BACHELOR'S THESIS

Department of Information Engineering and Computer Science

University of Trento

Italy

Supporting Joins and Numerical Computations over Encrypted Databases

Alex Pellegrini

Supervisors :

Dr. Muhammad Rizwan Asghar, Saarland University, Germany

Associate Prof. Dr. Bruno Crispo, University of Trento, Italy

September 2014

To my family relatives and friends

Abstract

Nowadays, the need of outourcing data has become necessary for many organisations as in-house storage is no longer worthy (in economical terms) due to everyday growing data. Data outsourcing introduces new challenges concerning trust on third parts, data security and confidentiality. To overcome such problems, new cryptographic techniques have been proposed. The system we propose will provide a way to secure outsourced data against unauthorised accesses and provide confidentiality against untrusted servers. At the same time, it provides a way to query encrypted databases. That is, evaluating encrypted queries over encrypted data without discolsing private information about the data or the query. In particular, this thesis will focus on providing a possible way to support joins between encrypted database tables. Furthermore, it describes a technique to store encrypted numerical values and evaluate range queries on such data.

Keywords: Encrypted Query Evaluation, Outsourced Databases Protection, Encrypted Searchable Data, Additive Homomorphic Encryption

Acknowledgements

There are a number of people I would like to thank, wihout whom this work might not have been carried out. They have made my studies and life a lot easier with their help and support.

My first thank goes to Associate Prof. Dr. Bruno Crispo for providing me the opportunity to join the CloudDB project. Thank to this, I have been introduced to computer security, which, I realised, I am very keen on. Furthermore, this gave me the possibility to learn a lot about cloud computing limitations and cryptographic techniques.

I would like to thank Dr. Muhammad Rizwan Asghar for his guidance and support during the last period, his passion inspired me a lot for my further studies and works. I am grateful for your help in improving the quality this work and my understanding at large. It is mainly thank to you behind my success in this work.

I would like to thank my mother Nicoletta Visintin and my father Claudio Pellegrini for their patience over these last years, to support and give essential advices in everyday life.

I would also like to thank my fellow students, in particular Alan, Davide, Federico, Francesco, Gabriele and all whom I studied with. I had a very good time these years with them, spent helping each other through every problem.

Alex Pellegrini
Trento, Italy
September 2014

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ABE Attribute Based Encryption

DCRA Decisional Composite Residuosity Assumption

HE Homomorphic Encryption

HVE Hidden Vector Encryption

KE Keyword Encryption

KMA Key Management Authority

PE Proxy Encryption

PECK Public key Encryption with Conjunctive Keyword

QP Query Processor

SDE Searchable Data Encryption

SQL Structured Query Language

TD Trapdoor Encryption

ToT Table of Tables

ToC Table of Columns

Table of Notations

A and B	Two generic sets
D	Generic Data
r	A random number
$PubParams$	Public encryption parameters
$O(n)$	Big-O notation, behaviour boundary
K_{pub}	Public key for a generic cryptosystem
n, z and y	Integer values
(c_1, c_2)	Structure of searchable ciphertexts
(e_1, e_2)	Structure of secure ciphertexts
(t_1, t_2)	Structure of trapdoors
k	Security parameter size
p and q	Two prime numbers of bit size k
$\mathbb{G}$	A unique subgroup of order q of $\mathbb{Z}_p^*$
g	A generator (primitive root)
$\phi(n)$	Euler's totient function (phi)
f	Keyed-hash function

s	A random key for f
H	Keyed collision-resistant hash function
x	The master secret key
x_{i1}	The client secret key for user i
x_{i2}	The server secret key for user i
$KE(D)$	Server side KE ciphertext of data D
$KE_i^*(D)$	Client side KE ciphertext of data D of user i
$PE(D)$	Server side PE ciphertext of data D
$PE_i^*(D)$	Client side PE ciphertext of data D of user i
T	Server side TD ciphertext of data Q
$TD_i^*(Q)$	Client side TD ciphertext of query Q of user i
$\wedge$	Logical conjunction
$\vee$	Logical disjunction
$\times$	Cartesian product

Part I

Introduction

Thesis Structure

This thesis is structured in four parts:

- **Part I**: Provides a brief overview of databases and query types, and a very quick explanation about why to use cryptography to protect databases in outsourced environments. It also surveys state of the art encryption schemes adopted during the last years to achieve main security goals.
- **Part II**: Describes the main ideas behind CloudDB system, by explaining the approach adopted along with cryptograpic techniques employed.
- **Part III**: Explains desing and implementation of how we implemented new features of CloudDB along with further cryptographic schemes adopted.
- **Part IV**: Concludes this thesis summarising main goals achieved during the work.

Chapter 1

Database Management Systems

Database Management Systems (DBMS) have always been widely used in a number of applications concerning the storage and manipulation of data. One could think of them as well-structured data archives, where information is supposed to be linked by some relationships. A database is basically a collection of tables containing plaintext data. Every table is made up by a set of fields, each of them characterised by some properties, such as data-type, sizes, relationships. The main purpose of a database is to provide a simple way to store large amounts of data and a likewise easy way to query such data to gather information bounded by some constraints. A database follows an user-defined schema to organise stored data, such a schema determines the logic organisation of information within the database. It defines every table and relation properties such as attributes names and types and further peculiarities as external dependecies such as foreign keys.

1.1 SQL Languages Overview

Usually, a user can access certain data, stored into the database, by issuing requests to the DBMS, which will fetch the desired data and return it. These requests are usually written by using a standard language called Structured Query Language (SQL).
These languages allow DMBS user to perform a lot of operations on the data, for instance:

- `CREATE` table;
- `DROP` a table or an entire database;

- `INSERT` rows into a table;
- `UPDATE` table;
- `DELETE` rows;
- `SELECT` data to gather information;

The last three operations, i.e. `SELECT`, `DELETE` and `UPDATE`, permit to users to specify also conditions and constraints to be evaluated before data is selected, deleted or updated. This is achieved by the use of the `WHERE` clause. A typical select query looks like:

```
SELECT * FROM 'table_name' WHERE <condition_tree>;
```

Whenever a query like this is submitted to the DBMS, the latter iterates over every `table_name`'s entry and tests whether the `condition_tree` is fully satisfied or not. If an entry satisfies the `condition_tree` it is returned to the user. Although we will consider the `WHERE` again during this thesis , we will pay more attention to the `JOIN` clause.

SQL languages allow the combination of two or more tables of the same database in order to retrieve related data belonging to different tables. It is useful to think to the JOIN process as a cartesian product between two or more sets, that is what it actually is. A new set is produced as a result that is made up by every possible combination of tuples within every set.
Let A and B be two different sets where $|A| = n$ and $|B| = m$, the cartesian product of the two is defined as follows:

$$A \times B = \{\ (a, b) \mid a \in A \wedge b \in B\ \}$$

Where the result's cardinality is :

$$|A \times B| = |A| \cdot |B| = n \cdot m$$

The corresponding SQL query is:

```
SELECT * FROM table1 JOIN table2;
```

This type of `JOIN` process is also known as cross join. SQL languages also support the `ON` clause to be used along with `JOIN`. This clause is used to specify to the DBMS two columns on which a comparison has to be performed when the joining takes place. Suppose every tuple $a \in A$ counts k attributes a_i (i.e., $|a| = k$) $\forall\ 0 \leq i < k$, and every tuple $b \in B$ counts l attributes b_j $\forall\ 0 \leq j < l$. Consider now a boolean function $f(x_1, x_2) : \mathbb{Z} \rightarrow \{0, 1\}$ that performs a comparison between two values. The definition of the product between two sets now becomes:

$$A \times B = \{\ (a, b) \mid a \in A \wedge b \in B, f(a_i, b_j) = 1\ \}$$

In this case, the corresponding SQL query would look like:

```
SELECT * FROM table1 AS t1
              JOIN table2 AS t2
              ON t1.attri = t2.attrj;
```

Where the binary operator = plays the role of the function f and `t1.attri` and `t2.attrj` are the two attributes a_i and b_j namely.

Moreover, SQL provides methods to perform calculations on stored data and return the computed value. Such methods are called aggregate functions and some examples are:

- `MIN` / `MAX`;
- `AVG` is used to compute the average value of a certain column;
- `COUNT` used to count, for example, the number of records satisfying a condition;
- `SUM` allows to compute, for example, the sum of the values of a numerical column;

We will take care mostly of `SUM` as the addition of encrypted values is one of the covered aspects of this work.
A simple SQL example of the `SUM` aggregate function looks like:

```
SELECT SUM(salary) AS total FROM employee;
```

This query, basically iterates over every entry of the table `employee` and sums up every value stored into the attribute field `salary`. When this process ends, the final values is returned to the user under the name `total`.

1.2 Encrypted Databases

Security in application and online services is somehow compromised by different kind of attacks and vulnerabilities. Deploying data management to third part entities can be very attractive when it comes to the need to store large data archives or performing operations on them.
Cloud computing is a paradigm that consists in sharing computers and other devices over a network, which are offered as a service to the end user, usually in form data storage and manipulation. This is obviously very attractive for a customer as it involves neither hardware or software purchase nor maintenance. On the other hand, a curious cloud service provider may learn about sensitive data, such as personal information, health data and credit card numbers.
It could be, of course, possible for an unauthorised entity to bypass the access control system on the cloud server and gain access to the stored data. The same cloud has to be supposed as untrusted as well.
What if we want to store some data in the cloud and keep it private? Moreover, we also want the cloud server to perform searches and computations on our behalf without ever knowing anything about the stored data or what we are looking for.
Cryptographic techniques could be applied to achieve this and at the same time overcome confidentiality violations. A solution has also to take in account that more than one user can access (read and write) the same data (*full-fledged multi-user* scheme). Trivial approaches like encrypting the whole data and share the secret key with every trusted user, are not appliable. That is, in the case we have to revoke access rights to an user, it is obviously mandatory to change the secret key and re-encrypt the whole data. Another problem is that every user who wants to read the stored data, has to entirely download the encrypted dataset to a local environment, decrypt it and then search for some desired information. This definitely does not scale for very large datasets.
A number of solutions have been proposed in the last decade to overcome such limitations and violations. We will see in the next chapter some of those approaches.

Chapter 2

State of the Art Schemes

In this chapter, we discuss existing solutions to mentioned issues describing key management processes as well as cryptographic approaches adopted to overcome confidentiality violations and ensure data and query privacy. First of all, we should give a general idea of what we mean for *Keyword Search*. Examined systems are then listed below based on features support.

2.1 Keyword Search

To start with we should define the concept of $Keyword\ Search$, considering a *Pubic Key Encryption* scheme (E). Suppose we want to store a collection of English poetries on the cloud. Of course, we do not want anybody (including the cloud server) to learn about our poetries so we store them in encrypted form. Suppose a poetry P_i is made up by n words $W_{i1}, W_{i2}, ..., W_{in}$. We may consider to send to the cloud server the ciphertext of the full poetry P_i and attach to it a *Searchable Data Encryption* (SDE) of each word W_i [1]. What is finally sent to the server is therefore:

$$E_{Kpub}(P_i) \parallel SDE_{Kpub}(W_{i1}) \parallel ... \parallel SDE_{Kpub}(W_{in})$$

where $Kpub$ is our public key. We want now to retrieve the poetry P containing a certain word W without disclosing any information about P and W. To achieve this, we encrypt W in such a way that the resulting ciphertext can be compared to each $SDE_{Kpub}(W')$ of every stored document. If a match is found, the matching document is returned and decrypted. In the following sections we will see how *Keyword Search* is employed to query outsourced encrypted data.

