A preliminary version of this paper appears in the proceedings of ACNS 2021. This is the full version.

Improved Structured Encryption for SQL Databases via Hybrid Indexing

David Cash[1], Ruth Ng[2], and Adam Rivkin[1]

[1] University of Chicago
[2] University of California San Diego
{davidcash,amrivkin}@uchicago.edu, ring@eng.ucsd.edu

Abstract. We introduce a new technique for indexing joins in encrypted SQL databases called *partially precomputed joins* which achieves lower leakage and bandwidth than those used in prior constructions. These techniques are incorporated into state-of-the-art structured encryption schemes for SQL data, yielding a *hybrid indexing* scheme with both partially and fully precomputed join indexes. We then introduce the idea of *leakage-aware query planning* by giving a heuristic that helps the client decide, at query time, which index to use so as to minimize leakage and stay below a given bandwidth budget. We conclude by simulating our constructions on real datasets, showing that our heuristic is accurate and that partially-precomputed joins perform well in practice.

1 Introduction

SQL applications are often deterred from using cloud storage solutions because they do not wish to grant a third party access to their sensitive data. Yet, in-house solutions often are less convenient than these large-scale ones and are vulnerable to compromise as well. This calls for a cryptographic solution which allows data on the cloud to be end-to-end encrypted so that the server never "sees" the sensitive data. This in turn poses a challenge when the server is called upon to perform SQL operations on the data.

Most current offerings of this technology depend heavily on property-revealing encryption (PRE), making them vulnerable to leakage abuse attacks (LAAs). For example, Always Encrypted either deterministically encrypts columns or stores them with an ordered index [4]. These techniques have been shown to offer little-to-no privacy in certain practical scenarios [37,26].

A more promising approach is structured encryption (StE) which uses auxiliary encrypted data structures (e.g. encrypted multimaps) to support a subset of SQL queries [15]. This is done by translating the SQL query into tokens which can be passed to the server to query the auxiliary structures. The outputs of this are compiled, decrypted and processed to retrieve the SQL query result. Security is measured by *leakage profiles*, which characterize what information a curious server can learn. In particular, StE-based constructions leak equal or less than PRE-based constructions and resist most known LAAs [20,37,8,9,25,26,27].

Our contributions. Our work can be grouped into three main contributions:

1. **Partially precomputed joins:** We introduce a new way to index (equi)joins which stems from the simple observation that when the server fully precomputes (FP) joins, the client has to download and decrypt a quadratic number of rows and the server learns the equality pattern of said rows. In our approach, the server partially precomputes (PP) joins: instead of indexing exactly which rows from the input table should be concatenated and returned, it just stores the set of rows from each input table that appears anywhere in the join output. At query time, the client downloads these sets and computes the join. When this is used to support SQL queries of the form "**select** * **from** id_1 **join** id_2 **on** $at_1 = at_2$", PP outperforms FP in both leakage and bandwidth at the cost of a logarithmic factor of client computation (in the worst case).
2. **Hybrid indexing:** When we incorporate PP joins into state-of-the-art StE schemes, we discover that some queries (e.g. those with a selection subquery) cannot be computed in the same way because the server does not know the equality pattern on the join columns (i.e. how the rows "match up"). So while PP joins are still the more secure choice, they sometimes incur more bandwidth than FP. To address this, we develop a hybrid StE scheme with both forms of indexing. The client chooses which to use at query time. We provide the first heuristic (that we are aware of) to enable this type of *leakage-aware client-side*

query planning, helping the client decide how to minimize leakage without exceeding a given bandwidth budget.

3. **Simulations on real data:** We quantify the effect of using FP and PP join indexing on bandwidth incurred by simulating our constructions on data from the City of Chicago's Data Portal and MySQL's sample Sakila database [2,3]. On simple (non-recursive) join queries, PP's bandwidth is on average 231 times less than FP's but more complex (recursive) queries are split down the middle as to which option used less bandwidth. We also demonstrate the accuracy of our heuristic under different client storage constraints. Assuming client storage comparable to that which is used in SQL Server, our heuristic chose a query plan with the maximal number of PP joins 79% of the time, and the optimal query plan 68% of the time.

RELATED WORK. Encrypted databases have been treated from a variety of perspectives. Structured encryption (StE) was defined by Chase and Kamara (CK) and is a special case of SSE, which was first defined by SWP [42].

We see our work as a direct extension and improvement upon SPX and OPX, two schemes which applied StE to the problem of indexing SQL databases [15,32,34]. Both our scheme and OPX address a similar query class to the one introduced in SPX, but lower leakage by using the hashset technique from OXT and primitives inspired by CJJJKRS [12,13]. In particular, our FpSj scheme in Section 4.2 bears many similarities to OPX with minor leakage improvements from using a single indexing data structure. Our PpSj and HybStI schemes (in Section 4.2 and Section 5 respectively) introduce a new technique which further lowers leakage and server storage. For non-recursive queries, there are also substantial bandwidth savings.

PRE-based solutions achieves higher query support at the cost of higher leakage [39,24,1,21,44], and are particularly susceptible to leakage abuse attacks [20,37,8,9,25,26,27].

Finally, encrypted search has also been attempted using alternate models and architectures including the database-provider model [28], MPC [16,6], ORAM [23] and trusted execution environments [35,11,5].

Other works have also partially delegated computation to the client, to reduce leakage or increase query support, though none have applied it to joins [43,17,19].

2 Preliminaries

We denote the empty string with ϵ. Given positive integer n, let $[n] = \{1, 2, \ldots, n\}$. Given tuples $\mathbf{t}_1 = (x_1, \ldots, x_n)$ and $\mathbf{t}_2 = (y_1, \ldots, y_m)$ we write $\mathbf{t}_1 \| \mathbf{t}_2$ as a shorthand for $(x_1, \ldots, x_n, y_1, \ldots, y_m)$. We extend set operations $\cap, \cup \in, \subseteq$ from sets to tuples by interpreting the tuples as sets.

Our algorithms often make use of dictionaries $\mathbf{D}$ which map labels $\ell \in \{0,1\}^*$ to values $\mathbf{D}[\ell] \in \{0,1\}^* \cup \{\perp\}$. We also adopt the shorthand $\mathbf{D}.\mathsf{Lbls} = \{\ell \in \{0,1\}^* : \mathbf{D}[\ell] \neq \perp\}$. A multimap $\mathbf{M}$ is an dictionary where $\mathbf{M}[\ell]$ is either a set of strings or $\perp$.

PSEUDOCODE. In pseudocode, we will assume that all integers , strings and sets are initialized to 0 , ϵ and $\emptyset$ respectively. For dictionaries and multimaps, they are initialized with all labels mapping to $\perp$. If S is a set or dictionary value, we write $S \xleftarrow{\cup} x$ in pseudocode as a shorthand for $S \leftarrow S \cup \{x\}$, initializing it first to $\emptyset$ if necessary. If $\mathbf{t}$ is a tuple, we similarly mean $\mathbf{t} \leftarrow \mathbf{t} \| (x)$ by writing $\mathbf{t} \leftarrow \mathbf{t} \xleftarrow{\cup} x$. Finally, we will write "Define X : *pred*" to set X (a function or constant) in such a way that the predicate *pred* is true. If there are undefined variables in *pred* we treat it as a random variable and expect that X is defined such that *pred* will always be true.

GAMES. Our work uses the code-based game-playing framework of BR [7]. Let G be a game and A an adversary. Then, we write $\Pr[\mathrm{G}(A)]$ to denote the probability that A plays G and the latter returns true. G may provide oracles to A, and if so we write $A^{\mathrm{O}_1, \cdots, \mathrm{O}_n}$ to denote that A is run with access to oracles $\mathrm{O}_1, \ldots, \mathrm{O}_n$.

SYMMETRIC ENCRYPTION, IND$-SECURITY. Symmetric Encryption (SE) scheme SE defines key set $\mathsf{SE.KS}$, encryption algorithm $\mathsf{SE.Enc}$ and decryption algorithm $\mathsf{SE.Dec}$. Encryption is randomized, taking a key $K_\mathrm{e} \in \mathsf{SE.KS}$ and a message $M \in \{0,1\}^*$ and returns a ciphertext $C \in \{0,1\}^*$. Decryption is deterministic and takes a key and ciphertext, returning a message. SE also defines a ciphertext length function $\mathsf{SE.cl}$. We require that if $C \leftarrow\!\!\$\ \mathsf{SE.Enc}(K_\mathrm{e}, M)$ then $|C| = \mathsf{SE.cl}(|M|)$ and $\Pr[\mathsf{SE.Dec}(K_\mathrm{e}, C) = M] = 1$. We want our

$\underline{\textbf{Game } \mathrm{G}^{\mathrm{ind\$}}_{\mathsf{SE}}(A)}$	$\underline{\textbf{Game } \mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A)}$
$b \leftarrow\!\!\$\ \{0,1\}$; $K_{\mathrm{e}} \leftarrow\!\!\$\ \mathsf{SE.KS}$	$b \leftarrow\!\!\$\ \{0,1\}$; $K_{\mathrm{f}} \leftarrow\!\!\$\ \mathsf{F.KS}$
$b' \leftarrow\!\!\$\ A^{\mathrm{Enc}}$; Return $b = b'$	$b' \leftarrow\!\!\$\ A^{\mathrm{Fn}}$; Return $b = b'$
$\underline{\textbf{Alg } \mathrm{Enc}(m)}$	$\underline{\textbf{Alg } \mathrm{Fn}(X)}$
$c_1 \leftarrow\!\!\$\ \mathsf{SE.Enc}(K_{\mathrm{e}}, m)$	If $\mathbf{C}[X] = \perp$ then $\mathbf{C}[X] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{F.ol}}$
$c_0 \leftarrow\!\!\$\ \{0,1\}^{\lvert c_1 \rvert}$; Return c_b	$c_1 \leftarrow\!\!\$\ \mathsf{F.Ev}(K_{\mathrm{f}}, X)$; $c_0 \leftarrow \mathbf{C}[X]$; Return c_b

Fig. 1. Games used in defining IND$ security of SE scheme SE (right) and PRF security of function family F (left)

SE schemes to protect the privacy of M, so ciphertexts should be indistinguishable from a random string of length $\mathsf{SE.cl}(|M|)$. We capture this with the game $\mathrm{G}^{\mathrm{ind\$}}_{\mathsf{SE}}$ in Fig. 1 and say that a scheme is IND$-secure if $\mathbf{Adv}^{\mathrm{ind\$}}_{\mathsf{SE}}(A) = 2\Pr[\mathrm{G}^{\mathrm{ind\$}}_{\mathsf{SE}}(A)] - 1$ is small for all adversaries A.

Function Families, PRF-security. A function family F defines a key set $\mathsf{F.KS}$ and an output length $\mathsf{F.ol}$. It defines a deterministic evaluation algorithm $\mathsf{F.Ev}\colon \mathsf{F.KS} \times \{0,1\}^* \to \{0,1\}^{\mathsf{F.ol}}$. We define PRF security for function family F via the game $\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}$ depicted in Fig. 1. We say that F is a PRF if $\mathbf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(A) = 2\Pr[\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A)] - 1$ is small for all adversaries A.

3 Structured Indexing for SQL data types

We now generalize CK's definition of structured encryption and provide a new framework for modeling encrypted SQL systems [15].

Abstract Data Types. An *abstract data type* ADT defines a domain set $\mathsf{ADT.Dom}$, a query set $\mathsf{ADT.QS}$, and a deterministic specification function $\mathsf{ADT.Spec} : \mathsf{ADT.Dom} \times \mathsf{ADT.QS} \to \{0,1\}^*$.

An example is the dictionary ADT DyAdt. $\mathsf{DyAdt.Dom}, \mathsf{DyAdt.QS}$ contain all possible dictionaries $\mathbf{D}$ and labels respectively (as defined in Section 2), and $\mathsf{DyAdt.Spec}(\mathbf{D}, \ell) = \mathbf{D}[\ell]$. Multimap ADT MmAdt is defined analogously.

Structured Indexing. We generalize Structured Encryption (StE) schemes (as defined by CK [15]) to *structured indexing* (StI) schemes. These are StE schemes without a decryption algorithm. The intuition here is that the handling of outsourced data often indexes the data in addition to encrypting it and we would like these encrypted indexes, whatever form they take, to achieve semantic security as well. Later, we show how this primitive allows us to modularize StE schemes. A StI scheme StI for ADT defines a set of keys $\mathsf{StI.KS}$ and the following algorithms:

- Randomized encryption algorithm $\mathsf{StI.Enc}$ which takes a key $K' \in \mathsf{StI.KS}$ and an element of $\mathsf{ADT.Dom}$ and returns an updated key K and index $\mathsf{IX} \in \{0,1\}^*$. This syntax generalizes that of CK by allowing key generation to occur within or outside $\mathsf{StI.Enc}$.
- Possibly randomized token generation algorithm $\mathsf{StI.Tok}$ which takes a key and a query from $\mathsf{ADT.QS}$, and returns fixed length token $\mathsf{tk} \in \{0,1\}^{\mathsf{StI.tl}}$.
- Deterministic evaluation algorithm $\mathsf{StI.Eval}$ which takes a token and index, and returns a ciphertext string $C \in \{0,1\}^*$.
- Finalization algorithm $\mathsf{StI.Fin}$ which takes K, q and an input string, and returns an output string.

Intuitively, the client indexes his data then encrypts this index with $\mathsf{StI.Enc}$, storing IX on the server. At query time, the client uses $\mathsf{StI.Tok}$ to generate a token and sends it to the server who runs $\mathsf{StI.Eval}$, returning C to the client. $\mathsf{StI.Fin}$ can be used for client-side post-processing of the data. Note that the output of $\mathsf{StI.Eval}$ need not be the input to $\mathsf{StI.Fin}$. In our indexing schemes the server will use the output of $\mathsf{StI.Eval}$ as "pointers" to retrieve rows of SQL data stored in a different data structure which in turn form the input to $\mathsf{StI.Fin}$.

Structured Encryption. We can now define StE as a special cases of StI. Intuitively, an StE scheme is an StI scheme where the data structure is also used to store query responses (as opposed to just indexing them). The output of evaluation can be fed into finalization for decryption and should return the query result. To highlight this, StE schemes have a *decryption algorithm* $\mathsf{StE.Dec}$ in place of a finalization algorithm which

Game $\mathrm{G}^{\mathrm{cor}}_{\mathsf{StE}}(A)$

$(\mathsf{DS}, st) \leftarrow\!\$\ A(\mathsf{s})$; $K' \leftarrow\!\$\ \mathsf{StE.KS}$
If $\mathsf{DS} \notin \mathsf{ADT.Dom}$ then return false
$(K, \mathsf{EDS}) \leftarrow\!\$\ \mathsf{StE.Enc}(K', \mathsf{DS})$
$A^{\mathrm{Tok}}(\mathsf{g}, \mathsf{EDS}, st)$; Return win

Oracle $\mathrm{Tok}(q)$

If $q \notin \mathsf{ADT.QS}$ then $\mathsf{win} \leftarrow \mathsf{false}$
$M \leftarrow \mathsf{StE.Dec}(K, \mathsf{StE.Eval}(\mathsf{StE.Tok}(K, q), \mathsf{EDS}))$
If $\mathsf{ADT.Spec}(\mathsf{DS}, q) \neq M$ then $\mathsf{win} \leftarrow \mathsf{false}$
Return tk

Game $\mathrm{G}^{\mathrm{ss}}_{\mathsf{StI},\mathcal{L},\mathcal{S}}(A)$

$(\mathsf{DS}, (q_1, \ldots, q_n), st) \leftarrow\!\$\ A(\mathsf{s})$
$b \leftarrow\!\$\ \{0,1\}$; $K' \leftarrow\!\$\ \mathsf{StI.KS}$
If $\mathsf{DS} \notin \mathsf{ADT.Dom}$ or $\{q_i\}_{i=1}^n \not\subseteq \mathsf{ADT.QS}$ then return false
If $b = 1$ then
  $(K, \mathsf{IX}) \leftarrow\!\$\ \mathsf{StI.Enc}(K', \mathsf{DS})$
  For $i \in [n]$ do $\mathsf{tk}_i \leftarrow\!\$\ \mathsf{StI.Tok}(K, q_i)$
Else
  $(\mathsf{IX}, (\mathsf{tk}_1, \ldots, \mathsf{tk}_n)) \leftarrow\!\$\ \mathcal{S}(\mathcal{L}(\mathsf{DS}, (q_1, \ldots, q_n)))$
$b' \leftarrow\!\$\ A(\mathsf{g}, \mathsf{IX}, (\mathsf{tk}_1, \ldots, \mathsf{tk}_n), st)$; Return $(b = b')$

Fig. 2. Games used in defining correctness for StE (structured encryption scheme for ADT) and semantic security for StI (structured indexing scheme for ADT) with respect to leakage algorithm $\mathcal{L}$ and simulator $\mathcal{S}$.

takes as input K, q, C and returns the query result. We define correctness via game $\mathrm{G}^{\mathrm{cor}}_{\mathsf{StE}}$ in Fig. 2 and say that StE is correct if the advantage of all adversaries A, defined $\mathbf{Adv}^{\mathrm{cor}}_{\mathsf{StE}}(A) = \Pr[\mathrm{G}^{\mathrm{cor}}_{\mathsf{StE}}(A)]$, is low. The correctness of our schemes will depend on the collision resistance of their function family primitives. Since we assume these are PRFs to prove security, we will also assume that their key-lengths are sufficient to ensure correctness.

We subdivide StE schemes into two types. We say that a scheme $\mathsf{StE}_{\mathrm{rr}}$ is *response revealing* (RR) if evaluation itself returns the query result. In other words, decryption must be such that $\mathsf{StE}_{\mathrm{rr}}.\mathsf{Dec}(K, q, C) = C$ for all K, q, C. An StE scheme that is not RR is *response hiding* (RH).

We refer to StE for the multimap and dictionary data types as multimap and dictionary encryption (MME/DYE) respectively. Our constructions make use of a specific dictionary encryption scheme adapted from CJJ+'s SSE scheme $\prod_{\mathrm{bas}}$ (2Lev in the Clusion library) [12,36]. In this scheme, the encrypted data structure is itself a dictionary $\mathbf{D}'$. We start by padding all values in the input dictionary to the same length, then for each label-value pair $\ell, \mathbf{D}[\ell]$, we do $\mathbf{D}'[\mathsf{F.Ev}(K_{\mathrm{f}}, \ell)] \leftarrow \mathsf{SE.Enc}(K_{\mathrm{e}}, \mathbf{D}[\ell])$ where F is a pseudorandom function family and SE is a symmetric encryption scheme. For completeness, we include the pseudocode of this dictionary encryption scheme (which we call Dye_π) in Appendix A. Our constructions also make use of a generic RR multimap encryption scheme. We adapt Dye_π to $\mathsf{Mme}^{\mathrm{rr}}_\pi$ (using a counter and label-dependent K_{e}) as an example of such a scheme in Appendix A.

Semantic security. We define semantic security for StI using game $\mathrm{G}^{\mathrm{ss}}_{\mathsf{StI},\mathcal{L},\mathcal{S}}$ depicted in Fig. 2, where StI is a StI scheme for ADT and $\mathcal{L}, \mathcal{S}$ are algorithms we refer to as the leakage algorithm and simulator respectively. The adversary runs in a setup and guessing phase, as indicated by the first argument to it. Its advantage is $\mathbf{Adv}^{\mathrm{ss}}_{\mathsf{StI},\mathcal{L},\mathcal{S}}(A) = 2\Pr[\mathrm{G}^{\mathrm{ss}}_{\mathsf{StI},\mathcal{L},\mathcal{S}}(A) = 1] - 1$. Note that when StI is an StE scheme we recover CK's non-adaptive security notion.

3.1 SQL Data Types

We now describe our notation for SQL data, queries and operations. We then define a class of ADTs we call *SQL data types* to construct StE schemes for.

SQL relations, databases, schemas. SQL relation R defines a tuple of distinct attributes $\mathsf{R.Ats} = (at_1, \ldots, at_n)$. Each attribute is a bitstring $at \in \{0,1\}^*$ and represents a "column" in the relation. R also defines a table R.T consisting of n-tuples of bitstrings representing the "rows" in the relation. Given a row $(x_1, \ldots, x_n) = \mathbf{r} \in \mathsf{R.T}$, we refer to the i-th entry of the row with $\mathbf{r}[at_i] = x_i$. We can initialize a relation with $\mathsf{NewRltn}(\mathbf{at})$ which returns the relation with $\mathsf{R.Ats} = \mathbf{at}$ and no rows.

We define a *database* to be a set of relations with disjoint attributes and their (distinct) identifiers, i.e. a set of the form $\mathsf{DB} = \{(id_1, \mathsf{R}_1), \ldots, (id_N, \mathsf{R}_N)\}$ where $i \neq j$ implies $id_i \neq id_j$ and $\mathsf{R}_i.\mathsf{Ats} \cap \mathsf{R}_j.\mathsf{Ats} = \emptyset$. We denote the identifier set of such a database as $\mathsf{DB.IDs} = \{id_i\}_{i \in [N]}$ and retrieve relations by identifier using $\mathsf{DB}[id_i] = \mathsf{R}_i$. Since database attributes are non-repeating, we allow the retrieval of a table by any of its attributes using getID (i.e. if $\mathsf{getID}(at, \mathsf{DB}) = id$ then $at \in \mathsf{DB}[id].\mathsf{Ats}$). Similarly, if $\mathbf{t} \subseteq \mathsf{DB}[id].\mathsf{Ats}$, then $\mathsf{getID}(\mathbf{t}, \mathsf{DB}) = id$.

We require that each $(id, \mathsf{R}) \in \mathsf{DB}$ has a *unique key attribute* $\mathsf{uk}(id) \in \mathsf{R.Ats}$. This functions as a "row number" which uniquely identifies each row. In other words, for all distinct $\mathbf{r}, \mathbf{r}' \in \mathsf{R.T}$, we have $\mathbf{r}[\mathsf{uk}(id)] \neq$

$\mathsf{R}_1.\mathsf{T}$

$\mathsf{uk}(id_1)$	at_1
aa	Alice
bb	Alice
cc	Bob
dd	Charlie
ee	David

$\mathsf{R}_2.\mathsf{T}$

$\mathsf{uk}(id_2)$	at_2	at_3
11	Alice	Math
22	Alice	Chem
33	Bob	CS
44	Eve	CS
55	Eve	Bio

$(\mathsf{R}_1 \bowtie_{at_1,at_2} \mathsf{R}_2).\mathsf{T}$

$\mathsf{uk}(id_1)$	at_1	$\mathsf{uk}(id_2)$	at_2	at_3
aa	Alice	11	Alice	Math
aa	Alice	22	Alice	Chem
bb	Alice	11	Alice	Math
bb	Alice	22	Alice	Chem
cc	Bob	33	Bob	CS

$(\sigma_{at_2,\texttt{Eve}}(\mathsf{R}_2)).\mathsf{T}$

$\mathsf{uk}(id_2)$	at_2	at_3
44	Eve	CS
55	Eve	Bio

$(\sigma_{at_3,\texttt{CS}}(\mathsf{R}_1 \bowtie_{at_1,at_2} \mathsf{R}_2)).\mathsf{T}$

$\mathsf{uk}(id_1)$	at_1	$\mathsf{uk}(id_2)$	at_2	at_3
cc	Bob	33	Bob	CS

Fig. 3. Examples of SQL relations $\mathsf{R}_1, \mathsf{R}_2$ and the output of join ($\bowtie$) and select (σ) operations on them.

