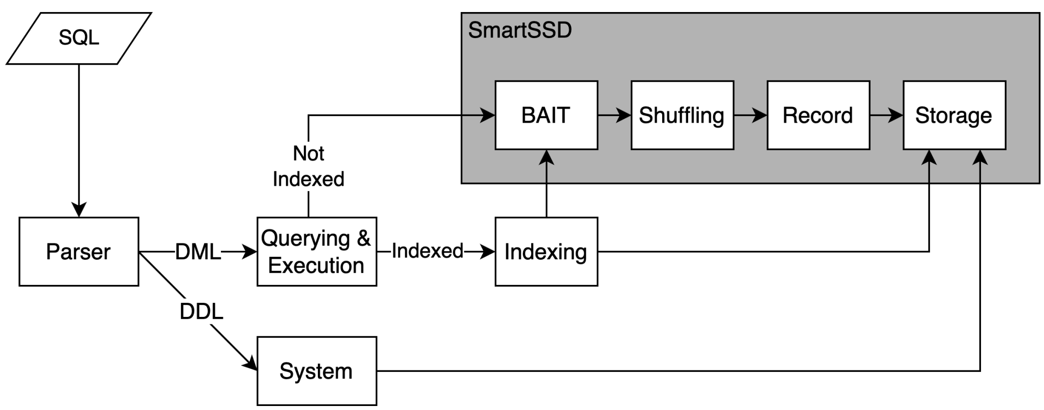

Figure 5.2 System Architecture with SmartSSD CSD

5.1.3 Acceleration Module

The SmartSSD CSD contains one Xilinx Kintex Ultrascale+ KU15P FPGA with 1.143 million SLCs (system logic cells) and approximate 300,000 LUTs (Look Up Tables). These SLCs and LUTs are key components in FPGA that play a crucial role in accelerating applications as they can be configured to implement various digital logic functions. With the help of the FPGA, an acceleration unit to accelerate the generation of deranged permutation has been designed.

As described in previously-mentioned sections, a deranged permutation plays a vital role in the shuffling. Generating a permutation is faster than generating a deranged permutation, especially as the length of the sequence increases. Since high efficient algorithms such as Fisher-Yates shuffle can help to generate the permutation, the generation of the deranged permutation can be concluded as two steps:

- Uniformly generating a permutation sequence at random.
- Verifying whether the generated permutation meets the criteria of being deranged. If it does, the algorithm outputs this permutation as the result. If not, the process is iteratively repeated, generating random permutations until a valid deranged permutation is obtained.

In general, the acceleration module covers the verifying that if the generated permutation is deranged based on linear array model. This acceleration module employes a linear array based on the FPGA to accelerate the check of deranged permutation. The linear array model is consisted of $2w$ identical processors (w is the size of one block) and these processors organized in two one-dimensional structure. Except for the first one, the $i-th$ processor, for $1 < i < w$, has local connection to processor $i-1$. All first w processors have the local connection to processor $i+w$. Every processor has a finite number of registers, and can execute basic instructions. This linear array model belongs to the SIMD class of computer architectures. During the execution of the derangement checking, processors work in synchronous manner and perform in parallel the same sequence of instructions. It is recommended that one selected processor, such as the first one, may have some additional features and can control work of the others. The structure of the model is shown in the figure 5.3.