2.2 Single User Schemes

Single user schemes are proposed to solve data confidentiality in outsourced environments when only one user has the key to encrypt and decrypt the data. The cloud server evaluates encrypted queries on ciphertext data and returns encrypted results. This way the server never learns anything about stored data, queries and encryption key. The secret key could also be shared among a group of users in order to make this kind of approach accessible by many users. However, this implies the fact that, when access is revoked to a user, a new secret key is distributed to the remaining set of users. The stored data has to be re-encrypted using the new secret key. This does not scale for large datasets.
Song *et al.* [2] are the first to address practical keyword search on encrypted data using symmetric encryption, where document is encrypted word by word. To perform search, the user sends keyword encrypted with the same key to the server, which tests each word in every document. This scheme reveals statistical information, such as the frequency of each word.

Goh [3] proposes an efficient secure index construction built using pseudo-random functions and Bloom filters, where each filter is randomised using a unique document identifier, they introduce false positives though. Chang and Mitzenmacher [4] proposed a more secure solution to indexing as it does not disclose the number of words in a document even though they do not support set updates (insertion and deletion). Kamara *et al.* [5] introduced a Dynamic Symmetric Searchable Encryption (DSSE) to support arbitrary updates.
Bucketisation has been proposed for reducing range queries to equality searches.

> *"Bucketization is one technique for executing queries over encrypted data on a Database As Service server. Encrypted records are divided into buckets, where each bucket has an ID and a range defined by the minimum and maximum values in the bucket. The client contains indexing information about the range of each bucket on the server. Client queries are then mapped to the set of buckets that contain any value satisfying the conditions of the query. The original queries are translated to bucket–level queries, which request the encrypted buckets containing the desired values."* [6]

Hacigumus *et al.* [7] propose a solution that enables SQL queries by using Bucketsation and support range queries after several interactions between the user and the server. Bucketisation introduces false positive results and is not scalable for large databases.

Order Preserving Encryption (**OPE**) is, instead, a very popular and efficient approach to support range queries, which was proposed by Boldyreva [8]. OPE leaks information about order relationship between ciphertexts. It is used by CryptDB to support range queries. CryptDB was introduced by Popa *et al* [9]. It is a single-user system which employs an implementation of Song *et al.* [2] scheme to support keyword search. Furthermore, it provides possibility of performing simple computations on encrypted data through homomorphic encryption based on Paillier cryptosystem [10].

2.3 Semi-Fledged Multi-User Schemes

Semi-fledged multi-user schemes are designed to allow a single writer and many reader sharing a decryption key. The matter involves the share of a public key among users authorised to read data. Such a key sharing implies the risk of key exposure. To put this problem right, Diffie-Hellman key exchange [11] could be applied. Otherwise, a regular key update process could be realised involving re-encrypting whole stored data every time the new decryption key is distributed. However, this approach is neither practical nor scalable, particularly when data sets are large. The basic idea is that most of the semi-fledged multi-user schemes adopt public-key encryption schemes to allow one writer and many readers. Basically, the writer shares its public-key with authorised users and encrypts the data to be stored with its private key. Using the writer public key, every user who possesses it can read.

Yang *et al.* [12] proposed a solution in which the database owner encrypts the data and assigns to each user a unique key for searching and reading the data. The main idea is that the data owner splits the master secret uniquely between each user and the server. Shen *et al.* [13] propose a symmetric key based predicated encryption scheme that achieves predicate privacy. Li *et al.* [14] propose a solution based on *Hidden Vector Encryption* (**HVE**) that uses multiple trusted authorities to distribute search capabilities to users. These solutions are rather inefficient due to the expensive pairing operations they involve. Lu and Tsudik proposed a solution based on *Attribute-Based Encryption* (**ABE**) [15] and blind Boneh-Boyen [16] weak signature scheme. Both HVE and ABE schemes are a sort of *Predicate Encryption*, or else an encryption paradigm where secret keys are associated with a a predicate and a and ciphertexts with an attribute. A secret key k_p can decrypt a ciphertext c_a only if the predicate p is appliable to the attribute a.

2.4 Full-Fledged Multi-User Schemes

Full-fledged mulri-user schemes allow multiple user to both read and write data without sharing any key. Key management is, therefore, scalable for a large number of users. Upon user removal, re-encryption is no longer needed. A new key pair is generated every time a new user is authorised to perform operations over encrypted data. Current full-fledged multi-user schemes have limitations. Current schemes support only keyword-based searches without supporting more complex queries, such as numeric inequalities and range queries and a coarse-grained access control model.

Hwang *et al.* [17] extend Public key Encryption with Conjunctive Keyword (**PECK**) to multi-user settings. Every user has to possess other users public keys in order to encrypt a message readable by all. This is obviously not scalable for a large number of users, moreover, when a new user is added to the system re-encrypting of all the data is required. Dong *et al.* [18] propose *Searchable Data Encryption* (**SDE**), a multi-user scheme that supports keyword search. The SDE is based on proxy encryption and does not require interactive protocols or pairing. As a result, the SDE is an efficient scheme for performing search on encrypted data.

Part II

CloudDB

Chapter 3

System Overview

CloudDB [19] is a system whose intent is to extend pre-existent works by protecting data confidentiality in outsourced environments while allowing multi-user access-policies and supporting complex SQL-like encrypted queries on encrypted databases. This does not limit its scope to keyword searches but allows an user to issue SQL-like queries inclusive of WHERE clause. The latter, in particular, can express conjunctions and disjunctions of equalities on `string` typed fields and both equalities and inequalities on numeric attributes .

3.1 The Model

CloudDB system follows the *Client-Server* paradigm and considers three main entities:

- **User** is a trusted authorised part of the system, which can store encrypted data into the database and query the latter to retrieve iformation by issuing complex encrypted queries. When a retrieve query is satisfied the user can decrypt the returned data.

- **Cloud Server** is the cloud server hosting the database. It evaluates incoming complex encrypted queries and returns encrypted results. Furthermore it also checks user access rights. It is supposed to be *honest* but *curious*, meaning that every operation is believed to be performed honestly but curious to learn about the data stored or exchanged as data and requests could be somehow analysed.

- **Key Management Authority** is a fully trusted entity that is intended to generate encryption keys. For each authorised user, it distributes an unique key pair between the same user and the server. Every key pair is computed starting from a master encryption key which is held by the KMA itself. It is also responsible of revoking keys to unauthorised users.

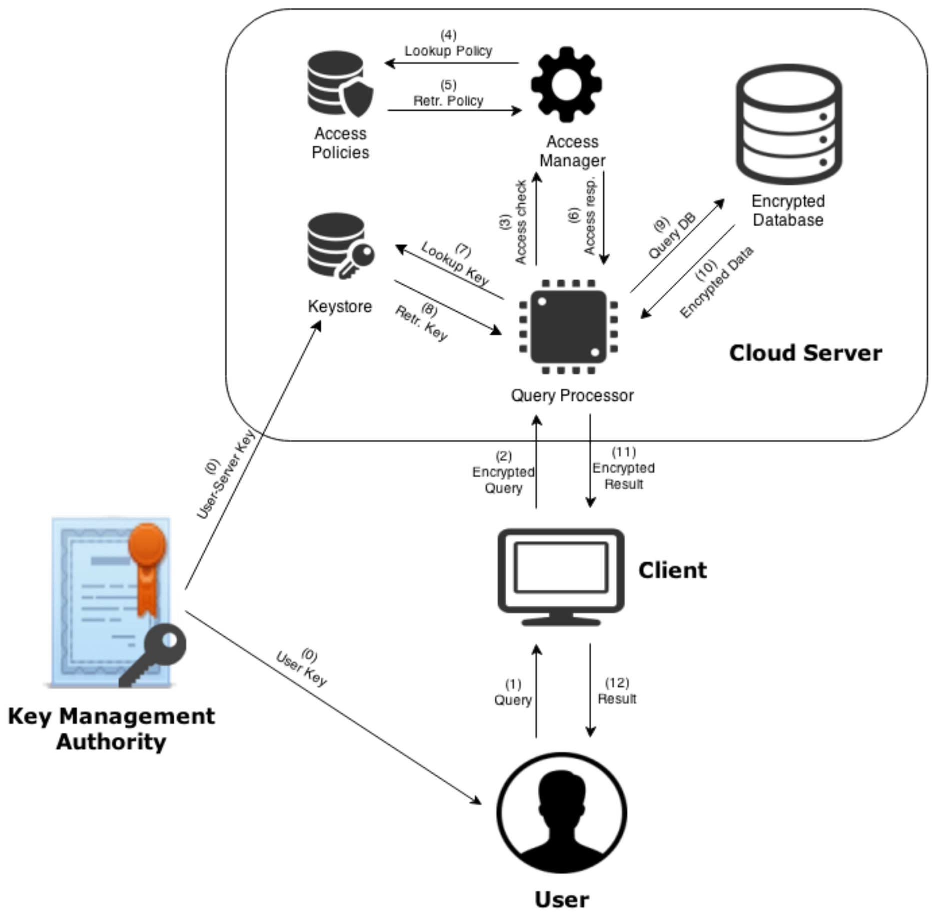

Figure 3.1: CloudDB model workflow

Figure 3.1 graphically explains the workflow of CloudDB by showing interactions between considered entities.
At the system boot, the Key Management Authority (KMA) is initialised

acoordingly to a choosen security parameter, that is it creates public encryption parameters and a master secret key. Every time a user is authorised to operate on the system, KMA generates an unique pair of keys, which are securely given to the new user and the server (0). This step is only performed once when the user is added to the system.
The user iterfaces to the system through the client (C), which is a graphical interface that permits composition of queries, saves them, loads stored queries. When the user submits a query (1) the client intercepts it, performs the first round of proxy trapdoor encryption on the same and sends the resulting ciphertext and user data to the cloud server (2).
The Query Processor (**QP**) placed on the server, receives the incoming queries and user data and checks whether the user is authorised to access operate on the database through the Access Manager (3). The latter queries the policies repository (4) to obtain access rules associated to the user (5). Once user policy is returned to the QP it returns an access denied error to C or retrieves the user-associated key from the keystore (7,8) accordingly.
Once QP obtained the user key, it can compute the server round of trapdoor encryption and executes the resulting encrypted query on the DBMS (9). The query results in a set of encrypted data (10), whcih is then partially decrypted by the QP by using, again, the user-associated key. QP sends the partially encrypted data to C (11) which runs the second round of proxy dercyption and displays the decrypted data to the user(12).

3.2 Encryption Schemes

Two encryption schemes are employed by the system, namely Keyword Encryption (**KE**) used to support equality encrypted match and Proxy Encryption (**PE**) to ensure data confidentiality and retrieval. Both schemes introduce randomness so as to relieve frequency analysis attacks. Every piece of data (database entry field) is stored under both encryptions.
Both client and server perform a round of encryption on every information flow with respect to the type of operation being performed, read (e.g., `SELECT`) or write (e.g., `INSERT` and `DELETE`).

3.2.1 Public Parameters Initialisation

First of all, we should describe how the KMA generates master secret key and public parameters with respect for choosen a security parameter. Moreover, it is explained how key pairs are distributead among authorised users. At system boot, the KMA runs an initialisation algorithm that takes a security parameter as input and outputs the set of public parameters while keeeping the master secret key private.

