Query Processing in Encrypted Cloud Databases

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Engineering

in

Computer Science and Engineering

by

Akshar Kaul

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

July 2013

TO

My Family and Friends

Acknowledgements

I would like to thank my advisor Prof. Jayant R. Haritsa for his encouragement and unmatched guidance throughout my project work. I have been extremely fortunate to work with him.

I would like to thank all my fellow labmates for their help and suggestions. Also I would like to thank all my CSA friends who have made my stay at IISc fun and memorable.

Finally, I am indebted with gratitude to my parents for their love and inspiration that no amount of thanks can suffice.

Abstract

With the increasing popularity of cloud computing, a new model known as Database-as-a-service (DAAS) model has emerged in recent years. In this model a service provider provides the database as a service to its clients over internet. One of the biggest problem in DAAS is "How to execute queries efficiently at the cloud resident server while maintaining data security". In this work we present ***PhantomDB****, which is a new framework for solving this problem. PhantomDB maintains data security by encrypting the data before storing it on the server. Each data item is encrypted under multiple encryption schemes. The Encryption schemes used by PhantomDB are carefully chosen to allow efficient query processing at the server. PhantomDB introduces the concept of Arithmetic Engine and Round Communication, which allow it to support all the standard SQL constructs with the exception of similarity operators. PhantomDB is able to support 16 out of 22 TPCH Benchmark queries. We also present a new hybrid storage model which takes unique features of PhantomDB into consideration while deciding the layout of relational data on disk. It helps in improving the performance of PhantomDB as compared to other current storage models.*

Contents

List of Tables

List of Figures

Keywords

Database-as-a-service, DAAS, Query Processing, encrypted cloud database.

Chapter 1

Introduction

In today's world cloud computing has acquired a very big role. It has led to the emergence of a new business model known as "Database-as-a-service" (*DAAS*) model [10]. In this model the "Database Service Providers" (*DSP*) provide database services to customers over the internet. DSP exposes its database servers to the client over internet. The client access these database servers using standard SQL interface. Client only has to configure his applications to use the database servers hosted by DSP, instead of database servers managed locally. The fact that DAAS model is gaining a lot of favor in industry is evident from emergence of cloud based SQL platforms from various vendors such as *Microsoft SQL Azure, Amazon RDS* etc. The rise in popularity of DAAS is due to numerous benefits it provides to client. Most prominent among these are:

- Client does not have to purchase expensive hardware and software. All these are owned by service provider. Since service provider buys these in bulk, economies of scale kick in and per unit cost decreases.
- All upgrades to software and hardware are handled by the service provider.
- Client does not have to hire professionals for administration and maintenance tasks (such as DB backup, restore, relocation).
- DAAS model allows *pay per usage*. Client only pays for the amount of service

which he uses. This is not possible in traditional set up (where client has to pay a fixed periodic cost to maintain the whole infrastructure).

The DAAS model, however, has its own set of challenges which are very different from those of the traditional databases. The main challenge in DAAS model is to ensure data security. Data security addresses the problem of protecting data stored at service provider from various threats. All corporations view their data as a valuable asset because it is used for making various strategic decisions. Any leakage of data can have disastrous effects on the company. This causes reluctance by many corporations in using the DAAS model since their data will then be stored at service provider's site and there will be a risk that service provider may view their data. The most intuitive way to mitigate this problem is to encrypt the data before sending it to service provider. But this raises the problem of "*Executing queries efficiently on the server*" since encrypted data hampers query processing.

Let us illustrate this with the help of an example: Consider the relation "part" as shown in Figure 1.1. Suppose the client wants to execute the query:

Select name

From part

Where price > 10000

In general the standard encryption schemes such as DES, AES etc. do not preserve order, i.e. if a < b then it is not mandatory that E(a) < E(b). If any of these encryption scheme is used to encrypt "part" while storing it on server, then we cannot execute this query on server.

pid	name	price	quantity
1	LED	30000	100
2	LCD	25000	150
3	TFT	10000	200

Figure 1.1: Relation "part"

One approach to solve this problem is to use a *Fully Homomorphic Encryption scheme* such as one proposed by C.Gentry in [7]. These schemes allow computation of any arbitrary function over encrypted data without the need to decrypt it. In this approach client encrypts his data using a Fully Homomorphic Encryption scheme before storing it at the server. When client wants to execute a query it is transformed into an equivalent query over the encrypted data. This allows server to process any valid SQL query on encrypted data and return the exact result set (still encrypted) to client. The client decrypts this encrypted result set to get the final answer. This approach offloads all the database processing work to the server. The problem with this approach is that operations on current Fully Homomorphic Encryption schemes are orders of magnitude (10^9 times) slower than the operations on plain text data. This makes them impractical for use in current systems.

Another approach to solve this problem is to use Service Provider only as *encrypted data storage*. In this approach client encrypts all the data before storing it at server. Whenever client has to execute a query, required part of database is transferred to the client machine, decrypted to recreate plain text database and then query is executed on this recreated plain text database using traditional query processing to get the result set. This approach has severe performance issues because for each query huge network traffic (to transfer required relations) is generated. This approach also negates many benefits of DAAS model since client now has to maintain his own database servers.

During our interaction with VP of a leading analytics company, which has been using DAAS for many years, a surprising thing that came to our knowledge is that they are using encrypted data storage approach for storing their sensitive data at server. Other non-sensitive data is stored at server in plain text. Hence it is clear that there is a real requirement in the industry to be able to execute queries in DAAS model efficiently at server while preserving security of data. This is the motivation for our project.

Many approaches have been proposed which lie in between these two extreme approaches e.g. bucketization based approach [8][12][9][11], CryptDB [15][6] etc. Bucketization based approach supports all standard SQL constructs but suffers from poor performance. It also requires the client to maintain a full-fledged database engine. On the other hand, CryptDB has good performance and does not require the client to maintain a database engine but it is able to support a very limited set of SQL constructs. These limitations prevent the use of these approaches in the industry.

In this work we propose a new approach, which we call ***PhantomDB***, to solve the data security problem in DAAS. PhantomDB maintains data security by encrypting data before storing it at the server. Each data item is encrypted under multiple encryption schemes. The encryption schemes chosen by PhantomDB have some special characteristics which allow server to perform certain operations on the cipher text itself. PhantomDB uses Order Preserving Encryption (supports range queries), Addition Homomorphic Encryption (supports aggregates), Multiplication Homomorphic Encryption and Search (supports word searches). Details of these encryption schemes are given in Section 4.1.

Each column of a tuple is encrypted separately. Also each data value is encrypted simultaneously under multiple encryption schemes. This leads to multiple encrypted columns for a single plain text column. The encryption schemes used to encrypt a column depend on its data type. A numeric value is encrypted under Order Preserving Encryption, Addition Homomorphic Encryption and Multiplication Homomorphic Encryption. A String value is encrypted under Order Preserving Encryption and Search Encryption. While storing a relation on the server, relation name and column names are randomized. This mapping from plain text relation and column names to randomized relation and column names is stored as metadata at the client. Details are given in Section 4.2. If we know the query workload in advance, then many optimizations can be done to the encrypted schema. These optimizations help us to increase the performance of PhantomDB. Details of these are given in Section 4.3.

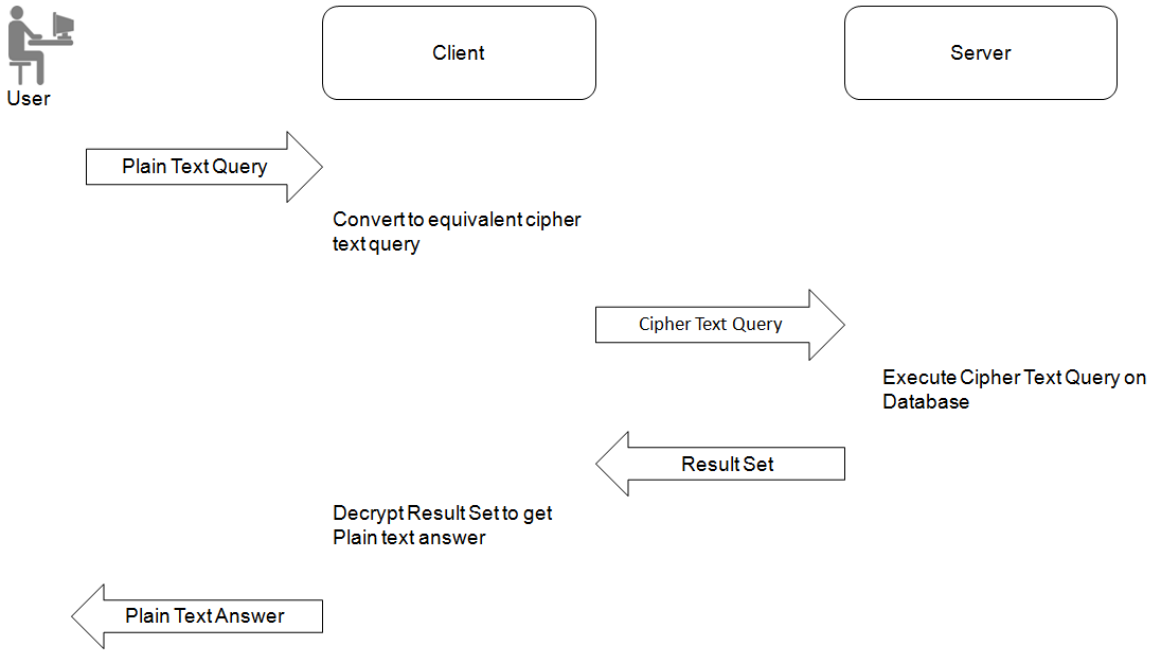

Figure 1.2: Query Execution Steps

Figure 1.2 shows the steps through which a query is executed in PhantomDB. User submits a query over plain text to the PhantomDB client. Client converts it into an equivalent query over the cipher text. This cipher text query is then given to server. Server executes this cipher text query on the data stored at database hosted by it. The result of this query execution (encrypted data) is returned to client, who then decrypts them to get final plain text answer. This plain text answer is then given to the user as output of his query.

