Computer Engineering Department
Faculty Of Engineering
Deanery of Higher Studies
Islamic University – Gaza
Palestine

New Methods of Query over Encrypted Data in Database

Ayman M. Al Derawi

Supervisor:

Dr. Mohammed Al Hanjouri

A Thesis Submitted in Partial
Fulfillment of the Requirements for
the Degree of Master in Computer
Engineering

1433 H
2012

New Methods of Query over Encrypted Data in Database

Ayman M. Al Derawi

Abstract

High secure data in databases is protected by encryption. When the data is encrypted, query performance decreases. In this work we propose three new mechanisms to query the encrypted data beside make a tradeoff between the performance and the security. We introduce three different methods, the first two works on encrypted data when the condition of the 'where' statement is '=' (searching for a specific whole word), the third one used when the condition of the 'where' statement is 'like' (searching for a part of the word). Our three methods based on replacing the select conditions on the encrypted data with another condition which is faster. In the first method we introduce a one-to-one mapping functions that is used as index for the plain data which will be encrypted; this function is also cannot be reversed without the key so the attackers cannot guess the plain text from the index. For example we will use AES as encryption/decryption function and SHA-1 as hashing function. In the second way we enhance the first way by putting the index on the memory, the index is implemented as a data structure Hash Map, this makes the response time faster but it needs a huge memory size, so this method cannot be used with the huge database size. In the third method we focus on the select statement contains a condition on a part of the cipher text, this makes it harder to implement the index without open a mapping between the plain text and the index. In the third way we work on two steps: creating the index and hiding the index. We notice that when the number of characters in the fuzzy query increase the response time enhance. We also notice that the response time for the fuzzy query contains the characters in the end of the word is better than the response time for the same query contains the characters in the start of the word and the response time for the fuzzy query contains the characters in the start of the word is better than the same query contains the character in the middle of the word. Our mechanisms work over many data-types. We implement our work as a layer above the DBMS; this makes our method compatible with any DBMS. The layer has common components for the three methods and specific components for each method. In the experiments we change the number of records in the database from 100 to 10,000 and measure the response time in mille second for the select query which have a condition on encrypted columns for each of the three proposed methods. The results of the experiments validate our approach. The experiments implement using a structure from a universal

benchmark TPC-H. The advantage of our work is that we enhance the response time of the query on the encrypted database beside maintain the security of the data. Our work can be used on equal and fuzzy conditions and can be implemented over any kind of DBMS.

Keywords- Encryption; Hashing; Querying over Encrypted Data; Numeric Index; Fuzzy Query.

الاستعلام عن البيانات المشفرة في قواعد البيانات

أيمن محمد الديراوي

ملخص

يتم حماية البيانات المهمة في قواعد البيانات بتشفير هذه البيانات. عندما يتم تشفير البيانات، تقل سرعة الاستعلام عنها. في هذا البحث نقترح ثلاث آليات جديدة للاستعلام عن البيانات المشفرة بجانب إجراء التوازن بين الأداء والأمان. الطريقة الأولى والثانية يعملان على البيانات المشفرة عندما يكون الشرط في البحث هو '=' (اي البحث عن كلمة بعينها)، الطريقة الثالثة تستخدم عندما يكون الشرط في البحث هو 'like' (البحث عن جزء من كلمة). تقوم فكرة الطرق الثلاث على استبدال شرط البحث عن البيانات المشفرة بشرط آخر اسرع في الاستعلام. في الطريقة الاولى قمنا ببناء فهرس للبيانات المراد تشفيرها باستخدام دوال لها خاصية (one-to-one) مع وجود شرط ان هذه الدوال لا يمكن عكسها بدون وجود المفتاح الخاص بها لذا فإنه لا يمكن معرفة البيانات الغير مشفرة باستخدام الفهرس. مثال على هذه الدوال والتي قمنا باستخدامها (AES) كدالة التشفير\فك التشفير و (SHA-1). في الطريقة الثانية قمنا بتطوير الطريقة الاولى وذلك بوضع الفهرس في الذاكرة، قمنا بتطبيق الفهرس على شكل هيكلية للبيانات (Hash Map)، هذه الطريقة قامت بتحسين وقت الاستجابة ولكنها تحتاج الى مساحة كبيرة من الذاكرة لذا فهذه الطريقة لا يمكن استخدامها مع قواعد البيانات كبيرة الحجم. في الطريقة الثالثة قمنا بالتركيز على الاستعلام المحتوي على شرط على جزء من البيانات المشفرة، هذا يجعل من الصعب تصميم الفهرس بدون اتاحة المجال للربط بين البيانات الغير مشفرة والفهرس. في الطريقة الثالة قمنا بالعمل على مرحلتين: تصميم الفهرس و اخفاء الفهرس، قمنا بملاحظة انه عندما يزيد عدد الاحرف في الاستعلام يتحسن وقت الاستجابة. قمنا ايضا بملاحظة ان وقت الاستجابة للاستعلام عن الاحرف الموجودة في نهاية الكلمة افضل من وقت الاستعلام عن الاحرف الموجودة في بداية الكلمة ووقت الاستعلام للاحرف الموجودة في بداية الكلمة افضل من وقت الاستعلام للاحرف الموجودة في وسط الكلمة .الطرق الثلاث تعمل على العديد من انواع البيانات. قمنا بتطبيق العمل كطبقة فوق نظام ادارة قواعد البيانات، وهذا يجعل أسلوبنا متوافق مع أي نظام لادارة قواعد البيانات. تحتوي هذه الطبقة على مكونات مشتركة للطرق الثلاث ومكونات اخرى خاصة بكل طريقة. اثناء التجارب قمنا بتغيير عدد الصفوف في قاعدة البيانات من 100 حتى 10,000 مع قياس وقت الاستجابة بالمللي ثانية للاستعلام المحتوي على شرط على العمود المشفر للطرق الثلاثة المقترحة. اثبتت النتائج صحة طرقنا المستخدمة. تم تنفيذ التجارب باستخدام معايير عالمية من (TPC-H). ميزات العمل الذي قمنا به اننا قمنا بتحسين وقت الاستجابة للاستعلام عن البيانات المشفرة في قواعد

البيانات مع المحافظة على سرية هذه البيانات. العمل الذي قمنا به يستخدم للاستعلام عن كل الشرط او جزء منه ويمكن تطبيقه مع اي نظام لادارة قواعد البيانات.

الاهداء

الى امي وابي الغاليين

الى زوجتي الصبورة الغالية

الى حلا المنورة حياتي

شكرا لكم جميعا، لولا دعمكم لما كان هذا العمل

ACKNOWLEDGMENT

I deeply thank my supervisor for all of his efforts to support me during my work, his immense support not only in advice, guidance, and inspiration, but also, he was one of the main reasons that support me to complete this thesis.

All my respect to Dr. Mohammed Al Hanjouri. The person who teaches me how to make an original papers and how to publish them, the person who makes me loves this study.

Table of Contents

List of Abbreviations

AES: Advanced Encryption Standard

SHA: Secure Hash Algorithm

DBMS: Database Management System

TPC: Transaction Processing Performance Council

SQL: Structured Query Language

XML: Extensible Markup Language

DES: Data Encryption Standard

GF: Gaussian Field

NSA: National Security Agency

MIT: Massachusetts Institute of Technology

MD: Message Digest

List of Figures

List of Tables

Chapter One

Introduction

Usually data is stored in databases to process and manage its relations; some data are classified as a high important data that needs to be high secured or on a level of security, the best way to secure such data is to encrypt it. Many encryption algorithms were studied and many designs of databases have prepared to put the considerations of encryption and security of the databases. In this work we focus on how to solve the problem of query the encrypted data in the database in a better response time beside maintain the security of this data.

1.1: Thesis Contribution

The traditional way to search an encrypted data is to decrypt all the data to plain text then find the target records. This way is obviously cost very time and has a bad performance especially with a large number of records; our objective is to find another ways which are faster.

We propose new methods to query encrypted data with many data types (string, character, numeric and date). Our methods have a good comparable response time with the traditional way. We also use an index over the data, the indexing information should be related with the data well enough to provide an effective query execution mechanism; on the other side, the relationship between indexes and data should not open the door to linking that can comprise the protection.

We introduce three methods, The core of first method based on using a one-to-one function that's generates a new value for the original data, this function must be one way function, so the attackers can't guess the original input value from the output value if using the same function. We use two such functions, the hash function and the encryption function. Another condition on this function is to be easy to use and have a good response time.

The second method enhance the first method by using a HASH_MAP a data structure that index the values, this enhance the first method response time and make it faster. But it requires a huge memory size so it's compatible with small databases or with distributed servers.

The two first methods work on any data types with the equal condition, the third method work on a fuzzy query, the fuzzy query is a query which has a 'like' statement. This makes it harder to execute the query because the data in the database becomes encrypted and the values of the characters changed, any mapping from the plain text to encrypted text will be a weak that can be used by the attackers to find the original plain text. The procedure to implement our method divides into two steps: first build the index by mapping the plain text into numeric index that can be ordered easily by the DBMS, second hide the values of this index. We explain in detail how to do the both steps and give a fully examples, then prove the enhancement by the results.

We have a compatible challenge, we do not know how the current DBMSs work and we cannot add changes to its cores, that's needs an open source DBMS. In order to solve this problem we have to make sure that our new method can adapt easily with the DBMS. The solution is by implementing a layer over the DBMS, this layer is a simple application that receives the clients request over the encrypted data, use our methods to process the request over the encrypted data and send updated request to the DBMS, receive the results from DBMS, do if needed any extra processing then return the results back to the client.

Our proposed methods implement on a standard database from a universal benchmark TPC-H and we use the data generator that is provided with this benchmark, all of our methods implement over the tables' layout from TPC-H and we test the experiments over it.

The experiments prove the theoretical idea behind our work and this follows by a comparison with the traditional way and a comparison between the methods cases itself.

1.2: Organization of the research

The research is consists of five chapters, they are organized as the follows: the first chapter is the introduction of our work. Chapter 2 is a related work. Chapter 3 is a background of the ready encryption algorithm and hash algorithm used in our work. Chapter 4 consists of the methodology used in our work; how to implement our work and the architecture for our methods. In chapter 5 we give a brief results built on a selected experiments and make a clear comparison between the experiments, analysis the results focus on the performance. Finally in chapter 6 we list the conclusion of our work.

Chapter Two

Related Work

2.1 Classification

The related work of our research can be classified into four main categories. The first category works on implementing a specific index for the encrypted data, our work can be classified to this category. The second one works on defining new special purpose encryption algorithms for the DBMS. The third category works on expecting the clients request on encrypted data, define expectation and statistics to propose the more requested encrypted data for each client and just decrypt this data. The fourth category works on proposing a secure schema for the encrypted database.

2.2: Related Work

In [1] the major challenges and design considerations related to database encryption were described. The article first presents an attack model and the main relevant challenges of data security, encryption overhead, key management, and integration footprint. Next, the article reviews related academic work on alternative encryption configurations; indexing encrypted data; and key management. Finally, the article concludes with a benchmark using the following design criteria: encryption configuration, encryption granularity and keys storage. Dawn Xiaodong Song [2] proposes a new encryption method that allows searching the encrypted data without decryption. However, the method is not adapted for database encryption. Hankan Hacijumus [3] proposes a way that has a weakness; it will output false joining records, which leads to the greatly increased cost of decrypting records and degraded performance of query. They propose a schema of executing SQL over encrypted data in the database-service-provider model. Then in [4] the writers proposed a new query method, in which the query is completed on the server side and the client side together, they have proposed bucket index, which support the range query for the numeric data. Then they add a technique that supports arithmetic computation [5]. In [6] Hore optimized the bucket index method on how to partition the bucket to get the trade between the security and query performance. The methods based on index is supported by DBMS (Data Base Management System), and focused on the query performance at the cost of storage space. There are also some researches on the fuzzy query of character string. Zhengfei Wang proposed a function to support fuzzy query over the encrypted character data [7] [8]. Their method named pairing coding method, it encodes every adjacent two characters in sequence and converted original string directly to another characteristic string by a hash function. This method can't deal with some characters, and could perform badly for big character string. Paper [9] had proposed characteristics matrix to express string and the matrix will also be compressed into a binary string as index. Every character string need a matrix size of 259x256, it is large and will lead to much computation; in addition, the length of index has come to more than hundred bits, which is not suitable for storage in database. In [10] the paper works on a group of users that wants to access a secure data on a server. The shared sensitive information requires more security and privacy protection, in this paper, two schemes were proposed which can search the encrypted documents without re-

encrypting all documents in a server even if group keys have to be updated. The schemes can support general database normalization for encrypted database. Their experiments show that their schemes are much more efficient than the comparables ones. Paper [11] only encrypts the sensitive field and it is also using bucket index to improve query performance. The order on numeric data is very useful. But on the character data, it has little effect. So the method in [11] is not fit for the character data. [12] Creates a B+ tree index for the data before encrypting them. When querying the encrypted data, firstly, it locates the encrypted records related to the querying predicate based on the B+ tree index; secondly, it decrypts the encrypted records to accomplish the results. Also, it must encrypt the B+ tree itself to protect it from leaking confidential information. According to the structure of the B+ tree, it encrypts each node of the B+ tree separately. The results of experiments in [12] show that the query performance over the encrypted data decreases about 20 percent compared with the plaintext query performance. In [13] the authors first survey the most relevant concepts of database security and summarize the most well-known techniques. They focus on access control systems, on which a large body of research has been devoted, and describe the key access control models and the role-based access control model. They also discuss security for advanced data management systems, and cover topics such as access control for XML. They then discuss current challenges for database security and some preliminary approaches that address some of these challenges. In [14] the paper presents a database encryption scheme that provides maximum security, whilst limiting the added time cost of encryption and decryption. On the other side, the queries such as sums, averages, counts and other statistical functions that aggregate over a range of data in the database cannot be performed directly in the proposed schema. Paper [15] presents a multilevel database security model which needs an open source database to implement over it. It also increases the complexity of access control. [16] Proposes an index mechanism of bucket index on the character data, which has close relationships with all of its characters. The index tries to translate the character string into a numeric data, on which the primary query will be processed to filtrate the records roughly. Only the rest records need to be decrypted. But the way cannot do the exception query over the database. Michael Mitzenmacher in [17] introduces the Compressed Bloom filter which is a simple randomized data structure for representing a set in order to support membership queries which can be used as index for the data. In [18] the authors generalize the traditional Bloom filter to Weighted Bloom Filter, which contains the information on the query frequencies and the membership likelihood of the elements into its optimal design. In [19] the writers propose a secure cipher index with great efficiency over the encrypted character data to improve the performance of the encrypted database, the writers also studied the influence of some parameters like the length of the character group. In [20] a bloom filter based index to support fuzzy query over encrypted character data is proposed on the principle of two-phase query. The proposed method performance decreases when increasing the length of the sensitive data. Solution for SQL querying over database system containing encrypted data of Yong Soon KIM and Eui Kyeong Hong in [21] is based on the UniSQL commercial relational database management system version 6.3 and cannot be adapted with any other DBMS. In [22] the paper address high-level authorization specifications and its efficient implementation in object oriented database scenario. The papers discuss three different types of access: Discretionary access control, Mandatory access control and role based access control in Relational DBMS and object oriented data bases. The paper identified some of the issues and current security models in object oriented database system. The

writers in [23] developed an inference violation detection system to protect sensitive data content, based on data dependency, database schema, and semantic knowledge. They gave an example for illustrating the use of the proposed technique to prevent multiple collaborative users from deriving sensitive information via inference. Their method is based on an ideal case when they always have a full knowledge of the data and the users. [24] Described and discussed various security issues in database. This paper is useful for planning of explicit and directive based database security requirements. [25] Is more specific compared with [24]. In [25] the writers focus on web database and how to protect the database from the SQL attacks.

Chapter Three

Background

3.1: Advanced Encryption Standard (AES) Algorithm

The Advanced Encryption Standard (AES) is a specification for the encryption of electronic data. Originally called Rijndael, the cipher was developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen [26].

AES has been adopted by the U.S. government and is now used worldwide. It replaces the Data Encryption Standard (DES). The algorithm described by AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data.

In our experiment we used AES-256 to encrypt the pre-selected column that's usually contains a high important data that is needed to be secured, the key of the AES will created according to standards and will kept on a safe place.

AES is based on a design principle known as a substitution-permutation network, and is fast in both software and hardware. Unlike its predecessor DES, AES does not use a Feistel network. AES is a variant of Rijndael which has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits. By contrast, the Rijndael specification *per se* is specified with block and key sizes that may be any multiple of 32 bits, both with a minimum of 128 and a maximum of 256 bits.

AES operates on a 4×4 column-major order matrix of bytes, termed the *state*, although some versions of Rijndael have a larger block size and have additional columns in the state. Most AES calculations are done in a special finite field.

The key size used for an AES cipher specifies the number of repetitions of transformation rounds that convert the input, called the plaintext, into the final output, called the ciphertext. The numbers of cycles of repetition are as follows:

- 10 cycles of repetition for 128 bit keys.
- 12 cycles of repetition for 192 bit keys.
- 14 cycles of repetition for 256 bit keys.

