Building Structurally Encrypted Relational Databases

Brown University

Zheguang Zhao

31st August, 2020

Abstract

End-to-end encrypted relational database management systems are the "holy grail" of database security and have been studied by the research community for the last 20 years. During this time, several systems that handle some subset of SQL over encrypted data have been proposed, including CryptDB (Popa et al., *SOSP '11*), ESPADA (Cash et al., *CRYPTO '13*), Blind Seer (Pappas et al., *IEEE S&P '14*) and Stealth (Ishai et al., *CT-RSA '16*).

CryptDB is based on property-preserving encryption (PPE) and has been shown to leak a considerable amount of information even in the snapshot model, which is the weakest adversarial model in this setting. And while ESPADA, Blind Seer and Stealth achieve much better leakage profiles, they suffer from two main limitations: (1) they cannot handle SQL queries that include join or project operations; and (2) they are not legacy-friendly which means that, unlike CryptDB and other PPE-based systems, they require a custom DBMS.

We design and build a new encrypted database management system called KafeDB that addresses all these limitations. KafeDB is based on structured encryption (STE) and, as such, achieves a leakage profile comparable to the ESPADA, Blind Seer and Stealth systems. KafeDB, however, handles a non-trivial subset of SQL which includes queries with joins and projections. In addition, KafeDB is *legacy-friendly*, meaning that it can be deployed on top of *any* relational database management system.

Contents

Chapter 1

Overview

1.1 Backgrounds

Data is being produced, collected and analyzed at unprecedented speed, volume and variety. For all the benefits of "big data", however, the constant occurrences of data breaches have raised serious concerns about the privacy and security of all the data that is being collected and managed, especially when data is sensitive like electronic health record or financial records. For example, in 2017, Yahoo! email breach affected 3 billion users. In 2016, the Democratic National Committee was hacked, and the documents leaked from the hack affected the 2016 U.S. Presidential election. On the other hand, surveillance programs have grown more pervasive and global than ever before, as revealed by Edward Snowden's leakage of classified information from National Security Agency in 2013. These trends all point at the fact that our data is not properly secured.

End-to-end encryption. While systems sometimes encrypt data in transit and at rest, data is decrypted and remains unencrypted when it is in use. An alternative way of deploying cryptography is *end-to-end* encryption. In this approach, the data is encrypted by the user before it even leaves its device. End-to-end encrypted systems and services provide much stronger security and privacy guarantees than the current generation of systems. The main challenge in building such systems, however, is that end-to-end encryption breaks many of the applications and services we rely on, including cloud computing, analytics, spam filtering, database queries and search. The area of *encrypted systems* aims to address the challenges posed by end-to-end encryption by producing practical systems that can operate on end-to-end encrypted data.

Encrypted databases. A key problem in this area is the problem of designing end-to-end *encrypted databases* (EDB); which are practical database management systems (DBMS) that operate on end-to-end encrypted databases. Roughly speaking, there are two kinds of databases: relational, which store data as tables and are queried using SQL; and non-relational (i.e., NoSQL), which do not store data as tables and are usually queried with lower-level query operations. Relational DBMSs are the most widely used and include products from major companies like Oracle, IBM, SAP and Microsoft.

PPE-based EDBs. The problem of relational EDBs is one of the "holy grails" of database security. It was first explicitly considered by Hacigümüs, Iyer, Li and Mehrotra [35] who described a quantization-based approach which leaks the range within which an item falls. In [47], Popa, Redfield, Zeldovich and Balakrishnan described a system called CryptDB that supports a non-trivial subset of SQL without quantization. CryptDB achieves this in part by making use of property-preserving encryption (PPE) schemes like deterministic and order-revealing (ORE) encryption [2, 12, 13], which reveal equality and order, respectively. Because CryptDB's PPE-based approach was efficient and legacy-friendly, it was quickly adopted by academic systems like Cipherbase [7] and commercial systems like SEEED [49] and Microsoft's SQL Server Always Encrypted [24]. While the security of PPE *primitives* had been formally studied by the cryptography community [3, 12, 15, 13, 14], their application to database systems was never formally analyzed or subject to any cryptanalytic evaluation (e.g., the first leakage analysis of the CryptDB system appeared in 2018 [39]). As a result, in 2015, Naveed, Kamara and Wright described practical data-recovery attacks against PPE-based EDBs in the snapshot model—which is the weakest possible adversarial model in this setting. In the setting of electronic medical records, for example, sensitive attributes of up to 99% of patients could be recovered with a snapshot attack (i.e., without even seeing any queries). Since then, several follow-up works have improved on the original NKW attacks [34, 27].

Given the high level of interest in EDBs from Academia, Industry and Government and the weaknesses of the quantization- and PPE-based solutions, the design of practical and cryptographically-analyzed relational EDBs remained an important open problem.

STE-based EDBs. There are several ways to design relational EDBs but each solution achieves some tradeoff between efficiency, query expressiveness and leakage. General-purpose primitives like fully-homomorphic encryption (FHE) and secure multi-party computation (MPC) can be used to support all of SQL without any leakage but at the cost of exceedingly slow query execution due to linear-time asymptotic complexity and very large constants. Oblivious RAM (ORAM) could also be used to handle all of SQL with very little leakage (i.e., mostly volume leakage) but at the cost of a poly-logarithmic multiplicative overhead in the size of the database.

More practical solutions can be achieved using structured encryption (STE) [21] which is a generalization of indexed-based searchable symmetric encryption (SSE) [50, 31, 20, 26]. STE schemes encrypt data structures in such a way that they can only be queried using a token that is derived from a query and the secret key. One way to use STE/SSE to design relational EDBs is to index each database column using an encrypted multi-map (EMM) [26]. This is, roughly speaking, the approach taken by systems such as ESPADA [18, 17, 37, 28], Blind Seer [45, 29] and Stealth [36].[1] We refer to this as *column indexing* and this leads to

[1]These systems are more complex than described here. They work in a multi-user setting and provide additional security properties that we do not consider in this work.

systems that can handle SQL queries of the form

SELECT * FROM table **WHERE** $\mathsf{att} = a$,

where a is a constant. When columns are indexed with more complex EMMs (e.g., that can also handle range queries) then column indexing yields systems that can handle queries of the form

SELECT * FROM T
WHERE $\mathsf{att}_1 = a$ **AND** $\mathsf{att}_2 \leq b$,

While column indexing results in fast query execution (i.e., sub-linear running time), systems based on this approach cannot handle SQL queries with project or join operations. This is a non-trivial limitation since joins are extremely common (e.g., [38] reports that 62.1% of Uber queries include joins). This was addressed by Kamara and Moataz who proposed the first STE-based solution to handle a non-trivial fraction of SQL [39]; more specifically, queries of the form

SELECT attributes **FROM** tables
WHERE $\mathsf{att}_1 = a$ **AND** $\mathsf{att}_2 = \mathsf{att}_3$,

which include projects and joins but not ranges. This scheme, called SPX, is asymptotically optimal for a subset of the queries above and (provably) leaks a lot less than known PPE-based solutions like CryptDB. In this work, we propose an extension of this scheme, called OPX, that handles any query of the form above in asymptotically optimal time. We note, however, that both SPX and OPX are only cryptographic constructions and not systems like ESPADA, Blind Seer and Stealth. A third approach to designing relational EDBs is to use trusted hardware like secure coprocessors or Intel SGX. Several systems, most notably TrustedDB [10] and StealthDB [54] take this direction. Though our system could leverage trusted hardware by running our client proxy in an enclave,[2] we do not investigate this direction given the security concerns around SGX.

Legacy-friendliness. The main advantages of PPE-based EDBs compared to STE-based EDBs are that the former are: (1) much easier to implement; and (2) *legacy-friendly* in the sense that the encrypted tables can be stored and queried by existing DMBSs without any modifications. In fact, the belief that STE-based solutions can only work on custom servers is a widespread and established belief in cryptography community and a large part of why PPE-based solutions are used in practice regardless of their leakage profiles.

1.2 Related Work

We already discussed related work on PPE-based and STE-based relational encrypted databases so we focus here on work in encrypted search and on other types of EDBs.

[2]SGX currently allows 90MB of working memory and our proxy is around 350kb in size.

Encrypted search. Encrypted search was first considered explicitly by Song, Wagner and Perrig in [50] which introduced the notion of searchable symmetric encryption (SSE). Goh provided the first security definition for SSE and a solution based on Bloom filters with linear search complexity. Chang and Mitzenmacher proposed an alternative security definition and construction, also with linear search complexity. Curtmola et al. introduced and formulated the notion of adaptive semantic security for SSE [26] together with the first sub-linear and optimal-time constructions. Chase and Kamara introduced the notion of structured encryption which generalizes SSE to arbitrary data structures [21].

Federated EDBs. Federated EDBs are systems that are composed of multiple autonomous encrypted databases. Most federated EDBs use secure multi-party computation (MPC) to query the constituent EDBs securely. In this model, multiple parties hold a piece of the database (either tables or rows) and a public query is executed in such a way that no information about the database is revealed beyond what can be inferred from the result and some additional leakage. Examples include SMCQL [11] and Conclave [55], which store the databases as secret shares and encryptions, respectively, and use MPC to execute the sensitive parts of a SQL query on the shared/encrypted data. We note that standard EDBs like `KafeDB` can be combined with MPC to yield a federated EDB.

Other EDBs. Other encrypted databases include ARX by Poddar, Boelter and Popa [46] and Jana by Galois [8]. While ARX is SSE-based, it is not a *relational* EDB since it is built on top of MongoDB. The authors choose to describe their queries using SQL for convenience but ARX does not store relational data or handle SQL/relational queries. Note that simply translating SQL queries to MongoDB queries using a SQL translator is not appropriate as this would alter the security/leakage guarantees claimed by ARX. The Jana system stores data either as MPC shares or encrypted using deterministic and order-preserving encryption depending on the efficiency/leakage tradeoff that is desired. Queries are then either handled using MPC or directly on the PPE-encrypted data. Jana currently has no formal leakage analysis or experimental results so it is not clear what its leakage profile or performance is in either mode of operation.

1.3 Main Results

The central problems we consider in this thesis are

1. Efficiency: how to make STE-based schemes worst-case optimal in time and space complexity? How to increase locality for I/O efficiency? How to enable encrypted query optimization?

2. Security: How to improve the security of complex queries for the STE-based schemes

3. Legacy Compliance: How to make STE-based schemes work on any standard relational databases?

We presented our solutions in two construction, the OPX and `pkfk` schemes, and a system `KafeDB`. We summarize our main results in Figure 1.1.

Efficiency	CryptDB [48, 43]	SPX [39]	OPX (Thesis)	`pkfk` (Thesis)
Time [53]	linear	quadratic	quadratic	**linear**
Space	linear	quadratic	quadratic	**linear**
Setup	linear	quadratic	quadratic	**linear**
Locality	Y	N	N	**Y**
Query Opt.	Y	N	**Y**	Y

(a) Efficiency (worst-case in DB size) comparisons.

Leakage	CryptDB	SPX	OPX (Thesis)	`pkfk` (Thesis)
Setup	$\forall$att $\in$ DB :VF,RC	$\sum_{\mathsf{DB}}$ J, $\sum_{\mathsf{DB}}$ T	J,T	T
Filtered Join	JP,RC	JP,RC	JP,RC	F-JP,F-RC
Conj. Filters	$\forall$att $\in Q$:VF,RC	$\forall$att $\in Q$:VF,RC	VF,RC	VF,RC

(b) Leakage comparisons for the query Q of filtered joins and conjunctive filters.

Leakage	Abbrev.	Description
Value frequency	VF	$\forall x \in \mathsf{dom}(\mathsf{att}), \mathsf{count}(x \in \mathsf{col}(\mathsf{att}))$
Row Collocation	RC	$\forall x \in \mathsf{dom}(\mathsf{att}), y \in \mathsf{dom}(\mathsf{att}'), \exists (x, y) \in \mathbf{T}$
Join Pattern	JP	$\forall \mathbf{r} \in \mathsf{col}(\mathsf{att}), \mathbf{r}' \in \mathsf{col}(\mathsf{att}'), \exists (\mathbf{r}, \mathbf{r}') \in \bowtie_{\mathsf{att}=\mathsf{att}'}$
Join Size	J	$\lvert$JP$\rvert$
Table Size	T	$\lvert\mathbf{T}\rvert$
Filtered Join Pattern	F-J	JP subject to filter only
Filtered Row Collocation	F-RC	RC subject to filter only

(c) Leakage descriptions.

Figure 1.1: Efficiency and security comparisons for the state-of-the-art PPE-based approache as in CryptDB, the initial STE-based approach SPX, and this thesis work in OPX, `pkfk`. The main results of this thesis are OPX, `pkfk` which include query optimziation, optimal complexity, locality and less leakage than the prior PPE- or STE-based solutions.

Encrypted Query Optimization. A relational query may consist of multiple operators typically organized as a query tree. Query optimization refers to re-ordering query tree such that the execution time may be reduced. For example, `filter pushdown` is a tpyical query optimization rule that pushes the filter before join, such that the worst-case quadratic join operator can benefit from a potentially large constant factor in query complexity reduction. For PPE-based databases, rules such as `filter pushdown` and `join reordering` can be done trivially by the client for encrypted queries as well. For STE-based approaches such as SPX, we need to change how the encrypted indexes are designed in order to enable this optimization. We proposed a solution in OPX (Ch. 4).

Query Complexity. The advantage of encrypting relational databases using PPE such as CryptDB [47] is that PPE allows the query operators to be executed directly over the ciphertexts in a similar way to the plaintexts, thereby achieving similar efficiency. The first STE-based scheme, SPX [39], on the other hand, has much reduced leakage profile, but suffers suboptimal query complexities, following the notion in **??**. Though in RAM model, SPX, OPX are shown to be optimal *in query output size*, the computational cost is better captured by the *query input size*, which becomes even more significant for example in external memory model. For plaintext, some query operations such as binary `JOIN`s are known to be worst-case $O(\mathsf{DB}^2)$, but if quantified by input size, linear binary join algorithms exist such as Hash Joins. Unfortunately SPX, OPX are both worst-case $O(\mathsf{DB}^2)$ also in query input size, so a pressing question is how to make STE-based approach match the plaintext query complexity. We addressed this question by proposing a new scheme (Ch. 6).

Space Complexity. The PPE-based schemes can achieve storage size linear in the plaintext database size. However for STE-based schemes like SPX the space complexity goes to worst-case quadratic in database size. The underlying issue is that it was unclear how to go beyond the straightforward (quadratic) way to index JOINs securely. Here we proposed a new way to represent JOIN that reduces the storage requirement to linear (Ch. 6).

Preprocessing Complexity. The STE-based schemes such as SPX require precomputing all possible binary joins among all attributes for indexing. This means that the cost of preprocessing is worst-case quadratic in both the database size and the schema size. It would be more ideal if we do not need to precompute the JOINs, but can still index JOINs securely. We proposed a linear algorithm to reduce preprocessing to one pass over the database, without precomputing each JOINs (Ch. 6).

Locality. The approach that the PPE-based scheme SPX takes is essentially to represent each relational operator such as fitler, join and projection in separate encrypted data structures. The computation of a query of multiple operators would have to search different encrypted data structures, which tend to result in random accesses to the storage device. However, due to encryption, the server cannot optimize the storage of these encrypted data structures for locality. We proposed a scheme that alters these encrypted data structures securely while permitting ciphertexts to be collocated (Ch. 6).

Security. Though STE-based schemes such as SPX provide much stronger security guarantee than PPE-based schemes, there are still cases where their leakage can be further reduced. For `filtered join`, the essential operation in the class of conjunctive queries, SPX can leak the joint frequency of the full JOIN, despite probably only a subset is needed for the result. For adversaries that observe both queries and results, this leakage can be devastating. Therefore it is an important open problem as to how to reduce this leakage. We proposed a solution in `pkfk` (Ch. 6).

Legacy Compliance. PPE-based schemes typically are easier to implement on top of a standard relational database systems. This has not been the case for STE-based approaches. We established a methodology called *emulation* that translates encrypted data structures and their operations to a domain specific language such as SQL. We found that the relation model which underlies all SQL databases can be used to model the dictionary-based STE constructions, and in particular for the counter-based constructions such as [17], a fixed-point operator is required in addition to relational algebra to express the computation. Emulation allows us to build a system called `KafeDB` which can work on top of any standard SQL databases (Ch. **??**).

Chapter 2

Preliminaries

2.1 Basic Notions

Notation. The set of all binary strings of length n is denoted as $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. $[n]$ is the set of integers $\{1,\ldots,n\}$. The output x of an algorithm $\mathcal{A}$ is denoted by $x \leftarrow \mathcal{A}$. Given a sequence $\mathbf{r}$ of n elements, we refer to its ith element as r_i or $\mathbf{r}[i]$. If S is a set then $\#S$ refers to its cardinality. Throughout, k will denote the security parameter.

Dictionaries and multi-maps. A dictionary DX with capacity n is a collection of n label/value pairs $\{(\ell_i, v_i)\}_{i\leq n}$ and supports get and put operations. We write $v_i := \mathsf{DX}[\ell_i]$ to denote getting the value associated with label ℓ_i and $\mathsf{DX}[\ell_i] := v_i$ to denote the operation of associating the value v_i in DX with label ℓ_i. A multi-map MM with capacity n is a collection of n label/tuple pairs $\{(\ell_i, \mathbf{v}_i)_i\}_{i\leq n}$ that supports Get and Put operations. We write $\mathbf{v}_i = \mathsf{MM}[\ell_i]$ to denote getting the tuple associated with label ℓ_i and $\mathsf{MM}[\ell_i] = \mathbf{v}_i$ to denote operation of associating the tuple $\mathbf{v}_i$ to label ℓ_i. Multi-maps are the abstract data type instantiated by an inverted index. In the encrypted search literature multi-maps are sometimes referred to as indexes, databases or tuple-sets (T-sets) [18, 17].

Relational databases. We denote a relational database $\mathsf{DB} = (\mathbf{T}_1, \ldots, \mathbf{T}_n)$, where each $\mathbf{T}_i$ is a two-dimensional array with rows corresponding to an entity (e.g., a customer or an employee) and columns corresponding to attributes (e.g., age, height, salary). For any given attribute, we refer to the set of all possible values that it can take as its *space* (e.g., integers, booleans, strings). We define the *schema* of a table $\mathbf{T}$ to be its set of attributes and denote it $\mathbb{S}(\mathbf{T})$. For a row $\mathbf{r} \in \mathbf{T}_i$, its table identifier $\mathsf{tbl}(\mathbf{r})$ is i and its row rank $\mathsf{rrk}(\mathbf{r})$ is its position in $\mathbf{T}_i$ when viewed as a list of rows. Similarly, for a column $\mathbf{c} \in \mathbf{T}_i^{\intercal}$, its table identifier $\mathsf{tbl}(\mathbf{c})$ is i and its column rank $\mathsf{crk}(\mathbf{c})$ is its position in $\mathbf{T}_i$ when viewed as a list of columns. For any row $\mathbf{r} \in \mathbf{T}$ and any column $\mathbf{c} \in \mathbf{T}$, we refer to the pairs $\chi(\mathbf{r}) \overset{def}{=} (\mathsf{tbl}(\mathbf{r}), \mathsf{rrk}(\mathbf{r}))$ and $\chi(\mathbf{c}) \overset{def}{=} (\mathsf{tbl}(\mathbf{c}), \mathsf{crk}(\mathbf{c}))$, respectively, as their *coordinates* in DB. For any attribute

$\mathsf{att} \in \mathbb{S}(\mathsf{DB})$ and constant a belonging to the attribute's domain, $\mathsf{DB}_{\mathsf{att}=a}$ is the set of rows $\left\{\mathbf{r} \in \mathsf{DB} : \mathbf{r}[\mathsf{att}] = a\right\}$.

SQL. In this work, we focus on the class of *conjunctive SQL* queries, which have the form,

```
SELECT attributes
FROM tables
WHERE att_1 = X_1 AND att_2 = X_2 ,
```

where X_i is either an attribute or a constant value. If X_i is a constant, then the predicate $\mathsf{att}_i = X_i$ is a *constant predicate* whereas if X_i is an attribute, then the predicate $\mathsf{att}_i = X_i$ is called a *join predicate*. A formula is a Boolean expression composed of constant and join predicates. We use standard relational algebra notation and denote the filtering operator by σ, the projection operator by π, the rename operator by ρ, the θ-join operator by $\underset{\theta}{\bowtie}$ and the cross join operator by $\times$.

Basic cryptographic primitives. A private-key encryption scheme is a set of three polynomial-time algorithms $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ such that Gen is a probabilistic algorithm that takes a security parameter k and returns a secret key K; Enc is a probabilistic algorithm that takes a key K and a message m and returns a ciphertext c; Dec is a deterministic algorithm that takes a key K and a ciphertext c and returns m if K was the key under which c was produced. Informally, a private-key encryption scheme is secure against chosen-plaintext attacks (CPA) if the ciphertexts it outputs do not reveal any partial information about the plaintext even to an adversary that can adaptively query an encryption oracle. We say a scheme is random-ciphertext-secure against chosen-plaintext attacks (RCPA) if the ciphertexts it outputs are computationally indistinguishable from random even to an adversary that can adaptively query an encryption oracle.[1] In addition to encryption schemes, we also make use of pseudo-random functions (PRF), which are polynomial-time computable functions that cannot be distinguished from random functions by any probabilistic polynomial-time adversary.

2.2 Structured Encryption

The design of DEX is based on structured encryption (STE)– a cryptographic primitive introduced by Chase and Kamara [21] which allows a user to encrypt an arbitrary data structure and to be able to privately query it later. DEX falls under the category of *structured encrypted database algorithms* (or STE-based EDB) for which we are going to detail both the syntax and security below.

[1]RCPA-secure encryption can be instantiated practically using either the standard PRF-based private-key encryption scheme or, e.g., AES in counter mode.

Syntax. A structured encrypted encryption scheme $\mathsf{STE} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Update})$ consists of three efficient algorithms. Setup takes as input a security parameter 1^k and a data structure DS and outputs a secret key K and an encrypted data structure EDS. Query is a two-party protocol between a client and a server. The client inputs its secret key K and a query q and the server inputs an encrypted data structure EDS. The client receives an encrypted response ct and the server receives $\perp$. Update is a two-party protocol between a client and a server. The client inputs its secret key K and an update u and the server inputs an encrypted data structure EDS. The client receives $\perp$ and the server receives an updated encrypted data structure EDS'.

Security. There are two adversarial models for STE: persistent adversaries and snapshot adversaries. A persistent adversary observes: (1) the encrypted data; and (2) the transcripts of the interaction between the client and the server when a query is made. A snapshot adversary, on the other hand, only receives the encrypted data after a query has been executed. Persistent adversaries capture situations in which the server is completely compromised whereas snapshot adversaries capture situations where the attacker recovers only a snapshot of the server's memory.

The security of STE is formalized using "leakage-parameterized" definitions following [26, 21]. In this framework, a design is proven secure with respect to a security definition that is parameterized with a specific leakage profile. Leakage-parameterized definitions for persistent adversaries were given in [26, 21] and for snapshot adversaries in [5].[2]

The leakage profile of a scheme captures the information an adversary learns about the data and/or the queries. Depending on the type of the adversary, the leakage can simply be the information the adversary learns by storing the encrypted database such as its size in the case of a snapshot adversary; or more sophisticated such as the size of the result tables or frequencies of SQL queries in the case of a persistent adversary. Each operation on the encrypted data structure is associated with a set of *leakage patterns* and this collections of sets forms the scheme's *leakage profile*.

We recall the informal security definition for STE and refer the reader to [26, 21, 5] for more details.

Definition 2.2.1 *[Security vs. persistent adversary (Informal)] Let* $\Lambda = \Big(\mathcal{L}_{\mathsf{S}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{U}}\Big) = \Big(\mathsf{patt}_1, \mathsf{patt}_2, \mathsf{patt}_3\Big)$ *be a leakage profile. An encrypted database algorithm* STE *is* Λ*-secure if there exists a* PPT *simulator that, given* $\mathsf{patt}_1(\mathsf{DB})$ *for an adversarially-chosen database* DB*,* $\mathsf{patt}_2(\mathsf{DB}, q_1, \ldots, q_t)$ *for adaptively-chosen queries* $(q_1, \ldots, q_t)$*, and* $\mathsf{patt}_3(\mathsf{DB}, u_1, \ldots, u_t)$ *for adaptively-chosen updates* $(u_1, \ldots, u_t)$ *can simulate the view of any* PPT *adversary. Here, the view includes the encrypted data structure and the transcript of the queries.*

Definition 2.2.2 *[Security vs. single snapshot adversary (Informal)] Let* $\Lambda = \mathcal{L}_{\mathsf{snap}}$ *be a snapshot leakage profile. An encrypted database algorithm* EDBA *is* Λ*-secure if there exists a* PPT *simulator that, given* $\mathcal{L}_{\mathsf{snap}}(\mathsf{DB}, \mathsf{op}_1, \cdots, \mathsf{op}_t)$ *for an adversarially-chosen database* DB

[2]Even though parameterized definitions were introduced in the context of SSE and STE, they can be (and have been) applied to other primitives, including to FHE-, PPE-, ORAM- and FE-based solutions.

and adaptively-chosen SQL queries or updates, can simulate the view of any PPT *adversary. Here the view includes the encrypted data collection only.*

The definition above can be naturally generalized to multiple snapshots, refer to the work by Amjad et al. [5].

Encrypted dictionaries and multi-maps. An encrypted dictionary EDX is an encryption of a dictionary DX that supports encrypted get and put operations. Similarly, an encrypted multi-map EMM is an encryption of a multi-map MM that supports encrypted get and put operations. Multi-map encryption schemes are structured encryption (STE) schemes for multi-maps and have been extensively investigated. Many practical constructions are known that achieve different tradeoffs between query and storage complexity, leakage and locality [26, 42, 17, 17, 19, 51, 16, 41]. Encrypted dictionaries can be obtained from any encrypted multi-map since the former is just an encrypted multi-map with single-item tuples.

2.3 Definitions

In this Section, we define the syntax and security of STE schemes. A STE scheme encrypts data structures in such a way that they can be privately queried. There are several natural forms of structured encryption. The original definition of [21] considered schemes that encrypt both a structure and a set of associated data items (e.g., documents, emails, user profiles etc.). In [22], the authors also describe *structure-only* schemes which only encrypt structures. Another distinction can be made between *interactive* and *non-interactive* schemes. Interactive schemes produce encrypted structures that are queried through an interactive two-party protocol, whereas non-interactive schemes produce structures that can be queried by sending a single message, i.e, the token. One can also distinguish between *response-hiding* and *response-revealing* schemes: the latter reveal the query response to the server whereas the former do not.

In this work, we focus on non-interactive structure-only schemes. Our main construction, opx, is response-hiding but makes use of response-revealing schemes as building blocks. As such, we define both forms below. At a high-level, non-interactive STE works as follows. During a setup phase, the client constructs an encrypted structure EDS under a key K from a plaintext structure DS. The client then sends EDS to the server. During the query phase, the client constructs and sends a token tk generated from its query q and secret key K. The server then uses the token tk to query EDS and recover either a response r or an encryption ct of r depending on whether the scheme is response-revealing or response-hiding.

Definition 2.3.1 *[Response-revealing structured encryption [21]] A response-revealing structured encryption scheme* $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$ *consists of three polynomial-time algorithms that work as follows:*

- $(K, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, \mathsf{DS})$*: is a probabilistic algorithm that takes as input a security parameter* 1^k *and a structure* DS *and outputs a secret key* K *and an encrypted structure* EDS.

- $\mathsf{tk} \leftarrow \mathsf{Token}(K, q)$*: is a (possibly) probabilistic algorithm that takes as input a secret key* K *and a query* q *and returns a token* tk*.*

- $\big\{\perp, r\big\} \leftarrow \mathsf{Query}(\mathsf{EDS}, \mathsf{tk})$*: is a deterministic algorithm that takes as input an encrypted structure* EDS *and a token* tk *and outputs either* $\perp$ *or a response.*

We say that a response-revealing structured encryption scheme Σ *is correct if for all* $k \in \mathbb{N}$*, for all* $\mathsf{poly}(k)$*-size structures* $\mathsf{DS} : Q \rightarrow \mathbf{R}$*, for all* (K, EDS) *output by* $\mathsf{Setup}(1^k, \mathsf{DS})$ *and all sequences of* $m = \mathsf{poly}(k)$ *queries* $q_1, \ldots, q_m$*, for all tokens* tk_i *output by* $\mathsf{Token}(K, q_i)$*,* $\mathsf{Query}(\mathsf{EDS}, \mathsf{tk}_i)$ *returns* $\mathsf{DS}(q_i)$ *with all but negligible probability.*

Definition 2.3.2 *[Response-hiding structured encryption [21]] A response-hiding structured encryption scheme* $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query}, \mathsf{Dec})$ *consists of four polynomial-time algorithms such that* Setup *and* Token *are as in Definition 2.3.1 and* Query *and* Dec *are defined as follows:*

- $\{\perp, \mathrm{ct}\} \leftarrow \mathsf{Query}(\mathsf{EDS}, \mathsf{tk})$*: is a deterministic algorithm that takes as input an encrypted structured* EDS *and a token* tk *and outputs either* $\perp$ *or a ciphertext* ct*.*

- $r \leftarrow \mathsf{Dec}(K, \mathrm{ct})$*: is a deterministic algorithm that takes as input a secret key* K *and a ciphertext* ct *and outputs a response* r*.*

We say that a response-hiding structured encryption scheme Σ *is correct if for all* $k \in \mathbb{N}$*, for all* $\mathsf{poly}(k)$*-size structures* $\mathsf{DS} : Q \rightarrow \mathbf{R}$*, for all* (K, EDS) *output by* $\mathsf{Setup}(1^k, \mathsf{DS})$ *and all sequences of* $m = \mathsf{poly}(k)$ *queries* $q_1, \ldots, q_m$*, for all tokens* tk_i *output by* $\mathsf{Token}(K, q_i)$*,* $\mathsf{Dec}_K\bigg(\mathsf{Query}\bigg(\mathsf{EDS}, \mathsf{tk}_i\bigg)\bigg)$ *returns* $\mathsf{DS}(q_i)$ *with all but negligible probability.*

Security. The standard notion of security for structured encryption guarantees that an encrypted structure reveals no information about its underlying structure beyond the setup leakage $\mathcal{L}_\mathsf{S}$ and that the query algorithm reveals no information about the structure and the queries beyond the query leakage $\mathcal{L}_\mathsf{Q}$. If this holds for non-adaptively chosen operations then this is referred to as non-adaptive semantic security. If, on the other hand, the operations are chosen adaptively, this leads to the stronger notion of adaptive semantic security. This notion of security was introduced by Curtmola *et al.* in the context of SSE [26] and later generalized to structured encryption in [21].

Definition 2.3.3 *[Adaptive semantic security [26, 21]] Let* $\Sigma = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$ *be a response-revealing structured encryption scheme and consider the following probabilistic experiments where* $\mathcal{A}$ *is a stateful adversary,* $\mathcal{S}$ *is a stateful simulator,* $\mathcal{L}_\mathsf{S}$ *and* $\mathcal{L}_\mathsf{Q}$ *are leakage profiles and* $z \in \{0,1\}^*$*:*

$\mathbf{Real}_{\Sigma,\mathcal{A}}(k)$*: given* z *the adversary* $\mathcal{A}$ *outputs a structure* DS*. It receives* EDS *from the challenger, where* $(K, \mathsf{EDS}) \leftarrow \mathsf{Setup}(1^k, \mathsf{DS})$*. The adversary then* adaptively *chooses a polynomial number of queries* $q_1, \ldots, q_m$*. For all* $i \in [m]$*, the adversary receives* $\mathsf{tk} \leftarrow \mathsf{Token}(K, q_i)$*. Finally,* $\mathcal{A}$ *outputs a bit* b *that is output by the experiment.*

Ideal$_{\Sigma,\mathcal{A},\mathcal{S}}(k)$: *given z the adversary $\mathcal{A}$ generates a structure* DS *which it sends to the challenger. Given z and leakage $\mathcal{L}_\mathsf{S}(\mathsf{DS})$ from the challenger, the simulator $\mathcal{S}$ returns an encrypted data structure* EDS *to $\mathcal{A}$. The adversary then* adaptively *chooses a polynomial number of operations $q_1, \ldots, q_m$. For all $i \in [m]$, the simulator receives a tuple $\Big(\mathsf{DS}(q_i), \mathcal{L}_\mathsf{Q}(\mathsf{DS}, q_i)\Big)$ and returns a token tk_i to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a bit b that is output by the experiment.*

We say that Σ is adaptively $(\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{Q})$-semantically secure if for all PPT *adversaries $\mathcal{A}$, there exists a* PPT *simulator $\mathcal{S}$ such that for all $z \in \{0,1\}^*$, the following expression is negligible in k:*

$$|\Pr[\,\mathbf{Real}_{\Sigma,\mathcal{A}}(k) = 1\,] - \Pr[\,\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(k) = 1\,]|$$

The security definition for *response-hiding* schemes can be derived from Definition 2.3.3 by giving the simulator $\Big(\perp, \mathcal{L}_\mathsf{Q}(\mathsf{DS}, q_i)\Big)$ instead of $\Big(\mathsf{DS}(q_i), \mathcal{L}_\mathsf{Q}(\mathsf{DS}, q_i)\Big)$.

