KafeDB: a Structurally-Encrypted Relational Database Management System

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

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

We design and build a new encrypted database management system called KafeDB that addresses all these limitations. KafeDB is based on structured encryption (STE) and, as such, achieves a leakage profile comparable to the ESPADA, Blind Seer and Stealth systems. KafeDB, however, handles a non-trivial subset of SQL which includes queries with joins and projections. In addition, KafeDB is *legacy-friendly*, meaning that it can be deployed on top of *any* relational database management system. We describe and benchmark our KafeDB prototype which is built on top of an unmodified instance of PostgreSQL. We found that for 68% of the TPC-H benchmark queries KafeDB took 1× to 100× longer than a standard/plaintext PostgreSQL server and incurred roughly a 40× overhead in storage.

I. Introduction

Data is being produced, collected and analyzed at unprecedented speed, volume and variety. For all the benefits of "big data", however, the constant occurrences of data breaches have raised serious concerns about the privacy and security of all the data that is being collected and managed, especially when data is sensitive like electronic health record or financial records.

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

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

PPE-based EDBs. The problem of relational EDBs is one of the "holy grails" of database security. It was first explicitly considered by Hacigümüs, Iyer, Li and Mehrotra [32] who described a quantization-based approach which leaks the range within which an item falls. In [42], Popa, Redfield, Zeldovich and Balakrishnan described a system called CryptDB that supports a non-trivial subset of SQL without quantization. CryptDB achieves this in part by making use of property-preserving encryption (PPE) schemes like deterministic and order-revealing (ORE) encryption [2], [13], [14], 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 [8] and commercial systems like SEEED [43] and Microsoft's SQL Server Always Encrypted [24]. While the security of PPE *primitives* had been formally studied by the cryptography community [3], [13], [16], [14], [15], 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 [36]). 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 [31], [26].

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

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

More practical solutions can be achieved using structured encryption (STE) [22] which is a generalization of indexed-based searchable symmetric encryption (SSE) [44], [29], [21], [25]. 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) [25]. This is, roughly speaking, the approach taken by systems such as ESPADA [19], [18], [34], [27], Blind Seer [40], [28] and Stealth [33].[1] We refer to this as *column indexing* and this leads to systems that can handle SQL queries of the form

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

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

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

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., [35] 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 [36]; more specifically, queries of the form

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

which include projects and joins but not ranges. This scheme, called SPX, is asymptotically optimal for a subset of the queries above and (provably) leaks a lot less than known PPE-based solutions like CryptDB. In this work, we propose an extension of this scheme, called OPX, that handles any query of the form above in asymptotically optimal time. We note, however, that both SPX and OPX are only cryptographic constructions and not systems like ESPADA, Blind Seer and Stealth. A third approach to designing relational EDBs is to use trusted hardware like secure coprocessors or Intel SGX. Several systems, most notably TrustedDB [11] and StealthDB [47] take this direction. Though our system could leverage trusted hardware by running our client proxy in an enclave,[2] we do not investigate this direction given the security concerns around SGX.

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

A. *Our Contributions*

In this work, we describe the design and implementation of an STE-based relational EDB system called `KafeDB` that: (1) handles a non-trivial subset of SQL; (2) comes with a rigorous security analysis; and, perhaps most surprisingly, (3) is legacy-friendly.

The OPX **construction.** `KafeDB` is based on a new STE scheme called OPX which is an extension of the SPX construction of Kamara and Moataz [36]. Unlike its predecessor, OPX supports query optimization and handles all conjunctive SQL queries optimally. It does this by using additional encrypted structures that are designed to optimally handle internal 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.

Emulation. A widely-held belief about STE schemes is that they are not legacy-friendly. This is illustrated by the fact that every STE-based EDBs like ESPADA, Blind Seer and Stealth are all custom-built. This is undesirable

[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.

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

in practice because database systems have been designed and optimized over the last 40 years to achieve peak performance and reliability. Indeed, STE's lack of legacy-friendliness has been a major reason for the adoption of PPE-based solutions in practice.

In this work, we introduce and formalize a new technique called *emulation* that makes STE schemes legacy-friendly. At a high-level, an emulator is a set of algorithms that "reshape" an encrypted data structure in such a way that it can be stored and queried as a different data structure. One of the main advantages of emulation is that it does not affect the leakage profile of the encrypted structure. It can, however, affect its storage and query complexity so an important goal when designing emulators is to minimize these overhead.

The introduction of emulation fundamentally changes the landscape of encrypted search and alters our understanding of what is possible. Indeed, it removes the only limitation of STE with respect to PPE, making it comparable to the latter in terms in efficiency and legacy-friendliness but superior in terms of security.

SQL emulators. While emulation is a general technique that can be used to make STE schemes legacy-friendly with respect to any target system, our focus here is on relational databases and on what we refer to as *SQL emulators*. More precisely, these are emulators that reshape encrypted structures as tables and "reformulates" the scheme's tokens as SQL queries. To design a legacy-friendly relational EDB, we construct a SQL emulator for the OPX scheme. This emulator is itself based on two other SQL emulators we introduce for the Pibase EMM construction of Cash et al. [18] and an encrypted set structure used by OPX.

Optimizations. One of the most important components of any DBMS is its query optimizer. Commercial query optimizers are the result of over 40 years of research from the database community and are major reason why real-world DBMSs are so efficient. It follows then that for KafeDB to be competitive at all with commercial systems, it has to support some form of query optimization. KafeDB uses some standard query optimizations like push-select-through-joins but also a set of custom optimizations designed specifically to work over encrypted data. This includes factoring many-to-many relationships and flattening multi-way joins.

Our prototype. We implemented our design, including the emulated variant of OPX, on top of an unmodified PostgreSQL server. The resulting system is called KafeDB and consists of several components. The first is the KafeDB client which includes a *database encryptor* that takes a relational database DB from an application and produces an OPX-encrypted and emulated database EDB'. The database encryptor then sends the emulated EDB to PostgreSQL server for storage. The second component of the KafeDB client is the *query parser* which parses a SQL query Q from the application, generates an optimized query tree and then a token tree from it. It is then used to generate a SQL query Q' that emulates the token tree which is sent to the PostgreSQL server. The PostgreSQL server then executes Q' on EDB' which results in a table of encrypted values which is returned to the KafeDB client who decrypts it before returning it to the application.