Each round consists of several processing steps, including one that depends on the encryption key itself. A set of reverse rounds are applied to transform ciphertext back into the original plaintext using the same encryption key.

3.2: High-level description of the algorithm

1. KeyExpansion—round keys are derived from the cipher key using Rijndael's key schedule
2. Initial Round
    1. AddRoundKey—each byte of the state is combined with the round key using bitwise xor
3. Rounds
    1. SubBytes—a non-linear substitution step where each byte is replaced with another according to a lookup table.
    2. ShiftRows—a transposition step where each row of the state is shifted cyclically a certain number of steps.
    3. MixColumns—a mixing operation which operates on the columns of the state, combining the four bytes in each column.
    4. AddRoundKey
4. Final Round (no MixColumns)
    1. SubBytes
    2. ShiftRows
    3. AddRoundKey

3.3: SubBytes Step

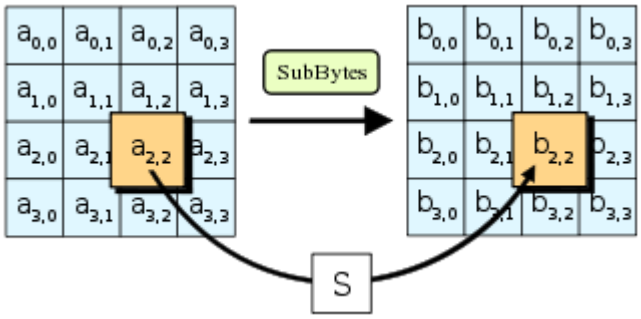

Figure 3.1: SubByte Step in AES [26]

In the SubBytes step (Figure 3.1), each byte in the *state* matrix is replaced with a SubByte using an 8-bit substitution box, the Rijndael S-box. This operation provides the non-linearity in the cipher. The S-box used is derived from the multiplicative inverse over **GF**(2^8), known to have good non-linearity properties. To avoid attacks based on simple algebraic properties, the S-box is constructed by combining the inverse function with an invertible affine transformation. The S-box is also chosen to avoid any fixed points, and also any opposite fixed points.

3.4: ShiftRows Step

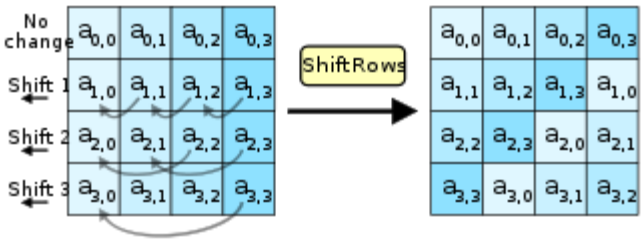

Figure 3.2: ShiftRows Step in AES [26]

The ShiftRows step (Figure 3.2) operates on the rows of the state; it cyclically shifts the bytes in each row by a certain offset. For AES, the first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively. For blocks of sizes 128 bits and 192 bits, the shifting pattern is the same. Row n is shifted left circular by n-1 bytes. In this way, each column of the output state of the ShiftRows step is composed of bytes from each column of the input state. (Rijndael variants with a larger block size have slightly different offsets). For a 256-bit block, the first row is unchanged and the shifting for the second, third and fourth row is 1 byte, 3 bytes and 4 bytes respectively—this change only applies for the Rijndael cipher when used with a 256-bit block, as AES does not use 256-bit blocks.

3.5: MixColumns Step

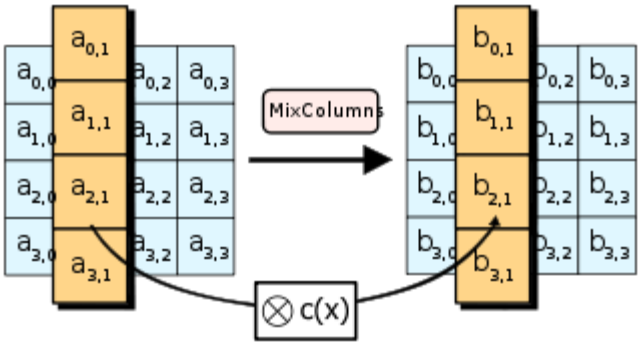

Figure 3.3: MixColumns Step in AES [26]

In the MixColumns step (Figure 3.3), the four bytes of each column of the state are combined using an invertible linear transformation. The MixColumns function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes. Together with ShiftRows, MixColumns provides diffusion in the cipher.

During this operation, each column is multiplied by the known matrix that for the 128 bit key is. The multiplication operation is defined as: multiplication by 1 means no change, multiplication by 2 means shifting to the left, and multiplication by 3 means

shifting to the left and then performing xor with the initial unshifted value. After shifting, a conditional xor with 0x1B should be performed if the shifted value is larger than 0xFF.

In more general sense, each column is treated as a polynomial over **GF**(2^8) and is then multiplied modulo x^4+1 with a fixed polynomial $c(x) = 0x03 \cdot x^3 + x^2 + x + 0x02$. The coefficients are displayed in their hexadecimal equivalent of the binary representation of bit polynomials from **GF**(2)[x]. The MixColumns step can also be viewed as a multiplication by a particular MDS matrix in a finite field.

3.6: The AddRoundKey Step

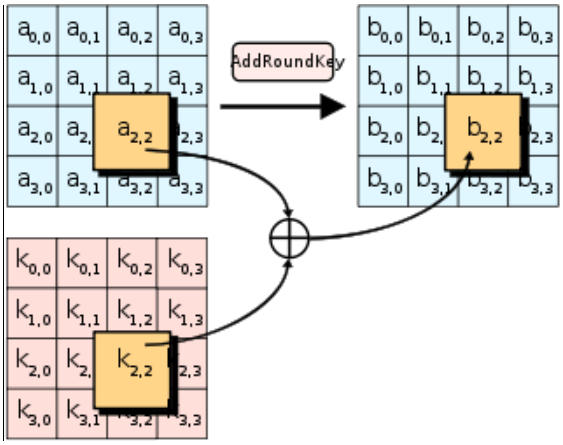

Figure 3.4: AddRoundKey Step in AES [26]

In the AddRoundKey step (Figure 3.4), the subkey is combined with the state. For each round, a subkey is derived from the main key using Rijndael's key schedule; each subkey is the same size as the state. The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR.

On systems with 32-bit or larger words, it is possible to speed up execution of this cipher by combining the SubBytes and ShiftRows steps with the MixColumns step by transforming them into a sequence of table lookups. This requires four 256-entry 32-bit tables, and utilizes a total of four kilobytes (4096 bytes) of memory — one kilobyte for each table. A round then is done with 16 table lookups and 12 32-bit exclusive-or operations, followed by four 32-bit exclusive-or operations in the AddRoundKey step.

If the resulting four kilobyte table size is too large for a given target platform, the table lookup operation can be performed with a single 256-entry 32-bit (i.e. 1 kilobyte) table by the use of circular rotates.

Using a byte-oriented approach, it is possible to combine the SubBytes, ShiftRows, and MixColumns steps into a single round operation.

3.7: Hashing: SHA-1 (Secure Hash Algorithm)

In our experiments we used SHA-1 as a hashing algorithm; SHA-1 is the most widely used of the existing SHA hash functions, and is employed in several widely used applications and protocols.

SHA-1 produces a 160-bit message digest based on principles similar to those used by Ronald L. Rivest of MIT in the design of the MD4 and MD5message digest algorithms, but has a more protective design [27].

The original specification of the algorithm was published in 1993 as the Secure Hash Standard, FIPS PUB 180, by US government standards agency NIST (National Institute of Standards and Technology). This version is now often referred to as SHA-0. It was withdrawn by NSA shortly after publication and was superseded by the revised version, published in 1995 in FIPS PUB 180-1 and commonly referred to as SHA-1. SHA-1 differs from SHA-0 only by a single bitwise rotation in the message schedule of its compression function; this was done, according to NSA, to correct a flaw in the original algorithm which reduced its cryptographic security. However, NSA did not provide any further explanation or identify the flaw that was corrected. Weaknesses have subsequently been reported in both SHA-0 and SHA-1. SHA-1 appears to provide greater resistance to attacks, supporting the NSA's assertion that the change increased the security.

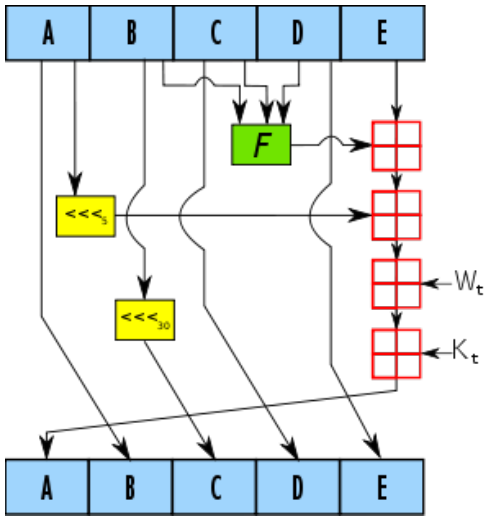

Figure 3.5: One iteration within the SHA-1 compression function: A, B, C, D and E are 32-bit words of the state; *F* is a nonlinear function that varies; $\lll_n$ denotes a left bit rotation by *n* places; *n* varies for each operation; W_t is the expanded message word of round t; K_t is the round constant of round t; $\boxplus$ denotes addition modulo 2^{32}.[27]

3.8: Hash Map

In computer science, a hash map is a data structure that uses a hash function to map identifying values, known as keys (e.g., a person's name), to their associated values (e.g., their telephone number) (Figure 3.6). Thus, a hash map implements an associative array. The hash function is used to transform the key into the index (the hash) of an array element (the slot or bucket) where the corresponding value is to be sought. In a well-dimensioned hash map, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table. At the heart of the hash map algorithm is a simple array of items; this is often simply called the hash table. Hash table algorithms calculate an index from the data item's key and use this index to place the data into the array. The implementation of this calculation is the hash function:

Index = f(key, array Length)

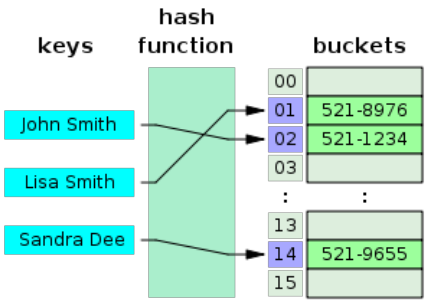

Figure 3.6: A small phone book as a hash table [28]

Chapter Four

Methodology

4.1: Layering Technique

In order to implement our work we need an open source database, the drawback for this technique is that our work can adapt only with this type of the database and can't work with the commercial databases like Oracle, MS SQL, MS Access, MySQL, … etc which surely are closed source. To solve this problem, we developed another way to implement our work to adapt with any kind of DBMS. We add a layer above any kind of DBMS, this layer have the responsibility to manage the way to query over encrypted data.

The drawback for adding the new layer is the response time; the results prove that the performance of adding the layer will be much better when working on encrypted data with the traditional way.

The client will work over the layer which will contact with DBMS figure (4.1).

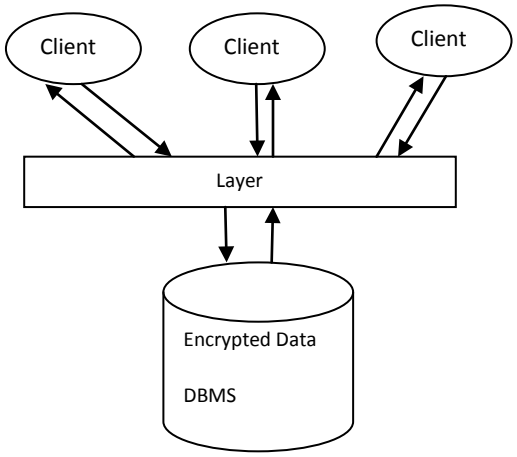

Figure 4.1: The Layer over the DBMS

The layer will provide the inner needed method for the methodology of process the encrypted data. The layer is better to be placed on the same place with the DBMS for two reasons:

1- Decreases the time of contacting with DBMS

2- Security purpose, the DBMS is usually placed on a safe place from the attackers.

The implementation of the layer can be as a service that has a port to connect with.

4.2: Architecture of the first Method

We implement two ways to work the first adding new hash column the second using the same encrypted column. The core of first method based on adding a new column for each encrypted column, this column contains a unique value for each appreciate plain value that will encrypted, In our method we used a hash algorithm to generate the 1-to-1 mapping from the plain data to unique hash values (Figure 4.2).

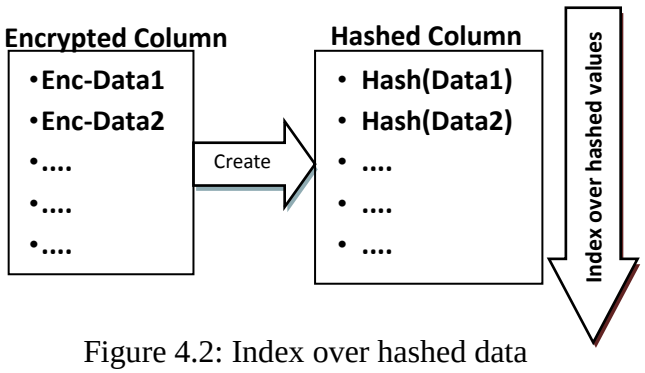

Figure 4.2: Index over hashed data

An index is build over the hashed column that makes the searching over the values in the hash column faster. By finding the needed hash value we find the needed plain text.

The second way is to work over the encrypted column directly without using any extra column.

An index is build over the encrypted column that makes the searching over the values in the encrypted column faster (Figure 4.3). By finding the needed encrypted value we find the needed plain text. That's done by using the same encryption/decryption algorithm with the same symmetric key which must be kept secret away from the attackers.

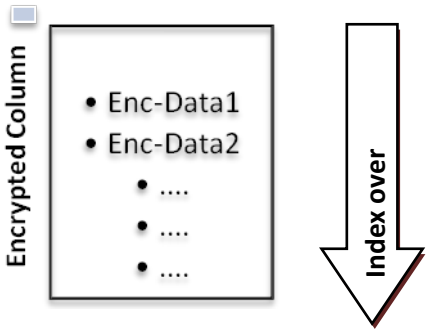

Figure 4.3: Index over encrypted data

The architecture of the layer is shown in figure (4.4). The queries from the client sent to the layer which has a subsystem called the Query Processor to check in the Meta data if there is any query on an encrypted column. The Meta data contains information of the encrypted columns in their tables and the corresponding hashed columns (if using the hash function) table 4.1.

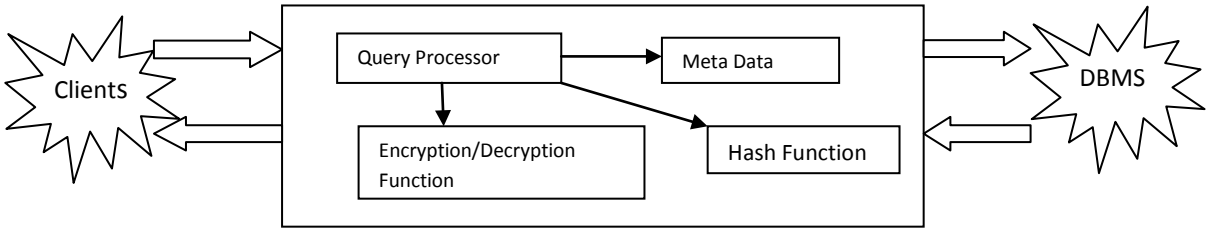

Figure 4.4: Architecture of the layer for the first method

This first way will cost more time especially with the insertion and updating on the encrypted column. Any insert or update statement must be followed by an inserting/updating value on the hash column.

Table 4.1: Meta data example

Table Name	Encrypted Column	Hash Column
CUSTOMER	C_PHONE	H_C_PHONE
…	…	…

When using the hash function as the one-way function the Query Processor replaces the client query with 'where' clause on encrypted data value with another one on the hashed data value. When using the encryption function as the function, the Query Processor replaces the client query with 'where' clause on encrypted data value with another with an encryption on the plain searched data. For example if table CUSTOMER has an encrypted column C_PHONE and the client query is:

SELECT * FROM CUSTOMER

WHERE C_PHONE = '02 526 544';

By using the tradition way we need to decrypt all the values of C_PHONE then check which one equals '02 526 544', this means a huge response time especially with a large number of records.

By using our technique and using the hash function, there will be another column appropriate for the C_PHONE contains the hashed values of C_PHONE named H_C_PHONE.

The query processor will replace the where statement to be

SELECT C_NAME FROM CUSTOMER

WHERE H_C_PHONE = HASH_VALUE('02 526 544');

By using the index over H_C_PHONE it will be fast and easy to find the row that has the value of '02 526 544' on C_PHONE without decrypt any value which mean a better response time.