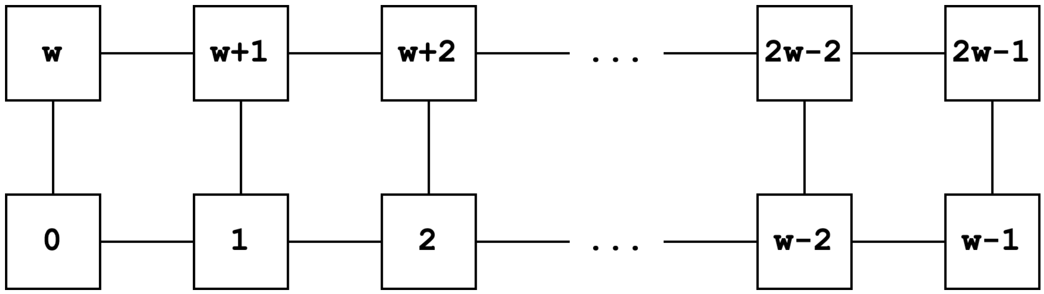

Figure 5.3 Linear array model of parallel computations

During the verification, all elements in the sequence before shuffling would be stored in the processors 0 to $w-1$, and all elements in the shuffled sequence would be stored in the processors w to $2w-1$. As one pair of processors, the processor i and the processor $i+w$ would conduct the comparison in parallel to accelerate the verification of derangement. The pseudocode is shown in below.

```
module Derangement_Checker (
  input [w-1:0] orig,
  input [w-1:0] perm,
  output reg is_derangement
);


reg flag = 1;


always @(*) begin
  integer i;
  flag = 1;


  for (i = 0; i < w; i = i + 1) begin
    if (orig[i] == perm[i]) begin
      flag = 0;
      break;
```

```
    end
  end

  is_derangement = flag;
end
endmodule
```

5.2 Experiment Evaluation and Analysis

To evaluate the efficiency of shuffling with SmartSSD CSD, a prototype of the design on a workstation with SmartSSD installed is implemented. The specifications of the SmartSSD CSD are detailed in Table 5.1. Notably, the bandwidth between DRAM and FPGA is around 20 GB/s, and the bandwidth between NAND flash storage and FPGA is around 3 GB/s.

Table 5.1 SmartSSD CSD Specifications

Form Factor	2.5" (U.2)
Storage Capacity	3.84 TB
Host Interface	PCIe Gen3x4
Sequential Read	Up to 3,300 MB/s
Sequential Write	Up to 2,000 MB/s
Random Write	Up to 800,000 IOPS
Random Read	Up to 110,000 IOPS
System Logic Cells	1.143 Million
Available LUTs	∼300 K
Accelerator-dedicated RAM	4 GB DDR4 @ 2400 Mbps

The prototype is written with C++ in AMD Vitis Unified Software Platform 2023.1. The experiments are conducted on a workstation machine with one Intel i7-10700F processor and 32GB of DDR4 memory. It's important to note that the SmartSSD CSD is designed for server environment, which typically equipped with powerful cooling system and native U.2 interface. To facilitate the experiments, one U.2 to PCIe x4 adapter and one additional 70mm × 70mm fan are used. These measures were taken to prevent the FPGA and NAND components from reaching their thermal thresholds, and they can be seen in Fig. 5.4. The SmartSSD CSD driver and the AMD Vitis development environment were both installed on an Ubuntu 20.04.6 system, with an additional installation of an older version of the Linux Kernel to accommodate the development requirement.