Unfortunately not all plain text queries can be directly converted to an equivalent query over cipher text. To handle such queries we introduce the concept of "*Arithmetic Engine*" and "*Round Communication*". Arithmetic engine allows client to apply a set of functions on the values returned by server. These functions are the same that are supported by SQL in its select block. Round communication allows execution of a single query in multiple rounds with server. These concepts allow PhantomDB to support majority of SQL constructs. Details of these concepts are given in Section 5.

PhantomDB is able to support all the standard SQL constructs. It supports only

full word matches using similarity operators (e.g. *where name Like "% akshar %"*). It does not support other arbitrary pattern searches (e.g. *where name like "%ram%"*). PhantomDB does not require the client to maintain a database engine. The performance of PhantomDB depends to a large extent on the number of round communications needed to execute the query.

Storage models used by current database engines are not suitable for storing encrypted data generated by PhantomDB and lead to performance degradation. We propose a new *hybrid storage model* which takes some unique features of the encrypted data generated by PhantomDB into consideration while deciding the layout of relational data on disk. This helps in improving the performance of PhantomDB. Details of this new storage model are given in Section 6. Additionally, this storage model is suitable for compression of data which helps in reducing the disk footprint of PhantomDB. The compression algorithms tested are given in Section 6.4. Experimental results of these algorithms are given in Section 7.2.

Chapter 2

Problem Framework

2.1 Solution Objectives

The main objective of PhantomDB is:
To develop a framework for DAAS model in which query processing can be efficiently done at server while preserving security of data.
We want server to do majority of the query processing work. Client may help server in processing of the query, but he will not do any query processing (or predicate evaluation) on data. This prevents any requirement of having a database engine at client side and allows PhantomDB to retain all the benefits of DAAS model.

2.2 Assumptions

Various assumptions which we make in PhantomDB are as follows:

1. Server is not trusted. Hence client does not share its secret encryption and decryption keys with server.

2. Client is fully trusted and won't be compromised. If we remove this assumption then security can never be guaranteed since adversary can compromise client and see all the data (client has all the encryption and decryption keys).

3. The communication channel between client and server is secure. This can be ensured by using various techniques such as TLS (Transport Layer Security) and SSL (Secure Sockets Layer).

4. All the encryption schemes used by PhantomDB are unbreakable. If any of the encryption scheme used by PhantomDB is broken in future then we can replace it with another unbroken encryption scheme having similar properties.

5. We have large disk space at the server. This assumption is practical since disk is very cheap now-a-days.

6. Client will not do any operation which involves more than one tuple (e.g. sorting, grouping).

2.3 Adversary Model

Adversary Model used by PhantomDB is *"Honest But Curious"* model. Various assumptions of this model are:

- Server does not tamper with the database engine. Hence database engine works correctly as is expected of it. It does not give any false tuples or incomplete answers for any query made to it.

- Server has access to all the encrypted data which client stores in the database.

- Server knows which encryption schemes client has used to encrypt the data but does not know which encryption scheme has been used to encrypt any particular column (server may try to guess it by seeing the length of column values).

- Server does not have any domain knowledge about the data being stored by client.

- Server cannot launch any active attack on the encryption scheme. He can only use brute force attack (or any passive attack specific to an encryption scheme) to break the encryption scheme.

- The goal of adversary is to determine what plain text data client has stored in the database.

We choose this adversary model for PhantomDB because it is a very practical model in which a very limited trust is put on the server.

Chapter 3

Architecture of PhantomDB

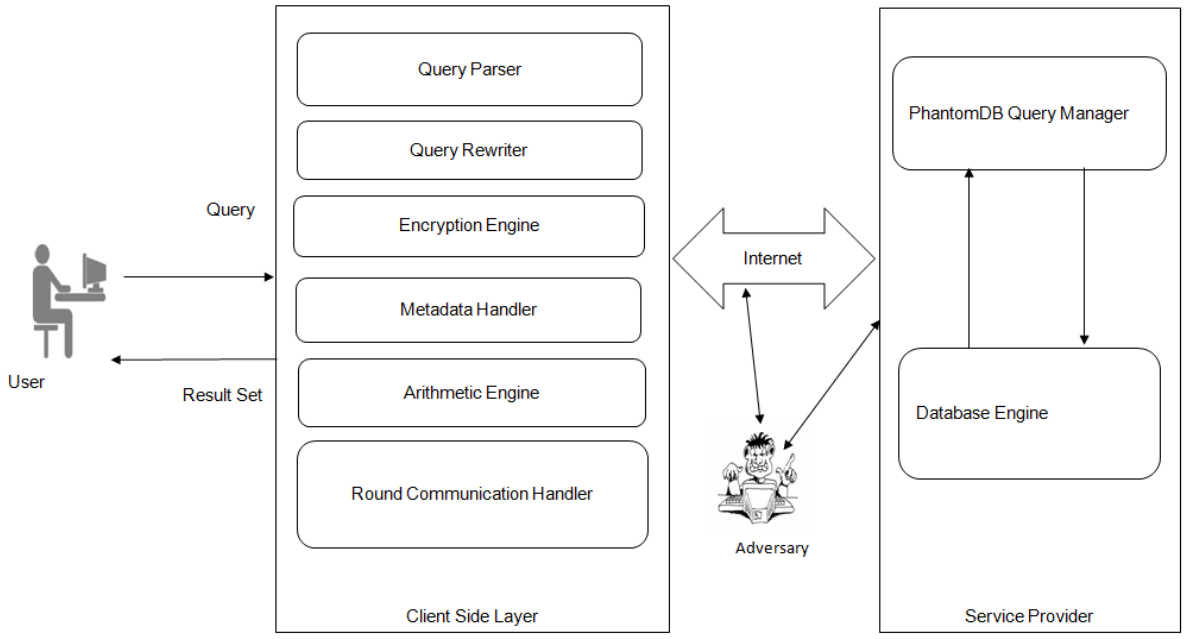

Figure 3.1: Architecture

The architecture of PhantomDB is shown in Figure 3.1. User issues plain text query. This plain text query is captured by the client side layer of PhantomDB. Query Parser is used to parse the query entered by user. This parser checks for syntactic and semantic validity of the query. If the query is valid then Query Rewriter re-writes it to an equivalent query over cipher text. Details of how this transformation is carried out is given in Section 5.1.

This transformed query is then transferred to PhantomDB Query Manager (PQM) layer at service provider. This layer executes the query submitted to it using Database Engine at service provider and Round Communication Handler (RCH) at client. We need PQM because current databases do not support round communication. PQM is stateful i.e. during communication with RCH it is blocked and cannot do further query processing till it gets the result back from RCH. RCH does not have to be stateful but it has to maintain all current valid round communication requests.

All the keys are stored in the encryption engine at client. It is responsible for encrypting and decrypting all data. This engine is not accessible to service provider. Metadata Handler is used to store and access all the metadata information needed at client. This metadata is used by query parser and query rewriter for their processing. Arithmetic Engine handles a set of functions over the cipher text values passed onto it. These functions are the same that are supported by SQL in its select block.

The adversary (can even be service provider) has access to all the messages that are being passed between client and server. He also has full access to all the internal modules of the service provider. He cannot modify PQM and Database Engine since our adversarial model (described in Section 2.3) does not allow it.

Chapter 4

Data Transformation

4.1 Encryption

PhantomDB uses encryption to preserve security of data stored at the server. Our choice of encryption scheme is based on the following observation:-

Even though there are large number of constructs in SQL only a small subset of these are used in most of the queries.

The most commonly used SQL constructs are equality checks, order comparisons, aggregation, multiplication and joins. There is no single practical encryption scheme which supports all of these constructs. However there exist practical encryption schemes which support a subset of these constructs. We use multiple such encryption schemes so as to cover all of these constructs.

PhantomDB uses attribute level encryption i.e. each column of a tuple is encrypted separately. We encrypt each column of the plain text tuple under multiple encryption schemes. This converts a single column of plain text relation into multiple columns of encrypted relation.

PhantomDB use one encryption key per encryption scheme for whole of the database. We chose this option, instead of other options such as one encryption key per column or one encryption key per relation, because it supports richest set of SQL among all the options.

Now we will discuss about various encryption schemes used by PhantomDB.

4.1.1 OPE

OPE stands for Order-Preserving Encryption Scheme. OPE is a deterministic encryption scheme whose encryption function preserves numerical ordering of the plain texts i.e.

$$if\ x < y\ then\ OPE_K(x) < OPE_K(y),\ for\ any\ key\ K.$$

OPE allows processing of equality and order predicates completely at the server on encrypted data itself. It also allows the server to process constructs like MIN, MAX, ORDER BY etc. A sample plain text query which can be processed using OPE is

Select name
From part
Where price > 10000

A single key (K_{OPE}) is used throughout the database for OPE. This allows server to join any two columns of the database. For example,

Select name, grade
From student,marks
Where student.sno = marks.sno

PhantomDB uses OPE scheme proposed in [3] which is the first provably secure order preserving encryption scheme.

4.1.2 ADD

ADD stands for Addition Homomorphic Encryption Scheme. ADD is an encryption scheme in which given only the public key and the encryptions of x and y we can compute the encryption of (x + y). We use Paillier Encryption scheme proposed in [14] for ADD. In this encryption scheme

$$E_K(x) * E_K(y) = E_K(x + y)\ ,\ for\ any\ key\ K$$

This encryption scheme allows server to compute aggregates over the cipher text. Queries containing aggregates are very frequent in decision support systems. For example,

Select sum(price)

From part

ADD allows PhantomDB to support these queries efficiently over encrypted relations. One thing to note here is that server will only know the encryption of aggregate and not the actual value of the aggregate.