Empirical evaluation. We evaluated KafeDB using the TPC-H benchmark [46] and compared its performance to a standard/plaintext instance of PostgreSQL. On 1GB database that consists of 8 million rows and 65 columns, KafeDB took 53 minutes to encrypt, emulate and store the database compared to 319 seconds for PostgreSQL to load the database; a 10× overhead. The total size of the encrypted database was 42GB compared to 4.5GB needed to store the plaintext database and all the associated indices. With respect to query efficiency, we found that when evaluating KafeDB and PostgreSQL on machines with the same amount of memory, 32% of the queries ran 1× to 20× slower, 36% ran 20× to 100× slower and 32% ran more than 100× slower. If, on the other hand, we evaluate the two systems on machines with the same memory to database size ratio, we found that 36% of the queries ran 1× to 20× slower, 45% ran 20× to 100× slower, and only 18% ran more than 100× slower.

We also evaluated the impact of our optimizations. The standard push-select-through-join optmization improved query time by 4× to 53×. Our many-to-many factorization optimization, however, improved query time by at least 2288× and reduced storage by 46×, brining the size of the encrypted database from 2TB to 42GB. Our multi-way join flattening optimization improved query time by 111×.

Limitations & comparison. KafeDB incurs a larger query overhead than some of the other STE-based EDBs. For example, Blind Seer [40] is only 1.3× to 25× slower than MySQL on a database of 10TB and ESPADA reports being 10× slower than a "warm" MySQL [19]. As reported in [33], the Stealth system is 5× to 100× slower than MySQL on a database of 10 million rows, depending on the selectivity of the query. We stress, however, that unlike these three systems, KafeDB is legacy-friendly and handles a non-trivial subset of SQL (i.e., SQL queries that include joins and projections). Currently, our system does not implement dynamism or ranges (these will be added in the future). Like ESPADA, Blind Seer[3] and Stealth, the current version of KafeDB is secure against semi-honest adversaries though we note that malicious behavior only affects correctness and not security.

II. Related Work

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

Encrypted search. Encrypted search was first considered explicitly by Song, Wagner and Perrig in [44] which introduced the notion of searchable symmetric

[3] We note that [28] achieves malicious security against the client with respect to query access control and not against a malicious server.

encryption (SSE). Goh provided the first security definition for SSE and a solution based on Bloom filters with linear search complexity. Chang and Mitzenmacher proposed an alternative security definition and construction, also with linear search complexity. Curtmola et al. introduced and formulated the notion of adaptive semantic security for SSE [25] together with the first sub-linear and optimal-time constructions. Chase and Kamara introduced the notion of structured encryption which generalizes SSE to arbitrary data structures [22].

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 [12] and Conclave [48], 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 [41] and Jana by Galois [9]. 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.

III. 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.

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) [19], [18].

Relational databases. We denote a relational database $\mathsf{DB} = (\mathbf{T}_1, \ldots, \mathbf{T}_n)$, where each $\mathbf{T}_i$ is a two-dimensional array with rows corresponding to an entity (e.g., a customer or an employee) and columns corresponding to attributes (e.g., age, height, salary). For any given attribute, we refer to the set of all possible values that it can take as its *space* (e.g., integers, booleans, strings). We define the *schema* of a table $\mathbf{T}$ to be its set of attributes and denote it $\mathbb{S}(\mathbf{T})$. For a row $\mathbf{r} \in \mathbf{T}_i$, its table identifier $\mathsf{tbl}(\mathbf{r})$ is i and its row rank $\mathsf{rrk}(\mathbf{r})$ is its position in $\mathbf{T}_i$ when viewed as a list of rows. Similarly, for a column $\mathbf{c} \in \mathbf{T}_i^\intercal$, its table identifier $\mathsf{tbl}(\mathbf{c})$ is i and its column rank $\mathsf{crk}(\mathbf{c})$ is its position in $\mathbf{T}_i$ when viewed as a list of columns. For any row $\mathbf{r} \in \mathbf{T}$ and any column $\mathbf{c} \in \mathbf{T}$, we refer to the pairs $\chi(\mathbf{r}) \overset{def}{=} (\mathsf{tbl}(\mathbf{r}), \mathsf{rrk}(\mathbf{r}))$ and $\chi(\mathbf{c}) \overset{def}{=} (\mathsf{tbl}(\mathbf{c}), \mathsf{crk}(\mathbf{c}))$, respectively, as their *coordinates* in DB. For any attribute $\mathsf{att} \in \mathbb{S}(\mathsf{DB})$ and constant a belonging to the attribute's domain, $\mathsf{DB}_{\mathsf{att}=a}$ is the set of rows $\big\{\mathbf{r} \in \mathsf{DB} : \mathbf{r}[\mathsf{att}] = a\big\}$.

SQL. In this work, we focus on the class of *conjunctive SQL* queries, 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 a *constant predicate* whereas if X_i is an attribute, then the predicate $\mathsf{att}_i = X_i$ is called a *join predicate.* A formula is a Boolean expression composed of constant and join predicates. We use standard relational algebra notation and denote the filtering operator by σ, the projection operator by π, the rename operator by ρ, the θ-join operator by $\underset{\theta}{\bowtie}$ and the cross join operator by $\times$.

Basic cryptographic primitives. A private-key encryption scheme is a set of three polynomial-time algorithms $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ such that Gen is a probabilistic algorithm that takes a security parameter k and returns a secret key K; Enc is a probabilistic algorithm that takes a key K and a message m and returns a ciphertext c; Dec is a deterministic algorithm that takes a key K and a ciphertext c and returns m if K was the key under which c was produced. Informally, a private-key encryption scheme is secure against chosen-plaintext attacks (CPA) if the ciphertexts it outputs do not reveal any partial information about the plaintext even to an adversary that can adaptively query an encryption oracle. We say a scheme is random-ciphertext-secure against chosen-plaintext attacks (RCPA) if the ciphertexts it outputs are computationally indistinguishable from random even to an adversary that can adaptively query an encryption oracle.[4] 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.

A. Structured Encryption

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

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

The security of STE is formalized using "leakage-parameterized" definitions following [25], [22]. 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 [25], [22] and for snapshot adversaries in [5].[5]

The leakage profile of a scheme captures the information an adversary learns about the data and/or the queries. 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 [25], [22], [5] for more details.

