

ROGUE: Updatable Matrix Lookup Arguments and Applications to Verifiable Databases

Christodoulos Pappas
Hong Kong University of Science and
Technology
Hong Kong, China
cpappas@connect.ust.hk

Zhuo Cai
Hong Kong University of Science and
Technology
Hong Kong, China
zcaiam@connect.ust.hk

Dimitrios Papadopoulos
Hong Kong University of Science and
Technology
Hong Kong, China
dipapado@cse.ust.hk

Abstract

Proving the correctness of computations over a large dataset via succinct non-interactive arguments of knowledge (SNARKs) entails the large overhead of “loading” the dataset in the SNARK. However, certain computations may only need to access a small fraction of the dataset (e.g., a database query that only accesses a subset of table rows and then computes an aggregation function). The standard way of *efficiently* proving such computations is to use *lookup arguments with sublinear prover complexity* to load only necessary data to the SNARK. Unfortunately, all prior schemes are *static*: even a single change to the dataset forces the prover to re-run an expensive pre-processing step, linear to the dataset size. The only exemption is the recent work of Dutta et al., (CCS’24) that proposed a lookup argument with *amortized* sublinear updates—based on re-running the pre-processing phase periodically, when too many changes have been accumulated. In this work, we present Rogue, the first lookup argument with sublinear prover time and updates that *always* take time proportional only to the number of incurred changes. Indeed, Rogue is actually a *matrix lookup argument*, supporting entire row lookups in time proportional to the number of rows (and independent of their size)! It has very good practical performance, e.g., for a $2^{20} \times 2^7$ matrix and 2^{10} row accesses, Rogue achieves $\times 21$ -942 and $\times 76$ -30000 faster lookups and updates, respectively, compared to prior works. We then use Rogue to build RogueDB, the first verifiable database system for arbitrary SQL queries that supports authenticated indexes, hence achieves prover time sublinear to the database. Compared with prior schemes with succinct proofs, vSQL (Zhang et al., IEEE S&P’17) and PoneglyphDB (Gu et al., SIGMOD’25), we get $\times 42.8$ - $\times 8624.1$ and $\times 149.6$ - $\times 11362.4$ faster prover times, for various SQL queries from the TPC-H benchmark.

CCS Concepts

• Security and privacy → Cryptography.

Keywords

SNARKs; Lookup Arguments

ACM Reference Format:

Christodoulos Pappas, Zhuo Cai, and Dimitrios Papadopoulos. 2026. ROGUE: Updatable Matrix Lookup Arguments and Applications to Verifiable Databases. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS ’26)*, November 15–19, 2026, The Hague, Netherlands. ACM, New York, NY, USA, 25 pages. <https://doi.org/10.1145/3830454.3832695>

1 Introduction

Succinct Non-Interactive Arguments of Knowledge (SNARKs) are cryptographic protocols that enable a *prover* to convince a *verifier* about the correctness of a computation by generating a small and easy-to-verify proof. In recent years, SNARKs have been extensively used to scale or bridge blockchains [45, 66], and ensure the correctness of database queries [37, 44, 75] or the integrity of machine learning tasks [13, 46, 57]. In many application scenarios, including the aforementioned ones, there is a need to prove the correctness of computations that access and possibly update a small number of elements from a potentially huge dataset. In that case, directly deploying the SNARK over the entire dataset may make the prover’s overhead prohibitively large. Alternatively, the prover can first commit to the large dataset using some type of *authenticated data structure* (ADS), such as a Merkle tree [49] or a cryptographic accumulator [17]. Then, generate a SNARK proof ensuring: (1) the correctness of the computation and (2) the validity of the ADS membership proofs for dataset elements accessed in the computation. This approach has been extensively studied in the literature [23, 51, 54, 60, 68], but it has a key limitation: proving (2) requires expensive cryptographic operations such as hash computations within the SNARK, potentially making it the main bottleneck of the prover time.

