Relational Database Encryption

Zheguang Zhao
Brown University, Rhode Island

Thesis Proposal
12th December, 2019
zheguang.zhao@gmail.com

Abstract

Given the increasing usage pattern of data outsourcing to the cloud and the omnipresent threats of data breaches, the problem of end-to-end relational database encryption has become a pressing one. Here, the data is encrypted by the client and uploaded to the untrusted server. The computation is carried out by the server directly over the encrypted data. The main challenge is to provide strong privacy, extensive functionality, high efficiency and legacy compliance, which have not been known to be simultaneously satisfiable in the literature.

This thesis presents a bottom-up, compositional approach to address the aforementioned requirements together as the problem of encrypted relational language. First an emulation technique is derived from the relational nature of the encrypted data structures under an algebraic view. Then a core set of encrypted relational operators are bootstrapped from the emulation of encrypted data structures. Language composition however presents several problems in efficiency and security. These are addressed by introducing dependence to the encrypted operators. Other relational operators are then simulated based on this core set of encrypted operators to constitute the encrypted relational language.

The ongoing work explores several ideas by integrating techniques from both the relational database and cryptography. A semi-encrypted multi-map is proposed to leverage the Third Normal Form with the same security but higher efficiency. To achieve the worst-case optimal joins over encrypted data, several variations are introduced to the encrypted data structures. Encrypted query optimization also deviates from the traditional context with additional emphasis on security. Continuing efforts are devoted to cost modeling of the encrypted relational language. Optimization rules are derived based the resulting cost analysis.

Finally, the various techniques proposed in this thesis are subdivided into several schemes for evaluation. All schemes are implemented in an open-source prototype system based on the algebraic core of Apache Spark SQL and thereby interfaced with any standard relational databases. The industry-standard TPC-H Benchmark is used to evaluate the efficiency and functionality.

Contents

1 Introduction

In the ubiquitous precense of cloud computing, more and more orgnizations choose to outsource their sensitive data to the cloud in order to levearge the the computational resource. But such usage has already raised concerns about the privacy of the data owners.

On the other hand, social networks, internet and mobile services collect sensitive users data for purposes such as advertisement targeting, sales and product support. But these sensitive user data are stored in the private cloud, and are also facing increasing danger of data breaches due to security loopholes.

The key problem is to design encrypted relational databases that can operate on end-to-end encrypted data. Here, the data is encrypted by the client and uploaded to the untrusted server. The computation is carried out by the server directly over the encrypted data. The main challenge is to provide strong privacy, extensive functionality, high efficiency and legacy compliance, which have not been known to be simultaneously satisfiable in the literature. Hacigümüş et al. [13] first studied this problem, but their approach is based on quantization and so leaks the frequency of data items in each range. Popa et al. [21] uses property-preserving encryption as part of the CryptDB prototype and so it also leaks nontrivial information about the data frequency and range. This leakage has been exploited in the leakage-abuse attacks in Naveed, Kamara, and Wright [19] and Cash et al. [5]. Fully homomophic encryption (FHE) or oblivious random-access machine (ORAM) provide highly secure building blocks but the resulting schemes are slow [9, 24]. The structured encryption (STE) provides reasonable security but is limited to either the key-value model or a reduced set of relational algebra, and is not known to be legacy compliant. Cash et al. [3] propose a encrypted multi-map for searchable symmetric encryption (SSE) with a focus in key-value data model. Its application to the relational database encryption is first studied in Kamara and Moataz [16]. Their approach inspires the following research questions that are examined together in this thesis

- Legacy complicance: How to design a relational database encryptionn scheme using purely the relational language and model
- Functionality: How to encrypt the relational language that preserves composition
- Security: How to improve the security for complex queries
- Efficiency: How to optimize encrypted queriestowards worst-case optimal time and storage

This thesis presents a bottom-up, compositional approach to address the aforementioned requirements together as the problem of encrypted relational language. First an emulation technique is derived from the relational nature of the encrypted data structures under an algebraic view. Then a core set of encrypted relational operators are bootstrapped from the emulation of encrypted data structures. Language composition however presents several problems in efficiency and security. These are addressed by introducing dependence to the encrypted operators. Other relational operators are then simulated based on this core set of encrypted operators to constitute the encrypted relational language.

The ongoing work explores several ideas by integrating techniques from both the relational database and cryptography. A semi-encrypted multi-map is proposed to leverage the third normal form (3NF) with the same security but higher efficiency. To achieve the worst-case optimal joins over encrypted data, several variations are introduced to the encrypted data structures.

Encrypted query optimization also deviates from the traditional context with additional emphasis on security. Continuing efforts are devoted to cost modeling of the encrypted relational language. Optimization rules are derived based the resulting cost analysis.

Finally, the various techniques proposed in this thesis are subdivided into several schemes for evaluation. All schemes are implemented in an open-source prototype system [26] based on the algebraic core of Apache Spark SQL [2] and thereby interfaced with any standard relational databases such as PostgreSQL [12]. The TPC-H Benchmark [8] is used to evaluate the efficiency and functionality.

2 Problem Setting

The user has a collection of relational data to outsource to the untrusted server that runs a standard relational databsae. The trusted client helps the user to encrypt the relational data and send them to the server. At query time, the user issues standard relational queries for example in the SQL language against the plaintext data model, the client translates such queries into equivalent queries but against the encrypted relational data model. The servrer takes the queries and efficiently run them and return the encrypted query results. The client decrypts the results and return them back to the user.

Within this setting, the focus is on how to design a relational database encryption scheme that involves mainly a trusted client protocol, and a simple de facto server/database protocol to realize both efficiency and privacy using the searchable symmetric encryption (SSE). The server is assumed to be semi-honest in the sense that it executes the queries and passively observes the execution. Occasionally the case of malicious server is also considered in this thesis.

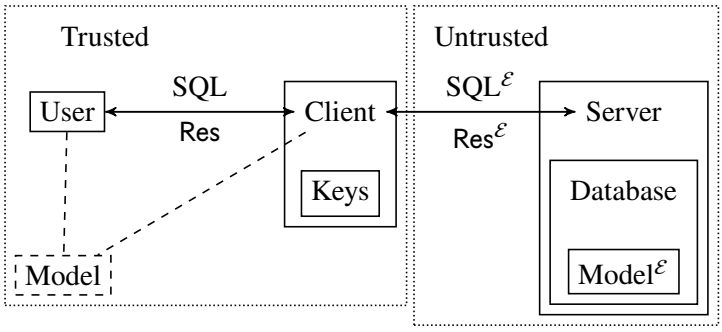

Figure 1: The problem setting of the encrypted relational database

Security. A SSE-based scheme admits leakage of information about plaintext relational data during both setup and query phases. It is then the job of the scheme designer to limit the leakage to a reasonable amount, where by reasonable it means it is sufficient to elude the known leakage-abuse attacks and the information it leaks is not significant for all its purposes. On the other hand, it is often necessary to provide a security argument based on reduction to certain hard problems to show that there is no more unknown leakage in the scheme than the ones claimed. This type of seucrity definition is presented here for a relational database encryption scheme $\mathbf{dex} = (\mathbf{Setup}, \mathbf{ProcessQuery})$ of probabilistic algorithms.

Definition 1 (Security of a relational database encryption scheme)**.** *A dynamic relational database encryption scheme* $\mathbf{dex} = (\mathbf{Setup}, \mathbf{ProcessQuery})$ *is* $\mathcal{L}$*-secure against adaptive attacks if for all efficient adversaries A there exists an efficient simulator S such that for the distributions of the same outcome of the games in Figure 2 are indistinguishable*

$$\left|\Pr\left(\mathsf{Real}^{A}_{\mathbf{dex}}(\lambda) = 1\right) - \Pr\left(\mathsf{Ideal}^{A}_{\mathbf{dex},\mathcal{L},S}(\lambda) = 1\right)\right| \leq \mathsf{negl}(\lambda) \tag{1}$$

$\mathbf{Real}_{\mathbf{dex}}(\lambda)$	$\mathbf{Ideal}_{\mathbf{dex},\mathcal{L},S}(\lambda)$
Init(RDB) 1. $(\mathsf{ERDB}, k) \sim \mathbf{Setup}(\mathsf{RDB}, \lambda)$ 2. **return** ERDB **Query**(Q_{RDB}) 1. $\left(\mathsf{Res}, Q_{\mathsf{ERDB}}\right) \sim \mathbf{ProessQuery}(k, Q_{\mathsf{RDB}})$ on ERDB 2. **return** Q_{ERDB} **Update**(Q_{RDB}) 1. $\left(\mathsf{Res}, Q_{\mathsf{ERDB}}\right) \sim \mathbf{ProcessQuery}(k, Q_{\mathsf{RDB}})$ on ERDB 2. **return** ERDB **Final**(b) 1. **output** b	**Init**(RDB) 1. $L \sim \mathcal{L}(\mathsf{RDB})$ 2. $\mathsf{ERDB} \sim S_{\mathbf{Init}}(L)$ 3. **return** ERDB **Query**(Q_{RDB}) 1. $L \sim \mathcal{L}(L, Q_{\mathsf{RDB}})$ 2. $\left(\mathsf{Res}, Q_{\mathsf{ERDB}}\right) \sim S_{\mathbf{Query}}(L)$ 3. **return** Q_{ERDB} **Update**(Q_{RDB}) 1. $L \sim \mathcal{L}(L, Q_{\mathsf{RDB}})$ 2. $\left(\mathsf{Res}, Q_{\mathsf{ERDB}}\right) \sim S_{\mathbf{Update}}(L)$ 3. **return** Q_{ERDB} **Final**(b) 1. **output** b

Figure 2: The interface of games in Definition 1. RDB denotes the relational database. ERDB denotes the encrypted counterpart. Q denotes the query. k is the secret key hold by the client, and λ the security parameter. $\mathcal{L}$ is the leakage function. Res stands for the query result. $\mathsf{negl}(\lambda)$ is a function that is asymtotically smaller than any inverse of polynomials in λ.

Functionality The relational language includes different dialects that are of different expressiveness. The langauge specifies various operators to compute on relational data. The goal of this thesis is to provide significant coverage to the relational language, and the same time without requiring the user to change their language usage.

Legacy Compliance Legacy compliance is an important property to the cryptographic scheme because it means the scheme can be implemented within or on top of an existing system without the need for customization. This property is no less crucial in the field of relational databases. Because the widely adopted standardization of the SQL language, much effort from both the industry and the research community has been devoted to improve the design from query optimization, parallelism to I/O and storage. A legacy compliant relational database scheme should be able to leverage these important results.

Efficiency Although the legacy compliance property opens up the opportunity to benefit from the advances in database research, it is not immediately clear whether or how such a scheme would be able to reap such benefits. For example, as will be shown in the thesis, the query optimziation has similarities but also differences compared to the plaintext databases. Another example is the worst-case optimal join algorithms that were only discovered within the last decade, but remain unclear how a relational database that is equipped with such algorithms would be encrypted while preserving the efficiency. This is also the focus of this thesis.

3 Preliminaries

3.1 Cryptographic Basics

Definition 2 (Pseudorandom functions)**.** *A set of functions* $\left\{\mathcal{F} : K \times \{0,1\}^n \to \{0,1\}^m \mid K \in \{0,1\}^\lambda\right\}$ *is pseudorandom if it satisifies the following*

- *Given* $x \in \{0,1\}^n$ *and* $k \in \{0,1\}^\lambda$*, there exists an efficient algorithm to compute* $\mathcal{F}(k,x)$
- *For any efficient algorithm* A*, the difference in the following two distributions is negligible in* λ

$$\left| \Pr_{k\sim\{0,1\}^\lambda}\left(A_{\mathcal{F}(k,\cdot)}(\lambda)=1\right) - \Pr_{f\sim U_f}\left(A_f(\lambda)=1\right)\right| \leq \mathsf{negl}(\lambda) \tag{2}$$

where U_f *denotes the set of all functions from* $\{0,1\}^n$ *to* $\{0,1\}^m$*.*

Definition 3 (CPA encryption)**.** *An encryption scheme* $\Sigma = (\mathsf{keygen}, \mathcal{E}, \mathcal{D})$ *has security against chosen-plaintext attacks if it satisifies the following*

- *Given* $m \in \Sigma.M$ *and* $k \sim \Sigma.\mathsf{keygen}(\lambda)$*, there exists an efficient algorihtm to compute* $\mathcal{E}(k,m)$
- *For any efficient algorithm* A*, any two messages* $m_L, m_R \in \Sigma.M$ *the difference in the following two distributions is negligible in* λ

$$\left| \Pr_{k\sim\mathsf{keygen}(\lambda)}\left(A_{\mathcal{E}}(m_L,\lambda)=1\right) - \Pr_{k\sim\mathsf{keygen}(\lambda)}\left(A_{\mathcal{E}}(m_R,\lambda)=1\right)\right| \leq \mathsf{negl}(\lambda) \tag{3}$$

3.2 Relational Database Basics

Relational Language. The relational language is a domain-specific language that expresses computation in the relational model The language specifies a spectrum of sub-langauges, each of which varies in capability, ranging from conjunctive queries, relational algebra that is quivalent to predicate logic, and datalog with recursion, and finally SQL which has recursion and window function to reach turing-completeness.

The Select-Project-Cross-product (SPC) algebra or equivalently the Select-Project-natural-Join-Rename (SPJR) algebra is at the core of the langauge taht expresses all conjunctive queries.

The relational alagebra extends the SPRJ algebra to express union, negation and aggregation. The algebra is closed under its operators over a set of relations.

The relational algebra cannot express transitive closure which is common in graphs. Datalog amends this aspect by adding fixed-point operator to give semnatics to recursive programs.

Finally, the Structured Query Language (SQL) specifies more operators including udpate, insertion and deleteion, and more importantly window function and recursive common table expression (details below) to bring the relational language to turing-completeness.

Relational Model. A relational model consists of a set of tables $T_1, T_2, \cdots, T_m$.

Table / Relation. A table or relation T has schema $\mathsf{Schema}(T) = (c_1, c_2, \cdots, c_j)$ and name $\mathsf{Name}T$, where c_j is an attribute or a column. When the context is clear, the symbol Schema and Name is dropped.

A table can be viewed as a matrix $T[i,j] = v_{i,j}$ for the cell at the i-th row and j-th column. The i-th row of the table can be denoted as $T[i,\cdot]$, and the j-th column $T[\cdot,j]$

A constant can also form a relation for example by the syntax $\rho_i\{1\}$, which describe a one-row, one column relationa with value 1 and column name i.

Query. A relational query connects the tables and attributes together using operators to perform comptuation on the data. Consider two main operations in relational algebra, the selection σ, join $\bowtie$ and projection π. One such query may read

$$\pi_{T_1.c_1,T_2.c_3}\sigma_{T_1.c_2=10}T_1 \bowtie_{T_1.c_1=T_2.c_1} T_2 \tag{4}$$

Which can also written in SQL

$$\begin{aligned} &\texttt{Select} \quad T_1.c_1, T_2.c_3 \\ &\texttt{From} \quad T_1, T_2 \\ &\texttt{From} \quad T_1.c_2 = 10 \quad \texttt{And} \quad T_1.c_1 = T_2.c_1 \end{aligned} \tag{5}$$

The result of a query is usually referred to as the query result or *result set*.

Query Tree A relational language can be viewed as a tree of operators. Such a tree also specifies precedence and associativity of the operators, thereby giving it an execution semantics. Two examples of such tree for the above example are in Figure 3

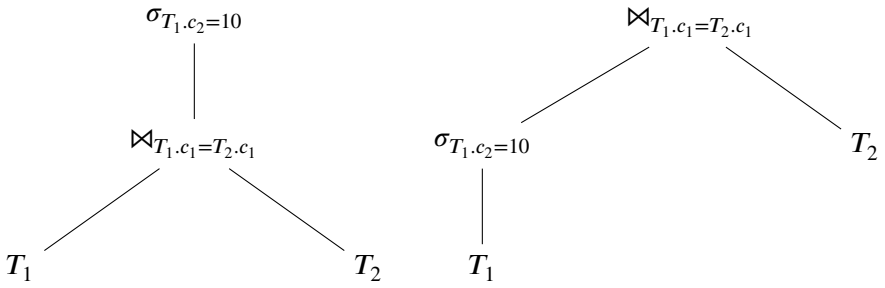

Figure 3: An exmaple of two equivalent query trees for the same query in (4).

Because a query tree gives a structure to a query, a tree traversal algorithm can be used to design a query rewrite program that transform a query tree to another query tree while preserving the semantics. This transformation forms the basis of query optimization, becasuse different query trees for a query may have different efficiency implication while all having the same semantics. This transformation also becomes the essential tool for designing query encryption in Section 5.

Because a query tree is equivalent to a query, in the rest of this thesis they are used interchangeably.

Formula. A boolean formula expresses propositional logic that is associated with an operator. The atom of the formula can refer to attributes or columns. So in this above example, the formula are $T_1.c_2 = 10$ and $T_1.c_1 = T_2.c_1$.

Predicate. An atom of the formula is also called a *predicate*. A predicate associated with a join is called the join predicate, and with a filter the filter predicate. Predicates can have correlation in their operands. Correlation between two predicates refers to the existence of attributes in these predicates that are related by the same table. In the above example, $T_1.c_2 = 10$ and $T_1.c_1 = T_2.c_1$ are correlated by the attributes c_1 and c_2 because both attributes are from the same table T_1.

Subquery. A subquery is a subexpression in a query. A subquery is typically an operand of an operator. The simplest subquery is just a table expression, such as the T_1 in the above example. However a subquery can be a complex query that has multiple operators, formula and tables.

View. A view is a virtual relation that is computed based on a query. A view can be thought of as a subquery or query plus a schema name.

Filter Operator. The filter operator or select operator in relational algebra takes a subquery either of the form of a table or a more complex query, and a condition of propositional boolean formula, and outputs rows in the subquery that satisify that the condition.

Join Operator. The join operator takes two subqueries and "multiply" the results by finding all possible matching pairs of tuples in the two subqueries that satisfy the formula.

There are several types of joins. The cross join or cross product $relcross$ computes cartesion product of two subqueries and does not take any formula. The inner join $\bowtie_{\mathsf{Formula}}$ computes based on the formula. The outer join in addition keeps unmatched rows from the specified left or right operands. The left semi join $T_1 \ltimes_{\mathsf{Formula}} T_2$ retains rows in the left operand that satisfies a formula that invovles T_2. The naturla join $\bowtie$ computes based on the implied formula that matches the same set of attributes in both left and right operands.

Other Operators. The projection operator for example $\pi_{c_1,c_2\to\mathsf{alias},\mathcal{F}k,c_3} T_1$ can retain columns from a subqery T_1, rename some of its attributes. and apply function to its columns.

The rename operator ρ only performs renaming of the subquery's attributes.

Recursive Common Table Expression. The relational algebra does not have means to express recursion. The need for recursion arises naturally in hierarchical data or graphs. Datalog and SQL extends beyond the relational algebra to include recursion, which is called the fixed-point operator in Datalog and the recursive common table expression in SQL [10]. The latter notion of recursion is used throughout this work. Here is an example

```
With Recursive View As
    BaseSubquery
  Union All
    RecursiveSubquery
```
(6)

The RCTE is a SQL expression, so it can compose with other SQL expressions to form a more complex query. In this case, an external subquery will reference the RCTE by name.

4 Emulation

The key technique used in this work is the so-called *emulation* of a cryptographic scheme for a domain specific language. The purpose of emulation is to address the requirements described in the problem setting, which can be phrased as follows

- Legacy compliance: can a cryptographic scheme be implemented on top of a domain specific language without any extension?
- Functionality: how much of a domian spec ific language can be implemented by using a cryptographic scheme as a building block?

These two questions are tightly connected and are treated separately in this section on Emulation and the next section on Encrypted SQL.

Cryptographic protocols are typically written in a turing-complete language or pseudocode and assumes various types of data structures. However many realistic applications are written using domain-specific languages that for one are not necessarily general-purposed, and secondly may not come with the prescribed data structures. For this reason many cryptographic protocols are not so-called legacy compliant, for they cannot be implemented on top of a given domain-specific language and its data model without modification and customization because there is a mismatch in the languages and data structures.

The mismatch between a cyrptographic scheme and a domian specific language also presents a challenge on the other direction, namely to implement an encrypted version of the domain specific language using the said cryptographic scheme. This is the aspect of functionality. It may not always be the case that the cryptographic protocol can be used as a building block to implement the entire encrypted domain specific language.