ADD is a non-deterministic encryption scheme. This means that encryption of same value two times will give different cipher texts with a very high probability. It is achieved by using randomization during encryption of data. This randomization is needed because deterministic Addition Homomorphic Encryption Schemes are not secure. This non-deterministic nature of ADD prevents the value of aggregate computed by server to be used for comparisons. For example, the predicate in having clause of following query cannot be evaluated by server

Select name, sum(quantity)

From part

Group By name

Having sum(quantity) = 100

To actually calculate the aggregate over a column, SUM (SQL construct) is replaced by a UDF (User Defined Function) which we have written to perform Paillier multiplication on a column encrypted with ADD.

4.1.3 MULT

MULT stands for Multiplication Homomorphic Encryption Scheme. MULT is an encryption scheme in which given only the public key and the encryptions of x and y we can compute encryption of (x * y). We use RSA Encryption Scheme proposed in [16] for MULT. In this encryption scheme

$$E_K(x) * E_K(y) = E_K(x * y) \text{ , for any key } K$$

This encryption scheme allows server to compute multiplication over the cipher text. Queries containing multiplication are also very common in decision support systems. For Example,

*Select price * quantity*

From part

Using MULT allows PhantomDB to support these queries efficiently over the encrypted relations. MULT is a deterministic encryption scheme i.e.

$$\text{if } m1 = m2 \text{ then } E_K(m1) = E_K(m2) \text{ , for any key } K.$$

This property of MULT allows PhantomDB to support queries containing equality predicates on multiplication of columns at the server. For example, PhantomDB is able to evaluate the predicate of following query on server

Select name

From part

*Where price = 2 * (Select MIN (price) From part)*

(This query selects all those rows whose price is twice of the minimum price).

We have written a UDF (User Defined Function) to perform multiplication for values encrypted under MULT. It is needed because this homomorphism is under modulo n (where n is a public parameter of RSA).

4.1.4 SEARCH

This encryption scheme is specially designed for textual data. It supports searching on the encrypted text. We use a modification[15] of scheme proposed in [17]. This encryption scheme allows only full word searches on the encrypted data. For example, if "Ship to IISc" is encrypted using this encryption scheme then we can search for "Ship", "to" or "IISc" but we cannot search for "hip". A sample SQL query which can be executed using SEARCH:

Select name, price

From part

Where name Like"% LED %"

This query selects all those rows whose name contains the word LED. The spaces around LED signify that it is a word. A sample SQL query which cannot be executed using SEARCH:

Select name, price

From part

Where name Like"%LED%"

In this query there are no spaces around LED, which means that LED can be part of a word also (e.g misled).

Hence we can say that SEARCH has a limitation that it does not support arbitrary regular expressions.

4.2 Metadata and Data Transformation

This section describes how PhantomDB transforms a plain text relation into an encrypted relation. PhantomDB encrypts each column of a tuple separately. The types of encryptions applied to a column depend upon its data type.

- If the column is of numeric data type then it is encrypted under OPE, ADD and MULT.

- If the column is of String data type then it is encrypted under OPE and SEARCH.

While storing a relation on server the relation name and column names are randomized. The mapping from plain text relation and column names to encrypted relation and column name is stored as metadata at the client. This randomization prevents server from knowing which columns are related, i.e. which columns are different encryptions of same plain text column, in most cases. It also prevents server from knowing which

encryption schemes have been used to encrypt a particular column of the encrypted relation. The server can only guess the encryption scheme used by looking at the size of values stored in the column.

For example, the relation "item" shown in Figure 4.1 gets converted into relation "item_enc" shown in Figure 4.2. Note that we have used simple names as randomized names for the sake of illustration. In fact, the relation name and column name of the relation shown in Figure 4.2 can be any valid string.

name	price	type
A	100	1
B	250	2
C	50	1

Figure 4.1: Relation "item"

n_ope	n_search	price_ope	price_add	price_mult	type_ope	type_add	type_mult
O1	S1	O2	A1	M1	O3	A2	M2
O4	S2	O5	A3	M3	O6	A4	M4
O7	S3	O8	A5	M5	O3	A6	M2

Figure 4.2: Relation "item_Enc"

4.3 Data Transformation Optimizations

If we know the query workload in advance then some optimizations can be done at data transformation time to the schema of encrypted database which helps in improving the performance of PhantomDB. Following are the optimizations which PhantomDB does if query workload is given to it:

4.3.1 Workload Balanced Transformation

We can remove those encrypted columns which are never used in query processing. Although we will need to maintain at least one encrypted column for each plain text column so that we don't lose any data. For example, if the "pid" column of relation "part" shown in Figure 1.1 is only used for equality and joins then we only need to encrypt it under OPE and not under ADD and MULT. This optimization will reduce disk space occupied by PhantomDB and also improve its query performance by reducing the encrypted relation size.

4.3.2 Precomputation

If some query uses a function of values from multiple columns, which cannot be computed from encryptions used by PhantomDB, then those values can be materialized explicitly during data loading into PhantomDB. For example, if some query on relation "*part*" shown in Figure 1.1 wants to order its answer on the value of "*price * quantity*" (OPE of it cannot be computed by server), then we can explicitly materialize this new column and store the values encrypted under OPE. This optimization will increase the disk space occupied by PhantomDB but will decrease query execution time for those queries which use these materialized columns.

This optimization assumes that we have a read only database. If we allow updates to the database then some update queries, such as

Update part

Set price = price + 10

can cause these pre-computed columns to become inconsistent. In that case we may need to re-calculate the value of this materialized column for whole relation.

Chapter 5

Query Processing

5.1 Query Rewriting

User submits a query over plain text to the client. This query is first checked for syntactic and semantic validity. If the query is valid, then it is transformed into an equivalent query over encrypted relations by the client. This transformed query is then given to server for execution. For example, suppose following plain text query was submitted by the user

Select name

From part

Where price > 10000

This query is transformed into the following equivalent cipher text query by the client

Select name_ope

From part_enc

Where price_ope > $OPE_{K_{OPE}}$*(10000)*

During this query rewriting many transformations need to be done. Most prominent among these are:

- All the constants in the query are replaced by their appropriate encryptions. In the above example query, *10000* is transformed into $OPE_{K_{OPE}}$*(10000)*.

- Relation names and column names are replaced by the randomized names assigned to them. In the above example query, *name*, *part* and *price* are transformed into *name_ope*, *part_enc* and *price_ope* respectively.

  This step uses the metadata stored at client side by PhantomDB. This replacement takes into account the operations being performed on the column and uses corresponding encrypted column which supports this operation. For example, if the column is to be used for aggregation then we use the column encrypted under ADD.

- SQL construct SUM and multiplication of columns is replaced by our UDF (User Defined Function) calls. For example, if the plain text query submitted by user is

  Select sum(price)

  From part

  Then it is converted into:

  Select PhantomDB_SUM(price_ADD)

  From part_enc

Similarly, many other transformations are carried out to transform a query over plain text to an equivalent query over the cipher text.

This transformed query is then transferred to the server. The server executes this transformed query as it would execute any normal query. The result set of this query execution is returned to client who then decrypts it to get the final answer. This final answer is returned to the user as answer to his query over plain text.

5.2 Arithmetic Engine

Unfortunately not all the plain text queries can be transformed into an equivalent query over the cipher text. For example, suppose the plain text query submitted by user is:

Select avg(price)

From part

We want to calculate the average (= Sum / Count) value of price in this query. Server can compute the sum and count individually but he cannot compute average since sum is encrypted under ADD while count is a number. There is no available function which takes as input a number encrypted under ADD (i.e. $\text{ADD}_{K_{ADD}}(\text{m1})$) and a plain number (m2) and returns m1/m2, without decrypting the number encrypted under ADD.

To execute such queries in which the server can compute parts of the final answer but not their function we introduce the concept of Arithmetic Engine (AE) at the client side layer of PhantomDB. Arithmetic Engine allows client to compute a set of function on the values returned by the server. These functions are the same that are supported by SQL in its Select block. The main features of Arithmetic Engine are:

- It is a client side resident engine.
- It has access to the encryption engine present at client side, which allows it to decrypt any encrypted value.
- It takes as input a list of values (plain and/or encrypted) along with a function F. The function F can be one of the following:
  1. The four arithmetic functions +, -, / and *.
  2. The modulus operator %.
  3. Extract (datetime-field From datetime-value)
     This function isolates a single field of a date time value. Depending on the field given as input this function returns date, month, year etc. from the datatime-value passed to it.
  4. Position (string-value In string-value)
     This function searches for the presence of a particular string in another string. First argument is the string being searched and second argument is the string in which search has to be done. This function does not support regular expression matching.

5. Char-length (character-string)

   This function returns the length in characters of a character string passed to it.

6. Abs (numeric-value)

   This function returns the absolute value of the number passed to it.

7. Substring (string-value From start [For length])

   This function is used to extract a substring from the source string. The first argument is the source string. The second argument is the starting index of the substring. The third argument is the length of the substring. This third argument is optional. If this third argument is not provided then the substring starting from start index to the end of source string is returned.

8. Upper (string-value)

   This function converts the string passed to it into all upper case characters.

9. Lower (string-value)

   This function converts the string passed to it into all lower case characters.

10. Trim (string-value)

    This function strips all the leading and trailing blanks from the string passed to it.

Any other kind of operators such as relational operators (>, <, == etc.) or logical operators (and, or etc.) are not permitted.

First of all AE decrypts all the encrypted values. Then it applies the function F on the plain text values. The output of the function is returned as the output of the Arithmetic Engine.

In the above example the transformed query sent to server will be:

Select sum(price_add), count(price_add)

From part_enc

Server computes both these values and returns them to the client. Client sends these two values along with a proper function F to the Arithmetic Engine. Arithmetic Engine decrypts the sum to get sum_plain. Then it computes sum_plain / count (described in F) to get the average value. This average value is then returned as the output of the Arithmetic Engine. The client then returns this value to the user as answer to his query.

Arithmetic Engine allows PhantomDB to support any of the above mentioned functions on values in the outermost select block of the query. Note that server does not have access to the Arithmetic Engine.