By using our technique and using the encryption function, the query processor will replace the where statement to be

SELECT C_NAME FROM CUSTOMER

WHERE C_PHONE = encrypt('02 526 544');

By using the index over C_PHONE it will be fast and easy to find the row that has the value of '02 526 544' on C_PHONE without needing to decrypt all the values which means a better response time.

Our method can works also with the range of values for example, if the client query on the encrypted column is:

SELECT * FROM CUSTOMER

WHERE C_PHONE IN (’02 526 878’, ’02 584 231’, ’03 874 214’);

The translated query will be

SELECT * FROM CUSTOMER

WHERE H_C_PHONE IN (HASH_VALUE(’02 526 878’),
HASH_VALUE(’02 584 231’), HASH_VALUE(’03 874 214’));

And if we used the encryption algorithm:

SELECT * FROM CUSTOMER

WHERE C_PHONE IN (encrypt(’02 526 878’), encrypt (’02 584 231’),

encrypt (’03 874 214’));

Analysis:

Using of encryption algorithm means that we don’t need to add a new column for the hashed values of the plain text but in the other side we need to use the key of the encryption/decryption algorithm to encrypt the client condition, decreasing the usage of the key means more security.

The security of our methods depends on the security of the key of the encryption/decryption algorithm and the strength of the algorithm itself: hash algorithm and encryption/decryption algorithm. In our implementation we used AES-256 as the encryption/decryption algorithm and SHA-1 as hash algorithm.

4.3: Architecture of the Second Method

The core of our method based on using the hash map. The data is stored on the Hash Map by using the function PutValue(Key, Value), the Key is an identifier to the Value, we use the function GetValue(Key): Value to get the Value by passing the Key. In our methodology the Key will be the plain text and the Value will be the Encrypted plain text; i.e. encrypted Key.

Value = Enc(Key)

This way will cost extra time especially with the insertion and updating on the encrypted column. Any insert or update statement must be followed by an inserting/updating value on the Hash Map. The time complexity for the Hash Map in big O notation is O(1) for the search and O(1) for the insert in average, O(n) for the search and O(1) for the insert in worst case, so the extra time needed for insert and update is acceptable.

The architecture of the layer is shown in figure (4.5). The queries from the client sent to the layer which has a subsystem called the Query Processor to check in the Meta data if there is any query on an encrypted column. The Meta data contains an instance of a data structure object called Hash Map. The Hash Map stores the mapping between the plain text and the encrypted text as KEY: VALUE, in which the KEY is the plain text and the VALUE is the encrypted value of the plain text. The Hash Map contains two main operations, PutValue and GetValue. PutValue(Key , Value), GetValue(Key): Value.

The Query Processor replaces the client query with 'a plain where' clause on encrypted data value (the where clause is a plain text) with another one with an encryption on the plain searched data. For example if table CUSTOMER has an encrypted column C_PHONE and the client query is:

SELECT * FROM CUSTOMER

WHERE C_PHONE = '02 526 544';

By using the tradition way we need to decrypt all the values of C_PHONE then check which one equals '02 526 544', this means a huge response time especially with a large number of records.

By using our technique and using the Hash Map, the query processor will first search the Hash Map for the Key = '02 526 544' and get the Value which will be the ENC_VALUE ('02 526 544'), then replaces the where statement to be:

SELECT C_NAME FROM CUSTOMER

WHERE C_PHONE = ENC_VALUE('02 526 544');

By using the index over C_PHONE it will be fast and easy to find the row that has the value of '02 526 544' on C_PHONE without needing to decrypt all the values which means a better response time.

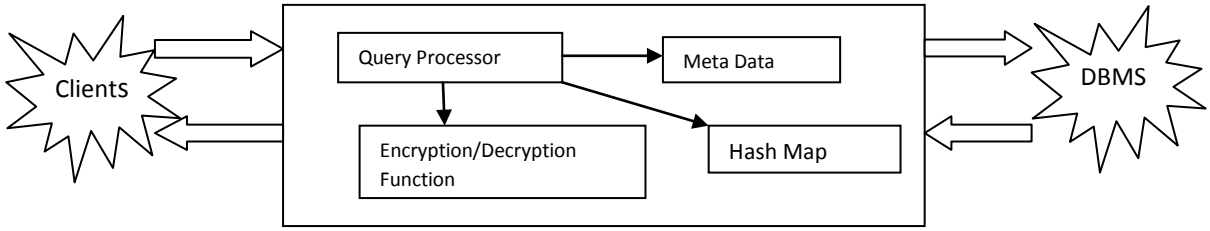

Figure 4.5: Architecture of the layer for the second method

Analysis:

The main difference between the first and the second method is that the first method works on the mapping which stored in the database (Hard Disk) but the second method works on the mapping which stored in the memory as a data structure (Hash Map) have O(1).

The second method didn't fit with the databases which have tables with a large number of records because it will take a huge memory and there will be a need for virtual memory which decrease the response time of this method.

Using of the distributed systems with distributed shared memory can be a part of solution for this problem (working with large databases).

The secure of the Hash Map depends on the security of its hash function; in our method we used SHA-1 as the hash algorithm for the Hash Map.

4.4: Architecture of the third Method

In the previous sections we work on the specific user query which includes "equal" expression in the where condition for example:

Select * from table1 where column1 = value

Our previous work works on any data type on a specific value. We introduce here another enhanced method which works on fuzzy query; the fuzzy query is a query which contains a "like" statement.

For example in the following SQL clause, the string '$s_1 s_2 s_3 \ldots s_x$' is the *match_expression*.

SELECT * FROM *table_name* WHERE *column_name* = '$s_1s_2s_3…s_x$'

It is an exact match, while in the fuzzy match, there will be some wildcard characters ('like', '%').

We will have a problem that the SQL clauses cannot execute over the encrypted data directly. In this work we propose a secure cipher index with great efficiency over the encrypted character data to improve the performance of the encrypted database. The idea is to map the target character data in the database into its index value, and translate the SQL clauses into a usable form to execute over the index attribute. Hash key and some other techniques are introduced to make the cipher index secure. In this schema, the same character data in different rows has different index value and the index value has no statistical characteristic, so it can avoid the problem of statistical attack and plain attack. When there is a query over the encrypted attribute, we filter out the rows through the index and only decrypt the matching ones, which could reduce the decryption cost greatly and prevent the information leaking.

Our work consists of two steps: creating the index and hiding the index.

We will use the following notations in our work

S: the string value on the target column

si : the ith character in S

E: the encryption function

E(S): the encrypted value for S

X(S): the indexed value for S

Xi: the ith character in X(S)

Hi: the numeric hashed value for si

Match_expression : The character value appeared in the where clause

Hash_Key: The key introduced to the hash function

4.4.1: Step1: Creating the index

For each target column (need to be encrypted), we add a new column named columnName_index which will be the index of the target column.

In order to enable the working on fuzzy query we will work on the character level, we will match each character in the target column to a numeric value.

In order to match the characters to numeric values we will use a hash function takes the character si and the hash_key as its input parameters and produces a hashed numeric value hi.

In our schema hi must be one digit numeric value so we limit the group of hash function output values between 0 and 9.

At this point it is easy for the attackers to get the plain text by making a statistical attack and plain attack, so in order to make it hard we will use a different hash_key for every table, the hash_keys must be stored in a safe place as a table like the one in the table 4.2:

Table 4.2: TABLES_HASHES example

Table Name	Hash_Key
REGION	X58@
CUSTOMERS	Ds85#ed
Table3	Hui
…	…

We named this table by TABLES_HASHES

If we choose the table REGION from the tpc-h tables layout and we chose the column NAME to be encrypted and for example the NAME value for the first row in table REGION is “GAZA”, and by assume the hash_key for this table is “X58@” the process of mapping the plain word to the index will be:

1- divide the word S

The word S will be divided into its characters si

So the word “GAZA” with be G, A, Z, A

2- Mapping each character to a numeric value of one digit

By using the hash function and the hash key “X58@” the output values will be like:

H(G, X58@) = 8

H(A, X58@) = 7

H(Z, X58@) = 4

H(A, X58@) = 7

This process can be shown in table 4.3.

Table 4.3: Creating the index

ID	REGIONKEY	NAME	E(NAME)	HASH_KEY	H(NAME)
1	5	GAZA	As$fs54a=	X58@	8747
2	5	GAZA	As$fs54a=	X58@	8747
3	6	CAIRO	Ds33##442	X58@	97563
…	…	…	…	…	…

Note that the character "A" in word "GAZA" will be matched to the same numeric value in the same row and equal to the value in the second ROW and the numeric value for the third ROW. This can be a port to the attackers to find the original plain text.
Note that the HASH_KEY is not a part of the table and we put it here for explain only.

4.4.2: Step 2: Hiding the index

In order to solve the above problem we introduce another function R and put a condition that this function must be calculated easily and its calculation time must be << the calculation time required for E, another condition that is R must be invertible.
Condition:

O(R) << O(E)

For example this function can be the XOR, XNOR, ADD, and MULTIPLICATION.

We introduce another table for mapping tables, rows, functions and the keys for these functions. We named this table TABLES_FUNCTIONS. We assume that TABLES_FUNCTIONS is stored in a safe place.

Table 4.4: TABLES_FUNCTIONS example

Table_Name	Row_ID	Function (R)	Key
REGION	1	XOR	HG452152
REGION	2	XNOR	FV21R4
Customer	1	ADD	OP854D
…	…	…	…

The functions are selected from the group G ={XOR, XNOR, ADD, MULTIP}, the keys are generated randomly.

The row id can be getting easily from the DBMS. We use more than one R to make it harder to the attacker to know the original index and we use different keys for the same purpose.
The next mission is to use these functions to hide the index, for the example table REGION we add another column INDEX_ .

The INDEX_ column is calculated as follows:

Suppose we need to create the INDEX_ for the row id = 1 in the table REGION, we first select the function R and the KEY from TABLES_FUNCTIONS.

INDEX_ = H(NAME) [FUNCTION] KEY

```
SELECT function, key from TABLES_FUNCTIONS
WHERE TABLE_NAME = 'REGION'
AND     ROW_ID = 1;
```

This select statement will returns FUNCTION = XOR and KEY = HG452152.
We use these outputs to generate the INDEX_ as :

INDEX_ = H(NAME) XOR HG452152= 8747 XOR HG452152

Let denotes the length of H(NAME) by L, and the length of the KEY by Lk

If L < Lk , then we add the first (Lk-L) bits from KEY to the end of the H(NAME).

If L > Lk, then we add the first (L-Lk) bits from the KEY to the end of the KEY.

By using this way we hide the length of the H(NAME) and make it harder for the attacker to guess the H(NAME).

The results of this operation are described in table.

Table 4.5: Hiding the index

ID	REGIONKEY	NAME	E(NAME)	HASH_KEY	H(NAME)	INDEX_
1	5	GAZA	As$fs54a=	X58@	8747	58KJBVNT
2	5	GAZA	As$fs54a=	X58@	8747	IOPSDE
3	6	CAIRO	Ds33##442	X58@	97563	8TFS3C
…	…	…	…	…	…	…

Note the following:

- The length of H(NAME) in ID = 1 and ID = 2 are equal but the length of their INDEX_ are different due to using a different keys.
- The same H(NAME) in ID = 1 and ID = 2 produces a different INDEX_.
- The character A in GAZA is mapped to different values in the same row in the ID = 1.
- The character A in GAZA is mapped to different values in row ID = 1 and row ID = 2.
- The character A in GAZA and CARIO is mapped to different values in row ID = 1 and row ID = 2 and row ID = 3.

4.4.3: Running Fuzzy query over the index

Suppose there is a select query over the encrypted data, and the tables in this query is indexed used our way.
The first select query:

SELECT * FROM REGION

WHERE NAME LIKE 'G%';

First we select the function (R) and the key from TABLES_FUNCTIONS. The DBMS can also have a cash of this information because it usually used, this make it faster.

SELECT ROW_ID, Function, Key

FROM TABLES_FUNCTIONS

WHERE TABLE_NAME = 'REGION';

After that we use ROW_ID, Function and Key to return the INDEX_ in table region to H(NAME) by using the equation:

$$H(NAME) = INDEX\_ (FUNCTION)^{-1} Key$$

So for the ROW_ID = 1 it will be:

$$H(NAME) = 58KJBVNT\ (XOR)^{-1}\ HG452152$$

$$= 58KJBVNT\ XOR\ HG452152$$

$$= 8747$$

For the ROW_ID = 2 it will be:

$$H(NAME) = IOPSDE\ (XOR)^{-1}\ FV21R4$$

$$= IOPSDE\ XOR\ FV21R4$$

$$= 8747$$

For the ROW_ID = 3 it will be:

H(NAME) = 8TFS3C (ADD)$^{-1}$ OP854D

= 8TFS3C SUB OP854D

= 97563

And so on.

Then we used TABLES_HASHES to get the HASH_KEY for table REGION.

SELECT HASH_KEY FROM TABLES_HASHES

WHERE TABLE_NAME = 'REGION';

Which return the HASH_KEY = X58@. We use this Hash key to get the mapping from the character to number.
The condition in the select statement will be changed to be:

SELECT * FROM REGION

WHERE = INDEX_ (FUNCTION) $^{-1}$ Key LIKE 'H(G, X58@) %';

=

SELECT * FROM REGION

WHERE H(NAME) LIKE '8 %';

It easily for the DBMS to execute the above query over the H(NAME) of the table REGION and returns the ROWS_IDs = 1,2 as a result of this query.

Table 4.6: Result of the query for ROWS_IDs = 1,2

ID	REGIONKEY	E(NAME)	H(NAME)	INDEX_
1	5	As$fs54a=	8747	58KJBVNT
2	5	As$fs54a=	8747	IOPSDE

The DBMS does not need to encrypt all the table, an just encrypt the returned records, this obviously decrease the time need to execute the user query over encrypted and enable the user to use a fuzzy query on that encrypted data.

Suppose the condition in the select statement will return data with different words, our way will work also.

Suppose the select statement is:

SELECT * FROM REGION

WHERE NAME LIKE '%A%';

First we select the function (R) and the key from TABLES_FUNCTIONS.
After that we use ROW_ID, Function and Key to return the INDEX_ in table region to H(NAME) by using the equation:

H(NAME) = INDEX_ (FUNCTION) $^{-1}$ Key

Then we used TABLES_HASHES to get the HASH_KEY for table REGION.
Which return the HASH_KEY = X58@. We use this Hash key to get the mapping from the character to number.
The condition in the select statement will be changed to be:

SELECT * FROM REGION

WHERE = INDEX_ (FUNCTION) $^{-1}$ Key LIKE '%H(A, X58@) %';

=

SELECT * FROM REGION

WHERE H(NAME) LIKE '%7 %';

It easily for the DBMS to execute the above query over the H(NAME) of the table REGION and returns the ROWS_IDs = 1,2, 3 as a result of this query.

Table 4.7: Result of the query for ROWS_IDs = 1,2, 3

ID	REGIONKEY	E(NAME)	H(NAME)	INDEX_
1	5	As$fs54a=	8747	58KJBVNT
2	5	As$fs54a=	8747	IOPSDE
3	6	Ds33##442	97563	8TFS3C

This proves that our way can be used widely in different fuzzy query.

Analysis: Collision Problem

In our method we map the characters values in the plain text to numeric values between 0 and 9, this can make a collision, for example:

> Character X1 can be mapped to numeric value N1 and Character X2 can be mapped to the same numeric value N1.

This problem happens because the range of the numeric values which is in our method between 0 and 9 is less than the range of the characters values.

The collision problem means that the select statement may return rows that are not satisfy the condition and they must be decrypted to filter out these rows.

Changing the scope of the numeric values and make it larger will solve the collision problem but in the other side will make the need to have a two digit numbers and this will make another problem in our schema, suppose the range of the numeric values is between 0 and 100, the hash function map the character value ‘x1’ to 3 and map the character value ‘x2’ to 33, it can be a miss understanding of is 33 maps to one ‘x2’ or maps to two ‘x1’? This means that the select statement which has the condition :

= INDEX_ (FUNCTION) $^{-1}$ Key LIKE ‘%3%’

will returns all the rows have ‘x1’ or ‘x2’ and they must be decrypted and filtered out.

Mapping the characters values to non-numeric data types will make the index slower or stop the index so it is not acceptable.

Architecture of the layer for the third method

The architecture of the layer is shown in figure (4.6). The queries from the client sent to the layer which has a subsystem called the Query Processor to check in the Meta data if there is any query on an encrypted column. The Query Processor replaces the client query with 'a plain where' clause on encrypted data value (the where clause is a plain text) with another one with which will work on our numeric index. The main responsibility for Query Processor is to map the searched text on fuzzy query to numeric values work on our index.

The Index_Maping_Info contains secure information on how to build the index and how to map the characters values to numeric values and important information about the functions used on building the index.