Lookup Arguments offer an alternative way to address this issue. Without loss of generality, they abstract the dataset as a lookup table and introduce specialized protocols for proving (2) that remove the necessity for expensive cryptographic operations within the SNARK. In general, lookup arguments can be split into two main categories that rely on fundamentally different techniques and are motivated by entirely different application scenarios. The first one, which includes popular schemes such as Lasso, LogUp, and Plookup [25, 32, 38, 56, 62], builds efficient lookup arguments with prover time *proportional to both the table size and the number of lookups*—which makes sense in applications where the number of lookups is comparable to or exceeds the table size, or whenever the table can be decomposed into very small tables, which is often the case for the “truth tables” of various non-linear operations [62]. We stress that this does not necessarily fit our target application



This work is licensed under a Creative Commons Attribution 4.0 International License. *CCS ’26, The Hague, Netherlands*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
<https://doi.org/10.1145/3830454.3832695>

scenario, e.g., loading specific rows from a huge relational database without having overhead proportional to the database itself.

The second category, which is the main focus of this work, constructs lookup arguments with *prover cost sublinear in the table size n* . The first such scheme was Caulk [70], which relies on the ADS+SNARK paradigm described above with two key innovations. It uses a *vector commitment scheme (VC)* as ADS that allows efficient aggregation of individual membership proofs into a single small proof. Moreover, it introduces a “lightweight” specialized protocol for showing the correctness of that aggregated proof without having to verify it inside the SNARK. Despite this, the overhead of Caulk is quadratic to the number of performed lookups, making it impractical for even moderate-sized instances. Subsequently, an extensive line of works [21, 22, 26, 29, 31, 59, 71] has significantly improved upon this, leading to a protocol with optimal prover complexity [22], i.e., linear in the number of lookups. For the remainder, we focus on this category of lookup arguments.

Updates remain troublesome. Despite improvements in lookup prover time, prior works have a major limitation: *they are not efficiently updatable*. In particular, even a *single change* of an entry of the committed table (insertion/deletion/modification) forces the prover to spend time *at least linear to n* . That is because (1) in order to achieve fast lookup times, the prover has to update all membership proofs after a single change and (2) all existing schemes rely on a VC based on the univariate KZG polynomial commitment [39], which needs $O(n \log n)$ time to update all n membership proofs! Naturally, this inability to efficiently handle table updates restricts the applicability of existing lookup arguments in many heavily “dynamic” applications, such as zkRollups [28, 47, 79] or verifiable databases [37, 44, 77], where the dataset may be frequently updated.

To the best of our knowledge, the only work that attempts to address this issue is the recent scheme of Dutta et al., [28] (CCS 2024), which relies on the following natural idea: when an update takes place, directly update the table but *postpone* the re-computation of individual membership proofs until the number of changes becomes “significantly” large. While this makes update operations fast, it clearly introduces another issue: the membership proofs from the original table can not be used anymore when generating a lookup proof, as they may be outdated. At a high level, to overcome this issue, [28] proposes a protocol to efficiently generate membership proofs on-the-fly, in time proportional to the number of accessed elements *and the Hamming distance between the original and the changed table*. Clearly, as the number of updates grows, the Hamming distance increases to such an extent that computing “fresh” new proofs becomes preferable. The authors of [28] analyze the optimal delay before new proofs are computed as a function of the Hamming distance. Their optimal configuration yields **amortized** update and lookup prover overheads of $O(\sqrt{nk} \log n)$ where k is the number of lookups/updates. In practice, the **worst-case** complexity of updates is unfortunately $O(n \log n)$, as all membership proofs are periodically recomputed from scratch. This leads to the following question: “*Is it possible to construct a lookup argument with worst-case sub-linear lookup and update overhead?*”

Our results. We answer the above affirmatively by proposing ROGUE. Our contributions can be summarized as follows:

- (1) We introduce ROGUE, the first fully updatable lookup argument, achieving $O(k \log n)$ update time, for k changes.
- (2) We build ROGUE as a *matrix-lookup* argument [21, 26], a generalization of lookup arguments that enables the prover to commit to matrices and subsequently lookup entire rows. Unlike existing matrix-lookup arguments, our scheme has a lookup prover time that is *independent* of the size of rows.
- (3) We then use ROGUE to construct ROGUEDB, the first verifiable database system which enables a prover to show the correctness of Select or Update queries *in time sub-linear to the table size*.