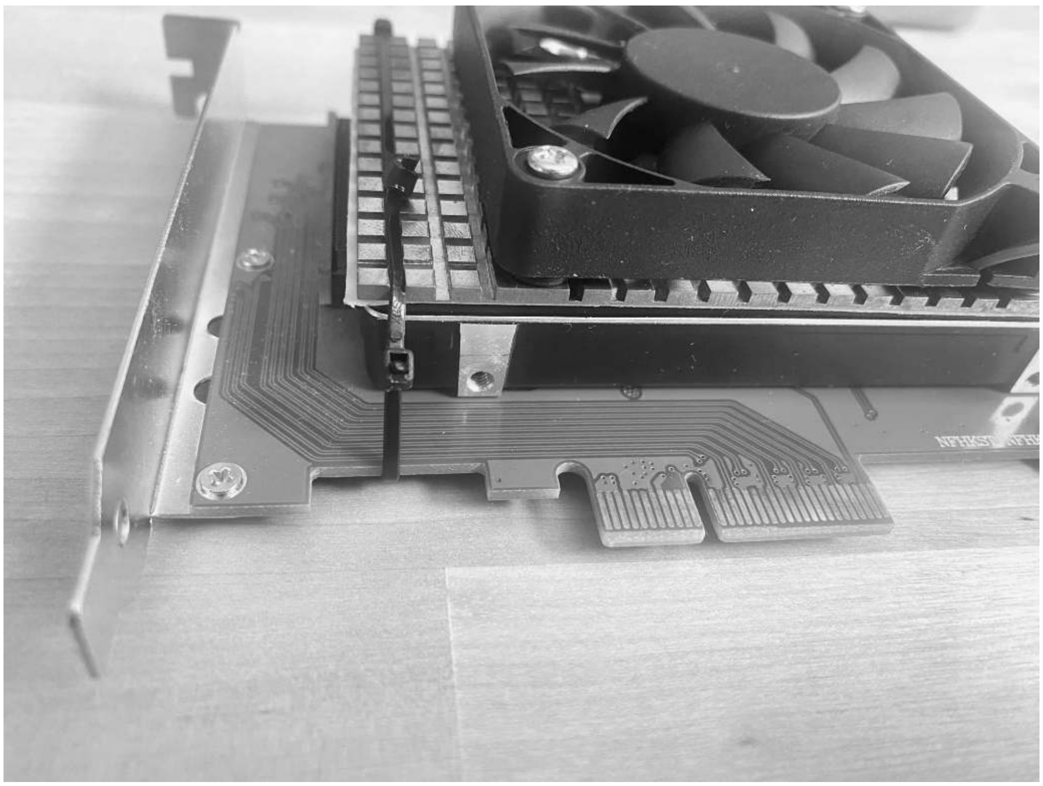

Figure 5.4 SmartSSD adapter and cooling.

In the performance evaluation, an IMDB review dataset is employed, as sourced from [91]. This dataset consists of numeric unique identifiers for reviews, accompanied by associated information such as reviewers, movies, ratings, and more. It contains over 5 million review records. To enhance its usability for our purposes, the dataset are converted from its original `json` format to `sql` format.

5.2.1 Shuffling Performance Evaluation

To simplify the performance evaluation process, the following assumptions must be clarified: 1) The shuffling process is initiated when the number of tuples l is a multiple of w (size of the block). This approach ensures that no remaining tuples are left during the conversion from a list of tuples to a matrix. 2) All columns within the relation have been indexed to maximize query performance.

To practically examine the performance of the design, the comparative analysis is conducted between two systems. In one system, SmartSSD CSD was enabled, and in the other, it was disabled. In the former, shuffling and storage operations were offloaded to the SmartSSD CSD, while in the latter, all processing was executed solely by the host CPU. The system performance evaluation uses a collection of insertion and update statements denoted as $\mathbb{Q}$. The total execution time without shuffling enabled is represented as $T^{\mathbb{Q}}_{\text{OFF}}$. Meanwhile, the execution time with shuffling enabled on a CPU-only platform is denoted as $T^{\mathbb{Q}}_{\text{CPU}}$, and the execution time with shuffling enabled with the involvement of SmartSSD CSD is denoted as $T^{\mathbb{Q}}_{\text{CSD}}$. The evaluation tests encompasses both INSERT and UPDATE operations involving 50,000 and 100,000 statements. The average results of 10 rounds are shown in Table 5.2.

Table 5.2 Running Times Under Three Cases

Operation	# of Statements	$T^{\mathbb{Q}}_{\text{OFF}}$	$T^{\mathbb{Q}}_{\text{CPU}}$	$T^{\mathbb{Q}}_{\text{CSD}}$
INSERT	50,000	5.27 s	5.88 s	5.42 s
INSERT	100,000	9.31 s	10.42 s	9.69 s
UPDATE	50,000	242.18 ms	306.85 ms	254.19 ms
UPDATE	100,000	387.11 ms	498.23 ms	399.65 ms

Based on the results, the following observations can be made:

- **Minor Impact of Shuffling Process**: The time consumed by the shuffling process is found to be relatively small when compared to the overall execution time. Moreover, the integration of SmartSSD CSD demonstrates its capability to further minimize the time allocated to shuffling. This suggests that the shuffling operation, when accelerated by SmartSSD CSD, exerts minimal influence on the system's overall performance. Consequently, a greater portion of computational resources can be directed towards other critical operations, such as data processing and index building.

- **Substantial Performance Enhancement**: The execution of statements involving SmartSSD CSD participation exhibits a notable improvement in relative performance from 5 to 10 × faster, when compared to scenarios where only the CPU is engaged in shuffling computations. This substantial enhancement underscores the significant reduction in overhead time associated with shuffling when SmartSSD CSD is employed.

5.2.2 Distribution and Information Entropy Analysis

For the purpose of eliminating the distribution traces of shuffled tuples, we ameliorate the shuffling algorithm. We conduct the same experiment as Section 5.1.1, and the distribution of monotonic data is reflected in Fig. 5.5. Our shuffling based on matrix could clearly state a better randomness on the data points distribution after shuffling.