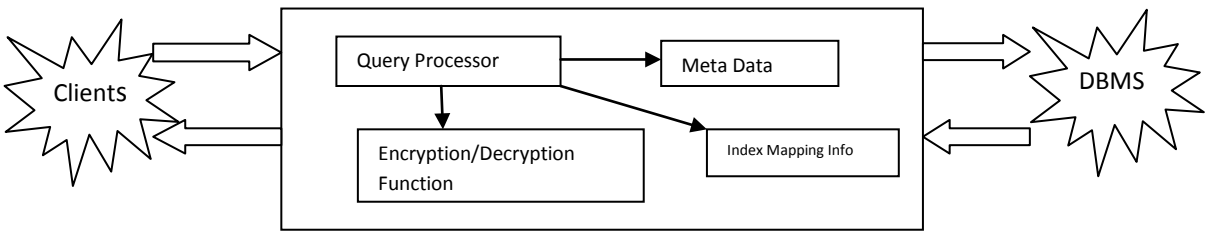

Figure 4.6: Architecture of the layer of the third method

Chapter Five

Experiments and Analysis of Performance

5.1: Experiments environment

The purpose of the experiments is to show the validity and the efficiency of our proposed approach.

According to TPC-H benchmark, the data in the database is automatically created by using the tool dbgen. TPC-H database include eight tables, of which used in our experiment is customer table. To encrypt data of the tables, AES -256 encryption algorithm implemented in Delphi is used. The experiments are conducted on a personal computer with Intel Core2 Due 2.10 GHz and 2.87 GB RAM. Relevant software components used are Windows 7 as the operating system and Oracle 11g R2 as the database server. The layer is implemented by using the Delphi as a programming language. We test the different methods by measure the response time of the query over the table has a number of records ranging from 100 to 10000 records.

Database Entities, Relationships, and Characteristics

The components of the TPC-H database are defined to consist of eight separate and individual tables. The relationships between columns of these tables are illustrated in Figure 5.1

Data Generator

The DBGEN program used to generate the data that populate the TPC-H Databases.

Querying over Encrypted data using traditional way

In the experiment, we test query execution time through comparing two different query approaches. The first way is the traditional way; decrypt all encrypted character data before querying them. The second way, which we propose in this work, is to decrypt the result records after filtering the records not related to querying conditions.

Algorithm 5.1: Query over encrypted data.

Purpose: To return the result of a query over encrypted data with a better response time than the traditional way.

INPUT: Query which has a where statement on an encrypted data.

OUTPUT: Collection of records satisfying with the where statement of the query.

Procedure:

(1) Receiving the query from the client

(2) Checking the metadata if the condition is over an encrypted column

(3) If so, changing the query to work over the index using the rules of our methods

(4) Executing the new query, returning the records satisfying the translated query conditions

(5) Decrypting the result records and obtaining the actual results

(6) Filter out the actual results if needed

(7) Returning the filtered results to the client

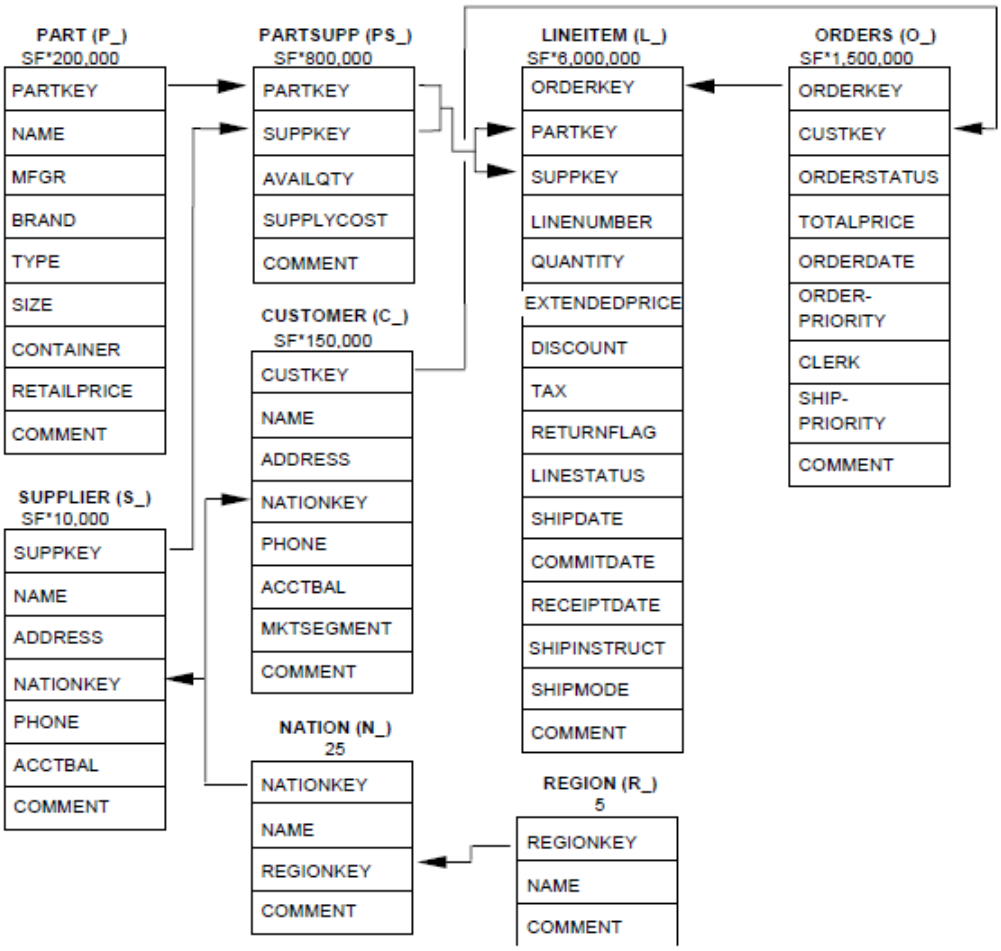

Figure 5.1: The TPC-H Schema

Required Tables

The following list defines the required structure (list of columns) of each table.

PART Table Layout
P_PARTKEY identifier SF*200,000 are populated
P_NAME variable text, size 55
P_MFGR fixed text, size 25
P_BRAND fixed text, size 10
P_TYPE variable text, size 25
P_SIZE integer
P_CONTAINER fixed text, size 10
P_RETAILPRICE decimal
P_COMMENT variable text, size 23
Primary Key: P_PARTKEY

SUPPLIER Table Layout
S_SUPPKEY identifier SF*10,000 are populated
S_NAME fixed text, size 25
S_ADDRESS variable text, size 40
S_NATIONKEY identifier Foreign key reference to N_NATIONKEY
S_PHONE fixed text, size 15
S_ACCTBAL decimal
S_COMMENT variable text, size 101
Primary Key: S_SUPPKEY

PARTSUPP Table Layout
PS_PARTKEY identifier Foreign key reference to P_PARTKEY
PS_SUPPKEY identifier Foreign key reference to S_SUPPKEY
PS_AVAILQTY integer
PS_SUPPLYCOST decimal
PS_COMMENT variable text, size 199
Compound Primary Key: PS_PARTKEY, PS_SUPPKEY

CUSTOMER Table Layout
C_CUSTKEY identifier SF*150,000 are populated
C_NAME variable text, size 25
C_ADDRESS variable text, size 40
C_NATIONKEY identifier Foreign key reference to N_NATIONKEY
C_PHONE fixed text, size 15
C_ACCTBAL decimal
C_MKTSEGMENT fixed text, size 10
C_COMMENT variable text, size 117
Primary Key: C_CUSTKEY

ORDERS Table Layout
O_ORDERKEY identifier SF*1,500,000 are sparsely populated
O_CUSTKEY identifier Foreign key reference to C_CUSTKEY
O_ORDERSTATUS fixed text, size 1
O_TOTALPRICE decimal
O_ORDERDATE date
O_ORDERPRIORITY fixed text, size 15
O_CLERK fixed text, size 15
O_SHIPPRIORITY integer
O_COMMENT variable text, size 79
Primary Key: O_ORDERKEY
Comment: Orders are not present for all customers. In fact, one-third of the customers do not have any order in the database. The orders are assigned at random to two-thirds of the customers. The purpose of this is to exercise the capabilities of the DBMS to handle "dead data" when joining two or more tables.

LINEITEM Table Layout
L_ORDERKEY identifier Foreign key reference to O_ORDERKEY
L_PARTKEY identifier Foreign key reference to P_PARTKEY, Compound Foreign Key Reference to (PS_PARTKEY, PS_SUPPKEY) with L_SUPPKEY
L_SUPPKEY identifier Foreign key reference to S_SUPPKEY, Compound Foreign key reference to (PS_PARTKEY, PS_SUPPKEY) with L_PARTKEY
L_LINENUMBER integer
L_QUANTITY decimal
L_EXTENDEDPRICE decimal
L_DISCOUNT decimal
L_TAX decimal
L_RETURNFLAG fixed text, size 1
L_LINESTATUS fixed text, size 1
L_SHIPDATE date
L_COMMITDATE date
L_RECEIPTDATE date
L_SHIPINSTRUCT fixed text, size 25
L_SHIPMODE fixed text, size 10
L_COMMENT variable text size 44
Compound Primary Key: L_ORDERKEY, L_LINENUMBER

NATION Table Layout
N_NATIONKEY identifier 25 nations are populated
N_NAME fixed text, size 25
N_REGIONKEY identifier Foreign key reference to R_REGIONKEY
N_COMMENT variable text, size 152
Primary Key: N_NATIONKEY

REGION Table Layout
R_REGIONKEY identifier 5 regions are populated
R_NAME fixed text, size 25
R_COMMENT variable text, size 152
Primary Key: R_REGIONKEY

5.2: Results for the First Method

We did experiments for the following cases:

1- The tradition method: query all the selected data with ignoring the where statement, decrypt the encrypted columns in the where statement, then filter the needed rows that have the values of the where statement.
We marked this method by: DEC_ALL

2- The enhanced hash method: replace the where statement on the encrypted columns with a where statement on the hash value of the searched plain text on the hash columns.
We marked this method by: HASH_METHOD

3- The enhanced encrypt method: replace the where statement on the encrypted columns with a where statement on the encrypt value of the searched plain text.
We marked this method by: ENC_METHOD

The results of each method are listed below in table 5.1

TABLE 5.1: QUERY TIME COST VS. NUMBER OF RECORD FOR THE FIRST METHOD.

No Of Records	100	500	1000	10000
DEC_ALL*	864	4013	6800	47578
HASH_METHOD*	32	35	37	34
ENC_METHOD*	35	31	39	37
HASH_METHOD	5	7	4	6
ENC_METHOD	7	5	5	6
DEC_ALL	821	4189	6882	47565

**Has selected encrypted columns*

In Table 5.1 we measured the query time in mille second when we try to execute a query have a condition on encrypted column for the methods: DEC_ALL, ENC_METHOD and HASH_METHOD when there is a select on encrypted column and when there is no select on the encrypted column. We increase the number of records on the target table from 100 to 10000 and listed the response time for all the methods.

Figure (5.2) shows the cost of query-execution time of the three kinds of querying methods when the size of the data increased from 100 to 100000 records. We measured the time in mille second. The experiments are done for the two cases; with selected encrypted column and without. We mark the results of the experiments with using a select statement having selection on an encrypted column by *.

For example suppose we have two encrypted columns x and y, and the query has a where condition on the encrypted column x, the query which has selection on an encrypted column (which marks by * in Figure 5.2) will be like:

SELECT (y or x) FROM table1 WHERE condition = x;

The query which will not have selection on an encrypted column will be like:

SELECT (any column except x and y) FROM table1 WHERE condition = x;

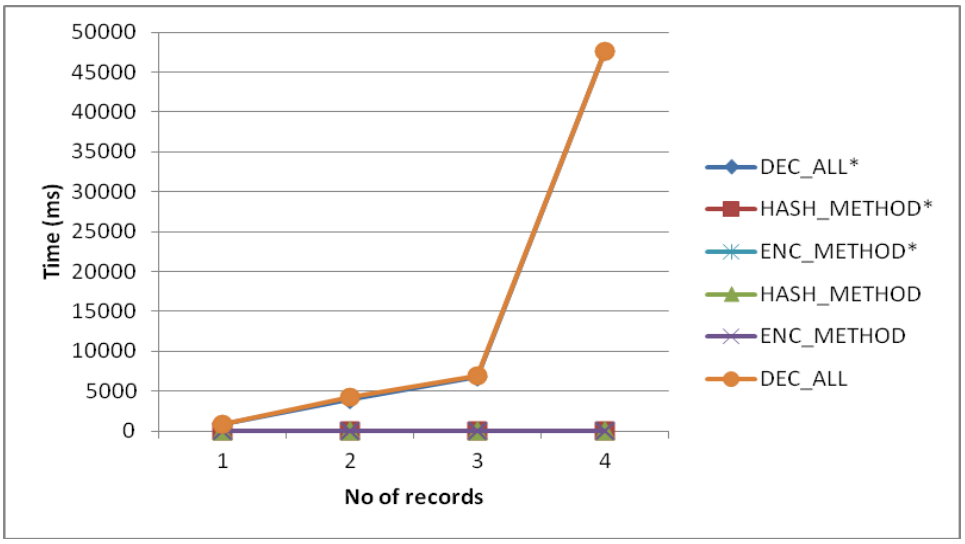

Figure 5.2. Comparing between the first method and the traditional method

We found that DEC_ALL is relatively costly and there is a huge difference between the tradition DEC_ALL method and our methods. This difference is obviously due to the number of records needs to decrypt in each of the methods. In the DEC_ALL, first, all the records in the table needs to be decrypt in the advance, then the decrypted records which are now a plain text have to be filtered as the condition in the where statement. The results of DEC_ALL are related to the number on records in the target table.

The results of HASH_METHOD and DEC_METHOD show that there is a much improvement in the response time in compare with the DEC_ALL method. This improvement due to needing to use the hash or decrypt function one time only, the other operations needed (replacements of the where conditions, … etc) are done in the memory

and need very little time in compare with the time needed when using the hash or decrypt functions.

The number of records in the table does not affect the response time; this is due to using the index so the values are ordered.

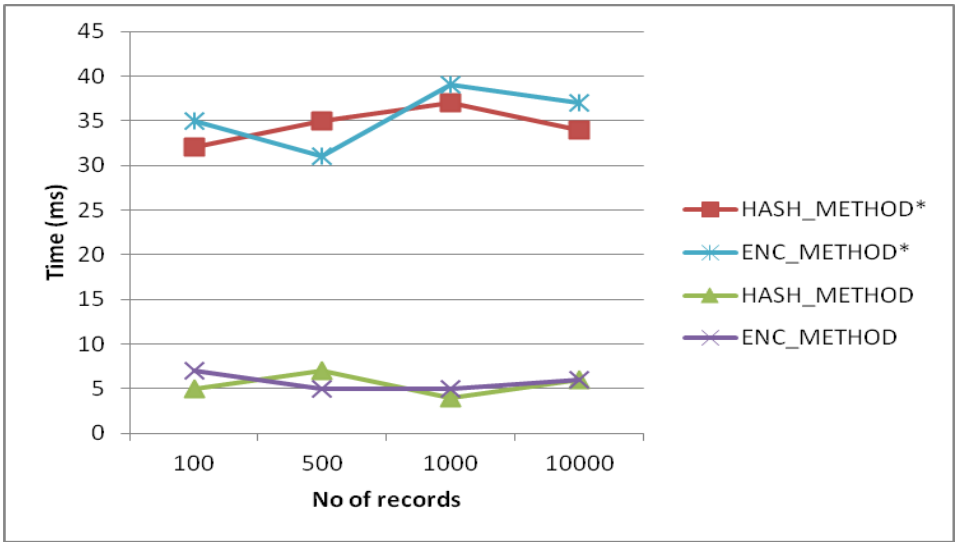

Figure 5.3: Results of executing the same query using HASH_METHOD and ENC_METHOD

In figure (5.3), a comparison in made between the HASH_METHOD and DEC_METHOD, we didn't include the results of DEC_ALL because they are relatively much bigger so the graph will not give us a meaningful view. The results of figure (5.3) show that in the first case, in which the select statement has a select on an encrypted column that the HASH_METHOD* and DEC_METHOD* are relatively equal in response time, this results can be changed if using another encryption algorithm or using the same algorithm (AES) with a smaller key size instead of 256, but of course it will affect the security of the encrypted data. In HASH_METHOD without a select statement on an encrypted column we don't use the cipher key which must be secure and hide in a safe place away from the clients. From the security side we can say is some cases when the select query has a where condition on an encrypted column but doesn't not select any encrypted column we don't use the cipher key which is more secure than using the decryption algorithm. The hash algorithm will cost much when there is an insert or update on a value on the encrypted column because we need an extra insert or update on the hash column, but this case (the insert and update statements) are not studied in this paper and we focus here on the select statement.

5.3: Results for the Second Method

We did experiments for the following cases:

1- The tradition method: query all the selected data with ignoring the where statement, decrypt the encrypted columns in the where statement, then filter the needed rows that have the values of the where statement.
We marked this method by: DEC_ALL

2- The enhanced method: replace the where statement on the encrypted columns with a where statement on the encrypt value of the searched plain text.
We marked this method by: ENH_HASH_METHOD

The results of each method are listed below in table 5.2.

