Abstract of "Building a Structurally-Encrypted Relational Database System" by Zheguang Zhao, Ph.D., Brown University, May 2021.

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`, `Monomi`, `ESPADA`, `Blind Seer` and `Stealth`. `CryptDB` and `Monomi` are based on property-preserving encryption (PPE) and have been shown to leak a considerable amount of information even in the snapshot model, which is the weakest adversarial model in this setting. 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 `Monomi`, they require a custom database system.

We design and build an encrypted database management system called `KafeDB` that addresses all these limitations. At the core of `KafeDB` are two new cryptographic schemes based on structured encryption (STE) called OPX and PKFK. In these schemes we propose STE-based techniques for optimal join, query optimization, leakage reduction and locality improvement over encrypted relational data. Overall `KafeDB` achieves a leakage profile comparable to the `ESPADA`, `Blind Seer` and `Stealth` systems. However `KafeDB` handles a non-trivial subset of SQL which includes queries with joins, selections and projections. In addition, `KafeDB` is *legacy-friendly*, meaning that it can be deployed on top of *any* relational database system. The TPC-H benchmark showed that `KafeDB` had 4.2× query overhead and 3.6× storage overhead over plaintext `PostgreSQL`.

Building a Structurally-Encrypted Relational Database System

by

Zheguang Zhao

B.S., University of Wisconsin at Madison, 2012

M.Sc., Brown University, 2016

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University.

Providence, Rhode Island

May, 2021

This dissertation by Zheguang Zhao is accepted in its present form
by the Department of Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____

Prof. Seny Kamara, Brown University, Advisor

Date _____ _____

Prof. Stanley Zdonik, Brown University, Advisor

Recommended to the Graduate Council

Date _____ _____

Prof. George Kollios, Boston University, Reader

Date _____ _____

Prof. Moti Yung, Columbia University, Reader

Approved by the Graduate Council

Date _____ _____

Prof. Andrew Campbell, Dean of the Graduate School

To my family.

Preface

This thesis results from my collaboration with several individuals. My academic advisors, Seny Kamara and Stan Zdonik, and my mentor, Tarik Moataz, helped me develop the subject. The other committee members, George Kollios and Moti Yung, and the external faculty member, Ugur Cetintemel, provided me with discussions and ideas. Several other individuals also helped shape my broader academic interests during the course of my study: Carsten Binnig, Lorenzo De Stefani, Subramanya R. Dulloor, Paul G. Dupius, Kris Ganjam, Maurice Herlihy, Tim Kraska, Andrey Y. Lokhov, Sherief Reda, Amitabha Roy, Richard E. Schwartz, Eli Upfal and Emanuel Zgraggen. The Brown University Graduate School and the National Science Foundation funded my research. Personally, I would like to thank my wife, Shuyuan, my other family members, and all of my friends for encouraging and supporting me during my study.

Contents

Chapter 1

Introduction

Data is being produced, collected and analyzed at unprecedented speed, volume and variety [69]. Individuals and enterprises have increasingly relied on outsourcing the storage and computation of their data to remote servers maintained by a third-party service provider. This usage pattern has been identified as "cloud computing" [75].

Cloud computing offers several advantages over the traditional approach of active management computation resources by the user. First, cloud-based solutions alleviate the burden for maintaining infrastructure from the user, and provide increasingly more reliable uptime and scalable computing and storage resources on demand. The user in turn can just focus on the data and the task. For developers, cloud-native applications have drastically reduced the gap between development and deployment, shortened engineering cycles, and enabled faster feature delivery to the end users. The cloud service providers invest heavily into research and development of better infrastructure, resulting in lower price and better quality of service.

In the past two decades, cloud computing is rapidly expanding to cover many aspects of daily life and business activities, ranging from emails and messaging, social media, file backups, data sharing, collaborative document editing, to voice assitance, machine translation and navigation. Not only individuals but organizations in goverments, e-commerce, healthcare, education and financial services increasingly rely on cloud services to provide digital access. The total market for cloud service revenue is forcasted to reach a total of 278.3 billion U.S. Dollars [39]. 58% of manu-

factoring industry rely on the cloud service to analyze their activities; the ratio remains as high as 50% for financial services and 40% for the education sector [89].

Meanwhile, 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 2014 private data for over 7 million users on Dropbox was stolen for extortion [86]. In 2016, the Democratic National Committee was hacked, and the documents leaked from the hack affected the 2016 U.S. presidential election [88]. New regulations such as the General Data Protection Regulation [95] and the California Consumer Privacy Act [70] have been put forth to prevent any discosure or access, either accidental, unlawful, unethical or unauthorized, to sensitive data. Under these regulations, the organizations that collect user data are required to implement data protection such as cryptographic access control.

On the other hand, surveillance programs have grown more pervasive and global, as revealed by Edward Snowden's leakage of classified information from National Security Agency in 2013 [53]. The cloud service providers should not be trusted with encrypting data alone, because they can be forced to turn over encryption keys under the pressure of the law enforcement.

End-to-end encryption. One way to address the privacy and security challenges of remote computation is to deploy 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 are a $40 billion industry which includes 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 [56] who described a quantization-based approach which leaks the range within which an item falls. In [84], 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 [5, 20, 23], 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 [12] and commercial systems like SEEED [87] and Microsoft's SQL Server Always Encrypted [40]. While the security of PPE *primitives* had been formally studied by the cryptography community [7, 20, 25, 23, 24], 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 [63]). 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 [55, 43].

EDBs on Trusted Hardware. An approach to designing relational EDBs is to use trusted hardware such as custom FPGAs or secure coprocessors like Intel SGX. Several systems, most notably Cipherbase [12], TrustedDB [16] and StealthDB [96] take this direction. However, the

security offered by systems relying on trusted hardware is weaker than end-to-end encryption. The effciency of the systems is also limited by the hardware, such as the memory size of SGX. Therefore Cipherbase and StealthDB are efficient mostly on a transactional workload, which typically is less memory intensive than an analytic workload. Lastly these systems also require intrusive software changes that may present challenges to adoption.

General-purpose approaches. For end-to-end encryption, 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 [49] (FHE) and secure multi-party computation [99, 100, 72] (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 large constants. Oblivious RAM (ORAM) could also be used to handle all of SQL with little leakage (i.e., mostly volume leakage) but at the cost of a poly-logarithmic multiplicative overhead in the size of the database.

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 approaches. More practical solutions can be achieved using structured encryption (STE) [36] which is a generalization of searchable symmetric encryption (SSE) [91, 42]. 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) [42]. This is, roughly speaking, the approach taken by systems such as ESPADA [31, 30, 59, 45], Blind Seer [81, 47] and Stealth [57].[1] We refer to this as *column indexing* and this leads to systems that can handle SQL queries of the form

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

[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.

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., [60] 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 [63]; 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 significantly less than known PPE-based solutions such as CryptDB.

In this work we take the STE-based approach to design encrypted relational database systems. During the course of our research, we needed to address the following challenges:

Query optimization. Relational query optimization has been extensively studied in the relational database literature [1]. Typical query optimization relies on relational-algebraic rules in combination with heuristics and cost estimation from the data distribution [52, 1, 15, 74]. The existing STE-based approach such as SPX [63] is not able to support query optimization because it only executes the operators in a fixed order. Even if we modify SPX to support the input query with reordered operators, it still could not reduce the overall query complexity. Without the support of query optimization, an STE-based scheme such as SPX often become suboptimal up to a potentially large constant in practice. In order to make a more practical STE-based relational database, we studied STE-based approach that not only admits query operators in various orders but also produces the effect of reduced query complexity (Ch. 3).

Complexity improvement. The query and storage complexity of relational data has been studied and improved continuously in the database literature for decades [1]. Prior STE-based constructions for encrypted relational data still cannot match the plaintext asymptotic complexity. For example, SPX [63] incurs worst-case quadratic overhead for both join queries and storage. This high overhead would hinder the scalability of the encrypted database system. To reduce the complexity gap, we designed the first STE-based join algorithm that matches the plaintext join (Ch. 4).

Leakage reduction. Existing STE-based schemes also come with nontrivial query leakage for complex relational queries that consists of multiple operators. For example, SPX leaks the patterns about the full join between two attributes, even if only a few rows between the two attributes are filtered for the query result. We investigated in how to reduce such leakage for queries of multiple operators (Ch. 4) while keeping the same asymptotic storage complexity.

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. In this work we developed a new technique called *emulation* (Ch. 5) to transform STE-based schemes such that they can be implemented on standard relational systems without compromising security and efficiency.

Locality improvement. Relational data contains much information about data locality, such as values that are colocated in the same rows, columns or tables based on logical relationship. This locality informs database systems how to improve query efficiency, such as (1) prefetching colocated values from slower storage to faster cache, or (2) storing colocated values in nearby storage blocks. However, this locality is lost in STE-based schemes such as SPX [63] mainly due to encryption. As a result the STE-based encrypted databases may have limited scalability. We study how to augment such an STE-based scheme in order to recover data locality and the security trade-off (Ch. 5).

Open-source Implementation. Finally, we combine the solutions to these challenges into two STE-based constructions called OPX and PKFK. Based on these schemes, we build the first STE-based relational database system called `KafeDB`. `KafeDB` 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. As a prototype, we built `KafeDB` on top of an unmodified instance of `PostgreSQL`. The TPC-H Benchmark [41] showed that `KafeDB` had 4.2× query overhead and 3.63× storage overhead.

1.1 Related Work

Encrypted search. Encrypted search was first considered explicitly by Song, Wagner and Perrig in [91] which introduced the notion of searchable symmetric encryption (SSE). Goh [50] provided the first security definition for SSE and a solution based on Bloom filters with linear search complexity. Chang and Mitzenmacher [35] proposed an alternative security definition and construction, also with linear search complexity. Curtmola et al. [42] introduced and formulated the notion of adaptive semantic security for SSE together with the first sub-linear and optimal-time constructions. Chase and Kamara [36] introduced the notion of structured encryption (STE) which generalizes SSE to arbitrary data structures. Kamara, Papamanthou and Roeder [67] gave the first optimal-time dynamic SSE scheme.

Cash et al. [31] proposed the first optimal-time scheme that handles conjunctive keyword search and Faber et al. show how to extend it to rich queries (e.g., range, substring and wildcard queries) [46] . Naveed et al. [79] propose an optimal-time dynamic SSE scheme based on blind storage. Cash et al. [30] show how to construct optimal-time SSE schemes with low I/O complexity and Cash and Tessaro [32] gave lower bounds on the locality of adaptively-secure SSE schemes. Asharov et al. build SSE schemes with optimal locality, optimal space overhead and nearly-optimal read efficiency [14]. Miers and Mohassel [76] presented an optimal time I/O efficient SSE construction. Garg et al. [48] presented a new SSE construction with reduced leakage leveraging oblivious RAM and garbled RAM techniques. Bost [27] proposed an efficient forward-secure SSE construction

based on trapdoor permutations. Kamara and Moataz [61] proposed non-interactive SSE schemes that handle boolean queries with arbitrary disjunctions and conjunctions with sub-linear search complexity. Bost and Fouque [28] described padding techniques that are more efficient than naive padding (adding dummy values to ensure all query responses are of the same volume) to protect against volume-based attacks. Kamara and Moataz [62] proposed the first volume-hiding encrypted multi-maps that do not rely on naive padding. Kamara et al. [65] described a general framework to design STE schemes that do not leak the query/search pattern.

SSE has also been considered in the multi-user setting [42, 59]. Pappas et al. [82] proposed a multi-user SSE construction based on garbled circuits and Bloom filters that can support Boolean formulas, ranges and stemming.

Other approaches for encrypted search include oblivious RAMs (ORAM) [51], secure multi-party computation [21], functional encryption [26] and fully-homomorphic encryption [49] as well as solutions based on deterministic encryption [20] and order-preserving encryption (OPE) [5, 23].

Attacks on SSE. While we do not consider the problem of designing an SSE scheme in this work, we do use SSE building blocks. Several works have proposed attacks that try to exploit the leakage of SSE. This includes the query-recovery attacks of Islam et al. [58], of Cash et al. [29], of Zhang et al. [102], and of Blackstone et al. [22]. Recently, Abdelraheem et al. [78], presented attacks on encrypted relational databases. We briefly mention here that although the attacks in [78] are ostensibly on relational EDBs, they are not related to or applicable to our constructions.

Locality in SSE. Locality has also been studied in SSE [36, 32]. To have optimal locality is to achieve minimum random access in each keyword search, i.e. for $DB(w)$ values associated with a keyword or label w, no more than constant amount of random access per search ($\mathcal{O}(1)$). Chase and Kamara [36] presented a construction that achieves optimal locality but at the cost of suboptimal storage size: $\mathcal{O}(K \cdot N)$ with K unique keywords and N keyword-value pairs. Cash and Tessaro [32] presented a lower bound for a SSE scheme that has constant locality and constant read efficiency (in $|DB(w)|$) has to incur storage size superlinear in N.

Distributed EDBs. Agarwal and Kamara studied the STE-based encrypted distributed hash tables [3] and key-value stores [4]. Adkins et al. [2] studied the design of end-to-end encrypted blockchain databases.

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 [19] and Conclave [97], 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 [83] and Jana by Galois [13]. 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.

1.2 Preliminaries

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.

Abstract data types. An *abstract data type* specifies the functionality of a data structure. It is a collection of data objects together with a set of operations defined on those objects. Examples include sets, dictionaries (also known as key-value stores or associative arrays) and graphs. The operations associated with an abstract data type fall into one of two categories: query operations, which return information about the objects; and update operations, which modify the objects. If the abstract data type supports only query operations it is *static*, otherwise it is *dynamic*. For simplicity we define data types as having a single operation and note that the definitions can be extended to capture multiple operations in the natural way. We model a dynamic data type $\mathbf{T}$ as a collection of four spaces $\mathbf{D} = \{\mathbf{D}_k\}_{k\in\mathbb{N}}$, $\mathbb{Q} = \{\mathbb{Q}_k\}_{k\in\mathbb{N}}$, $\mathbb{R} = \{\mathbb{R}_k\}_{k\in\mathbb{N}}$ and $\mathbb{U} = \{\mathbb{U}_k\}_{k\in\mathbb{N}}$ and two maps $\mathsf{qu} : \mathbf{D} \times \mathbb{Q} \to \mathbb{R}$ and $\mathsf{up} : \mathbf{D} \times \mathbb{U} \to \mathbf{D}$, where $\mathbf{D}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{U}$ are, respectively, $\mathbf{T}$'s object, query, response and update spaces. A data type $\mathbf{T}$ is often described by its maps $(\mathsf{qu}, \mathsf{up})$ from which the object, query, response and update spaces can be deduced. The spaces are ensembles of finite sets of finite strings indexed by the security parameter. We assume that $\mathbb{R}$ includes a special element $\perp$ and that $\mathbf{D}$ includes an empty object d_0 such that for all $q \in \mathbb{Q}$, $\mathsf{qu}(d_0, q) = \perp$.

Data structures. A type-$\mathbf{T}$ data *structure* is a representation of data objects in $\mathbf{D}$ in some computational model (as mentioned, here it is the word RAM). Typically, the representation is optimized to support qu as efficiently as possible; that is, such that there exists an efficient algorithm Query that computes the function qu. For data types that support multiple queries, the representation is often optimized to efficiently support as many queries as possible. As a concrete example, the dictionary type can be represented using various data structures depending on which queries one wants to support efficiently. Hash tables support Get and Put in expected $O(1)$ time whereas balanced binary search trees support both operations in worst-case $\log(n)$ time.

Definition 1.2.1 (Structuring scheme)**.** *Let* $\mathbf{T} = (\mathsf{qu} : \mathbf{D} \times \mathbb{Q} \to \mathbb{R}, \mathsf{up} : \mathbf{D} \times \mathbb{U} \to \mathbf{D})$ *be a dynamic type. A type-*$\mathbf{T}$ *structuring scheme* $\mathsf{SS} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Update})$ *is composed of three polynomial-time algorithms that work as follows:*

- $\mathsf{DS} \leftarrow \mathsf{Setup}(d)$*: is a possibly probabilistic algorithm that takes as input a data object* $d \in \mathbf{D}$ *and outputs a data structure* DS*. Note that* d *can be represented in any arbitrary manner as long as its bit length is polynomial in* k*. Unlike* DS*, its representation does not need to be optimized for any particular query.*

- $r \leftarrow \mathsf{Query}(\mathsf{DS}, q)$*: is an algorithm that takes as input a data structure* DS *and a query* $q \in \mathbb{Q}$ *and outputs a response* $r \in \mathbb{R}$*.*

- $\mathsf{DS} \leftarrow \mathsf{Update}(\mathsf{DS}, u)$*: is a possibly probabilistic algorithm that takes as input a data structure* DS *and an update* $u \in \mathbb{U}$ *and outputs a new data structure* DS*.*

Here, we allow Setup and Update to be probabilistic but not Query. This captures most data structures but the definition can be extended to include structuring schemes with probabilistic query algorithms. We say that a data structure DS *instantiates* a data object $d \in \mathbf{D}$ if for all $q \in \mathbb{Q}$, $\mathsf{Query}(\mathsf{DS}, q) = \mathsf{qu}(d, q)$. We denote this by $\mathsf{DS} \equiv d$. We denote the set of queries supported by a structure DS as $\mathbb{Q}_{\mathsf{DS}}$; that is,

$$\mathbb{Q}_{\mathsf{DS}} \stackrel{def}{=} \Big\{ q \in \mathbb{Q} : \mathsf{Query}(\mathsf{DS}, q) \neq \perp \Big\}.$$

Similarly, the set of responses supported by a structure DS is denoted $\mathbb{R}_{\mathsf{DS}}$.

Definition 1.2.2 (Correctness)**.** *Let* $\mathbf{T} = (\mathsf{qu} : \mathbf{D} \times \mathbb{Q} \to \mathbb{R}, \mathsf{up} : \mathbf{D} \times \mathbb{U} \to \mathbf{D})$ *be a dynamic type. A type-*$\mathbf{T}$ *structuring scheme* $\mathsf{SS} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Update})$ *is perfectly correct if it satisfies the following properties:*

1. *(static correctness) for all* $d \in \mathbf{D}$*,*

$$\Pr\left[\, \mathsf{DS} \equiv d : \mathsf{DS} \leftarrow \mathsf{Setup}(d) \,\right] = 1,$$

   where the probability is over the coins of Setup*.*

2. *(dynamic correctness) for all* $d \in \mathbf{D}$ *and* $u \in \mathbb{U}$*, for all* $\mathsf{DS} \equiv d$*,*

$$\Pr\left[\, \mathsf{Update}(\mathsf{DS}, u) \equiv \mathsf{up}(d, u) \,\right] = 1,$$

where the probability is over the coins of Update.

Note that the second condition guarantees the correctness of an updated structure whether the original structure was generated by a setup operation or a previous update operation. Weaker notions of correctness (e.g., for data structures like Bloom filters) can be derived from Definition 1.2.2.

Basic data structures. We use structures for several basic data types including dictionaries and multi-maps which we recall here. A dictionary structure DX of capacity n holds a collection of n label/value pairs $\{(\mathtt{label}_i, v_i)\}_{i \leq n}$ and supports get and put operations. We write $v_i := \mathsf{DX}[\mathtt{label}_i]$ to denote getting the value associated with label $\mathtt{label}_i$ and $\mathsf{DX}[\mathtt{label}_i] := v_i$ to denote the operation of associating the value v_i in DX with label $\mathtt{label}_i$. A multi-map structure MM with capacity n is a collection of n label/tuple pairs $\{(\mathtt{label}_i, \mathbf{v}_i)_i\}_{i \leq n}$ that supports get and put operations. Similarly to dictionaries, 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).

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) [31, 30].

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 $\chi(\mathbf{r}) \stackrel{def}{=} (\mathsf{tbl}(\mathbf{r}), \mathsf{rrk}(\mathbf{r}))$ and $\chi(\mathbf{c}) \stackrel{def}{=} (\mathsf{tbl}(\mathbf{c}), \mathsf{crk}(\mathbf{c}))$, respectively, as their *coordinates* in DB. We also use $\chi_{\mathbf{r}}$ to denote the coordinate for row $\mathbf{r}$. 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 $\{\mathbf{r} \in \mathsf{DB} : \mathbf{r}[\mathsf{att}] = a\}$.

SQL and relational queries. In this work, we focus on the class of *conjunctive SQL* queries or the *SPC algebra* [1], which have the form,

`SELECT` attributes `FROM` tables
`WHERE` $\mathsf{att}_1 = X_1$ `AND` $\mathsf{att}_2 = X_2$,

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 referred to as a *select predicate* or *filter predicate* or *constant predicate*, whereas if X_i is an attribute, then the predicate $\mathsf{att}_i = X_i$ is called a *join predicate*. We use standard relational algebra notation and denote the *select operator* or *filter operator* by σ, the *project operator* by π, the *rename operator* by ρ, the *θ-join operator* by $\underset{\theta}{\bowtie}$ with the join predicate θ, the *cross-join operator* by $\times$, and the *(left) semi-join operator* by $\ltimes$. We follow the semantics of all the above operators as defined in [1]. We also use *fixed-point operators*, which is an extension over relational algebra to express recursion [6]. For example, $\bigcup_i^\infty \mathrm{Q}_i$ specifies a union of a sequence of queries $\{\mathrm{Q}_i\}_i$ which stops at maximum i for which Q_i results in an empty set.

1.3 Definitions

In this Section, we define the syntax and security of STE schemes.

1.3.1 Basic Cryptographic Primitives

We make use of standard private-key encryption schemes, pseudo-random functions and hash functions.

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.[2]

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.

1.3.2 Structured Encryption

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 [36] considered schemes that encrypt both a structure and a set of associated data items (e.g., documents, emails, user profiles etc.). In [37], 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*

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

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, PKFK, 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 1.3.1 (Response-revealing structured encryption [36])**.** *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*.*
- $\{\perp, r\} \leftarrow \mathsf{Query}(\mathsf{EDS}, \mathsf{tk})$*: 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* $\mathsf{DS} : Q \to \mathbf{R}$*, 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 1.3.2 (Response-hiding structured encryption [36])**.** *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 1.3.1 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* $\mathsf{DS} : Q \to \mathbf{R}$*, 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* $\mathsf{Token}(K, q_i)$*,* $\mathsf{Dec}_K\left(\mathsf{Query}\left(\mathsf{EDS}, \mathsf{tk}_i\right)\right)$ *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 [42] and later generalized to structured encryption in [36].

The security of STE is formalized using "leakage-parameterized" definitions following [42, 36]. 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 [42, 36] and for snapshot adversaries in [9].[3]

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 [42, 36, 9] for more details.

[3]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.

Definition 1.3.3 (Adaptive semantic security [42, 36])**.** *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.*

$\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathcal{S}}(k)$*: 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* $(\mathsf{DS}(q_i), \mathcal{L}_{\mathsf{Q}}(\mathsf{DS}, q_i))$ *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 there exists a* PPT *simulator* $\mathcal{S}$ *such that for all* PPT *adversaries* $\mathcal{A}$*, 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 1.3.3 by giving the simulator $(\perp, \mathcal{L}_{\mathsf{Q}}(\mathsf{DS}, q_i))$ instead of $(\mathsf{DS}(q_i), \mathcal{L}_{\mathsf{Q}}(\mathsf{DS}, q_i))$.

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 [65] 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 $\mathsf{rid}_{k,t}(\mathsf{DS}, q_1, \ldots, q_t) = (\mathsf{DS}[q_1], \ldots, \mathsf{DS}[q_t])$. 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 $\mathsf{rlen}_{k,t}(\mathsf{DS}, q_1, \ldots, q_t) = (|\mathsf{DS}[q_1]|, \ldots, |\mathsf{DS}[q_t]|)$;

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 [42, 67, 30, 30, 32, 92, 27, 65]. Encrypted dictionaries can be obtained from any encrypted multi-map since the former is just an encrypted multi-map with single-item tuples.

Chapter 2

KafeDB: System Architecture

2.1 Introduction

In this work, we tackle the key challenges that impede the development and deployment of encrypted databases. Our contributions can be summarized as follows:

- *(design principles)* we identify and discuss five key principles for the design of practical and secure relational EDBs. These include a reasonable leakage profile, efficiency, legacy friendliness, optimization friendliness and expressiveness. Achieving any strict subset of these four requirements is insufficient.

- *(construction)* we describe the first encrypted database scheme that follows all the design principles outlined above. To achieve this we make two important technical contributions: (1) we show, for the first time, how to design *optimization-friendly* STE schemes; and (2) we introduce a new technique called *emulation* that makes STE-based solutions legacy-friendly. This new scheme is called OPX and is an extension of the SPX construction of Kamara and Moataz [63].

- *(architecture)* we propose an architecture for encrypted database management systems that integrates the needed cryptographic components into a traditional DBMS architecture. This is, in part, done by introducing a *crypto engine* that is responsible for providing end-to-end

encryption and an *emulation engine* that is responsible for making the encrypted databases and queries "comprehensible" to a standard and unmodified DBMS.

- *(prototype)* we describe the implementation and evaluation of a new system called `KafeDB` based on our architecture, our OPX construction and our emulators. `KafeDB` runs on top of an *unmodified* `PostgreSQL` server. Our initial prototype demonstrates the feasibility of our architecture and approach. We evaluate its performance empirically using the `TPC-H` benchmark and report promising initial results: about an order of magnitude query and storage overhead over standard `PostgreSQL`, but offering considerably stronger security guarantees than `CryptDB` and much more expressiveness than `ESPADA`, `Blind Seer` and `Stealth`. To improve upon the initial results, we will need more sophisticated techniques in both cryptography and database systems.

2.2 Design Principles

Designing a relational EDB is, arguably, the most challenging problem in encrypted search. Existing RDBMs are the result of over 40 years of research and development so competing with the performance of commercial DBMSs over encrypted data is a tall order. To achieve this level of performance, it stands to reason that EDBs need to inherit as many of these advances as possible. With this in mind, we outline five principles that are necessary for the design of practical EDBs.

Adversarial models & leakage. There are two main adversarial models considered in encrypted search: (1) *persistent adversaries* which have access to the encrypted database and can view all the query operations that are executed on it; and *snapshot* adversaries which only have access to the encrypted database. Persistent adversaries model attackers that corrupt the server and stay long enough to view some number of queries. Snapshot adversaries model attackers that corrupt the server and exfiltrate a snapshot of its memory and disk. All cryptographic solutions that support search on encrypted data in sub-linear time leak information against persistent adversaries. This is true of PPE-based, STE-based and ORAM-based solutions. However, it is known that both STE and ORAM can lead to solutions with no leakage against snapshot adversaries [9].

The security of an encrypted search solution is characterized by its *leakage profile* which is a formal description of the information an adversary learns from observing and interacting with the scheme. More precisely, in the the persistent model, a leakage profile consists of: (1) *setup leakage* which describes the information the adversary learns by just observing the encrypted database; and (2) *query leakage* which describes the information the adversary learns by observing the execution of queries. For dynamic schemes, which support the addition and deletion of data, the leakage profile also includes *add leakage* and *delete leakage*. This approach to characterizing leakage was introduced in [42, 36] and we refer to [65] for additional details.

Principal #1: minimal leakage. An important design goal for any encrypted database is to minimize the information a persistent adversary is able to recover. At a minimum, this means that there should be no known practical attack against the scheme. Furthermore, the scheme's leakage profile should have the following characteristics:

- *(minimal setup leakage)* the setup leakage of the scheme should include at most the "shape" of the database; i.e., the number of columns and rows of each table.
- *(output-dependent query leakage)* when a query is executed, the adversary should, at most, learn information related to result of the query and not to the entire database or column. Furthermore, the information that is leaked should, at most, be statistical information about the query or result like whether a query has been queried in the past, or the number of rows that contain a similar value.

To be clear, leakage profiles with these characteristics are not provably immune to attacks. But, given our current understanding and the state-of-the-art results in cryptanalysis [22], these leakage profiles seem difficult to attack in practice. For more discussion about leakage attacks we refer the reader to [22] and the discussions and references therein. [1]

Principle #2: low asymptotic overhead. The system should be competitive with a standard plaintext DBMS with respect to query execution and storage. High performance imposes efficiency

[1] Since our current design and system does not yet handle range queries, we do not consider cryptanalytic work focused on encrypted range schemes.

requirements on the system's database encryption scheme. Specifically, it should achieve the same asymptotic complexity as a plaintext database and preferably with small constants. As an example, schemes that add a linear or even polylogarithmic multiplicative overhead over a plaintext query are unacceptable in practice. The same applies for the scheme's round and storage complexities.

Principle #3: optimization friendliness. In addition to low asymptotic overhead, the underlying database encryption scheme should be *optimization-friendly* in the sense that it should support the execution of optimized query plans and, in particular, of plans that are optimized by commercial query optimizers.

Principle #4: rich query expressiveness. The system should support a non-trivial subset of SQL and, at a minimum, the class of *conjunctive* SQL queries (or the SPC algebra) which have the form:

`SELECT` attributes `FROM` tables

`WHERE` $\mathsf{att}_1 = a$ `AND` ... `AND` $\mathsf{att}_2 = \mathsf{att}_3$ `AND` ...

This requires being able to handle select, project and join operators.

Principle #5: legacy friendliness. While building an entirely new encrypted DBMS from the ground up is an interesting technical question, designing a scheme that can work on top of an existing, unmodified DBMSs is more appealing from a practical standpoint. If the resulting system is competitive with plaintext systems, achieves the required security and provides rich query expressiveness, there is almost no reason to build a new DBMS from scratch and lose over 40 years of advances in database research and engineering. Ideally, the design should be database-agnostic in the sense that it should not depend on a particular DBMS.

2.3 System Architecture

`KafeDB` has a three-tier architecture composed of the application, the client and the server, as shown in Figure 2.1. Both the application and the client are assumed to operate in a trusted environment,

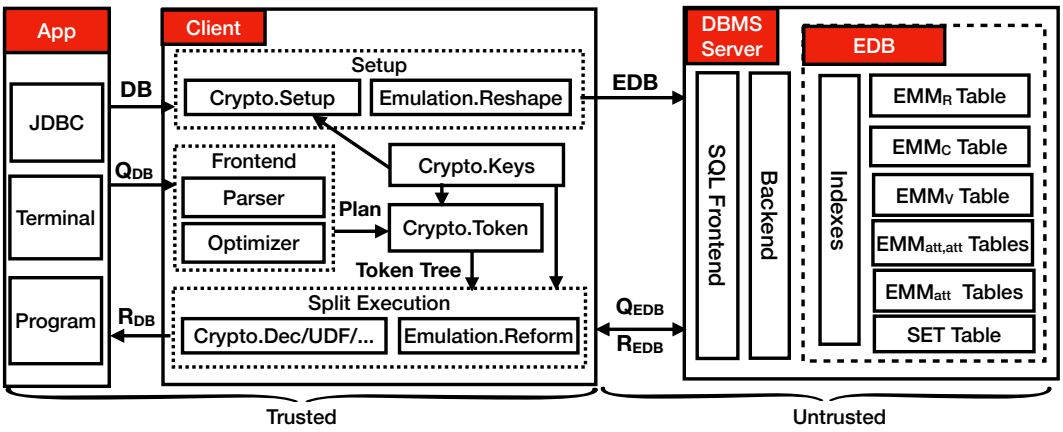


Figure 2.1: The KafeDB system architecture.

whereas the server is untrusted. The client encrypts the application's database and queries and sends them to the server who executes them. The key material is stored by the client so the server never sees the data or queries in plaintext. Note that the client is stateless and only stores the schema of the plaintext database and the cryptographic keys.

