5-2020

Secure and Efficient Models for Retrieving Data from Encrypted Databases in Cloud

Sultan Ahmed A Almakdi
University of Arkansas, Fayetteville

Secure and Efficient Models for Retrieving Data from Encrypted Databases in Cloud

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science

by

Sultan Ahmed A Almakdi
King Khalid University
Bachelor of Science in Computer Science, 2010
University of Colorado at Denver
Master of Science in Computer Science, 2014

May 2020
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council.

Brajendra Panda, Ph.D.
Dissertation Director

Susan Gauch, Ph.D.
Committee Member

Mark Arnold, Ph.D.
Committee Member

Miaoqing Huang, Ph.D.
Committee Member

Abstract

Recently, database users have begun to use cloud database services to outsource their databases. The reason for this is the high computation speed and the huge storage capacity that cloud owners provide at low prices. However, despite the attractiveness of the cloud computing environment to database users, privacy issues remain a cause for concern for database owners since data access is out of their control. Encryption is the only way of assuaging users' fears surrounding data privacy, but executing Structured Query Language (SQL) queries over encrypted data is a challenging task, especially if the data are encrypted by a randomized encryption algorithm. Many researchers have addressed the privacy issues by encrypting the data using deterministic, onion layer, or homomorphic encryption. Nevertheless, even with these systems, the encrypted data can still be subjected to attack. In this research, we first propose an indexing scheme to encode the original table's tuples into bit vectors (BVs) prior to the encryption. The resulting index is then used to narrow the range of retrieved encrypted records from the cloud to a small set of records that are candidates for the user's query. Based on the indexing scheme, we then design three different models to execute SQL queries over the encrypted data. The data are encrypted by a single randomized encryption algorithm, namely the Advanced Encryption Standard AES-CBC. In each proposed scheme, we use a different (secure) method for storing and maintaining the index values (BVs) (i.e., either at user's side or at the cloud server), and we extend each system to support most of relational algebra operators, such as *select*, *join*, etc. Implementation and evaluation of the proposed systems reveals that they are practical and efficient at reducing both the computation and space overhead when compared with state-of-the-art systems like CryptDB.

Acknowledgments

First of all, I would like to thank Prof. Brajendra Panda for his endless support and guidance while pursuing my Ph.D. Without his guidance and advises, this dissertation wouldn't have been completed.

Second, special thanks go out to my family for their love and motivations during the journey of pursuing the PhD. Without their unlimited support and motivations, I would not be able to complete the research.

Third, I want to acknowledge Najran University and the ministry of higher education in Saudi Arabia for offering me the opportunity to study in the United States. Without their financial support, I would not be able to make this achievement.

Last but not least, I would like to thank Dr. Mark Arnold, Dr. Susan Gauch, and Dr. Miaoqing Huang for being members of my committee, and for their valuable comments and suggestions during this dissertation. Also, I thank staff members at the computer science department and the University of Arkansas in general.

Table of Contents

List of Figures

List of Tables

1 Introduction

In the contemporary electronic era, both individuals and organizations need scalable data storage and high-performance computing units to process and store their data. Historically, only large organizations/ companies have been able to own such units, as they were not affordable for most individuals and small companies. With the rise of cloud computing, however, this problem has been solved, as users can now rent storage and computational units as needed at an affordable price. Most cloud providers provide databases as a service, which allow individual users and companies to outsource their data and access them at any time, from any location. According to the report in [1], the compound annual growth rate (CAGR) of cloud database market is anticipated to be 46.78% in 2023 [Figure 1.1]. However, given that privacy breaches are one of the most common threats in the cloud computing environment, many people have expressed concerns about privacy when outsourcing sensitive data. For instance, untrustworthy cloud service providers might steal personal customer information—such as email addresses, mailing addresses, and phone numbers—and sell that information to third parties, who can then use it to send irritating advertisements to users via email, mail, and telephone.

More importantly, attackers who target a cloud provider can gain access to customers' sensitive personal information, such as social security numbers (SSNs). This has serious consequences, as criminals can use these data to impersonate customers in situations such as financial transactions (e.g., telephone banking). Thus, sensitive data are restricted from being processed or sold to a third party. Therefore, significant evolutions in the cloud computing environment could make such services unattractive to consumers if changes occur without also providing appropriate solutions for privacy breach issue. Such an issue must be tackled if cloud

providers are to gain the trust of users and organizations so that they will outsource sensitive data without worrying about data leakages.

Data encryption effectively solves the problem of privacy breaches by ensuring that cloud providers cannot learn from the data they store. The easiest way is to encrypt the entire table and outsource it. Then to process a query, the entire table must be retrieved and decrypted. However, the application of this technique conflicts with purpose of the databases and the critical functionalities of cloud environments (e.g., searching). Other researchers have used a proxy (i.e., a trusted third-party server), rather than the user, as an additional component to carry out the encryption and decryption processes. To make this approach practical, each datum must be encrypted with more than one encryption algorithm to support various query types [2]. However, in the case of very large data sets, this approach comes with the penalty of a significant computational burden, as each datum might be decrypted more than once. Various researchers

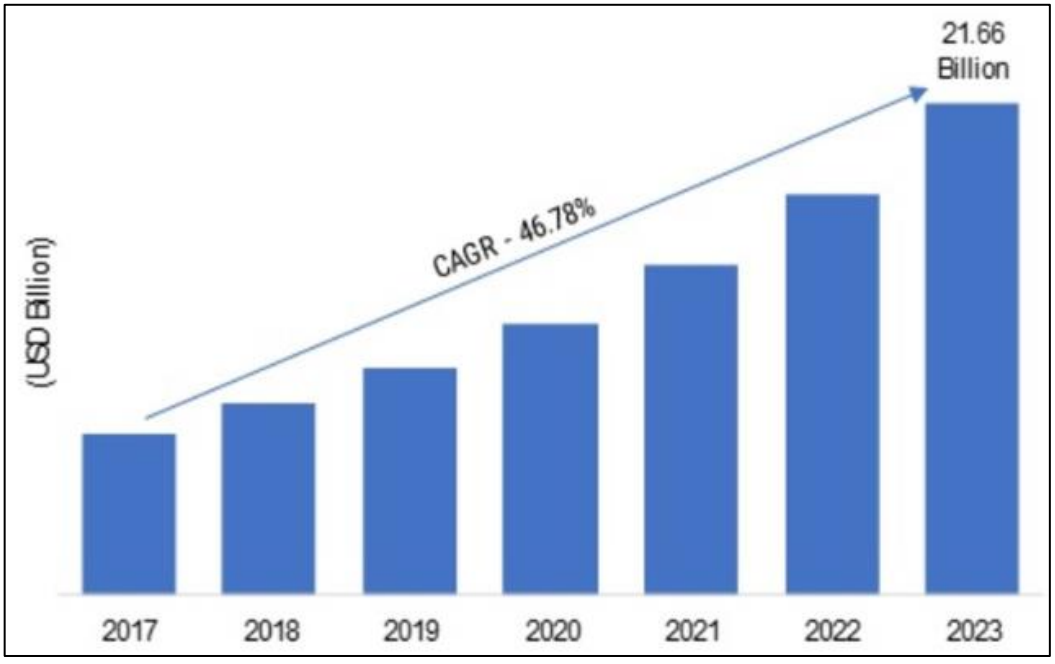


Figure 1.1 Anticipated market growth in 2023
Source: https://www.marketresearchfuture.com/reports/cloud-database-market-6847

have proposed numerous systems using different encryption techniques to protect data confidentiality. In the following subsection, we explore the common encryption algorithms that have been adopted in state-of-the-art research on cloud database security.

1.1 Encryption Algorithms

1.1.1 Order-Preserving Encryption

Order-preserving encryption (OPE) is a functional encryption technique to encrypt data such that range queries (e.g., maximum, minimum, and inequality operators) can be implemented on encrypted data without encrypting the operands or decrypting the data [3], [4]. This type of encryption uses a function to compare the order of the ciphertexts to allow comparison operations of the encrypted numeric data. This type of algorithm preserves the original data order. For example, if c_1 is the ciphertext of m_1, and c_2 is the ciphertext of m_2, then the comparison of c_1 and c_2 is as follows:

$$(c_1 < c_2 \text{ if } m_1 < m_2); (c_1 > c_2 \text{ if } m_1 > m_2); \text{ or } (c_1 = c_2 \text{ if } m_1 = m_2)$$

The security of this type of encryption is downgraded if the adversary infers the ciphertext of a certain plaintext. Also, this type of encryption is vulnerable to inference attacks as in [5], [6].

1.1.2 Deterministic Encryption

In deterministic encryption (DE) schemes, given the encryption key k and the messages m, the ciphertext of m is always the same when encrypted with k, even in multiple executions of DE. While DE can be used in keyword searches, it does not preserve the order if used with numeric values. AES-SIV is an example of a deterministic algorithm. DE is widely used in securing cloud databases. However, privacy can be compromised in this scheme if the attacker is able to identify the ciphertext of a certain plaintext word (e.g., if he establishes the ciphertext of “Alice,” then he is able to determine which tuples contain “Alice”).

1.1.3 Homomorphic Encryption

Homomorphic encryption (HOM) is a method of encryption that allows for the performance of certain arithmetic operations (i.e., addition and multiplication) on ciphertexts without the need to decrypt them. There are two types of HOM: partially HOM and fully HOM. Partially HOM supports either addition or multiplication, while fully HOM supports both operations. HOM can secure numeric data in cloud environments; however, encrypting with HOM produces long ciphertexts, since this type of encryption is based on an asymmetric encryption system. Also, computation performance on HOM ciphertexts is downgraded as the volume of ciphertexts increases [7].

1.1.4 Randomized Encryption

Randomized encryption intends to produce different ciphertexts for each plaintext, i.e., no more than one ciphertext has the same plaintext. Essentially, in this scheme, in addition to the secret key, an initialization vector (IV) must be XORed with the first block of the plaintext, and a new ciphertext must therefore be obtained each time the algorithm is executed [8]. While this encryption technique provides the highest security level for outsourced databases, it has a major drawback in cloud databases—namely, the inability to execute SQL queries over ciphertexts.

1.1.5 Onion Encryption

The term “onion layers encryption” was first developed by the authors in [2]. In onion encryption, cloud servers are able to execute different SQL statements while data remain secret. In the onion encryption, each layer is a ciphertext of a specific encryption methodology (e.g., DET, OPE, HOM, or RAN). The inner layer is the ciphertext of the algorithm with the lowest security level, while the outer layer is the ciphertext with the highest security level (i.e., randomized). The main flaw of this approach is the intensive computation that results from decrypting all of the encryption

layers, which leads to slower query processing. In addition, the space required for the encrypted databases is about 3.75 times that required for the unencrypted databases [2].

1.2 Summary of Contributions

In this dissertation, we first designed a novel indexing scheme for randomized encrypted databases, which is based on defining a partitioning tree (PT) for all domain partitions for sensitive columns in a table (i.e., dividing each column into sub-columns where each sub-column represents a set of values). The PT is then used to encode records into bit vectors (BVs), wherein each bit position is mapped to a specific partition and is only set to one if the value belongs to the set of values represented by that partition. The BVs are used to retrieve only part of the outsourced encrypted records. Second, we suggested three secure prototypes based on our proposed indexing technique. In each model, we use a different method of storing and handling the BVs (either on a private cloud server or on a third-party cloud server). Third, for each system, we proposed different algorithms to process most of the relational algebra operators on encrypted data without revealing the confidentiality of data. Fourth, we conducted several experiments to evaluate different aspects of the proposed systems—including, but not limited to, time overhead and space overhead—-against three well-known approaches: CryptDB [2], columns-based fragmentation [9], and one block-based encryption [10]. Our experiments showed that the proposed systems outperformed most of the competing approaches in both time and space overhead.

The rest of this document is organized as follow: in section 2, we explain different security enforcement schemes for the cloud databases, then we survey the state-of–the-art approaches to secure outsourced databases. Section 3 details the proposed systems in this dissertation, followed by the implementation and evaluation details in section 4. Finally, we provided a conclusion of this dissertation and discussed some of the future work in section 5.

2 Related Work

The powerful features of the cloud computing environment—such as enormous capacity and high-performance computing units—have attracted database owners in both small and large companies. These features have made it necessary to obtain and maintain the incredibly expensive storage and computation units to process vast databases at affordable prices. In spite of the advantages of cloud computing, data security is the main drawback of using cloud services to outsource databases. The work of securing databases began when networking and internet technology advanced and were adopted by major companies. This, in turn, fueled the need for databases as a service for individuals and entities requiring high storage capacity and computation. Data privacy, integrity, and confidentiality are in danger when databases are outsourced, since the database owner loses control over who can access and read their data.

To address these issues, researchers began to develop various approaches to protect user data from unauthorized access and data breaches, using data access control, data encryption, or both [11], [12]. Data access control functions by regulating what object O a subject S can access and what operations Op that S can perform on O. Encryption, in contrast, works by encoding plaintext data into ciphertext (i.e., an unreadable format). Each method has its advantages and disadvantages. For example, access control is intended to increase computing performance, while encryption decreases it. In addition, encryption protects against data breaches from both internal and external attacks, whereas data privacy can be compromised in such cases when the access control mechanism is enforced, since adversaries might bypass predefined access roles to access data. Regardless of the heavy computational work required by encryption schemes, encryption is still the preferred security mechanism for most database users.

2.1 Security Issues in Cloud Computing

Despite the powerful features of cloud computing, there are many issues and vulnerabilities that can be exploited by malicious actors against outsourced data. One type of security issues—privileges abuse—involves the use of legitimate privileges for malicious purposes (e.g., a user in company A with the privilege to view company A's sales records who uses their privileges to fetch sales tuples and pass them to competitor company B). Another vulnerability in the cloud environment occurs when a user is assigned privileges that exceed what is necessary to perform their job. This can create a potential threat if such privileges are abused, either by the user or an attacker compromising the user's account. Cloud environments may also be vulnerable to SQL injection, i.e., injecting SQL queries to act against the objective of an application. The injected SQL statement is inserted into an executable statement which, in turn, can fetch data that the attacker wants to obtain (e.g., injecting a query to retrieve all of the records in a given table). Malicious insider attacks are a form of attacks that are performed from inside the cloud organization, with very little chance of detection. The attacker can access sensitive data and leak or maliciously process them in a way that violates system policies.

A data breach is defined as the accessing or obtaining of sensitive information—such as medical records, student information, employees' salaries, and so on—in an unauthorized manner. Such problems occur when data access is not restricted and when API access control is weak. Other cloud attacks may exploit the following: weak authentication; unpatched services; or insecure system architecture (e.g., keeping sensitive and non-sensitive data in the same database without implementing any form of encryption for the sensitive data).

One solution to concerns about the above-mentioned vulnerabilities is applying data encryption to the sensitive data. For example, in an SQL injection attack, if the attacker injects a

query to fetch the entire set of the database, the attacker will learn nothing if the sensitive data are encrypted. However, security level differs by the type of encryption used. Weak encryption techniques can be compromised by cryptographic attacks, which exploit a vulnerability in the cryptographic, such as a weakness in a cipher, key management scheme, code, or cryptographic protocol. In this dissertation, we focus on encryption strategy as a method of protecting data from the cloud database vulnerabilities listed above. In the next sub-section, we survey the most popular encryption schemes that have been used by state-of-the-art database security systems in the field of cloud computing.

2.2 Encryption-Based Solutions to Protect Cloud Databases

Encryption is defined as encoding plaintext into unreadable formats, which preserves the confidentiality and privacy of the data. There are two types of encryptions: symmetric and asymmetric. In symmetric encryption, only one encryption key (i.e., the secret key *sk*) is used to encrypt and decrypt the data. The *sk* is known by both the encryptor and decryptor entities; however, it must be kept secret. The most popular symmetric encryption algorithms are the Advanced Encryption Standard (AES) and Blowfish algorithms. In [13], the authors conducted a comparative analysis of the AES and Blowfish algorithms and found that AES was faster than Blowfish by nearly 200 ms when used to encrypt or decrypt the same file. Asymmetric encryption, on the other hand, involves using two keys—a public key and a private key—to encrypt and decrypt the data. To guarantee confidentiality, the receiver's public key is used to encrypt the data, while the private key, associated with the receiver's public key, is the only key used in the decryption process and must be kept secret. Asymmetric encryption systems are computationally intensive and slow down the encryption and decryption processes [14]. Some examples of well-

known asymmetric encryption systems include RSA, ElGamal, Diffie-Hellman, and ECC. For more details about asymmetric encryption algorithms, see [15], [16], and [17].

When choosing an encryption system, a variety of factors must be considered, including security level, computation overhead, and complexity, among others. For instance, the computation overhead of asymmetric encryption is higher than that of symmetric encryption because, in an asymmetric system, the usage of CPU cycles is higher than in a symmetric scheme. Moreover, the security level in both systems differs based on the type of algorithm used (e.g., randomized algorithms are more secure than deterministic algorithms) and whether a strong *sk* was used to encrypt the data. In a symmetric system, the security level is high, and the chance of compromising the *sk* (for, e.g., a *sk* length of 256 bits in AES) is virtually nonexistent. Accordingly, symmetric algorithms are more widely used than asymmetric algorithms.

Data encryption is the only solution for protecting outsourced databases that prevents data leakage resulting from any form of unauthorized data access in the cloud. However, it is challenging to execute SQL queries over encrypted databases. The existing literature on cloud database security offers a variety of techniques to overcome this problem and deal with outsourced encrypted databases. While most state-of-the-art systems aim to provide security, efficiency (i.e., time required to execute SQL queries) varies depending on the technique used; the higher security level provided, the lower performance level achieved. In the following subsection, we explore each strategy used to deal with outsourced encrypted databases. We then review the research offering solutions within each strategy.

2.3 Current Approaches to Process SQL Queries over Encrypted Databases

Early attempts to secure databases encrypted the whole database, with each record encrypted as one block (e.g., if a table has four sensitive columns before encryption, the encrypted table will

have just one column to store the encrypted values). However, as mentioned earlier, problems occur when SQL queries must be executed over the encrypted data. The simplest solution to this matter is to fetch the entire outsourced table and decrypt it, then execute the query. While this approach can work well for small databases, it suffers from higher computation costs when applied to larger databases (e.g., a table holding millions of records).

Researchers have proposed numerous solutions to avoid retrieving entire outsourced encrypted databases by classifying records into categories before proceeding to the encryption process. The encrypted table will have an additional column(s) to store the category of the record. By using these categories, the end-user is able to retrieve only part of the encrypted data. In addition, the amount of fetched encrypted tuples is impacted by how data are categorized in the cloud (i.e., the more data categories there are, the fewer data are fetched). To address the issue of categorizing data, many authors (e.g., [18]–[20]) have proposed approaches to dividing attributes into categories that can be used to query encrypted data. The techniques are based on dividing each attribute into ranges. The main encrypted table in the cloud will then have additional attributes—as many as the number of partitions among all attributes—to hold numeric values. The whole record will then be encrypted as one block and stored as an attribute value in the cloud. The early attempts technique was developed by [10], who proposed different techniques to execute relational algebra operators over the encrypted records. Their solution assigned an identifier for each value in the tuple, then used those identifiers to retrieve only encrypted records whose identifiers matched the requested identifier. Nevertheless, there are several limitations to using such an approach, including vulnerability to statistical attacks, as mentioned in [21], and heavy client-side computation due to decrypting every retrieved record's data (because all the fields of each record are encrypted as one block). The authors in [22] proposed a system to build an index for the plain

data, then encrypt each page of the index individually. To execute a query, the corresponding page is loaded and decrypted. However, since all pages are encrypted with one key, the security of this scheme is downgraded. In addition, the size of the index will continue to grow, which could impact performance. To improve the security level, a unique encryption key could be used to encrypt each page of the index. In [21], the researchers suggested building a B-tree index, maintained on the client-side, over the plaintext data. In [23], the authors introduced a single values level encrypted index and suggested splitting the index into sub-indexes, i.e., each sub-index is for encrypted values using the same key in the column. The authors in [24] proposed a none–order-preserving index for the encrypted database. This index does not require interaction with the user once the query is submitted. The security of this scheme is higher than that of models based on order-preserving indexes, which may be vulnerable to statistical attacks. Hahn et al. [25] propose a system to join encrypted databases. The idea is based on applying a selection operation first, then enforce the join over selected data. That only leaks the frequency of use and access patterns. This method is interesting; however, the delay is high because of the asymmetric cryptosystem they use.

In [2], Popa et al. developed CryptDB as the first practical system for executing Standard Query Language (SQL) queries over encrypted databases. Two attack scenarios were addressed using onion layers encryption: cloud attack and proxy attack. Each datum is encrypted by more than one encryption algorithm in which the outer layer ciphertexts produced by a randomized encryption algorithm. CryptDB uses a proxy to perform the crypto operations for the user. One of the drawbacks of CryptDB is that, because of the excessive crypto operations and many layers of decryption, it introduces a high computational burden. In addition, because it is challenging to execute an analytical load to encrypted data on a server, CryptDB was improved in [26] to support

complex queries and large data sets. MONOMI solves this problem by splitting the execution into two sets: a set of queries to outsourced encrypted data and a set to be executed on decrypted data on the user's side. Authors in [27] proposed an enhanced version of CryptDB to accelerate query processing. Instead of using AES, they used AES-NI, which was reflected in the speed of the query processing time. They also suggested improvements to the hardware to accelerate query processing in CryptDB. There are many different systems proposed on top of CryptDB, such as the one presented in [28].

Liu et al. [29] proposed a fully homomorphic order-preserving encryption system (FHOPE) to execute complex SQL queries over encrypted numeric data. This system allows cloud providers to run arithmetic and comparison operators over encrypted data without repeating the encryption, thus helping to resist homomorphic order-preserving attacks. The downside of that study is that the authors conducted their experiments using tables with less than 9,000 records. For improved measuarment of the efficiency and scalability of this system, the tables should have more records (e.g., 100,000 or more). A variety of studies related to this system are provided in references [30]–[34].

Cui et al. proposed P-McDb [35], a privacy-preserving search approach that allows users to execute queries over encrypted data. To avoid inference attack, this system requires two cloud servers, one for database re-randomizing and shuffling and one for data storing and searching. Instead of a total search, P-McDb supports partial searches of encrypted records that are described as a sub-linear manner. Further, P-McDb is a multi-user system. In the case of a user revocation, the data cannot be re-encrypted. Another limitation of this system is that communication with two cloud providers will add more latency when compared to other systems such as those described in [2]. More proposed systems related to P-McDb are described in references [9], and [36]–[39].

Osama et al., in [40], proposed different approaches for partitioning attributes of tables into multiple sub-columns based on the attribute's domain values. The methods were tested and introduced various delays. They use an order-preserving mapping function, which enables cloud servers to run different types of SQL-queries. The major disadvantage of this research is that only attributes with numeric values but not with string values were considered. Moreover, such a system only supports select statements.

In [41], the researchers proposed a secure database (SDB) approach, a system that divides data into sensitive and non-sensitive, with only sensitive data being encrypted. The initial idea was to split the sensitive data into two shares. The data owner (DO) keeps one share, and the second share is kept by the cloud service provider (CSP). Assuming the CSP is *curious*, the CSP can learn nothing from its share unless it obtains the DO's share. Also, the SDB allows different operators to share the same encryption, thus providing secure query processing with data interoperability. Similar studies can be found in [26], [42], and [43].

As presented in [44], some researchers used a technique called "Bucketization," in which the tuples are mapped to more than one bucket. This technique enables a "database as a service" (DAS) server to execute SQL-style queries over encrypted data. Each bucket contains a set of encrypted records ranging from the minimum to maximum value and assigned an identification (ID). Several studies based on this approach have been conducted [45]–[48].

Some researchers ([35], [49], [50]) have addressed cloud database privacy by adopting what is called a hybrid cloud. The technique is based on dividing data into sensitive and non-sensitive. Then, the sensitive data or attributes are outsourced to the user's private cloud while the non-sensitive data are migrated to the public cloud. The problem is that, because most users

consider their data to be sensitive, this scheme is not practical for users of non-sensitive data. Also, the complexity of integration for this solution is high.

On the other hand, Amjad et al. [9] proposed a technique to prevent untrusted and suspicious cloud service providers from being able to learn from private data. This technique is based on vertical fragmentation, in which each sensitive encrypted column is outsourced to a different cloud server (slave cloud). In contrast, while the whole encrypted table is stored at the central server (master cloud). Because the encryption algorithms [2] and the proxy were used in this system, the proxy performed all the work of interpreting queries, encryption, and decryption. One of the limitations of this work is more communication delays, especially if the query condition contains more than one clause. Another example of research that uses this technique is [51].