$\mathbf{r}[\mathsf{uk}(id')]$. Given some $\mathbf{r} \in \mathsf{DB}[id]$ we refer to the tuple $(id, \mathbf{r}[\mathsf{uk}(id)])$ as its *coordinates* and note that it uniquely identifies that row within the database. Additionally, we refer to the values in a "column" with $\mathsf{rng}(at, \mathsf{DB}) = \{\mathbf{r}[at] : \mathbf{r} \in \mathsf{DB}[\mathsf{getID}(at, \mathsf{DB})]\}$.

A database's schema communicates all information about DB except the tables: $\mathsf{Schema}(\mathsf{DB}) = \{(id, \mathsf{R.Ats}) : (id, \mathsf{R}) \in \mathsf{DB}\}$. As shorthand, if $\mathsf{scma} = \mathsf{Schema}(\mathsf{DB})$ then $\mathsf{scma}[id] = \mathsf{DB}[id].\mathsf{Ats}$ and $\mathsf{getID}(at, \mathsf{scma}) = \mathsf{getID}(at, \mathsf{DB})$. In our schemes, the client stores $\mathsf{Schema}(\mathsf{DB})$ as part of the key in order to appropriately format data returned by the server. This is a result of our explicit handling of schemas, coordinates and attributes, something which was left implicit in prior work.

SQL OPERATIONS. In our work, we address the secure computation of SQL (equi)joins and (equality) selections. These operations work as follows.

The selection operation is parametrized by a pair of bitstrings (at, x), takes a relation R_1 with $at \in \mathsf{R.Ats}$ as input, and returns $\mathsf{R} = \sigma_{(at,x)}(\mathsf{R}_1)$ where:

$$\mathsf{R.Ats} = \mathsf{R}_1.\mathsf{Ats} \text{ and } \mathsf{R.T} = \{\mathbf{r} \in \mathsf{R}_1.\mathsf{T} : \mathbf{r}[at] = x\}.$$

In Fig. 3, we provide an example of such a selection on a relation in a database.

The join infix operation is a function parametrized by two equal-length tuples of attributes $\mathbf{t}_1, \mathbf{t}_2$. It takes two relations $\mathsf{R}_1, \mathsf{R}_2$ with disjoint attribute sets where $(at_1^i, \ldots, at_n^i) = \mathbf{t}_i \subseteq \mathsf{R}_i.\mathsf{Ats}$. It returns $\mathsf{R} = \mathsf{R}_1 \bowtie_{\mathbf{t}_1,\mathbf{t}_2} \mathsf{R}_2$ where:

$$\mathsf{R.Ats} = \mathsf{R}_1.\mathsf{Ats} \| \mathsf{R}_2.\mathsf{Ats} \qquad \text{and} \qquad \mathsf{R.T} = \{\mathbf{r}_1 \| \mathbf{r}_2 : \mathbf{r}_1 \in \mathsf{R}_1.\mathsf{T},\ \mathbf{r}_2 \in \mathsf{R}_2.\mathsf{T}, \forall i \in [n], \mathbf{r}_1[at_i^1] = \mathbf{r}_2[at_i^2]\}.$$

In the case of a join on singleton tuples, we abbreviate $\bowtie_{(at),(at')}$ as $\bowtie_{at,at'}$. In Fig. 3, we provide an example such a join. Attribute tuples can be empty in which case it returns the Cartesian product of the input rows. This is also known as the "cross" operation $\times$.

ADT FOR SQL DATABASES. We say that an ADT SqlDT is a *SQL data type* if its domain elements $\mathsf{DB} \in \mathsf{SqlDT}$ are *SQL databases* which take the form $\mathbf{DB} = (\mathsf{DB}, \alpha)$ where DB is as defined in Section 3.1 and $\alpha \in \{0,1\}^*$ is the auxiliary data. The purpose of α is to allow annotations on DB consistent with real world applications. In this work, we use α to indicate the allowed joins, and $\mathsf{SqlDT.Spec}$ always returns either a relation or $\perp$.

3.2 Constructing StE for SQL Data Types Using Encrypted Indexes

Our end goal is structurally encrypted databases supporting response-hiding SQL queries. We build these by constructing StI schemes for classes of SQL queries, then converting these into StE schemes for SQL data types via a generic transform. We now describe this conversion, then dedicate the remainder of this work to the abovementioned StI schemes.

StE, StI FOR SqlDT. Intuitively, our StE schemes handle the indexing and storage of SQL data separately. We do the former with an StI scheme and the latter with an RH dictionary encryption scheme. This modularization simplifies pseudocode and reduces the problem of designing secure StE schemes to that of StI schemes.

Alg $\mathsf{StE.Enc}(K'_{\mathrm{i}}, \mathbf{DB})$
$(K_{\mathrm{d}}, ED, \mathsf{DS}) \leftarrow\!\$\ \mathsf{EncRows}(\mathbf{DB})$
$(K_{\mathrm{i}}, \mathsf{IX}) \leftarrow\!\$\ \mathsf{StI.Enc}(K'_{\mathrm{i}}, \mathsf{DS})$
Return $\big((K_{\mathrm{d}}, K_{\mathrm{i}}), (ED, \mathsf{IX})\big)$

Subroutine $\mathsf{EncRows}\big((\mathsf{DB}, \alpha)\big)$
For $(id, \mathsf{R}) \in \mathsf{DB}$ do
  For $\mathbf{r} \in \mathsf{R.T}$ do
    $\mathbf{D}[(id, \mathbf{r}[\mathsf{uk}(id)])] \leftarrow \mathbf{r}$
$(K_{\mathrm{d}}, ED) \leftarrow\!\$\ \mathsf{Dye}_\pi.\mathsf{Enc}(\mathbf{D})$
For $(id, \mathsf{R}) \in \mathsf{DB}$ do
  For $\mathbf{r} \in \mathsf{R.T}$ do
    $\ell \leftarrow (id, \mathbf{r}[\mathsf{uk}(id)])$
    $\mathbf{T}[\ell] \leftarrow \mathsf{Dye}_\pi.\mathsf{Tok}\big(K_{\mathrm{d}}, \ell\big)$
Return $\big(K_{\mathrm{d}}, ED, (\mathsf{DB}, \alpha, \mathbf{T})\big)$

Alg $\mathsf{StE.Tok}\big((K_{\mathrm{d}}, K_{\mathrm{i}}), q\big)$
$\mathsf{tk} \leftarrow\!\$\ \mathsf{StI.Tok}(K_{\mathrm{i}}, q)$; Return tk

Alg $\mathsf{StE.Eval}\big(\mathsf{tk}, (ED, \mathsf{IX})\big)$
$P \leftarrow \mathsf{StI.Eval}(\mathsf{tk}, \mathsf{IX})$; Return $\mathsf{EvalRows}(P, ED)$

Subroutine $\mathsf{EvalRows}\big((P_1, \ldots, P_n), ED\big)$
For $i \in [n]$ do
  $C_i \leftarrow \{(c_1, \ldots, c_{n'}) : (\mathsf{rt}_1, \ldots, \mathsf{rt}_{n'}) \in P_i\ ,\ c_j = \mathsf{Dye}_\pi.\mathsf{Eval}(\mathsf{rt}_j, ED)\}$
$C \leftarrow (C_1, \ldots, C_n)$; Return C

Alg $\mathsf{StE.Dec}\big((K_{\mathrm{d}}, K_{\mathrm{i}}), q, C\big)$
Return $\mathsf{StI.Fin}(K_{\mathrm{i}}, q, \mathsf{DecRows}(K_{\mathrm{d}}, C))$

Subroutine $\mathsf{DecRows}\big(K_{\mathrm{d}}, (C_1, \ldots, C_n)\big)$
For $i \in [n]$ do
  $M_i \leftarrow \{(m_1, \ldots, m_{n'}) : (c_1, \ldots, c_{n'}) \in P_i\ ,\ m_j = \mathsf{Dye}_\pi.\mathsf{Eval}(\mathsf{rt}_j, ED)\}$
$M \leftarrow (M_1, \ldots, M_n)$; Return M

Alg $\mathcal{L}\big(\mathbf{DB}, (q_1, ..., q_n)\big)$
$(K, ED, \mathsf{DS}) \leftarrow\!\$\ \mathsf{EncRows}(\mathbf{DB})$ using a random function in place of $\mathsf{F.Ev}(K_{\mathrm{f}}, \cdot)$
$lk^{\mathrm{i}} \leftarrow\!\$\ \mathcal{L}^{\mathrm{i}}(\mathsf{DS}, (q_1, \ldots, q_n))$
Return (lk^{i}, N, L) where L, N are the max row length and # of rows in $\mathbf{DB}$

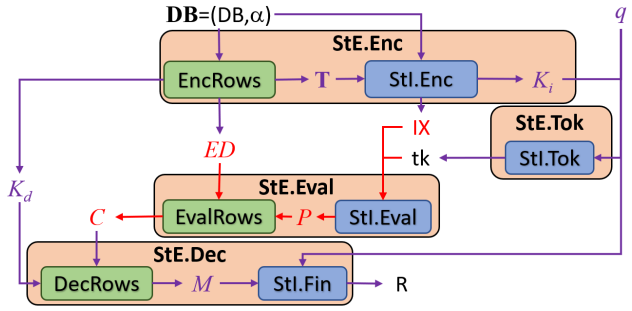

Fig. 4. Algorithms and for structured encryption scheme $\mathsf{StE} = \mathbf{SqlStE}[\mathsf{StI}, \mathsf{SE}, \mathsf{F}]$ expressed both in pseudocode (top) and diagrammatically (bottom left), and leakage algorithm for StE (bottom right). Here, Dye_π is the RH dictionary encryption scheme Dye_π in Appendix A (which uses SE, F as primitives) and $\mathcal{L}^{\mathrm{i}}$ is StI's leakage profile.

More formally, we construct an StE scheme for SQL data type SqlDT using the transform **SqlStE** which which takes uses an StI scheme for SqlDT_1 (described below), symmetric encryption scheme SE and function family F. We capture the syntax and pseudocode of StE's algorithms in Fig. 4. Note that $\mathsf{StE.KS} = \mathsf{StI.KS}$ and Dye_π is the RH dictionary encryption scheme given in Appendix A which uses SE, F as primitives. It is used in $\mathsf{EncRows}, \mathsf{EvalRows}, \mathsf{DecRows}$, which encrypt, retrieve and decrypt the rows of database DB. We used a specific RH dictionary encryption scheme because pathological alternatives may introduce circular security issues, preventing a more general approach.

We now describe how the algorithms in StI and $\mathsf{StE} = \mathbf{SqlStE}[\mathsf{StI}, \mathsf{SE}, \mathsf{F}]$ work. During $\mathsf{StE.Enc}$, algorithm $\mathsf{EncRows}$ will store the rows of DB in an encrypted dictionary ED using $\mathsf{Dye}_\pi.\mathsf{Enc}$. It also prepares a token dictionary $\mathbf{T}$ which maps each row coordinate to a token for Dye_π. SQL data type SqlDT_1 is the same as SqlDT except that its domain elements now take the form $\mathsf{DS} = (\mathsf{DB}, \alpha, \mathbf{T})$ where $(\mathsf{DB}, \alpha) \in \mathsf{SqlDT.Dom}$. The output of $\mathsf{StE.Enc}$ is ED and the index returned by $\mathsf{StI.Enc}(\mathsf{DS})$.

StE's tokens are those generated by StI. As such, the server's first step in $\mathsf{StE.Eval}$ is to run $\mathsf{StI.Eval}$. We require that $\mathsf{StI.Eval}$ returns a *pointer tuple* $P = (P_1, \ldots, P_n)$ which is a tuple of sets of tokens. The tokens in each P_i come from $\mathbf{T}$ and point to rows from the same table. Algorithm $\mathsf{EvalRows}$ replaces each token with relevant (encrypted) row from ED and returns *ciphertext tuple* $C = (C_1, \ldots, C_n)$, the output of $\mathsf{StE.Eval}$.

During $\mathsf{StE.Dec}$, algorithm $\mathsf{DecRows}$ decrypts each ciphertext to get *plaintext tuple* $M = (M_1, \ldots, M_n)$. $\mathsf{StI.Fin}$ takes these decrypted rows and performs any final client-side post-processing, returning the final output relation R.

In this work, we will define three different SQL data types, each with its own StI scheme(s). To demonstrate that all of these can be used to construct secure RH StE for their respective data type via **SqlStE**,

we demonstrate that the semantic security of StE reduces to that of its primitives. The proof follows from a standard hybrid argument and is given in Appendix B.

Theorem 1. *Let* $\mathsf{StE} = \mathbf{SqlStE}[\mathsf{StI}, \mathsf{SE}, \mathsf{F}]$ *be a correct StE scheme for* SqlDT*. Then given algorithms* $\mathcal{L}^{\mathrm{i}}, \mathcal{S}^{\mathrm{i}}$ *and adversary* A *we can define* $\mathcal{L}$ *as in Fig. 4 and construct* $\mathcal{S}, A_{\mathrm{s}}, A_{\mathrm{f}}, A_{\mathrm{i}}$ *such that:*

$$\mathbf{Adv}^{\mathrm{ss}}_{\mathsf{StE},\mathcal{L},\mathcal{S}}(A) \leq \mathbf{Adv}^{\mathrm{ind\$}}_{\mathsf{SE}}(A_{\mathrm{s}}) + \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(A_{\mathrm{f}}) + \mathbf{Adv}^{\mathrm{ss}}_{\mathsf{StI},\mathcal{L}^{\mathrm{i}},\mathcal{S}^{\mathrm{i}}}(A_{\mathrm{i}}).$$

4 Partially Precomputed Joins

We demonstrate our framework from Section 3 in action with two SQL data types: JnDT and SjDT. The former only supports non-recursive join queries and is presented for the purpose of introducing partially precomputed (PP) join indexing. The latter allows recursive queries, cluster joins and equality selections, and demonstrates how OPX's techniques can be modified to use PP joins.

4.1 Indexing of Non-Recursive Joins

JOIN DATA TYPE JnDT. We define $\mathsf{JnDT.Dom}$ to contain (DB, α) such that DB is a database and α is the set of join queries supported (i.e. if A is the set of attributes in DB that are not unique key attributes, then $\alpha \subseteq \{(at_1, at_2) \in A \times A : \mathsf{getID}(at_1, \mathsf{DB}) \neq \mathsf{getID}(at_2, \mathsf{DB})\}$. Our goal here is to capture SQL queries of the form "id_1 **join** id_2 **on** $at_1 = at_2$" where $id_1, id_2 \in \mathsf{DB.IDs}$ and $at_i \in \mathsf{DB}[id_i].\mathsf{Ats}$.

We allow queries to be any pair of attributes (i.e. $\mathsf{JnDT.QS} = \{(at_1, at_2) : at_i \in \{0,1\}^*\}$), but $\mathsf{JnDT.Spec}$ only computes the join if $(at_1, at_2) \in \alpha$:

$$\mathsf{JnDT.Spec}\big((at_1, at_2), (\mathsf{DB}, \alpha)\big) = \mathsf{DB}[\mathsf{getID}(at_1, \mathsf{DB})] \bowtie_{at_1, at_2} \mathsf{DB}[\mathsf{getID}(at_2, \mathsf{DB})]$$

and returns $\perp$ otherwise. From here on we assume that all queries made are "non-trivial" meaning they return relations with at least one row.

FP INDEXING. FpJn is an StI scheme that "fully precomputes" joins and is modeled after SPX's handling of "type-2 selections" and OPX's handling of "leaf joins" [32,34]. The intuition here is that the output relation for each possible join query is precomputed and pointers to the rows therein are stored as an entry in a RR encrypted multimap. FpJn's detailed algorithms and leakage profile are given in Fig. 5. Note that $\mathsf{FpJn.KS} = \mathsf{Mme.KS}$ and that each row in the output of a particular join is indexed as a pair of pointers to rows in DB.

Since join queries are handled directly by Mme the leakage and efficiency of FpJn depends entirely on Mme. For the rest of this discussion, we will assume Mme is one of the mainstream multimap encryption schemes (e.g. [12,18,15]) with the "standard" leakage profile consisting the label space size $|\mathbf{M}.\mathsf{Lbls}|$, multimap size $\sum_{\ell \in \mathbf{M}.\mathsf{Lbls}} |\mathbf{M}[\ell]|$, query pattern (equality pattern of queries $\ell_1, \ldots, \ell_n$) and query responses $\mathbf{M}[\ell_1], \ldots, \mathbf{M}[\ell_n]$.

Notice that when a query (at_1, at_2) is made in FpJn, the query responses reveal the equality pattern of columns at_1, at_2 for rows that appear in the join output. To illustrate, if the query is made on $\mathsf{DB} = \{(id_1, \mathsf{R}_1), (id_2, \mathsf{R}_2)\}$ where $\mathsf{R}_1, \mathsf{R}_2$ are as depicted in Fig. 3, the server learns that the first two rows of each R_i all have the same value in their at_1, at_2 columns, but won't reveal anything about the last two rows of each R_i apart from the fact that they are not returned in the join. Note that in the worst case, the join returns all rows from both relations and the search pattern leakage reveals the entire equality pattern of both columns. This leakage is comparable to PRE-based techniques like deterministic encryption or adjustable joins (an observation also made by DPPS [20]). This is significant because, as discussed in Section 1, LAAs are highly effective against PRE and can be applied in this case. Beyond the worst case, FP indexing leaks strictly less than PRE-based solutions but this does not make them immune to LAAs. In particular, we believe that attacks (such as those using ℓ_p-optimization or graph matching [37,8]) can be extended to make use of partial equality patterns and cross column correlations, and be effective against FpJn's leakage.

We also note that FpJn achieves lower leakage than the analogous indexing in SPX or OPX because it uses a single multimap. The latter schemes had one encrypted multimap for each attribute (i.e. $\mathbf{M}_{at_1}$ indexes all joins $(at_1, at_2) \in \alpha$) this leaks additional metadata and tells the adversary when two queries join on the same at_1.

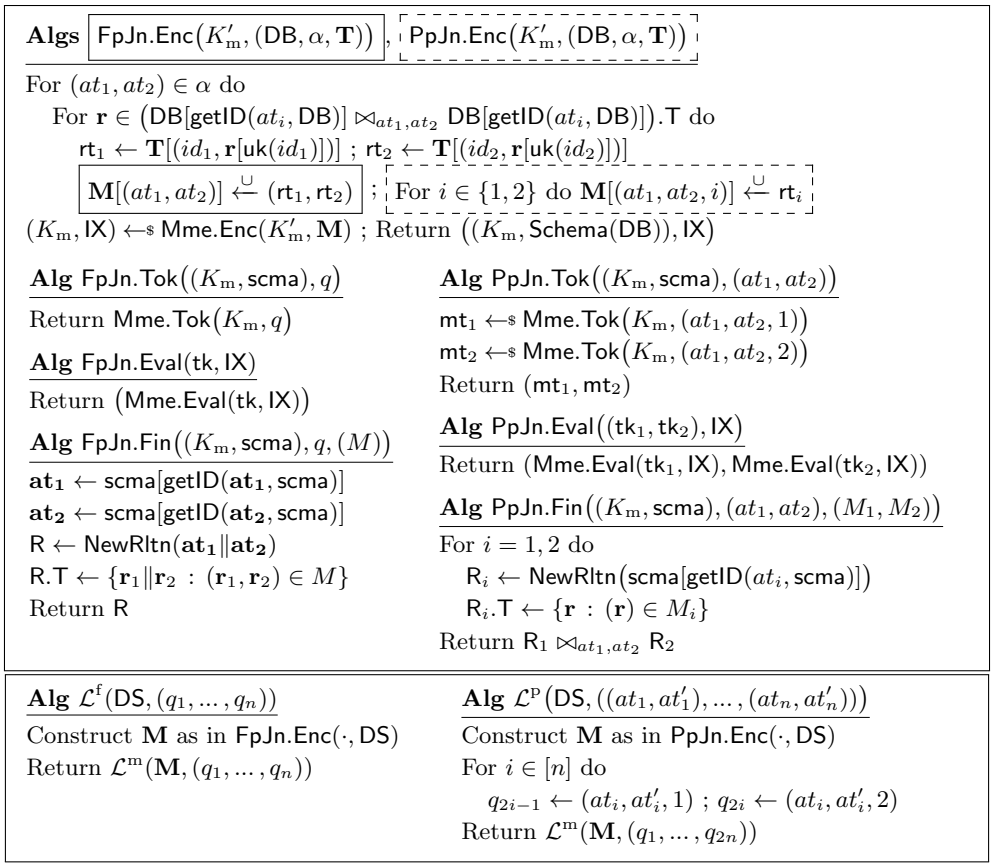

Fig. 5. Algorithms of StI schemes FpJn, PpJn (top) and their leakage algorithms (bottom) where Mme is a RR multimap encryption scheme. Note that in the encryption algorithm, boxed code belongs only to the respective algorithm.

PP INDEXING. We introduce a new StI scheme PpJn which performs "partially precomputed" indexing, whose algorithms are also depicted in Fig. 5. PpJn.Enc proceeds in the same way as FpJn.Enc but we store the rows from each input relation separately. In other words, if $\mathbf{M}_f, \mathbf{M}_p$ are the multimaps constructed in the respective setup algorithms, then $\mathbf{M}_p[(at_1, at_2, i)] = \{\mathsf{rt}_i : (\mathsf{rt}_1, \mathsf{rt}_2) \in \mathbf{M}_f[(at_1, at_2)]\}$ for $i = 1, 2$ and $(at_1, at_2) \in \alpha$. Notice that this means the client needs to reassemble the output relation from the two sets of rows in StI.Fin. We recommend that the client do so by sorting then joining the columns, avoiding the quadratic time nested loop join where rows are compared pairwise.