The client is central to the KafeDB architecture, and most of its modules have to be carefully designed such that any given standard relational database can be used on the server. The most important modules are the *crypto engine* and the *emulation engine*, which are used throughout the data management cycle for data setup, query optimization and execution.

Crypto engine. In KafeDB, end-to-end encryption is handled by a crypto engine that implements the database encryption scheme. It is responsible for encrypting the database and queries and for decrypting the results. Currently, KafeDB's crypto engine implements two of our STE-based constructions, OPX and PKFK, but future versions could be based on new and improved schemes.

Emulation engine. Once a database or query is encrypted it is handed to the emulation engine which is responsible for transforming them into relational tables and SQL queries to be processed by the server. Note that the tables and SQL queries output by the emulation engine *are not* the same as the tables and SQL queries produced by the application. In fact they are completely different since the latter are representations of the STE tables and queries of the application. Again, KafeDB's emulation engine currently implements a specific emulator designed for this work but it

could be replaced in the future with a different emulator.

Setup. Our current focus is on analytical workloads, therefore we design `KafeDB` to bulk load new data through the setup module at the client[2]. The setup module invokes the crypto engine to encrypt the data into encrypted structures, and then it uses the emulation engine to reshape them into tables and indexes.

Query optimizer. Due to encryption, the `KafeDB` server cannot maintain statistics over the tables and is, therefore, limited in how much it can optimize queries. In fact, since its underlying database encryption scheme achieves minimal setup leakage, the only information the server learns at setup time is the size of the database. Because of this, `KafeDB` does most of its query optimization at the client. One of the main technical contributions of OPX is its ability to support any SPC query plan. More precisely, the encrypted structures and query protocols used by OPX are carefully designed so that the supported operations (i.e., joins, filters, projections) can be queried in any order. This flexibility results in `KafeDB` being optimization-friendly since it can process query plans that are produced by standard query optimizers. Besides filter pushdown, we also identified two additional optimizations that are particularly effective in reducing some of the costs introduced by OPX. These include: *many-to-many join factorization* and *multi-way join flattening*. The former transforms each many-to-many join into two many-to-one joins (this requires an additional encrypted table at setup) and the latter transforms a sequence of multiple joins (i.e. a left deep tree) into separate pairs of joins (i.e. a bushy tree).

Split execution. `KafeDB`'s STE-based crypto engine currently handles conjunctive SQL queries on encrypted data. For more complex queries, we use split execution as introduced in [94]. Given a query, the client splits it into two kinds of subqueries: conjunctive subqueries, which are supported by the OPX crypto engine, and other subqueries which are not. The conjunctive subqueries are processed by the crypto engine and the others are executed locally using the results of the conjunctive subqueries.

[2]We defer the extension on secure fine-grained updates that are ACID-compliant to follow-up work.

2.3.1 The STE-based Crypto Engine

The Crypto Engine in our architecture is designed to be a generic component that can plug in any STE-based database encryption scheme. The main challenge for this generic design is to have a sophisticated Emulation Engine that can renders any given STE-scheme legacy friendly without changing security and efficiency.

By relegating much complication of legacy compatibility to emulation, we can therefore concentrate on improving the STE-based scheme on three aspects: security, expressivity and functionality, while having the confidence that these cryptographic improvement can be translated directly into the practical improvement of the whole `KafeDB` system. This is possible because the security of `KafeDB` has to rely on that of the specific STE-based scheme in the Crypto Engine. Second, the functionality such as language expressivity of `KafeDB` also depends on what the underlying STE-based scheme offers. Lastly the efficiency of `KafeDB` may also vary given which STE-based scheme is used.

In this section, we give an intuitive description of our Crypto Engine based on two STE-based schemes OPX (Ch. 3) and PKFK (Ch. 4) and of our emulation techniques (Sec. 5.1), and refer to for the formal treatment and analysis. First, however, we provide some technical background that is necessary to understand our construction.

2.3.2 Overview of OPX and PKFK

Both OPX and PKFK schemes can be divided into two parts: (1) a setup phase during which the client outputs the encrypted database; and (2) a query phase during which the client sends an encrypted query. Here, we only focus on the important ideas behind the scheme.

Setup. The setup takes as input the plaintext database DB and a security parameter (i.e., the length of the cryptographic keys), and constructs various representations of DB that are amendable to structured encryption. The details of representations vary between OPX and PKFK. For example, OPX creates six data structures that capture different representations of the database:

- *(row representation)* MM_R is a multi-map that maps each row identifier to a tuple composed

of the cells of the row;

- *(column representation)* MM_C is a multi-map that maps each column identifier to a tuple composed of the cells of the the column;
- *(filter representation)* MM_V is a multi-map that maps the unique values in every column to the rows that contain that value;
- *(partial join representation)* $\{\mathsf{MM}_{\mathbf{c},\mathbf{c}'}\}_{\mathbf{c},\mathbf{c}'}$ is a set of multi-maps that correspond to a pair of *joinable* columns in the database. Each multi-map maps row identifiers in one column to the row identifiers in the other column that have equal cell value;
- *(full join representation)* $\{\mathsf{MM}_{\mathbf{c}}\}_{\mathbf{c}}$ is a set of multi-maps, each of which corresponds to a column $\mathbf{c}$ of the database. Each multi-map maps a column identifier $\mathbf{c}'$ to the pairs of row identifiers that have the same value in both $\mathbf{c}$ and $\mathbf{c}'$;
- *(partial filter representation)* SET is a set-membership structure that checks whether a cell in a specific row contains a specific value.

All the multi-maps are encrypted using a multi-map encryption scheme. The set structure is encrypted using a custom encrypted set scheme that we detail in the full version of this work. An important aspect of OPX and PKFK is the ability to make use of any combination of these structures to answer any conjunctive query. To do so, both schemes leverage a key design technique in structured encryption called *structural chaining*. In its simplest form, it works as follows: the client sends an encrypted query that can only be used with one of the encrypted structure. Once the server executes this encrypted query, it reveals an intermediary response which is composed of other precomputed encrypted queries that were stored in the encrypted structure during setup. The server can then use these encrypted queries to query other encrypted structures. As a concrete example, in OPX, the client sends an encrypted SQL query to retrieve all rows that are equal to some specific value. The server will take this encrypted query and run it against EMM_V, the encrypted multi-map of MM_V, which outputs all the necessary encrypted queries to run against EMM_R.

Query. The query phase takes as input an optimized SQL query tree whose nodes are relational operators. Then it proceeds to replace each node with a corresponding token for a specific encrypted structure. The emulation engine then emulates the token tree as an encrypted SQL query and sends it to the server. The server executes the encrypted query leveraging the structural chaining discussed above.

Efficiency. One advancement of OPX and PKFK over prior works is on efficiency. OPX is the first STE-based scheme that can support query optimization such that its encrypted query can incur lower complexity at execution. Its storage overhead over plaintext comes from the creation of multiple (encrypted) database representations. While most of these representations do not significantly increase the asymptotic storage overhead, the partial and full join representations can incur a worst-case quadratic blowup in database size. We use an idea of query rewrite called many-to-many join factorization to circumvent this blowup. At a high level, this new query rewrite rule factors a many-to-many join (e.g. foreign key to foreign key) into two many-to-one joins (e.g. foreign key to primary key) such that each only incurs linear complexity.

It turns out that if the complexity measure is changed to the *database size* as in the database literature([101, 73]), then both OPX and SPX are suboptimal: quadratic time and space in input database size. To improve, the PKFK scheme finally bridges the complexity gap using the latter, more typical complexity measure. The key improvement comes from the discovery of STE-based optimal join algorithm (Sec. 4.3.1). Furthermore through PKFK we study how to increase locality in STE-based schemes, in particular through an STE-based colocation technique (Ch. 5).

Security. Because both OPX and PKFK use encrypted multi-maps extensively, their leakage depend on the leakage of their underlying encrypted multi-map constructions. In our current instantiation, OPX and PKFK leak a combination of query equality and response identity patterns. At a high level, the quality equality reveals frequency information on how the client accesses the database such as if and when the client sends the same query. The server can also learn which query touches which rows, as well as the rows touched rows that are common between different encrypted queries through the response identity patterns. However OPX and PKFK—and therefore

KafeDB— *provably* leak significantly less than PPE-based schemes and systems like CryptDB [84] and Monomi [94]. Furthermore, their leakage profile is not prone to any known practical attack. We refer the formal security proofs to their respective chapters.

2.3.3 Overview of Emulation

While STE-based solutions are efficient and more secure than PPE-based solutions they have an important limitation: they require a custom server and, therefore, are not legacy-friendly. To address this, we introduce a new technique called *emulation* that can make STE-based schemes like OPX legacy-friendly. While the notion of emulation is generally applicable, in this project we focused on designing *SQL emulators*; that is emulators to make OPX run on any unmodified relational RDBMS. The main advantages of our emulator are: (1) it does not impact OPX or PKFK's efficiency; (2) it preserves its security; and (3) it is agnostic to the underlying relational DBMS. An emulator consists of two algorithms: a *reshape* algorithm and a *reform* algorithm. We will provide details in Ch. 5.

Reshape. This algorithm transforms the encrypted database, which consists of a set of EMMs, into a set of relational tables. It relies on sub-emulators that transform the individual EMMs into tables. In our current KafeDB implementation, we use the Pibase EMM from [30] so our sub-emulator is designed for that particular construction. At a high level, the sub-emulators parse each EMM into a set of label/tuple pairs which it then inserts as a row into a table. It then creates a plaintext index on the (encrypted) label column. For example in Figure 2.1, we summarize all the generated tables for OPX.

Reform. This algorithm transforms the (encrypted) query tree into a normal SQL query. Here, we use Common Table Expressions (CTEs) to capture the recursive nature of Pibase EMM queries. The main challenges in the reform algorithm is (1) relational algebra does not have certain language constructs that are used to describe pre-emulated query algorithms, such as the counter-based loop; (2) the efficiency of the reformed query may not be as efficient as the pre-emulated query algorithm. We address these challenges in more details in Ch. 5.

2.3.4 Overview of Colocation

Relational data contains much information about data locality, such as values that are colocated in the same rows, columns or tables based on logical relationship. This locality informs database systems how to improve query efficiency, such as (1) by prefetching colocated values from slower storage to faster cache, or (2) by storing colocated values in nearby storage blocks. However, this locality is lost in STE-based schemes such as SPX [63] and PKFK mainly due to encryption. As a result the STE-based databases may have limited scalability. For example in Figure 2.1, the encrypted structures emulated as tables used in OPX and stored on the `KafeDB` server do not convey any colocation of rows or columns as in the plaintext data model. Therefore in Chapter 5 we study the STE-based technique to increase locality by colocaiton and its security trade-off.

Chapter 3

OPX: Query Optimization

3.1 Introduction

Though SPX [63] supports SPC algebra, it is not efficient enough to yield a system that is competitive with commercial plaintext database management systems (DBMS). This stems from several reasons which we now discuss.

Query processing and optimization. Database systems process SQL queries in a series of steps. First, a SQL query is converted into a *logical query tree* which is a tree-based representation of the query where each node is a relational algebra operator. Query trees are evaluated bottom up by evaluating the operators at the leaves on the appropriate database tables. The intermediate table that results from an operation is then passed on to its parent node until the final result table is output by the root. The initial query tree is then converted by a *query optimizer* to an equivalent but optimized query tree using various optimization techniques.

Query optimizers are one of the most important components of a DBMS and a large part of why commercial systems are so efficient. It follows then that for encrypted database systems to be competitive with commercial systems, they must support some form of query optimization. However, the SPX construction does not allow for query optimization because it only handles queries in heuristic normal form (HNF) which is a very specific form of query tree.

An overview of SPX. We briefly recall how SPX works at a high-level. First, note that any conjunctive SQL query can be represented as an SPC query [34] which, in turn, can be represented as a query tree with select/filter, projection and cross product operations. SPX makes use of two kinds of encrypted data structures: encrypted multi-maps (EMM) which map encrypted labels to encrypted tuples and encrypted dictionaries which map encrypted labels to encrypted values. A database $\mathsf{DB} = (\mathbf{T}_1, \ldots, \mathbf{T}_n)$ is encrypted as $\mathsf{EDB} = (\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, \mathsf{EDX})$ where EMM_R and EMM_C store encryptions of the rows and columns in the database, respectively; where EMM_V is used to process filter operations and where EDX is an encrypted dictionary that stores a set of encrypted multi-maps $\{\mathsf{EMM}_{\mathbf{c},\mathbf{c}'}\}_{\mathbf{c},\mathbf{c}' \in \mathsf{DB}}$ that are used to process joins between columns $\mathbf{c}$ and $\mathbf{c}'$. Given a query tree, SPX evaluates leaf operations by querying one of its EMMs directly and then uses various algorithms to process the internal operations on intermediate results. While the leaf operations are handled optimally thanks to the EMMs, internal operations are not necessarily handled in optimal or even sub-linear time.

Sub-optimality of correlated queries. Another source of SPX's sub-optimality is comes from how it handles correlated queries. Roughly speaking, a conjunctive SQL query is uncorrelated if the terms of its `WHERE` clause include attributes/columns that are in different tables. The query trees of uncorrelated queries are relatively simple: they have height 1 with leaves that are either join or filter operations and a root that is a Cartesian product. From the discussion above, one can see that only leaf filters in SPX are evaluated optimally by directly querying the EMMs. Correlated queries, on the other hand, have query trees of height 2 or more which means they have internal operations which, as discussed above, are not necessarily handled optimally.

Our contributions. In this work, we describe an extension of the SPX construction, called OPX, that supports query optimization. It does this by using additional encrypted structures that are designed to optimally handle internal operations by taking inputs from outputs of other operations. These additional structures include an encrypted set structure to handle internal filters and an additional set of encrypted multi-maps to handle internal joins. These additional structures increase the storage overhead but only concretely; asymptotically-speaking OPX has the same

storage overhead as SPX. The leakage profile of OPX is also similar to that of SPX. In addition to executing internal operations more efficiently, OPX has the advantage that it can handle any query tree; not just HNF trees. This is an important feature because it means that OPX can be used to query trees that have been optimized by standard query optimizers.

3.2 The OPX Construction

We now describe our main construction, OPX, which is a response-hiding STE scheme for relational databases that supports conjunctive SQL queries. It uses a response-revealing multi-map encryption scheme Σ_{MM}, the adaptively-secure encrypted multi-map scheme $\Sigma^{\pi}_{\mathsf{MM}}$ by Cash et al. [30], a symmetric encryption scheme SKE, a pseudo-random function F, and a random oracle H. We describe the scheme in detail in Figures (3.1), (3.2), (3.3), and (3.4), and provide a high level description below.

Remark on notation. We note that the syntax of OPX matches Definition 1.3.2 but its queries q are query trees and its tokens tk are token trees; that is, trees where each node is a sub-token. We make this explicit here by referring to query trees as QT and to token trees as TT. Throughout, while processing a query tree, we denote by $\mathbf{R_{in}}$ the set of inputs to an operation/node and by $\mathbf{R_{out}}$ the set of outputs of that operation/node.

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

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

Let $\Sigma_{\mathsf{MM}} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Get})$ be a response-revealing multi-map encryption scheme, $\Sigma^{\pi}_{\mathsf{MM}} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Get})$ be the response-revealing multi-map encryption scheme in [30], $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme, $F : \{0,1\}^k \times \{0,1\}^\star \to \{0,1\}^m$ be a pseudo-random function, and $H : \{0,1\}^\star \to \{0,1\}^m$ be a random oracle. 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 SET;
  2. initialize multi-maps MM_R, MM_C and MM_V;
  3. initialize multi-maps $(\mathsf{MM}_{\mathsf{att}})_{\mathsf{att} \in \mathsf{DB}^{\mathsf{T}}}$;
  4. initialize multi-maps $(\mathsf{MM}_{\mathsf{att},\mathsf{att}'})_{\mathsf{att},\mathsf{att}' \in \mathsf{DB}^{\mathsf{T}}}$ such that $\mathsf{dom}(\mathsf{att}) = \mathsf{dom}(\mathsf{att}')$;
  5. sample two keys $K_1, K_F \xleftarrow{\$} \{0,1\}^k$;
  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 $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$, 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 $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$,
      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 $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$,
      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 rows $\mathbf{r}_i$ and $\mathbf{r}_j$ in $\mathbf{c}$ and $\mathbf{c}'$, such that $\mathbf{c}[i] = \mathbf{c}'[j]$,
            1. compute $\mathsf{rtk}_i \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_R, \chi(\mathbf{r}_i)\Big)$;
            2. compute $\mathsf{rtk}_j \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_R, \chi(\mathbf{r}_j)\Big)$;
            3. 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 3.1: The OPX scheme (Part 1).

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

  13. for all $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$,
      a. for all $v \in \mathbf{c}$,
          i. compute $K_v \leftarrow F_{K_F}(\chi(\mathbf{c}) \| v)$;
          ii. set for all $\mathbf{r} \in \mathsf{DB}_{\mathbf{c}=v}$,
          $$\mathsf{SET} := \mathsf{SET} \bigcup \left\{ H\big(K_v \| \mathsf{rtk}\big) \right\},$$
          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 $\mathbf{c}[i] = \mathbf{c}'[j]$,
              1. add $\big(\mathsf{rtk}_i, \mathsf{rtk}_j\big)$ to $\mathbf{t}$ where
              $$\mathsf{rtk}_i \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}_i)) \quad \text{and} \quad \mathsf{rtk}_j \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K_R, \chi(\mathbf{r}_i j).$$
          iii. for all rtk s.t. $(\mathsf{rtk}, \cdot) \in \mathbf{t}$, set
          $$\mathsf{MM}_{\mathbf{c},\mathbf{c}'}\Big[\mathsf{rtk}\Big] := \Big(\mathsf{rtk}'\Big)_{(\mathsf{rtk},\mathsf{rtk}') \in \mathbf{t}}$$
      b. compute $(K_{\mathbf{c},\mathbf{c}'}, \mathsf{EMM}_{\mathbf{c},\mathbf{c}'}) \leftarrow \Sigma^{\pi}_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}_{\mathbf{c},\mathbf{c}'}\big)$;
  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 3.2: The OPX 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 $N \equiv \mathbf{R}_{\mathbf{in}}^{(l)} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R}_{\mathbf{in}}^{(r)}$ 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 $\mathbf{R}_{\mathbf{in}}^{(l)}$ and $\mathbf{R}_{\mathbf{in}}^{(r)}$, 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 3.3: The OPX scheme (Part 3).

- $\mathsf{Query}(\mathsf{EDB}, \mathsf{tk})$:
  1. parse EDB as $(\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, (\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}' \in \mathsf{DB}^\intercal}, \mathsf{SET}, (\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c} \in \mathsf{DB}^\intercal})$.
  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 $N \equiv (\mathsf{rtk}, \mathtt{pos})$, then for all rtk in $\mathbf{R_{in}}$ in the column at position $\mathtt{pos}$, if
     $$H(\mathsf{rtk} \| \mathsf{rtk}) \notin \mathsf{SET}$$
     then it removes the row from $\mathbf{R_{in}}$. Finally, it sets $\mathbf{R_{out}} := \mathbf{R_{in}}$;
     - if $N \equiv (\mathsf{jtk}, \mathtt{pos})$, then it computes
     $$\Big((\mathsf{rtk}_1, \mathsf{rtk}'_1), \ldots, (\mathsf{rtk}_s, \mathsf{rtk}'_s)\Big) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}(\mathsf{jtk}, \mathsf{EMM}_{\mathtt{pos}}),$$
     and sets
     $$\mathbf{R_{out}} := \Big((\mathsf{rtk}_i, \mathsf{rtk}'_i)\Big)_{i \in [s]};$$
     - if $N \equiv (\mathsf{etk}, \mathtt{pos}_1, \mathtt{pos}_2)$, 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
     $$(\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s) \leftarrow \Sigma^{\pi}_{\mathsf{MM}}.\mathsf{Query}(\mathsf{ltk}, \mathsf{EMM}_{\mathtt{pos}_1, \mathtt{pos}_2}),$$
     where $\mathsf{rtk} = \mathbf{r}[\mathsf{att}_{\mathtt{pos}_2}]$, and appends the new rows $\Big(\mathsf{rtk}_i\Big)_{i \in [s]} \times \mathbf{r}$ to $\mathbf{R_{out}}$;
     - if $N \equiv (\mathtt{pos}_1, \mathtt{pos}_2)$, then it sets
     $$\mathbf{R_{out}} := \mathbf{R}^{(l)}_{\mathbf{in}} \bowtie_{\mathtt{pos}_1 = \mathtt{pos}_2} \mathbf{R}^{(r)}_{\mathbf{in}},$$
     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 $N \equiv (\mathtt{pos}_1, \cdots, \mathtt{pos}_z)$, then it computes $\mathbf{R_{out}} := \pi_{\mathtt{pos}_1, \cdots, \mathtt{pos}_z}(\mathbf{R_{in}})$;
     - 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 3.4: The OPX scheme (Part 4).

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

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

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 $\{\mathsf{MM}_{\mathbf{c}}\}_{\mathbf{c}\in\mathsf{DB}^{\mathsf{T}}}$. For all columns $\mathbf{c}, \mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}$ 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 $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$,

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

It initializes a set SET and computes for each column $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$, 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

$$\mathsf{SET} := \mathsf{SET} \bigcup \Big\{ H(K_v \| \mathsf{rtk}) \Big\},$$

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 $\mathbf{c}, \mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}$ 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

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

then computes

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

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. 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

$$\mathsf{stk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_V, \langle a, \chi(\mathsf{att})\Big);$$

- **(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 $(\mathsf{rtk}, \mathtt{pos})$ where

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

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

- **(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 $(\mathsf{jtk}, \mathtt{pos})$ 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 $\mathtt{pos}$ 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 $(\mathsf{etk}, \mathtt{pos}_1, \mathtt{pos}_2)$ where

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

  and $\mathtt{pos}_1$, $\mathtt{pos}_2$ 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 $(\mathtt{pos}_1, \mathtt{pos}_2)$ where $\mathtt{pos}_1$ and $\mathtt{pos}_2$ are the column positions of att_1 and att_2 in $\mathbf{R}_{\mathbf{in}}^{(l)}$ and $\mathbf{R}_{\mathbf{in}}^{(r)}$, 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 $\pi_{\mathsf{att}_1,\cdots,\mathsf{att}_z}(\mathbf{R}_{\mathbf{in}})$ then set the corresponding node to

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

  where $\mathtt{pos}_i$ is the column position of att_i in $\mathbf{R}_{\mathbf{in}}$.

- **(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

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

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

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

$$H(\mathsf{rtk} \| \mathsf{rtk}) \notin \mathsf{SET}$$

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

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

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

and sets

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

- **(internal join):** if N has form $(\mathsf{etk}, \mathtt{pos}_1, \mathtt{pos}_2)$, 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

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

  where $\mathsf{rtk} = \mathbf{r}[\mathsf{att}_{\mathtt{pos}_2}]$, 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 $(\mathtt{pos}_1, \mathtt{pos}_2)$, then it sets

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

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

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

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

- **(internal projection):** if N is an internal node of form $(\mathtt{pos}_1, \cdots, \mathtt{pos}_z)$, then it computes

$$\mathbf{R_{out}} := \pi_{\mathtt{pos}_1, \cdots, \mathtt{pos}_z}(\mathbf{R_{in}})$$

- **(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 $\mathbf{R}_{\mathbf{in}}^{(l)}$ and $\mathbf{R}_{\mathbf{in}}^{(r)}$ are the left and right input respectively.

Now, it replaces each cell rtk in $\mathbf{R}_{\mathbf{out}}^{\mathsf{root}}$ by

$$\mathrm{ct} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}(\mathsf{rtk}, \mathsf{EMM}_R).$$

3.2.1 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 3.2.1. *If Σ_{mm} is optimal, then the time and space complexity of the* Query *algorithm presented in Section (3.2) is optimal.*

The proof of the theorem is in Section 3.4.

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 [42, 36, 67, 31, 30] 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 $O(\sum_{\mathbf{c} \in \mathsf{DB}^{\intercal}} \#\mathbf{c})$, 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 $O(\sum_{\mathbf{c} \in \mathsf{DB}^{\intercal}} \sum_{v \in \mathbf{c}} \#\mathsf{DB}_{\mathsf{att}(\mathbf{c})=v})$ storage. Similarly, SET contains the pseudo- random evaluation of the coordinate of all rows in the database and therefore requires $O(\sum_{\mathbf{c} \in \mathsf{DB}^{\intercal}} \sum_{v \in \mathbf{c}} \#\mathsf{DB}_{\mathsf{att}(\mathbf{c})=v})$. For

each $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$, 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 $(\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c}\in\mathsf{DB}^{\mathsf{T}}}$ has size

$$O\Big(\sum_{\mathbf{c}\in\mathsf{DB}^{\mathsf{T}}}\sum_{\mathbf{c}':\mathsf{dom}(\mathsf{att}(\mathbf{c}'))=\mathsf{dom}(\mathsf{att}(\mathbf{c}))} \#\mathsf{DB}_{\mathsf{att}(\mathbf{c})=\mathsf{att}(\mathbf{c}')}\Big).$$

Similarly, for all $\mathbf{c}, \mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}$, 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 $(\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^{\mathsf{T}}}$ 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 $O((\sum_i \|\mathbf{T}_i\|_c)^2)$ 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 Sec. (3.6), we provide a concrete example of the storage overhead of an encrypted database.

3.3 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 [63], 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. [30]. 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.

3.3.1 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[1]

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

where $\mathcal{L}_{\mathsf{S}}^{\mathsf{mm}}$, $\mathcal{L}_{\mathsf{S}}^{\pi}$, n and $\|\mathbf{T}_i\|$ are the setup leakage of Σ_{MM}, the setup leakage of $\Sigma_{\mathsf{MM}}^{\pi}$ 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* LT 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 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

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

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

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

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

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 $\mathcal{P}(\mathsf{pnode})$ captures the leakage produced when the server queries EMM_C to retrieve the encrypted content of the column att. More precisely, $\mathcal{P}(\mathsf{pnode})$ 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 $\mathcal{P}(\mathsf{pnode})$ 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

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

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

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

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 $\mathcal{S}(\mathsf{snode})$ captures the leakage produced when the server queries EMM_V to retrieve some row tokens. More precisely, $\mathcal{S}(\mathsf{snode})$ 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 $\mathcal{S}(\mathsf{snode})$ 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, $\mathcal{S}(\mathsf{snode})$ 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

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

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,

$$\begin{aligned}\mathcal{J}(\mathsf{jnode}) = &\Big(\mathtt{leaf}, f(\mathsf{att}_1), \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}\Big(\mathsf{MM}_{\mathsf{att}_1}, \Big\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2) \Big\rangle\Big),\\ &\Big\{\mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r}_1), \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}}(\mathsf{MM}_R, \chi(\mathbf{r}_2)\Big\}_{(\mathbf{r}_1,\mathbf{r}_2)\in \mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}}\Big),\end{aligned}$$

In this case, $\mathcal{J}(\mathsf{jnode})$ 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,

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

where $\mathbf{R_{in}}[\mathsf{att}]$ denotes the cell values in $\mathbf{R_{in}}$ at attribute att. In this case, $\mathcal{J}(\mathsf{jnode})$ 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 $\Sigma^{\pi}_{\mathsf{MM}}$ 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 $\Sigma^{\pi}_{\mathsf{MM}}$ reveals if and when the same query is evaluated (search pattern) as well as the response identifiers (access pattern). If jnode has form $\mathbf{R}^{(l)}_{\mathbf{in}} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R}^{(r)}_{\mathbf{in}}$ then,

$$\mathcal{J}(\mathsf{jnode}) = \Big(\mathtt{inter}, f(\mathsf{att}_1), f(\mathsf{att}_2)\Big),$$

In this case, $\mathcal{J}(\mathsf{jnode})$ captures the leakage produced when the server removes all the rows in $\mathbf{R}^{(l)}_{\mathbf{in}} \times \mathbf{R}^{(r)}_{\mathbf{in}}$ 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

$$\Lambda := \Lambda \bigcup \Big\{\mathcal{J}(\mathsf{jnode})\Big\}.$$

Finally, it sets

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

3.3.2 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 3.3.1. *If F is a pseudo-random function, SKE is RCPA secure, $\Sigma^{\pi}_{\mathsf{MM}}$ is adaptively $(\mathcal{L}^{\pi}_{\mathsf{S}}, \mathcal{L}^{\pi}_{\mathsf{Q}})$-secure, and Σ_{MM} is adaptively $(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}})$-secure, then OPX is adaptively $(\mathcal{L}^{\mathsf{opx}}_{\mathsf{S}}, \mathcal{L}^{\mathsf{opx}}_{\mathsf{Q}})$-secure in the random oracle model.*

The proof of Theorem 3.3.1 is in Section 3.5.