Information entropy is one of the most important feature of randomness [92]. High entropy is crucial for security-critical applications, and high entropy means that the outcomes are more unpredictable and random. Let m be the procedure of shuffling; the formula for calculating the information entropy can be expressed as follows:

$$H(m) = -\sum_{i=1}^{n} p(x_i) \cdot \log_2(p(x_i))$$

where n represents the number of possible positions for the tuple to move during the shuffling, and $p(x_i)$ represents the probability of selecting each position x_i. Table 5.3 records the different information entropy with various column size under multi-level

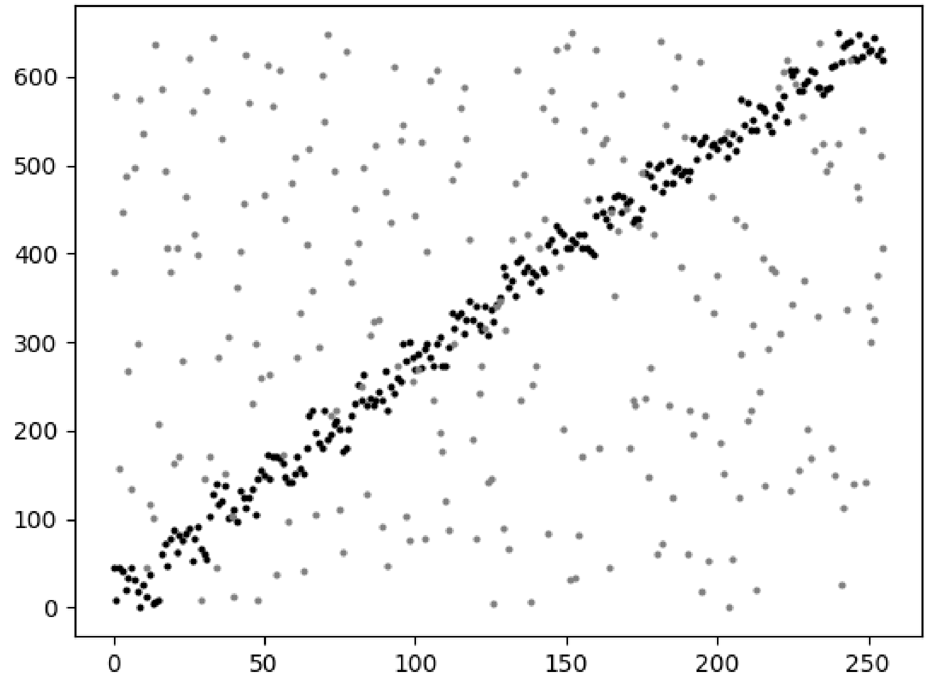

Figure 5.5 Distribution of monotonic data under matrix shuffling. Blacks dots are data points before shuffling, while gray dots are shuffled data points. Horizontal coordinate represents the index of the data, and vertical coordinate holds the value.

shuffling and matrix shuffling. We can clearly see that the matrix shuffling could provide a much larger information entropy for a better randomness.

Table 5.3 Information Entropy Under Different Shuffling Algorithm

Column Size	64	256	1024	1024
Multi-level Shuffling	24	64	160	4096
Matrix Shuffling	295.995	1683.996	8769.006	43250.047

5.3 Contribution

In this chapter, a solution for enhancing database security through the use of SmartSSD-accelerated is described for cryptographic shuffling. The approach accelerates the shuffling algorithm using SmartSSD CSD, including software and hardware architectures, and optimizes the algorithm further using the built-in FPGA. Several performance evaluations have been conducted to show the performance improvements the SmartSSD CSD brings to the shuffling algorithm. The proposed solution provides a promising direction for enhancing database security and protecting sensitive information.

Chapter 6

Conclusion

In summary, this research has made significant contributions to the field of database security by introducing an innovative shuffling algorithm. The paper discusses the significance of database security in protecting an organization's valuable data assets in the face of increasing cyber threats The shuffling algorithm in relational database is a novel way to enhance the database security and preserve the data privacy without modifying the data themselves. Efforts have been made to improve the efficiency of the shuffling algorithm through novel storage data structures, column-oriented storage schemes, and computational storage device integration. With the assistance of column-oriented database and SmartSSD computational storage devices, the proposed shuffling algorithm could promote the security with just a minor overhead on the performance.

The contributions in this dissertation are listed below:

1. The shuffling could not only protect the offline database data storage but also apply the shuffling to the running database.