TABLE 5.2: QUERY TIME COST VS. NUMBER OF RECORD FOR THE SECOND METHOD.

No Of Records	100	500	1000	10000
DEC_ALL*	864	4013	6800	47578
ENH_HASH_METHOD*	9	8	8	9
DEC_ALL	821	4189	6882	47565
ENH_HASH_METHOD	4	6	5	6

**Has selected encrypted columns*

Figure (5.4) shows the cost of query-execution time of the two kinds of querying methods when the size of the data increased from 100 to 100000 records. We measured the time in mille second.

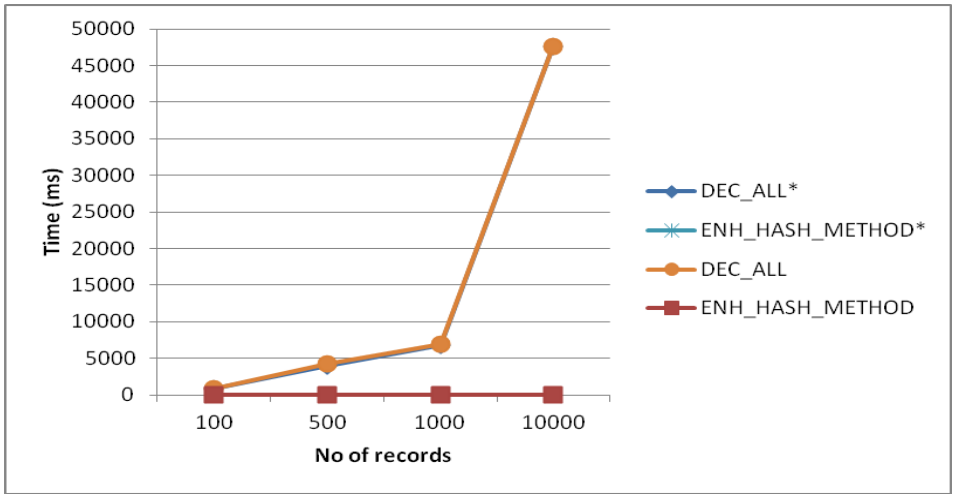

Figure 5.4: Comparing between the second method and the traditional method

The results prove that our method have a better response time in compare with DEC_ALL. This difference is obviously due to the number of records needs to decrypt in each of the methods. In the DEC_ALL, first, all the records in the table needs to be decrypt in the advance, then the decrypted records which are now a plain text have to be filtered as the condition in the where statement. The results of DEC_ALL are related to the number on records in the target table.

The results of ENH_HASH_METHOD show that there is a much improvement in the response time in compare with the DEC_ALL method. This improvement due to needing to use the decryption function one time only, the other operations needed (replacements of the where conditions and search the Hash Map) are done in the memory and need very little time in compare with the time needed when using the decryption functions.

The number of records in the table does not affect the response time; this is due to using the Hash Map to search the needed record in the database table which column is indexed so the values are ordered.

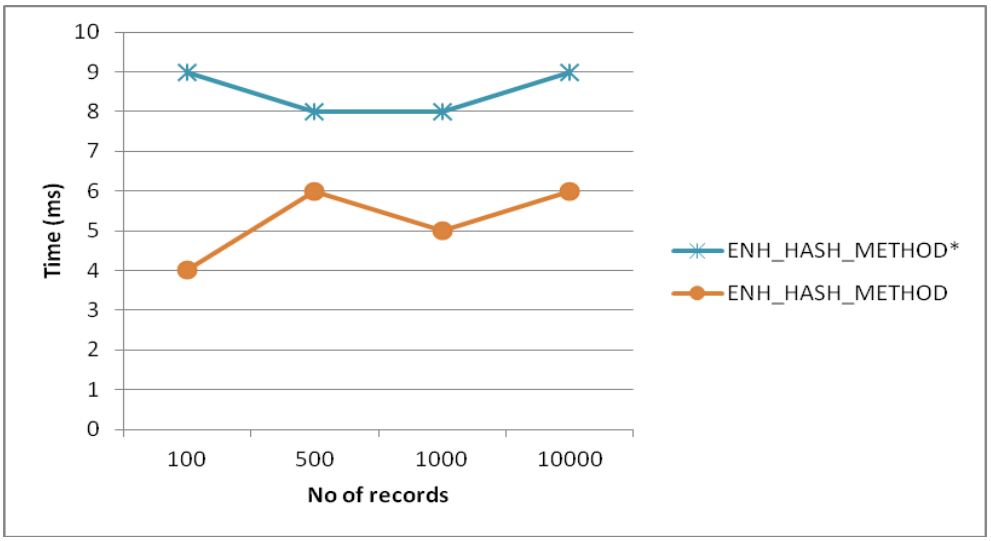

Figure 5.5. Results of executing the same query using ENH_HASH_METHOD

In figure (5.5), a comparison in made between the ENH_HASH_METHOD and ENH_HASH_METHOD*, we didn't include the results of DEC_ALL because they are relatively much bigger so the graph will not give us a meaningful view. The results of figure (5.5) show that in the first case, in which the select statement has a select on an encrypted column that the ENH_HASH_METHOD* have a little more response time due to the time needed to decrypt the encrypted column. Using Hash Map will cost when there is an insert or update on a value on the encrypted column, but will be less than the time needed when use the HASH_METHOD from the first method but this case (the insert and update statements) are not studied in this paper and we focus here on the select statement.

5.4: Results for the third method

The TPC-H schema is used and we work here on the same table of the previous experiments (CUSTOMER) table.

The comparison is done between the traditional method and our method, we used fuzzy query on one character, two characters, three characters and four characters, this characters are located in the start of the word, middle and end of the word; for example:

SELECT * FROM CUSTOMERS
WHERE NAME LIKE 'j%';

SELECT * FROM CUSTOMERS
WHERE NAME LIKE 'jo%';

SELECT * FROM CUSTOMERS
WHERE NAME LIKE 'joh%';

SELECT * FROM CUSTOMERS
WHERE NAME LIKE 'john%';

SELECT * FROM CUSTOMERS
WHERE NAME LIKE '%o%';

SELECT * FROM CUSTOMERS
WHERE NAME LIKE '%oh%';

SELECT * FROM CUSTOMERS
WHERE NAME LIKE '%ohn%';

SELECT * FROM CUSTOMERS
WHERE NAME LIKE '%ohns%';

SELECT * FROM CUSTOMERS

WHERE NAME LIKE '%n';

SELECT * FROM CUSTOMERS

WHERE NAME LIKE '%on';

SELECT * FROM CUSTOMERS

WHERE NAME LIKE '%son';

SELECT * FROM CUSTOMERS

WHERE NAME LIKE '%nson';

The comparison is made as the following:

1- The tradition method: query all the selected data with ignoring the where statement, decrypt the encrypted columns in the where statement, then filter the needed rows that have the values of the where statement.

We marked this method by: DEC_ALL

2- Our new method, translate the fuzzy query on the encrypted column and using our index to find out the target condition, then decrypt the results and filter out them to remove the collision based on the condition.

We marked this method by: FUZZY_METHOD

We make the experiments and write down the results when the number of the characters in the fuzzy condition changes from 1 to 4 and their location changed.

5.4.1: First Group: Number of characters in the fuzzy query = 1

We test the response time when the location of the character was in the start of the word, in the middle of the word and in the last of the word.

First case: when the location of the character is in the start of the word:

TABLE 5.3: QUERY TIME COST VS. NUMBER OF RECORD FOR THE FIRST CASE.

LOCATION = START

No Of Records	100	500	1000	10000
DEC_ALL*	836	4200	6890	47576
FUZZY_METHOD_START*	21	84	252	4530
FUZZY_METHOD_START	15	76	240	4512
DEC_ALL	821	4189	6882	47565

**Has selected encrypted columns*

Figure (5.6) shows the cost of query-execution time of the two kinds of querying methods when the size of the data increased from 100 to 100000 records. We measured the time in mille second. The experiments are done for the two cases; with selected encrypted column and without. We mark the results of the experiments with using a select statement having selection on an encrypted column by *.

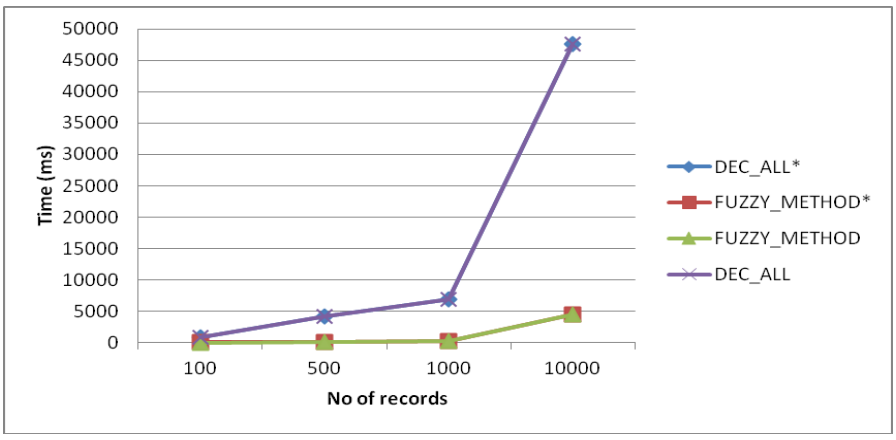

Figure 5.6: Results of executing the same query using the traditional method and FUZZY_METHOD_START for number of characters =1

The results of FUZZY_METHOD show that there is a much improvement in the response time in compare with the DEC_ALL method. This improvement due to needing decrypt few records in compare with DEC_ALL which needs to decrypts all the records. The time needed to build and hide the index and translate the condition to work over the index is much slower than the time needed to decrypt all the records.

Analysis:

The number of records in the table affects the response time; this is due to the response time for the operation of translating the condition to work over our index

INDEX_ (FUNCTION) $^{-1}$ Key LIKE 'H(G, X58@) %'

Directly proportional with the number of rows in the table.

The another issues is the collision problem, when the number of rows increases the collision increases that's because the probability of finding the same characters increases when the number of rows increase, that's increase the number of rows need to be decrypted and filtered out to remove the collision.

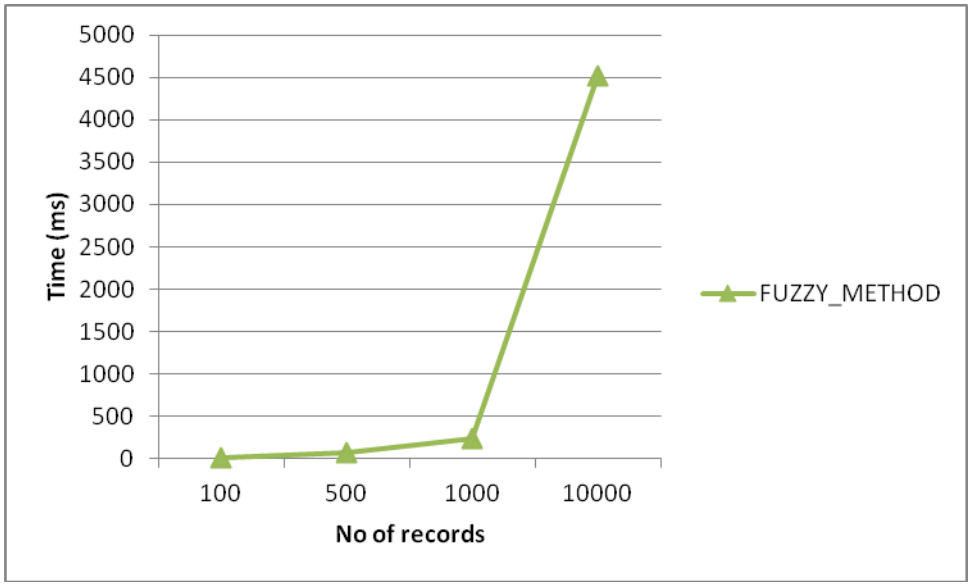

Figure 5.7: Results of executing the same query using FUZZY_METHOD_START for number of characters =1

In figure (5.7), we draw the results of FUZZ_QUERY_START, we found that after the size of the table exceed 1000 records the response time increases fast, this due to the problem of collision and increasing the number of rows.

Second case: when the location of the character is in the middle of the word:

TABLE 5.4: QUERY TIME COST VS. NUMBER OF RECORD FOR THE SECOND CASE

No Of Records	100	500	1000	10000
DEC_ALL*	836	4200	6890	47576
FUZZY_METHOD-MIDDLE*	56	140	510	7211
FUZZY_METHOD-MIDDLE	50	124	450	6211
DEC_ALL	821	4189	6882	47565
FUZZY_METHOD-START	15	76	240	4512

**Has selected encrypted columns*

Figure (5.8) shows the cost of query-execution time of the two kinds of querying methods when the size of the data increased from 100 to 100000 records.

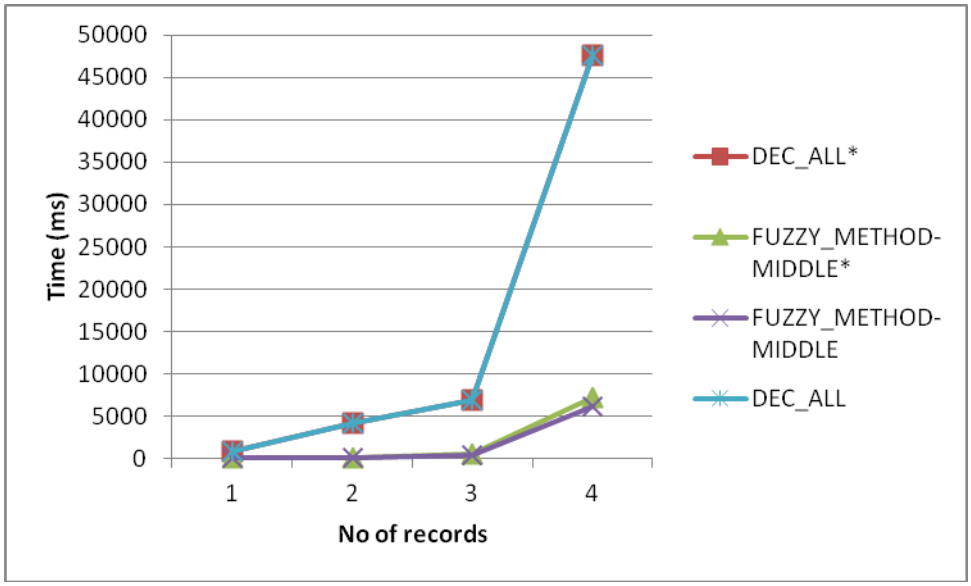

Figure 5.8. Results of executing the same query using the traditional method and FUZZY_METHOD_MIDDLE for number of characters =1

Analysis:

We found that the response time increases when searching the character in the middle of the word, that's because the probability of finding this character in the middle of the word is higher than the probability of finding the same character in the start of the word which increases the collision. This issue is also a data related. We work here on the data generated by a standard benchmark.

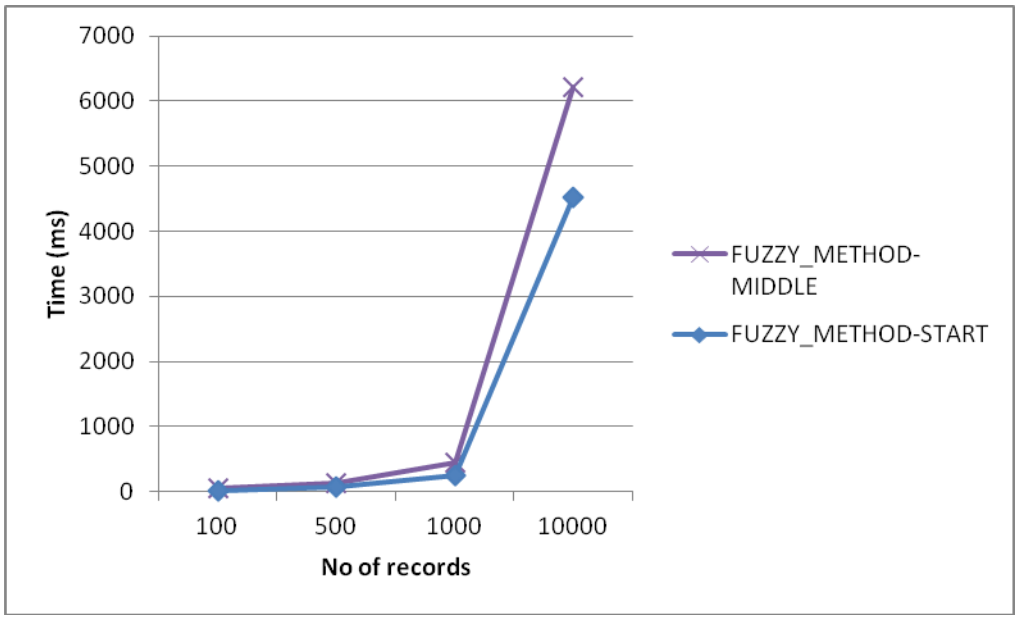

Figure 5.9: Comparing between FUZZY_METHOD-START and FUZZY_METHOD-MIDDLE

Third case: when the location of the character is in the end of the word:

Table 5.5: Query time cost vs. Number of record for the third case.