Consider using symmetric searchable encryption (SSE) to build an encrypted key-value model such as in the document retrieval application. One main building block in SSE is the encrypted multi-map which is typically described as a dictionary. A dictionary is an abstract data type that has efficient instantiation as a hash table. The key-value model is also a dictionary. So there is a perferct symmetry of the abstraction such that a SSE encrypted multi-map can be simplily stored as key-value data and the operations are also mirrored. In terms of functionality, such symmetry also exists because the main operations in both the encrypted multi-map and the key-value model is the retrieval of values associated with the keys. So it is not a far fetch to design a legacy-compliant, functionally-complete key-value model encryption scheme

However SSE has not been known to implement any relational database encryption scheme that is legacy compliant. The main barrier seems to be that the relational model and the relational query language at the logical level only present one type of data structure, namely the relations or tables. Its purpose is for modeling entities and relationships that commonly arise in real-world applications. So there appears to be a substantial difference between an encrypted multi-map and the relational model. On the other hand, the query langauge in relational model is also substantially more complex than the encrypted multi-map. It is perhaps unclear a priori how much coverage a SSE scheme can provide for a relational query language. Furthermore, the efficiency also may not be preserved during the emulation.

Finally, even if one could show that an SSE encrypted multi-map can be used to build a legacy-compliant, functionally-complete ecrypted relational database, the overall security of the scheme would require a careful analysis.

To address these issues, this thesis takes a bottom-up, compositional approach where the first step is to investigate if and how an encrypted data structure can be *emulated*, or implemented, purely using the relational language and model. The goal is however to defer the problem of how to encrypt entire

relational language, but rather just to establish how it is possible to descrbie an encrypted data structure with the relational language. There are good reasons to do so first, because relational language is compositional by nature, so the distant hope is that the problem of entire encrypted language would get much easier once the constituents are also relational.

To explain why emulation is possible between a cryptographic scheme and a domain-speicifc language, the nature of computation on the crypotgraphic scheme needs to be viewed under the new light of the said language. Given that the relational algebra is an algebraic language, an algebraic view needs to be established for the encrypted data structures under SSE. Then the parallel between the relational language and the encrypted data structures is drawn with the recursive common table expression also in the algebraic sense. Finally the connection of these two constructs is thereby established rather naturally. This result will lay down the foundation for consrtucting Encrypted SQL in the next section.

4.1 An Algebraic View of Multi-maps

A multi-map maps each label to a set of values. The operations on a multi-map include search, insertion and deletion. One can search for all values that are mapped to a label, add a new value for a label, or delete an existing value for a label.

Typically multi-maps are defined and viewed as an *abstract data type*. However the connection between multi-maps, encrypted multi-maps and emulated encrypted multi-maps may be better explained via an *algebraic* view of the mulit-maps. Algebraically, a multi-map and its operations can be viewed as an elementary map with recursion, or so-called *recursive map*. This view will be instrumental in the emulation of encrypted multi-maps.

Let MM be the multi-map of each label $l \in L$ and a set of values $\{v \in V\}$

$$\mathsf{MM} : L \to 2^V \tag{7}$$

where L is the domain of MM, Dom(MM), and 2^V denotes the powerset of V and is the codomain of MM. With slight abuse of notation, denote $\mathsf{MM}(l)$ as the image of l under MM.

A multi-map has the following operations

Definition 4 (The Multi-map)**.** *A multi-map* $\mathsf{MM} : L \to 2^V$ *has the following operations*

- $\mathsf{MM}(l)$ *returns all values associated with label* l
- $\mathsf{MM} + \{l \to v\}$ *returns a new* MM *with the label-value pair added*
- $\mathsf{MM} - \{l \to v\}$ *returns a new* MM *with the label-value pair removed*

Perhaps the first thing to notice is that a multi-map is not so much different from a *map*, or a function, except for the codomain is set of sets rather than a set of elements. So the multi-map can be defined as an element-to-set map

$$\mathsf{MM} : \left\{ l \to V_l \mid l \in \text{Dom}(\mathsf{MM}) \quad \text{and} \quad V_l \in \mathsf{MM}(l) \right\} \tag{8}$$

where $V_l = \left\{ v_1, \cdots, v_{\|V_l\|} \right\} = \mathsf{MM}(l)$ denotes the subset of V that maps from l.

The fact that the multi-map has a codomain of a powerset is quite an annoyance for relational model. The relational model advocates for atomic or elemenatray values in a relation. A map of elementary codomain is more amendable to the relational model. Can a multi-map be defined only on an elementary map?

So the first step towards this simplification is to decompose a multi-map into two structures, namely a multi-map and a map

$$\mathsf{MM} \stackrel{\text{def}}{=} \mathsf{MM}_1 \circ \mathsf{M}_2 \tag{9}$$

where the multi-map MM_1 is an element-to-set function

$$\mathsf{MM}_1 : \{l \to \{(l,i)\} \mid \forall l \in \mathrm{Dom}(\mathsf{MM}) \quad \text{and} \quad \forall i \in [\|\mathsf{MM}(l)\|]\} \tag{10}$$

and the map M_2 is a set-to-set function that internally uses a tuple-to-element map $\mathsf{M}_{\mathsf{elm}}$

$$\mathsf{M}_2 : \left\{\{(l,i)\} \to \left\{\mathsf{M}_{\mathsf{elm}}(l,i)\right\} \mid \forall (l,i) \in \mathsf{MM}_1(l)\right\} \tag{11}$$

where

$$\mathsf{M}_{\mathsf{elm}} : \left\{(l,i) \to v_i \mid \forall (l,i) \in \mathrm{Dom}(\mathsf{M}_2) \quad \text{and} \quad \forall v_i \in \mathsf{MM}(l)\right\} \tag{12}$$

It turns out one can further reduce the multi-map to just the elementary map with a recursion

Definition 5 (The Multi-map as a Recursive Map)**.** *A multi-map* MM *can be represented as an elementary map* $\mathsf{M}_{\mathsf{elm}}$ *with recursion functions* rec,rec^+ *and* rec^-

$$\mathsf{MM} \stackrel{\text{def}}{=} \left(\mathsf{M}_{\mathsf{elm}}, \mathsf{rec}, \mathsf{rec}^+, \mathsf{rec}^-\right) \tag{13}$$

where the operations are defined as

- $\mathsf{MM}(l) = \mathsf{rec}(l,1)$
- $\mathsf{MM} + \{l \to v\} = \mathsf{M}_{\mathsf{elm}} \cup \mathsf{rec}^+(l,v,1)$
- $\mathsf{MM} - \{l \to v\} = \mathsf{M}_{\mathsf{elm}} \setminus \mathsf{rec}^-(l,v,1)$

and the map is defined as

$$\mathsf{M}_{\mathsf{elm}} : \left\{(l,i) \to v_i \mid \forall (l,i) \in \mathrm{Dom}(\mathsf{M}_2) \quad \textit{and} \quad \forall v_i \in \mathsf{MM}(l)\right\} \tag{14}$$

and the recursion for search is

$$\mathsf{rec}(l,i) = \begin{cases} \left\{\mathsf{M}_{\mathsf{elm}}(l,i)\right\} \cup \mathsf{rec}(l,i+1) & \textit{if} \quad \exists (l,i) \in \mathrm{Dom}(\mathsf{M}_{\mathsf{elm}}) \\ \emptyset & \textit{else} \end{cases} \tag{15}$$

$$\mathsf{MM} + \{l \to v\} \stackrel{\text{def}}{=} \mathsf{M}_{\mathsf{elm}} \cup \mathsf{rec}^+(l,v,1) \tag{16}$$

and the recursion for insertion is

$$\mathsf{rec}^+(l,v,i) = \begin{cases} \mathsf{rec}^+(l,v,i+1) & \textit{if} \quad \exists (l,i) \in \mathrm{Dom}(\mathsf{M}_{\mathsf{elm}}) \quad \textit{and} \quad v \neq \mathsf{M}_{\mathsf{elm}}(l,i) \\ \emptyset & \textit{if} \quad v = \mathsf{M}_{\mathsf{elm}}(l,i) \\ \{(l,i) \to v\} & \textit{else} \end{cases} \tag{17}$$

and the recursion for deleteion is

$$\mathsf{rec}^-(l,v,i) = \begin{cases} \mathsf{rec}^-(l,v,i+1) & \textit{if} \quad \exists (l,i) \in \mathrm{Dom}(\mathsf{M}_{\mathsf{elm}}) \quad \textit{and} \quad v \neq \mathsf{M}_{\mathsf{elm}}(l,i) \\ \{(l,i) \to v\} & \textit{if} \quad v = \mathsf{M}_{\mathsf{elm}}(l,i) \\ \emptyset & \textit{else} \end{cases} \tag{18}$$

The recursion differs from the traditional for-loop iteration because each value set from label l, $\mathsf{MM}(l)$, may have a different size.

4.2 Encrypted Multi-maps

The recursive map definition of a multi-map is important because it implies that a relational language that can describe both maps and recursion should be able to describe a mulit-map. The next step is to find an encrypted multi-map construction that still preserves the recursive map structure.

In this section, a particular scheme for encrypted multi-maps is recalled and recast into the algebraic view. The scheme is then shown to preserve the recursive map structure of a multi-map.

An encrypted multi-map can be defined as

Definition 6 (Encrypted multi-map)**.** *An encrypted multi-map* $\mathsf{EMM} : L \to 2^V$ *with key* $k \in \{0,1\}^\lambda$ *for a map* $\mathsf{MM} : L \to 2^V$ *has the following operations*

- **Search** : $\mathsf{EMM}(k, l)$ *returns all values associated with label* l
- **Insert** : $\mathsf{EMM} + (k, \{l \to v\})$ *returns a new* EMM *with label-value pair added*
- **Delete** : $\mathsf{EMM} - (k, \{l \to v\})$ *returns a new* EMM *with label-value pair removed*
- **Setup** : $\mathsf{EMM}, k \sim \mathcal{T}(\mathsf{MM}, \lambda)$ *is a probablistic transformation from a multi-map to an encrypted multi-map*

where the key space $K = \{0,1\}^\lambda$

First the correctness of any encrypted multi-map scheme is defined as

Definition 7 (Correctness of an encrypted multi-map)**.** *Given a multi-map* MM *and its encrypted multi-map* $\mathsf{EMM} \sim \mathcal{T}(\mathsf{MM}, \lambda)$ *under some probablistic transformation* $\mathcal{T}$ *with security parameter* λ*, the operations of* MM *and* EMM *are computationally indistinguishable for any label* $l \in L$ *and any value* $v \in V$

$$\Pr\left(\mathsf{MM}(l) = \mathsf{EMM}(k, l)\right) = 1 - \mathsf{negl}(\lambda) \tag{19}$$

This should hold even in the precense of updates

$$\mathsf{MM} \pm \{l, v\} \quad \textit{and} \quad \mathsf{EMM} \pm \{l, v\} \tag{20}$$

Definition 8 (Security of an enrypted multi-map)**.** *An encrypted multi-map* EMM *is* $\mathcal{L}$*-secure against adaptive attacks if for all efficient A there exists an efficient S such that*

$$\Pr\left[A(\lambda, \mathsf{EMM}) = 1\right] - \Pr\left[S(\lambda, \mathcal{L})\right] = \mathsf{negl}(\lambda) \tag{21}$$

The Π_{bas} scheme can be formally defined as

Definition 9 (Π_{bas})**.** *The* Π_{bas} *scheme defines a tuple of functions* $(\mathsf{trapd}, \mathsf{labf}, \mathsf{valf}, \mathsf{rec}, \mathsf{rec}^{\pm})$ *defined over a map* $\mathsf{M}_{\Pi_{\text{bas}}}$ *where* $\mathsf{trapd}, \mathsf{labf}, \mathsf{valf}$ *are always computed on the client and* $\mathsf{rec}, \mathsf{rec}^{\pm}$ *are always computed on the server. The* Π_{bas} *implements* EMM *by*

- **Search**$_{\Pi_{\text{bas}}}$: $\mathsf{EMM}_{\Pi_{\text{bas}}}(k, l) = \mathsf{rec}(\mathsf{trapd}(k, l, 1), 1)$
- **Insert**$_{\Pi_{\text{bas}}}$: $\mathsf{EMM}_{\Pi_{\text{bas}}} + (k, \{l \to v\}) = \mathsf{M}_{\Pi_{\text{bas}}} \cup \mathsf{rec}^{+}(\mathsf{trapd}(k, l, 1), \mathsf{valf}(\mathsf{trapd}(k, l, 2), v), 1)$
- **Delete**$_{\Pi_{\text{bas}}}$: $\mathsf{EMM}_{\Pi_{\text{bas}}} - (k, \{l \to v\}) = \mathsf{M}_{\Pi_{\text{bas}}} \setminus \mathsf{rec}^{-}(\mathsf{trapd}(k, l, 1), \mathsf{valf}(\mathsf{trapd}(k, l, 2), v), 1)$
- **Setup**$_{\Pi_{\text{bas}}}$: $\mathcal{T}_{\Pi_{\text{bas}}}(\mathsf{MM}, \lambda) = (\mathsf{trapd}, \mathsf{labf}, \mathsf{valf})(\mathsf{MM}, \lambda)$

The Π_{bas} scheme can be seen as a transformation from the recursive map representation of a multi-map to another recursive map structure that preserves correctness and security. The transformation between a mutli-map and an encrypted multi-map is a tuple of random functions $\mathcal{T}_{\Pi_{\text{bas}}} = (\mathsf{trapd}, \mathsf{labf}, \mathsf{valf})$ with key k unifromaly drawn from all λ-bit strings. The transformation takes the recursive map $\mathsf{M}_{\mathsf{elm}}$ of a multi-map MM, derives a secret key k at random, and performs the following replacement

- Let the trapdoor function be

$$\mathsf{trapd}_t = \mathsf{trapd}(k, l, t) = \mathcal{F}(k, l||t) \quad \text{for} \quad t = 1, 2 \tag{22}$$

- label (l, i) is replaced with the pseudorandom value

$$\mathsf{labf}(\mathsf{trapd}_1, i) = \mathcal{F}(\mathsf{trapd}_1, i) \tag{23}$$

- the value v_i is replaced with the encrypted value

$$\mathsf{valf}(\mathsf{trapd}_2, v_i) = \mathcal{E}(\mathsf{trapd}_2, v_i) \tag{24}$$

  where the symmetric decryption is the inverse

$$\mathsf{valf}^{-1}(\mathsf{trapd}_2, v_i) = \mathcal{D}(\mathsf{trapd}_2, \mathcal{E}(\mathsf{trapd}_2, v_i)) = v_i \tag{25}$$

Let the resulting relation be $\mathsf{M}_{\Pi_{\text{bas}}}$. It remains to be shown that $\mathsf{M}_{\Pi_{\text{bas}}}$ is a map that implements the EMM correctly and securely. But before moving forward, whether $\mathsf{M}_{\Pi_{\text{bas}}}$ is still a map under the transformation requires some closer inspection, because the search and update protocols are defined based on this property. This turns out to also be the necessary condition for a $\mathsf{M}_{\Pi_{\text{bas}}}$-based EMM to be correct.

The first point to note is the randomness. The tranformation functions are random functions but the randomness hinges entirely upon the secret key k in the pseudorandom function trapd. But once trapd is determined given k, the other functions are all deterministic functions.

The second aspect is the distinctness. This is a necessary condition for the correctness of EMM. The hope is that these transformation functions are all one-to-one[1] with the knowledge of the secret key. For an negative example, suppose trapd were not one-to-one. Then some labels $l' \neq l''$ from MM are mapped under such trapd to the same trapdoors $k_{l',1} = k_{l'',1}$. Then labf would produce the same transformed label $\mathsf{labf}(k_{l',1}, \cdot) = \mathsf{valf}(k_{l'',1}, \cdot)$. This collision means the transformation lost information about the distinctness of labels in MM, and so a correct EMM could not be possibly constructed.

Luckily the distinctness can be established with overwhelming probability, provided that the domain and the co-domain of the function is of the same size. Because the randomness hinges upon trapd, so it is necessary and sufficient that trapd is indistinguishable from one-to-one for the rest to follow. Note that trapd is a pseudorandom function, and so is not guaranteed to be invertible. But the Switch Lemma suggests that a pseudorandom function is computationally indistinguishable from a pseudorandom permutation which is always invertible with knowledge of the key assuming large enough key size λ. So if the key space of is chosen to be of such size λ, then trapd is effectively one-to-one. Finally, it is safe to write $\mathsf{M}_{\Pi_{\text{bas}}}$ as a map

$$\mathsf{M}_{\Pi_{\text{bas}}} : \left\{ \mathsf{labf}(\mathsf{trapd}_1, i) \to \mathsf{valf}(\mathsf{trapd}_2, v_i) \mid l \in \text{Dom}(\mathsf{MM}) \quad \text{and} \quad v_i \in \mathsf{MM}(l) \right\} \tag{26}$$

[1] Only for correctness. For security, one-to-one valf is not sufficient for precluding leakage of intersection among value sets from different labels, and so it needs to be strengthen.

The search on $\mathsf{M}_{\Pi_{\text{bas}}}$ can then be defined as the recursive part of the recursive map

$$\mathsf{EMM}_{\Pi_{\text{bas}}}(k, l) = \mathsf{rec}(\mathsf{trapd}(k, l, 1), 1) \tag{27}$$

where the recursion reads

$$\mathsf{rec}(\mathsf{trapd}_1, i) = \begin{cases} \left\{ \mathsf{valf}^{-1}\Big(\mathsf{trapd}_2, \mathsf{M}_{\Pi_{\text{bas}}}(\mathsf{labf}(\mathsf{trapd}_1, i))\Big) \right\} \cup \mathsf{rec}(\mathsf{trapd}_1, i+1) & \text{if} \quad \exists \mathsf{labf}(l, i) \in \mathrm{Dom}(\mathsf{M}_{\Pi_{\text{bas}}}) \\ \emptyset & \text{else} \end{cases} \tag{28}$$

Protocol 1. $\mathbf{Setup}_{\Pi_{\text{bas}}}(\mathsf{MM}, \lambda)$

1. *Compute* $\mathsf{M}_{\mathsf{elm}}$ *from* MM *per definition in (12)*
2. *Generate secret key* $k \sim \{0,1\}^{\lambda}$
3. *Compute the map* $\mathsf{M}_{\Pi_{\text{bas}}}$
4. *Store the secret key* k *on the client*
5. *Send the map* $\mathsf{M}_{\Pi_{\text{bas}}}$ *to the server*

Protocol 2. $\mathbf{Search}_{\Pi_{\text{bas}}}(k, l)$

- *Client: On input* (k, l)
  1. *Reconstruct the derived keys* $(\mathsf{trapd}_1, \mathsf{trapd}_2)$ *using secret key* k
  2. *Send* $t = (\mathsf{trapd}_1, \mathsf{trapd}_2)$ *to the server*
- *Server: On input* $\big(\mathsf{trapd}_1, \mathsf{trapd}_2\big)$
  1. *Compute* $\mathsf{rec}(\mathsf{trapd}_1, 1)$

Leakage The setup leakage of Π_{bas} is the sum of values associated with each label and is optimal. The query leakage has two components, the access pattern and the query pattern. The access pattern includes the the values that are mapped to the label. The query pattern is repitition of queried labels. the trapdoor generated for the queried label, which is deterministic and so cause the query repitition to be leaked.

4.3 An Algebraic View of Recursive Common Table Expression

Because the relational algebra cannto express recursion, there is no hope in emulating the multi-maps and encrypted multi-maps within just the relational agebra. The next dialect beyond the relaional algebra is the datalog that includes the fixed-point operator which provides semantics to recursion [1]. Similarly SQL specifies the recurisve common table expression [10]. However the semantics of the recursive common table expression is not very clear, and so it presents a challange to connect with the encrypted multi-maps via the recurisve map definition. Here inspired by the relational algebra, the recursive common table expression is also viewed under the algebraic lens.

Definition 10 (Recursive Common Table Expression)**.** *A recursive common table expression (RCTE) is a SQL construct that expresses recursion via a union between two subqueries, the base query and the recursive query. The syntax reads*

$$\begin{array}{l} \texttt{With Recursive}\ \ \mathsf{View}\ \ \texttt{As} \\ \quad \mathsf{BaseSubquery} \\ \texttt{Union All} \\ \quad \mathsf{RecursiveSubquery} \end{array} \tag{29}$$

The semantics of the RCTE defines that the recursive view is equivalent to

$$\mathsf{View} = \mathsf{rec}_{\mathsf{rcte}}(1) \tag{30}$$

where the recursive function is defined as

$$\mathsf{rec}_{\mathsf{rcte}}(i) = \begin{cases} \Delta\mathsf{View}_i \cup \mathsf{rec}_{\mathsf{rcte}}(i+1) & \textit{if} \quad \Delta\mathsf{View}_i \neq \emptyset \\ \emptyset & \textit{else} \end{cases} \tag{31}$$

with the view increment

$$\Delta\mathsf{View}_i = \begin{cases} \mathsf{BaseSubquery} & \textit{if} \quad i = 1 \\ \mathsf{RecursiveSubquery} \mid \Delta\mathsf{View}_{i-1} & \textit{else} \end{cases} \tag{32}$$

and the vertical bar in expression $Q \mid V$ means the query Q is computed as if only given the result of view V.