2. Two shuffling schemes are proposed for different scenarios, the schemes include the multi-level shuffling and matrix-shuffling.

3. Attribute association aware feature is proposed to mitigate the risk of statistical attack.

4. A new data storage data structure is proposed for the shuffling without actually moving the data inside the storage.

5. Hardware accelerators have been utilized to promote the shuffling algorithm, especially the deranged permutation generation on which the shuffling relies.

In the realm of relational database design, the shuffling algorithm exhibits notable advantages and contributions. However, it is essential to acknowledge that certain facets of this design could benefit from further improvement. Improving these aspects represents a critical avenue for future research endeavors aimed at enhancing the design's security, performance, and usability.

At present, the remarkable performance of the algorithm is contingent upon column-oriented storage. Nevertheless, it is noteworthy that there exist scenarios where row-oriented storage outperforms column-oriented storage in terms of both performance and compatibility. As elucidated in Chapter 4, data manipulation is more facile in row-oriented storage, while specific queries faster execution in column-oriented storage. The potential establishment of a converter holds promise in influencing the overall performance of the design.

Notably, certain modules in the current implementation require refinement. Areas such as the indexing of BAIT and the acceleration of modules beyond shuffling within the relational database are identified as areas requiring attention. Undertaking further research endeavors is imperative to address these deficiencies and fortify the overall design.

Bibliography

[1] "Cost of a Data Breach Report 2023," IBM Security, Tech. Rep., Jun. 2023.

[2] A. Hope, *4 million impacted in colorado department of health care ibm moveit data breach*, Aug. 2023. [Online]. Available: `https://www.cpomagazine.com/cyber-security/4-million-impacted-in-colorado-department-of-health-care-ibm-moveit-data-breach/`.

[3] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: Protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ACM, 2011, pp. 85–100.

[4] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Annual cryptology conference*, Springer, 2013, pp. 353–373.

[5] P. Grubbs, T. Ristenpart, and V. Shmatikov, "Why your encrypted database is not secure," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017, pp. 162–168.

[6] S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of computer and system sciences*, vol. 28, no. 2, pp. 270–299, 1984.

[7] K. G. Kogos, K. S. Filippova, and A. V. Epishkina, "Fully homomorphic encryption schemes: The state of the art," in *2017 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, IEEE, 2017, pp. 463–466.

[8] X. Song and Y. Wang, "Homomorphic cloud computing scheme based on hybrid homomorphic encryption," in *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, IEEE, 2017, pp. 2450–2453.

[9] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," *ACM SIGOPS Operating Systems Review*, vol. 32, no. 5, pp. 81–91, 1998.

[10] E. Riedel, G. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia applications," in *Proceedings of 24th Conference on Very Large Databases*, Citeseer, 1998, pp. 62–73.

[11] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: Opportunities and challenges," en, in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York New York USA: ACM, Jun. 2013, pp. 1221–1230, ISBN: 978-1-4503-2037-5. DOI: `10.1145/2463676.2465295`. [Online]. Available: `https://dl.acm.org/doi/10.1145/2463676.2465295`.

[12] Y. Kang, Y.-s. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart ssd," en, in *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, IEEE, May 2013, pp. 1–12, ISBN: 978-1-4799-0218-7. DOI: `10.1109/MSST.2013.6558444`. [Online]. Available: `http://ieeexplore.ieee.org/document/6558444/`.

[13] G. F. Estabrook and R. C. Brill, "The theory of the taxir accessioner," *Mathematical Biosciences*, vol. 5, no. 3-4, pp. 327–340, 1969.

[14] G. Graefe, "B-tree indexes for high update rates," *ACM Sigmod Record*, vol. 35, no. 1, pp. 39–44, 2006.

[15] A. Braginsky and E. Petrank, "A lock-free b+ tree," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, 2012, pp. 58–67.

[16] S. M. Rumble, A. Kejriwal, and J. Ousterhout, "Log-structured memory for {dram-based} storage," in *12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014, pp. 1–16.

[17] M. Stonebraker, D. J. Abadi, A. Batkin, *et al.*, "C-store: A column-oriented dbms," in *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, 2018, pp. 491–518.

[18] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[19] J. Lánskỳ and M. Zemlicka, "Compression of a dictionary," in *Proceedings of the Dateso 2006 Annual International Workshop on DAtabases, TExts, Specifications and Objects. CEUR-WS*, vol. 176, 2006, pp. 11–20.