Definition III.1 (Security vs. persistent adversary (Informal)). *Let* $\Lambda = \big(\mathcal{L}_\mathsf{S}, \mathcal{L}_\mathsf{Q}, \mathcal{L}_\mathsf{U}\big) = \big(\mathsf{patt}_1, \mathsf{patt}_2, \mathsf{patt}_3\big)$ *be a leakage profile. An encrypted database algorithm* STE *is* Λ*-secure if there exists a* PPT *simulator that, given* $\mathsf{patt}_1(\mathsf{DB})$ *for an adversarially-chosen database* DB*,* $\mathsf{patt}_2(\mathsf{DB}, q_1, \ldots, q_t)$ *for adaptively-chosen queries* $(q_1, \ldots, q_t)$*, and* $\mathsf{patt}_3(\mathsf{DB}, u_1, \ldots, u_t)$ *for adaptively-chosen updates* $(u_1, \ldots, u_t)$ *can simulate the view of any* PPT *adversary. Here, the view includes the encrypted data structure and the transcript of the queries.*

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 [25], [38], [18], [18], [20], [45], [17], [37]. Encrypted dictionaries can be obtained from any encrypted multi-map since the former is just an encrypted multi-map with single-item tuples.

IV. The OPX Scheme

We describe the OPX scheme which extends the SPX construction of [36]. It uses as building blocks a response-revealing multi-map encryption scheme Σ_MM, a variant of the Pibase construction of Cash et al. [18] we denote Σ^π_MM and a pseudo-random function F. In Appendix B, we provide a concrete example that walks through our indexing approach.

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

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

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

It then computes

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

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

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

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

[5] 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.

It then computes

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

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

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

and sets

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

It then computes

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

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

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

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

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

and

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

and sets

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

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

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

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

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

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

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

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 $\mathbf{c}, \mathbf{c}' \in \mathsf{DB}^\intercal$ that have the same domain, it initiates an empty tuple $\mathbf{t}$ that it populates as follows. For all rows $\mathbf{r}_i$ and $\mathbf{r}_j$ in column $\mathbf{c}$ and $\mathbf{c}'$, respectively, that verify

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

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

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

and

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

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

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

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 $K = (K_1, K_R, K_C, K_V, \{K_{\mathbf{c}}\}_{\mathbf{c}\in\mathsf{DB}^\intercal}, \quad K_F, \{K_{\mathbf{c},\mathbf{c}'}\}_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^\intercal})$ and $\mathsf{EDB} = (\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, (\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^\intercal}$, 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.[6] 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 $(\mathsf{rtk}, \mathsf{pos})$ where
$$\mathsf{rtk} \leftarrow F_{K_F}\Big(\chi(\mathsf{att})\|a\Big),$$
and pos denotes the position of att in $\mathbf{R_{in}}$.
- **(leaf join):** if N is a leaf node of form $\mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{T}_2$ then set the corresponding node in TT to $(\mathsf{jtk}, \mathsf{pos})$ where
$$\mathsf{jtk} \leftarrow \Sigma_{\mathsf{MM}}.\mathsf{Token}\Big(K_{\mathsf{att}_1}, \Big\langle \chi(\mathsf{att}_1), \chi(\mathsf{att}_2)\Big\rangle\Big),$$
and pos denotes the positions of att_1 in $\mathbf{R_{in}}$.
- **(internal join):** if N is an internal node of form $\mathbf{T} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R_{in}}$, then set the corresponding node in TT to $(\mathsf{etk}, \mathsf{pos}_1, \mathsf{pos}_2)$ where
$$\mathsf{etk} := K_{\mathsf{att}_1,\mathsf{att}_2},$$
and pos_1, pos_2 denote the positions of att_1 and att_2 in $\mathbf{R_{in}}$, respectively.
- **(intermediate internal join):** if N is an internal node of form $\mathbf{R}^{(l)}_{\mathbf{in}} \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{R}^{(r)}_{\mathbf{in}}$ then set the corresponding node in TT to $(\mathsf{pos}_1, \mathsf{pos}_2)$ where pos_1 and pos_2 are the column positions of att_1 and att_2 in $\mathbf{R}^{(l)}_{\mathbf{in}}$ and $\mathbf{R}^{(r)}_{\mathbf{in}}$, respectively.

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

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

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

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

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

where $\mathtt{pos}_i$ is the column position of att_i in $\mathbf{R_{in}}$.
- **(leaf scalars):** if N is a node of form $[a]$ then set the corresponding node to $[\mathsf{Enc}_{K_1}(a)]$.
- **(cross product):** if N is a node of form $\times$ then keep it with no changes.

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

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

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

and sets $\mathbf{R_{out}} := (\mathsf{rtk}_1, \cdots, \mathsf{rtk}_s)$.
- **(internal constant select):** if N has form $(\mathsf{rtk}, \mathtt{pos})$, then for all rtk in $\mathbf{R_{in}}$ in the column at position $\mathtt{pos}$, if

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

then it removes the row from $\mathbf{R_{in}}$. Finally, it sets $\mathbf{R_{out}} := \mathbf{R_{in}}$.
- **(leaf join):** if N has form $(\mathsf{jtk}, \mathtt{pos})$, then it computes

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

and sets

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

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

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

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

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

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

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

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

V. Analysis of OPX

We now discuss the security of OPX. Given a database DB and a query tree QT, OPX reveals a leakage tree LT that has the same structure as QT but where each node reveals a leakage pattern. Because the leakage profile can get relatively complex depending on the query we only give high-level intuition here and provide a detailed and formal analysis in the full version of this work [6].

We note that the overview we provide here assumes that the scheme's underlying multi-map encryption schemes are instantiated with the current state-of-the-art optimal-time and space constructions [25], [22], [38], [19], [18]. If the underlying schemes are instantiated with zero-leakage or "almost zero-leakage" constructions like the ones found in [37], OPX leaks considerably less at the cost of increased query time. In the full version of this work [6], we describe OPX's leakage profile when using such constructions.

Selections. During a selection operation, the adversary can learn the number of matching rows, the frequency with which a particular row has been accessed and its size. If many queries are performed on the same table and column, then the adversary can eventually build a histogram over the column. Now, depending on the composition of the query tree an adversary could get a more detailed histogram if additional *internal* selections are performed on the same attribute.