5.3 Round Communication

Even Arithmetic Engine is not enough to support all kinds of plain text queries. For example, suppose the plain text query submitted by user is:

Select i1.name

From item i1

Where i1.price > (Select Avg(i2.price) From item i2 Where i2.type = i1.type)

In this query the average value is needed for predicate evaluation and not for output to user. Hence Arithmetic Engine cannot help in query execution.

To execute such queries in which the server cannot compute the values needed for predicate evaluation, we introduce the concept of Round Communication (RC). Round Communication allows client to supply any intermediate values which server needs for predicate evaluation but cannot be computed over encrypted values. The server will transfer appropriate values to client, who will then compute the function of these values needed by server for predicate evaluation and return it to server. The server will then continue query processing using the value returned by client. To tell server that it needs to do Round Communication in order to get a value, we introduce a new keyword (RC<id>) in transformed query. RC tells the PhantomDB Query Manager (PQM) at server that the value needed for this predicate evaluation needs a Round Communication. In the above example, the transformed query sent to server will be:

Select i1.name_ope

From item_enc i1

Where i1.price_ope > (RC1 (Select sum(i2.price_add), count(i2.price_add)

From item_enc i2 Where i2.type_ope = i1.type_ope))

For each row of relation i1, server executes the query associated with RC1. The values (sum, count) are then transferred to client. Client computes the average value by decrypting sum and dividing it by count. This average value is then encrypted under OPE and returned to server. The server then uses this value to evaluate the predicate.

A single query may need many Round Communications depending upon the type of predicates. In such case an identifier, <id>, is used to differentiate between the various Round Communication calls. For example, consider the following plain text query submitted by user:

Select i1.name

From item i1

Where i1.price > (Select Avg(i2.price) From item i2 Where i2.type = i1.type)

And i1.quantity < (Select Avg(i3.quantity) From item i3 Where i3.type = i1.type)

It will be transformed into the following query:

Select i1.name_ope

From item_enc i1

Where i1.price_ope > (RC1 (Select sum(i2.price_add), count(i2.price_add)

From item_enc i2 Where i2.type_ope = i1.type_ope))

And i1.quantity_ope < (RC2 (Select sum(i3.quantity_add), count(i3.quantity_add)

From item_enc i3 Where i3.type_ope = i1.type_ope))

Note that we have two predicates which need round communication for their execution. In order for client to distinguish between them they are given a unique identifier.

One important thing to note here is that client will not do any predicate evaluation during Round Communication. Predicate evaluation requires a more powerful engine than the one required by Arithmetic Engine and Round Communication. In fact we will need a full database engine at the client side to support predicate evaluation. We don't want it because a full database engine at the client side will negate most benefits of DAAS.

Another important thing to note here is that Round Communication makes the server stateful. While server is waiting for client to respond to his call, he cannot do further query processing. Removing this statefulness is a direction for future work.

The concepts of Arithmetic Engine and Round Communication can be used together in a single query also. For example, consider the following plain text query

Select avg(quantity)
From part
Where price > (Select avg(price) From part)

It is transformed into following query over the cipher text:

Select sum(quantity_add), count(quantity_add)
From part_enc
Where price_ope > (RC1 (Select sum(price_add), count(price_add) From part_enc))

In this transformed query sum(price_add) and count(price_add) are used for Round Communication whereas sum(quantity_add) and count(quantity_add) are used as input for Arithmetic Engine to calculate the average.

Using the concept of Arithmetic Engine and Round Communication PhantomDB is able to support majority of SQL constructs. It is evident from the fact that it supports *16 out of 22* TPCH benchmark queries. PhantomDB is not able to support other 6

queries because they have regular expression matching (not supported by SEARCH) as predicates.

5.4 Query Rewriting to Reduce Round Communication

It is possible to reduce (or even eliminate) the need for Round Communication in some queries by intelligent transformation of plain text query to cipher text query. For example, consider the following plain text query

Select name

From part

Where price + price = quantity

Following naïve transformation of this query results in a cipher text query which needs Round Communication

Select name_ope

From part_enc

Where (RC1 SUM(price_add, price_add)) = quantity_ope

In each Round Communication client will decrypt the value of "price_add + price_add" (encrypted under ADD), then encrypt it under OPE and return it to the server. This is needed beacuse ADD is not a deterministic encryption scheme and hence it cannot be used for processing equality predicate. However if we had transformed the plain text query into the following semantically equivalent cipher text query:

Select name_ope

From part_enc

*Where (MULT(2) * price_mult) = quantity_mult*

then the transformed query can be executed fully on server without the need for any Round Communication. Formalizing the techniques for such query re-writing, which minimize the need for Round Communication, is a direction for future work.
The total number of Round Communication calls needed to execute a query also depend on the plan selected by server to execute the transformed query.

5.5 Classification of various query components

Any query consists of following blocks:

- Select
- From
- Where
- Group By
- Having
- Order By

Depending on the type of expressions contained in them each of these block makes different number of Arithmetic Engine (AE) and Round Communication(RC) calls. In this section we will describe different types of these blocks depending on the number of Arithmetic Engine and Round Communication calls made by them.
We will consider the following relations:

- Relation A with attributes a1, a2 and a3.
- Relation B with attributes b1, b2 and b3.

5.5.1 Select Block

The Select block (without aggregate functions) makes only Arithmetic Engine calls. It can be classified into following types depending on the number of Arithmetic Engine calls made:

1. No Arithmetic Engine call required

   For these types of queries we don't need to call the Arithmetic Engine. For example, consider the following plain text query:

   Select a1
   From A

   This query can be transformed to the following equivalent query over the cipher text:

   Select a1_ope
   From A_enc

   The values returned from the server are then decrypted and returned to the user.

2. 1 Arithmetic Engine call per query

   For these types of queries we need to call the Arithmetic Engine once per query. For example, consider the following plain text query:

   Select avg(a1)
   From A

   This query is transformed to the following query over the cipher text:

   Select sum(a1_add) , count(a1_add)
   From A_enc

   The values returned by the server are decrypted and passed onto the arithmetic engine for calculation of average value.

3. 1 Arithmetic Engine call per output row

   For these types of queries we need to call the Arithmetic Engine once per output row returned by the server. For example, consider the following plain text query:

   *Select a1 * (1 - a2)*
   From A

   This query is transformed to the following query over the cipher text:

   Select a1_ope, a2_ope
   From A_enc

   For each row returned by the server, the values are decrypted and then passed onto the arithmetic engine for calculation. The values returned by arithmetic engine are then given to the user.

The classification of select block with aggregates is given in Section 5.5.3

5.5.2 Where Block

The objects in the where block have the following form:

LHS operator RHS

LHS corresponds to the data coming from the current relation while the RHS corresponds to the data coming from other relations.

1. RHS of the condition

   This corresponds to the data coming from other relations for predicate evaluation. It can be classified into following types depending on the number of Round Communication calls needed:

   (a) No Round Communication needed

   For these types of objects we only need to do proper query transformation and then they can be evaluated at the server. For example, consider the following plain text query:

Select a1

From A

Where a2 > 50

This query can be transformed to the following equivalent query over the cipher text:

Select a1_ope

From A_enc

Where a2_ope > OPE(50)

(b) One Round Communication per query

For these types of objects one Round Communication per query is needed for predicate evaluation. For example, consider the following plain text query:

Select a1

From A

Where a2 > (Select avg(b2) from B)

This query is transformed to the following query over the cipher text:

Select a1_ope

From A_enc

Where a2_ope > (RC1 Select sum(b2_add), count(b2_add) from B_enc)

(c) One Round Communication per row

For these types of objects one Round Communication per row is needed for predicate evaluation. For example, consider the following plain text query:

Select a1

From A

Where a2 > (Select avg(b2) From B Where b3 = a3)

This query is transformed to the following query over the cipher text:

Select a1_ope

From A_enc

Where a2_ope > (RC1 Select sum(b2_add), count(b2_add) From B_enc

Where b3_ope = a3_ope)

2. LHS of the condition

   This corresponds to the data coming from the current relation for predicate evaluation. It can be classified into the following types depending on the number of Round Communication calls needed:

   (a) No Round Communication needed

   For these types of objects we only need to do proper query transformation and then they can be evaluated at the server. For example, consider the following plain text query:

   Select a1

   From A

   Where a2 > 10

   This query can be transformed to the following equivalent query over the cipher text:

   Select a1_ope

   From A_enc

   Where a2_ope > OPE(10)

   (b) Round Communication needed

   For these types of objects one Round Communication per row is needed for predicate evaluation. For example, consider the following plain text query:

   Select a1

   From A

   *Where a2 * (1 - a3) > 1000*

   This query is transformed to the following query over the cipher text:

Select a1_ope

From A_enc

Where (RC1 a2_ope, a3_ope) > OPE(1000)

5.5.3 Group By Block

1. Group By attribute

   These attributes determine the group to which a row belongs to. These attributes can be divided into the following type:

   (a) No Round Communication needed

   The group to which these attribute belong to can be determined by server without the need for any Round Communication. For example, consider the following plain text query:

   Select count(a1)

   From A

   Group By a1

   This query can be transformed to the following equivalent query over the cipher text:

   Select count(a1_ope)

   From A_enc

   Group By a1_ope

   (b) Round Communication needed

   The group to which these attribute belong to cannot be determined by server. It needs Round Communication to determine the corresponding group. For example, consider the following plain text query:

   Select count(a1)

   From A

   *Group By a1 * (1 - a2)*

This query is transformed to the following query over the cipher text:

Select count(a1_ope)
From A_enc
Group By (RC1 a1_ope, a2_ope)

2. Select with Group By

   The Select block of a query with Group By usually has aggregate functions. There are certain limitations on these objects. It is because of this reason that they are being classified here instead of Section 5.5.1. These objects can be classified into the following type depending upon the number of Round Communications needed:

   (a) No Round Communication needed

   For these type of objects we only need to do proper query transformation and then they can be evaluated at the server. For example, consider the following plain text query:

   Select sum(a1)
   From A
   Group By a2

   This query can be transformed to the following equivalent query over the cipher text:

   Select sum(a1_add)
   From A_enc
   Group By a2_ope

   (b) Round Communication needed

   For these type of objects Round Communication is needed to calculate the value to be returned by server. For example, consider the following plain text query:

   *Select sum (a1 * (1 - a2))*

From A

Group By a3

This query is transformed to the following query over the cipher text:

Select sum (RC1 a1_ope, a2_ope)

From A_enc

Group By a3_ope

5.5.4 Having Block

The objects in the Having block have the following form:

LHS operator RHS

LHS corresponds to the data coming from the current relation while the RHS corresponds to the data coming from other relations.