3.3.3 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 $\Sigma^{\pi}_{\mathsf{MM}}$ from [30] 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 [65],

VLH or AVLH [64] if built using the adaptively-secure $\Sigma^{\pi}_{\mathsf{MM}}$ 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

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

To instantiate such a scheme, one can use oblivious RAM (ORAM) simulation techniques [51] in a black-box fashion, or more customized/advanced schemes such as the oblivious tree structures (OTS) [98] or the TWORAM construction [48] with a careful parametrization of the block-sizes, or the AZL construction based on the piggy-backing scheme PBS [65]. 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

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

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

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

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

$$\mathcal{J}(\mathsf{jnode}) = \left(\mathtt{leaf}, f(\mathsf{att}_1), \left\{|\mathbf{r}_1|, \mathrm{AccP}(\mathbf{r}_1), |\mathbf{r}_2|, \mathrm{AccP}(\mathbf{r}_2)\right\}_{(\mathbf{r}_1,\mathbf{r}_2)\in \mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2}}\right),$$

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

$$\mathcal{J}(\mathsf{jnode}) = \left(\mathtt{in}, \langle f(\mathsf{att}_1), f(\mathsf{att}_2)\rangle, \left\{|\mathbf{r}_1|, \mathrm{AccP}(\mathbf{r}_1)\right\}_{\substack{(\mathbf{r}_1,\mathbf{r}_2)\in \mathsf{DB}_{\mathsf{att}_1=\mathsf{att}_2} \\ \wedge \chi(r_2)\in \mathbf{R_{in}}[\mathsf{att}_2]}}\right),$$

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

$$\mathcal{J}(\mathsf{jnode}) = \left(\mathtt{inter}, f(\mathsf{att}_1), f(\mathsf{att}_2)\right),$$

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

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 3.2.1 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 [42, 36, 67, 31, 30]. 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

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

  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

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

  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

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

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

$$O\Big(m \cdot \log\Big(m \cdot \sum_{i=1}^{n} \|\mathbf{T}_i\|_c\Big)\Big),$$

  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 [42, 36, 67, 31, 30]. 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 3.2.1 that the execution of the OPX query algorithm and its plaintext counterpart have exactly the same asymptotics.

3.4 Proof of Theorem 3.2.1

Theorem 3.2.1. *If the query algorithm of Σ_{mm} is optimal, then the time and space complexity of the* Query *algorithm presented in Section (3.2) 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 $O(m)$.[2]

  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 from the input $\mathbf{R}_{\mathbf{in}}$ which has search complexity equal to $O(\#\mathbf{R}_{\mathbf{in}}[\mathsf{att}])$ which is the number of cells of the attribute att in $\mathbf{R}_{\mathbf{in}}$. 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

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

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

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

  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.

■

3.5 Proof of Theorem 3.3.1

Theorem 3.3.1. *If F is a pseudo-random function, SKE is RCPA secure, $\Sigma^{\pi}_{\mathsf{MM}}$ is adaptively $(\mathcal{L}^{\pi}_{\mathsf{S}}, \mathcal{L}^{\pi}_{\mathsf{Q}})$-secure, and Σ_{MM} is adaptively $(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}, \mathcal{L}^{\mathsf{mm}}_{\mathsf{Q}})$-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 $\mathsf{EMM}_R \leftarrow \mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_R))$, $\mathsf{EMM}_C \leftarrow \mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_C))$, $\mathsf{EMM}_V \leftarrow \mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_V))$, for all $\mathbf{c} \in \mathsf{DB}^{\mathsf{T}}$, $\mathsf{EMM}_{\mathbf{c}} \leftarrow \mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_{\mathbf{c}}))$, and for all $\mathbf{c}, \mathbf{c}' \in \mathsf{DB}^{\mathsf{T}}$, $\mathsf{EMM}_{\mathbf{c},\mathbf{c}'} \leftarrow \mathcal{S}^{\pi}_{\mathsf{MM}}(\mathcal{L}^{\pi}_{\mathsf{S}}(\mathsf{MM}_{\mathbf{c},\mathbf{c}'}))$. Given (n, ρ), it instantiates an empty set SET, and inserts $r_{i,j} \xleftarrow{\$} \{0,1\}^k$ in SET for $i \in [n]$ and $j \in [\rho]$. $\mathcal{S}$ outputs

$$\mathsf{EDB} = (\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, (\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c}\in\mathsf{DB}^{\mathsf{T}}}, \mathsf{SET}, (\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^{\mathsf{T}}}).$$

Recall that OPX is response-hiding so $\mathcal{S}$ receives $(\perp, \mathcal{L}^{\mathsf{opx}}_{\mathsf{Q}}(\mathsf{DB}, \mathsf{QT}))$ as input in the $\mathbf{Ideal}_{\mathsf{SPX},\mathcal{A},\mathcal{S}}(k)$

experiment. Given this input, $\mathcal{S}$ parses $\mathcal{L}_{\mathsf{Q}}^{\mathsf{opx}}(\mathsf{DB}, \mathsf{QT})$ as a leakage tree. It then instantiates a token tree TT with the same structure. It samples uniformly at random a key $K_1 \xleftarrow{\$} \{0,1\}^k$, 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 $(\mathtt{scalar}, |a|)$ then it sets TT_N to $[\mathsf{Enc}_{K_1}(\mathbf{0}^{|a|})]$. Otherwise if N has form $(\mathtt{cross}, \perp)$, then it sets TT_N to $\times$.
- **(Projection).** If N has form $\Big(\mathtt{leaf}, \mathcal{L}_{\mathsf{Q}}^{\mathsf{mm}}\Big(\mathsf{MM}_C, \chi(\mathsf{att})\Big)\Big)$ then it sets

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

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

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

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

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

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

  then it sets,

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

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

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

  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

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

– it samples a key $K_{g(a\|\mathsf{att})} \xleftarrow{\$} \{0,1\}^k$;

– 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

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

– 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

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

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

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

and

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

it then sets

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

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

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

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

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

then if $\langle f(\mathsf{att}_1), f(\mathsf{att}_2)\rangle$ has never been queried before, and by leveraging the key-equivocation of $\Sigma^{\pi}_{\mathsf{MM}}$, it generates a key such that[3]

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

[3]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.

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

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

- **(Join case-3).** If N has form $\Big(\mathtt{inter}, f(\mathsf{att}_1), f(\mathsf{att}_2)\Big)$, 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 $\mathbf{Real}_{\mathsf{OPX},\mathcal{A}}(k)$ outputs 1 is negligibly-close to the probability that $\mathbf{Ideal}_{\mathsf{OPX},\mathcal{A},\mathcal{S}}(k)$ outputs 1. We do this using the following sequence of games:

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

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

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

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

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

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 $\mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_{\mathbf{c}_i}))$ 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

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

$\mathsf{Game}_{3+\#\mathsf{DB}^{\mathsf{T}}}$: 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.

$\mathsf{Game}_{3+\#\mathsf{DB}^\intercal+i}$ for $i \in [(\#\mathsf{DB}^\intercal)^2]$: is the same as $\mathsf{Game}_{2+\#\mathsf{DB}^\intercal+i}$, except that $\mathsf{EMM}_{\mathbf{c}_i,\mathbf{c}'_i}$ is replaced with the output of $\mathcal{S}_{\mathsf{MM}}(\mathcal{L}^{\mathsf{mm}}_{\mathsf{S}}(\mathsf{MM}_{\mathbf{c}_i,\mathbf{c}'_i}))$, 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

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

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

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

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 $\mathbf{Ideal}_{\mathsf{OPX},\mathcal{A},\mathcal{S}}(k)$.

■

3.6 A Concrete Example of Indexed HNF

Similar to [63], 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 $\mathbb{S}(\mathbf{T}_1) = (\mathsf{ID}, \mathsf{Name}, \mathsf{Course})$ and that of $\mathbf{T}_2$ is $\mathbb{S}(\mathbf{T}_2) = (\mathsf{Course}, \mathsf{Department})$. The tables are described in Figure (3.5).

Figure (3.6) shows the result of applying our method to index the database $\mathsf{DB} = (\mathbf{T}_1, \mathbf{T}_2)$, as detailed in Section (3.2). 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.

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

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

Course	Department
16	CS
18	Math

Figure 3.5: Plaintext database DB.

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

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

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

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

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

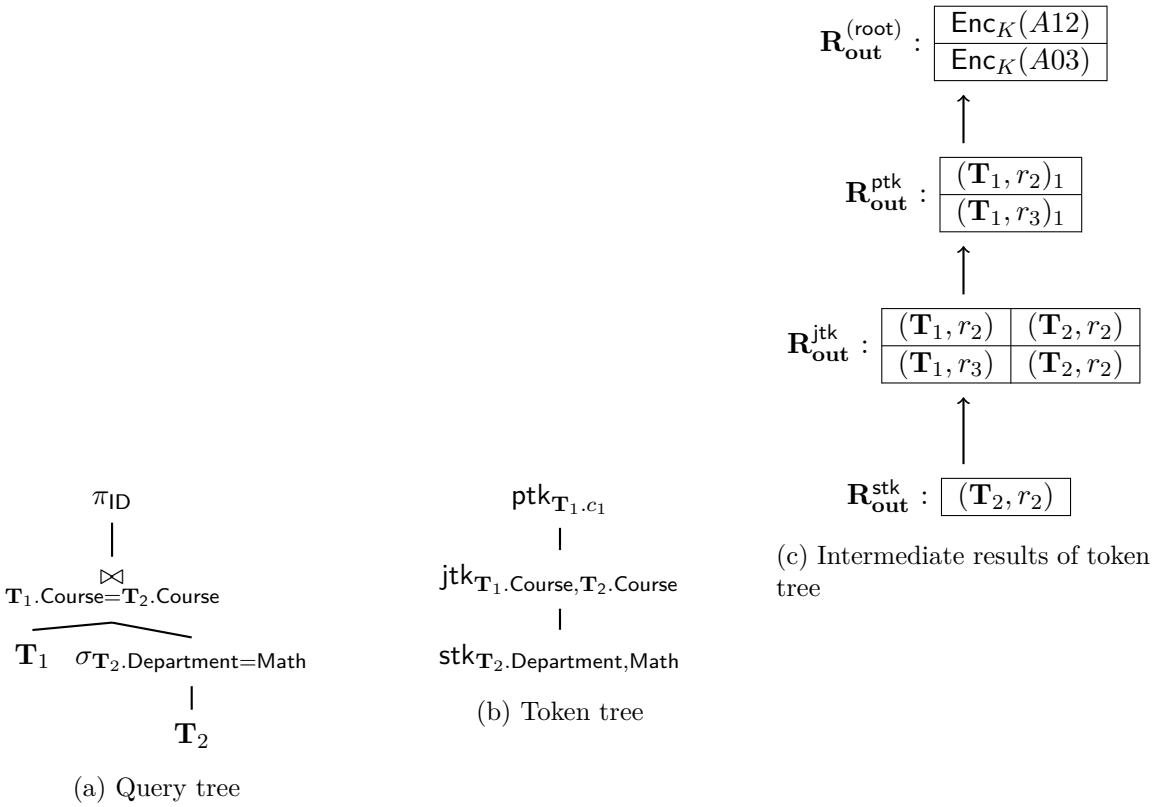
Figure 3.6: Indexed database.

This SQL query can be rewritten as a query tree, see Figure (3.7c), and then translated, based on opx protocol into a token tree as depicted in Figure (3.7b).[5]

We detail in Figure (3.7c) 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

[5]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.



(a) Query tree

(b) Token tree

(c) Intermediate results of token tree

Figure 3.7: A query tree translated to a token tree which is then executed using the indexed database.

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.[6] 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

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

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 database, the $4\times$ 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.

3.7 Limitations of OPX

Although OPX represents the first STE-based scheme that implements relational query optimization, it however still leaves several other important problems open. We briefly overview these problems below and will address them in the subsequent work (Ch. 4, 5).

Suboptimality. Although SPX and OPX are analyzed to be asymptotically linear in query output size, they however have higher asymptotic complexity than the plaintext query algorithms, because the plaintext complexity is measured using a different notion: asymptotically linear in query *input* size in the database literature (e.g. [101, 73]). For example, a hash join can run in time linear in the table size, and takes up storage space linear in the table size. But an SPX or OPX join can take time and space quadratic in table size, because a join may have worst-case quadratic output size. In other words, linearity in output size does not yield a query algorithm that has quadratic output size. Therefore here we momentarily leave this problem unaddressed here as to how to design STE-based join algorithm that matches the plaintext join complexity in terms of query input size rather than the output size. Instead, we will present in Ch. 4 a solution to optimal join and matching overall query/storage complexity to plaintext in the same notion as the database literature, while here for OPX we focus on solving a different efficiency problem: how to support query optimization for queries with multiple operators. In the later chapter we will change to use the query input size as complexity measure as in the database literature, and construct an optimal STE-based join algorithm and conjunctive query algorithm (Ch. 4).

Query leakage. Conjunctive queries that contain multiple selections and joins in both SPX and OPX may leak information about each full join, which roughly is about which rows in the entire tables have equal attribute values, even if only a subset of such rows are filtered for. This leakage may lead to discovery of entire joint frequency between two tables, therefore it is important to reduce its disclosure. This problem is formally addressed in a different scheme (Ch. 4).

Locality. A higher-level encrypted database scheme such as OPX (and also SPX) represents plaintext database in a way that loses much locality information in the plaintext data model, such which values correspond to the same row (i.e. same record), the same column (i.e. same attribute) or the same table (i.e. same concept). It therefore presents a challenge to scalability. The key is recover such locality through STE structures, and we study this problem more in-depth in Ch. 5.

Chapter 4

PKFK: Improved Efficiency and Security

4.1 Introduction

In [11] (Ch. 2), we outlined key design principles for encrypted database systems and a system architecture to achieve them. We also described our preliminary efforts to build a system called `KafeDB` that embodies these principles. In this chapter, we present the new STE-based scheme called PKFK that has most improved efficiency and security over prior schemes such as SPX [64] and OPX [66] (Ch. 3).

The PKFK construction. The original version of `KafeDB` is based on the STE-based database encryption scheme OPX [66] (Ch. 3) which is itself an extension of the SPX scheme [63]. While SPX/OPX handle a non-trivial fraction of SQL, they come with several limitations including quadratic complexity, lack of support for query optimization and relatively large leakage for conjunctive queries. The PKFK scheme successfully addresses the limitations of SPX/OPX. In particular, the PKFK scheme achieves (1) linear storage and query complexity in database size, (2) support for both standard and custom query optimization, and (3) reduced leakage for conjunctive queries. Morever, the PKFK scheme allows flexibility for its encrypted structures to be reshaped in order to gain more access locality. We developed SQL emulators to specifically leverage this property. Our

benchmark (Ch. 6) showed that PKFK had only 4.2× query overhead and 3.6× storage overhead, representing two orders of magnitude improvement over SPX/OPX and similar performance to the PPE-based CryptDB.

4.2 Limitations of SPX

The SPX scheme [63] is the first STE-based construction that handles a nontrivial subset of SQL queries, namely the class of conjunctive queries, which consists of selection (or filter), join and projection operators. However SPX has several limitations that hinder the development of practical and scalable systems. In this chapter we investigate new techniques and constructions to overcome these limitations. We first illustrate these limitations using an example of SPX in Figure 4.1. Note that the limitations except for query optimization apply also to the OPX scheme [66] (Ch. 3).

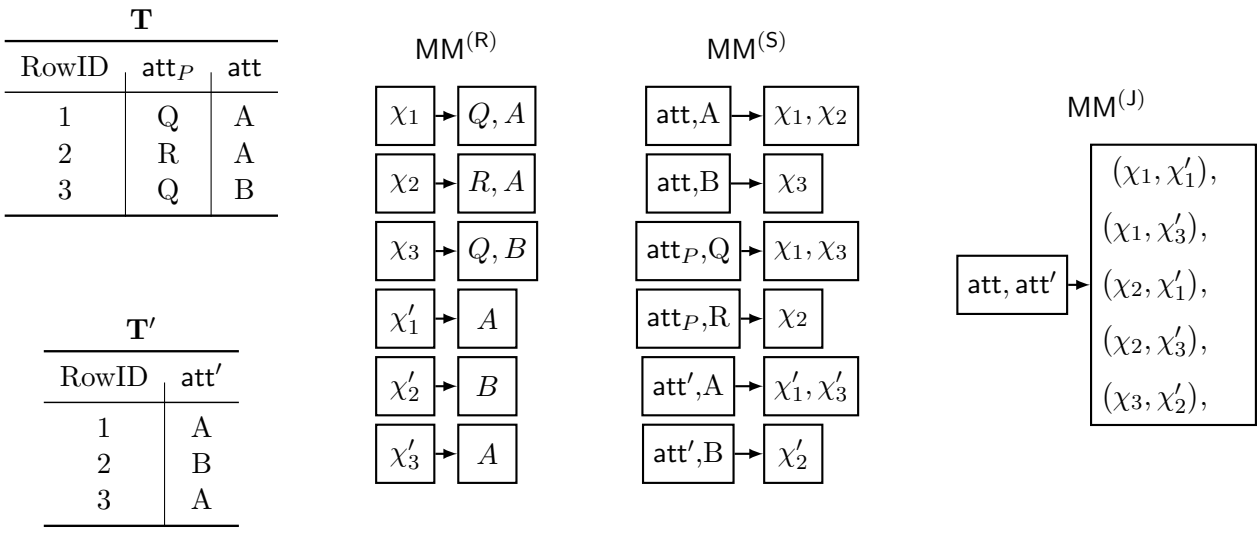


Figure 4.1: An example of SPX [63] construction for plaintext tables $\mathbf{T}$ and $\mathbf{T}'$ for the selection attribute att_P and join attributes att and att'. The notation $\mathsf{MM}^{(\mathsf{S})}$ corresponds to MM_V, $\mathsf{MM}^{(\mathsf{R})}$ to MM_R, and $\mathsf{MM}^{(\mathsf{J})}$ to $\mathsf{MM}_{\mathsf{att}}$ in [63]. Each row coordinate for $\mathbf{T}$ is denoted as χ_i and for $\mathbf{T}'$ as χ'_i.

Example of SPX. First we represent rows in two tables $\mathbf{T}, \mathbf{T}'$ as a multimap $\mathsf{MM}^{(\mathsf{R})}$. We denote the row coordinates to the corresponding encrypted multimap for the ith row in $\mathbf{T}$ as χ_i and for $\mathbf{T}'$

as χ'_i. To represent a join between these two tables, say between columns $\mathsf{att}, \mathsf{att}'$ as in the query

$$\mathbf{T} \bowtie_{\mathsf{att}=\mathsf{att}'} \mathbf{T}',$$

we first need to find the pairs of all rows that constitute the join result. This step effectively requires to compute the join query. Then we compute the pair of row coordinates (χ_i, χ'_j) for each pair of joined rows, and store all such pairs as a tuple inside another multimap for join as

$$\mathsf{MM}^{(\mathsf{J})}[(\mathsf{att}, \mathsf{att}')] = \Big((\chi_i, \chi'_j)\Big)_{\forall i,j:\mathsf{att}[i]=\mathsf{att}'[j]}$$

SPX then encrypts these structures using black-box encrypted multi-maps $\mathsf{EMM}^{(\mathsf{R})}, \mathsf{EMM}^{(\mathsf{S})}, \mathsf{EMM}^{(\mathsf{J})}$, which can be instantiated using standard constructions such as [42, 36, 67, 31, 30].

Quadratic setup and storage complexity. SPX for the join representation will take time and storage proportional to the join size, which is worst-case quadratic in table size $\mathbf{T}$: $\mathcal{O}(|\mathbf{T}| \cdot |\mathbf{T}'|)$. This can be seen by counting the length of the tuple in $\mathsf{MM}^{(\mathsf{J})}$ associated with the join attributes $(\mathsf{att}, \mathsf{att}')$. This cost is significantly higher than the plaintext which is worst-case linear in table size $\mathcal{O}(\mathbf{T})$. Moreover, it is also less desirable to "precompute" the join during setup in order to compute its encrypted representation. Ideally the encryption should take time and space linear in the plaintext database to set up.

Quadratic join complexity. To compute the encrypted join, the server needs to query the corresponding $\mathsf{EMM}^{(\mathsf{J})}$ for the entire tuple associated with this join, whose length is proportional to the join size. This means the encrypted join algorithm is worst-case quadratic $\mathcal{O}(|\mathbf{T}| \cdot |\mathbf{T}'|)$, which is higher than the linear complexity in plaintext join (such as [17]). This complexity gap is problematic for a conjunctive query because its complexity is typically dominated by joins. In this work we construct the first STE-based join that matches the plaintext asymptotic complexity using the same notions in the relational database literature.

Unoptimized conjunctive query. Suppose the query is a filtered join, say the same join between columns $\mathsf{att}, \mathsf{att}'$ but also filtered by $\mathsf{att}_P = R$,

$$\sigma_{\mathsf{att}_P=R}(\mathbf{T} \bowtie_{\mathsf{att}=\mathsf{att}'} \mathbf{T}')$$

In plaintext, this query will be optimized by the *selection pushdown* rule such that the filter would precede the join

$$(\sigma_{\mathsf{att}_P=R}\mathbf{T}) \bowtie_{\mathsf{att}=\mathsf{att}'} \mathbf{T}'$$

Doing so will ensure that the intermediate input to the join is effectively reduced by the filter, and in this case to of size one for only one matched row, Row 2 in $\mathbf{T}$ with $\mathsf{att}_P = R$. Therefore such optimized query would only incur one access for the filtered join. However, SPX does not support such query optimization, therefore it will still retrieve the entire tuple of row coordinates for the whole join from $\mathsf{EMM}^{(\mathsf{J})}$, in this case 5 pairs of row coordinates that constitute the total unfiltered join size. Therefore SPX may still incur the full join cost for a partially-filtered join. In order to have practical solution, we need to investigate how to support relational query optimization in an STE-based scheme.

Large conjunctive query leakage. In the above filtered join example, because the server in SPX still retrieves the entire tuple of the row coordinate pairs even though only one such pair is filtered, it leaks the *equality pattern* for the entire join regardless of the filter. This equality pattern means which row in column att is equal to those in column att'. But this leakage can also be used to infer the number of unique values in the entire domain of columns $\mathsf{att}, \mathsf{att}'$, and their frequency distributions. This leakage for the entire domain seems undesirable especially since this example query only concerns one row from each table. For example if the table stores information about individuals as rows, we do not want to leak information about other individuals when the query filters down to just one individual. In this work we address how to reduce this leakage.

Lack of leagcy compatibility. The data structures $\mathsf{EMM}^{(\mathsf{R})}, \mathsf{EMM}^{(\mathsf{J})}$ and the query algorithms are not tables and relational algebra or SQL, therefore these schemes are believed to require custom

changes to the relational database systems. We propose a general framework called emulation, to turn an STE-based scheme into one that uses different data structures and query languages, such that the STE-scheme can be implemented on legacy systems with no custom modification.

Lack of locality. Exploiting data and access locality is crucial for database efficiency for large data. Such locality information is typically represented in the plaintext table. For example, values on the same row can be accessed together by multiple select and join operators. If the same values are stored in contiguous locations then multiple operators of the same row will only need $\mathcal{O}(1)$ sequential read to the secondary storage. However SPX dose not preserve such locality. N operators on the same row will not only incur $\mathcal{O}(N)$ reads, but these reads are no longer sequential. For example in Figure 4.1, because the select and join operators are represented in two separate encryptd multimaps $\mathsf{EMM}^{(\mathsf{S})}, \mathsf{EMM}^{(\mathsf{J})}$, the same query above with a filter and a join will require two separate accesses to $\mathsf{EMM}^{(\mathsf{S})}$ $\mathsf{EMM}^{(\mathsf{J})}$ for the same row, say $(\mathbf{T}, 3)$, and another access to $\mathsf{EMM}^{(\mathsf{R})}$ to retrieve said row. We study techniques to increase locality in an STE-based scheme in order to further enhance its efficiency with security trade-off.

4.3 Techniques

To overcome the limitations outlined above, we examine the sources of these limitations and propose new techniques to address them. Our optimal join algorithm (Sec. 4.3.1) uses new graph structures to reduce the join complexity to match the plaintext join. Our query optimization technique (Sec. 4.3.2) implements standard database optimization in an encrypted setting. Our leakage reduction technique (Sec. 4.3.3) reduces the new leakage introduced in query optimization while managing the storage blowup. Beyond these techniques, in Chapter 5 we also introduce Emulation to render STE-based schemes legacy-friendly, and Colocation to increase locality. Finally we combine these techniques in a new construction called PKFK [1] (Sec. 4.4 and 4.4.4).

[1]The name is associated with the relational database concept Primary Key and Foreign Key. It serves as a tribute to the inspiration we draw from the relational database literature.

4.3.1 Optimal Join

The join operator is one of the most expensive operators in relational databases. Its query complexity typically dominates the execution of conjunctive relational queries. Therefore, designing efficient join algorithms for different computational models and hardware settings has long been studied and remains an active area of research in relational databases (e.g. [101, 73, 18, 68]). For example, the hash join algorithm achieves linear storage and time complexity in the input table size [101, 73], which is optimal.

Optimal encrypted joins. The key for an encrypted relational database to be scalable is to achieve efficient encrypted joins. Informally, we say that an encrypted join operation is optimal if it only adds a constant overhead to the optimal plaintext join, which is asymptotically linear in input size for both time and space. Formal definition is presented in Section 4.6.

PPE-based joins. PPE-based joins such as the adjustable join in `CryptDB` [84, 85] and `Monomi` [94] can be shown to match the query and storage complexity of plaintext joins. For example, the adjustable join can be implemented using the hash join algorithm, because the equality between elements in two columns is revealed at query time, which allows the hash join to proceed. The followup work [77] strengthens the security of the adjustable join by reducing the leakage of transitive joins, however its construction is about four times more expensive in storage and query time. In general, PPE-based joins leak the equality within all columns and their correlations with the join columns.

STE-based joins. STE-based joins are a more secure alternative to the PPE-based joins. However, known constructions such as the one provided in SPX [63] and OPX [66] have higher asymptotic complexity than plaintext joins. More precisely in SPX [63] a join between two tables $\mathbf{T}$ and $\mathbf{T}'$ actually incurs worst-case quadratic complexity $\mathcal{O}(|\mathbf{T}| \cdot |\mathbf{T}'|)$ for both query and storage. By contrast, an optimal plaintext join such as the hash join [101, 73] runs in expected $\mathcal{O}(|\mathbf{T}| + |\mathbf{T}'|)$ time and has $\mathcal{O}(|\mathbf{T}| + |\mathbf{T}'|)$ size. Therefore, our goal in this work is to design the first STE-based join that matches the complexity of the plaintext join in the same notion used in the database literature

(e.g. [101, 73]).

Surrogate Join Graph

The key to our construction for an optimal STE-based join is a nonstandard representation of the join. We first examine the underlying reason for the quadratic blowup for the currently-known STE-based joins using join graphs, and then introduce our solution based on a graph transformation called the surrogate join graph.

Join graphs. To gain some intuition behind the quadratic blowup in SPX's join, we first use a graph to represent the join, called the *join graph* [71]. A node in the join graph is a row (or an identifier or pointer to a row), whereas an edge between two nodes means the two corresponding rows satisfy the join predicate and are paired up in the join result. Overall the join graph is a bipartie graph, where the nodes split naturally into two sets for the two tables. An example of a join graph is shown in Figure 4.2.

The SPX structures can be seen as storing all the edges of the join graph inside an encrypted multi-map EMM, which stores only one tuple of the form $((\mathsf{att}, att'), E)$ for the entire edge set $E = \big((u_i, v_j) : u_i \in U, v_j \in V\big)$. But in general the edge set can be much larger than each node set, such as worst-case quadratic size. Even if the EMMconstruction is optimal in space and time, this means (1) the EMMfor join would still incur quadratic storage blowup, and (2) just by reading this EMMfor the join would require quadratic amount of read operations.

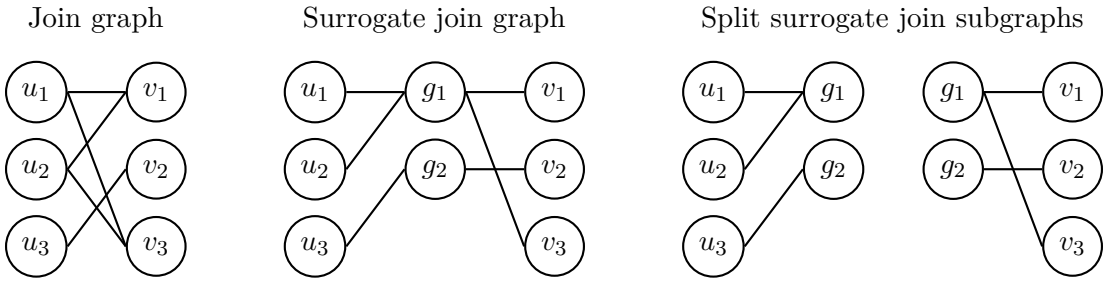


Figure 4.2: A join graph, its surrogate join graph and split surrogate join subgraphs for the join example in Figure 4.1. Each node u_i corresponds to χ_i and node v_i to χ'_i. Each surrogate node is denoted as g_i.

Surrogate join graphs. To avoid the quadratic blowup, we need to avoid directly materializing the entire edge set of the join graph in an encrypted multi-map. Instead, what we seek here is to first reduce the edge set such that its size goes from being quadratic to linear. This is possible because a join graph may have redundancy within its edges. For example, multiple nodes may be connected to the same set of nodes, which reflects the basic property of a join, where multiple rows from $\mathbf{T}$ may be joined with the same set of rows from $\mathbf{T}'$. To reduce such redundancy, we morph the join graph into a new graph called the *surrogate join graph*, by first adding a set of special nodes and then rearranging the overall connectivity, while keeping the join semantics intact.

More precisely, let the join graph be a bipartie graph $G = (U, V, E)$ where both U and V are node sets and E is an edge set. U is a set of row coordinates for $\mathbf{T}$ and V for $\mathbf{T}'$. For example, each element u_i in U is a row coordinate χ_i for the encrypted multi-map $\mathsf{EMM}^{(\mathsf{R})}$ which can be used to retrieved a unique encrypted row in $\mathbf{T}$. The edge set E consists of pairs (u, v) where $u = \chi$ for $\mathbf{T}$ and $v = \chi'$ for T'.

We then create a new graph $G_S = (U, V, S, E_S)$, by first adding a set of nodes S, called *surrogates*, such that each surrogate node corresponds to a unique subset of nodes in V that some node in U connects to in the original join graph G. Then we define the new edge set E_S by connecting the node sets U and V through the surrogates S in the following way:

(1) Let R_u be the subset of nodes in V that u in U connects to in the original join graph G;

(2) For each unique R_u over all $u \in U$, add a surrogate with a label $s = \mathsf{id}(R_u)$ that uniquely identifies R_u in V;

(3) For each (u, v) in E, we add a length-two path (u, g, v) (i.e. two edges) to the set E_S with the surrogate node $g = \mathsf{id}(R_u)$

x Figure 4.2 shows an example of a surrogate join graph, for the join example in Figure 4.1.

First, the above definition of E_S is correct in that it preserves the join semantics in E, because for each edge (u, v) in E there exists a unique length-two path in E_S with the same endpoints $(u, \cdot, v)$.