Bouganim et al. [52] introduced a hardware/software system to address the problem of confidentiality leakage in the outsourced databases. The idea is that the user maintains and controls a mediator smartcard that is plugged in on the side. This smartcard is responsible for encrypting the data before putting them into the database and decrypting data before sending them to the user. The major disadvantage of this technique is that the user is limited by the capacity of the smartcard and cannot benefit from the storage provided by the cloud services. Similar studies can be found in [52]–[54].

SafeBox [55] is a system based on an approach called access security broker (CASB). This approach allows users to search and share encrypted data while protecting sensitive information from being leaked if an attacker gains access to the cloud server (CS). This technique can be applied over encrypted databases or files and supports keyword-based searches within the encrypted contents. Several studies that use CASB can be found in [56] and [57]. Also, a detailed survey about the use of brokers in the Cloud can be found in [58].

3 Methodology

3.1 Introduction

The primary goal of this research is to address the significant drawbacks of some of the state-of-the-art research in the field of cloud database security. They are described below.

Onion layers encryption means encrypting each datum using different encryption algorithms. The inner layer is the ciphertext of the algorithm with the lowest security level, while the outer layer is the ciphertext of the algorithm with the highest security level (i.e., randomized encryption algorithm) [2]. When it comes to search for a value, the whole column's values must be updated to the next layer (take off layers) This process might be executed more than once to achieve the desired result. For a large encrypted table, more excessive crypto operations are performed, leading to substantial computational overhead. To enable the cloud server to remove and adjust layers, the secret key is passed to the server, making the system vulnerable to an in-session attack. Also, the trusted but curious cloud provider(s) could learn about the data if the security layer is adjusted to a low-security level layer. To overcome this limitation, our proposed approaches are designed to encrypt each datum in the table using only a randomized encryption algorithm (AES-CBC). Also, we fetch only those encrypted rows from the cloud server that are related to the query of the user, which reduces undesirable computations. We eliminate passing the secret keys to the cloud server to ensure that curious cloud provider(s) cannot learn from the outsourced database.

In systems that use vertical fragmentation (i.e., column-based fragmentation), the table is fragmented throughout a multi-cloud. So, each column is outsourced to a different cloud to preserve privacy and speed up the query processing [9]. This approach might be practical for tables that have a few attributes but not for tables that have hundreds of attributes. The reason is because

the table owner must have multiple accounts with more than one cloud server. If two or more cloud providers collude, privacy will be compromised.

Communication delay is another concern when using such systems. To address this issue, the developed systems require only one server to outsource the encrypted table, a feature that will minimize communication costs, improve privacy, and accelerate query processing.

Homomorphic encryption (HE) is a technique in which SQL queries can be executed over the ciphertexts as if the data were not encrypted. HE is used to encrypt only numeric values and support arithmetic operators over encrypted numeric data. However, the space required to store the ciphertexts is too large. Also, the integrity of encrypted data is not preserved in such systems because the attacker can change the ciphertexts undetected. To date, this type of encryption supports only addition and multiplication over encrypted digital data. To overcome this, we use a symmetric randomized algorithm AES-CBC to preserve integrity because the modification of a ciphertext leads to an incorrect decryption result.

3.2 Encryption Strategy

To provide privacy and a high level of security, all of our approaches used AES-CBC to encrypt sensitive data. Only the query manager (QM) keeps and maintains the secret keys (SKs). We used a symmetric algorithm rather than an asymmetric algorithm to gain a higher level of security and faster crypto processing.

3.3 The Query Manager (QM)

In our work [59]–[61], we defined the query manager (QM) as a trusted server that resides in an organization's or company's private cloud. It works as an intermediary between users and the Cloud and is responsible for processing queries and encoding BVs for each table (we explain this step in detail in the discussion of partitioning trees). Also, the proposed prototypes support

individuals' cases in which QM would be light software residing in the end user's system. For an organization, it performs the same functions as the QM server. While we assume that the user can encrypt only columns that have sensitive data, the proposed approaches even support encrypting all of a table's attributes.

3.4 Partitioning Tree (PT)

As stated in our previous studies [59-61], the PT is the primary element of all proposed systems in which the query is appropriately rewritten for execution by the cloud server. The owner of the table participates in the construction of the PT by specifying which columns are sensitive and should be encrypted. Then, the table owner defines the possible partitions for each column and indicates whether the partitions are ranges of values or non-ranges of values. Thus, the values in each column are partitioned into multi-partitions in which each partition includes a set of values. Then, the QM builds the PT based on these specifications. As shown in Figure 3.1., the name of the table is the root of the tree, and the second-level nodes are the sensitive columns that have to be encrypted. Nodes in the third level, each of which is assigned an ID, represent the partitions of all the sensitive columns.

Table 3.1 The Students Table

ID	Name	SSN	VisaType	Department
01	Alice	12701	J-1	MATH
02	Ryan	25678	F-2	BUSN
03	Mark	46932	F-1	CSCI
04	John	42213	J-2	PHYS

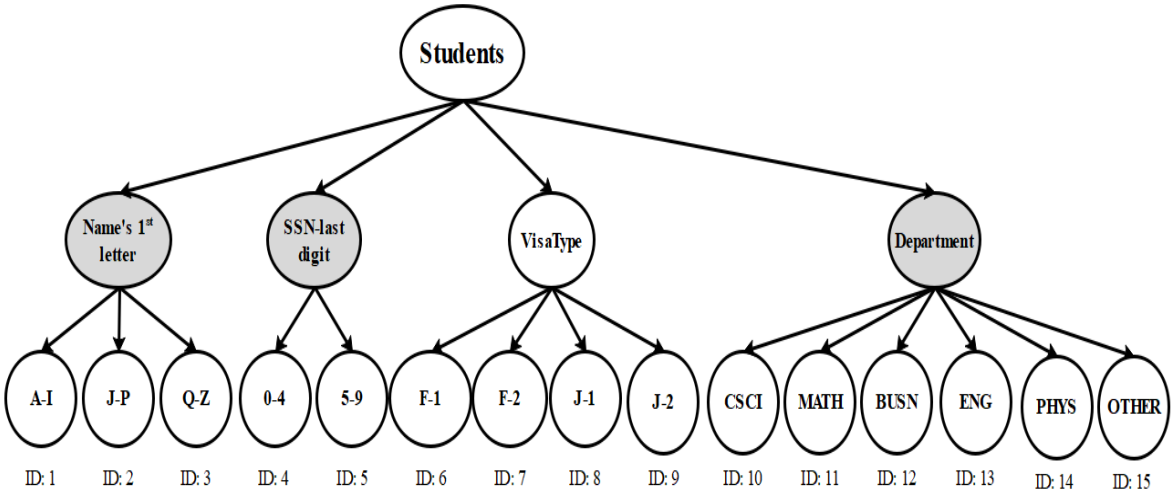


Figure 3.1 An example of the partitioning tree (PT) of the students table (Table 3.1)

The second-level nodes are assigned a color of either white or gray. Gray nodes imply that the partitions of the column are ranges of values (e.g., the Name column must have range partitions based on the first letter of the name, while students Visa Type have non-range partitions, such as F-1, F-2, etc.). The PT is stored locally in the QM. We encourage the data owner to define as many as possible partitions for each SC, which can narrow the range of retrieved encrypted records from the outsourced table and, in turn, increase the performance of the proposed systems, and achieve faster query processing.

We are not concerned about memory consumption in our solutions because the sensitive information in every row is encoded into bits (i.e., the smallest computation unit). Based on the PT, this does not consume memory or searching time. In section 3.6, we explain in detail how and where to store bit vectors in each proposed system. Algorithm 1 shows how the QM constructs the PT of the students' table (Table 3.1) and Figure 3.1 shows the PT of the students' table.

Algorithm1: PT construction

Input: file.txt containing the name of the table, sensitive columns, and partitions.
Create a tree and add a node, represent the table name (TN), as the root of the tree.
Int Id=0
For each sensitive column SC_i
    $Node_i$ = TN.addchild ('the name of the SC_i')
    If SC_i contains range of values
        $Node_i$.AssignColor ('Gray')
    Else
        $Node_i$.AssignColor ('White')
    While (partitions! = null)
        $Node_j$ = $Node_i$.addchild('partition value')

3.5 Encoding Approach

The encoding process is an essential step in the proposed systems. The proposed scheme is used to encode records' sensitive data into bit vectors (BVs), which can be exploited to retrieve the required encrypted records (i.e., candidate records for the user's query without the need for decrypting data). The QM parses each record to get the names of the sensitive columns and their values. It then uses the PT to encode the tuples before proceeding to the encryption process, to bit vectors (BVs). Each bit position in the BV is mapped to a partition node from the third level nodes (e.g., the first bit in the BV is mapped to the node having the ID =1). The encoding process is accomplished as follows:

1) For each record R_j in the main table T, the QM creates a BV having a length equal to the number of nodes in the third level of the corresponding PT. It then initializes all its bits to zeroes (e.g., if the bit vector has 10 bits, the number of nodes at the third level of the PT is 10).

2) For each record R_j, the bit b_m that mapped to the partition node PN_m under SC_i is set to one if the datum equals the values represented by PN_m. Then, the QM assigns an index to the newly created BV.

For the sake of clarity, the encoded BVs of the records in Table 3.1 are as follows:

1 <100100010010000>
2 <001010100001000>
3 <010101000100000>
4 <010100010000010>

Figure 3.2 The BVs of Records in Table 3.1

How and where to store the BVs differs for each proposed system. In section 3.6, for each prototype, we present the details of how and where to store the BVs (i.e., either locally at the QM or by migrating them to the cloud server). We also discuss the security and the potential threats in each model and show how to solve them. Algorithm 2 delineates the process of the encoding step.

Algorithm 2: Encoding algorithm

1: **Define** a vector V
2: **For** each record R_j in Table T
Parse R_j to get the value(s) of the sensitive columns *SCs*
3: **Define** a bit vector BV_j of length n where n = the number of nodes in the 3rd level of the T
4: **For** each SC_i in R_j
5: **If** value v = value of the n^{th} sub-node of the SC_i **OR** $v \in$ value of the n^{th} sub-node of SC_i
6: **Set** n^{th} bit in BV_j to 1
7: **Else**
8: **Set** n^{th} bit in BV_j to 0
9: **End for**

3.6 Developed Prototypes

3.6.1 Bit Vectors as a Matrix (BVM)

In this section, we discuss the model that uses bit verctors as a matrix. Some passages, figures, tables, and algorithms presented in subsections (3.6.1.1 and 3.6.1.2) have been quoted verbatim from our published work in [59].

3.6.1.1 System Description

In this prototype, we store and process the BVs of each outsourced table locally at the QM. We assume that the QM is a trusted server residing in either the end user's machine as an application or in the private cloud (Figure 3.3). Because the only thing outsourced in this prototype is the encrypted table, the highest level of security is provided. The outsourced encrypted table preserves the structure of the original table. However, we add a column (used as a foreign key) to store the rows' indices (during the encoding process, the QM assigns a unique index number to every encrypted row and its BV). Further, we need these indices to fetch the encrypted records. To process a query, the QM needs to load the BVs to the main memory from the hard disk drive (HDD) and perform a rapid look-up to find which records are candidates for the user's query. Then, it pushes their indices into a list and rewrites the query to fetch any record whose index is on the list. In the following sub-sections, we present the supported statements for the relational algebraic operations.

3.6.1.2 Basic Operations

The basic operations in database applications *are insert, select, update, alter,* and *delete* statements. We extended this prototype to support these operations, as explained below.

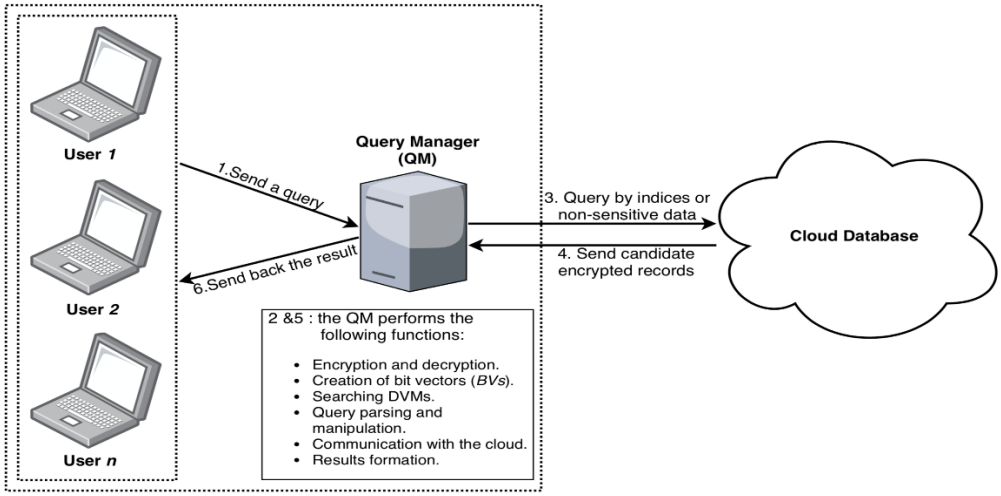


Figure 3.3 BVM Architecture.

Insertion statements are straightforward. The QM receives the *insert* query, creates a new BV for the newly inserted record, appends it to the corresponding bit vectors matrix (BVM), and then encrypts sensitive data and sends it to the Cloud.

Select is an essential statement in all database applications. In our approach, it is also part of the execution of other statements like *update* and *delete*. In the QM, the process of executing the *select* statement algorithm is to check which of the following cases is applicable and then enforce it.

Case 1: None of the column(s) in the query condition is sensitive, so they are not stored in encrypted form in the Cloud. In this case, the QM directly searches and retrieves the data corresponding to the query condition from the Cloud. For example, in Table 3.1, Figure 3.1, and Figure 3.2, if the query condition is "WHERE ID = 03", the QM retrieves the records directly from the Cloud that satisfy this condition.

Case 2: All column(s) in the query condition are sensitive, so they are stored in encrypted form in the Cloud. In this case, for each column that appears in the query condition, the QM retrieves the indices of corresponding bit vector(s) (BVs) from the BVM. Then it performs logical

AND/OR operations based on conditions among indexes returned for each column. For example, if the query condition is "Name = Mark AND visa type = F1", the QM will find any BV having the bit that mapped to the node representing the values of the PD "Q-Z." Because "Mark" belongs to this group (i.e., PD "Q-Z"), any BV has the bit mapped to this partition is set to one will be added to a list L[]. The QM will do the same for visa type, then perform the logical operation AND between the two lists and rewrite the query accordingly to fetch the candidate records.

Algorithm 3: select

1: Receive user query < Table name, List of columns {c_1, c_2, c_3 …}, list of values {v_1, v_2, v_3 …}>
2: **Check** query columns used in search condition
3: CASE 1:
4: **If** none of columns mentioned in query is sensitive
5: **Search** the corresponding table in cloud
6: Return data.
7: CASE 2:
8: **If** each column C_i is sensitive
9: **For** each value *v* being searched for under column C_i
10: **Search** the column representing that domain value in PT then for each BV_j entry under domain value in C_i
11: **If** bit =1
12: **Add** the index of BV_j to List *Li* [ ]
13: **Define** list *F* where *F* [ ] is the final list that contains the indices from the lists (L_1 [ ], L_2 [ ],…, *Li* [ ]) after performing AND or OR (based on query conditions) operations between them.
14: **Return** *F*
15: **Retrieve** encrypted records from the cloud based on the list *F* [ ].
16: **Decrypt** and return data
17: CASE 3:
18: **If** mixed columns (Sensitive and Non-sensitive)
19: **Do** Case 1 AND Case 2
20: **Remove** duplication if found
21: Return data

The ***update*** process is one of the functions provided by the QM. Through this function, the user issues a query to update record(s). The QM identifies record(s) using our *select* algorithm. Then based on the results of *select* algorithm, the QM issues a query to retrieve encrypted record(s) from the Cloud, decrypts them to find the exact records that match the query conditions, and issues a query to update the encrypted record and its BV. The steps are shown in Algorithm 4.

Algorithm 4: update

1: Receive user query < Table name, List of columns {c_1, c_2, c_3 …}, List of values {v_1, v_2, v_3 …}>
2: **Use** search Algorithm (Algorithm 3) to find the candidate record(s)
3: **Fetch** the record(s) from the encrypted table
4: **Decrypt** and find the exact record(s)
5: **For** each update value *uv* in record R_j
6: **If** *uv* falls under non-sensitive column
7: **Update** the old value with *uv* in the outsourced table
8: **If** *uv* falls under a sensitive column SC_i
9: **For** each SC_i in the update query
10: **If** the *uv* falls under a *PD* other than the previous *PD*
11: **Set** the bit that mapped to this *PD* to 1 and unset the rest of bits that mapped to other *PDs* for this $SC_{i..}$
12: **Encrypted** the updated values
13: **Send** data back to the cloud

The ***delete*** process is used to delete a record from a table. In this case, the QM uses the search algorithm to find the candidate record(s), then removes the record(s) from the outsourced encrypted table and deletes the corresponding BVs from the BVM. Algorithm 5 outlines the steps in the deletion process.

Algorithm 5: delete

1: **Use** Algorithm 3 (select algorithm) to find the required record.
2: **Add** the index of each fetched record satisfying the delete condition into list L [].
3: **Generate** a delete query to delete any record from the outsourced encrypted table its index is in L [].
3: **Delete** the BVs of the deleted records from the corresponding BVM.

Alter is one of the fundamental operations in any database that allows users to drop/add columns from/to relations. However, in this model, the QM first determines whether the dropped or added column is a sensitive column. In the case of "drop," the QM will look at the corresponding partitioning tree (PT) and drop all partition nodes of the predecessor node that represents the dropped column. It then deletes all bits mapped to the deleted nodes from the BVM. The QM then forwards the alter query to the Cloud, which will execute the query to drop the encrypted column. In the "add column" case, the QM asks the user if the column is sensitive. If it is sensitive, then it will add it with its partitions to the PT after receiving information from the user.

3.6.1.3 Relational Algebra Operators

3.6.1.3.1 Join

Most current research in database security does not support the *join* operator because dealing with encrypted databases it is not a straightforward task, especially if AES-CBC is the encryption algorithm. The simple solution for such a task is to retrieve all encrypted tables from the Cloud and perform the *join* operator after decrypting them. However, this is not the optimal choice when it comes to massive tables. In this model, to avoid unnecessary computation, we need to move as much of the join computation as possible to the cloud site without decrypting data, leaving minimal work for the QM. To make BVM both practical and efficient in the *join* operator, we need to consider the following cases for the join condition:

1) The join condition has only non-sensitive columns.

2) The join condition involves only sensitive columns having limited distinct value partitions such as USA visa types.

3) The join condition contains only sensitive columns that have range partitions such as salary.

4) The join condition has at least two of the previous cases. So, we design an algorithm to enable the cloud provider to implement a join operator over encrypted tuples without decrypting the encrypted attributes.

To solve the first case, in which the join condition involves only non-sensitive attributes (i.e., unencrypted attributes), the QM will first determine if the attributes are sensitive or not. If they are non-sensitive, it will forward the query to the cloud database, which will implement the join query and return the join result to the QM. The QM then decrypts the join result and removes duplication if found. In the second case, the join condition contains only sensitive attributes that are not ranges of values. In this case, the QM creates a list L_i [ ] for every partition of the attribute mentioned in the join condition, where L_i [ ] will have the index of any record having its bit that mapped to partition node *i* is set to 1, before searching the BVM. Then the QM will rewrite the query to join all the tuples from both tables based on the indices of these lists. For example, there are two tables, A and B, both having the same attributes: ID, name, rank, department, salary. Now, assume that the query was to join them where A.rank = B.rank. Then let us say that the partition domains for rank attributes are manager, secretary, and employee. Now, the QM will create three lists for each table. The first list will contain the indices of the bit vectors in the corresponding BVM that have their bits mapped to the PD "manager" set to 1. The second list will hold the indices that have the bits mapped to the PD "secretary" set to 1, and the third list will contain the indices that have the bits mapped to the PD "employee" set to 1. In the next step, the QM rewrites the query to join the

tuples that have their indices in list L_i [ ] from Table A with the tuples that have their indices in list L_i [ ] from Table B. In the third step, the Cloud will execute the query and return the result to the QM, which will decrypt and remove any duplications before sending the result back to the user.

In the third case, the join condition contains only sensitive columns that are ranges of values. This case is similar to the second case. However, in this case, the mapping between joined partitioning lists can be one too many. To illustrate this, consider Figure 3.4. Assume we want to join tables A and B by equality of salary. The salary column in Table A has three PDs that are [(10,000 to 20,000), (20,001 to 30,000), (30,001 to 40,000)], and the salary column has two PDs that are [(10,000 to 25,000), (25,001 to 40,000)]. We need to make sure that the QM rewrites the join query in a way that it maps the PDs from Table A to the corresponding PDs from Table B. To do that, the QM will create *n* lists for each table where *n* is the number of PDs in the table that has the fewest PDs in the joining column. So, in the above example, n =2 since Table B has the least PDs. In the first list L_1[ ] in table A, the QM adds indices with bits that represent PD_1, or PD_2 is 1. In the second list L_2[ ] of table A, the QM adds indices with bits that represent PD_2 or PD_3 as 1. The following figure, Figure 3.4, shows how the PDs are mapped. The QM rewrites the query before forwarding it to the Cloud. After getting the join result back from the Cloud for each tuple, the QM decrypts only the encrypted column's value (only the encrypted columns involved in the join condition) and enforces the join condition. If the join condition is satisfied, the QM proceeds to decrypt all of the tuple's values before moving to the next tuple. If the join condition is not met, the QM will not decrypt the entire tuple's values and will move to the next tuple. We do so to avoid unnecessary decryption processes for those tuples that do not meet the join condition. The

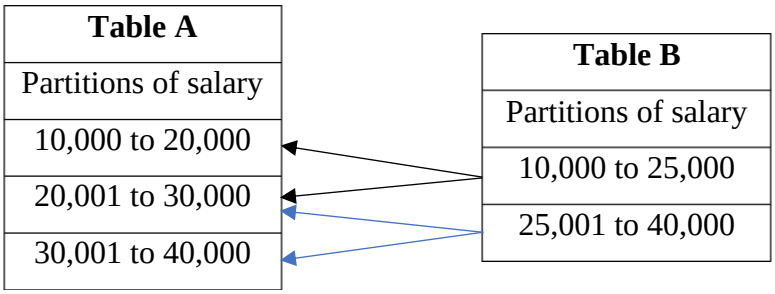


Figure 3.4 An example of joining two tables by the equality of salary values

last case is when the join condition involves two or more of the previous cases. In that case, the QM might rewrite the query. Algorithm 6 below shows how the QM performs the join.

Algorithm 6: Join

Input: the QM Parses user's query to get tables' names, attributes, and values.
QM checks the PT of each table in the join and performs the following:
Case 1
If join column(s) is non-sensitive (not present in the PT) THEN
    Forward the query to the cloud server (CS)
    Define LinkedHashSet *s*
    For every fetched record *i*
        decrypt all values in *i*
        push *i* to *s*
    send *s* to the user
Case 2
If join column(s) is a sensitive AND not ranges, THEN
    For each PD_j under the node that represent a SC_i in PT of Table x
        Create a list Lx_j [ ]
        For each BV in BVM_x
            If the bit mapped to PD_j is not 0
                Push the BV' index to Lx_j [ ]
        For each PD_j under the node that represent a SC_i in the PT of Table x_1
            If the value v of PD_j of SC_i from table x_1 equals to the v of PD_j of SC_i from Table x_2
                PD_j of $x_1 \bowtie PD_j$ of x_2
        Rewrite the query and send it to the cloud
        Define LinkedHashSet *s*

23: **For** every fetched record *i*
24: **decrypt** all values in *i*
25: **push** *i* to *s*
26: **Send** *s* to the user
27: Case 3
28: **If** join column(s) is a sensitive AND ranges, THEN
29: **For** each PD_j under the node that represent the SC_i in PT of Table x
30: **Create** a list Lx_j [ ]
31: **For** each BV in BVM_x
32: **If** the bit mapped to PD_j is not 0
33: **Push** the BV' index to Lx_j [ ]
34: **For** each PD_j under the node that represent the SC_i in PT of Table x_1
35: **If** the PD_j of SC_i from table x_1 contains at least one value v where v ∈ PD_j from table x_2
36: PD_j of x_1 ⋈ PD_j of x_2
37: **Rewrite** the query and send it to the cloud
38: **Define** LinkedHashSet *s*
39: **For** each fetched record *i*
40: **Decrypt** only the join columns' values
41: **If** the join condition satisfied
42: **Decrypt** the whole tuple's values
43: **Push** *i* to *s*
44: **Else**
45: **Proceed** to the next encrypted tuple
46: **Send** *s* to the user
47: Case 4
48: The query *involves* two or more of the above cases.