1. RHS of the condition

   This corresponds to the data coming from other relations for predicate evaluation. It can be classified into following types depending on the number of Round Communication calls needed:

   (a) No Round Communication needed

   For these types of objects we only need to do proper query transformation and then they can be evaluated at the server. For example, consider the following plain text query:

   Select sum(a1)

   From A

   Group By a2

   Having count(a1) > 5

   This query can be transformed to the following equivalent query over the cipher text:

Select sum(a1_add)

From A_enc

Group By a2_ope

Having count(a1_ope) > 5

(b) One Round Communication per query

For these types of objects one Round Communication per query is needed for predicate evaluation. For example, consider the following plain text query:

Select sum(a1)

From A

Group By a2

Having count(a1) > (Select avg(b1) From B)

This query is transformed to the following query over the cipher text:

Select sum(a1_add)

From A_enc

Group By a2_ope

Having count(a1_ope) > (RC1 Select sum(b1_add), count(b1_add) From B_enc)

(c) One Round Communication per group

For these types of objects one Round Communication per group is needed for predicate evaluation. For example, consider the following plain text query:

Select sum(a1)

From A

Group By a2

Having count(a1) > (Select avg(b1) From B Where b2 = a2)

This query is transformed to the following query over the cipher text:

Select sum(a1_add)

From A_enc

Group By a2_ope

Having count(a1_ope) > (RC1 Select sum(b1_add), count(b1_add) From B_enc Where b2_ope = a2_ope)

2. LHS of the condition

   This corresponds to the data coming from the current relation for predicate evaluation. It can be classified into the following types depending on the number of Round Communication calls needed:

   (a) No Round Communication needed

   For these types of objects we only need to do proper query transformation and then they can be evaluated at the server. For example, consider the following plain text query:

   Select a1

   From A

   Group By a1

   Having count(a2) > 5

   This query can be transformed to the following equivalent query over the cipher text:

   Select a1_ope

   From A_enc

   Group By a1_ope

   Having count(a2_ope) > 5

   (b) Round Communication needed

   For these types of objects one Round Communication per row is needed for predicate evaluation. For example, consider the following plain text query:

   Select a3

   From A

Group By a3

*Having sum(a1 * (1 - a2)) > 1000*

This query is transformed to the following query over the cipher text:

Select a3_ope

From A_enc

Group By a3_ope

Having (RC2 sum(RC1 a1_ope, a2_ope)) > OPE(1000)

5.5.5 Order By Block

The attributes in this block are used for ordering of the output tuples. These can be classified into the following types depending on the number of Round Communication calls needed:

1. No Round Communication needed

   For these types of objects we only need to do proper query transformation and then they can be evaluated at the server. For example, consider the following plain text query:

   Select a1

   From A

   Order By a1

   This query can be transformed to the following equivalent query over the cipher text:

   Select a1_ope

   From A_enc

   Order By a1_ope

2. Round Communication needed

   For these types of objects one Round Communication per row is needed for ordering of rows. For example, consider the following plain text query:

Select a1

From A

*Order By a1 * (1 - a2)*

This query is transformed to the following query over the cipher text:

Select a1_ope

From A_enc

Order By (RC1 a1_ope, a2_ope)

Chapter 6

Storage Model

Storage model defines how relational data is actually stored on the hard disk. Storage models used by current database engines are not suitable for PhantomDB because of data explosion in PhantomDB.

6.1 Data Explosion in PhantomDB

Using multiple encryptions of a single data value in PhantomDB leads to the problem of Data Explosion. For achieving good security in OPE we require the cipher text to be at least 128 bits long. ADD and MULT require cipher text to be at least 2048 bits long. This means that a single 32 bit integer value gets converted into 4224 (128+2048+2048) bits of encrypted data.
For string data values the cipher text is almost of the same size as plain text. The main reason for this is that they don't use homomorphic encryptions.
This huge data explosion for numeric data leads to an increased disk space requirement for PhantomDB.

6.2 Current Storage Models

All the current database engines use one of the following storage model:

(a) **Horizontal Storage Model**

In this storage model values from all columns of a row are stored continuously on a disk block. Consecutive rows of a relation are stored one after another in disk blocks. It is the most widely used storage model. This storage model is not suitable for PhantomDB because increased table size (due to data explosion) means that queries which require sequential scan of a relation will be very slow as they have to read a very large amount of data.

(b) **Vertical Storage Model**

In this storage model all values in a single column of the relation are stored continuously on disk blocks. Values of a single column from consecutive rows are stored one after another in disk blocks. Different columns of a relation are stored separately. This model is not suitable for PhantomDB because query processing will lead to a large number of joins between various columns storing the row data.

6.3 Hybrid Storage Model

In order to get a good performance with PhantomDB we propose a new Hybrid Storage Model which is more suitable for storing encrypted data generated by PhantomDB. This storage model takes into consideration the unique features of encrypted data generated by PhantomDB. In this storage model the encrypted data is stored as follows:

- All the columns corresponding to Order Preserving Encryptions and Search Encryptions are stored using a horizontal storage model. The idea behind this is that most of the predicates are based on these encryptions and by keeping them together on a disk block multiple predicates on a relation can be evaluated simultaneously. Also since the data explosion due to OPE and SEARCH is very less, compared to ADD and MULT, even a sequential scan of data stored in this model will not be orders of magnitude slow.

- All the columns corresponding to ADD and MULT encryptions are stored using

vertical storage model. The idea behind this is that these encryptions are mostly used for aggregates (which are a function of all the values stored in a particular column). Hence keeping each column separately allows efficient processing of these aggregates.

The hybrid storage model for the relation "item_enc" (Figure 4.2) is shown in Figure 6.1. Experimental results (Section 7.1) show that Hybrid Storage Model leads to better query performance of PhantomDB than existing storage models.

n_ope	n_search	price_ope	type_ope
O1	S1	O2	O3
O4	S2	O5	O6
O7	S3	O8	O9

price_add
A1
A3
A5

type_add
A2
A4
A6

price_mult
M1
M3
M5

type_mult
M2
M4
M6

Figure 6.1: Hybrid Storage model

6.4 Compression in Hybrid Storage Model

ADD and MULT are the main culprits for data explosion problem of PhantomDB. Keeping ADD and MULT in vertical storage gives us the opportunity to compress them. This helps in reducing the disk footprint of PhantomDB. We have done experimental

evaluation to determine the compression achieved for ADD and MULT with following compression algorithms:

1. **BurrowsWheeler Transform (BWT)** [13]

   It is also called block-sorting compression. The idea is to apply a reversible transformation to a block of text to form a new block that contains the same characters, but is easier to compress by simple compression algorithms. The transformation tends to group characters together so that the probability of finding a character close to another instance of the same character is increased substantially. Text of this kind can easily be compressed with fast locally-adaptive algorithms, such as move-to-front coding in combination with Huffman or arithmetic coding.

   This algorithm is used in BZip.

2. **LZ77 (Lempel-Ziv 1977)** [18]

   LZ77 achieves compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the input (uncompressed) data stream. A match is encoded by a pair of numbers called a length-distance pair. To spot matches, the encoder must keep track of some amount of the most recent data, such as the last 2 KB, 4 KB, or 32 KB. The structure in which this data is held is called a sliding window. The encoder needs to keep this data to look for matches, and the decoder needs to keep this data to interpret the matches the encoder refers to.

   This algorithm is used in GZip.

3. **LZMA (Lempel-Ziv Markov-chain)** [1]

   LZMA uses a dictionary compression algorithm (a variant of LZ77 with huge dictionary sizes and special support for repeatedly used match distances), whose output is then encoded with a range encoder (using a complex model to make a probability prediction of each bit). The dictionary compressor finds matches using sophisticated dictionary data structures, and produces a stream of literal symbols and phrase references, which is encoded one bit at a time by the range encoder.

This algorithm is used in 7-Zip.

4. **PPMD (Prediction by partial matching)** [5]
   Prediction by Partial Matching is a method to predict the next symbol depending on n previous symbols. If no prediction can be made based on all n context symbols a prediction is attempted with (n - 1) symbols. This process is repeated until a match is found or no more symbols remain in context. At that point a fixed prediction is made.

Experimental results for above mentioned compression algorithms applied on ADD and MULT (Section 7.2) show that compression is a very promising idea to reduce the disk footprint of PhantomDB. The experiments also show that the decompression time is very low for most of the algorithms. This is very important because data will be decrypted numerous times and we don't want the decompression time to become a bottleneck for query processing.

Chapter 7

Experimental Results

7.1 Query Processing

In this section we present experimental results for query processing using PhantomDB. The experimental setup consists of a machine with Intel Core2 Quad Core 3.0 GHz processor, 8 GB memory, 4*300 GB 15000 RPM SAS running Ubuntu Linux 12.04. Both the client and server processes were running on the same machine. The database used was "MySQL". Data generation tool provided by TPCH was used to generate the data for experiments. We flush Operating System buffer cache as well as database query and table cache before running each query. We use primary key definition specified by TPCH. For the encrypted database Order Preserving Encryptions of columns specified as primary key in TPCH were made the primary key. No secondary indexes were created. We replace the calls for operations on ADD and MULT by our custom UDF's (User Defined Functions).