This small change in indexing technique has substantial impact on bandwidth and security. In the worst case, the number of rows sent with FP is quadratic while PP's is linear. This bandwidth reduction occurs because two sets of rows are sent instead of their cross product. Notice that modulo some metadata information (i.e. the multimap sizes), the PP leakage can be derived from the FP leakage meaning that PP indexing is no worse than FP indexing. In fact, if more than one row is returned to any query PP leakage is strictly lower. To illustrate, when join query (at_1, at_2) is made to the aforementioned database in Fig. 3, the adversary sees that the first three rows of both tables were returned and can infer that each row has at least one matching value in the other column – nothing specific about their equality patterns.

In summary, PpJn is the superior indexing choice for JnDT because its leakage is strictly lower, bandwidth is no worse and efficiency is comparable.

SEMANTIC SECURITY. The security of FpJn, PpJn reduce to that of Mme. The proof follows directly from the definition of Mme's semantic security and is given in Appendix C.

Theorem 2. *Let $\mathcal{L}, \mathcal{S}$ be the leakage algorithm and simulator for* Mme. *Let $\mathcal{L}^f, \mathcal{L}^p$ be the leakage algorithms given in Fig. 5. Then, given adversary A these exists adversary A_m and simualtor $\mathcal{S}^p$ such that:*

$$\mathbf{Adv}^{ss}_{\mathsf{FpJn},\mathcal{L}^f,\mathcal{S}}(A) \le \mathbf{Adv}^{ss}_{\mathsf{Mme},\mathcal{L},\mathcal{S}}(A) \text{ and } \mathbf{Adv}^{ss}_{\mathsf{PpJn},\mathcal{L}^p,\mathcal{S}^p}(A) \le \mathbf{Adv}^{ss}_{\mathsf{Mme},\mathcal{L},\mathcal{S}}(A_m).$$

4.2 PP indexing for recursive queries

SjDT. We expand the query support of JnDT to include equality selections, cluster joins (joins on more than one attribute) and recursively defined queries. The resultant query class is similar to the SPJ algebra defined by CM [14] except for the omission of the projection operation which we note can be handled as a post-processing step requiring no cryptographic techniques.

We capture this via the SQL data type SjDT. Its domain is unchanged from JnDT.Dom except that α allows tuple pairs in addition to attribute pairs. Below we describe the forms, evaluation and SQL equivalent of $q \in \mathsf{SjDT.QS}$. (Note that the $\mathtt{r}, \mathtt{s}, \mathtt{j}$ flags are included in SjDT queries for domain separation.) These are defined recursively so $q_i, \mathbf{q}_i$ are themselves queries of the respective type.

Query Type	SjDT query q	$\mathsf{SjDT.Spec}(q, \mathbf{DB})$	SQL query $\mathbf{q}$
Relation retrieval	$(\mathtt{r}, id)$	$\mathsf{DB}[id]$ where $\mathbf{DB} = (\mathsf{DB}, \alpha)$	**select** * **from** id
(Equality) selections	$(\mathtt{s}, at, x, q_1)$	$\sigma_{(at,x)}(\mathsf{SjDT.Spec}(q_1, \mathbf{DB}))$	**select** * **from** $\mathbf{q}_1$ **where** $at = c$
(Equi)joins	$(\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, q_1, q_2)$ where $(\mathbf{t}_1, \mathbf{t}_2) \in \alpha$	$(\mathsf{SjDT.Spec}(q_1, \mathbf{DB})) \bowtie_{\mathbf{t}_1, \mathbf{t}_2} (\mathsf{SjDT.Spec}(q_2, \mathbf{DB}))$	**select** * **from** $\mathbf{q}_1$ **join** $\mathbf{q}_2$ **on** $\mathbf{t}_1 = \mathbf{t}_2$

We say that queries of the form $(\mathtt{r}, id)$, $(\mathtt{s}, at, x, (\mathtt{r}, id))$ or $(\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, (\mathtt{r}, id_1), (\mathtt{r}, id_2))$ are *non-recursive* and all others are *recursive*. We require that all attributes in each $\mathbf{t}_i$ come from the same relation in DB (i.e. $\mathbf{t}_i \subseteq \mathsf{DB}[id].\mathsf{Ats}$ for some $id \in \mathsf{DB.IDs}$). While allowing cluster joins may lead to an exponential-size index, a judicious database administrator would not allow this – cluster joins are rarely used and usually known in advance.

Hashset filtering. To minimize the leakage of recursive queries in our StI schemes we employ the filtering hashset technique introduced in OXT [13]. We now review this technique and establish some notation for it.

This filtering hashset is a set denoted HS containing outputs of a function family F where $\mathsf{F.KS} = \{0,1\}^{\mathsf{F.ol}}$. In our algorithms, the hashset will be used to associate predicate bitstrings with a row tokens (from $\mathbf{T}$). Later, given a predicate p's key $K = \mathsf{F.Ev}(K_\mathrm{f}, p)$ we can filter a set of row tokens, retaining only those which satisfy the predicate. We formalize this via the following algorithms:

Alg $\mathsf{HsEnc}(K_\mathrm{f}, \mathsf{SET})$
For $(p, \mathsf{rt}) \in \mathsf{SET}$ do
  $\mathsf{HS} \xleftarrow{\cup} \mathsf{F.Ev}(\mathsf{F.Ev}(K_\mathrm{f}, p), \mathsf{rt})$
Return HS

Alg $\mathsf{HsFilter}(K, (P_1, \ldots, P_n), \mathsf{HS})$
For $i \in [n]$ do
  For $\mathsf{rt} \in \mathbf{rt} \in P_i$ if $\mathsf{F.Ev}(K, \mathsf{rt}) \in \mathsf{HS}$ then $S \xleftarrow{\cup} \mathbf{rt}$
  If $S \neq \emptyset$ then $P_i \leftarrow S$
Return $(P_1, \ldots, P_n)$

For notational convenience in our pseudocodes, HsFilter takes as input a tuple set $P = (P_1, \ldots, P_n)$. It then attempts to filter each P_i and retain only the tuples where at least one rt satisfies the predicate. However, if no such tuple exists, it does not perform the filtering at all.

PP indexing for SjDT. We are now ready to extend the PP indexing technique introduced in Section 4 to construct StI for SjDT. On a high level, we do so by using an inverted index (similar to those used for SSE) to handle selections and a filtering hashset to handle recursively defined queries. The result is StI scheme PpSj whose algorithms are depicted in Fig. 6.

Now we provide some intuition for PpSj's algorithms. The scheme has two server-side data structures: an encrypted multimap and a hashset. The multimap is used to index non-recursive queries my mapping a query-derived label to the relevant rows in the database. For example, the label for relation retrieval query $(\mathtt{r}, id)$ is the query itself and its values are row tokens associated to rows in $\mathsf{DB}[id]$ (i.e. $\{(\mathbf{T}[(id, \mathbf{r}[\mathsf{uk}(id)])]) : \mathbf{r} \in \mathsf{DB}[id].\mathsf{T}\}$). Note that the latter are singleton tuples because we required that pointer tuples be made out of tuples of tokens. The hashset is used to filter the sets in a pointer tuple during a recursive query. For example, when processing the query $(\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, (\mathtt{s}, at, x, (\mathtt{r}, id_1)), (\mathtt{r}, id_2))$ (a select followed by a join), the server would use the multimap to retrieve row tokens for each of the non-recursive subqueries (i.e. $(\mathtt{s}, at, x, (\mathtt{r}, id_1))$ and $(\mathtt{r}, id_2)$). The token would also include two keys which can be used with HsFilter which tests if the rows being pointed to (in $\mathsf{DB}[id_1]$ or $\mathsf{DB}[id_2]$) are in $\mathsf{DB}[id_1] \bowtie_{\mathbf{t}_1, \mathbf{t}_2} \mathsf{DB}[id_2]$.

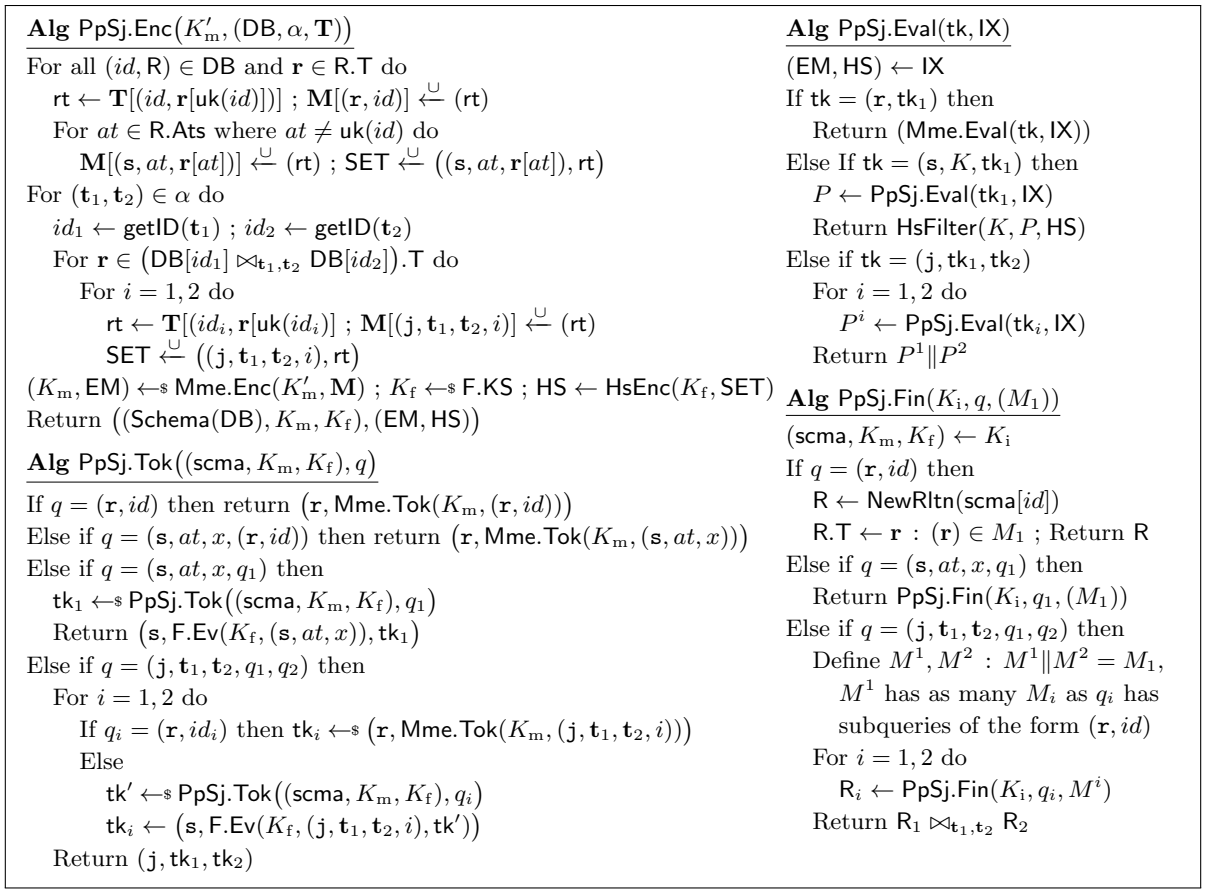

Fig. 6. Algorithms for PpSj the StI scheme for SjDT using PP indexing.

FP INDEXING FOR SjDT. We analogously extend FpJn introduced in Section 4 to construct FpSj, an StI for SjDT. Just like with PpSj, non-recursive queries will be added to the encrypted multimap that is used to index the non-recursive joins while all recursive queries are filtered using the hashset. The only subtlety in this extension is the handling of "internal joins" which are queries of the form $q = (\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, (\mathbf{r}, id), q_1)$ (or $q = (\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, q_1, (\mathbf{r}, id))$) because we want to limit the row tokens leaked from id to those who join with some row returned by q_1. Similar to OPX, we construct an index where each row token returned in the subquery will "point to" the tokens of the rows joined to in $\mathsf{DB}[id]$. As alluded to in Section 3.2, this self-referential indexing (where Mme tokens are stored in $\mathbf{M}$) may introduce circular security issues if pathological Mme primitives are used. We avoid this by indexing internal joins with a specific, non-pathological primitive (as was done in OPX). To avoid the increased leakage and complexity of an additional data structure, we will assume that Mme is the $\mathsf{Mme}^{\mathrm{rr}}_{\pi}$ scheme recounted in Appendix A and co-locate this index with the one used for non-recursive queries. Notice that this subtlety does not come up in PpSj because we do not reveal join equality patterns so all recursive joins can be handled similarly.

The StE scheme $\mathsf{StE} = \mathbf{SqlStE}[\mathsf{FpSj}, \mathsf{SE}, \mathsf{F}]$ is essentially the same as OPX with minor improvements in leakage (analogous to those described in our discussion of FpJn in Section 4) and a slightly revised approach to "internal joins". For this reason, we defer a complete description of FpSj to Appendix E.

PpSj LEAKAGE PROFILE. While a pseudocode description of PpSj's leakage profile may seem convoluted, we believe the intuition behind it enables helpful comparisons with FpSj and OPX [34]. As such, we aim to give some intuition by describing the components of PpSj's leakage profile via a running example, deferring a full description of PpSj's leakage algorithm and the associated security proof to Appendix D. Below, we assume that MME primitives have the "standard" leakage profile (as described in Appendix A).

Our example database contains $\mathsf{R}_1, \mathsf{R}_2$ from Fig. 3. If no queries are made, the server-side data structures reveal only *metadata leakage*. This includes the number of values in the multimap, the maximum-length of

a value in the multimap and the number of F outputs in the hashset. The leakage of FpSj is comparable but on OPX it is higher because different data structures are used to index different SQL operations.

We will refer to all other forms of leakage as "query dependent leakage". This is where PP indexing has substantial savings over FP and OPX.

Now lets assume the client makes the following queries: $q_1 = (\mathtt{s}, at_3, \mathtt{CS}, (\mathtt{r}, id_2))$, $q_2 = (\mathtt{s}, at_2, \mathtt{Eve}, (\mathtt{r}, id_2))$, $q_3 = (\mathtt{r}, id_1)$ and $q_4 = \big(\mathtt{j}, at_1, at_2, (\mathtt{r}, id_1), (\mathtt{s}, at_3, \mathtt{CS}, (\mathtt{r}, id_2))\big)$. The server will receive four tokens, where $\mathsf{tk}_1, \mathsf{tk}_2, \mathsf{tk}_3$ are such that $\mathsf{tk}_i = (\mathtt{r}, \mathsf{mt}_i)$ and $\mathsf{tk}_4 = \big(\mathtt{j}, (\mathtt{r}, \mathsf{mt}_4), (\mathtt{j}, K, \mathsf{mt}_5)\big)$. Here, each mt_i is a token for Mme while K is a hashset key. Just from inspecting these, the adversary learns the *recursion structure* of the queries. Specifically, he learns that the first three queries were non-recursive while q_4 was a join followed by a select. This leakage is slightly lower in $\mathsf{FpSj}, \mathsf{PpSj}$ compared to OPX because the adversary cannot differentiate between non-recursive selections and relation retrievals.

The Mme tokens leak *the multimap query pattern* and *multimap responses*. The former reveals whenever the associated query or subquery is repeated. In our case, the adversary learns that $\mathsf{mt}_1, \mathsf{mt}_5$ are associated to the same query. Note that this does not extend to $\mathsf{mt}_3, \mathsf{mt}_4$ because the latter is in a join. From the multimap query responses he "sees" the row tokens that are returned by each $\mathsf{Mme.Eval}(\mathsf{mt}_i, \mathsf{EM})$. This reveals the equality pattern of the rows returned by each associated query/subquery. For example, this reveals that q_1, q_2 both return two rows, one of which is shared. On join queries, we enjoy similar leakage savings as described in the non-recursive case. For example, tk_4 will reveal that three rows are returned from the left relation (i.e. id_1) but doesn't say anything about whether they are in the final output relation or how they "match up" with rows from the right relation. In FpSj and OPX, both of the above are revealed.

Finally, the hashset keys reveal the *hashset key query pattern* and *hashset filtering results*. The former reveals when the exact same predicate is repeated and is detectable because the keys would be the same. The latter is because the adversary is free to apply hashset keys (in the tokens) to filter all the row tokens he can retrieve from EM thereby learning the *hashset filtering results*. This means that using K he can learn that one row returned by q_2 satisfies the predicate associated to K even though it is not in the output of q_4. Similarly, he learns that two rows returned by q_1 satisfies the predicate but only one is returned by q_4. Using FpSj the adversary would additionally learn which row returned in q_3 is "paired up" with this row in the q_4 output.

Leakage comparison. From the above discussion, one might expect decreasing query-dependent leakage from PpSj to FpSj to OPX. While the leakage for FpSj can always be derived from OPX, the comparison of PpSj to FpSj is not as straightforward because they sometimes do not return the same rows when recursive queries are made (which we discuss in more detail below).

However, when restricted to non-recursive queries, PpSj's query-dependent leakage is strictly superior for the same reasons that PpJn was superior in Section 4. Extending this, we can upper bound the leakage lk_{p} of PpSj on queries $q_1, \ldots, q_n$ with its leakage lk'_{p} on the minimal set of non-recursive queries $q'_1, \ldots, q'_m$ with which the server can still deduce the pointer tuples it should return on $q_1, \ldots, q_n$. Doing the same for FpSj, we have $lk_{\mathrm{f}} \leq lk'_{\mathrm{f}}$ as well. Then, via the above observation about non-recursive queries we have $lk'_{\mathrm{p}} \leq lk'_{\mathrm{f}}$, with the inequality being strict if at least one join query with at least two rows is made. Our being able to *bound* PpSj*'s query-dependent leakage lower than* FpSj*'s* gives credence to the intuition that PpSj is the more secure variant in practice.

Efficiency drawback of PpSj. Comparing the bandwidth of $\mathsf{PpSj}, \mathsf{FpSj}$ is also not clear cut: On non-recursive queries, PpSj will perform equal or better than FpSj but on recursive queries the converse is sometimes true.

Consider the query $q = (\mathtt{s}, at_3, \mathtt{CS}, (\mathtt{j}, at_1, at_2, (\mathtt{r}, id_1), (\mathtt{r}, id_2)))$ in our toy example. With FpSj, the server returns pointers to the two rows that feature in the output relation (i.e. those with coordinates $(id_1, \mathtt{cc}), (id_2, \mathtt{33})$) but PpSj returns four (i.e. with coordinates $(id_1, \mathtt{aa}), (id_1, \mathtt{bb}), (id_1, \mathtt{cc}), (id_2, \mathtt{33})$) because without the equality pattern over the join columns and it cannot filter out the first and second rows of R_1.

More generally, this overhead may occur anytime that a recursive query (involving at least one join) is made and grows with query complexity. Depending on the data and query workload, this overhead ranges from negligible to quite substantial, something we explore further in Section 6.

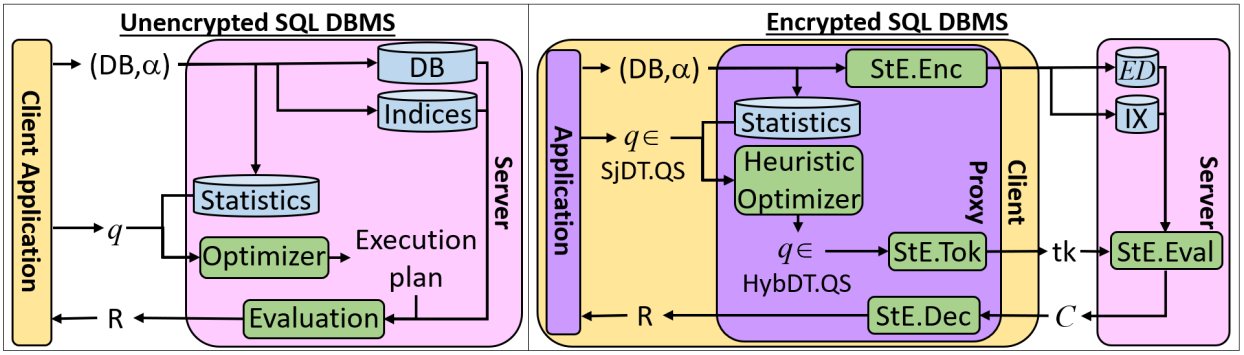

Fig. 7. Data/ query processing in unencrypted SQL databases (left) and the analogous processes using **SqlStE** with hybrid indexing (right).

5 Hybrid indexing

We showed that the choice between FP and PP indexing depends heavily on query load. This motivates our hybrid StI scheme that postpones this decision till query time. We first cover the technical details of supporting both indexing techniques, then give a heuristic for the client to choose between them.

HYBRID DATA PROCESSING. We give a new ADT where each join in a query is annotated with the desired indexing technique, HybDT. This ADT is equivalent to SjDT except that its join queries take the form $(op, \mathbf{t}_1, \mathbf{t}_2, q_1, q_2)$ where $op \in \{\mathtt{fp}, \mathtt{pp}\}$. When evaluating HybDT.Spec, these are both functionally equivalent to the analogous SjDT join query's $(\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, q_1, q_2)$.

The hybrid system we envision makes the same assumption as in (unencrypted) SQL DBMSes – that client queries are unoptimized and have no canonical form – and therefore mirrors its data flow as depicted in Fig. 7. It also borrows its architecture (i.e. use of a client-side proxy) from existing encrypted SQL solutions [39,43]. The client's SjDT query is annotated using a heuristic optimizer to get a HybDT query. This latter query is then tokenized, evaluated and decrypted using hybrid indexing scheme HybStI in StE = **SqlStE**[HybStI, SE, F].

As best we know, no existing work has looked into query optimization in StE schemes. We believe this area to be of independent interest because unlike encrypted systems where optimization runs on the server (with full access to the data) and is solely interested in maximizing efficiency, optimization in encrypted SQL DBMSes should be done (at least partially) by the proxy with only precomputed statistics about the data and may additionally seek to minimize leakage. We initiate this study with our heuristic below.

HybStI DETAILS. This StI merges FpSj, PpSj by essentially storing both kinds of indexes on the server. More specifically, HybStI.Enc will merge the multimaps and hashsets generated by PpSj, FpSj (avoiding repetition where possible) so that it can take join tokens of either form. When a HybDT join query is made, the client indicates which index to use in its query with *op*.

We believe the intuition for how HybStI works is straightforward, so we defer a full discussion on its pseudocode, leakage algorithm and security to Appendix F. The only subtlety comes when a query contains both FP and PP joins. Notice that pointer tuples in this case will contain more than one P_i (unlike FpSj) and the tuples in at least one P_i will contain more than one rt (unlike PpSj). As such, after the client performs the PP joins in HybStI.Fin some column reordering may be necessary.

HybStI LEAKAGE. We will describe HybStI's leakage profile in comparison to that of PpSj and FpSj. The metadata leakage is comparable, with each size (multimap or hashset) being the sum of respective FpSj and PpSj sizes. The recursion structure leakage is technically higher but only because we leak the join annotations that weren't present in the other two schemes.