3.6.1.3.2 Union

Union is one of the widely used operations in database systems in which tuples from two or more tables are merged. However, to execute a union operator, the number of attributes and datatype of both tables must be compatible. Even though the union operation removes the duplication from the union results, dealing with encrypted data complicates this step. To accomplish this, we could enable the cloud server to execute the union operator over encrypted tables and then send the result back to the QM. Doing so will move the union computation to the cloud server leaving only

decryption and duplication removal to the QM. The QM will decrypt each tuple in the union result set and add it into a LinkedHashSet as soon as it receives it from the Cloud. Note that both tables must be encrypted with the same secret key to execute the union algorithm. Algorithm 7 delineates the processes.

Algorithm 7: union

```
1:  Forward the user's query to the cloud server
2:  Define a vector v
3:  For each fetched record i in the union result set
4:      Decrypt i
5:      Add i to v
6:  v.distinct()
7:  Send v to the user.
8:  v.clear()
```

3.6.1.3.3 Intersection

Intersection is the process of finding the common subset out of two or more sets. However, executing intersection over encrypted databases is not easy without decrypting the data. If the tables to be intersected have a large number of records, the user is going to add a significant computational overhead by decrypting the whole set of the encrypted tuples from the intersected tables before executing the intersection operator. As a result, we cannot benefit from the cloud services because the computation is moved to the user's side rather than on the cloud side. Some of the previously proposed systems in [2], [9], [10] can process queries over encrypted databases but will experience delays if the tables to be intersected have large numbers of tuples.

In this model, our goal is to move the computation as much as possible to the cloud side while eliminating unnecessary decryption processes at the QM. Moreover, we want to execute the intersection operator partially in the cloud database server leaving only the elimination of duplication at the QM. In this way, we exploit the high computational speed provided by a cloud

database server to accelerate the query processing time. We explain the simulation of intersection as follows:

- The user sends the query to the QM, which is going to parse it to remove the headers of tables and columns.
- If the intersect operation involves *k* columns out of *n* columns, where *n* denotes the total number of columns in a table, the QM uses our *join* algorithm to join both tables by *k* - columns and then rewrites the query. Otherwise, the QM chooses all *SCs* that are not in ranges to join the tables using our *join* algorithm before rewriting the query.
- The QM sends the translated query to the cloud database server, which will execute the query and send back the *join* result to the QM.
- Before returning the *intersecting* result to the user, the QM decrypts the encrypted *join* set and pushes it to a hash list to remove duplicates, if found.

To illustrate the process, consider this example. Suppose we have two tables, A and B, and we want to carry out an intersection between them. Assume both tables have the same columns and the same PT as those shown in Figure 3.5.

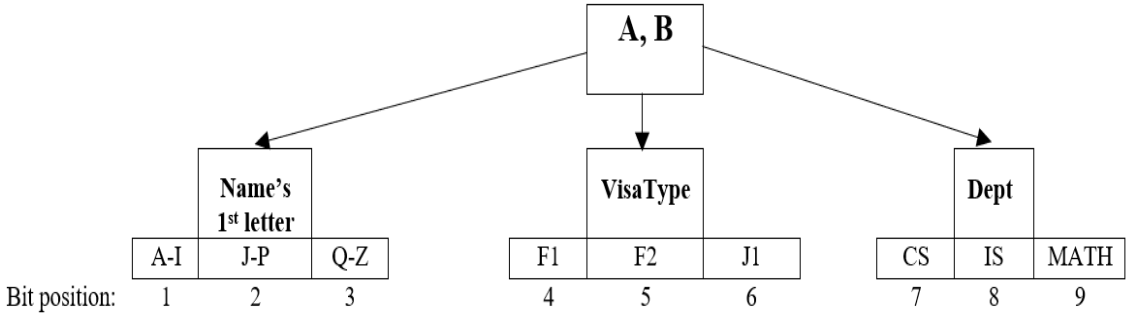


Figure 3.5 An example of a PT for Table A and Table B.

The translation process of the intersection query is accomplished as follows:

1: Join any record from table A with any record from table B if and only if the BVs of both records have the 1st, 4th , and **7**th positions are set to 1.

2: Join any record from table A with any record from table B if and only if the BVs of both records have the 1st, 4th , and **8**th positions are set to 1.

3: Join any record from table A with any record from table B if and only if the BVs of both records have the 1st, 4th , and **9**th positions are set to 1.

4: Join any record from table A with any record from table B if and only if the BVs of both records have the 1st, 5th , and **7**th positions are set to 1.

5: Join any record from table A with any record from table B if and only if the BVs of both records have the 1st, 5th , and **8**th positions are set to 1.

6: Join any record from table A with any record from table B if and only if the BVs of both records have the 1st, 5th , and **9**th positions are set to 1.

7: Join any record from table A with any record from table B if and only if the BVs of both records have the 1st, 6th , and **7**th positions are set to 1.

8: Join any record from table A with any record from table B if and only if the BVs of both records have the 1st, 6th , and **8**th positions are set to 1.

9: Join any record from table A with any record from table B if and only if the BVs of both records have the 1st, 6th , and **9**th positions are set to 1.

10: Join any record from table A with any record from table B if and only if the BVs of both records have the 2nd, 4th , and **7**th positions are set to 1.

11: Join any record from table A with any record from table B if and only if the BVs of both records have the 2nd, 4th , and **8**th positions are set to 1.

12: Join any record from table A with any record from table B if and only if the BVs of both records have the 2nd, 4th , and **9**th positions are set to 1.

13: Join any record from table A with any record from table B if and only if the BVs of both records have the 2nd, 5th , and **7**th positions are set to 1.

14: Join any record from table A with any record from table B if and only if the BVs of both records have the 2nd, 5th , and **8**th positions are set to 1.

15: Join any record from table A with any record from table B if and only if the BVs of both records have the 2nd, 5th , and **9**th positions are set to 1.

16: Join any record from table A with any record from table B if and only if the BVs of both records have the 2nd, 6th , and **7**th positions are set to 1.

17: Join any record from table A with any record from table B if and only if the BVs of both records have the 2nd, 6th , and **8**th positions are set to 1.

18: Join any record from table A with any record from table B if and only if the BVs of both records have the 2nd, 6th, and **9**th positions are set to 1.

19: Join any record from table A with any record from table B if and only if the BVs of both records have the 3rd, 4th , and **7**th positions are set to 1.

20: Join any record from table A with any record from table B if and only if the BVs of both records have the 3rd, 4th , and **8**th positions are set to 1.

21: Join any record from table A with any record from table B if and only if the BVs of both records have the 3rd, 4th , and **9**th positions are set to 1.

22: Join any record from table A with any record from table B if and only if the BVs of both records have the 3rd, 5th, and **7**th positions are set to 1.

23: Join any record from table A with any record from table B if and only if the BVs of both records have the 3rd, 5th , and **8**th positions are set to 1.

24: Join any record from table A with any record from table B if and only if the BVs of both records have the 3rd, 5th , and **9**th positions are set to 1.

25: Join any record from table A with any record from table B if and only if the BVs of both records have the 3rd, 6th , and **7**th positions are set to 1.

26: Join any record from table A with any record from table B if and only if the BVs of both records have the 3rd, 6th , and **8**th positions are set to 1.

27: Join any record from table A with any record from table B if and only if the BVs of both records have the 3rd, 6th, and **9**th positions are set to 1.

3.6.1.3.4 Difference

To execute the difference in this prototype, the intersection is first enforced between the tables to find the common tuples. Second, the QM sends a query to retrieve all tuples except the join result set. Third, the QM decrypts the result and sends back the result.

3.6.1.3.5 Duplication Removal

Enforcing duplication removal over a query result (encrypted tuples) at the Cloud is impossible because we use a non-deterministic encryption algorithm (AES-CBC). Therefore, we leave the execution of this operator to be accomplished at the QM before sending back the user's result. That means the QM will decrypt the encrypted tuples retrieved from the Cloud. If the query

contains the duplication elimination keyword "*distinct*," the QM will define a LinkedHashSet data structure that does not allow duplicated elements in the set. It will then add each decrypted tuple to the set. Note that the translated query to be executed by the cloud server will not have the "*distinct*" keyword. Further, the keyword "*distinct*" usually appears in *select* queries, in which case the algorithm to eliminate duplication is the *select* algorithm, and we add three more steps to execute the *distinct* operator. See algorithm 8 below.

Algorithm 8: duplication removal

```
1:  execute steps 1-9 of select algorithm
2:  if distinct keyword is present in the original query
3:      define LinkedHashSet s
4:      for each fetched tuple i
5:          decrypt values of i
6:          push i to s
7:      send s to the user.
```

3.6.1.3.6 Aggregation and Sort

To implement the aggregation and sort operators (max, min, and count) over encrypted tables in the Cloud, we consider two cases for sensitive columns, ranges, and non-range columns. In non-range columns, we can process the query locally at the QM with no need to communicate with the Cloud. In such a case, we avoid the decryption computation that results from retrieving encrypted records from the Cloud. Consequently, we will achieve faster query processing. Specifically, the QM searches the BVM after looking up the corresponding PT using our search algorithm to obtain a list of all BVs' indices that satisfy the query condition. Note that the QM might not need to do further computations such as decryption processes and will, therefore, send the query result back to the user. For the sake of clarity, consider Table 3.1, Figure 3.1, and Figure 3.2, suppose a user send the following query:

select count(name) from students where department = 'computer science';

then, the query is processed as below:

- The QM will search the PT and find that Dept is a non-range column (node's color is gray). Then it will search the BVM (Figure 3.2) and push the index of any BV having the bit representing the PD "computer science" is not zero into a list L [].
- The QM counts the number of the elements in L [] and returns the number to the user.

On the other hand, the query process is divided into two phases. The first phase is accomplished at the QM, while the second phase is handled in the Cloud. The QM executes the aggregation or sort operators over decrypted records after the QM processes the query to retrieve only candidate encrypted tuples that are related to the query. For example, the query

Select count ('name') from student where the name ='Alice';

is processed as follows:

- The QM looks up the PT and will find the name is a range column (node's color is not gray). It will then search the corresponding BVM and add to the list L [] the indices of all BVs that have the bit assigned to the PD "A-F" is one.
- The QM then looks for any encrypted record in which its index is present in L[] from the Cloud using this query syntax:

  select name from the student where index in ("elements of L [] separated by commas ").
- The QM decrypts every fetched tuple's name and increments the count value only if the decrypted name value equals 'Alice.'
- The QM returns the value of the count variable to the user.

The SUM and AVERAGE functions are processed similarly as a count, but we sum the decrypted numbers. If the operation is average, we divide the sum over the number of decrypted values that meets the query conditions. The sorting operator can be executed similarly to the aggregation

operator. However, we need to run the sort operator over decrypted data before sending the result back to the user. Algorithm 9 shows the process.

Algorithm 9: aggregate functions

```
1:  Do steps 1 to 6 of select algorithm.
2:  If all SCs are non-ranges
3:      If the operator is count
4:          Do steps 9 to 12 of select algorithm
5:          Let x = Count the number of Li []
6:          Return x
7:      If the operator is max
8:          Let m = the value of right most PD of the SC predecessor node in the PT
9:          Return m
10:     If the operator is min
11:         Let n = the value of left most PD of the SC predecessor node in the PT
12:         Return n
13: If all SCs are ranges
14:     If the operator is count
15:         Do steps 9 to 15 of select algorithm
16:         Define List Lk[]
17:         For each fetched record i
18:             Decrypt i
19:             If i meets the query condition
20:                 Add it to Lk []
21:     Return size of Lk [] to the user.
22:     If the operator is max
23:         For the right most PD of the SC
24:             Do steps 10 to 15 of select algorithm
25:             Define a variable x
26:             For each fetched record i
27:                 Decrypt i
28:                 If i > x then
29:                     x= i
30:     Return x to the user
31:     If the operator is min
32:         For the left most PD of the SC
33:             Do steps 10 to 15 of select algorithm
34:             Define a variable x
35:             For each fetched record i
```

```
36:                 Decrypt i
37:                 If i < x  then
38:                     x= i
39:     Return x to the user.
```

3.6.1.3.7 Project

In project queries, the QM does a column-based retrieval; it will select all tuples for specific column(s). Further, we do not need to perform a PT lookup in the project. However, we need to decrypt the whole set of retrieved tuples at the QM. We do not need to remove duplication of doing any filtration at the QM.

3.6.2 Bit Vectors as Column(n) (BVSAC)

In this section, we discuss the model that stores bit verctors as an additional column in the main encrypted table. Some passages, figures, tables, and algorithms presented in this section have been quoted verbatim from our published work in [60].

3.6.2.1 System Description

We designed this model to execute different relational algebraic queries over encrypted data. In addition, we divided the computation into two sides: a client-side and cloud provider (CP) side, in which we shift the majority of the computation to the CP site by rewriting the queries into ways that enable the CP to execute them over encrypted data. The architecture of this model is shown in Figure 3.6. We developed an algorithm for each query category (e.g., select, join, union, intersection, etc.) to allow the CPs to execute such query categories over encrypted tuples. We used our encoding scheme, presented in section 3.5, to create BVs. However, instead of storing the BVs locally at the QM, we migrated them to the CP. On the CP side, the encrypted table

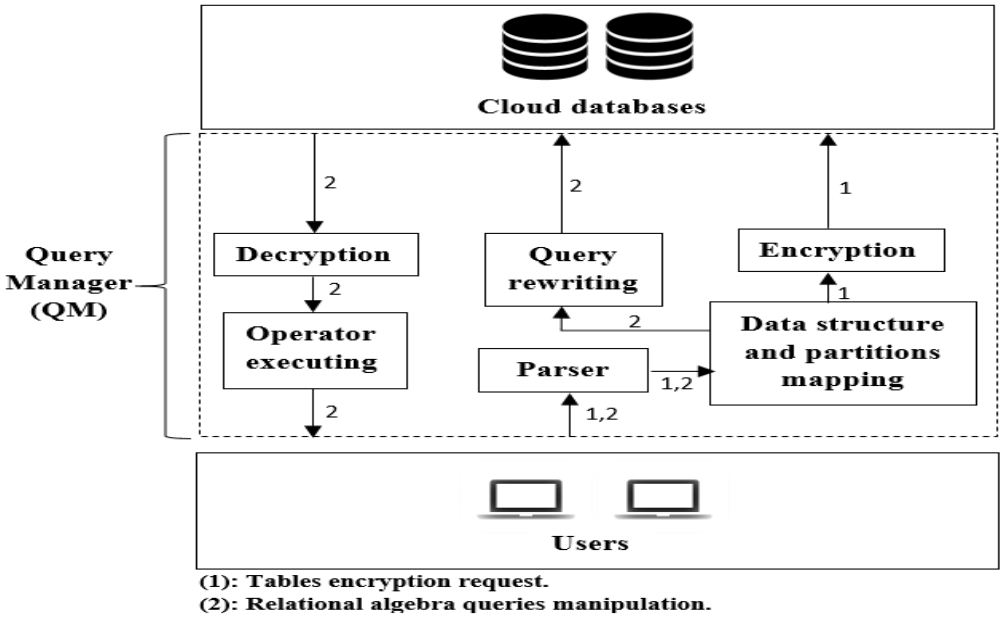


Figure 3.6 BVSAC Architecture.

contains an additional column (reference column) to store a BV for each tuple. The data in the reference column is a BIT(*n*), where *n* is the number of bits. We are not concerned about the growth of the BVs because they are in bits stored in the Cloud. Because the maximum number of bits that can be stored in an attribute of type BIT is 64 bits, we need more than one attribute to store bits. Later, when it comes to translating the query, the QM can take the bits position and find the remainder of the modulo 64. So, when the bit position is 91, then 91 mod 64 =27, which implies that the new bit is located in the second bit's attributes at position 27.

Even though it is difficult for an adversary to infer the data distribution or the possible values of a column from the BVs, we encrypt the tables' names and the headers' sensitive columns using a deterministic encryption algorithm (AES) in which the ciphertexts for each plaintext are always identical. For example, the attribute "Students-Rank" contains a limited range of values like senior, grad, junior, etc. An attacker might be able to infer possible values, although they are randomly encrypted with AES-CBC (which always produces different ciphertexts for a plaintext). Having more than one ciphertext share the same prefix is not a concern because the adversary cannot get the plaintext unless he obtains the secrete key (sk). In this case, the sk is not passed to the cloud server.

For example, we want to encrypt the table of graduate students, as shown in Table 3.2. Using the owner's sk, the QM encrypts the table name and the names of sensitive columns using AES-DET. It then creates a new table at the cloud server. Using the PT presented in Figure 3.7 and our encoding algorithm described in section 3.5, the QM encodes the tuples into bit vectors (BVs). The outsourced encrypted table is similar to Table 3.3. Note that the encrypted table has an index column in which each index number is a part of each row's initialization vector (iv)). To show how a query is processed using this system, assume a user submits the following query:

Table 3.2 Graduate students

ID	Name	Rank	Visa type	Department
110	Alice	freshman	F1	Computer science
111	Sara	senior	J1	Computer engineering
112	John	junior	None	Information system
113	Ryan	Sophomore	J2	Math

"*select* name from Graduate_students where Department = "Computer Science"

The QM handles the query as follows:

1) The QM encrypts the name of the table and the sensitive columns' headers using the sk of the table's owner and obtains the ciphertexts as below:

$CT_A = E_{AES\text{-}DET}$ ("Graduate students", sk)

$CT_B = E_{AES\text{-}DET}$ ("Name", sk)

2) The QM searches the PT to find the bits' positions, then rewrites the query as:

"*select* CT_B from CT_A where reference&32 > 0"

3) The cloud server returns only the encrypted tuples that have the bit at position 32 (2^5) = 1.

SCs	Name's first letter				Rank					Visa Type					Department					
PDs	A-F	G-L	M-R	S-Z	Fresh	Sen	Jun	Soph	Grad	F1	F2	J1	J2	None	CS	CE	IS	Bus	Math	other
BPs	524288	262144	131072	65536	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
	2^{19}	2^{18}	2^{17}	2^{16}	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

SCs: denotes sensitive columns.
PDs: denotes partitions domains.
BPs: denotes bits' positions (e.g. the 1st bit in the bit vector is the bit that is located at position 1 while the last bit in the bit vector is located at position 524288.

Figure 3.7 The PT of Table 3.2

Table 3.3 The encrypted graduate students table

$E_{AES\text{-}DET}$ (“Graduate students”, sk), encrypted version of the outsourced graduate students table. Note that the headers of the columns are encrypted using AES-$_{DET}$ where $ct_{n=}$ $E_{AES\text{-}DET}$ (“Name”, sk), $ct_{r=}$ $E_{AES\text{-}DET}$ (“Rank”, sk), $ct_{vt=}$ $E_{AES\text{-}DET}$ (“Visa type”, sk), $ct_{d=}$ $E_{AES\text{-}DET}$ (“Department”, SK).

Index	ID	ct_n	ct_r	ct_{vt}	ct_d	Reference
01	110	*&^	*_^%	*/d	^%^H	10001000010000100000
02	111	%^&	/+$	&^/	&&%$	00010100000100010000
03	112	)(#	%$/*	+-*&	)*#R	01000010000001001000
04	113	$#!	!@~K	*/f	@$%*	00100001000010000010

3.6.2.2 Condition Rewriting

To allow the cloud server (CS) to search encrypted data, the QM must rewrite the query conditions in ways that allow the CS to execute the queries over encrypted tables. This translation process is a mandatory step before executing any query in any category. The QM rewrites the queries based on the stored (PT) of the table(s) involved in the query. As in [10], there are three conditions:

1) A condition is containing a column and a value.
2) A condition is containing only columns.
3) Both 1 and 2.

In both Type 1 and Type 2, we have five possible operators {>, <, =, ≥, ≤ } while in Type 3, we have three operators {V, ⌐, Λ}. In Type 1, the QM parses the query to get the table name, attribute headers, and values. After that, the QM finds the bit’s positions that represent the PD_i under which value *v* falls. If the operation is “=,” the QM replaces values with its corresponding bit position. If the operator is “≥” or “>,” the QM locates the positions of all PDs that are less than *v*. Hence, the QM rewrites the query by replacing *v* with the positions separated by OR, removes (“>” or “≥”), and adds “=” instead. For example, we have a table T, and it has a salary column attribute partitioned into three PDs as {(1000-5000), (5001-9000), and (9001-12000)}. Let us say that the partitions are mapped to the 32, 16, and 8, respectively. Assume that the user sends a query to fetch

any tuple in which “salary > 8000.” Then, the condition of the query after translation is “reference &16=1 OR reference &8=1.” We address the operations (“≤” or “<”) in a similar way as with “>” or” ≥.” However, we take the PDs ≤ *v*.

In Type 2, for every table involved in the condition, the QM identifies the bits’ positions mapped to all PDs of the attributes A. If the operator is “ =,” the QM rewrites the query by substituting the attributes’ headers with pairs of the PDs’ separated by the OR operation (pair1 ∨ pair2 ∨ pair3 …). For example, (PD_i from attribute A, ∧ PD_i from attribute B, where PD_i=PD_j). To illustrate this, review Table 3.2 and assume that we have another table called (students info table), and both tables share the same PT as in Figure 3.8. Suppose the condition is “where GraduateStudents.name = students_info.name”. Then, let us say that:

$CT_{A=}$ $E_{AES\text{-}DET}$ (“graduate students”, sk)

$CT_{B=}$ $E_{AES\text{-}DET}$ (“students_info”, sk)

Assuming that both tables were encrypted using the same sk, the translated query condition is as follows:

“WHERE ((CT_A.Reference&524288>0 AND CT_B.Reference&262144>0) OR

(CT_A.Reference&262144>0 AND CT_B.Reference&131072>0) OR

(CT_A.Reference&131072>0 AND CT_B.Reference &65536>0) OR

(CT_A.Reference&65536>0 AND CT_B.Reference &32768>0))”

We follow the same procedures if the columns are numeric, and the operators are in {>, <, ≥, ≤}. In Type 3, each condition can be one or more of the above two types separated by AND/ OR logical operators.

3.6.2.3 Basic Operations

Insert statements, as in the previous model, enable this model to insert encrypted record(s). To perform the insertion, we follow the same steps in the BVM system. However, instead of appending the BV to the BVs matrix locally at the QM, we insert it as a value for the reference column.

Select statements execution in this model is accomplished as follows:

The QM gets the user's query, rewrites it according to the tables' PT, and sends the translated select query to the cloud server (note that the QM translates the clauses that involve only sensitive columns). Then, the cloud server executes the translated query and sends back the result (encrypted tuples for all candidate records). Now, the QM decrypts and applies the selection operation to filter unrelated data before returning the result to the user. We illustrated the process in the example in the previous sub-section. Algorithm 10 outlines the steps to be performed by the QM to execute the *select* statement.

Algorithm 10: select

```
Input < Table name, List of all columns, List of all data items di(s) >
If none of columns c_i mentioned in the query are sensitive
    Forward the query to CSP
      Decrypt encrypted data
      Send result back to the user
If all column(s) c_i mentioned in the query are sensitive
    For each data item di / value v being searched for under column c_i
        Find the bit's position that mapped to the partition domain PD that di falls under
        in the table's PT.
Rewrite the query and substitute values by bits' positions.
Send the translated query to CSP to retrieve candidate tuples (CTs)
Decrypt CTs then implement select again to filter out unrelated records.
Send result back to the user
```

To execute ***update*** statements, we use the *select* algorithm that is presented in Algorithm 10 to fetch candidate encrypted records from the Cloud. Then, if the new updated value(s) belong to a different partition domain, the QM finds the exact record(s) and sends an updated query to the Cloud to update the encrypted record(s) and the bits.

In ***delete*** statements, the QM uses Algorithm 10 to obtain the encrypted records from the Cloud. It then identifies the exact record(s) after decrypting the candidate records. Immediately, a delete query is issued by the QM to the Cloud to delete the encrypted records.

The process for the ***alter*** query is similar to the first model. We use the partitioning tree (PT) to determine whether or not a column is sensitive. If the column is sensitive, the QM will identify the node and its children nodes. In other words, the QM finds the node numbers that represent bits positions of the encrypted column. Then, the QM issues an alter query to delete the bit positions from the reference column and the encrypted table. The QM then removes the node of this column and its children nodes from the PT. If a sensitive column is added, the process is reversed. The QM adds a node to the second level and adds its partitions as children nodes in the third level. It will then add *n* bits where *n* equals the number of new children nodes added to the reference column.

3.6.2.4 Relational Algebra Operators

3.6.2.4.1 Join

To enable the proposed model to support the join operator, we must consider different cases for the join condition: 1) the join condition involves only non-sensitive columns, 2) the join condition involves only sensitive columns that have limited distinct values, and 3) the join condition involves sensitive columns that may have too many distinct values. The first case is straightforward because

the QM is required only to forward the query to the cloud database server (CDBS). Then, it decrypts the result and removes the duplication.