Joins. Join operations reveal the most leakage. The adversary learns the *number* of rows with equal values at the given pair of attributes. In addition, it can also learn the frequency with which these rows have been accessed in the past if the join is followed by additional projections or selections. If the join is internal, the adversary learns a bit more information: it can also learn *which* rows in the two columns have the same value. Finally, if the query tree includes a sub-tree of the form $\big(\mathbf{T}_1 \bowtie_{\mathsf{att}_1=\mathsf{att}_2} \mathbf{T}_2\big) \bowtie_{\mathsf{att}_2=\mathsf{att}_3} \mathbf{T}_3$ then the leakage will reveal the equality of rows in $\mathbf{T}_1[\mathsf{att}_1]$ and $\mathbf{T}_3[\mathsf{att}_3]$.

Projections. Projections disclose the size of the column and the frequency with which the rows have been accessed.

Cross product. Cross product operations reveal the least information. If it has the form $\mathbf{T} \times [a]$, where a is a constant then the adversary learns $|a|$. If it has form $\mathbf{T}_1 \times \mathbf{T}_2$ it learns the size of both tables (which is already revealed at setup).

A. Efficiency

We now turn to analyzing the search and storage efficiency of our construction. Given a query tree QT query we show in the Theorem below—whose proof is in the full version of this work [6]—that the search complexity of opx is asymptotically optimal.

Theorem V.1. *If Σ_{mm} is optimal, then the time and space complexity of the* Query *algorithm presented in Section (IV) is optimal.*

OPX has the same asymptotic storage complexity as SPX but, in practice, requires more storage. 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 [25], [22], [38], [19], [18] produce encrypted structures with storage overhead that is linear in the sum of the tuple sizes. If we use such a scheme as the underlying multi-map encryption scheme, EMM_R and EMM_C are $O(\sum_{\mathbf{r}\in\mathsf{DB}} \#\mathbf{r})$ and $O\big(\sum_{\mathbf{c}\in\mathsf{DB}^\intercal} \#\mathbf{c}\big)$, respectively, since the former maps the coordinates of each row in DB to their (encrypted) row and the latter maps the coordinates of very column to their (encrypted) columns. Since EMM_V maps the cells in DB to tokens for their rows, it requires $O\big(\sum_{\mathbf{c}\in\mathsf{DB}^\intercal}\sum_{v\in\mathbf{c}} \#\mathsf{DB}_{\mathsf{att}(\mathbf{c})=v}\big)$ storage. Similarly, SET contains the pseudo-random evaluation of the coordinates of all rows and therefore requires $O\big(\sum_{\mathbf{c}\in\mathsf{DB}^\intercal}\sum_{v\in\mathbf{c}} \#\mathsf{DB}_{\mathsf{att}(\mathbf{c})=v}\big)$. For each $\mathbf{c} \in \mathsf{DB}^\intercal$, map $\mathsf{EMM}_{\mathbf{c}}$ maps each pair $(\mathbf{c}, \mathbf{c}')$ such that $\mathsf{dom}(\mathsf{att}(\mathbf{c})) = \mathsf{dom}(\mathsf{att}(\mathbf{c}'))$, to a tuple of tokens for rows in $\mathsf{DB}_{\mathsf{att}(\mathbf{c})=\mathsf{att}(\mathbf{c}')}$. As such, the collection $(\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c}\in\mathsf{DB}^\intercal}$ has size

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

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

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

VI. Emulation

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

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

Definition VI.1 (SQL emulator)**.** *Let* $\mathsf{STE} = (\mathsf{Setup}, \mathsf{Token}, \mathsf{Query}, \mathsf{Resolve})$ *be a response-hiding structured encryption scheme and* $\mathsf{SQL} = (\mathsf{Setup}, \mathsf{Exec})$ *be a relational DBMS. An SQL emulator* $\mathsf{Emu} = (\mathsf{Reshape}, \mathsf{Reform})$ *for* STE *is a set of two polynomial-time algorithms that work as follows:*

- $\mathsf{DB} \leftarrow \mathsf{Reshape}(\mathsf{EDS})$*: is a possibly probabilistic algorithm that takes as input an encrypted structure* EDS *generated using* $\mathsf{STE.Setup}$ *and outputs a database* $\mathsf{DB} = (\mathbf{T}_1, \ldots, \mathbf{T}_n)$.
- $\mathsf{Q} \leftarrow \mathsf{Reform}(\mathbb{S}(\mathsf{DB}), \mathsf{tk})$ *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* $\mathsf{Token}(K, q)$*,* $\mathsf{SQL.Exec}(\mathsf{DB}, \mathsf{Q}) = \mathsf{STE.Query}(\mathsf{EDS}, \mathsf{tk})$.

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

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

A. A SQL Emulator for Pibase

We recall the Pibase scheme from [18]. The Setup algorithm of Pibase samples a key K and instantiates a dictionary DX. For each label $\ell \in \mathbb{L}$, it generates two label keys $K_{\ell,1}$ and $K_{\ell,2}$ by evaluating a pseudo-random function F_K on on $\ell \| 1$ and $\ell \| 2$, respectively. Then, for each value v_i in the tuple $\mathbf{t}_\ell = (v_1, \cdots, v_m)$ associated with ℓ, it creates an encrypted label $\ell'_i := F_{K_{\ell,1}}(i)$ which is the evaluation of $F_{K_{\ell,1}}$ on a counter. It then inserts an encrypted label/value pair $(\ell'_i, \mathsf{Enc}_{K_2}(v_i))$ in the dictionary DX. The encrypted multi-map EMM consists of the dictionary DX. $\mathsf{EMM} = \mathsf{DX}$ is sent to the server.

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

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

```
CREATE TABLE $\mathbf{T}$.name, label, value.
```

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

```
Create Index On $\mathbf{T}$.name (label).
```

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

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

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

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

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

B. A SQL Emulator for OPX *Set Structure*

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

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

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

```
CREATE TABLE $\mathbf{T}$.name, label.
```

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

```
Create Index On $\mathbf{T}$.name (label).
```

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

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

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

```
SELECT ($S$).* FROM ($S$) WHERE EXISTS(
          SELECT label FROM $\mathbf{T}$.name
          WHERE label = $\mathsf{UDF}_F(\mathsf{tk}, S[\mathtt{pos}])$
),
```

where S is the input of the server.