For the same reason that PpSj and FpSj's query-dependent leakages were not directly comparable, they also cannot be compared with that of HybStI. However, like we did in Section 4.2, we can upper bound HybStI's query-dependent leakage on $q_1, \ldots, q_n \in$ HybDT.QS with that of $q'_1, \ldots, q'_m$, the minimal set of non-recursive queries in HybDT.QS (with consistent join annotation) with which the server can still compute its output on $q_1, \ldots, q_n$. This leakage is no better than the analogous bound in PpSj and no worse than that

of FpSj, this confirms the intuition that hybrid indexing achieves an *intermediate level of query-dependent leakage* compared to solely using FP or PP indexing.

Leakage-aware query planning. The join annotation selected by our query planning heuristic will minimize leakage without exceeding a predetermined bandwidth limit. More specifically, suppose the user supplies a query $q \in \mathsf{SjDT}$ with J joins and a bandwidth limit L indicating the maximum number of rows from ED that can be returned in the ciphertext tuple. We estimate the bandwidth of all possible HybDT queries, then select an annotation by:

1. Eliminating options which exceed L rows. If none remain, return $\perp$.
2. Maximize number of PP joins
3. If multiple choices remain, minimize bandwidth.

We argue that our setup is realistic because (1) we expect the J joins made in a query to be modest enough for the client to evaluate all 2^J HybDT queries, (2) bandwidth measurement can be reduced to the number of rows from ED sent as they are padded to the same length, and (3) is it common for SQL applications to limit bandwidth to prevent the client from maxing out its memory.

Alg $\mathsf{EvalBW}(q)$

If $q = (\mathtt{r}, id)$ then $\mathbf{B}[(id)] \leftarrow \mathcal{N}(id)$
Else if $q = (\mathtt{s}, at, x, q_1)$ then
  $\mathbf{B} \leftarrow \mathsf{EvalBW}(q_1)$; $id \leftarrow \mathsf{getID}(at, \mathsf{scma})$
  $\mathbf{B}[(id)] \leftarrow \mathbf{B}[(id)] \cdot \mathcal{H}_{at}(x)$
Else if $q = (op, \mathbf{t}_1, \mathbf{t}_2, q_1, q_2)$ then
  $\mathbf{B} \leftarrow \mathsf{EvalBW}(q_1) \cup \mathsf{EvalBW}(q_2)$
  For $i = 1, 2$
    Define $\mathbf{i_i} : \mathsf{getID}(\mathbf{t}_i, \mathsf{scma}) \in \mathbf{i_i} \in \mathbf{B}.\mathsf{Lbls}$
  If $op = \mathtt{fp}$ then
    $N \leftarrow \frac{\mathcal{F}(\mathbf{t_1}, \mathbf{t_2}) \cdot \mathbf{B}[\mathbf{i_1}] \cdot \mathbf{B}[\mathbf{i_2}]}{\mathcal{N}(\mathsf{getID}(\mathbf{t_1}, \mathsf{scma})) \cdot \mathcal{N}(\mathsf{getID}(\mathbf{t_2}, \mathsf{scma}))}$
    $\mathbf{B}[\mathbf{i_1} \| \mathbf{i_2}] \leftarrow \mathbf{2} \cdot \mathbf{N}$
    $\mathbf{B}[\mathbf{i_1}] \leftarrow \perp$; $\mathbf{B}[\mathbf{i_2}] \leftarrow \perp$
  Else if $op = \mathtt{pp}$ then
    For $i = 1, 2$ do
      $\mathbf{B}[\mathbf{i_i}] \leftarrow \mathcal{P}_{\mathbf{i}}(\mathbf{t_1}, \mathbf{t_2}) \cdot \frac{\mathbf{B}[\mathbf{i_i}]}{\mathcal{N}(\mathsf{getID}(\mathbf{t_i}, \mathsf{scma}))}$
Return $\mathbf{B}$

Schema scma
Return $\mathsf{Schema}(\mathsf{DB})$

Table size $\mathcal{N}(id)$
Return $|\mathsf{DB}[id].\mathsf{T}|$

Freq. histogram $\mathcal{H}_{at}(x)$
$\mathsf{R} \leftarrow \sigma_{(at,x)}(\mathsf{DB}[\mathsf{getID}(at, \mathsf{scma})])$
Return $\frac{|\mathsf{R.T}|}{\mathcal{N}(id)}$

FP join size $\mathcal{F}(\mathbf{t}_1, \mathbf{t}_2)$
For $i = 1, 2$ do $id_i \leftarrow \mathsf{getID}(\mathbf{t}_i, \mathsf{scma})$
$\mathsf{R} \leftarrow \mathsf{DB}[id_1] \bowtie_{\mathbf{t}_1, \mathbf{t}_2} \mathsf{DB}[id_2]$
Return $|\mathsf{R.T}|$

PP join sizes $\mathcal{P}_j(\mathbf{t}_1, \mathbf{t}_2)$
For $i = 1, 2$ do $id_i \leftarrow \mathsf{getID}(\mathbf{t}_j, \mathsf{scma})$
$\mathsf{R} \leftarrow \mathsf{DB}[id_1] \bowtie_{\mathbf{t}_1, \mathbf{t}_2} \mathsf{DB}[id_2]$
Return $|\{\mathbf{r}[\mathsf{uk}(id_j)] : \mathbf{r} \in \mathsf{R.T}\}|$

Fig. 8. EvalBW algorithm (left) defined in terms of precomputed statistics (right) stored on the client. Our heuristic assumes that q incurs bandwidth $\sum_{\mathbf{i} \in \mathbf{B}.\mathsf{Lbls}} \mathbf{B}[\mathbf{i}]$ where $\mathbf{B} = \mathsf{EvalBW}(q)$.

To complete this setup, we need a way for the client to estimate the bandwidth of a query with only partial information about **DB** computed during setup. These precomputed statistics are listed on the right of Fig. 8 and the bandwidth estimation algorithm is EvalBW. Intuitively, EvalBW will populate a dictionary $\mathbf{B}$ with entries $\mathbf{B}[\mathbf{i}]$ representing the bandwidth for the ciphertext set containing rows from all $\mathsf{DB}[id]$ where $id \in \mathbf{i}$. We estimate that a query $q \in \mathsf{HybDT.QS}$ incurs bandwidth $\sum_{\mathbf{i} \in \mathbf{B}.\mathsf{Lbls}} \mathbf{B}[\mathbf{i}]$ where $\mathbf{B} = \mathsf{EvalBW}(q)$. We will next explore the tradeoffs involved in storing these statistics.

Memory tradeoffs. Notice that the client storage required for the precomputed statistics (as given in Fig. 8) increases with number of joins (i.e. $|\alpha|$) and size of histograms (i.e. $|\mathsf{rng}(at, \mathsf{DB})|$ for each at). In practice, data may be too complex or client devices may be too memory strapped (e.g. mobile devices) to store this in full. We describe two tradeoffs application designers can explore to better fit their system requirements.

When it is unfeasible to store full frequency histograms for some at, the client can partition $\mathsf{rng}(at, \mathsf{DB})$ into ranges and store this bucketed frequency histogram. EvalBW will approximate $\mathcal{H}_{at}(x)$ by assuming that values within a bucket are uniformly distributed. This approach is used in SQL server and the literature recommends 200 equiDepth (as opposed to equiWidth) buckets [41,10]. In the extreme case, the client uses a single bucket and needs only store $|\mathsf{rng}(at, \mathsf{DB})|$ and uses $\mathcal{H}_{at}(x) \approx \frac{1}{|\mathsf{rng}(at, \mathsf{DB})|}$. Note that this only works when the elements of $\mathsf{rng}(at, \mathsf{DB})$ can be closely approximated and ordered. For example, this may not work

Query Type	Indexed Data	Chicago data set					Sakila data set				
		JnDT		SjDT		HybDT	JnDT		SjDT		HybDT
		FpJn	PpJn	FpSj	PpSj	HybStl	FpJn	PpJn	FpSj	PpSj	HybStl
Non-recursive join	MM lbls	1249	2498	1249	2498	3747	631	1262	631	1262	1893
	MM vals	1.495e10	2.796e7	1.495e10	2.796e7	1.498e10	5.103e8	2.201e6	5.103e8	2.201e6	5.125e8
Recursive join	MM lbls	–	–	2.796e7	–	2.796e7	–	–	2.202e6	–	2.202e6
	MM vals	–	–	1.496e10	–	1.496e10	–	–	5.107e8	–	5.107e8
	HS vals	–	–	7.477e9	2.796e7	7.505e9	–	–	2.552e8	2.201e6	2.574e8
Relation retrieval	MM lbls	–	–	15	15	15	–	–	15	15	15
	MM vals	–	–	4.010e5	4.010e5	4.010e5	–	–	4.409e4	4.409e4	4.409e4
Select	MM lbls	–	–	1.082e6	1.082e6	1.082e6	–	–	1.190e5	1.190e5	1.190e5
	MM vals	–	–	5.749e6	5.749e6	5.749e6	–	–	2.945e5	2.945e5	2.945e5
	HS vals	–	–	5.749e6	5.749e6	5.749e6	–	–	2.945e5	2.945e5	2.945e5
Total	MM lbls	1249	2498	2.905e7	1.085e6	2.905e7	631	1262	2.321e6	1.203e5	2.322e6
	MM vals	1.495e10	2.796e7	2.991e10	3.412e7	2.994e10	5.103e8	2.201e6	1.021e9	2.540e6	1.023e9
	HS vals	–	–	7.483e9	3.371e7	7.511e9	–	–	2.555e8	2.496e6	2.577e8

Fig. 9. Simulated server storage for each data set using each of our schemes in terms of multimap (MM) labels/ values and hashset (HS) values broken down by the query type being indexed (i.e. relation retrievals, non-recursive/ recursive joins, or selections).

with a "name" column because the names in $\mathsf{rng}(at, \mathsf{DB})$ are not dense in any easily enumerated set. In general, bucketing sacrifices the accuracy of EvalBW to reduce client memory. We study this tradeoff more in Section 6.

Above, we assumed the client would pre-compute and store the join sizes. When this is infeasible due to memory constraints, the client can alternatively compute join sizes using table sizes and the $\mathcal{H}_{at}(x)$ during EvalBW whenever $\mathsf{rng}(at)$ is enumerable. Notice that we can express each co-occurrence frequency as a function of the relevant occurrence frequencies. With a single attribute join, let $X = \mathsf{rng}(at_1, \mathsf{DB}) \cap \mathsf{rng}(at_2, \mathsf{DB})$, $N_i = \mathcal{N}(\mathsf{getID}(at_i, \mathsf{scma}))$ then

$$\mathcal{F}(at_1, at_2) = N_1 \cdot N_2 \cdot \sum_{x \in X} \mathcal{H}_{at_1}(x) \cdot \mathcal{H}_{at_2}(x) \quad \text{and} \quad \mathcal{P}_j(at_1, at_2) = N_j \cdot \sum_{x \in X} \mathcal{H}_{at_j}(x).$$

We can extend this to a cluster join $(\mathbf{t}_1, \mathbf{t}_2)$ where $\mathbf{t}_j = (at_1^j, \ldots, at_n^j)$. We substitute the above histogram values for $\mathcal{H}_{\mathbf{t}_j}(x_1, \ldots, x_n)$ and take the sum over all $(x_1, \ldots, x_n)$ where $x_i \in \mathsf{rng}(at_i^1, \mathsf{DB}) \cap \mathsf{rng}(at_i^2, \mathsf{DB})$. These frequencies are approximated by assuming that columns are independently distributed: $\mathcal{H}_{\mathbf{t}_i}(x_1, \ldots, x_n) \approx \prod_{i \in [n]} \mathcal{H}_{at_i^j}(x_i)$. Note also that accuracy issues are compounded if frequency histograms are themselves estimated using bucketing. In general, approximating join sizes trades efficiency (of EvalBW) and accuracy (for cluster joins) to reduce memory.

6 Simulations on Real-World Datasets

To get some indication of how our schemes would fare in practice we simulate the storage and bandwidth they would incur in a real-world context. We show that in practice, PP indexing is likely to be more storage efficient than FP. We also confirm three claims made in this work: (1) PP indexing has equal or better bandwidth than FP on non-recursive joins (i.e. JnDT queries), (2) On recursive selects and joins (i.e. SjDT queries), the analogous choice is data and query dependent, and (3) our heuristic is accurate in finding optimal hybrid query execution plans.

We note that our goal here is not to make broad statements about all SQL data nor to perform a full system evaluation. We see our simulations more as a sanity check which might motivate large-scale implementations of our schemes. Additionally, we are not aware of any benchmarks with just join and select queries so we generate our own as described below.

Simulation setup. Our simulation dataset uses all relations from the MySQL Sakila benchmark [3] and the following fifteen frequently accessed relations from Chicago's Open Data Portal: Bike_Racks, Census_Data,

[3] We excluded the film_text relation since it is a subset of the film relation

Join category	# joins	Ratio of FpJnto PpJn BW		
		Min	Ave	Max
One-one	237	1.0	1.0	1.0
One-many	711	1.0	1.8	2.0
Many-many	932	1.5	465	8000

Fig. 10. Breakdown of all possible non-recursive join queries which returns at least one row by join types. For each type, we simulated the number of rows that would be sent using FP and PP indexing, and report the minimum, average and maximum overhead incurred.

Crimes_2019, Employee_Debt, Fire_Stations, Grafitti, Housing, IUCR_Codes, Land_Inventory, Libraries, Lobbyists, Police_Stations, Reloc_Vehicles, Street_Names, Towed_Vehicles. In total, our setup involved 30 relations, 175 attributes and 219,992 rows. In Appendix G, we present some additional summary statistics to better understand our source data. We also provide a full, annotated source code for our simulations in [40].

We include in α all single-attribute joins that return at least one row. This helps to filter out meaningless join queries (e.g. joining on "language" and "actor"). We consider joins within the Sakila relations and joins within the Chicago relations, but we do not attempt joins between the two independent sources. We generate recursive queries with J joins and S selections by selecting uniformly at random J distinct joins from α as well as S attributes and elements of their domains, discarding queries that return no rows. When $J \geq 2$ we only use input tables with less than 1000 rows to avoid very large output relations.

Server storage. With the above setup we can get an idea of how much server-side storage would be required by each of our indexing schemes. Recall that our schemes make use of a RR multimap primitive and/or a hashset filtering primitive. Therefore, in Fig. 9 we report the number of multimap [4] labels and values as well as the values in hashset HS for each of our StI schemes. We present our simulation results for the two datasets separately since the Chicago data set contains many more rows and would dominate the Sakila statistics. Additionally, we also show a breakdown of these statistics in terms of the queries they index to better understand the cost of each type of query support.

A number of observations can be made from this data. In our simulation we see that even though there are more selections to index (as evidenced by the number of labels), the multimap size (i.e. number of values) is dominated by join indexes. We expect this cost to be lower in a real system because a judicious database administrator can reduce the set of supported joins (α) to a smaller number than we did. Our simulation also brings forth another advantage of PP join indexing – it is more storage efficient by several orders of magnitude. This is because each row token is stored at most once per join (the same thing which causes PP to have better bandwidth) and, in the case of SjDT, there is no need for the "internal join" indexing which essentially doubles the multimap's labels and values. Finally, for the above reason, the storage overhead of hybrid indexing over FP indexing is very small so systems which currently use indexing schemes like FP (e.g. OPX or SPX) can upgrade its security at low cost.

Join categories. We partition joins into three classes which behave quite differently: *one-one*, *one-many* and *many-many*. We say that a join $\mathsf{R} \leftarrow \mathsf{R}_1 \bowtie_{at_1,at_2} \mathsf{R}_2$ is one-one if each row in $\mathsf{R}_1, \mathsf{R}_2$ occurs at most once in R. It is one-many if the above is true for one relation but not for the other. It is many-many if there exists rows in both $\mathsf{R}_1, \mathsf{R}_2$ which occur more than once in R. We record the breakdown of these classes in our datasets in Fig. 10.

StI for JnDT. In Section 4 we showed that PP indexing has superior bandwidth on non-recursive join queries. We demonstrate that these savings by computing all 1880 possible joins in α and report our findings in Fig. 10. As one would expect, PP indexing always performs equal or better to FP – they perform equally for one-one joins but there are moderate and significant savings for one-many and many-many joins respectively.

StI for SjDT. In Section 4.2 we noted that neither PP nor FP joins are strictly superior when it comes to recursive SjDT queries. We demonstrate this using our datasets. For each combination of 1 to 3 joins and 0 to 2 selects, we randomly sampled 25 queries and report the results in Fig. 11. As can be seen, neither scheme can reliably achieve the optimal bandwidth. While FpSj performed better on average, its maximum overhead exceeds that of PpSj in about half the cases.

[4] Note that in the case of FpSj, HybStI, this includes the multimap for internal joins.

Query type	Ratio of BW to ideal					
	FpSj			PpSj		
	Min	Ave	Max	Min	Ave	Max
1 $\bowtie$, 0 σ	1.0	9.6	37	1.0	1.0	1.0
1 $\bowtie$, 1 σ	1.0	1.6	4.0	1.0	60	302
1 $\bowtie$, 2 σ	1.0	1.3	2.0	1.0	90	500
2 $\bowtie$, 0 σ	1.0	3.3	57	1.3	13	54
2 $\bowtie$, 1 σ	1.0	15	201	1.0	41	201
2 $\bowtie$, 2 σ	1.0	14	121	1.0	93	535
3 $\bowtie$, 0 σ	1.0	7.2	48	2.4	9.1	17
3 $\bowtie$, 1 σ	1.0	6.5	63	2.6	23	60
3 $\bowtie$, 2 σ	1.0	5.0	61	2.3	30	84

Query type	Bucketed $B = 1$				Bucketed $B = 200$				Full histograms			
	Correct	Wrong			Correct	Wrong			Correct	Wrong		
		R1	R2	R3		R1	R2	R3		R1	R2	R3
1 $\bowtie$, 0 σ	14	11	0	0	25	0	0	0	25	0	0	0
1 $\bowtie$, 1 σ	6	17	0	6	12	0	12	1	16	0	8	1
1 $\bowtie$, 2 σ	5	0	0	20	14	0	1	10	15	0	0	10
2 $\bowtie$, 0 σ	5	0	0	20	21	3	0	1	25	0	0	0
2 $\bowtie$, 1 σ	15	0	1	9	18	6	1	0	24	0	1	0
2 $\bowtie$, 2 σ	17	8	0	0	20	0	1	4	23	0	0	2
3 $\bowtie$, 0 σ	5	20	0	0	8	10	7	0	21	2	2	0
3 $\bowtie$, 1 σ	2	23	0	0	19	1	5	0	25	0	0	0
3 $\bowtie$, 2 σ	7	18	0	0	16	4	5	0	24	0	1	0

Fig. 11. Left: On randomly generated queries involving the indicated number of joins ($\bowtie$) and selects (σ), we report the minimum, average and maximum ratios of rows sent using each indexing technique compared to the theoretical minimum possible. Right: Using the same queries, we report the accuracy of our heuristic under three different client storage settings. When a suboptimal query execution plan is returned, we report the point at which our heuristic fails (with R3 being the closest to success).

HYBRID STI. In Section 5 we provided a heuristic for client-side leakage-aware query planning. We demonstrate its efficacy when frequency histograms are estimated via three bucketing options: $B = |\mathsf{rng}(at, \mathsf{DB})|$ (full histograms), $B = 200$ and $B = 1$. We use the same 225 queries as the SjDT simulations and set the bandwidth limit L for each $q \in \mathsf{SjDT}$ to be the mean incurred by all 2^J possible HybDT queries to ensure that the optimization is non-trivial. Additionally, join sizes $\mathcal{F}, \mathcal{P}_1, \mathcal{P}_2$ are estimated using the histogram. Therefore, our simulation is conservative and we expect our heuristic to perform better in applications with a fixed L and precomputed join sizes.

In Fig. 11 we show how our heuristic performed for each query type and histogram estimation technique. When the optimal join annotation is not returned we note which "level" the heuristic failed at, where the levels are defined in relation to our definition of "optimality" given in Section 5. In particular, an R1 failure means the returned q' exceeds bandwidth limit L when StE.Eval is run, an R2 failure means q' used more FP joins than was necessary to reduce bandwidth below L and an R3 failure means q' was not the smallest bandwidth option which uses the minimal number of FP joins while meeting L.

Unsurprisingly, there is a direct tradeoff between client memory and the heuristic's accuracy: across all 225 queries, the heuristic returned the optimal q' on 198 with full histograms but only 143 and 76 when $B = 200$ and $B = 1$ respectively. More interestingly, our heuristic seems to improve when the search space increases: when there is one join the heuristic performed slightly better averaged across all three B values than guessing (58.7% vs 50%) but when there are three it performs significantly better (56.4% vs 12.5%). This demonstrates that our heuristic works when it is most needful since we expect the bandwidth overhead from an incorrect choice to increase with query complexity.

7 Conclusion

Our work introduces partially precomputed join indexing and incorporates it into a hybrid StE scheme. While we did not explore it in this work, we believe that our schemes can be extended to support dynamic queries and adaptive security via multimap primitives of the same kind. We believe the former can be achieved in a similar way to KM's extension of SPX to SPX$^+$. To achieve the latter, our schemes can be reframed in JN's model for adaptive compromise [30]. Future work can also extend our query support, possibly by incorporating cryptographic techniques for range queries or aggregations [22,29]. Higher query support would also enable more rigorous testing using real-world applications and query benchmarks. Stronger security can be achieved using lower-leakage indexing primitives [31,38,33].

We also introduce leakage-aware query planning which we believe to be of independent interest as it incorporates structured indexing into DBMS architecture, which may help StE become a part of commercial DBMSes. Future work could improve our heuristic's efficiency and accuracy, or develop analogous hybrid schemes for other query classes.

8 Acknowledgements

We would like to thank the anonymous reviewers for their comments on our work. We are also grateful to Mihir Bellare and Francesca Falzon for discussion and insights. Cash was supported in part by NSF CNS 1703953. Ng was supported by DSO National Laboratories. Rivkin was supported by the Liew Family College Research Fellows Fund.