E_S also removes the redundancy in E, because for all $m = |R_u|$ edges that share the same first

node u, say $(u, v_1), \cdots, (u, v_m)$ in E, there is only one corresponding edge (u, g) in E_S as part of the path $(u, g, \cdot)$. Vice versa, for all n edges that share the same second node, say $(u_1, v), \cdots, (u_n, v)$ in E, there is also just one corresponding edge (g, v) in E_S as part of the path $(\cdot, g, v)$. Overall, E_S is linear in the node set size $\mathcal{O}(|U| + |V|)$, and the node size increase by surrogates S is also linear in $\mathcal{O}(|V|)$.

Examples. In the worst case where the join is a cartesion product, each node from U is matched with every node in V, that means there will only be one surrogate in $S = \{g\}$, and every node in U is connected with a surrogate g once, and similarly for V. So the worst-case size for E_S is still linear $\mathcal{O}(|U| + |V|)$.

Another example is when the join is one-to-one. This means that each node in U is matched with a unique node V. This will result in the largest possible surrogate set S, which still equals to $|V|$ at maximum.

An Optimal STE-based Join

Because the edge complexity of the surrogate join graph is reduced to linear in the node size, we can construct an STE-based join with linear query and storage complexity based on this edge set.

Split surrogate join subgraphs. We first construct the surrogate join graph $G_S = (U, S, V, E_S)$ for the join $\mathbf{T}_1 \bowtie \mathbf{T}_2$ where U and V are row coordinates of $\mathbf{T}_1$ and $\mathbf{T}_2$, S is the surrogates, and E_S is the edge set. Then we split G_S into two subgraphs by duplicating the surrogates S, denoted as $G_1 = (U, S, E_1)$ and $G_2 = (S, V, E_2)$, for example as shown in Figure 4.2. We then store the edge set of each subgraph as a multi-map. There are however two ways to encode the edge sets.

With interaction. One straightforward way is to let $\mathsf{MM}_1^{(\mathsf{J})}$ map a join attribute pair $(\mathsf{att}, \mathsf{att}')$ to all the edges in the left subgraph E_1, and let $\mathsf{MM}_2^{(\mathsf{J})}$ map each surrogate to a tuple of nodes that are connected to it by some edges in E_2 in the right subgraph. However this idea would introduce an additional round of $\mathcal{O}(U)$-size interaction, because the server would need to first return all the encrypted surrogates from G_1 in $\mathsf{EMM}_1^{(\mathsf{J})}$ to the client, and then wait for the client to decrypt the

surrogates, compute and send back search tokens for the $\mathsf{EMM}_2^{(\mathsf{J})}$.

Without interaction. To eliminate this extra interaction, we can precompute and store a search token gtk (called a "surrogate token") for each surrogate g for $\mathsf{EMM}_2^{(\mathsf{J})}$ in the first multi-map $\mathsf{MM}_1^{(\mathsf{J})}$ as a tuple $((\mathsf{att}, \mathsf{att}'), (u, \mathsf{gtk}))$. An example is shown in Figure 4.3. Then the join computation goes as follows:

(1) The client sends over a join token based on the column ids for attributes att and att'.

(2) Then the server first search $\mathsf{EMM}_1^{(\mathsf{J})}$ and decrypt the surrogate token gtk for each u.

(3) For each surrogate token gtk that corresponds to a u in (2), the server then queries the $\mathsf{EMM}_2^{(\mathsf{J})}$ for the set of v's, called R_u, that join with u, and outputs (u, R_u).

Notice that the algorithm has nested loops in step (2) and (3), which might suggest it is still a quadratic algorithm. However there is redundancy in the search of $\mathsf{EMM}_2^{(\mathsf{J})}$ of step (3) because the surrogate tokens for different nodes u from step (2) may be the same. So after we add standard *memoization* for the duplicate surrogate tokens gtk in between step (2) and step (3), we effectively achieve worst-case (in join size) expected (in memoization) linear in both time and space $\mathcal{O}(T)$ for table size T. This matches the plaintext hash join complexity, and improves significantly from the worst-case quadratic $\mathcal{O}(T^2)$ in SPX join.

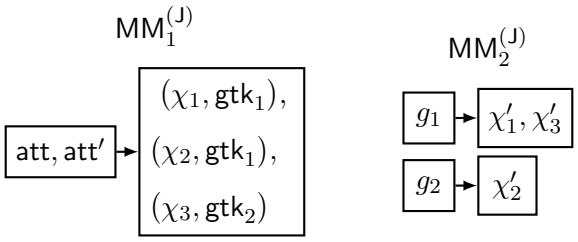


Figure 4.3: Multi-maps based on the split surrogate join subgraphs in Figure 4.2. Each node u_i corresponds to row coordinate χ_i for the table $\mathbf{T}$ and each node v_i corresponds to row coordinate χ'_i for the table $\mathbf{T}'$. Each surrogate g_i corresponds to the search token gtk_i for $\mathsf{EMM}_2^{(\mathsf{J})}$. These multi-maps replace the quadratic multi-map $\mathsf{MM}^{(\mathsf{J})}$ in Figure 4.1 to achieve optimal STE-based join.

Storage complexity. Although the optimal encrypted join between each pair of attributes only requires linear space complexity in table size as $\mathcal{O}(T)$, the total number of joins in a database may still be large. We define the join set to be the set of attributes in a database that can be joined under an equality predicate. The overall storage complexity for the join set using STE-based optimal join then becomes $\mathcal{O}(T \cdot \#\mathsf{JoinSet})$ for the whole database.

Although in the worst case the size of the join set may be quadratic in the total number of attributes, in practice the join set size tends to be a small constant. For example, the TPC-H benchmark which models a data warehouse has only 20 attributes in 8 tables. Moreover, a relational database with normalization tends to further reduce the join set size to be on the order of the number of tables, because all joins tend to be among primary or foreign keys. On the other hand, the number of rows T in a table can be very large, for example on the order of 10^6 in TPC-H. Therefore, under the assumption that the relational database is dominated by the number of rows T, we can drop the join set size as a relatively insignificant constant and view the asymptotic storage complexity to be $\mathcal{O}(T)$. This is an improvement over SPX where though under the same assumption the complexity is nonetheless quadratic $\mathcal{O}(T^2)$.

We establish formally that the STE-based join as outlined above matches the plaintext join complexity for not only one join but also conjunctive joins (i.e. multi-way joins).

Theorem 4.3.1. *If the encrypted multi-map scheme Σ_{mm} is optimal, then the STE-based join outlined above has the optimal asymptotic complexity: (1) the query time complexity is linear in input table size; and (2) the storage complexity is also linear in table size.*

The proof is provided in Sec. 4.6.

4.3.2 Query Optimization

Relational queries consist of a diverse set of operators, where the operators within a query can be reordered or rewritten into other operators to lower the overall query complexity. For a simple example, a filtered-join query can choose to apply filter first on the tables before doing the join, thereby potentially saving a large fraction of computation. This problem, called query optimization, has been extensively studied in the relational database literature [1]. Typical query optimization

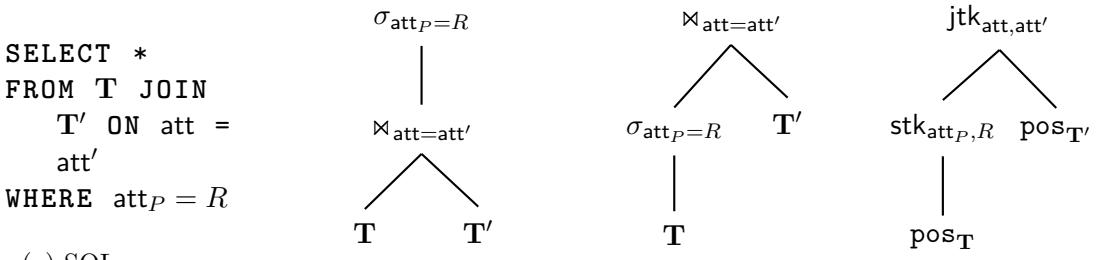
relies on relational-algebraic rules in combination with heuristics and cost estimation from the data distribution [52, 1, 15, 74]. This work focuses on an important class of relational queries, called the conjunctive relational query, which consists of joins, selections (filters) and projections, all in conjunction with each other[1].

The SPX construction [63] is not able to support query optimization because it only executes the operators in a fixed order which can lead to suboptimal performance in practice.

In order to make a more practical STE-based relational database, we study how design schemes that can execute query operators in various orders and introduce several new optimizations.

Filtered Joins

The filter join is one of the most common constituents in conjunctive queries. It is a join not only between two tables, but also in conjunction to a selection on either table, or two selections on both tables. One example is shown in Figure 4.4. For simplicity, we use a single filtered join to illustrate the details of the construction, but the technique applies to generally to arbitrary composition of multiple filtered joins in a query.



(a) SQL. (b) A query tree. (c) Selection pushdown. (d) A token tree.

Figure 4.4: An example of a filtered join query.

Query trees as input. Different from prior STE works for keyword search or boolean queries [31, 61], relational queries have more complex structures due to different types of relational operators [1]. A relational query is typically represented as a *query tree*, which encodes information of execution ordering. For example in Figure 4.4, the nodes in a query tree represent operators, such as join ($\bowtie$), select (σ), project (π) and table scan. If we view each edge as a directed edge pointing from

the child to the parent, then the execution order corresponds to a topological order.

Plaintext optimization. The filtered join is typically optimized using *selection pushdown* or *push-select-over-join* [1]. For example in Figure 4.4 the optimized query tree has both selections pushed below the join. This rule is based on the observation that if the selection is evaluated, then the filtered table(s) would make a smaller input to the join. Suppose in Figure 4.4 the fraction of VISA-paying customers and that of the vegetable suppliers are α. Then the input to the join after the selection would be $\mathcal{O}(\alpha(C+S))$ for total C customers and S suppliers. If the optimal STE-based join is used then the overall filtered join complexity is reduced by the factor of α as in $\mathcal{O}(2\alpha(C+S))$.

Suboptimal approach. The existing STE-based approach such as SPX cannot support selection pushdown, as it executes the join and the filter independently and intersects the results. Therefore regardless of the ordering of the selections and join, the same query above for example would incur the full join complexity J plus the selection complexity as $\mathcal{O}(J+\alpha(C+S))$.

Even if we modified SPX with an optimal join (such as Sec 4.3.1) with complexity $J=\mathcal{O}(C+S)$, here the total complexity would still be suboptimal as $\mathcal{O}((1+\alpha)(C+S))$, which would always be larger than the plaintext complexity as $\mathcal{O}(2\alpha(C+S))$ unless trivially the select operator selected the entire table such that $\alpha=1$.

Selection pushdown on STE-based join. To support selection pushdown over the optimal STE-based join, the server needs to avoid computing the entire surrogate join graph when only a subset of the row coordinate nodes are selected.

But such constrained computation of the join is impossible with the definition of $\mathsf{EMM}_1^{(\mathsf{J})}$ (Sec. 4.3.1), because its labels are based on the join attributes only. Therefore we need to change its definition to incorporate the selected row coordinates.

Concretely, we define a new $\mathsf{EMM}^{(\mathsf{XJ})}$ for the join between att and att' to play the role of $\mathsf{EMM}_1^{(\mathsf{J})}$ (Sec. 4.3.1). We show an example in Figure 4.5. This encrypted multi-map $\mathsf{EMM}^{(\mathsf{XJ})}$ encrypts a set of label-value pairs, each of the form (χ, gtk) for a row coordinate χ and a surrogate token

gtk. To compute a filtered join, the client sends the join token $\mathsf{jtk} = K^{(\mathsf{XJ})}$ which is the key for $\mathsf{EMM}^{(\mathsf{XJ})}$, and the select token stk for $\mathsf{EMM}^{(\mathsf{S})}$. First the server queries $\mathsf{EMM}^{(\mathsf{S})}$ by stk to obtain the filtered row coordinates. Then for each such selected row coordinate χ, the server computes the token $\mathsf{tk} = \Sigma_{\mathsf{MM}}.\mathsf{Token}(\mathsf{jtk}, \chi)$ for $\mathsf{EMM}^{(\mathsf{XJ})}$, and queries it for only the corresponding surrogate tokens under the selection. Then the rest of the STE-based optimal join algorithm will be carried out only on this set of surrogate tokens on $\mathsf{EMM}_2^{(\mathsf{J})}$ (Sec. 4.3.1). Note that here the server computes the token algorithm, which is different from the typical STE setting.

For example in Figure 4.5, if push-down selection results in χ_3 only, then the server only uncovers one surrogate token gtk_2 from $\mathsf{EMM}^{(\mathsf{XJ})}$, and finally it computes only χ_2' that joins with χ_3 from $\mathsf{EMM}_2^{(\mathsf{J})}$. The unselected part of the join for χ_1, χ_2 and χ_1', χ_3' is not involved.

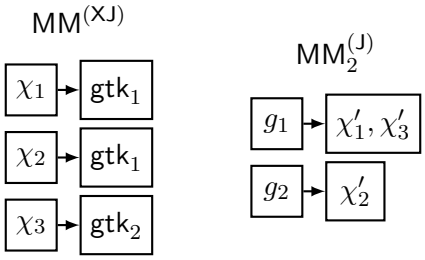


Figure 4.5: The multi-map $\mathsf{MM}^{(\mathsf{XJ})}$ for selection pushdown on the STE-based filtered join, based on the example in Figure 4.3 with the difference being that $\mathsf{MM}^{(\mathsf{XJ})}$ replaces $\mathsf{MM}_1^{(\mathsf{J})}$.

Leakage. The selection pushdown is susceptible to an active attack where the persistent attacker uses the revealed row coordinates from other queries to search the encrypted data structure $\mathsf{EMM}^{(\mathsf{XJ})}$ for part of the join that was not selected. We addressed the reduction of such leakage in Section 4.3.3.

Token trees. To represent the execution order of the tokens associated with a query, we introduce the token tree, an analog to the query tree. The client converts a query tree into a token tree that may consist of join nodes of *join tokens* (jtk), select nodes of *select tokens* (stk), and relative positions of tables ($\mathtt{pos}$) present in the query tree.

A node's type and subtree give information about the execution strategy. For example in Figure 4.4, the leaf select node $\mathsf{stk}_{\mathsf{att},R}$ with the child table position $\mathtt{pos}_{\mathbf{T}}$ tells where the result of

the search for $\mathsf{stk}_{\mathsf{att},R}$ in $\mathsf{EMM}^{(\mathsf{S})}$ should be placed in the result table.

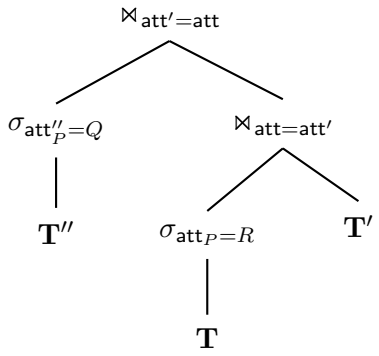
An edge indicates the output of the child is used as input to the parent. For example in Figure 4.4, the result of the leaf select node (i.e. search result using select token stk), which is a set of row coordinates for table $\mathbf{T}$ corresponding to the selection $\mathsf{att} = R$, will become input to the parent join node $\mathsf{jtk}_{\mathsf{att},\mathsf{att}'}$.

Extension to conjunctive relational queries. When the relational query becomes more complicated such as involving multiple filtered joins, another query optimization rule is typically applied to reorder them, called the *join reordering* [1]. The reason why such optimization makes sense is because a filtered join tends to have smaller result size than the full join, so for multiple filtered joins, it is often beneficial to "chain" them together in an order that reduces intermediate data size. A common heuristic is to order the operators by the estimated selectivity [1].

To support join reordering, we reuse the encrypted multi-maps for filtered joins (Sec. 4.3.2), with the generalization that we allow the row identifiers to come from arbitrary token subtrees. Figure 4.6 shows the query of two joins and two filters, its optimized query tree and the corresponding token tree. The select nodes and join nodes not only follow the same optimized order as the plaintext selections and joins, but the overall query complexity is also reduced because the token tree nodes only compute based on the output from the subtree.

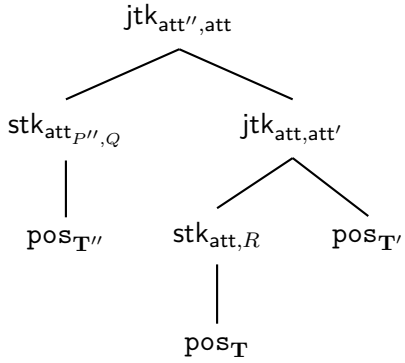
```
SELECT *
FROM T
  JOIN T′ ON
     att = att′
  JOIN T″ ON
     att = att″
WHERE att_P = R
  AND att″_P = Q
```

(a) The SQL query.



(b)

(c) An optimized query tree.



(d) The token tree.

Figure 4.6: An example of conjunctive relational query.

Conjunctive Selections

Within a conjunctive relational query, selections on the same table are all conjunctive, and the semantics is to find all rows that satisfy all of the selection predicates simultaneously. In this work we focus on one form of selection predicate which is an attribute equal to a constant value, $\mathsf{att} = v$. Overall a conjunctive selection is $\mathbf{T}.\mathsf{att}_1 = v_1 \wedge \cdots \wedge \mathbf{T}.\mathsf{att}_m = v_m$ over the same table $\mathbf{T}$.

Baseline solution. One straightforward approach is to use an encrypted multi-map to query for each selection predicate, and then take the intersection over all resulting row sets. SPX [63] for example can be seen as taking this approach for conjunctive selection.

Baseline leakage. The baseline approach above is unable to support the reordering of selection predicates, because different ordering would still result in the same execution time. Because the execution essentially queries for each predicate independently, the total complexity is $\mathcal{O}(\sum_{i=1}^{m} \mathsf{DB}(\mathbf{T}.\mathsf{att}_i = v_i))$.

On the other hand, this approach also incurs additional leakage for the conjunction, which includes the entire set of row ids for each predicate. Next, we first discuss how to improve the efficiency through enabling selection reordering, then in the next subsection we will turn our focus to leakage reduction.

Selection reordering. In applications, selection predicates typically have varying selectivity i.e. the ratio of result size versus total size. An often more efficient execution strategy is to order the selection predicates $P_1, \cdots, P_m$ based on increasing selectivity $P_{(1)}, \cdots, P_{(m)}$ such that

(1) The predicate with lowest selectivity (i.e., with smallest result set) $P_{(1)}$ is computed first;

(2) To compute $P_{(1)} \wedge \cdots \wedge P_{(j+1)}$, for each row coordinate χ in the result set of $P_{(1)} \wedge P_{(j)}$, check if it satisfies the next predicate $P_{(j+1)}$, and repeat for all $j < m$ for m selection predicates.

This above algorithm essentially ties the computation of the subsequent selection predicates to the first one, which is chosen typically by the query optimizer for its lowest selectivity. Each subsequent predicate usually further reduces this result set by retaining only the intersection. In the worst

case no such reduction occurs for any of the subsequent selection predicates, so each predicate will take $\mathcal{O}(|P_{(1)}|)$ steps to check, and overall $\mathcal{O}(|P_{(1)}| \cdot m)$ for predicate result size $|P|$ and m selection predicates. Therefore as long as the first predicate is chosen for lower selectivity, then the algorithm is almost always better the baseline approach (only equal when chosen as largest selectivity or all selection predicates share the same selectivity).

To support selection reordering in STE-based conjunctive selections, we use a technique from Cash et al. [31] to construct an additional SET which supports a membership check that whether a row referenced by the row coordinate χ contains a selected value $v_{(j)}$ in predicate $P_{(j)}$ without revealing to the server $P_{(j)}$. This functionality can be useful when the row coordinates are the result set of the part of the conjunction, because then the server can evaluate the next term $\mathsf{att}_{(j)} = v_{(j)}$ in the conjunction by testing SET.

More abstractly this functionality can be seen as evaluation of a function $f(K_j, \chi)$ with inputs $K_j = F(K, \mathsf{id}(T) \parallel \mathsf{att}_{(j)} \parallel v_{(j)})$ and row coordinate χ. Here we define the function f as a PRF or a random oracle $H(K_j \parallel \chi)$, and during setup the client precomputes f over all inputs and save the results in the SET to store on the server.

For querying with conjunction $P_1, \cdots, P_m$, the server takes $(\mathsf{stk}_1, K_2, \cdots, K_m)$ as an input, which corresponds to reordered selection predicates $P_{(1)}, P_{(2)}, \cdots, P_{(m)}$ such as based on expected selectivity. The server first queries $\mathsf{EMM}^{(\mathsf{S})}$ using stk_1 for $P_{(1)}$. Then for each χ decrypted in the result, the server checks the SET for the existence of value $f(K_2, \mathsf{id}(T) \parallel \mathsf{att}_2 \parallel v_2)$ for $P_{(2)}$, and so on until $P_{(m)}$, while each step retains only χ for which the check on SET succeeds.

Selection reordering leakage. The extension outlined above allows the scheme to support selection reordering, but the attacker can still infer information about each predicate in the conjunction. For example, the attacker can store the input K_j to the function f for the predicate P_j, and evaluate f on different row coordinates that are revealed at a different time by other queries. In the worst case where the attacker collects all such row coordinates for table $\mathbf{T}$, the attacker can infer the complete result set for each predicate P_j, in which case the overall leakage becomes equal to the baseline approach.

4.3.3 Leakage Reduction

Compared to SPX [63], the techniques for filtered join (Sec. 4.3.2) and selection reordering (Sec. 4.3.2) improve efficiency significantly, but can still incur the same leakage. For example, a filtered join query by itself only leaks on the filtered portion of the join, which is an improvement over SPX (which leaks in the full join). But in the worst case when all the row identifiers for the filtered column are known, the leakage is the same as SPX. Because filtered joins are one of the most common building blocks for relational queries, we focus on leakage reduction for filtered joins in this work, and leave the extension to selection reordering in the followup work.

Filtered Join Leakage Reduction

When viewed abstractly, the STE-based filtered join (Sec. 4.3.2) essentially introduces an outsourced token computation $g(J, \chi)$ for $\mathsf{EMM}^{(\mathsf{XJ})}$, where $g(J, \chi) = \Sigma_{\mathsf{MM}}.Token(J, \chi)$ with key $J = K^{(\mathsf{XJ})}$, where the client provides the key J and the server iterates over each row coordinate χ from a pushdown selection to compute g. However, the client input J is too "generous" in the sense that it allows the server to compute the function g also on other unselected row coordinates that are revealed by other queries. Therefore the key idea to reduce this leakage is to constrain the domain of row coordinates in the function g by only those that are selected by pushdown selection P.

With interaction. The simplest approach is to let only the client compute g. This however introduces additional interaction, where for each pushdown selection P the server needs to send back all the selected row coordinates. The client then computes a token $g(J, \chi) = \Sigma_{\mathsf{MM}}.\mathsf{Token}(J, \chi)$ for $\mathsf{EMM}^{(\mathsf{XJ})}$ for each row coordinate χ received, and sends all such tokens to the server. The server then queries $\mathsf{EMM}^{(\mathsf{XJ})}$ using these tokens to uncover the surrogate tokens for the rest of the join. Overall this approach can incur $\mathcal{O}(T)$ bandwidth for each pushdown selection.

Constrained computation To eliminate the interaction, we constrain the computation of g by the pushdown selection predicate P, denoted as $g(C_P; J, \chi)$ for the constraint C_P. The server

should only be able to compute the constrained g for the key J on the subset of row coordinates that are selected under P.

One way to realize such constrained function g is to build additional encrypted data structure to limit the server's access to the tokens to $\mathsf{EMM}^{(\mathsf{XJ})}$. We redefine the encrypted multi-map $\mathsf{EMM}^{(\mathsf{XJ})}$ to be over the plaintext set of tuples, each of the form $((C_P, \chi), \mathsf{gtk})$ for a value C_P, a row coordinate χ and a surrogate token gtk. The function g corresponds to the outsourced computation of a token for the tuple $((C_P, \chi), \mathsf{gtk})$ in $\mathsf{EMM}^{(\mathsf{XJ})}$ given the client input C_P and key $K^{(\mathsf{XJ})}$. Here for leakage reduction C_P needs to not only depend on the join attributes but also the selection P, for example C_P being the result of a pseudorandom function $F(K_C, \mathsf{att}_P \parallel x \parallel \mathsf{att} \parallel \mathsf{att}')$ for a secret key K_C and selection on $\mathsf{att}_P = x$. Now when given such C_P as client input, the server's token computation of g is limited to only for each row coordinate χ under the pushdown selection P, namely $\Sigma_{\mathsf{MM}}.\mathsf{Token}(K^{(\mathsf{XJ})}, (C_p, \chi))$. This means only the surrogate tokens associated with the rows under the pushdown selection P may be uncovered from $\mathsf{EMM}^{(\mathsf{XJ})}$ for the rest of the join.

An example. Figure 4.7 shows such an example for the filtered join $\sigma_{\mathsf{att}_P=R}\mathbf{T} \bowtie_{\mathsf{att}=\mathsf{att}'} \mathbf{T}'$. The client sends the tokens: $\mathsf{stk}_R = \Sigma_{\mathsf{MM}}.\mathsf{Token}(K^{(\mathsf{S})}, \mathsf{att}_P \parallel R)$ for selection, and $K^{(\mathsf{XJ})}$ and $C_R = F(K_C, \mathsf{att}_P \parallel R \parallel \mathsf{att} \parallel \mathsf{att}')$. The server first uses stk_R to query $\mathsf{EMM}^{(\mathsf{S})}$ for the only row coordinate χ_2, then it computes a token $\mathsf{tk} = \Sigma_{\mathsf{MM}}.\mathsf{Token}(K^{(\mathsf{XJ})}, (C_R, \chi_2))$ and queries $\mathsf{EMM}^{(\mathsf{XJ})}$ for surrogate token gtk_1, then it uses gtk_1 to query $\mathsf{EMM}_2^{(\mathsf{J})}$ for the row coordinates (χ_1', χ_2') to join with χ_1. The server finally uses the row coordinates to query the $\mathsf{EMM}^{(\mathsf{R})}$ for the encrypted rows and return the filtered join result back to the client.

Query complexity. This leakage reduction approach does not add to the query complexity for filtered join (Sec. 4.3.2), because only the filtered portion of the join will be queried.

Storage requirement. This improvement in leakage approach results in an increase of storage, but overall the overhead is still constant in table size under common assumptions. The increase comes from the fact that a join can correlate with different filtered attributes in a table $\mathbf{T}$, where the total number of such correlation goes up to table width L. Since each tuple encrypted in

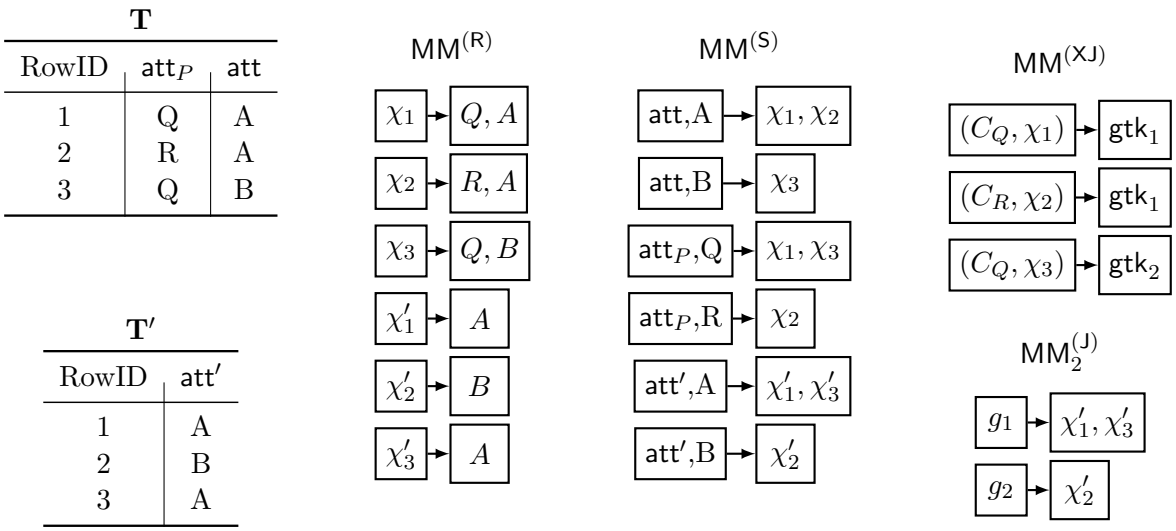


Figure 4.7: The modified multi-map $\mathsf{MM}^{(\mathsf{XJ})}$ for leakage reduction on the STE-based filtered join of $\sigma_{\mathsf{att}_P=x}\mathbf{T} \bowtie_{\mathsf{att}=\mathsf{att}'} \mathbf{T}'$, based on the example in Figure 4.1 and 4.3.

$\mathsf{EMM}^{(\mathsf{XJ})}$ for this join is specific to a filtered attribute and a row in the customer table, there will be $\mathcal{O}(L \cdot T)$ tuples for each row in table $\mathbf{T}$ for this join between att and att'. Additionally there will be $\mathcal{O}(T)$ tuples for $\mathsf{EMM}_2^{(\mathsf{J})}$ in table $\mathbf{T}$ for the same join but in reverse order. So overall the complexity becomes $\mathcal{O}((L \cdot T + T) \cdot \#\mathsf{JoinSet})$.

However, as argued in Section 4.3.1, the join set size is a small constant dominated by the table length T. Similarly, the table width L is also a small constant dominated by T. Therefore asymptotically the storage complexity is still $\mathcal{O}(T)$.

Join filtered by conjunctive selection. In a more general setting a join can be filtered by a conjunction of selections on the same table. To minimize the query complexity it is beneficial to push the conjunctive selections down the tree so that they are evaluated ahead of the join, which reduces the input size to the join. The leakage reduction approach here ties the leakage of the join to the first selection in the conjunction, which is typically chosen by the query optimizer to be the selection with the smallest output.

Join without any filters. A join without any filter on either table can also be handled by having a constraint value C that correspond to a selection predicate that always return true. We defer the

full details to the next section.

Join filtered on both sides. In general a filtered join may have selection on both sides of input, such as $(\sigma_P \mathbf{T}) \bowtie (\sigma_{P'} \mathbf{T}')$. The extension is to exploit the symmetry by treating this both-sided filtered join as two one-sided filtered joins. With this symmetry for two-sided filtered joins also extended to unfiltered joins, we no longer need $\mathsf{MM}^{(\mathsf{J})}$ (such as in Figure 4.7), and can just use $\mathsf{MM}^{(\mathsf{XJ})}$ for joins in general. We defer the full details to the next section.