In the second case, the QM uses the data structure DSs of the tables in the *join* condition to match the PDs of the tables. For example, we have two tables, T_1 and T_2. In Table T_1, each PD_j under a sensitive column SC_i is joined with each PD_j under a sensitive column SC_i in T_2 if and only if the PD_j from T_1 equals the PD_j from T_2. Note that the QM does this only for columns that are involved in the join condition.

The third case is based on the range of PDs. If PD_i has at least one common element, the QM joins the PDs from both tables and ensures that each PD_i of the SC from Table T_1 is joined with each PD_i of the SC from Table T_2. For example, assume the salary PDs in Tables T_1 are [PD_1 (10,000–15,000), PD_2 (15,001–20,000), PD_3 (20,001–25,000), and PD_4 (25,001–30,000)], and in T_2 are [PD_1 (10,000–20,000), and PD_2 (20,001–30,000)]. The QM joins PD_1 from Table T_2 with PD_1, PD_2 from Table T_1 because PD_1 of Table T_2 contains elements from both PD_1 and PD_2 of Table T_1, and so on. The QM then rewrites the query and sends it to the Cloud. The Cloud returns the join result to the QM, which decrypts only the columns involved in the join condition and checks whether the plaintexts satisfy the join condition. The QM decrypts the whole tuple only if the two values satisfy the join condition. Otherwise, the QM skips to the next tuple. We do so to eliminate unnecessary decryption processes. The QM eliminates duplicates after decrypting the whole result. Algorithm 11 shows the steps of the ***join*** operation.

Algorithm 11: join

1: Input < Tables names, List of all columns, List of all data items *di*(s) >
2: **If** none of columns mentioned in the join condition are sensitive
3: **Forward** the query to CDBS
4: **Decrypt** encrypted data and **Remove** duplication
Send result back to the user
5: **If** all columns mentioned in the query are sensitive

6: **If** the SCs have limited distinct values
7: **Join** the bit's position of each partition domain PD_i of a SC_k from table T_m with the equivalent PD_i of a SC_x from table T_n
8: **If** the SCs have range values
9: **Join** the bit's position of each partition domain PD_i of a SC_k from table T_m with each PD_i of a SC_x from table T_n if it has at least 1 common value.
10: **Rewrite** the query and substitute columns' names by bits' positions separated by OR operation.
11: **Send** the translated query to CDBS to retrieve candidate tuples (CTs)
12: **Decrypt** CTs and **remove** duplication
13: **Send** result back to the user

3.6.2.4.2 Union

To process union queries, we follow the same processes presented in the previous system for BVM.

3.6.2.4.3 Intersection

The intersection operation uses the same two conditions as the union operation. However, instead of retrieving all encrypted tuples of both tables, we extract only the tuples that are common to both tables. We must simulate the intersection operation because it is impossible to apply it over encrypted tuples. To do that, in the CDBS, we use the inner join between the two tables. The join condition is based on PDs that are not ranges because range-based PDs return more candidate tuples. The CDBS then executes a query to choose the tuples from Table A that exist in the join result. After that, the CDBS returns the joining result to the QM, which eliminates duplicates after the decryption process. For example, Table A and Table B have attributes (Name, Visa type, Rank). Then we perform the inner join operation in which the condition is visa type = visa type AND Rank= Rank). Note that, in the condition, we substitute the columns' names with the positions of the corresponding bits to enable the CDBS to execute the operation. This substitution filters out too many uncommon tuples in the Cloud, resulting in fewer decryption operations by the QM. Algorithm 8 illustrates that processes.

Algorithm 12: intersection

1: Input < Tables names>
2: **Find** all common sensitive columns that are not ranges between the tables
3: **For** each table T_i
4: **Find** bits' positions of PDs
5: **Initiates** an inner join query where the joining condition is based on the equivalence of the bits obtained from step 4 for each table.
6: **Send** the query to CDBS
7: **Decrypt** the result of join.
8: **Implement** *distinct* over decrypted data to **Remove** duplication
9: **Send** result back to the user

3.6.2.4.4 Difference

To invoke the *difference* operator, we follow the same procedures as in the BVM system.

3.6.2.4.5 Duplication Removal

To remove the duplication, we follow the same steps as the BVM system.

3.6.2.4.6 Aggregate and Sort

In this operation, we have two cases. First, the condition clause is based on column(s) in which its partitions domain PD values are not ranges. In this case, the CDBS efficiently implements the aggregation operation over encrypted data and sends back the result to the QM, which needs only to decrypt the result and send it back to the user. Note that the QM may not implement the aggregation operator again over the decrypted data because the candidate tuples retrieved from the CDBS are the exact query result.

For example, consider Table 3.2, Table 3.3, and Figure 3.7**.** A query select count(*) from GraduateStudents where department = "Computer science"; is translated to "select count(*) from $E_{AES\text{-}DET}$("Graduate students", SK) where reference &32 >0". It returns only the number of tuples that satisfy the condition because the condition is based on the partition domain "computer

science" that is not a range of values. The second case is the condition clause, which is based on column(s) in which its partitions domains' values are ranges. In addition to the operations in the first case, the QM implements the aggregation operator again over decrypted candidate tuples. The reason is that candidate tuples have tuples unrelated to the query because the PDs are ranges. In the normal operation, the whole calculation is done at the QM. However, we minimize the range of encrypted records retrieved from the CDBS by only those that satisfy the condition clause of the average query. Algorithm 13 illustrates these steps.

Algorithm 13: Aggregation operator

```
1:  Input < Tables names, List of all columns, List of all data items di(s) >
2:  If none of columns mentioned in the aggregation query are sensitive
3:      Forward the query to CDBS
4:          Decrypt encrypted data and Remove duplication
            Send result back to the user
5:  If all columns mentioned in the query are sensitive
6:      Reconstruct the query by substituting data items/ values by the bits' positions that
        mapped to the PDs that contain data items.
7:      Send the query to CDBS
8:          If the SCs in the aggregation query are not ranges
9:              Decrypt the result
10:             Send result back to the user
11:         If the SCs in the aggregation query are ranges
12:             Decrypt the result
13:             Execute aggregation operator again over the result
14:             Send result back to the user
```

To enforce ***sort*** operators over encrypted data, we can first make the cloud server filters out unrelated records according to the PDs of the PT. Then the QM decrypts the returned sorted result from the Cloud and executes the sort operator over them again. The sorting computation performed at the QM before returning the final result to the user and after retrieving candidate tuples is significantly small because the CDBS sends back only those candidate records that fall under a specific partition domain(s). However, the CDBS will return a group of records that are

not sorted except by the partition domain. For example, to retrieve tuples that have an income ranging from 50k to 60k, the cloud server returns all encrypted records that have income within this range unsorted. Therefore, the QM sorts them after the decryption process.

3.6.2.4.7 Project

To execute the project operator, we follow the steps for the BVM system.

3.6.3 Bit Vectors as an Independent Table (BVSIT)

In this section, we discuss the model that uses bit verctors as a separate table. Some passages, figures, tables, and algorithms presented in subsections (3.6.3.1 and 3.6.3.2) have been quoted verbatim from our published work in [61].

3.6.3.1 Model Description

In this prototype, we store the bit vectors (BVs) in an independent table in the cloud server. We add an index column to both the encrypted table and the independent table (i.e., the bits table), in which the index of each record in the encrypted table equals the index of the corresponding record in the bits table. To ensure that the bits table does not leak any data from bits representations, we encrypt the indices in the bits table to ensure that an attacker cannot determine what record in the bits table is intended for the encrypted record *x* in the encrypted table. Furthermore, we perform record-based shuffling in the bits table so that the records will not have the same order as the main encrypted table. Figure 3.6 shows the architecture of the proposed model.

For ***security*** in the BVSIT, data in all sensitive columns are encrypted with a symmetric encryption algorithm (AES-CBC). This version is a randomized encryption algorithm in which the plaintexts have different ciphertexts. By doing this, we enforce maximum security and make this model resistant to various attacks such as chosen plaintext attacks and inference attacks. In addition, having the bits table stored in the Cloud without index encryption could lead to data leakage and enable the attacker to identify which record in the bits table represents a specific encrypted record in the main encrypted table(i.e., record linkage attack). The attacker could do this by matching the indices in both tables. To avoid this scenario, we encrypt the index column in the bits table using AES-DET and perform record-based shuffling in the bits table. This encryption

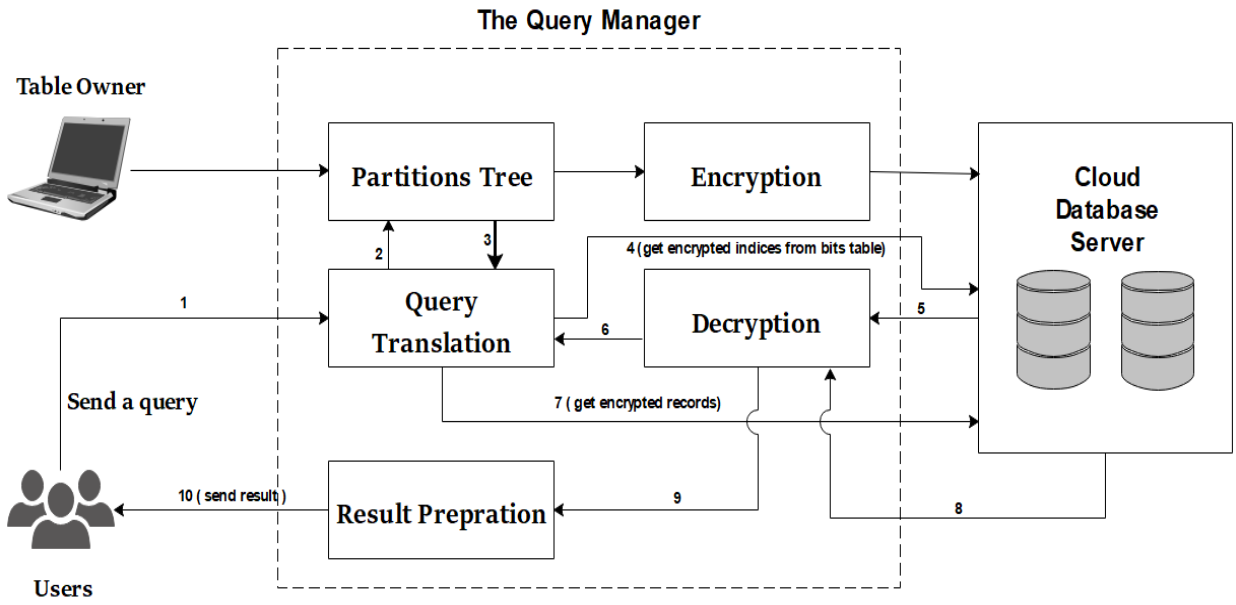


Figure 3.9 BVSIT Architecture.

ensures that the attacker cannot infer which record in the bits table represents the encrypted record *x*.

3.6.3.2 Basic Operations

Before explaining how to support basic SQL operations in this model, consider the employees' table presented in Table 3.6. Given the PT (shown in Figure 3.10), the bits table is shown in Table 3.4, and the encrypted table is shown in Table 3.5.

In ***select*** statements, we use the PT to rewrite the select queries appropriately so that the cloud server can execute SQL queries over the encrypted data with no need to modify the functionality of the database system. Because the PT is stored locally in the QM, the QM will conduct a quick lookup to determine whether the attributes in the select conditions are sensitive. Then, the QM will issue two queries. The first query retrieves the encrypted indices of all candidate tuples for the select query from the bits table. The second query retrieves the encrypted tuples

Table 3.4 Bits table of Table 3.5

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
CT(3)	0	1	0	1	0	0	1	0	0	1	0	0	0
CT(4)	0	1	0	1	0	1	0	0	0	0	0	1	0
CT(1)	1	0	0	1	0	0	1	0	0	0	1	0	0
CT(2)	0	0	1	0	1	0	0	1	0	0	0	1	0

Table 3.5 Encrypted employees table

Index	ID	Name	SSN	Rank	Salary
1	^%#	)@(	$@^#	@^#S	FS^@
2	(#*	^#*	)@H2	+_@	%@*&
3	(#(	(#&	$@%	@^#@	@%TW
4	)^@	)^#H	!@M	*@^#	+_)#FQ

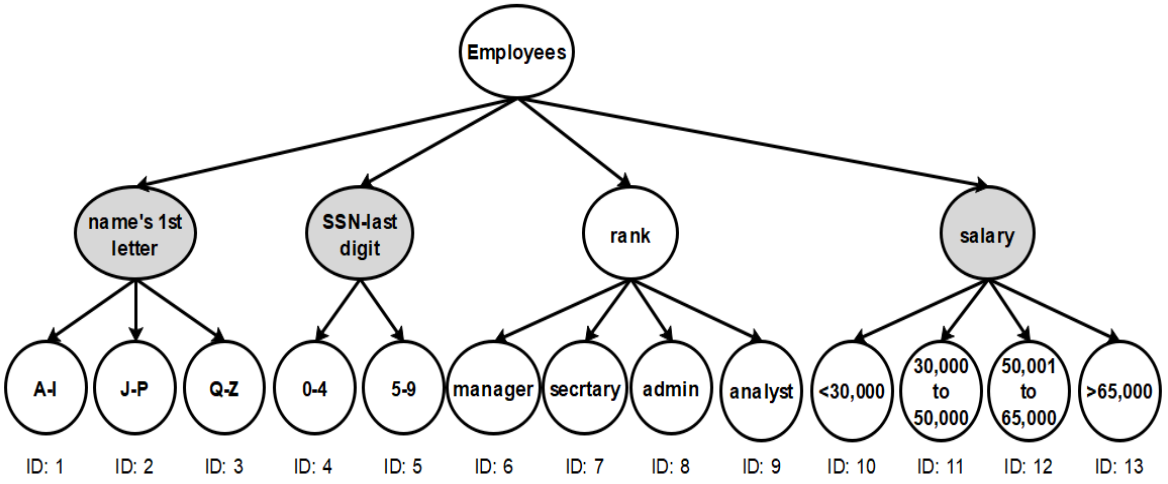


Figure 3.10 The Partitioning Tree (PT) of the employees table.

Table 3.6 The employees' table

ID	Name	SSN	Rank	Salary
01	Alice	12701	Secretary	30,000
02	Ryan	25678	Admin	60,000
03	Mark	46932	Secretary	29,000
04	John	42213	Manager	55,000

(whose indices were decrypted after being retrieved from the previous query) from the encrypted table. For clarity, consider Table 3.4, Table 3.5, and Figure 3.10 for the following query:

select name from employees where rank = “manager” or salary = “60,000”.

Then, the query is processed as follows:

1) The QM uses the PT of the employee table to locate the nodes representing the partitions under which the values fall (i.e., “manager” and “60,000”).
2) According to the PT, “manager” is mapped to the node of ID = 6, and “60,000” falls under the values represented by the node ID = 12.
3) The query is translated to another query to select the index from the bits table, where “6” = 1 or “12” =1; this enables the CSP to return any index that has a value of 1 in either column 6 or column 12.
4) The QM will retrieve the encrypted indices (CT (4) and CT (2)) and decrypt them to obtain the plain indices (2,4).
5) A new query is issued to retrieve the encrypted tuples from the main encrypted table. The query is:

select name from encrypted_employees where index in (2,4).

6) The QM will decrypt the data and ensure that rank = “manager” and salary = ‘60,000’ before returning the result to the user.

The same process is followed if the condition has either more or fewer clauses. As in step three, the operation “AND” and “OR” remain unchanged in the query condition.

To make this model sufficiently flexible and practical for all types of users, ***insert*** is enabled to support insert queries. Before the encryption process begins, the QM uses Algorithm 1 to encode the inserted record(s) to the bit vector(s). The insert query leads to the sending of two insert queries to the Cloud, one to insert the encrypted record into the encrypted table and the other to insert the bits in the bit vector into the independent bits table. To do the shuffling, the system will occasionally and automatically retrieve the entire bits table, re-encrypt the indices, and perform a column-based, then record-based, shuffling before outsourcing the table once again. The

reason to re-encrypt the indices is that if they are not re-encrypted, the cloud provider can easily find the recently inserted record(s).

In ***update*** queries, the QM determines if the new value is intended for a sensitive column by conducting a lookup of the PT. The select rewriting method is used to accomplish this step so that only the candidate-encrypted tuples of the update query are retrieved. Following this, the QM finds the exact record(s) after the decryption process and issues two queries to update both tables in the Cloud (i.e., the encrypted table and the bits table). For example, we want to update Mark's salary (the third record in Table I) to 50,000. The QM uses a select operation to retrieve Mark's record and finds that the new salary falls under the node "12" that is mapped to column "12" in the bits table. The QM will then issue two queries to update the new salary (i.e., the ciphertext of the new salary value) in the encrypted table and update the value of column "10" to 0 and column "12" to 1.

For the ***delete*** query, the QM uses the select method described earlier to retrieve the candidate-encrypted records from the Cloud, decrypt them, and then find the required record(s). The query is processed as follows: 1) The QM sends a select query to obtain the encrypted indices of the candidate records from the bits table that is outsourced to the Cloud. 2) The QM decrypts the encrypted indices and issues another query to retrieve the encrypted tuples from the main encrypted table; the query will be a select query based on the indices obtained from the first query. 3) The QM will decrypt the encrypted tuples and filter the tuples that are unrelated to the query. 4) The index of any record related to the delete query will be kept by the QM, which will then issue two deletion queries, one to delete the record from the encrypted table and the other to delete the corresponding record from the bits table.

The ***alter*** query is applied in the following process. When deleting a column, we first search the second level of the corresponding PT to determine whether the column is sensitive (this step occurs locally at the QM). In essence, the query is forwarded to the Cloud if the column is non-sensitive. Otherwise, the QM finds the IDs of all children nodes (i.e., sub-nodes at level 3) of the node that represents the sensitive column. Then, the node and its children nodes are deleted. Following this, the QM issues a query to delete all columns that represent the deleted partitions from the bits table in the Cloud. It then sends another query to delete the encrypted column from the Cloud. Note that we do not focus on updating the IDs of the level-three nodes at the PT. We can keep their IDs, and we eliminate any unnecessary computations that could result from updating them. To add a column, we follow the same deleting procedure but in the opposite way. We add a node at the second level of the PT and then add sub-nodes that represent the partition domains of the column. The *n* columns are then added to the bits table at the Cloud where *n* = the number of sub-nodes recently added to level three of the PT. We also add a column to the encrypted table to store the encrypted data of the newly added column.

3.6.3.3 Relational Algebra Operations

3.6.3.3.1 Join

As with the BVM system, implementing join over encrypted data without decrypting data or modifying the functionality of the DBMS is a difficult task, particularly in our case where we use a randomized encryption algorithm to encrypt sensitive data. However, to enable the cloud provider to execute join without decrypting the data, we developed a strategy to accomplish this while leaving most of the computation to the Cloud. In the following section, we explain how we process the join operator.

The QM gets the join query and parses it to obtain the name of the tables and the sensitive columns. Note that we have the same cases as the previous model. If the join condition involves only non-sensitive columns, the query is simply forwarded to the Cloud without any modification. Otherwise, the QM must perform a quick search to the partitioning trees (PTs) of the tables involved in the join query to find which partitions of the joining columns from both tables can be joined. To accomplish that, the QM rewrites the join query in a way to enable the CP to join any partition p_i of the SC from Table A with any partition p_j of the SC from Table B if and only if the p_i equals p_j or has common elements with p_j. We do not want to miss any record. However, we want to eliminate any unrelated records from both tables. After that, the QM rewrites the query accordingly to fetch the encrypted indices from the outsourced bits tables of the joined tables. The QM then issues *n* queries where *n* is the total number of partitions in the joining SCs from both tables. So, if the joining SC in Table A has three partitions and joining SC in Table B has five partitions, the QM issues eight queries to the Cloud to retrieve the encrypted indices from the corresponding bits tables. After that, the QM decrypts the encrypted indices retrieved by each query and adds them to lists. Based on the lists obtained from the previous step, the final join query is then formed to fetch the candidate encrypted tuples from the main encrypted tables and send it to the cloud server.

For example, we want to joint two tables, A and B, by Visa Type. For simplicity, we assume the two tables have the same PT and that the partitions of the Visa Type attribute are (F1, F2, J1, J2). In the bits table of Table A, named BOA (bits of A), the columns that represent Visa Type partitions are 5, 6, 7, and 8. In the bits table of Table B (named BOB), the columns are 2, 3, 4, and 5. To execute the join query, the following queries are sent to the Cloud to retrieve the encrypted indices from bits tables:

Select indx AS A from BOA where ‘5’ =1; *Select* indx AS B from BOB where ‘2’ =1.

Select indx AS C from BOA where ‘6’ =1; *Select* indx AS D from BOB where ‘3’ =1.

Select indx AS E from BOA where ‘7’ =1; *Select* indx AS F from BOB where ‘4’ =1.

Select indx AS C from BOA where ‘8’ =1; *Select* indx AS D from BOB where ‘5’ =1.

Note that we avoid the joining query (e.g., select BOA.indx AS A, BOB.indx AS B from BOA, and BOB where BOA.5 = 1 & BOB.2= 1) because it will slow the process. Our goal is to speed up the query processing time. However, to minimize the impact of the communication delay, we can send all the queries together as one string (i.e., as one prepared statement) to the Cloud. Therefore, the best way to achieve our goal (fast query processing) is by using the queries presented above. After completing this step, the QM defines eight lists, each of which contains the decrypted indices from a specific query (i.e., list 1 will have the decrypted indices retrieved from A, list 2 will include indices from B, and so on). The QM then rewrites the query in the following format:

SELECT A.Name, B.name from (A inner join B on (A.indx in (elements of list 1) and B.indx in (element of list 2) OR A.indx in (elements of list 3) and B.indx in (element of list 4) OR A.indx in (elements of list 5) and B.indx in (element of list 6) OR A.indx in (elements of list 7) and B.indx in (element of list 8));

After the join is completed in the Cloud, the QM decrypts the result (only decrypting the joining columns and enforcing the condition, if satisfied; then proceeding to decrypt the remaining fields, otherwise skipping to the next tuple). The QM then removes any duplication before sending the result back to the user (using the same process followed in the previous two models).

3.6.3.3.2 Union

To implement the union operation between two or more tables, we need to fetch all encrypted tuples from all the unionid encrypted tables, then decrypt them and apply the *distinct* operator to remove the duplication. Note that we assume both tables were encrypted using the same secret key (sk). We follow the same processes as in the previous model (BVM) using an algorithm similar to that in the BVM model.

3.6.3.3.3 Intersection

The intersection operation is similar to the intersection operation in the BVM model. However, we intersect the bits tables before proceeding to intersect the encrypted tables in the Cloud. We use the join operation to simulate the intersection process and filter out uncommon tuples from the tables involved in the intersection operation. We follow the same procedures used for the intersection operation, as in the previous model.

3.6.3.3.4 Duplication Elimination, Aggregation Functions, and Project

These operators are treated similarly to those in the BVM model.

4 Experiments and Evaluation

In the previous chapter, we explained in detail the proposed models, and for each model, we presented the developed algorithms to execute different relational algebra operators. As stated earlier, this dissertation aims to implement, evaluate, and compare the performance of the proposed prototypes for different types of statements using various metrics (e.g., the execution delay of queries, space requirements at both the QM and the cloud server, and the percentage of the computation overhead at the QM versus at the cloud server).

Furthermore, we compare our proposed prototypes to three systems: CryptDB [2], the one block-based technique (OBT; i.e., each row is encrypted as one block) [10], and the column-based fragmentation technique (CBF; i.e., each sensitive column is migrated to a cloud; read [9] for more details). In the discussion subsection, we make a comprehensive comparison between the proposed systems and the competing systems used in the evaluation. In addition, we recommend which model is suitable for single users and which model is ideal for multi-user systems after investigating the results of all the models.

4.1 Experimental Setup

We used a PC with 6GB of RAM, 1TB HDD, and a Core i5 processor with 2.8 GHz to conduct all the experiments for all the systems (BVM, BVSAC, BVSIT, OBT, CBF, and CryptDB). To implement the functions of the QM in the proposed models, we used Java to simulate each task as a java class or method. MySQL server was used on the user's machine and we used Java Database Connectivity (JDBC) as a connector from Java to the MySQL engine. All the experiments were performed on the local machine; therefore, the communication delay variable was removed from all the reported delays.

We implemented the OBT and CBF because their implementations are not available online unlike CryptDB, which is available for public use on GitHub [62]. While implementing the CBF, we adopted the implementation in [63] to encrypt numerical values in a way that preserves the order (OPE) and in [64] to support the additive homomorphic property. In our models, we used the randomized version of the AES-CBC to encrypt sensitive data. In our systems, each tuple's data are transmitted to the encryptor class as soon as the encoding step has been accomplished. The encryptor and decryptor classes call numerous cryptographic packages, including the "javax.crypto" package offering the classes and interfaces for crypto tasks; more information can be found in [65], and the table owner's secret key (SK) and the pre-generated initialization vectors (IVs) can be used to encrypt or decrypt each tuple (each tuple needs a unique IV as explained in Chapter 3). The SK is 256 bits, and each IV is 128 bits (the IV size equals the block size in the AES). To store the bit vectors (BVs), we stored them locally at the QM in the first model, and we wrote them in a text file for future use (in the future, the QM just reads the BVs set from the file.txt and loads them to the data structure). In the other two models, we migrated the set of BVs to the cloud.