No Of Records	100	500	1000	10000
DEC_ALL*	836	4200	6890	47576
FUZZY_METHOD-MIDDLE*	56	140	510	7211
FUZZY_METHOD-MIDDLE	50	124	450	6211
DEC_ALL	821	4189	6882	47565
FUZZY_METHOD-START	15	76	240	4512
FUZZY_METHOD-END	15	55	212	3800
FUZZY_METHOD-END*	22	66	235	4400
FUZZY_METHOD*	21	84	252	4530

**Has selected encrypted columns*

Figure (5.10) shows the cost of query-execution time of the two kinds of querying methods when the size of the data increased from 100 to 100000 records.

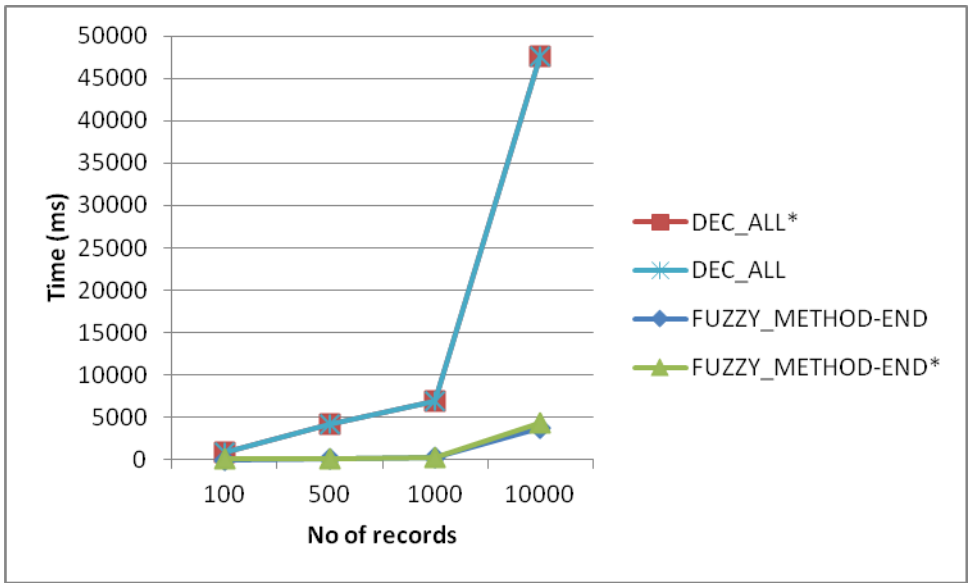

Figure 5.10: Results of executing the same query using the traditional method and FUZZY_METHOD_END for number of characters =1

Analysis:

We found that the response time decreases when searching the character in the end of the word, that's because the probability of finding this character in the end of the word is smaller than the probability of finding the same character in the start or middle of the word which decreases the collision figure 5.11. This issue is also a data related. We work here on the data generated by a standard benchmark.

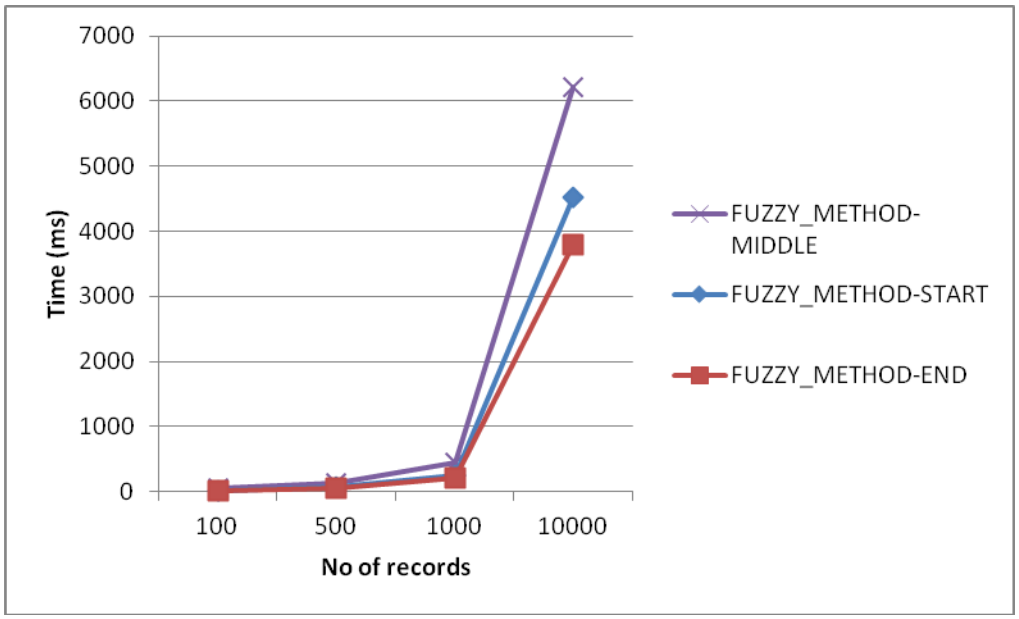

Figure 5.11: Comparing between FUZZY_METHOD_START, FUZZY_METHOD_MIDDLE and FUZZY_METHOD_END

5.4.2: Second Group: Number of characters in the fuzzy query = 2

We test the response time when the location of the character was in the start of the word, in the middle of the word and in the last of the word.

First case: when the location of the character is in the start of the word:

SELECT * FROM CUSTOMERS

WHERE NAME LIKE ‘jo%’;

Table 5.6: Query time cost vs. Number of record. LOCATION=START; LENGTH=2

No Of Records	100	500	1000	10000
FUZZY_METHOD-START	13	66	200	4120
DEC_ALL	821	4189	6882	47565

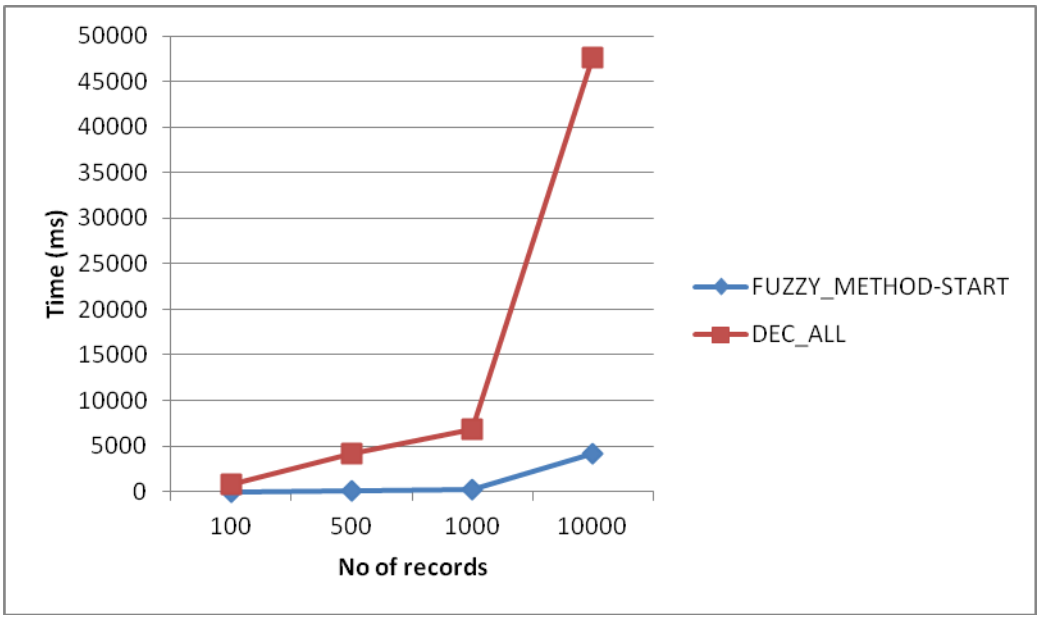

Figure 5.12: Results of executing the same query using the traditional method and FUZZY_METHOD_START for number of characters =2

Second case: when the location of the character is in the middle of the word:

SELECT * FROM CUSTOMERS

WHERE NAME LIKE '%oh%';

Table 5.7: Query time cost vs. Number of record LOCATION = MIDDLE; LENGHT = 2

No Of Records	100	500	1000	10000
FUZZY_METHOD-MIDDLE	40	110	390	4982
DEC_ALL	821	4189	6882	47565

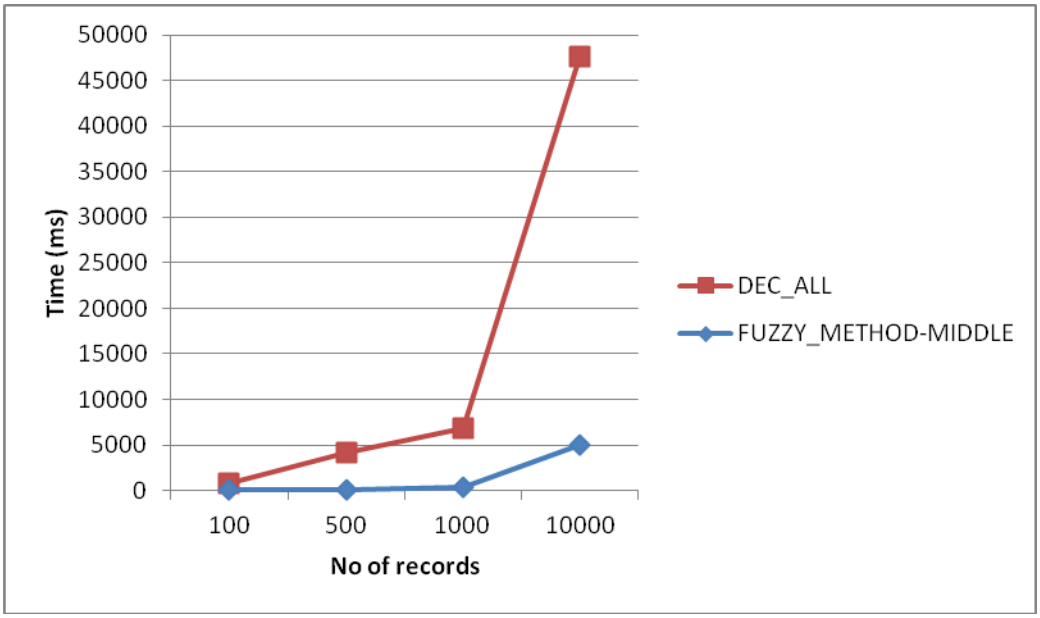

Figure 5.13: Results of executing the same query using the traditional method and FUZZY_METHOD_MIDDLE for number of characters =2

Third case: when the location of the character is in the end of the word:

SELECT * FROM CUSTOMERS

WHERE NAME LIKE '%on';

Table 5.8: Query time cost vs. Number of record LOCATION = END; LENGHT = 2

No Of Records	100	500	1000	10000
FUZZY_METHOD-END	12	56	175	3965
DEC_ALL	821	4189	6882	47565

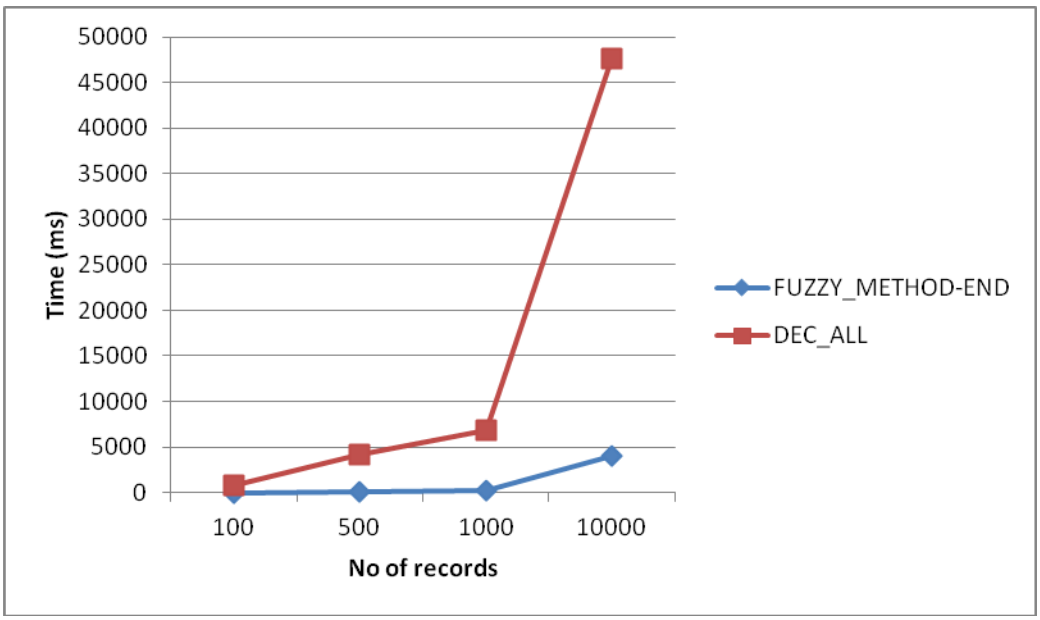

Figure 5.14: Results of executing the same query using the traditional method and FUZZY_METHOD_END for number of characters =2

Comparison between three methods

Table 5.9: Query time cost vs. Number of record LOCATION = START, MIDDLE,END; LENGHT = 2

No Of Records	100	500	1000	10000
FUZZY_METHOD-START	13	66	200	4120
FUZZY_METHOD-MIDDLE	40	110	390	4982
FUZZY_METHOD-END	12	56	175	3965

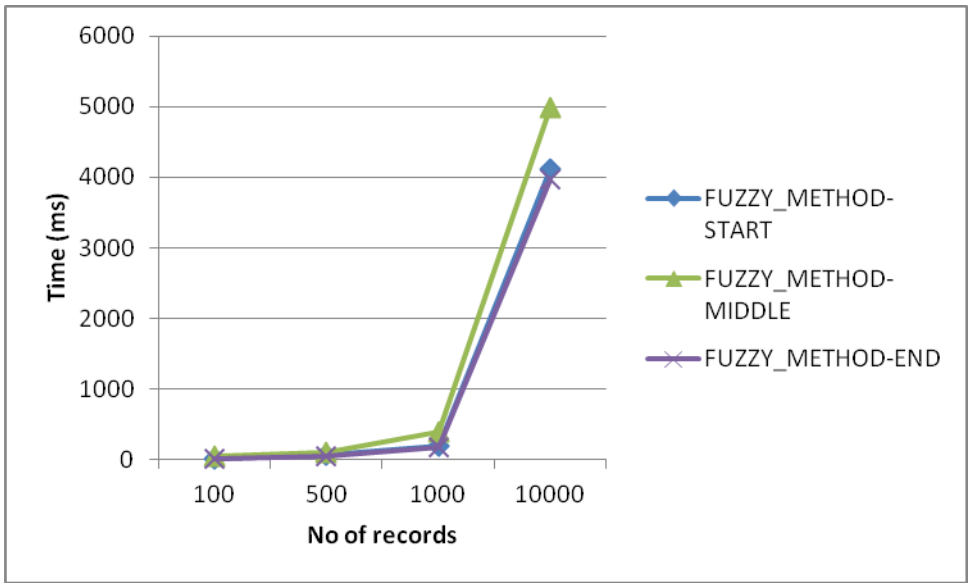

Figure 5.15: Results of executing the same query using the FUZZY_METHOD_START, FUZZY_METHOD_MIDDLE and FUZZY_METHOD_END for number of characters =2

Analysis:

We found that the response time decreases when searching the character in the end of the word, that's because the probability of finding this character in the end of the word is smaller than the probability of finding the same character in the start or middle of the word which decreases the collision. This issue is also a data related. We work here on the data generated by a standard benchmark. This result is like the results of the first group.

We found that when the number of characters increases the execution time of FUZZY_QUERY is decrease, that's because the probabilities of finding the same characters in the same order decreases so the collision decreases.

5.4.3: Third Group: Number of characters in the fuzzy query = 3

We test the response time when the location of the character was in the start of the word, in the middle of the word and in the last of the word.