[20] S. Golomb, "Run-length encodings (corresp.)," *IEEE transactions on information theory*, vol. 12, no. 3, pp. 399–401, 1966.

[21] S.-i. Minato, "Zero-suppressed bdds for set manipulation in combinatorial problems," in *Proceedings of the 30th International Design Automation Conference*, 1993, pp. 272–277.

[22] K. Osborne, R. Johnson, T. Pöder, K. Osborne, R. Johnson, and T. Pöder, "Hybrid columnar compression," *Expert Oracle Exadata*, pp. 65–104, 2011.

[23] *Samsung SmartSSD*, en. [Online]. Available: `https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html` (visited on 08/30/2023).

[24] *Smartssd computational storage drive installation and user guide*, en, Oct. 2021. [Online]. Available: `https://www.xilinx.com/content/dam/xilinx/support/documents/boards_and_kits/accelerator-cards/1_3/ug1382-smartssd-csd.pdf` (visited on 08/30/2023).

[25] Y. Wang, J. Wang, and X. Chen, "Secure searchable encryption: A survey," *Journal of communications and information networks*, vol. 1, pp. 52–65, 2016.

[26] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," in *Proceedings of the VLDB Endowment*, VLDB Endowment, vol. 6, 2013, pp. 289–300.

[27] R. Poddar, T. Boelter, and R. A. Popa, "Arx: A strongly encrypted database system.," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 591, 2016.

[28] A. Papadimitriou, R. Bhagwan, N. Chandran, *et al.*, "Big data analytics over encrypted datasets with seabed," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 587–602.

[29] P. Paillier, *Paillier encryption and signature schemes.* 2005.

[30] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, IEEE, 2000, pp. 44–55.

[31] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *International conference on the theory and applications of cryptographic techniques*, Springer, 2004, pp. 506–522.

[32] D. Boneh and B. Waters, "Conjunctive, subset, and range queries on encrypted data," in *Theory of Cryptography: 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007. Proceedings 4*, Springer, 2007, pp. 535–554.

[33] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner, "Rich queries on encrypted data: Beyond exact matches," in *European Symposium on Research in Computer Security*, Springer, 2015, pp. 123–145.

[34] K. Xue, S. Li, J. Hong, Y. Xue, N. Yu, and P. Hong, "Two-cloud secure database for numeric-related sql range queries with privacy preserving," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 7, pp. 1596–1608, 2017.

[35] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Pump up the volume: Practical database reconstruction from volume leakage on range queries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 315–331.

[36] S. Wu, Q. Li, G. Li, D. Yuan, X. Yuan, and C. Wang, "Servedb: Secure, verifiable, and efficient range queries on outsourced database," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, IEEE, 2019, pp. 626–637.

[37] L. Tang, T. Li, Y. Jiang, and Z. Chen, "Dynamic query forms for database queries," *IEEE transactions on knowledge and data engineering*, vol. 26, no. 9, pp. 2166–2178, 2013.

[38] D. Cash, J. Jaeger, S. Jarecki, *et al.*, "Dynamic searchable encryption in very-large databases: Data structures and implementation," *Cryptology ePrint Archive*, 2014.

[39] S. Bajaj and R. Sion, "Trusteddb: A trusted hardware based database with privacy and data confidentiality," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011, pp. 205–216.

[40] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan, "A secure coprocessor for database applications," in *2013 23rd Interna-*

tional Conference on Field programmable Logic and Applications, IEEE, 2013, pp. 1–8.

[41] B. Fuhry, H. J. Jain, and F. Kerschbaum, "Encdbdb: Searchable encrypted, fast, compressed, in-memory database using enclaves," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, 2021, pp. 438–450.

[42] C. Priebe, K. Vaswani, and M. Costa, "Enclavedb: A secure database using sgx," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 264–278.

[43] S. Eskandarian and M. Zaharia, "Oblidb: Oblivious query processing for secure databases," *arXiv preprint arXiv:1710.00458*, 2017.

[44] D. Vinayagamurthy, A. Gribov, and S. Gorbunov, "Stealthdb: A scalable encrypted database with full sql query support.," *Proc. Priv. Enhancing Technol.*, vol. 2019, no. 3, pp. 370–388, 2019.