Simply put, the recursive view result consists of

1. The result of the base subquery
2. The result of the recurisve subquery based on the last (base or recursive) subquery result, if the preivous subquery result is nonempty

It is important to emphasize that the recursive subquery only observes the *last* base or recursive subquery, rather than *all* of the previous subqueries. One intuitive example is the typical graph traversal where the next node depends only on the previous node, not necessarily on the entire path. This may appear to be a restiction but in fact it does not preclude other types of recursion that may depend on maximum d number of subqueries in the past, where d is a positive constant. An example of $d = 2$ is the Fibonacci sequence. This type of $d > 1$ recursive dependence can also be easily specified via expanding the schema of the recursive view, and consequently is enforced on the schema of the base subquery and the recursive subquery.

Immediately this definition of the RCTE makes it straightforward to adapt to the recursive map definition of an encrypted multi-map. Such connection is used in the next on emulation.

4.4 Emulation of Encrypted Multi-Maps

The goal of emulation is to express encrypted multi-maps and their operations using a relational language under the relational model. The advantage of doing so is to achieve maximum legacy compliance, meaning that the resulting scheme can run on any existing standard relational database vendors. The result

of the emulation can then be used to compose more complex language features. The main challenge is that, first, structured encryption is not known to be legacy compliant for relational databases, and the encrypted multimaps such as Π_{bas} are unclear to model relationally. On the other hand, the relational language family comes with different scope or domain specificity, ranging from relational agebra, datalog to SQL. So it is of central concern to identify the necessary and sufficient relational language constructs that can emulate an SSE scheme.

Concretely, the emulation under the relational model restricts the capability of the database server to only execute a given query written in standard SQL, without assuming any extension or customization on the database server even in the form of a stored procedure

Protocol 3. *Database server under emulation*

- *Server: On input query Q:*
  1. *Execute Q*
  2. *Return query result* $\mathsf{Res}(Q)$ *to client*

Because a Π_{bas} encrypted multi-map can be viewed algebraically as an recursive map, so the key question is how can an relational language express such structure. What conveniently follows from this algebraic view is a straightforward positive and negative result

- A map is just an algebraic relation, so it is natural to express by any relational language
- The relational algebra cannot express Π_{bas} because it does not support recursion

So this result suggests taht the relational algebra needs to be extended with the recursive operator so as to expres Π_{bas} EMM.

First, simply store the recursive map representation of EMM, the EMM as relation of two attributes, `label` and `value`

$$T_{\Pi_{\text{bas}}}(\texttt{label}, \texttt{value}) \tag{33}$$

where the tuple

$$(\texttt{label}, \texttt{value}) \stackrel{\text{def}}{=} \left\{ (l', v') \mid l' \in \text{Dom}(\mathsf{M}_{\Pi_{\text{bas}}}), v' \in \mathsf{EMM}(l') \right\} \tag{34}$$

Note that the label l' and value v' refer to the encrypted form in the recursive map of an EMM rather than in the plaintext multi-map. The detials are shown in Protocol 4.

To search for all the values associated with a label, the client first reconstructs the derived keys using the secret key, and then uses these derived keys to compose a query to send to the database server. The query makes use of the recursive common table expression (Defition 10) to express the recursion of the search operation on encrypted multi-map. The details of the query is shown in Protocol 5.

Protocol 4. *Emulation for* $\mathbf{Setup}_{\Pi_{\text{bas}}}(\mathsf{MM})$

- *Client: On input (*MM*)*
  1-4. *Compute* EMM *from* MM *as Step 1-4 in Protocol 1*
  5. *Create table by sending query to the database server*

$$\texttt{Create Table} \quad T_{\Pi_{\text{bas}}}(\texttt{label}, \texttt{value}) \tag{35}$$

6. *For each label-value pair $l' \to r'$ in* EMM*, send the query to the database server*

$$\texttt{Insert Into} \quad T_{\Pi_{\text{bas}}} \quad \texttt{Values} \quad (l', r') \tag{36}$$

Protocol 5. *Emulation for* $\mathbf{Search}_{\Pi_{\text{bas}}}$

- *Client: On input* (k, l)

1. *Reconstruct the derived keys* $(\mathsf{trapd}_1, \mathsf{trapd}_2)$ *as Step 1 in Protocol 2*
2. *Send this following query to the database server*

Query 1 (**RCTE** for $\mathbf{Search}_{\Pi_{\text{bas}}}$)

$$\begin{aligned}
&\texttt{With Recursive View As} \\
&\quad \pi_{\texttt{value},i}\, \sigma_{\texttt{label}=\mathcal{F}(\mathsf{trapd}_1,i)} \Big(T_{\Pi_{\text{bas}}} \times \rho_i\{1\} \Big) \\
&\quad \texttt{Union All} \\
&\quad \pi_{\texttt{value},i} T_{\Pi_{\text{bas}}} \ltimes_{\texttt{label}=\mathcal{F}(\mathsf{trapd}_1,i+1)} \mathsf{View} \times \pi_{i+1\to i} \mathsf{View} \\
&\pi_{D(\mathsf{trapd}_2,\texttt{value})\to\texttt{value}} \mathsf{View}
\end{aligned} \tag{37}$$

Efficiency The lowerbound of the encrypted multi-map lookup is $\omega(1)$ meaning one access to retrieve all associated values. In relational model all values are "flatten" into multiple rows, so this lowerbound can only be achieved if these values are packed into one row. The reasonable expected runtime for the relational model without packing is then $O(\mathsf{MM}(l))$ for a label w, meaning only have the same number of accesses as the mumber of values to retrieve.

The efficiency of the RCTE is not immediately clear from its query form. But the semantics in Π_{bas} RCTE admits an execution that is as efficient as the straightforward procedral implementation with hash tables. One observation is that there are joins involved in the base query and the recursive query. But these are specific type of joins that admit optimizations. Therefore the efficiency is preserved at the semantis level as the following theorem establishes.

Theorem 1 (RCTE preserves efficiency)**.** *The RCTE for* Π_{bas} *incurs the same linear time read access* $O(\mathsf{MM}(l))$ *as* Π_{bas} *implemented by a hash table, for each label* l*.*

Proof sketch. Different implementation of the RCTE may have different observed efficiency, but here the argument is laid out for whether the *semantics* of RCTE admits an implementation that is as efficient as claimed.

The cross product in the base query has a constant right operand, so this join simply can be replaced by having a constant column added to each row. In addition, there is a hash index on the label of the $T_{\Pi_{\text{bas}}}$ to ensure amortized constant time per access. This means that the base query for counter $i = 1$ only incur at most one access to retrieve the first value if any.

The recursive query bases its input View on previous base or recursive query. The first recursive query will base on the base query. The left semi join indicates that the result of the input View is used to filter the $T_{\Pi_{\text{bas}}}$ based on the label attribute. This filter has size equal to the size of the input View. Because the base query only had one output, this semi join will only incur one filter access on $T_{\Pi_{\text{bas}}}$ to retrieve again at most one value.

So by induction, this RCTE only incurs $O(\mathsf{MM}(l))$. □

Leakage The RCTE can be shown to leak the same amount of information as the Π_{bas} scheme, based on the observation that these two have the same recursive map structure

Theorem 2 (RCTE has the same leakage)**.** *If the $\mathcal{F}$ and $\mathcal{D}$ executed on the database server are a pseudorandom function and the decryption algorithm in a CPA-secure cipher, then the RCTE has the same leakage as the Π_{bas}.*

Proof sketch. This should be easy to argue by comparing two games where one executes RCTE and one executes the Π_{bas}, and make the reduction of distinguisher to the security of $\mathcal{F}$ and $\mathcal{D}$. The key to the reduction is to see that there is a parallel in the recurive map structure in the emulation and in the Π_{bas}, and therefore gives no advantage to a successful distinguisher who then would have been forced to break the $\mathcal{F}$. □

4.5 Encrypted Sets

A set of labels is an abstract data type that allows one to ask membership question such as whether a certain label exists in a set. Given a finite set $\mathsf{Set} = \{l_1, \cdots, l_S\}_{i=1}^{S}$

$$\mathsf{Set}(l) = \begin{cases} \mathsf{true} & \text{if} \quad l \in \text{Dom}(\mathsf{Set}) \\ \mathsf{false} & \text{else} \end{cases} \tag{38}$$

If the set is stored as a hash set, then the lookup takes amortized worst-case constant time access and linear space in the number of labels.

An encrypted set of labels aims to support the same semantics but with added privacy that the labels as well as the lookup labels are concealed to the untrusted server. For a given Set, the encrypted set is defined as

$$\mathsf{ESet} : \{\mathsf{trapd}(l) \mid l \in \text{Dom}(\mathsf{Set})\} \tag{39}$$

where

$$\mathsf{trapd} = \mathcal{F}(k, l) \tag{40}$$

Emulation. The emulation of an encrypted set first models the encrypted set as a single-attribute relation

$$T_{\mathsf{ESet}} : (\texttt{label}) \tag{41}$$

where the label column stores the trapdoors. Then a hash index is built on the label column to preserves the amortized worst-case constant time lookup as in a hash set implementation.

To query the encrypted set given a single plaintext label l, the query reads

$$\sigma_{\texttt{label}=\mathsf{trapd}(l)} T_{\mathsf{ESet}} \tag{42}$$

which is equivalent to a semi join query

$$\rho_{\texttt{label}}\{\mathsf{trapd}(l)\} \ltimes_{\texttt{label}} T_{\mathsf{ESet}} \tag{43}$$

The query returns an empty set to indicate a false lookup result, or else the randomized label to indicate a true lookup result.

5 Encrypted SQL

The SQL language encryption problem considers how to encrypt and outsource relational data, and query the data directly on the server efficiently and securely. One aspect of the encrypted SQL is that the data under the relational model are encrypted using CPA-secure ciphers, and therefore are not amendable to direct computation. Viewed from the SQL standpoint, the relational model on the ciphertext is completely changed to one that can no longer relate directly the entities, but rather has to go through a collection of "special" encrypted tables to reconstruct any relationship. This change in the encrypted relatioanl model also adds a dimension of complexity to the problem of encrypted SQL.

Another aspect of the encrypted SQL is that the relational language is complex and compositional. Therefore this thesis takes the compositional approach by focusing on encrypting a core set of relational operators using emulation of a set of encrypted data structures. The key idea here is that if each one of these operators can be transformed from operating on plaintext data to operating on encrypted data through encrypted multi-maps, then its operator semantics remains the same, and therefore can still compose to form complex but semantically equivalent queries over the encrypted data.

The challenge is to design a scheme that translates a query against the plaintext relatioanl model to a query against the ciphertext relational model with only relational operators that the unmodified database server can understand and execute.

The previous section on emulation provides a path towards this goal. What emulation provides is a way to represent and operate encrypted data structures relationally. The rest of the work is then to study how to bootstrap from the emulation to implement encrypted relational operators such as filters and joins.

There are different ways to build encrypted operators from encrypted data structures, and the differences result in trade-offs in security and effciency. This is the central part of this thesis to thouroughly examine these differences.

First, the encrypted multi-maps proposed in the literature [16] is applied directly through emulation, and results in the so-called independent encrypted operators in Section 5.2 (The **dex-spx** scheme, Section 8.2). Both the security and the efficiency are suboptimal and subsequently improved by taking the new approach to design the dependent encrypted operators in Section 5.3 (The **dex-cor** scheme).

Further improvement is to apply the database normal form to provide (1) partioning of the encrypted data structure and (2) a new relational-model-inspired encrypted multi-map that requires less storage and runtime complexities, while preserving the same security. This is to be studied in Section 5.4 (The **dex-nf** scheme).

Then the focus shifts to treat the worst-case suboptimality of the encrypted joins. The central piece is to instead of indexing on the row ids, index on the domains and therefore renders the worst-case optimal join algorithms [20] possible.

Towards SQL-completeness necessitates the study of other encrypted operators and predicates, such as disjunction, negation, range, wildcards, and aggregation. These are the focus of Section 5.6.

5.1 Encrypted Tables

Let table T be a matrix of cells $T[i, j] = v_{i,j}$ for $i \in I, j \in J$. The encryption of table T requires preprocessing. This preprocessing includes randomly permuting the table T by its rows and columns respectively, and assigning a row identifier or *row id* to each randomly permuted row. This produces a randomly permuted table

Definition 11 (Randomly Permuted Table withn Row Identifier)**.** *A table T is randomly permuted and assigned with a row identifier or row id as*

$$T^{\mathcal{P}} \underset{k_I \sim 0,1^\lambda, k_J \sim 0,1^\lambda}{\sim} [1, \cdots, I]' || \mathcal{P}\big(k_I, \mathcal{P}\big(k_J, T[\cdot, j]_{j \in J}\big)[i, \cdot]_{i \in I}\big) \tag{44}$$

where the symbol $V||M$ denotes concatenating vector V to matrix M, and the symbol V' denotes transpose of row vector V. The schema of the randomly permuted table remains the same as the schema of the original table, except that the order of the columns are permuted

$$\mathsf{Schema}(T^{\mathcal{P}}) = \mathsf{rid} || \mathcal{P}(k_J, \mathsf{Schema}(T)) \tag{45}$$

Throughout this thesis, each plaintext table is assumed to have undergone this preprocessing transformation implicitly before obtaining the encrypted table. Because the random permutation is a random 1-to-1 onto function, so in the following text the plaintext table T is used to refer to either the original table T or the randomly permuted table $T^{\mathcal{P}}$, to save some verbosity, unless when explicit differentiation is necessary to avoid ambiguity.

A table is encrypted by applying CPA-secure encryption on each of its cell under the secret key. The encrypted table, denoted as $T^{\mathcal{E}}$, contains

$$\big(\mathcal{E}(k, v_{i,j})\big)_{i \in I, j \in J} \quad \text{for} \quad \forall v_{i,j} = T^{\mathcal{P}}[i, j] \tag{46}$$

The table name and scheme need to be randomized by a pseudorandom function. For table $T^{\mathcal{P}}$ with schema $(\mathsf{rid}, c_1, \cdots, c_J)$, the encrytped table $T^{\mathcal{E}}$ has name

$$\mathsf{Name}(T^{\mathcal{E}}) = \mathcal{F}(k, \mathsf{Name}(T)) \tag{47}$$

and the schema is computed based on the permuted schema in $T^{\mathcal{P}}$

$$\mathsf{Schema}(T^{\mathcal{E}}) = \big(\mathsf{rid}, ({c_j}^{\mathcal{E}})_{j \in J}\big) = \big(\mathsf{rid}, (\mathcal{F}(k, \mathsf{Name}(c_j)))_{j \in J}\big) \tag{48}$$

The full description is listed in Definition 12.

Definition 12 (Encrypted Table)**.** *An encrypted table for table $T : \big(c_1, \cdots, c_J\big)$ is with name* $\mathsf{Name}(T^{\mathcal{E}}) = \mathcal{F}(k, \mathsf{Name}(T))$ *and schema*

$$\mathsf{Schema}(T^{\mathcal{E}}) : \Big(\mathsf{rid}, \mathcal{F}(k, \mathsf{Name}(c_j))_{j=1}^{J}\Big) \tag{49}$$

where $\mathsf{rid} = 1, 2, \cdots, I$ *is the row id in the randomly permuted table $T^{\mathcal{P}}$ by Definition 11. The encrypted table's content is computed based on the same permutation $T^{\mathcal{P}}$ as*

$$\big(\mathcal{E}(k, v_{i,j})\big)_{i \in I, j \in J} \quad \textit{for} \quad \forall v_{i,j} = T^{\mathcal{P}}[i, j] \tag{50}$$

Because the encrypted table cells are computed using CPA-secure encryption, they can no longer support direct computation. Instead, each table cell is indexed by some encrypted data structures through the row id and randomized column name. So the row ids and the encrypted data structures constitute the basis of encrypted operators that operate on the encrypted table. This aspect will be studied in depth in the next sections.

5.1.1 Encrypted Table Operations

The operations on the encrypted tables are limited. Only two operations are possible, the row selection by row id, and the column projection by the randomized column name. Combining row id and column name determines a cell in an encrypted table.

Seucrity. The random permutation used in the preprocessing effectively hides the location mapping between a plaintext table and an enrypted table.

Using a CPA-secure cipher produces ciphertexts for all table cells such that the ciphertexts for values of the same length are indistinguishable from one another. This has stronger security than the Deterministic encryption where the same plaintexts are mapped to the same ciphertexts under the same key.

Because values can of varying lenghts for example in the string data type or the binary data type. For variable-length data types the values need to be padded to the same length during encryption.

Efficiency. The total number of encryptions is equal to the number of cells in a table, therefore the total encryption takes linear time in the number of values. The entire table retrieval also takes linear time in the number of values. Therefore the efficiency is deemed optimal.

5.2 Independent Encrypted Operators

An encrypted operator uses encrypted data structures to implement its semantics over encrypted tables or subqueries of encrypted operators. An encrypted operator is said to be *independent* if it only operates over the encrypted table, and so the only way to compose with other subqueries is through plain relational joins.

5.2.1 Independent Encrypted Filter

First the definition of encrypting a single filter is considered

Definition 13 (An Independence Encrypted Filter)**.** *Given a table* $T : (c, \cdots)$ *and its encrypted counterpart* $T^{\mathcal{E}} : (\mathsf{rid}, c^{\mathcal{E}}, \cdots)$*, construct a multi-map* $\mathsf{EMM}^{\perp}_{\sigma}$ *emulated as a relation* $T^{\perp}_{\sigma} : (\texttt{label}, \texttt{value})$ *such that the plaintext query* $\sigma_{T.c=v}T$ *for any domain value* $v \in \mathrm{Dom}(T.c)$ *can be translated into a ciphertext query*

$$\sigma^{\mathcal{E},\perp}_{T.c=v} T^{\mathcal{E}} \stackrel{\mathsf{def}}{=} T^{\mathcal{E}} \ltimes_{\mathsf{rid}=\texttt{value}} \mathsf{EMM}^{\perp}_{\sigma}(T.c = v) \tag{51}$$

that returns the same encrypted reuslt set. The resulting encrypted filter has the schema

$$\mathsf{Schema}(\sigma^{\mathcal{E},\perp}) : (\mathsf{rid}, c, \cdots) \tag{52}$$

Most of the action happens in how to instatiate the encrypted multi-map in the problem definition. If using a scheme that does not allow for emulation under relational model but a procedural implementation, then it would require extension to the database in order to execute this query.

Kamara and Moataz [16] describes as part of the **spx** scheme that an encrypted multi-map can be constructed to map the domain of the filtered column to the row ids

$$\mathsf{EMM}^{\perp}_{\sigma} : \left\{ v \to \left\{ \mathsf{rid}_i \right\}_{i=1}^{I_v} \mid v \in \mathrm{Dom}(T.c) \right\} \tag{53}$$

Then the associated row ids can be used to look up the rows in the data table to construct the reuslt set.

This thesis then considers how to emulate Kamara and Moataz's approach in relational model using the recurisve map definition. This becomes the basis of the fisrt **dex** scheme, the **dex**-**spx** scheme. First, The Π_{bas} multi-map is used here because it is amendable to emulation under the relational model. Then the recursive map for the domain of $T.c$ is computed in Π_{bas} as

$$\mathsf{M}_\sigma^\perp : \left\{\mathcal{F}(\mathcal{F}(k, v||1), i) \to \mathcal{E}(\mathcal{F}(k, v||2), \mathsf{rid}_i) \mid v \in \text{Dom}(T.c) \quad \text{and} \quad i \in I_v\right\} \tag{54}$$

where the inner application of the pseudorandom function produces the trapdoors for the domain value $T.c = v$

$$\mathcal{F}(k, v||1) \quad \text{and} \quad \mathcal{F}(k, v||2) \tag{55}$$

And the recusion part of the recurisve map follows Section 4.2.