4.2 Datasets and Partitioning Tree

We randomly generated four tables. We defined a list of values for each attribute and let a java program constructs tuples by randomly picking values from the lists. The sizes of the tables (i.e., the number of records) were 10k, 20k, 50k, and 100k records. We had 24 attributes in total for all tables, and we considered all of them, except ID attributes, as sensitive attributes. In our study, although we could use the proposed models with small tables, we focused on the large tables since it is easier to test the penalties introduced by each scheme. Table 4.1 presents the structure of the main tables. For each table, we created a table in the cloud according to the created algorithm for

each model. We built a partitioning tree (PT) for all the tables based on Table 4.2. We generated the tables so that they were fairly distributed to the PT. For example, for the attribute (Name), not all the records were mapped to the first partition (node #1) under the name predecessor; instead, approximately 33% of the records were mapped to the first node, 33% assigned to the second node, and 34% assigned to the third node. We considered the same technique for the rest of the attribute partitions.

Table 4.1 The structure of the original (plain) students' table

Name of the Attribute	Datatype	Storage Required (bytes)
ID	int	4
Name	varchar	20
SSN	int	4
Visa_Type	varchar	6
Salary	int	4
Department	varchar	20

Table 4.2 The sensitive attributes and the number of partitions for students table

Attribute	Sensitive?	Number of Partitions
ID	No	0
Name	Yes	3
SSN	Yes	2
Visa_Type	Yes	4
Salary	Yes	5
Department	Yes	6
Total number of partitions		20

4.3 Evaluation

The evaluation consisted of the following:

1) Testing and evaluating basic database operations (create, select, insert, update, and delete statements) execution cost
2) Testing and evaluating aggregation operations (sum, average, count, max, and min) execution cost
3) Testing and evaluating joining and setting operations (join, union, and intersection) execution cost
4) Identifying space requirements for each system

For each part, we considered different factors that play a role in the efficiency, such as the number of clauses in the query conditions, what the logic operation (AND/OR) is in the condition, and what encrypted attributes to retrieve for the tuples. In the discussion, we explore the factors that make the proposed models more efficient and what makes them inefficient. In addition, we discuss the impact of the PT size on the efficiency of the proposed models.

4.3.1 Execution Delay Comparison

4.3.1.1 Original Database Encryption and Insert Statements

In the proposed models, the original database encryption step involves parsing records, generating indices, building BVs, encrypting sensitive data, and inserting the encrypted data into the encrypted table in the cloud server. In Table 4.3, we compare the time taken by each system to encrypt each table. As seen in Table 4.3 and Figure 4.1, the BVSAC experienced the least encryption and insertion delays among all systems because the QM does not need to store nor manage the BVs locally at the QM as in the BVM, which is why we see that the delay in the BVM is higher than the BVSAC. The second fastest model is the OBT, which encrypts every tuple as one block and adds M columns, where M is the number of sensitive columns, to store the integer

values of the identifiers. On the other hand, the CBF experienced the highest delay since the encryption process required *N* insertion (*N* = number of columns + 1) into *N* different tables in different cloud servers. The second slowest model is CryptDB in both the creation and insertion processes due to heavy computation results from the onion layer encryption. In summary, the proposed models are faster than the CryptDB and CBF models in both the creation and insertion processes.

Table 4.3 The delay of the original database encryption comparison among all systems in minutes

N.R	BVM	BVSAC	BVSIT	CryptDB	CBF	OBT
10k	12	11	25	44	55	13
20k	26	23	48	66	112	24
50k	65	54	115	158	291	55
100k	147	112	223	308	578	113

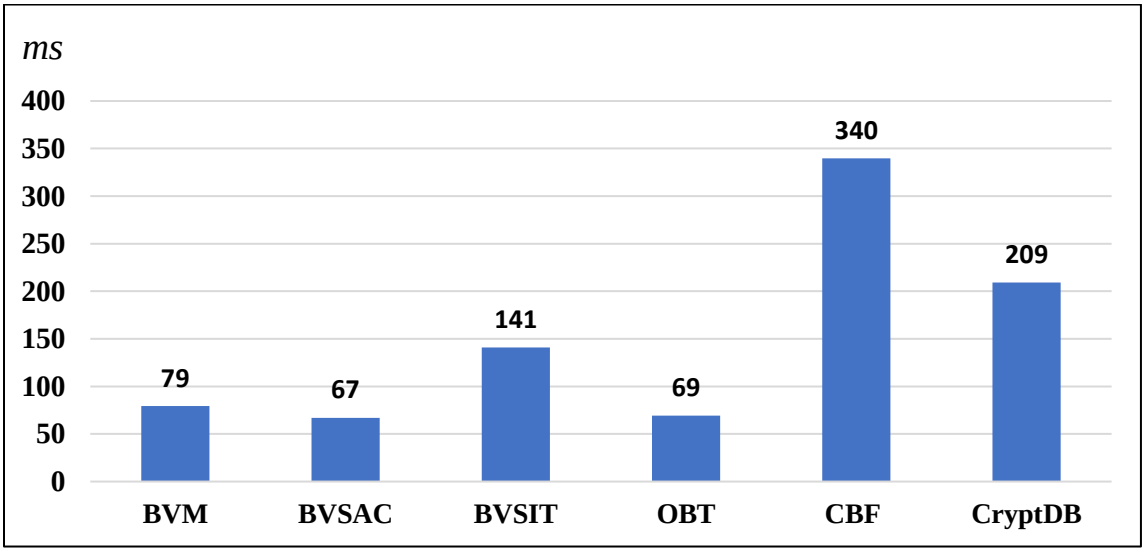


Figure 4.1 The delay comparison of insert statements for all systems, in milliseconds

4.3.1.2 Basic Statements

In this section, we examine the delay in executing different basic statements (*select, update, and delete*) for all systems. Furthermore, we focus more on the *select* statements because they are the main operations in most database applications. In addition, most state-of-the-art research focuses on *select* for the same reason.

4.3.1.2.1 Experiment 1

In this experiment, we calculated the average percentage of fetched encrypted tuples from the encrypted tables for the proposed models. We also studied how the number of clauses in the query condition can contribute to narrowing the range of the fetched set. Figure 4.2 illustrates how our models dramatically drop the average retrieved encrypted candidate records for select statements to about 31% when only one clause is present in the query condition, while all the models fetched the least percentage when the condition clause had three clauses. On the other hand, without using any of the proposed prototypes to manage the randomized encrypted database (i.e., no indexing), we must retrieve the entire outsourced encrypted table.

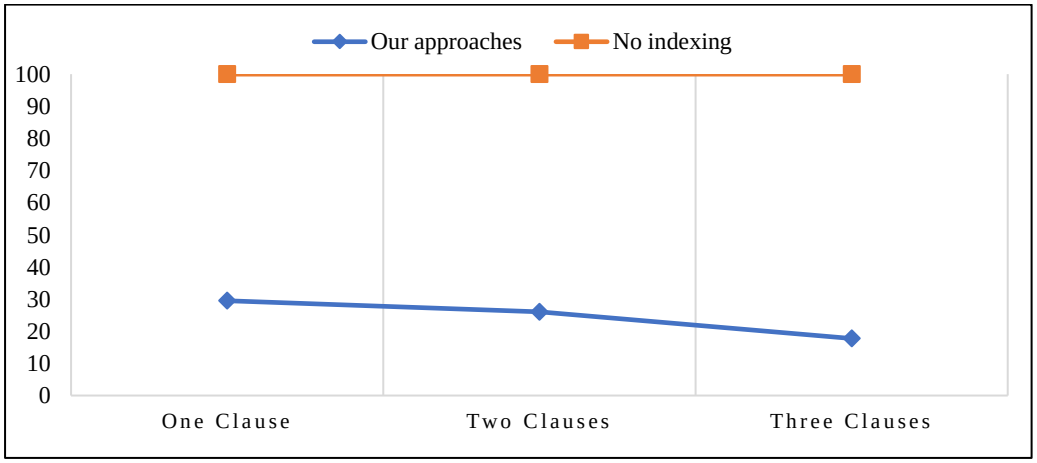


Figure 4.2 The percentage of retrieved encrypted tuples for all proposed models

4.3.1.2.2 Experiment 2

As mentioned in Chapter 3, we aimed to reduce the crypto computation at the QM when possible to avoid computation overhead, which speeds up the query execution time. To accomplish this, we only decrypted the column values that were present in the query condition before proceeding to decrypt the encrypted values of the whole record. For the sake of clarity, we have, for example, the following *select* statement:

select * from students WHERE name = 'Alice'

Using any of our proposed models, as depicted in Figure 4.3**,** we only fetch less than 35% of the entire set of the outsourced encrypted records. This depends on how the name column was partitioned beforehand; in our case, we had three partitions, and the more partitions that are defined, the lower percentage we obtain and vice versa. However, we do not immediately decrypt all the encrypted data of the fetched set. Instead, we decrypt only the name values to determine if the name is equal to "Alice" (i.e., we start by decrypting the column mentioned in the query condition). If the name equals "Alice," then we decrypt the entire record; otherwise, we skip to the next one and so on. In Figure 4.3, we can see that from the whole fetched encrypted set, the average of the entirely decrypted rows was less than 24% of the fetched rows. Accordingly, our models are efficient since we are not only narrowing the range of retrieved outsourced data, but we are also eliminating the unnecessary computation (crypto computation) whenever possible.

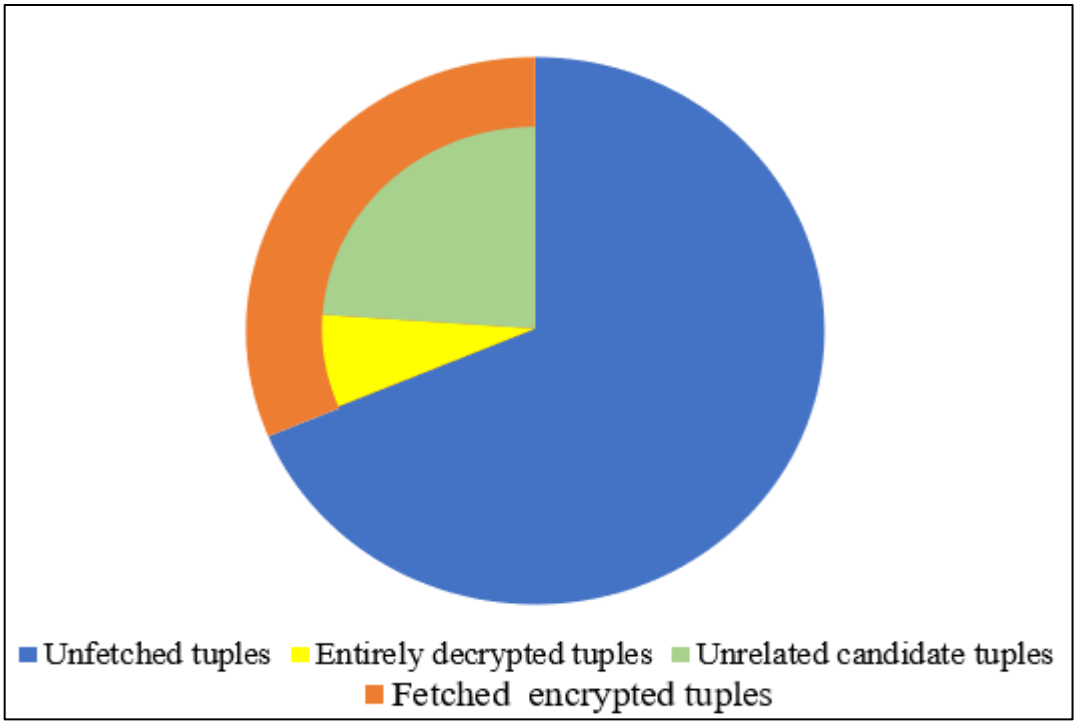


Figure 4.3 Average percent of entirely decrypted tuples

4.3.1.2.3 Experiment 3

4.3.1.2.3.1 *Select* (*) Latency

In this study, we focused on *select* statements since they are the widely used statements in the database systems. As explained in the previous chapter, the *select* statement is also part of the most proposed relational algebra algorithms (e.g., delete and update). Therefore, we deeply examined, tested, and evaluated the *select* statements with a different number of clauses and both cases of select single column values, such as, *select* name and select all (i.e., *select* *). Furthermore, as a part of this experiment, we compared the performance of the proposed models against the CryptDB, OBT, and CBF systems.

Table 4.4 presents the total runtime for all systems when executing *select* statements to retrieve single column values. The runtime we measured was the time from query parsing until the

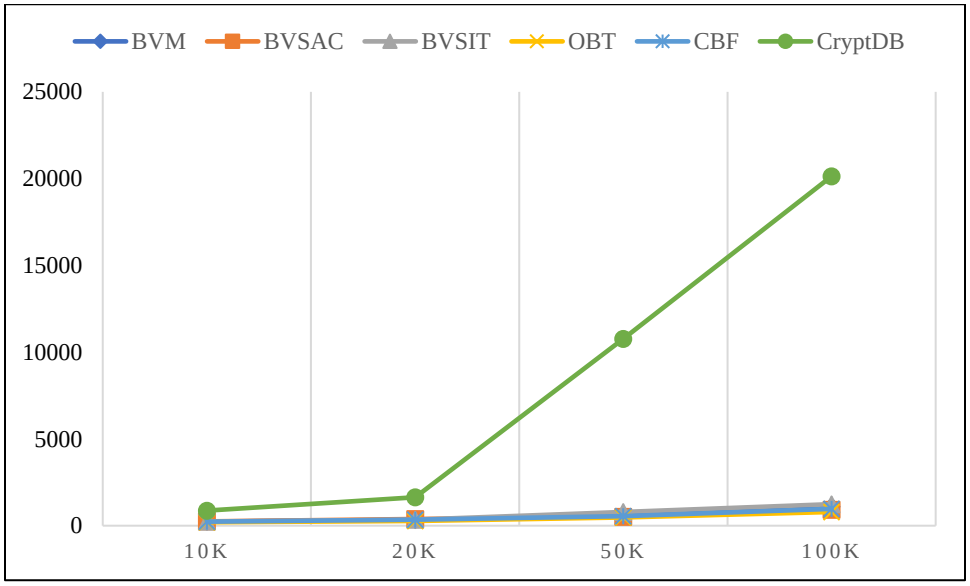
final query result was formed in milliseconds (*ms*). In the first case, we measured the average runtime when the condition clause of the queries features only one clause. The delay is the average delay of executing a *select* statement on each sensitive column. Furthermore, we tested *select* * statements when the condition had two and three clauses. Figure 4.4, Figure 4.5, and Figure 4.6 present the total execution time of *select* * statements for each model. As seen in the figures, the BVSAC averaged the shortest runtime of all the proposed models for all cases (i.e., single clause and multiple clauses) when the database sizes were 50k records or more. This result means that the BVSAC performs better, in term of execution time, with larger databases and has a higher start-up time than the other proposed models, which is why it is slower than some other systems for processing databases with 20k and 10k. Another factor that speeds up query processing is that we benefit from the bitwise operations, provided by MySQL, in the cloud because we store the bits as an independent column along with each encrypted record. Furthermore, we eliminated the manipulation of the indices at the QM in contrast to the BVM and BVSIT. For smaller databases (10k or less), the BVSIT experienced the fastest execution time of the proposed systems because the bitwise operations were performed in a column-based manner, allowing for rapid searching over the bits table; however, when databases grew, the performance of the BVSIT downgraded as the amount of the decrypted indices at the QM grew (i.e., the indices fetched from the bit table). The BVM model is the second fastest for databases with more than 50k rows, and the main factor that affects the performance of the BVM is the time for loading the BVs from the hard drive to the main memory and then searching them. Forming the final lists of the candidate record indices is another factor that affects the delay in this model.

In terms of comparing the proposed prototypes with the other approaches, as seen in Table 4.4, Figure 4.4, Figure 4.5, and Figure 4.6, the proposed models were faster than all the competing

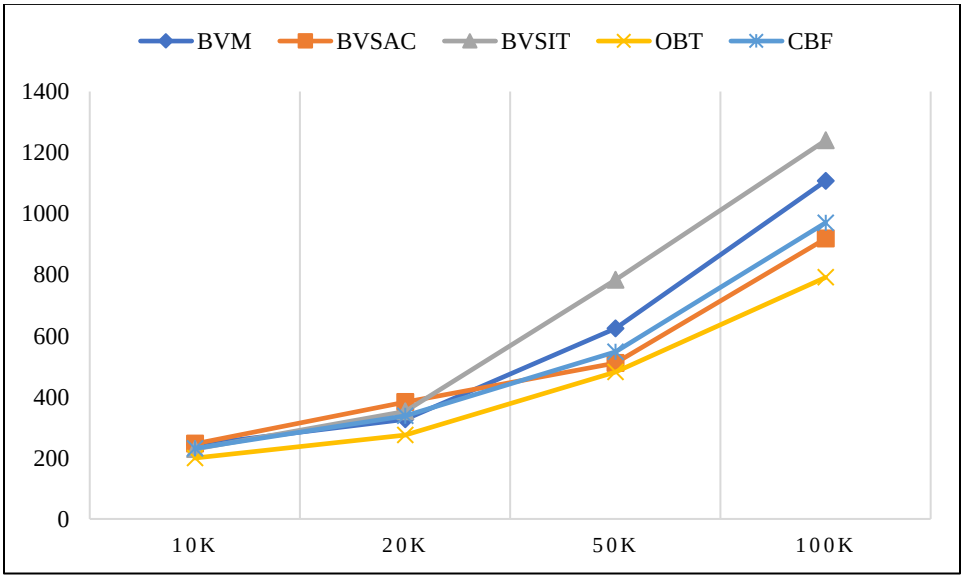
systems except for the OBT for *select* statements with two or three clauses. The OBT experienced the least delay since the amount of decrypted data was less than in our approaches (all values are stored as one block leading to fewer bytes to decrypt for each row). Cell-based encryption produces longer ciphertexts (i.e., the blocks less than 16 bytes will be padded) and then higher decryption overhead. However, when the *select* query is not to select all (*select* *), our models perform better than the OBT because we eliminate decrypting whole rows in our models, whereas the OBT does not. When the query condition involved a single clause, the CBF system experienced delays comparable to our systems, but when the number of clauses in the *select* statements was two or three, its delay was almost double that of our system's delays since two or three tables are searched to form the final *select* query (the query that retrieves the records from the main encrypted table in the master cloud). Finally, the CryptDB incurs the highest delay among the systems as a result of the decryption of the onion layers overhead. The delay also increases when the statement condition has two or three clauses because more onion layers are required to be slipped off in different columns.

Table 4.4 Delays in milliseconds of executing (*select* *).
The average required records are 8% for one clause, 5% for two clauses, and 3% for three clauses.

	1 clause						2 clauses						3 clauses					
	BVM	BVSAC	BVSIT	OBT	CBF	CryptDB	BVM	BVSAC	BVSIT	OBT	CBF	CryptDB	BVM	BVSAC	BVSIT	OBT	CBF	CryptDB
10k	242	246	229	200	231	870	136	153	133	98	238	1329	115	132	112	77	324	1558
20k	326	382	352	275	338	1633	259	263	215	179	483	2016	213	224	192	142	542	12626
50k	624	510	783	481	547	10764	479	367	614	289	746	18969	440	303	545	221	893	21634
100k	1107	918	1240	791	970	20135	840	702	1128	562	1478	37078	835	610	1021	493	1604	59910

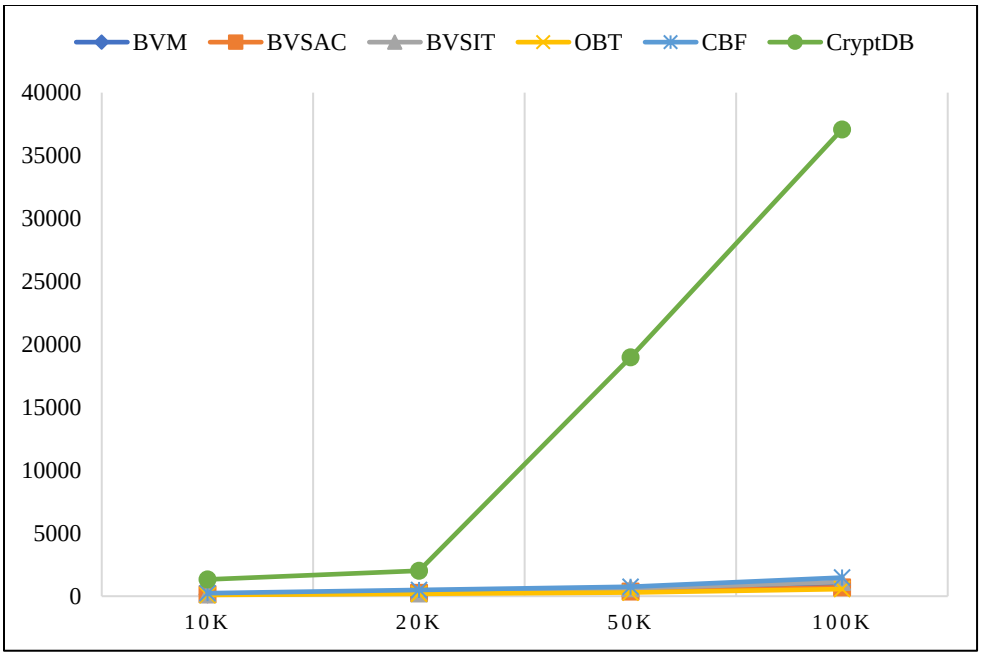


(a) Average total processing time for all models, including CryptDB.

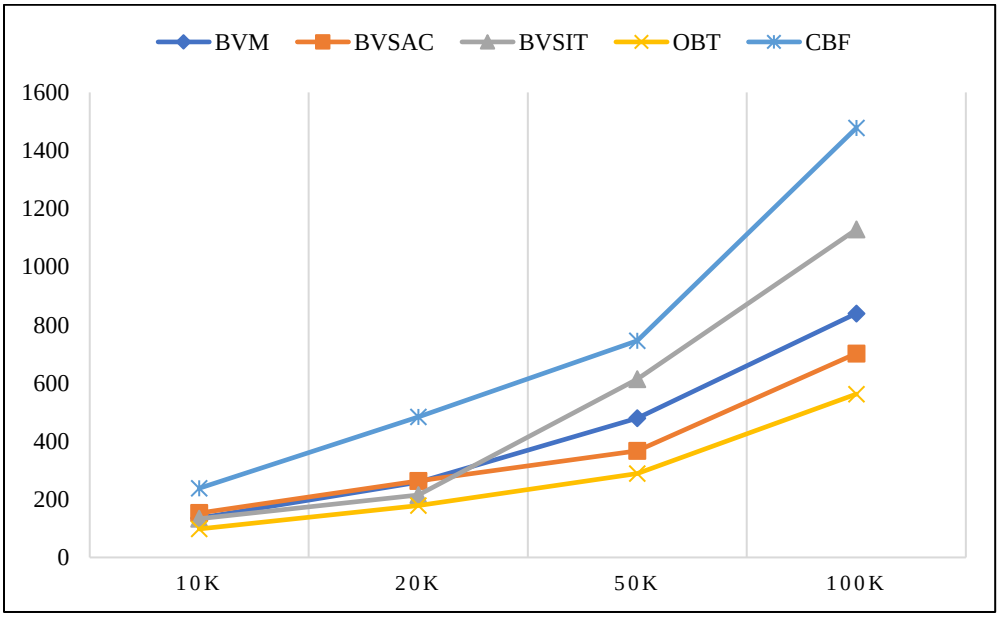


(b) Average total processing time for all models, excluding CryptDB.

Figure 4.4 The delay comparison of executing *select* all statements (*select* *) for all models in ms when the query condition contains only one clause.