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

C. A SQL Emulator for OPX

Our SQL emulator for OPX, $\mathsf{Emu}_{\mathrm{OPX}} = (\mathsf{Reshape}, \mathsf{Reform})$, makes use of the two emulators described in Sections VI-A and VI-B. The first, $\mathsf{Emu}_{\mathsf{PB}} = (\mathsf{Reshape}, \mathsf{Reform})$ is for the Pibase multi-map encryption scheme [18] whereas the second $\mathsf{Emu}_{\mathsf{SET}} = (\mathsf{Reshape}, \mathsf{Reform})$ is for the SET structure [18]. Given an OPX encrypted database $\mathsf{EDB} = (\mathsf{EMM}_R, \mathsf{EMM}_C, \mathsf{EMM}_V, \mathsf{SET}, (\mathsf{EMM}_{\mathbf{c}})_{\mathbf{c}\in\mathsf{DB}^\intercal}, (\mathsf{EMM}_{\mathbf{c},\mathbf{c}'})_{\mathbf{c},\mathbf{c}'\in\mathsf{DB}^\intercal})$, $\mathsf{Emu}_{\mathrm{OPX}}.\mathsf{Reshape}$ simply applies the $\mathsf{Emu}_{\mathsf{PB}}.\mathsf{Reshape}$ algorithm to all the encrypted multi-maps and $\mathsf{Emu}_{\mathsf{SET}}.\mathsf{Reshape}$ to the set structure SET. This results in an emulated encrypted database $\mathsf{EEDB} = \big(\mathbf{T}_R, \mathbf{T}_C, \mathbf{T}_V, \{\mathbf{T}_{\mathbf{c}}\}_{\mathbf{c}}, \{\mathbf{T}_{\mathbf{c},\mathbf{c}'}\}_{\mathbf{c},\mathbf{c}'}, \mathbf{T}_{\mathsf{SET}}\big)$.

The Reform algorithm takes as input a token tree TT and produces a SQL query plan by traversing the tree in post-order and for each node N doing the following:

- **(leaf select):** if N has form stk, it replaces it with
$$\mathrm{Q}_{\mathsf{stk}} := \mathsf{Emu}_{\mathsf{PB}}.\mathsf{Reform}(\mathbb{S}(\mathbf{T}_V), \mathsf{stk});$$
- **(internal constant select):** if N has form $(\mathsf{rtk}, \mathsf{pos})$, it replaces it with
$$\mathrm{Q}_{\mathsf{rtk}} := \mathsf{Emu}_{\mathsf{SET}}.\mathsf{Reform}(\mathbb{S}(\mathbf{T}_{\mathsf{SET}}), \mathsf{rtk});$$
- **(leaf join):** if N has form $(\mathsf{jtk}, \mathsf{pos})$, it replaces it with
$$\mathrm{Q}_{\mathsf{jtk}} := \mathsf{Emu}_{\mathsf{PB}}.\mathsf{Reform}(\mathbb{S}(\mathbf{T}_{\mathsf{pos}}), \mathsf{jtk});$$
- **(internal join):** if N has form $(\mathsf{etk}, \mathsf{pos}_1, \mathsf{pos}_2)$, it replaces it with
$$\mathrm{Q}_{\mathsf{etk}} := \mathsf{Emu}_{\mathsf{PB}}.\mathsf{Reform}(\mathbb{S}(\mathbf{T}_{\mathsf{pos}_1,\mathsf{pos}_2}), \mathsf{etk});$$
- **(intermediate internal join):** if N has form $(\mathsf{pos}_1, \mathsf{pos}_2)$, then it replaces it with
$$\mathbf{R}_{\mathbf{in}}^{(l)} \bowtie_{\mathsf{pos}_1=\mathsf{pos}_2} \mathbf{R}_{\mathbf{in}}^{(r)};$$
- **(leaf projection):** if N is a leaf node of form ptk then it replaces it with
$$\mathrm{Q}_{\mathsf{ptk}} := \mathsf{Emu}_{\mathsf{PB}}.\mathsf{Reform}(\mathbb{S}(\mathbf{T}_C), \mathsf{ptk});$$
- **(cross product):** if N is a node of form × then it does nothing.

VII. The KafeDB Architecture

KafeDB is a relational EDB system. Our implementation is built on top of PostgreSQL but because of our emulation technique, we could have built KafeDB on top of any relational database management system—even a managed DBMS in the cloud. KafeDB does not require any modifications to applications or servers. It is completely transparent to both. Figure 1 provides a high-level description of KafeDB's design.

Overview. KafeDB is composed of the KafeDB client which runs on a trusted device (preferably the same machine as the application) and a DBMS server which runs on an untrusted device. The KafeDB client sits between the application and the server and handles all the application queries. When the application creates a database, the client sets up and emulates an OPX-encrypted database on the server. Thanks to our emulation techniques, the OPX-encrypted database is represented as a standard SQL database. At query time, the client receives a SQL query from the application, converts it into an OPX token tree and emulates it into a standard SQL query plan. This query plan is then executed by the server. The server returns an encrypted result to the client who decrypts it and executes any additional processing (e.g., grouping, aggregating etc.) before returning the result to the application. As illustrated in Figure 1, the KafeDB client is composed of several modules which we now describe.

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

Parser. The parser module is a standard SQL parser that transforms the SQL query into a query plan/tree. KafeDB uses the SQL parser from SparkSQL [10].

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

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

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

Optimizations. One of the most important components of any DBMS is its query optimizer which transforms a query plan into a physical plan which can be executed as efficiently as possible. Commercial query optimizers are the result of over 40 years of research from the database community and are major reason why real-world DBMSs are so efficient. It follows then that for KafeDB to be competitive at all with a commercial system, it has to

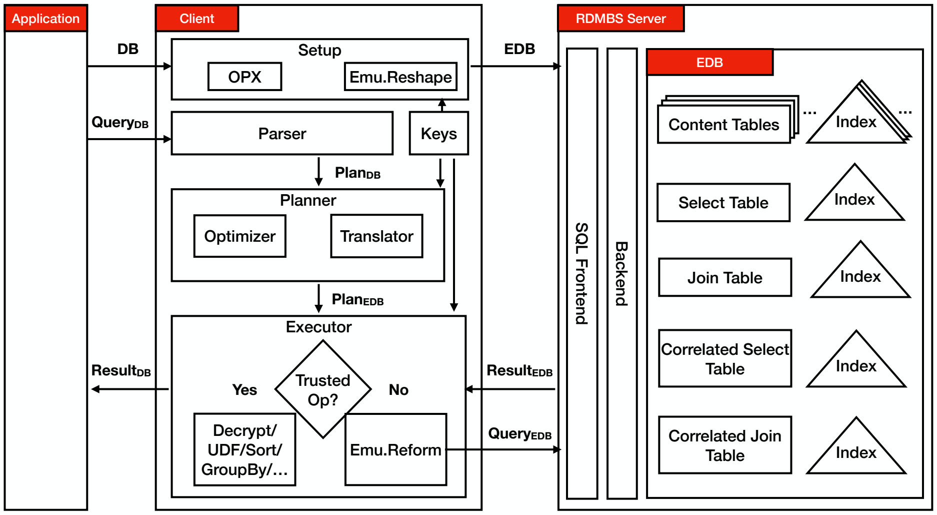


Fig. 1: Overview of KafeDB's architecture over an RDMBS server such as PostgreSQL.

support some form of query optimization. Due to space limitations, we describe KafeDB's optimizations in more detail in Appendix A.