Init(k): KMA generates two prime numbers p and q such that $q \mid p-1$. Moreover, it outputs a finite cyclic group $\mathbb{G}$ such that it is the unique subgroup of order q of $\mathbb{Z}_p^*$ and a primitive root $g \in \mathbb{G}$. A random value x is chosen from $\mathbb{Z}_q^*$, which will be part of the master secret key, and the public parameter $h = g^x$ is computed. Furthermore, a secure hash function H and keyed-hash function f_s along with a random s key for it.

Algorithm 1: Public Parameters Generation

Public parameters are then published as $PubParams = (\mathbb{G}, g, q, h, f_s, H)$.

Every time a new user i is authorised to perform operations on the system, KMA chooses a random $x_{i1} \in \mathbb{Z}_q^*$ and computes $x_{i2} = x - x_{i1}$. Finally, x_{i1} is securely given to user i while x_{i2} is the user$_i$-server key and is so given to the server. Latter saves the received key into the keystore.

3.2.2 Proxy Encryption

Proxy Encryption (PE) follows the El Gamal encryption scheme [20], which is shown in Figure 3.2, and is employed in order to ensure privacy of data in outsourced environments such as cloud servers. This scheme allows CloudDB to support multi-user operations. The scheme used to perform Proxy Encryption is divided in two rounds, *ClientProxyEnc/Dec* and *ServerProxyEnc/Dec*, one performed by the client and one by the server.

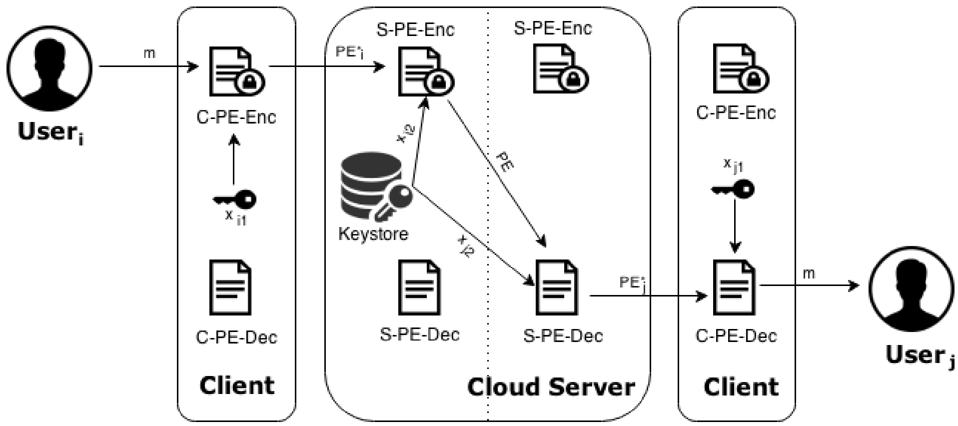

Figure 3.2: Elgamal proxy encryption scheme.

We shall now go through the description of *Proxy Encryption* algorithms implemented in CloudDB.

ProxyEnc

$\mathcal{PE}$-ClientEnc(x_{i1}, D): To encrypt a message D, user i chooses a random $r_D \leftarrow \mathbb{Z}_q$ and computes the ciphertext $PE_i^*(D) = (e_1', e_2')$ where $e_1' = g^{r_D}$ and $e_2' = g^{r_D x_{i1}} D$. The client finally passes $PE_i^*(D)$ to the server.

Algorithm 2: PE Client Encryption

$\mathcal{PE}$-ServerEnc($x_{i2}, PE_1^*(D)$): Before storing received encrypted data the proxy server computes the value $PE(D) = (e_1, e_2)$ where $e_1 = e_1'$ and $e_2 = e_2' \cdot (e_1')^{x_{i2}} = g^{r_D x} D$ with x the master secret key.

Algorithm 3: PE Server Encryption

ProxyDec

$\mathcal{PE}$-ServerDec($x_{j2}, PE(D)$): Before sending retrieved ciphertext to the client of user j, the server decrypts $PE(D) = (e_1, e_2)$ using the key x_{j2}. $PE_j^*(D) = (e_1', e_2')$ with $e_1' = e_1 = g^{r_D}$ and $e_2' = e_2 \cdot e_1^{-x_{j2}} = g^{r_D x_{j1}} D$. The server then sends $PE_j^*(D)$ to the client.

Algorithm 4: PE Server Decryption

$\mathcal{PE}$-ClientDec($x_{j1}, PE_j * (D)$): User j receives the incoming encrypted message $PE_j^*(D)$ and decrypts it as follws:
$e_2' \cdot e_1^{-x_{j1}} = g^{r_D x_{j1}} D \cdot (g^{r_D})^{-x_{1j}} = D$.

Algorithm 5: PE Client Decryption

3.2.3 Keyword Encryption

KeywordEncryption (KE) is employed, as already said, to support encrypted equality match on the cloud sever. It is used to perform read (i.e., search) operation on encrypted data. Encrypted match assumes that every search query is turned into a *Trapdoor* (TD) value and tested on KE values stored into the encrypted databse [18]. We show, in what follows, how KE works in CloudDB.

KeywordEnc

$\mathcal{KE}$-ClientEnc(x_{i1}, D): In order to create an SDE of data D the user i choses a random number $r_D \leftarrow \mathbb{Z}_q$ and $\sigma_D \leftarrow f_s(D)$ and computes the ciphertext $KE_i^*(D) = (c_1', c_2', c_3')$. We have that $c_1' = g^{r_D + \sigma_D}$, $c_2' = c_1'^{x_{i1}}$ and $c_3' = H(h^{r_D})$. The client then sends $KE_i^*(D)$ to the proxy server to be processed.

Algorithm 6: KE Client Encryption

$\mathcal{KE}$-ServerEnc($x_{i2}, KE_i^*(D)$): Before storing KE(D), to make data searchable, the server computes a round of keyword encryption on $KE_i^*(D)$. The server computes $KE = (c_1, c_2)$ where $c_1 = (c_1')^{x_{i2}} \cdot c_2' = c_1'^{x} = (g^{r_D + \sigma_D})^x = h^{r_D + \sigma_D}$ and $c_2 = c_3' = H(h^{r_D})$. Upon data insertion, $KE(D)$ is stored into the database along with the $PE(D)$.

Algorithm 7: KE Server Decryption

Both PE and KE algorithms are run during data insertion process and both ciphertext variants of a piece of data are stored into the database. Only *ProxyEnc/Dec* is run upon data retrieval instead.

3.3 Operations

In this section, it is described how data is written to the server, and so to the database, and how encrypted search flow works as well as how the *Match* algorithm is defined.

3.3.1 Data Writing

Suppose an user performs a **write** operation such as issuing an `INSERT` query (it could also be an `UPDATE`). The data being sent will obviously become searchable and retrievable therefore it is encrypted using both KE and PE encryption variants. Encryption is performed, as previously said, in two rounds performed by both the client and the server. The process to encrypt and store data is straightforward as shown in Figure 3.3

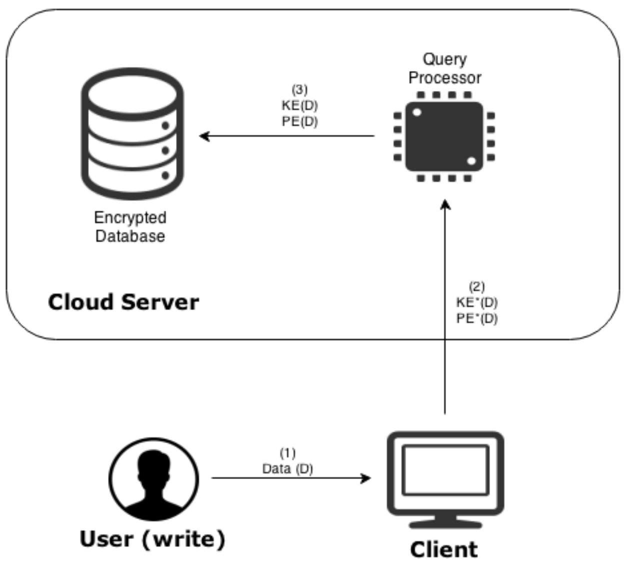

Figure 3.3: Data writing workflow.

For the sake of simplicity, it has been omitted the communication with keystore as well as the one with access management system, which are the same for every operation and can be seen in Figure 3.1. For every piece of information, the system stores both KE and PE ciphertexts. The reader will not fail to notice from previous section that the result of both PE and KE schemes is made up by two elements as $PE = (e_1, e_2)$ while $KE = (c_1, c_2)$. That is, for every information the database holds four fields.

3.3.2 Data Reading (Encrypted Search)

In order to perform a search over the encrypted data held into the database, a user has to compose (1) an SQL-like query (Q) and submit it to the client (e.g., `SELECT`). The latter handles the process of turning Q into a client-side trapdoor by running a round of TD (2), initially defined by Dong *et al.*[18] explained below. This trapdoor value is finally sent to the server. When the server obtains such a value it takes care of running the second round of *Trapdoor* producing the complete TD (3) value of Q ready to be tested on encrypted data. Equality test is accomplished running the *Match* algorithm which takes a KE and TD values as inputs and returns true if and only if there is a match between ciphertexts. The process is summarised in Figure 3.4

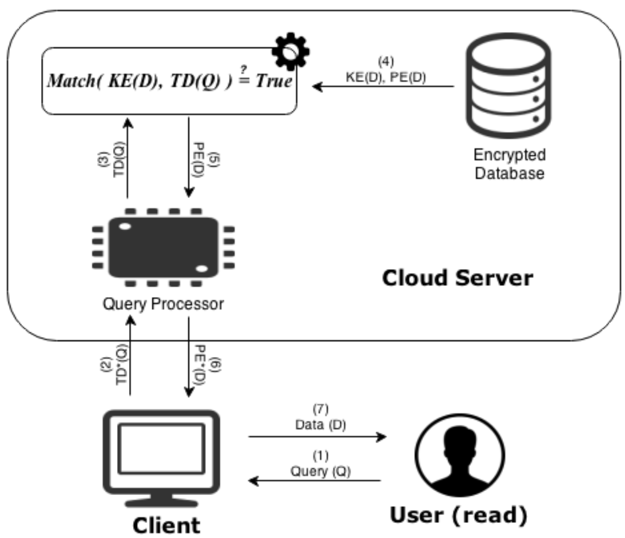

Figure 3.4: Data reading workflow.

We now move to describe how *Trapdoor* algorithm is defined. It is, as usual, split into two parts, one of which is run by the client and the other by the server.

Trapdoor Encryption

$\mathcal{TD}$-ClientEnc(x_{i1}, Q): User i issues the query Q to the client which, first of all, chooses a renadom number $r_Q \leftarrow \mathbb{Z}_q^*$ and computes $\sigma_Q = f_s(Q)$. Next, composes the message $TD_i^*(Q) = (t_1', t_2')$ where $t_1' = g^{\sigma_Q - r_Q}$ and $t_2' = h^{r_Q} \cdot g^{-x_{i1} r_Q} \cdot g^{x_{i1}\sigma_Q} = g^{x_{i2} r_Q} \cdot g^{x_{i1}\sigma_Q}$. $TD_i^*(Q)$ is finally sent to the proxy server for the second round of encryption.

Algorithm 8: TD Client Encryption

$\mathcal{TD}$-ServerEnc($x_{i2}, TD_i^*(Q)$): The server recieves $TD_i^*(Q) = (t_1', t_2')$ and computes the final ciphertext $T = t_2' \cdot (t_1')^{x_{i2}} = g^{x\sigma_Q} = h^{\sigma_Q}$.