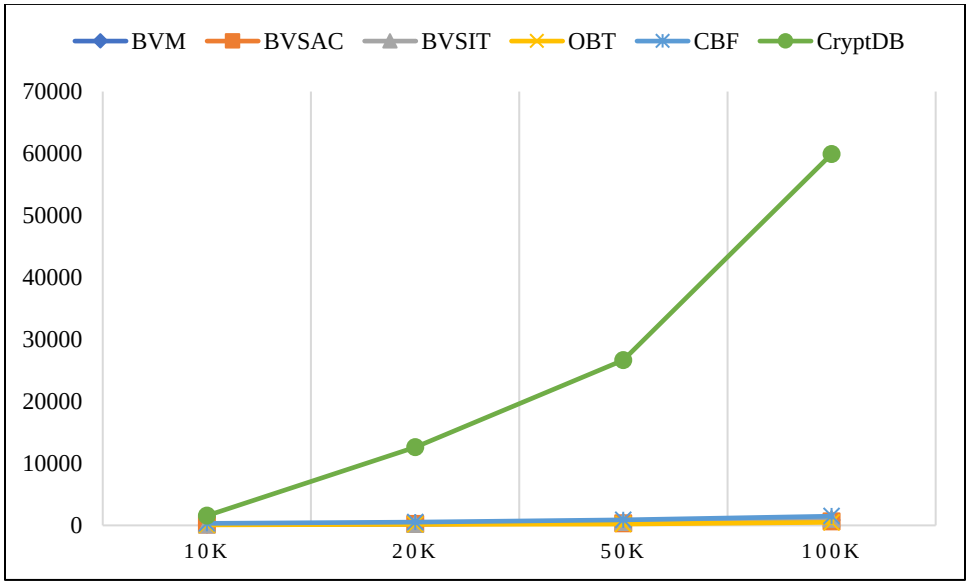


(a) Average total processing time for all models, including CryptDB.

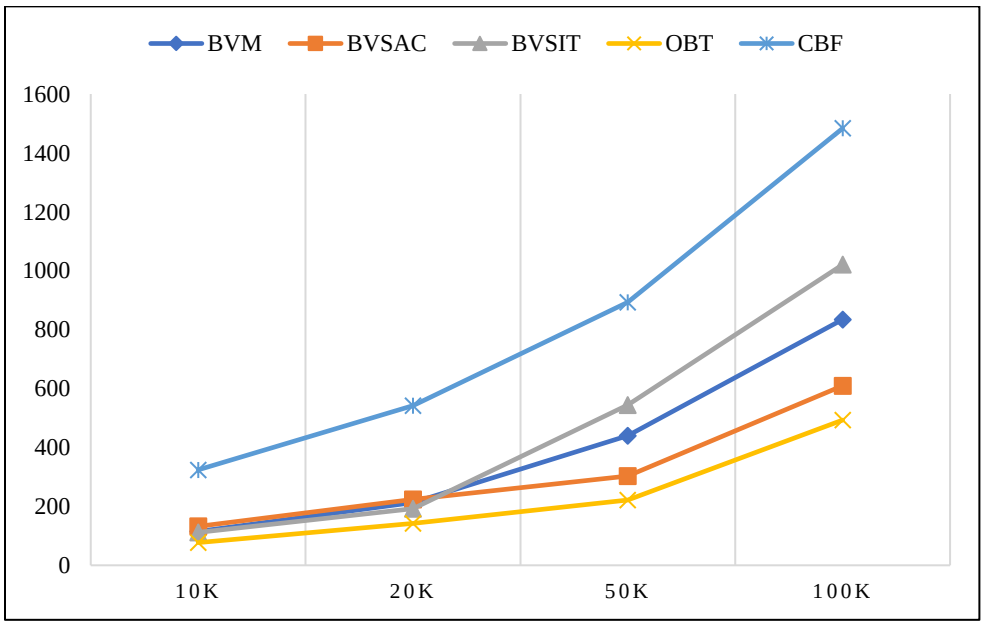


(b) Average total processing time for all models, excluding CryptDB.

Figure 4.5 The delay comparison of executing select all statements (*select* *) for all models in ms when the query condition contains two clauses.



(a) Average total processing time for all models, including CryptDB.



(b) Average total processing time for all models, excluding CryptDB.

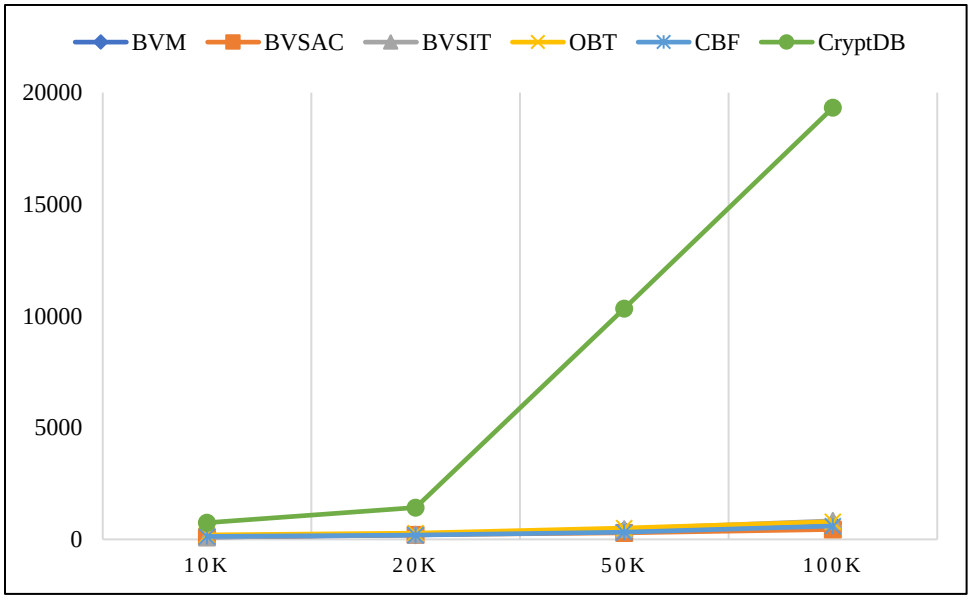
Figure 4.6 The delay comparison of executing select all statements (*select* *) for all models in ms when the query condition contains three clauses.

4.3.1.2.3.2 Select (col) Latency

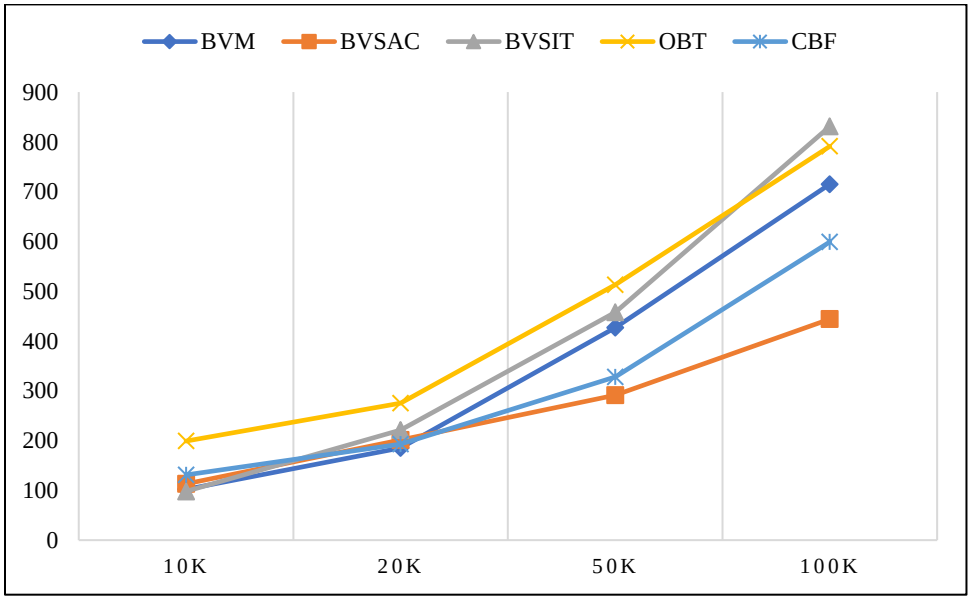
Table 4.5 presents the total query processing time of implementing *select* statements with one, two, and three clauses in the WHERE condition to retrieve a single column's values instead of *select* *. To make a simpler comparison, in Figure 4.7, Figure 4.8, and Figure 4.9, we plotted the numbers from Table 4.5. Examining the figures reveals that the performance of the proposed models increased in the same order as implementing *select* *. However, our models tend to be slower than the OBT for executing *select* * over different database sizes, but with single-column selection, our models except the BVSIT beat the OBT when executed over different dataset sizes. The BVSIT performs better than the OBT for the database of 20k or fewer records but not for more extensive databases (i.e., 50k and 100k records) because the number of decrypted indices fetched from the bits table grows, which adds more computation overhead to the QM.

Table 4.5 Delays in milliseconds of executing (*select* single column).
The average required records are 8% for one clause, 5% for two clauses, and 3% for three clauses.

	1 clause						2 clauses						3 clauses					
	DVM	BVSAC	BVSIT	OBT	CBF	Cryptdb	DVM	BVSAC	BVSIT	OBT	CBF	Cryptdb	DVM	BVSAC	BVSIT	OBT	CBF	Cryptdb
10k	102	113	98	199	131	746	57	76	55	98	145	1194	46	59	47	77	178	1491
20k	185	201	221	275	193	1417	125	139	107	179	251	1746	100	112	92	141	379	12491
50k	427	291	458	513	328	10332	264	229	308	289	572	18429	209	161	274	221	809	21364
100k	715	444	831	791	599	19325	534	330	611	562	903	35998	467	289	536	493	1107	54370

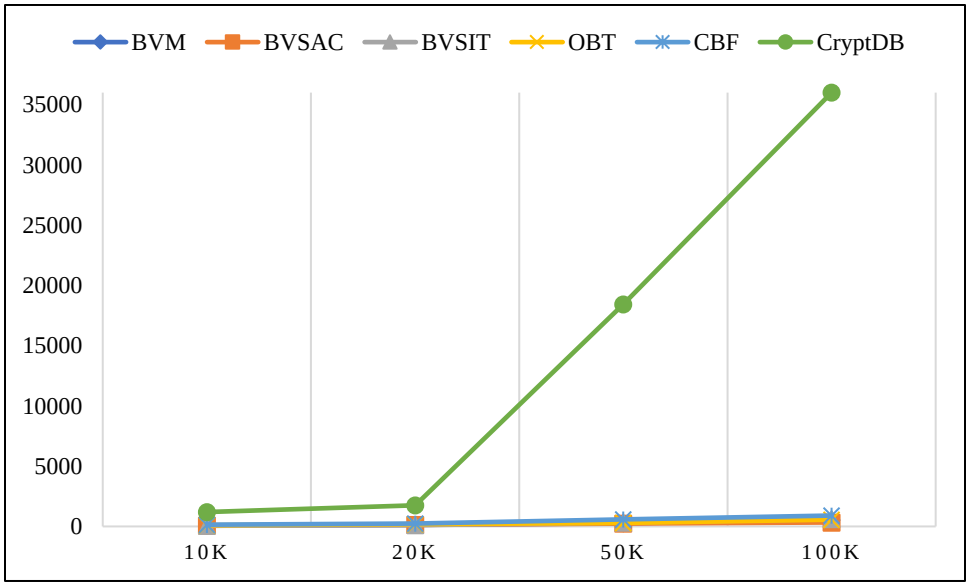


(a) Average total processing time for all models, including CryptDB.

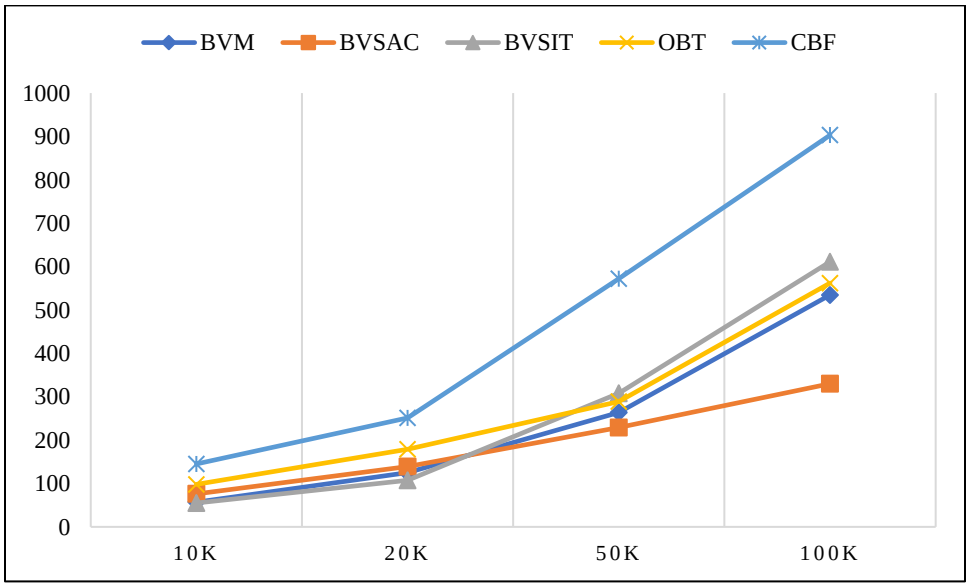


(a) Average total processing time for all models, excluding CryptDB.

Figure 4.7 The delay comparison of executing *select* statements to retrieve a single column's value for all models in *ms* when the query condition contains only one clause

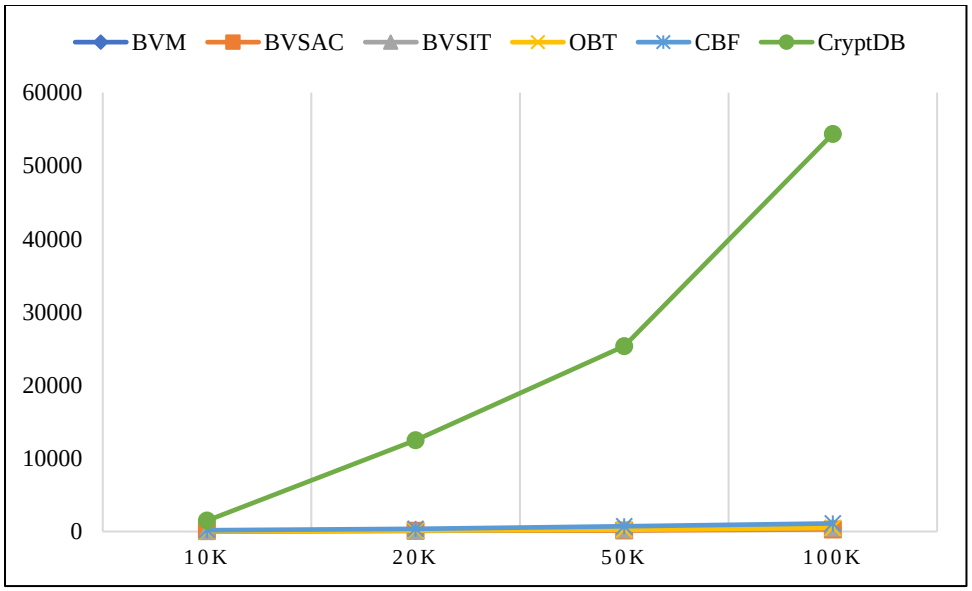


(a) Average total processing time for all models, including CryptDB.

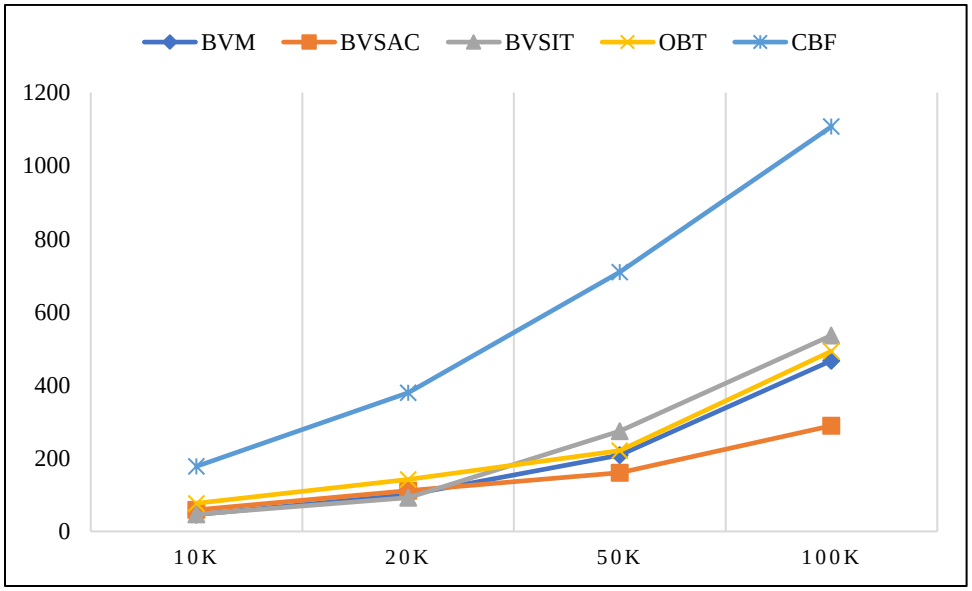


(b) Average total processing time for all models, excluding CryptDB.

Figure 4.8 The delay comparison of executing *select* statements to retrieve only one column's value for all models in *ms* when the query condition contains two clauses.



(a) Average total processing time for all models, including CryptDB.



(b) Average total processing time for all models, excluding CryptDB.

Figure 4.9 The delay comparison of executing *select* statements to retrieve only one column's value for all models in *ms* when the query condition contains three clauses.

4.3.1.2.3.3 Throughput

Throughput is defined as the amount of data transferred at a given time. By measuring the throughput, we can tell which system is more responsive to user's queries when the requested data are increased since the end user is the one who will be affected by the system slowdown. To measure the throughput, we executed a set of queries to retrieve % 25 then %50 of the records from a table holding 100,000 records. Then, we measure the amount of plain data (records' data after decryption) and divide it by the time taken by each system to deliver the required data to the user [66].

$$\text{Throughput} = \frac{\sum_{k=0}^{n} \text{unencrypted record's size (in bytes)}}{\text{Total time taken to deliver the data (MS)}}$$, where n= the total number of required records.

As seen in Table 4.6, the proposed systems achieved a higher throughput when compared with CryptDB and CBF systems. Further, BVSAC achieved a higher throughput among the proposed systems in this research which makes it the best choice for end users who seek a faster responsive system. OBT has the highest throughput and that is because it requires the least bytes requirements among all systems to encrypt data.

In Figure 4.10, we show the percent of the throughput for each system and we compare the systems with the throughput of MySQL when dealing with unencrypted data. To calculate the percent, we multiply the system throughput by 100 and divide it by the unencrypted throughput.

$$\text{Throughput percent} = \frac{\text{System's throughput} * 100}{\text{Unencrypted throughput}}$$

By examining Figure 4.10, we can say that our systems achieved a reasonable throughput percent and as the amount of required data increased (up to 50% of the rows), the throughput dropped slightly and still outperform CryptDB and CBF.

Table 4.6 The amount of plain data in kB that each system can deliver to the user per second.

	kilobytes per second (kB/s)						
	BVM	**BVSAC**	**BVSIT**	**OBT**	**CBF**	**CryptDB**	**MySql**
Requested rows = 25%	794	1083	778	1440	727	66	4720
Requested rows = 50%	971	1221	947	1447	816	57	10114

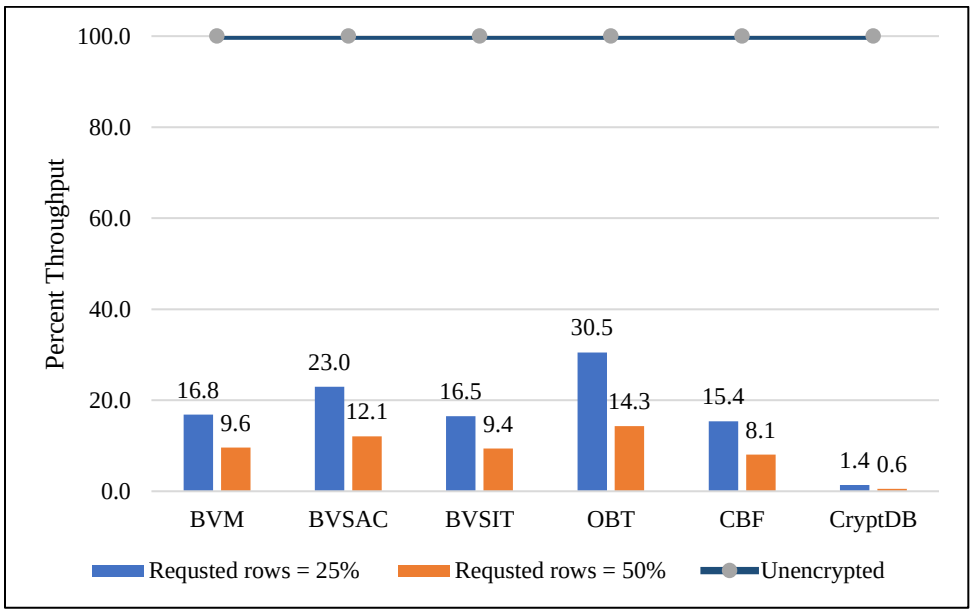


Figure 4.10 The percent of throughput of all systems compared with the throughput of MySQL when requesting unencrypted data. Note, the requested data are fetched by a select query from a table with 100,000 records.

4.3.1.2.4 Experiment 4: Update and Delete Statements

Figure 4.11 depicts the average time cost taken by each system to execute *update* statements. In this experiment, we executed *update* statements with only one predicate. As seen in Figure 4.11, the update time cost is high in all systems and is the result of updating an encrypted field in the database systems. The x-axis represents the number of rows affected by the *update* statements,

which means selecting the required data and then issuing 100 *insert* statements in the first case, 200 *insert* statements in the second, and so on. The BVSAC and OBT systems have a close update delay, 6,812 and 7,130 ms, respectively. However, in the OBT, the update process is risky when two or more substrings in the decrypted block match the updated value (i.e., any substring from the updated record that matches the new substring will be updated to the new substring). Thus, *update* in the OBT is not recommended. By zooming to the delays of the proposed models, the BVSIT is the slowest system because the *update* must be done in both the main encrypted table and the bits table, and it is the second slowest model. In addition, the BVM experienced a slightly higher delay than the BVSAC but still performed faster than the CBF, which is the third slowest model in this comparison. In CryptDB, the update cost is the highest of the compared systems. In summary, the BVSAC is the most efficient model for *update* statements.

Figure 4.12 demonstrates the time taken by each model to delete different numbers of records (100, 200, 500, and 1,000 tuples) when the delete condition has only one clause. The delete process selects the required rows and then deletes them. The delete is efficient in all the proposed models because the delete is performed after executing the *select* operation to retrieve the needed tuples. Instead of sending a single query to delete each record, we maintain the index of the record and then issue one query at the end to delete any record of its index in the delete query, that is, *delete* from TABLE_NAME where index in ( ). On the other hand, CryptDB is the slowest system to execute *delete* statements for the same reasons we mentioned earlier (i.e., onion layers decryption).

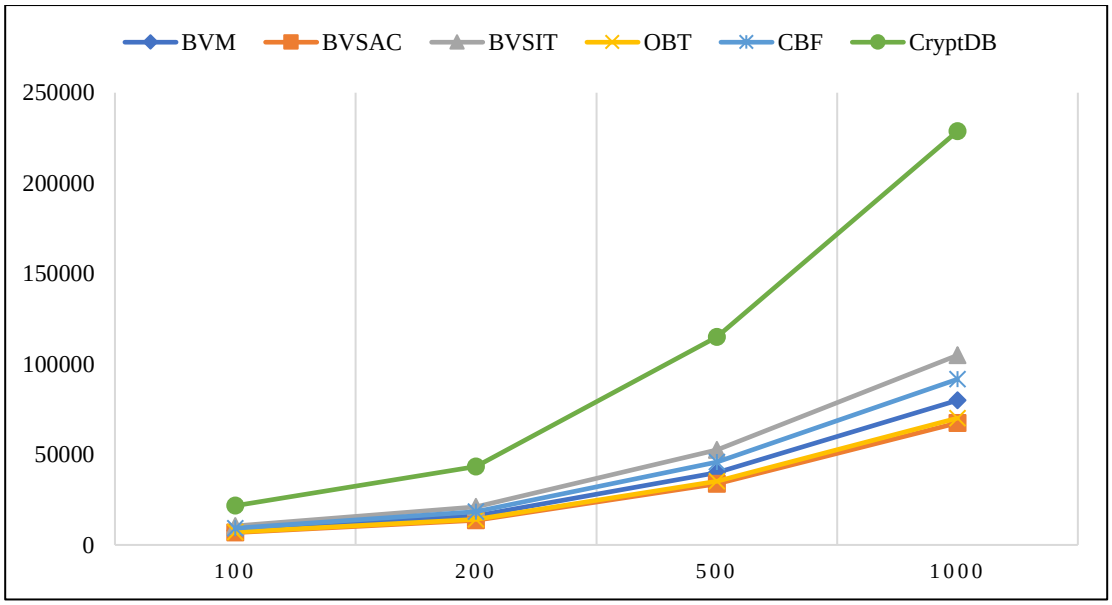


Figure 4.11 Comparison of the average delay of update statements for all models to update a number of existing tuples (100, 200, 500, and 1,000 tuples).