VIII. Empirical Evaluation

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

Implementation. The KafeDB client makes use of and extends several components of Apache Spark SQL [10]. Specifically, it uses and extends Spark SQL's algebraic core for query translation and optimization, its parser to parse plaintext SQL queries into a query plan, and its executor to facilitate split execution. The KafeDB server can be any DBMS but in this evaluation we use PostgreSQL 9.6.2 [30]. The KafeDB client is written in Scala 2.12 and consists of 1599 lines of code. In addition, our framework includes 398 lines for testing, 392 lines to load the TPC-H benchmark, 578 lines to execute the TPC-H benchmark, all calculated using IntelliJ IDEA [1]. the query parser and executor code required for all other modules composing KafeDB Client are inherited from Apache Spark SQL. Our implementation is available for download in an anonymized form here [7]. 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 [39] in the DEX Client and by the pgcrypto module in PostgreSQL 9.6.2.[7]

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

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

Data generation.

SK: *@sam: can you fill thisin*

The TPC-H benchmark specifies the cardinality or the number of rows of each table, multiplied by a scale factor n. We chose a scale factors of 1, ? and ? which lead to about 8, ? and ? million rows and 1, ?, and ? GB of data, respectively.

Each attribute value is sampled uniformly at random from its domain. All filtered and joined attributes are known a-priori by looking to the queries and the database schema. For KafeDB, we only index these specific attributes. We index the same filtered and joined attributes for the (plaintext) PostgreSQL evaluation. This indexing strategy helps to ensure the best possible query performance for both PostgreSQL and KafeDB.

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

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

on the KafeDB server but in the clear. We summarize the composition of these query portions in Table I, and refer the reader to [7] for more details. All queries are run in a uniformly randomized order. The benchmark is first warmed up by executing all the TPC-H queries and discarding the results. The runtime is averaged over 10 runs with satisfactory relative standard error.

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

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

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

A. Setup Time

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

SK: *@sam: can you fill in*

For ? and ? GB databases, KafeDB took ? and ? seconds, respectively.

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

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

TABLE II: TPC-H setup time with scale factor 1.

B. Storage Overhead

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

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

The ratio between contents and indices is similar for both KafeDB and PostgreSQL: both use more storage for their auxiliary structures, namely 86% for the emulated EMMs in KafeDB, and 70% for indexing in PostgreSQL. Because, intuitively speaking, the emulated EMMs play the role of indices over the encrypted contents, dedicating a large percentage of storage for them is not surprising. We defer the details of storage breakdown to the full version [6].

SK: *@sam: can you fill in*

For ? and ? GB databases, KafeDB required ? and ? GBs of storage respectively.

C. Query Efficiency

We now examine the query efficiency of KafeDB.[8] Our results show that KafeDB incurs a slowdown of 1× to 100× for most queries and that query efficiency improves with more memory.

Equal memory vs. equal ratio. Table IV describes the distribution of the slowdown incurred by KafeDB over PostgreSQL over all 22 TPC-H queries. In the equal memory setting, KafeDB runs 7 queries with a 20× slowdown, 8 queries with a slowdown between 20× to 100×, and another 7 queries with more than 100× slowdown. In the equal ratio setting, this improves to 8 queries with a 20× slowdown, 10 between 10× to 100×, and only 4 beyond 100×. All queries

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

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

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

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

TABLE III: `TPC-H` storage breakdown in GB (ratio over total size).

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

TABLE IV: Distribution of the slowdown of `TPC-H` encrypted queries.

see an improvement in the equal ratio setting, but some benefit more than others. For instance, `q20` improves by about 9× but `q10` and `q11` only by 3×. Half of the queries run with less than 31× slowdown. Note that the mean value, while important to assess, is considerably skewed in our case as 5 queries among the 22 have a higher impact on the average; for example, `q5` with a 2232× slowdown, `q19`, `q20`, and `q21` with over a 280× slowdown.

Granular query efficiency. As shown in Table IV, for both memory settings the majority of queries run within 1× to 100× slower on `KafeDB` than on `PostgreSQL`. A higher memory ratio benefits `KafeDB` as its 90%-trimmed average query time shortens from 96× slowdown in the equal memory setting to 63× in the equal ratio setting. We also want to emphasize that some queries do very well such as `q1`, `q16`, and `q17` with a slowdown of less than 6×. Due to space constraints, we defer the details of the individual query runtimes to the full version of this paper [6].

SK: *@sam: can you fill in*

For a ? GB database, query ? ran the fastest, taking ? seconds and query ? ran the slowest requiring ? seconds. For a ? GB database, query ? ran the fastest, taking ? seconds and qeury ? ran the slowest taking ? seconds.

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

D. Optimizations

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

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

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

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

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

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

[1] Letters indicate initials of relation names.

[2] Standard error divided by mean

TABLE V: `Push select through join` optimization.

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

TABLE VI: Many-to-many join factorization and join flattening optimizations.

References

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

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

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

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

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

[6] Anonymous. Safedb: a structurally-encrypted relational database management system. https://drive.google.com/file/d/1oIBYJj37MOUh8NSpia8tUmK35AwHDsMB/view?usp=sharing, 2020.

[7] Anonymous. X. https://anonymous.4open.science/r/acda0b77-6248-47bc-be51-e8c2a9196370/, 2020.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] T. Chiba and T. Onodera. Workload characterization and optimization of tpc-h queries on apache spark. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 112–121. IEEE, 2016.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

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

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

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

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

[40] 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.

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

[42] 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.

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

[44] 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.

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

[46] T. P. P. C. (TPC). TPC-DS benchmark. http://www.tpc.org/tpcds/.

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

[48] 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.