Algorithm 9: TD Server Encryption

T is the final trapdoor value of Q ready to be testesd on encrypted data. The server now retrieves encrypted data from the encrypted database (4) and takes advantage of the $Match$ algorithm to test TD on KE ciphertexts. $Match$ is of straightforward implementation and is defined as follows:

Match

Match($KE(D), T$): This algorithm takes $T = h^{\sigma_Q}$ and $KE(D) = (c_1, c_2)$ as inputs where $c_1 = g^{xr_D + x\sigma_D}$, and $c_2 = H(h^{r_D})$ (see Algorithm 7) and checks whether $c_2 \stackrel{?}{=} H(c_1 \cdot T^{-1})$. If a match is found `true` is returned `false` otherwise. The most important role is here played by the keyed-hash function f_s, indeed if and only if $\sigma_Q = \sigma_D$.

Algorithm 10: Match Algorithm

If $Match$ returns `true` for a certain piece of data, the corresponding PE ciphertext is passed to the query processor (5) in order to perform *Server-ProxyDec* (see Section 3.2.2). Such a ciphertext is decrypted so as only the user i can fully decrypt it (see Figure 3.2). It is then sent to the client of user i to perform the second round of decryption (6) and the final result is then returned to user i (7).

Chapter 4

Implementation Details

We pass now to the description of CloudDB implementation, showing adopted technologies and approaches. It is made up by three main aplications which implement a *Client* a *Server* and the *Key Management Authority*. The model follows the *Client-Server* scheme, even though KMA is an external trustworthy entity that runs only when a new user is authorised to use the system.

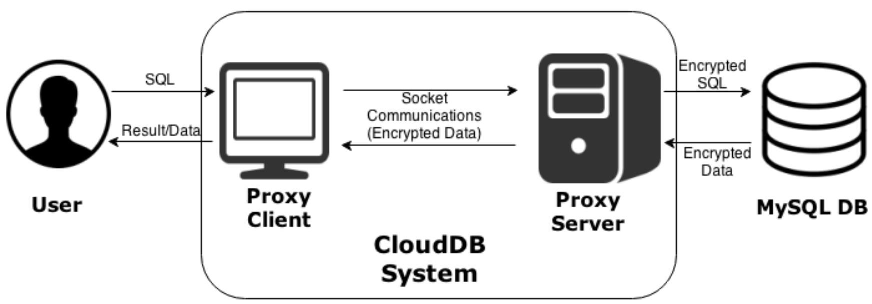

Figure 4.1: CloudDB architecture

An user should be aware of how SQL queries have to be written, naming conventions and encrypted databases limitations. A specific grammar and parser have been set up in order to accept SQL-like queries. To achieve this, Java-Cup along with JFlex are used.
A small snippet of the grammar, written to accept SQL-like `SELECT` queries, can be seen in Listing 4.1:

Listing 4.1: A grammar snippet.

```
query ::=  SELECT ALL FROM ID:table_name
                     {: RESULT = new Query("SELECT", table_name,
                        null, null, null); :}
                     |
                     SELECT id_list:columns FROM ID:table_name
                     {: RESULT = new Query("SELECT", table_name,
                        columns, null, null); :}
              ...
}
```

Consider the following `SELECT` query:

`Q4.0.1:   SELECT * FROM personnel;`

`Q4.0.1` matches the first query structure shown in the grammar of Listing 4.1. The parser creates and returns, therefore, a new `Query` java-object of type `"SELECT"` and `table_name=personnel`. Every query is intercepted, by the client application, and parsed in order to extract query type (e.g. `CREATE, INSERT`),SQL keywords and actual user data.

User data is encrypted applying the client round of both proxy and keyword encryptions (see Sections 3.2.2 and 3.2.3) and then sent to the server application, which applies the second round of both encryptions and executes the query, according to its type. We shall now describe how each kind of query is treated and evaluated through the system.

4.1 Create Table

A `CREATE` query allows table creation, specifying a table name and many column names. Table and column names are considered confidential data same as data contained in tables. This means that those names have to be encrypted as well. Encryption implies data expansion accordingly to the chosen security parameter. For a security parameter of k bits, the system permits $k/8$ characters long names. For instance, selecting $k = 1024$ the security parameter allows a user to provide names of 128 characters, as 1 character is stored in 1 byte.

PE and KE produce ciphertexts of length approximatively $\frac{k}{128} * 39$ characters. For example, choosing a security parameter of $k = 256$ means ciphertexts' length of about 78 characters. This becomes a problem when it comes to table creation because MySQL permits maximum table and colum names length of 64 characters. This means that it is impossible to use ciphertext as actual table or column name. There is no way to secure those names other than save their ciphertexts (PE and KE) in auxiliary tables and use unique references as actual names. The concept of `Table of Tables` and `Table of Columns` come to hand in this case.

4.1.1 Table of Tables

`Table of Tables` (ToT) is a simple table that stores table names' ciphertexts (PE and KE) coming from `CREATE` queries. Such a table is supposed to exist since system installation. The table belows shows a ToT containing six records.

ID	NamePE	NameKE
1	..3498..	..1682..
...	...	...
6	..9653..	..5311..

Table 4.1: Table of Tables structure

Each record stores information about a table name, this also means that in the encrypted database six tables exists. Lets consider the following `CREATE` query:

```
Q4.1.1: CREATE TABLE personnel(name, age, address);
```

Table name `personnel` is encrypted (for simplicity, table and column names treatment is split in two sections) and both PE and KE ciphertexts are stored in ToT, as shown in Table 4.2.

ID	NamePE	NameKE
1	..3498..	..1682..
...	...	...
6	..9653..	..5311..
7	..1762..	..4431..

Table 4.2: Table of Tables structure (after insertion)

The newly inserted record has `ID = 7` (highlighted in yellow), we use the conjunction of `tab+ID` as actual table name, in this case it will be `tab7`. `Q4.1.1` is thus transformed to:

```
Q4.1.2:   CREATE TABLE tab7(name, age, address);
```

In the next section, we explain the very similar treatment adopted to store column names before query execution.

4.1.2 Table of Columns

Data expansion affects also column names, therefore, we set up another auxiliary table called `Table of Columns` (ToC). Each record of ToC stores column name ciphertexts (PE and KE) and the belonging table id, as shown in Table 4.3. The number of records in ToC tells also how many columns there are in the database for each table.

ID	TableID	NamePE	NameKE
1	1	..3498..	..1682..
2	1	..5173..	..4493..
..	...	...	...
12	6	..7541..	..6359..

Table 4.3: Table of Columns structure

Column names `name, age` and `address` in `Q4.1.2` are encrypted and saved in ToC. In this case three rows are added to ToC as in Table 4.4 having `ID = 13,14` and `15` (highlighted in cyan).

ID	TableID	NamePE	NameKE
1	1	..3498..	..1682..
2	1	..5173..	..4493..
..	...	...	...
13	7	..2659..	..9925..
14	7	..1963..	..4330..
15	7	..9715..	..6181..

Table 4.4: Table of Columns structure

The conjunction of `col+ID` are used to compose unique column names. In this case column `name, age` and `address` are translated to `col13, col14` and `col15` respectively. `Q4.1.2` becomes therefore:

```
Q4.1.3:   CREATE TABLE tab7(col13, col14, col15);
```

Since for every future value we have to support both retrieval and search, `Q4.1.3` is not very correct because for every value, the system is supposed to be able to distighish between PE and KE ciphertexts. A more reliable version of `Q4.1.3` would be:

```
Q4.3: CREATE TABLE tab7(col13_PE, col13_KE,
                        col14_PE, col14_KE,
                        col15_PE, col15_KE);
```

After being executed `tab7` is created in the database and will be built as Table 4.5.

ID	col13_PE	col13_KE	col14_PE	col14_KE	col15_PE	col15_KE
...	...	...	...	...	...	...

Table 4.5: Example table

In the above table, the field `ID` is an unique record identifier.
In the following sections we will see how ToT and ToC are integrated with every operations.

4.2 Data Insertion

Upon every insertion, SQL queries are intercepted, table and column names are encrypted using TD, while the actual data to be inserted passes through both PE and KE schemes. Suppose to insert a record into the `personnel` table created before using the following query:

```
Q4.2.1: INSERT INTO personnel(name, age, address)
                    VALUES('Alice', 23, 'Copenhagen');
```

When the server receives the encrypted query, it checks whether the table exists in the database. To do this it looks for a match (see $Match$ algorithm, Section 3.3.2) in the ToT using the table name TD ciphertext. If a match is found, the field `ID` is retrieved and used, first, as a filter for column existence check and then for actual insertion. Column existence check is performed using column TD ciphertexts. If every column exists, then the server creates table and column names using unique identifiers. The new query will look like (for reading simplicity PE and KE fields are omitted):

```
Q4.2.2: INSERT INTO tab7(col13, col14, col15)
                    VALUES(PE/KE('Alice'),
                    PE/KE(23),
                    PE/KE('Copenhagen'));
```

`Q4.2.2` is executed by the DBMS and inserts the encrypted record in `tab7`.

4.3 Data Retrieval

Retrieving particular data from an encrypted database obviously requires some more steps to be performed compared to traditional databases. First of all, any `SELECT` query is intercepted, as usual, and table name, column names and policies are encrypted using TD (see also Section 3.3.2). We split the explanation in two parts, one for searching with conditions and one for searching without conditions.

4.3.1 Searching Without Conditions

Suppose we want to retrieve names and ages from the same `personnel` table we created before. To achieve this, we can submit the following `SELECT` query:

```
Q4.3.1:   SELECT name, age FROM personnel;
```

The client generates TD ciphertexts for column names `name` and `age` and table name `personnel`. The server applies the second round of TD encryption and then checks for table and column existence in ToT and ToC respectively. If both checks return `true`, then column name PE ciphertexts are retrieved, along with table and columns identifiers (`tab7`, `col13` and `col14` namely). `Q4.3.1` is thus translated to:

```
Q4.3.2:   SELECT col13, col14 FROM tab7;
```

Fields `col13` and `col14` PE ciphertexts are retrieved from every record in table `tab7`. After the server round of PE decryption, the result is passed to the client, which can now complete decryption and display data to the user.

4.3.2 Searching With Conditions

In the case we want to retrieve only certaing records, according to some specific conditions, we can express those conditions through the `WHERE` clause. As in traditional databases, the `WHERE` clause allows a user to fetch records containing specific values. A very simple example of `SELECT` query with `WHERE` clause would be:

```
Q4.3.3:   SELECT * FROM personnel WHERE name = 'Alice';
```

We will only show how `WHERE` clause is evaluated, without repeating the same things many times.
Both sides of condition of policy in query `Q4.3.3` are encrypted using TD. Once the correct table has been found (i.e. `tab7`), the server checks the existence of the column appearing in the condition (i.e. `col13`). If a match is found, the server loops over every row in `tab7` and applies $Match$ on KE ciphertext of field `col13` and TD of `'Alice'`. Every time $Match$ returns `true`, the current record is retrieved.

Suppose now to issue a more complex query, containing a more convoluted `WHERE` policy, for instance:

```
Q4.3.4: SELECT * FROM personnel
                 WHERE name = 'Alice'
                 OR (name = 'Bob' AND address = 'Madrid');
```

In `Q4.3.4` one can see also SQL conjunctions (i.e. `AND`) and disjunctions (i.e. `OR`). In this case, the system represents the policy as a tree, where every leaf contains a condition and every internal node is a conjunction or disjunction.