First case: when the location of the character is in the start of the word:

SELECT * FROM CUSTOMERS

WHERE NAME LIKE ‘joh%’;

Table 5.10: Query time cost vs. Number of record LOCATION = START; LENGHT = 3

No Of Records	100	500	1000	10000
FUZZY_METHOD-START	12	51	165	3502
DEC_ALL	821	4189	6882	47565

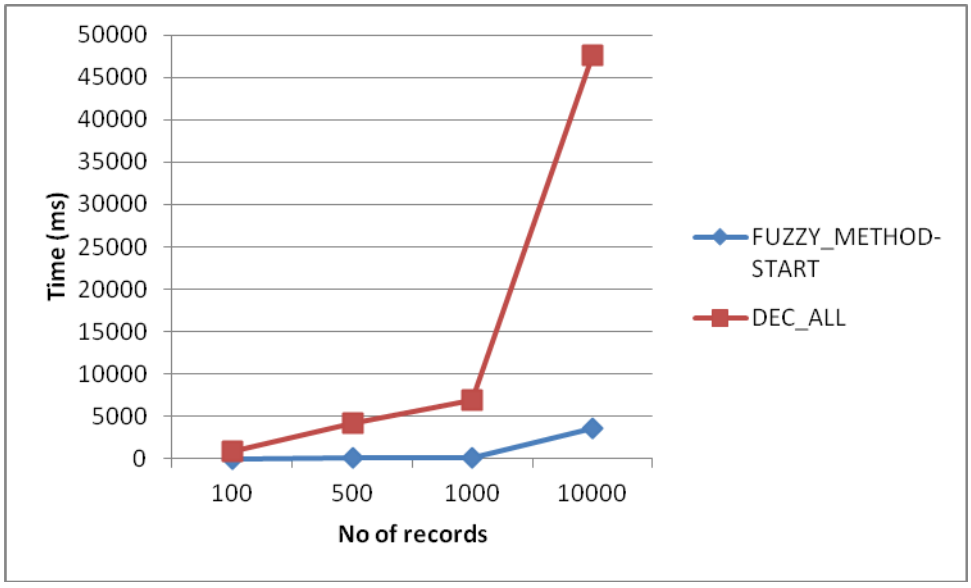

Figure 5.16: Results of executing the same query using the traditional method and FUZZY_METHOD_START for number of characters =3

Second case: when the location of the character is in the middle of the word:

SELECT * FROM CUSTOMERS

WHERE NAME LIKE '%ohn%';

Table 5.11: Query time cost vs. Number of record LOCATION = MIDDLE; LENGHT = 3

No Of Records	100	500	1000	10000
FUZZY_METHOD-MIDDLE	33	86	301	4210
DEC_ALL	821	4189	6882	47565

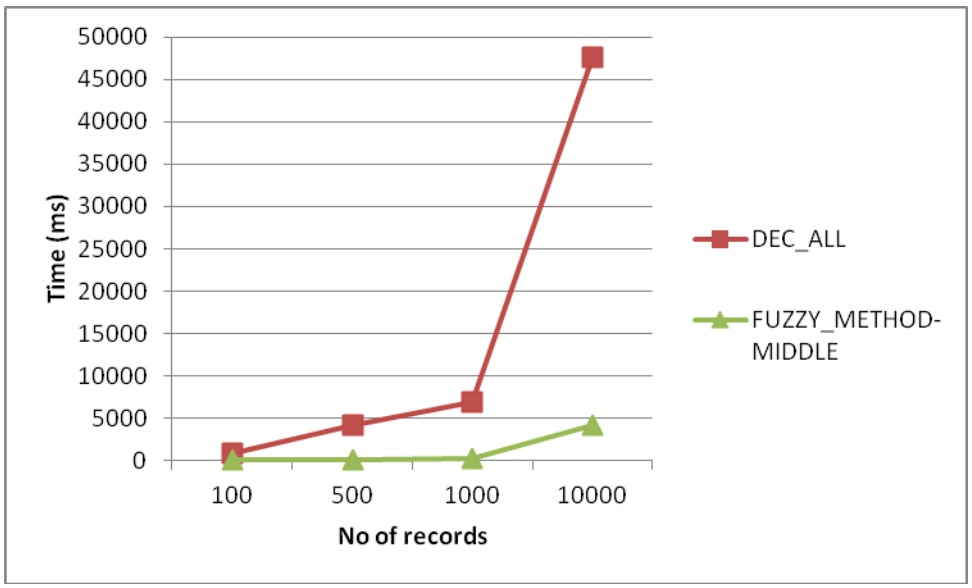

Figure 5.17: Results of executing the same query using the traditional method and FUZZY_METHOD_MIDDLE for number of characters =3

Third case: when the location of the character is in the end of the word:

SELECT * FROM CUSTOMERS

WHERE NAME LIKE '%son';

Table 5.12: Query time cost vs. Number of record LOCATION = END; LENGHT = 3

No Of Records	100	500	1000	10000
FUZZY_METHOD-END	12	45	134	3215
DEC_ALL	821	4189	6882	47565

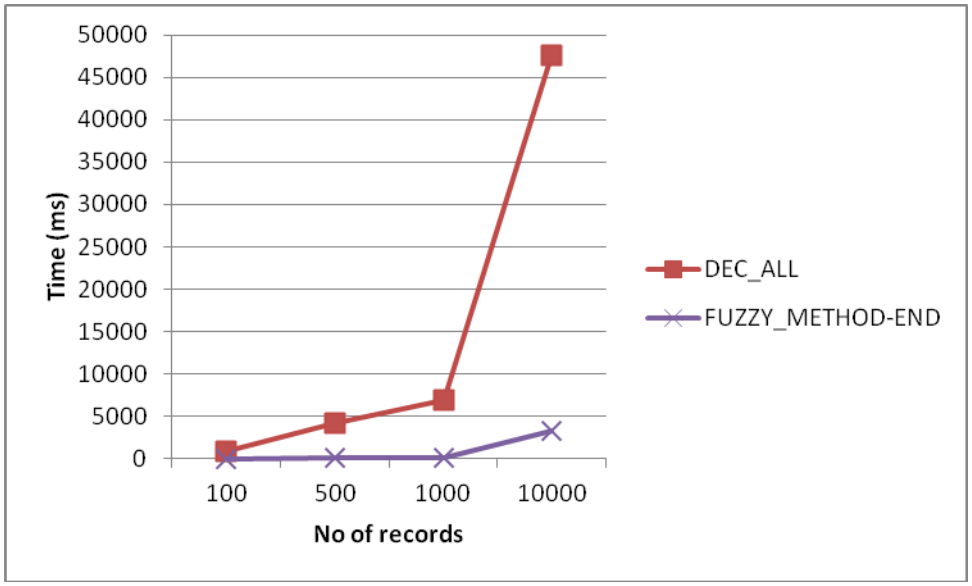

Figure 5.18: Results of executing the same query using the traditional method and FUZZY_METHOD_END for number of characters =3

Comparison between three methods

Table 5.13: Query time cost vs. Number of record LOCATION = START, MIDDLE, END; LENGHT = 3

No Of Records	100	500	1000	10000
FUZZY_METHOD-START	12	51	165	3502
FUZZY_METHOD-MIDDLE	33	86	301	4210
FUZZY_METHOD-END	12	45	134	3215

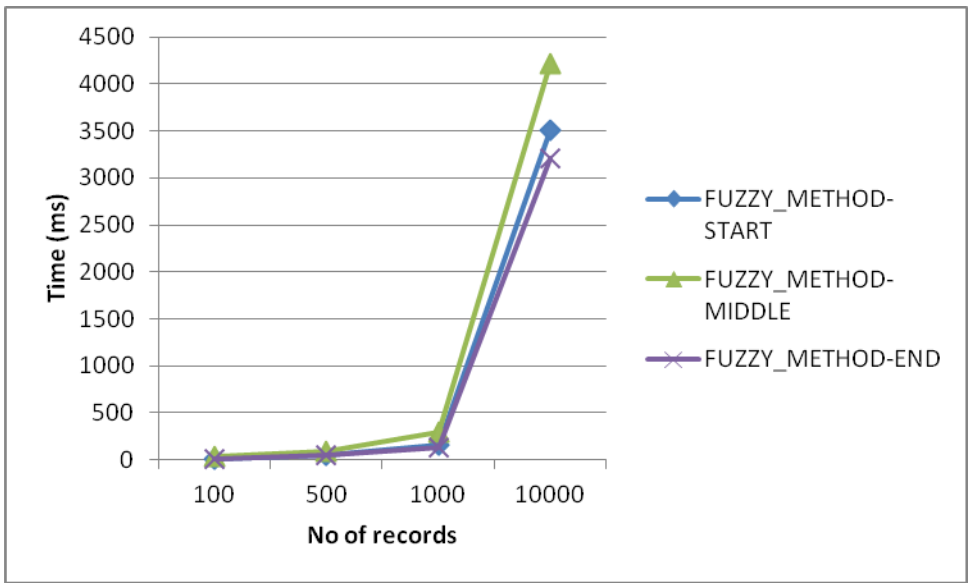

Figure 5.19: Results of executing the same query using FUZZY_METHOD_START, FUZZY_METHOD_MIDDLE AND FUZZY_METHOD_END for number of characters =3

5.4.4: Fourth Group: Number of characters in the fuzzy query = 4

We test the response time when the location of the character was in the start of the word, in the middle of the word and in the last of the word.

First case: when the location of the character is in the start of the word:

SELECT * FROM CUSTOMERS

WHERE NAME LIKE 'john%';

Table 5.14: Query time cost vs. Number of record LOCATION = START; LENGHT = 4

No Of Records	100	500	1000	10000
FUZZY_METHOD-START	7	36	70	1540
DEC_ALL	821	4189	6882	47565

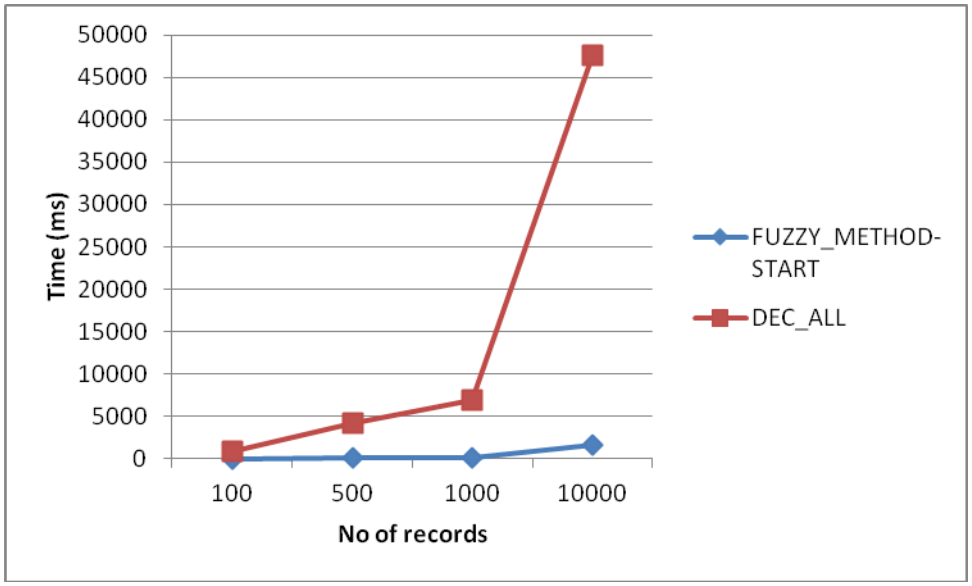

Figure 5.20: Results of executing the same query using traditional method and FUZZY_METHOD_START for number of characters =4

Second case: when the location of the character is in the middle of the word:

SELECT * FROM CUSTOMERS

WHERE NAME LIKE '%ohns%';

Table 5.15: Query time cost vs. Number of record LOCATION = MIDDLE; LENGHT = 4

No Of Records	100	500	1000	10000
FUZZY_METHOD-MIDDLE	12	46	120	2541
DEC_ALL	821	4189	6882	47565

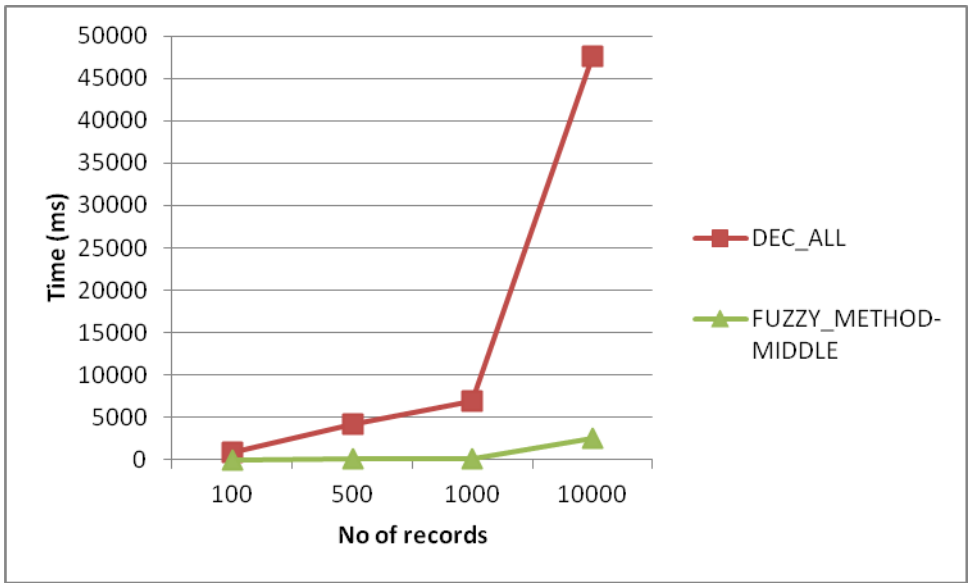

Figure 5.21: Results of executing the same query using traditional method and FUZZY_METHOD_MIDDLE for number of characters =4

Third case: when the location of the character is in the end of the word:

SELECT * FROM CUSTOMERS

WHERE NAME LIKE '%nson';

Table 5.16: Query time cost vs. Number of record LOCATION = END; LENGHT = 4

No Of Records	100	500	1000	10000
FUZZY_METHOD-END	6	24	36	541
DEC_ALL	821	4189	6882	47565

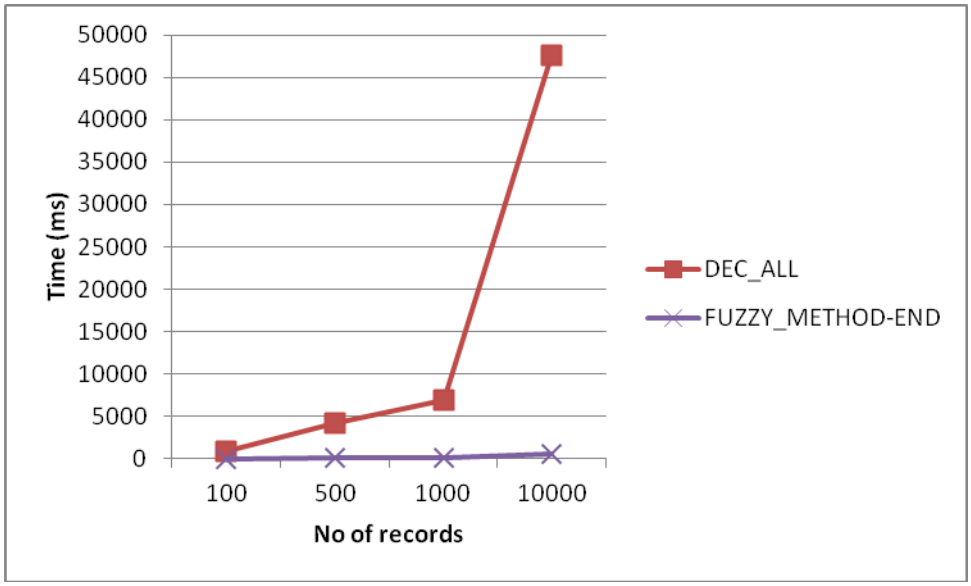

Figure 5.22: Results of executing the same query using traditional method and FUZZY_METHOD_END for number of characters =4

Comparison between three methods

Table 5.17: Query time cost vs. Number of record LOCATION = START, MIDDLE, END; LENGHT = 4

No Of Records	100	500	1000	10000
FUZZY_METHOD-START	7	36	70	1540
FUZZY_METHOD-MIDDLE	12	46	120	2541
FUZZY_METHOD-END	6	24	36	541

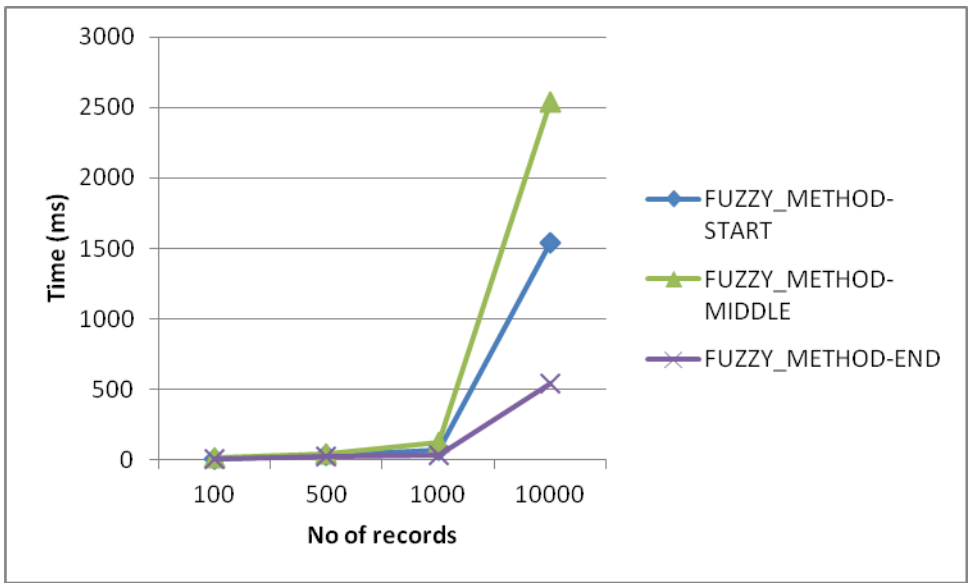

Figure 5.23: Results of executing the same query using FUZZY_METHOD_START , FUZZY_METHOD_MIDDLE and FUZZY_METHOD_END for number of characters =4