PhantomDB does not support floating point arithmetic. In our experiments all the floating point values were rounded to the nearest integer.

It is important to note here that these numbers are conservative in the sense that many optimizations such as caching of encrypted and decrypted values at client etc. have not been implemented. These optimizations will increase the performance of PhantomDB.

Storage Overheads

Table 7.1 shows the size of database before and after it is transformed by PhantomDB. The database sizes shown here are without the use of compression on ADD and MULT. If we use all the encryptions specified by PhantomDB then the database size increases by 36.8 times. However if we use Workload Balanced Transformation (WBT), specified in Section 4.3, then the increase in database size is 6.4 times.

	Disk Space
Plain Text DB	754 MB
PhantomDB	27 GB
PhantomDB WBT	4.68 GB

Table 7.1: Database Size

Time Overheads

PhantomDB is able to execute *16 out of 22* TPCH benchmark queries. In this section we present the time taken for execution of TPCH Query 1 and Query 4. Query 4 can be executed fully on server without any need for Round Communication and Arithmetic Engine. Query 1 on the other hand requires both Round Communication as well as Arithmetic Engine.

We used the following approaches while evaluating PhantomDB:

1. **PhantomDB**

   In this approach we use PhantomDB along with the horizontal storage model.

2. **PhantomDB HSM**

   In this approach we use PhantomDB along with the hybrid storage model.

3. **PhantomDB WBT**

   In this approach we use PhantomDB along with the Workload Balanced Transformation (specified in Section 4.3). It uses hybrid storage model.

4. PhantomDB PC

In this approach we use PhantomDB along with the pre-computations (specified in Section 4.3). It uses hybrid storage model.

Table 7.2 shows the time taken to execute Query 4, at varying selectivity. PhantomDB is up to 6 times slower in executing the query as compared to plain text database.

	0%	50%	100%
Plain Text DB	2	11	15
PhantomDB HSM	6	68	88

Table 7.2: Time taken (in seconds) to execute Q4 at varying selectivity

	0%	25%	100%
Plain Text DB	7	10	20
PhantomDB	350	67934	270319
PhantomDB HSM	71	67657	270040
PhantomDB WBT	72	67655	270041
PhantomDB PC	61	257	1122

Table 7.3: Time taken (in seconds) to execute Q1 at varying selectivity

Table 7.3 shows the time taken to execute Query 1 by PhantomDB (using various approaches) and Plain Text database. Query 1 requires a table scan on lineitem table of TPCH. PhantomDB is 50 times slower than Plain Text database at 0% selectivity (no output) also because of the data explosion (encrypted lineitem table is 45 times larger than plain text lineitem table). Using hybrid storage model (PhantomDB HSM) brings down the performance degradation to 10 times. If we use schema corresponding to Workload Balanced Transformation (PhantomDB WBT) then we get performance comparable to hybrid storage model. This is because many of the ADD and MULT columns are not required for processing TPCH workload and are hence removed from the schema in WBT. The performance of PhantomDB (even when using hybrid storage

model or workload balanced transformation) degrades rapidly when the selectivity of Query 1 is increased. However, the degradation of performance is less severe if we had used pre-computations (PhantomDB PC).

	DB	RC	#RC
PhantomDB	353	67578	1501796
PhantomDB HSM	73	67584	1501796
PhantomDB WBT	76	67579	1501796
PhantomDB PC	257	0	0

Table 7.4: Breakup of time taken (in seconds) and number of Round Communication (RC) calls made for executing Q1 at 25% selectivity

	Encryption	Decryption
OPE (Number)	11	11
OPE (String)	190	190
ADD	35	70
MULT	2	70
SEARCH	0.2	–

Table 7.5: Time taken (in ms) for single encryption and decryption.

To understand why there is such a severe performance degradation in PhantomDB, PhantomDB HSM and PhantomDB WBT we look into time taken for Database operations and Round Communication (RC) when executing Query 1 at 25% selectivity (Table 7.4). From the table it is clear that most of the time is spent on round communication. Since our server is stateful (i.e. it has to wait for client to respond to Round Communication call) this slows down the whole query processing.

In order to understand why round communication is taking so much time we look at time taken to do single encryption and decryption (Table 7.5). For Query 1 it takes 0.05s to do all encryptions and decryptions per round communication. Multiplying it with the

number of Round Communication calls (1501796) we expect that client will spend about 75090s (= 1501796 * 0.05) in encryption and decryption alone.

Hence we can conclude that biggest bottleneck in query execution of PhantomDB is encryption engine. The performance of PhantomDB can be improved drastically by using specialized hardware for encryption-decryption, which will bring down the time taken for encryption and decryption.

Communication Overheads

PhantomDB also leads to more data being transferred between server and client. Continuing our analysis of Query 1 and Query 4 we found that:

- Query 1 leads to 49 times more data being transferred per row of the output. This is because it requires many aggregates as part of the output (each aggregate is encrypted under ADD which takes 2048 bits). However this is not a big problem because for aggregate queries the output cardinality is very less as compared to relation size. Hence the absolute amount of data transferred is not big in most practical cases.

- Query 4 leads to 1.5 times more data being transferred per row of the output.

- Query 1 leads to transfer of 4736 bytes of data per Round Communication. Query 4 has no such overheads.

From the above experimental results we can conclude that the performance of queries in PhantomDB depends upon the approach chosen. It also depends upon the number of round communications needed to execute the query. We also saw that using a hybrid storage model improves the performance of PhantomDB.

7.2 Compression

In this section we present experimental results for compression of ADD and MULT data. The uncompressed data was generated using a Java program and stored in a file. First we generated a plain text file containing numbers. These numbers were then encrypted to generate the uncompressed encrypted file. The uncompressed data (encrypted values) was stored as binary data in the files. This simulates how a database engine will actually store the data of a relation in vertical storage model. We then compress the file using all the compression algorithms enumerated in Section 6.3.

The machine used for experiment was SUN Ultra 24, Intel Core2 Quad Core 3.0 GHz processor, 8 GB memory, 4*300 GB 15000 RPM SAS running Ubuntu Linux 12.04 and Windows Vista Business. We use gzip in Linux to compress the file using LZ77. For other compression algorithms we use 7-zip [1], which was installed on Windows. The compression algorithms were compared on following metrics:

Compression Ratio

Compression Ratio (CR) is defined as

$$CR = \frac{compressed\ data\ size}{uncompressed\ data\ size}$$

Compression Algorithms with lower Compression Ratio are better than algorithms with higher Compression Ratio. We fixed the uncompressed (encrypted) file size at 4 GB for our experiments. We varied the number of distinct values that were used to generate this uncompressed file. Figures 7.1 and 7.2 show Compression Ratio that were achieved for files storing data encrypted under ADD and MULT, respectively. Figure 7.3 shows the Compression Ratio that were achieved for the plain text file. Various inferences which we can draw from these figures are:

- Compression Ratio increases as we increase the number of distinct values used to generate the same amount of data. This means that columns having low domain

size are more suitable for compression.

- Compression Ratios for MULT are significantly better than Compression Ratios for ADD.
- LZMA performs better than other algorithms. For cases with few distinct values all the algorithms are comparable, but for cases with higher number of distinct values LZMA outperforms all the other algorithms.
- Compression Ratio for plain text are better than the Compression Ratio for data encrypted under ADD. However it is worse than the Compression Ratio for data encrypted under MULT.
- Compression Ratio for LZMA is better for data encrypted under ADD than for plain text data until 100000 distinct values are used.