Our Updatable Lookup Argument. ROGUE deviates from existing lookup arguments in that it commits the table using the *multi-linear version of KZG* [53] after first computing its *multi-linear extension (MLE)* (see Section 2 for the definition of MLE). This is a first step towards constructing efficient update and matrix lookup algorithms. However, although encoding tables as multilinear polynomials makes it easier to handle updates, there are several challenges that remain when proving update correctness, as we elaborate next.

Efficient Updates. To prove updates, (e.g., show that the table $\mathbf{T} \in \mathbb{F}^n$ has been updated correctly to $\mathbf{T}'$ at the indexes $\mathcal{I} = \{i_1, \dots, i_k\} \in [n]^k$) we have to generate a SNARK proof showing that: (i) all elements in indexes not in $\mathcal{I}$ remain unchanged (i.e., $\mathbf{T}[i] = \mathbf{T}'[i]$, for all $i \in [n] \setminus \mathcal{I}$) and (ii) all elements of $\mathbf{T}'$ in $\mathcal{I}$ have been correctly replaced with the elements of a vector $\mathbf{F}' \in \mathbb{F}^k$, (i.e., $\mathbf{T}'[i_j] = \mathbf{F}'[j]$, for all $i_j \in \mathcal{I}$). Crucially, the verification algorithm should only use the commitments of $\mathbf{T}, \mathbf{T}', \mathbf{F}'$ and $\mathcal{I}$ to validate the proof (see Section 2 for a formal description of the update relation) and the prover time should only be proportional to k —the number of changes.

Starting from (i), observe that even naively checking its validity requires $n - k \approx O(n)$ comparisons. Thus, constructing a proving algorithm for (i) that takes time only proportional to k is non-trivial¹. To this end, we make use of the following crucial observations: First, if (i) holds, then the vector $\mathbf{d} = \mathbf{T}' - \mathbf{T}$ would have only k non-zero elements. This indicates $\mathbf{d}$ is sparse and that someone can evaluate the MLE of $\mathbf{d}$, in time proportional to its non-zero elements. Secondly, and most importantly, if (i) does not hold, then, for any random point $\mathbf{r} \in \mathbb{F}^{\log n}$, the evaluation of the MLE of $\mathbf{d}$ at $\mathbf{r}$ equals $y = \sum_{i_j \in \mathcal{I}} \beta(i_j, \mathbf{r})(\mathbf{T}'[i_j] - \mathbf{T}[i_j])$, only with negligible probability! Based on these observations, we propose the following protocol: Given a random point $\mathbf{r}$, the prover generates (a) a SNARK proof showing the correct computation of y using our lookup algorithm to access all $\mathbf{T}[i_j], \mathbf{T}'[i_j]$, for $i_j \in \mathcal{I}$ and (b) a multi-linear KZG opening proof for the commitment of the MLE of $\mathbf{d}$ which is $C_{\mathbf{T}'} C_{\mathbf{T}}^{-1}$.

Unfortunately, it is still unclear how to prove (a) and (b) in $\tilde{O}(k)$ time. For instance, in (a) the prover does not have the membership proofs of $\mathbf{T}'$, which are necessary for efficiently generating lookup proofs for $\mathbf{T}'$. To overcome that issue, we use the algorithm proposed in [63] that enables the prover to update all proofs in $\tilde{O}(k)$ time and not in $\tilde{O}(n)$! As for (b), observe that the MLE of $\mathbf{d}$ has n coefficients—albeit most of them are 0. Consequently, naively generating a proof for (b) requires $O(n)$ time. To overcome that issue, we “exploit” the sparsity of that polynomial and propose a

¹In fact, this was also the main technical challenge of the previous work [28].

Table 1: Asymptotic comparison of lookup arguments. n is the size of the table, k is the number of lookups/updates. The first three rows are schemes without support for efficient updates ($\times$ indicates that Update requires re-running PreProcess from scratch). Symbol $*$ indicates amortized complexity for [28], $|\text{CRS}|$ the size of the prover’s public parameters and $\mathbb{F}$, $\mathbb{G}$, $\mathbb{P}$ denote field, group and pairing operations. Finally, $K \in [1, \sqrt{n}]$.