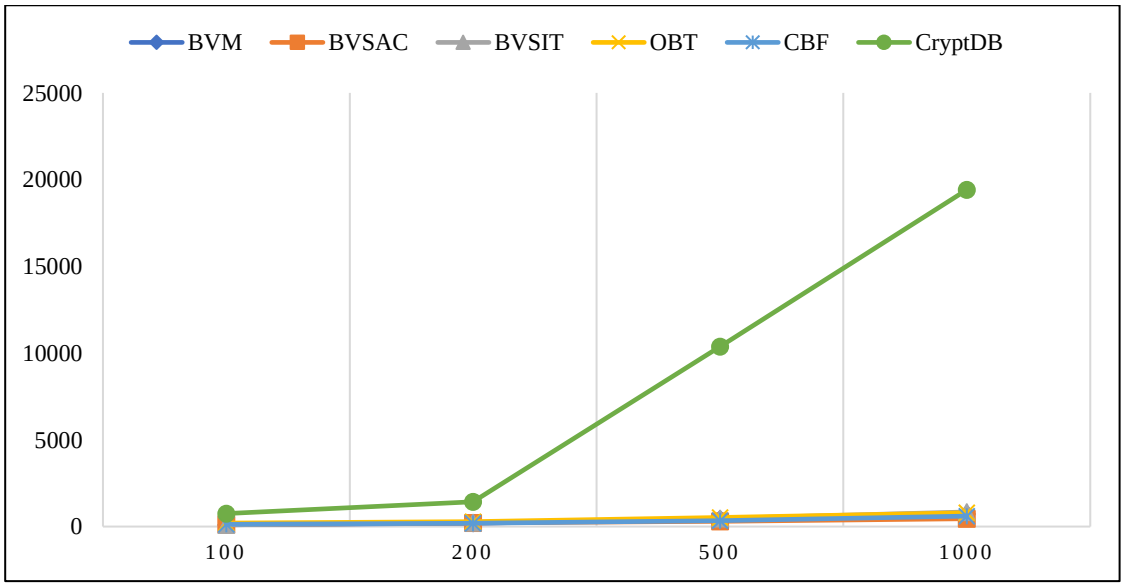


Figure 4.12 Comparison of the average delay of delete statements for all models to delete a different number of tuples (100, 200, 500, and 1,000 tuples).

4.3.1.1 Aggregation Functions

4.3.1.1.1 Experiment 5

Table 4.7 and Table 4.8 list the average execution times of the aggregation operators' (sum, average, count, max, and min) queries for the proposed prototypes along with the competing systems. All the aggregate queries executed in this comparison have one predicate (i.e., we did not execute the aggregate functions on the entire column's values), and the *average* number of records that meet the aggregate condition in all the queries is approximately 6%. The cost resulted from the aggregate functions relying on the cost of *select* statements in the proposed models because first, the query is rewritten to fetch the required rows (e.g., "select rows where department = Business"), then while decrypting the result, the QM enforces the aggregate function if the query condition is met (refer to the aggregate algorithms of the proposed systems in Chapter 3). In the OBT system, the procedures we followed in the proposed systems were followed, but the whole encrypted rows were fetched, which results in heavier computation overhead at the QM. In the CBF and CryptDB, the encryption algorithm used to encrypt the numerical field to support the addition operation over the encrypted values is homomorphic encryption (i.e., the Paillier algorithm). The Paillier algorithm relies on multiplying the modular to produce the encrypted sum, which requires heavier computation than the proposed models. In the implementation of the CBF, we adopted the Paillier algorithm in [64] to encrypt the extended salary column; however, to execute the aggregate function, the indices of the rows in column *A* that meet the aggregate condition must first be retrieved from the corresponding table at cloud *x* that maintains the extended column *A*. Then, the aggregate query was rewritten to apply the aggregate function over the encrypted values of the records that their indices are in the query predicate. From both tables (Table 4.7 and Table 4.8), we can conclude that the proposed models achieved a faster execution

delay than the others for smaller databases, but for larger databases, the BVSAC is the winner, while the BVSIT starts to experience heavier computations, as stated in the previous experiment. CryptDB is the slowest model in all database sizes due to the intensive decryption operations.

Table 4.7 Delay comparison in milliseconds of executing SUM, AVERAGE, AND COUNT functions. (only one predicate in the queries)

	SUM						AVERAGE						COUNT					
	DVM	BVSAC	BVSIT	OBT	CBF	Cryptdb	DVM	BVSAC	BVSIT	OBT	CBF	Cryptdb	DVM	BVSAC	BVSIT	OBT	CBF	Cryptdb
10k	110	121	107	206	143	878	111	121	107	206	147	878	104	115	101	200	131	746
20k	196	212	232	284	212	1718	198	214	234	286	208	1718	189	205	225	277	193	1417
50k	440	304	473	525	361	10762	441	305	474	527	363	10762	431	295	464	516	328	10332
100k	733	460	850	807	645	20185	734	462	853	806	647	20185	722	449	839	796	599	19325

Table 4.8 Delay comparison in milliseconds of executing MAX and MIN functions. (only one predicate in the queries)

	Average delay of MAX and MIN					
	BVM	BVSAC	BVSIT	OBT	CBF	CryptDB
10k	112	123	109	208	147	878
20k	201	217	237	289	208	1714
50k	447	311	480	532	363	10762
100k	741	468	858	815	647	20185

4.3.1.2 Join, Union, and Intersection

4.3.1.2.1 Experiment 6

In this study, we compared the delay of the proposed prototypes for join and union queries. In join queries, we did not include the competing systems in this experiment because the *join* has not been implemented in CryptDB (as the author stated in [62]) and the CBF. Therefore, we excluded all the competing systems from this experiment, and we performed the experiment on tables with the specifications displayed in Table 4.9.

Table 4.9 Specifications of the joined tables

Table size (number of rows)	Number of rows matching the join condition
500	100
1000	200
10000	1000

We joined two tables by the equality of the encrypted ID, and their structures are in Table 4.11 and Table 4.10 below. The ID column in both tables is partitioned into 10 partitions based on the last digit in the ID as (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9). Figure 4.13 demonstrates that the delay of the BVSAC is the highest, followed by the BVSIT, and the BVM has the least delay. The high delays in the join queries in all the proposed models are due to the nature of the join calculation and the amount of the decrypted data. Then, after enforcing the join condition (at the QM), the decrypted tuples were pushed into a LinkedHashSet to remove the duplication. In conclusion, although we experienced high join delays, we executed the join statements using the proposed systems over databases that were encrypted with a randomized encryption algorithm, such as AES-CBC.

Figure 4.14 demonstrates the average total time of processing the union queries of all the proposed systems. In all the proposed prototypes, the QM intercepts the union query and forwards it to the server without modification. Therefore, the cost presented in Figure 4.14 is the average delay of processing the union statements taken by all the models (we took the average execution time of all the systems since the delay of the union queries is roughly close for all the models). The delay we measured is from the time the QM intercepts the query until the final query result is formed. To remove the duplications, we pushed the decrypted tuples from both tables to a LinkedHashSet, which ensures no duplicated records exist. As seen in Figure 4.14, our models experienced a linear delay growth as the sizes of the tables increased.

In Figure 4.15, we show the latency, in milliseconds, of the intersection operator when executed on different tables with a different number of records. To perform this experiment, we reduced the tables' sizes because we experienced execution failures due to the lake of memory. Moreover, the leading cause to get this kind of error is the massive join computations that result from implementing the intersection operator. Note that the intersection queries were to intersecting tables based on all columns (i.e., not a partial intersection). As seen in Figure 4.15, The BVM system is the most efficient in performing the interaction, while the BVSAC is the least efficient in terms of latency. The factors that cause the variation in the delay are the same factors that affect delays in *join* queries.

Table 4.10 The structure of original international students table

Name of the Attribute	Data type	Is sensitive?
ID	int	Yes
Name	varchar	Yes
Address	int	Yes
Citizenship	varchar	Yes

Table 4.11 The structure of TA students table.

Name of the Attribute	Data type	Is sensitive?
ID	int	Yes
Name	varchar	Yes
SSN	int	Yes
Visa_Type	varchar	Yes
Salary	int	Yes
Department	varchar	Yes

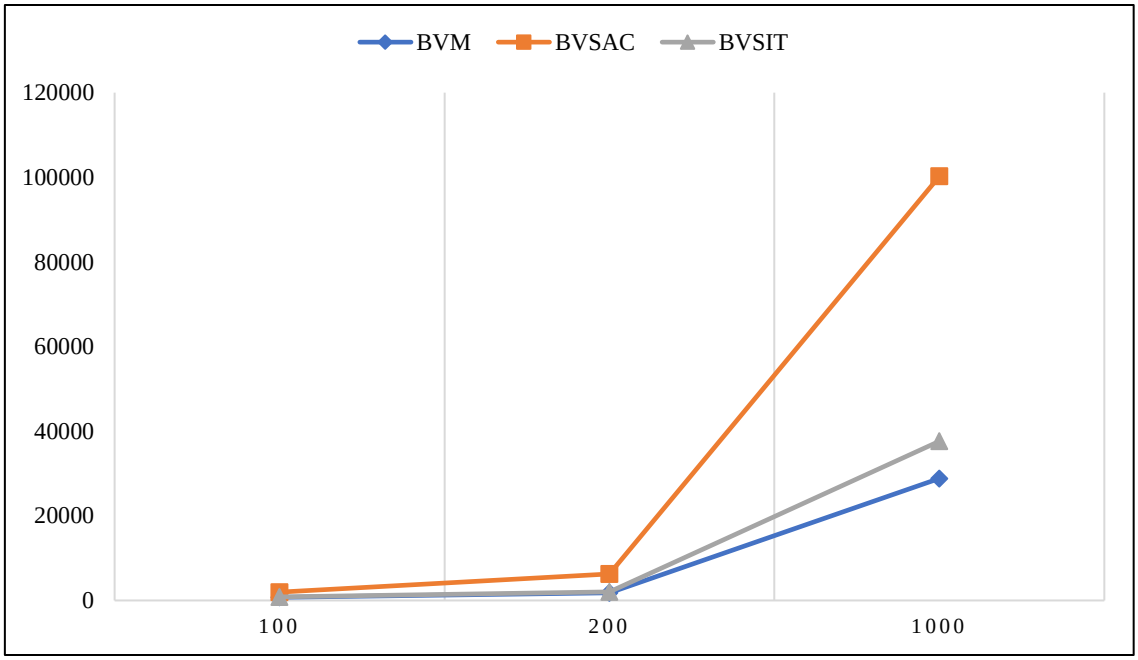


Figure 4.13 Join delay comparison among the proposed systems

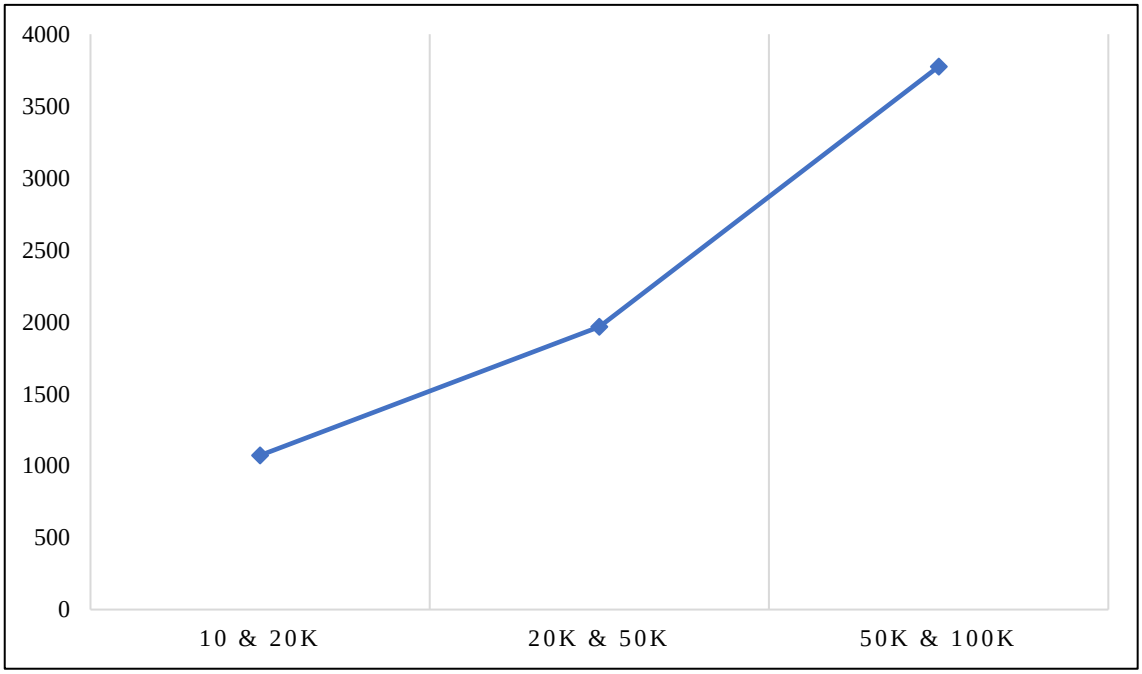


Figure 4.14 Union delay comparison among the proposed systems

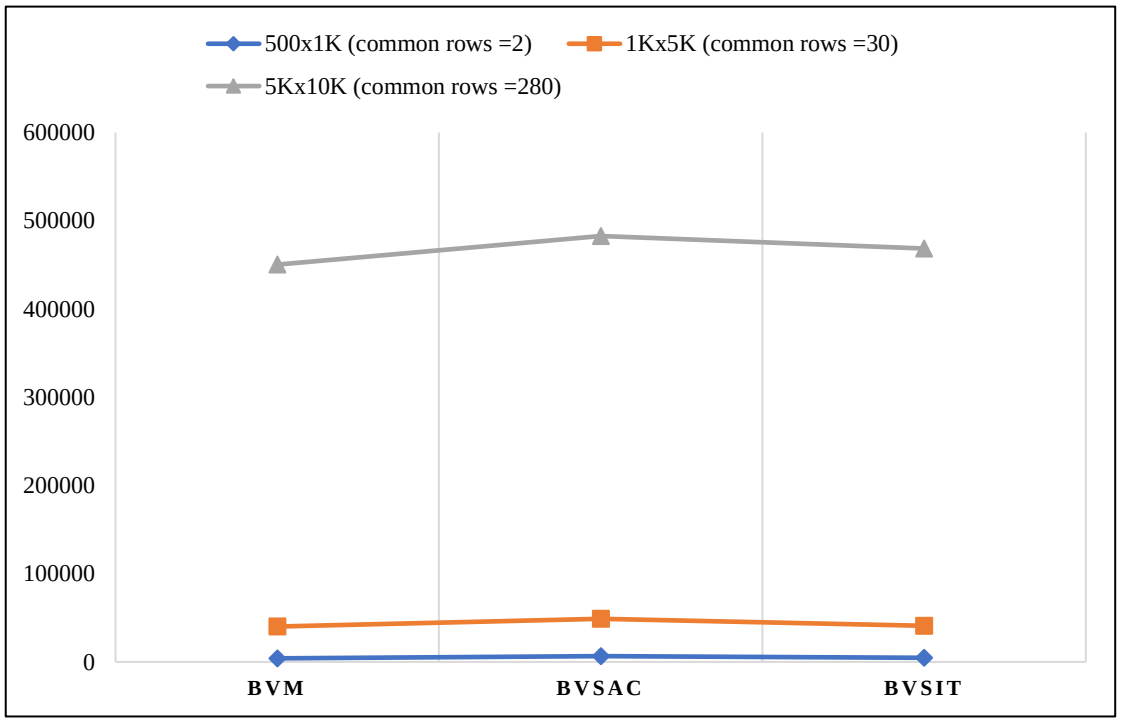


Figure 4.15 Intersection latency comparison among the proposed systems

4.3.1.3 Query Rewriting at the QM

4.3.1.3.1 Experiment 7

Figure 4.16 depicts the average time taken by each model to rewrite a query. The processes required to rewrite a query are presented in Table 4.12.

Table 4.12 Query rewriting stages

	Query parsing	PT search	BVs store and search (at QM)	Indices decryption
BVM	✓	✓	✓	x
BVSAC	✓	✓	x	x
BVSIT	✓	✓	x	✓

To calculate the average, we executed a set of queries for different cases of *select* statements with one, two, and three clauses. We focused on the *select* statements because they are part of the *update* and *delete* statements, so examining the *select* statement is enough to measure the cost of translating queries in each model; we excluded the crypto operations from this percentage.

In Figure 4.16, the pie chart represents the total average delay taken by all the proposed prototypes in translating a query. According to this chart, the BVM has the largest portion of delay (about 52%), followed by BVSIT (about 38%), leaving only 10 % of the total cost to the BVSAC. The BVM presents the highest percentage because of the BV preparation and processing. Afterward, the QM needs to prepare a list containing the indices of the candidate records to be fetched from the cloud. This step is mandatory before the translation process. In the BVSIT, although the QM does not maintain the BVs, which we migrated to the cloud, the main reason for the delay is the decryption processes of the fetched BV indices. Another factor contributing to this delay is that the QM must communicate twice with the MySQL engine to process a query. In the BVSAC, the QM searches the PT then rewrites the query; therefore, it has the least delay percentage.

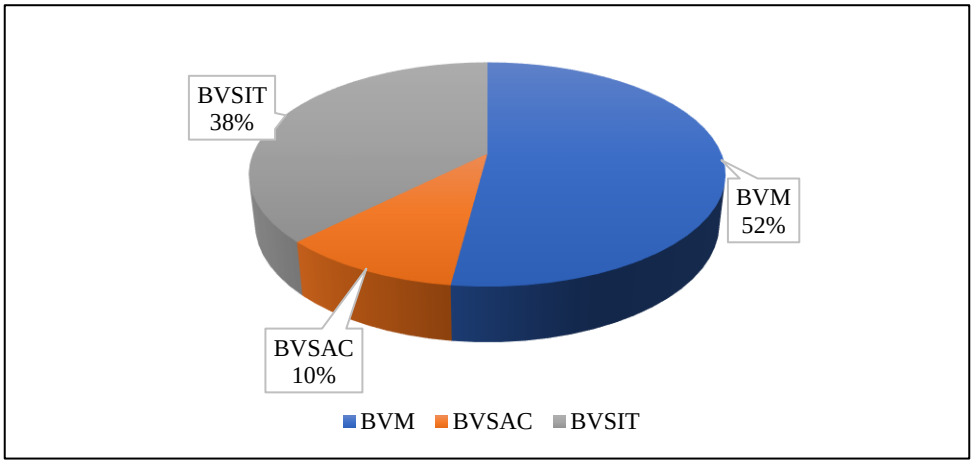


Figure 4.16 The average query translation cost at the QM for each model

4.3.1.4 Cloud Server Computation versus QM Computation

Figure 4.17 illustrates the average percentage of the computation time at the QM compared to the server when a query is executed by the proposed prototypes and by the other systems. The figure demonstrates that the CryptDB has succeeded in moving over 90% of the query processing computation delay to the server, leaving less than 10% of the computation delay to the QM (the QM is equivalent to the proxy in both the CryptDB and CBF). Despite this, the penalty is the higher query processing delay, as demonstrated in the previous experiments.

In our proposed prototypes, we can see that BVM has most of the computation overhead at the QM because in addition to the crypto operation, the BVs are processed locally at the QM. About 60% of the computation is at the QM in the BVSIT due to decrypting the BVs' encrypted indices after fetching them from the server. The OBT has about 53% of the computation at the QM because the QM is entirely decrypting the fetched records, even for queries not based on *select* * statements.

On the other hand, the BVSAC and CBF experienced a close percentage of overhead at the QM and cloud (about 50% of the computation moved to the server) because in the BVSAC, the BVs are searched at the server, and we avoid processing them at the QM, as explained earlier. Furthermore, in the CBF, the arrangements of different sets of indices and the rewriting of the query after performing conjunctive operations between the set of indices are the leading causes of obtaining a higher computation percentage at the QM.

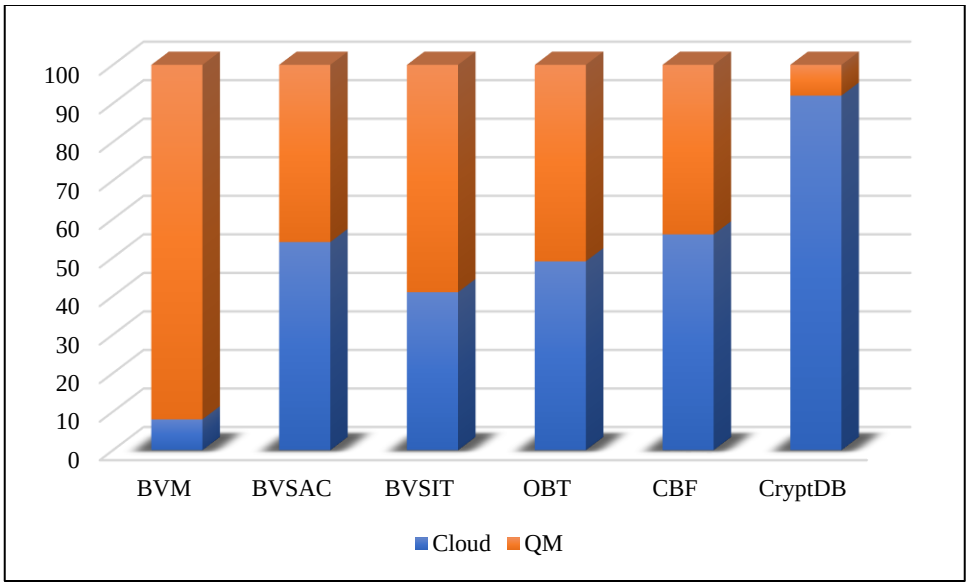


Figure 4.17 The percentage of the computation at the QM versus at the cloud server

4.3.2 Space Requirement

In this section, we evaluate each system in terms of the space requirement at both the cloud server and the QM as, for the proposed systems, a system could require not only space to store the encrypted table at the server but also space to maintain the BVs. In Table 4.13, for each proposed model (BVM, BVSAC, and BVSIT), we present the total space required for each record at the server in bytes. The index column is an integer that required 4 bytes, while the rest of the columns (the columns that will be encrypted) required 33 bytes (32 bytes since the secret key is 32 bytes + 1 byte to record the datatype length [67]. In addition, we present the space required to store the BVs on the server for each model. In the same table, we present the storage requirement per row for the OBT, CBF, and CryptDB. For the OBT scheme, all the values are stored as one block in one column; however, a column (datatype of int) is still required for the identifier of each sensitive column in the main table. In total, there are six columns in the plain table, which require six columns for identifiers. In the CBF, each numerical column (i.e., at the extended table in the salve

server) needs 257 bytes for Paillier ciphertexts plus 9 bytes for OPE ciphertexts plus 17 bytes for the randomized ciphertext at the main encrypted table that will be at the master server. For string columns, only deterministic ciphertexts are migrated to the extended columns and randomized ciphertexts at the main encrypted table at the master cloud. For the index column, we need 28 bytes (4 bytes for each extended table, which is six + the main table). On the other side, in CryptDB, each numerical column is extended to three columns (iv [17 bytes], Onion Ord [17 bytes], and Onion Add [257 bytes]), while the string column is extended to two columns (iv [17 bytes] and Onion Eq [17 ~ 33 bytes]).

As seen in Table 4.13 and Figure 4.18, the OBT has the lowest storage requirement at the server side, followed by the BVM then the BVSAC. The BVSIT requires a higher space requirement than the BVSAC since we store the BVs as an independent table rather than a column in the same encrypted table, as in the BVSAC. The CBF needs about 1,046 bytes for each record, which is the highest space requirement among the systems. CryptDB scores the second-highest space requirements, about 1,023 bytes/record.

Table 4.14 indicates the space requirements to store the BVs of the students' table (Table 4.1) at the QM for different sizes. We can conclude that the BVM requires 25,000 bytes to store the BVs of the table that holds 10,000 records, while other systems require no hard drive space to store any data to process a query. According to this, the space overhead is high in the BVM for executing a query because the whole set of the BVs is loaded from the hard drive into the data structures in the main memory.