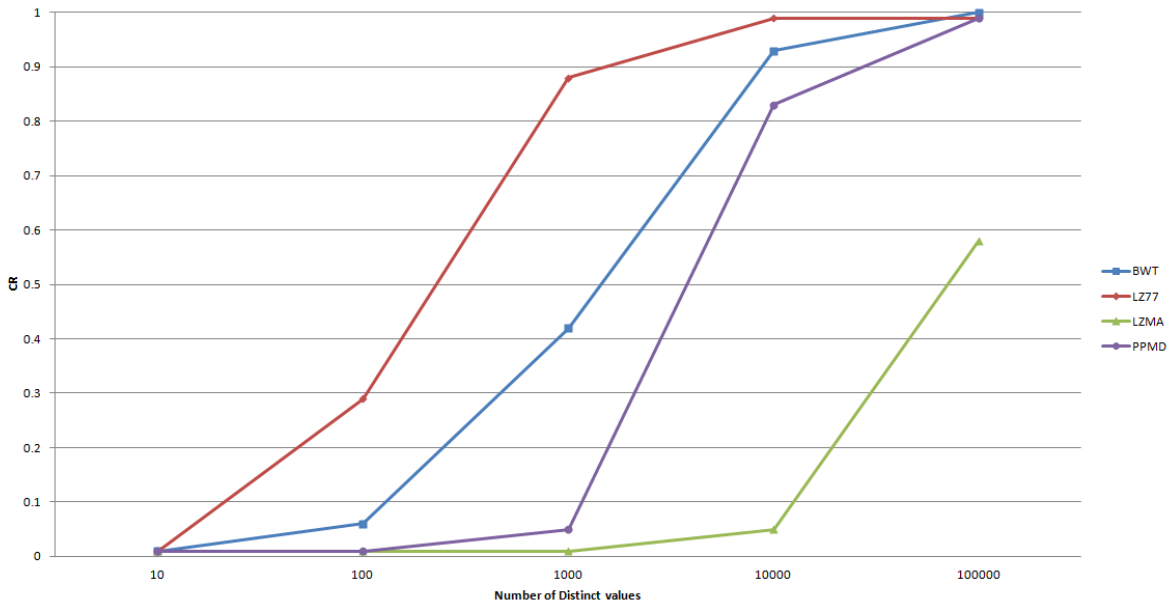

Figure 7.1: Compression Ratio for ADD

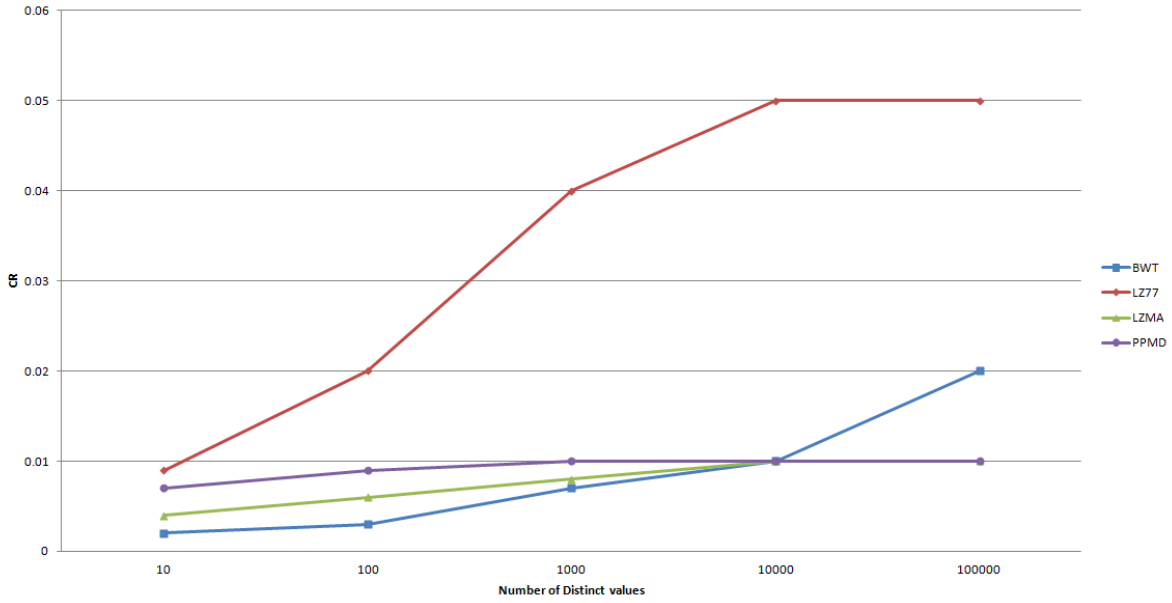

Figure 7.2: Compression Ratio for MULT

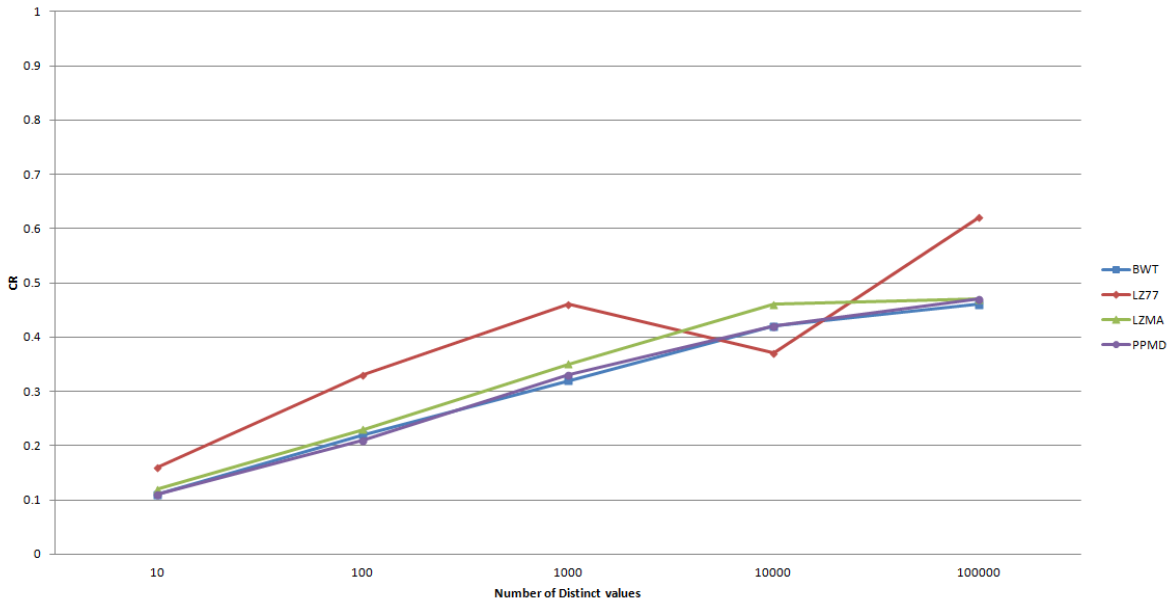

Figure 7.3: Compression Ratio for plain text data

Time

Figure 7.4 shows the amount of time it takes to compress a 4 GB file containing encrypted data using various compression algorithms, when the number of distinct values used to generate the file is varied. Figure 7.5 shows the amount of time it takes to decompress the corresponding compressed files. Various inferences which we can draw from these figures are:

- It takes more time to compress a file than to decompress the corresponding compressed file.
- Both compression time as well as decompression time increase as we increase the number of distinct values used to generate the uncompressed file. The rate of increase varies significantly across different algorithms.
- LZMA (which was best choice in terms of Compression Ratio) has significantly higher compression time compared to other algorithms for most cases. However decompression time for LZMA is amongst the lowest. This is very important because data will be compressed only once (during insertion) but it has to be decompressed numerous times. Hence LZMA is an acceptable choice in terms of compression and decompression times.

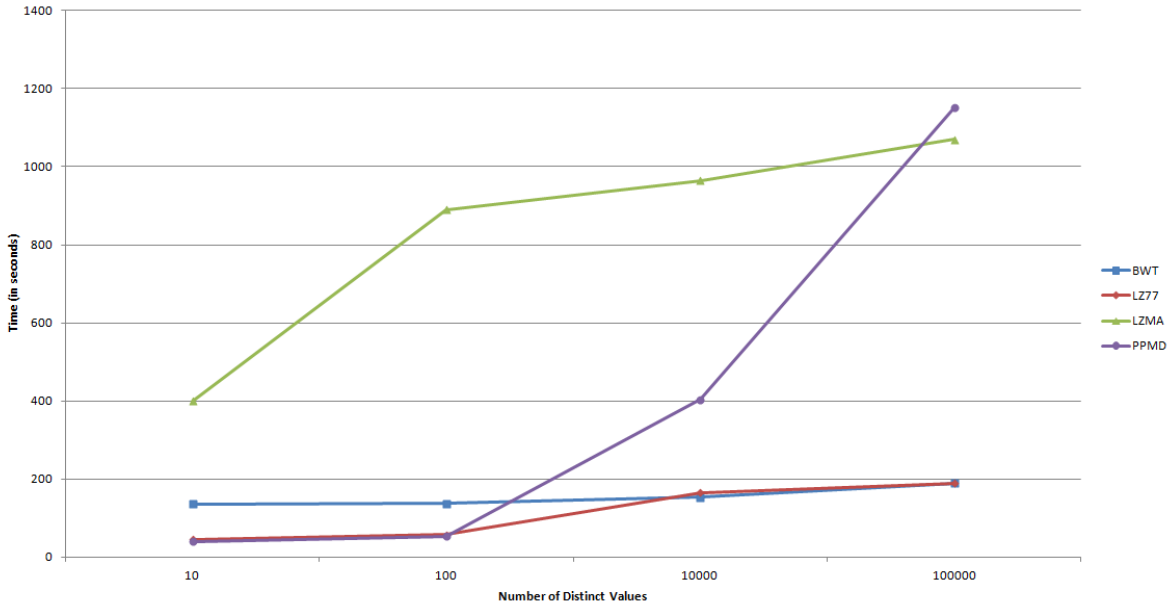

Figure 7.4: Compression Time

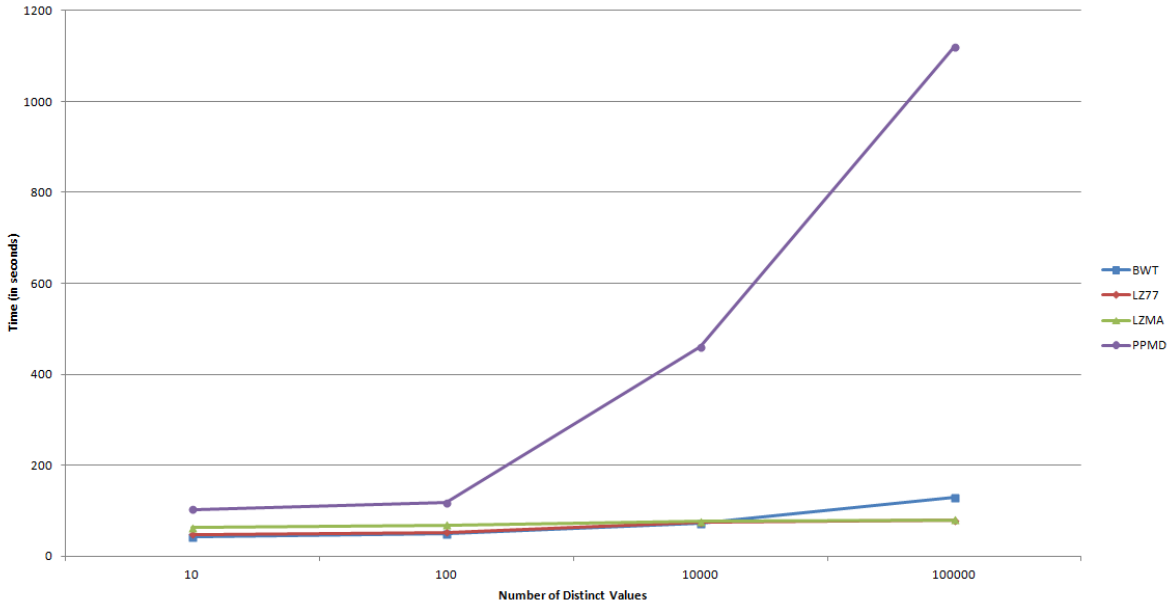

Figure 7.5: Decompression Time

Memory Usage

Figure 7.6 shows the amount of main memory used by various compression algorithms. From the figure it is clear that memory used by LZMA is significantly higher than other algorithms. However in terms of absolute numbers, LZMA uses about 200 MB of memory which is not very as large compared to total memory (in GB's) in current systems.