4.4 The PKFK Scheme

In this section, we describe our new PKFK construction. The scheme's performance is comparable to PPE-based schemes like CryptDB [94] but provides stronger security guarantees. It also provides efficiency and leakage improvements over the SPX construction of [63] and the OPX scheme of [11]—both of which are STE-based. We first provide a high level intuition before diving into the scheme's details.

4.4.1 Overview

Roughly speaking, the scheme represents the database using various multi-map structures that are then encrypted using a multi-map encryption scheme. Compared to SPX and OPX, PKFK uses a new compact and highly efficient design that: (1) lowers the query and space complexity from quadratic to linear in table size; (2) does not require the pre-computation of the joins; and (3) reduces the leakage of conjunctive queries. Moreover, unlike SPX which only supports queries in their normal form (i.e., with no support for query optimization), PKFK supports not only standard query optimizationbut also custom optimization rules we introduce. We found that PKFK has only 4.2× median query overhead and 3.6× storage overhead over a plaintext query, representing about two orders of magnitude improvement over SPX and OPX and similar performance to CryptDB. The full details of evaluation is presented in Chapter 6.

In the following, we divide our overview in two: (1) a setup phase during which the client generates the encrypted database; and (2) a query phase during which the client generates an

encrypted query for execution.

4.4.2 Setup

Given a plaintext database DB, the setup phase creates five collections of structures that capture different representations of the database:

- *(row representation)*: $\mathsf{MM}^{(\mathsf{R})}$ is a multi-map that maps each row identifier to a tuple composed of the content of the row;
- *(filter representation)*: $\mathsf{MM}^{(\mathsf{S})}$ is a multi-map that maps the unique values in every column that supports selection;
- *(partial filter representation)*: SET is a set structure that checks whether a column in a specific row (i.e. a table cell) contains a particular value.
- *(join representation)*: $\mathsf{MM}^{(\mathsf{XJ})}$ is a multi-map that maps a row under a constraint to a surrogate in the surrogate join graph (Sec. 4.3.1).

Every multi-map is encrypted using a multi-map encryption scheme, whereas the set structure is encrypted using a custom encrypted set scheme that we detail in the next section. As mentioned above, a key feature of PKFK is its ability to support optimized conjunctive queries which it answers by making use of a combination of the encrypted structures. In order to do so, PKFK uses a structured encryption design technique called *structural chaining*. In a nutshell it works as follows: the client generates an encrypted query that it sends to the server. Depending on the type of the query, the server executes a query algorithm using a specific encrypted structure. The result of this execution is a set of intermediary results that the server uses to query another structure and so on. As a concrete example, in PKFK, the client sends a SQL query to retrieve all rows in table $\mathbf{T}$ whose attribute equal to a particular value. The server then takes the encrypted form of this query, first searches the encrypted filter representation $\mathsf{EMM}^{(\mathsf{S})}$ which outputs the intermediary results that it then uses to search the row representation $\mathsf{EMM}^{(\mathsf{R})}$ to uncover the encrypted query result.

Linear setup time. Another improvement made in PKFK is that it does not require precomputation of all joins. Such requirement is inherent in SPX [64] and OPX [66] (Ch. 3), so they take prohibitive time to set up (i.e. quadratic in database size). By getting rid of this requirement, PKFK only takes linear time to set up.

Row representation. The row representation in $\mathsf{MM}^{(\mathsf{R})}$ supports the lookup of an encrypted row given its coordinate. Each row $\mathbf{r}$ in the plaintext database is associated with a pointer or coordinate $\chi(\mathbf{r}) = (\mathsf{tbl}(\mathbf{r}), \mathsf{rrk}(\mathbf{r}))$, which is defined to be the unique pair of table identifier and the row rank. Each plaintext row is encrypted element-wise by a symmetric key encryption scheme *ske*.

Filter representation. The filter representation in $\mathsf{MM}^{(\mathsf{S})}$ supports the lookup of row coordinates given a selection predicate $\mathsf{att} = a$, which is encoded as a pair $(\chi(\mathsf{att}), a)$. The resulting row coordinates point to the rows that satisfy the selection predicate, and can be used to look up the encrypted rows in $\mathsf{MM}^{(\mathsf{R})}$. One detail to note for encryption is that to make an non-interactive scheme, each row coordinate resulting from the lookup is in the form of a row token rtk for the encrypted row representation $\mathsf{EMM}^{(\mathsf{R})}$.

Partial filter representation. The partial filter representation in SET supports conjunctive selection, such as the selection P_i in conjunction $P_1 \wedge P_2 \wedge \cdots \wedge P_m$ for any $i > 1$. The way to use SET is to first search $\mathsf{MM}^{(\mathsf{S})}$ to obtain the row coordinates, then check if each resulting row coordinate $\mathbf{r}$ satisfy the predicate $P_i \equiv \mathsf{att}_i = a_i$. To support this check, the SET stores the ciphertext for each row coordinate $\mathbf{r}$ and the filter attribute value at $\mathbf{r}$: $\mathbf{r}[\mathsf{att}_i]$.

Join representation. The join representation in $\mathsf{MM}^{(\mathsf{XJ})}$ is not as straightforward as other representations, because it is not a direct representation of the join, but rather builds on top of the *surrogate join graph* (Sec. 4.3.1). Another complication comes from the need to support several types of operations: (1) unfiltered joins, (2) one-sided filtered joins and (3) both-sided filtered joins. The unfiltered join needs to look up all pairs of row coordinates and their surrogates under both join attributes. This intuitively supports the join computation by matching the rows between

two attributes by the same surrogate. To add support for filtered joins, we need to condition the resulting pairs additionally on the selection. This is achieved by adding the encoding for selection (att_p, a) and the join attributes $(\mathsf{att}, \mathsf{att}')$ into the labels of $\mathsf{MM}^{(\mathsf{XJ})}$.

4.4.3 Query

The query phase is divided into two parts. The client first gives the plaintext query to the Token algorithm to obtain a collection of tokens. These tokens are structured as a *token tree* (Sec. 4.3.2) to reflect the operator ordering in the plaintext query tree, such as selections first and then followed by joins and so on. Such ordering is typically decided by a query optimizer based on algebraic rules and heuristics (more details in [1]). The capability of work with query trees as output from query optimizers was first proposed in OPX [66] (Ch. 3).

Token generation. The Token algorithm takes as input an a-priori optimized SQL query tree whose nodes are relational operators. It then replaces every node in the query tree with a corresponding token. Each token is generated in such a way that it is going to be used against a specific encrypted structure that represents a relational operator (Sec.4.4.2). One detail to note here is in order for the Query algorithm to assemble the query results in a table, the token tree needs to store information about where the search result of each token belong in the query result table. This positional information is stored in the `pos` constant along with each token in the token tree.

Query execution. The token tree is then sent to the server to execute as input to the Query algorithm. The Query algorithm traverses the token tree, and for each token it searches corresponding the encrypted structure, and puts the search results in a recursively built result table at the correct position. Often the subsequent tokens require input from the results of the previous tokens, and such dependency is captured by the token tree structure and followed in the Query algorithm. The final result is a table of encrypted rows that correspond to the plaintext query result.

4.4.4 Pseudo-Code of PKFK

PKFK uses as building blocks a response-revealing multi-map encryption scheme Σ_{MM}, a symmetric encryption scheme SKE, a random oracle H, and a pseudo-random function F. We listed the pseudo-code of PKFK in three parts:

1. Setup: Fig. 4.8 and 4.9;
2. Token: Fig. 4.10 and 4.11;
3. Query: Fig. 4.12 and 4.13.

4.5 Security and Leakage

Similar to SPX [64] and OPX (Ch. 3), PKFK is a black-box construction and therefore its concrete leakage profile depends on which underlying data structure encryption scheme it uses. We already analyzed the concrete leakage profile of OPX in Section 3.3 based on additional assumption that $\Sigma^{\pi}_{\mathsf{MM}}$ from [30] is replaced with an almost leakage free multi-map encryption scheme. For PKFK, we follow the same assumption to analyze its concrete leakage profile. In particular, we describe the leakage profile of PKFK when instantiated with standard multi-map encryption schemes [42, 36, 67, 31, 30]. 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), also known as the access pattern and the search pattern in the SSE literature, respectively. The query equality reveals if and when a query is repeated while the response identity reveals the identities of the responses. In the following we will give a high level intuition on what each pattern will disclose.

Improvement over SPX and OPX. The main improvement of PKFK is for joins. In particular for a filtered join query, SPX leaks the join pattern for the full tables with respect to the join attributes, regardless of the filter; OPX leaks the join pattern for any row seen in this filter and in all other queries on the same table. By contrast, PKFK only leaks the join patterns for the filtered rows in this query alone.

Let $\Sigma_{\mathsf{MM}} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Get})$ be a response-revealing multi-map encryption scheme, $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme, $F : \{0,1\}^k \times \{0,1\}^\star \to \{0,1\}^m$ be a pseudo-random function, and $H : \{0,1\}^\star \to \{0,1\}^m$ be a random oracle. Consider the DB encryption scheme $\mathsf{PKFK} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query}, \mathsf{Dec})$ defined as follows [a]:

- $\mathsf{Setup}(1^k, \mathsf{DB})$:
  1. initialize a SET
  2. initialize a collection of multi-maps
  $$\mathsf{MM}^{(\mathsf{R})}, \mathsf{MM}^{(\mathsf{S})}, \mathsf{MM}^{(\mathsf{XJ})}$$
  3. sample keys $K_E, K_F, K_S, K_C, K_G \xleftarrow{\$} \{0,1\}^k$;
  4. **(Row rep.)**
     a. for each table $\mathbf{T} \in \mathsf{DB}$, for each row $\mathbf{r} = (r_1, \cdots, r_{\#\mathbf{r}}) \in \mathbf{T}$,
     $$\mathsf{MM}^{(\mathsf{R})}\big[\chi(\mathbf{r})\big] := \big(\mathsf{SKE.Enc}_{K_E}(r_1) \| \cdots \| \mathsf{SKE.Enc}_{K_E}(r_{\#\mathbf{r}})\big)$$
     b. compute
     $$\big(K^{(\mathsf{R})}, \mathsf{EMM}^{(\mathsf{R})}\big) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}^{(\mathsf{R})}\big)$$
  5. **(Filter rep.)** for each table $\mathbf{T} \in \mathsf{DB}$,for each column $\mathbf{c} \in \mathsf{FilterSet}(\mathbf{T})$,
     a. for each unique value $v \in \mathbf{c}$,
        i. for each row $\mathbf{r} \in \mathbf{T}_{\mathbf{c}=v}$, compute the row token
        $$\mathsf{rtk}_{\mathbf{r}} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\big(K^{(\mathsf{R})}, \chi(\mathbf{r})\big)$$
        ii. set
        $$\mathsf{MM}^{(\mathsf{S})}\big[\chi(\mathbf{c}) \| v\big] := \big(\mathsf{rtk}_{\mathbf{r}}\big)_{\mathbf{r} \in \mathbf{T}_{\mathbf{c}=v}}$$
     b. compute
     $$\big(K^{(\mathsf{S})}, \mathsf{EMM}^{(\mathsf{S})}\big) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}^{(\mathsf{S})}\big)$$
  6. **(Partial filter rep.)** for each table $\mathbf{T} \in \mathsf{DB}$, for each column $\mathbf{c} \in \mathbf{T}$,
     a. for each unique value $v \in \mathbf{c}$,
        i. compute $K_v \leftarrow F(K_S, \chi(\mathbf{c}) \| v)$
        ii. set for each $\mathbf{r} \in \mathbf{T}_{\mathbf{c}=v}$,
        $$\mathsf{SET} := \mathsf{SET} \bigcup \big\{H(K_v \| \mathsf{rtk})\big\}$$
        where $\mathsf{rtk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K^{(\mathsf{R})}, \chi(\mathbf{r}))$

(Continued in Fig. 4.9)

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

Figure 4.8: The PKFK scheme.

(Continued from Fig. 4.8)

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

  6. **(Join rep.)**

     a. for each table $\mathbf{T} \in \mathsf{DB}$, for each column pairs $(\mathbf{c}, \mathbf{c}') \in \mathsf{JoinSet}(\mathsf{DB})$ where $\mathsf{tbl}(\mathbf{c}) = \mathbf{T}$,

        i. for each row $\mathbf{r}$ in table $\mathbf{T}$,

           1. compute the row token

$$\mathsf{rtk}_{\mathbf{r}} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\big(K^{(\mathsf{R})}, \chi(\mathbf{r})\big)$$

           2. compute the surrogate for cell $v = \mathbf{r}[\mathbf{c}]$

$$g_v \leftarrow F\big(K_G, \chi(\mathbf{c}) \| \chi(\mathbf{c}') \| v\big)$$

           3. **(filtered join)** for each correlated filter column $\bar{\mathbf{c}} \in \mathsf{FilterSet}(\mathbf{T})$ where $\mathbf{T} = \mathsf{tbl}(\mathbf{c})$,

              a. compute the constraint for the filter value $w = \mathbf{r}[\bar{\mathbf{c}}]$

$$C_w \leftarrow F\big(K_C, \chi(\mathbf{c}) \| \chi(\mathbf{c}') \| \chi(\bar{\mathbf{c}}) \| w\big)$$

              b. set

$$\mathsf{MM}^{(\mathsf{XJ})}\big[(C_w, \mathsf{rtk}_{\mathbf{r}})\big] := \big((\mathsf{rtk}_{\mathbf{r}}, g_v)\big)$$

        ii. **(unfiltered join)** compute the null constraint for the unfiltered join

$$C_\perp = F\big(K_C, \chi(\mathbf{c}) \| \chi(\mathbf{c}') \| \chi(\mathbf{c}) \| \perp\big)$$

        iii. set

$$\mathsf{MM}^{(\mathsf{XJ})}\big[C_\perp\big] := \big(\mathsf{rtk}_{\mathbf{r}}, g_v\big)_{\mathbf{r} \in \mathbf{T}, v = \mathbf{r}[\mathbf{c}]}$$

     b. compute

$$\big(K^{(\mathsf{XJ})}, \mathsf{EMM}^{(\mathsf{XJ})}\big) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}^{(\mathsf{XJ})}\big)$$

  7. output

$$K = \big(K_E, K_F, K_S, K_C, K_G, K^{(\mathsf{R})}, K^{(\mathsf{S})}, K^{(\mathsf{XJ})}\big)$$

     and

$$\mathsf{EDB} = \big(\mathsf{SET}, \mathsf{EMM}^{(\mathsf{R})}, \mathsf{EMM}^{(\mathsf{S})}, \mathsf{EMM}^{(\mathsf{XJ})}\big)$$

Figure 4.9: The PKFK scheme.

- $\mathsf{Token}(K, \mathsf{QT})$:
  1. parse K as $\big(K_E, K_F, K_S, K_C, K_G, K^{(\mathsf{R})}, K^{(\mathsf{S})}, K^{(\mathsf{XJ})}\big)$;
  2. initialize a token tree TT as a tree with the same structure as the query tree QT;
  3. initialize a multi-map P for table positions in the query tree QT;
  4. Recursively visit each query tree node $N \in \mathsf{QT}$ in post-order traversal, where its subtree(s) are $\mathsf{QT}_{\mathbf{in}}$ (and $\mathsf{QT}'_{\mathbf{in}}$),
     a. set the table positions for the query subtree rooted at node N

$$P\big[N\big] := \begin{cases} (\mathbf{T}) & \text{if } N \equiv \mathbf{T} \\ \mathrm{P}\big[\mathrm{Q}_{\mathbf{in}}\big] \bigcup P\big[\mathrm{Q}'_{\mathbf{in}}\big] & \text{else} \end{cases}$$

     b. **(leaf filter)** if $N \equiv \sigma_{\mathsf{att}=v}\big(\mathbf{T}\big)$,
        i. compute the select token

$$\mathsf{stk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\big(K^{(\mathsf{S})}, \chi(\mathsf{att}) \| v\big)$$

        ii. set the token tree node with table position $\mathtt{pos} = index(\mathbf{T}, P[N])$

$$N_{\mathsf{TT}} \leftarrow \big(\mathsf{stk}, \mathtt{pos}\big)$$

     c. **(internal filter)** else if $N \equiv \sigma_{\mathsf{att}=v}\big(\mathsf{QT}_{\mathbf{in}}\big)$,
        i. compute the partial filter key

$$K_v \leftarrow F\big(K_S, \chi(\mathsf{att}) \| v\big)$$

        ii. set the token tree node with table position $\mathtt{pos} = index(\mathbf{T}, P[N])$

$$N_{\mathsf{TT}} \leftarrow \big(K_v, \mathtt{pos}\big)$$

     d. **(project)** else if $N \equiv \pi_{\mathsf{att}_1, \cdots, \mathsf{att}_z}\big(\mathsf{QT}_{\mathbf{in}}\big)$
        i. let $\mathbf{T}_i = \mathsf{tbl}(\mathsf{att}_i)$; compute the table position for $i = 1, \cdots, z$

$$\mathtt{pos}_i := index(\mathbf{T}_i, P[N])$$

        and the attribute index

$$c_i := index(\mathsf{att}_i, \mathbb{S}(\mathbf{T}_i))$$

        ii. set the token tree node

$$N_{\mathsf{TT}} \leftarrow \big((\mathtt{pos}_1, c_1), \cdots, (\mathtt{pos}_z, c_z)\big)$$

     e. **(constant)** else if $N \equiv [a]$, set $N_{\mathsf{TT}} = \big[\mathsf{SKE.Enc}(K_E, a)\big]$
     f. **(cross product)** else if $N \equiv \mathsf{QT}_{\mathbf{in}} \times \mathsf{QT}'_{\mathbf{in}}$, set $N_{\mathsf{TT}} := \mathsf{TT}_{\mathbf{in}} \times \mathsf{TT}'_{\mathbf{in}}$;

(Continued in Fig. 4.11)

Figure 4.10: The PKFK scheme.

(Continued from Fig. 4.10)

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

  4. *g.* **(join)** else if $N \equiv \mathsf{QT}_{\mathbf{in}} \bowtie_{\mathsf{att}=\mathsf{att}'} \mathsf{QT}'_{\mathbf{in}}$,
     - *i.* **(left side, filtered)** if left subtree $\mathsf{QT}_{\mathbf{in}}$ has selection $\mathsf{att}_P = v$ on table $\mathsf{tbl}(\mathsf{att})$, then compute the left select constraint
     $$C \leftarrow F\big(K_C, \chi(\mathsf{att}) \| \chi(\mathsf{att}') \| \chi(\mathsf{att}_P) \| v\big)$$
     - *ii.* **(left side, unfiltered)** else compute the left null constraint
     $$C \leftarrow F\big(K_C, \chi(\mathsf{att}) \| \chi(\mathsf{att}') \| \chi(\mathsf{att}) \| \perp\big)$$
     - *iii.* **(right side, filtered)** if right subtree $\mathsf{QT}'_{\mathbf{in}}$ has selection $\mathsf{att}_{P'} = v'$ on table $\mathsf{tbl}(\mathsf{att}')$, then compute the right select constraint
     $$C' \leftarrow F\big(K_C, \chi(\mathsf{att}) \| \chi(\mathsf{att}') \| \chi(\mathsf{att}_{P'}) \| v'\big)$$
     - *iv.* **(left side, unfiltered)** else compute the right null constraint
     $$C' \leftarrow F\big(K_C, \chi(\mathsf{att}) \| \chi(\mathsf{att}') \| \chi(\mathsf{att}') \| \perp\big)$$
     - *v.* set the token tree node with table positions $\mathtt{pos} = \mathit{index}(\mathbf{T}, P[N])$ and $\mathtt{pos}' = \mathit{index}(\mathbf{T}', P[N])$
     $$N_{\mathsf{TT}} \leftarrow \big(C, C', K^{(\mathsf{XJ})}, \mathtt{pos}, \mathtt{pos}'\big)$$
  5. output the token tree TT.

Figure 4.11: The PKFK scheme.

- $\mathsf{Query}(\mathsf{EDB}, \mathsf{TT})$:

  1. parse EDB as $\big(\mathsf{SET}, \mathsf{EMM}^{(\mathsf{R})}, \mathsf{EMM}^{(\mathsf{S})}, \mathsf{EMM}^{(\mathsf{XJ})}\big)$;
  2. for every node N in post-order traversal in the token tree TT with its input table(s) from its subtree(s) $\mathsf{TT}_{\mathbf{in}}$ (and $\mathsf{TT}'_{\mathbf{in}}$) as $\mathbf{R}_{\mathbf{in}}$ (and $\mathbf{R}'_{\mathbf{in}}$) and its output table as $\mathbf{R}_{\mathbf{out}}$,
     - *a.* **(leaf filter)** if $N \equiv (\mathsf{stk}, \mathtt{pos})$,
       - *i.* compute
       $$\big(\mathsf{rtk}_1 \cdots, \mathsf{rtk}_s\big) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}(\mathsf{EMM}^{(\mathsf{S})}, \mathsf{stk})$$
       - *ii.* set its output
       $$\mathbf{R}_{\mathbf{out}} \leftarrow \mathbf{R}_{\mathbf{out}} \bigcup \big(\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s\big)^{\mathsf{T}}$$
     - *b.* **(internal filter)** else if $N \equiv (K, \mathtt{pos})$,
       - *i.* for each row $\mathbf{r} \in \mathbf{R}_{\mathbf{in}}$,
         1. if $H(K \| \mathbf{r}[\mathtt{pos}]) \in \mathsf{SET}$, then set
         $$\mathbf{R}_{\mathbf{out}} \leftarrow \mathbf{R}_{\mathbf{out}} \bigcup \{\mathbf{r}\}$$
     - *c.* **(project)** else if $N \equiv \big((\mathtt{pos}_1, c_1), \cdots, (\mathtt{pos}_z, \mathbf{c}_z)\big)$,
       - *i.* for each row $\mathbf{r} \in \mathbf{R}_{\mathbf{in}}$,
         1. for all $j = 1, \cdots z$,
            - *a.* look up the encrypted row
            $$\mathbf{r}_j \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}(\mathsf{EMM}^{(\mathsf{R})}, \mathbf{r}[\mathtt{pos}_j])$$
            - *b.* update the cell
            $$\mathbf{r}[\mathtt{pos}_j] \leftarrow \mathbf{r}[\mathtt{pos}_j] \| \mathbf{r}_j[c_j]$$
       - *ii.* set $\mathbf{R}_{\mathbf{out}} \leftarrow \mathbf{R}_{\mathbf{in}}$;
     - *d.* **(constant)** else if $N \equiv [\mathrm{ct}]$, set $N_{\mathsf{TT}} = \big[\mathrm{ct}\big]$
     - *e.* **(cross product)** else if $N \equiv \mathsf{TT}_{\mathbf{in}} \times \mathsf{TT}'_{\mathbf{in}}$, set output $\mathbf{R}_{\mathbf{out}} \leftarrow \mathbf{R}_{\mathbf{in}} \times \mathbf{R}'_{\mathbf{in}}$;

(Continued in Fig. 4.13)

Figure 4.12: The PKFK scheme.

(Continued from Fig. 4.12)

- $\mathsf{Query}(\mathsf{EDB}, \mathsf{TT})$:

3. *f.* **(join)** else if $N \equiv (C, C', K, \mathtt{pos}, \mathtt{pos}')$,
   - *i.* **(left side, unfiltered)** if $\mathsf{TT}_{\mathbf{in}}$ does not have any leaf filter node at $\mathtt{pos}$,
     1. compute the token
     $$\mathsf{tk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K, C)$$
     2. compute the pairs of row tokens and surrogates $E = (\mathsf{rtk}_i, g_i)_i$ on left-side table as
     $$E \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}\big(\mathsf{EMM}^{(\mathsf{XJ})}, \mathsf{tk}\big)$$
   - *ii.* **(left side, filtered)** else, for each row $\mathbf{r} \in \mathbf{R}_{\mathbf{in}}$ and the row token $\mathsf{rtk} = \mathbf{r}[\mathtt{pos}]$,
     1. compute token
     $$\mathsf{tk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K, C \| \mathsf{rtk})$$
     2. compute the pairs of row tokens and surrogates $E = (\mathsf{rtk}_i, g_i)_i$ on left-side table as
     $$E \leftarrow E \bigcup \Sigma_{\mathsf{MM}}.\mathsf{Query}\big(\mathsf{EMM}^{(\mathsf{XJ})}, \mathsf{tk}\big)$$
   - *iii.* **(right side, unfiltered)** if $\mathsf{TT}_{\mathbf{in}}$ does not have any leaf filter node at $\mathtt{pos}'$,
     1. compute the token
     $$\mathsf{tk}' \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\big(K, C'\big)$$
     2. compute the pairs of row tokens and surrogates $E' = (\mathsf{rtk}'_j, g'_j)_j$ on right-side table as
     $$E' \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}\big(\mathsf{EMM}^{(\mathsf{XJ})}, \mathsf{tk}'\big)$$
   - *iv.* **(right side, filtered)** else, for each row $\mathbf{r}' \in \mathbf{R}'_{\mathbf{in}}$ and the row token $\mathsf{rtk}' = \mathbf{r}'[\mathtt{pos}']$
     1. compute the token
     $$\mathsf{tk}' \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\big(K, C' \| \mathsf{rtk}'\big)$$
     2. compute the pairs of row tokens and surrogates $E' = (\mathsf{rtk}'_j, g'_j)_j$ on right-side table as
     $$E' \leftarrow E' \bigcup \Sigma_{\mathsf{MM}}.\mathsf{Query}\big(\mathsf{EMM}^{(\mathsf{XJ})}, \mathsf{tk}'\big)$$
   - *v.* compute the joined result table using relational join[a]
   $$\mathbf{R}_{\mathbf{out}} \leftarrow \mathbf{R}_{\mathbf{in}} \underset{\mathtt{pos}=\mathsf{rtk}.}{\bowtie} E \underset{g.=g'.}{\bowtie} E' \underset{\mathsf{rtk}'.=\mathtt{pos}'}{\bowtie} \mathbf{R}'_{\mathbf{in}}$$
4. for each row token $\mathsf{rtk}_{i,j}$ at the i-th row and j-th column in table $\mathbf{R}_{\mathbf{out}}$, replace it with the encrypted row
$$\mathbf{R}_{\mathbf{out}}[i, j] \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Query}\big(\mathsf{EMM}^{(\mathsf{R})}, \mathsf{rtk}_{i,j}\big)$$
5. output the result table $\mathbf{R}_{\mathbf{out}}$.

[a] Concretely to achieve optimal time complexity any linear-time plaintext join algorithm will do, for example the Surrogate Join (Fig. 4.14, Sec. 4.6) proposed as part of this work.

Figure 4.13: The PKFK scheme.

Join pattern. For a *filtered* join between two tables $\mathbf{T}$ and $\mathbf{T}'$ on join predicate $\mathsf{att} = \mathsf{att}'$ with a filter $\mathsf{att}_P = a$ on $\mathbf{T}$ and a filter $\mathsf{att}'_P = a'$ on $\mathbf{T}'$, an adversary learns how many times the unique combinations of filter and join attributes $(\mathsf{att}, \mathsf{att}', \mathsf{att}_P)$ and $(\mathsf{att}, \mathsf{att}', \mathsf{att}'_P)$ have been queried before, the number of rows in the filtered join result, how many times each row has joined with another row in the join result, and how many times each row in the join result has been accessed in the past. Note that this filtered join pattern is restricted to the result of this filtered join rather than the full join. Even if an adversary observes other rows independently, say from other queries, it would not be able to uncover more join patterns beyond this filtered join. For one-sided filtered join, say a filter $\mathsf{att}_P = a$ on $\mathbf{T}$ but none on $\mathbf{T}'$, the join pattern is also restricted to one-sided filtered join result. Lastly, for *unfiltered* joins, an adversary learns the join pattern as described above for all rows in $\mathbf{T}$ and $\mathbf{T}'$ with respect to the join attributes.

Select pattern. For the selection pattern, the leaf filter for $\mathsf{att} = a$ reveals to an adversary how many times this attribute and value pair (att, a) has appeared in past queries, how many rows match this filter, and how many times each row has been accessed in past queries. Each internal filter also reveals how many rows (1) were previously observed from the result of this query subtree or past queries, and (2) match this internal filter; and how frequently each row in the result has been accessed in the past.

Project pattern. The project pattern reveals to an adversary the number of columns being projected on, how frequent each column has been accessed in the past, and the ciphertext length of each projected cell.

Row pattern. The row pattern includes the number of rows that match the query tree, the size of each row (i.e. the number of columns in a table), and how frequently each row in the result has been accessed in the past.

Cross-product and constant patterns. The cross-product and constant do not involve querying any data structure encryption scheme, so their leakage only includes the input and output size.

Variant. Note that the leakage profile of PKFK can be further improved with slight modifications to the 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 project node. We defer the details of this variant to the full version of this work.

4.6 Efficiency

We now turn to analyzing the query and storage efficiency of PKFK.

Asymptotic improvement. PKFK's query and storage complexity remain linear in database size under natural assumptions. This complexity is lower than the worst-case quadratic blowup in SPX [63] and OPX [66] (Ch. 3). The main complexity improvement in PKFK over prior works comes from its optimal join algorithm, which matches the plaintext complexity in the sense that it is asymptotically linear in query *input* size (or database size) [101, 73]. Note that this is different from the analysis of SPX and OPX which used a different optimality notion for joins that is asymptotically linear in query *output* size. PKFK also improves the storage complexity from quadratic to linear, and does not require precomputing all possible joins, therefore it requires one pass over the plaintext database to set up the encryption.

Concrete efficiency. Though PKFK asymptotically matches the plaintext complexity, it is still important to understand the hidden constants in its concrete efficiency. We implemented PKFK

in KafeDB (Ch. 2), and evaluates performance against OPX, SPX and PPE-based systems such as CryptDB. The TPC-H benchmark showed that KafeDB had 4.2× query overhead and 3.63× storage overhead over plaintext PostgreSQL. The overhead achieved by using PKFK was two to three orders of magnitude over OPX and SPX on the same scale factor in the benchmark. More details are presented in Chapter 6.

4.6.1 Optimal Join

We first analyze the efficiency of the join in PKFK for both a single join and conjunctive joins. The key idea here is that we construct an optimal plaintext join algorithm called GJoin using the join representation $\mathsf{MM}^{(\mathsf{XJ})}$ (Sec. 4.4) in PKFK, and then by optimality of the encrypted multi-map scheme used to encrypt the join representation, we derive the PKFK join with the same complexity.

Computation vs output. First we introduce the optimality notion. The output complexity of a join is worst-case quadratic in table size T, so merely outputing the join result may incur quadratic time. However computing the join does not necessarily need more than $\mathcal{O}(T)$ time. Indeed, there exist plaintext join algorithms that require only one pass over the tables and requires only linear time to compute (e.g. [101, 73]). The main reason why computing the join can run in time $\mathcal{O}(T)$ is because it can use more efficient data structure to represent the join result table.