Table 4.13 Storage requirement per record at the server (cloud) of the encrypted student table (Table 4.1) in all the systems

Name of the Attribute	Data type	BVM (bytes)	BVSAC (bytes)	BVSIT (bytes)	OBT	CBF	CryptDB
index	int	4	4	4	65 + 24	28	0
ID	varbinary	17	17	17		284	291
Name	varbinary	33	33	33		66	50
SSN	varbinary	17	17	17		284	291
Visa_Type	varbinary	17	17	17		34	50
Salary	varbinary	17	17	17		284	291
Department	varbinary	33	33	33		66	50
Bit vectors storages		0	4	37*	0	0	0
Total Space required		**202**	**206**	**239**	**89**	**1046**	**1023**

*37 = 20 bytes for storing bits as individual columns (we have 20 bits that require 20 columns of bit datatype) + 17 bytes for the encrypted index column

Table 4.14 Bits' storage requirements at the QM when the students' table has a different number of tuples

Model	Bits' storage requirement in bytes at the QM			
	10,000 records	**20,000 records**	**50,000 records**	**100,000 records**
BVM	25,000	50,000	125,000	250,000
BVSAC	0	0	0	0
BVSIT	0	0	0	0

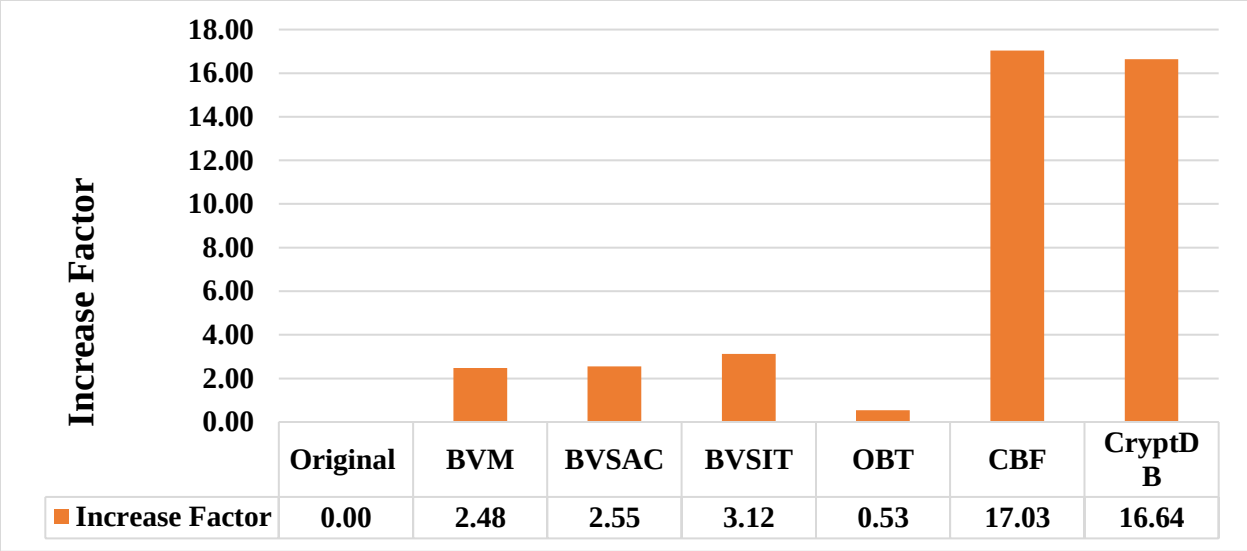


Figure 4.18 Comparison of the increase factor of encrypted table space among all systems. The increase factor (IF) = (x-y) /y , where x= the space required to store the encrypted table, y= the space required to store the original (unencrypted) table.

4.3.3 Factors that Increase the Efficiency of the Proposed Systems

The number of partitions of a column plays a significant role in the efficiency of our proposed models. The more partitions there are, the fewer records are fetched from the cloud, and the efficiency of the models are influenced by the number of fetched encrypted records from the cloud. Since we encoded the tuples data into bits based on attribute partitioning, the bits searching is quick, and we are not concerned about the length of the PT, as this contributes to the reduction of the retrieved encrypted tuples from the cloud.

Having a balanced tuples distribution among the partitions of every sensitive attribute is another challenge the data owner must account for. For illustration, one can suppose column A is partitioned into A1, A2, and A3; 50% of the records have their values of A fall under partition A1, 30% fall under A2, and 20% are mapped to A3. If a select query executed to retrieve a record falls under A1, then 50% of the encrypted records are retrieved, while if the query result exists in the records mapped to A3, 20% of the encrypted records are fetched from the cloud. How to partition

the columns of a table exceeds the scope of this dissertation; however, there is a penalty that studies have proposed to partition the columns, such as in [18], [20]. In [40], four methods can be used by the data owner to partition the columns into different partition domains, and each method works differently to divide each column into partitions based on its values. In future work, we will study this issue to design a partitioning algorithm to perform a fair partitioning for a table's columns.

4.4 Discussion

As observed in the conducted experiments, we evaluated each proposed model; then, we compared the models in terms of query processing speed, computation overhead, storage requirements at the QM, and the cloud server. In the terms of the speed, the main cause of the delay was the amount of data to be decrypted; however, all the proposed prototypes intended to reduce the amount of retrieved encrypted data from the cloud to less than 35%. The cause of having a variation in a query processing speed is linked to how and where the BVs are stored. The delay of the BVM is mainly affected by the BV transportation from the hard drive HDD to the memory, while in the BVSIT, we avoided this matter by storing the BVs as an independent relational table in the cloud. In the BVSAC, the BVs were stored as an additional column in the same encrypted table in the cloud. For single users restricted by the allowed cloud space, the BVM is the optimal model because it requires the minimum additional space to manage and process the encrypted relational tables. However, the computation overhead and the memory consumption at the QM will likely be a problem if more than a user shares the same computation resource since the whole set of BVs of the table must be loaded to the main memory to execute a query. Thus, the BVM is not recommended for a multi-user system, and the optimal model is the BVSAC, followed by the BVSIT. The BVSAC experienced the shortest execution time, the second-lowest space

requirement in the cloud, and the lowest storage requirements at the QM, meaning it is suitable for both single- and multi-user systems.

In terms of security, all the proposed models were at the highest security level. In the CBF, if the cloud providers did not collude, the security was high; however, it is not guaranteed, which downgrades the security level. In CryptDB, the authors suggested not putting back layers, which also reduces the security level. The OBT was secure if a randomized encryption algorithm was used; otherwise, the security level was reduced. For tables with excessive columns, such as thousands, the performance of the OBT would be jeopardized since the data of all the fields for each row are concatenated in one block. In the CBF, thousands of clouds were needed to store the extended columns, which makes this system impractical. In contrast, the proposed systems avoided such a problem because we performed cell-based encryption. We stored all the data on the same cloud server, which accelerated the query processing time for scalable databases. In Table 4.15, we provide a summary of the advantages and disadvantages of the proposed systems in this research and the systems we compared our systems with.

Table 4.15 Summary of the advantages and disadvantages of the schemes evaluated in this dissertation.

	Advantages	Disadvantages
BVM	✓ High security level. ✓ Has the least cloud space requirements among the proposed systems and the second-least among all systems. ✓ Faster query processing with smaller databases. ✓ Can support both single- and multi-key encryption. ✓ Keeps the same structure of the original table. ✓ Uses one cloud to outsource the encrypted data. ✓ The optimal choice for individual users who are limited by the space in the cloud and seek a faster execution time.	× Higher overhead (in terms of computation and space) at the QM because of query translation. × Most of the computations are performed at the QM, which makes it unsuitable for multi-user systems. × Does not support addition and multiplication (+,*).
BVSAC	✓ High security level. ✓ Has the least overhead (in terms of computation and space) at the QM. ✓ Processes queries over larger databases faster than others. ✓ Can support both single- and multi-key encryption. ✓ Keeps the same structure as the original table. ✓ Uses one cloud to outsource the data. ✓ Moves about 50% of the computation to the cloud server. ✓ Is the optimal choice for dealing with huge databases in both single- and multi-user systems that are not limited by the space in the cloud and seek a faster execution time. ✓ Has the least insertion and update latency, which makes it the best option for applications that might experience a high volume of these two operations.	× Experiences a higher delay than other proposed systems when dealing with smaller databases. × Higher delay than other proposed systems in the join and intersection operators. × Does not support addition and multiplication (+,*).

Table 4.15 (Cont.)

	Advantages	Disadvantages
BVSIT	✓ High security level. ✓ Works better with smaller databases. ✓ Moves about 39% of the computation to the cloud server. ✓ Can support both single- and multi-key encryption. ✓ Keeps the same structure as the original table. ✓ Is the optimal choice for dealing with small databases in both single- and multi-user systems that are not limited by the space in the cloud and seek a faster execution time.	× Experiences the highest delay in comparison to the other proposed systems when applied over large databases. × Has the highest insert and update delay. × Has the highest space requirement for the outsourced table among the other proposed systems. × Needs to communicate with the cloud server twice to process each query. × Does not support addition and multiplication (+,*).
OBT	✓ Requires the minimum space requirement among all systems. ✓ Secure if a randomized encryption algorithm is used. ✓ Uses one cloud to outsource the encrypted data.	× Loses the main table structure. × Does not support multi-key-based encryption. × Works only for databases that have a small number of attributes. × An inaccurate update could happen if the value to be updated matches two or more sub-strings in the same row. × Does not support the intersection operator. × Does not support addition and multiplication (+,*). × Supports only select * but not select (column).
CBF	✓ Enables the cloud server to execute SQL operators over encrypted data. ✓ Supports the functions of CryptDB and achieves processes queries faster than CryptDB.	× Not practical for single users since *N* accounts in *N* different clouds need to be maintained. × Experiences an exponential delay growth as the number of predicates in a query increase. × The security level is downgraded if more than one cloud provider colludes. × Works only for tables with a small number of attributes. × Requires the most space among all the systems to outsource the encrypted table.

Table 4.15 (Cont.)

	Advantages	Disadvantages
CryptDB	✓ Moves over 91% of the computation overhead to the cloud server. ✓ Supports most relational algebra operators. ✓ Can support both single- and multi-key encryption. ✓ Supports single- and multi-user systems.	× The security level is low because the secret key (*sk*) is passed to the cloud server. Although the server provider is trusted, they might be curious. × It is vulnerable to in-session attacks and inference attacks. × Authors suggest not to put back layers since that will cause computation overhead; however, that will downgrade the security level. × Requires the second-most amount of space among all the systems to outsource the encrypted table. × Works for small tables only, because if it is applied to larger tables, the computation delay will be incredibly high. × The data owner needs to install their own user-defined functions (UDFs) as an extension to the MySQL server in the cloud server; however, most cloud providers do not allow clients to install their own software in the cloud server. Therefore, CryptDB is not currently applicable in the real cloud environment.

5 Conclusion and Future Work

5.1 Summary

Cloud computing is an attractive computing environment for all kinds of users and companies. But, privacy breaches, not only by malicious attackers but also by curious providers, is the downside of this type of service, because users lose access control over outsourced data. There are many solution for this problem and data encryption is the effective one. However, executing SQL queries over encrypted data is challenging, especially if a randomized encryption algorithm, like AES-CBC, is used for the encryption. In this research, we first introduce the QM, a trusted server, which works as an intermediate between the cloud server and user(s) and performs all the crypto processes. In addition, we design a novel indexing technique based on predefining partitions for each sensitive attribute, and then encode each tuple to bits, accordingly. The bits are used to retrieve candidate tuples for a specific query that minimize the range of the retrieved encrypted tuples. Based on this encoding scheme, we proposed three different secure systems that each use a different way to store and maintain the index data (i.e., the bit vectors [BVs]) either locally at the QM or by migrating the BVs to the cloud sever. For each proposed prototype, we design different algorithms to accomplish query execution of different SQL relational algebra operators, and we make it resistant to attack scenarios, such as inference attacks. We test our models by implementing them and comparing their performance with state-of-the-art systems like CryptDB. We evaluate them in terms of their execution time requirement and space requirements. We find that the proposed systems require both less execution time and space when compared with most other competing systems.

5.2 Future work

As a future work, we aim to extend the proposed prototypes to support addition and multiplication operations. That could be achieved by using a different secure algorithm to encrypt the numeric attributes. Also, since the number of nodes (number of partitions) in the PT used by proposed systems in this dissertation impacts the efficiency of the developed systems, we aim to develop a fair partitioning algorithm that can divide sensitive attributes into *n* partitions where *n* is a threshold defined by system administrator. This would allow faster query processing speed by restricting the number of encrypted tuples retrieval from the cloud.

6 References

[1] "Cloud Database Market." [Online]. Available: source: https://www.marketresearchfuture.com/reports/cloud-database-market-6847.

[2] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Processing queries on an encrypted database.," *Commun. ACM*, vol. 55, no. 9, pp. 103–111, 2012.

[3] 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.

[4] Y. D. Jang and J. H. Kim, "A comparison of the query execution algorithms in secure database system," *Int. J. Electr. Comput. Eng.*, vol. 6, no. 1, pp. 337–343, 2016.

[5] M. Naveed, S. Kamara, and C. V Wright, "Inference attacks on property-preserving encrypted databases," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 644–655.

[6] D. Pouliot and C. V Wright, "The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1341–1352.

[7] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," *IEEE Trans. Comput.*, vol. 64, no. 3, pp. 698–706, 2013.

[8] F. Oggier and M. J. Mihaljević, "An information-theoretic security evaluation of a class of randomized encryption schemes," *IEEE Trans. Inf. forensics Secur.*, vol. 9, no. 2, pp. 158–168, 2013.

[9] A. Alsirhani, P. Bodorik, and S. Sampalli, "Improving database security in cloud computing by fragmentation of data," in *2017 International Conference on Computer and Applications (ICCA)*, 2017, pp. 43–49.

[10] H. Hacigümü\cs, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL over encrypted data in the database-service-provider model," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 2002, pp. 216–227.

[11] M. Nabeel, E. Bertino, M. Kantarcioglu, and B. Thuraisingham, "Towards privacy preserving access control in the cloud," in *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, 2011, pp. 172–180.

[12] Y. Zhu, H. Hu, G.-J. Ahn, D. Huang, and S. Wang, "Towards temporal access control in cloud computing," in *2012 Proceedings IEEE INFOCOM*, 2012, pp. 2576–2580.

[13] J. Raigoza and K. Jituri, “Evaluating performance of symmetric encryption algorithms,” in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2016, pp. 1378–1379.

[14] M. B. Yassein, S. Aljawarneh, E. Qawasmeh, W. Mardini, and Y. Khamayseh, “Comprehensive study of symmetric key and asymmetric key encryption algorithms,” in *2017 international conference on engineering and technology (ICET)*, 2017, pp. 1–7.

[15] M. Chakraborty, B. Jana, T. Mandal, and M. Kule, “An Performance Analysis of RSA Scheme Using Artificial Neural Network,” in *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2018, pp. 1–5.

[16] T. P. Innokentievich and M. V. Vasilevich, “The Evaluation of the cryptographic strength of asymmetric encryption algorithms,” in *2017 Second Russia and Pacific Conference on Computer Technology and Applications (RPC)*, 2017, pp. 180–183.

[17] A. Boicea, F. Radulescu, C.-O. Truica, and C. Costea, “Database encryption using asymmetric keys: a case study,” in *2017 21st International Conference on Control Systems and Computer Science (CSCS)*, 2017, pp. 317–323.

[18] O. M. Ben Omran and B. Panda, “Efficiently Managing Encrypted Data in Cloud Databases,” in *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*, 2015, pp. 266–271.

[19] O. M. Ben Omran and B. Panda, “Facilitating secure query processing on encrypted databases on the cloud,” in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, 2016, pp. 307–312.

[20] O. M. Ben Omran and B. Panda, “A new technique to partition and manage data security in cloud databases,” in *The 9th International Conference for Internet Technology and Secured Transactions (ICITST-2014)*, 2014, pp. 191–196.

[21] E. Damiani, S. D. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, “Balancing confidentiality and efficiency in untrusted relational DBMSs,” in *Proceedings of the 10th ACM conference on Computer and communications security*, 2003, pp. 93–102.

[22] B. Iyer, S. Mehrotra, E. Mykletun, G. Tsudik, and Y. Wu, “A framework for efficient storage security in RDBMS,” in *International Conference on Extending Database Technology*, 2004, pp. 147–164.

[23] E. Shmueli, R. Waisenberg, Y. Elovici, and E. Gudes, “Designing secure indexes for encrypted databases,” in *IFIP Annual Conference on Data and Applications Security and Privacy*, 2005, pp. 54–68.

[24] W.-K. Wong, K.-W. Wong, H.-Y. Yue, and D. W. Cheung, “Non-order-preserving Index for Encrypted Database Management System,” in *International Conference on Database and Expert Systems Applications*, 2017, pp. 190–198.

[25] F. Hahn, N. Loza, and F. Kerschbaum, “Joins over encrypted data with fine granular security,” *Proc. - Int. Conf. Data Eng.*, vol. 2019-April, pp. 674–685, 2019.

[26] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, “Processing analytical queries over encrypted data,” in *Proceedings of the VLDB Endowment*, 2013, vol. 6, no. 5, pp. 289–300.

[27] Y. ZHUANG, C. WEI, L. I. Jian, and W. LI, “Performance Enhanced for CryptDB Based on AES-NI Acceleration,” *DEStech Trans. Comput. Sci. Eng.*, no. ameit, 2017.

[28] A. Kumar and M. Hussain, “Secure query processing over encrypted database through cryptdb,” in *Recent Findings in Intelligent Computing Techniques*, Springer, 2018, pp. 307–319.

[29] G. Liu, G. Yang, H. Wang, Y. Xiang, and H. Dai, “A Novel Secure Scheme for Supporting Complex SQL Queries over Encrypted Databases in Cloud Computing,” *Secur. Commun. Networks*, vol. 2018, 2018.

[30] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[31] C. Gentry, S. Halevi, and N. P. Smart, “Fully homomorphic encryption with polylog overhead,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2012, pp. 465–482.

[32] D. Liu and S. Wang, “Nonlinear order preserving index for encrypted database query in service cloud environments,” *Concurr. Comput. Pract. Exp.*, vol. 25, no. 13, pp. 1967–1984, 2013.

[33] Z. Liu, X. Chen, J. Yang, C. Jia, and I. You, “New order preserving encryption model for outsourced databases in cloud environments,” *J. Netw. Comput. Appl.*, vol. 59, pp. 198–207, 2016.

[34] K. Li, W. Zhang, C. Yang, and N. Yu, “Security analysis on one-to-many order preserving encryption-based cloud data search,” *IEEE Trans. Inf. Forensics Secur.*, vol. 10, no. 9, pp. 1918–1926, 2015.

[35] S. Cui, M. R. Asghar, S. D. Galbraith, and G. Russello, “P-McDb: Privacy-preserving search using multi-cloud encrypted databases,” in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, 2017, pp. 334–341.

[36] D. Cash *et al.*, “Dynamic searchable encryption in very-large databases: data structures and implementation.,” in *NDSS*, 2014, vol. 14, pp. 23–26.

[37] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, “Leakage-abuse attacks against searchable encryption,” in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015, pp. 668–679.

[38] Y. Zhang, J. Katz, and C. Papamanthou, “All your queries are belong to us: The power of file-injection attacks on searchable encryption,” in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 707–720.

[39] E. Stefanov, C. Papamanthou, and E. Shi, “Practical Dynamic Searchable Encryption with Small Leakage.,” in *NDSS*, 2014, vol. 71, pp. 72–75.

[40] O. M. Omran, “Data Partitioning Methods to Process Queries on Encrypted Databases on the Cloud,” 2016.

[41] S. Shastri, R. Kresman, and J. K. Lee, “An Improved Algorithm for Querying Encrypted Data in the Cloud,” in *2015 Fifth International Conference on Communication Systems and Network Technologies*, 2015, pp. 653–656.

[42] M. R. Asghar, G. Russello, B. Crispo, and M. Ion, “Supporting complex queries and access policies for multi-user encrypted databases,” in *Proceedings of the 2013 ACM workshop on Cloud computing security workshop*, 2013, pp. 77–88.

[43] W. K. Wong, B. Kao, D. W. L. Cheung, R. Li, and S. M. Yiu, “Secure query processing with data interoperability in a cloud database environment,” in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 1395–1406.

[44] T. Raybourn, “Bucketization Techniques for encrypted databases: Quantifying the impact of Query Distributions,” Bowling Green State University, 2013.

[45] B. Hore, S. Mehrotra, and G. Tsudik, “A privacy-preserving index for range queries,” in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, 2004, pp. 720–731.

[46] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu, “Secure multidimensional range queries over outsourced data,” *VLDB J.*, vol. 21, no. 3, pp. 333–358, 2012.

[47] T. Raybourn, J. K. Lee, and R. Kresman, “On Privacy Preserving Encrypted Data Stores,” in *Multimedia and Ubiquitous Engineering*, Springer, 2013, pp. 219–226.

[48] J. Wang, X. Du, J. Lu, and W. Lu, “Bucket-based authentication for outsourced databases,” *Concurr. Comput. Pract. Exp.*, vol. 22, no. 9, pp. 1160–1180, 2010.

[49] J. Li, Y. K. Li, X. Chen, P. P. C. Lee, and W. Lou, “A hybrid cloud approach for secure authorized deduplication,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 5, pp. 1206–1216, 2014.

[50] M. Tao, J. Zuo, Z. Liu, A. Castiglione, and F. Palmieri, “Multi-layer cloud architectural model and ontology-based security service framework for IoT-based smart homes,” *Futur. Gener. Comput. Syst.*, vol. 78, pp. 1040–1051, 2018.

[51] V. H. Hacigumus, B. R. Iyer, and S. Mehrotra, “Query optimization in encrypted database systems.” Google Patents, Mar-2010.

[52] L. Bouganim and P. Pucheral, “Chip-secured data access: Confidential data on untrusted servers,” in *Proceedings of the 28th international conference on Very Large Data Bases*, 2002, pp. 131–142.

[53] S. Bajaj and R. Sion, “Trusteddb: A trusted hardware-based database with privacy and data confidentiality,” *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 3, pp. 752–765, 2013.

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

[55] G. Wang, C. Liu, Y. Dong, H. Pan, P. Han, and B. Fang, “SafeBox: A scheme for searching and sharing encrypted data in cloud applications,” in *2017 International Conference on Security, Pattern Analysis, and Cybernetics (SPAC)*, 2017, pp. 648–653.

[56] C. Liu, G. Wang, P. Han, H. Pan, and B. Fang, “A cloud access security broker based approach for encrypted data search and sharing,” in *2017 International Conference on Computing, Networking and Communications (ICNC)*, 2017, pp. 422–426.

[57] X. Li, H. Ma, F. Zhou, and W. Yao, “T-broker: A trust-aware service brokering scheme for multiple cloud collaborative services,” *IEEE Trans. Inf. Forensics Secur.*, vol. 10, no. 7, pp. 1402–1415, 2015.

[58] S. S. Chauhan, E. S. Pilli, R. C. Joshi, G. Singh, and M. C. Govil, “Brokering in interconnected cloud computing environments: A survey,” *J. Parallel Distrib. Comput.*, vol. 133, pp. 193–209, 2019.

[59] S. Almakdi and B. Panda, “Secure and Efficient Query Processing Technique for Encrypted Databases in Cloud,” in *2019 2nd International Conference on Data Intelligence and Security (ICDIS)*, IEEE, 2019, pp. 120–127.

[60] S. Almakdi and B. Panda, “Designing a Bit-Based Model to Accelerate Query Processing Over Encrypted Databases in Cloud,” in the 9th International Conference on Cloud Computing (CloudComp 2019), Springer, Cham, 2019. Accepted.

[61] S. Almakdi and B. Panda, “A Secure Model to Execute Queries Over Encrypted Databases in the Cloud,” in *2019 IEEE International Conference on Smart Cloud (SmartCloud)*, IEEE, 2019. Accepted.

[62] R. A. Popa, “CryptDB,” 2014. [Online]. Available: https://github.com/CryptDB/cryptdb.

[63] A. Madkour, “OPE.” [Online]. Available: https://github.com/aymanmadkour/ope.

[64] “Paillier.” [Online]. Available: https://www.csee.umbc.edu/~kunliu1/research/Paillier.html.

[65] “Package javax.crypto.” [Online]. Available: https://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html#package_description.

[66] J. Douglas, “Querying Over Encrypted Databases in A Cloud Environment,” Boise State University, 2019.

[67] “MySql Storage Requirments.” [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/storage-requirements.html.

7 Publications

"Secure and Efficient Query Processing Technique for Encrypted Databases in Cloud", Sultan Almakdi and Brajendra Panda, The 2nd International Conference on Data Intelligence and Security (ICDIS 2019), Texas-USA. **Published**

"Designing a Bit-Based Model to Accelerate Query Processing Over Encrypted Databases in Cloud", Sultan Almakdi and Brajendra Panda, The 9th International Conference on Cloud Computing (CloudComp 2019), Sydney-Australia. **Accepted**

"A Secure Model to Execute Queries Over Encrypted Databases in the Cloud", Sultan Almakdi and Brajendra Panda, The 4th IEEE International Conference on Smart Cloud (SmartCloud 2019), Tokyo-Japan. **Accepted**