References

1. encrypted-bigquery-client. `https://github.com/google/encrypted-bigquery-client`, 2015.
2. City of chicago data portal. `https://data.cityofchicago.org/`, 2021.
3. Sakila sample database. `https://dev.mysql.com/doc/sakila/en/`, 2021.
4. P. Antonopoulos, A. Arasu, K. D. Singh, K. Eguro, N. Gupta, R. Jain, R. Kaushik, H. Kodavalla, D. Kossmann, N. Ogg, et al. Azure sql database always encrypted. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1511–1525, 2020.
5. S. Bajaj and R. Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering*, 26(3):752–765, 2013.
6. 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.
7. M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 409–426. Springer, 2006.
8. V. Bindschaedler, P. Grubbs, D. Cash, T. Ristenpart, and V. Shmatikov. The tao of inference in privacy-protected databases. *Proceedings of the VLDB Endowment*, 11(11):1715–1728, 2018.
9. L. Blackstone, S. Kamara, and T. Moataz. Revisiting leakage abuse attacks. Cryptology ePrint Archive, Report 2019/1175, 2019. `https://eprint.iacr.org/2019/1175`.
10. N. Bruno and L. Gravano. *Statistics on query expressions in relational database management systems*. PhD thesis, Columbia University, 2003.
11. Y. Cao, W. Fan, Y. Wang, and K. Yi. Querying shared data with security heterogeneity. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 575–585, 2020.
12. D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: data structures and implementation. In *NDSS*, volume 14, pages 23–26. Citeseer, 2014.
13. D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Annual cryptology conference*, pages 353–373. Springer, 2013.
14. A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90, 1977.
15. M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 577–594. Springer, 2010.
16. S. S. Chow, J.-H. Lee, and L. Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *NDSS*. Citeseer, 2009.
17. V. Ciriani, S. D. C. Di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Keep a few: Outsourcing data while maintaining confidentiality. In *European Symposium on Research in Computer Security*, pages 440–455. Springer, 2009.
18. R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. Cryptology ePrint Archive, Report 2006/210, 2006. `https://eprint.iacr.org/2006/210`.
19. E. Damiani, S. D. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational dbmss. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 93–102, 2003.
20. I. Demertzis, D. Papadopoulos, C. Papamanthou, and S. Shintre. Seal: Attack mitigation for encrypted databases via adjustable leakage. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.
21. S. Evdokimov and O. Günther. Encryption techniques for secure database outsourcing. In *European Symposium on Research in Computer Security*, pages 327–342. Springer, 2007.
22. S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich queries on encrypted data: Beyond exact matches. Cryptology ePrint Archive, Report 2015/927, 2015. `https://eprint.iacr.org/2015/927`.
23. S. Garg, P. Mohassel, and C. Papamanthou. Tworam: efficient oblivious ram in two rounds with applications to searchable encryption. In *Annual International Cryptology Conference*, pages 563–592. Springer, 2016.

24. P. Grofig, I. Hang, M. Härterich, F. Kerschbaum, M. Kohler, A. Schaad, A. Schröpfer, and W. Tighzert. Privacy by encrypted databases. In *Privacy Technologies and Policy - Second Annual Privacy Forum, APF 2014, Athens, Greece, May 20-21, 2014. Proceedings*, pages 56–69, 2014.
25. P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 315–331, 2018.
26. P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 655–672. IEEE, 2017.
27. Z. Gui, O. Johnson, and B. Warinschi. Encrypted databases: New volume attacks against range queries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 361–378, 2019.
28. H. Hacigümüş, 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.
29. T. Hackenjos, F. Hahn, and F. Kerschbaum. Sagma: Secure aggregation grouped by multiple attributes. *ACM SIGMOD Record*, 2020.
30. J. Jaeger and N. Tyagi. Handling adaptive compromise for practical encryption schemes. In *Annual International Cryptology Conference*, pages 3–32. Springer, 2020.
31. S. Kamara and T. Moataz. Encrypted multi-maps with computationally-secure leakage. *IACR Cryptol. ePrint Arch.*, 2018:978, 2018.
32. S. Kamara and T. Moataz. Sql on structurally-encrypted databases. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 149–180. Springer, 2018.
33. S. Kamara, T. Moataz, and O. Ohrimenko. Structured encryption and leakage suppression. In *Annual International Cryptology Conference*, pages 339–370. Springer, 2018.
34. S. Kamara, T. Moataz, S. Zdonik, and Z. Zhao. An optimal relational database encryption scheme. Cryptology ePrint Archive, Report 2020/274, 2020. `https://eprint.iacr.org/2020/274` Accessed: 2020-02-29.
35. M. Kantarcıoglu and C. Clifton. Security issues in querying encrypted data. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 325–337. Springer, 2005.
36. E. S. Lab. The clusion library. `https://github.com/encryptedsystems/Clusion`, 2020.
37. M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655, 2015.
38. S. Patel, G. Persiano, K. Yeo, and M. Yung. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 79–93, 2019.
39. R. A. Popa, C. M. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.
40. A. Rivkin. Hybrid indexing simulations. `https://github.com/AdamRivkin/Hybrid-Indexing-Simulations`, 2021.
41. J. Sack. Optimizing your query plans with the sql server 2014 cardinality estimator, 2014.
42. D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 44–55. IEEE, 2000.
43. S. L. Tu, M. F. Kaashoek, S. R. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. 2013.
44. Z. Yang, S. Zhong, and R. N. Wright. Privacy-preserving queries on encrypted data. In *European Symposium on Research in Computer Security*, pages 479–495. Springer, 2006.

A CJJ+'s Multimap/Dictionary Encryption Schemes

CJJ+'s RH dictionary encryption scheme. In our StE scheme constructed using **SqlStE** (in Section 3) we use a specific RH dictionary encryption scheme to store the rows in DB. We formalize this as Dye_π whose algorithms are in Fig. 12. The primitives (given as input to **SqlStE**) used in Dye_π are symmetric encryption scheme SE and function family F. Note that $\mathsf{Dye}_\pi.\mathsf{KS} = \mathsf{F.KS} \times \mathsf{SE.KS}$.

Example RR multimap encryption scheme. In our StI schemes such as $\mathsf{PpJn}, \mathsf{PpSj}, \mathsf{HybStI}$ we use a RR multimap Mme as a primitive. We give an example of such a scheme $\mathsf{Mme}^{\mathrm{rr}}_\pi$ which is also based on Π_{bas}. Its algorithms are in algorithms are in Fig. 12. The primitives are as in Dye_π but we require that $\mathsf{SE.KS} = \{0,1\}^{\mathsf{F.ol}}$. Note that $\mathsf{Mme}^{\mathrm{rr}}_\pi.\mathsf{KS} = \mathsf{F.KS}$.

$\underline{\textbf{Alg } \mathsf{Dye}_\pi.\mathsf{Enc}\big((K_\mathrm{f}, K_\mathrm{e}), \mathbf{D}\big)}$
Pad all values in $\mathbf{D}$ to the same length
For $\ell \in \mathbf{D}.\mathsf{Lbls}$ do $\mathbf{D}'[\mathsf{F.Ev}(K_\mathrm{f}, \ell)] \leftarrow\!\!\$\ \mathsf{SE.Enc}(K_\mathrm{e}, \mathbf{D}[\ell])$
Return $\big((K_\mathrm{f}, K_\mathrm{e}), \mathbf{D}'\big)$

$\underline{\textbf{Alg } \mathsf{Dye}_\pi.\mathsf{Tok}\big((K_\mathrm{f}, K_\mathrm{e}), \ell\big)}$
$\mathsf{tk} \leftarrow \mathsf{F.Ev}(K_\mathrm{f}, \ell)$; Return tk

$\underline{\textbf{Alg } \mathsf{Dye}_\pi.\mathsf{Eval}(\mathsf{tk}, \mathbf{D}')}$
Return $\mathbf{D}'[\mathsf{tk}]$

$\underline{\textbf{Alg } \mathsf{Dye}_\pi.\mathsf{Dec}\big((K_\mathrm{f}, K_\mathrm{e}), C\big)}$
Unpad and return $\mathsf{SE.Dec}(K_\mathrm{e}, C)$

$\underline{\textbf{Alg } \mathsf{Mme}^{\mathrm{rr}}_\pi.\mathsf{Enc}(K_\mathrm{f}, \mathbf{M})}$
Pad all values in $\mathbf{M}$ to the same length
For $\ell \in \mathbf{M}.\mathsf{Lbls}$ do
  $K_\mathrm{e} \leftarrow \mathsf{F.Ev}(K_\mathrm{f}, \ell\|0)$; $K \leftarrow \mathsf{F.Ev}(K_\mathrm{f}, \ell\|1)$
  For $v \in \mathbf{M}[\ell]$ do
    $\mathbf{D}[\mathsf{F.Ev}(K, \mathrm{ctr})] \leftarrow\!\!\$\ \mathsf{SE.Enc}(K_\mathrm{e}, v)$; $\mathrm{ctr} \leftarrow \mathrm{ctr} + 1$
Return $(K_\mathrm{f}, \mathbf{D})$

$\underline{\textbf{Alg } \mathsf{Mme}^{\mathrm{rr}}_\pi.\mathsf{Tok}(K_\mathrm{f}, \ell)}$
Return $\big(\mathsf{F.Ev}(K_\mathrm{f}, \ell\|0), \mathsf{F.Ev}(K_\mathrm{f}, \ell\|1)\big)$

$\underline{\textbf{Alg } \mathsf{Mme}^{\mathrm{rr}}_\pi.\mathsf{Eval}\big((K_\mathrm{e}, K), \mathbf{D}\big)}$
While $\mathbf{D}[\mathsf{F.Ev}(K, \mathrm{ctr})] \neq \bot$ do
  $x \leftarrow \mathsf{SE.Dec}\big(K_\mathrm{e}, \mathbf{D}[\mathsf{F.Ev}(K, \mathrm{ctr})]\big)$
  $\mathrm{ctr} \leftarrow \mathrm{ctr} + 1$; Unpad x then $S \xleftarrow{\cup} x$
Return S

Fig. 12. Algorithms for RH dictionary encryption scheme Dye_π and RR multimap encryption scheme $\mathsf{Mme}^{\mathrm{rr}}_\pi$.

B Proof of Theorem 1

Theorem 1. *Let* $\mathsf{StE} = \mathbf{SqlStE}[\mathsf{Stl}, \mathsf{SE}, \mathsf{F}]$ *be a correct StE scheme for* SqlDT*. Then given algorithms* $\mathcal{L}^\mathrm{i}, \mathcal{S}^\mathrm{i}$ *and adversary* A *we can define* $\mathcal{L}$ *as in Section 3.2 and construct* $\mathcal{S}, A_\mathrm{s}, A_\mathrm{f}, A_\mathrm{i}$ *such that:*

$$\mathbf{Adv}^{\mathrm{ss}}_{\mathsf{StE},\mathcal{L},\mathcal{S}}(A) \leq \mathbf{Adv}^{\mathrm{ind\$}}_{\mathsf{SE}}(A_\mathrm{s}) + \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(A_\mathrm{f}) + \mathbf{Adv}^{\mathrm{ss}}_{\mathsf{Stl},\mathcal{L}^\mathrm{i},\mathcal{S}^\mathrm{i}}(A_\mathrm{i}).$$

Proof. The adversaries, simulator and games $\mathrm{G}_0, \mathrm{G}_1, \mathrm{G}_2, \mathrm{G}_3$ are given in Fig. 13. Notice that the $\mathsf{EncRows}$ algorithm used in the adversaries and games is given at the top, and uses two oracles $\textsc{Enc}, \textsc{Fn}$ which the algorithms define. Let b be the challenge bit selected in $\mathrm{G}^{\mathrm{ss}}_{\mathsf{StE},\mathcal{L},\mathcal{S}}(A)$.

Notice that we can express $\mathbf{Adv}^{\mathrm{ss}}_{\mathsf{StE},\mathcal{L},\mathcal{S}}(A) = \Pr[\mathrm{G}^{\mathrm{ss}}_{\mathsf{StE},\mathcal{L},\mathcal{S}}(A)|b=1] - \Pr[\mathrm{G}^{\mathrm{ss}}_{\mathsf{StE},\mathcal{L},\mathcal{S}}(A)|b=0] = \Pr[\mathrm{G}_3] - \Pr[\mathrm{G}_0]$. In $b=1$ case, this follows directly from the definition of A_i. In the $b=0$ case, this follows from the definition of $\mathcal{L}^\mathrm{i}, \mathcal{S}^\mathrm{i}$.

The only difference between G_0 and G_1 is whether $\mathsf{IX}, \mathsf{tk}_1, \ldots, \mathsf{tk}_n$ are generated using Stl's algorithms or $\mathcal{S}$. In both cases, $\mathbf{D}'$'s values are encrypted using $\mathsf{SE.Enc}$. This is the same differentiation going on in the semantic security game so $\mathrm{G}^{\mathrm{ss}}_{\mathsf{Stl},\mathcal{L}^\mathrm{i},\mathcal{S}^\mathrm{i}}(A_\mathrm{i}) = \Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_0]$. Similarly the difference between G_1 and G_2 is whether the values in $\mathbf{D}'$ are the output of $\mathsf{SE.Enc}$ or random strings which is what is going on in the IND\$-security game $\mathrm{G}^{\mathrm{ind\$}}_{\mathsf{SE}}(A_\mathrm{s})$, so $\mathbf{Adv}^{\mathrm{ind\$}}_{\mathsf{SE}}(A_\mathrm{s}) = \Pr[\mathrm{G}_2] - \Pr[\mathrm{G}_1]$. Once again, the difference between G_2 and G_3 is whether the labels in $\mathbf{D}'$ (i.e. the tokens in $\mathsf{Dye}_\pi.\mathsf{Enc}$) are generated using $\mathsf{F.Ev}$ or a random function which is what is going on in the PRF-security game $\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A_\mathrm{f})$, so $\mathbf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(A_\mathrm{f}) = \Pr[\mathrm{G}_3] - \Pr[\mathrm{G}_2]$.

Combining all the above equations gives the desired bound on $\mathbf{Adv}^{\mathrm{ss}}_{\mathsf{StE},\mathcal{L},\mathcal{S}}(A)$.

C Proof of Theorem 2

Theorem 2. *Let* $\mathcal{L}, \mathcal{S}$ *be the leakage algorithm and simulator for* Mme*. Let* $\mathcal{L}^\mathrm{f}, \mathcal{L}^\mathrm{p}$ *be the leakage algorithms given in Fig. 5. Then, for all adversaries* A*:*

$$\mathbf{Adv}^{\mathrm{ss}}_{\mathsf{FpJn},\mathcal{L}^\mathrm{f},\mathcal{S}}(A) \leq \mathbf{Adv}^{\mathrm{ss}}_{\mathsf{Mme},\mathcal{L},\mathcal{S}}(A).$$

Additionally, there exists adversary A_m *and simulator* $\mathcal{S}^\mathrm{p}$ *such that:*

$$\mathbf{Adv}^{\mathrm{ss}}_{\mathsf{PpJn},\mathcal{L}^\mathrm{p},\mathcal{S}^\mathrm{p}}(A) \leq \mathbf{Adv}^{\mathrm{ss}}_{\mathsf{Mme},\mathcal{L},\mathcal{S}}(A_\mathrm{m}).$$

Proof. The first result follows directly from the definition of $\mathsf{FpJn}, \mathcal{L}^\mathrm{f}$. The second result requires us to define $A_\mathrm{m}, \mathcal{S}^\mathrm{p}$, which we do in Fig. 14. In both of these, tokens are just concatenated and deconcatenated as needed by the definition of PpJn. The result follows immediately.

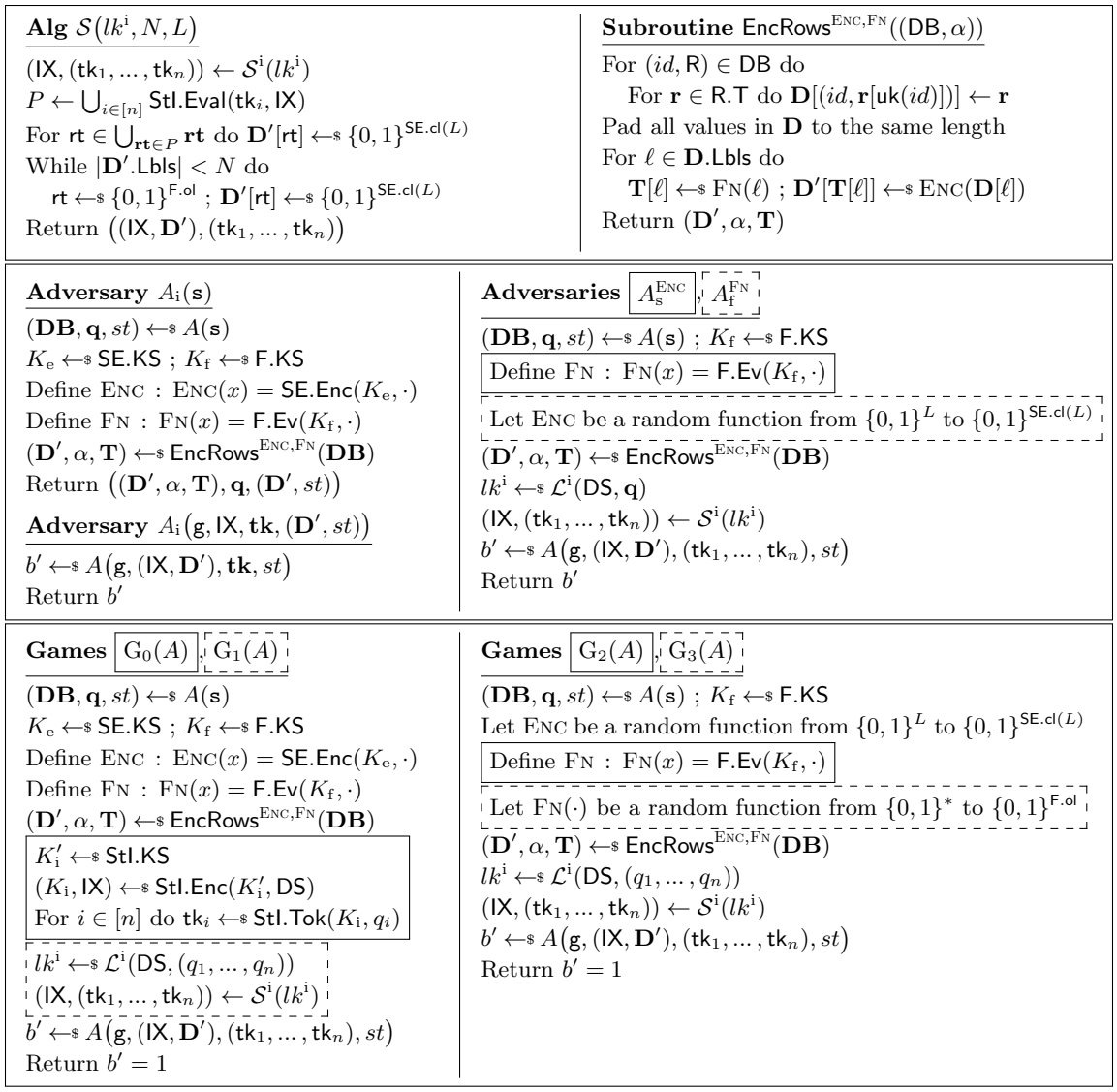

Fig. 13. Simulator, adversaries and games used in the proof of Theorem 1.

D Leakage Profile and Security Proof for PpSj

LEAKAGE. In Section 4.2 we overviewed the different forms of leakage that make up PpSj's leakage profile. Here, we provide a full pseudocode of $\mathcal{L}^{\mathrm{p}}$ for completeness and give some intuition to aid in reading it.

$\mathcal{L}^{\mathrm{p}}$'s pseudocode is given in the top left of Fig. 15. It calls the three subroutines which compute the query-dependent leakage. RS is first called on each of the $q_1, \ldots, q_n$. Through this, counters $c_{\mathrm{q}}, c_{\mathrm{p}}$ are maintained which count the number of accesses to $\mathbf{M}$ (to retrieve a value) and HS (to filter based on a predicate). The labels or predicates associated to each of these subqueries are logged in the vectors $\mathbf{q}, \mathbf{p}$. The $r_1, \ldots, r_n$ returned during these calls are part of the leakage. It reveals the "structure" of each query that was made.

Vector $\mathbf{q}$ are the queries made to the multimap primitive and is therefore an input to $\mathcal{L}$. The output of this makes up the multimap leakage (e.g. multimap query equality pattern) and will be returned by $\mathcal{L}^{\mathrm{p}}$. Vector $\mathbf{p}$ is given as input to QP and HF which compute the equality pattern and filtering results on the hashset predicates respectively.

Theorem 3. *Let $\mathcal{L}, \mathcal{S}$ be the leakage algorithm and simulator for* Mme *respectively. Let $\mathcal{L}^{\mathrm{p}}, \mathcal{S}^{\mathrm{p}}$ be as defined in Fig. 15 and let* F *be the function family used. Then for all adversaries A there exists adversaries $A_{\mathrm{m}}, A_{\mathrm{f}}$ such that:*

$$\mathbf{Adv}^{\mathrm{ss}}_{\mathsf{PpSj},\mathcal{L}^{\mathrm{p}},\mathcal{S}^{\mathrm{p}}}(A) \leq \mathbf{Adv}^{\mathrm{ss}}_{\mathsf{Mme},\mathcal{L},\mathcal{S}}(A_{\mathrm{m}}) + (p+1) \cdot \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(A_{\mathrm{f}}).$$

Adversary $A_{\mathrm{m}}(\mathbf{s})$	**Alg** $\mathcal{S}^{\mathrm{p}}(lk^{\mathrm{m}})$
Return $A(\mathbf{s})$	$(\mathsf{EM}, (\mathsf{tk}_1, \ldots, \mathsf{tk}_{2n})) \leftarrow \mathcal{S}^{\mathrm{m}}(lk^{\mathrm{m}})$
Adversary $A_{\mathrm{m}}(\mathsf{g}, \mathsf{IX}, (\mathsf{tk}_1, \ldots, \mathsf{tk}_{2n}), st)$	$\mathbf{tk} \leftarrow \big((\mathsf{tk}_1, \mathsf{tk}_2), \ldots, (\mathsf{tk}_{2n-1}, \mathsf{tk}_{2n})\big)$
$\mathbf{tk} \leftarrow \big((\mathsf{tk}_1, \mathsf{tk}_2), \ldots, (\mathsf{tk}_{2n-1}, \mathsf{tk}_{2n})\big)$	Return $(\mathsf{EM}, \mathbf{tk})$
$b' \leftarrow\!\!\$\ A(\mathsf{g}, \mathsf{IX}, \mathbf{tk}, st)$; Return b'	

Fig. 14. Simulators (right) and adversaries (left) used in the proof of Theorem 2.

Here, p is the number of distinct predicates used in constructing HS.

Proof. Adversary A_{m} is given in Fig. 15. In the same diagram, we see A_1, A_2 which are both PRF adversaries playing $\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}$. We define A_{f} to randomly pick one at run time and use it.