[45] R. Álvarez-Sánchez, A. Andrade-Bazurto, I. Santos-González, and A. Zamora-Gómez, "Aes-ctr as a password-hashing function," in *International Joint Conference SOCO'17-CISIS'17-ICEUTE'17 León, Spain, September 6–8, 2017, Proceeding 12*, Springer, 2018, pp. 610–617.

[46] M. Coles and R. Landrum, "Transparent data encryption," in *Expert SQL Server 2008 Encryption*, Springer, 2009, pp. 127–150.

[47] S. Gaetjen, D. Knox, and W. Maroulis, *Oracle Database 12c Security*. McGraw-Hill Education Group, 2015.

[48] T. Dalenius and S. P. Reiss, "Data-swapping: A technique for disclosure control," *Journal of statistical planning and inference*, vol. 6, no. 1, pp. 73–85, 1982.

[49] K. Muralidhar and R. Sarathy, "Data shuffling—a new masking approach for numerical data," *Management Science*, vol. 52, no. 5, pp. 658–670, 2006.

[50] J. Chung, K. Lee, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Ubershuffle: Communication-efficient data shuffling for sgd via coding theory," *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[51] C. Meehan, A. R. Chowdhury, K. Chaudhuri, and S. Jha, "Privacy implications of shuffling," in *International Conference on Learning Representations*, 2021.

[52] A. Bittau, Ú. Erlingsson, P. Maniatis, *et al.*, "Prochlo: Strong privacy for analytics in the crowd," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 441–459.

[53] T. Gao and Z. Chen, "Image encryption based on a new total shuffling algorithm," *Chaos, solitons & fractals*, vol. 38, no. 1, pp. 213–220, 2008.

[54] N. A. Ali, A. M. S. Rahma, and S. H. Shaker, "3d content encryption using multi-level chaotic maps," *Iraqi Journal of Science*, pp. 2521–2532, 2023.

[55] A. A. Tamimi and A. M. Abdalla, "An audio shuffle-encryption algorithm," in *The world congress on engineering and computer science*, 2014.

[56] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten, "Monetdb: Two decades of research in column-oriented database," *IEEE Data Engineering Bulletin*, 2012.

[57] B. Imasheva, N. Azamat, A. Sidelkovskiy, and A. Sidelkovskaya, "The practice of moving to big data on the case of the nosql database, clickhouse," in *Optimization of Complex Systems: Theory, Models, Algorithms and Applications*, Springer, 2020, pp. 820–828.

[58] D. Vohra, "Apache parquet," in *Practical Hadoop Ecosystem*, Springer, 2016, pp. 325–335.

[59] S. Melnik, A. Gubarev, J. J. Long, *et al.*, "Dremel: Interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, 2010.

[60] J. Wang, D. Park, Y.-S. Kee, Y. Papakonstantinou, and S. Swanson, "Ssd in-storage computing for list intersection," in *Proceedings of the 12th International Workshop on Data Management on New Hardware*, 2016, pp. 1–7.

[61] M. Torabzadehkashi, S. Rezaei, A. HeydariGorji, H. Bobarshad, V. Alves, and N. Bagherzadeh, "Computational storage: An efficient and scalable platform for big data and hpc applications," *Journal of Big Data*, vol. 6, pp. 1–29, 2019.

[62] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, "Smartssd: Fpga accelerated near-storage data analytics on ssd," *IEEE Computer architecture letters*, vol. 19, no. 2, pp. 110–113, 2020.

[63] S. Salamat, A. Haj Aboutalebi, B. Khaleghi, J. H. Lee, Y. S. Ki, and T. Rosing, "Nascent: Near-storage acceleration of database sort on smartssd," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 262–272.

[64] M. Soltaniyeh, V. Lagrange Moutinho Dos Reis, M. Bryson, X. Yao, R. P. Martin, and S. Nagarakatte, "Near-storage processing for solid state drive based recommendation inference with smartssds®," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, 2022, pp. 177–186.

[65] E. B. Tavakoli, A. Beygi, and X. Yao, "Rpknn: An opencl-based fpga implementation of the dimensionality-reduced knn algorithm using random projection," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 4, pp. 549–552, 2022.

[66] N. Hedam, M. Tychsen Clausen, P. Bonnet, S. Lee, and K. Friis Larsen, "Delilah: Ebpf-offload on computational storage," in *Proceedings of the 19th International Workshop on Data Management on New Hardware*, 2023, pp. 70–76.