Now a problem arises when the scheme is to generalize to multiple columns, and some of the columns may have overlapping domain values. If this happens, then the trapdoors as defined above would be the same, and therefore lead to the same label for the same domain value but in two different columns. This results in additional setup leakage of the number of overlapping domain values in two different columns. This is a nontrivial leakage that can be avoided by changing the definition of the encrypted multi-map to map from the pairs of column name and the domain value

$$\mathsf{EMM}_\sigma^\perp : \left\{(T.c, v) \to \left\{\mathsf{rid}_i\right\}_{i=1}^{I_v} \mid v \in \text{Dom}(T.c)\right\} \tag{56}$$

which leads to the modification of the Π_{bas} recursive map

$$\mathsf{M}_\sigma^\perp : \left\{\mathcal{F}(\mathcal{F}(k, T.c||v||1), i) \to \mathcal{E}(\mathcal{F}(k, T.c||v||2), \mathsf{rid}_i) \mid v \in \text{Dom}(T.c) \quad \text{and} \quad i \in I_v\right\} \tag{57}$$

Notice that the trapdoors are now dependent on both the column name and the domain value. By definition each column has only unique domain values. So the aforementioned leakage of overlapping domain values in multiple columns is avoided. This same trick is used for other operators too.

This encrypted multi-map can be readily emulated in the relational model as a table with schema, as described in Section 4.4.

$$\begin{aligned} &T_\sigma^\perp(\texttt{label}, \texttt{value}) = \\ &\left\{(\mathcal{F}(\mathcal{F}(k, \mathsf{Name}(T.c)||v||1), i), \mathcal{E}(\mathcal{F}(k, \mathsf{Name}(T.c)||v||2), \mathsf{rid}_i)) \mid v \in \text{Dom}(T.c) \quad \text{and} \quad i \in I_v\right\} \end{aligned} \tag{58}$$

So the abstract query of encrypted filter is instantiated with the RCTE expression on $T_\sigma^\perp$

Query 2 (**RCTE**$_\sigma^\perp(T.c = v)$)

$$\begin{aligned} &\texttt{With Recursive View As} \\ &\quad \pi_{\texttt{value},i}\, \sigma_{\texttt{label}=\mathcal{F}(\mathsf{trapd}_1,i)}\left(T_\sigma \times \rho_i\{1\}\right) \\ &\quad \texttt{Union All} \\ &\quad \pi_{\texttt{value},i}\left(T_\sigma \ltimes_{\texttt{label}=\mathcal{F}(\mathsf{trapd}_1,i)} \pi_{i+1\to i}(\mathsf{View})\right) \\ &\pi_{D(\mathsf{trapd}_2,\texttt{value})}\mathsf{View} \end{aligned} \tag{59}$$

where the trapdoors are

$$\mathsf{trapd}_j = \mathcal{F}(k, \mathsf{Name}(T.c)||v||j) \quad \textit{for} \quad j = 1, 2 \tag{60}$$

This RCTE can be substituted back into the problem definition in place of the encrypted multi-map to complete the solution.

Security. The security of the encrypted filter hinges upon the security of the encrypted multi-map, because the outer layer of the operation is only a semi join based on the cleartext rid which is entirely based on the result of the encrypted multi-map first, as shown in (51). So for the setup leakage, the encrypted filter leaks the total number rows in the column. This is equivalent to the setup leakage in the encrypted table. The query leakge consists of the encrypted rows that match the filter.

Efficiency. For the same reason that the outer layer of the operation is only the semi join, the efficiency of the RCTE can be kept at amotized the same nubmer of accesses as the size of the matching result set. This amoritzed complexity is based on the fact the hash index is built on the rid column in the data table T. So for each decrypted value in the result of the encrypted multi-map, there is only amortized constant access through this hash index to retrieve the matching row in the data table if any.

5.2.2 Independent Encrypted Join

The independent encrypted join can be phrased as follows

Definition 14 (An Independent Encrypted join)**.** *Given two tables $T_1 : (c, \cdots)$, $T_2 : (d, \cdots)$, and their encrypted counterparts $({}^{\mathcal{E}}T_1) : (\mathsf{rid}_{T_1}, {c_1}^{\mathcal{E}}, \cdots), ({}^{\mathcal{E}}T_2) : (\mathsf{rid}_{T_2}, {d_1}^{\mathcal{E}}))$, construct an encrypted multi-map $\mathsf{EMM}^{\perp}_{\bowtie}$ and its emulation $T^{\perp}_{\bowtie} : (\mathtt{label}, \mathtt{value}_1, \mathtt{value}_2)$ such that the join query $T_1 \bowtie_{c=d} T_2$ can be translated into an encrypted query*

$$ {T_1}^{\mathcal{E}} \bowtie^{\mathcal{E},\perp}_{T_1.c=T_2.d} {T_2}^{\mathcal{E}} \stackrel{\mathsf{def}}{=} \mathsf{EMM}(T_1.c = T_2.d) \bowtie_{\mathtt{value}_1=\mathsf{rid}_{T_1}} T_1 \bowtie_{\mathtt{value}_2=\mathsf{rid}_{T_2}} T_2 \tag{61} $$

The resulting encrypted join has the schema

$$ \mathsf{Schema}(\bowtie^{\mathcal{E},\perp}) : \left(\mathsf{rid}_{T_1}, \mathsf{rid}_{T_2}, c, \cdots, d, \cdots\right) \tag{62} $$

The **spx** scheme [16] defines an encrypted multi-map for this problem. It maps between each row id from the left operatnd table to the set of row ids in the right that are to join to form the row id result set that satisfy the join condition.

$$ \mathsf{EMM}^{\perp}_{\bowtie} : \left\{\left(\mathsf{Name}(T_1.c), \mathsf{Name}(T_2.d)\right) \to \left\{(\mathsf{rid}_1, \mathsf{rid}_2)\right\} \mid \left\{(\mathsf{rid}_1, \mathsf{rid}_2)\right\} \in Q\right\} \tag{63} $$

where

$$ Q = \pi_{\mathsf{rid}_1, \mathsf{rid}_2} T_1 \bowtie_{c=d} T_2 \tag{64} $$

One may notice that this multi-map definition seems quite unbalanced, because there is always only one label corresponding to the table column names under the join, which is mapped to all the row rid pairs in the join of two tables.

Using the approach in this thesis, the multi-map for join as defined in [16] can be viewed as a recursive map that is amendable to emulation, for example as instantiated by Π_{bas} as

$$ \mathsf{M}^{\perp}_{\bowtie} : \left\{\mathcal{F}\left(\mathsf{trapd}_1, i\right) \to \left(\mathcal{E}\left(\mathsf{trapd}_2, \mathsf{rid}_1\right), \mathcal{E}\left(\mathsf{trapd}_2, \mathsf{rid}_2\right)\right) \mid \left(\mathsf{rid}_1, \left\{\mathsf{rid}_2\right\}\right) \in Q\right\} \tag{65} $$

where the trapdoors are

$$ \mathsf{trapd}_t = \mathcal{F}(k, \mathsf{Name}(T_1.c)||\mathsf{Name}(T_2.c)||t) \quad \text{for} \quad t = 1, 2 \tag{66} $$

Then the emulation proceeds with a table $T^{\perp}_{\bowtie} : (\mathtt{label}, \mathtt{value}_1, \mathtt{value}_2)$ that stores the label and value of the recursive map $\mathsf{M}_{\Pi_{\text{bas}}}$ respectively. Note that now there are two columns correspond to the row id pair as the value. The emulation query reads

Query 3 ($\mathbf{RCTE}^{\perp}_{\bowtie}(T_1.c = T_2.c)$)

$$
\begin{array}{l}
\texttt{With Recursive View As} \\
\quad \pi_{\texttt{value}_1,\texttt{value}_2,i}\, \sigma_{\texttt{label}=\mathcal{F}(\texttt{trapd}_1,i)} \left(T^{\perp}_{\bowtie} \times \rho_i\{1\} \right) \\
\quad \texttt{Union All} \\
\quad \pi_{\texttt{value}_1,\texttt{value}_2,i} \left(T^{\perp}_{\bowtie} \ltimes_{\texttt{label}=\mathcal{F}(\texttt{trapd}_1,i)} \pi_{i+1\to i}(\mathsf{View}) \right) \\
\pi_{D(\texttt{trapd}_2,\texttt{value}_1)\to T_1{}^{\mathcal{E}}.\texttt{rid},D(\texttt{trapd}_2,\texttt{value}_2)\to T_2{}^{\mathcal{E}}.\texttt{rid}} \mathsf{View}
\end{array}
\tag{67}
$$

Then this query is to replace the encrypted multi-map expression in the problem definition to complete the solution.

Security. The leakage of the independent encrypted join reduces to the encrypted multi-map that it uses. In this case the setup leakage includes the number of join pairs of row ids that satisify the join condition. The query leakage on the other hand includes the encrypted result set of the join.

Efficiency. The setup cost is to compute the all pair of row ids for the join. This takes worst-case quadartic time in the size of each table.

To compute the join, the encrypted join needs to read worst-case quadratic number of rows from the encrypted multi-map table as input.

After computing the join pairs of row ids from the RCTE, the encrypted join proceeds to have two joins with the data tables, each of which takes worst-case quadratic number of access to each table.

So the overall computation of encrypted join takes worst-case quadratic time in terms of table length. This matches the asymptotic suboptimal worst case for many join algorithms implemented in the relational databases. But this complexity can be improved. One area of improvement is the quadratic input size. This can help not only cut down the storage size but also gives a large constant improvement in the computation. This direction is pursued in Section 5.4 and in the **dex-nf** scheme in Section 8.4. The other area of improvement is toward the worst-case optimality, and is addresed in Section 5.5 and in the **dex-dom** scheme in Section 8.5.

5.2.3 Composition of Independent Encrypted Operators

For a more complex query that may consist of multiple filters and joins, the problem of composing multiple independent encrypted filters and joins arises. Consider the conjunctive queries of 3 filter predicates and 2 join predicates, named from p_1 to p_5

$$\sigma_{p_1 \wedge p_2} T_1 \bowtie_{p_3} \sigma_{p_4} T_2 \bowtie_{p_5} T_3 \tag{68}$$

One would hope such a direct symbol manipulation for each operator would just work

$$\sigma^{\mathcal{E}}_{p_1 \wedge p_2} T_1{}^{\mathcal{E}} \bowtie^{\mathcal{E}}_{p_3} \sigma^{\mathcal{E}}_{p_4} T_2{}^{\mathcal{E}} \bowtie^{\mathcal{E}}_{p_5} T_3{}^{\mathcal{E}} \tag{69}$$

But this cannot be done for independent encrypted operators for two reasons. The independent encrypted operators by themselves are not composable. They only become composable through additional relational natural inner joins and semi joins on the rids. Also, an encrypted operator does not take *compound* formula

such as $p_1 \wedge p_2$. Instead, the operator with compound formula needs to be decomposed into multiple operators with predicates.

$$\left(\left({T_1}^{\mathcal{E}} \bowtie^{\mathcal{E},\perp}_{p_3} {T_2}^{\mathcal{E}}\right) \ltimes_{\mathsf{rid}_1} \left(\sigma^{\mathcal{E},\perp}_{p_1} {T_1}^{\mathcal{E}}\right) \ltimes_{\mathsf{rid}_1} \left(\sigma^{\mathcal{E},\perp}_{p_2} {T_1}^{\mathcal{E}}\right)\right) \bowtie_{\mathsf{rid}_2} \left(\sigma^{\mathcal{E},\perp}_{p_4} {T_2}^{\mathcal{E}}\right) \bowtie_{\mathsf{rid}_2} \left({T_2}^{\mathcal{E}} \bowtie^{\mathcal{E},\perp}_{p_5} {T_3}^{\mathcal{E}}\right) \quad (70)$$

The query trees for the above three queries are shown in Figure 4. This composition also generalizes to arbitrary number of filters and joins.

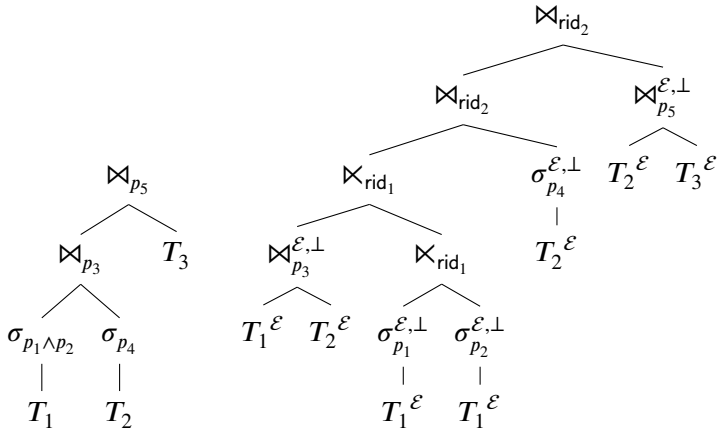

Figure 4: The composition of independent encrypted operators. The left query tree is in (68). The right query is the encrypted query. Notice the excessive table accesses by the independent encrypted operators compared to the plaintext operators. This is the main cause of inefficiency and insecurity.

Another aspect of handling queries against multiple tables is that the encrypted multi-maps for each encrypted operators need to be combined. If left as an independent table, each of the encrypted multi-maps would leak the size of the encrypted result set at setup time. To mitigate this, the encrypted multi-maps for all attributes that the operators can operate on are combined by concatenation. Then the setup leakage reduces to the total size of the universe of operators on the entire database.

This compositional approach forms basis for the **spx** scheme in [16] and in this thesis the emulated **dex-spx** scheme (full details in Section 8.2). Both the security and the efficiency can be improved and is one of the focuses in this thesis in developing other **dex** schemes.

Security. The leakage is straightforward. The setup leakage includes the total size of the universe of opeators on the entire database. This means the sum of row count for each attribute values and each pair of joinnable attributes.

The query leakage depends on the number of encrypted operators and formula, and is simply concatenation of the leakage of each operator and the joined encrypted result set. The reason is because the natural inenr and semi joins that are used to connect the reuslt sets from the encryptd operators only operate on the cleartext row ids, and therefore leaks the structure of the joins on these row ids.

However this query leakage is suboptimal for both the formula and the operators. For an exmaple formula $a \wedge b$, both a and b are filterd indepedently and checked for equality later in the natural semi join. This means the leakage for conjunctive formula leaks not just the intersection but also the leakage of both operands. Ideally, the formula $a \wedge b$ would be checked based on the result of the filter on a first, and the leakage would be only the base filter and the intersection.

This queruy leakage is also suboptimal for the operators. The observation is that some of the filters and joins are dependent in the sense that the result of a join can depend on the result of the filter and vice versa. This means treating them independently would result in accessing some rows that only satisfy individual operator, but not all. These excessive accesses become part of the leakage that this thesis aims to mitigate in Section 5.3. At a high level, these excessive leakages are introduced due to inablity to capture the *dependence* between encrypted operators as well as the plaintext counterparts.

Efficiency. The independent encxrypted filter is asymtotically suboptimal for conjunctive formula. Treating conjunctive clauses independently such as for the example of $a \wedge b$ also means the effiency degrades from worst-case linear time in the minimal size of the two operands to the total size of both tables, and in case of external memory, the worst-case quasi-linear time in an extra sorting phase.

The excessive accesses of rows that only satisfy individual operators may also miss large divisive factors in the runtime complexity. These factors could be very significant in the case of filtered join, such as in the example of filter on T_1 and join T_1 with T_2. Suppose the filter on T_1 has low selectivity such as $1/10$ of T_1. This factor of 10 ideally should be divided by the worst-case quadratic time in the subsequent join of T_1 and T_2. This would have been a huge reduction because the numerator is a quadratic term. In the database theory this optimization is called the push-selection-through-join and pipeline execution of predicates. But the independent encrypted operators are not able to express this optimization and so result in suboptimal time.

5.3 Dependent Encrypted Operators

The independent encrypted join and filter operators are the only encrypted operators supported in the **dex-spx** scheme (full details in Section 8.2). But as shown in the previous section, although the independent encrypted operators can experess conjunctive queries as in the **spx** scheme in [16] and **dex-spx** scheme in this thesis, the reuslting scheme is suboptimal for both security and the efficiency.

One of the key ideas to improve is to introducoe dependence amont encrypted opreators. Dependence can happen in two dimensions. One is within the formula associcated with an operator, and the other is between operators. Both of these dependences are to be addressed using the same approach introduced in this section and the following subsections. The highlevel approach is to use the independent opeator once, and the subsequent operators that are dependent on this operator become the so-called *dependent encrypted operators*. This approach is implemented in the **dex-cor** scheme in full detail in Section 8.3.

For encrypted operators, the key is to operate not just on an encrypted table but also on any subquery of other encrypted operators. The change is based on innovation in the encrypted data structures.

5.3.1 Dependent Encrypted Filter

Two filters cannot have dependence other than over cross join because they by definition operate on separate tables. Therefore the dependent encrypted filter only concerns with two cases

1. Compound formula present within the operator
2. Filter after a subquery of other encrypted operators

Both are fundamentally the same problem

Definition 15 (A Dependent Encrypted Filter)**.** *Given a table* $T : (c_1, c_2, \cdots)$ *and its encrypted counterpart* $T^{\mathcal{E}} : (\mathsf{rid}, {c_1}^{\mathcal{E}}, {c_2}^{\mathcal{E}}, \cdots)$*, and either*

1. *A subformula* $Q = \bigvee_{i=1}^{I-1} \left(c_i = v_i\right)$ *that is the prefix of the compound formula* $c_1 = v_1 \wedge \cdots C_{I-1} \wedge c_I = v_I$*; or*
2. *A subquery* Q *that operates on tables including* T

and such Q *is translated into an encrypted subquery* $Q^{\mathcal{E}}$*, design an encrypted set* ESet *such that the a query of either*

1. *A compound formula:* $\sigma_{Q \wedge c_I = v_I} T$
2. *An atom for a subquery:* $\sigma_{c_I = v_I} Q$

can be expressed as an encrypted query

$$\sigma^{\mathcal{E},\propto}_{T.c_I = v_I} Q^{\mathcal{E}} \stackrel{\text{def}}{=} Q^{\mathcal{E}} \ltimes \mathsf{ESet}^{\propto}_{\sigma}(T.c_I = v_I) \tag{71}$$

Note that in the problem definition a semi join is used, which is different from an inner join in semantics. A semi join is a generalization of filter that uses conjunctive join predicates. In terms of efficiency, its semantics admits linear time execution in terms of the the minimum size of the two operand relations if both are indexed with constant time access on the joined attribute, or else in addition plus the total size of unindexed operand relation(s). The query means to check for each row in the subquery $Q^{\mathcal{E}}$ whether it satisfies the predicate, which is described by the encrypted data structure.

The best encrypted data stucture to use here is an encrypted set. This is because smenatically what is needed in the problem query is the ability to check if a given row satisfies a specified atom. Such questions are naturally fit for a set. In particular, an encrypted set needs to tell whether a given row id in $T.\mathsf{rid}$ contains a given value v

$$\mathsf{ESet}^{\propto}_{\sigma}(T.c = v) : \left\{\mathsf{trapd}(T.c, v, \mathsf{rid}_i) \mid \mathsf{rid}_i \in T.\mathsf{rid}\right\} \tag{72}$$

where the trapdoor serves as a function to encode the set labels

$$\mathsf{trapd}(T.\mathsf{rid}_i) = \mathcal{F}(\mathsf{trapd}_{\sigma}, \mathsf{rid}_i) \tag{73}$$

and

$$\mathsf{trapd}_{\sigma} = \mathcal{F}(k, \mathsf{Name}T.c||v) \tag{74}$$

Then this encrypted set can answer whether a given value is present in the given row under the given column.

To emulate this encrypted set, create an one-attribute, hash-indexed table on the set labels as

$$T^{\propto}_{\sigma} : (\texttt{label}) \tag{75}$$

Then given the subquery as described in the problem definition, the query reads

$$Q^{\mathcal{E}} \ltimes_{\mathsf{trapd(rid)}=\texttt{label}} T^{\propto}_{\sigma} \tag{76}$$

Security. The setup leakage of the encrypted set is the size of the rows in the column. The query leakage now depends on the input subquery. The guarantee is that for previous conjunctive atoms, this encrypted filter will not access more than what has already been leaked in the intermediate result set up until this point. It may still leak the intersection of this atom under consideration with the previous atoms.

Efficiency. Because of the dependence, if the input subquery has reduced result set size due to selectivity of the operators, then this operator will incur only amortized worst-case linear time in terms of the input subquery size.

5.3.2 Dependent Encrypted Join

For joins, there are only two cases, the *filtered join*, the *filter after join* and the *join over join*. By induction, these cases will generalize to arbitrary number of dependent operators.