Now we can proceed via a standard hybrid argument. Let $b_{\mathrm{p}}, b_{\mathrm{f}}, b_{\mathrm{m}}$ be the challenge bits in $\mathrm{G}^{\mathrm{ss}}_{\mathsf{PpSj},\mathcal{L}^{\mathrm{p}},\mathcal{S}^{\mathrm{p}}}$, $\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}$ and $\mathrm{G}^{\mathrm{ss}}_{\mathsf{Mme},\mathcal{L},\mathcal{S}}$ respectively.

From the various advantage definitions, we have that $\mathbf{Adv}^{\mathrm{ss}}_{\mathsf{PpSj},\mathcal{L}^{\mathrm{p}},\mathcal{S}^{\mathrm{p}}}(A) = \Pr[\mathrm{G}^{\mathrm{ss}}_{\mathsf{PpSj},\mathcal{L}^{\mathrm{p}},\mathcal{S}^{\mathrm{p}}}(A)|b_{\mathrm{p}} = 1] - \Pr[\mathrm{G}^{\mathrm{ss}}_{\mathsf{PpSj},\mathcal{L}^{\mathrm{p}},\mathcal{S}^{\mathrm{p}}}(A)|b_{\mathrm{p}} = 0]$, $\mathbf{Adv}^{\mathrm{ss}}_{\mathsf{Mme},\mathcal{L},\mathcal{S}}(A_{\mathrm{m}}) = \Pr[\mathrm{G}^{\mathrm{ss}}_{\mathsf{Mme},\mathcal{L},\mathcal{S}}(A_{\mathrm{m}})|b_{\mathrm{m}} = 1] - \Pr[\mathrm{G}^{\mathrm{ss}}_{\mathsf{Mme},\mathcal{L},\mathcal{S}}(A_{\mathrm{m}})|b_{\mathrm{m}} = 0]$, and $\mathbf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(A_1) = \Pr[\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A_1)|b_{\mathrm{f}} = 1] - \Pr[\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A_1)|b_{\mathrm{f}} = 0]$. Notice also that $\Pr[\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A_2)|b_{\mathrm{f}} = 0, c = i] = \Pr[\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A_2)|b_{\mathrm{f}} = 1, c = i+1]$ for $i \in [p-1]$ and $\Pr[\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A_2)|b_{\mathrm{f}} = 1, c = j] - \Pr[\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A_2)|b_{\mathrm{f}} = 1, c = j] \leq \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(A_2)$ for $j \in [p]$. This means that

$$p \cdot \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(A_2) \geq \Pr[\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A_2)|b_{\mathrm{f}} = 1, c = p] - \Pr[\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A_1)|b_{\mathrm{f}} = 0, c = 1].$$

Notice that A_{m} in $\mathrm{G}^{\mathrm{ss}}_{\mathsf{Mme},\mathcal{L},\mathcal{S}}$ uses the game to simulate multimap encryption and performs the rest itself as it happens in the "real world" of $\mathrm{G}^{\mathrm{ss}}_{\mathsf{PpSj},\mathcal{L}^{\mathrm{p}},\mathcal{S}^{\mathrm{p}}}(A)$. This gives $\Pr[\mathrm{G}^{\mathrm{ss}}_{\mathsf{PpSj},\mathcal{L}^{\mathrm{p}},\mathcal{S}^{\mathrm{p}}}(A)|b_{\mathrm{p}} = 1] = \Pr[\mathrm{G}^{\mathrm{ss}}_{\mathsf{Mme},\mathcal{L},\mathcal{S}}(A_{\mathrm{m}})|b_{\mathrm{m}} = 1]$. Similarly, A_1 simulates multimap encryption as in the "ideal world" of $\mathrm{G}^{\mathrm{ss}}_{\mathsf{Mme},\mathcal{L},\mathcal{S}}$ and defers the filtering key production to $\textsc{Fn}$ which gives us $\Pr[\mathrm{G}^{\mathrm{ss}}_{\mathsf{Mme},\mathcal{L},\mathcal{S}}(A_{\mathrm{m}})|b_{\mathrm{m}} = 0] = \Pr[\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A_1)|b_{\mathrm{f}} = 1]$. When A_2 plays $\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A_2)$, if $c = p$ then all the K_i will be randomly selected. This means $\Pr[\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A_1)|b_{\mathrm{f}} = 0] = \Pr[\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A_2)|b_{\mathrm{f}} = 1, c = p]$. Over p hybrids, we get to the version where all the $\mathsf{F.Ev}(K_i, \cdot)$ (where K_i is not revealed to the adversary) are simulated with random functions, giving us $\Pr[\mathrm{G}^{\mathrm{prf}}_{\mathsf{F}}(A_1)|b_{\mathrm{f}} = 0, c = 1] = \Pr[\mathrm{G}^{\mathrm{ss}}_{\mathsf{PpSj},\mathcal{L}^{\mathrm{p}},\mathcal{S}^{\mathrm{p}}}(A)|b_{\mathrm{p}} = 0]$ because this selects all of HS elements as $\mathcal{S}^{\mathrm{p}}$ does.

E FpSj Details

PSEUDOCODE. The pseudocode for FpSj is given in Fig. 16.

Notice that $\mathsf{FpSj.KS} = \mathsf{Mme}^{\mathrm{rr}}_{\pi}.\mathsf{KS} = \mathsf{F.KS}$. The flags $\mathtt{iij}, \mathtt{ij}$ used come from the terms used for query classification by KMZZ in [34] where recursive joins are split into "internal joins" (i.e. queries of the form $(\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, (\mathtt{r}, id_1), q_2)$ or $(\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, (\mathtt{r}, id_1), q_2)$ and "intermediate internal joins" (i.e.those of form $(\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, q_1, q_2)$).

As alluded to in Section 4.2, the biggest difference between FP and PP indexing of SjDT queries is the handling of internal joins. As was done in OPX, this necessitates the use of a specific RR multimap encryption primitive (namely $\mathsf{Mme}^{\mathrm{rr}}_{\pi}$ from Appendix A) to minimize leakage. In the FpSj's encryption algorithm, we manually add entries to the server-side data structure of $\mathsf{Mme}^{\mathrm{rr}}_{\pi}$ (i.e. dictionary $\mathbf{D}$) to index these joins.

LEAKAGE. The leakage algorithm $\mathcal{L}^{\mathrm{f}}$ for FpSj is given in Fig. 17.

As mentioned in Section 4.2, the differences between this and $\mathcal{L}^{\mathrm{p}}$ all stem from their different handling of joins. As depicted in Fig. 15, we break down the latter's query-dependent leakage into the recursion structure ($\mathbf{r}$ computed by RS), leakage due to queries to the underlying multimap encryption scheme (lk computed using $\mathcal{L}$), hashset filtering results (SET' computed using HF), hashset query patterns (computed using $\mathsf{QP}(\mathbf{p})$) and the total number of hashset predicates made (c_p). For examples and intuition of each of these forms of leakage, see the examples given in Section 4.2.

With $\mathcal{L}^{\mathrm{f}}$, we must compute leakage due to three different types of join queries (leaf, internal or internal intermediate). For leaf joins, the difference in leakage for the two SjDT StI schemes is exactly that of the two JnDT schemes given in Section 4. For internal intermediate joins, these are handled entirely using hashset filtering, much like the recursive joins in PpSj. As such, the leakage is comparable (in that we reveal the

Alg $\mathcal{L}^{\mathrm{p}}\big(\mathsf{DS}, (q_1, \ldots, q_n)\big)$

Construct $\mathbf{M}, \mathsf{SET}$ as in $\mathsf{PpSj.Enc}(\cdot, \mathsf{DS})$
For $i = 1, \ldots, n$ do
  $(r_i, \mathbf{q}, \mathbf{p}, c_{\mathrm{q}}, c_{\mathrm{p}}) \leftarrow \mathsf{RS}(q_i, \mathbf{q}, \mathbf{p}, c_{\mathrm{q}}, c_{\mathrm{p}})$
$\mathbf{r} \leftarrow (r_1, \ldots, r_n)$; $lk \leftarrow\!\!\$\ \mathcal{L}(\mathbf{M}, \mathbf{q})$
$\mathsf{SET}' \leftarrow \mathsf{HF}(\mathbf{p}, \bigcup_{q \in \mathbf{q}} \mathbf{M}[q], \mathsf{SET})$
Return $(\mathbf{r}, lk, \mathsf{QP}(\mathbf{p}), c_{\mathrm{p}}, \mathsf{SET}', |\mathsf{SET}'|)$

Subroutine $\mathsf{HF}\big((p_1, \ldots, p_n), S, \mathsf{SET}\big)$

For all $i \in [n]$ and $\mathsf{rt} \in S$ do
  If $(p_i, \mathsf{rt}) \in \mathsf{SET}$ then $\mathsf{SET}' \xleftarrow{\cup} (i, \mathsf{rt})$
Return SET'

Subroutine $\mathsf{QP}\big((p_1, \ldots, p_n)\big)$

For all $i, j \in [n]$ if $p_i = p_j$ then
  $\mathbf{P}[i, j] \leftarrow 1$ else $\mathbf{P}[i, j] \leftarrow 0$
Return $\mathbf{P}$

Subroutine $\mathsf{RS}(q, \mathbf{q}, \mathbf{p}, c_{\mathrm{q}}, c_{\mathrm{p}})$

If $q = (\mathbf{r}, id)$ then $\mathbf{q} \xleftarrow{\cup} (\mathbf{r}, id)$; $r \leftarrow (\mathtt{m}, c_{\mathrm{q}})$; $c_{\mathrm{q}} \leftarrow c_{\mathrm{q}} + 1$
Else if $q = (\mathbf{s}, at, x, (\mathbf{r}, id))$ then
  $\mathbf{q} \xleftarrow{\cup} (\mathbf{s}, at, x)$; $r \leftarrow (\mathtt{m}, c_{\mathrm{q}})$; $c_{\mathrm{q}} \leftarrow c_{\mathrm{q}} + 1$
Else if $q = (\mathbf{s}, at, x, q_1)$ then
  $(r_1, \mathbf{q}, \mathbf{p}, c_{\mathrm{q}}, c_{\mathrm{p}}) \leftarrow \mathsf{RS}(q_1, \mathbf{q}, \mathbf{p}, c_{\mathrm{q}}, c_{\mathrm{p}})$
  $\mathbf{p} \xleftarrow{\cup} (\mathbf{s}, at, x)$; $r \leftarrow (\mathbf{p}, c_{\mathrm{p}}, r_1)$; $c_{\mathrm{p}} \leftarrow c_{\mathrm{p}} + 1$
Else if $q = (\mathbf{t}_1, \mathbf{t}_2, q_1, q_2)$ then
  For $i = 1, 2$ do
    If $q_i = (\mathbf{r}, id)$ then
      $\mathbf{q} \xleftarrow{\cup} (\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, i)$; $r_i \leftarrow (\mathtt{m}, c_{\mathrm{q}})$; $c_{\mathrm{q}} \leftarrow c_{\mathrm{q}} + 1$
    Else
      $(r'_i, \mathbf{q}, \mathbf{p}, c_{\mathrm{q}}, c_{\mathrm{p}}) \leftarrow \mathsf{RS}(q_i, \mathbf{q}, \mathbf{p}, c_{\mathrm{q}}, c_{\mathrm{p}})$
      $\mathbf{p} \xleftarrow{\cup} (\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, i)$; $r_i \leftarrow (\mathbf{p}, c_{\mathrm{p}}, r'_i)$; $c_{\mathrm{p}} \leftarrow c_{\mathrm{p}} + 1$
  $r \leftarrow (\mathtt{j}, r_1, r_2)$
Return $(r, \mathbf{q}, \mathbf{p}, c_{\mathrm{q}}, c_{\mathrm{p}})$

Alg $\mathcal{S}^{\mathrm{p}}((r_1, \ldots, r_n), lk, \mathbf{P}, c_{\mathrm{p}}, \mathsf{SET}', N)$

$(EM, \mathbf{mt}) \leftarrow\!\!\$\ \mathcal{S}(lk)$
For $i = 1, \ldots, c_{\mathrm{p}}$ do
  If $\exists c \in [i]$ where $\mathbf{P}[c, i] = 1$ then $K_i \leftarrow K_c$
    else $K_i \leftarrow\!\!\$\ \mathsf{F.KS}$
For $(i, \mathsf{rt}) \in \mathsf{SET}'$ do $\mathsf{HS} \xleftarrow{\cup} \mathsf{F.Ev}(K_i, \mathsf{rt})$
While $|\mathsf{HS}| < N$ do $x \leftarrow\!\!\$\ \{0,1\}^{\mathsf{F.ol}}$; $\mathsf{HS} \xleftarrow{\cup} x$
For $i = 1, \ldots, n$ do
  $\mathsf{tk}_i \leftarrow \mathsf{QuerySim}\big(r_i, \mathbf{mt}, (K_1, \ldots, K_{c_{\mathrm{p}}})\big)$
Return $\big((EM, \mathsf{HS}), (\mathsf{tk}_1, \ldots, \mathsf{tk}_n)\big)$

Subroutine $\mathsf{QuerySim}(r, (\mathsf{mt}_1, \ldots, \mathsf{mt}_{c_{\mathrm{q}}}), (K_1, \ldots, K_{c_{\mathrm{p}}}))$

If $r = (\mathtt{m}, i)$ then return $(\mathbf{r}, \mathsf{mt}_i)$
Else if $r = (\mathbf{p}, i, r_1)$ then
  Return $\big(\mathbf{s}, K_i, \mathsf{QuerySim}(r_1, \mathbf{mt}, \mathbf{k})\big)$
Else if $r = (\mathtt{j}, r_1, r_2)$
  For $i = 1, 2$ do
    If $r_i = (\mathtt{m}, i)$ then $\mathsf{tk}_i \leftarrow (\mathbf{r}, \mathsf{mt}_i)$
    Else if $r_i = (\mathbf{p}, j, r')$ then
      $\mathsf{tk}' \leftarrow \mathsf{QuerySim}(r', \mathbf{mt}, \mathbf{k})$; $\mathsf{tk}_i \leftarrow \big(\mathbf{s}, \mathsf{tk}', K_j\big)$
  Return $(\mathtt{j}, \mathsf{tk}_1, \mathsf{tk}_2)$

Alg $A_{\mathrm{m}}(\mathbf{s})$

$(\mathsf{DS}, (q_1, \ldots, q_n), st) \leftarrow\!\!\$\ A(\mathbf{s})$
Construct $\mathbf{M}, \mathsf{SET}$ as in $\mathsf{PpSj.Enc}(\cdot, \mathsf{DS})$
For $i = 1, \ldots, n$ do
  $(r_i, \mathbf{q}, \mathbf{p}, c_{\mathrm{q}}, c_{\mathrm{p}}) \leftarrow \mathsf{RS}(q_i, \mathbf{q}, \mathbf{p}, c_{\mathrm{q}}, c_{\mathrm{p}})$
Return $(\mathbf{M}, \mathbf{q}, (\mathsf{SET}, \mathbf{p}, (r_1, \ldots, r_n), st))$

Alg $A_{\mathrm{m}}(\mathsf{g}, EM, \mathbf{mt}, (\mathsf{SET}, \mathbf{p}, \mathbf{r}, st))$

$(p_1, \ldots, p_{c_{\mathrm{p}}}) \leftarrow \mathbf{p}$
$(r_1, \ldots, r_n) \leftarrow \mathbf{r}$
$K_{\mathrm{f}} \leftarrow\!\!\$\ \mathsf{F.KS}$
For $i = 1, \ldots, c_{\mathrm{p}}$ do
  $K_i \leftarrow \mathsf{F.Ev}(K_{\mathrm{f}}, p_i)$
$\mathbf{k} \leftarrow (K_1, \ldots, K_{c_{\mathrm{p}}})$
For $i \in [n]$ do
  $\mathbf{tk} \xleftarrow{\cup} \mathsf{QuerySim}(r_i, \mathbf{mt}, \mathbf{k})$
Return $A(\mathsf{g}, (EM, \mathsf{HS}), \mathbf{tk}, st)$

Alg $\boxed{A_1^{\mathrm{FN}}}, \boxed{A_2^{\mathrm{FN}}}$

$(\mathsf{DS}, (q_1, \ldots, q_n), st) \leftarrow\!\!\$\ A(\mathbf{s})$
Construct $\mathbf{M}, \mathsf{SET}$ as in $\mathsf{PpSj.Enc}(\cdot, \mathsf{DS})$
For $i = 1, \ldots, n$ do $(r_i, \mathbf{q}, \mathbf{p}, c_{\mathrm{q}}, c_{\mathrm{p}}) \leftarrow \mathsf{RS}(q_i, \mathbf{q}, \mathbf{p}, c_{\mathrm{q}}, c_{\mathrm{p}})$
$(EM, \mathbf{mt}) \leftarrow\!\!\$\ \mathcal{S}(\mathcal{L}(\mathbf{M}, \mathbf{q}))$
$(p_1, \ldots, p_{c_{\mathrm{p}}}) \leftarrow \mathbf{p}$; $\boxed{c \leftarrow\!\!\$\ [p]}$; $\boxed{\mathrm{ctr} \leftarrow 1}$
For $(p, \mathsf{rt}) \in \mathsf{SET}$ do
  $\boxed{\text{If } K_p = \epsilon \text{ then } K_p \leftarrow\!\!\$\ \mathrm{FN}(p)}$
  $\boxed{\mathsf{HS} \xleftarrow{\cup} \mathsf{F.Ev}\big(K_p, \mathsf{rt}\big)}$
  If $K_p = \epsilon$ then
    If $\mathrm{ctr} < c$ or $p \in \mathbf{p}$ then $K_p \leftarrow\!\!\$\ \mathsf{F.KS}$; $\mathrm{ctr} \leftarrow \mathrm{ctr} + 1$
    Else if $\mathrm{ctr} = c$ then $K_p \leftarrow \perp$; $\mathrm{ctr} \leftarrow \mathrm{ctr} + 1$
  If $K_p = \epsilon$ then $x \leftarrow\!\!\$\ \{0,1\}^{\mathsf{F.ol}}$; $\mathsf{HS} \xleftarrow{\cup} x$
  Else if $K_p = \perp$ then $\mathsf{HS} \xleftarrow{\cup} \mathrm{FN}(\mathsf{rt})$
  Else $\mathsf{HS} \xleftarrow{\cup} \mathsf{F.Ev}(K_p, \mathsf{rt})$
For $i \in [n]$ do $\mathbf{tk} \xleftarrow{\cup} \mathsf{QuerySim}(r_i, \mathbf{mt}, (K_{p_1}, \ldots, K_{p_{c_{\mathrm{p}}}}))$
Return $A(\mathsf{g}, (EM, \mathsf{HS}), \mathbf{tk}, st)$

Fig. 15. Leakage profile (top), simulator (middle) and adversaries (bottom) used in the proof of Theorem 3. In $\mathcal{L}^{\mathrm{p}}$, $\mathsf{RS}, \mathcal{L}, \mathsf{HF}, \mathsf{QP}$ compute the recursion structure leakage, Mme's leakage profile, hashset filtering results and hashset query pattern respectively, as discussed in Section 4.2. In $\mathcal{S}^{\mathrm{p}}$, $\mathcal{S}$ is a simulator for Mme. Note that when A_{f} (from Theorem 3) is run it randomly selects one of A_1, A_2 and runs it.

Alg $\mathsf{FpSj.Enc}\big(K_{\mathrm{m}}, (\mathsf{DB}, \alpha, \mathbf{T})\big)$

For all $(id, \mathsf{R}) \in \mathsf{DB}$ and $\mathbf{r} \in \mathsf{R.T}$ do
  $\mathsf{rt} \leftarrow \mathbf{T}[(id, \mathbf{r}[\mathsf{uk}(id)])]$; $\mathbf{M}[(\mathtt{r}, id)] \xleftarrow{\cup} (\mathsf{rt})$
  For $at \in \mathsf{R.Ats}$ where $at \neq \mathsf{uk}(id)$ do
    $\mathbf{M}[(\mathtt{s}, at, \mathbf{r}[at])] \xleftarrow{\cup} (\mathsf{rt})$; $\mathsf{SET} \xleftarrow{\cup} \big((\mathtt{s}, at, \mathbf{r}[at]), \mathsf{rt}\big)$
For $(\mathbf{t}_1, \mathbf{t}_2) \in \alpha$ do
  $id_1 \leftarrow \mathsf{getID}(\mathbf{t}_1)$; $id_2 \leftarrow \mathsf{getID}(\mathbf{t}_2)$
  For $\mathbf{r} \in \big(\mathsf{DB}[id_1] \bowtie_{\mathbf{t}_1, \mathbf{t}_2} \mathsf{DB}[id_2]\big).\mathsf{T}$ do
    For $i = 1, 2$ do $\mathsf{rt}_i \leftarrow \mathbf{T}[(id_i, \mathbf{r}[\mathsf{uk}(id_i)])]$
    $\mathbf{M}[(\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2)] \xleftarrow{\cup} (\mathsf{rt}_1, \mathsf{rt}_2)$; $\mathbf{M}_1[(\mathbf{t}_1, \mathbf{t}_2, \mathsf{rt}_1, 1)] \xleftarrow{\cup} \mathsf{rt}_2$
    $\mathbf{M}_1[(\mathbf{t}_1, \mathbf{t}_2, \mathsf{rt}_2, 2)] \xleftarrow{\cup} \mathsf{rt}_1$; $\mathsf{SET} \xleftarrow{\cup} \big((\mathtt{iij}, \mathbf{t}_1, \mathbf{t}_2), (\mathsf{rt}_1, \mathsf{rt}_2)\big)$
$(K_{\mathrm{m}}, \mathbf{D}) \leftarrow\!\!\$\ \mathsf{Mme}^{\mathrm{rr}}_{\pi}.\mathsf{Enc}(K_{\mathrm{m}}, \mathbf{M})$
For $(\mathbf{t}_1, \mathbf{t}_2, \mathsf{rt}, i) \in \mathbf{M}_1.\mathsf{Lbls}$ do
  For $j = 0, 1$ do $K_j \leftarrow \mathsf{F.Ev}(\mathsf{F.Ev}(K_{\mathrm{m}}, (\mathtt{ij}, \mathbf{t}_1, \mathbf{t}_2, i)), \mathsf{rt} \| j)$
  $\{\mathsf{rt}_1, \ldots, \mathsf{rt}_n\} \leftarrow \mathbf{M}_1[(\mathbf{t}_1, \mathbf{t}_2, \mathsf{rt}, i)]$
  For $k \in [n]$ do
    Pad rt_k to $\mathbf{M}$'s max. value length
    $\mathbf{D}[\mathsf{F.Ev}(K_0, k)] \leftarrow\!\!\$\ \mathsf{SE.Enc}(K_1, \mathsf{rt}_k)$
$K_{\mathrm{f}} \leftarrow\!\!\$\ \mathsf{F.KS}$; $\mathsf{HS} \leftarrow \mathsf{HsEnc}(K_{\mathrm{f}}, \mathsf{SET})$
Return $\big((\mathsf{Schema}(\mathsf{DB}), K_{\mathrm{m}}, K_{\mathrm{f}}), (\mathbf{D}, \mathsf{HS})\big)$