Modeling leakage. Every STE scheme is associated with leakage which itself can be composed of multiple *leakage patterns*. The collection of all these leakage patterns forms the scheme's *leakage profile*. Leakage patterns are (families of) functions over the various spaces associated with the underlying data structure. For concreteness, we borrow the nomenclature introduced in [41] and recall some well-known leakage patterns that we make use of in this work. Here $\mathbf{D}$ and $\mathbb{Q}$ refer to the space of all possible data objects and the space of all possible queries for a given data type. In this work, we consider the following leakage patterns:

- the *query equality pattern* is the function family $\mathsf{qeq} = \{\mathsf{qeq}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{qeq}_{k,t} : \mathbf{D}_k \times \mathbb{Q}_k^t \to \{0,1\}^{t\times t}$ such that $\mathsf{qeq}_{k,t}(\mathsf{DS}, q_1, \ldots, q_t) = M$, where M is a binary $t \times t$ matrix such that $M[i,j] = 1$ if $q_i = q_j$ and $M[i,j] = 0$ if $q_i \neq q_j$. The query equality pattern is referred to as the search pattern in the SSE literature;

- the *response identity pattern* is the function family $\mathsf{rid} = \{\mathsf{rid}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{rid}_{k,t} : \mathbf{D}_k \times \mathbb{Q}_k^t \to [2^{[n]}]^t$ such that $\mathsf{rid}_{k,t}\Big(\mathsf{DS}, q_1, \ldots, q_t\Big) = (\mathsf{DS}[q_1], \ldots, \mathsf{DS}[q_t])$. The response identity pattern is referred to as the access pattern in the SSE literature;

- the *response length pattern* is the function family $\mathsf{rlen} = \{\mathsf{rlen}_{k,t}\}_{k,t\in\mathbb{N}}$ with $\mathsf{rlen}_{k,t} : \mathbf{D}_k \times \mathbb{Q}_k^t \to \mathbb{N}^t$ such that $\mathsf{rlen}_{k,t}(\mathsf{DS}, q_1, \ldots, q_t) = \Big(|\mathsf{DS}[q_1]|, \ldots, |\mathsf{DS}[q_t]|\Big)$;

Chapter 3

A Tour of the Main Results

3.1 Overview

The first step towards encrypting relational data using structured encryption (STE) is to represent the relational model as some data structures that have STE constructions, and if we do so with care we should end up with a secure relational database. This approach is first proposed in [39], and serves as a foundation upon which this thesis developes. The bulk of this thesis will be focusing on how to improve the security and efficiency from the baseline approach in [39] from both the data structure and the encryption. We will see different constructions and eventually arrive at a native STE data structure for relational model for the best performance.

This baseline approach in SPX [39] however opens up many unsolved problems, amongst which we explore mainly three aspects in this thesis to derive new results

- Encypted Query Optimization
- Emulation for Legacy Compliance
- Optimal Efficiency and Better Security

We will first give an informal account of the main reults we achieve in these aspects in the following sections, and give formal treatment in the subsequent chapters.

3.2 Mapping the Relational Model

We fisrt review the idea that the relational model can be represented as a set of multimaps, which is essentially the idea first proposed in the SPX scheme [39]. The resulting exeuction is called the *indexed exectuion.*

The idea of representing the relational model as a set of multimaps is shown in an example in Figure 3.1. Abstractlly, we first restrict ourselves to the class of conjunctive queries, i.e. queries that have only conjuctive filters and joins, (and of course projections). Then we encrypt each cell in the table using RCPA-secure cipher. This by definition makes the encrypted cells not queryable, because all ciphertexts are computationally indistinguishable to randomly generated bits. The key step to enable relational query computation over the encrypted cells is to associate each encrypted row in the table a row identifier, or a rid, say as an extra primary key $\mathsf{att}_{\mathsf{rid}}$. Then we index all the values for filtering and joining through the rids alone.

Row Identifiers. Each row identifier $\mathsf{rid}$is mapped to its associated row in the table,

$$\mathsf{rid} \to \{\mathrm{ct} \mid \mathrm{ct} \in \sigma_{\mathsf{att}_{\mathsf{rid}}=\mathsf{rid}} \mathbf{T}\}$$

Where all these mappings are stored in $\mathsf{MM}_{\mathsf{rid}}$ (the MM_R in [39]).

Filters. The filter operator $\sigma_{\mathsf{att}=x}$ can be represented as mapping between the label and values

$$\mathbf{T}.\mathsf{att} \| x \to \{\mathsf{rid} \mid \forall \mathsf{rid} \in \pi_{\mathsf{att}_{\mathsf{rid}}} \sigma_{\mathsf{att}=x}(\mathbf{T})\}$$

Nonexisting filter value can be represented as the absence of such mapping. All mappings are stored in a multimap MM_σ (the MM_V in [39]).

Joins. Similarly, we can encode the mappings for join as

$$\mathbf{T}.\mathsf{att} \| \mathbf{T}'.\mathsf{att}' \to \{(\mathsf{rid}, \mathsf{rid}') \mid \forall \sigma_{\mathsf{att}_{\mathsf{rid}}, \mathsf{att}_{\mathsf{rid}'}} \mathbf{T} \bowtie_{\mathsf{att}=\mathsf{att}'} \mathbf{T}'\}$$

and store all such mappings in $\mathsf{MM}_\bowtie$ (the $\mathsf{MM}_{\mathsf{att}}$ in [39]).

Projections. We can also encode the mappings for projections as

$$\mathbf{T}.\mathsf{att} \to \{\mathrm{ct} \mid \mathrm{ct} \in \pi_{\mathsf{att}} \mathbf{T}\}$$

and store all such mappings in MM_π (the MM_C in [39]).

An Example Query. Notice how we can process the filter query $\sigma_{\mathsf{att}=x}\mathbf{T}$ by using these two multimaps $\mathsf{MM}_\sigma, \mathsf{MM}_{\mathsf{rid}}$,

1. For each $\mathsf{rid} \in \mathsf{MM}_\sigma[\mathbf{T}.\mathsf{att}\|x]$,
   (a) Output $\mathsf{MM}_{\mathsf{rid}}[\mathsf{rid}]$

Or using set-oriented language,

$$\sigma_{\mathsf{att}=x}\mathbf{T} = \{r \mid \forall \mathsf{rid} \in \mathsf{MM}_\sigma[\mathbf{T}.\mathsf{att}\|x], \exists r \in \mathsf{MM}_{\mathsf{rid}}[\mathsf{rid}]\}$$

Customer		
rid	Name	Nation
c1	Abe	US
c2	Bob	US
c3	Bob	Canada
c4	Cay	Canada

(a) Customer Table.

Supplier		
rid	Name	Nation
s1	Intel	US
s2	IBM	US
s3	Apple	US
s4	RIM	Canada
s5	WestJet	Canada

(b) Supplier Table.

$$\mathsf{EMM}_\sigma \begin{pmatrix} C.Nation\|US & \rightarrow & (c_1, c_2) \\ C.Nation\|Canada & \rightarrow & (c_3, c_4) \\ C.Name\|Abe & \rightarrow & (c_1) \\ C.Name\|Bob & \rightarrow & (c_2, c_3) \\ C.Name\|Cay & \rightarrow & (c_4) \\ S.Nation\|US & \rightarrow & (s_1, s_2, s_3) \\ S.Nation\|Canada & \rightarrow & (s_4, s_5) \end{pmatrix}$$

(c) A filter multimap.

$$\mathsf{EMM}_\bowtie \begin{pmatrix} \text{C.Nation}\|\text{S.Nationn} & \rightarrow & ((c_1, s_1) \\ & & (c_1, s_2) \\ & & (c_1, s_3) \\ & & (c_2, s_1) \\ & & (c_2, s_2) \\ & & (c_2, s_3) \\ & & (c_3, s_4) \\ & & (c_3, s_5) \\ & & (c_4, s_4) \\ & & (c_4, s_5)) \end{pmatrix}$$

(d) A join multimep.

$$\mathsf{EMM}_\pi \begin{pmatrix} C.Name & \rightarrow & (\mathsf{cid}_{C.Name}, \mathsf{Enc}(Abe), \mathsf{Enc}(Bob), \mathsf{Enc}(Bob), \mathsf{Enc}(Cay)) \\ C.Nation & \rightarrow & (\mathsf{cid}_{C.Name}, \mathsf{Enc}(US), \mathsf{Enc}(US), \mathsf{Enc}(Canada), \mathsf{Enc}(Canada)) \\ S.Name & \rightarrow & (\mathsf{cid}_{S.Name}, \mathsf{Enc}(Intel), \mathsf{Enc}(IBM), \mathsf{Enc}(Apple), \mathsf{Enc}(RIM), \mathsf{Enc}(WestJet)) \\ S.Nation & \rightarrow & (\mathsf{cid}_{S.Nation}, \mathsf{Enc}(US), \mathsf{Enc}(US), \mathsf{Enc}(US), \mathsf{Enc}(Canada), \mathsf{Enc}(Canada)) \end{pmatrix}$$

(e) A projection multimap.

$$\mathsf{EMM}_{\mathsf{rid}} \begin{pmatrix} c_1 & \rightarrow & (c_1, \mathsf{Enc}(Abe), \mathsf{Enc}(US)) \\ c_2 & \rightarrow & (c_2, \mathsf{Enc}(Bob), \mathsf{Enc}(US)) \\ c_3 & \rightarrow & (c_3, \mathsf{Enc}(Bob), \mathsf{Enc}(Canada)) \\ c_4 & \rightarrow & (c_4, \mathsf{Enc}(Cay), \mathsf{Enc}(Canada)) \\ s_1 & \rightarrow & (s_1, \mathsf{Enc}(Intel), \mathsf{Enc}(US)) \\ s_2 & \rightarrow & (s_2, \mathsf{Enc}(IBM), \mathsf{Enc}(US)) \\ s_3 & \rightarrow & (s_3, \mathsf{Enc}(Apple), \mathsf{Enc}(US)) \\ s_4 & \rightarrow & (s_4, \mathsf{Enc}(RIM), \mathsf{Enc}(Canada)) \\ s_5 & \rightarrow & (s_5, \mathsf{Enc}(WestJet), \mathsf{Enc}(Canada)) \end{pmatrix}$$

(f) A row identifier multimap.

Figure 3.1: Example tables and SPX multimaps that encode the class of conjuctive queries over the tables. We rename each multimap with a suffix as relational algebra operator to signifiy the usage.

Encryption. All multimaps can then be encrypted using an existing EMM construction. For example, the Π_{bas} EMM construction in [17] has the following leakage profile

- setup leakage: the size of the EMM (the number of label-value pairs)
- Query leakage: the response of the query (the values associated with the queried label)

Then to analyze the total leakage of the SPX scheme, we just need to analyze the leakage by each EMM used in the scheme, and for a query of multiple operators, the total leakage will then be the cancatenation of each underlying EMM's leakage. [39] shows that the setup leakage is just the number of total size (number of cells) of the entire database, and the total size of all the supported joins in the database. To see this, just examine the size of $\mathsf{EMM}_\sigma, \mathsf{EMM}_\pi, \mathsf{EMM}_{\mathsf{rid}}$, and see that each of them essentially encode each table cell in the database once, and the $\mathsf{EMM}_\bowtie$ encodes the total number of pairs of rids in each join. The query leakage on the other hand is the concatenation of the query leakages of all the EMMs involved in processing the operators:

- Filter: the set of row identifiers (i.e. the value frequency) under the filter
- Join: the set of row identifier pairs (i.e. the joint frequency) under the join

We will subsequently see that there are several asepcts that this baseline approach in SPX does not yet address.

3.3 Encrypted Query Optimization

Query optimization refers to reordering the query operators in order to achieve better performance. This is important because query operators may have different complexities.

For exampe, the following query has two equivalent forms

$$\sigma_{\mathbf{T}_1.\mathsf{att}_1=x}(\mathbf{T}_1 \bowtie_{\mathbf{T}_1.\mathsf{att}_2=\mathbf{T}_2.\mathsf{att}_2} \mathbf{T}_2) \quad \equiv \quad (\sigma_{\mathbf{T}_1.\mathsf{att}_1=x}\mathbf{T}_1) \bowtie_{\mathbf{T}_1.\mathsf{att}_2=\mathbf{T}_2.\mathsf{att}_2} \mathbf{T}_2$$

Both forms can be express algorithmically using the multimaps as

```
1. Let $\mathbf{R}_\sigma = \mathsf{MM}_\sigma[\mathbf{T}_1.\mathsf{att}_1 \| x]$
2. For each $(\mathbf{T}_1.\mathsf{rid}, \mathbf{T}_2.\mathsf{rid}') \in \mathsf{MM}_\bowtie[\mathbf{T}_1.\mathsf{att}_2 \| \mathbf{T}_2.\mathsf{att}_2]$,
    (a) If $\mathbf{T}_1.\mathsf{rid}$ exists in $\mathbf{R}_\sigma$, then output
            $\mathsf{MM}_{\mathsf{rid}}[\mathbf{T}_1.\mathsf{rid}] \| \mathsf{MM}_{\mathsf{rid}}[\mathbf{T}_2.\mathsf{rid}']$
```

Notice that there are only one unique way of expressing the indexed execution, though there are two query forms. Looking closer to the indexed execution, we see that it is always the retrieving $O(T + T^2)$ number of pairs from the MM_σ and $\mathsf{MM}_\bowtie$ for table size T. With tpyical query optimziation that exists in relational database, the second query form with filter pushdown in the above example can run in $O(sT^2)$ for a potentially small selectivity $0 \leq s \leq 1$.

New encoding. The key impediment towards implementing query optimization via reordering is that the multimap for join $\mathsf{MM}_\bowtie$ always retrieve $O(T^2)$ join pairs. So the solution is to change the definition of the label to capture the potential dependency on a set of pre-filtered row identifiers. For example, if we relax the label for $\mathsf{MM}_\bowtie$ to be $\mathbf{T}_1.\mathsf{rid} \| \mathbf{T}_1.\mathsf{att}_2 \| \mathbf{T}_2.\mathsf{att}_2$, then we can express the second query form above as

```
1. For each $\mathsf{rid}_\sigma \in \mathsf{MM}_\sigma[\mathbf{T}_1.\mathsf{att}_1 \| x]$,
    (a) For each $(\mathbf{T}_1.\mathsf{rid}, \mathbf{T}_2.\mathsf{rid}') \in \mathsf{MM}_\bowtie[\mathsf{rid}_\sigma \| \mathbf{T}_1.\mathsf{att}_2 \| \mathbf{T}_2.\mathsf{att}_2]$, output
            $\mathsf{MM}_\mathsf{rid}[\mathbf{T}_1.\mathsf{rid}] \| \mathsf{MM}_\mathsf{rid}[\mathbf{T}_2.\mathsf{rid}']$
```

This achieves the optimization effect with filter pushdown to the query complexity $O(sT^2)$.

Token trees. Another important step towards encrypted query optimization is the token representation. In SPX the tokens associated with the operators in a query is treated as a sequence. However, query optimziation roots in the partial ordering of operators. This semantics is typically captured by the query tree in relational settings. Therefore we introduce a new token representation, the *token tree*, that essentially mimics the the query tree to capture operator reordering.

The token tree and the new encoding of encrypted data structures for operator reordering is developed into a full scheme called OPX with full details in Chapter 4.

3.4 Emulation for Legacy Compliance

Legcay compliance means that our scheme can be implemented using the standard relational database without requiring internal modifications. This feature is particularaly welcomed for speeding up the adoption of the technology. However perhaps it is not obvious at the first glance that a sequence of EMM operations can be expressed as SQL. The key is to represent EMMs as tables, and EMM operations as SQL queries. We defer much syntatical detail to Chapter 5, but only explain the high level idea here.

Some EMM constructions such as [17] has dictionaries as underlying data structure. A dictionary is just a set of key-value pairs. Therefore we can use table of two attributes to store the dictionary. We have also seen in the previous section that a simple query can be written using a set-oriented language, which has analog in relational algebra. Therefore we can translate the operations for EMMs into a relational langauge.

For example, the query $\sigma_{\mathbf{T}.att=x}\mathbf{T}$ can be expressed algorithmically as

1. For each $\mathsf{rid} \in \mathsf{MM}_\sigma[\mathbf{T}.\mathsf{att}\|x]$,

    (a) Output $\mathsf{MM}_{\mathsf{rid}}[\mathsf{rid}]$

Or using set-oriented language,

$$\sigma_{\mathsf{att}=x}\mathbf{T} = \{r \mid \forall \mathsf{rid} \in \mathsf{MM}_\sigma[\mathbf{T}.\mathsf{att}\|x], \exists r \in \mathsf{MM}_{\mathsf{rid}}[\mathsf{rid}]\}$$

The set-oriented expression can then be turned into relational algebra, provided that we have boost the $\mathsf{EMM.Query}(\cdot)$ and $\mathsf{EMM.Token}(\cdot)$ functions into *operators* over a column of inputs

$$\mathsf{EMM.Token}(\begin{bmatrix}\mathsf{rid}_1\\ \mathsf{rid}_2\\ \vdots\end{bmatrix}) = \begin{bmatrix}\mathsf{tk}(\mathsf{rid}_1)\\ \mathsf{tk}(\mathsf{rid}_2)\\ \vdots\end{bmatrix}$$

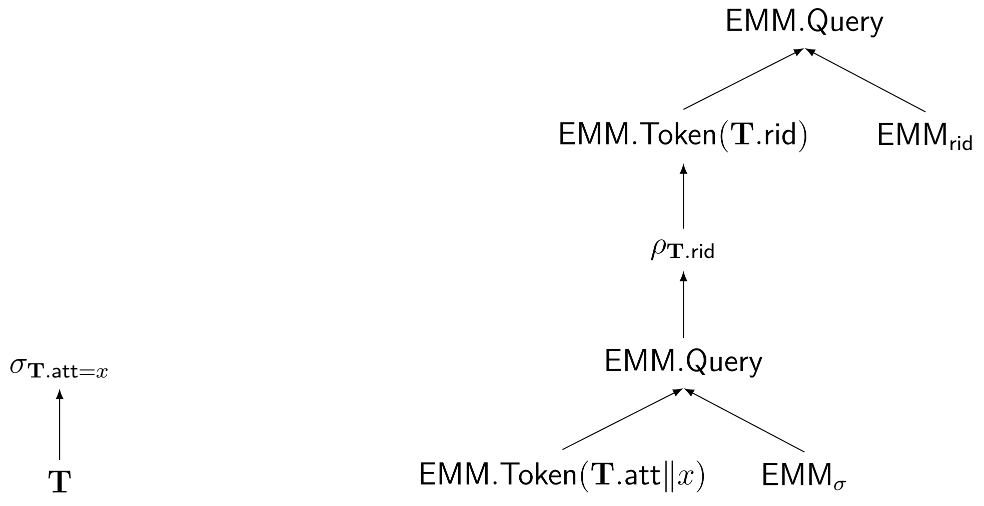
and

$$\mathsf{EMM.Query}(\begin{bmatrix}\mathsf{tk}(\mathsf{rid}_1)\\ \mathsf{tk}(\mathsf{rid}_2)\\ \vdots\end{bmatrix}, \mathsf{EMM}) = [\mathsf{EMM.Query}(\mathsf{tk}(\mathsf{rid}_1), \mathsf{EMM})\|\mathsf{EMM.Query}(\mathsf{tk}(\mathsf{rid}_1), \mathsf{EMM})\| \cdots]^T$$

where A^T denotes transpose of A. So the corresponding encrypted relational expression for the above example is shown in Figure 3.2.

Finally to obtain full legacy compliance, we need to further emulate the new operators $\mathsf{EMM.Query}$ and $\mathsf{EMM.Token}$ to obtain a SQL standard query. This process makes use of a SQL construct called common table expression.

We developed the system called `KafeDB` and the emulation for OPX first. More details is presented in Chapter 5.


$\sigma_{\mathbf{T}.\mathsf{att}=x}$
$\mathbf{T}$
EMM.Query
EMM.Token($\mathbf{T}$.rid)
$\mathsf{EMM}_{\mathsf{rid}}$
$\rho_{\mathbf{T}.\mathsf{rid}}$
EMM.Query
EMM.Token($\mathbf{T}$.att$\|x$)
EMM_{σ}

(a) Example query tree

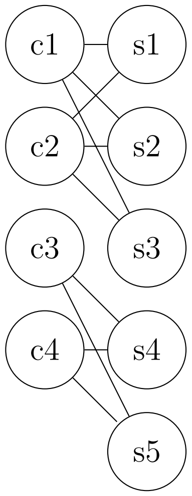
(b) Example token tree

Figure 3.2: Example query tree and token tree

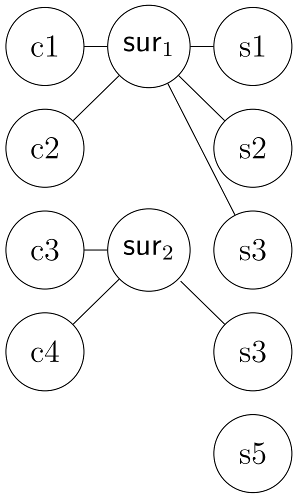
3.5 Optimal Efficiency

In the previous section we mentioned that a filtered join can only processed in $O(T + T^2)$ in SPX, which is improved by enabling query optimziation in OPX to $O(sT^2)$ for table size T and selctivity s in terms of query complexity. However, query input complexity can be further reduced to match the optimal plaintext processing such as hash join algorithm to be linear in input size $O(sT + T)$.

To achieve this optimal filtered join efficiency, we need to develope further insight into how join is represented in this indexed execution. In particular, we need to precompute and materialize all the join pairs of row identifiers into the multimap. This however has a lot of redundancy, because multiple rows from the same table may be joined with the same set of rows in the other table. For example in Figure 3.1, we can see that Abe and Bob from US are joined with the same set of suppliers, namely Intel, IBM and Apple from US. If we just naively store all these pairs, we end up with quadratic numbrer of such paris in the multimap. We visaully represent all these paris in a *join graph* in Figure 3.3.



(a) The join graph.



(b) The surrogate join graph.

Figure 3.3: Example of join $Customer \bowtie_{Nation} Supplier$.

How do we reduce the redundancy? We create a new set of *surrogate* nodes in the join graph to represent the sharing of the edges. For instance, Abe and Bob from US connect to the same surrogate, which is in turned connect to the same set of suppliers. It turns out that the total number of surrogates we need to insert into the join graph is equal to the number of unique values in the joined attribute. On the other hand, each customer and each supplierw would only need to connect to one surrogate. This reduces the number of edges to $O(T)$. We call the resulting graph the *surrogate join graph*. We then store the surrogate join graph in two multimaps, both only encode $O(T)$ pairs.

The surrogate join graph allows us to achieve three improvements over [39] simultaneously

- Optimal query complexity: $O(sT + T)$ (versus $O(T^2)$ in [39])

- Optimal space complexity: $O(T+T)$ (versus $O(T^2)$ in [39])
- Optimal setup complexity: $O(T)$ (versus $O(T^2)$ in [39])

for table size T (i.e. the number of rows)and selectivity s. We defer the full treatment of the topic in Chapter 6.

3.6 Leakage Reductnion for Filtered Joins

In SPX [40] a filtered join leaks the *full* join pattern. The full join pattern can be very revealing, because the adversary may be able to infer statistics such as the number of unique values in each joint attribute, and the frequency of each value.

For example in Figure 3.1, consider the filtered join

$$\sigma_{\text{C.Name=Bob}}\text{Customer} \bowtie_{\text{C.Nation=S.Nation}} \text{Supplier}$$

which can be expressed as the execution in SPX as

1. Client computes tokens and keys

   $\mathsf{tk}_\sigma \leftarrow \mathsf{EMM}_\sigma.\mathsf{Token}_{K_\sigma}(\text{C.Name}\|\text{Bob}), \quad \mathsf{tk}_\bowtie \leftarrow \mathsf{EMM}_\bowtie.\mathsf{Token}_{K_\bowtie}(\text{C.Nation}\|\text{S.Nation})$

2. Server computes query result

   (a) Compute the filter and join separately

   $$\mathbf{R}_1 \leftarrow \Big(\mathsf{EMM}_\sigma.\mathsf{Query}(\mathsf{tk}_\sigma)\Big), \quad \mathbf{R}_2 \leftarrow \Big(\mathsf{EMM}_\bowtie.\mathsf{Query}(\mathsf{tk}_\bowtie)\Big)$$

   (b) Correlate the filter and join

   $$\mathbf{R}_3 \leftarrow \Big\{(\mathsf{rtk},\mathsf{rtk}')\|\forall(\mathsf{rtk},\mathsf{rtk}') \in \mathbf{R}_2, \exists \mathsf{rtk} \in \mathbf{R}_1\Big\}$$

   (c) Compute the rows

   $$\mathbf{R}_4 \leftarrow \Big\{(\mathsf{EMM}_{\mathsf{rid}}.\mathsf{Query}(\mathsf{rtk}), \mathsf{EMM}_{\mathsf{rid}}.\mathsf{Query}(\mathsf{rtk}'))\|(\mathsf{rtk},\mathsf{rtk}') \in \mathbf{R}_3\Big\}$$

Notice that the intermediate results for filter predicate and join predicate $\mathbf{R}_1, \mathbf{R}_2$ are computed independently, where the join token $\mathsf{tk}_\bowtie$ is the same for the given joint attributes, regardless of the filter attributes. This means that the server reveals the full join pattern for the unfiltered join as well as any filtered joins for these joint attributes.

The OPX scheme that we proposed in this work reduces the comptuation of a filtered join from the need to computing the full join, but it still leaks the full join pattern for malicious adversary.

For the above example, the OPX expressess the execution as

1. Client computes tokens (same as SPX)
2. Server compute query result as
    (a) Compute the filter (same as SPX)
    (b) Compute the filtered join

$$\mathbf{R}_2 \leftarrow \left\{ \mathsf{EMM}_{\bowtie}.\mathsf{Query}(\mathsf{EMM}_{\bowtie}.\mathsf{Token}_{\mathsf{tk}_{\bowtie}}(\mathsf{rtk})) \mid \mathsf{rtk} \in \mathbf{R}_1 \right\}$$

    (c) Compute the rows (same as SPX)

The only difference bewteen OPX and SPX for this example is the way that filtered join is computed. Most noticeably, OPX allows *the server to compute the actual join token*, based on the client's join token. This new mechanism unfortunately leaks the full join pattern just like SPX, because the client join token $\mathsf{tk}_{\bowtie}$ is the same for any filtered join on the same joint attributes. For example, the above query leaks the joint pattern not only between Bob and his suppliers, but also between other customers such as Abe and Cay, because the (malicious) server can use the same client join token $\mathsf{tk}_{\bowtie}$ to reveal more joint pattern once any other *unrelated* query has revealed the other customer's row tokens rtk.

To reduce this leakage, we want the query leakge to be only dependent on the response, which in this case the *filtered* join pattern. To this end, we develope a new encoding in the EMM for joins. We show how this encoding works in the same example

1. Client computes the same tokens as OPX, except for join token

$$\mathsf{tk}_{\bowtie} \leftarrow \mathsf{EMM}_{\bowtie}.\mathsf{Token}_{K_{\bowtie}}(\text{C.Name}\|\text{US}\|\text{C.Nation}\|\text{S.Nation})$$

2. Server compute query result (same as OPX)

Here the new encoding essentially makes the client's join token depends on the filter value. This ensures that the server will only be able to compute actual join tokens based on this client's token, which limits the leakage of joint patterns that are not part of the filter.

The detail construction is called `pkfk`, and we present the details and the evaluation in Ch. 6.

3.7 Collocation

Chapter 4

Encrypted Query Optimization

4.1 Overview

In this section we outline the problem of encrypted query optimization and the solution. We will forgo much of the formalism and details and focus instead on the conceptual approachn and intuition.

Query optimization has been an important topic in relational database research since the inception of the field. Relational queries are rooted in the relational algebra, which admits multiple semantically equivalent, yet not equally efficient forms. This means that query optimizataion has to take place before the query execution to try to look for a more efficient query structure in order to reduce the execution time.

Query optimization tpyically relies on both the structure of the query and the statistics of the data. In a PPE-based encrypted database, the statistics may be available via the leakage, such as the ordering (i.e. whether the data are sorted) of the frequency (i.e. how many unique values and their frequencies), which may still be useful for a query optimizer. But for a STE-based scheme, such statsitics are not leakaed from the data at setup time, so we assume that the ideal encrypted query optimizer can only optimize queries based on the query structure alone. In the furture work, we will look into how to incorporate statistics from the past queries or from the data into a STE-based scheme.

Curiously, the first STE-based approach, SPX, only admits rather limited query optimization. In particular, the arguablely most widely used optimization rules, the `filter pushdown`, `filter reordering` and `join reordering`, are not possible to implement in SPX, leaving much efficiency enhancement off the table (the pun intended). So our most pressing need is to design an STE-based scheme that allows for `filter pushdown` and `join/filter reordering`.

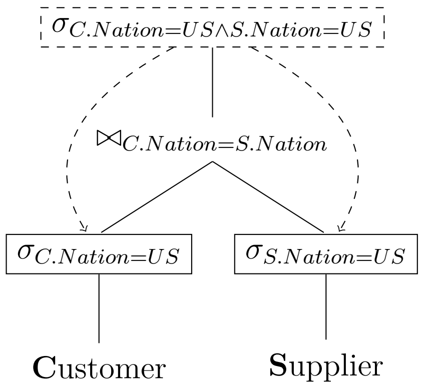
First, we review how `filter pushdown` and `join/filter reordering` work. Then we explain at a conceptual level why SPX cannot support these rules. Finally we introduce two new techniques that will enable these rules without compromising the security or even with better security in one case.

Id	Name	Nation
c1	Abe	US
c2	Bob	US
c3	Cay	Canada
c4	Bob	Canada

(a) Customer

Id	Name	Nation
s1	Intel	US
s2	IBM	US
s3	RIM	Canada

(b) Supplier



$\sigma_{C.Nation=US \wedge S.Nation=US}$
$\bowtie_{C.Nation=S.Nation}$
$\sigma_{C.Nation=US}$
$\sigma_{S.Nation=US}$
Customer
Supplier

(c) Query tree with `filter pushdown` applied.

C.Id	S.Id	C.Name	S.Name	Nation
c1	s1	Abe	Intel	US
c2	s1	Bob	Intel	US
c1	s2	Abe	IBM	US
c2	s2	Bob	IBM	US

(d) Query result.

Figure 4.1: Example of two tables and a filtered join: Customer and Supplier on Nation and filtered by US.

4.1.1 Optimization Rules

In general, query optimization follows the principle of cutting down intermediate data size for operators towards the bottom of the query tree, so that the upper level operators have smaller inputs. The `filter pushdown` rule says that if the query has a filter and a join that are *correlated*, or they touch on the same relation, then the filter can be evaluated first before the join. This can be visually represented as a *query tree*. We show an exmaple of a filtered join with said rule applied in Figure 4.1. Suppose we joined the two tables first and then applied the filter. This strategy would result in a full join between Customer and Supplier first, which involve potentially large number of customers and suppliers that would be filtered out later. Therefore pushing the filter before the join would limit the input to the join by a potentially small fraction of the original tables. This means a worst-case quadratic operation like JOIN can benefit from a potentially large constant factor reduction at query complexity.