The problem of dependent encrypted join is

Definition 16 (Dependent encrypted join)**.** *Let* $Q : (a_1, b_1, \cdots)$ *be a subquery that contains* $T_1 : (a_1, b_1, \cdots)$*. Let its encrypted counterpart be* $Q^{\mathcal{E}} : (\mathsf{rid}_1, {a_1}^{\mathcal{E}}, {b_1}^{\mathcal{E}}, \cdots)$*, and the encrypted table be* ${T_1}^{\mathcal{E}} : (\mathsf{rid}_1, {a_1}^{\mathcal{E}}, \cdots)$*. Let* $T_2 : (a_2, b_2 \cdots)$ *be a different table to be joined with* Q*, and its encrypted counterpart* ${T_2}^{\mathcal{E}} : (\mathsf{rid}_2, {a_2}^{\mathcal{E}}, {b_2}^{\mathcal{E}}, \cdots)$ *design encrypted data structures such that a dependent encrypted join can express*

$$Q \bowtie_{T_1.a_1 = T_2.a_2} T_2 \quad for \quad T_1 \in Q \tag{77}$$

as

$$Q^{\mathcal{E}} \bowtie^{\mathcal{E},\propto}_{T_1.a_1 = T_2.a_2} {T_2}^{\mathcal{E}} \stackrel{\mathsf{def}}{=} \mathsf{EMM}(Q^{\mathcal{E}}, T_1.a_1 = T_2.a_2) \bowtie_{\mathtt{value}=\mathsf{rid}_2} {T_2}^{\mathcal{E}} \quad for \quad {T_1}^{\mathcal{E}} \in Q^{\mathcal{E}} \tag{78}$$

such that the encrypted result set corresponds to the plaintext query's result set.

Note that the new encrypted multi-map takes the entire encrypted subquery as an argument, thereby capturing the dependence on the output of this subquery. Such an encrypted multi-map reads

$$\mathsf{EMM} : \left\{\mathsf{rid}_1 \to \left\{\mathsf{rid}_2\right\} \in S\right\} \tag{79}$$

where

$$S = {}_{\mathsf{rid}_1}G_{||}\pi_{\mathsf{rid}_1,\mathsf{rid}_2}\mathcal{P}(k, T_1) \bowtie_{a_1=a_2} \mathcal{P}(k, T_2) \tag{80}$$

This encrypted mulit-map can be seen as taking the row id pairs of the join result, groiuped by the row ids for the left operand table, and concatenate the row ids for the right operand table to a set.

This multi-map can be instantiated by Π_{bas} and viewed as the recurisve map that is amendable to emulation

$$\mathsf{M}^{\propto}_{\bowtie} : \left\{\mathcal{F}\left(\mathsf{trapd}_1, i\right) \to \mathcal{E}\left(\mathsf{trapd}_2, \mathsf{rid}_2\right) \mid (\mathsf{rid}_1, \left\{\mathsf{rid}_2\right\}) \in S\right\} \tag{81}$$

where the trapdoors are

$$\mathsf{trapd}_t = \mathcal{F}\left(k, \mathsf{trapd}_{\bowtie}||\mathsf{rid}_1||t\right) \quad \text{for} \quad t = 1, 2 \tag{82}$$

and

$$\mathsf{trapd}_{\bowtie} = \mathcal{F}\left(k, \mathsf{Name}(T_1.a_1)||\mathsf{Name}(T_2.a_2)\right) \tag{83}$$

Note that the row ids for the left operand which could be either a subquery or a table, is used to construct the trapdoors for the encrypted multi-map, and this serves as the basis for modeling the dependence.

First create a table to store the encrypted map as

$$T_{\bowtie} : (\mathtt{label}, \mathtt{value}) \tag{84}$$

where the label and value columns store those of the recursive map $\mathsf{M}^{\propto}_{\bowtie}$. Then the encrypted query first establishes the encrypted multi-map expression by emulation $\mathsf{EMM}(Q^{\mathcal{E}}, T_1.a_1 = T_2.a_2)$ as

Query 4 ($\mathbf{RCTE}^{\propto}_{\bowtie}(T_1.a_1 = T_2.a_2)$)

$$
\begin{aligned}
&\texttt{With Recusive View As}\\
&\quad \pi_{Q^{\mathcal{E}}.*,{T_1}^{\mathcal{E}}.\mathsf{rid}_1,\texttt{value},i}\, T_{\bowtie} \bowtie_{\texttt{label}=\mathcal{F}(\mathcal{F}(\mathsf{trapd}_{\bowtie},{T_1}^{\mathcal{E}}.\mathsf{rid}_1||1),i)} \left(Q^{\mathcal{E}} \times \rho_i\{1\}\right)\\
&\quad \texttt{Union All}\\
&\quad \pi_{Q^{\mathcal{E}}.*,{T_1}^{\mathcal{E}}.\mathsf{rid}_1,\texttt{value},i}\left(T \_\bowtie \ltimes_{\texttt{label}=\mathcal{F}(\mathcal{F}(\mathsf{trapd}_{\bowtie},{T_1}^{\mathcal{E}}.\mathsf{rid}_1||1),i)} \pi_{i+1\rightarrow i}\mathsf{View}\right)\\
&\pi_{Q^{\mathcal{E}}.*,\mathcal{D}(\mathcal{F}(\mathsf{trapd}_{\bowtie},{T_1}^{\mathcal{E}}.\mathsf{rid}_1),\texttt{value})\rightarrow\texttt{value}}\mathsf{View}
\end{aligned}
\tag{85}
$$

The trapdoors are computed as described above based on the predicate $T_1.a_1 = T_2.a_2$. Then the composition with T_2 proceeds with

$$
\mathbf{RCTE}^{\propto}_{\bowtie}\left(Q^{\mathcal{E}}, T_1.a_1 = T_2.a_2\right) \bowtie_{\texttt{value}=\mathsf{rid}_2} {T_2}^{\mathcal{E}} \tag{86}
$$

This query completes the solution to the problem definition above. Note that the encrypted subquery $Q^{\mathcal{E}}$ is embedded inside the RCTE to achieve desirable effect of dependence, where the output of the subquery is directly used as input to the recursion. Therefore the output of the RCTE is a *dependent* result set of the subquery. This can be represented as a query tree in Figure 5.

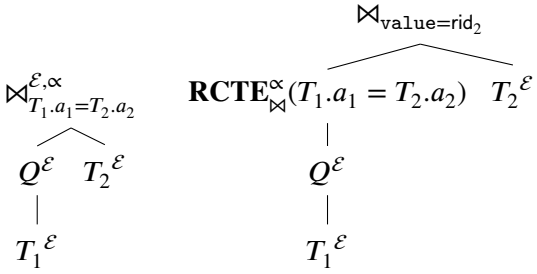

Figure 5: The dependent encrypted join. The right query expands the the encrypted join operator in the left query with the RCTE expression.

5.3.3 Composition of Dependent Encrypted Operators

The dependence of encrypted operators lead to optimiziation of the efficiency and security of complex queries. For the same example in (68), the independent operators introduce excessive table accesses. These can be reduced by using the dependent operators instead

$$
\left(\sigma^{\mathcal{E},\propto}_{p_2}\sigma^{\mathcal{E},\propto}_{p_1}{T_1}^{\mathcal{E}}\right) \bowtie^{\mathcal{E},\propto}_{p_3} \left(\sigma^{\mathcal{E},\propto}_{p_4}{T_2}^{\mathcal{E}}\right) \bowtie^{\mathcal{E},\propto}_{p_5} {T_3}^{\mathcal{E}} \tag{87}
$$

The corresponding query tree is in Figure 6.

5.3.4 Conjunctive Filter Formula

Interestingly the dependent operators are not always better than the independent ones. For the above example, the *entire* table ${T_1}^{\mathcal{E}}$ needs to be accessed to determine the intermediate result of the first filter predicate p_1. This incurs the worst case of the dependent filter that the dependent subquery is actually the entire table scan. Though semantically correct, this excessive access can be reduced by using an

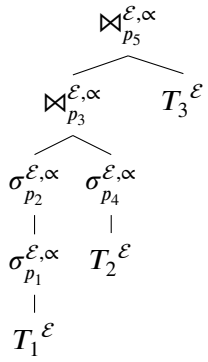

Figure 6: The composition of dependent encrypted operators for the same plaintext query in (68).

independent filter operator first as $\sigma^{\mathcal{E},\perp}_{p_1}{T_1}^{\mathcal{E}}$, and the subsequent conjunctive predicates on the same table remain the same by using the dependent ones. This reduces the first predicate access to only the result set. This introduces a constant divisive factor to the efficiency of conjunctive filter formula.

5.3.5 Conjunctive Join Formula

Composition in (86) can be extended for the case of dependence in the formula. In this case the the compound formula contains multiple join predicates, and all are concerned with the same two tables. For example

$$Q \bowtie_{T_1.a_1=T_2.a_2 \wedge T_1.b_1=T_2.b_2 \wedge T_1.c_1=T_2.c_2} T_2 \tag{88}$$

As analyzed in the previous Section 5.2, the independent approach would result in three encrypted joins and combine them subsequently using two natural joins. This may be reasonable if the subquery $Q = T_1$, but for more complicated subquery that might already result in reduction in input size, it may be worth to introduce dependence again as

$$\mathbf{RCTE}^{\propto}_{\bowtie}\big(\mathbf{RCTE}^{\propto}_{\bowtie}\big(\mathbf{RCTE}^{\propto}_{\bowtie}\big(Q^{\mathcal{E}}, T_1.a_1 = T_2.a_2\big), T_1.b_1 = T_2.b_2\big), T_1.c_1 = T_2.c_2\big) \bowtie_{\texttt{value=rid}_2} {T_2}^{\mathcal{E}} \tag{89}$$

Security. The security is improved because now the excessive access is reduced because the dependence is introduced such that subsequent join operator is guaranteed to not enlarge the existing access to the intermediate result set of prior operators. Therefore if the first operator produces small accesses on a table thanks to selectivity, then the dependent join operator would not access more of the said table. Note that this is not true for the independent case.

Efficiency. The dependent join operator also improves the efficiency by introducing a potentially large divisive factor for the worst-case quadratic input complexity for the join based on the dependent filtered subquery. This improvement has been validated in the database query optimization literature, and for the first time this optimization is available to the encrypted relational database using SSE.

5.3.6 Simulation of Independent Join

Recall that an independent join always operate on a table rather than on a subquery. So if the encrypted subquery in an dependent join equals to the encrypted table of ${T_1}^{\mathcal{E}}$, then the RCTE for dependent join has

the same semantics as the independent encrypted join. This means that the dependent encrypted join can be used just as an independent encrypted join.

Furthermore, the efficiency of the independent join is also improved significantly throught the simulation by dependent join.

Theorem 3 (Efficiency of Simulated Indepdent Join)**.** *The efficiency of the independent join is improved through the simulation from the dependent join by a factor of worst-case quadratic to linear in the table size.*

Proof sketch. The best way to see this is by comparing the query tree of an indepedent join and that of a simulated indepednet join. Consider the join between T_1 and T_2 after a subquery Q that involves T_1

$$Q \bowtie_{T_1.c_1=T_2.c_2} T_2 \quad \text{for} \quad T_1 \in Q \tag{90}$$

Assume the subquery Q has been translated to an encrypted subquery $Q^{\mathcal{E}}$ of the same form, so the comparison in question is just the tranlation of the last join by the independent join versus by the simulated independent join. The resulting query trees are shown in Figure 7. The left qeury uses the independent join, and the right query uses the simualted independent join.

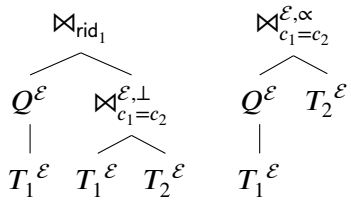

Figure 7: The comparison of an independent join and a simulated independent join in the proof of Theorem 3.

At the first glance, the left query has redundancy in the use of ${T_1}^{\mathcal{E}}$, and also it uses an additional inner join to "glue" together the results from the encrypted subquery and the encrypted join. By contract, the right query mimics closely the structure of the plaintext query, by having exactly one encrypted join in place of the plaintext join. This redundancy of operators contribute to the inefficiency, as analyzed below.

The efficiency of the query can be characterized by the intermdidate data size. The two branches under the root node in the left query has output size worst-case $O(Q^{\mathcal{E}})$ and $O(T^2)$ respectively, assuming the size of each table is $O(T)$. By contrast, the two branches under the root node in right query has only output size worst-case $O(Q^{\mathcal{E}})$ and $O(T)$ respectively. So the right query has smaller intermediate data size.

Lastly, the computation at the root node of the left tree has worst-case $O(Q^{\mathcal{E}} \cdot T^2)$, compared to the right tree $O(Q^{\mathcal{E}} \cdot T)$.

Combining the intermediate data and the computation, the right tree using the simulated independent join has asymtotically improves the efficiency from quadratic to linear in the table size.

Because the argument uses the generaic subquery Q which can represent any query structure, so by induction the improvement is established. □

5.3.7 Security against the Malicious Server

Proposed work

So far the focus has been on the semi-honest server. A malicious server can deviate from the prescribed protocol, for example by modifying the client queries. The seucrity against the semi-honest in general does not imply the security against the malicious server.

The dependent join described in the preivous section has improved security for the semi-honest server, but unforunately the security can be shown to be the same as the independnet join against the malicious server. The high-level idea is that the master trapdoor for the join is computed based on the column names, and so even though the derived trapdoors for the encrypted multi-map are depenent on the row ids of the previous subquery, the master trapdoor remains the same for the row ids that do not belong to the previous subquery. A malicious adversay can then use this master trapdoor to uncover not only the result of the dependent join but also the join as if being independent. So this makes the dependent join leaks the same amount of information as the independent join.

To improve the security against the malicious adversary, the master trapdoor needs to be less generous than based on just the column names. The result of the entire query can be first viewed as a join graph of intermediate results, and the application of operators can be viewed as a traversal of subgraph. Then the seucrity can be viewed as how to allow traversal of one subgraph to not disclose any other subgraphs. The solution then is to use the information about entire traversal path to construct the trapdoor. The storage will likely to increase, but luckily with a multiplicative the width dimension of a table, which is often the much smaller dimension than the length.

5.4 Encrypted Normal Form

Proposed work

Many data warehouses tend to organize their data model in the so-called 3rd normal form, for example the TPC-H Benchmark [8]. In a nutshell, it refers to a representation of the entites and relationships among the data that has reduced redundancy and anamolies. In this regime, joins are typically between primary keys and foreign keys.

So far the encrypted operators are agnostic about the normal form, and therefore results in suboptimal efficiency mianly in several ways.

- The worst-case quadratic storage size for each join. This storage blowup is closely associated with the efficiency in the form of quadratic input size to the join operator.
- Indirection between the encrypted data structures and the base data tables. The encrypted multi-maps used in encrypted operators are by definition have their values (row ids) encrypted under CPA-secure cipners. The subset of these row ids are revealed by the trapdoors during query time to recover the association with the base data, by means of a join. So in the case of dependent operators, this results in additive quadratic terms in the time complexity, each of which is in terms of the size of the intermediate data.

The main cause of these problems is that the encrypted multi-map were designed for the *key-value model*, for example the document keyword store[2] A document keyword store has the following features

- Each document may have different number of keywords
- Each keyword may associate with multiple different, possibly intersecting documents

This theis however considers the relational data model, and it has its own features

[2]The document keyword store is commonly referred to as the “database” in the crytography literature.

- The relational data are structured in rows and columns in tables. Each row has the same number of cells, and each column has the same number of rows.
- In general, each cell stores the atomic or elementary values

In contrast to the document keyword store, the implications are

- Each row in a column can only contain one unique value
- Each unique value can appear in multiple rows in a column, but never intersecting

5.4.1 Semi-Encrypted Multi-Map

Proposed work

This section describes a new *semi-encrypted multi-map* that leverages the unique featuers in relational model. The semi-encryptd multi-map leaves its values (row ids) in cleartext. Perhaps surprisingly, this less secure building block can be used to build encrypted SQL with the same security as the encrypted multi-map. This requires a formal security reduction to establish.

In terms of efficiency, the cleartext values (row ids) in the encrypted multi-map admits two important optimization

- Direct indexing from the database
- Partioning and collocating the emulated semi-encrypted multi-map with the base data table

These two new optimizations significantly improve the performance of the encrypted operators and the overal encrypted query, by removing the additive quadratic terms in terms of the size of the intermediate data.

5.5 Worst-Case Optimal Encrypted Join

Proposed work

The worst-case complexity is $O(T^{1.5})$ rather than the commonly used join algorithms that incur quadratic time in size of table T. Several new join algorithms that match this lower bound have been discovered in the last decade.

The encrypted joins used so far do not match the lower bound. So this section studies how to make encrypted joins worst-case optimal by borrowing ideas from Cash et al. [4].

5.6 Other Encrypted Operations and Predicates

Proposed work

5.6.1 Semi-joins, Anti-joins and Set Operations

Proposed work

This section discusses how to use simulation to implement semi-joins, anti-joins, negations, set operations, etc.

5.6.2 Negation

Proposed work

This seciton pursues the idea of simulating negation by the encrypted set for dependent encrypted filters and the set difference operator. However the surprising result is that the ordering of the negation and other conjunctive or disjunctive clauses may result in different leakage.

5.6.3 Range Query

Proposed work

Range predicates are implemented using native encrypted multi-maps built on top of the ideas in Faber et al. [11].

5.6.4 Insertion, Deletion and Updates

Proposed work

The main contribution in this part is non-interactive database updates by building on emulation. This marks an advance beyond the **spx** work in Kamara and Moataz [16].

5.7 Encrypted Query Translation

The encrypted query translation to map a *plaintext query* to a *encrypted query*. The plaintext query is composed by the user who only has the plaintext data model in mind. The encrypted query is computed by the client upon seeing the plaintext query, which satisfies two important properties

1. The encrypted query is against the encrypted data model under a **dex** scheme
2. The encrypted query consists only of encrypted operators
3. The encrypted query preserves the same semantics as the plaintext query by returning the same result set under encryption for all possible database instances.

A simple translation algorithm can be defined as follows

Protocol 6 (Encrypted Query Translation)**.**

- *Input: Plaintext* Query

1. $Q \leftarrow$ *Parse* Query *as query tree*
2. *Traverse the tree* Q *using post-order*
   (a) ${Q_l}^{\mathcal{E}} \leftarrow$ *Visit the left subtree*
   (b) ${Q_r}^{\mathcal{E}} \leftarrow$ *Visit the right subtree*
   (c) *Visit the current root node*
   (d) *Translate the operator* op *in the root node into an encrypted operator* ${Q_{\mathsf{root}}}^{\mathcal{E}} = \mathsf{op}^{\mathcal{E}}$ *or a tree of encrypted operators* ${Q_{\mathsf{root}}}^{\mathcal{E}} = \{\mathsf{op}^{\mathcal{E}}\}$
   (e) *Connect* ${Q_{\mathsf{root}}}^{\mathcal{E}}$ *with left and right subtrees* ${Q_l}^{\mathcal{E}}, {Q_r}^{\mathcal{E}}$
   (f) *Return* ${Q_{\mathsf{root}}}^{\mathcal{E}}$

The deatils of the translation from each plaintext operator is studied further in Section 6 on Query Optimization. This thesis proposes four **dex** schemes that make different choices in the encrypted operator construction. The full details are presented in Section 8.

6 Query Optimization

Proposed work

The previous section on Encrypted SQL establishes how to translate plaintext query into an encrypted query that can be readily executed on the server running an unmodified relational database. Howeve it is unclear whether the encrypted query is optimal in the sense that it admits the most efficient implementation or exectuion strategy in the database. It is an issue reminiscent of the query optimization in the plaintext database context, but has several important distinctions. First, traditionally the query is opitmized against the *same* data model, whereas in the case of **dex** the translation process produces two distinct but related queries against two data models. In particular it is worthwhile to investigate the following questions:

1. What does query optimiation mean?
2. Is it true that optimality in the plaintext data model implies the optimality in the the encrypted data model?
3. Does query optimization leak more information about the plaintext query or data?

Query optimiztion is an important topic in relational databases. It naturally arises for complex queries. The complexity of a query depends on the number of operators and their ordering. Due to the expressiveness of the relational langauge, the same query can have multiple semantically equivalanet queries, but each may admit different implementation details, and so maximizing the query efficiency means to find an equivalent query that admits the most efficient implementation.

Query optimization can happen in both the physical level and the logical level of the database. The physical level usually covers storage and indexing techniques, buffer management and data statistics. The logical level, the algebraic equivalence of the relational language is used to transform the query to another query that can be implemented more efficiently.