Appendix A
Query Optimization

One of the most important components of any DBMS is its query optimizer which transforms a query plan into a physical plan which can be executed as efficiently as possible. It follows then that for KafeDB to be competitive at all with a commercial system, it has to support some form of query optimization.

Overview of query optimization. Standard query optimization works in several phases. First, the optimizer transforms a *logical query plan* into an equivalent but more efficient logical plan. To do this, the optimizer uses a series of relational identities to transform and re-arrange the order of operations. This works because it is well-known that, in practice, executing certain operations before others tends to yield large gains in efficiency. During the second phase, the optimizer transforms the new logical query plan into a *physical query plan* by choosing concrete algorithms to execute each operation. Because KafeDB works over encrypted data, query optimization is more complex and works differently than in traditional DBMSs. In particular, in KafeDB logical plans are not transformed into physical plans but first into *OPX plans* and then into an *emulated plans*. An OPX plan associates to every logical operation either a standard/plaintext relational operator or an encrypted query on an one of the OPX encrypted structures. The OPX plan is then converted to an emulated plan by replacing each encrypted query by its SQL emulation.

Standard optimizations. KafeDB uses several standard optimization rules like push-select-through-join which inverses the order of selects and joins whenever a select is followed by a join. To see why this optimization works, suppose a logical plan performs a join on two tables of size n and then a select that filters a ε fraction of the resulting n^2 rows. The cost of these two operations is cn^2 for some constant c. If on the other hand, we execute the select operation on both tables first and then join the results, the cost is $cn + \varepsilon_1\varepsilon_2 n^2$ for some constant c and where ε_1 and ε_2 are the fraction of rows selected on the first and second table, respectively.

A. *Custom Optimizations*

In addition to the standard optimizations described above, KafeDB uses a set of custom optimizations that are needed to counterbalance potential inefficiencies incurred by the use of the OPX construction.

Factoring many-to-many joins. Equi-joins can have three forms: one-to-one where each row of the first table can only be joined with a single row of the second table; one-to-many (or many-to-one) where each row of the first table can be joined to multiple rows of the second table; and many-to-many where each row of the first table can be joined with many rows of the second table and each row of the second table can be joined with many rows of the first table. These different joins have different complexities. Specifically, for two tables each with n rows, one-to-one joins are $O(n)$, one-to-many (and many-to-one) joins are $O(n)$ but many-to-many joins are $O(n^2)$. Recall that OPX handles equi-joins between two tables $\mathbf{T}_1$ and $\mathbf{T}_2$ using an EMM of size $O(|\mathbf{T}_1 \bowtie \mathbf{T}_2|)$ which, in the worst-case, is quadratic. To improve on this KafeDB does two things.

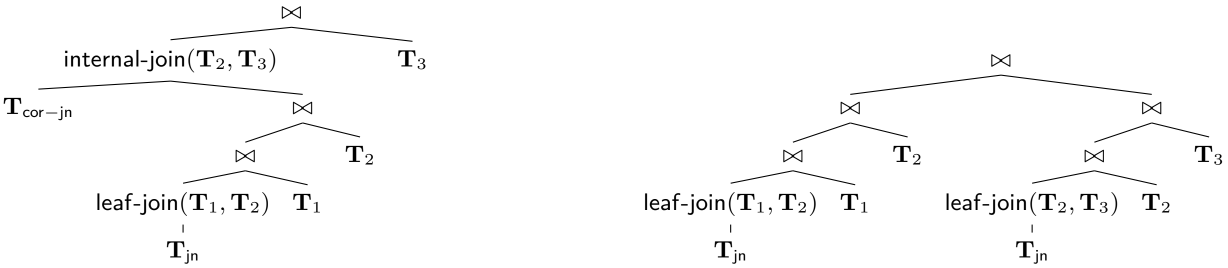
At setup time it converts all many-to-many relationships into two many-to-one relationships by introducing additional tables. For example, given two tables $\mathbf{T}_1$ and $\mathbf{T}_2$ that model a many-to-many relationship, a third table P is created such that $\mathbf{T}_1$ and P have a many-to-one relationship and P and $\mathbf{T}_2$ have a one-to-many relationship. Note that this can be done at setup time using the database schema. Then, at query time, KafeDB factors many-to-many joins into two many-to-one (or one-to-many) joins using the identity $\mathbf{T}_1 \bowtie \mathbf{T}_2 = (\mathbf{T}_1 \bowtie \mathrm{P}) \bowtie \mathbf{T}_2$.

This optimization has several advantages. The first is that it guarantees that the storage will be linear rather than quadratic. Second, it improves the I/O complexity of joins since only $O(n)$ reads are needed as opposed to $O(n^2)$. Note, however, that the worst-case computational cost is still $O(n^2)$ if the result of the join is quadratic. We note that if the database is in third-normal form—which is common for analytical workloads—then the many-to-many relationships are already factored using a third table so the storage overhead will be linear

Flattening multi-way joins. A multi-way join is a sequence of multiple joins, e.g., $\mathbf{T}_1 \bowtie \mathbf{T}_2 \bowtie \mathbf{T}_3$. Multi-way joins can be executed in several ways, e.g., for the above 3-way join we have,

$$\begin{aligned}\mathbf{T}_1 \bowtie \mathbf{T}_2 \bowtie \mathbf{T}_3 &= (\mathbf{T}_1 \bowtie \mathbf{T}_2) \bowtie \mathbf{T}_3 \\ &= (\mathbf{T}_1 \bowtie \mathbf{T}_2) \bowtie (\mathbf{T}_2 \bowtie \mathbf{T}_3)\end{aligned}$$

In a standard DBMS the second form, which is pipelined, is usually more efficient than the third form which is flattened. The reason is that the pipelined variant requires fewer joins. In KafeDB, however, the emulations of these 3-way joins have different complexity. Figure 2 shows the emulations of both the pipelined and the flattened variants. While the emulated flattened form has more joins than the emulated pipelined form, the total complexity of the pipelined form is $O(\mathbf{T}_1\mathbf{T}_2 + \mathbf{T}_1\mathbf{T}_2\mathbf{T}_3)$ whereas the complexity of the


⋈
internal-join($\mathbf{T}_2, \mathbf{T}_3$)
$\mathbf{T}_3$
$\mathbf{T}_{\text{cor-jn}}$
⋈
⋈
$\mathbf{T}_2$
leaf-join($\mathbf{T}_1, \mathbf{T}_2$)
$\mathbf{T}_1$
$\mathbf{T}_{\text{jn}}$
⋈
⋈
⋈
⋈
$\mathbf{T}_2$
⋈
$\mathbf{T}_3$
leaf-join($\mathbf{T}_1, \mathbf{T}_2$)
$\mathbf{T}_1$
leaf-join($\mathbf{T}_2, \mathbf{T}_3$)
$\mathbf{T}_2$
$\mathbf{T}_{\text{jn}}$
$\mathbf{T}_{\text{jn}}$

(a) Emulated pipelined form.