The other two rules, the `join/filter reordering` follow this very principle. When we consider complex conjunctive queries that involve multiple filters and joins, it is often benefitial to process the "smaller" (or lower selectivity) filters or joins first, in order to avoid a larger input to the subsequent operators. The benefit for these two rules are the same that they introduce a potentially large constant reduction to the query complexity.

4.1.2 The Problem of the SPX Scheme

The prior state-of-the-art STE-based scheme SPX [39] does not support the query optimization rules. As such SPX leaves a potentially large constant gap in query complexity when compared to the plaintext exectuion. The root of the problem is that SPX encrypts for each relational operator on each attribute "independently", meaning that we have to execute each operator by itself, without considering the output of the other operators, and then in the end combine all results together using a multi-way join. This approach is incompatible with the query optimization where operators may dependend each other's outputs.

Filtered Joins. We show a conceptual picture of SPX in Figure 4.2 where we only focus on the functional aspect. In particular, when viewed as functions, SPX has mainly two encrypted multimaps, one that maps each attribute value to a list of priamry keys, or Ids, and one that maps the string of two attribute names to a list of Id pairs. The former indexes a filter, and the latter indexes a join. Notice how it is impossible to use output of a filter to the input of a join: the EMM_σ outputs a list of Ids for the filter predicate $C.Name = Nation$, but then this list of Ids cannot be used to lookup the associated joins, because the $\mathsf{EMM}_\bowtie$ only accepts the join attribute names and outputs the full join results for the Ids. So we have to independnetly search $\mathsf{EMM}_\bowtie$ to retrieve all the Id pairs, though some of them such as $(c_3, s_3), (c_4, s_4)$ do not satisfy the filter.

Conjunctive Filters. Another issue with SPX is that the conjunctive filters leak the value frequencies and row collocations *for each filter*. For example, if our query is

```
SELECT * FROM Customer
WHERE Nation = US and Name = Abe
```

Here SPX essentially treates the predicate $Nation = US$ and $Name = Bob$ separately, and will retrieve the resulting Ids associated with each predicate, though only the intersection of the Ids is needed for the result. This approach incurs a multiplicative factor in number of filters in the query complexity, because each filter would in the worst-case computes a result size equal to the row cardinality of the table, and so total is $O(q\dot{\mathsf{DB}})$ for q conjunctive filters. On the other hand, the order of the filters may matter, if we again draw inspiration from the plaintext execution. In the example, $Name = Abe$ only has selectivity 1, meaning only one row matches the predicate, but the other predicate on $Nation$ has selectivity 2. This means that if we were to filter on $Name = Abe$ first, we would end up with just one row c_1 for *Abe*, and we just need to check its $Nation$ attribute and see if it is US. This approach would reduce the processing to the $O(q \cdot \min_q s_q \mathsf{DB}))$ for q conjunctive filters and each with selectivity s_q.

Another downside of the SPX approach is the security: the leakage will be the union of the leakage of each filter. A better leakage proflie would be the leakage of just one filter and its intersection with other filters.

$$\mathsf{EMM}_\sigma : \mathbf{T}.\mathsf{att} \rightarrow (Id, \cdots), \quad \mathsf{EMM}_{\bowtie} : \{\text{“}\mathsf{att}_1, \mathsf{att}_2''\} \rightarrow (\mathbf{T}_1.Id, \mathbf{T}_2.Id)$$

(a) The domain and range interpretation of the encrypted multimaps for filters and joins.

$$\mathsf{EMM}_\sigma \begin{pmatrix} C.Nation\|US & \rightarrow & (c_1, c_2) \\ C.Nation\|Canada & \rightarrow & (c_3, c_4) \end{pmatrix}, \quad \mathsf{EMM}_\sigma \begin{pmatrix} S.Nation\|US & \rightarrow & (s_1, s_2) \\ S.Nation\|Canada & \rightarrow & (s_3) \end{pmatrix},$$
$$\mathsf{EMM}_\sigma \begin{pmatrix} C.Name\|Abe & \rightarrow & (c_1) \\ C.Name\|Bob & \rightarrow & (c_2, c_4) \\ C.Name\|Cay & \rightarrow & (c_3) \end{pmatrix}, \quad \mathsf{EMM}_\sigma \begin{pmatrix} S.Name\|Intel & \rightarrow & (s_1) \\ S.Name\|IBM & \rightarrow & (s_2) \\ S.Name\|RIM & \rightarrow & (s_3) \end{pmatrix},$$
$$\mathsf{EMM}_{\bowtie} \left(\text{“}C.Nation, S.Nation'' \quad \rightarrow \quad ((c_1, s_1), (c_2, s_1), (c_1, s_2), (c_2, s_2), (c_3, s_3), (c_4, s_4))\right)$$

(b) Example application for the example in Fig. 4.1

Figure 4.2: The conceptual picture for the SPX [39] scheme. SPX treats each relational operator independently and cannot support query optimziation rules.

4.1.3 Solution Sketch

In order to support query optimization, we need to capture the dependency amongst query operators in the encrypted data structures or the EMMs. We need to change how the encrypted multimaps are defined functionally.

Looking back at Figure 4.2, we notice that the key for adding dependency is to add Ids to the domain of the encrypted multimaps for joins, $\mathsf{EMM}_{\bowtie}$

$$\mathsf{EMM}_{\bowtie} : (\{\text{“}\mathsf{att}_1, \mathsf{att}_2''\}, \mathbf{Id}) \rightarrow Id$$

which we apply conceptually to the example as

$$\mathsf{EMM}_{\bowtie} \begin{pmatrix} \text{“}C.Nation, S.Nation'', \mathbf{c1} & \rightarrow & s_1 \\ \text{“}C.Nation, S.Nation'', \mathbf{c2} & \rightarrow & s_1 \\ \text{“}C.Nation, S.Nation'', \mathbf{c1} & \rightarrow & s_2 \\ \text{“}C.Nation, S.Nation'', \mathbf{c2} & \rightarrow & s_2 \\ \text{“}C.Nation, S.Nation'', \mathbf{c3} & \rightarrow & s_3 \\ \text{“}C.Nation, S.Nation'', \mathbf{c4} & \rightarrow & s_3 \end{pmatrix}$$

This way, when we obtain the results of the filter from EMM_σ, say

$$c_1, c_2$$

Then we can use this output as an input to the $\mathsf{EMM}_{\bowtie}$ to compute the join that are only associated with c_1, c_2, which have been filtered.

Post-join Filters. For post-join filters, such as the filter $\sigma_{S.Nation=US}S$ on the right subtree in the example of Figure 4.1, the situation is different than the pre-join filter like above. Essentially, after a join, we already have join pairs of Ids $(c_i, s_j)_k$, now we just want to filter out the pairs whose second element s_j satisfy the post-join filter predicate. This cannot be done by the EMM_σ above because functionally we are given the Id s_j and want to test the *existence*, i.e. whether this Id s_j "exists" in the filtered result. For this formulation of post-join filter as existence or membership test, we just need to implement a set

$$\mathsf{SET}_\exists : (Id, \mathsf{att})$$

which we apply conceptually to the example as

$$\mathsf{SET}_\exists \begin{pmatrix} (s_1, US) \\ (s_2, US) \\ (s_3, Canada) \end{pmatrix}$$

Then we can filter the result of the previously executed join by the following: for each s_j in $(c_i, s_j)_k$, check if (s_j, US) is in the $\mathsf{SET}_\exists$, if not, remove the pair. The retained pairs are the filtered join results.

In terms of security, we want this encrypted set $\mathsf{SET}_\exists$ to not leak the filter value US. We will present the cryptographic details in late sections.

4.2 The OPX Scheme

We describe the OPX scheme which extends the SPX construction of [39]. It uses as building blocks a response-revealing multi-map encryption scheme Σ_{MM}, a variant of the Pibase construction of Cash et al. [17] we denote Σ^π_{MM} and a pseudo-random function F. In Appendix 4.3, we provide a concrete example that walks through our indexing approach.

Variant of Pibase. As one of our building blocks, we need a multi-map encryption scheme that achieves a slightly stronger variant of adaptive security. More precisely, it needs to achieve a sort of ”key equivocation” by which mean that a simulator should be able to output a simulated encrypted multi-map and simulated tokens and, at a later time, produce a key that is indistinguishable from a real key even to an adversary that holds the encrypted multi-map and tokens. This can be achieved by simply instantiating the PRF in Pibase with a random oracle and programming it appropriately during simulation.

Setup. The Setup algorithm takes as input a database $\mathsf{DB} = (\mathbf{T}_1, \cdots, \mathbf{T}_n)$ and a security parameter k. It first samples a key $K_1 \xleftarrow{\$} \{0,1\}^k$, and then initializes a multi-map MM_R such that for all rows $\mathbf{r} \in \mathsf{DB}$, it sets

$$\mathsf{MM}_R\Big[\chi(\mathbf{r})\Big] := \Big(\mathsf{Enc}_{K_1}(r_1), \cdots, \mathsf{Enc}_{K_1}(r_{\#\mathbf{r}}), \chi(\mathbf{r})\Big),$$

It then computes

$$(K_R, \mathsf{EMM}_R) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\Big(1^k, \mathsf{MM}_R\Big).$$

It initializes a multi-map MM_C such that for all columns $\mathbf{c} \in \mathsf{DB}^\intercal$, it sets

$$\mathsf{MM}_C\Big[\chi(\mathbf{c})\Big] := \Big(\mathsf{Enc}_{K_1}(c_1), \cdots, \mathsf{Enc}_{K_1}(c_{\#\mathbf{c}}), \chi(\mathbf{c})\Big),$$

It then computes

$$(K_C, \mathsf{EMM}_C) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\Big(1^k, \mathsf{MM}_C\Big).$$

It initializes a multi-map MM_V, and for each $\mathbf{c} \in \mathsf{DB}^\intercal$, all $v \in \mathbf{c}$ and $\mathbf{r} \in \mathsf{DB}_{\mathbf{c}=v}$, it computes

$$\mathsf{rtk}_\mathbf{r} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_R, \chi(\mathbf{r})\Big),$$

and sets

$$\mathsf{MM}_V\Big[\Big\langle v, \chi(\mathbf{c})\Big\rangle\Big] := \Big(\mathsf{rtk}_\mathbf{r}\Big)_{\mathbf{r}\in\mathsf{DB}_{\mathbf{c}=v}}.$$

It then computes

$$(K_V, \mathsf{EMM}_V) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}(1^k, \mathsf{MM}_V).$$

It initializes a set of multi-maps $\{\mathsf{MM}_\mathbf{c}\}_{\mathbf{c}\in\mathsf{DB}^\intercal}$. For all columns $\mathbf{c}, \mathbf{c}' \in \mathsf{DB}^\intercal$ that have the same domain such that $\mathsf{dom}(\mathsf{att}(\mathbf{c})) = \mathsf{dom}(\mathsf{att}(\mathbf{c}'))$, it initiates an empty tuple $\mathbf{t}$ that it populates as follows. For all rows $\mathbf{r}_i$ and $\mathbf{r}_j$ in column $\mathbf{c}$ and $\mathbf{c}'$, respectively, that verify

$$\mathbf{c}[i] = \mathbf{c}'[j],$$

it inserts $(\mathsf{rtk}_i, \mathsf{rtk}_j)$ in $\mathbf{t}$ where

$$\mathsf{rtk}_i \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}_i))$$

and

$$\mathsf{rtk}_j \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}_i j)),$$

and sets

$$\mathsf{MM}_\mathbf{c}\Big[\langle\chi(\mathbf{c}), \chi(\mathbf{c}')\rangle\Big] := \mathbf{t}.$$

It then computes, for all $\mathbf{c} \in \mathsf{DB}^\intercal$,

$$(K_\mathbf{c}, \mathsf{EMM}_\mathbf{c}) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\Big(1^k, \mathsf{MM}_\mathbf{c}\Big).$$

It initializes a set SET and computes for each column $\mathbf{c} \in \mathsf{DB}^\intercal$, and for all $v \in \mathbf{c}$, a key K_v such that

$$K_v \leftarrow F_{K_F}(\chi(\mathbf{c})\|v),$$

where $K_F \xleftarrow{\$} \{0,1\}^k$. Then for all rows $\mathbf{r}$ in $\mathsf{DB}_{\mathbf{c}=v}$, it sets

$$\mathsf{SET} := \mathsf{SET} \bigcup \left\{ F_{K_v}(\mathsf{rtk}) \right\},$$

where $\mathsf{rtk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}))$. It then initializes a set of multi-maps $\{\mathsf{MM}_{\mathsf{att},\mathsf{att}'}\}$ for $\mathsf{att}, \mathsf{att}' \in \mathbb{S}(\mathsf{DB})$ and $\mathsf{dom}(\mathsf{att}) = \mathsf{dom}(\mathsf{att}')$. For all columns $\mathbf{c}, \mathbf{c}' \in \mathsf{DB}^{\intercal}$ that have the same domain, it initiates an empty tuple $\mathbf{t}$ that it populates as follows. For all rows $\mathbf{r}_i$ and $\mathbf{r}_j$ in column $\mathbf{c}$ and $\mathbf{c}'$, respectively, that verify

$$\mathbf{c}[i] = \mathbf{c}'[j],$$

it inserts $(\mathsf{rtk}_i, \mathsf{rtk}_j)$ in $\mathbf{t}$ where

$$\mathsf{rtk}_i \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}_i))$$

and

$$\mathsf{rtk}_j \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}_i j)).$$

Then for all rtk such that $(\mathsf{rtk}, \cdot) \in \mathbf{t}$, it sets

$$\mathsf{MM}_{\mathbf{c},\mathbf{c}'}\left[\mathsf{rtk}\right] := \left(\mathsf{rtk}'\right)_{(\mathsf{rtk},\mathsf{rtk}')\in\mathbf{t}}$$

then computes

$$(K_{\mathbf{c},\mathbf{c}'}, \mathsf{EMM}_{\mathbf{c},\mathbf{c}'}) \leftarrow \Sigma^{\pi}_{\mathsf{MM}}.\mathsf{Setup}\left(1^k, \mathsf{MM}_{\mathbf{c},\mathbf{c}'}\right).$$

Finally, it outputs a key $K = (K_1, K_R, K_C, K_V, \{K_{\mathbf{c}}\}_{\mathbf{c}\in\mathsf{DB}^{\intercal}}, K_F, \{K_{\mathbf{c},\mathbf{c}'}\}_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^{\intercal}})$ and $\mathsf{EDB} = (\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, (\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^{\intercal}}, \mathsf{SET}, (\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c}\in\mathsf{DB}^{\intercal}})$.

Token. The Token algorithm takes as input a key K and a query tree QT and outputs a token tree TT.[1] The token tree is a copy of QT and first initialized with empty nodes. The algorithm performs a post-order traversal of the query tree and, for every visited node N, does the following:

- **(leaf select)** if N is a leaf node of form $\sigma_{\mathsf{att}=a}(\mathbf{T})$ then set the corresponding node in TT to
$$\mathsf{stk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\left(K_V, \langle a, \chi(\mathsf{att})\right).$$

[1]Every query in the SPC algebra can be represented as a query tree QT which is a a tree-based representation of the query. A query can have several query tree representations each leading to a different query complexity when executed.

- **(internal constant select):** if N is an internal node of form $\sigma_{\mathsf{att}=a}(\mathbf{R_{in}})$ then set the corresponding node in TT to $(\mathsf{rtk}, \mathsf{pos})$ where

$$\mathsf{rtk} \leftarrow F_{K_F}\Big(\chi(\mathsf{att}) \| a\Big),$$

and pos denotes the position of att in $\mathbf{R_{in}}$.

- **(leaf join):** if N is a leaf node of form $\mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{T}_2$ then set the corresponding node in TT to $(\mathsf{jtk}, \mathsf{pos})$ where

$$\mathsf{jtk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_{\mathsf{att}_1}, \Big\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2)\Big\rangle\Big),$$

and pos denotes the positions of att_1 in $\mathbf{R_{in}}$.

- **(internal join):** if N is an internal node of form $\mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R_{in}}$, then set the corresponding node in TT to $(\mathsf{etk}, \mathsf{pos}_1, \mathsf{pos}_2)$ where

$$\mathsf{etk} := K_{\mathsf{att}_1,\mathsf{att}_2},$$

and pos_1, pos_2 denote the positions of att_1 and att_2 in $\mathbf{R_{in}}$, respectively.

- **(intermediate internal join):** if N is an internal node of form $\mathbf{R}^{(l)}_{\mathbf{in}} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R}^{(r)}_{\mathbf{in}}$ then set the corresponding node in TT to $(\mathsf{pos}_1, \mathsf{pos}_2)$ where pos_1 and pos_2 are the column positions of att_1 and att_2 in $\mathbf{R}^{(l)}_{\mathbf{in}}$ and $\mathbf{R}^{(r)}_{\mathbf{in}}$, respectively.

- **(leaf projection):** if N is a leaf node of form $\pi_{\mathsf{att}}(\mathbf{T})$ then set the corresponding node to ptk where

$$\mathsf{ptk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_C, \chi(\mathsf{att}_i)\Big).$$

- **(internal projection):** if N is an internal node of form $\pi_{\mathsf{att}_1,\cdots,\mathsf{att}_z}(\mathbf{R_{in}})$ then set the corresponding node to

$$\Big(\mathsf{pos}_1, \cdots, \mathsf{pos}_z\Big),$$

where pos_i is the column position of att_i in $\mathbf{R_{in}}$.

- **(leaf scalars):** if N is a node of form $[a]$ then set the corresponding node to $[\mathsf{Enc}_{K_1}(a)]$.

- **(cross product):** if N is a node of form $\times$ then keep it with no changes.

Query. The algorithm takes as input the encrypted database EDB and the token tree TT. It performs a post-order traversal of tk and, for each visited node N, does the following:

- **(leaf select):** if N has form stk, it computes

$$(\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}\Big(\mathsf{EMM}_V, \mathsf{stk}\Big),$$

  and sets $\mathbf{R_{out}} := (\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s)$.

- **(internal constant select):** if N has form $(\mathsf{rtk}, \mathsf{pos})$, then for all rtk in $\mathbf{R_{in}}$ in the column at position pos, if

$$F_{\mathsf{rtk}}(\mathsf{rtk}) \notin \mathsf{SET},$$

  then it removes the row from $\mathbf{R_{in}}$. Finally, it sets $\mathbf{R_{out}} := \mathbf{R_{in}}$.

- **(leaf join):** if N has form $(\mathsf{jtk}, \mathsf{pos})$, then it computes

$$\Big((\mathsf{rtk}_1, \mathsf{rtk}'_1), \ldots, (\mathsf{rtk}_s, \mathsf{rtk}'_s)\Big) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}(\mathsf{EMM}_{\mathsf{pos}}, \mathsf{jtk}),$$

  and sets

$$\mathbf{R_{out}} := \Big((\mathsf{rtk}_i, \mathsf{rtk}'_i)\Big)_{i \in [s]}.$$

- **(internal join):** if N has form $(\mathsf{etk}, \mathsf{pos}_1, \mathsf{pos}_2)$, then for each row $\mathbf{r}$ in $\mathbf{R_{in}}$, it computes $\mathsf{ltk} \leftarrow \Sigma^{\pi}_{\mathsf{MM}}.\mathsf{Token}(\mathsf{etk}, \mathsf{rtk})$, and

$$(\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s) \leftarrow \Sigma^{\pi}_{\mathsf{MM}}.\mathsf{Query}(\mathsf{EMM}_{\mathsf{pos}_1, \mathsf{pos}_2}, \mathsf{ltk}),$$

  where $\mathsf{rtk} = \mathbf{r}[\mathsf{att}_{\mathsf{pos}_2}]$, and appends the new rows

$$\Big(\mathsf{rtk}_i\Big)_{i \in [s]} \times \mathbf{r}$$

  to $\mathbf{R_{out}}$.

- **(intermediate internal join):** if N has form $(\mathsf{pos}_1, \mathsf{pos}_2)$, then it sets

$$\mathbf{R_{out}} := \mathbf{R}^{(l)}_{\mathbf{in}} \bowtie_{\mathsf{pos}_1 = \mathsf{pos}_2} \mathbf{R}^{(r)}_{\mathbf{in}}.$$

- **(leaf projection):** if N is a leaf node of form ptk then it computes

$$(\mathrm{ct}_1, \cdots, \mathrm{ct}_s) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}\Big(\mathsf{EMM}_C, \mathsf{ptk}\Big),$$

  and sets $\mathbf{R_{out}} := (\mathrm{ct}_1, \cdots, \mathrm{ct}_s)$.

ID	Name	Course
A05	Alice	16
A12	Bob	18
A03	Eve	18

Course	Department
16	CS
18	Math

Figure 4.3: Plaintext database DB.

- **(internal projection):** if N is an internal node of form $(\mathtt{pos}_1, \cdots, \mathtt{pos}_z)$, then it computes
$$\mathbf{R_{out}} := \pi_{\mathsf{pos}_1,\cdots,\mathsf{pos}_z}(\mathbf{R_{in}}).$$
- **(cross product):** if N is a node of form $\times$ then it computes
$$\mathbf{R_{out}} := \mathbf{R}^{(l)}_{\mathbf{in}} \times \mathbf{R}^{(r)}_{\mathbf{in}},$$
where $\mathbf{R}^{(l)}_{\mathbf{in}}$ and $\mathbf{R}^{(r)}_{\mathbf{in}}$ are the left and right input respectively.

Now, it replaces each cell rtk in $\mathbf{R}^{\mathsf{root}}_{\mathbf{out}}$ by
$$\mathrm{ct} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}(\mathsf{EMM}_R, \mathsf{rtk}).$$

4.3 A Concrete Example of Indexed Execution

Similar to [39], our examples also rely on a small database DB composed of two tables $\mathbf{T}_1$ and $\mathbf{T}_2$ that have three and two rows, respectively. The schema of $\mathbf{T}_1$ is $\mathbb{S}(\mathbf{T}_1) =$ (ID, Name, Course) and that of $\mathbf{T}_2$ is $\mathbb{S}(\mathbf{T}_2) =$ (Course, $\mathsf{Department}$). The tables are described in Figure (4.3).

Figure (4.4) shows the result of applying our method to index the database $\mathsf{DB} = (\mathbf{T}_1, \mathbf{T}_2)$, as detailed in Section (4.2). There are five multi-maps MM_R, MM_C, MM_V, $\mathsf{MM}_{\mathsf{Course}}$, $\mathsf{MM}_{\mathsf{T}_2.\mathsf{Course},\mathsf{T}_1.\mathsf{Course}}$, and a set SET. e detail below how the indexing works for this example.

The first multi-map, MM_R, maps every row in each table to its encrypted content. As an instance, the first row of $\mathbf{T}_1$ is composed of three values $(A05, \text{Alice}, 16)$ that will get encrypted and stored in MM_R. Since DB has five rows, MM_R has five pairs. The second multi-map, MM_C, maps each column of every table to its encrypted content. Similarly, as DB is composed of five columns in total, MM_C has five pairs. The third multi-map, MM_V, maps every unique value in every table to its coordinates in the plaintext table. For example, the value 18 in $\mathbf{T}_1$ exists in two positions, in particular, in the second and third row. The join multi-map, $\mathsf{MM}_{\mathsf{Course}}$, maps the columns' coordinates to the pair of rows that have the same value. In our example, as the first row of both tables contains 16, and the second and third rows of $\mathbf{T}_1$ and the second row of $\mathbf{T}_2$ contain 18, the label/tuple pair

$$\bigg(\mathbf{T}_1 \| \mathbf{c}_3 \| \mathbf{T}_2 \| \mathbf{c}_1, \Big((\mathbf{T}_1 \| r_1, \mathbf{T}_2 \| r_1), (\mathbf{T}_1 \| r_2, \mathbf{T}_2 \| r_2)\Big), (\mathbf{T}_1 \| r_3, \mathbf{T}_2 \| r_2)\Big)\bigg)$$

MM_R	
$\mathsf{T}_1\|r_1$	$\mathsf{Enc}_K(\text{A05})$, $\mathsf{Enc}_K(\text{Alice})$, $\mathsf{Enc}_K(16)$
$\mathsf{T}_1\|r_2$	$\mathsf{Enc}_K(\text{A12})$, $\mathsf{Enc}_K(\text{Bob})$, $\mathsf{Enc}_K(18)$
$\mathsf{T}_1\|r_3$	$\mathsf{Enc}_K(\text{A03})$, $\mathsf{Enc}_K(\text{Eve})$, $\mathsf{Enc}_K(18)$
$\mathsf{T}_2\|r_1$	$\mathsf{Enc}_K(16)$, $\mathsf{Enc}_K(\text{CS})$
$\mathsf{T}_2\|r_2$	$\mathsf{Enc}_K(18)$, $\mathsf{Enc}_K(\text{Math})$

MM_C	
$\mathsf{T}_1\|c_1$	$\mathsf{Enc}_K(\text{A05})$, $\mathsf{Enc}_K(\text{A12})$, $\mathsf{Enc}_K(\text{A03})$
$\mathsf{T}_1\|c_2$	$\mathsf{Enc}_K(\text{Alice})$, $\mathsf{Enc}_K(\text{Bob})$, $\mathsf{Enc}_K(\text{Eve})$
$\mathsf{T}_1\|c_3$	$\mathsf{Enc}_K(16)$, $\mathsf{Enc}_K(18)$, $\mathsf{Enc}_K(18)$
$\mathsf{T}_2\|c_1$	$\mathsf{Enc}_K(16)$, $\mathsf{Enc}_K(18)$
$\mathsf{T}_2\|c_2$	$\mathsf{Enc}_K(\text{CS})$, $\mathsf{Enc}_K(\text{Math})$

$\mathsf{MM}_{\mathsf{Course}}$	
$\mathsf{T}_1\|c_3\|\mathsf{T}_2\|c_1$	$(\mathsf{T}_1\|r_1, \mathsf{T}_2\|r_1)$, $(\mathsf{T}_1\|r_2, \mathsf{T}_2\|r_2)$, $(\mathsf{T}_1\|r_3, \mathsf{T}_2\|r_2)$

$\mathsf{MM}_{\mathsf{T}_2.\mathsf{Course},\mathsf{T}_1.\mathsf{Course}}$	
$\mathsf{T}_2\|r_1$	(T_1, r_1)
$\mathsf{T}_2\|r_2$	(T_1, r_2), (T_1, r_3)

MM_V	
$\mathsf{T}_1\|c_1\|\text{A05}$	T_1, r_1
$\mathsf{T}_1\|c_1\|\text{A12}$	T_1, r_2
$\mathsf{T}_1\|c_1\|\text{A03}$	T_1, r_3
$\mathsf{T}_1\|c_2\|\text{Alice}$	T_1, r_1
$\mathsf{T}_1\|c_2\|\text{Bob}$	T_1, r_2
$\mathsf{T}_1\|c_2\|\text{Eve}$	T_1, r_3
$\mathsf{T}_1\|c_3\|16$	T_1, r_1
$\mathsf{T}_1\|c_3\|18$	$(\mathsf{T}_1, r_2), (\mathsf{T}_1, r_3)$
$\mathsf{T}_2\|c_1\|16$	T_2, r_1
$\mathsf{T}_2\|c_1\|18$	T_2, r_2
$\mathsf{T}_2\|c_2\|\text{CS}$	T_2, r_1
$\mathsf{T}_2\|c_2\|\text{Math}$	T_2, r_2

SET
$\mathsf{T}_1\|r_1\|c_1\|\text{A05}$
$\mathsf{T}_1\|r_2\|c_1\|\text{A12}$
$\mathsf{T}_1\|r_3\|c_1\|\text{A03}$
$\mathsf{T}_1\|r_1\|c_2\|\text{Alice}$
$\mathsf{T}_1\|r_2\|c_2\|\text{Bob}$
$\mathsf{T}_1\|r_3\|c_2\|\text{Eve}$
$\mathsf{T}_1\|r_1\|c_3\|16$
$\mathsf{T}_1\|r_2\|c_3\|18$
$\mathsf{T}_1\|r_3\|c_3\|18$
$\mathsf{T}_2\|r_1\|c_1\|16$
$\mathsf{T}_2\|r_2\|c_1\|18$
$\mathsf{T}_2\|r_1\|c_2\|\text{CS}$
$\mathsf{T}_2\|r_2\|c_2\|\text{Math}$

Figure 4.4: Indexed database.

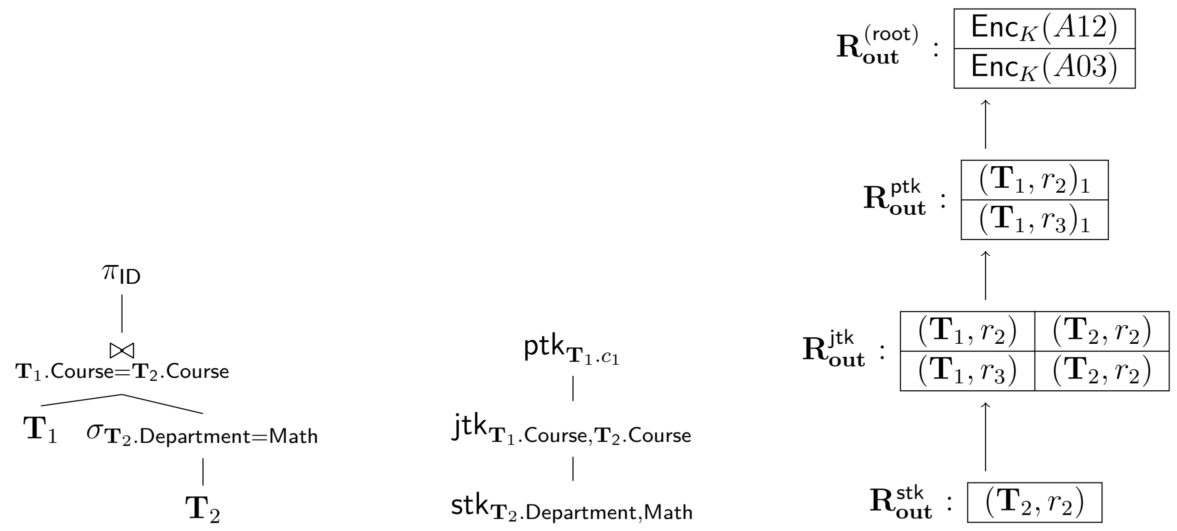
is added to $\mathsf{MM}_{\mathsf{Course}}$. The correlated join multi-map, $\mathsf{MM}_{\mathsf{T}_2.\mathsf{Course},\mathsf{T}_1.\mathsf{Course}}$, maps every row in each table to all rows that contain the same value. In our example, for the attribute Course, the first row in $\mathbf{T}_2$ maps to the first row in $\mathbf{T}_1$ while the second row in $\mathbf{T}_2$ maps to second and third rows in $\mathbf{T}_1$. Finally, the set structure SET stores all values in every row and every attribute.

A concrete query. Let us consider the following simple SQL query

$$\texttt{SELECT } \mathbf{T}_1.\text{ID} \texttt{ FROM } \mathbf{T}_1, \mathbf{T}_2 \texttt{ WHERE } \mathbf{T}_2.\text{Department} = \text{Math} \ \mathsf{AND}\ \mathbf{T}_2.\text{Course} = \mathbf{T}_1.\text{Course}.$$

This SQL query can be rewritten as a query tree, see Figure (**??**), and then translated, based on opx protocol into a token tree as depicted in Figure (4.5b).[2]

[2]For sake of clarity, this example of token tree generation does not accurately reflect the token protocol of opx, but only gives a high level idea of its algorithmic generation.


π_{ID}
$\bowtie_{\mathbf{T}_1.\mathsf{Course}=\mathbf{T}_2.\mathsf{Course}}$
$\mathbf{T}_1$
$\sigma_{\mathbf{T}_2.\mathsf{Department}=\mathsf{Math}}$
$\mathbf{T}_2$
$\mathsf{ptk}_{\mathbf{T}_1.c_1}$
$\mathsf{jtk}_{\mathbf{T}_1.\mathsf{Course},\mathbf{T}_2.\mathsf{Course}}$
$\mathsf{stk}_{\mathbf{T}_2.\mathsf{Department},\mathsf{Math}}$
$\mathbf{R}^{(\mathsf{root})}_{\mathbf{out}}$:
$\mathsf{Enc}_K(A12)$
$\mathsf{Enc}_K(A03)$
$\mathbf{R}^{\mathsf{ptk}}_{\mathbf{out}}$:
$(\mathbf{T}_1, r_2)_1$
$(\mathbf{T}_1, r_3)_1$
$\mathbf{R}^{\mathsf{jtk}}_{\mathbf{out}}$:
$(\mathbf{T}_1, r_2)$
$(\mathbf{T}_2, r_2)$
$(\mathbf{T}_1, r_3)$
$(\mathbf{T}_2, r_2)$
$\mathbf{R}^{\mathsf{stk}}_{\mathbf{out}}$:
$(\mathbf{T}_2, r_2)$

(a) Query tree (b) Token tree (c) Intermediate results of token tree

Figure 4.5: A query tree translated to a token tree which is then executed using the indexed database.