Achieving linear-time and space computation for joins is important, because for a complicated query of multiple joins, the output step does not necessarily need to happen at each single join, but rather only takes place if the query reaches the end and needs to materialize results physically as a table. So it is often possible to delay the output for all the joins until the very end of the query.

Therefore we define optimality to be mainly on the complexity of join computation. Note however that the output step in an optimal join algorithm should still not exceed the join result size, because this is the minimum cost to physically write each row in the joined result.[2]

[2]This requirement to tie the output step's complexity to the output size rules out the hypothetical join algorithm that does not perform any computation but simply returning the two input tables, and uses the output step to compute the actual join.

The Surrogate Join Algorithm

The Surrogate Join or GJoin algorithm (Fig. 4.14) used in our proof can be seen as the PKFK join but (1) without encryption, i.e. with only the $\mathsf{MM}^{(\mathsf{XJ})}$ [3] and (2) instantiated with a concrete underlying join algorithm such as for Step (3.f.v) in the Query algorithm (Fig. 4.13). So the overall complexity of GJoin is equal to the PKFK join assuming an optimal multi-map encryption scheme.

To better analyze the efficiency improvement of GJoin, note that it separates the *computation* of the join(s) versus the *output* of the final result of the join(s). At a high level, it recursively defines two algorithms, one on the leaf join (i.e., join with tables as input) and the other on the internal join (i.e. join with other joins as input). Both are called $\mathsf{GJoin}(\cdot)$ with only difference in the algorithm's signature. The internal join algorithm recursively calls itself on its two subtrees, until it reaches the leaf, in which case it calls the leaf join algorithm.

Pointer sets. GJoin makes use of $\mathsf{PSET}_{\mathbf{T}}$, which is a set of pointers to the rows of a table $\mathbf{T}$, $\mathsf{PSET}_{\mathbf{T}} = \bigcup_{\mathbf{r}\in\mathbf{T}} \chi(\mathbf{r})$. Here we use the same $\chi(\mathbf{r})$ to denote both coordinate and pointer of $\mathbf{r}$ due to the equivalence. We assume "dereferencing" each pointer, or getting the row $\mathbf{r}$ given its coordinate $\chi(\mathbf{r})$ takes $\mathcal{O}(1)$ time.

Single join. The algorithm first constructs two pointer sets to input tables $\mathbf{T}$ and $\mathbf{T}'$, and then populates two E sets called $E_{\mathbf{T}}$ and $E_{\mathbf{T}'}$ with tuples of the same form as $\mathsf{MM}^{(\mathsf{XJ})}$, which are pairs of row pointers and surrogates. Note that here for simplicity we only consider equi-join of the form $\mathsf{att} = \mathsf{att}'$, where the surrogates are simple join attribute values. In general the surrogates can be defined over a more complex join predicate θ in a θ-join. The computation of the join is then done through the intersection of $E_{\mathbf{T}}$ and $E_{\mathbf{T}'}$ based on each tuple's second element i.e. the surrogate. If there is a match then the whole tuple is retained in each E set. This step essentially retains only row pointers that can be later used to construct this join's output.

Multiple joins. The recursive case in $\mathsf{GJoin}(\mathrm{Q}_J)$ for muli-way joins Q_J first picks one of join in Q_J, say $\mathrm{Q} \bowtie_{\mathbf{T}.\mathsf{att}=\mathbf{T}'.\mathsf{att}'} \mathrm{Q}'$, which divides the conjunctive join clauses into two subtrees Q, Q'.

[3] For simplicity we do not consider filters, which is a simple extension to GJoin.

Then it recursively calls $\mathsf{GJoin}(\cdot)$ on the two subtrees, which after reaching the base case (i.e. leaf join), will each return a list of $(E_i)_i$ sets, each corresponding to a table $\mathbf{T}_i$ in a join clause in each subtree. Let the two lists be $\mathcal{E}_{\mathrm{Q}}$ and $\mathcal{E}_{\mathrm{Q}'}$. On the other hand, it also calls the leaf join algorithm GJoin on the join $\mathbf{T}.\mathsf{att} = \mathbf{T}'.\mathsf{att}'$ to obtain two E sets $E_{\mathbf{T}}, E_{\mathbf{T}'}$. It then uses these two E sets to update the E sets for the same tables $\mathbf{T}, \mathbf{T}'$ found in $\mathcal{E}_{\mathrm{Q}}$ and $\mathcal{E}_{\mathrm{Q}'}$. This is done through intersection on the row pointers. Each of this step takes at most $\mathcal{O}(T)$ time, and does not increase the total storage size. Finally, if there are $M-1$ joins between M tables, then there will be M total E sets returned to represent results of Q.

Output. The output step first constructs two indexes (say hash tables) for each E set from GJoin. Then it uses depth-first search to enumerate combinations of row pointers in all m E sets.

Proof of Theorem 4.3.1

Theorem 4.3.1 (STE-based join in PKFK is optimal). *If the multi-map encryption scheme Σ_{mm} is optimal, then the join in PKFK has optimal asymptotic complexity.*

Proof. **Time complexity.** For simplicity, assumes all tables in the database have the same number of rows T and number of attributes L. For the base case: one join between two tables, computing the pointer sets PSET and the E sets takes $\mathcal{O}(T)$ steps. The intersection of E sets takes at most $\mathcal{O}(T)$ steps, and the updated E sets can never increase beyond size $\mathcal{O}(T)$ (or a better bound is $\mathcal{O}(\min(|Q|, T))$). Finally the algorithm returns just the two updated E sets as an intermediate representation of the *computation* of the join. It is thus clear that the leaf GJoin algorithm always takes time $\mathcal{O}(T)$ and space $\mathcal{O}(\min(|Q|, T))$ for leaf join query output size $\mathcal{O}(T^2)$.

For the recursive case for m-way joins $Q = \mathbf{T}_1 \bowtie_1 \cdots \bowtie_{m-1} \mathbf{T}_m$, it splits the $m-1$ joins into two parts over the k-th join $\bowtie_k$. Its recursive calls to GJoin on these two parts return a list of $L = (E_i)_i$ sets, each corresponds to a table $\mathbf{T}_i$. The real computation happens in taking the pointer sets, calls the leaf join GJoin on $\bowtie_k$, then updates the corresponding E sets via the same intersection except by the second element in each tuple, namely the row pointers. This step takes at most $\mathcal{O}(T)$ time, and does not increase the total storage size beyond $\mathcal{O}(T)$. Finally, for m total

E sets produced in GJoin recursively, assuming m is a small constant dominated by T, we obtain $\mathcal{O}(T)$ time and space for recursive GJoin.

For the output step, building the indexes (say hash tables) on m total E sets takes $\mathcal{O}(m \cdot \min(|Q|, T))$ time by passing over each E set once where each E set has size $\mathcal{O}(\min(|Q|, T))$. For enumeration of joined rows, because row pointers in each E set are certain to be in the final result, it takes asymptotically as many steps as the join result size $\mathcal{O}(|Q|)$. Again assuming m is dominated by T, the output step takes $\mathcal{O}(|Q|)$ time but $\mathcal{O}(T)$ space.

The final step is to note the equivalence of the E sets with the $\mathsf{MM}^{(\mathsf{XJ})}$.

Storage complexity. First consider the case of unfiltered joins. For a single join, for each row $\mathbf{r}$ in table $\mathbf{T}$, there is only one tuple $(\chi(\mathbf{r}), \mathbf{r}[\mathsf{att}])$ in $\mathsf{MM}^{(\mathsf{XJ})}$. Similar is the case for $\mathbf{T}'$. So there are in total $2T$ tuples in $\mathsf{MM}^{(\mathsf{XJ})}$. Instantiated by Σ_{MM}, the storage complexity is therefore $\mathcal{O}(T)$. This matches the plaintext storage complexity for the two attributes.

For filtered joins, the filter attributes are in either $\mathbf{T}$ or $\mathbf{T}'$. For each such filter attribute att_P, we need to add one tuple $(C_w\chi(\mathbf{r}), \mathbf{r}[\mathsf{att}])$ where the constraint C_w is unique per combination of join attributes $(\mathsf{att}, \mathsf{att}')$, filter attribute att_P and filter value $\mathbf{r}[\mathsf{att}_P] = w$. Therefore at most we only add L tuples per row $\mathbf{r}$ for this join in $\mathsf{MM}^{(\mathsf{XJ})}$. So the total size of $\mathsf{MM}^{(\mathsf{XJ})}$ is $(L \cdot T)$.

For total M possible joins in the database, $\mathsf{MM}^{(\mathsf{XJ})}$ has $(M \cdot L \cdot T)$ entries for the entire database. Here we follow the natural assumption that T dominates M and L which are typically a small constant. So the total storage for $\mathsf{EMM}^{(\mathsf{XJ})}$ is $\mathcal{O}(T)$.

■

4.6.2 Optimal Query and Storage

We extend the proof to STE-based Queryalgorithm in PKFK by adding in other representations such as the filter representation (Sec. 4.4), and argue that these do not increase the asymptotic complexity of the plaintext query algorithm. Then using the same strategy, by assuming optimal encrypted multi-map construction we then derive the same complexity for the PKFK Query algorithm.

1. $\mathsf{GJoin}(\mathbf{T} \bowtie_{\mathbf{T}.\mathsf{att}=\mathbf{T}.\mathsf{att}'} \mathbf{T}', \mathsf{PSET}_{\mathbf{T}}, \mathsf{PSET}_{\mathbf{T}'})$:
   a. given a join $\mathbf{T} \bowtie_{\mathbf{T}.\mathsf{att}=\mathbf{T}.\mathsf{att}'} \mathbf{T}'$ and two pointer sets $\mathsf{PSET}_{\mathbf{T}}, \mathsf{PSET}_{\mathbf{T}'}$ for table $\mathbf{T}$ and $\mathbf{T}'$:
   b. initialize two empty sets $E_{\mathbf{T}}, E_{\mathbf{T}'}$;
   c. for each element $\chi(\mathbf{r}) \in \mathsf{PSET}_{\mathbf{T}}$, add a pair $(\chi(\mathbf{r}), g)$ to $E_{\mathbf{T}}$ for surrogate $g = \mathbf{r}[\mathsf{att}]$[a];
   d. for each element $\chi(\mathbf{r}') \in \mathsf{PSET}_{\mathbf{T}'}$, add a pair $(\chi(\mathbf{r}'), g')$ to $E_{\mathbf{T}'}$ for surrogate $g' = \mathbf{r}'[\mathsf{att}']$;
   e. update $E_{\mathbf{T}}$ by intersecting it with $E_{\mathbf{T}'}$ based on the surrogates as $E_{\mathbf{T}} \leftarrow E_{\mathbf{T}} \bigcap |_2 E_{\mathbf{T}'}$; similarly update $E_{\mathbf{T}'}$;
   f. Return $(E_{\mathbf{T}}, E_{\mathbf{T}'})$.
2. $\mathsf{GJoin}(\mathrm{Q} \bowtie_{\mathbf{T}.\mathsf{att}=\mathbf{T}'.\mathsf{att}'} \mathrm{Q}')$:
   a. given a join $\mathrm{Q} \bowtie_{\mathbf{T}.\mathsf{att}=\mathbf{T}'.\mathsf{att}'} \mathrm{Q}'$
   b. run $\mathsf{GJoin}(\mathrm{Q})$ to get a list of $\mathcal{E}_{\mathrm{Q}} \equiv (E_T)_{T \in \mathrm{Q}}$; similarly run $\mathsf{GJoin}(\mathrm{Q}')$ to get $\mathcal{E}_{\mathrm{Q}'} \equiv (E_T)_{T \in \mathrm{Q}'}$;
   c. let pointer sets $\mathsf{PSET}_{\mathbf{T}}$ be $E_{\mathbf{T}}|_1 \in \mathcal{E}_{\mathrm{Q}}$ and $\mathsf{PSET}_{\mathbf{T}'}$ be $E_{\mathbf{T}'}|_1 \in \mathcal{E}_{\mathrm{Q}'}$;
   d. run $\mathsf{GJoin}(\mathbf{T} \bowtie_{\mathbf{T}.\mathsf{att}=\mathbf{T}'.\mathsf{att}'} \mathbf{T}', \mathsf{PSET}_{\mathbf{T}}, \mathsf{PSET}_{\mathbf{T}'})$ with pointer sets to get a pair of $(E^J_{\mathbf{T}}, E^J_{\mathbf{T}'})$;
   e. update $E_{\mathbf{T}}$ in $\mathcal{E}_{\mathrm{Q}}$ by intersecting it with $E^J_{\mathbf{T}}$ on the row pointers as $E_{\mathbf{T}} \leftarrow E_{\mathbf{T}} \bigcap |_1 E^J_{\mathbf{T}}$; and similarly update $E_{\mathbf{T}'} \leftarrow E_{\mathbf{T}'} \bigcap |_1 E^J_{\mathbf{T}'}$ in $\mathcal{E}_{\mathrm{Q}'}$;
   f. return the updated $\mathcal{E}_{\mathrm{Q}} \cup \mathcal{E}_{\mathrm{Q}'}$.
3. $\mathsf{ExecJoin}(Q)$:
   a. **(Computation)** given query $\mathrm{Q} = \mathbf{T}_1 \bowtie_1 \cdots \bowtie_{m-1} \mathbf{T}_m$, run $\mathsf{GJoin}(\mathrm{Q})$ to get a list of $\mathcal{E}_{\mathrm{Q}} \equiv (E_i)_{i \in [m]}$;
   b. **(Output)** for each $E_i \in \mathrm{Q}$:
      i. create a dictionary $\mathsf{MM}^{(1)}_i$ that maps each $r \in E_i|_1$ to some $g \in E_i|_2$ such that $(r, g) \in E_i$;
      ii. create a inverse lookup index $\mathsf{MM}^{(2)}_i$ that maps each $g \in E_i|_2$ to some subset $R_i \subseteq E_i|_1$ such that for all $r \in R_i$ we have $(r, g) \in E_i$;
      iii. define a recursion $\mathsf{OutputJoinRow}(i, r, X)$:
         1. if $i \geq m$: output X;
         2. for each $g \in \mathsf{MM}^{(1)}_i[r]$,
            a. for each $r' \in \mathsf{MM}^{(2)}_{i+1}[g]$: run $\mathsf{OutputJoinRow}(i+1, m, r', X \cup \{r'\})$
      iv. for each $r_1 \in E_1|_1$: run $\mathsf{OutputJoinRow}(1, r_1, (r_1))$;

[a] This surrogate is defined for θ-join predicate $\theta \equiv \mathsf{att} = \mathsf{att}'$ for example, though the surrogate definition can be extended to more complex θ.

Figure 4.14: The Surrogate Join GJoin algorithm.

Theorem 4.6.1 (PKFK Query algorithm is optimal)**.** *If the encrypted multi-map scheme* Σ_{mm} *is optimal, then the* Query *algorithm in* PKFK *has the optimal asymptotic complexity: (1) the query time complexity is linear in input table size; and (2) the storage complexity is also linear in table size.*

Proof. We present a sketch here due to the simplicity. The proof leverages Theorem 4.3.1 by extending it with filters. We extend the GJoin algorithm (Fig. 4.14) with a multi-map MM_S that maps filter attribute values to row pointers, and a set SET_S for all filter attribute values. Then we check that this multi-map is equivalent to $\mathsf{MM}^{(\mathsf{S})}$, and this set is equivalent to SET. To check the time complexity, we just need to verify that the only operation added are intersections between MM_S, SET_S and E sets, therefore only require $\mathcal{O}(T)$ time. MM_S and SET_S require $\mathcal{O}(L \cdot T)$ for L filter attributes, but since L is typically a small constant dominated by T, so the storage complexity is also $\mathcal{O}(T)$ when instantiated with an optimal multi-map encryption scheme.

■

Chapter 5

Emulation and Colocation

5.1 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, emulation takes an encrypted data structure (e.g., an encrypted multi-map) and finds a way to represent it as another data structure (e.g., a table) without significant 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. However challenges still remain particularly in how to bridge the semantics and complexity gaps between the STE-based query algorithm and its relational emulation.

Emulation of STE-based schemes. STE-based schemes such as SPX [64] and PKFK make black-box usage of multiple STE structures, therefore the emulation is naturally divided into two levels. First at the lower level we emulate each of the concrete STE structures (and their query algorithms) as tables and SQL subqueries. Then at the higher level we recursively transform token trees by composing these lower-level emulated queries into relational queries.

Concretely, given an STE-based scheme, emulation should define two algorithms, one that maps the output of STE-based Setup to tables called *emulated tables*, and the other that maps outputs of the STE-based Token to relational queries called *emulated queries*, which also specify the semantics of the STE-based Query. As a result, the emulated queries can be readily executed on any standard relational database while preserving the security of the original STE-based scheme.

Definition 5.1.1. *[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* 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* $\mathsf{Token}(K, q)$*,* $\mathsf{SQL.Exec}(\mathsf{DB}, Q) = \mathsf{STE.Query}(\mathsf{EDS}, \mathsf{tk})$*. We extend the same syntax also to a collection of tokens such that the* Reform *works on a collection or column of* tk*s*

Semantics gaps. STE schemes such as Pibase [30] often rely on loop-based iterations, which presents a challenge to emulation because relational algebra does not have a loop-based construct [1]. Furthermore in black-box constructions such as SPX [64] and PKFK, a single relational query consists of multiple tokens for querying multiple STE structures. Therefore we developed two general techniques to render STE structures amenable to relational emulation: (1) restructure

loop-based iteration as recursion, and (2) extend query specification to multiple tokens. We will show that solving (1) enables emulation to use relational operators beyond relational algebra; and for (2) how to leverage the relational semantics of relational algebra to extend the Reform algorithm in Def. 5.1.1 to a collection or column of tokens.

Security. 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. Similarly, the emulated/reshaped structure, DB, and the emulated/reformed token, Q, reveal nothing beyond the query leakage of EDS and tk.

Efficiency gaps. 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 that is: (1) larger than the original structure EDS; and (2) less query-efficient. In this work we aim for emulators that do not affect the efficiency of the pre-emulated structure.

5.1.1 Emulation for Encrypted Multi-Maps

STE structures like encrypted multi-maps can be instantiated with different constructions, each of which may need to be emulated differently. In PKFK we make use of the Pibase construction by Cash et al. [30] and our colocation-friendly variant (Sec. 5.2). Here we show how to emulate Pibase, and defer the details for colocation-friendly variant later in Section 5.2.

Overview of Pibase. We recall the $\mathsf{Pibase} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$ scheme from [30]. The Setup algorithm of Pibase samples a key K and instantiates a history-independent 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 $(\ell'_i, \mathsf{Enc}_{K_{\ell,2}}(v_i))$ in the dictionary DX. The encrypted multi-map EMM consists of the dictionary DX. EMM = DX is sent to the server.

To Query 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.

Emulation for Pibase. The Pibase EMM is stored as a dictionary, which can be viewed as a relation of label and value pairs, so it can be readily stored as a table of two columns. However as mentioned previously, some challenges still remain for Pibase emulation. First, relational algebra does not have a direct counterpart to loop-based iteration, which is needed by Pibase's Query algorithm when iterates over a counter. For relational queries, the same algorithm is also repetitively invoked, for example when PKFK's Queryalgorithm iterates over multiple row tokens from the pushdown selection, or iterates over multiple surrogates from the left-hand side of the join to compute for the right-hand side of the join.

To address these issues, we used two techniques for Pibase query emulation:

1. we restructure the iteration-based Pibase Query as recursion;
2. we extend the same emulation for a collection of tokens.

Recursion in (1) allows us to use the fixed-point operator [6] or equivalently the common table expression in SQL [44]. The relational semantics of the resulting relational query is used to address (2).

The SQL emulator for Pibase is detailed in Figure 5.1 and works as follows. Given an encrypted multi-map EMM, the Reshape algorithm creates a table $\mathbf{T}$ and schema $\mathbb{S}(\mathbf{T}) = (\texttt{label}, \texttt{value})$ by executing

```
CREATE TABLE T (label, value).
```

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

```
INSERT INTO T VALUES (ℓ1, e1) ..., (ℓm, em).
```

The table $\mathbf{T}$ thus contains two columns, one for encrypted labels and one for encrypted values.

To query $\mathbf{T}$ for a set of plaintext labels L, given a collection or column of tokens under the column name K_L located in a table also denoted as K_L, the Reform algorithm outputs the following relational query using the fixed-point operator [6]

$$\bigcup_{i=1}^{\infty} \mathbf{T} \underset{\texttt{label}=\mathsf{Udf}_F(K_L,i)}{\bowtie} K_L$$

This query is recursively defined as a union over all i-th nonempty joins, until the last join returns an empty set. This corresponds to invoking the Pibase Query algorithm on multiple tokens in K_L, and for each such token it iterates over a counter i to compute the i-th encrypted label until none is found, which happens when $i = \max_{\ell \in L} \#\mathsf{MM}[\ell]$. Note that for each token the number of encrypted labels may differ. Also, Udf_F, the built-in user-defined function [1] for a PRF, accepts a collection or column of values as argument. This emulation strategy leverages the relational semantics to circumvent the lack of loop-based iteration in relational algebra.

Reducing overhead. Although the above emulated query is semantically correct, its specification is not tight in the sense that it allows for unnecessary computation. Each join in the union is invariably between the same two tables, the emulated EMMtable $\mathbf{T}$ and the table of tokens K_L (where the counter i is incremented in the join predicate for encrypted labels). This means that at each counter i the query still needs to check for all tokens in K_L for matching encrypted labels in $\mathbf{T}$, even for the subset of tokens that produce no labels at previous counter $i-1$. This means even in case of varying response lengths, all tokens in K_L result in the same amount of computation which is the maximum response length over all token in K_L. Assuming each recursive join is linear-time in input size, each i-th join in this emulated query takes $\mathcal{O}(|\mathbf{T}| + \#K_L)$ time, for total of $\max_{\ell \in L} \#mm[\ell]$ joins. So the total time is $\mathcal{O}\big(\max_{\ell \in L} \#\mathsf{MM}[\ell] \cdot (|\mathbf{T}| + \#K_L)\big)$. For uneven response lengths for each token, the redundant computation can be significant.

To improve this, we leverage a simple observation that Pibase assigns contiguous integers to the counters, so we only need to check for the subset of tokens that had encrypted labels at the previous counter. So we restructure the recursion by introducing an intermediate view $\mathbf{G}_i$ for each

counter i, such that the overall query becomes

$$\bigcup_{i=1}^{\infty} \mathbf{G}_i : \mathbf{G}_i = \mathbf{T} \underset{\texttt{label}=\mathsf{Udf}_F(K_L,i)}{\ltimes} \begin{cases} K_L & \text{if } i = 1 \\ \mathbf{G}_{i-1} & \text{else} \end{cases}$$

Because the intermediate view $\mathbf{G}_i$ only contains the subset of tokens in K_L that has at least i labels, the next view $\mathbf{G}_{i+1}$ will only compute on this subset of tokens. Overall this modified query only takes time $\mathcal{O}(\max_{\ell \in L} \#\mathsf{MM}[\ell] \cdot |\mathbf{T}| + \sum_{\ell \in L} \#\mathsf{MM}[\ell]))$.

To do away with the $\max_{\ell \in L} \#\mathsf{MM}[\ell] \cdot |\mathbf{T}|$ additive factor inherent in each recursive join, we can pre-build an index (e.g. hash index) on $\mathbf{T}$'s `label` column to help reducing the cost of each recursive join, by the SQL statement

```
CREATE INDEX ON T (label).
```

With this modification, the emulated query takes only time $\mathcal{O}(\sum_{\ell \in L} \#\mathsf{MM}[\ell])$, which matches the pre-emulation scheme for $\#L$ tokens.

Storage complexity. The storage overhead of the emulated encrypted multi-map is equal to the size of the encrypted table, $|\mathbf{T}| = O(\sum_{\ell \in \mathbb{L}} \#\mathsf{MM}[\ell])$ where $\mathbb{L}$ denotes the set of plaintext labels stored in MM, plus the size of the plaintext index created over $\mathbf{T}$.`label` column, which is also $O(|\mathbf{T}|)$. So the overall size of the emulated encrypted multi-map is equal to $O(\sum_{\ell \in \mathbb{L}} \#\mathsf{MM}[\ell])$.

SQL syntax. The tightened emulated query can be written in SQL syntax using the common table expression [44, 90],

```
WITH RECURSIVE G(i, 𝕊(T), 𝕊(K_L)) AS {
  SELECT 1 AS i, 𝕊(T), 𝕊(K_L) FROM T JOIN K_L ON T.label = Udf_F(K_L, i)
  UNION ALL
  SELECT i+1 AS i, 𝕊(T), 𝕊(K_L) FROM T JOIN G ON T.label = G.label
  WHERE T.label = Udf_F(K_L, i+1))
}
SELECT 𝕊(T), 𝕊(K_L) FROM G.
```

- Reshape(EMM):
  1. create a table $\mathbf{T}$ with columns `label` and `value` by executing
     $$\texttt{CREATE TABLE } \mathbf{T}(\texttt{label}, \texttt{value})$$
  2. parse EMM as a dictionary DX;
  3. for each label/value pair (ℓ, e) in DX, inserts the row $\mathbf{r} = (\ell, e)$ in $\mathbf{T}$ by executing
     $$\texttt{INSERT INTO } \mathbf{T} \texttt{ VALUES } (\ell_1, e_1) \ldots, (\ell_m, e_m)$$
  4. create a hash index on $\mathbf{T}$'s `label` column by executing
     $$\texttt{CREATE INDEX ON } \mathbf{T}(\texttt{label})$$
  5. output $\mathbf{T}$.
- Reform($\mathbb{S}(\mathbf{T}), K_L$):
  1. output a relational query
     $$\bigcup_{i=1}^{\infty} \mathbf{G}_i : \mathbf{G}_i = \mathbf{T} \underset{\texttt{label}=\mathsf{Udf}_F(K_L,i)}{\bowtie} \begin{cases} K_L & \text{if } i = 1 \\ \mathbf{G}_{i-1} & \text{else} \end{cases}$$
     or equivalently a SQL query
     ```
     WITH RECURSIVE G(i, S(T), S(K_L)) AS {
       SELECT 1 AS i, S(T), S(K_L) FROM T JOIN K_L ON T.label = Udf_F(K_L, i)
       UNION ALL
       SELECT i + 1 AS i, S(T), S(K_L) FROM T JOIN G ON T.label = G.label
       WHERE T.label = Udf_F(K_L, i + 1))
     }
     SELECT S(T), S(K_L) FROM G
     ```

Figure 5.1: The Pibase emulator.

5.1.2 Emulation for Sets

PKFK makes use of a set structure SET for the partial filter representation to handle conjunctive selections (Sec. 4.4). We first recall abstractly how this set structure is built and queried. We then proceed to describe its emulation.

Overview of SET Given a set of n pairs $S = \{(a_1, b_1), \cdots, (a_n, b_n)\}$, the client and the server jointly compute a membership function $f(S, (a, b))$ for the client input a and the server input b, and the server to learn the output of f without knowing a beyond its repetition. To construct this

function f, the client computes for each $(a, b) \in S$ an element $y = H(K_a \| b)$ where $K_a = F(K, a)$ for a secret key K, a random oracle H and a pseudo-random function F, and adds y to SET. The client finally stores SET on the server.

To test for membership for (a, b) in S, the client sends $K_a = F_{K_S}(a)$, the server inputs b, computes $e = H(K_a \| b)$ and outputs true if e exists in SET.

Emulation of SET For emulation, we need to extend the input and output of the set membership function to a collection of inputs in the form of tables because of the relational semantics of relational algebra. We show the details in Figure 5.2. In particular, we extend the server input to be a column of b's as B, and the output of the set membership to be a column of SET elements y's as Y such that the $y_i \in Y$ if and only if the corresponding input (a_i, b_i) is in S. For example in PKFK, the client input a is a token representing a selection predicate P, and the server input B contains row tokens from other conjunctive selections, then the output Y contains row tokens in B that also satisfy P.

Concretely, given SET as constructed above, the $\mathsf{Reshape}$ algorithm creates a table $\mathbf{T}$ and schema $\mathbb{S}(\mathbf{T}) = (\texttt{label})$ by executing

```
CREATE TABLE T(label).
```

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

```
CREATE INDEX ON T(label).
```

For every element y_i in SET, it inserts a row (y_i) into $\mathbf{T}$ by executing

```
INSERT INTO T VALUES (y_1), ..., (y_#SET).
```

It then outputs the table $\mathbf{T}$. To query $\mathbf{T}$, given the client input $K_a = F(K, a)$ and the server input of a column B in table $\mathsf{tbl}(B)$, the Reform algorithm outputs the relational query

$$\mathsf{tbl}(B) \underset{\mathsf{Udf}_F(K_S, K_a \| B) = \texttt{label}}{\ltimes} \mathbf{T}$$

This query makes use of the *semi join* [1] to filter $\mathsf{tbl}(B)$ based on a join predicate $(\mathsf{Udf}_F(K_S, K_a \| B) = \texttt{label})$ for each element in the column `label` of emulated set $\mathbf{T}$. This is

- $\mathsf{Reshape}(\mathsf{EDB})$:
  1. create a table $\mathbf{T}$ and schema $\mathbb{S}(\mathbf{T}) = (\texttt{label})$ by executing
     $$\texttt{CREATE TABLE } \mathbf{T}(\texttt{label})$$
  2. create an index over $\mathbf{T}$ by executing
     $$\texttt{CREATE INDEX ON } \mathbf{T}(\texttt{label})$$
  3. for every element y_i in SET, it inserts a row (y_i) into $\mathbf{T}$ by executing
     $$\texttt{INSERT INTO } \mathbf{T} \texttt{ VALUES } (y_1), \ldots, (y_{\#\mathsf{SET}})$$
- $\mathsf{Reform}(\mathbb{S}(\mathsf{EDB}), B)$:
  1. output the relational query
     $$\mathsf{tbl}(B) \underset{\mathsf{Udf}_F(K_S, K_a \| B) = \texttt{label}}{\ltimes} \mathbf{T}$$
     or equivalently the SQL query
     $$\texttt{SELECT } \mathbb{S}(\mathsf{tbl}(B)) \texttt{ FROM } \mathsf{tbl}(B) \texttt{ LEFT SEMI JOIN } \mathbf{T} \texttt{ ON } \mathsf{Udf}_F(K_S, a \| B) = \texttt{label}$$

Figure 5.2: The SET emulator.

equivalent to querying SET multiple times, for each server input $b \in B$ and the fixed the client input a. Equivalently the SQL query is

$$\texttt{SELECT } \mathbb{S}(\mathsf{tbl}(B)) \texttt{ FROM } \mathsf{tbl}(B)$$
$$\texttt{LEFT SEMI JOIN } \mathbf{T} \texttt{ ON } \mathsf{Udf}_F(K_S, a \| B) = \texttt{label}$$

Efficiency. Each check of the join predicate in the semi-join-based emulated query would have required a scan of each element of the emulated set $\mathbf{T}$. But with a hash index created on the `label` column in $\mathbf{T}$, the join predicate takes only $\mathcal{O}(1)$ time. For the total of $\#B$ server inputs per one client input a, the query takes $\mathcal{O}(\#B)$ time. The size of $\mathbf{T}$ equals the total number of pairs in the set S plus the size of the hash index which is also linear in the size of S, so in total $O(\#S)$.