The relational datatbase encryption schemes introduced in this thesis present a new challenge to query optimization. For an encrypted relational database, the user query is first tranformed into a client query using encrypted relational language against the encrypted data model. The encrypted data model is very different from the plaintext data model because it hides many details about the data, such as the distribution information, while it only leaks a small amount of information during setup and query phrase. Therefore query optimization becomes quite different and limited for the encrytped query by the security design.

Second, the legacy compliance property of a **dex** scheme dictates that the control of the physical level optimization is left solely to the discretion of the database. Therefore the best an encryption scheme can do is to ask the database to build enough access methods (e.g. indices) such that the theoretical efficiency of the encrypted query can be preserved at runtime.

Therefore the approach taken in this thesis is that the logical level optimization is integrated into the translation process on the client, as a two-phase process. First the plaintext query is optimized based on a set of rules. Then the encrypted query is optimized again by a different set of rules. Cruicially, these rules are only limited to information within the leakage and nothing more. Notice that this process is outside of the untrusted server and the databass, and resides soley in the trusted client. The physical level optimization concerns mainly the auxiliary indices on the database during setup time.

Encrypted query optimization exhibits both similarities and differences versus the tranditional query optimization. For example, the push-selection-through-join rule is the same for encrypted queries, but how the related operators are composed via either independent or dependent operators become a unique

problem to encrypted queries. Join order selection has the same minimization target between plaintext and encrypted queries. But many-to-many join fractorization is a brand new problem for encrypted queries that does not have parallel in the traditional query optimziation world.

It is also important to point out that the boundary for optimization in traditional databases is between operators. But for then encrypted query, the encrypted operators are based on emulation, therefore optimization requires additional inspection *inside* the encrypted operator.

A full study of encrypted query optimziation requires quantitative cost models for the encrypted operators. The cost model construction is an important proposed work in this thesis.

In the next sections, a quantitative understanding is presented, and the empirical evidence is gathered in Section 10. These reuslts will pave way for a full quantitative analysis of the encrypted query optimziation rules.

6.1 Cost Modeling

Proposed work

A cost model is a mathematical model that estimates the efficiency of different relational operators and their composition. A new cost model is needed to accurately describe the cost of the encrypted operators that are built on top of emulation and simulation.

In the following sections, a less formal approach is taken to motivate the definition of the problem and derive the new optimization rules for this problem.

6.2 An Motivating Example

An example is helpful in showing the subtleties in the **dex** query optimization problem and serves as motivation to the proposed solution later. Given a pliantext data model that contains tables $\left\{T_i : (a_i, b_i, c_i, \cdots)\right\}$, and their encrypted counterparts $\left\{T_i{}^{\mathcal{E}} : (\mathsf{rid}_i, a_i{}^{\mathcal{E}}, b_i{}^{\mathcal{E}}, c_i{}^{\mathcal{E}}, \cdots)\right\}$, and the emulated encrypted data structures. Consider a 3-way join, 2-conjuctive-clause filter plaintext query

$$\sigma_{a_1=10 \wedge b_1=20} T_1 \bowtie_{c_1=c_2} T_2 \bowtie_{d_2=d_3} T_3 \tag{91}$$

It has several possible representation in query trees. Figure 8 shows one naive composition with the filter with two conjuctive clauses applied at the very top of the tree. Note that the execution goes from bottom up on a query tree, so this means the filter is applied last. The right query tree is obtained from pushing the filter through the joins, all the way down to the table. This advantage of this optimization is well validated in research and practice, because the filter usually introduces a large divisive factor based on its selectivity to the worst-case quadratic time of the joins.

6.3 Delay of Data Table Joins

This rule requires inspection inside and between the encrypted operators, and applies to both independent and dependent types. First, consider only using the encrypted operators. The reuslting query tree is shown in Figure 9. Note the four major branches of the tree correspond to the encrypted operator for the two conjunctive clauses of the filter and the two joins respectively. The two conjucntive clauses in the plaintext filter are now broken up into two separate independent encrypted filters, and the semi join is used to combine results from them.

One immediate observation is that though this encrypted query is semantically correct, it is not optimal. There are multiple appearances of the same tables in the tree, such as T_1 for three times and T_2 for

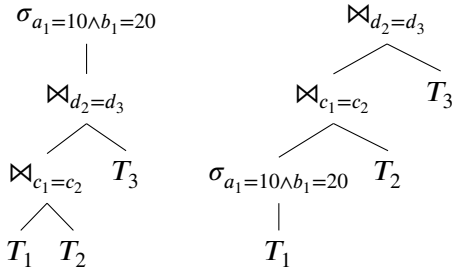

Figure 8: Two equivalent query trees for the example plaintext query in (91). The right query tree is obtained from pushing selection through join on the left query tree.

two times, but they are semnatically only joined once as in the plaintext query. As a result the "gluing" operators have 1 semi join and 3 inenr joins.

Also interestingly, this encrypted query results from no matter which plaintext query tree is used in Figure 8. This reason is that the independent encrypted operators can only be composed indendently by definition. So this is the first sign that the optimality in the plaintext query does not necessarily imply the optimality in the encrypted query in a **dex** scheme.

Conjecture 1 (Query Optimality)**.** *The plaintext query optimality does not imply the encrypted query optimality.*

So query optimization has to be defined also on the encrypted query. The idea is that the data tables like T_i can be delayed in the tree as much as possible to reduce the excessive access. Because the row ids of each table that must appear in the result set can be computed solely based on the emulated encrypted data structures. This optimization is unique to the encrypted operators and has no analogy in the traditional query optimziation.

Rule 1. *Delay data table joins in an encrypted query until all of the row ids that are needed from the data tables are determined from the encrypted data structures.*

This optimization is shown in Figure 10.

By the way, this rule also hints on a conjectured necessary condition on the optimal *privacy* and *efficiency* of an encrypted query, namely

Conjecture 2 (Necessary Condition on Optimal Privacy and Efficiency of an Encrypted Query)**.** *If an encrypted query is optimally private and efficient, then the total access (revealed row ids) to the data tables is minimized to just the rows in each table that participate in the result set.*

Intuitively, any excessive access to the data tables for a given query leaks more than semantically required to answer the query.

6.4 Dependent Encrypted Operators For Correlated Filters and Joins

Another observation based on the previous section is that the indepednent operators are rather ill-suited for the correlation in the conjunctive filter clauses and the 3-way joins. The final, optimized encrypted query still has to incur excessive access in the following ways

- Row ids that satisfy *either* of the two conjunctive filter clauses are read, although only the *intersection* is needed.

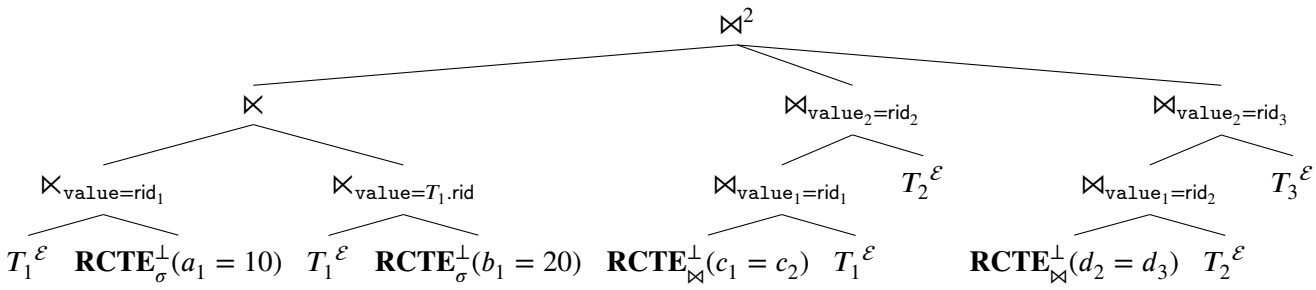

Figure 9: The encrypted query using independent encrypted operators only.

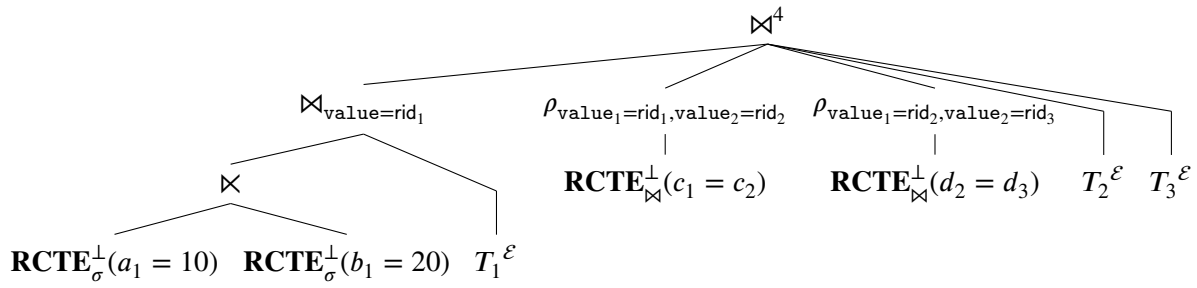

Figure 10: The encrypted query with independent encrypted operators only, with Rule 1 applied.

- One of the joins that involved the table T_1 that is also filtered on, but all row ids that participate in the join are read, instead of just the filtered ones.

Now to further improve the query, the depednet encrypted operators have to be used instead. This leads to the second rule of encrypted query optimiztaion, namely

Rule 2. *Use dependent encrypted operators for correlated predicates.*

The result is shown as the left query in Figure 11. It can be shown that the excessive access is reduced by a potentially large divisive factor, because of the depednence of input-output data introduced between operators. The final structure of the encrypted query tree looks very similar to the plaintext query structure.

Now a different question is: can optimization rules compose? In this case, the only data tables that can be delayed are T_1 and T_2. So Rule 1 can be applied afterwards to obtain the right query in the same figure.

6.5 Simulated Independent Joins For Unfiltered Joins

It turns out that dependent operators are not always better than independent ones. Suppose the plaintext query is changed to have unfiltered joins

$$T_1 \bowtie_{c_1=c_2} T_2 \bowtie_{d_2=d_3} T_3 \tag{92}$$

There are three possibilities for encrypted queries. First is to use independent joins, second to use depedent joins, and last to use *simulated* independent joins. Here simulation means to use dependent joins

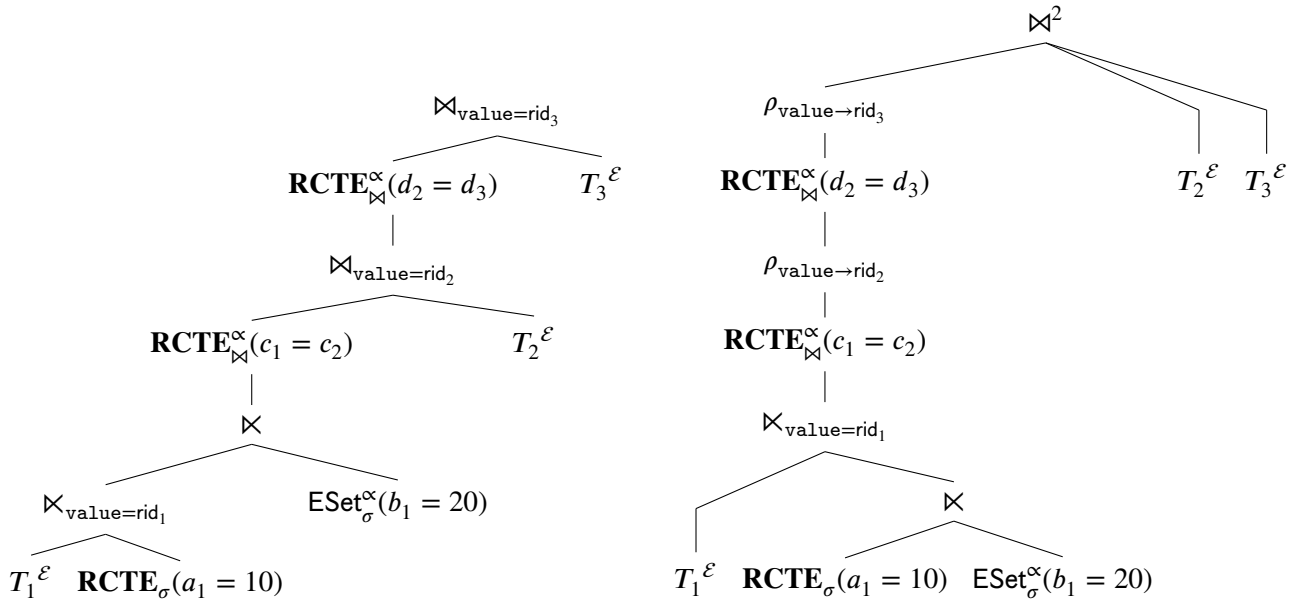

Figure 11: The encrypted queries using dependent encrypted operators under Rule 2. The right query tree is obtained by subsequently applying Rule 1.

to expfress independent joins. The first type is shown in Figure 12. The last two types are shown in Figure 13.

Perhaps reminiscent of the database literature, the two independent queries is akin to the "bushy join tree" and the right query the "left-deep join tree".

The first observation is that all three queries incur worst-case $O(T^3)$ time. Compared to plaintext query, all three encrypted queries match the typical worst-case time of two-way join algorithms, but none matches the worst-case optimal time. If a typical worst-case suboptimal two-way join algorithm is used, then the plaintext query incurs $O(T^3)$ time, and if a worst-case optimal join algorithm is used, then $O(T^{1.5})$. The improvement of the encrypted join to match the optimal worst case will be studied in the **dex-dom** scheme and so not yet the focus here.

Another observation is that the simulated independence is more efficient than the independence in that it reduces the "glue" joins from 5 to 3. So in the following only the simulated independence is compared against the dependence.

Now let the focus be on just the nature of dependence. Perhaps one intuition might be that the right query with dependent oeprators could be better because it has less "glue" joins than the left query. But the experimental results on *normalized* tables indicate that there is a significant difference in empirical efficiency that favors the simulated independent operators. This remains an open question to invetigate in this thesis in a cost model.

But a qualitative argument for the superiority of independent operators can be made as follows. One major factor in the query performance is the intermediate data size. The query with independent operators tend to have lower tree height. Generally speaking the deeper the join tree the larger intermediate data might potentially be.

Suppose the main focus is just on the input and output of the RCTEs in the query trees. The left query in the figure has two simulated independent RCTEs, so their input sizes are $O(T)$ each and output

sizes are $O(T^2)$ each[3]. But for the right query, the two RCTEs are "chained" together on a path, and so the input and output size of the second RCTE is actually $O(T^2)$ and $O(T^3)$. This argument needs to be gounded into a formal cost model for RCTEs and encrypted oeprators, and is part of the proposed work.

Rule 3. *Simulated independent joins are better than independent joins.*

Rule 4. *Use simulated independent joins for unfiltered joins.*

Now perhpas a more important question is that what happens if there is correlated filter predicates to the joins, but the filter does not introduce a noticeable cutdown to the intermediate data size as more joins are chained along the same path? Especially when the data under the joins have weak statitistical correlation, then it would not take more than a few joins for the filter to lose the effect on cutting down the filter. This can be shown easily with the probability theory. So the point is that the case of unfiltered joins is more applicable than it seems, because correlated filters can be ineffective due to the statistical properties of the underlying data, and so the filtered joins can be well approximated by the unfiltered joins.

Rule 5. *If the filtered joins can be approximated by the unfiltered joins, apply Rule 4.*

But the caveat is to exactly know when the filtered joins can be aprpoxaimted by the unfiltered joins. The knowledge would have to come from plaintext data distribution. It is certainly doable if the client keeps also the statistics about the plaintext data, but the question is whether applying this rule on a query would perhaps reveal more information about the plaintext query and the data. The information revealed is that the filtered attributes and the join attributes have weak statistical correlations. This may or may not constititue significance in particular real-world applciations, and it is important to study and know its implications to attacks.

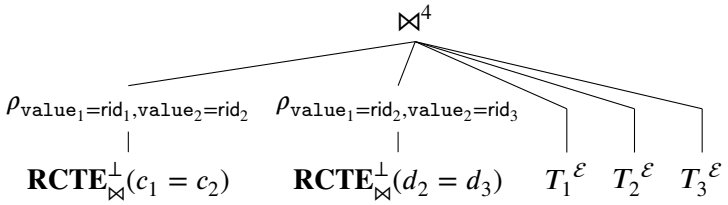

Figure 12: The encrypted query for plaintext query (92) with independent encrypted operators.

6.6 Many-to-Many Join Factorization

Consider even simpler example with just one join

$$T_1 \bowtie_{c_1=c_2} T_2$$

But this join is a *many-to-many* join.

The straightforward solution of using just one encrypted join is shown in Figure 14. However this solution would result in an encrypted

multi-map of worst-case quadratic size $O(T^2)$, whereas the plaintext query only requires $O(T)$ storage. In terms of efficiency, this solution also makes the query suboptimal in efficiency measured by

[3]Note that the join size of these two RCTEs is only $O(T^3)$ not $O(T^4)$ because the total join size is bounded by $O(T^3)$

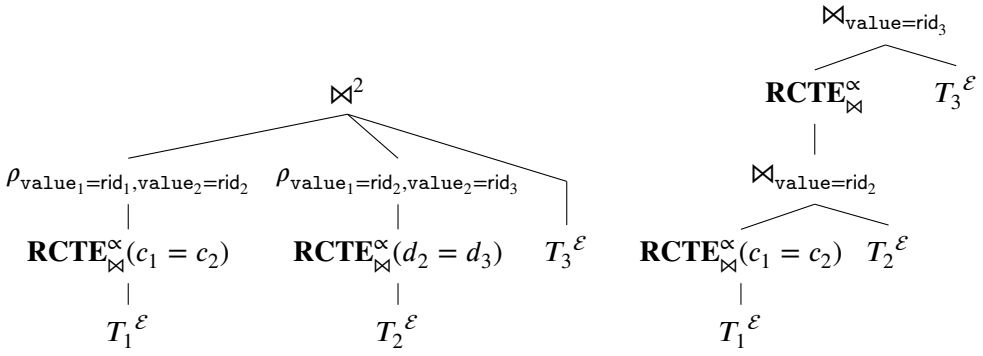

Figure 13: The encrypted queries for plaintext query (92). The left query uses *simulated* independent opeartors, and the right query uses dependent operators.

intermediate size. The RCTE has output size $O(T^2)$, and as a result the top join has input size $O(T^2)$. But a plaintext join would only requires $O(T)$ input size from each table. This is an instance where the space complexity is closely related to the time complexity.

So the hope is through reduce the space complexity, the time complexity can be reduced too. This is indeed the outcome of this optimization rule

Rule 6. *Factor the many-to-many join into two many-to-one joins, and only index these two joins in the encrypted data structure.*

As shown in Figure 15, to reduce this suboptimal join input size, the join is factorized into two many-to-one joins by means of normalization of the two operand tables into three tables, and then only during the setup time compute the encrypted multi-map entries for row ids in these two many-to-one joins. This can be done by extracting the third table that presents a primary key to which the foreign keys in the two original tables reference. Unlike many-to-many joins, a many-to-one join has bounded computational cost of the size of the largest table, or $O(\max(T_{\mathsf{left}}, T_{\mathsf{right}})$. Therefore the input sizes of the encrypted query reduces to $O(\max(T_1, T_3) + \max(T_3, T_4)) = O(T)$, assuming that $T_1 = T_4$ are the sizes of the foreign key table, or the "many" side of the join, and T_3 the size of the third primary key table, or the "one" side of the join. This optimziation rule matches the optimal worst-case bound for the input size for a many-to-many join.

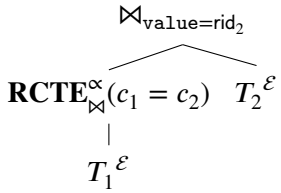

Figure 14: The many-to-many join query with one encrypted join.

6.7 Join Order Selection

Proposed work

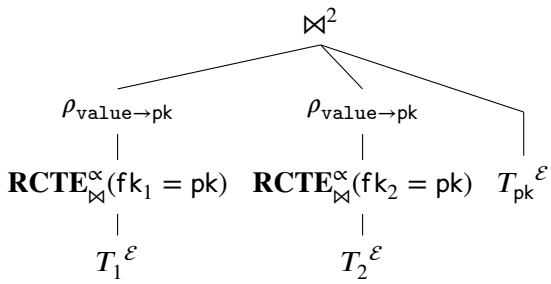

Figure 15: The factorized many-to-many join into two many-to-one joins.

When a query has multiple dependent joins, the order of which these joins are composed can have significant implication of the performance. However, determining the optimal join order among N joins is an NP-hard problem where the search space is N factorial [15, 18]. Moreoever, if the physical level access methods are also considered for each join, then the search space is even larger. So typicall heuristics based on data statistics are applied to simplify the search.