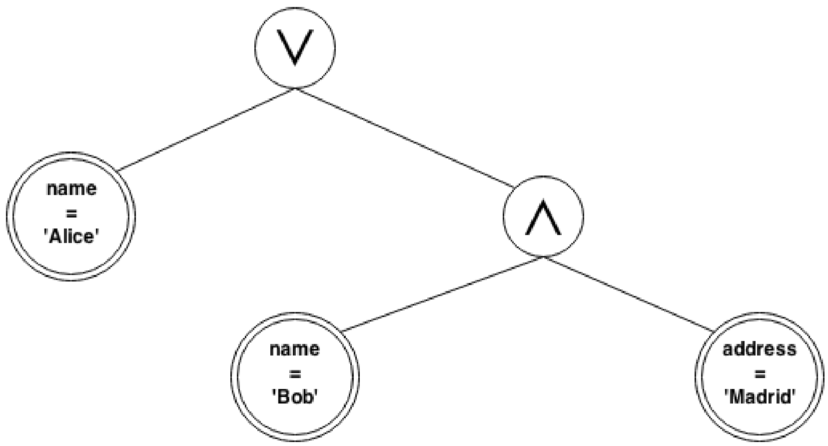

Figure 4.2: Example condition policy tree for Q4.3.4

The policy evalutaing algorithm follows the Deep-First-Search (**DFS**) paradigm, using also tree-pruning techniques. That is, in the tree in Figure 4.2 the root node is a disjunction, or else, if at least one of its child nodes is satisfied it results satisfied too. That is, if we find a satisfied child node, we can stop the algorithm, without inspecting the rest of the conditions.

Part III

Contributions

Chapter 5

Numerical Data

Every time, so far in this dissertation, we took in consideration a piece of data we always referred to `string` type as it was the only data-type supported by the system. The introduction of the numerical (`integer`) data type implies the fact that the system has to be revised in order to grant the full integration of such an information type with the pre-existing data and features. Traditional databases allow the storage of numerical data and provide some built-in function in order to perform computation over that data for example:

- `SUM()`
- `MAX()`
- `COUNT()`
- `AVG()`

This chapter is going through the description of how numerical data is treated as well as how range queries are evaluated on numerical attributes including also cryptographic techniques employed to support such processes.

5.1 Homomorphic Encryption

Homomorphic Encryption (**HE**) generates ciphertext that mantains certain properties as the corresponding plaintexts and on which it is possible to carry out some specific computations. The resulting ciphertext, once decrypted, matches the result of the same computation performed on plaintext values.

Consider an encryption scheme $\boldsymbol{\mathcal{E}}$ and a binary operation $*$ between two numerical values. $\boldsymbol{\mathcal{E}}$ is homomorphic with respect for $*$ if holds:

$$\boldsymbol{\mathcal{E}}(x_1) * \boldsymbol{\mathcal{E}}(x_2) = \boldsymbol{\mathcal{E}}(x_1 * x_2)$$

It is clear that decrypting the result using the corresponding decryption algorithm $\boldsymbol{\mathcal{D}}$ along with the correct key we obtain:

$$\boldsymbol{\mathcal{D}}(\boldsymbol{\mathcal{E}}(x_1 * x_2)) = (x_1 * x_2)$$

A very straightforward example of partially homomorphic encryption scheme is **RSA** [21]. It is a very popular asymmetric algorithm which bases its security on the factorization problem. Suppose e and d the encryption and decryption exponent used in RSA respectively where $ed \equiv 1 \ (mod\ \phi(n))$ (modular inverse relation) and n is the product of two large prime numbers. We have that RSA is homomorphic with respect for the product function:

$$\boldsymbol{\mathcal{E}}(x_1) \cdot \boldsymbol{\mathcal{E}}(x_2) = x_1^e \cdot x_2^e = (x_1 x_2)^e = \boldsymbol{\mathcal{E}}(x_1 x_2)$$

RSA scheme can, therefore, be employed to perform multiplication on encrypted values.

Our aim was first to support aggreate queries like `SUM()`, where only addition is involved as a computation. We solved this by using the Paillier cryptosystem [10]. It is a probabilistic additive homomorphic cryptosystem based on Decisional Composite Residuosity Assumption (**DCRA**) or else given a composite n and an integer z is hard to decide if z is a nth root modulo n^2, i.e., whether there exists an integer y such that $z \equiv y^n mod\ n^2$. That is, given two ciphertexts c_1 and c_2, corresponding to two plaintext messages m_1 and m_2 namely, it is possible to compute the encryption of $m_1 + m_2$ applying a certain function $f(c_1, c_2)$. It is meant to be used to encrypt numerical data as far as numerical computations on string values, in most cases, do not make much sense. It is used for data retrieval instad of PE in case of numerical data. Here, we describe the Paillier cryptosystem adapted to CloudDB client-server scheme. Again, two round of encryption and decryption are needed ($ClientHomomorphicEnc/Dec$ and $ServerHomomorphicEnc/Dec$). Furthermore, such a cryptosystem works under other assumptions, or rather it needs different public parameters than the other encryption schemes (PE, KE and TD) employed. For these reasons we adjusted it to work for a client server model, which is CloudDB, and using the same key distribution. First of all we, shall describe the public parameters published by the KMA in order to have a clear idea of how the algorithms work. This algorithm is run inside the same *Init(k)* defined above for KMA boot.

***HomomorphicInit(k')*:** takes as input the security parameter $k' = k/4$ and computes $n = p \cdot q$, where p and q are two prime numbers of bit length k'. It then chooses an integer g of order $(p-1)(q-1)/2 = \phi(n)/2$ by computing $g = a^{2n^2}$, where a is a random integer from $\mathbb{Z}^*_{n^2/2}$. The algorithm then outputs the tuple (n, g).

Algorithm 11: HE Public Parameters Generation

The output of *HomomorphicInit* takes now part of the final set of public parameters publicised by *Init* (Algorithm 1), which becomes

$$PubParams = (\mathbb{G}, g, q, h, f_s, H, homoN, homoG).$$

In the definition, we omit the fact that every computation is modulo n^2. However, in the following definitions, n and g will refer to *homoN* and *homoG* parameters, respectively

HomomorphicEnc

$\mathcal{HE}$-ClientEnc(x_{i1}, D): Given an integer D, user i first chooses a random number $r_D \leftarrow [1 \ldots n/4]$, next it computes $HE_i^*(D) = (e'_1, e'_2)$, where $e'_1 = g^{rD}$ and $e'_2 = g^{x_{i1}rD}(1 + Dn)$. $HE_i^*(D)$ is then sent to the server.

Algorithm 12: HE Client Encryption

$\mathcal{HE}$-ServerEnc($x_{i2}, HE_i^*(D)$): The server re-encrypts the incoming ciphertext by computing $HE(D) = (e_1, e_2)$ where $e_1 = e'_1 = g^{rD}$ and $e_2 = e_1^{x_{i2}} \cdot e'_2 = g^{rDx_{i2}} g^{rDx_{i1}}(1 + Dn) = h^{rD}(1 + Dn)$. $HE(D)$ is then stored.

Algorithm 13: HE Server Encryption

HomomorphicDec

$\mathcal{HE}$-ServerDec($x_{j2}, HE(D)$): Once $HE(D)$ is retrieved the server decrypts it, with respet for a user j, to $HE_j * (D) = (e'_1, e'_2)$. It sets $e'_1 = e_1 = g^{rD}$ and computes $e'_2 = e_2 \cdot e_1^{-x_{j2}} = g^{x_{j1}rD}(1 + Dn)$. The server sends the ciphertext back to the client of user j.

Algorithm 14: HE Server Decryption

$\mathcal{HE}$**-ClientDec(**$x_{j1}, HE_j^*(D)$**):** User j decrypts the received ciphertext by computing $\lambda = e_2' \cdot (e_1')^{-x_{j1}} = (1 + Dn)$. To get data D, it calculates $(\lambda - 1)/n = D$.

Algorithm 15: HE Client Decryption

Paillier cryptosystem defines the encryption function as additively homomorphic by performing the product between two ciphertexts. This means that the product of two ciphertexts decrypts to the sum of the corresponding plaintexts. We shall now describe how homomorphic sum is defined in our system.

HomomorphicSum

$\mathcal{HE}$**-Sum(**$HE(D_1), HE(D_2)$**):** The product between encrypted values $HE(D_1) = (e_1, e_2)$ and $HE(D_2) = (e_3, e_4)$ is performed computing $HE(D) = (e_5, e_6)$ where $e_5 = e_1 \cdot e_3 = g^{rD_1 + rD_2}$ and $e_2 \cdot e_4 = g^{x(rD_1 + rD_2)}((1 + D_1 n) \cdot (1 + D_2 n)) = g^{x(rD_1 + rD_2)}(1 + D_1 n + D_2 n + D_1 D_2 n^2)$.

Algorithm 16: HE Sum

Applying both rounds of homomorphic decryption on the resulting ciphertext $HE(D)$, we get back the value $D_1 + D_2$ as from the last computation in $\mathcal{HE}$**-ClientDec** we have:

$$D_1 + D_2 + D_1 D_2 n \ mod(n) \equiv D_1 + D_2 \ mod(n)$$

where $n \in \bar{0}$ and $\bar{a}$ denotes the residue class of an integer a in $\mathbb{Z}/n\mathbb{Z}$ [22]. One can easily point out that this encryption scheme can also support encrypted multiplication. To achieve this, the server has obviously to raise a ciphertext to the power of another ciphertext. Anyway, this would take very long based on the size of the security paramter.

5.2 Input Limitations

For a generic Paillier cryptosystem implementation, with a security parameter k, we have a public parameter $n = p \cdot q$, where p and q are two prime numbers of k bits and n is obviously $2k$ bits long. $2k$ is also the maximum input length, in terms of bits of a number. In our implementation, we adapted this cryptosystem to enable the usage of the same key pairs it uses for other encryption schemes (i.e., PE, KE and TD). In our implementation, k is reduced (only for homomorphic encryption) four times so as to have

$k' = k/4$. This implies that the homomorphic public parameter n will have bit length of $2 \cdot (k/4) = k/2$. Therefore, $\mathbf{k/2}$ will be the maximum length, always in terms of number of bits, of an numeric input

5.3 Table Creation

A user can express the will to make a certain column contain numerical values at table creation time. The grammar (see beginning of Chapter 4) has been extended in order to accept a mandatory parameter after the column's name, which defines the data type stored in it. A user can use the keyword `integer(k)` to create a numerical column, which will be stored as the couple (HE, KE). HE is used instead of PE for data retrieval while KE supports search as usual. k stands for the representation bit number of data stored in a numerical column. This supplementary information is fundamental in order to evaluate range policies on numerical data.
The following query can be considered the newest version of Query `Q4.1.1`:

```
Q5.3.1: CREATE TABLE personnel(name string,
                                age integer(3),
                                address string);
```

As the reader can see, every column name is now followed by a data type. Those types will be stored in ToC as additional information. ToC will then look like:

ID	**TableID**	**NamePE**	**NameKE**	**Type**
1	1	..3498..	..1682..	0
2	1	..5173..	..4493..	7
..	...	...	...	...
13	7	..2659..	..9925..	0
14	7	..1963..	..4330..	3
15	7	..9715..	..6181..	0

Table 5.1: Table of Columns Structure (Type added).