Alg $\mathsf{FpSj.Tok}(K_{\mathrm{i}}, q)$

$(\mathsf{scma}, K_{\mathrm{m}}, K_{\mathrm{f}}) \leftarrow K_{\mathrm{i}}$
If $q = (\mathtt{r}, id)$ then return $(\mathtt{r}, \mathsf{Mme}^{\mathrm{rr}}_{\pi}.\mathsf{Tok}(K_{\mathrm{m}}, (\mathtt{r}, id)))$
Else if $q = (\mathtt{s}, at, x, (\mathtt{r}, id))$ then
  Return $\big(\mathtt{r}, \mathsf{Mme}^{\mathrm{rr}}_{\pi}.\mathsf{Tok}(K_{\mathrm{m}}, (\mathtt{s}, at, x))\big)$
Else if $q = (\mathtt{s}, at, x, q_1)$ then
  Return $\big(\mathtt{s}, \mathsf{F.Ev}(K_{\mathrm{f}}, (\mathtt{s}, at, x)), \mathsf{FpSj.Tok}(K_{\mathrm{i}}, q_1)\big)$
Else if $q = (\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, (\mathtt{r}, id_1), (\mathtt{r}, id_2))$ then
  Return $\big(\mathtt{r}, \mathsf{Mme}^{\mathrm{rr}}_{\pi}.\mathsf{Tok}(K_{\mathrm{m}}, (\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2))\big)$
Else if $q = (\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, q_1, (\mathtt{r}, id))$ then
  Return $(\mathtt{ij}, \mathsf{F.Ev}(K_{\mathrm{m}}, (\mathtt{ij}, \mathbf{t}_1, \mathbf{t}_2, 1)), 1, \mathsf{FpSj.Tok}(K_{\mathrm{i}}, q_1))$
Else if $q = (\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, (\mathtt{r}, id), q_1)$ then
  Return $(\mathtt{ij}, \mathsf{F.Ev}(K_{\mathrm{m}}, (\mathtt{ij}, \mathbf{t}_1, \mathbf{t}_2, 2)), 2, \mathsf{FpSj.Tok}(K_{\mathrm{i}}, q_1))$
Else if $q = (\mathtt{j}, \mathbf{t}_1, \mathbf{t}_2, q_1, q_2)$ then
  For $i = 1, 2$ do $\mathsf{tk}_i \leftarrow\!\!\$\ \mathsf{FpSj.Tok}(K_{\mathrm{i}}, q_i)$
  Return $\big(\mathtt{iij}, \mathsf{F.Ev}(K_{\mathrm{f}}, (\mathtt{iij}, \mathbf{t}_1, \mathbf{t}_2)), \mathsf{tk}_1, \mathsf{tk}_2\big)$

Alg $\mathsf{FpSj.Eval}(\mathsf{tk}, (\mathbf{D}, \mathsf{HS}))$

If $\mathsf{tk} = (\mathtt{r}, \mathsf{mt})$ then
  Return $(\mathsf{Mme}^{\mathrm{rr}}_{\pi}.\mathsf{Eval}(\mathsf{mt}, \mathbf{D}))$
Else if $\mathsf{tk} = (\mathtt{s}, K, \mathsf{tk}_1)$ then
  $P \leftarrow \mathsf{FpSj.Eval}(\mathsf{tk}_1, (\mathbf{D}, \mathsf{HS}))$
  Return $\mathsf{HsFilter}(K, P, \mathsf{HS})$
Else if $\mathsf{tk} = (\mathtt{ij}, K, i, \mathsf{tk}_1)$ then
  $(P_1) \leftarrow \mathsf{FpSj.Eval}(\mathsf{tk}_1, (\mathbf{D}, \mathsf{HS}))$
  For $\mathsf{rt} \in \mathbf{rt} \in P_1$ do
    For $j = 0, 1$ do $K_j \leftarrow \mathsf{F.Ev}(K, \mathsf{rt} \| j)$
    $x \leftarrow \mathbf{D}[\mathsf{F.Ev}(K_0, c_{\mathsf{rt}})]$
    While $x \neq \bot$ do
      $S \xleftarrow{\cup} (\mathsf{SE.Dec}(K_1, x), \mathbf{rt})$
      $c_{\mathsf{rt}} \leftarrow c_{\mathsf{rt}} + 1$
      $x \leftarrow \mathbf{D}[\mathsf{F.Ev}(K_0, c_{\mathsf{rt}})]$
  If $i = 1$ then
    $P_1 \leftarrow \{(\mathsf{rt}) \| \mathbf{rt} : (\mathsf{rt}, \mathbf{rt}) \in S\}$
  Else
    $P_1 \leftarrow \{\mathbf{rt} \| (\mathsf{rt}) : (\mathsf{rt}, \mathbf{rt}) \in S\}$
  Return (P_1)
Else if $\mathsf{tk} = (\mathtt{iij}, K, \mathsf{tk}_1, \mathsf{tk}_2)$ then
  For $i = 1, 2$ do
    $(P_i) \leftarrow \mathsf{FpSj.Eval}(\mathsf{tk}_i, (\mathbf{D}, \mathsf{HS}))$
  For $\mathbf{rt}_1 \in P_1$ and $\mathbf{rt}_2 \in P_2$ do
    For $\mathsf{rt}_1 \in \mathbf{rt}_1$ and $\mathsf{rt}_2 \in \mathbf{rt}_2$ do
      If $\mathsf{F.Ev}(K, (\mathsf{rt}_1, \mathsf{rt}_2)) \in \mathsf{HS}$ then
        $P_0 \xleftarrow{\cup} \mathbf{rt}_1 \| \mathbf{rt}_2$
  Return (P_0)

Alg $\mathsf{FpSj.Fin}(K_{\mathrm{i}}, q, (M_1))$

$(\mathsf{scma}, K_{\mathrm{m}}, K_{\mathrm{f}}) \leftarrow K_{\mathrm{i}}$
Using scma and q, compute the
  attributes $\mathbf{at}$ in $\mathsf{SjDT.Spec}(q, \mathsf{DS})$
$\mathsf{R} \leftarrow \mathsf{NewRltn}(\mathbf{at})$
For $(m_1, \ldots, m_n) \in M_1$ do
  $\mathsf{R.T} \xleftarrow{\cup} m_1 \| \ldots \| m_n$
Return R

Fig. 16. Algorithms for FpSj, the StI scheme for SjDT using FP indexing.

equality pattern and filtering results of the predicates) with the only subtlety coming from the fact that FpSj associates the join predicate with a pair of row tokens, thereby leakage the equality pattern of the join (but restricted to the rows that have been retrieved by the subqueries).

This leaves the leakage from internal intermediate joins. Recall that the indexing of such joins involved manual additions to the output of $\mathsf{Mme}^{\mathrm{rr}}_{\pi}.\mathsf{Enc}$ (i.e. $\mathbf{D}$). As such, the leakage algorithm must include information to simulate these entries. This includes the final number of values in $\mathbf{D}$ (i.e. M), the length of these values (i.e. ℓ), the query pattern of such joins (i.e. $\mathsf{QP}(\mathbf{j})$), query responses to these (i.e. $\mathbf{I}$) and the number of such joins (i.e. c_{j}).

SECURITY. The security of FpSj Its security is given in Theorem 4 below, which uses the simulator $\mathcal{S}^{\mathrm{f}}$ given in Fig. 17.

Theorem 4. *Let $\mathcal{L}, \mathcal{S}$ be the leakage algorithm and simulator for $\mathsf{Mme}^{\mathrm{rr}}_{\pi}$ respectively (given in [12]). Let F, SE be the primitives used in $\mathsf{Mme}^{\mathrm{rr}}_{\pi}$ and FpSj's algorithms. Let $\mathcal{L}^{\mathrm{f}}, \mathcal{S}^{\mathrm{f}}$ be as defined in Fig. 17 and Fig. 17*

respectively. Then for all adversaries A there exists adversaries $A_{\mathrm{f}}, A_{\mathrm{s}}$ such that:

$$\mathbf{Adv}^{\mathrm{ss}}_{\mathsf{PpSj},\mathcal{L}^{\mathrm{p}},\mathcal{S}^{\mathrm{p}}}(A) \leq (m+m_1)\mathbf{Adv}^{\mathrm{ind\$}}_{\mathsf{SE}}(A_{\mathrm{s}}) + (m+m_1+p+1).$$

Here, m, m_1 are the number of labels in $\mathbf{M}, \mathbf{M}_1$ respectively and p is the number of distinct predicates used in constructing HS*.*

Proof. This proof is quite similar to Theorem 3, except that we can now reduce security straight to SE, F because we assumed the use of the $\mathsf{Mme}^{\mathrm{rr}}_{\pi}$ multimap encryption scheme. We therefore omit a full description of the adversaries and proof except to say that the multiplicative factors in the bound come from the number of SE and F keys that are used in FpSj.Enc (including those which are an output of F.Ev).

F HybStl Details

PSEUDOCODE. HybStl's algorithms merge the techniques used in $\mathsf{FpSj}, \mathsf{PpSj}$ to support hybrid queries from HybDT. The pseudocode is given in Fig. 18. Note that HybFinalize is a subroutine recursively called by HybStl.Fin to perform client-side (i.e. PP) joins.

As mentioned in Section 5 the only subtlety in this handling is that the ordering of attributes in the tuple sets passed into HybStl.Fin may not be consistent with the desired output relation. This edge case may occur when a recursive join query is made which makes use of both types of indexing. This can be easily remedied because HybStl.Fin has access to schema scma and query q. Therefore, it can compute the desired order of attributes in the output relation.

LEAKAGE. For completeness, the leakage profile of HybStl is described via pseudocode in Fig. 19. This leakage algorithm merges our two previous ones (i.e. $\mathcal{L}^{\mathrm{f}}$ and $\mathcal{L}^{\mathrm{p}}$) in the straightforward way. In particular, the only difference between $\mathcal{L}^{\mathrm{f}}$ (Fig. 17) and $\mathcal{L}^{\mathrm{h}}$ is the recursion structure of partially precomputed joins which are handled in the style of $\mathcal{L}^{\mathrm{p}}$ (Fig. 15).

SECURITY. As alluded to above, the proof of HybStl's security (with respect to $\mathcal{L}^{\mathrm{h}}$) is very similar to the result for Theorem 4. As such, we state the security bound below but omit the proof for brevity.

Theorem 5. *Let $\mathcal{L}$ be the leakage algorithm and simulator for $\mathsf{Mme}^{\mathrm{rr}}_{\pi}$ (given in [12]) and $\mathcal{L}^{\mathrm{h}}$ be as defined in Fig. 19. Let F, SE be the primitives used in $\mathsf{Mme}^{\mathrm{rr}}_{\pi}$ and HybStl's algorithms. Then for all adversaries A there exists $A_{\mathrm{f}}, A_{\mathrm{s}}, \mathcal{S}^{\mathrm{h}}$ such that:*

$$\mathbf{Adv}^{\mathrm{ss}}_{\mathsf{HybStl},\mathcal{L}^{\mathrm{h}},\mathcal{S}^{\mathrm{h}}}(A) \leq (m+m_1)\mathbf{Adv}^{\mathrm{ind\$}}_{\mathsf{SE}}(A_{\mathrm{s}}) + (m+m_1+p+1).$$

Here, m, m_1 are the number of labels in $\mathbf{M}, \mathbf{M}_1$ respectively and p is the number of distinct predicates used in constructing HS*.*

G Supplementary Content for Simulations

The Data Portal stores each Chicago relation separately and intends each relation to be useful on its own – independent from the other relations. The Sakila database also has 15 relations, with a total of 46,238 rows and 88 attributes. Unlike the Chicago database, the Sakila relations have a clear logical structure in the schema such that each relation has a role defined relative to the other relations. Details about the Sakila schema can be found on the MySQL website. By including one example database without a structured schema and one with, we hope to model two different use cases – one where the DBA treats each relation as existing independently and one where the DBA carefully pre-plans the entire organization.

To give an idea of the data distribution in our data sets, we give some summary statistics about each in Table 20. We report each relation's name, number of attributes, number of rows, and minimum, mean, and maximum attribute densities. The density of an attribute is the average occurrence frequency of the values in that column. In other words, for relation $\mathsf{DB}[id]$ at's attribute density is $\frac{|\mathsf{rng}(at)|}{|\mathsf{DB}[id].\mathsf{T}|}$.

Alg $\mathcal{L}^{\mathrm{f}}\big(\mathsf{DS}, (q_1, \ldots, q_n)\big)$

Construct $\mathbf{M}, \mathbf{M}_1, \mathsf{SET}$ as in $\mathsf{FpSj.Enc}(\cdot, \mathsf{DS})$
For $i = 1, \ldots, n$ do
  $(r_i, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}}) \leftarrow \mathsf{RS}(q_i, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}})$
$\mathbf{r} \leftarrow (r_1, \ldots, r_n)$; $lk \leftarrow\!\!\$\ \mathcal{L}(\mathbf{M}, \mathbf{q})$; $S \leftarrow \bigcup_{q \in \mathbf{q}} \mathbf{M}[q]$
While $S \neq S'$ do $S \leftarrow S'$; $(\mathsf{SET}, S', \mathbf{I}) \leftarrow \mathsf{IJ}(\mathbf{j}, S', \mathbf{I})$
$\mathsf{SET}' \leftarrow \mathsf{HF}(\mathbf{p}, S, \mathsf{SET})$
Define M : # of vals in $\mathbf{M}$ and $\mathbf{M}_1$
Define ℓ : max. length val in $\mathbf{M}, \mathbf{M}_1$
Return $(\mathbf{r}, lk, \mathsf{QP}(\mathbf{p}), c_{\mathrm{p}}, \mathsf{SET}', |\mathsf{SET}'|, \mathbf{I}, M, \ell, \mathsf{QP}(\mathbf{j}), c_{\mathrm{j}})$

Subroutine $\mathsf{IJ}\big((j_1, \ldots, j_n), S, \mathbf{I}\big)$

For $i \in [n]$ $\mathsf{rt} \in S$ do
  $(\mathbf{t}_1, \mathbf{t}_2, k) \leftarrow j_i$
  If $\mathbf{M}_1[(\mathbf{t}_1, \mathbf{t}_2, \mathsf{rt}, i)] \neq \perp$ then
    $S' \leftarrow S' \cup \mathbf{M}_1[(\mathbf{t}_1, \mathbf{t}_2, \mathsf{rt}, i)]$; $\mathbf{I}[(\mathsf{rt}, i)] \leftarrow \mathbf{M}_1[(\mathbf{t}_1, \mathbf{t}_2, \mathsf{rt}, i)]$
Return $(S', \mathbf{I})$

Subroutine $\mathsf{HF}(\mathbf{p}, S, \mathsf{SET})$

$(p_1, \ldots, p_n) \leftarrow \mathbf{p}$
For all $i \in [n]$
  If $p_i = (\mathtt{iij}, \mathbf{t}_1, \mathbf{t}_2)$ then
    For $\mathbf{rt} \in S \times S$ do
      If $(p_i, \mathbf{rt}) \in \mathsf{SET}$ then
        $\mathsf{SET}' \xleftarrow{\cup} (i, \mathbf{rt})$
  Else
    For $\mathsf{rt} \in S$ do
      If $(p_i, \mathsf{rt}) \in \mathsf{SET}$ then
        $\mathsf{SET}' \xleftarrow{\cup} (i, \mathsf{rt})$
Return SET'

Subroutine $\mathsf{QP}\big((t_1, \ldots, t_n)\big)$

For all $i, j \in [n]$ if $t_i = t_j$ then
  $\mathbf{T}[i, j] \leftarrow 1$ else $\mathbf{T}[i, j] \leftarrow 0$
Return $\mathbf{T}$

Subroutine $\mathsf{RS}(q, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}})$

If $q = (\mathbf{r}, id)$ then $\mathbf{q} \xleftarrow{\cup} (\mathbf{r}, id)$; $r \leftarrow (\mathtt{m}, c_{\mathrm{q}})$; $c_{\mathrm{q}} \leftarrow c_{\mathrm{q}} + 1$
Else if $q = (\mathbf{s}, at, x, (\mathbf{r}, id))$ then $\mathbf{q} \xleftarrow{\cup} (\mathbf{s}, at, x)$; $r \leftarrow (\mathtt{m}, c_{\mathrm{q}})$; $c_{\mathrm{q}} \leftarrow c_{\mathrm{q}} + 1$
Else if $q = (\mathbf{s}, at, x, q_1)$ then
  $(r_1, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}}) \leftarrow \mathsf{RS}(q_1, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}})$; $\mathbf{p} \xleftarrow{\cup} (\mathbf{s}, at, x)$; $r \leftarrow (\mathbf{s}, c_{\mathrm{p}}, r_1)$; $c_{\mathrm{p}} \leftarrow c_{\mathrm{p}} + 1$
Else if $q = (\mathbf{t}_1, \mathbf{t}_2, (\mathbf{r}, id_1), (\mathbf{r}, id_2))$ then $\mathbf{q} \xleftarrow{\cup} (\mathbf{j}, \mathbf{t}_1, \mathbf{t}_2)$; $r \leftarrow (\mathtt{m}, c_{\mathrm{q}})$; $c_{\mathrm{q}} \leftarrow c_{\mathrm{q}} + 1$
Else if $q = (\mathbf{t}_1, \mathbf{t}_2, q_1, (\mathbf{r}, id))$ or $q = (\mathbf{t}_1, \mathbf{t}_2, (\mathbf{r}, id), q_1)$ then
  $(r_1, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}}) \leftarrow \mathsf{RS}(q_1, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}})$; If $q = (\mathbf{t}_1, \mathbf{t}_2, q_1, (\mathbf{r}, id))$ then $i = 1$ else $i = 2$
  $\mathbf{j} \xleftarrow{\cup} (\mathbf{t}_1, \mathbf{t}_2, i)$; $r \leftarrow (\mathtt{ij}, c_{\mathrm{j}}, i, r_1)$; $c_{\mathrm{j}} \leftarrow c_{\mathrm{j}} + 1$
Else if $q = (\mathbf{t}_1, \mathbf{t}_2, q_1, q_2)$ then
  For $i = 1, 2$ do $(r_i, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}}) \leftarrow \mathsf{RS}(q_i, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}})$
  $\mathbf{p} \xleftarrow{\cup} (\mathtt{iij}, \mathbf{t}_1, \mathbf{t}_2)$; $r \leftarrow (\mathtt{iij}, c_{\mathrm{p}}, r_1, r_2)$; $c_{\mathrm{p}} \leftarrow c_{\mathrm{p}} + 1$
Return $(r, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}})$

Alg $\mathcal{S}^{\mathrm{f}}(\mathbf{r}, lk, \mathbf{P}, c_{\mathrm{p}}, \mathsf{SET}', N, \mathbf{I}, M, \ell, \mathbf{J}, c_{\mathrm{j}})$

$(r_1, \ldots, r_n) \leftarrow \mathbf{r}$; $(\mathbf{D}, \mathbf{mt}) \leftarrow\!\!\$\ \mathcal{S}(lk)$
For $i = 1, \ldots, c_{\mathrm{p}}$ do
  If $\exists c \in [i]$ where $\mathbf{P}[c, i] = 1$ then $K_i \leftarrow K_c$ else $K_i \leftarrow\!\!\$\ \mathsf{F.KS}$
For $(i, x) \in \mathsf{SET}'$ do $\mathsf{HS} \xleftarrow{\cup} \mathsf{F.Ev}(K_i, x)$
While $|\mathsf{HS}| < N$ do $x \leftarrow\!\!\$\ \{0,1\}^{\mathsf{F.ol}}$; $\mathsf{HS} \xleftarrow{\cup} x$
For $(\mathsf{rt}, i) \in \mathbf{I}.\mathsf{Lbls}$ do
  If $\exists c \in [i]$ where $\mathbf{J}[c, i] = 1$ then $K'_i \leftarrow K'_c$ else $K'_i \leftarrow\!\!\$\ \mathsf{F.KS}$
  $\{\mathsf{rt}'_1, \ldots, \mathsf{rt}'_n\} \leftarrow \mathbf{I}[(\mathsf{rt}, i)]$; For $k = 0, 1$ do $K''_k \leftarrow \mathsf{F.Ev}(K'_i, \mathsf{rt} \| k)$
  For $k \in [n]$ do
    Pad rt'_k to length ℓ then $\mathbf{D}[\mathsf{F.Ev}(K''_0, k)] \leftarrow \mathsf{SE.Enc}(K''_1, \mathsf{rt}'_k)$
While $|\mathbf{D}.\mathsf{Lbls}| < M$ do $x \leftarrow\!\!\$\ \{0,1\}^{\mathsf{F.ol}}$; $\mathbf{D}[x] \leftarrow\!\!\$\ \{0,1\}^{\ell}$
For $i = 1, \ldots, n$ do
  $\mathsf{tk}_i \leftarrow \mathsf{QuerySim}(r_i, \mathbf{mt}, (K_1, \ldots, K_{c_{\mathrm{p}}}), (K'_1, \ldots, K'_{c_{\mathrm{j}}}))$
Return $\big((\mathbf{D}, \mathsf{HS}), (\mathsf{tk}_1, \ldots, \mathsf{tk}_n)\big)$

Subroutine $\mathsf{QuerySim}(r, \mathbf{mt}, \mathbf{k}, \mathbf{k}')$

$(\mathsf{mt}_1, \ldots, \mathsf{mt}_{c_{\mathrm{q}}}) \leftarrow \mathbf{mt}$
$(K_1, \ldots, K_{c_{\mathrm{p}}}) \leftarrow \mathbf{k}$; $(K'_1, \ldots, K'_{c_{\mathrm{j}}}) \leftarrow \mathbf{k}'$
If $r = (\mathtt{m}, i)$ then return $(\mathbf{r}, \mathsf{mt}_i)$
Else if $r = (\mathbf{p}, i, r_1)$ then
  $\mathsf{tk} \leftarrow \mathsf{QuerySim}(r_1, \mathbf{mt}, \mathbf{k}, \mathbf{k}')$
  Return $(\mathbf{s}, K_i, \mathsf{tk})$
Else if $r = (\mathtt{ij}, i, j, r_1)$ then
  $\mathsf{tk} \leftarrow \mathsf{QuerySim}(r_1, \mathbf{mt}, \mathbf{k}, \mathbf{k}')$
  Return $(\mathtt{ij}, K'_i, j, \mathsf{tk})$
Else if $r = (\mathtt{iij}, i, r_1, r_2)$ then
  For $j = 1, 2$ do
    $\mathsf{tk}_j \leftarrow \mathsf{QuerySim}(r_j, \mathbf{mt}, \mathbf{k}, \mathbf{k}')$
  Return $(\mathtt{iij}, K_i, \mathsf{tk}_1, \mathsf{tk}_2)$

Fig. 17. Top: Leakage profile for FpSj. Here, $\mathcal{L}$ is the leakage algorithm for $\mathsf{Mme}^{\mathrm{rr}}_{\pi}$ and subroutines $\mathsf{IJ}, \mathsf{HF}, \mathsf{QP}, \mathsf{RS}$ compute the leakage associated to internal joins, hashset filtering, hashset query patterns and query recursion structures, as discussed in Section 4.2 and Appendix E. Bottom: Simulator used in the proof of Theorem 4 where $\mathcal{S}$ is a simulator for $\mathsf{Mme}^{\mathrm{rr}}_{\pi}$.