5.4.5: Comparison between the four groups

We made a comparison between the three cases when the location in the character is in the start and middle and end for each of the three groups LENGTH = 1, 2, 3, 4

First Case: Start of the word

Table 5.18: Query time cost vs. Number of record LOCATION = START; LENGHT = 1,2,3,4

No Of Records	100	500	1000	10000
FUZZY_METHOD-START = 1	15	76	240	4512
FUZZY_METHOD-START = 2	13	66	200	4120
FUZZY_METHOD-START = 3	12	51	165	3502
FUZZY_METHOD-START = 4	7	36	70	1540

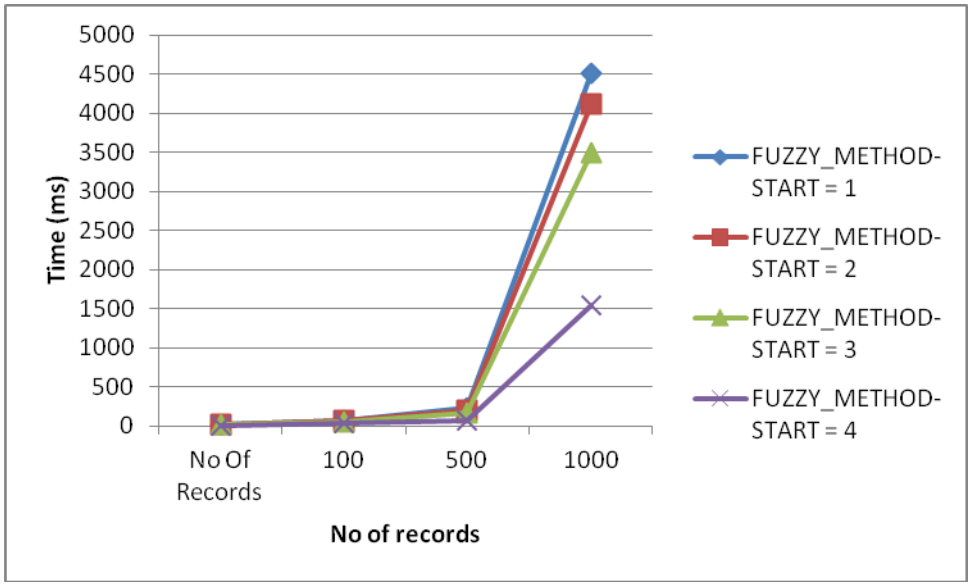

Figure 5.24: Results of executing the same query using FUZZY_METHOD_START for number of characters =1,2,3,4

Second case: Middle of the word

Table 5.19: Query time cost vs. Number of record LOCATION = MIDDLE; LENGHT = 1,2,3,4

No Of Records	100	500	1000	10000
FUZZY_METHOD-MIDDLE = 1	50	124	450	6211
FUZZY_METHOD-MIDDLE = 2	40	110	390	4982
FUZZY_METHOD-MIDDLE = 3	33	86	301	4210
FUZZY_METHOD-MIDDLE = 4	12	46	120	2541

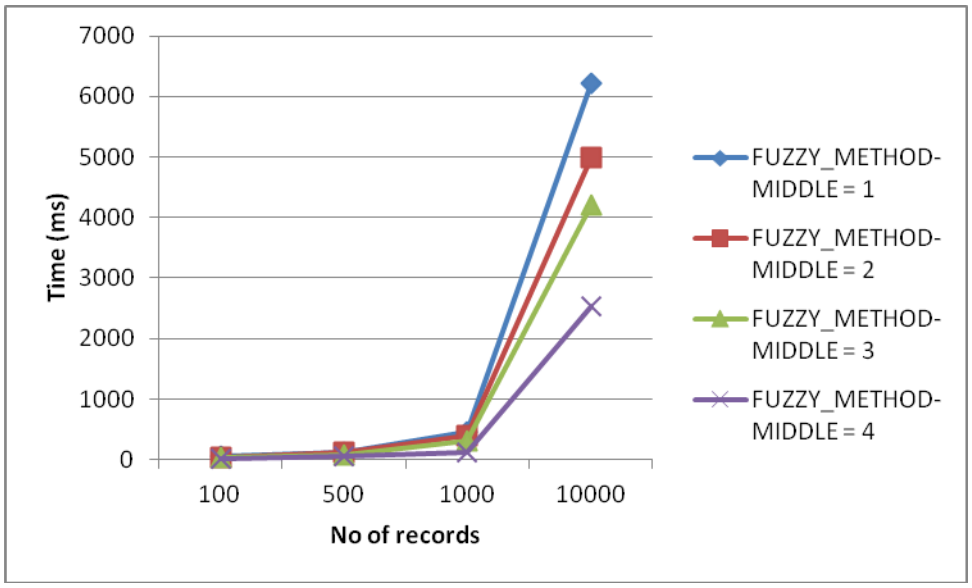

Figure 5.25: Results of executing the same query using FUZZY_METHOD_MIDDLE for number of characters =1,2,3,4

Third Case: End of the word

Table 5.20: Query time cost vs. Number of record LOCATION = END; LENGHT = 1,2,3,4

No Of Records	100	500	1000	10000
FUZZY_METHOD-END = 1	15	55	212	3800
FUZZY_METHOD-END = 2	12	56	175	3965
FUZZY_METHOD-END = 3	12	45	134	3215
FUZZY_METHOD-END = 4	6	24	36	541

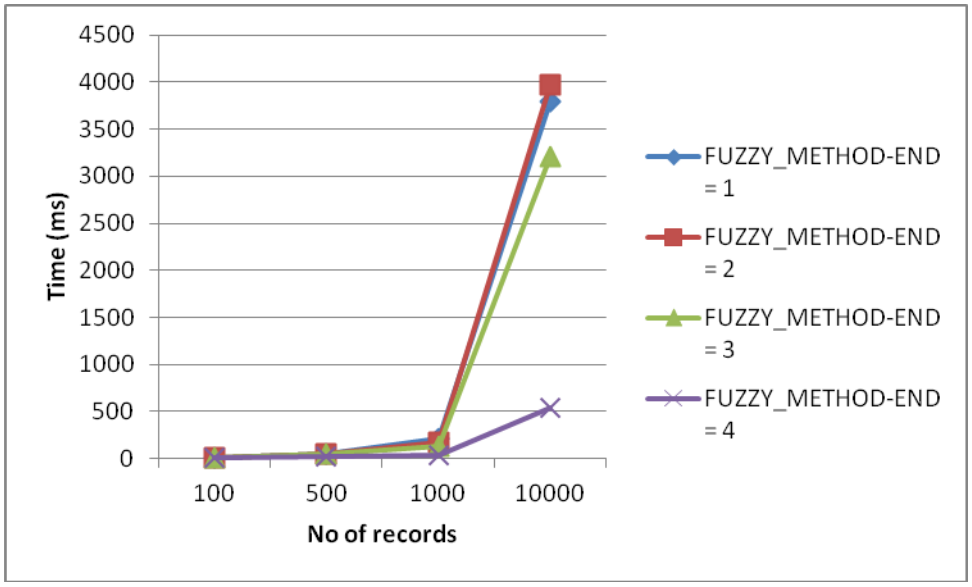

Figure 5.26: Results of executing the same query using FUZZY_METHOD_END for number of characters =1,2,3,4

Analysis:

According to the results we found that when the length of the characters increases the response time enhanced and decreses that is due to decreasing the collision, that's means that the results rows need to be filtered out is decresed.
This can be proved by the following:
If we limit the range of charaters to be from a-z that is 26 characters.
If the characters are distributed randomly and have the same probapility each character have the probapility of 1/26 = 0.0385, by using our fuzzy method and by limit the group of numeric index to be from 0 – 9 that is 10 numbers. So the probapilty of having a collision for each chracter is 10/26 = 0.385

The probabilty of having a character x and collision is:

0.0385 * 0.385 = 0.0148225

If the number of concatanated characters to be find is 2 then the probability of having a collision is :

0.385 * 0.385 = 0.148225 < 0.385

That’s proves when the number of characters increases the probability of collison decreases and the response time enhanced.

Chapter Six

Conclusion and Future Work

6.1 Summary and Concluding Remarks

We proposed three new methods of query over encrypted data in databases, our methods did not affect the inner structure of the DBMS because it was implemented as a layer above the DBMS, this layer hided the details of the DBMS so we did not need to have an open source database. This point makes that our methods can adapt with any kind of the known DBMS like MS SQL, Oracle and MS Access.. Etc. We implemented our layer by using the Delphi as a programming language, for each of the proposed ways there were a common components and for every one there were a specific components. A universal benchmark TPC-H was used to implement the layout and the tables and its relations and the data was generated according to this benchmark. For the first method we used one way functions in our case this were the AES and SHA-1, the query from the client changed in the layer to be able to work over the encrypted data. The results of this method provided a huge enhancement in the response time for the query over encrypted data when used this way. We proved this by tested the results of experiments that were measuring the response time for the method when the number of records in the database changed. In the second method we used the data structure HASH MAP, this object was used to enhance the response time for the first method by storing the mapping of plain text and encrypted text in the memory. The Hash Map stored the mapping between the plain text and the encrypted text as KEY: VALUE, in which the KEY is the plain text and the VALUE is the encrypted value of the plain text. The Hash Map contains two main operations, PutValue and GetValue. PutValue(Key , Value), GetValue(Key): Value. This method needed a huge memory, so this method is not adapt with a large database contains tables have many records; this method can work in distributed systems environment.

The results of our experiments for the two first methods provided that the second method have a better response time, this was due the second method worked on the memory and the first method worked on the hard disk which is slower than the memory.

In the third method we worked on the fuzzy queries. We built an index over the target plain text and mapped the plain text to numeric values to be the index for the plain text which was encrypted, and then we hided the structure of this index. This process must have a response time faster than the time needed for the encryption, we did this by using a simple functions and the experiments shows that there was a noticed enhancement in the response time in compared with the traditional way when the number of records in the target table increased. We gave detailded examples on how to build the index and how to query the data.

6.2 Recommendations and Future Work

In the future we will try to fix the limitations of the first method which are the extra time and hard disk space needed for insert or update a new record in the database and the issue that this method works fine with the equal condition in the database and does not work with the greater or smaller conditions on numeric data and does not work also with the fuzzy query. The second method also does not work with the greater or smaller conditions on numeric data and does not work with the fuzzy query, the second method also needs a huge memory when the number of records in the database is large so this method is not adapted with the large database size, in future we will try to fix this problem. In future we will study how to solve the collision problem for third method and study other cases when the target characters need to be found is not beside each other's and we will try to make our method more secure by making the hiding of the index a more complex operation. We will also try to make our methods work with the join query and study how to adapt our methods with the more complex queries.

References

[1] Erez Shmueli, Ronen Vaisenberg, Yuval Elovici and Chanan Glezer, "Database Encryption – An Overview of Contemporary Challenges and Design Considerations" *SIGMOD Record, September* 2009 (Vol. 38, No. 3)

[2] Dawn Xiaodong Song, David Wagner, and Adrian Perring. Practical Techniques for Searches on Encrypted Data, *IEEE Symposium on Security and Privacy*, 2000, pp. 44-55.

[3] H. Hacigumus , Bala Iyer and Sharad Mehrotra, "Providing Database as a Service", *Data Engineering, Proceedings. 18th International Conference , 2002*

[4] H. Hacigumus, B. Iyer, C. Li and S. Mehrotra, "Executing SQL over encrypted data in the database service provider model," In *ACM SIGMOD Conference*, 2002, pp. 216-227.

[5] H. Hacigumus, B. Iyer, and S. Mehrotra. "Efficient execution of aggregation queries over encrypted relational databases". In the *proceedings of Database Systems for Advanced Applications* (DASFAA), 2004, pp. 125-136

[6] B. Hore, S. Mehrotra and G. Tsudik. "A Privacy-Preserving Index for Range Queries". In *Proceedings of the 30th VLDB Conference*, 2004, pp. 720–731.

[7] Z. Wang, J. Dai, W. Wang and B.L. Shi, "Fast Query over Encrypted Character Data in Database". *Communications In Information and Systems*, 2004, pp.289-300

[8] Zheng-Fei Wang, Wei Wang and Bai-Le Shi , "Storage and Query over Encrypted Character and Numerical Data in Database", *Computer and Information Technology. The Fifth International Conference, 2005*

[9] H. Zhu, J. Cheng and R. Jin, "Execution Query over Encrypted Character Strings in Databases," Frontier of Computer Science and Technology, 2007, pp. 90-97

[10] H. APark, D. Lee, J. Zhan and G. Blosser, "Efficient Keyword Index Search over Encrypted Documents of Groups" *ISI 2008*, June 17-20

[11] Yu Han, Zhao Liang Niu Xiamu, "Research on a new method for database encryption and cipher index". Acta Electronica Sinica, No. 12A 2005

[12] Z. Wang, A. Tang and W. Wang, "Fast Query over Encrypted Data Based on b+ Tree", *International Conference on Apperceiving Computing and Intelligence Analysis* (ICACIA), 23-25 Oct. 2009.

[13] Bertino, E.; Sandhu, R., "Database security – concepts, approaches and challenges", *IEEETransactions on Dependable and Secure Computing*, VOL. 2, NO. 1, JANUARY-MARCH 2005

[14] S. Sesay, Z. Yang, J. Chen and D. Xu, "A Secure Database Encryption Scheme". *Consumer Communications and Networking Conference* (CCNC), 2005, pp. 49-53

[15] W. Baohua, M. Xiniang and L. Danning, "A Formal Mutilevel Database Security Model", *IEEE International Conference on Computational Intelligence and Security*, 13-17 Dec. 2008

[16] Y. Zhang, W. Li and X. Niu, "A Method of Bucket Index over Encrypted Character Data in Database". *Intelligent Information Hiding and Multimedia Signal Processing*, 2007, pp. 186-189

[17] Michael Mitzenmacher , "Compressed Bloom Filters", *IEEE/ACM Transactions on Networking*, VOL. 10, NO. 5, October 2002

[18] Jehoshua Bruck , Jie Gao and Anxiao (Andrew) Jiang, "Weighted Bloom Filter" *ISIT 2006*, Seattle, USA, July 9 14, 2006

[19] Yong Zhang, Wei-xin Li and Xia-Mu Niu, "A Secure Cipher Index Over Encrypted Character Data in Database", *Proceedings of the Seventh International Conference on Machine Learning and Cybernetics*, Kunming, 12-15 July 2008

[20] Lianzhong Liu and Jingfen Gai, "Bloom Filter Based Index for Query over Encrypted Character Strings in Database", World Congress on Computer Science and Information Engineering, 2009

[21] Yong Soon KIM and Eui Kyeong Hong, "Considerations of Extending SQL on Encrypted Data in UniSQL", *Advanced Communication Technology, The 9th International Conference* on 12-14 Feb. 2007

[22] Premchand B. Ambhore,B.B.Meshram and V.B.Waghmare "A Implementation of Object Oriented Database Security", *Software Engineering Research, Management & Applications. 5th ACIS International Conference , 2007*

[23] Yu Chen and Wesley W. Chu, Fellow "Protection of Database Security via Collaborative Inference Detection", *IEEE Transactions on Knowledge and Data Engineering*, VOL. 20, NO. 8, August 2008

[24] Sohail IMRAN and Irfan Hyder, "Security Issues in Databases", *Second International Conference on Future Information Technology and Management Engineering, 2009*

[25] Xu Ruzhi, Guo jian and Deng Liwu, "A Database Security Gateway to the Detection of SQL Attacks", *3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, 2010

[26] Wikipedia, the free encyclopedia. Advanced Encryption Standard [Online]. Available: http://en.wikipedia.org/wiki/Advanced_Encryption_Standard [10/11/2012]

[27] Wikipedia, the free encyclopedia. Secure Hash Algorithm [Online]. Available: http://en.wikipedia.org/wiki/SHA-1[12/11/2012]

[28] Wikipedia, the free encyclopedia. Hash Table [Online]. Available: http://en.wikipedia.org/wiki/Hash_table [14/11/2012]