(b) Emulated flattened form.

Fig. 2: Comparison of emulated pipelined and flattened multi-way joins.

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

Course	Department
16	CS
18	Math

Fig. 3: Plaintext database DB.

flattened form is $O(\mathbf{T}_1\mathbf{T}_2 + \mathbf{T}_2\mathbf{T}_3)$, where we write $\mathbf{T}_i$ to mean $|\mathbf{T}_i|$ for visual clarity.

Handling compound keys. A compound key is a key that consists of more than one attribute. If KafeDB does not recognize compound keys and encrypts the tables as-is using OPX, this will result in very large EMMs. For example, if the database has two tables $\mathbf{T}_1(\mathsf{att}_1, \mathsf{att}_2, \dots)$ and $\mathbf{T}_2(\mathsf{att}_1, \mathsf{att}_2, \dots)$ with compound keys $(\mathsf{att}_1, \mathsf{att}_2)$, then OPX's $\mathsf{EMM}_{\mathsf{att}_1}$ and $\mathsf{EMM}_{\mathsf{att}_2}$ will be $O(2\mathbf{T}_1\mathbf{T}_2)$. A better approach is to transform the tables by creating a new attribute att_3 in both tables that hold pairs $(\mathsf{att}_1, \mathsf{att}_2)$ before encrypting with OPX. This reduces the size of the two EMMs to $O(\mathbf{T}_1\mathbf{T}_2)$ and to $O(\max(\mathbf{T}_1, \mathbf{T}_2))$ in the case of a many-to-one relationship.

Appendix B
A Concrete Example of Indexing in OPX

Similar to [36], 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).

Figure (4) shows the result of applying our method to index the database $\mathsf{DB} = (\mathbf{T}_1, \mathbf{T}_2)$, as detailed in Section (IV). There are five multi-maps MM_R, MM_C, MM_V, $\mathsf{MM}_{\mathsf{Course}}$, $\mathsf{MM}_{\mathsf{T_2.Course},\mathsf{T_1.Course}}$, and a set SET. We detail below how the indexing works for this example.

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

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

is added to $\mathsf{MM}_{\mathsf{Course}}$. The correlated join multi-map, $\mathsf{MM}_{\mathsf{T_2.Course},\mathsf{T_1.Course}}$, 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,

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

This SQL query can be rewritten as a query tree, see Figure (5a), and then translated, based on the OPX protocol into a token tree as depicted in Figure (5b).[9]

We detail in Figure (5c) the intermediary results of the token tree execution using the indexed database and provide below a high level description of how it works.

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

[9] 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.

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

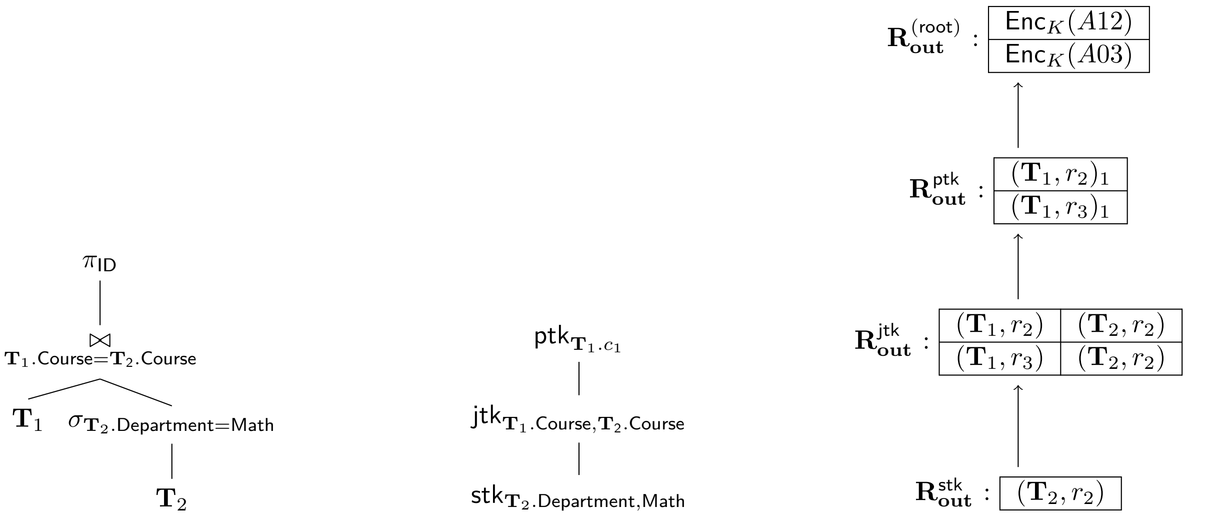
$\mathsf{MM}_{\mathsf{Course}}$	
$\mathsf{T}_1 \| c_3 \| \mathsf{T}_2 \| c_1$	$(\mathsf{T}_1 \| r_1, \mathsf{T}_2 \| r_1)$, $(\mathsf{T}_1 \| r_2, \mathsf{T}_2 \| r_2)$, $(\mathsf{T}_1 \| r_3, \mathsf{T}_2 \| r_2)$

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

Fig. 4: Indexed database.


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

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

Fig. 5: A query tree translated to a token tree which is then executed using the indexed database.

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

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

Concrete storage overhead. The plaintext database DB is composed of thirteen cells excluding the tables attributes.[10] The indexed structure consists of fifty eight pairs. Assuming that a pair and a cell have the same bit length, our indexed representation of the database has a multiplicative storage overhead of 4.46. In particular, each of the multi-maps MM_R, MM_C, MM_V and the set SET have the same size as the plaintext database (i.e., 13 pairs). This explains the 4× factor. It is worth emphasizing that even if one considers a larger database, the 4× factor remains unchanged. The additive component of the multiplicative factor, i.e., the 0.46, will vary, however, from one database to another depending on the number of columns with the same domain and the number of equal rows in these columns.

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