Alg $\mathsf{HybStI.Enc}\big(K_{\mathrm{m}}, (\mathsf{DB}, \alpha, \mathbf{T})\big)$

For all $(id, \mathsf{R}) \in \mathsf{DB}$ and $\mathbf{r} \in \mathsf{R.T}$ do
  $\mathsf{rt} \leftarrow \mathbf{T}[(id, \mathbf{r}[\mathsf{uk}(id)])]$; $\mathbf{M}[(\mathbf{r}, id)] \xleftarrow{\cup} (\mathsf{rt})$
  For $at \in \mathsf{R.Ats}$ where $at \neq \mathsf{uk}(id)$ do
    $\mathbf{M}[(\mathtt{s}, at, \mathbf{r}[at])] \xleftarrow{\cup} (\mathsf{rt})$; $\mathsf{SET} \xleftarrow{\cup} \big((\mathtt{s}, at, \mathbf{r}[at]), \mathsf{rt}\big)$
For $(\mathbf{t}_1, \mathbf{t}_2) \in \alpha$ do
  $id_1 \leftarrow \mathsf{getID}(\mathbf{t}_1)$; $id_2 \leftarrow \mathsf{getID}(\mathbf{t}_2)$
  For $\mathbf{r} \in \big(\mathsf{DB}[id_1] \bowtie_{\mathbf{t}_1, \mathbf{t}_2} \mathsf{DB}[id_2]\big).\mathsf{T}$ do
    For $i = 1, 2$ do
      $\mathsf{rt}_i \leftarrow \mathbf{T}[(id_i, \mathbf{r}[\mathsf{uk}(id_i)])]$
      $\mathbf{M}[(\mathsf{pp}, \mathbf{t}_1, \mathbf{t}_2, i)] \xleftarrow{\cup} (\mathsf{rt}_i)$
      $\mathsf{SET} \xleftarrow{\cup} \big((\mathsf{pp}, \mathbf{t}_1, \mathbf{t}_2, i), \mathsf{rt}_i\big)$
    $\mathbf{M}[(\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2)] \xleftarrow{\cup} (\mathsf{rt}_1, \mathsf{rt}_2)$
    $\mathbf{M}_1[(\mathbf{t}_1, \mathbf{t}_2, \mathsf{rt}_1, 1)] \xleftarrow{\cup} \mathsf{rt}_2$; $\mathbf{M}_1[(\mathbf{t}_1, \mathbf{t}_2, \mathsf{rt}_2, 2)] \xleftarrow{\cup} \mathsf{rt}_1$
    $\mathsf{SET} \xleftarrow{\cup} \big((\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2), (\mathsf{rt}_1, \mathsf{rt}_2)\big)$
$(K_{\mathrm{m}}, \mathbf{D}) \leftarrow\!\!\$\ \mathsf{Mme}^{\mathrm{rr}}_{\pi}.\mathsf{Enc}(K_{\mathrm{m}}, \mathbf{M})$
For $(\mathbf{t}_1, \mathbf{t}_2, \mathsf{rt}, i) \in \mathbf{M}_1.\mathsf{Lbls}$ do
  For $j = 0, 1$ do
    $K_j \leftarrow \mathsf{F.Ev}(\mathsf{F.Ev}(K_{\mathrm{m}}, (\mathtt{ij}, \mathbf{t}_1, \mathbf{t}_2, i)), \mathsf{rt} \| j)$
  $\{\mathsf{rt}_1, \ldots, \mathsf{rt}_n\} \leftarrow \mathbf{M}_1[(\mathbf{t}_1, \mathbf{t}_2, \mathsf{rt}, i)]$
  For $k \in [n]$ do
    Pad rt_k to $\mathbf{M}$'s max. value length
    $\mathbf{D}[\mathsf{F.Ev}(K_0, k)] \leftarrow\!\!\$\ \mathsf{SE.Enc}(K_1, \mathsf{rt}_k)$
$K_{\mathrm{f}} \leftarrow\!\!\$\ \mathsf{F.KS}$; $\mathsf{HS} \leftarrow \mathsf{HsEnc}(K_{\mathrm{f}}, \mathsf{SET})$
Return $\big((\mathsf{Schema}(\mathsf{DB}), K_{\mathrm{m}}, K_{\mathrm{f}}), (\mathbf{D}, \mathsf{HS})\big)$

Alg $\mathsf{HybStI.Tok}(K_{\mathrm{i}}, q)$

$(\mathsf{scma}, K_{\mathrm{m}}, K_{\mathrm{f}}) \leftarrow K_{\mathrm{i}}$
If $q = (\mathbf{r}, id)$ then return $\big(\mathbf{r}, \mathsf{Mme}^{\mathrm{rr}}_{\pi}.\mathsf{Tok}(K_{\mathrm{m}}, (\mathbf{r}, id))\big)$
Else if $q = (\mathtt{s}, at, x, (\mathbf{r}, id))$ then
  Return $\big(\mathbf{r}, \mathsf{Mme}^{\mathrm{rr}}_{\pi}.\mathsf{Tok}(K_{\mathrm{m}}, (\mathtt{s}, at, x))\big)$
Else if $q = (\mathtt{s}, at, x, q_1)$ then
  Return $\big(\mathtt{s}, \mathsf{F.Ev}(K_{\mathrm{f}}, (\mathtt{s}, at, x)), \mathsf{HybStI.Tok}(K_{\mathrm{i}}, q_1)\big)$
Else if $q = (\mathsf{pp}, \mathbf{t}_1, \mathbf{t}_2, q_1, q_2)$ then
  For $i = 1, 2$ do
    If $q_i = (\mathbf{r}, id_i)$ then
      $\mathsf{tk}_i \leftarrow\!\!\$\ \big(\mathbf{r}, \mathsf{Mme}^{\mathrm{rr}}_{\pi}.\mathsf{Tok}(K_{\mathrm{m}}, (\mathsf{pp}, \mathbf{t}_1, \mathbf{t}_2, i))\big)$
    Else
      $\mathsf{tk}' \leftarrow\!\!\$\ \mathsf{HybStI.Tok}(K_{\mathrm{i}}, q_i)$
      $\mathsf{tk}_i \leftarrow \big(\mathtt{s}, \mathsf{F.Ev}(K_{\mathrm{f}}, (\mathsf{pp}, \mathbf{t}_1, \mathbf{t}_2, i)), \mathsf{tk}'\big)$
  Return $(\mathsf{pp}, \mathsf{tk}_1, \mathsf{tk}_2)$
Else if $q = (\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2, (\mathbf{r}, id_1), (\mathbf{r}, id_2))$ then
  Return $\big(\mathbf{r}, \mathsf{Mme}^{\mathrm{rr}}_{\pi}.\mathsf{Tok}(K_{\mathrm{m}}, (\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2))\big)$
Else if $q = (\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2, q_1, (\mathbf{r}, id))$ then
  Return $(\mathtt{ij}, \mathsf{F.Ev}(K_{\mathrm{m}}, (\mathtt{ij}, \mathbf{t}_1, \mathbf{t}_2, 1)), \mathsf{HybStI.Tok}(K_{\mathrm{i}}, q_1))$
Else if $q = (\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2, (\mathbf{r}, id), q_1)$ then
  Return $(\mathtt{ij}, \mathsf{F.Ev}(K_{\mathrm{m}}, (\mathtt{ij}, \mathbf{t}_1, \mathbf{t}_2, 2)), \mathsf{HybStI.Tok}(K_{\mathrm{i}}, q_1))$
Else if $q = (\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2, q_1, q_2)$ then
  For $i = 1, 2$ do $\mathsf{tk}_i \leftarrow\!\!\$\ \mathsf{HybStI.Tok}(K_{\mathrm{i}}, q_i)$
  Return $\big(\mathtt{iij}, \mathsf{F.Ev}(K_{\mathrm{f}}, (\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2)), \mathsf{tk}_1, \mathsf{tk}_2\big)$

Alg $\mathsf{HybStI.Eval}(\mathsf{tk}, \mathsf{IX})$

$(\mathbf{D}, \mathsf{HS}) \leftarrow \mathsf{IX}$
If $\mathsf{tk} = (\mathbf{r}, \mathsf{tk}_1)$ then return $(\mathsf{Mme}^{\mathrm{rr}}_{\pi}.\mathsf{Eval}(\mathsf{tk}, \mathsf{IX}))$
Else If $\mathsf{tk} = (\mathtt{s}, K, \mathsf{tk}_1)$ then
  Return $\mathsf{HsFilter}\big(K, \mathsf{HybStI.Eval}(\mathsf{tk}_1, \mathsf{IX}), \mathsf{HS}\big)$
Else if $\mathsf{tk} = (\mathsf{pp}, \mathsf{tk}_1, \mathsf{tk}_2)$
  Return $\mathsf{HybStI.Eval}(\mathsf{tk}_1, \mathsf{IX}) \| \mathsf{HybStI.Eval}(\mathsf{tk}_2, \mathsf{IX})$
Else if $\mathsf{tk} = (\mathtt{ij}, K, \mathsf{tk}_1)$ then
  $(P_1, \ldots, P_n) \leftarrow \mathsf{HybStI.Eval}(\mathsf{tk}_1, \mathsf{IX})$
  Define j : $\exists \mathsf{rt} \in \mathbf{rt} \in P_j$
    where $\mathbf{D}[\mathsf{F.Ev}(\mathsf{F.Ev}(K, \mathsf{rt} \| 0), 0)] \neq \bot$
  For $\mathsf{rt} \in \mathbf{rt} \in P_j$ do
    For $i = 1, 2$ do $K_i \leftarrow \mathsf{F.Ev}(K, \mathsf{rt} \| i)$
    While $\mathbf{D}[\mathsf{F.Ev}(K_0, c_{\mathsf{rt}})] \neq \bot$ do
      $\mathbf{rt} \xleftarrow{\cup} \mathsf{SE.Dec}(K_1, \mathbf{D}[\mathsf{F.Ev}(K_0, c_{\mathsf{rt}})])$
      $P' \xleftarrow{\cup} \mathbf{rt}$; $c_{\mathsf{rt}} \leftarrow c_{\mathsf{rt}} + 1$
  Return $P \setminus \{P_j\} \| (P')$
Else if $\mathsf{tk} = (\mathtt{iij}, K, \mathsf{tk}_1, \mathsf{tk}_2)$ then
  For $i = 1, 2$ do
    $(P^i_1, \ldots, P^i_{n_i}) \leftarrow \mathsf{HybStI.Eval}(\mathsf{tk}_i, (\mathbf{D}, \mathsf{HS}))$
  Define j_1, j_2 : $\exists \mathsf{rt}_i \in \mathbf{rt}_i \in P^i_{j_i}$
    where $\mathsf{F.Ev}(K, (\mathsf{rt}_1, \mathsf{rt}_2)) \in \mathsf{HS}$
  For $\mathsf{rt}_1 \in \mathbf{rt}_1 \in P^1_{j_1}$ and $\mathsf{rt}_2 \in \mathbf{rt}_2 \in P^2_{j_2}$ do
    If $\mathsf{F.Ev}(K, (\mathsf{rt}_1, \mathsf{rt}_2)) \in \mathsf{HS}$ then $P' \xleftarrow{\cup} \mathbf{rt}_1 \| \mathbf{rt}_2$
  If $P' \neq \emptyset$ then return $P_1 \setminus \{P^1_{j_1}\} \| P_2 \setminus \{P^2_{j_2}\} \| (P')$

Alg $\mathsf{HybStI.Fin}\big((\mathsf{scma}, K_{\mathrm{m}}, K_{\mathrm{f}}), q, (M_1, \ldots, M_n)\big)$

Using scma and q, compute $\mathbf{at_1}, \ldots, \mathbf{at_n}, \mathbf{at}$, the attributes in $M_1, \ldots, M_n, \mathsf{HybDT.Spec}(q, \mathsf{DS})$ respectively
For $i \in [n]$ do
  $\mathsf{R}_i \leftarrow \mathsf{NewRltn}(\mathbf{at_i})$
  $\mathsf{R}_i.\mathsf{T} \leftarrow \{m_1 \| \ldots \| m_{n'} : (m_1, \ldots, m_{n'}) \in M_i\}$
$\mathsf{R} \leftarrow \mathsf{HybFinalize}\big(q, (\mathsf{R}_1, \ldots, \mathsf{R}_n)\big)$
If $\mathsf{R.Ats} \neq \mathbf{at}$ then reorder attributes in R accordingly
Return R

Subroutine $\mathsf{HybFinalize}(q, \mathbf{R})$

If $q = (\mathbf{r}, id)$ then $(\mathsf{R}) \leftarrow \mathbf{R}$; Return R
Else if $q = (\mathtt{s}, at, x, q_1)$ then return $\mathsf{HybFinalize}(q_1, \mathbf{R})$
Else if $q = (\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2, q_1, q_2)$ then
  $(\mathsf{R}_1, \ldots, \mathsf{R}_n) \leftarrow \mathbf{R}$
  Define j : $\mathbf{t}_1 \cup \mathbf{t}_2 \subseteq \mathsf{R}_j.\mathsf{Ats}$
  Partition $\mathbf{R} \setminus \{\mathsf{R}_j\}$ into $\mathbf{R}_1, \mathbf{R}_2$ where $\mathbf{R}_i$ contains all the attributes in $\mathsf{HybDT}(q_i, \mathsf{DS})$
  $\mathsf{R} \leftarrow \mathsf{HybFinalize}(q_1, \mathbf{R}_1 \| (\mathsf{R}_j))$
  $\mathsf{R} \leftarrow \mathsf{HybFinalize}(q_2, \mathbf{R}_2 \| (\mathsf{R}))$
Else if $q = (\mathsf{pp}, \mathbf{t}_1, \mathbf{t}_2, q_1, q_2)$ then
  Partition $\mathbf{R}$ into $\mathbf{R}_1, \mathbf{R}_2$ where $\mathbf{R}_i$ contains all attributes in $\mathsf{HybDT}(q_i, \mathsf{DS})$
  Return $\mathsf{HybFinalize}(q_1, \mathbf{R}_1) \bowtie_{\mathbf{t}_1, \mathbf{t}_2} \mathsf{HybFinalize}(q_2, \mathbf{R}_2)$

Fig. 18. Algorithms for HybStI, the StI scheme for HybDT using hybrid indexing. HybFinalize is a recursively called subroutine used in HybStI.Fin.

Alg $\mathcal{L}^{\mathrm{h}}\big(\mathsf{DS}, (q_1, \ldots, q_n)\big)$

Construct $\mathbf{M}, \mathbf{M}_1, \mathsf{SET}$ as in $\mathsf{FpSj.Enc}(\cdot, \mathsf{DS})$
For $i = 1, \ldots, n$ do $(r_i, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}}) \leftarrow \mathsf{RS}(q_i, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}})$
$\mathbf{r} \leftarrow (r_1, \ldots, r_n)$; $lk \leftarrow\!\!\$\ \mathcal{L}(\mathbf{M}, \mathbf{q})$; $S \leftarrow \bigcup_{q \in \mathbf{q}} \mathbf{M}[q]$; While $S \neq S'$ do $S \leftarrow S'$; $(\mathsf{SET}, S', \mathbf{I}) \leftarrow \mathsf{IJ}(\mathbf{j}, S', \mathbf{I})$
Define M : # of vals in $\mathbf{M}$ and $\mathbf{M}_1$
Define ℓ : max. length val in $\mathbf{M}, \mathbf{M}_1$
Return $(\mathbf{r}, lk, \mathsf{QP}(\mathbf{p}), c_{\mathrm{p}}, \mathsf{HF}(\mathbf{p}, S, \mathsf{SET}), |\mathsf{SET}'|, \mathbf{I}, M, \ell, \mathsf{QP}(\mathbf{j}), c_{\mathrm{j}})$

Subroutine $\mathsf{RS}(q, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}})$

If $q = (\mathtt{r}, id)$ then $\mathbf{q} \xleftarrow{\cup} (\mathtt{r}, id)$; $r \leftarrow (\mathtt{m}, c_{\mathrm{q}})$; $c_{\mathrm{q}} \leftarrow c_{\mathrm{q}} + 1$
Else if $q = (\mathtt{s}, at, x, (\mathtt{r}, id))$ then $\mathbf{q} \xleftarrow{\cup} (\mathtt{s}, at, x)$; $r \leftarrow (\mathtt{m}, c_{\mathrm{q}})$; $c_{\mathrm{q}} \leftarrow c_{\mathrm{q}} + 1$
Else if $q = (\mathtt{s}, at, x, q_1)$ then
  $(r_1, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}}) \leftarrow \mathsf{RS}(q_1, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}})$; $\mathbf{p} \xleftarrow{\cup} (\mathtt{s}, at, x)$; $r \leftarrow (\mathtt{s}, c_{\mathrm{p}}, r_1)$; $c_{\mathrm{p}} \leftarrow c_{\mathrm{p}} + 1$
Else if $q = (\mathtt{pp}, \mathbf{t}_1, \mathbf{t}_2, q_1, q_2)$ then
  For $i = 1, 2$ do
    If $q_i = (\mathtt{r}, id)$ then
      $\mathbf{q} \xleftarrow{\cup} (\mathtt{pp}, \mathbf{t}_1, \mathbf{t}_2, i)$; $r_i \leftarrow (\mathtt{m}, c_{\mathrm{q}})$; $c_{\mathrm{q}} \leftarrow c_{\mathrm{q}} + 1$
    Else
      $(r_i', \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}}) \leftarrow \mathsf{RS}(q_i, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}})$; $\mathbf{p} \xleftarrow{\cup} (\mathtt{pp}, \mathbf{t}_1, \mathbf{t}_2, i)$; $r_i \leftarrow (\mathtt{p}, c_{\mathrm{p}}, r_i')$; $c_{\mathrm{p}} \leftarrow c_{\mathrm{p}} + 1$
  $r \leftarrow (\mathtt{pp}, r_1, r_2)$
Else if $q = (\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2, (\mathtt{r}, id_1), (\mathtt{r}, id_2))$ then $\mathbf{q} \xleftarrow{\cup} (\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2)$; $r \leftarrow (\mathtt{m}, c_{\mathrm{q}})$; $c_{\mathrm{q}} \leftarrow c_{\mathrm{q}} + 1$
Else if $q = (\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2, q_1, (\mathtt{r}, id))$ or $q = (\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2, (\mathtt{r}, id), q_1)$ then
  $(r_1, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}}) \leftarrow \mathsf{RS}(q_1, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}})$; If $q = (\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2, q_1, (\mathtt{r}, id))$ then $i = 1$ else $i = 2$
  $\mathbf{j} \xleftarrow{\cup} (\mathbf{t}_1, \mathbf{t}_2, i)$; $r \leftarrow (\mathtt{fp}, c_{\mathrm{j}}, i, r_1)$; $c_{\mathrm{j}} \leftarrow c_{\mathrm{j}} + 1$
Else if $q = (\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2, q_1, q_2)$ then
  For $i = 1, 2$ do $(r_i, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}}) \leftarrow \mathsf{RS}(q_i, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}})$
  $\mathbf{p} \xleftarrow{\cup} (\mathtt{fp}, \mathbf{t}_1, \mathbf{t}_2)$; $r \leftarrow (\mathtt{iij}, c_{\mathrm{p}}, r_1, r_2)$; $c_{\mathrm{p}} \leftarrow c_{\mathrm{p}} + 1$
Return $(r, \mathbf{q}, \mathbf{p}, \mathbf{j}, c_{\mathrm{q}}, c_{\mathrm{p}}, c_{\mathrm{j}})$

Fig. 19. Leakage algorithm used in Theorem 5, the proof of security for hybrid StI scheme HybStI. The subroutines HF, IJ, QP are given in Fig. 17.

Chicago R name	|R.Ats|	|R.T|	*at* densities		
			Min	Ave	Max
Bike_Racks	12	5164	4e-4	0.53	1.0
Census_Data	9	78	0.72	0.91	1.0
Crimes_2019	30	1.7e5	6e-6	0.17	1.0
Employee_Debt	7	1.4e4	3e-3	0.10	0.46
Fire_Stations	7	92	0.01	0.65	1.0
Grafitti	5	67	0.09	0.68	1.0
Housing	14	915	0.03	0.32	0.56
IUCR_Codes	4	404	5e-3	0.51	1.0
Land_Inventory	19	2.0e5	3e-4	0.41	1.0
Libraries	9	81	0.01	0.63	1.0
Lobbyists	7	1537	0.03	0.22	0.66
Police_Stations	15	23	0.04	0.87	1.0
Reloc_Vehicles	20	2672	2e-3	0.51	1.0
Street_Names	7	2582	2e-3	0.31	1.0
Towed_Vehicles	10	3339	1e-3	0.19	0.99

Sakila R name	|R.Ats|	|R.T|	*at* densities		
			Min	Ave	Max
Address	8	603	2e-3	0.93	1.0
Actor	4	208	0.02	0.56	1.0
Category	3	16	0.06	0.69	1.0
City	4	600	2e-3	0.55	1.0
Country	3	109	0.01	0.67	1.0
Customer	10	599	2e-3	0.5	1.0
Film	14	1002	1e-3	0.37	1.0
Film_Actor	3	5462	1e-4	0.07	0.18
Film_Cat	3	1000	1e-3	0.34	1.0
Inventory	4	5481	2e-4	0.30	1.0
Language	3	6	0.17	0.72	1.0
Payment	7	1.6e4	1e-4	0.44	1.0
Staff	11	2	0.5	0.86	1.0
Store	4	2	0.5	0.88	1.0
Rental	7	1.6e4	1e-4	0.47	1.0

Fig. 20. Summary statistics for the Chicago (left) and Sakila (right) data used in our simulations.