The `Type` field contains the number of bits of representation of data stored in a column. For example, in ToC represented Table 5.1, the `age` column, or else, the column having `ID` set to `14`, is a numerical column that contains numbers representable on 3 bits. Columns having `Type` set to `0` are

string columns.
As explained, numerical values are encrypted using HE, for what concerns data retrieval, and KE for data search. After being translated (see Section 4.1), `Q5.3.1` becomes:

```
Q5.3.2: CREATE TABLE tab7(col13_PE, col13_KE,
                          col14_HE, col14_KE,
                          col15_PE, col15_KE);
```

`Q5.3.2` is then executed by the DBMS creating the table:

ID	col13_PE	col13_KE	col14_HE	col14_KE	col15_PE	col15_KE
...	...	...	...	...	...	...

Table 5.2: Personnel Table with Numerical Column.

5.4 Data Insertion

Upon data insertion, the client does not know whether to encrypt a certain value using PE or HE for data retireval because every information is only mantained by the server. This means that an extra communication is required between client and server for every data insertion, as shown in Figure 5.1. The client learns, therefore, data types and perform encryptions consequently.

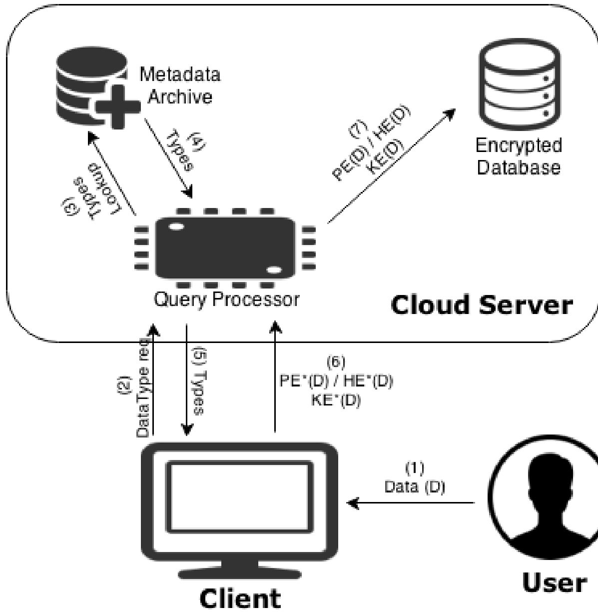

Figure 5.1: Multiple data types insertion workflow.

A naive solution would create both PE and HE ciphertexts despite the data-type and the server then discards the useless ones.

5.4.1 Bag of Bits

Whenever an integer value is about to be stored on the cloud, the client transforms it into its binary representation on **k** bits and creates the bag of bit structure for it. The bag of bits is a set of **k** ciphertexts each of those generated from the KE encryption of strings of length **k**. Every string is made up by a bit of the binary representation of the value and filled with $*$ cahracters.
Suppose we want to insert the value 5 in a column that represents numbers on $k = 3$ bits. The client creates k KE ciphertexts as follows:

- KE($1 * *$)
- KE($*0*$)
- KE($* * 1$)

having 101 the 3-bits binary representation of 5. This set is then sent to the server to perform the second round of KE encryption on each element. Every column has a separate bag of bit table, in order to shorten and the new set of entries is attached to the correct bag of bits table. Consider the following `INSERT` query:

```
INSERT INTO personnel(age) VALUES(5);
```

Every bag of bits table is uniquely named within the database using the original table an column identificators. Consider the previous example where `personnel` table is identified by the id $tab7$ and the `age` colum by the id $col14$ the bag of bits table for this column will be called *tab7_col14_bags* and is made up by the two fields required in order to store the KE ciphertext, an unique identifier and the id of the actual numeric value in the original table. The above query is lastly transformed in the following queries:

```
1.  INSERT INTO tab_7(col14_HE, col14_KE) VAL-
            UES(HE(5), KE(5));


2. INSERT INTO tab7_col14_bags(idVal, bag)
              VALUES(id, HE(1**)),
              (id, HE(*0*)), (id, HE(**1));
```

and submitted to the underlying MySql DBMS. Query no. **1** stores HE and KE ciphertexts into table *tab*_7. Furthermore the id of the newly inserted record is returned and saved in the id variable. Query no. **2** stores the entire bag of bits of 5 into table *tab7_col14_bags* along with the id of the actual corresponding value in *tab*_7. Bag of bits structure is used to evaluate range queries over encrypted data as described in the nex section.

5.5 Data Retrieval

An user that wants to query the cloud database to retrieve particular information can act as usual. When a search query is submitted the server retrieves data type informations for every column from the Metadata Archive and thus apply the correct decryption algorithm as server-side round. Equality constraints are carryed out normally comparing TD ciphertext, taken from the policy, and every KE ciphertext, retrieved from the database, using the $Match$ algorithm.

5.5.1 Evaluating Range Queries

A more detailed explanation is required when it comes to evaluating range queries. In this case, the bag of bit apporach comes into play. First of all, the user is expected to specify the representation bit length (the same k defined when table was created) for every column appearing in the `WHERE` clause. This way the system can meaningfully compare TD values in the policy with KE values belonging to bag of bits of a certain column.
For example:

```
Q1:  SELECT * FROM personnel WHERE age > 4#3;
```

The value 3 is supposed to be the same value given to k, stored under the field `Type` of ToC (see Section 5.3), when the `personnel` table was created. The client can, therefore, retrieve the value of k splitting `4#3` on `#` an then build corresponding condition tree. We know that the numbers bigger than 4 representable on 3 bits are:

- 5 = `101`
- 6 = `110`
- 7 = `111`

This means that a satisfying number should be represented on 3 bits having the first bit set to `1` and either the second or the third set to `1`. The strings being constructed are hence $1**$, $*1*$ and $**1$. Once the server round of TD has been applied the condition tree will look like Figure 5.2

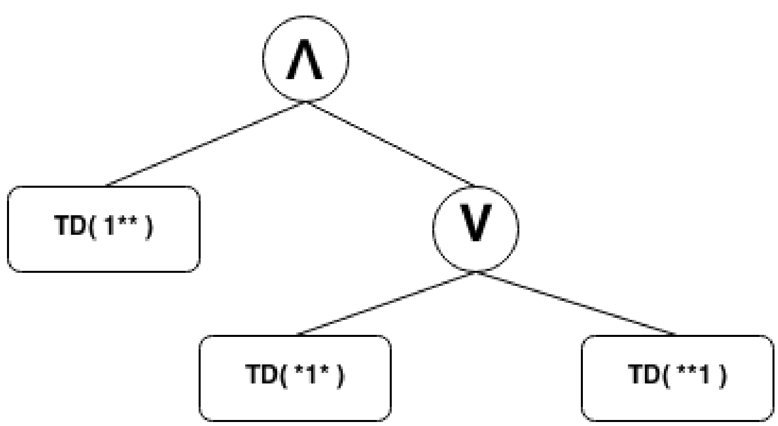

Figure 5.2: Example range query condition tree.

The server loads the bags of bits of numerical values, stored in `personnel`, in a customised data structure based on `HashMaps`. Every bag of bits table (i.e., the bag of bits of each column) is stored in an HashMap referenced, or else having values ids as keys.

col14	id1	TD(1**)
		TD(*0*)
		TD(**1)
	id2	TD(1**)
		TD(*0*)
		TD(**0)
	id3	TD(1**)
		TD(*1*)
		TD(**1)

Table 5.3: Column bag of bits HashMap.

Table 5.3 represents a possible `HashMap` java object for column $col14$ (i.e. `age` column as assumed above) that contains bag of bits of three numerical values 5, 4, 7 namely. This is done for every numrical column considered in a query. The set of hashmaps containing value of each column are collected in a bigger `HashMap` where elements are accessible by column names as shown in Table 5.4.

tab7	col14	HashMapCol13

Table 5.4: Table bag of bits hashmap.

In Table 5.4 HashMapCol14 is the structure shown in Table 5.3. `tab7` has only one numerical column, i.e., `age`, the only entry in Table 5.4. Accessing a particular bag of bits using such a data structure becomes relatively fast, as the worst-case access time will become $O(n)$, where n is the number of columns belonging to a certain table, in the case `HashMap` degenerates into a linked list. This holds for insertion and deletions as well.

Proxy server loads the whole content of the table `personnel` into a `ResultSet` java object including unique ids of each record.
Looping over `ResultSet`, the server retrieves ids of records one by one, and column `ID` from the encrypted policy. This way it can access the bag of bit of corresponding value in the current record. Once the correct bag of bit

has been loaded, the server parses the condition tree and checks whether an element of the bag satisfies the current tree-node. If the root-node of the condition tree is satisfied `true` is returned, `false` otherwise. If `true` is returned, then the current record satisfies the condition tree and it is ready be decrypted. In the case of a bag of bits like the one in Table 5.3 and query `Q1` records having id 1 and 3 satysfy the condition tree.

Range policies containing `>= N` (greater-or-equal than) and `<= N` (less-or-equal than) conditions (always for numerical attributes) are easily supported as we can translate such comparison operator to `> N-1` and `< N+1` namely. In fact, the following query:

```
SELECT * FROM personnel WHERE age >= 5#3;
```

and query `Q1` are evaluated equally.

KE ciphertexts are stored along HE thus numerical equalities are evaluated as usual using the $Match$ algorithm.

5.6 Homomorphic Cryptography Performances

As previously said, HE substituted PE when it comes to the storage of numerical data. We tested our Paillier Cryptosystem adaption and implementation against pre-existing PE algorithm. In what follows we tested HE encryption, decryption and sum mechanisms. The reader shoud be aware of the fact that encryption and decryption measurements are not performed on both client and server separately, but as a unique process instead.
Every test is based on the time taken, by each method, performing 100 iterations.

Hardware Specifications
Processor 1,8 GHz Intel Core i5
Memory 4 GB 1600 MHz DDR3

5.6.1 Homomorphic Encryption Performances

The following chart shows how HE behaves, with respect for PE, in function of the security parameter size variation.

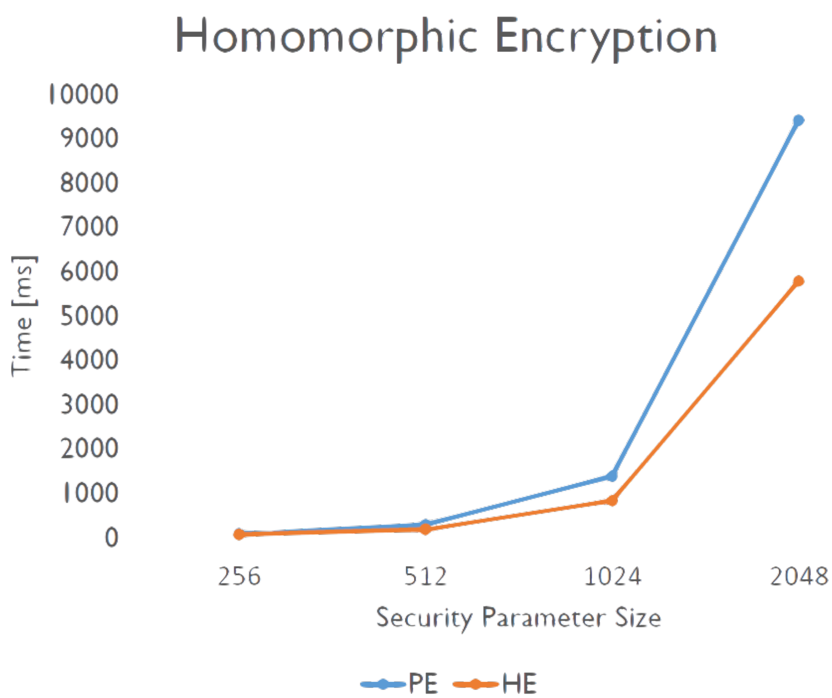

Figure 5.3: Homomorphic Encryption Performances.