One crucial aspect in the join order selection for the encrypted query is that the selection should not compromise the privacy. This means that the heuristics are better based at most only on the leakage function of the scheme. For all the **dex** schemes, this means the table sizes.

On the other hand, encrypted join operators have the independent type and the dependent type. Therefore their differences in structure and efficiency need to be taken into count when constructing the join tree. A multi-way join can be represented as either a left-deep tree or a bushy tree. In general, a left-deep tree can only use dependnet joins, and a bush tree can use either dependent or independent joins depending on the tree strucutre. This creates a new dimension of consideration in join selection which traditionally only has to consider the join size.

This thesis focuses on extending existing join order selection problem into encrytped join operators of both independent and dependent types. There are well-established algorihtms that only take the table sizes into consideration, for example in [22, 25, 17].

7 Encrypted Relational Database

The previous sections on Encrypted SQL and Query Optimization have established the ingredients for an encrypted relational database that is conceived for the problem in the beginning of the thesis. In particular, Encrypted SQL presents how to use structured encryption to design efficient and secure encrypted relational operators that can compose complex, semantically-equivalant encrypted queries. Query Optimization discusses how to optimize encrypted queries.

Building on this knowledge so far, this section aims at explaining how to design an end-to-end encrypted relational database that solves the problem posted at the beginning of the thesis. The focus however is solely on the abstraction. System implementation is deferred to Section 9.

The abstract design of an encrypted relational database is presented in Figure 16.

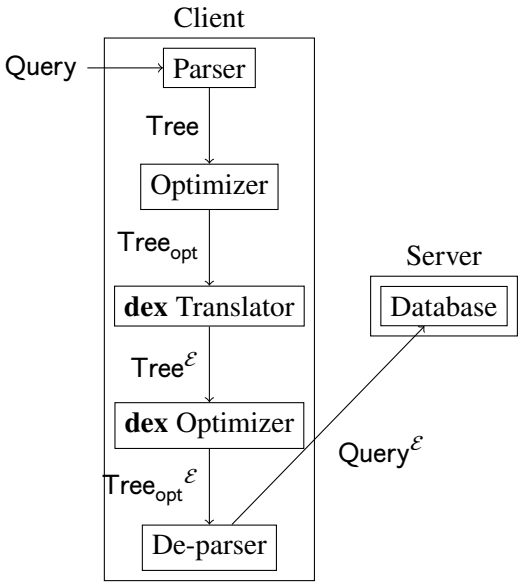

Figure 16: The encrypted relational database.

8 Schemes

The ideas and techniques introduced in the previous secitons on Encrypted SQL and Query Optimization are incorporated into different **dex** schemes with varying security and efficiency.

The **dex**-**spx** scheme is the closest to the **spx** as proposed by Kamara and Moataz. The encrypted multi-maps in **spx** are dirctly emulatd in **dex**-**spx**. The encrypted filters and joins are independent, and their dependnecies are simulated using plain database semi-joins. **dex**-**spx** corrects an error in the query algorithm in **spx**. Specifically, **spx** uses cross products to join together results from encrypted multi-maps, but inner joins and semi-joins were meant.

The **spx** specifies a form so-called Generalized Heuristic Normal Form, which in the database theory is the SPC algebra with one of its rewrite rules, the push-selection-through-joins rule [1, Figure 6.2]. Therefore all **dex** schemes directly work with the SPC algebra.

Another difference between the **spx** scheme and the **dex** schemes in terms of leakage is that **dex** schemes do not pad the data tables during the encryption, so the sizes of the data tables are part of the setup leakage.

Table 1 lists different techniques used in the schemes. Table 2 and Table 3 compare the relational language coverage. The performance of these schemes are listed in the Table 4. The setup leakage of each scheme is listed in Table 5. The query leakage for join in each scheme is listed in Table 6. The query leakage for conjunctive filter in each scheme is listed in Table 7. The query leakage for dependent filters and joins for each scheme is listed in Tabler 8. The quyery leakage for conjunction, disjunction and negation in each scheme is listed in Table 9.

Scheme	Indep. Filter	Indep. Join	Dep. Filter	Dep. Join	Normal Form	W.C. Optimal Join
dex-spx	✓	✓				
dex-cor	✓		✓	✓		
dex-nf	✓		✓	✓	✓	
dex-dom	✓		✓			✓

Table 1: The techniques used in **dex** schemes.

Scheme	Conjunction	Disjunction	Negation
dex-spx	native		
dex-cor	native	sim.	sim.
dex-nf	improved	sim.	sim.
dex-dom	native	sim.	sim.

Table 2: The predicates supported in **dex** schemes.

Scheme	Filter	Cross Join	Inner Join	Outer Join	Semi Join	Anti Join	Ins./Del./Up.
dex-spx	native	sim.	native				sim.
dex-cor	native	sim.	native	sim.	sim.	sim.	sim.
dex-nf	improved	sim.	improved	sim.	sim.	sim.	sim.
dex-dom	native	sim.	improved	sim.	sim.	sim.	sim.

Table 3: The operators supported in **dex** schemes.

Scheme	Join Space	Join Input	Join Time	Filter Space	Filter Time
dex-spx	$O(L^2 \cdot \mathsf{idx})$	$O(L^2)$	$O(L^2 \cdot \mathsf{ptr})$	$O(L \cdot \mathsf{idx})$	$O(L \cdot \mathsf{ptr})$
dex-cor	$O(L^2 \cdot \mathsf{idx})$	$O(L^2/\mathsf{sel})$	$O(L^2 \cdot \mathsf{ptr}/\mathsf{sel})$	$O(L \cdot \mathsf{idx})$	$O(L \cdot \mathsf{ptr})$
dex-nf	$O(L)^*$	$O(L)^*$	$O(L^2/\mathsf{sel})^*$	$O(L)$	$O(L)$
dex-dom	$O(L)$	$O(L)$	$O(L^{1.5}/\mathsf{sel})$	$O(L)$	$O(L)$

Table 4: Efficiency of the **dex** schemes. L is number of rows in a table. $\mathsf{ptr}, \mathsf{idx}, \mathsf{sel}$ are factors introduced by indirect access, hash indices and selectivity respectively. Join complexity is per two joinnable attributes. Filter complexity is per attribute. The **dex-nf** complexities marked with $*$ only apply to normal-form keyed attributes.

Scheme	Joins	Filters	Tables
dex-spx	$N_{\bowtie}$	N_σ	$n_{\mathsf{row}}, n_{\mathsf{col}}, n_{\mathsf{cell}}$
dex-cor	$N_{\bowtie}$	N_σ	above
dex-nf	$n_{\mathsf{pk}}, n_{\mathsf{fk}}$	n_σ	above
dex-dom	N_{Dom}	n_σ	above

Table 5: The setup leakage of **dex** schemes. N denotes the count statistics over the entire database. For example $N_{\bowtie}$ is the sum of result size of all join oeprators in the entire database. n denotes the row count over one table. For example n_{pk} is the row count for the primary key. N is the sum of n over all tables in the database.

Scheme	1 Join 1 Atom	1 Join M Atoms	W Joins
dex-spx	A	$\left(A_m\right)_1^M$	$\left(J_w\right)_1^W$
dex-cor	A	$A_1, \left(A_m \mid A_{m-1}\right)_2^M$	$J_1, \left(J_w \mid J_{w-1}\right)_2^W$
dex-nf	above	above	above
dex-dom	above	above	above

Table 6: The query leakage of **dex** scheme for conjunctive non-cross joins. A denotes the leakage of the encrypted result that satisfies the atom. J denotes the leakage of the join that includes all of its atoms. $A_2 \mid A_1$ denotes the leakage of the *conditional* encrypted result set of A_2 based on that of A_1.

Scheme	1 Filter 1 Atom	1 Filter M Atoms	W Filters
dex-spx	A	$\left\{A_m\right\}_1^M$	$\left(F_w\right)_1^W$
dex-cor	above	$A_1, \left(A_m \mid A_{m-1}\right)_2^M$	$F_1, \left(F_w \mid F_{w-1}\right)_2^W$
dex-nf	above	above	above
dex-dom	above	above	above

Table 7: The query leakage of **dex** schemes for conjunctive filters. A denotes the leakage of the encrypted result set that satisfies the atom. F denotes the leakage of the filter including all of its atoms. $A_2 \mid A_1$ denotes the leakage of the *conditional* encrypted result set of A_2 based on that of A_1.

Scheme	Filter before Join	Filter after Join
dex-spx	F, J	F, J
dex-cor	$J \mid F$	$F \mid J$
dex-nf	above	above
dex-dom	above	above

Table 8: The query leakage in **dex** schemes for correlated filters and joins. F and J are the leakage of the filter and join operator respectively. $F \mid J$ denotes the leakage of the filter is conditioned on that of the join.

Scheme	$Q \wedge A$	$Q \vee A$	$\neg A$
dex-spx	Q, A	unsupported	unsupported
dex-cor	$A \mid Q$	$Q, A, A \mid Q$	A
dex-nf	above	above	above
dex-dom	above	above	above

Table 9: The query leakage of the **dex** schemes for conjunction, disjunction and negation. A and Q denotes the atom and the subformula on the same table(s) of the operator, repsectively.

8.1 Overview

A **dex** scheme consists of a set of client functions that transform a relatioanl database to an encrypted database, and transoform a plaintext query to a ciphertext query

- **Setup**(RDB, λ): encrypts the relational database with emulated encrypted multi-maps under the security parameter.
  - **EncryptTables**(RDB, k): encrypts just the data tables in the relational database under the secret key.
- **ProcessQuery**(Query, k): translates the query under the secret key key, sends the translated query to the server and process the query result.
  - **TranslateQuery**(Query, k): translates the query under the secret key.
  - **TranslateFormula**(Formula, k, $\cdots$): translates the propositional logic formula associated with a relational operator. The argument list varies for different schemes.

A **dex** scheme specifies a set of encrypted data structures that are emulated as tables. So it may also consists of client functions that create these tables. These functions are calld during **Setup**.

- **CreateEmm**$_{\mathsf{op}}$(RDB, k)
- **CreateESet**$_{\mathsf{op}}$(RDB, k)

The server hosts a relational database without any extension or customization, and only executes standard SQL queries

Protocol 7. *Database server query execution*

- *Server: On input query Q:*
  1. *Execute Q*
  2. *Return query result* Res(Q) *to client*

For all schemes, these protocols are of the same form

Protocol 8. Setup(RDB, λ)

- *Client on input* (RDB, λ)
  1. *Derive secret key* $k \sim \{0,1\}^{\lambda}$
  2. *For each table in* RDB*, permute the rows in place.*
  3. *For each table in* RDB*, assign a row id to each i-th row such that* rid $\leftarrow i$
  4. *For each encrypted multi-map* $\mathsf{EMM}_{\mathsf{op}}$ *specified by the scheme, run* **CreateEmm**$_{\mathsf{op}}$.
  5. *For each encrypted set* $\mathsf{ESet}_{\mathsf{op}}$ *specified by the scheme, run* **CreateESet**$_{\mathsf{op}}$.

Protocol 9. EncyptTables(RDB, k)

- *Client on intput* (RDB, k)

1. *For each table* T *in* RDB
   (a) *Generate randomized table name and schema*

$$T^{\mathcal{E}} = \mathcal{F}(k, \mathbf{Name}(T)) \quad \textit{and} \quad \mathsf{Schema}(T^{\mathcal{E}}) = (\mathcal{F}(k, c) \mid \forall c \in \mathsf{Schema}(T)) \tag{93}$$

   (b) *Create encrypted table with randomized schema by sending query to server*

$$\texttt{Create Table} \quad T^{\mathcal{E}}(\mathsf{Schema}(T^{\mathcal{E}})) \tag{94}$$

   (c) *For each row* r *in the table* T
      i. *Create encrypted row* $r^{\mathcal{E}}$

$$r^{\mathcal{E}} = \{\mathcal{E}(k, r[c]) \mid \forall c \in \mathsf{Schema}(T)\} \tag{95}$$

      ii. *Insert encrypted row to the encrypted table by sending query to server*

$$\texttt{Insert Into} \quad T^{\mathcal{E}} \quad \texttt{Values} \quad r^{\mathcal{E}} \tag{96}$$

8.2 DEX-SPX

Protocol 10. $\mathbf{CreateEmm}^{\mathbf{spx}}_{\sigma}(\mathsf{RDB}, k)$

- *Client on input* (RDB, k)

1. *Create a table for storing encrypted multi-map* EMM_{σ} *with schema* (`label`, `value`) *by sending query to server*

$$\texttt{Create Table} \quad T^{\perp}_{\sigma}(\texttt{label}, \texttt{value}) \tag{97}$$

2. *Create a hash index on* $T^{\perp}_{\sigma}$.`label` *to ensure amortized constant time access by sending query to the server*

$$\texttt{Create Index On} \quad T^{\perp}_{\sigma} \quad \texttt{Using Hash} \quad (\texttt{label}) \tag{98}$$

3. *For each table* T *in* RDB*, for each column* c *in* T*, for each unique value* v *in* c
   (a) *Derive trapdoors* $\mathsf{trapd}_1, \mathsf{trapd}_2 = \mathcal{F}(k, \mathbf{Name}(c)||v||1), \mathcal{F}(k, \mathbf{Name}(c)||v||2)$
   (b) *For each* i*-th row that contains* v *and its row id* rid_i *in*

$$(\mathsf{rid})_{i=1}^{I} \leftarrow \pi_{\mathsf{rid}}\sigma_{c=v}T \tag{99}$$

      i. *Create label-value pair*

$$(l, v) = \left(\mathcal{F}(\mathsf{trapd}_1, i), \mathcal{E}(\mathsf{trapd}_2, v)\right) \tag{100}$$

      ii. *Insert new row to* $T^{\perp}_{\sigma}$ *by sending query to server*

$$\texttt{Insert Into} \quad T^{\perp}_{\sigma} \quad \texttt{Values} \quad (l, v) \tag{101}$$

Protocol 11. $\mathbf{CreateEmm}^{\mathbf{spx}}_{\bowtie}(\mathsf{RDB}, k)$

- *Client on input* (RDB, k)

1. *Create a table for storing encrypted multi-map* $\mathsf{EMM}_{\bowtie}$ *with schema* $(\texttt{label}, \texttt{value}_1, \texttt{value}_2)$ *by sending query to server*

$$\texttt{Create Table} \quad T_{\bowtie}^{\perp}(\texttt{label}, \texttt{value}_1, \texttt{value}_2) \tag{102}$$

2. *Create a hash index on* $T_{\bowtie}^{\perp}.\texttt{label}$ *to ensure amortized constant time access by sending query to the server*

$$\texttt{Create Index On} \quad T_{\bowtie}^{\perp} \quad \texttt{Using Hash} \quad (\texttt{label}) \tag{103}$$

3. *For each pair of columns that are joinable* $T_1.c_1, T_2.c_2$,
   (a) *Derive trapdoors* $\mathsf{trapd}_j = \mathcal{F}(k, \mathbf{Name}(c_1)||\mathbf{Name}(c_2)||j)$ *for* $j = 1$ *or* 2
   (b) *Compute the inner join with corresponding row ids*

$$(\mathsf{rid}_1, \mathsf{rid}_2)_{i=1}^{I} \leftarrow \sigma_{\mathsf{rid}_1, \mathsf{rid}_2} T_1 \bowtie_{c_1 = c_2} T_2 \tag{104}$$

   (c) *For each row id pairs* $\left(\mathsf{rid}_1, \mathsf{rid}_2\right)_i$ *in the above result and a counter* i
      i. *Compute the label and the associated value pair*

$$(l, v_1, v_2) = \left(\mathcal{F}(\mathsf{trapd}_1, i), \mathcal{E}(\mathsf{trapd}_2, \mathsf{rid}_{1,i}), \mathcal{E}(\mathsf{trapd}_2, \mathsf{rid}_{2,i})\right) \tag{105}$$

      ii. *Insert into encrypted multi-map the label and value pair by sending the query to the server*

$$\texttt{Insert Into} \quad T_{\bowtie}^{\perp} \quad \texttt{Values} \quad (l, v_1, v_2) \tag{106}$$

Protocol 12. $\mathbf{TranslateQuery}^{\mathbf{spx}}(\mathsf{Query}, k)$

- *Client on input* (Query, k)

1. *If the root of* Query *is a table name* T
   (a) *Reconstruct randomized table name* $T^{\mathcal{E}} = \mathcal{F}(k, T)$
   (b) *Return this subquery with* rid *renamed to* $T^{\mathcal{E}}.\mathsf{rid}$

$$\rho_{\mathsf{rid} \to \mathsf{rid}_i}(T^{\mathcal{E}}) \tag{107}$$

2. *Else if the root of* Query *is a project operator* $\pi(\mathsf{Subquery})$
   (a) *Reconstruct randomized projected schema*

$$\mathsf{Schema}_{\mathcal{F}} \leftarrow (\mathcal{F}(k, c) \mid \forall c \in \pi) \tag{108}$$

   (b) *Recursively translate the subquery tree*

$$\mathsf{Subquery}' \leftarrow \mathbf{TranslateQuery}^{\mathbf{spx}}(\mathsf{Subquery}, k) \tag{109}$$

   (c) *Return a query tree with randomized projected schema*

$$\pi_{\mathsf{Schema}_{\mathcal{F}}} \mathsf{Subquery}' \tag{110}$$

3. *Else if the root of* Query *is a filter operator* $\sigma_{\mathsf{Formula}}(T)$

(a) *Recursively translate the subquery tree*

$$\mathsf{Subquery}' \leftarrow \textbf{TranslateQuery}^{\textbf{spx}}(\mathsf{Subquery}, k) \tag{111}$$

(b) *Translate the formula as*

$$\mathsf{Subquery}_{\sigma} \leftarrow \textbf{TranlsateFormula}^{\textbf{spx}}(\mathsf{Formula}, k) \tag{112}$$

(c) *Reconstruct the randomized table name from* T *as* $T^{\mathcal{E}} = \mathcal{F}(k, T)$

(d) *Return a query tree that left semi-joins the two subquery trees*

$$\mathsf{Subquery}' \ltimes_{T^{\mathcal{E}}.\mathtt{rid}=\mathtt{value}} \mathsf{Subquery}_{\sigma} \tag{113}$$

4. *Else if the root of* Query *is a corss join operator*

$$\mathsf{Subquery}_1 \times \mathsf{Subquery}_2 \tag{114}$$

(a) *Recursively translate the left and right sbuquery trees*

$$\mathsf{Subquery}'_j \leftarrow \textbf{TranslateQuery}^{\textbf{spx}}(\mathsf{Subquery}_j, k) \quad \textit{for} \quad j = 1, 2 \tag{115}$$

(b) *Return this query that cross join the left and right subquery trees*

$$\mathsf{Subquery}'_1 \times \mathsf{Subquery}_2 \tag{116}$$

5. *Else if the root of* Query *is an inner join operator*

$$\mathsf{Subquery}_1 \bowtie_{\mathsf{Formula}} \mathsf{Subquery}_2 \tag{117}$$

(a) *Recursively translate the left and right subquery trees*

$$\mathsf{Subquery}'_j \leftarrow \textbf{TranslateQuery}^{\textbf{spx}}(\mathsf{Subquery}_j, k) \quad \textit{for} \quad j = 1, 2 \tag{118}$$

(b) *Translate the formula as*

$$\mathsf{Subquery}_{\bowtie} \leftarrow \textbf{TranslateFormula}^{\textbf{spx}}(\mathsf{Formula}, k) \tag{119}$$

(c) *Return a query tree that natural joins the three subquery trees*

$$\mathsf{Subquery}'_1 \bowtie \mathsf{Subquery}_{\bowtie} \bowtie \mathsf{Subquery}'_2 \tag{120}$$

Protocol 13. $\textbf{TranslateFormula}^{\textbf{spx}}(\mathsf{Formula}, \mathbf{k})$

- *Client on input* $(\mathsf{Formula}, k)$

1. *If the root of* Formula *is a filter predicate* $T.c = v$ *for column* c *and value* v

(a) *Reconstruct the randomized table name from* T *as* $T^{\mathcal{E}} = \mathcal{F}(k, T)$

(b) *Reconstruct the trapdoors*

$$\mathsf{trapd}_j = \mathcal{F}(k, \mathsf{Name}(c)||v||j) \quad \textit{for} \quad j = 1, 2 \tag{121}$$

(c) *Return the recursive common table expression that operates on the* EMM *for filter*