We detail in Figure (4.5c) the intermediary results of the token tree execution using the indexed database and provide below a high level description of how it works.

The server starts by fetching from MM_V the tuple corresponding to $\mathbf{T}_2 \| c_2 \| 18$, which is equal to $\{(\mathbf{T}_2, r_2)\}$. This represents the first intermediary output $\mathbf{R}^{\mathsf{stk}}_{\mathbf{out}}$ which is also the input for the next node. For each element in $\mathbf{R}^{\mathsf{stk}}_{\mathbf{out}}$, the server fetches the corresponding tuple in $\mathsf{MM}_{\mathbf{T}_2.\mathsf{Course},\mathbf{T}_1.\mathsf{Course}}$, which is equal to $\{(\mathbf{T}_1, r_2), (\mathbf{T}_1, r_3)\}$. Now, the second intermediary output $\mathbf{R}^{\mathsf{jtk}}_{\mathbf{out}}$ is composed of all row coordinates from $\mathbf{T}_1$ that match $\mathbf{T}_2$. For the internal projection node, given $(1, \mathsf{in})$, the server will simply output the row tokens in the first attribute as $\mathbf{R}^{\mathsf{ptk}}_{\mathbf{out}}$.

Finally, the server fetches tuples from the MM_R that correspond to the remaining row tokens, as the final result of $\mathbf{R}^{\mathsf{root}}_{\mathbf{out}}$, which is equal to

$$\mathbf{R}^{\mathsf{root}}_{\mathbf{out}} = (\mathsf{Enc}_K(A12), \mathsf{Enc}_K(A03)).$$

Concrete storage overhead. The plaintext database DB is composed of thirteen cells excluding the tables attributes.[3] The indexed structure consists of fifty eight pairs. Assuming that a pair and a cell have the same bit length, our indexed representation of the database has a multiplicative storage overhead of 4.46. In particular, each of the multi-maps MM_R, MM_C, MM_V and the set SET have the same size as the plaintext database (i.e., 13 pairs). This explains the $4\times$ factor. It is worth emphasizing that even if one considers a larger

[3]Note that our calculation does not take into account the security parameter and consider every (encrypted) cell as a one unit of storage.

database, the 4× factor remains unchanged. The additive component of the multiplicative factor, i.e., the 0.46, will vary, however, from one database to another depending on the number of columns with the same domain and the number of equal rows in these columns.

4.3.1 Efficiency

We now turn to analyzing the search and storage efficiency of our construction.

Query complexity. Given a potentially optimized query tree QT of an SPC query, we show that the search complexity of opx is asymptotically optimal.

Theorem 4.3.1 *[] If Σ_{mm} is optimal, then the time and space complexity of the* Query *algorithm presented in Section (**??**) is optimal.*

The proof of the theorem is in Appendix A.1.

Storage complexity. The storage complexity of OPX is similar to that of SPX asymptotically, but is larger concretely. This is because OPX needs two additional encrypted structures: a collection of encrypted multi-maps $(\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^\intercal}$) and an encrypted set SET.

For a database $\mathsf{DB} = (\mathbf{T}_1, \ldots, \mathbf{T}_n)$, opx produces three encrypted multi-maps EMM_R, EMM_C, EMM_V, two collections of encrypted multi-maps $(\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^\intercal}$ and $(\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c}\in\mathsf{DB}^\intercal}$, and a set structure SET. For ease of exposition, we assume that each table is composed of m rows. Also, note that standard multi-map encryption schemes [26, 21, 42, 18, 17] produce encrypted structures with storage overhead that is linear in the sum of the tuple sizes. Using such a scheme as the underlying multi-map encryption scheme, we have that EMM_R and EMM_C are $O(\sum_{\mathbf{r}\in\mathsf{DB}} \#\mathbf{r})$ and $O\Big(\sum_{\mathbf{c}\in\mathsf{DB}^\intercal} \#\mathbf{c}\Big)$, respectively, since the former maps the coordinates of each row in DB to their (encrypted) row and the latter maps the coordinates of very column to their (encrypted) columns. Since EMM_V maps each cell in DB to tokens for the rows that contain the same value, it requires $O\Big(\sum_{\mathbf{c}\in\mathsf{DB}^\intercal}\sum_{v\in\mathbf{c}} \#\mathsf{DB}_{\mathsf{att}(\mathbf{c})=v}\Big)$ storage. Similarly, SET contains the pseudo- random evaluation of the coordinate of all rows in the database and therefore requires $O\Big(\sum_{\mathbf{c}\in\mathsf{DB}^\intercal}\sum_{v\in\mathbf{c}} \#\mathsf{DB}_{\mathsf{att}(\mathbf{c})=v}\Big)$. For each $\mathbf{c} \in \mathsf{DB}^\intercal$, an encrypted multi-map $\mathsf{EMM}_{\mathbf{c}}$ maps each pair of form $(\mathbf{c}, \mathbf{c}')$ such that $\mathsf{dom}(\mathsf{att}(\mathbf{c})) = \mathsf{dom}(\mathsf{att}(\mathbf{c}'))$ to a tuple of tokens for rows in $\mathsf{DB}_{\mathsf{att}(\mathbf{c})=\mathsf{att}(\mathbf{c}')}$. As such, the collection $(\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c}\in\mathsf{DB}^\intercal}$ has size

$$O\Bigg(\sum_{\mathbf{c}\in\mathsf{DB}^\intercal}\ \sum_{\mathbf{c}':\mathsf{dom}(\mathsf{att}(\mathbf{c}'))=\mathsf{dom}(\mathsf{att}(\mathbf{c}))} \#\mathsf{DB}_{\mathsf{att}(\mathbf{c})=\mathsf{att}(\mathbf{c}')}\Bigg).$$

Similarly, for all $\mathbf{c}, \mathbf{c}' \in \mathsf{DB}^\intercal$, an encrypted multi-map $\mathsf{EMM}_{\mathsf{att},\mathsf{att}'}$ maps the coordinate of each row $\mathbf{r}$ in the column att to all the coordinates of rows $\mathbf{r}'$ in att' that have the same value such that $\mathbf{r}[\mathsf{att}] = \mathbf{r}'[\mathsf{att}']$. The size of $(\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^\intercal}$ is exactly the same as the earlier collection.

Note that the expression above will vary greatly depending on the number of columns in DB that have the same domain. In the worst case, all columns will have a common domain and the expression will be a sum of $O\Big(\Big(\sum_i \|\mathbf{T}_i\|_c\Big)^2\Big)$ terms of the form $\#\mathsf{DB}_{\mathsf{att}(\mathbf{c})=\mathsf{att}(\mathbf{c}')}$. In the best case, none of the columns will share a domain and both collections will be empty. In practice, however, we expect there to be some relatively small number of columns with common domains. In Appendix (4.3), we provide a concrete example of the storage overhead of an encrypted database.

4.4 Security and Leakage of OPX

We show that OPX is adaptively-semantically secure with respect to a well-specified leakage profile. Similar to the leakage profile SPX [39], the profile of OPX is composed of a "black-box component" in the sense that it comes from the underlying STE schemes, and a "non-black-box component" that comes from OPX directly. In this section, we will first describe and prove this leakage profile in a black-box manner, i.e., without assuming any specific instantiation of the underlying STE schemes except for $\Sigma^{\pi}_{\mathsf{MM}}$ which is a concrete response-revealing multi-map encryption scheme by Cash et al. [17]. Then, as a second step, we consider two instantiations with different concrete leakage profiles that illustrate the impact on the overall leakage profile of OPX. In particular, depending on the chosen concrete instantiation, we will show that the resulting leakage profile can be significantly different.

4.4.1 Black-Box Leakage Profile

In the following, we describe the setup and query leakage of OPX without any assumption on how the underlying data structure encryption schemes work.

Setup leakage. The setup leakage captures what a persistent adversary learns by only observing the encrypted structure and before observing any query execution. The setup leakage of OPX is equal to the setup leakage of SPX along with the setup leakage of Σ_{DX} and the number of cells of all tables in the database such that[4]

$$\mathcal{L}^{\mathsf{opx}}_{\mathsf{S}}\Big(\mathsf{DB}\Big) = \Bigg(\left(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_{\mathbf{c}})\right)_{\mathbf{c}\in\mathsf{DB}^{\intercal}}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}\left(\mathsf{MM}_{R}\right), \mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}\left(\mathsf{MM}_{C}\right), \\ \mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}\left(\mathsf{MM}_{V}\right), \left(\mathcal{L}^{\pi}_{\mathsf{S}}(\mathsf{MM}_{\mathbf{c},\mathbf{c}'})\right)_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^{\intercal}}, n\cdot\sum_{i=1}^{n}\|T_i\|_c \Bigg),$$

where $\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}$, $\mathcal{L}^{\pi}_{\mathsf{S}}$, n and $\|\mathbf{T}_i\|$ are the setup leakage of Σ_{MM}, the setup leakage of $\Sigma^{\pi}_{\mathsf{MM}}$ which is equal to the sum of all tuple sizes in a given multi-map, the number of tables, and the number of columns in the ith table, respectively.

Query leakage. The query leakage captures what a persistent adversary learns when it observes the token and query execution. The query leakage of OPX is represented as a *leakage tree* that has the same form as of the query tree QT. In particular, the query leakage, denoted here Λ, starts empty and is then populated in a recursive manner as the query execution

[4]Note that this information will be revealed to the adversary through the size of the set structure SET

goes through in a post-order traversal of the nodes of QT. In particular, for every node N visited in QT, the query leakage is constructed as follows.

Cross product. If the node $N \equiv \mathsf{xnode}$, then this is is a *cross product* pattern which is defined as

$$\mathcal{X}(\mathsf{xnode}) = \begin{cases} \Big(\mathtt{scalar}, |a|\Big) & \text{if } \mathsf{xnode} \equiv [a]; \\ \Big(\mathtt{cross}, \perp\Big) & \text{if } \mathsf{xnode} \equiv \times; \end{cases}$$

This pattern captures what the server learns when it executes a scalar node or a cross product node. The query leakage is now equal to

$$\Lambda := \Lambda \bigcup \Big\{ \mathcal{X}(\mathsf{xnode}) \Big\}.$$

Projection. If $N \equiv \mathsf{pnode}$, then this is a *projection pattern* which is defined as

$$\mathrm{P}(\mathsf{pnode}) = \begin{cases} \Bigg(\mathtt{leaf}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\Big(\mathsf{MM}_C, \chi(\mathsf{att})\Big)\Bigg) & \text{if } \mathsf{pnode} \equiv \pi_{\mathsf{att}}(\mathbf{T}); \\ \Bigg(\mathtt{in}, f(\mathsf{att}_1), \cdots, f(\mathsf{att}_z)\Bigg) & \text{if } \mathsf{pnode} \equiv \pi_{\mathsf{att}_1, \cdots, \mathsf{att}_z}(\mathbf{R_{in}}); \end{cases}$$

where $f \xleftarrow{\$} \Big\{\{0,1\}^* \rightarrow \{0,1\}^{\log(\rho)}\Big\}$ is a uniformly sampled function and ρ is the total number of attributes in DB. The projection pattern captures the leakage produced when the server executes a projection node, whether it is a leaf or an internal node. If the node pnode in QT is a *leaf projection*, then $\mathrm{P}(\mathsf{pnode})$ captures the leakage produced when the server queries EMM_C to retrieve the encrypted content of the column att. More precisely, $\mathrm{P}(\mathsf{pnode})$ reveals the Σ_{MM} query leakage on the coordinates of the projected attribute. Otherwise, if the node pnode is an *internal projection* in QT, then $\mathrm{P}(\mathsf{pnode})$ reveals the position of the attributes, $\mathsf{att}_1, \cdots, \mathsf{att}_z$, in $\mathbf{R_{in}}$ – the intermediary result table given as input to pnode. The query leakage is now equal to

$$\Lambda := \Lambda \bigcup \Big\{ \mathrm{P}(\mathsf{pnode}) \Big\}.$$

Selection. If $N \equiv \mathsf{snode}$, then this is a *selection pattern* which is defined as

$$\mathsf{SELECT}(\mathsf{snode}) = \begin{cases} \Bigg(\mathtt{leaf}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\Big(\mathsf{MM}_V, \Big\langle a, \chi(\mathsf{att})\Big\rangle\Big), \Big(\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}_R, \chi(\mathbf{r})\Big)_{\mathbf{r} \in \mathsf{DB}_{\mathsf{att}=a}}\Bigg) & \text{if } \mathsf{snode} \equiv \sigma_{\mathsf{att}=a}(\mathbf{T}); \\ \Bigg(\mathtt{in}, f(\mathsf{att}), g(a \| \mathsf{att}), \Big(\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}_R, \chi(\mathbf{r})\Big)_{\chi(\mathbf{r}) \in \mathbf{R_{in}} \wedge \mathbf{r}[\mathsf{att}]=a}\Bigg) & \text{if } \mathsf{snode} \equiv \sigma_{\mathsf{att}=a}(\mathbf{R_{in}} \end{cases}$$

where $g \xleftarrow{\$} \Big\{\{0,1\}^* \rightarrow \{0,1\}^{\log(\gamma)}\Big\}$ is a uniformly sampled function, and γ is the sum of distinct values in every column in the entire database. The selection pattern captures the leakage produced when the server executes a selection node, whether it is a leaf or an internal node. If the node snode is a *leaf selection* node, then $\mathsf{SELECT}(\mathsf{snode})$ captures the leakage produced when the server queries EMM_V to retrieve some row tokens. More precisely, $\mathsf{SELECT}(\mathsf{snode})$ reveals the Σ_{MM} query leakage on the coordinates of the attribute att and the constant a. It also reveals the Σ_{MM} query leakage on all coordinates of rows whose cell values at attribute att match the constant a. Otherwise, if the node snode is an *internal selection* node, then $\mathsf{SELECT}(\mathsf{snode})$ captures the leakage produced when the server removes all row tokens in the intermediate result set $\mathbf{R_{in}}$ that do not belong to the set structure SET. In particular, $\mathsf{SELECT}(\mathsf{snode})$ reveals the Σ_{MM} query leakage on all coordinates of rows $\mathbf{r}$ in $\mathbf{R_{in}}$ that match the constant a at the attribute att. The query leakage is now equal to

$$\Lambda := \Lambda \bigcup \Big\{ \mathsf{SELECT}(\mathsf{snode}) \Big\}.$$

Join. If $N \equiv \mathsf{jnode}$, then this is a *join pattern* which is defined as follows. If jnode has form $\mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{T}_2$ then,

$$\mathsf{join}(\mathsf{jnode}) = \Bigg(\mathtt{leaf}, f(\mathsf{att}_1), \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\Bigg(\mathsf{MM}_{\mathsf{att}_1}, \Bigg\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2) \Bigg\rangle\Bigg), \\ \Bigg\{ \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}_R, \chi(\mathbf{r}_1), \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}_R, \chi(\mathbf{r}_2) \Bigg\}_{(\mathbf{r}_1,\mathbf{r}_2)\in \mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}} \Bigg),$$

In this case, $\mathsf{join}(\mathsf{jnode})$ captures the leakage produced when the server retrieves some $\mathsf{EMM}_{\mathsf{att}_1}$ which it in turn queries to retrieve row tokens. More precisely, it reveals if and when $\mathsf{EMM}_{\mathsf{att}_1}$ has been accessed in the past. In addition, it reveals the query leakage of Σ_{MM} on the coordinates of att_1 and att_2 and, for every pair of rows $(\mathbf{r}_1, \mathbf{r}_2)$ in $\mathsf{DB}_{\mathsf{att}_{i,1}=\mathsf{att}_{i,2}}$, it reveals the Σ_{MM} query leakage on their coordinates. If jnode has form $\mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R_{in}}$ then,

$$\mathsf{join}(\mathsf{jnode}) = \Bigg(\mathtt{in}, \langle f(\mathsf{att}_1), f(\mathsf{att}_2) \rangle, \Bigg(\mathcal{L}_{\mathsf{Q}}^{\pi}\Bigg(\mathsf{MM}_{\mathsf{att}_1,\mathsf{att}_2}, \chi(\mathbf{r}) \Bigg)\Bigg)_{\chi(\mathbf{r})\in \mathbf{R_{in}}[\mathsf{att}_2]}, \Bigg\{ \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}_R, \chi(\mathbf{r}_1) \Bigg\}_{\substack{(\mathbf{r}_1,\mathbf{r}_2)\in \mathsf{DB}_{\mathsf{att}_1=\mathsf{att}} \\ \wedge \chi(\mathbf{r}_2)\in \mathbf{R_{in}}[\mathsf{att}_2}}$$

where $\mathbf{R_{in}}[\mathsf{att}]$ denotes the cell values in $\mathbf{R_{in}}$ at attribute att. In this case, $\mathsf{join}(\mathsf{jnode})$ captures the leakage produced when the server retrieves $\mathsf{EMM}_{\mathsf{att}_1,\mathsf{att}_2}$ which it in turn queries to retrieve row tokens. More precisely, it reveals if and when $\mathsf{EMM}_{\mathsf{att}_1,\mathsf{att}_2}$ has been accessed in the past. In addition, it reveals the query leakage of $\Sigma_{\mathsf{MM}}^{\pi}$ on the coordinates of rows $\mathbf{r}$ that belong to $\mathbf{R_{in}}[\mathsf{att}]$ and, for every pair of rows $(\mathbf{r}_1, \mathbf{r}_2)$ in $\mathsf{DB}_{\mathsf{att}_{i,1}=\mathsf{att}_{i,2}}$ such that $\chi(\mathbf{r}_2) \in \mathbf{R_{in}}[\mathsf{att}_2]$, it reveals the Σ_{MM} query leakage on their row coordinates. In particular, the concrete query leakage of $\Sigma_{\mathsf{MM}}^{\pi}$ reveals if and when the same query is evaluated (search pattern) as well as the response identifiers (access pattern). If jnode has form $\mathbf{R_{in}}^{(l)} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R_{in}}^{(r)}$ then,

$$\mathsf{join}(\mathsf{jnode}) = \Bigg(\mathtt{inter}, f(\mathsf{att}_1), f(\mathsf{att}_2) \Bigg),$$

In this case, $\mathsf{join}(\mathsf{jnode})$ captures the leakage produced when the server removes all the rows in $\mathbf{R}_{\mathbf{in}}^{(l)} \times \mathbf{R}_{\mathbf{in}}^{(r)}$ to only keep those which have the same cell value at both attributes att_1 and att_2. The query leakage is now equal to

$$\Lambda := \Lambda \bigcup \left\{ \mathsf{join}(\mathsf{jnode}) \right\}.$$

Finally, it sets

$$\mathcal{L}_{\mathsf{Q}}^{\mathsf{opx}}(\mathsf{DB}, \mathsf{QT}) := \Lambda.$$

4.4.2 Security of OPX

We now prove that OPX is adaptively semantically-secure with respect to the leakage profile described in the previous sub-section.

Theorem 4.4.1 *[]If F is a pseudo-random function,* SKE *is RCPA secure, $\Sigma_{\mathsf{MM}}^{\pi}$ is adaptively $\left(\mathcal{L}_{\mathsf{S}}^{\pi}, \mathcal{L}_{\mathsf{Q}}^{\pi}\right)$-secure, and Σ_{MM} is adaptively $\left(\mathcal{L}_{\mathsf{S}}^{\mathsf{mm}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\right)$-secure, then* OPX *is adaptively $(\mathcal{L}_{\mathsf{S}}^{\mathsf{opx}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{opx}})$-secure in the random oracle model.*

The proof of Theorem 4.4.1 is in Appendix (A.2).

4.4.3 Concrete Leakage Profile

In this section, we are interested in the leakage profile of OPX when the underlying data structure encryption schemes are instantiated with specific constructions and a well-specified concrete leakage profile. Note that in this section, we make the additional assumption that $\Sigma_{\mathsf{MM}}^{\pi}$ from [17] is replaced with an almost leakage free multi-map encryption scheme. However, this scheme needs to verify some key-equivocation property which is the case for the volume hiding schemes like PBS [41], VLH or AVLH [40] if built using the adaptively-secure $\Sigma_{\mathsf{MM}}^{\pi}$ scheme as the underlying multi-map encryption scheme.

(Almost) Leakage-free data structure encryption schemes. We make the assumption that the underlying response-revealing multi-map encryption scheme Σ_{mm} is almost-leakage free in that it leaks the response length pattern, known as the volume pattern, and the response identity pattern such that

$$\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}, q) = \left(\mathsf{rlen}, \mathsf{rid} \right).$$

To instantiate such a scheme, one can use oblivious RAM (ORAM) simulation techniques [32] in a black-box fashion, or more customized/advanced schemes such as the oblivious tree structures (OTS) [56] or the TWORAM construction [30] with a careful parametrization of the block-sizes, or the AZL construction based on the piggy-backing scheme PBS [41]. These constructions however incur an additional overhead, and some of them, work under new trade-offs. Note that if a construction is response-hiding, then it may require one round of interaction to reveal the response. Note that the leakage profile of OPX can be further improved by using *completely* leakage-free data structures that can also hide the volume pattern, but we defer the details to the full version of this work.

In the following, we describe the concrete leakage profile of OPX when instantiated with a (almost) leakage-free data structure encryption. Specifically, when the node is an xnode, the revealed cross-product pattern remains the same. If the node is a pnode, then the projection pattern added to Λ is now equal to

$$\mathrm{P}(\mathsf{pnode}) = \begin{cases} \Big(\texttt{leaf}, \big(|c_j|\big)_{j\in[\#\mathbf{c}[\mathsf{att}]]}, \mathrm{AccP}\big(\mathsf{att}\big)\Big) & \text{if } \mathsf{pnode}_i \equiv \pi_{\mathsf{att}}(\mathbf{T}); \\ \Big(\texttt{in}, f(\mathsf{att}_1), \cdots, f(\mathsf{att}_z)\Big) & \text{if } \mathsf{pnode}_i \equiv \pi_{\mathsf{att}_1,\cdots,\mathsf{att}_z}(\mathbf{R_{in}}). \end{cases}$$

where $\mathrm{AccP}(\mathsf{att})$ denotes if and when the attribute att has been accessed before.

If the node is an snode, then the revealed selection pattern added to Λ is now equal to

$$\mathsf{SELECT}(\mathsf{snode}) = \begin{cases} \Big(\texttt{leaf}, \Big\{|\mathbf{r}|, \mathrm{AccP}\big(\mathbf{r}\big)\Big\}_{\mathbf{r}\in\mathsf{DB}_{\mathsf{att}=a}}\Big) & \text{if } \mathsf{snode} \equiv \sigma_{\mathsf{att}=a}(\mathbf{T}); \\ \Big(\texttt{in}, g(a\|\mathsf{att}), \Big\{|\mathbf{r}|, \mathrm{AccP}\big(\mathbf{r}\big)\Big\}_{\chi(\mathbf{r})\in\mathbf{R_{in}}\wedge\mathbf{r}[\mathsf{att}]=a}\Big) & \text{if } \mathsf{snode} \equiv \sigma_{\mathsf{att}=a}(\mathbf{R_{in}}); \end{cases}$$

If the node is a jnode, then the revealed join pattern added to Λ is now equal to

$$\mathsf{join}(\mathsf{jnode}) = \Big(\texttt{leaf}, f(\mathsf{att}_1), \Big\{|\mathbf{r}_1|, \mathrm{AccP}(\mathbf{r}_1), |\mathbf{r}_2|, \mathrm{AccP}(\mathbf{r}_2)\Big\}_{(\mathbf{r}_1,\mathbf{r}_2)\in\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}}\Big),$$

if jnode has form $\mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{T}_2$ and,

$$\mathsf{join}(\mathsf{jnode}) = \Big(\texttt{in}, \langle f(\mathsf{att}_1), f(\mathsf{att}_2)\rangle, \Big\{|\mathbf{r}_1|, \mathrm{AccP}(\mathbf{r}_1)\Big\}_{\substack{(\mathbf{r}_1,\mathbf{r}_2)\in\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2} \\ \wedge\chi(r_2)\in\mathbf{R_{in}}[\mathsf{att}_2]}}\Big),$$

if jnode has form $\mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R_{in}}$ and,

$$\mathsf{join}(\mathsf{jnode}) = \Big(\texttt{inter}, f(\mathsf{att}_1), f(\mathsf{att}_2)\Big),$$

if jnode has form $\mathbf{R}^{(l)}_{\mathbf{in}} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R}^{(r)}_{\mathbf{in}}$.

Variant. Note that the leakage profile of OPX can be further improved with some slight modifications to the main opx construction. In particular, if the underlying response-revealing multi-map is replaced with a response-hiding scheme, then the access pattern, $\mathrm{AccP}(\mathbf{r})$, of an accessed row, $\mathbf{r}$, can be completely hidden. Note that even the response length of the intermediary results will not be disclosed as the underlying scheme is leakage-free as per our assumption. For example, in the case of a *leaf select node*, the output will now be a set of row coordinates, instead of row tokens. And in order to proceed to the next node, the client and server need to interact to first decrypt the row coordinate and execute the next operation. Note that this approach will not incur any additional query overhead to what is added by using leakage-free schemes; however it will add additional interaction between the client and the server. The concrete leakage profile of this modified scheme will be the type of nodes composing the query plan, i.e., whether the node is a join, select, or a cross-product node. We defer the details of this variant to the full version of this work.

Efficiency. We have shown in Theorem 4.3.1 that both the OPX query algorithm and the equivalent plaintext execution on the same query tree QT have exactly the same query complexity if the underlying multi-map and dictionary encryption schemes are instantiated using standard techniques [26, 21, 42, 18, 17]. However, in the (almost) leakage-free setting, the query complexity of opx is higher for the simple reason that the cost of querying a leakage-free data structure encryption scheme is higher than the one of querying a standard (optimal) scheme. More precisely, at any step where the client and server execute a Σ_{mm} query protocol, then the query complexity will be higher depending on the executed node. We describe below this impact in more details.

- **(case 1):** If the node is a *leaf selection node* of the form $\sigma_{\mathsf{att}=a}(\mathbf{T})$, then the overhead is equal to

$$O\left(\#\mathsf{DB}_{\mathsf{att}=a} \cdot \log\left(m \cdot \sum_{i=1}^{n} \|\mathbf{T}_i\|_c\right)\right).$$

  where m is the maximum number of cells in a table; instead of $O(m)$ – the query complexity of a plaintext execution on the same node.

- **(case 2):** If the node is a *leaf join node* of the form $\mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} (\mathbf{T}_2)$, then the overhead is equal to

$$O\left(\#\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2} \cdot \log\left(\sum_{\mathsf{att}\in\mathbb{S}(\mathsf{DB})} \sum_{\substack{\mathsf{att}'\in\mathbb{S}(\mathsf{DB})\\ \mathsf{dom}(\mathsf{att})=\mathsf{dom}(\mathsf{att}')}} \#\mathsf{DB}_{\mathsf{att}=\mathsf{att}'}\right)\right),$$

  where $\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}$ is the tuple composed of all joined pairs between columns att_1 and att_2; instead of $O(\#\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2})$ – the query complexity of a plaintext execution on the same node.

- **(case 3):** If the node is an *internal join node* of the form $\mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} (\mathbf{R_{in}})$, then the overhead is equal to

$$O\left(\sum_{\chi(\mathbf{r})\in\mathbf{R_{in}}[\mathsf{att}_2]} \left(\#\mathsf{DB}_{\mathsf{att}_1=\mathtt{value}_{\mathsf{att}_2}(\mathbf{r})} \cdot \log\left(\sum_{\mathsf{att}\in\mathbb{S}(\mathsf{DB})} \sum_{\substack{\mathsf{att}'\in\mathbb{S}(\mathsf{DB})\\ \mathsf{dom}(\mathsf{att})=\mathsf{dom}(\mathsf{att}')}} \#\mathsf{DB}_{\mathsf{att}=\mathsf{att}'}\right)\right)\right),$$

where $\texttt{value}_{\mathsf{att}_2}(\mathbf{r})$ is the cell value of row $\mathbf{r}$ at attribute att_2; instead of $O(\sum_{\chi(\mathbf{r})\in\mathbf{R_{in}}[\mathsf{att}_2]}(\#\mathsf{DB}_{\mathsf{att}_1=\texttt{value}_{\mathsf{att}_2}(\mathbf{r})}))$– the query complexity of a plaintext execution on the same node.

- **(case 5):** If the node is a *leaf projection node* of the form $\pi_{\mathsf{att}}(\mathbf{T})$, then the overhead is equal to
$$O\left(m \cdot \log\left(m \cdot \sum_{i=1}^{n} \|\mathbf{T}_i\|_c\right)\right),$$
where m is the maximum number of cells in the table.

- **(case 5):** if the node is a *scalar node*, a *cross-product node*, an *intermediate internal join*, an *internal projection node*, or an *internal selection node*, then the query complexity is similar to a plaintext execution as no multi-map or dictionary query executions are required in the process.

Note that using (almost) leakage-free data structures to instantiate OPX does not incur any asymptotical storage overhead.

Standard data structure encryption schemes. In this section, we describe the leakage profile of OPX if instantiated with standard data structure encryption schemes [26, 21, 42, 18, 17]. By *standard*, we refer to a class of well-studied data structure encryption schemes that reveal the *response identity pattern* (rid), and the *query equality pattern* (qeq), known as the access pattern and the search pattern in the SSE literature, respectively. The search pattern reveals if and when a query is repeated while the access pattern reveals the identities of the responses. The concrete leakage profile of opx when instantiated with these standard data structures is the same as the one detailed in the abstract section except that we replace the black box notation $\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}$ with rid and qeq on the same inputs. Below, we give a high level intuition on what each pattern will disclose.

Select pattern. Independently of the type of the selection node, then an adversary can learn the number of rows containing the same value as well as the frequency with which a particular row has been accessed, and also the size of that row. If many queries have been performed on the same table and the same column, then the adversary can build a frequency histogram of that specific column's contents. Now depending on the composition of the query tree, an adversary can build a more detailed histogram if more *internal* selection are performed on the same attribute.

Join pattern. Among all patterns, the join pattern leaks the most. The adversary learns the number of rows that have equal values in a given pair of attributes. In addition, it learns the frequency with which these rows have been accessed in the past, eventually following the execution of a different type of nodes such as a projection or a selection. Similar to the selection pattern, the adversary can build therefore a histogram summarizing the frequency of apparition of rows that it gets richer with more operations down the query tree. If the join node is internal, then the adversary learns a bit more information as for every row, it knows exactly the rows in a different attribute that have the same value. The adversary can help the adversary for example to trace back to the leaf join leakage information it collected to identify the exact rows that have the same values. This is also true in general for all the information the adversary collects from different nodes as long as the operations are correlated. Finally, if the node is an *intermediate internal node*, then the execution of such a node leads to the propagation of the frequency information cross different attributes.

Projection pattern. This pattern simply discloses the number of rows in a specific attributes (size of the column) along with the frequency with which these rows have been accessed.

Note that we dismissed a discussion on the cross-product pattern as it is self-explanatory and does not involve querying any data structure encryption scheme.

Efficiency. With respect to efficiency, we have shown in Theorem 4.3.1 that the execution of the OPX query algorithm and its plaintext counterpart have exactly the same asymptotics.

4.5 The OPX Protocol

We detail the pseudo-code of OPX in Figures (4.6), (4.8) and (4.9).