Figure 5.3 shows that HE results to be meaningfully faster than PE, mainly when the security parameter size becomes sufficiently large.
We run the full encryption of the number `71` a hundred times for every value of the security parameter size.

5.6.2 Homomorphic Decryption Performances

Decryption measurements have been carried out on pre-computed ciphertexts, corresponding to the plaintext number `71`. For every security parameter size value, both *Proxy Decryption* and *Homomorphic Decryption* run a hundred times each.

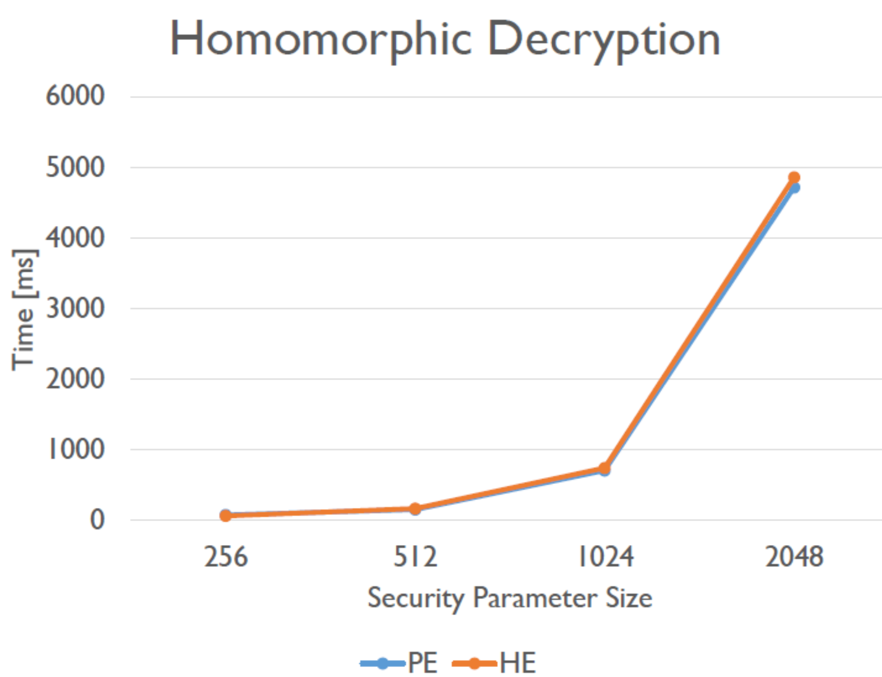

Figure 5.4: Homomorphic Decryption Performances.

Figure 5.4 shows that there is a minimal gap betweenthe two lines, but a careful observation shows that *Proxy Decryption* is slightly faster than *Homomorphic Encryption* .This is due to the fact that, the two schemes do exactly the same operations on server side, but on client side HE becomes a bit more complicated (see Algorithm 15). A more accurate and heavy test could actually clarify which performs better. Anyway, HE significantly limits the input length, for this reason PE is still used to protect `string` typed values.

5.6.3 Homomorphic Sum Performances

Algorithm 16 is a very fast tool provided to the server in order to compute encrypted sum. The server can in fact easily compute the modular product of two ciphertexts, that will result in plaintext values' sum.

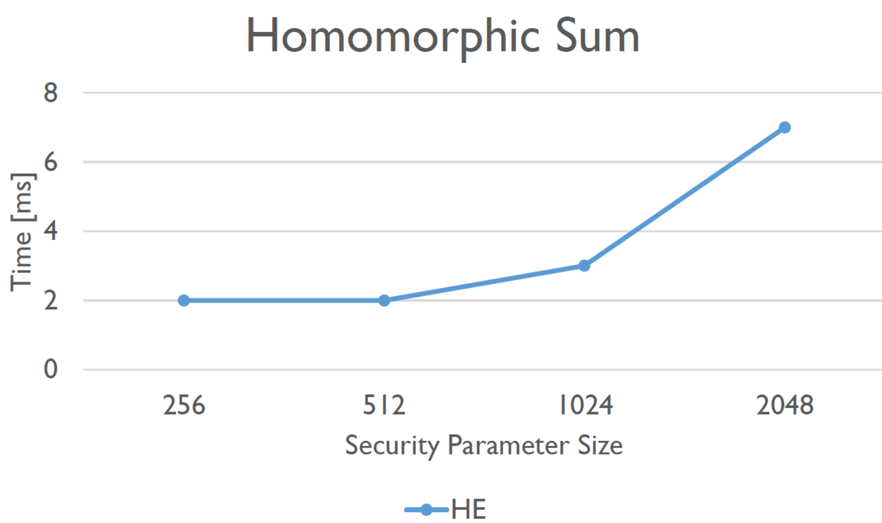

Figure 5.5: Homomorphic Sum Performances.

As Figure 5.5 shows the sum procedure is fast regardless to the size of the security parameter.
As a conclusion we could say that such an implementation, of the Paillier Cryptosystem, is relatively fast, with respect for the PE scheme. Moreover, it allows fast encrypted sum computations.

Chapter 6

Supporting Joins

Another important topic of this thesis is how joins are supported on encrypted outsourced databased. The most important thing that one has to bear in mind is that every value is stored on the cloud as the couple `PE/HE,KE`. This means that every value is encrypted introducing randomness (see Section 3.2). The *Match* algorithm defined in Algorithm 10 becomes useless because in case of join the server has to compare a record against another record. That is, we no longer have TD ciphertext as input for *Match*, we only have KE ciphertexts that somehow support encrypted match. The proposed solution is taking advantage of the underlying DBMS join system but at the same time it leaks equality information about stored values. Anyway it does not compromise data confidentiality as its security is the same as for *Trapdoor* encryption scheme see [18] for the proof.

6.1 Table Creation

The main idea is to use the DBMS join system in order to support joins. To do this, we store TD ciphertexts, along with PE or HE, instead of KE in order to no longer introduce randomness. This results in a trivial string equality match when it comes to encrypted match or record comparison (join). Furthermore, only one field in the actual storage is required to store a TD ciphertext, optimising space consumption. This happens only for columns that are supposed to take part in a join process.
Of course, the system has to know in advance which column are joinable since table creation. For this purpose `JOINABLE` has been introduced as a

new keyword in the grammar. A `CREATE` query now becomes:

```
Q6.1.1: CREATE TABLE personnel(name string JOINABLE,
                               age integer(7) JOINABLE,
                               address string);
```

The above query specifies a table named `personnel` two string columns `address` and `name`, where the latter is also `JOINABLE` and an integer column `age`, again, `JOINABLE`. Suppose again that `personnel` table is identified by $tab7$ and $col13, col14$ and $col15$ identify name, address and age respectively. The table will become on the cloud:

col13_PE	col13_TD	col14_HE	col14_TD	col15_PE	col15_KE
...	...	...	...	...	...

Table 6.1: Example table with joinable columns.

Hilighted text in Table 6.1 show how joinable columns are stored after execution of Query `Q6.1.1`. A new boolean (`0` or `1`) field has been added to the Table of Columns which stores information about whether a column is joinable or not. This information is only held by the server. ToC looks then like:

ID	**TableID**	**NamePE**	**NameKE**	**Type**	**Joinable**
1	1	..3498..	..1682..	0	1
2	1	..5173..	..4493..	7	0
..	...	...	...	...	...
13	7	..2659..	..9925..	0	1
14	7	..1963..	..4330..	3	1
15	7	..9715..	..6181..	0	0

Table 6.2: Table of Columns Structure (Joinable added).

The example content ot ToC in Table 6.2 shows how it is updated after execution of Query `Q6.1.1`. The `Joinable` field has been added and joinable column are hilighted in yellow. ToT is not affected by this new feature.

6.2 Data Insertion

Every time an user issues an `INSERT` query the client needs an extra communication with the server in order to retrieve information about columns. To achieve this the client uses the same socket communication used to retrieve information about data type shown in Figure 5.1. The server has been enabled to send back data types as welll as joinable information.
Doing so, the client can now apply the correct encryption scheme for both data retrieval (PE or HE) and search (KE or TD). Consider the following `INSERT` query:

```
Q6.2.1: INSERT INTOpersonnel(name, age, address)
                VALUES('Alice', 5, 'Copenhagen');
```

After querying the server for data types and joinability of columns, the client encrypts Query `Q6.2.1` to:

```
Q6.2.2: INSERT INTO tab7(col13_PE, col13_TD,
                     col14_HE, col14_TD, col15_PE, col15_KE)
                     VALUES(PE('Alice'), TD('Alice'),
                     HE(5), TD(5),
                     PE('Copenhagen'), TD('Copenhagen'));
```

Query `Q6.2.2` is then executed, inserting the encrypted record into `tab7` (i.e., `personnel`) shown in Table 6.1. In the remaining sections we explain data retrieval using the `JOIN` clause.

6.3 Cross Join

In this section is described how `SELECT` queries containing `JOIN` clause are evaluated by CloudDB. *Cross Join* is the simplest join scenario as it returns the cartesian product (see Section 1.1) of the records of tables in the join. For a join between two tables it combines each row of the first table with each row in the second table. An example of the syntax for a cross join query looks like:

```
Q6.3.1:   SELECT * FROM personnel JOIN salary;
```

or simply listing tables using a comma as a separator:

```
SELECT * FROM personnel, salary;
```

Suppose we want to submit Query Q6.3.1 to CloudDB. After two rounds of TD encryption the proxy server checks whether the two tables exist in the database by looking them up in the Table of Tables. If both tables are found the query is then translated to:

```
Q6.3.2:   SELECT * FROM tab7 JOIN tab4;
```

where tab7 and tab4 identify personnel and salary tables namely. This query is submitted to the DBMS and The result is treated the same way as the result of a traditional SELECT query.
From now on, in our examples, we assume that col8 and col9 are the identifiers of columns name and amount of table tab4 (salary).

6.3.1 Selecting Fields

To select only certain fields from the result of a join it is necessary to specify them after the SELECT clause. Of course this will generate a conflict if, for instance, two tables share the name of a field, and we want to select both fields. An example of a conflicting selection would be:

```
SELECT name, name FROM personnel JOIN salary;
```

where both personnel and salary tables have a field called name. To overcome this confusion, one has to specify the table from which the field has to be taken, like it is used to do for traditional SQL. Table name has to be prepended to column name with a dot character like table.column so a correct JOIN query would look like:

```
Q6.3.3: SELECT personnel.age, salary.name
               FROM personnel JOIN salary;
```

However, this implies some more checks on the server side as first it has to check whether both tables exist, then check if they actually have the specified fields. After both round of TD encryption Query Q6.3.3 will become:

```
Q6.3.4: SELECT tab7.col14_HE, tab4.col8_PE FROM tab7 JOIN tab4;
```

and then executed by the DBMS.