5.1.3 Emulation for STE-based Relational Databases

STE-based relational database schemes such as SPX and PKFK make use of black-box STE structures like encrypted multi-maps and sets. Here for emulation of the STE-based scheme, we also

make black-box use of the emulation of the constructions of STE structures. We show the technique by focusing on the PKFK scheme. At a high level, for Setup, the Reshape algorithm takes as input the EDB, and emulates each encrypted structure by calling the associated emulator. For the Query algorithm with the token tree TT as input, the Reform algorithm recursively walks TT and depending on the node type, emulates each token into a relational subquery. Finally all such subqueries are incorporated into a single relational query. For more details we refer to the pseudo-code in

1. Reshape: Figure 5.4;
2. Reform: Figure 5.5 and 5.6.

Example. An example of emulation using Pibase [30] as the EMM construction is shown in Figure 5.3.

$\mathbf{T}^{(\mathsf{S})}$

label	value
$F(K_{(\mathsf{att},A),1}, 1)$	$E(K_{(\mathsf{att},A),2}, \mathsf{rtk}_1)$
$F(K_{(\mathsf{att},A),1}, 2)$	$E(K_{(\mathsf{att},A),2}, \mathsf{rtk}_2)$
$F(K_{(\mathsf{att},B),1}, 1)$	$E(K_{(\mathsf{att},B),2}, \mathsf{rtk}_3)$
$F(K_{(\mathsf{att}_P,Q),1}, 1)$	$E(K_{(\mathsf{att}_P,Q),2}, \mathsf{rtk}_1)$
$F(K_{(\mathsf{att}_P,Q),1}, 2)$	$E(K_{(\mathsf{att}_P,Q),2}, \mathsf{rtk}_3)$
$F(K_{(\mathsf{att}_P,R),1}, 1)$	$E(K_{(\mathsf{att}_P,R),2}, \mathsf{rtk}_2)$
$F(K_{(\mathsf{att}',A),1}, 1)$	$E(K_{(\mathsf{att}',1),2}, \mathsf{rtk}'_1)$
$F(K_{(\mathsf{att}',A),1}, 2)$	$E(K_{(\mathsf{att}',2),2}, \mathsf{rtk}'_3)$
$F(K_{(\mathsf{att}',B),1}, 1)$	$E(K_{(\mathsf{att}',3),2}, \mathsf{rtk}'_2)$

Figure 5.3: An example of emulation for the Pibase construction [30] for $\mathsf{EMM}^{(\mathsf{S})}$ in Figure 4.7. In general the rows should have been permuted due to history-indepence requirement.

5.2 Colocation

Relational data contains much information about data locality, such as values that are colocated in the same rows, columns or tables based on logical relationship. This locality informs database systems on how to improve query efficiency, by (1) prefetching colocated values from slower storage to faster cache, or (2) storing colocated values in nearby storage blocks. However, this locality

- $\mathsf{Reshape}(\mathsf{EDB})$:
  1. parse EDB as $\mathsf{SET}, \mathsf{EMM}^{(\mathsf{R})}, \mathsf{EMM}^{(\mathsf{S})}, \mathsf{EMM}^{(\mathsf{XJ})}$
  2. emulate each structure as

$$\mathsf{DB}_{\mathsf{SET}} \leftarrow \mathsf{Emu}_{\mathsf{SET}}.\mathsf{Reshape}(\mathsf{SET}), \quad \mathsf{DB}^{(\mathsf{R})}_{\mathsf{EMM}} \leftarrow \mathsf{Emu}_{\mathsf{EMM}}.\mathsf{Reshape}(\mathsf{EMM}^{(\mathsf{R})}),$$

$$\mathsf{DB}^{(\mathsf{S})}_{\mathsf{EMM}} \leftarrow \mathsf{Emu}_{\mathsf{EMM}}.\mathsf{Reshape}(\mathsf{EMM}^{(\mathsf{S})}), \quad \mathsf{DB}^{(\mathsf{XJ})}_{\mathsf{EMM}} \leftarrow \mathsf{Emu}_{\mathsf{EMM}}.\mathsf{Reshape}(\mathsf{EMM}^{(\mathsf{XJ})}),$$

  3. output $\mathsf{EDB} = (\mathsf{DB}_{\mathsf{SET}}, \mathsf{DB}^{(\mathsf{R})}_{\mathsf{EMM}}, \mathsf{DB}^{(\mathsf{S})}_{\mathsf{EMM}}, \mathsf{DB}^{(\mathsf{XJ})}_{\mathsf{EMM}})$.

Figure 5.4: The PKFK emulator Reshape algorithm.

- $\mathsf{Reform}(\mathbb{S}(\mathsf{EDB}), \mathsf{TT})$:
  1. parse $\mathbb{S}(\mathsf{EDB})$ as $(\mathbb{S}(\mathsf{DB}_{\mathsf{SET}}), \mathbb{S}(\mathsf{DB}^{(\mathsf{R})}_{\mathsf{EMM}}), \mathbb{S}(\mathsf{DB}^{(\mathsf{S})}_{\mathsf{EMM}}), \mathbb{S}(\mathsf{DB}^{(\mathsf{XJ})}_{\mathsf{EMM}}))$
  2. for each node N in token tree TT recursively visited in post-order traversal,
     a. let the result of visiting subtree(s) be $\mathrm{Q}_{\mathbf{in}}$ (and $\mathrm{Q}'_{\mathbf{in}}$);
     b. **(leaf filter)** if $N \equiv (\mathsf{stk}, \mathtt{pos})$, update the output query

$$\mathrm{Q}_{\mathbf{out}} \leftarrow \underset{\mathtt{value}}{\pi} \mathsf{Emu}_{\mathsf{MM}}.\mathsf{Reform}(\mathbb{S}(\mathsf{DB}^{(\mathsf{S})}_{\mathsf{EMM}}), \mathsf{stk})$$

     with schema $\mathbb{S}(\mathrm{Q}_{\mathbf{out}}) = (\mathtt{value})$
     c. **(internal filter)** else if $N \equiv (K_v, \mathtt{pos})$, update the output query
        i. get the emulated query

$$\mathrm{Q}_{\mathsf{SET}} \leftarrow \mathsf{Emu}_{\mathsf{SET}}.\mathsf{Reform}(\mathbb{S}(\mathsf{DB}_{\mathsf{SET}}), \mathrm{Q}_{\mathbf{in}}.\mathtt{pos})$$

        with schema $\mathbb{S}(\mathrm{Q}_{\mathsf{SET}}) = (\mathrm{Q}_{\mathbf{in}})$
        ii. update the output query

$$\mathrm{Q}_{\mathbf{out}} \leftarrow \mathrm{Q}_{\mathsf{SET}}$$

        with schema $\mathbb{S}(\mathrm{Q}_{\mathbf{out}}) = \mathbb{S}(\mathrm{Q}_{\mathbf{in}})$
     d. **(project)** else if $N \equiv \big((\mathtt{pos}_1, c_1), \cdots, (\mathtt{pos}_z, \mathbf{c}_z)\big)$,
        i. set $\mathrm{Q}_{\mathbf{out}} \leftarrow \mathrm{Q}_{\mathbf{in}}$
        ii. for all $j = 1, \cdots, z$,
           1. get the emulated query

$$\mathrm{Q}_j \leftarrow \pi_{\mathtt{label}, \mathtt{value}.c_j} \mathsf{Emu}_{\mathsf{EMM}}.\mathsf{Reform}(\mathbb{S}(\mathsf{DB}^{(\mathsf{R})}_{\mathsf{EMM}}), \mathrm{Q}_{\mathbf{out}}.\mathtt{pos}_j)$$

           with schema $\mathbb{S}(\mathrm{Q}_{\mathrm{Jn}}) = (\mathtt{label}, \mathtt{value}.c_j)$
           2. update the output query

$$\mathrm{Q}_{\mathbf{out}} \leftarrow \mathrm{Q}_{\mathbf{out}} \underset{\mathtt{pos}_j = \mathtt{label}}{\ltimes} \mathrm{Q}_j$$

     e. **(constant)** else if $N \equiv [\mathrm{ct}]$, update the output query $\mathrm{Q}_{\mathbf{out}} \leftarrow [\mathrm{ct}]$;
     f. **(cross product)** else if $N \equiv \mathsf{TT}_{\mathbf{in}} \times \mathsf{TT}'_{\mathbf{in}}$, update the output query $\mathrm{Q}_{\mathbf{out}} \leftarrow \mathrm{Q}_{\mathbf{in}} \times \mathrm{Q}'_{\mathbf{in}}$;

(Continued in Fig. 5.6)

Figure 5.5: The PKFK emulator Reform algorithm.

(Continued from Fig. 5.5)

- $\mathsf{Reform}(\mathbb{S}(\mathsf{EDB}), \mathsf{TT})$:

  2. *g.* **(join)** else if $N \equiv (C, C', K, \mathtt{pos}, \mathtt{pos}')$
     - *i.* **(left side, unfiltered)** if $\mathsf{TT}_{\mathbf{in}}$ does not have any leaf filter node at $\mathtt{pos}$,
       1. compute the token
          $$\mathsf{tk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K, C)$$
       2. get the emulated query
          $$\mathrm{Q}_E \leftarrow \underset{\mathtt{value}}{\pi} \mathsf{Reform}(\mathbb{S}(\mathsf{DB}^{(\mathsf{XJ})}_{\mathsf{EMM}}), \mathsf{tk})$$
          with schema $\mathbb{S}(\mathrm{Q}_E) = (\mathtt{value}) = ((\mathsf{rtk}, g))$
     - *ii.* **(left side, filtered)**
       1. get the token query
          $$\mathrm{Q}_{\mathsf{tk}} \leftarrow \underset{\Sigma_{\mathsf{MM}}.\mathsf{Token}(K, \mathsf{Udf}_F(C, \mathtt{pos}))}{\pi} \mathrm{Q}_{\mathbf{in}}$$
          with schema $\mathrm{Q}_{\mathsf{tk}} = (\mathsf{att}_{\mathsf{tk}})$
       2. get the emulated query
          $$\mathrm{Q}_E \leftarrow \underset{\mathtt{value}}{\pi} \mathsf{Emu}_{\mathsf{EMM}}.\mathsf{Reform}(\mathbb{S}\mathsf{DB}^{(\mathsf{XJ})}_{\mathsf{EMM}}, \mathrm{Q}_{\mathsf{tk}}.\mathsf{att}_{\mathsf{tk}})$$
          with schema $\mathbb{S}(\mathrm{Q}_E) = (\mathtt{value}) = ((\mathsf{rtk}, g))$ i.e. a column called $\mathtt{value}$ of pairs where the first element is called rtk and second element called g;
     - *iii.* **(right side, unfiltered)** if $\mathsf{TT}'_{\mathbf{in}}$ does not have any leaf filter node at $\mathtt{pos}'$,
       1. compute the token
          $$\mathsf{tk}' \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K, C')$$
       2. get the emulated query
          $$\mathrm{Q}'_E \leftarrow \underset{\mathtt{value}'}{\pi} \mathsf{Reform}(\mathbb{S}(\mathsf{DB}^{(\mathsf{XJ})}_{\mathsf{EMM}}), \mathsf{tk}')$$
          with schema $\mathbb{S}(\mathrm{Q}'_E) = (\mathtt{value}') = ((\mathsf{rtk}', g'))$
     - *iv.* **(right side, filtered)**
       1. get the token query
          $$\mathrm{Q}_{\mathsf{tk}'} \leftarrow \underset{\Sigma_{\mathsf{MM}}.\mathsf{Token}(K, F(C', \mathtt{pos}'))}{\pi} \mathrm{Q}'_{\mathbf{in}}$$
          with schema $\mathrm{Q}_{\mathsf{tk}'} = (\mathsf{att}_{\mathsf{tk}'})$
       2. get the emulated query
          $$\mathrm{Q}'_E \leftarrow \underset{\mathtt{value}'}{\pi} \mathsf{Emu}_{\mathsf{EMM}}.\mathsf{Reform}(\mathbb{S}\mathsf{DB}^{(\mathsf{XJ})}_{\mathsf{EMM}}, \mathrm{Q}_{\mathsf{tk}'}.\mathsf{att}_{\mathsf{tk}'})$$
          with schema $\mathbb{S}(\mathrm{Q}'_E) = (\mathtt{value}') = ((\mathsf{rtk}', g'))$
     - *v.* update the output query
       $$\mathrm{Q}_{\mathbf{out}} \leftarrow \mathrm{Q}_{\mathbf{in}} \underset{\mathtt{pos}=\mathtt{value}.\mathsf{rtk}}{\bowtie} \mathrm{Q}_E \underset{\mathtt{value}.g=\mathtt{value}'.g'}{\bowtie} \mathrm{Q}'_E \underset{\mathtt{value}'.\mathsf{rtk}'=\mathtt{pos}'}{\bowtie} \mathrm{Q}'_{\mathbf{in}}$$
       with schema $\mathbb{S}(\mathrm{Q}_{\mathbf{out}}) = \mathbb{S}(\mathrm{Q}_{\mathbf{in}}) \cup \mathbb{S}(\mathrm{Q}_E) \cup \mathbb{S}(\mathrm{Q}'_E) \cup \mathbb{S}(\mathrm{Q}'_{\mathbf{in}})$
  3. output query $\mathrm{Q}_{\mathbf{out}}$.

Figure 5.6: The PKFK emulator Reform algorithm.

is lost in STE-based schemes such as SPX [63] and PKFK mainly due to encryption. As a result STE-based databases may have limited scalability. In this section, we study how to augment an STE-based scheme to recover such data locality and what is the security trade-off.

Locality in relational databases. The relational model organizes data logically into the same columns, rows or tables according to relational semantics, such as attributes associated with the same entity (e.g. `Name`,`Age` and `Address`), and entities associated with the same concept (e.g. `Customers`) [1]. We call this logical notion of data locality as *colocation*, roughly speaking, to mean the knowledge of logically-related data likely to be accessed together. For instance, data colocated per row, column or table may be more likely accessed together in the presence of a projection, selection or join.

Colocation does not only imply one solution, i.e., to physically store the data sequentially together, though this way the random accesses may be reduced. Another way is to have the database system prefetch colocated values even if they are physically apart in order to hide slower random accesses with computation. These two designs can also be combined to achieve the best query performance. Reducing or hiding random accesses by exploiting colocation has become a central theme in the database research literature [101, 1, 73, 17, 68].

Locality in the STE literature. Locality has also been studied in SSE and STE [36, 32], however for a rather different data model and more stringent notion than colocation. The data model in consideration is mainly about keyword search, or key-value stores in the NoSQL literature [93, 33]. Locality here is to achieve minimum random access in each keyword search. We contend that the locality notion used in relational databases is more complex than what is considered in the STE literature, because the relational model is more complicated in two ways: (1) values may be colocated on the same row, column or table, and (2) queries may be composed of multiple operators that access values spanning across multiple such logical locations. Even on plaintext, the problem remains challenging to achieve minimum $\mathcal{O}(1)$ random access for all queries on all tables simultaneously while maintaining optimal query complexity. Instead, we aim at a different goal: to preserve the colocation through the STE structures for the server to reduce or hide overall random accesses.

5.2.1 Colocation in STE-based Schemes

Colocation in STE-based schemes means to disclose to the server which parts of the STE structure(s) belong logically to the same row, column or table, such that the server can use such knowledge to optimize either storage or query execution. Therefore colocation enhances the locality of an STE-based scheme at the cost of adding a new set of patterns in the leakage profile, called the colocation patterns. Our approach for STE-based colocation introduces small modification to the pre-emulation scheme. At a high level, we insert a set of values called the colocation tags to the structures during pre-emulation. When PKFK is instantiated with a specific EMM scheme based on a Pibase variant, these tags are revealed also during setup, and thus adding precisely the colocation patterns to the setup leakage. Then we leverage the property of emulation that preserves leakage to make colocation also legacy-friendly.

In the following, we introduce steps to turn an STE-based scheme such as PKFK into one with colocation. We then analyze the security trade-off in Sec. 5.2.3.

Colocation tags. First, we augment the PKFK scheme's Setup algorithm by inserting a set of values called the colocation tags inside each structure for each operator representation (e.g. inside each tuple of each multi-map). Details of the modification are in Figure 5.9 and 5.10. Each colocation tag reveals which parts within and across the structures correspond to some unique table, column or row, such as a tuple of pseudorandom values $\big(F_{K_1}(\mathsf{tbl}(\mathbf{T})), F_{K_2}(\chi(\mathbf{r})), F_{K_3}(\chi(\mathsf{att}))\big)$ for coordinates of the row $\mathbf{r}$ and the column att in a table $\mathbf{T}$ (except for $\mathsf{MM}^{(\mathsf{R})}$ that omits component on the column coordinate). Then the rest of the PKFK scheme and the emulation follows the existing definition. This step alone will increase the query leakage by an additional component called the colocation pattern. This is because searching a structure for each operator in the query in addition reveals the colocation tags in the response, which indicates in the response which ciphertexts from different encrypted structures correspond to a unique plaintext row, column or table.

Pibase Variant. In the second step, we increase the setup leakage by revealing these colocation tags, so that the server can exploit this colocation information to either store parts of the encrypted structures under the same colocation tag in nearby storage, or prefetch them into a faster cache at

query time using a custom prefetching algorithm. Here we propose an encrypted multi-map variant based on Pibase [30] (Sec. 5.2.2) which reveals the colocation tags for each tuple in the multi-map.

Legacy-friendly colocation. One way to leverage the colocation tags revealed by our Pibase variant is to implement a custom prefetching algorithm in the encrypted database system to pre-load parts of the encrypted structures under the same colocation tag into faster cache during the Query algorithm. However this customization breaks legacy-friendliness. So we also consider how to achieve legacy-friendliness for colocation, by additionally requiring the colocation to work with emulation. Instead of engineering custom prefetching algorithms, we rely on the relational model alone to convey the colocation information of the emulated structures to the underlying database. This means colocation takes emulated structures, examines the revealed colocation tags, and reorganizes the parts of the emulated structures based on the same colocation tag into the same row, column or table. As such, the underlying relational database can view the colocated encrypted structures as tables whose rows and columns reflect the colocation of the plaintext relational model. One key step towards legacy-friendly colocation is the emulation of the Pibase variant (Sec. 5.2.2).

Security. Colocation requires us to increase the leakage of PKFK with a new set of patterns called the colocation patterns. This set of patterns is similar to access patterns (e.g. which row, column or table is accessed by the query). However, although the server may learn which ciphertexts in the encrypted structures are likely to be accessed together, such information can only help the server to infer the dimension of each plaintext table. This is because every value in a plaintext table always colocates with the same number of values row-wise or column-wise due to the structural constraint of a table. Therefore revealing such colocation patterns through STE at most leaks the table dimensions. One subtlety arises is that the table dimensions leaked can be combined with other leakage to deduce more information. For example in PKFK, the table dimensions can be used to calculate the per-table JoinSet size (i.e. the total number of attribute pairs that can be joined), and the per-table FilterSet size (i.e. the total number of attributes that can be filtered). More details are discussed in Sec. 5.2.3.

Examples. An example of Pibase variant after emulation for the row representation $\mathsf{EMM}^{(\mathsf{R})}$ and the filter representation $\mathsf{EMM}^{(\mathsf{S})}$ is shown in Figure 5.7. The colocation tags are highlighted in boxed texts. Note that the rows should have been permuted per the requirement of history independence [30]. The server can instrument a custom prefetching algorithm, for example when processing the select token stk for $\mathbf{T}.\mathsf{att} = Q$, the server can predict the row token accesses for the same table under the colocation tag for $\mathbf{T}$, thereby caching the relevant rows in $\mathbf{T}^{(\mathsf{R})}$ (i.e. the first three) into faster cache.

However the custom prefetching algorithm requires changes to the database systems. In order to achieve legacy friendliness with colocation, the server can instead store the parts of the encrypted structures according to the colocation tags into the same table. This way the relational model conveys the locality information to the server, thereby leveraging the existing locality optimization in the database system. Figure 5.8 shows such an example. The emulated structures for $\mathsf{EMM}^{(\mathsf{R})}$ and $\mathsf{EMM}^{(\mathsf{S})}$ are reorganized into two tables $\mathbf{T}_{\mathsf{co}}$ and $\mathbf{T}'_{\mathsf{co}}$, corresponding to the two distinct colocation tags for $\mathbf{T}$ and $\mathbf{T}'$. The parts in $\mathsf{EMM}^{(\mathsf{R})}$ and $\mathsf{EMM}^{(\mathsf{S})}$ under the same tags for rows and columns are also stored as rows and columns.

5.2.2 Variant of the Pibase Encrypted Multi-Map

In the following, we detail our new variant of Pibase [30]. This variant is an encryption scheme for multi-map structures that have the following structure: every label/tuple pair $(\ell, \mathbf{v})$ can be written as

$$\Big(\ell, ((v_{1,1}, v_{v_{1,2}}), \cdots, (v_{\#\mathbf{v},1}, v_{v_{\#\mathbf{v},2}}))\Big).$$

That is, every tuple is composed of multiple pairs. This variant works similarly to Pibase except that instead of encrypting every value in the tuple, it only encrypts the second value of every pair composing the tuple. We detail this construction in Figure 5.11.

Efficiency and security analysis. This variant of Pibase has the same storage overhead and query complexity of Pibase. The main difference is the setup leakage. In particular, Pibase only reveals the number of pairs N composing the multi-map while this variant reveals the first value of

$\mathbf{T}^{(\mathsf{R})} = \mathsf{Emu}_{\mathsf{Pibase}}(\mathsf{EMM}^{(\mathsf{R})})$

label	value
$F(K_{\chi_1,1},1)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}),F_{K_{\mathsf{co}}}(\chi_1)),}\ E(K_{(\chi_1),2},E_{K_E}(Q)\|E_{K_E}(A))$
$F(K_{\chi_2,1},1)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}),F_{K_{\mathsf{co}}}(\chi_2)),}\ E(K_{(\chi_2),2},E_{K_E}(R)\|E_{K_E}(A))$
$F(K_{\chi_3,1},1)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}),F_{K_{\mathsf{co}}}(\chi_3)),}\ E(K_{(\chi_3),2},E_{K_E}(Q)\|E_{K_E}(B))$
$F(K_{\chi'_1,1},1)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}'),F_{K_{\mathsf{co}}}(\chi'_1)),}\ E(K_{\chi'_1,2},E_{K_E}(A))$
$F(K_{\chi'_2,1},1)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}'),F_{K_{\mathsf{co}}}(\chi'_2)),}\ E(K_{\chi'_2,2},E_{K_E}(B))$
$F(K_{\chi'_3,1},1)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}'),F_{K_{\mathsf{co}}}(\chi'_3)),}\ E(K_{\chi'_3,2},E_{K_E}(A))$

$\mathbf{T}^{(\mathsf{S})} = \mathsf{Emu}_{\mathsf{Pibase}}(\mathsf{EMM}^{(\mathsf{S})})$

label	value
$F(K_{(\mathsf{att},A),1},1)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}),F_{K_{\mathsf{co}}}(\chi_1),F_{K^{(\mathsf{S})}_{\mathsf{co}}}(\chi_2)),}\ E(K_{(\mathsf{att},A),2},K_{\mathbf{T},1})$
$F(K_{(\mathsf{att},A),1},2)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}),F_{K_{\mathsf{co}}}(\chi_2),F_{K^{(\mathsf{S})}_{\mathsf{co}}}(\chi_2)),}\ E(K_{(\mathsf{att},A),2},K_{\mathbf{T},2})$
$F(K_{(\mathsf{att},B),1},1)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}),F_{K_{\mathsf{co}}}(\chi_3),F_{K^{(\mathsf{S})}_{\mathsf{co}}}(\chi_2)),}\ E(K_{(\mathsf{att},B),2},K_{\mathbf{T},3})$
$F(K_{(\mathsf{att}_P,Q),1},1)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}),F_{K_{\mathsf{co}}}(\chi_1),F_{K^{(\mathsf{S})}_{\mathsf{co}}}(\chi_1)),}\ E(K_{(\mathsf{att}_P,Q),2},K_{\mathbf{T},1})$
$F(K_{(\mathsf{att}_P,Q),1},2)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}),F_{K_{\mathsf{co}}}(\chi_2),F_{K^{(\mathsf{S})}_{\mathsf{co}}}(\chi_1)),}\ E(K_{(\mathsf{att}_P,Q),2},K_{\mathbf{T},3})$
$F(K_{(\mathsf{att}_P,R),1},1)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}),F_{K_{\mathsf{co}}}(\chi_3),F_{K^{(\mathsf{S})}_{\mathsf{co}}}(\chi_1)),}\ E(K_{(\mathsf{att}_P,R),2},K_{\mathbf{T},2})$
$F(K_{(\mathsf{att}',A),1},1)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}'),F_{K_{\mathsf{co}}}(\chi'_1),F_{K^{(\mathsf{S})}_{\mathsf{co}}}(\chi'_1)),}\ E(K_{(\mathsf{att}',1),2},K_{\mathbf{T}',1})$
$F(K_{(\mathsf{att}',A),1},2)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}'),F_{K_{\mathsf{co}}}(\chi'_2),F_{K^{(\mathsf{S})}_{\mathsf{co}}}(\chi'_1)),}\ E(K_{(\mathsf{att}',2),2},K_{\mathbf{T}',3})$
$F(K_{(\mathsf{att}',B),1},1)$	$\boxed{(F_{K_{\mathsf{co}}}(\mathbf{T}'),F_{K_{\mathsf{co}}}(\chi'_3),F_{K^{(\mathsf{S})}_{\mathsf{co}}}(\chi'_1)),}\ E(K_{(\mathsf{att}',3),2},K_{\mathbf{T}',2})$

Figure 5.7: An example of emulation for the Pibase variant construction (Sec. 5.2.2) for $\mathsf{EMM}^{(\mathsf{R})}$ and $\mathsf{EMM}^{(\mathsf{S})}$ in Figure 4.7. The boxed texts are the colocation tags. In general the rows should have been permuted due to history-indepence requirement.

every pair composing all tuples of the multi-map. More precisely, the setup leakage is equal to

$$\mathcal{L}_S(\mathsf{MM}) = \Big\{ v : \exists \ell \in \mathbb{L}_{\mathsf{MM}}\ s.t.\ (v,\cdot) \in \mathsf{MM}[\ell] \Big\}.$$

Observe that if the first values of all pairs are values generated uniformly at random from $\{0,1\}^k$, then the setup leakage of the Pibase variant will be equal to N which is then equivalent to the standard Pibase.

$\mathbf{T}_{\mathsf{co}}$

$\mathtt{label}^{(\mathsf{S})}_{\mathsf{att}}$	$\mathtt{value}^{(\mathsf{S})}_{\mathsf{att}}$	$\mathtt{label}^{(\mathsf{S})}_{\mathsf{att}_P}$	$\mathtt{value}^{(\mathsf{S})}_{\mathsf{att}_P}$	$\mathtt{label}^{(\mathsf{R})}$	$\mathtt{value}^{(\mathsf{R})}$
$F\big(K_{(\mathsf{att},A),1},1\big)$	$E\big(K_{(\mathsf{att},A),2},K_{\mathbf{T},1}\big)$	$F\big(K_{(\mathsf{att}_P,Q),1},1\big)$	$E\big(K_{(\mathsf{att}_P,Q),2},K_{\mathbf{T},2}\big)$	$F\big(K_{\chi_1,1},1\big)$	$E\big(K_{\chi_1,2},E_{K_E}(Q)\|E_{K_E}(A)\big)$
$F\big(K_{(\mathsf{att},A),1},2\big)$	$E\big(K_{(\mathsf{att},A),2},K_{\mathbf{T},2}\big)$	$F\big(K_{(\mathsf{att}_P,Q),1},2\big)$	$E\big(K_{(\mathsf{att}_P,Q),2},K_{\mathbf{T},3}\big)$	$F\big(K_{\chi_2,1},1\big)$	$E\big(K_{\chi_2,2},E_{K_E}(R)\|E_{K_E}(A)\big)$
$F\big(K_{(\mathsf{att},B),1},1\big)$	$E\big(K_{(\mathsf{att},B),2},K_{\mathbf{T},3}\big)$	$F\big(K_{(\mathsf{att}_P,R),1},1\big)$	$E\big(K_{(\mathsf{att}_P,R),2},K_{\mathbf{T},2}\big)$	$F\big(K_{\chi_3,1},1\big)$	$E\big(K_{\chi_3,2},E_{K_E}(Q)\|E_{K_E}(B)\big)$

$\mathbf{T}'_{\mathsf{co}}$

$\mathtt{label}^{(\mathsf{S})}_{\mathsf{att}'}$	$\mathtt{value}^{(\mathsf{S})}_{\mathsf{att}'}$	$\mathtt{label}^{(\mathsf{R})}$	$\mathtt{value}^{(\mathsf{R})}$
$F\big(K_{(\mathsf{att}',A),1},1\big)$	$E\big(K_{(\mathsf{att}',1),2},K_{\mathbf{T}',1}\big)$	$F\big(K_{\chi'_1,1},1\big)$	$E\big(K_{\chi'_1,2},E_{K_E}(A)\big)$
$F\big(K_{(\mathsf{att}',A),1},2\big)$	$E\big(K_{(\mathsf{att}',2),2},K_{\mathbf{T}',3}\big)$	$F\big(K_{\chi'_2,1},1\big)$	$E\big(K_{\chi'_2,2},E_{K_E}(B)\big)$
$F\big(K_{(\mathsf{att}',B),1},1\big)$	$E\big(K_{(\mathsf{att}',3),2},K_{\mathbf{T}',2}\big)$	$F\big(K_{\chi'_3,1},1\big)$	$E\big(K_{\chi'_3,2},E_{K_E}(A)\big)$

Figure 5.8: An example of the server leveraging colocation tags in Figure 5.7 to increase locality by reorganizing storage format. Each row coordinates for $\mathbf{T}$ is denoted as χ_i and for $\mathbf{T}'$ as χ'_i. The parts of emulated structures are stored together corresponding to the same row, column or table as revealed by the colocation tags.