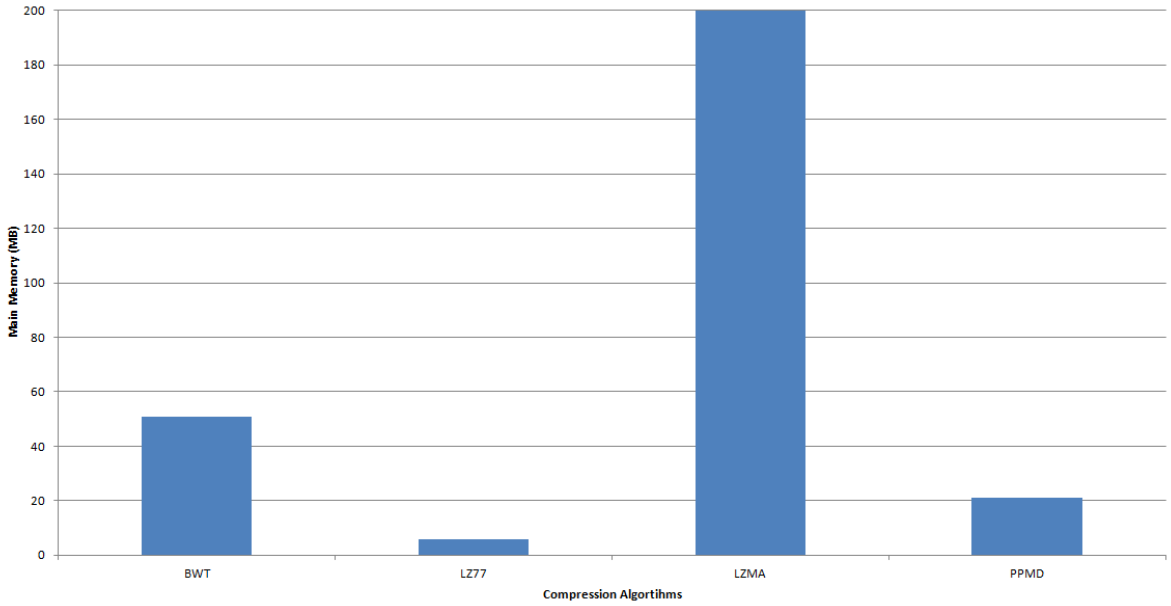

Figure 7.6: Main Memory Used

From all the above metrics we can conclude that:

- Compression helps in reducing the amount of disk space taken by ADD and MULT (main culprits for data explosion in PhantomDB). Hence we can reduce the disk footprint of PhantomDB significantly by using compression on ADD and MULT columns.
- LZMA is our preferred algorithm for compressing ADD and MULT data. This is because even though it has high compression time and high memory usage than

other algorithms, it gives us low compression ratio and its decompression time (critical for query processing) is also amongst the lowest.

Chapter 8

Related Work

Many approaches have been proposed to solve the problem of "*Executing SQL queries efficiently in DAAS model while preserving security*". These approaches can be categorized into software based or hardware based approaches. Sections 8.1 and 8.2 discuss about some software based approaches while Sections 8.3 and 8.4 discuss about some hardware based approaches.

8.1 CryptDB

This was proposed by Popa et al.[15] [6]. In it each attribute is encrypted separately. The encryption schemes used by CryptDB are:

(a) RND: which is a randomized encryption scheme and does not support any operation on cipher text.

(b) DET: which is a deterministic encryption scheme and supports equality predicates on cipher text.

(c) OPE: which is an order preserving encryption scheme similar to the one used by PhantomDB.

(d) JOIN: which is an encryption scheme that supports equi-joins on cipher texts.

(e) SEARCH: which is a word search encryption scheme similar to the one used by PhantomDB.

CryptDB uses the concept of "onions". An onion consists of multiple layers of encryption on a single data value. For example consider the "price" column of the relation "item" shown in Figure 4.1. One of the onion (equality onion) that will be created for it is $\text{RND}_{K1}(\text{DET}_{K2}(\text{JOIN}_{K3}(\text{m})))$ where m represents a value from price column. Multiple such onions are created for a single column. Issuing a query with equality predicate on price will result in client giving the key K1 to the server, who will use it to decrypt the outer layer of price column's equality onion and bring the security down to DET level. Notice that there is no way of increasing the security back. Hence security of CryptDB can be brought down to its lowest level by issuing queries which require least secure encryption schemes for processing.
This removal of onion layer also leads to performance jitters. Any query which needs to remove a layer of onion runs slower when it is executed the first time because of the time taken to remove the onion layer. Subsequent executions of the same query will be substantially faster because the layer has already been removed.

PhantomDB is a better approach than CryptDB because it can handle a super set of queries handled by CryptDB. This is evident from the fact that PhantomDB supports 16 (out of 22) TPCH queries while CryptDB does not support any.

8.2 Bucketization Approach

This approach which was proposed by Hacigumus et al. [8] splits the domain of possible column values into partitions. Each partition is given a unique id. These partition id are stored on the server in plain text. It uses row level encryption i.e. whole row is encrypted as a single data value.

Whenever a user submits a query the partition ids are used to do first level of filtering at the server. All the rows which pass this filter are transferred to client. The client decrypts these rows to create a plain text database and does final query processing on

this recreated database. The amount of rows transferred to client depends upon the coarseness of partitioning. In worst case whole relations have to be shipped to client. For example, "*Select AVG(Price) from part;*" requires bringing whole encrypted relation to client in bucketization approach but PhantomDB is able to process it very efficiently and requires transfer of only two values (Sum and Count).

The main drawback of this approach is that it requires a full-fledged database engine at the client side. This is because result set returned by the server contains false positives which have to be filtered out at the client side. This requirement negates most advantages of the DAAS model.

Subsequent papers in this approach have focused on efficient query splitting between the client and the server [9], efficient partitioning of the possible values to balance security and performance [12] [11].

8.3 Chip Secured Data Access

This approach proposed by Bouganim and Pucheral [4] requires the use of smart card, which is defined as a tamper proof hardware. A smart card is installed at the server site by client. This smart card contains the encryption engine as well as a database engine. All the data processing happens inside the smart card. The server is only used as the encrypted storage engine. The smart card has limited processing power and memory. Hence it becomes a bottleneck when this solution is scaled.

8.4 GhostDB

This approach proposed by Anciaux et al. [2] also requires the use of smart card. The data is divided into public and private parts. The public part is stored at the server in plain text while the private data is stored at server in encrypted form. All the processing on public data happens on the server while processing on private data happens on the smart card. This approach is also not feasible due to scalability issues of smart cards.

Chapter 9

Conclusions

This work presented PhantomDB which is a new framework for efficient query processing in DAAS model. PhantomDB is able to support 16 out of 22 TPCH benchmark queries. PhantomDB uses encryption to ensure security of data stored at the server. A data item is simultaneously encrypted under multiple, carefully chosen encryption schemes to produce multiple cipher texts. PhantomDB introduced the concept of Arithmetic Engine and Round Communication to support those queries which cannot be executed by server alone. PhantomDB also proposed a new hybrid storage model that takes unique features of encrypted data generated by PhantomDB into consideration while deciding how to store data on disk. This new hybrid storage model improves the performance of PhantomDB as compared to other current storage models. Another advantage of this hybrid storage model is that it allows compression to be used for reducing the disk footprint of PhantomDB.

Chapter 10

Future Work

Various directions for future work in PhantomDB are:

- Extending PhantomDB to handle analytic queries and data mining algorithms efficiently.
- In some DAAS models client has access to a secure co-processor at server. This co-processor is tamper proof and hence client trusts it fully. PhantomDB can be extended to such cases. The main challenge here is that co-processor has a very limited processing power and memory.
- One of the main features of PhantomDB is that a single data value is encrypted simultaneously under different encryption schemes. Formalizing the security achieved in this case is also an interesting area.

Bibliography

[1] `http://www.7-zip.org/`.

[2] N. Anciaux, M. Benzine, L. Bouganim, P. Pucheral, and D. Shasha. Ghostdb: querying visible and hidden data without leaks. In *SIGMOD Conference*. ACM, 2007.

[3] A. Boldyreva, N. Chenette, Y. Lee, and A. Oneill. Order-preserving symmetric encryption. In *Advances in Cryptology-EUROCRYPT 2009*. Springer, 2009.

[4] L. Bouganim and P. Pucheral. Chip-secured data access: Confidential data on untrusted servers. In *VLDB*, 2002.

[5] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. In *IEEE Transactions on Communications 32*. IEEE, 1984.

[6] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational cloud: a database service for the cloud. In *CIDR*, 2011.

[7] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

[8] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *SIGMOD Conference*. ACM, 2002.

[9] H. Hacigümüs, B. R. Iyer, and S. Mehrotra. Query optimization in encrypted database systems. In *DASFAA*. Springer, 2005.

[10] H. Hacigümüs, S. Mehrotra, and B. R. Iyer. Providing database as a service. In *ICDE*. IEEE Computer Society, 2002.

[11] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu. Secure multidimensional range queries over outsourced data. In *VLDB J.*, volume 21, 2012.

[12] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *VLDB*, 2004.

[13] D. J. W. Michael Burrows. A block sorting lossless data compression algorithm. In *Technical Report, Digital Equipment Corporation*. Digital Equipment Corporation, 1994.

[14] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*. Springer, 1999.

[15] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: processing queries on an encrypted database. In *Commun. ACM*, volume 55, 2012.

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

[17] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2000.

[18] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. In *IEEE Transactions on Information Theory*, volume 23, 1977.