Let $\Sigma_{\mathsf{DX}} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Get})$ be a response-revealing dictionary encryption scheme, $\Sigma_{\mathsf{spx}} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$ be a the SPX encryption scheme from [39], $\Sigma_{\mathsf{MM}} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Get})$ be a response-revealing multi-map encryption scheme and $F : \{0,1\}^k \times \{0,1\}^\star \to \{0,1\}^\star$ be a pseudo-random function. Consider the DB encryption scheme $\mathsf{opx} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query}, \mathsf{Dec})$ defined as follows [a]:

```
• $\mathsf{Setup}(1^k, \mathsf{DB})$:
    1. initialize a set $\mathsf{SET}$;
    2. initialize multi-maps $\mathsf{MM}_R$, $\mathsf{MM}_C$ and $\mathsf{MM}_V$;
    3. initialize multi-maps $(\mathsf{MM}_{\mathsf{att}})_{\mathsf{att} \in \mathsf{DB}^\intercal}$;
    4. initialize multi-maps $(\mathsf{MM}_{\mathsf{att},\mathsf{att}'})_{\mathsf{att},\mathsf{att}' \in \mathsf{DB}^\intercal}$ such that $\mathsf{dom}(\mathsf{att}) = \mathsf{dom}(\mathsf{att}')$;
    5. sample two keys $K_1, K_F \overset{\$}{\leftarrow} \{0,1\}^k$;
    6. for all $\mathbf{r} \in \mathsf{DB}$ set
           $\mathsf{MM}_R\big[\chi(\mathbf{r})\big] := \Big(\mathsf{Enc}_{K_1}(r_1), \ldots \mathsf{Enc}_{K_1}(r_{\#\mathbf{r}}), \chi(\mathbf{r})\Big)$;
    7. compute $(K_R, \mathsf{EMM}_R) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}_R\big)$;
    8. for all $\mathbf{c} \in \mathsf{DB}^\intercal$, set
           $\mathsf{MM}_C\big[\chi(\mathbf{c})\big] := \Big(\mathsf{Enc}_{K_1}(c_1), \ldots \mathsf{Enc}_{K_1}(c_{\#\mathbf{c}}), \chi(\mathbf{c})\Big)$;
    9. compute $(K_C, \mathsf{EMM}_C) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}_C\big)$;
    10. for all $\mathbf{c} \in \mathsf{DB}^\intercal$,
        (a) for all $v \in \mathbf{c}$ and $\mathbf{r} \in \mathsf{DB}_{\mathbf{c}=v}$,
            i. compute $\mathsf{rtk}_{\mathbf{r}} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_R, \chi(\mathbf{r})\Big)$,
        (b) set
               $\mathsf{MM}_V\Big[\big\langle v, \chi(\mathbf{c})\big\rangle\Big] := \Big(\mathsf{rtk}_{\mathbf{r}}\Big)_{\mathbf{r} \in \mathsf{DB}_{\mathbf{c}=v}}$;
    11. compute $(K_V, \mathsf{EMM}_V) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}_V\big)$;
    12. for all $\mathbf{c} \in \mathsf{DB}^\intercal$,
        (a) for all $\mathbf{c}' \in \mathsf{DB}^\intercal$ such that $\mathsf{dom}(\mathsf{att}(\mathbf{c}')) = \mathsf{dom}(\mathsf{att}(\mathbf{c}))$,
            i. initialize an empty tuple $\mathbf{t}$;
            ii. for all rows $\mathbf{r}_i$ and $\mathbf{r}_j$ in $\mathbf{c}$ and $\mathbf{c}'$, such that $\mathbf{c}[i] = \mathbf{c}'[j]$,
                A. compute $\mathsf{rtk}_i \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_R, \chi(\mathbf{r}_i)\Big)$;
                B. compute $\mathsf{rtk}_j \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_R, \chi(\mathbf{r}_j)\Big)$;
                C. add $(\mathsf{rtk}_i, \mathsf{rtk}_j)$ to $\mathbf{t}$;
            iii. set
                   $\mathsf{MM}_{\mathbf{c}}\Big[\Big\langle \chi(\mathbf{c}), \chi(\mathbf{c}')\Big\rangle\Big] := \mathbf{t}$;
        (b) compute $(K_{\mathbf{c}}, \mathsf{EMM}_{\mathbf{c}}) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}_{\mathbf{c}}\big)$;
```

[a] Note that we omit the description of Dec since it simply decrypts every cell of R.

Figure 4.6: opx: an optimal relational DB encryption scheme (Part 1).

```
• Token(K, QT):
    13. for all $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$,
        (a) for all $v \in \mathbf{c}$,
            i. compute $K_v \leftarrow F_K(\chi(\mathbf{c}) \| v)$;
            ii. set for all $\mathbf{r} \in \mathsf{DB}_{\mathbf{c}=v}$,
                    $\mathsf{SET} := \mathsf{SET} \bigcup \left\{ F_{K_v}\left(\mathsf{rtk}\right) \right\}$,
                where $\mathsf{rtk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}))$;
    14. for all $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$,
        (a) for all $\mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}$ such that $\mathsf{dom}(\mathsf{att}(\mathbf{c}')) = \mathsf{dom}(\mathsf{att}(\mathbf{c}))$,
            i. initialize an empty tuple $\mathbf{t}$;
            ii. for all $\mathbf{r}_i, \mathbf{r}_j \in [m]$ such that $\mathbf{c}[i] = \mathbf{c}'[j]$,
                A. add $\left(\mathsf{rtk}_i, \mathsf{rtk}_j\right)$ to $\mathbf{t}$ where
                    $\mathsf{rtk}_i \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}_i))$ and $\mathsf{rtk}_j \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}_i j)$.
            iii. for all rtk s.t. $(\mathsf{rtk}, \cdot) \in \mathbf{t}$, set
                    $\mathsf{MM}_{\mathbf{c},\mathbf{c}'}\left[\mathsf{rtk}\right] := \left(\mathsf{rtk}'\right)_{(\mathsf{rtk},\mathsf{rtk}') \in \mathbf{t}}$
        (b) compute $(K_{\mathbf{c},\mathbf{c}'}, \mathsf{EMM}_{\mathbf{c},\mathbf{c}'}) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\left(1^k, \mathsf{MM}_{\mathbf{c},\mathbf{c}'}\right)$;
    15. output $K = (K_1, K_R, K_C, K_V, \{K_{\mathbf{c}}\}_{\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}}, K_F, \{K_{\mathbf{c},\mathbf{c}'}\}_{\mathbf{c},\mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}})$ and $\mathsf{EDB} = (\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, (\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}}, \mathsf{SET}, (\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}})$.
```

Figure 4.7: opx: an optimal relational DB encryption scheme (Part 2).

- $\mathsf{Token}(K, \mathsf{QT})$:
    1. initialize a token tree TT with empty nodes and with the same structure as QT;
    2. for every node N accessed in a post-traversal order in QT ,
        (a) if $N \equiv \sigma_{\mathsf{att}=a}(\mathbf{T})$ then set TT_N to
        $$\mathsf{stk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_V, \langle a, \chi(\mathsf{att})\Big);$$
        (b) if $N \equiv \sigma_{\mathsf{att}=a}(\mathbf{R_{in}})$ then set TT_N to $(\mathsf{rtk}, \mathtt{pos})$ where
        $$\mathsf{rtk} \leftarrow F_{K_F}\Big(\chi(\mathsf{att})\|a\Big)$$
        and $\mathtt{pos}$ denotes the position of att in $\mathbf{R_{in}}$.
        (c) if $N \equiv \mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{T}_2$ then set TT_N to $(\mathsf{jtk}, \mathtt{pos})$ where
        $$\mathsf{jtk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_{\mathsf{att}_1}, \Big\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2)\Big\rangle\Big),$$
        and $\mathtt{pos}$ is the position of attribute att_1 in $\mathbf{R_{in}}$.
        (d) if $N \equiv \mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R_{in}}$ then set the corresponding node in TT to $(\mathsf{etk}, \mathtt{pos}_1, \mathtt{pos}_2)$ where
        $$\mathsf{etk} := K_{\mathsf{att}_1,\mathsf{att}_2};$$
        and $\mathtt{pos}_1$, $\mathtt{pos}_2$ are the positions of the attributes att_1, att_2 in $\mathbf{R_{in}}$, respectively.
        (e) if $N \equiv \mathbf{R}^{(l)}_{\mathbf{in}} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R}^{(r)}_{\mathbf{in}}$ then set TT_N to $(\mathtt{pos}_1, \mathtt{pos}_2)$ where $\mathtt{pos}_1$ and $\mathtt{pos}_2$ are the column positions of att_1 and att_2 in $\mathbf{R}^{(l)}_{\mathbf{in}}$ and $\mathbf{R}^{(r)}_{\mathbf{in}}$, respectively.
        (f) if $N \equiv \pi_{\mathsf{att}}(\mathbf{T})$ then set TT_N to ptk where
        $$\mathsf{ptk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_C, \chi(\mathsf{att}_i)\Big).$$
        (g) if $N \equiv \pi_{\mathsf{att}_1,\cdots,\mathsf{att}_z}(\mathbf{R_{in}})$ then set TT_N to
        $$\Big(\mathtt{pos}_1, \cdots, \mathtt{pos}_z\Big),$$
        where $\mathtt{pos}_i$ is the column position of att_i in $\mathbf{R_{in}}$.
        (h) if $N \equiv [a]$ then set TT_N to $[\mathsf{Enc}_{K_1}(a)]$.
        (i) if $N \equiv \times$ then set TT_N to $\times$.
    3. output TT.

Figure 4.8: opx: an optimal relational DB encryption scheme (Part 2).

- $\mathsf{Query}(\mathsf{EDB}, \mathsf{tk})$:
  1. parse EDB as $\big(\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, \mathsf{EDX}_1, \mathsf{EDX}_2, \mathsf{SET}\big)$;
  2. for every node N accessed in a post-traversal order in TT,
     - if $N \equiv \mathsf{stk}$, it computes
       $$(\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}\Big(\mathsf{stk}, \mathsf{EMM}_V\Big),$$
       and sets $\mathbf{R_{out}} := (\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s)$;
     - if $N \equiv (\mathsf{rtk}, \mathsf{pos})$, then for all rtk in $\mathbf{R_{in}}$ in the column at position pos, if
       $$F_{\mathsf{rtk}}(\mathsf{rtk}) \notin \mathsf{SET}$$
       then it removes the row from $\mathbf{R_{in}}$. Finally, it sets $\mathbf{R_{out}} := \mathbf{R_{in}}$;
     - if $N \equiv (\mathsf{jtk}, \mathsf{pos})$, then it computes
       $$\Big((\mathsf{rtk}_1, \mathsf{rtk}'_1), \ldots, (\mathsf{rtk}_s, \mathsf{rtk}'_s)\Big) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}(\mathsf{jtk}, \mathsf{EMM}_{\mathsf{pos}}),$$
       and sets
       $$\mathbf{R_{out}} := \Big((\mathsf{rtk}_i, \mathsf{rtk}'_i)\Big)_{i\in[s]};$$
     - if $N \equiv (\mathsf{etk}, \mathsf{pos}_1, \mathsf{pos}_2)$, then for each row $\mathbf{r}$ in $\mathbf{R_{in}}$, it computes $\mathsf{ltk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(\mathsf{etk}, \mathsf{rtk})$, and
       $$(\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}(\mathsf{ltk}, \mathsf{EMM}_{\mathsf{pos}_1,\mathsf{pos}_2}),$$
       where $\mathsf{rtk} = \mathbf{r}[\mathsf{att}_{\mathsf{pos}_2}]$, and appends the new rows $\Big(\mathsf{rtk}_i\Big)_{i\in[s]} \times \mathbf{r}$ to $\mathbf{R_{out}}$;
     - if $N \equiv (\mathsf{pos}_1, \mathsf{pos}_2)$, then it sets
       $$\mathbf{R_{out}} := \mathbf{R}^{(l)}_{\mathbf{in}} \bowtie_{\mathsf{pos}_1=\mathsf{pos}_2} \mathbf{R}^{(r)}_{\mathbf{in}},$$
       where $\mathbf{R}^{(l)}_{\mathbf{in}}$ and $\mathbf{R}^{(r)}_{\mathbf{in}}$ are the left and right input respectively;
     - if $N \equiv \mathsf{ptk}$ then it computes
       $$(\mathrm{ct}_1, \cdots, \mathrm{ct}_s) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}\Big(\mathsf{ptk}, \mathsf{EMM}_C\Big)$$
       and sets $\mathbf{R_{out}} := (\mathrm{ct}_1, \cdots, \mathrm{ct}_s)$;
     - if $N \equiv (\mathsf{pos}_1, \cdots, \mathsf{pos}_z)$, then it computes $\mathbf{R_{out}} := \pi_{\mathsf{pos}_1,\cdots,\mathsf{pos}_z}(\mathbf{R_{in}})$;
     - if $N \equiv \times$ then it computes
       $$\mathbf{R_{out}} := \mathbf{R}^{(l)}_{\mathbf{in}} \times \mathbf{R}^{(r)}_{\mathbf{in}};$$
  3. it replaces each cell rtk in $\mathbf{R}^{\mathsf{root}}_{\mathbf{out}}$ by $\mathrm{ct} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}(\mathsf{rtk}, \mathsf{EMM}_R)$;
  4. output $\mathbf{R}^{\mathsf{root}}_{\mathbf{out}}$.

Figure 4.9: opx: an optimal relational DB encryption scheme (Part 3).

Chapter 5

Legacy Compliance

5.1 Overview

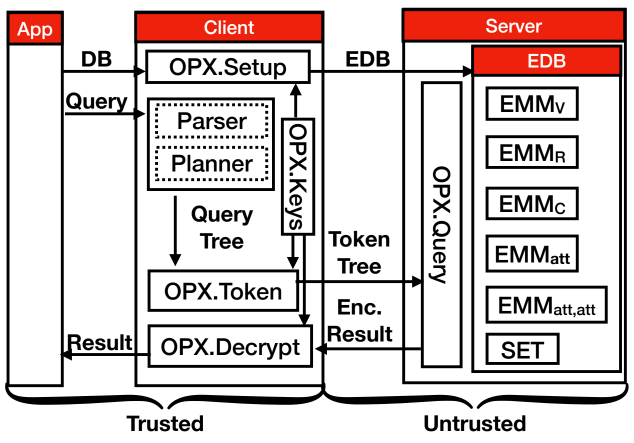
The main limitation of STE is its use of non-standard query algorithms which limits its applicability since it requires re-architecting existing storage systems. In fact, this lack of "legacy-friendliness" is widely considered to be the main reason practical encrypted search deployments use PPE-based designs. Legacy-friendliness is an important property in practice, especially in the context of database systems which have been optimized over the last 40 years.

In this section, we show that the common belief that STE is not legacy-friendly is not true. We introduce a new technique called *emulation* that makes STE schemes legacy-friendly. At a high level, the idea is to take an encrypted data structure (e.g., an encrypted multi-map) and find a way to represent it as a table, without any additional storage or query overhead. The access to the encrypted data structure will also need to be translated into relatioanl queries. The benefits of emulation are twofold: (1) low-overhead emulator essentially makes an STE scheme legacy-friendly; and (2) it preserves the STE scheme's security.

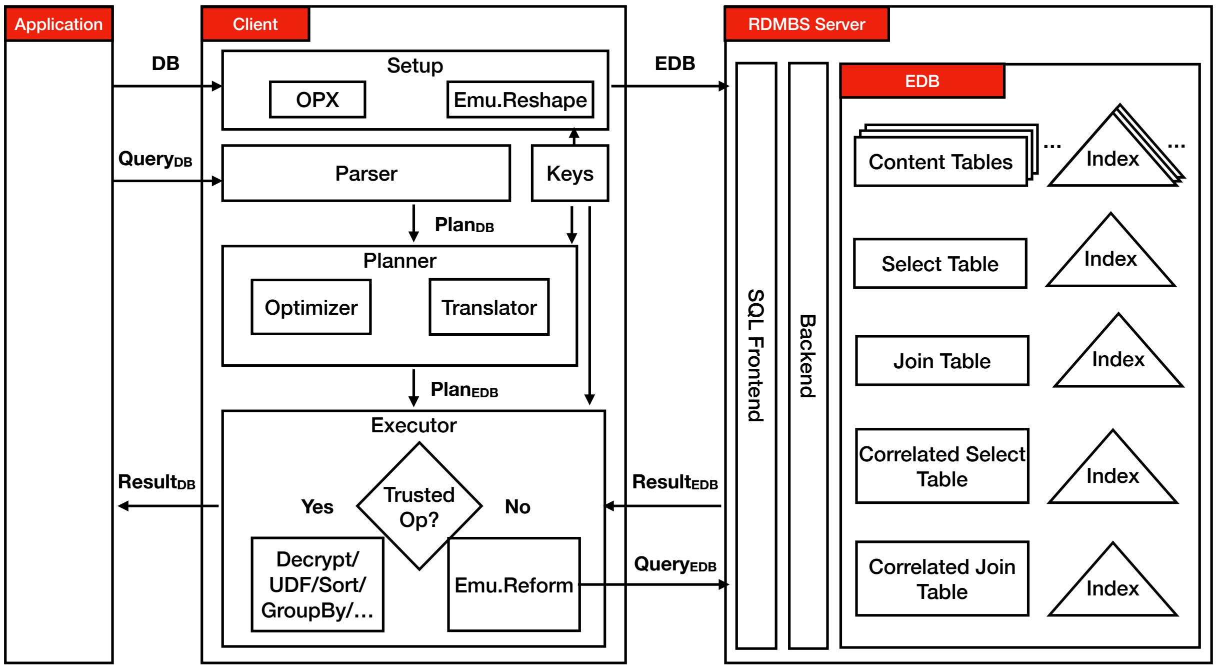
5.2 The `KafeDB` Architecture

`KafeDB` is a relational EDB system. Our implementation is built on top of `PostgreSQL` but because of our emulation technique, we could have built `KafeDB` on top of any relational database management system—even a managed DBMS in the cloud. `KafeDB` does not require any modifications to applications or servers. It is completely transparent to both. We depict `KafeDB`'s design for OPX in Figure 5.1b, contrasting with the pre-emulation architecture for OPX in Figure 5.1a. We first describe the architecture of `KafeDB`, and then present full treatment of emulation in Section 5.3.

Overview. `KafeDB` is composed of the `KafeDB` client which runs on a trusted device (preferably the same machine as the application) and a DBMS server which runs on an untrusted device. The `KafeDB` client sits between the application and the server and handles



(a) System Architecture for OPX before emulation.



(b) System Architecture KafeDB for OPX after emulation.

all the application queries. When the application creates a database, the client sets up and emulates an OPX-encrypted database on the server. Thanks to our emulation techniques, the OPX-encrypted database is represented as a standard SQL database. At query time, the client receives a SQL query from the application, converts it into an OPX token tree and emulates it into a standard SQL query plan. This query plan is then executed by the server. The server returns an encrypted result to the client who decrypts it and executes any additional processing (e.g., grouping, aggregating etc.) before returning the result to the application. As illustrated in Figure ??, the KafeDB client is composed of several modules which we now describe.

Setup. The setup module takes as input a relational database encrypts it with OPX and emulates/reshapes the result back into a standard relational database which it then sends to the server.

Parser. The parser module is a standard SQL parser that transforms the SQL query into a query plan/tree. `KafeDB` uses the SQL parser from SparkSQL [9].

Planner. The planner module has two components: an optimizer and translator. The *optimizer* transforms a query plan into an equivalent query plan that can be executed more efficiently. `KafeDB` uses a variety of query optimizations techniques. Some are well-known optimizations used by standard DBMSs and others are specific to `KafeDB` and customized for working with encrypted data. The *translator* then transforms the optimized query plan into a partial token tree where some nodes are plaintext operations to be executed on the client, and some nodes are tokens to be executed on the server.

Executor. The executor takes as input a partial token tree and replaces each tokenized node with a SQL expression that it generates by emulating/reforming the token. Note that after every tokenized node has been emulated, the tree is entirely composed of SQL expressions. The executor now does a *split execution* of this query plan by sending the SQL subqueries of supported operations to the server (and decrypting the results) and executing the unsupported operations at the client. The executor then sends the final result back to the application.

Server. The server is an unmodified standard SQL DBMS such as `PostgreSQL`. The server stores encrypted *content tables*, by which we mean encryptions of the original database tables, and *auxiliary tables* and *auxiliary indices* that result from the emulation of the OPX encrypted data structures.

Optimizations. One of the most important components of any DBMS is its query optimizer which transforms a query plan into a physical plan which can be executed as efficiently as possible. Commercial query optimizers are the result of over 40 years of research from the database community and are major reason why real-world DBMSs are so efficient. It follows then that for `KafeDB` to be competitive at all with a commercial system, it has to support some form of query optimization. Due to space limitations, we describe `KafeDB`'s optimizations in more detail in Appendix **??**.

5.3 Emulation

The main limitation of STE is its use of non-standard query algorithms which limits its applicability since it requires re-architecting existing storage systems. In fact, this lack of "legacy-friendliness" is widely considered to be the main reason practical encrypted search deployments use PPE-based designs. Legacy-friendliness is an important property in practice, especially in the context of database systems which have been optimized over the last 40 years.

In this section, we show that the common belief that STE is not legacy-friendly is not true. We introduce a new technique called *emulation* that makes STE schemes legacy-friendly. At a high level, the idea is to take an encrypted data structure (e.g., an encrypted multi-map) and find a way to represent it as another data structure (e.g., a graph) without any additional storage or query overhead. Intuitively, emulation is a more sophisticated version of the classic data structure problem of simulating a stack with two queues. Designing storage- and query-efficient emulators can be challenging depending on the encrypted structure being emulated and the target structure (i.e., the structure we wish to emulate on top of). The benefits of emulation are twofold: (1) low-overhead emulator essentially makes an STE scheme legacy-friendly; and (2) it preserves the STE scheme's security.

Definition 5.3.1 *[SQL emulator] Let* $\mathsf{STE} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query}, \mathsf{Resolve})$ *be a response-hiding structured encryption scheme and* $\mathsf{SQL} = (\mathsf{Setup}, \mathsf{Exec})$ *be a relational DBMS. An SQL emulator* $\mathsf{Emu} = (\mathsf{Reshape}, \mathsf{Reform})$ *for* STE *is a set of two polynomial-time algorithms that work as follows:*

- $\mathsf{DB} \leftarrow \mathsf{Reshape}(\mathsf{EDS})$*: is a possibly probabilistic algorithm that takes as input an encrypted structure* EDS *generated using* $\mathsf{STE.Setup}$ *and outputs a database* $\mathsf{DB} = (\mathbf{T}_1, \ldots, \mathbf{T}_n)$*.*

- $Q \leftarrow \mathsf{Reform}(\mathbb{S}(\mathsf{DB}), \mathsf{tk})$ *is a possibly probabilistic algorithm that takes as input the schema of the emulated structure* $\mathbb{S}(\mathsf{DB})$ *and an* STE *token* tk *and outputs a SQL query* Q*.*

We say that Emu *is correct if for all* $k \in \mathbb{N}$*, for all* DS*, for all* (K, st, EDS) *output by* $\mathsf{STE.Setup}(1^k, \mathsf{DS})$*, for all* DB *output by* $\mathsf{Reshape}(\mathsf{EDS})$*, for all queries* $q \in \mathbb{Q}$*, for all tokens* tk *output by* $\mathsf{Token}(K, q)$*,* $\mathsf{SQL.Exec}(\mathsf{DB}, Q) = \mathsf{STE.Query}(\mathsf{EDS}, \mathsf{tk})$*.*

Security and efficiency. Since emulators operate strictly on the encrypted structures and the tokens produced by their underlying STE schemes, it follows trivially that an emulated/reshaped structure, DB, reveals nothing beyond the setup leakage of the original encrypted structure EDS. Similarly, the emulated/reshaped structure, DB, and the emulated/reformed token, Q, reveal nothing beyond the query leakage of EDS and tk.

While emulators preserve the security of their underlying STE scheme, they do not necessarily preserve their efficiency. In fact, the restructuring step could lead to an emulated structure EEDS that is: (1) larger than the original structure EDS; and (2) less query-efficient. The main challenge in designing emulators, therefore, is to design restructuring and query algorithms that do not affect the efficiency of the pre-emulated structure.

5.3.1 A SQL Emulator for Pibase

We recall the Pibase scheme from [17]. The Setup algorithm of Pibase samples a key K and instantiates a dictionary DX. For each label $\ell \in \mathbb{L}$, it generates two label keys $K_{\ell,1}$ and $K_{\ell,2}$ by evaluating a pseudo-random function F_K on on $\ell\|1$ and $\ell\|2$, respectively. Then, for each value v_i in the tuple $\mathbf{t}_\ell = (v_1, \cdots, v_m)$ associated with ℓ, it creates an encrypted label $\ell'_i := F_{K_{\ell,1}}(i)$ which is the evaluation of $F_{K_{\ell,1}}$ on a counter. It then inserts an encrypted label/value pair $(\ell'_i, \mathsf{Enc}_{K_2}(v_i))$ in the dictionary DX. The encrypted multi-map EMM consists of the dictionary DX. $\mathsf{EMM} = \mathsf{DX}$ is sent to the server.

To Get the value associated with a label ℓ, the client sends the label key $K_{\ell,1}$ to the server who does the following. It evaluates the pseudo-random function $F_{K_{\ell,1}}$ on counter value i and uses the result to query DX. More precisely, it computes $\mathrm{ct}_i := \mathsf{DX}[F_{K_{\ell,1}}(i)]$ and if $\mathrm{ct}_i \neq \perp$ it sends it back to the client and increments i and continues otherwise it stops.

A SQL emulator. Our SQL emulator for Pibase works as follows. Given an encrypted multi-map EMM, the $\mathsf{Reshape}$ algorithm creates a table $\mathbf{T}$ with name $\mathbf{T}.\mathsf{name} = \mathsf{EMM}.\mathsf{name}$ and schema $\mathbb{S}(\mathbf{T}) = (\mathtt{label}, \mathtt{value})$ by executing

```
CREATE TABLE T.name, label,value.
```

For efficiency reasons, an index is created over the table $\mathbf{T}$ by executing,

```
Create Index On T.name (label).
```

It then parses EMM as a dictionary DX and, for each label/value pair (ℓ, e) in DX, inserts the row $\mathbf{r} = \Big(\ell, e\Big)$ in $\mathbf{T}$ by executing

```
INSERT INTO T.name (label,value) VALUES $(\ell_1, e_1) \ldots, (\ell_m, e_m)$.
```

Given a token $\mathsf{tk} = K_{\ell_1}$, the Reform algorithm outputs the following SQL common table expression,

```
WITH RECURSIVE G(label,value,$i$) AS (
    SELECT T.label,T.value,1 FROM T WHERE T.label = $\mathsf{UDF}_F\Big(K_{\ell_1}, i\Big)$
    UNION ALL
    SELECT T.label,T.value,$i+1$ FROM T,G
    WHERE T.label = $\mathsf{UDF}_F\Big(K_{\ell_1}, i+1)\Big)$
        and T.label = G.label
)
SELECT value FROM G,
```

Efficiency. The storage overhead of the emulated encrypted multi-map is equal to the size of the encrypted table, $O(\sum_{\ell \in \mathbb{L}} \#\mathsf{MM}[\ell])$, plus the size of the plaintext index created over the table $\mathbf{T}$, $O(\#\mathbb{L})$. Since the latter is dominated by the former, the overall size of the emulated encrypted multi-map is equal to $O(\sum_{\ell \in \mathbb{L}} \#\mathsf{MM}[\ell])$. The emulated get operation runs in $O(\#\mathsf{MM}[\ell])$ which is optimal due to the index created on $\mathbf{T}$.

5.3.2 A SQL Emulator for OPX Set Structure

OPX makes use of a set structure SET, refer to Section **??** for more details. In the following, we are going to describe abstractly how this set structure is built and queried. Given multiple sets $\mathbb{S} = \{S_1, \cdots, S_n\}$ such that $S_i = \{e_{i,1}, \cdots, e_{i,s_i}\}$, for all $i \in [n]$. It first samples two keys $K_1, K_2 \xleftarrow{\$} \{0,1\}^k$ and creates a new empty set SET that it populates as follows. For each $i \in [n]$, and each element $e_{i,j}$, for $j \in [s_i]$, it computes $\mathsf{SET} := \mathsf{SET} \cup \{F_{K^\star}(F_{K_1}(e_{i,j}))\}$, where $K^\star := F_{K_2}(i \| e_{i,j})$, and F is a pseudo-random function. The client outputs two keys K_1, K_2 and SET.

Now, given a set of values $\mathrm{ct} = \{\mathrm{ct}_1, \cdots, \mathrm{ct}_m\}$ and a position of a set i, the client and server want to test if there are any elements in S_i equal simultaneously to some value v and one of the ct_j, for $j \in [m]$. The client sends a token $\mathsf{tk} := F_{K_2}(i \| v)$, and then the server checks if $F_{\mathsf{tk}}(\mathrm{ct}_j) \in \mathsf{SET}$, for $j \in [m]$. The server outputs true or false depending on the membership result for each $j \in [m]$.

A SQL emulator. Given a set structure SET constructed as above, the Reshape algorithm creates a table $\mathbf{T}$ with name $\mathbf{T}.\mathtt{name} = \mathsf{SET}.\mathtt{name}$ and schema $\mathbb{S}(\mathbf{T}) = \mathtt{label}$ by executing,

```
CREATE TABLE T.name, label.
```

For efficiency, an index is created over the database $\mathsf{DB} = \mathbf{T}$ by executing

```
Create Index On T.name (label).
```

For every element e in SET, it inserts the row $\mathbf{r} = \ell$ in $\mathbf{T}$ by executing

```
INSERT INTO T.name label VALUES (e_1),...,(e_{s_i}).
```

It then outputs the table $\mathbf{T}$. Given a token $\mathsf{tk} = F_{K_2}(\mathsf{pos} \| v)$ and a position pos, the Reform algorithm outputs the SQL query,

```
SELECT (S).* FROM (S) WHERE EXISTS(
         SELECT label FROM T.name
         WHERE label = UDF_F(tk, S[pos])
),
```

where S is the input of the server.

Efficiency. The execution of the SQL query Q on the database $\mathsf{DB} = \mathbf{T}$ is $O(\#\mathbf{R_{in}})$ due to the index created on the `label` column. The size of $\mathbf{T}$ is $O(\sum_{i=1}^{n} \#S_i)$.

5.4 Empirical Evaluation

In this section, we evaluate how `KafeDB` performs in practice. In particular, we are interesting in assessing: (1) setup time; (2) query efficiency; (3) storage efficiency; and (4) the impact of our optimizations on all these dimensions.

Implementation. The KafeDB client makes use of and extends several components of Apache Spark SQL [9]. Specifically, it uses and extends Spark SQL's algebraic core for query translation and optimization, its parser to parse plaintext SQL queries into a query plan, and its executor to facilitate split execution. The KafeDB server can be any DBMS but in this evaluation we use PostgreSQL 9.6.2 [33]. The KafeDB client is written in Scala 2.12 and consists of 1599 lines of code. In addition, our framework includes 398 lines for testing, 392 lines to load the TPC-H benchmark, 578 lines to execute the TPC-H benchmark, all calculated using IntelliJ IDEA [1]. the query parser and executor code required for all other modules composing KafeDB Client are inherited from Apache Spark SQL. Our implementation is available for download in an anonymized form here [6]. For the cryptographic building blocks, we use AES in CBC mode with PKCS7 padding for symmetric encryption, and HMAC-SHA-256 for pseudo-random functions. Both primitives are provided by Bouncy Castle 1.64 [44] in the DEX Client and by the pgcrypto module in PostgreSQL 9.6.2.[1]

Testing environment. We conducted our experiments on Amazon Elastic Compute Cloud (EC2) [4]. Following the typical hardware setting in the research literature [23], we chose to keep the memory higher than the database size with a ratio roughly equally to 4. For this evaluation, we used of two kinds of EC2 instances: (1) t2.xlarge, which have 16GB of memory; and (2) m5.8xlarge, which have 128GB of memory. For both instances, we make use of 1TB of Elastic Block Store for disk storage.

Data model. For data generation, we use the standard DBMS benchmark Transaction Processing Performance Council H (TPC-H), which models data-driven decision support for business environments centered around a data warehouse scenario.

Data generation. We use the TPC-H benchmark of scale factor 1, which leads to about 8.6 million rows and $1GB$ of data. Each attribute value is sampled uniformly at random from its domain. All filtered and joined attributes are known a-priori by looking to the queries and the database schema. For KafeDB, we only index these specific attributes. We index the same filtered and joined attributes for the (plaintext) PostgreSQL evaluation. This indexing strategy helps to ensure the best possible query performance for both PostgreSQL and KafeDB.