6.4 Join Using ON Clause

SQLs provide also the `ON` clause to be used along with `JOIN`. It is a very powerful tool to combine records which have a particular field containing the same value or having certain properties. The `ON` clause makes the join return a filtered combination of records. Suppose we want to join only records that contains the same value for a certain field. Again, here it is necessary to specify both table name and column name in order to avoid name conflicts. An issued query should look like:

```
Q6.4.1: SELECT * FROM personnel JOIN salary
                      ON personnel.name = salary.name;
```

In this case, the server can correctly distinguish between `name` column of `personnel` and `salary` table. The join hereby happens on records that store the same value for the field `name` in both tables. The server splits both side of the filter (`personnel.id = salary.id`) on the dot so as to retrieve table and column names. It then checks existence of tables and related filter columns. A further check is then made on columns as they must be joinable (i.e., searchable component stored as TD). If every check is successfull `Q6.4.1` is translated using unique table and column identifiers:

```
Q6.4.2: SELECT * FROM tab7 JOIN tab4
                      ON tab7.col13_TD = tab4.col8_TD
```

and submitted to the MySql DBMS which can easily evaluate the equality between *Trapdoor* ciphertexts and return the set of recods. An user can also specify a list of fields to retrieve in the `SELECT` clause. The result is treated as usual, or else it is partially decrypted by the server and then sent back to the client.

6.5 Join Along With Where Clause

Derived from the fact that `JOIN` can be considered an enlargement of `SELECT` queries, one can think about filtering the result of a join. This can be normally achieved by adding condition policies using `WEHRE` clause. Again, it is necessary to specify table names inside condition because we are considering more than one table. Consider the following `SELECT` query:

```
Q6.5.1: SELECT personnel.address
               FROM personnel JOIN salary
               ON personnel.name = salary.name
               WHERE salary.amount <= 135#10
               AND personnel.age > 21#7;
```

In this case, the resulting set of combined records is not directly sent to the client but is filtered by policy evaluation. The policy expresses a constraint on two numerical columns belonging to `salary` and `personnel` tables. This has not been chosen casually. The case of a numerical policy helps the reader to better understand the *bag of bits* structure. Here we need two bag of bits hashmaps, one per table like Table 5.4. Anyway, it is useful merge every bag of bit in a single hashmap referenced by column names as usual. To overcome name conflicts, it is enough to use the conjunction of table and column unique identifiers as keys.

JoinResult	tab7col13	HashMapTab7Col14
	tab4col8	HashMapTab4Col8
	tab4col9	HashMapTab4Col9

Table 6.3: Table bag of bits hashmap for query Q6.5.1

However, another problem comes up here. As shown in Table 5.3 every bag of bits is referenced by the id of the record in the actual table. This means that if two table are joined and a policy like in `Q6.5.1` exists we have also to retrieve two sets of record ids. `Q6.5.1` is translated like follows before being executed:

```
Q6.5.2: SELECT tab7.idVal AS tab4_idVal,
               tab4.idVal AS tab2_idVal, tab7.col15_PE
               FROM tab7 JOIN tab4
               ON tab7.col13_TD = tab4.col8;
```

The result is loaded in a `ResultSet` java object. For every record both conditions of `WHERE` policy are evaluated retrieving the correct bag of bits.

The server first retrieves the hashmap containing bag of bits related to a certain column, e.g., for the first condition `salary.amount` the key `tab4col8` si used. Next the server retrieves the correct set of KE ciphertexts making up the bag of bits of the value in current record. This is achieved by quering the just retreived hashmap using the correct record id, that's why record ids of both tables are required. In the case of the first condition, the server has to retrieve the record id of table `salary`, thus taking value stored in field `tab4_idVal`. Every time a record satisfies the entire policy tree, is added to the set of records ready to be decrypted and given to the client as a query response.

6.6 Join Performances

We shortly describe a relatively small join test performed on our system. We measured execution time of joining two tables, both made up by a hundred records, in function of the security parameter size. Our approach leaks, to the server, equality information about records, but it reveals to be fast in terms of record comparison. This is due to the fact that we use the DBMS join mechanism (e.g., for optimised set products). Moreover, it introduces a very small overhead due to the process of fetching tables and columns information, query translation and preparation, data loading and TD ciphertext comparison.
Performances of cross join between, for instance, two tables T_1 and T_2 of three and two columns respectively and both made up by a hundred record, are comparable to the performances obtained retrieving a full table T made up by five columns and composed by `10.000` records, of course up to the overhead due to the length of TD ciphertexts to be compared.
In this small test we run the following query, varying the security parameter size:

```
Q6.6.1: SELECT * FROM personnel JOIN salary
                      ON personnel.id = salary.userid;
```

For each value of the security parameter value size, the query has been run 50 times, and the average time is reported in the following table, inclusive of join overhead.

Security Parameter	Time [s]	Overhead [ms]
256	0.349154	17
512	0.968221	21
1024	4.136385	31
2048	27.22329	71
4096	207.348315	186

Table 6.4: Average execution time joining T_1 and T_2.

The reader should be aware that times in Table 6.4 are inclusive of decryption procedures and graphical rendering times.
The following chart graphically shows the scalability trend of the join mechanism in function of the security parameter size.

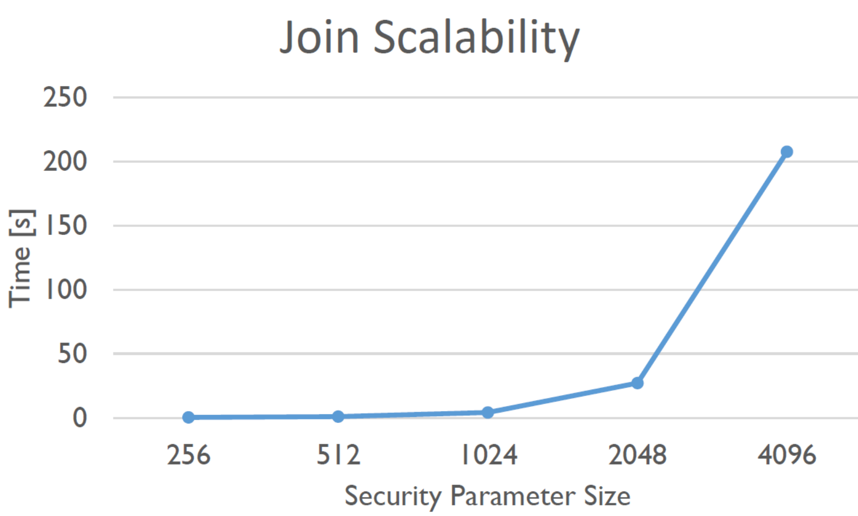

Figure 6.1: Join Scalability Trend.

`Q6.6.1` produces an output table T of dimension 100×5 (rows by columns). It seems to take very long to join two datasets encrypted using a security parameter size of `4096`. We then created a new table T_3 (of dimension 100×5) that reflects exactly the two joined tables, containing the same data, and realised that retrieving whole data from T_3 takes very few less than the join. This means that the overhead reported in Table 6.4 is the time needed, to the server, to perform the actual join (It is not shown in Figure 6.1 as it would be impossible to see).

Part IV

Conclusions

Chapter 7

Conclusions and Future Works

This thesis proposes a possible solution to the problem of supporting joins between encrypted tables in outsourced databases while protecting data. Furthermore, it proposes also a solution to process range queries on numerical data, including the possibility to perform addition operation between encrypted values, even though latter has not been implemented yet.

Numerical values are encrypted using an homomorphic encryption scheme to protect data. This allows the server to carry out addition operations without even knowing neither addends nor the result. The bag of bits structure has been defined in order to achieve range policy evaluation. Next step could be the implementation of aggregate functions based on addition (e.g., `COUNT` and `SUM`).

Joins are supported by storing deterministic ciphertexts of values in joinable columns. This way the system can take advantage of the DBMS's join mechanism to perform optimised joining between two or more tables. This discloses, by the way, equality information of ciphertexts but still ensures data confidentiality.

Bibliography

[1] Dan Boneh, Rafail Ostrovsky, Giovanni Di Crescenzo, Giuseppe Persiano. *Public Key Encryption with keyword Search*. 2004.

[2] Dawn Xiaodong Song, David Wagner, Adrian Perrig. *Practical Techniques for Searches on Encrypted Data.*

[3] Eu-Jin Goh. *Secure Indexes*. 2004.

[4] Yan-Cheng Chang, Michael Mitzenmacher. *Privacy Preserving Keyword Searches on Remote Encrypted Data*. 2005.

[5] Seny Kamara, Charalampos Papamanthou, Tom Roeder. *Dynamic searchable symmetric encryption*. 2012.

[6] Tracey Raybourn. *Bucketization Techniques for Encrypted Databases: Quantifying the Impact of Query Distributions*. 2013.

[7] Hakan Hacigumus, Bala Iyer, Chen Li, Sharad Mehrotra. *Executing sql over encrypted data in the database-service-provider model*. 2002.

[8] Alexandra Boldyreva, Nathan Chenette, Adam O'Neill. *Order-preserving encryption revisited: Improved security analysis and alternative solutions*. 2011.

[9] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. *Cryptdb: protecting confidentiality with encrypted query processing*. 2011.

[10] Pascal Paillier. *Public-key cryptosystems based on composite degree residuosity classes*. 1999.

[11] Withfield Diffie, Martin E. Hellman. *New Directions in Cryptography*. 1976.

[12] Yanjiang Yang, Haibing Lu, Jian Weng. *Multi-user private keyword search for cloud computing*. 2011.

[13] Emily Shen, Elaine Shi, Brent Waters. *Predicate privacy in encryption systems*. 2008.

[14] Ming Li, Shucheng Yu, Ning Cao, Wenjing Lou. *Authorized private keyword search over encrypted data in cloud computing*. 2011.

[15] Vipul Goyal, Omkant Pandey, Amit Sahai, Brent Waters. *Attribute-based encryption for fine-grained access control of encrypted data*. 2006.

[16] Mira Belenkiy, Jan Camenisch, Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Hovav Shacham. *Randomizable proofs and delegatable anonymous credentials*. 2009.

[17] Yong Ho Hwang, Pil Joong Lee. *Public key encryption with conjunctive keyword search and its extension to a multi-user system*. 2007.

[18] Changyu Dong, Giovanni Russello, Naranker Dulay. *Shared and Searchable Encrypted Data for Untrusted Servers*. 2009.

[19] Mihaela Ion Muhammad Rizwan Asghar, Giovanni Russello, Bruno Crispo. *Supporting Complex Queries and Access Policies for Multi-user Encrypted Databases*. 2013. `http://dl.acm.org/citation.cfm?doid=2517488.2517492`.

[20] Taher Elgamal. *A public key cryptosystem and a signature scheme based on discrete logarithms*. 2004.

[21] Rivest R.L., Shamir A., Adleman L.M. *A method for obtaining digital signatures and public-key cryptosystems*. 1978.

[22] Kenneth Ireland, Michael Rosen. *A Classical Introduction to Modern Number Theory*. 2010.

[23] Yanbin Lu, Gene Tsudik. *Enhancing data privacy in the cloud*. 2011.

[24] Jonathan Katz, Amit Sahai, Brent Waters. *Predicate Encryption Supporting Disjunctions, Polynomial Equations, and Inner Products*. 2013.

Biography

Alex Pellegrini was born in Cles (TN) Italy on December 4, 1992. He is a Computer Science Student at the University of Trento. Alex was an Exchange Student in fall 2013 at the Department of Computer Science at the Technical University of Denmark (DTU), Denmark. During his stay at DTU, he took courses concerning algorithms, data mining and data security. To get his bachelor degree, he studied how to protect data in outsourced environments, and how to evaluate complex queries over encrypted data.

Homepage: `http://rexos.github.io`