Emulation

A SQL emulator. The emulator for the Pibase variant is similar to the emulator of Pibase (Sec. 5.1.1, Fig. 5.1). In particular, Reform is exactly the same, while Reshape differs only on the number of columns. In particular, instead of having a table composed only of two columns, we have a table composed of three. The additional column holds the value in clear in every entry of the dictionary. Recall that an entry in the Pibase variant has the form $(\ell,(v,\mathrm{ct}))$ where v is a value in clear.

Efficiency. The efficiency of this emulator is exactly the same as the one of the Pibase SQL emulator.

5.2.3 Security Trade-off

Our STE-based colocation approach introduces new leakage, the colocation pattern, so that the server can increase locality by (1) reorganizing storage and/or (2) implementing prefetching. Colocation therefore comes with a price: it weakens the security of PKFK. So it is important to

- $\mathsf{Setup}(1^k, \mathsf{DB})$:
  1. initialize a SET
  2. initialize a collection of multi-maps
  $$\mathsf{MM}^{(\mathsf{R})}, \mathsf{MM}^{(\mathsf{S})}, \mathsf{MM}^{(\mathsf{XJ})}$$
  3. sample keys $K_E, K_F, K_S, K_C, K_G, \boxed{K_{\mathsf{co}}, K^S_{\mathsf{co}}, K^{(\mathsf{S})}_{\mathsf{co}}, K^{(\mathsf{XJ})}_{\mathsf{co}}} \xleftarrow{\$} \{0,1\}^k$;
  4. **(Row rep.)**
     a. for each table $\mathbf{T} \in \mathsf{DB}$, for each row $\mathbf{r} = (r_1, \cdots, r_{\#\mathbf{r}}) \in \mathbf{T}$,
     $$\mathsf{MM}^{(\mathsf{R})}\big[\chi(\mathbf{r})\big] := \Big(\boxed{\big(F_{K_{\mathsf{co}}}(\mathsf{tbl}(\mathbf{T})), F_{K_{\mathsf{co}}}(\chi(\mathbf{r}))\big)}, \mathsf{SKE.Enc}_{K_E}(r_1) \| \cdots \| \mathsf{SKE.Enc}_{K_E}(r_{\#\mathbf{r}})\Big)$$
     b. compute
     $$\big(K^{(\mathsf{R})}, \mathsf{EMM}^{(\mathsf{R})}\big) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}^{(\mathsf{R})}\big)$$
  5. **(Filter rep.)** for each table $\mathbf{T} \in \mathsf{DB}$,for each column $\mathbf{c} \in \mathsf{FilterSet}(\mathbf{T})$,
     a. for each unique value $v \in \mathbf{c}$,
        i. for each row $\mathbf{r} \in \mathbf{T}_{\mathbf{c}=v}$, compute the row token
        $$\mathsf{rtk}_{\mathbf{r}} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\big(K^{(\mathsf{R})}, \chi(\mathbf{r})\big)$$
        ii. set
        $$\mathsf{MM}^{(\mathsf{S})}\big[\chi(\mathbf{c})\|v\big] := \Big(\boxed{\big(F_{K_{\mathsf{co}}}(\mathsf{tbl}(\mathbf{T})), F_{K_{\mathsf{co}}}(\chi(\mathbf{r})), F_{K^{(\mathsf{S})}_{\mathsf{co}}}(\chi(\mathbf{c}))\big)}, \mathsf{rtk}_{\mathbf{r}}\Big)_{\mathbf{r} \in \mathbf{T}_{\mathbf{c}=v}}$$
     b. compute
     $$\big(K^{(\mathsf{S})}, \mathsf{EMM}^{(\mathsf{S})}\big) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}^{(\mathsf{S})}\big)$$
  6. **(Partial filter rep.)** for each table $\mathbf{T} \in \mathsf{DB}$, for each column $\mathbf{c} \in \mathbf{T}$,
     a. for each unique value $v \in \mathbf{c}$,
        i. compute $K_v \leftarrow F(K_S, \chi(\mathbf{c})\|v)$
        ii. set for each $\mathbf{r} \in \mathbf{T}_{\mathbf{c}=v}$,
        $$\mathsf{SET} := \mathsf{SET} \bigcup \Big\{\boxed{\big(F_{K_{\mathsf{co}}}(\mathsf{tbl}(\mathbf{T})), F_{K_{\mathsf{co}}}(\chi(\mathbf{r})), F_{K^S_{\mathsf{co}}}(\chi(\mathbf{c}))\big),} H(K_v \| \mathsf{rtk})\Big\}$$
        where $\mathsf{rtk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}(K^{(\mathsf{R})}, \chi(\mathbf{r}))$

(Continued in Fig. 5.10)

Figure 5.9: Colocation in PKFK by modifying Setup with the colocation tags shown as boxed texts.

(Continued from Fig. 5.9)

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

  6. **(Join rep.)**
     a. for each table $\mathbf{T} \in \mathsf{DB}$, for each column pairs $(\mathbf{c}, \mathbf{c}') \in \mathsf{JoinSet}(\mathsf{DB})$ where $\mathsf{tbl}(\mathbf{c}) = \mathbf{T}$,
        i. for each row $\mathbf{r}$ in table $\mathbf{T}$,
           1. compute the row token

$$\mathsf{rtk}_{\mathbf{r}} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\big(K^{(\mathsf{R})}, \chi(\mathbf{r})\big)$$

           2. compute the surrogate for cell $v = \mathbf{r}[\mathbf{c}]$

$$g_v \leftarrow F\big(K_G, \chi(\mathbf{c}) \| \chi(\mathbf{c}') \| v\big)$$

           3. **(filtered join)** for each correlated filter column $\bar{\mathbf{c}} \in \mathsf{FilterSet}(\mathbf{T})$ where $\mathbf{T} = \mathsf{tbl}(\mathbf{c})$,
              a. compute the constraint for the filter value $w = \mathbf{r}[\bar{\mathbf{c}}]$

$$C_w \leftarrow F\big(K_C, \chi(\mathbf{c}) \| \chi(\mathbf{c}') \| \chi(\bar{\mathbf{c}}) \| w\big)$$

              b. set

$$\mathsf{MM}^{(\mathsf{XJ})}\big[F(C_w, \mathsf{rtk}_{\mathbf{r}})\big] := \Big(\boxed{\big(F_{K_{\mathsf{co}}}(\mathsf{tbl}(\mathbf{T})), F_{K_{\mathsf{co}}}(\chi(\mathbf{r})), F_{K_{\mathsf{co}}^{(\mathsf{XJ})}}(\chi(\mathbf{c}))\big),}(\mathsf{rtk}_{\mathbf{r}}, g_v)\Big)$$

        ii. **(unfiltered join)** compute the null constraint for the unfiltered join

$$C_\perp = F\big(K_C, \chi(\mathbf{c}) \| \chi(\mathbf{c}') \| \chi(\mathbf{c}) \| \perp\big)$$

        iii. set

$$\mathsf{MM}^{(\mathsf{XJ})}\big[F(C_\perp, 0)\big] := \Big(\boxed{\big(F_{K_{\mathsf{co}}}(\mathsf{tbl}(\mathbf{T})), F_{K_{\mathsf{co}}}(\chi(\mathbf{r})), F_{K_{\mathsf{co}}^{(\mathsf{XJ})}}(\chi(\mathbf{c}))\big),}\mathsf{rtk}_{\mathbf{r}}, g_v\Big)_{\mathbf{r} \in \mathbf{T}, v = \mathbf{r}[\mathbf{c}]}$$

     b. compute

$$\big(K^{(\mathsf{XJ})}, \mathsf{EMM}^{(\mathsf{XJ})}\big) \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Setup}\big(1^k, \mathsf{MM}^{(\mathsf{XJ})}\big)$$

  7. output

$$K = \big(K_E, K_F, K_S, K_C, K_G, \boxed{K_{\mathsf{co}}, K_{\mathsf{co}}^S, K_{\mathsf{co}}^{(\mathsf{S})}, K_{\mathsf{co}}^{(\mathsf{XJ})},} K^{(\mathsf{R})}, K^{(\mathsf{S})}, K^{(\mathsf{XJ})}\big)$$

     and

$$\mathsf{EDB} = \big(\mathsf{SET}, \mathsf{EMM}^{(\mathsf{R})}, \mathsf{EMM}^{(\mathsf{S})}, \mathsf{EMM}^{(\mathsf{XJ})}\big)$$

Figure 5.10: Colocation in PKFK by modifying Setup with the colocation tags shown as boxed texts.

Let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme, $F : \{0,1\}^k \times \{0,1\}^\star \to \{0,1\}^m$ be a pseudo-random function. Consider the MM encryption scheme $\mathsf{Pibase} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query})$ defined as follows:

- $\mathsf{Setup}(1^k, \mathsf{MM})$:
  1. initialize an empty dictionary DX and sample a key $K \xleftarrow{\$} \{0,1\}^k$;
  2. for every label $\ell \in \mathbb{L}_{\mathsf{MM}}$,
     a. compute $K_1 \| K_2 \leftarrow F_K(\ell)$;
     b. initialize a counter count;
     c. for every pair (v_1, v_2) in $\mathsf{MM}[\ell]$,
        – compute $e = F_{K_1}(\mathsf{count})$ and $\mathrm{ct} = \mathsf{SKE.Enc}(K_2, v_2)$;
        – set $\mathsf{DX}[e] = (v_1, \mathrm{ct})$;
        – increment count;
  3. output the key K and $\mathsf{EMM} = \mathsf{DX}$.
- $\mathsf{Token}(K, \ell)$:
  1. compute $K_1 \| K_2 = F_K(\ell)$;
  2. output $\mathsf{tk} = (K_1, K_2)$.
- $\mathsf{Query}(\mathsf{EMM}, \mathsf{tk})$:
  1. parse EMM as DX and tk as (K_1, K_2);
  2. initialize an empty set Result and counter count;
  3. while $\mathsf{out} \neq \perp$,
     a. compute $e = F_{K_1}(\mathsf{count})$;
     b. compute $\mathsf{out} \leftarrow \mathsf{DX}[e]$;
     c. if $\mathsf{out} = (v, \mathrm{ct})$, then add $v, \mathsf{SKE.Dec}(K_2, \mathrm{ct})$ to Result;
     d. increment counter count;
  4. output Result.

Figure 5.11: Pibase variant.

understand its impact.

Setup leakage. There are two cases. Without legacy-friendliness i.e., not using the Pibase variant, the colocated PKFK scheme (Fig. 5.9 and 5.10) with a standard encrypted multi-map instantiation such as [42, 36, 67, 31, 30] does not change the setup leakage of the non-colocated PKFK scheme: it leaks the total number of rows in the database ($|\mathsf{DB}|$), the total number of filter-able attributes in the database ($\#\mathsf{FilterSet}(\mathsf{DB})$), and the total number of join-able attributes in the database ($\#\mathsf{JoinSet}(\mathsf{DB})$),

$$\mathcal{L}_S(\mathsf{DB}) = \Big(|\mathsf{DB}|, \#\mathsf{FilterSet}(\mathsf{DB}), \#\mathsf{JoinSet}(\mathsf{DB})\Big)$$

However to achieve legacy-friendly colocation, we instantiate the colocated the PKFK scheme with the Pibase variant. The setup leakage of the colocated PKFK scheme increases, which amounts to per-table statistics as

$$\mathcal{L}_S(\mathsf{EDB}_{\mathsf{co}}) = \Big(\#\{\mathbf{T} \in \mathsf{DB}\}, \Big(|\mathbf{T}|, \#\mathsf{FilterSet}(\mathbf{T}), \#\mathsf{JoinSet}(\mathbf{T})\Big)_{\mathbf{T}\in\mathsf{DB}}\Big)$$

This can be seen as the union of per-structure setup leakage:

- $\mathsf{MM}^{(\mathsf{R})}$ leaks the total number of tables ($\#\mathbf{T}$) and the total number of rows per table ($|\mathbf{T}|$);
- additionally $\mathsf{MM}^{(\mathsf{S})}$ leaks the number of filter-able attributes per table ($\#\mathsf{FilterSet}(\mathbf{T})$);
- additionally $\mathsf{MM}^{(\mathsf{XJ})}$ leaks the number of join-able attributes that involve a table ($\#\mathsf{JoinSet}(\mathbf{T})$);
- finally SET leaks as much as $\mathsf{MM}^{(\mathsf{S})}$.

Therefore for setup the colocation patterns entail refining the per-database dimensional statistics to be per-table in the leakage profile.

Query leakage. The query leakage of the colocated PKFK scheme is increased by the colocation patterns. Intuitively speaking, the colocation patterns inform the server whether two query

responses on the same or different encrypted structures belong to the same plaintext row, column or table. This means that the server learns the access pattern of each plaintext row, column and table. Next we describe the concrete leakage profile of PKFK when instantiated with a (almost) leakage-free data structure encryption [65]. The leakage profile of the *colocated* PKFK can be viewed as a modification based on the leakage profile of the *non-colocated* PKFK (Sec. 4.5), where the additional component is mainly the access pattern of rows, columns and tables.

Selection pattern. The *leaf filter* leaks the following additional information in the selection pattern: how frequently the filtered table, the attribute, and the pair of attribute value has been accessed in past queries. We do not consider colocation of the SET structure in this work, so the *internal filter* does not leak more.

Join pattern. The *filtered join* leaks the following additional information in the join pattern: how frequently the triplet of filter attribute and join attributes has been accessed in past queries, how frequently each joined table has been accessed in past queries. For *unfiltered join*, instead of the aforementioned triplet it is the access frequency on the pair of join attributes.

Project pattern. The projection leaks the following additional information in the project pattern: how frequently each projected table has been accessed in past queries.

Constant and cross product pattern. These patterns stay the same, because constant and cross product do not involve encrypted structures.

Chapter 6

Evaluation

In this section, we evaluate how OPX (Ch. 3) and PKFK (Ch. 4) performs in practice. To do this, we implemented OPX and PKFK and integrated it into an encrypted relational database system `KafeDB` (Ch. 2). Its code and a design are publicly available [10, 11]. We also implemented SPX [63] and similarly integrated them into `KafeDB`. In summary, we compared OPX and PKFK with: SPX [63] (STE-based), `CryptDB` [84] and `Monomi` [94] (PPE-based). In particular, we assessed the following efficiency metrics: (1) setup time, (2) query efficiency, and (3) storage efficiency. Our evaluation demonstrates that

1. PKFK achieves similar query and storage overhead compared to the PPE-based scheme in `CryptDB`;
2. PKFK improves one to two orders of magnitude over the STE-based schemes SPX and OPX.

Implementation. We implemented PKFK, SPX and OPX in the `KafeDB` system. 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` [54]. The source code is anonymized and can be downloaded at [10]. 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` [80] 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) [8]. Following the typical hardware setting in the research literature [38], we chose to keep the memory higher than the database size to accommodate more complex queries. We used EC2 instance of type t2.2xlarge with 32GB of RAM and 1.2TB of Elastic Block Store for disk storage.

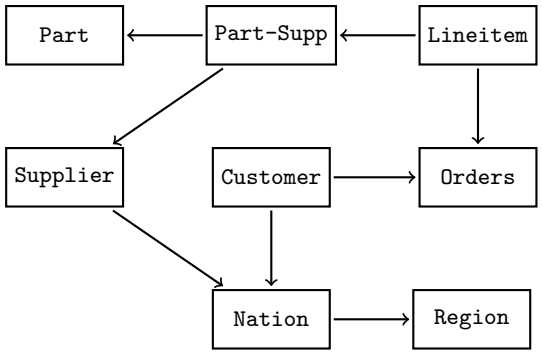
TPC-H Benchmark. For data generation, we use the standard DBMS benchmark Transaction Processing Performance Council H (TPC-H), which models data-driven decision support for business environments centered around a data warehouse scenario. We used the TPC-H benchmark of scale factor 1, which leads to about 8.6 million rows and 4.4GB of data. Each attribute value was sampled uniformly at random from its domain. All queries were run in a uniformly randomized order. The benchmark was first warmed up by executing all the TPC-H queries and discarding the results. The statistics were summarized over 10 runs.

The schema consists of 8 tables that represent real-world entities and relationships such as suppliers and customers, parts and orders. Figure 6.1a illustrates the schema as a directed graph where the nodes are tables, and each arrow represents a many-to-one relationship. In Table 6.1b, we provide a more detailed description of each table, including the number of attributes and the number of rows.

Data generation. We use the TPC-H benchmark of scale factor 1, which leads to about 8.6 million rows and about 4.4GB of data. Each attribute value is sampled uniformly at random from its domain.

Comparisons. For the purpose of this evaluation, we also compare our efficiency numbers to those of CryptDB and Monomi from [94]. We note that the original CryptDB's system [84] only supports 4 out of the 22 TPC-H queries, so the results we recall here are from a modified version of CryptDB in [94] that supports the full TPC-H. We also note that the hardware setup differs

[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	2×10^5
Supplier	7	10×10^3
Part-Supp	7	8×10^5
Customer	8	1.5×10^5
Nation	4	25
Region	3	5
Lineitem	17	6×10^6
Orders	9	1.5×10^6

(b) The cardinality is for scale factor 1.

Figure 6.1: Description of the TPC-H database schema [41].

slightly in [94] where most noticeably the authors used a machine with slightly smaller RAM of 24GB compared to the 32GB of RAM we use in our setting.[2] Since the code of [94] 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 implemented the STE schemes SPX [63] and its optimized variant OPX [66] in KafeDB as another baseline to demonstrate the improvement in our new scheme PKFK.

6.1 Query Efficiency

We compared all schemes on TPC-H queries q1-q22. The relative slowdown reflects the multiplicative overhead incurred by each system over the query efficiency of plaintext PostgreSQL. We summarized all results in Figure 6.2. Our benchmark demonstrated that PKFK achieved comparable query efficiency to DTE-based approach such as in CryptDB and Monomi [94]. On the other hand, PKFK outperformed the STE-based precursor SPX [63] and its optimized variant OPX (Ch: 3) for one to two orders of magnitude, due to (1) reduced asymtotic complexity for joins and (2) increased access locality via new techniques such as colocation and join direction optimization.

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×

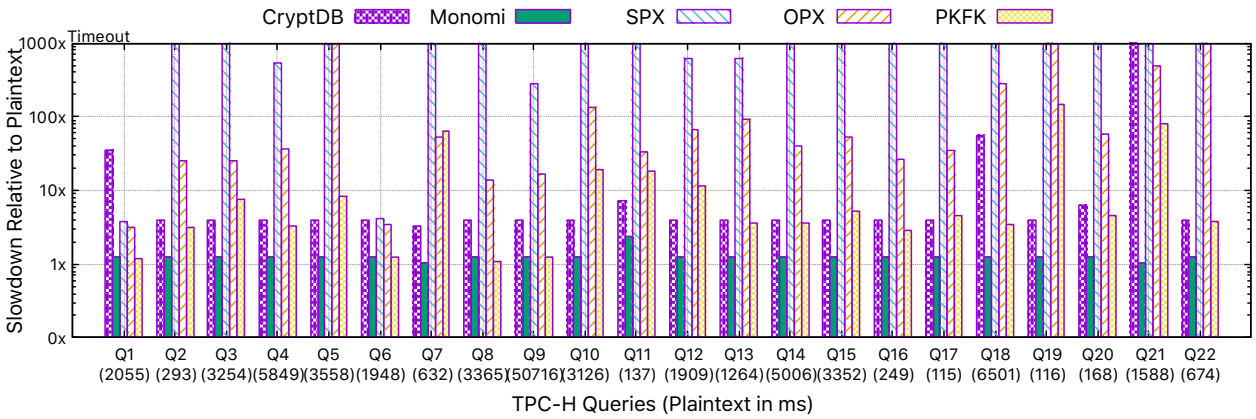
[2]The authors in [94], however, stated that their evaluation numbers were similar across different hardware setups.

	CryptDB[94]	Monomi[94]	SPX [63]	OPX (Ch: 3)	PKFK (Ch: 4)
Median	3.92×	1.24×	1610.5×	45.6×	4.2×
Min	1.04×	1.03×	3.7×	3.2×	1.09×
Max	55.9×	2.33×	1701.9×	1407.9×	147.9×

(a) Summary of slowdown relative to plaintext.

Colocation	Selection Pushdown	Join Ordering
2.92×	19.8×	12.6×

(b) Average improvement in PKFK.



(c) TPC-H query time.

Figure 6.2: TPC-H query efficiency comparison for DTE-based and STE-based schemes.

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.

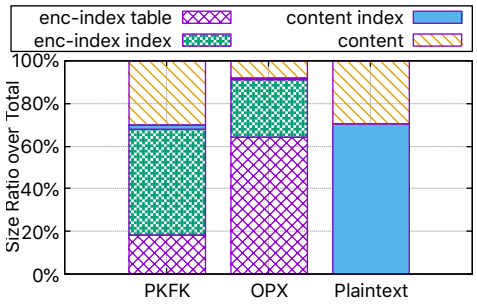
PKFK vs. STE-based approaches. PKFK improves over SPX/OPX on average by over one to two orders of magnitude. The majority of the queries, 16 out of 22, in PKFK incur less than 10× slowdown compared to plaintext PostgreSQL. The slowest query q19 finished a little over 100×. We attribute this improvement to the three novel techniques used in PKFK, namely the reduced complexity for joins, the colocation and the query optimizations.

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 individually enabled. Figure 6.2b summarizes our results. In particular, PKFK with colocation improved 2.92× on average for query time. The effectiveness of colocation is mainly due to the increased locality where a query now operates on common indexes and table rows. Note that SPX and OPX, in order to process a single query, require operating on different indexes and tables which lead to poor locality. The selection pushdown or push-select-through-join rule brought 19.8 average speedup by reducing intermediate data size. Join direction optimization showed 12.6× average speedup by increasing access locality by following the more efficient join ordering recommended by the query optimizer.

6.2 Storage Overhead

System	Size	Setup time
Plaintext	4.442GB	5.99min
CryptDB[94]	4.21×	-
Monomi[94]	1.72×	-
SPX [63]	252.22×	60.17×
OPX	13.17×	10.37×
PKFK	3.63×	8.26×

(a) TPC-H storage and setup time relative to plaintext.



(b) Breakdown of TPC-H storage in percentage.

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

(c) TPC-H storage breakdown in MB.

Figure 6.3: TPC-H storage size and setup time.

We compare the storage overhead across different systems in Figure 6.3. 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 SPX/OPX. We provide below more details about our comparison.

PKFK **vs. DTE-based approaches.** Figure 6.3a shows that PKFK incurs a storage overhead of 3.64×, which is slightly lower than CryptDB's 4.21× 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.** PKFK is about two orders of magnitute smaller in footprint than SPX and about 4× smaller than its optimized variant OPX. This reduction is mainly because PKFK achieves storage complexity linear in table size, whereas both SPX and OPX requires quadratic complexity[3]. Another contributing factor is the new emulation technique that colocate the encrypted indexes in the same table as the encrypted contents to eliminate redundant information.

Storage breakdown. To better understand the storage overhead of PKFK, we provide in Figure 6.3c 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 emulated table for encrypted indexes or structures;
- (`enc-index index`): the plaintext indexes created on top of the encrypted index table;
- (`content table`): the tables containing the content of the database;
- (`content index`): the indexes created on top of the content 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 OPX, 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 colocated encrypted indexes do not need to store duplicate cell values such as the

[3] OPX can rely on database technique such as third normal form to trade off this quadratic blowup for additional query overhead.

row token identifier. Moreover, with the colocation, sharing indexes becomes possible. Note that some indexes become redundant and with PKFK, we can afford removing them without hurting the system functionalities or efficiency.

Remarks. In Figure 6.3b, 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 indexes over the content is about the same as that of `PostgreSQL` - around 70%. On the other hand, the ratio for OPX, around 90%, is much more skewed towards encrypted indexes, which signifies a more index-intensive payload.

6.3 Setup Time

Figure 6.3a summarizes that the setup time of PKFK improves about 20% over OPX, 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 colocation of encrypted data indexes with the table contents, which reduces the amount of data to stored and therefore written on disk.

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[2] D. Adkins, A. Agarwal, S. Kamara, and T. Moataz. Encrypted blockchain databases. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 241–254, 2020.

[3] A. Agarwal and S. Kamara. Encrypted distributed hash tables. *IACR Cryptol. ePrint Arch.*, 2019:1126, 2019.

[4] A. Agarwal and S. Kamara. Encrypted distributed hash tables. Technical Report 2019/1126, IACR ePrint Cryptography Archive, 2019.

[5] 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.

[6] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 110–119, 1979.

[7] 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.

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

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

[10] Anonymous. `KafeDB` source code. https://anonymous.4open.science/r/1f937dff-91e1-4a22-9b89-1b3939cfb18f/, 2020.

[11] Anonymous. Encrypted databases: from theory to systems. In *CIDR*, 2021.

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

[13] 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.

[14] G. Asharov, M. Naor, G. Segev, and I. Shahaf. Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In *ACM on Symposium on Theory of Computing (STOC '16)*, 2016.

[15] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 261–272, 2000.

[16] 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.

[17] C. Balkesen, J. Teubner, G. Alonso, and M. Özsu. Main-memory hash joins on modern processor architectures. *IEEE Transactions on Knowledge and Data Engineering*, 27:1754–1766, 2015.

[18] C. Barthels, G. Alonso, T. Hoefler, T. Schneider, and I. Müller. Distributed join algorithms on thousands of cores. *Proc. VLDB Endow.*, 10:517–528, 2017.

[19] 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.

[20] 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.

[21] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security (CCS 2008)*, pages 257–266. ACM, 2008.

[22] L. Blackstone, S. Kamara, and T. Moataz. Revisiting leakage abuse attacks. In *Network and Distributed System Security Symposium (NDSS '20)*, 2020.

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

[24] 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.

[25] 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.

[26] D. Boneh, G. di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Advances in Cryptology – EUROCRYPT '04*, volume 3027 of *Lecture Notes in Computer Science*, pages 506–522. Springer, 2004.

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

[28] R. Bost and P.-A. Fouque. Thwarting leakage abuse attacks against searchable encryption – a formal approach and applicaitons to database padding. Technical Report 2017/1060, IACR Cryptology ePrint Archive, 2017.

[29] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM Conference on Communications and Computer Security (CCS '15)*, pages 668–679. ACM, 2015.

[30] 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.

[31] 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.

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

[33] R. Cattell. Scalable sql and nosql data stores. *Acm Sigmod Record*, 39(4):12–27, 2011.

[34] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *ACM Symposium on Theory of Computing (STOC '77)*, pages 77–90. ACM, 1977.

[35] 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.

[36] 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.

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

[38] 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.

[39] L. Columbus. Roundup of cloud computing forecasts and market estimates. 2018.

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

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

[42] 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.

[43] 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.

[44] A. Eisenberg and J. Melton. Sql: 1999, formerly known as sql3. *ACM Sigmod record*, 28(1):131–138, 1999.

[45] 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.

[46] 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*, pages 123–145. Springer, 2015.

[47] 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.

[48] 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.

[49] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

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

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

[52] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218. IEEE, 1993.

[53] G. Greenwald. *No place to hide: Edward Snowden, the NSA, and the US surveillance state*. Macmillan, 2014.

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

[55] 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.

[56] 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.

[57] 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.

[58] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS '12)*, 2012.

[59] 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.

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

[61] S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sublinear complexity. In *Advances in Cryptology - EUROCRYPT '17*, 2017.

[62] S. Kamara and T. Moataz. Encrypted multi-maps with computationally-secure leakage. *IACR Cryptol. ePrint Arch.*, 2018:978, 2018.

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

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

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

[66] S. Kamara, T. Moataz, S. Zdonik, and Z. Zhao. Opx: An optimal relational database encryption scheme. Technical report, IACR ePrint Cryptography Archive, 2020.

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

[68] P. Koutris, S. Salihoglu, and D. Suciu. Algorithmic aspects of parallel query processing. *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[69] D. Laney et al. 3d data management: Controlling data volume, velocity and variety. *META group research note*, 6(70):1, 2001.

[70] C. S. Legislature. California consumer privacy act of 2018. 2018.

[71] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*, pages 615–629, 2016.

[72] Y. Lindell and B. Pinkas. A proof of security of yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[73] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14:709–730, 2002.

[74] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 659–670, 2004.

[75] P. Mell, T. Grance, et al. The nist definition of cloud computing. 2011.

[76] I. Miers and P. Mohassel. Io-dsse: Scaling dynamic searchable encryption to millions of indexes by improving locality. Cryptology ePrint Archive, Report 2016/830, 2016. http://eprint.iacr.org/2016/830.

[77] I. Mironov, G. Segev, and I. Shahaf. Strengthening the security of encrypted databases: Non-transitive joins. In *Theory of Cryptography Conference*, pages 631–661. Springer, 2017.

[78] T. A. Mohamed Ahmed Abdelraheem and C. Gehrmann. Inference and record-injection attacks on searchable encrypted relational databases. Technical Report 2017/024, 2017. http://eprint.iacr.org/2017/024.

[79] M. Naveed, M. Prabhakaran, and C. Gunter. Dynamic searchable encryption via blind storage. In *IEEE Symposium on Security and Privacy (S&P '14)*, 2014.

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

[81] 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.

[82] 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.

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

[84] 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.

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

[86] H. Salcedo. Google drive, dropbox, box and icloud reach the top 5 cloud storage security breaches list. 2014.

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

[88] R. Satter, J. Donn, and C. Day. Inside story: How russians hacked the democrats' emails. 2017.

[89] D. A. Services. Cloud computing and business intelligence market study. 2020.

[90] G. Shalygina and B. Novikov. Implementing common table expressions for mariadb. In *Second Conference on Software Engineering and Information Management (SEIM-2017)*, pages 12–17, 2017.

[91] 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.

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

[93] M. Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53(4):10–11, 2010.

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

[95] E. Union. Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation). 2016.

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

[97] 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.

[98] 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.

[99] A. Yao. Protocols for secure computations. In *IEEE Symposium on Foundations of Computer Science (FOCS '82)*, pages 160–164. IEEE Computer Society, 1982.

[100] A. Yao. How to generate and exchange secrets. In *IEEE Symposium on Foundations of Computer Science (FOCS '86)*, pages 162–167. IEEE Computer Society, 1986.

[101] H. Zeller and J. Gray. An adaptive hash join algorithm for multiuser environments. In *VLDB*, volume 90, pages 186–197, 1990.

[102] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium*, 2016.