Query generation. TPC-H specifies 22 queries that are common in the business environment. TPC-H queries are all complicated enough to require split execution between KafeDB client and server (Sec.5.2). To analyze the cost of our encryption scheme, we measure the query portion executed on the encrypted data on the KafeDB server and obmit the time spent on the KafeDB client for the post-decryption query portion. For the baseline, we measure the same query portion on the plaintext PostgreSQL as on the KafeDB server but in the clear. We summarize the composition of these query portions in Table 6.1, and refer the reader to

[1]We were limited to using AES in CBC mode because that is the only mode supported by PostgreSQL.

[6] for more details. All queries are run in a uniformly randomized order. The benchmark is first warmed up by executing all the TPC-H queries and discarding the results. The runtime is averaged over 10 runs with satisfactory relative standard error.

Composition	q1,6	q4,13,14	q12,16,22	q3,11	q17	q18	q19,20	q21	q8	q9	q10	q2	q5	q7
Filters	-	-	1	1	2	-	4	3	1	-	1	2	1	2
Joins	-	1	1	2	1	2	1	4	8	6	3	4	6	5

Table 5.1: Number of outsourced filters and joins after TPC-H query is processed for split execution.

Experimental setting. We want to compare the query performance of KafeDB to PostgreSQL in two different settings:

- *(equal memory)* in this setting, we use the same hardware for both systems. In particular, this experiment runs KafeDB and PostgreSQL on a m5.8xlarge EC2 instance.
- *(equal ratio)* in this setting, we the systems on different instances but keeping the ratio of memory to database size the same. Specifically, we run PostgreSQL on a t2.xlarge instance and KafeDB on a m5.8xlarge instance.

The goal of the first setting is to quantify the performance overhead incurred by KafeDB when run on the exact same hardware setup as PostgreSQL. In the second setting, we compare query performance between KafeDB and PostgreSQL while maintaining an equal memory to database size ratio, here equal to 4. This goal of this setting is maintain a similar level of caching effect between both systems.

5.4.1 Setup Time

In this section, we compare the setup time for KafeDB to the loading time of PostgreSQL. In general KafeDB requires more computation to build the encrypted data structures during emulation. In our implementation, many of the setup operations are parallelized over the 32 Virtual CPUs of the m5.8xlarge EC2 instance. Table 5.2 summarizes the setup time for a 1GB database for both KafeDB and PostgreSQL. In particular, KafeDB took 3210 seconds to setup the database whereas PostgreSQL only took 319 seconds to load the database. This amounts to a 10× slowdown. Note that our KafeDB setup benefits from the compound key and many-to-many join factorization optimizations. With neither, the setup ran out of storage. Without the latter, the setup took around 6 hours, where the additional overhead was spent on indexing directly over many-to-many relationships.

System	Time opt.(sec)	Time unopt.(hour)[1]
KafeDB	3210	6
PostgreSQL	319	-

[1] Without many-to-many key optimization.

Table 5.2: TPC-H setup time with scale factor 1.

5.4.2 Storage Overhead

The storage cost of KafeDB has three componenets: (1) the encrypted content tables; (2) the emulated encrypted multi-maps; and (3) the any additional needed indexing.

Total storage overhead. The size of the KafeDB and PostgreSQL databases are summarized in Table 5.3. KafeDB generates 8 encrypted content tables and 4 tables that represent the emulated encrypted multi-maps. PostgreSQL, on the other hand, stores only 8 plaintext content tables. The overall storage of KafeDB is 10× higher compared to PostgreSQL. If we consider only the contents tables, the overhead is 3.7× and this comes from the fact that, in KafeDB, values like integers or strings are encrypted as 128-bit blocks. As part of its emulation, KafeDB also generates a new column that stores the row identifiers for each content table. The storage overhead due to indexing the content tables is tiny in KafeDB compared to the indices used by PostgreSQL; namely less than 10%. This is because the indices in KafeDB are generated *only* for row identifiers, whereas PostgreSQL is set to index all the attributes that are relevant to the queries. On the other hand the emulated encrypted multi-maps, together with the indices they require, amount to about 12× the size of the indexes in PostgreSQL. Note that this larger ratio is due to using structured encryption as our underlying cryptographic primitive, whereas PostgreSQL uses standard indices such as B+trees and hash tables.

The ratio between contents and indices is similar for both KafeDB and PostgreSQL: both use more storage for their auxiliary structures, namely 86% for the emulated EMMs in KafeDB, and 70% for indexing in PostgreSQL. Because, intuitively speaking, the emulated EMMs play the role of indices over the encrypted contents, dedicating a large percentage of storage for them is not surprising. We defer the details of storage breakdown to Appendix 5.5.

System	Content tables	Indices	EMM tables	Indices	Total
KafeDB	4.88 (13%)	0.26 (0.7%)	21 (50%)	16 (36%)	42
PostgreSQL	1.32 (30%)	3.11 (70%)	-	-	4.5

Table 5.3: TPC-H storage breakdown in GB (ratio over total size).

Setting	1-20x	20-100x	> 100x	Min	Max	Median	Mean	90%-Trim Mean
Equal memory	7	8	7	1.4x	2232x	41x	188x	96x
Equal ratio	8	10	4	1.0x	1876x	31x	143x	63x

Table 5.4: Distribution of the slowdown of `TPC-H` encrypted queries.

5.4.3 Query Efficiency

We now examine the query efficiency of `KafeDB`.[2] Our results show that `KafeDB` incurs a slowdown of $1\times$ to $100\times$ for most queries and that query efficiency improves with more memory.

Equal memory vs. equal ratio. Table 5.4 describes the distribution of the slowdown incurred by `KafeDB` over `PostgreSQL` over all 22 `TPC-H` queries. In the equal memory setting, `KafeDB` runs 7 queries with a $20\times$ slowdown, 8 queries with a slowdown between $20\times$ to $100\times$, and another 7 queries with more than $100\times$ slowdown. In the equal ratio setting, this improves to 8 queries with a $20\times$ slowdown, 10 between $10\times$ to $100\times$, and only 4 beyond $100\times$. All queries see an improvement in the equal ratio setting, but some benefit more than others. For instance, `q20` improves by about $9\times$ but `q10` and `q11` only by $3\times$. Half of the queries run with less than $31\times$ slowdown. Note that the mean value, while important to assess, is considerably skewed in our case as 5 queries among the 22 have a higher impact on the average; for example, `q5` with a $2232\times$ slowdown, `q19`, `q20`, and `q21` with over a $280\times$ slowdown.

Granular query efficiency. As shown in Table 5.4, for both memory settings the majority of queries run within $1\times$ to $100\times$ slower on `KafeDB` than on `PostgreSQL`. A higher memory ratio benefits `KafeDB` as its 90%-trimmed average query time shortens from $96\times$ slowdown in the equal memory setting to $63\times$ in the equal ratio setting. We also want to emphasize that some queries do very well such as `q1`, `q16`, and `q17` with a slowdown of less than $6\times$. Due to space constraints, we defer the execution time of each query to Appendix 5.5.

Discussion. There are several reasons why `KafeDB`'s query time is higher than `PostgreSQL`. First, in `KafeDB` a SQL query is transformed to an OPX query and then emulated back into a SQL query which is usually more complex than the original query. Second, encryption prevents the underlying DBMS to use certain optimizations like bit-map indexes which require frequency information. Third, `KafeDB` requires the DBMS to query the emulated EMMs before being able to query the content tables which adds overhead.

[2]Recall that we report the execution time for *split* `TPC-H` queries; refer to the query generation paragraph or Sec. 5.2 for more details on the split execution.

5.4.4 Optimizations

We now evaluate the effectiveness of the optimizations proposed in Section ??. All experiments are conducted on a `m5.8xlarge` instance. To evaluate these optimizations, we extracted certain operations from the `TPC-H` queries and evaluated the operation on `KafeDB` with and without the optimization.

Push Select Through Join

To evaluate the push-select-through-join optimization we extracted the joins and filters from some of the `TPC-H` queries and reordered them. We list 4 such queries in Table 5.5 that represent varying numbers of joins and relationships. The results show that all queries perform better when the optimization is applied. The speedups vary from 4× to 53×.

Many-to-many Join Factorization

The many-to-many join factorization optimization is important `KafeDB` because many-to-many relationships may result in worst-case quadratic storage. This, in turn, can increase the memory footprint during query execution. For example in `TPC-H`, the `Customer` and `Supplier` over the same `Nation` is many-to-many.

Table 5.6 reports the time for `KafeDB` to compute a join between the `Customer` and `Supplier` tables with and without this optimization. The join produces 60 million rows from the 150 thousand rows of `Customer` and 10 thousand rows of `Supplier`. Although the result is only 4% of the worst-case quadratic join size, the unoptimized computation of the join (the first row the Table) took more than 24 hours. The optimized computation (second row of the Table) took only 12 minutes.

Multi-Way Join Flattening

Multi-way join flattening can be beneficial when one of the tables is small. This occurs, for example in the joins between `Supplier`, `Nation` and `Customer` of `TPC-H` which is a sub-query of `q5`. In this case, the primary key table, `Nation`, only has 25 rows which is less than 1% of the rows in the foreign key tables, `Customer` and `Supplier`.The original pipelined query plan joins `Supplier` with `Nation` and the result is then joined with `Customer`. The flattened plan is reported on the third line of Table 5.6 and shows an improvement of about 20×.

Query[1]	Mean(ms)	Relative Error [2]
$C \bowtie O \bowtie L \bowtie \sigma(N)$	163920.8	0.28%
$\sigma(N) \bowtie C \bowtie O \bowtie L$	30996.4	0.38%
$O \bowtie L \bowtie \sigma(C)$	152831.7	0.20%
$\sigma(C) \bowtie O \bowtie L$	32833.0	0.22%
$PS \bowtie \sigma(P) \bowtie S$	49309.6	0.55%
$\sigma(P) \bowtie PS \bowtie S$	919.1	1.86%
$L \bowtie \sigma(P) \bowtie S$	96383.2	0.38%
$\sigma(P) \bowtie L \bowtie S$	6176.4	0.28%

[1] Letters indicate initials of relation names.
[2] Standard error divided by mean

Table 5.5: `Push select through join` optimization.

Query[1]	Mean	Relative Error
$S \bowtie C$	> 24h	-
$S \bowtie N \bowtie C$	774956.4ms	1.33%
$(S \bowtie N) \bowtie (N \bowtie C)$	37757.1ms	0.91%

Table 5.6: Many-to-many join factorization and join flattening optimizations.

5.5 Detailed Evaluation Results

Query time. Table 5.7 reports the server execution time by `KafeDB` and the slowdown compared to `PostgreSQL` for each `TPC-H` query. Note that two hardware setups, the *equal memory* and the *equal ratio*, were evaulated to examine the effect of caching.

Storage. Table 5.8 summarizes the storage breakdown for each tables in `TPC-H` under the scale factor 1. The *selection table, join table, correlated join table, correlated selection table* are the result of the emulation of the EMMs in OPX.

			Slowdown	
Query	Mean (in ms)	Relative Error	Equal memory	Equal ratio
q1	2149.5	1.38%	1.4	1.0
q2	2445.4	0.52%	15.8	12.3
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
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
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

Table 5.7: `TPC-H` query server execution time and slowdown comparison between `KafeDB` and `PostgreSQL`.

	total(bytes)		index(bytes)		table(bytes)	
table	`KafeDB`	`PostgreSQL`	`KafeDB`	`PostgreSQL`	`KafeDB`	`PostgreSQL`
region	32k	64k	16k	48k	8192	8192
orders	604m	603m	45m	396m	559m	206m
supplier	3936k	4680k	328k	2840k	3600k	1832k
customer	64m	75m	4640k	46m	60m	29m
partsupp	302m	337m	24m	194m	278m	143m
nation	32k	80k	16k	64k	8192	8192
lineitem	4090m	3338m	181m	2419m	3909m	919m
part	82m	85m	6184k	55m	75m	30m
sel. table	13G	-	4926m	-	8439m	-
join table	6578m	-	3071m	-	3507m	-
cor. join table	8332m	-	3071m	-	5261m	-
cor. sel table	9G	-	4926m	-	4219m	-

Table 5.8: `TPC-H` storage size comparison between `KafeDB` and `PostgreSQL`.

Chapter 6

Optimal Efficiency and Leakage Reduction

6.1 Overview

In previous chapters, we proposed a new STE-based scheme OPX that allows for encrypted query optimization, which gives us potentially large constant factor reduction in query complexities. We then built a system `KafeDB` that uses emulation to implement OPX on top of any standard relational database without requiring any custom modification. However, we discovered two issues with the proposed solutions pertaining to efficiency and security.

Quadratic complexity. There are large gaps in terms of query and space efficiency between OPX/`KafeDB` and our baseline, PPE-based schemes CryptDB/Monomi. We identified two issues

1. The worst-case $O(\mathsf{DB}^2)$ cost for querying and space requirement.
2. The poor locality of access across multiple EMMs

Query leakage for filtered JOINs. The OPX scheme also does not address the query leakage for the filtered JOINs, which is the building block for conjunctive queries. For filtered JOINs, the OPX scheme leaks the join pattern for the full join, though only a subset of the full join is filtered on. This leakage can be potentially very large if the query involves multiple filtered JOINs over multiple attributes. For example, if there are filtered JOINs over m attributes, then the leakage of the query would be the join patterns over all rows for all m attributes. Such leakage would include the joint frequencies of all these attributes, which may be used to compute value frequencies in each attribute, and the value row collocation among these m attributes.

In this chapter, we study how to

1. Reduce the quadratic query and space complexity to linear in the plaintext database size.

2. Reduce the query leakage for filtered JOINs to just the joint pattern of the reuslt rows.
3. Use a new collocation technique to merge encrypted data structures to achieve better locality.

6.2 Reducing Join Compelxity through Surrogates

To reduce the STE-based join complexity from quadratic to linear for both time and space, we first investigate where this overhead comes from by looking at the representation of the join in the encrypted data structure. We will then build a better representation that will lead to less complexity.

6.2.1 Join Graphs

One way to represent the join is through a graphical representation, or a *join graph*. We show in an example in Figure 6.1. Join graph is a bipartie graph, where each party consists of nodes that represent rows in a table. If two rows are joined in the result, then there is an edge between their two corresponding nodes. In the worst case, each row in a table is paried with all rows in the other table, resulting in a cartesion product, or a *complete* bipartie graph. A complete bipartie graph has $\mathbf{T}_1 \cdot \mathbf{T}_2$ edges where $\mathbf{T}_1, \mathbf{T}_2$ are the number of nodes (or rows) in each party (or table).

Comparing the join graph and the EMMs in SPX and OPX, we find that essentially these structures store the *edges* of the join graph, which has worst-case quadratic cost in both query complexity (in terms of input relaiton size) and storage complexity. So the edge complexity of the join graph corresponds to the query and space complexity of an STE join.

6.2.2 Surrogate Join Graphs

To reduce the edge complexity, we notice that there is redundancy in the edges of the join graph. In particular, the join graph forms *partitions*, where each partition is a complete bipartie subgraph. Each node in a partition is connected to the same set of nodes, which is the other party.

We introduce a *surrogate* for each node. Nodes connected with the same surrogate are part of the same partition. We call the resulting graph a *surrogate join graph*. The surrogate join graph preserves the joint relationship in the join graph by establishing a bijection bewteen an edge in the join graph, and a path between the same two nodes (from different parties) in the surrogate join graph through the same surrogate. We call this path the *surrogate path*. We show the surrogate join algorithm in Figure 6.3.

Though the total number of nodes increases linearly through adding the surrogates, the edge complexity of the surrogate join graph however is reduced down to linear in the size of the tables, $\mathbf{T}_1 + \mathbf{T}_2$.

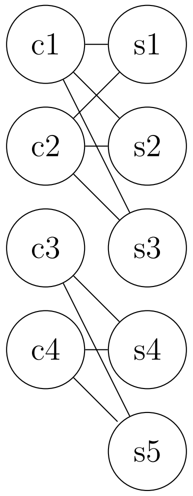
What are the surrogates? It turns out that they have one-to-one correspondence with the *joint values*. So why not just use the joint values? There is good reason why we choose to

Customer		
Id	Name	Nation
c1	Abe	US
c2	Bob	US
c3	Bob	Canada
c4	Cay	Canada

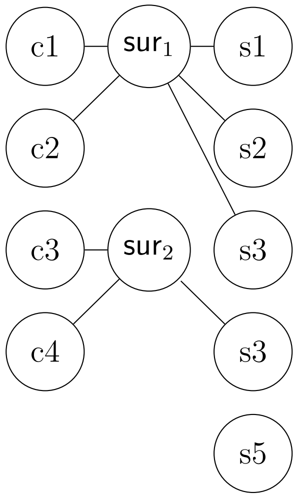
(a) Customer Table.

Supplier		
Id	Name	Nation
s1	Intel	US
s2	IBM	US
s3	Apple	US
s4	RIM	Canada
s5	WestJet	Canada

(b) Supplier Table.



(c) The join graph.



(d) The surrogate join graph.

Figure 6.1: Example of join $Customer \bowtie_{Nation} Supplier$.

think about them in such slightly abstract manner, becase later when we need to use them to build encrypted joins, we will only need to rely on this correspondence, not the values per se.

We need two encrypted multimaps to encode all the edges, because an edge in a join graph corresponds to a surrogate path in a surrogate join graph, and a surrogate path always consists of two edges. Essentially, we first factor the surrogate join graph into two bipartie graphs, where one of such graphs encodes $\mathbf{T}_1$ and S, and the other S and $\mathbf{T}_2$, for tables $\mathbf{T}_1, \mathbf{T}_2$ and surrogates S. Then we encode each graph in an EMM. We show this in an example in Figure 6.2.

6.3 Reducing Filtered Join Leakage

Filtered JOINs are the building block for the class of conjunctive queries. A filtered JOIN is essentially a join that is correlated with one or more filters. For simplicity we consider only one such correlated filter. Correlation between two operators here just mean that they operate on the same table. So for example,

```
SELECT *
FROM $\mathbf{T}_1$, $\mathbf{T}_2$, $\mathbf{T}_3$
WHERE $\mathbf{T}_1.\mathsf{att}_1 = T_2.\mathsf{att}_1$ AND $\mathbf{T}_1.\mathsf{att}_2 = c$
AND $\mathbf{T}_2.\mathsf{att}_3 = \mathbf{T}_3.\mathsf{att}_3$
```

In this example, the filtered join is between $\mathbf{T}_1, \mathbf{T}_2$, whereas $\mathbf{T}_2, \mathbf{T}_3$ is not a filtered join.

$$\mathsf{EMM}\begin{pmatrix} \text{C.Nation}\|\text{S.Nationn} & (c_1, s_1) \\ & (c_1, s_2) \\ & (c_1, s_3) \\ & (c_2, s_1) \\ & (c_2, s_2) \\ & (c_2, s_3) \\ & (c_3, s_4) \\ & (c_3, s_5) \\ & (c_4, s_4) \\ & (c_4, s_5) \end{pmatrix}$$

(a) SPX EMM_C.

$$\mathsf{EMM}\begin{pmatrix} c_1 & (c_1, s_1) \\ & (c_1, s_2) \\ & (c_1, s_3) \\ c_2 & (c_2, s_1) \\ & (c_2, s_2) \\ & (c_2, s_3) \\ c_3 & (c_3, s_4) \\ & (c_3, s_5) \\ c_4 & (c_4, s_4) \\ & (c_4, s_5) \end{pmatrix}$$

(b) OPX $\mathsf{EMM}_{\text{C.Nation,S.Nation}}$.

$$\mathsf{EMM}\begin{pmatrix} c_1 & \mathsf{sur}_1 \\ c_2 & \mathsf{sur}_1 \\ c_3 & \mathsf{sur}_2 \\ c_4 & \mathsf{sur}_2 \\ \mathsf{sur}_1 & (s_1, s_2, s_3) \\ \mathsf{sur}_2 & (s_4, s_5) \end{pmatrix}$$

(c) `pkfk` $\mathsf{EMM}_{\bowtie}$.

Figure 6.2: Example EMMs for the join in Fig. 6.1. Notice the redundancy in SPX and OPX EMMs for c_i's and s_j's, whereas `pkfk` EMM only store each c_i and s_j once. In general, each SPX and OPX stores worst-case $O(T^2)$ number of (c_i, s_j) pairs, whereas `pkfk` EMM only stores $O(T)$ number of them for table size T.

6.3.1 Join tokens in SPX and OPX

In STE-based schemes SPX, OPX, a filtered join leaks the joint pattern for the *entire* join. Can we confine the leakage to only the filtered join? First let us develope some understanding why such leakage happens.

The SPX scheme essentially encode each operation in an encrypted multimap independent of each other. This independence manifests by the way that the query tokens are constructed. The filter token for $\sigma_{\mathbf{T}_1.\mathsf{att}_2=c}$ is constructed as

$$\mathsf{tk}_\sigma = \mathcal{F}_{K_\sigma}(\mathbf{T}_1.\mathsf{att}_2 \| c)$$

The join token for $\bowtie_{\mathbf{T}_1.\mathsf{att}_1=\mathbf{T}_2.\mathsf{att}_1}$ is constructed as

$$\mathsf{tk}_\bowtie = \mathcal{F}_{K_\bowtie}(\mathbf{T}_1.\mathsf{att}_1 \| \mathbf{T}_2.\mathsf{att}_1)$$

So at query execution, both tokens need to be sent to the server and allow the server to

SurJoin($\mathbf{T} \bowtie_{\mathsf{att}=\mathsf{att}'} \mathbf{T}'$):

- On input relations $\mathbf{T}, \mathbf{T}'$, attributes $\mathsf{att}, \mathsf{att}'$

1. Designate a *surrogate* for each *domain value* in $\mathsf{att} \cup \mathsf{att}'$ conceptually. Semantically this is equivalent to a surrogate function

$$\mathsf{sur}(\mathbf{r}) := \mathcal{F}(\mathbf{r}), \forall \mathbf{r} \in \mathsf{att} \cup \mathsf{att}', \text{ where } \mathcal{F} \text{ is a random function}$$

   Notice that it is not necessary to precompute the join nor the union $\mathsf{att} \cup \mathsf{att}'$, but rather it just iterates over each value in each column once.

2. Set the bipartie subgraph $G = (V, E)$ for mapping $\mathbf{r} \Rightarrow \mathsf{sur}(\mathbf{r})$ where

$$V = \bigcup_{\mathbf{r}} \mathsf{sur}(\mathbf{r}) \bigcup_{\mathbf{r}} \mathbf{r}, \quad E = \bigcup_{\mathbf{r}} (\mathbf{r}, \mathsf{sur}(\mathbf{r})), \quad \forall \mathbf{r} \in \mathsf{att}$$

3. Set the bipartie subgraph $G' = (V', E')$ for mapping $\mathsf{sur}(\mathbf{r}') \Leftarrow \mathbf{r}'$ where

$$V' = \bigcup_{\mathbf{r}'} \mathsf{sur}(\mathbf{r}') \bigcup_{\mathbf{r}'} \mathbf{r}', \quad E' = \bigcup_{\mathbf{r}'} (\mathsf{sur}(\mathbf{r}'), \mathbf{r}'), \quad \forall \mathbf{r}' \in \mathsf{att}'$$

4. Return surrogate join graph as the union of subgraphs $G \cup G'$, or as the join of the edge relations via the surrogates $E \bowtie_{\mathsf{sur}} E'$

Figure 6.3: Algorithm to find surrogate join graph

compute the filtered row ids and all the joint row ids. Consequently, the server sees all the joint row ids and the joint pattern therein.

The inefficiency in the SPX scheme is improved by the OPX scheme, where the encrypted labels in the encrypted multimap are as a function on the input of not only the token and the frequency counter, but also the row ids,

$$\ell_{\bowtie} = \mathcal{F}_{\mathsf{tk}_{\bowtie}}(\texttt{row-id} \| \texttt{ctr})$$

where the join token is still the same as in SPX.

Alhough the access is reduced in OPX for filtered joins by avoiding computing the full join, the leakage on the other hand is not improved in OPX, because the server still gets hold of the join token that can potentially be used to compute the unfiltered part of the join given *other* queries. As long as some other unrelated query, say a filter on the same table on some other attribute, has revealed any row id of the unfiltered part, the server can still use the join token to discover the joint pattern on the unfiltered row id.

6.3.2 Conditioning the join token on the filter attribute

To reduce the leakage to just the filtered joint pattern, we have to further condition the token derivation on the filter value. For example, the join token for $\sigma_{\mathbf{T}_1.\mathsf{att}_2=c} \bowtie_{\mathbf{T}_1.\mathsf{att}_1=\mathbf{T}_2.\mathsf{att}_1}$ is derived based on three pieces of information, $\mathbf{T}_1.\mathsf{att}_2, \mathbf{T}_1.\mathsf{att}_1, \mathbf{T}_2.\mathsf{att}_1$,

$$\mathsf{tk}_{\bowtie,\mathsf{att}_2\|c} = \mathcal{F}_{K_{\bowtie}}(\mathbf{T}_1.\mathsf{att}_2 \| c \| \mathbf{T}_1.\mathsf{att}_1 \| \mathbf{T}_2.\mathsf{att}_2)$$

This EMM construction construction for filtered join does not have precedence in the literature yet, so we show the details of the construction for the running example in Figure 6.4. Notice that a filtered join leaks the joint pattern only for the filtered portion, for example query $C \bowtie_{Payment=Visa \wedge Industry=Phone} S$ would only leak the joint pattern for customer-supplier pair (c_1, s_3) for even malicious adversary, whereas for SPX and OPX all customer-supplier pairs would be leaked (for semi-honest and malicious adversary respectively).

By definition, a join can be filtered based on different attributes from the same table. This means that for each join and for each potentail filter, we need to precompute the tokens for the search token and the labels. The number of labels we need to compute is still linear in the table length (number of rows), though quadratic in the table width (number of attributes). In typical data warehouse, the table size is dominated by the table length rather than the table width. Moreoever, the table width can be viewed as a constant because the table schema rarely changes, whereas the table can grow in rows. So assuming the table width is a constant, the filtered join with the new leakage reduction applied is still asymtotically optimal in space.

$$\mathsf{EMM}\begin{pmatrix} Payment\|Visa & (c_1, c_2) \\ Payment\|Masters & (c_3, c_4) \\ Industry\|IT & (s_1, s_2) \\ Industry\|Phones & (s_3, s_4) \\ Industry\|Airline & (s_5) \end{pmatrix}$$

(a) Filter EMM.

$$\begin{bmatrix} \mathcal{F}(\mathsf{tk}_{\bowtie, Payment\|Visa}, c_1) & \mathsf{Enc}(\mathsf{tk}_{\bowtie, Payment\|Visa}, \mathsf{sur}_1) \\ \mathcal{F}(\mathsf{tk}_{\bowtie, Payrment\|Visa}, c_2) & \mathsf{Enc}(\mathsf{tk}_{\bowtie, Payment\|Visa}, \mathsf{sur}_1) \\ \mathcal{F}(\mathsf{tk}_{\bowtie, Payment\|Masters}, c_3) & \mathsf{Enc}(\mathsf{tk}_{\bowtie, Payment\|Masters} \mathsf{sur}_2) \\ \mathcal{F}(\mathsf{tk}_{\bowtie, Payment\|Masters}, c_4) & \mathsf{Enc}(\mathsf{tk}_{\bowtie, Payment\|Masters} \mathsf{sur}_2) \\ \mathcal{F}(\mathcal{F}(\mathsf{tk}_{\bowtie, Industry\|IT}, \mathsf{sur}_1), 1) & \mathsf{Enc}(\mathcal{F}(\mathsf{tk}_{\bowtie, Industry\|IT}, \mathsf{sur}_1), s_1) \\ \mathcal{F}(\mathcal{F}(\mathsf{tk}_{\bowtie, Industry\|IT}, \mathsf{sur}_1), 2) & \mathsf{Enc}(\mathcal{F}(\mathsf{tk}_{\bowtie, Industry\|IT}, \mathsf{sur}_1), s_2) \\ \mathcal{F}(\mathcal{F}(\mathsf{tk}_{\bowtie, Industry\|Phone}, \mathsf{sur}_1), 3) & \mathsf{Enc}(\mathcal{F}(\mathsf{tk}_{\bowtie, Industry\|Phone}, \mathsf{sur}_1), s_3) \\ \mathcal{F}(\mathcal{F}(\mathsf{tk}_{\bowtie, Industry\|Phone}, \mathsf{sur}_2), 1) & \mathsf{Enc}(\mathcal{F}(\mathsf{tk}_{\bowtie, Industry\|Phone}, \mathsf{sur}_2), s_4) \\ \mathcal{F}(\mathcal{F}(\mathsf{tk}_{\bowtie, Industry\|Airline}, \mathsf{sur}_2), 2) & \mathsf{Enc}(\mathcal{F}(\mathsf{tk}_{\bowtie, Industry\|Airline}, \mathsf{sur}_2), s_5) \end{bmatrix}$$

(b) `pkfk` Filtered join EMM.

Figure 6.4: Example of `pkfk` filtered join construction, where $\mathsf{tk}_{\bowtie} = \mathcal{F}(K_{\bowtie}, C.Nation\|S.Nation)$ and $\mathsf{tk}_{\bowtie,x} = \mathcal{F}(K_{\bowtie}, x\|C.Nation\|S.Nation)$. Notice that a filtered join leaks the joint pattern only for the filtered portion, for example query $C \bowtie_{Payment=Visa \wedge Industry=Phone} S$ would only leak the joint pattern for customer-supplier pair (c_1, s_3).

6.4 Encrypted Table

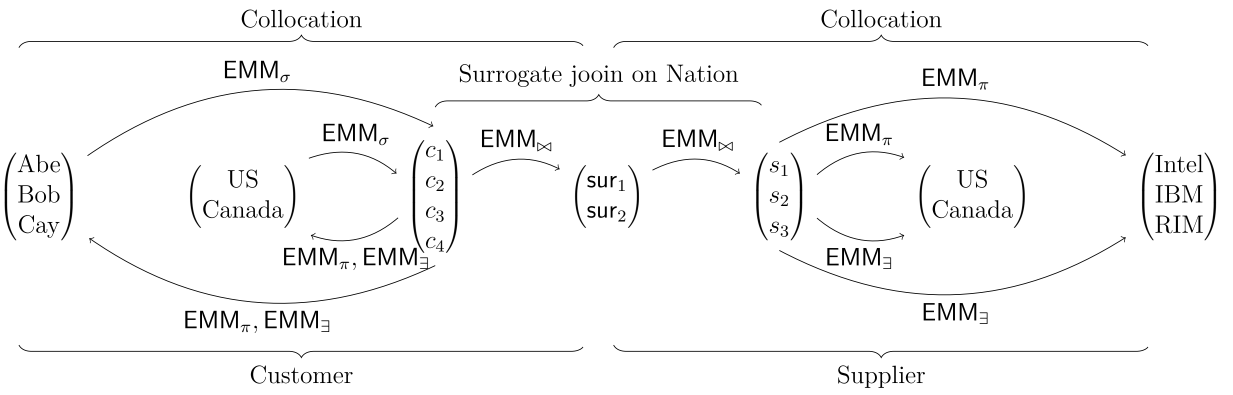


Figure 6.5: Example of the conceptual construction for the class of queries that joins Customer with Supplier on Nation with any conjunctive filters and projections. For instance, $\pi_{\text{C.name}}\sigma_{\text{C.nation=US}\wedge\text{C.name=Bob}}\text{Customer} \bowtie_{\text{nation}} \sigma_{\text{S.nation=US}}\text{Supplier}$. This query can be computed via chaining of $\mathsf{EMM}_\sigma, \mathsf{EMM}_\exists, \mathsf{EMM}_\bowtie, \mathsf{EMM}_\pi$ on Customer and $\mathsf{EMM}_\bowtie, \mathsf{EMM}_\exists, \mathsf{EMM}_\pi$ on Supplier.

.

Encrypted multimpas (EMMs) originated from the document-keyword model, where the typical application is document retrieval based on keywords. The operations an EMM supports are also very simple, such as retrieving the value(s) associatd with the label.

On the other hand, the data structure underlies the relational model is the table. A table supports multiple operations, such as filter on a column, projection on a column, or join (with another table) on a column.