Query 5 (**RCTE**$_\sigma$)

$$\begin{aligned}&\texttt{With Recursive}\ \ \mathsf{View}\ \ \texttt{As}\\&\quad \pi_{\texttt{value},i}\,\sigma_{\texttt{label}=\mathcal{F}(\mathsf{trapd}_1,i)}\big(T_\sigma^{\perp}\times\rho_i\{1\}\big)\\&\quad \texttt{Union All}\\&\quad \pi_{\texttt{value},i}\Big(T_\sigma^{\perp}\ltimes_{\texttt{label}=\mathcal{F}(\mathsf{trapd}_1,i)}\pi_{i+1\to i}(\mathsf{View})\Big)\\&\pi_{D(\mathsf{trapd}_2,\texttt{value})\to T^{\mathcal{E}}.\mathsf{rid}}\mathsf{View}\end{aligned}\tag{122}$$

2. *Else if the root of* Formula *is an equi-join predicate* $T_1.c_1 = T_2.c_2$
   (a) *Reconstruct the randomized table names as* ${T_j}^{\mathcal{E}} = \mathcal{F}(k, T_j)$ *for* $j = 1, 2$
   (b) *Reconstrct the trapdoors*

$$\mathsf{trapd}_j = \mathcal{F}(k, \mathsf{Name}(c_1)||\mathsf{Name}(c_2)||j) \quad for \quad j = 1, 2 \tag{123}$$

   (c) *Return the recursive common table expression that operates on the* EMM *for equi-join*

Query 6 (**RCTE**$_\bowtie$)

$$\begin{aligned}&\texttt{With Recursive}\ \ \mathsf{View}\ \ \texttt{As}\\&\quad \pi_{\texttt{value}_1,\texttt{value}_2,i}\,\sigma_{\texttt{label}=\mathcal{F}(\mathsf{trapd}_1,i)}\big(T_\sigma^{\perp}\times\rho_i\{1\}\big)\\&\quad \texttt{Union All}\\&\quad \pi_{\texttt{value}_1,\texttt{value}_2,i}\Big(T_\sigma^{\perp}\ltimes_{\texttt{label}=\mathcal{F}(\mathsf{trapd}_1,i)}\pi_{i+1\to i}(\mathsf{View})\Big)\\&\pi_{D(\mathsf{trapd}_2,\texttt{value}_1)\to {T_1}^{\mathcal{E}}.\mathsf{rid},D(\mathsf{trapd}_2,\texttt{value}_2)\to {T_2}^{\mathcal{E}}.\mathsf{rid}}\mathsf{View}\end{aligned}\tag{124}$$

3. *Else if the root of* Formula *is a conjunction*

$$\mathsf{Subformula}_1 \wedge \mathsf{Subformula}_2 \tag{125}$$

   (a) *Recursively translate the left and right operand*

$$\mathsf{Subquery}_j \leftarrow \textbf{TranslateFormula}(\mathsf{Subquery}_j, k) \quad for \quad j = 1, 2 \tag{126}$$

   (b) *Return the query that natural semi-joins both subqueries*

$$\mathsf{Subquery}_1 \ltimes \mathsf{Subquery}_2 \tag{127}$$

8.3 DEX-COR

Protocol 14. $\textbf{CreateESet}^{\textbf{cor}}_{\wedge}$

- *Client on input* (RDB, λ)

1. *Create a table to store the encrypted set by sending this query to the server*

$$\texttt{Create Table}\ \ T_\sigma^{\alpha}(\texttt{value}) \tag{128}$$

2. *For each table* T *in* RDB*, for each column* c

(a) *For each value v in c and its row id* rid

i. *Derive the trapdoor*

$$\mathsf{trapd}_{c,v} = \mathcal{F}(k, \mathsf{Name}(c)||v) \tag{129}$$

ii. *Compute the encrypted set value*

$$w = \mathcal{F}(\mathsf{trapd}_{c,v}, \mathsf{rid}) \tag{130}$$

iii. *Insert into the encrypted set table by sending this query to the server*

$$\texttt{Insert Into} \quad T_{\sigma}^{\propto} \quad \texttt{Values} \quad (w) \tag{131}$$

Protocol 15. CreateEmm$^{\mathbf{cor}}_{\bowtie}$

- *Client on input* (RDB, k)

1. *Create a table for storing encrypted multi-map* $\mathsf{EMM}_{\bowtie}$ *with schema* $(\texttt{label}, \texttt{value}_1, \texttt{value}_2)$ *by sending query to server*

$$\texttt{Create Table} \quad T_{\bowtie}^{\propto}(\texttt{label}, \texttt{value}_1, \texttt{value}_2) \tag{132}$$

2. *Create a hash index on* $T_{\bowtie}^{\propto}.\texttt{label}$ *to ensure amortized constant time access by sending query to the server*

$$\texttt{Create Index On} \quad T_{\bowtie}^{\propto} \quad \texttt{Using Hash} \quad (\texttt{label}) \tag{133}$$

3. *For each pair of columns that are joinable* $T_1.c_1, T_2.c_2$,

(a) *Derive the master trapdoor*

$$\mathsf{trapd}_{\bowtie} = \mathcal{F}(k, \mathsf{Name}(c_1)||\mathsf{Name}(c_2)) \tag{134}$$

(b) *Compute the inner join with corresponding row ids, grouped by the* rid_1 *and collect all the corresponding* rid_2 *in a set*

$$\left(\mathsf{rid}_1, \left\{\mathsf{rid}_2\right\}\right)_{r=1}^{R} \leftarrow {}_{\mathsf{rid}_1}G_{\mathsf{collect}(\mathsf{rid}_2)}\sigma_{\mathsf{rid}_1,\mathsf{rid}_2}T_1 \bowtie_{c_1=c_2} T_2 \tag{135}$$

(c) *For each* $\left(\mathsf{rid}_1, \left\{\mathsf{rid}_2\right\}_{i=1}^{I_r}\right)_r$ *in the above result*

i. *Derive the specific trapdoors based on the master trapdoor* $\mathsf{trapd}_{\bowtie}$

$$\mathsf{trapd}_j = \mathcal{F}(\mathsf{trapd}_{\bowtie}, \mathsf{rid}_1||j) \quad \textit{for} \quad j = 1, 2 \tag{136}$$

ii. *For each i-th* $\mathsf{rid}_{2,i}$ *in* $\left\{\mathsf{rid}_2\right\}_{i=1}^{I_r}$

A. *Compute the label and the associated value pair*

$$(l, v) = \left(\mathcal{F}(\mathsf{trapd}_1, i), \mathcal{E}(\mathsf{trapd}_2, \mathsf{rid}_{2,i})\right)) \tag{137}$$

B. *Insert into encrypted multi-map the label and value pair by sending the query to the server*

$$\texttt{Insert Into} \quad T_{\bowtie}^{\propto} \quad \texttt{Values} \quad (l, v) \tag{138}$$

Protocol 16. TranslateQuery$^{\mathbf{cor}}$ (RDB, k)

- *Client on input* (RDB, k)

  1-4. Same as **TranslateQuery**$^{\textbf{spx}}$ *1-4.*

  5. Else if the root of Query *is a join operator*

$$\mathsf{Subquery}_1 \bowtie_{\mathsf{Formula}} \mathsf{Subquery}_2 \tag{139}$$

  (a) Recursively translate the left and right subquery trees

$$\mathsf{Subquery}'_j \leftarrow \textbf{TranslateQuery}^{\textbf{cor}}(\mathsf{Subquery}_j, k) \quad \textit{for} \quad j = 1, 2 \tag{140}$$

  (b) Translate the formula based on $\mathsf{Subquery}'_1$

$$\mathsf{Subquery}^{\bowtie}_{1'} \leftarrow \textbf{TranslateFormula}^{\textbf{cor}}(\mathsf{Formula}, k, \mathsf{Subquery}'_1) \tag{141}$$

  (c) Return a query tree that inner joins the two subqueries

$$\mathsf{Subquery}^{\bowtie}_{1'} \bowtie \mathsf{Subquery}'_2 \tag{142}$$

Protocol 17. **TranslateFormula**$^{\textbf{cor}}$(Formula, k, Subquery)

- *Client on input* (Formula, k, Subquery)

  1. If the root of Formula *is a filter predicate* $T.c = v$ *for column c and value v*

  (a) Reconstruct the randomized table name as $T^{\mathcal{E}} = \mathcal{F}(k, T)$

  (b) If Subquery *is empty, this is the first atom in the* Formula

  i. Same as Step 1.(a)-(b) in **TranslateFormula**$^{\textbf{spx}}$

  (c) Else Subquery *is nonempty, this is non-first atom in the* Formula

  i. Reconstruct trapdoor

$$\mathsf{trapd}_{c,v} = \mathcal{F}(k, \mathsf{Name}(c)||v) \tag{143}$$

  ii. Return this query

  Query 7

$$\mathsf{Subquery} \ltimes_{\mathcal{F}(\mathsf{trapd}_{c,v}, T^{\mathcal{E}}.\mathsf{rid})=\texttt{value}} T^{\propto}_{\sigma} \tag{144}$$

  2. If the root of Formula *is an equi-join predicate* $T_1.c_1 = T_2.c_2$

  (a) Reconstruct the randomized table names ${T_j}^{\mathcal{E}} = \mathcal{F}(k, T)$ *for* $j = 1, 2$

  (b) Reconstruct the trapdoor

$$\mathsf{trapd}_{\bowtie} = \mathcal{F}(k, \mathsf{Name}(c_1)||\mathsf{Name}(c_2)) \tag{145}$$

  (c) Return the recursive common table expression that operates on the Subquery *as input to the* EMM

Query 8

$$\begin{aligned}&\texttt{With Recusive View As}\\&\quad \pi_{\mathsf{Subquery}.*,T_1{}^{\mathcal{E}}.\mathsf{rid},\texttt{value},i} T_{\sigma}^{\perp} \bowtie_{\texttt{label}=\mathcal{F}(\mathcal{F}(\mathsf{trapd}_{\bowtie},T_1{}^{\mathcal{E}}.\mathsf{rid}||1),i)} \left(\mathsf{Subquery} \times \rho_i\{1\}\right)\\&\quad \texttt{Union All}\\&\quad \pi_{\mathsf{Subquery}.*,T_1{}^{\mathcal{E}}.\mathsf{rid},\texttt{value},i} \left(T_{\sigma}^{\perp} \ltimes_{\texttt{label}=\mathcal{F}(\mathcal{F}(\mathsf{trapd}_{\bowtie},T_1{}^{\mathcal{E}}.\mathsf{rid}||1),i)} \pi_{i+1\to i}(\mathsf{View})\right)\\&\pi_{\mathsf{Subquery}.*,\mathcal{D}(\mathcal{F}(\mathsf{trapd}_{\bowtie},T_1{}^{\mathcal{E}}.\mathsf{rid}),\texttt{value})\to T_2{}^{\mathcal{E}}.\mathsf{rid}} \mathsf{View}\end{aligned} \tag{146}$$

3. *Else if the root of* Formula *is a conjunction*

$$\mathsf{Subformula}_1 \wedge \mathsf{Subformula}_2 \tag{147}$$

(a) *Recursively translate the left and right operand*

$$\mathsf{Subquery}_j \leftarrow \textbf{TranslateFormula}^{\textbf{cor}}(\mathsf{Subquery}_j, k, \mathsf{Subquery}) \quad \textit{for} \quad j = 1,2 \tag{148}$$

(b) *Return the query that natural semi-joins both subqueries*

$$\mathsf{Subquery}_1 \ltimes \mathsf{Subquery}_2 \tag{149}$$

8.4 DEX-NF

Proposed work

8.5 DEX-DOM

Proposed work

9 System Implementation

All the **dex** schemes are implemented in a system $\Sigma_{\mathbf{dex}}$, open-source at [26], and evaluated by the TPC-H benchmark [8]. A comprehensive study requires the schemes to run against different standard relational database systems such as Postgres [12], MySQL [7] and SparkSQL [2]. Therefore it is important that the system can interface with different relational databases. To this end, $\Sigma_{\mathbf{dex}}$ takes SparkSQL [2]'s alegebra core, extends it with a cryptographic module and several **dex**-specific modules that implement the following features

- Interface with any database endpoint as data store using JDBC [14]
- Parallel relational data encryption to database endpoint
- Encrypted SQL algebra
- Query translation between a plaintext query and an encrypted query
- Encrypted SQL optimization rules
- Secret key management on the trusted client
- Automatically purging of query logs and cached data on the database endpoint

As a result, $\Sigma_{\mathbf{dex}}$ can plug-and-play on any SQL relational databases and big data systems alike.

10 Evaluation

Proposed work

A through evaluation of **dex** should include empirical evidence for

- Efficiency and storage size differences in each **dex** scheme in Section 8
- Effectiveness of query optimization rules in Section 6
- Plug-and-play on any standard relational database and big data system
- Substantial SQL converage
- Substantial data model complexity
- Realistic workload

Currently, the relational database is chosen as PostgreSQL [12].

10.1 TPC-H

The TPC-H Benchmark [8] is well-established standard benchmark for data warehouse workload. The experiment is run on Services [23]. The instance type is chosen by consulting the hardware setup in Chiba and Onodera [6]. Chiba and Onodera use dedicated server of a memory size equal to 10 times over the TPC-H data size. So the Amazon instance type is chosen to match this ratio as closely as possible. Note that due to encryption the encrypted data size is larger. So this differential accounts for the choice of larger instance type to run the $\Sigma_{\mathbf{dex}}$. This means there are two possible comparions, one against plaintext database running on the *same* larger instance, and the other against plaintext database running on smaller instance that is appropriate to its size.

10.1.1 Results for the dex-cor Scheme

The time elapses for **dex-cor** scheme for the various queries are summarized in Table 10. Only the encrypted portion of each query are measured. The encrypted portion of a query is a set of subqueries that consist of only encrypted operators, and so are executed on the server.

The storage size for the encrypted tables and emulated encrypted data structures are summarized in Table 11.

11 Conclusion

dex-cor	mean(ms)(m44xlarge)	rel.err.	slowdown vs. Postgres	slowdown vs. Postgres(t22xlarge)
q1	2149.5	1.38%	1.4	1.0
q10	217837.6	0.16%	115.0	32.7
q11	993.3	2.15%	13.8	5.4
q12	38976.8	0.25%	32.7	26.9
q13	61460.2	0.27%	84.0	64.9
q14	110169.4	0.15%	40.5	29.7
q15	100299.6	0.29%	41.0	33.3
q16	478.2	1.99%	3.0	2.6
q17	437.2	2.29%	6.0	4.0
q18	280982.8	0.28%	66.6	50.0
q19	31976.2	0.36%	373.1	324.3
q2	2445.4	0.52%	15.8	12.3
q20	27546.1	0.51%	284.6	35.7
q21	447845.6	0.22%	441.9	354.6
q22	55829.3	0.36%	134.1	108.7
q3	40152.1	0.38%	22.8	17.2
q4	119831.3	0.20%	30.6	23.1
q5	4645337.0	0.62%	2231.8	1875.5
q7	47910.1	0.30%	131.8	81.6
q8	117817.6	0.44%	57.4	38.8
q9a	464302.6	0.20%	10.8	12.3
q9b	356243.1	0.32%	8.3	9.4

Table 10: TPC-H Benchmark elapse time for query 1 to 20, coded with prefix q and query number. The times are measured only on the encrypted portion of each query, including decryption of the results. The queries and benchmark implementation are published in [26]. The runtimes are measured on Amazon AWS instance type m44xlarge and t22xlarge. TPC-H scale factor is 1.

tale name	row est.	attrs	page est.	total(bytes)	index(bytes)	table(bytes)
region	5	4(3)	1	32 k (64 k)	16 k(48 k)	8192
orders	$1.5e+06$	10(9)	71484(26405)	604 m(603 m)	45 m(396 m)	559 m(206 m)
supplier	10000	8(7)	447(226)	3936 k(4680 k)	328 k(2840 k)	3600 k(1832 k)
customer	150000	9(8)	7620(3758)	64 m(75 m)	4640 k(46 m)	60 m (29m)
partsupp	800000	7(5)	35580(18242)	302 m(337 m)	24 m(194 m)	278 m (143 m)
nation	25	5(4)	1	32 k(80 k)	16 k(64 k)	8192
lineitem	$6.00139e+06$	19(16)	500251(117594)	4090 m(3338 m)	181 m(2419 m)	3909 m(919 m)
part	200000	10(9)	9656(3832)	82 m(85 m)	6184 k(55 m)	75 m(30 m)
$T_{\sigma}^{\perp}$	$8.74646e+07$	2	1079906	13 G	4926 m	8439 m
$T_{\bowtie}^{\propto}$	$5.45298e+07$	2	673291	8332 m	3071 m	5261 m

Table 11: TPCH benchmark **dex-cor** versus plaintext Postgres storage size. The plaintext Postgres storage size is shown in parenthesis. TPC-H scale factor is 1.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Vol. 8. Addison-Wesley Reading, 1995.

[2] Michael Armbrust et al. "Spark sql: Relational data processing in spark". In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM. 2015, pp. 1383–1394.

[3] David Cash et al. "Dynamic searchable encryption in very-large databases: data structures and implementation." In: *NDSS*. Vol. 14. Citeseer. 2014, pp. 23–26.

[4] David Cash et al. "Highly-scalable searchable symmetric encryption with support for boolean queries". In: *Annual Cryptology Conference*. Springer. 2013, pp. 353–373.

[5] David Cash et al. "Leakage-abuse attacks against searchable encryption". In: *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. ACM. 2015, pp. 668–679.

[6] Tatsuhiro Chiba and Tamiya Onodera. "Workload characterization and optimization of tpc-h queries on apache spark". In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2016, pp. 112–121.

[7] Oracle Corporation. *MySQL*. `https://www.mysql.com/`. 2019.

[8] Transaction Processing Performance Council. "TPC-H benchmark specification". In: *Published at http://www. tcp. org/hspec. html* 21 (2008), pp. 592–603.

[9] Alexander Degitz, Jens Köhler, and Hannes Hartenstein. "Access Pattern Confidentiality-Preserving Relational Databases: Deployment Concept and Efficiency Evaluation." In: *EDBT/ICDT Workshops*. 2016.

[10] Andrew Eisenberg and Jim Melton. "SQL: 1999, formerly known as SQL3". In: *ACM Sigmod record* 28.1 (1999), pp. 131–138.

[11] Sky Faber et al. "Rich queries on encrypted data: Beyond exact matches". In: *European Symposium on Research in Computer Security*. Springer. 2015, pp. 123–145.

[12] The PostgreSQL Global Development Group. *PostgreSQL*. `https://www.postgresql.org/`. 2019.

[13] Hakan Hacigümüş et al. "Executing SQL over encrypted data in the database-service-provider model". In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM. 2002, pp. 216–227.

[14] Graham Hamilton and Rick Cattell. "Jdbc: A java sql api". In: *Sun Microsystems* 1 (1996), p. 997.

[15] Toshihide Ibaraki and Tiko Kameda. "On the optimal nesting order for computing n-relational joins". In: *ACM Transactions on Database Systems (TODS)* 9.3 (1984), pp. 482–502.

[16] Seny Kamara and Tarik Moataz. "SQL on structurally-encrypted databases". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2018, pp. 149–180.

[17] Guido Moerkotte and Thomas Neumann. "Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products". In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment. 2006, pp. 930–941.

[18] Guido Moerkotte and Wolfgang Scheufele. “Constructing optimal bushy processing trees for join queries is NP-hard”. In: *Technical reports* 96 (1996).

[19] Muhammad Naveed, Seny Kamara, and Charles V Wright. “Inference attacks on property-preserving encrypted databases”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 644–655.

[20] Hung Q Ngo et al. “Worst-case optimal join algorithms”. In: *Journal of the ACM (JACM)* 65.3 (2018), p. 16.

[21] Raluca Ada Popa et al. “CryptDB: protecting confidentiality with encrypted query processing”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 85–100.

[22] P Griffiths Selinger et al. “Access path selection in a relational database management system”. In: *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM. 1979, pp. 23–34.

[23] Amazon Web Services. *Amazon EC2*. `https://aws.amazon.com/ec2/`. 2019.

[24] Benjamin Hong Meng Tan et al. “Efficient Private Comparison Queries over Encrypted Databases using Fully Homomorphic Encryption with Finite Fields.” In: *IACR Cryptology ePrint Archive* 2019 (2019), p. 332.

[25] Bennet Vance and David Maier. “Rapid bushy join-order optimization with cartesian products”. In: *ACM SIGMOD Record*. Vol. 25. 2. ACM. 1996, pp. 35–46.

[26] Zheguang Zhao. *encrypted-spark*. `https://github.com/zheguang/encrypted-spark/tree/dex`. 2019.