[67] J.-H. Kim, Y.-R. Park, J. Do, S.-Y. Ji, and J.-Y. Kim, "Accelerating large-scale graph-based nearest neighbor search on a computational storage platform," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 278–290, 2022.

[68] *Abg/dbsake*, `https://github.com/abg/dbsake`, Accessed: 2019-06-26.

[69] C. W. Probst, R. R. Hansen, and F. Nielson, "Where can an insider attack?" In *Formal Aspects in Security and Trust: Fourth International Workshop, FAST 2006, Hamilton, Ontario, Canada, August 26-27, 2006, Revised Selected Papers 4*, Springer, 2007, pp. 127–142.

[70] K. Pearson, "X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 50, no. 302, pp. 157–175, 1900.

[71] J. Lee Rodgers and W. A. Nicewander, "Thirteen ways to look at the correlation coefficient," *The American Statistician*, vol. 42, no. 1, pp. 59–66, 1988.

[72] R. A. Fisher, "Statistical methods for research workers," in *Breakthroughs in statistics*, Springer, 1992, pp. 66–70.

[73] M. M. Vopson, "Estimation of the information contained in the visible matter of the universe," *AIP Advances*, vol. 11, no. 10, 2021.

[74] C. E. Shannon, "Communication theory of secrecy systems," *Bell system technical journal*, vol. 28, no. 4, pp. 656–715, 1949.

[75] *Iso/iec 7812-1:2017*, `http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/07/04/70484.html`, Accessed: 2019-06-26.

[76] R. Elmasri and S. Navathe, *Fundamentals of database systems*. Addison-Wesley Publishing Company, 2010.

[77] L. Bouganim and P. Pucheral, "Chip-secured data access: Confidential data on untrusted servers," in *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, Elsevier, 2002, pp. 131–142.

[78] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004, pp. 563–574.

[79] J. F. Korsh and P. S. LaFollette, "Constant time generation of derangements," *Information processing letters*, vol. 90, no. 4, pp. 181–186, 2004.

[80] A. Crotty, V. Leis, and A. Pavlo, "Are you sure you want to use mmap in your database management system," in *CIDR 2022, Conference on Innovative Data Systems Research. https://db. cs. cmu. edu/papers/2022/p13-crotty. pdf*, 2022.

[81] W. Effelsberg and T. Haerder, "Principles of database buffer management," *ACM Transactions on Database Systems (TODS)*, vol. 9, no. 4, pp. 560–595, 1984.

[82] R. Panneerselvam, *Database Management Systems*. PHI Learning Pvt. Ltd., 2011.

[83] A. K. Dwivedi, C. Lamba, and S. Shukla, "Performance analysis of column oriented database vs row oriented database," *International Journal of Computer Applications*, vol. 50, no. 14, 2012.

[84] *Cyan4973/xxHash*, Dec. 2020. [Online]. Available: `https://github.com/Cyan4973/xxHash` (visited on 12/03/2020).

[85] M. C. Ah Kioon, Z. S. Wang, and S. Deb Das, "Security analysis of md5 algorithm in password storage," in *Applied Mechanics and Materials*, Trans Tech Publ, vol. 347, 2013, pp. 2706–2711.

[86] P. R. Jelenković and A. Radovanović, "Least-recently-used caching with dependent requests," *Theoretical computer science*, vol. 326, no. 1-3, pp. 293–327, 2004.

[87] *Imdb data files available for download*, Dec. 2020. [Online]. Available: `https://datasets.imdbws.com/`.

[88] E. Prouff, "Dpa attacks and s-boxes," in *International Workshop on Fast Software Encryption*, Springer, 2005, pp. 424–441.

[89] E. Tomes and N. Altiparmak, "A comparative study of hdd and ssd raids' impact on server energy consumption," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2017, pp. 625–626.

[90] G. Koo, K. K. Matam, T. I, *et al.*, "Summarizer: Trading communication with computing near storage," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 219–231.

[91] *IMDb Review Dataset - ebD*, en. [Online]. Available: `https://www.kaggle.com/datasets/ebiswas/imdb-review-dataset` (visited on 09/01/2023).

[92] S.-Y. Li and B. H. Miguel Angel, "A novel image protection cryptosystem with only permutation stage: Multi-shuffling process," *Soft Computing*, pp. 1–18, 2023.