The SPX and OPX schemes take the approach to use EMMs to describe table operations. For example a filter on a table is encoded as an EMM label-value pair where the label is based on the filter attribute and value, and the value is based on a list of elements from the column that matched the filter. A projection is encoded as a label-value pair where the label is based on the attribute name and the value is based on the list of elements in the column. The upside of this approach is that we can define the scheme on top of any existing EMM constructions, and build the leakage of the scheme on top of the EMMs.

However, The downside of this approach is that we no longer have access locality. What happens if the query projects on two attributes? To see this suppose we have a simple query that filters and projects on the same table

$$\sigma_{\mathbf{T}.\mathsf{att}_1=c}$$

6.5 Empirical Evaluation

In this section, we evaluate how `pkfk` performs in practice. We also compare `pkfk` against the recently introduced STE-based system `KafeDB` and PPE-based systems CryptDB [47] and Monomi [52]. In particular, we are interested in assessing the following efficiency metrics: (1) setup time, (2) query efficiency, and (3) storage efficiency. Our evaluation demonstrates that

1. `pkfk` achieves comparable query and storage to CryptDB while providing stronger security guarantees;

2. `pkfk` achieves one order of magnitude improvement over the previous state-of-the-art STE-based approach, `KafeDB`.

Implementation. The `pkfk` client and server implementations are based on a modification of `KafeDB`'s open-source implementation [6]. Specifically, the client uses and extends `Spark SQL`'s algebraic core for query translation and optimization, its parser to parse plaintext SQL queries into a query plan, and its executor to facilitate split execution. The `pkfk` server can be any DBMS but in this evaluation we use `PostgreSQL 9.6.2` [33]. Our implementation contains 1872 lines of codes calculated using CLOC [**?**] and is available for download in an anonymized form here [6]. For the cryptographic building blocks, we use `AES` in `CBC mode` with `PKCS7` padding for symmetric encryption, and `HMAC-SHA-256` for pseudo-random functions. Both primitives are provided by `Bouncy Castle 1.64` [44] in the `pkfk` client and by the `pgcrypto` module in `PostgreSQL 9.6.2`.[1]

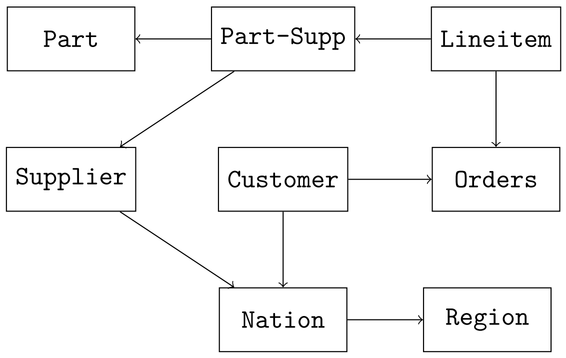
Testing environment. We conducted our experiments on Amazon Elastic Compute Cloud (`EC2`) [4]. Following the typical hardware setting in the research literature [23], we chose to keep the memory higher than the database size to accommodate more complex queries. We used `EC2` instance of type `t2.xlarge`, which is configured with 16GB of RAM and 1TB of `Elastic Block Store` for disk storage.

Data model. For data generation, we use the standard DBMS benchmark Transaction Processing Performance Council H (`TPC-H`), which models data-driven decision support for business environments centered around a data warehouse scenario.

The schema consists of 8 tables that represent real-world entities and relationships such as suppliers and customers, parts and orders. Figure 6.6a illustrates the schema as a directed graph where the nodes are tables, and each arrow represents a many-to-one relationship. In Table 6.6b, we provide a more detailed description of each table, including the number of attributes and the number of rows.

Data generation. The `TPC-H` benchmark, given a scale factor, automatically populates the database. We use the `TPC-H` benchmark of scale factor 1, which leads to about 8.6 million

[1] We were limited to using `AES` in `CBC mode` because that is the only mode supported by `PostgreSQL 9.6`.



(a) The boxes represent tables. A directed edge represent a many-to-one relationship.

Table	Number of attributes	Cardinality
`Part`	9	200×10^3
`Supplier`	7	10×10^3
`Part-Supp`	7	800×10^3
`Customer`	8	150×10^3
`Nation`	4	25
`Region`	3	5
`Lineitem`	17	6×10^6
`Orders`	9	1.5×10^6

(b) The cardinality is for $n = 1$ scale factor. For $n \geq 1$, multiply the cardinality by n.

Figure 6.6: Description of the `TPC-H` database schema [25].

rows and 4.4GB of data. Each attribute value is sampled uniformly at random from its domain. Note that all filtered and joined attributes are known a-priori by looking to the queries and the database schema.

Query generation. `TPC-H` specifies 22 queries that are common in the business environment. In the following, we use the notation $\texttt{q}\ell$ to refer to the ℓth query in the `TPC-H` benchmark. `TPC-H` queries are all complicated enough to require split execution between `pkfk` client and the server, refer to Section 5.2.[2] For our query efficiency evaluation, we measure the time spent on the portion of the query executed on the `pkfk` server, and omit the time spent on the `pkfk` client. The latter time is related to the decryption of the response as well as handling the unprocessed portion of the complex queries. Similarly, for our baseline comparison with `PostgreSQL`, we measure the time spent processing the same query portion in plaintext. We summarize the composition of these query portions in Table 6.1, and refer the reader to [6] for more details. All queries are run in a uniformly randomized order. The benchmark is first warmed up by executing all the `TPC-H` queries and discarding the results. The runtime is averaged over 10 runs.

Composition	q1,6	q4,13,14	q12,16,22	q3,11	q17	q18	q19	q20	q21	q8	q9	q10	q2	q5	q7
Filters	-	-	1	1	2	-	4	2	2	1	-	1	2	1	2
Joins	-	1	1	2	1	2	1	3	4	8	6	3	4	6	5

Table 6.1: Number of outsourced filters and joins after `TPC-H` query is processed on the server.

Comparisons. For the purpose of this evaluation, we also compare our efficiency numbers to those of CryptDB and Monomi from [52]. We note that the original CryptDB's system [47] only supports 4 out of the 22 `TPC-H` queries, so the results we recall here are from a modified

[2]Complex queries refer to range predicates, negation, disjunction, grouping and user-defined aggregations.

version of CryptDB in [52] that supports the full TPC-H. We also note that the hardware setup differs slightly in [52] where most noticeably the authors used a machine with slightly larger RAM of 24GB compared to the 16GB of RAM we use in our setting.[3] Since the code of [52] is not open-source, and in order to draw fair comparisons, we only report the query and storage multiplicative overheads incurred by these systems over a plaintext PostgreSQL. We also compare against KafeDB where we re-evaluated the open source implementation available in [6] on the same hardware setup.

6.5.1 Query Efficiency

We compare the relative slowdown for all TPC-H queries q1-q22 for STE-based systems KafeDB and pkfk, DET-based approaches CryptDB and Monomi, and plaintext PostgreSQL. The relative slowdown reflects the multiplicative overhead incurred by each system over the query efficiency of plaintext PostgreSQL. We summarize all results in Figure 6.7. Our results demonstrate that pkfk achieves comparable query efficiency to DET-based approach. In addition, with our new design and emulation techniques such as collocation and join order optimizations, we are one order of magnitude better, in terms of multiplicative factor, than state-of-the-art STE-based KafeDB. We provide below a more detailed comparison.

pkfk vs. DTE-based approaches. Compared to CryptDB, pkfk achieves comparable performance, while providing better security guarantees. The median slowdown for pkfk is only 4.2× over a plaintext PostgreSQL, comparable to the 3.92× in CryptDB and 1.24× in Monomi. In terms of the distribution, we noticed that more than half of the queries, 13 out of 22, in pkfk finished shorter than the median query time of CryptDB.

pkfk vs. STE-based approaches. pkfk improves over KafeDB on average by over an order of magnitude. The majority of the queries, 16 out of 22, in pkfk incur $1-10\times$ slowdown compared to plaintext PostgreSQL. Only one query, q19, finished a little over 100×. We attribute this improvement to the two novel techniques used in pkfk, the collocation and the join order optimization.

Optimizations. In order to better assess the efficiency impact of each of the pkfk techniques, we created an evaluation setup in which our techniques are gradually enabled. Figure 6.7b summarizes our results. In particular, we identified that pkfk with just collocation provides 2.94× speedup over KafeDB, with an additional 5.90× speedup when join order optimization is enabled. The effectiveness of collocation is mainly due to the increased locality where a query now operates on common indices and table rows. Note that KafeDB, in order to process a single query, requires operating on different indices and tables which lead to poor locality. On the other hand, join order optimization leverages the asymmetry of each join, which helps minimizing the iterations needed to reconstruct the join pairs. Note

[3]The authors in [52], however, stated that their evaluation numbers were similar across different hardware setups.

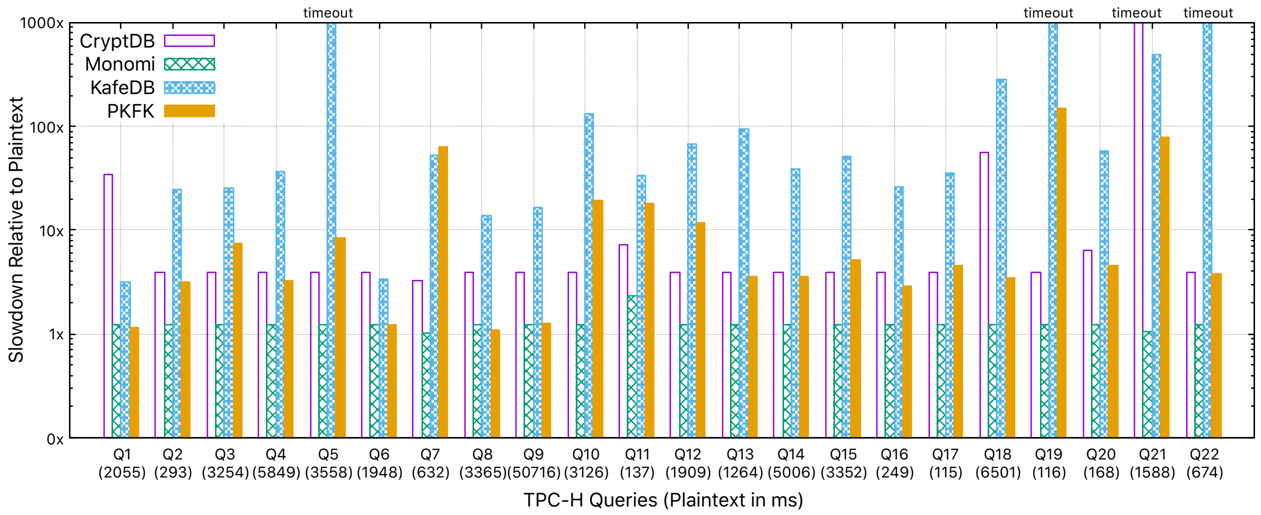
that more iterations leads to longer processing pipelines in the form of recursive common table expressions.

	CryptDB	Monomi	`KafeDB`	`pkfk`
Median	3.92×	1.24×	45.6×	4.2×
Min.	1.04×	1.03×	3.2×	1.09×
Max.	55.9×	2.33×	1407.9×	147.9×

(a) Summary of slowdown relative to plaintext

	collocation	join-order	total
Speedup	2.92×	5.90×	17.24×

(b) Improvement of techniques used in `pkfk` over `KafeDB`



(c) Comparison of `TPC-H` query slowdown

Figure 6.7: `TPC-H` query efficiency comparison for STE-based systems and DTE-based systems

6.5.2 Storage Overhead

We compare the storage overhead across different systems in Figure 6.8. Our results show that: (1) **`pkfk`** achieves comparable if not better storage overhead than DTE-based approaches, and (2) **`pkfk`** greatly reduces the storage overhead when compared to `KafeDB`. We provide below more details about our comparison.

`pkfk` vs. DTE-based approaches. Figure 6.8a shows that `pkfk` incurs a storage overhead of 3.64×, which is slightly lower than CryptDB which incurs 4.21× multiplicative overhead. Monomi achieves better storage overhead with only 1.72× blowup, mainly because Monomi partially uses format-preserving encryption scheme (FFX) that has weaker security.

pkfk vs. STE-based approaches. KafeDB requires 13.7× more storage then plaintext PostgreSQL. With pkfk, we are able to bring this blowup down to only 3.64× - which amount for a 72% improvement over KafeDB. This reduction is mainly due to the new design of the underlying structured encryption scheme, but also to the new emulation techniques. In particular, with pkfk, we avoid the quadratic complexity of joins' pre-processing as it becomes now only linear.

Storage breakdown. To better understand the storage overhead of pkfk, we provide in Figure 6.8c a more granular depiction of how storage overhead is distributed cross different components. In particular, we break the storage overhead down into four different components:

- (`enc-index table`): the encrypted index table - the emulated form of the encrypted indices;
- (`enc-index index`): the plaintext indices created on top of the encrypted index table;
- (`content table`): the tables containing the content of the database;
- (`content index`): the indices created on top of thecontent tables.

Note that, conceptually, the encrypted indexing in pkfk plays a similar role to the content indexing in plaintext PostgreSQL in that it facilitates efficient search. Compared to KafeDB, the significant storage reduction of the first two components, namely, the `enc-index table` and `enc-index index` components, is mainly due to the quadratic-to-linear storage improvement made possible by our new structured encryption scheme design. In particular, our results show that pkfk has over an order of magnitude reduction in `enc-index table` size. Such reduction can be attributed to the fact that collocated encrypted indices do not need to store duplicate cell values such as the row token identifier. Moreover, with the collocation, sharing indices becomes possible. Note that some indices become redundant and with pkfk, we can afford removing them without hurting the system functionalities or efficiency.

Remarks. In Figure 6.8b, we observe that pkfk achieves a balanced storage profile of indexing and contents, similar to that of the plaintext PostgreSQL. In particular, our results show that the relative ratio between indexing and contents in pkfk, namely the sum of all encrypted and plaintext indices over the content is about the same as that of PostgreSQL - around 70%. On the other hand, the ratio for KafeDB, around 90%, is much more skewed towards encrypted indices, which signifies a more index-intensive payload.

6.5.3 Setup Time

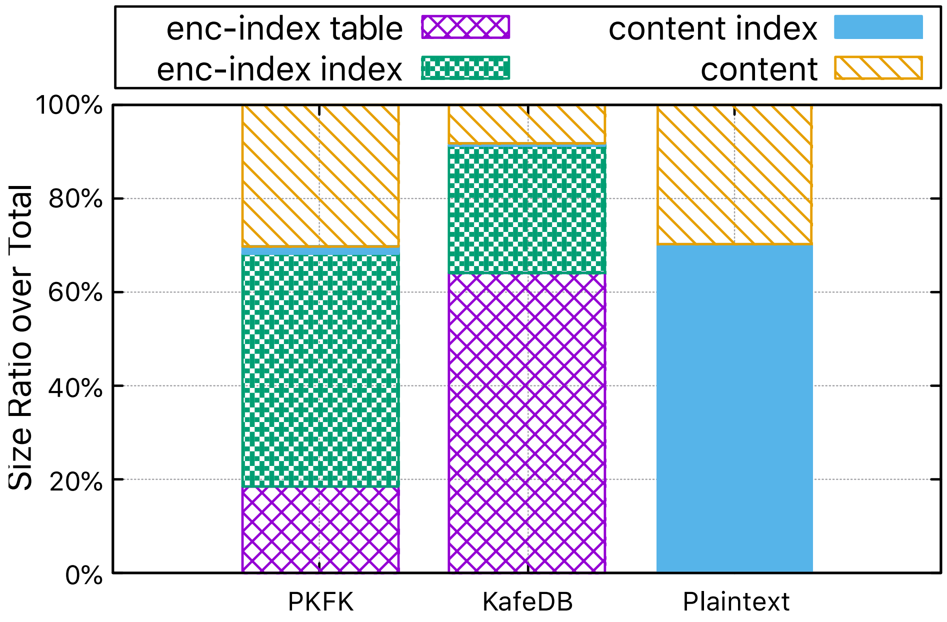
Figure 6.8a summarizes that the setup time of pkfk improves about 20% over KafeDB, from 10.3× to 8.2× over the plaintext setup time. The improvement can also be attributed to the new structured encryption design involving only a linear pre-processing of joins, instead of quadratic, as well as the collocation of encrypted data indices with the tables' content, which reduces the amount of data to stored and therefore written on disk.

6.6 The `pkfk` Protocol

We detail the pseudo-code of `pkfk` in Figures (**??**), (6.11), (**??**) and (**??**).

System	Size	Setup time
Plaintext	4.442GB	5.99min
CryptDB	4.21×	-
Monomi	1.72×	-
`KafeDB`	13.17×	10.37×
`pkfk`	3.63×	8.26×

(a) Storage and setup time.



(b) Storage breakdown in percentage.

System	total	enc-index table	enc-index index	content index	content table
`pkfk`	16131.1	2986.8	7998.4	260.0	4886.0
`KafeDB`	58506.9	37367.0	15994.0	260.0	4886.0
Plaintext	4442.7	n/a	n/a	3112.9	1329.8

(c) Storage breakdown in MB.

Figure 6.8: `TPC-H` storage size and setup time.

Let

- $\Sigma_{\mathsf{EDX}} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Get})$ be a response-revealing dictionary encryption scheme
- $\Sigma_{\mathsf{EMM}} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Get})$ be the response-revealing multimap encryption scheme
- $F : \{0,1\}^k \times \{0,1\}^\star \to \{0,1\}^\star$ be a pseudo-random function

The DB encryption scheme `pkfk` $= (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query}, \mathsf{Dec})$ is defined as follows [a]:

$\mathsf{Setup}(1^k, \mathsf{DB})$:

```
1. Sample client keys $K \leftarrow \$\{0,1\}^k, K_{\mathsf{sur}} \leftarrow \$\{0,1\}^k, K_{\bowtie} \leftarrow \$\{0,1\}^k, K_{\exists} \leftarrow \$\{0,1\}^k$.
2. For each table $\mathbf{T} \in \mathsf{DB}$ with $M$ rows and $N$ columns,
    (a) Permute $\mathbf{T}$ by rows and columns.
    (b) Let the table name and the attributes be anonymized.
    (c) Allocate a two-dimensional array $\mathsf{A}_{\mathbf{T}}$ with $M$ rows and $N$ columns, where $M, N$ are the
        row and column dimensions of $\mathbf{T}$.
    (d) For $j = 1, \cdots, N$,
        • For $i = 1, \cdots, M$,
            i. Assign a counter $P$ for each value $v = \mathbf{T}[i,j]$ to indicate that $v$ has duplicated
               $P$ times between rows $\mathbf{T}[1,j]$ to $\mathbf{T}[i,j]$
            ii. Compute the filter token and label
                $\mathsf{tk} \leftarrow \mathcal{F}_{K_\sigma}(\mathsf{att}_j \| \mathbf{T}[i,j]); \quad \ell_\sigma \leftarrow \mathcal{F}_{\mathsf{tk}}(P)$
            iii. Compute the existence filter token and label
                $\mathsf{tk} \leftarrow \mathcal{F}_{K_\exists}(\mathsf{att}_j \| \mathbf{T}[i,j]); \quad \ell_\exists \leftarrow \mathcal{F}_{\mathsf{tk}}(i)$
            iv. Allocate an empty list $S$ for storing join surrogate labels and ciphertexts.
```

[a]We omit the description of Dec because it simply decrypts every cell of the two-dimensioanl result array with the client key.

Figure 6.9: `pkfk` scheme, the setup algorithm

(Continued)

```
2. (d) • i. For each $\mathsf{att}_q \in \mathsf{DB}$ that are joinable with $\mathsf{att}_j$,
            – Compute the surrogate under the join
                  $\mathsf{sur} \leftarrow \mathcal{F}_{K_{\mathsf{sur}}}(\mathbf{T}[i,j] \| \mathsf{att}_j \| \mathsf{att}_q)$
            – Assign a counter $P$ for this surrogate $\mathsf{sur}$ to indicate the number of dulicated
              $\mathsf{sur}$ between rows $\mathbf{T}[1,j]$ and $\mathbf{T}[i,j]$
            – For the unfiltered join between $\mathsf{att}_j$ and $\mathsf{att}_q$,
                * Derive the token key
                      $\mathcal{K}_{\mathsf{tk}} \leftarrow \mathcal{F}_{K_{\bowtie}}(\mathsf{att}_j \| \mathsf{att}_q)$
                * Compute the token with $K_1$, encrypt the join surrogate, and append to
                  the list
                      $\mathsf{tk} \leftarrow \mathcal{F}_{\mathcal{K}_{\mathsf{tk}}}(i); \quad \mathrm{ct} \leftarrow \mathsf{Enc}_{\mathsf{tk}}(\mathsf{sur}); \quad S \leftarrow S \| \mathrm{ct}$
                * Compute for the other join direction with $K_2$
                      $\mathsf{tk} \leftarrow \mathcal{F}_{\mathcal{K}_{\mathsf{tk}}}(\mathsf{sur}); \quad \ell \leftarrow \mathcal{F}_{\mathsf{tk}}(P); \quad S \leftarrow S \| \ell$
            – For each $\mathsf{att}_r \in \mathbf{T}, 1 \leq r \leq N$ that may be filtered for the join between $\mathsf{att}_j$
              and $\mathsf{att}_q$,
                * Derive the token key
                      $\mathcal{K}_{\mathsf{tk}} \leftarrow \mathcal{F}_{K_{\bowtie}}(\mathbf{T}[i,r] \| \mathsf{att}_j \| \mathsf{att}_q)$
                * Compute the token with $K_1$, encrypt the join surrogate, and append to
                  the list
                      $\mathsf{tk} \leftarrow \mathcal{F}_{\mathcal{K}_{\mathsf{tk}}}(i); \quad \mathrm{ct} \leftarrow \mathsf{Enc}_{\mathsf{tk}}(\mathsf{sur}); \quad S \leftarrow S \| \mathrm{ct}$
                * Compute for the other join direction with $K_2$
                      $\mathsf{tk} \leftarrow \mathcal{F}_{\mathcal{K}_{\mathsf{tk}}}(\mathsf{sur}); \quad \ell \leftarrow \mathcal{F}_{\mathsf{tk}}(P); \quad S \leftarrow S \| \ell$
         ii. Set the cell in $\mathsf{A}_{\mathbf{T}}$ to the concatenation of the encrypted content, the filter label
             $\ell_\sigma$, the existence filter label $\ell_\exists$, and the join surrogate labels and ciphertexts $S$
                  $\mathsf{A}_{\mathbf{T}}[i,j] \leftarrow \mathsf{Enc}_{K_{\mathsf{DB}}}(\mathbf{T}[i,j]) \| \ell_\sigma \| \ell_\exists \| S$
       • Store the schema for each column in $\mathsf{A}_{\mathbf{T}}[\cdot,j]$ in a multimap
             $\mathsf{MM}_{\mathsf{Schm}}[\mathbf{T},j] = \mathsf{att}_j \| \mathsf{att}_\sigma \| \mathsf{att}_\exists \| \{(\mathsf{att}_j \| \mathsf{att}_q) \| (\mathsf{att}_q \| \mathsf{att}_j)\}_q$
         where $(\mathsf{att}_q)_q$ are the list of attributes joinable to $\mathsf{att}_j$.
3. Output $K = (K_{\mathsf{DB}}, K_{\mathsf{sur}}, K_\sigma, K_\exists, K_{\bowtie}), \mathsf{EDB} = (\{\mathsf{A}_{\mathbf{T}}\}, \mathsf{MM}_{\mathsf{Schm}})$
```

Figure 6.10: **pkfk** scheme, the setup algorithm (cont.)

$\mathsf{Token}(K, \mathsf{QT})$:

- Input: client keys K, a query tree QT against DB.
- Output: a token tree TT.

1. For each node N in QT processed in post-order,
   - Let $\mathsf{QT}_{\mathbf{in}}$ denote the schema of the subtrees.
   - Let the attribute names and table names in QT be already anonymized.
   - If $N \equiv \mathbf{T}$, a table scan of table $\mathbf{T}$, then set
     $$\mathsf{TT}_N \leftarrow \mathsf{A}_{\mathbf{T}}$$
   - If $N \equiv \pi_{\mathbf{T}.\mathsf{att}} \mathsf{QT}_{\mathbf{in}}$, then set the token tree node
     $$\mathsf{TT}_N \leftarrow \mathsf{project}(\mathbf{T}.\mathsf{att})$$
2. If $N \equiv \sigma_{\mathbf{T}.\mathsf{att}=v} \mathsf{QT}_{\mathbf{in}}$, then
   - If $\mathbf{T}.\mathsf{att} = v$ is the first filter on $\mathbf{T}$ in $\mathsf{QT}_{\mathbf{in}}$, then compute the token and set the token tree node
     $$\mathsf{tk} \leftarrow \mathcal{F}_{K_\sigma}(\mathsf{att} \| v); \quad \mathsf{TT}_N \leftarrow \mathsf{filter}(\mathsf{att}, \mathsf{tk})$$
   - Else, compute the token key and set the token tree node
     $$K_{\mathsf{tk}} \leftarrow \mathcal{F}_{K_\exists}(\mathsf{att} \| v); \quad \mathsf{TT}_N \leftarrow \mathsf{filter\text{-}conj}(\mathsf{att}, K_{\mathsf{tk}})$$
3. If $N \equiv \mathsf{QT}^{(l)}_{\mathbf{in}} \bowtie_{\mathbf{T}_l.\mathsf{att}_l = \mathbf{T}_r.\mathsf{att}_r} \mathsf{QT}^{(r)}_{\mathbf{in}}$, a join, then
   - If exists a correlated filtered attribute $\mathbf{T}_l.\mathsf{att}^{(l)}_\sigma = v_l$ in $\mathsf{QT}^{(l)}_{\mathbf{in}}$, then compute the token key input
     $$\mathbf{c}_l = \mathsf{att}^{l}_{\sigma} \| \mathsf{att}_l \| \mathsf{att}_r$$
   - If exists a correlated filtered attribute $\mathbf{T}_r.\mathsf{att}^{(r)}_\sigma = v_r$ in $\mathsf{QT}^{(r)}_{\mathbf{in}}$, then compute the token key input
     $$\mathbf{c}_r = \mathsf{att}^{(r)}_{\sigma} \| \mathsf{att}_r \| \mathsf{att}_l$$
   - Compute the token keys and set the token tree node
     $$K^l_{\mathsf{tk}}, K^r_{\mathsf{tk}} \leftarrow \mathcal{F}_{K_\bowtie}(\mathbf{c}_l), \mathcal{F}_{K_\bowtie}(\mathbf{c}_r); \quad \mathsf{TT}_N \leftarrow \mathsf{join}(\mathbf{c}_l \| \mathbf{c}_r, K^{(l)}_{\mathsf{tk}} \| K^{(r)}_{\mathsf{tk}})$$
4. Output the token tree TT

Figure 6.11: `pkfk` scheme, the token algortihm

```
Query(TT, EDB)
  • Input: a token tree $\mathsf{TT}$, the encrypted database $\mathsf{EDB} = (\{\mathsf{A}_{\mathbf{T}}\}, \mathsf{MM}_{\mathsf{Schm}})$
  • Output: a query result table $\mathbf{R}$ against $\mathsf{EDB}$
  1. For each node $\mathsf{TT}_N$ in the token tree $\mathsf{TT}$ processed in post order,
      • Let $\mathbf{R}_{\mathbf{in}}$ be the result table of the subtree of $\mathbf{R}_N$
      • If $\mathsf{TT}_N \equiv \mathsf{A}_{\mathbf{T}}$, then set the query node
            $\mathbf{R}_N \leftarrow \mathsf{A}_{\mathbf{T}}$
      • Else if $\mathsf{TT}_N \equiv \mathsf{project}(\mathbf{T}.\mathsf{att})$, retain said column,[a]
            $\mathbf{R}_N \leftarrow \mathbf{R}_N[\cdot, \mathbf{T}.\mathsf{att}]$
      • Else if $\mathsf{TT}_N \equiv \mathsf{filter}(\mathsf{att}, \mathsf{tk})$, then
        (a) Initialize
            $\mathbf{R}_N \leftarrow \mathbf{R}_{\mathbf{in}}$
        (b) For $P = 1 \cdots$ until $\mathbf{R}_P = \emptyset$,
            – Filter and set $\mathbf{R}_N$
            $\mathbf{R}_P \leftarrow \big\{\mathbf{R}_N[i,\cdot] \mid \forall i, \mathbf{R}_N[i,\mathsf{att}][\mathsf{att}_\sigma] = \mathcal{F}_{\mathsf{tk}}(P)\big\}; \quad \mathbf{R}_N \leftarrow \mathbf{R}_P$
      • Else if $\mathsf{TT}_N \equiv \mathsf{filter\text{-}conj}(\mathsf{att}, K_{\mathsf{tk}})$, filter and set $\mathbf{R}_N$
            $\mathbf{R}_N \leftarrow \big\{\mathbf{R}_N[i,\cdot] \mid \forall i, \exists \mathbf{R}_{\mathbf{in}}[i,\mathsf{att}][\mathsf{att}_\exists] = \mathcal{F}_{\mathsf{tk}}(i)\big\}$
      • Else if $\mathsf{TT}_N \equiv \mathsf{join}(\mathbf{c}_l \| \mathbf{c}_r, K_l \| K_r)$, join and set $\mathbf{R}_N$
        (a) Let $\mathbf{T}_l, \mathbf{T}_r$ be the tables that contain $\mathsf{att}_l, \mathsf{att}_r$ repsectively.
        (b) Initialize $\mathbf{R}_l \leftarrow \mathbf{R}_{\mathbf{in}}^{(l)}, \mathbf{R}_r \leftarrow \mathbf{R}_{\mathbf{in}}^{(r)}$.
        (c) For each $i$-th cell in the $\mathsf{att}_l$ column $\mathbf{R}_l[\cdot, \mathsf{att}_l]$, find its original row coordinate in $\mathsf{A}_{\mathbf{T}_l}$
            (for example by carrying the original coordinate around),
            $r_i \leftarrow \mathsf{RowCoord}_{\mathsf{A}_{\mathbf{T}_l}}(\mathbf{R}_l[i, \mathsf{att}_l])$
        (d) Decrypt each $i$-th surrogate located as the $\mathbf{c}_l$-th element in $\mathbf{R}_l[i, \mathsf{att}_l]$,
            $\mathsf{tk}_i \leftarrow \mathcal{F}_{K_l}(r_i); \quad \mathsf{sur}_i \leftarrow \mathsf{Dec}_{\mathsf{tk}_i}(\mathbf{R}_l[i, \mathsf{att}_l][\mathbf{c}_l]); \quad \mathbf{R}_l[i, \mathsf{att}_l][\mathbf{c}_l] \leftarrow \mathsf{sur}_i$
        (e) For each $i$-th row in $\mathbf{R}_l$, join rows in $\mathbf{R}_r$ with the same surrogate,
            – Compute the $i$-th token for the $i$-th surrogate,
            $\mathsf{tk}_i \leftarrow \mathcal{F}_{K_r}(\mathsf{sur}_i)$
            – For $P = 1, \cdots$, until $\Delta\mathbf{R}_P = \emptyset$,
            $\Delta\mathbf{R}_P \leftarrow \big\{\mathbf{R}_r[i', \cdot] \mid \exists i', \mathbf{R}_r[i', \mathsf{att}_r][\mathbf{c}_r] = \mathcal{F}_{\mathsf{tk}_i}(P)\big\}; \quad \mathbf{R}_P \leftarrow \mathbf{R}_P \cup \Delta\mathbf{R}_P$
            – Pair the $i$-th row in $\mathbf{R}_l$ once with each row in $\mathbf{R}_P$,
            $\mathbf{R}_i \leftarrow \big\{\mathbf{R}_l[i, \cdot] \| \mathbf{R}_P[i', \cdot] \mid \forall i' \in [\#\mathrm{rows}(\mathbf{R}_P)]\big\}$
        (f) Set
            $\mathbf{R}_N \leftarrow \{\mathbf{R}_i \mid \forall i \in [\#\mathrm{rows}(\mathbf{R}_l)]\}$
```

[a] As mentioned in Setup, we use the attribute name to indicate the column index and the list index that can be computed from $\mathsf{MM}_{\mathsf{Schm}}$.

Figure 6.12: `pkfk` scheme, the query algorithm

Chapter 7

Conclusion

Appendix A

Appendix

A.1 Proof of Theorem 4.3.1

Theorem 4.3.1. *If the query algorithm of Σ_{mm} is optimal, then the time and space complexity of the* Query *algorithm presented in Section (??) is optimal.*

Proof. A query tree QT can be composed of four different types of nodes: (1) a cross-product node xnode, (2) a projection node pnode, (3) a selection node snode, and a (4) a join node jnode. We will show that for each type of nodes, the search and space complexity on plaintext text relational database is asymptotically equal to the search and space complexity required by the Query algorithm of opx. We assume in this proof that the plaintext database has indices to speed-up lookup operations on every attribute.

- **(case 1):** if the node is a cross-product node, then the output of the node, xnode, in a plaintext database given a left and a right input $\mathbf{R}_{\mathbf{in}}^{(l)}$ and $\mathbf{R}_{\mathbf{in}}^{(r)}$, respectively, is equal to

  $$\mathbf{R}_{\mathbf{out}} = \mathbf{R}_{\mathbf{in}}^{(l)} \times \mathbf{R}_{\mathbf{in}}^{(r)},$$

  which is the exact same operation performed by the Query algorithm of opx when the node is a cross-product node.

- **(case 2):** if the node is a projection node, then there are two possible cases. If the node pnode has form $\pi_{\mathsf{att}}(\mathbf{T})$, a leaf projection node, then a plaintext database will require a work linear in $O(m)$ to fetch all the cells of the attribute att and where m is the number of cell in the column. On the other hand, opx performs a Query operation on EMM_C to fetch the corresponding encrypted cells. Assuming that Σ_{MM} has an optimal search complexity, the amount of work is also linear in $O(m)$.[1]

  The second case is when the projection node has form $\pi_{\mathsf{att}}(\mathbf{R}_{\mathbf{in}})$, an interior projection node. In this case, a plaintext database will simply select the corresponding columns

[1]Note that we are not accounting for the security parameter in our computation and only focusing on the number of cells.

from the input $\mathbf{R_{in}}$ which has search complexity equal to $O(\#\mathbf{R_{in}}[\mathsf{att}])$ which is the number of cells of the attribute att in $\mathbf{R_{in}}$. In the Query algorithm of opx, the exact same operation is performed and therefore, the same complexity is required.

- **(case 3):** if the node is a selection node, there there are three possible cases. If the node snode has form $\sigma_{\mathsf{att}=a}(\mathbf{T})$, a leaf selection node, then a plaintext database will require a work linear in $O(\#\mathsf{DB}_{\mathsf{att}=a})$ which is the number of cells in the attribute att equal to a. On the other hand, opx performs a Query operation on EMM_C to fetch the corresponding cells in $\mathsf{DB}_{\mathsf{att}=a}$. Assuming that Σ_{MM} has an optimal search complexity, then the amount of work is equal to $O(\#\mathsf{DB}_{\mathsf{att}=a})$.

  The second case is when the selection node has form $\sigma_{\mathsf{att}=a}(\mathbf{R_{in}})$, an interior selection node. In this case, a plaintext database has to go linearly over the entire column att in $\mathbf{R_{in}}$ to only output the rows in $\mathbf{R_{in}}$ with the cell at the attribute att equal to the constant a. That is, the search complexity is equal to $O(\mathbf{R_{in}}[\mathsf{att}])$. On the other hand, opx tests for each row in $\mathbf{R_{in}}[\mathsf{att}]$ whether it exists in SET. Assuming that test membership in SET is optimal, then the search complexity is equal to $O(\mathbf{R_{in}}[\mathsf{att}])$

  The third case is when the selection node has form $\sigma_{\mathsf{att}_1=\mathsf{att}_2}(\mathbf{R_{in}})$, an interior variable select node. In this case, a plaintext will simply remove any row in $\mathbf{R_{in}}$ such that the cell values are not equal. This has search complexity equal to $O(\#\mathbf{R_{in}}[\mathsf{att}_1])$. On the other hand, opx similarly removes all rows that have equal equal cell value at both columns att_1 and att_2. Clearly, the plaintext and encrypted operations have the same search and space complexity.

- **(case 4):** if the node is a join node, then there are two possible cases. If the node jnode has form $\mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{T}_2$, then a plaintext database would at least require $O(\#\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2})$ which is the result of the join operation on the columns att_1 and att_2. On the other hand, opx queries $\mathsf{EMM}_{\mathsf{att}_1}$ to fetch the join result. Assuming that Σ_{MM} has an optimal search complexity, then the search complexity is equal to $O\#(\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2})$.

  The second case occurs when the join node has form $\mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R_{in}}$, an interior join node. In this case, a plaintext database has to go over every cell at the attribute att_2 and checks if there are any rows in table $\mathbf{T}$ at attribute att_1 that are equal to the value in the selected cell. The search complexity is equal to

  $$O\Bigg(\max(\#\mathbf{R_{in}}[\mathsf{att}_2], \#\mathbf{R_{out}}[\mathsf{att}_2])\Bigg),$$

  which is itself equal to the maximum value of either (1) the number of cells in $\mathbf{R_{in}}[\mathsf{att}]$ or (2) the size of joinable rows which is equal to $\#\mathbf{R_{out}}[\mathsf{att}_2]$ (or equivalently to $\#\mathbf{R_{out}}[\mathsf{att}_1]$). On the other hand, opx queries $\mathsf{EMM}_{\mathsf{att}_1,\mathsf{att}_2}$ to fetch the joinable result. Similar to the plaintext scenario, opx will for each row token in $\mathbf{R_{in}}[\mathsf{att}_2]$ fetch the joinable rows, if any, from $\mathsf{EMM}_{\mathsf{att}_1,\mathsf{att}_2}$. Since $\Sigma^{\pi}_{\mathsf{MM}}$ has an optimal search complexity, then the search complexity is equal to $O(\max(\#\mathbf{R_{in}}[\mathsf{att}_2], \#\mathbf{R_{out}}[\mathsf{att}_2]))$ as the same operation is performed.

Finally, opx will query EMM_R to retrieve all the encrypted rows corresponding to the rows tokens in $\mathbf{R}^{\mathsf{root}}_{\mathbf{in}}$. Assuming that Σ_{mm} has an optimal search complexity, then this step will require $O(\#\mathbf{R}^{\mathsf{root}}_{\mathbf{in}})$. Note that this operation would add exactly the same complexity as the sum of the output size of the child nodes, and therefore would not have an impact on the final asymptotic result.

To sum up, we have shown that whatever the type of the node, both the plaintext and opx query algorithm executions require the same space and search complexities.

■

A.2 Proof of Theorem 4.4.1

Theorem 4.4.1. *If F is a pseudo-random function, SKE is RCPA secure, $\Sigma^{\pi}_{\mathsf{MM}}$ is adaptively $\left(\mathcal{L}^{\pi}_{\mathsf{S}}, \mathcal{L}^{\pi}_{\mathsf{Q}}\right)$-secure, and Σ_{MM} is adaptively $\left(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\right)$-secure, then opx is adaptively $(\mathcal{L}^{\mathsf{opx}}_{\mathsf{S}}, \mathcal{L}^{\mathsf{opx}}_{\mathsf{Q}})$-secure in the random oracle model.*

Proof. Let $\mathcal{S}_{\mathsf{MM}}$ and $\mathcal{S}^{\pi}_{\mathsf{MM}}$ be the simulators guaranteed to exist by the adaptive security of Σ_{MM} and $\Sigma^{\pi}_{\mathsf{MM}}$ and consider the OPX simulator $\mathcal{S}$ that works as follows. Given $\mathcal{L}^{\mathsf{opx}}_{\mathsf{S}}(\mathsf{DB})$, $\mathcal{S}$ simulates EDB by computing $\mathsf{EMM}_R \leftarrow \mathcal{S}_{\mathsf{MM}}\left(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_R)\right)$, $\mathsf{EMM}_C \leftarrow \mathcal{S}_{\mathsf{MM}}\left(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_C)\right)$, $\mathsf{EMM}_V \leftarrow \mathcal{S}_{\mathsf{MM}}\left(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_V)\right)$, for all $\mathbf{c} \in \mathsf{DB}^{\intercal}$, $\mathsf{EMM}_{\mathbf{c}} \leftarrow \mathcal{S}_{\mathsf{MM}}\left(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_{\mathbf{c}})\right)$, and for all $\mathbf{c}, \mathbf{c}' \in \mathsf{DB}^{\intercal}$, $\mathsf{EMM}_{\mathbf{c},\mathbf{c}'} \leftarrow \mathcal{S}^{\pi}_{\mathsf{MM}}\left(\mathcal{L}^{\pi}_{\mathsf{S}}(\mathsf{MM}_{\mathbf{c},\mathbf{c}'})\right)$. Given (n, ρ), it instantiates an empty set SET, and inserts $r_{i,j} \overset{\$}{\leftarrow} \{0,1\}^k$ in SET for $i \in [n]$ and $j \in [\rho]$. $\mathcal{S}$ outputs

$$\mathsf{EDB} = (\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, (\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c}\in\mathsf{DB}^{\intercal}}, \mathsf{SET}, (\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^{\intercal}}).$$

Recall that OPX is response-hiding so $\mathcal{S}$ receives $\left(\perp, \mathcal{L}^{\mathsf{opx}}_{\mathsf{Q}}(\mathsf{DB}, \mathsf{QT})\right)$ as input in the $\mathbf{Ideal}_{\mathrm{SPX},\mathcal{A},\mathcal{S}}(k)$ experiment. Given this input, $\mathcal{S}$ parses $\mathcal{L}^{\mathsf{opx}}_{\mathsf{Q}}(\mathsf{DB}, \mathsf{QT})$ as a leakage tree. It then instantiates a token tree TT with the same structure. It samples uniformly at random a key $K_1 \overset{\$}{\leftarrow} \{0,1\}^k$, and creates s set $\mathsf{SET}^{\star}$ such that $\mathsf{SET}^{\star} := \mathsf{SET}$. For each node N, retrieved in a post-order traversal from the leakage tree, it simulates the corresponding node in the token tree TT as follows.

- **(Cross product).** If N has form $\left(\mathtt{scalar}, |a|\right)$ then it sets TT_N to $[\mathsf{Enc}_{K_1}(\mathbf{0}^{|a|})]$. Otherwise if N has form $\left(\mathtt{cross}, \perp\right)$, then it sets TT_N to $\times$.

- **(Projection).** If N has form $\left(\mathtt{leaf}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\left(\mathsf{MM}_C, \chi(\mathsf{att})\right)\right)$ then it sets

$$\mathsf{TT}_N \leftarrow \mathcal{S}_{\mathsf{MM}}\left(\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\left(\mathsf{MM}_C, \chi(\mathsf{att})\right)\right),$$

  If N has form $\left(\mathtt{in}, f(\mathsf{att}_1), \cdots, f(\mathsf{att}_z)\right)$, then it sets TT_N to $\left(f(\mathsf{att}_1), \cdots, f(\mathsf{att}_z)\right)$.

- **(Selection case-1).** If N has form

$$\left(\texttt{leaf}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\left(\mathsf{MM}_V, \left\langle a, \chi(\mathsf{att})\right\rangle\right), \left(\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}_R, \chi(\mathbf{r})\right)_{\mathbf{r}\in\mathsf{DB}_{\mathsf{att}=a}}\right)$$

then it first sets for all $\mathbf{r} \in \mathsf{DB}_{\mathsf{att}=a}$,

$$\mathsf{rtk}_{\mathbf{r}} \leftarrow \mathcal{S}_{\mathsf{MM}}\left(\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}_R, \chi(\mathbf{r}))\right),$$

then it sets,

$$\mathsf{TT}_N \leftarrow \mathcal{S}_{\mathsf{MM}}\left(\left(\mathsf{rtk}_{\mathbf{r}}\right)_{\mathbf{r}\in\mathsf{DB}_{\mathsf{att}=a}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\left(\mathsf{MM}_V, \left\langle a, \chi(\mathsf{att})\right\rangle\right)\right).$$

- **(Selection case-2).** If N has form

$$\left(\texttt{in}, f(\mathsf{att}), g(a\|\mathsf{att}), \left(\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}_R, \chi(\mathbf{r})\right)_{\chi(\mathbf{r})\in\mathbf{R_{in}}\wedge\mathbf{r}[\mathsf{att}]=a}\right)$$

then if $g(a\|\mathsf{att})$ has never been revealed before,

  - for all $\mathbf{r} \in \mathsf{DB}$ such that $\chi(\mathbf{r}) \in \mathbf{R_{in}}$ and $\mathbf{r}[\mathsf{att}] = a$, it sets

$$\mathsf{rtk}_{\mathbf{r}} \leftarrow \mathcal{S}_{\mathsf{MM}}\left(\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}_R, \chi(\mathbf{r}))\right)$$

  - it samples a key $K_{g(a\|\mathsf{att})} \overset{\$}{\leftarrow} \{0,1\}^k$;
  - for each $\mathbf{r} \in \mathsf{DB}$ such that $\chi(\mathbf{r}) \in \mathbf{R_{in}}$ and $\mathbf{r}[\mathsf{att}] = a$, it picks and removes uniformly at random a value r in $\mathsf{SET}^\star$ and sets

$$H(K_{g(a\|\mathsf{att})}\|\mathsf{rtk}_{\mathbf{r}}) := r;$$

  - it sets

$$\mathsf{TT}_N \leftarrow (K_{g(a\|\mathsf{att})}, f(\mathsf{att})).$$

Otherwise, if $g(a\|\mathsf{att})$ has been revealed before then,

  - for all $\mathbf{r} \in \mathsf{DB}$ such that $\chi(\mathbf{r}) \in \mathbf{R_{in}}$ and $\mathbf{r}[\mathsf{att}] = a$, it sets

$$\mathsf{rtk}_{\mathbf{r}} \leftarrow \mathcal{S}_{\mathsf{MM}}\left(\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}_R, \chi(\mathbf{r}))\right)$$

  - for all $\mathbf{r} \in \mathsf{DB}$ such that $\chi(\mathbf{r}) \in \mathbf{R_{in}}$ and $\mathbf{r}[\mathsf{att}] = a$, if $H(K_{g(a\|\mathsf{att})}\|\mathsf{rtk}_{\mathbf{r}})$ has not been set yet, then it picks and removes uniformly at random a value $r \in \mathsf{SET}^\star$ and sets

$$H(K_{g(a\|\mathsf{att})}\|\mathsf{rtk}_{\mathbf{r}}) := r;$$

– it sets

$$\mathsf{TT}_N \leftarrow (K_{g(a\|\mathsf{att})}, f(\mathsf{att})).$$

- **(Join case-1).** If N has form

$$\Bigg(\texttt{leaf}, f(\mathsf{att}_1), \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\Bigg(\mathsf{MM}_{\mathsf{att}_1}, \Bigg\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2)\Bigg\rangle\Bigg),$$
$$\Bigg\{\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}_R, \chi(\mathbf{r}_1), \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}_R, \chi(\mathbf{r}_2)\Bigg\}_{(\mathbf{r}_1,\mathbf{r}_2)\in\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}}\Bigg),$$

then it sets for all $(\mathbf{r}_1, \mathbf{r}_2) \in \mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}$,

$$\mathsf{rtk}_1 \leftarrow \mathcal{S}_{\mathsf{MM}}\Bigg(\mathcal{L}_{\mathsf{Q}}^{\mathsf{MM}}\Bigg(\mathsf{MM}_R, \chi(\mathbf{r}_1)\Bigg)\Bigg)$$

and

$$\mathsf{rtk}_2 \leftarrow \mathcal{S}_{\mathsf{MM}}\Bigg(\mathcal{L}_{\mathsf{Q}}^{\mathsf{MM}}\Bigg(\mathsf{MM}_R, \chi(\mathbf{r}_2)\Bigg)\Bigg),$$

it then sets

$$\mathsf{TT}_N \leftarrow \Bigg(\mathcal{S}_{\mathsf{MM}}\Bigg(\Bigg\{\mathsf{rtk}_{\mathbf{r}_1}, \mathsf{rtk}_{\mathbf{r}_2}\Bigg\}_{(\mathbf{r}_1,\mathbf{r}_2)\in\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\Bigg(\mathsf{MM}_{\mathsf{att}_1}, \Bigg\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2)\Bigg\rangle\Bigg)\Bigg), f(\mathsf{att}_1)\Bigg)$$

- **(Join case-2).** If N has form

$$\Bigg(\texttt{in}, \langle f(\mathsf{att}_1), f(\mathsf{att}_2)\rangle, \Bigg(\mathcal{L}_{\mathsf{Q}}^{\pi}\Bigg(\mathsf{MM}_{\mathsf{att}_1,\mathsf{att}_2}, \chi(\mathbf{r})\Bigg)\Bigg)_{\chi(\mathbf{r})\in\mathbf{R_{in}}[\mathsf{att}_2]}, \Bigg\{\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}(\mathsf{MM}_R, \chi(\mathbf{r}_1)\Bigg\}_{\substack{(\mathbf{r}_1,\mathbf{r}_2)\in\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}\\ \wedge\chi(\mathbf{r}_2)\in\mathbf{R_{in}}[\mathsf{att}_2]}}\Bigg),$$

then it first computes for all $(\mathbf{r}_1, \mathbf{r}_2) \in \mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}$ and $\chi(\mathbf{r}_2) \in \mathbf{R_{in}}[\mathsf{att}_2]$,

$$\mathsf{rtk}_1 \leftarrow \mathcal{S}_{\mathsf{MM}}\Bigg(\mathcal{L}_{\mathsf{Q}}^{\mathsf{MM}}\Bigg(\mathsf{MM}_R, \chi(\mathbf{r}_1)\Bigg)\Bigg)$$

then if $\langle f(\mathsf{att}_1), f(\mathsf{att}_2)\rangle$ has never been queried before, and by leveraging the key-equivocation of $\Sigma_{\mathsf{MM}}^{\pi}$, it generates a key such that[2]

$$K_{f(\mathsf{att}_1),f(\mathsf{att}_2)} \leftarrow \mathcal{S}_{\mathsf{MM}}^{\pi}\Bigg(\{\mathsf{rtk}_{\mathbf{r}}\}_{\mathbf{r}}, \Bigg(\mathcal{L}_{\mathsf{Q}}^{\pi}\Bigg(\mathsf{MM}_{\mathsf{att}_1,\mathsf{att}_2}, \chi(\mathbf{r})\Bigg)\Bigg)_{\chi(\mathbf{r})}\Bigg)$$

otherwise if $\langle f(\mathsf{att}_1), f(\mathsf{att}_2)\rangle$ has been queried before, it uses the previously generated key and sets

$$\mathsf{TT}_N \leftarrow \Bigg(K_{f(\mathsf{att}_1),f(\mathsf{att}_2)}, f(\mathsf{att}_1), f(\mathsf{att}_2)\Bigg)$$

[2]Note that the key will be generated based on all previously simulated row tokens on that particular column; and this is why we omit the indices from the notation in order to capture this aspect.

- **(Join case-3).** If N has form $\Big(\texttt{inter}, f(\mathsf{att}_1), f(\mathsf{att}_2)\Big)$, then it sets

$$\mathsf{TT}_N \leftarrow (f(\mathsf{att}_1), f(\mathsf{att}_2))$$

It remains to show that for all probabilistic polynomial-time adversaries $\mathcal{A}$, the probability that $\mathbf{Real}_{\mathrm{OPX},\mathcal{A}}(k)$ outputs 1 is negligibly-close to the probability that $\mathbf{Ideal}_{\mathrm{OPX},\mathcal{A},\mathcal{S}}(k)$ outputs 1. We do this using the following sequence of games:

Game_0 : is the same as a $\mathbf{Real}_{\mathrm{OPX},\mathcal{A}}(k)$ experiment.

Game_1 : is the same as Game_0, except that EMM_C is replaced with the output of $\mathcal{S}_{\mathsf{MM}}\big(\mathcal{L}_{\mathsf{S}}^{\mathsf{mm}}(\mathsf{MM}_C)\big)$ and every *leaf projection* node of form $\pi_{\mathsf{att}}(\mathbf{T})$ is replaced with the output of

$$\mathcal{S}_{\mathsf{MM}}\bigg(\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\bigg(\mathsf{MM}_C, \chi(\mathsf{att})\bigg)\bigg),$$

Game_2 : is the same as Game_1, except that EMM_V is replaced with the output of $\mathcal{S}_{\mathsf{MM}}\big(\mathcal{L}_{\mathsf{S}}^{\mathsf{mm}}(\mathsf{MM}_V)\big)$ and, every *leaf select* node of form $\sigma_{\mathsf{att}=a}(\mathbf{T})$ is replaced with the output of

$$\mathcal{S}_{\mathsf{MM}}\bigg(\Big(\mathsf{rtk}_{\mathbf{r}}\Big)_{\mathbf{r}\in\mathsf{DB}_{\mathsf{att}=a}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\bigg(\mathsf{MM}_V, \Big\langle a, \chi(\mathsf{att})\Big\rangle\bigg)\bigg).$$

Game_{2+i} for $i \in [\#\mathsf{DB}^{\mathsf{T}}]$: is the same as Game_{1+i}, except that $\mathsf{EMM}_{\mathbf{c}_i}$ is replaced with the output of $\mathcal{S}_{\mathsf{MM}}\big(\mathcal{L}_{\mathsf{S}}^{\mathsf{mm}}(\mathsf{MM}_{\mathbf{c}_i})\big)$ and, every *leaf join* node of form $\mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{T}_2$ is replaced with the output of

$$\Bigg(\mathcal{S}_{\mathsf{MM}}\bigg(\Big\{\mathsf{rtk}_{\mathbf{r}_1}, \mathsf{rtk}_{\mathbf{r}_2}\Big\}_{(\mathbf{r}_1,\mathbf{r}_2)\in\mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\bigg(\mathsf{MM}_{\mathsf{att}_1}, \Big\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2)\Big\rangle\bigg)\bigg), f(\mathsf{att}_1)\Bigg)$$

$\mathsf{Game}_{3+\#\mathsf{DB}^{\mathsf{T}}}$: is the same as $\mathsf{Game}_{2+\#\mathsf{DB}^{\mathsf{T}}}$, except that SET is replaced by a set composed of values generated uniformly at random, and every *internal select* node of the form $\sigma_{\mathsf{att}=a}(\mathbf{R_{in}})$ is replaced with $(K_{g(a\|\mathsf{att})}, f(\mathsf{att}))$, where $K_{g(a\|\mathsf{att})}$ is generated as detailed above.

$\mathsf{Game}_{3+\#\mathsf{DB}^{\mathsf{T}}+i}$ for $i \in [(\#\mathsf{DB}^{\mathsf{T}})^2]$: is the same as $\mathsf{Game}_{2+\#\mathsf{DB}^{\mathsf{T}}+i}$, except that $\mathsf{EMM}_{\mathbf{c}_i,\mathbf{c}'_i}$ is replaced with the output of $\mathcal{S}_{\mathsf{MM}}\big(\mathcal{L}_{\mathsf{S}}^{\mathsf{mm}}(\mathsf{MM}_{\mathbf{c}_i,\mathbf{c}'_i})\big)$, and every *internal join* node of form $\mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R_{in}}$ is replaced with the output of

$$\Bigg(\mathcal{S}_{\mathsf{MM}}^{\pi}\bigg(\{\mathsf{rtk}_{\mathbf{r}}\}_{\mathbf{r}}, \bigg(\mathcal{L}_{\mathsf{Q}}^{\pi}\bigg(\mathsf{MM}_{\mathsf{att}_1,\mathsf{att}_2}, \chi(\mathbf{r})\bigg)\bigg)_{\chi(\mathbf{r})}\bigg), f(\mathsf{att}_1), f(\mathsf{att}_2)\Bigg)$$

$\mathsf{Game}_{4+\#\mathsf{DB}^\intercal+(\#\mathsf{DB}^\intercal)^2}$: is the same as $\mathsf{Game}_{3+\#\mathsf{DB}^\intercal+(\#\mathsf{DB}^\intercal)^2}$ except that EMM_R is replaced with the output of $\mathcal{S}_{\mathsf{MM}}\Big(\mathcal{L}_{\mathsf{S}}^{\mathsf{mm}}(\mathsf{MM}_R)\Big)$ and every row token $\mathsf{rtk}_{\mathbf{r}}$ for a row $\mathbf{r}$ is replaced with the output of[3] of

$$\mathcal{S}_{\mathsf{MM}}\left(\mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\left(\mathsf{MM}_R, \left\langle \mathsf{tbl}(\mathbf{r}), \mathsf{rrk}(\mathbf{r})\right\rangle\right)\right)$$

where $\mathrm{ct}_j \leftarrow \mathsf{Enc}_{K_1}(r_j)$.

$\mathsf{Game}_{5+\#\mathsf{DB}^\intercal+(\#\mathsf{DB}^\intercal)^2}$: is the same as $\mathsf{Game}_{4+\#\mathsf{DB}^\intercal+(\#\mathsf{DB}^\intercal)^2}$, except that every SKE encryption ct of a message m is replaced with $\mathrm{ct} \leftarrow \mathsf{Enc}_{K_1}(\mathbf{0}^{|m|})$.

Note that $\mathsf{Game}_{5+\#\mathsf{DB}^\intercal+(\#\mathsf{DB}^\intercal)^2}$ is identical to $\mathbf{Ideal}_{\mathrm{OPX},\mathcal{A},\mathcal{S}}(k)$.

■

[3]Note that we are making the assumption that all attributes have the same domain, otherwise, there would be a number of games smaller than $(\#\mathsf{DB}^\intercal)^2$.

Bibliography

[1] Intellij idea. https://www.jetbrains.com/idea/.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *ACM SIGMOD International Conference on Management of Data*, pages 563–574, 2004.

[3] G. Amanatidis, A. Boldyreva, and A. O'Neill. Provably-secure schemes for basic query support in outsourced databases. In *Working conference on Data and applications security*, pages 14–30, 2007.

[4] I. Amazon.com. Amazon elastic compute cloud, 2019.

[5] G. Amjad, S. Kamara, and T. Moataz. Breach-resistant structured encryption. In *Proceedings on Privacy Enhancing Technologies (Po/PETS '19)*, 2019.

[6] Anonymous. `KafeDB` source code. https://anonymous.4open.science/r/66286761-2631-4311-b65d-983570892591/, 2020.

[7] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *CIDR*, 2013.

[8] D. W. Archer, D. Bogdanov, L. Kamm, Y. Lindell, K. Nielsen, J. I. Pagter, N. P. Smart, and R. N. Wright. From keys to databases – real-world applications of secure multi-party computation. Cryptology ePrint Archive, Report 2018/450, 2018. https://eprint.iacr.org/2018/450.

[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394. ACM, 2015.

[10] S. Bajaj and R. Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Trans. Knowl. Data Eng.*, 26(3):752–765, 2014.

[11] J. Bater, G. Elliott, C. Eggen, S. Goel, A. Kho, and J. Rogers. Smcql: secure querying for federated databases. *Proceedings of the VLDB Endowment*, 10(6):673–684, 2017.

[12] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In A. Menezes, editor, *Advances in Cryptology – CRYPTO '07*, Lecture Notes in Computer Science, pages 535–552. Springer, 2007.

[13] A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill. Order-preserving symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2009*, pages 224–241, 2009.

[14] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: improved security analysis and alternative solutions. In *Advances in Cryptology - CRYPTO '11*, pages 578–595, 2011.

[15] A. Boldyreva, S. Fehr, and A. O'Neill. On notions of security for deterministic encryption, and efficient constructions without random oracles. In *Advances in Cryptology - CRYPTO '08*, pages 335–359. 2008.

[16] R. Bost. Sophos - forward secure searchable encryption. In *ACM Conference on Computer and Communications Security (CCS '16)*, 20016.

[17] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[18] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO '13*. Springer, 2013.

[19] D. Cash and S. Tessaro. The locality of searchable symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2014*, 2014.

[20] Y. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security (ACNS '05)*, volume 3531 of *Lecture Notes in Computer Science*, pages 442–455. Springer, 2005.

[21] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT '10*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010.

[22] M. Chase and S. Kamara. Structured encryption and controlled disclosure. Technical Report 2011/010.pdf, IACR Cryptology ePrint Archive, 2010.

[23] T. Chiba and T. 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)*, pages 112–121. IEEE, 2016.

[24] M. Corp. Always Encrypted. https://msdn.microsoft.com/en-us/library/mt163865(v=sql.130).aspx.

[25] T. P. P. Council. Tpc benchmark™h standard specification revision 2.18.0. 2018.

[26] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security (CCS '06)*, pages 79–88. ACM, 2006.

[27] F. B. Durak, T. M. DuBuisson, and D. Cash. What else is revealed by order-revealing encryption? In *ACM Conference on Computer and Communications Security (CCS '16)*, 2016.

[28] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich queries on encrypted data: Beyond exact matches. In *European Symposium on Research in Computer Security (ESORICS '15). Lecture Notes in Computer Science*, volume 9327, pages 123–145, 2015.

[29] B. A. Fisch, B. Vo, F. Krell, A. Kumarasubramanian, V. Kolesnikov, T. Malkin, and S. M. Bellovin. Malicious-client security in blind seer: a scalable private dbms. In *IEEE Symposium on Security and Privacy*, pages 395–410. IEEE, 2015.

[30] S. Garg, P. Mohassel, and C. Papamanthou. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *Advances in Cryptology - CRYPTO 2016*, pages 563–592, 2016.

[31] E.-J. Goh. Secure indexes. Technical Report 2003/216, IACR ePrint Cryptography Archive, 2003. See http://eprint.iacr.org/2003/216.

[32] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[33] T. P. G. D. Group. Postgresql 9.6.2. https://www.postgresql.org/ftp/source/v9.6.2/, 2017.

[34] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *IEEE Symposium on Security and Privacy (S&P '17)*, 2017.

[35] H. Hacigümücs, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 216–227, 2002.

[36] Y. Ishai, E. Kushilevitz, S. Lu, and R. Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In K. Sako, editor, *Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings*, volume 9610 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 2016.

[37] S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In *ACM Conference on Computer and Communications Security (CCS '13)*, pages 875–888, 2013.

[38] N. M. Johnson, J. P. Near, and D. X. Song. Practical differential privacy for SQL queries using elastic sensitivity. *CoRR*, abs/1706.09479, 2017.

[39] S. Kamara and T. Moataz. SQL on Structurally-Encrypted Data. In *Asiacrypt*, 2018.

[40] S. Kamara and T. Moataz. Computationally volume-hiding structured encryption. In *Advances in Cryptology - Eurocrypt' 19*, 2019.

[41] S. Kamara, T. Moataz, and O. Ohrimenko. Structured encryption and leakae suppression. In *Advances in Cryptology - CRYPTO '18*, 2018.

[42] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security (CCS '12)*. ACM Press, 2012.

[43] I. Mironov, G. Segev, and I. Shahaf. Strengthening the security of encrypted databases: Non-transitive joins. In *IACR Cryptol. ePrint Arch.*, 2017.

[44] T. L. of the Bouncy Castle. Bouncy castle java release 1.64. `http://bouncycastle.org/latest_releases.html`, 2019.

[45] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S.-G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 359–374. IEEE, 2014.

[46] R. Poddar, T. Boelter, and R. A. Popa. Arx: A Strongly Encrypted Database System. Technical Report 2016/591.

[47] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, 2011.

[48] R. A. Popa and N. Zeldovich. Cryptographic treatment of cryptdb's adjustable join. 2012.

[49] SAP Software Solutions. SEEED. `https://www.sics.se/sites/default/files/pub/andreasschaad.pdf`.

[50] D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE Symposium on Research in Security and Privacy*, pages 44–55. IEEE Computer Society, 2000.

[51] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[52] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *Proc. VLDB Endow.*, 6:289–300, 2013.

[53] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC '82*, 1982.

[54] D. Vinayagamurthy, A. Gribov, and S. Gorbunov. Stealthdb: a scalable encrypted database with full SQL query support. *PoPETs*, 2019(3):370–388, 2019.

[55] N. Volgushev, M. Schwarzkopf, B. Getchell, M. Varia, A. Lapets, and A. Bestavros. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 3. ACM, 2019.

[56] X. S. Wang, K. Nayak, C. Liu, T. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226. ACM, 2014.