Scheme	PreProc.	Prove	Verify & $ \pi $	Update	$ \text{CRS} $
Caulk [70]	$n \log n \mathbb{G}$	$k^2 \mathbb{F}$	$1 \mathbb{G}$	$\times$	$n \mathbb{G}$
CQ-based [18, 26, 29, 78]	$n \log n \mathbb{G}$	$k \log k \mathbb{G}$	$1 \mathbb{G}$	$\times$	$n \mathbb{G}$
Campanelli et al. [22]	$n \log n \mathbb{G}$	$k \mathbb{G}$	$\log k \mathbb{G}$	$\times$	$n \mathbb{G}$
Dutta et al. [28]	$n \log n \mathbb{G}$	$\sqrt{nk} \log n^* \mathbb{G}$	$1 \mathbb{G}$	$\sqrt{nk} \log n^* \mathbb{G}$	$n \mathbb{G}$
ROGUE (Ours)	$n \log n \mathbb{G}$	$k \log n + K \mathbb{P}$	$\log^2 n \mathbb{P}$	$k \log n + K \mathbb{P}$	$n/K \mathbb{G}$

specialized KZG evaluation algorithm that takes only $\tilde{O}(k)$ time! What remains is to prove (ii) (i.e., the correctness of changes). This can be directly achieved using our lookup algorithm, which we will describe next.

Our Lookup Algorithm. To prove lookups we have to generate a proof showing that for a given vector $\mathbf{F}$ and index set $\mathcal{I}$, it holds that $\mathbf{T}[i_j] = \mathbf{F}[j]$, for all $i_j \in \mathcal{I}$, where the verification algorithm only uses the commitments of $\mathbf{F}$, $\mathcal{I}$ and $\mathbf{T}$. To achieve this, we rely on the protocol that aggregates multi-linear KZG proofs as described in [63]. Unfortunately, this alone is not enough since it requires the verifier to obtain $\mathbf{F}$ and $\mathcal{I}$, leading to prohibitive proof sizes and verification times. Instead, we propose a novel protocol that aggregates multi-linear KZG proofs when the verifier only holds the commitments to $\mathbf{F}$ and $\mathcal{I}$.

Fast Matrix Lookups. Actually, ROGUE is a *matrix lookup argument* [21, 26], in that it can encode a matrix $\mathbf{T} \in \mathbb{F}^{n \times m}$ and subsequently lookup entire rows of that committed matrix. This definition is a straightforward extension of lookup arguments to the case where every lookup returns entire “rows” of m consecutive elements. Standard lookup arguments can achieve this naively by aggregating the individual proofs of elements of rows. However, ROGUE is the first matrix lookup argument to achieve complexity *independent of the row size m* . Exploiting the properties of multi-linear extension polynomials, we can create membership proofs for entire rows of the matrix instead of individual elements. This allows us to generate lookup proofs for k rows by only accessing their commitments and aggregating their proofs.

Table 1 gives a performance comparison summary on a single table ($m = 1$). Observe that our scheme not only significantly outperforms prior work in both lookup and update time, but also provides an additional key advantage: it offers a trade-off between prover time and the size of the prover’s public parameters. Specifically, the prover can reduce the public parameter size to $O(\sqrt{n})$ at the cost of only a modest increase in lookup and update time to $O(k \log n + \sqrt{n})$, which incurs minimal concrete overhead in practice. Contrary to this, all existing lookup arguments require public parameters of size at least $O(n)$.

Verifiable Databases with Efficient Index Support. One of the most widely studied use-cases of verifiable computation is *verifiable database queries*, where the proofs pertain to the correct evaluation of query operators over outsourced databases (e.g., [43, 69, 75]). In fact, there has been a recent resurgence of interest for constructing verifiable database schemes (VDS) for relational databases and SQL queries, motivated by real-world applications such as efficiently and verifiably enabling smart contracts to access historical node data [11], or verifiable Blockchain analytics [10] and financial data reporting [7] (see [19] for additional interesting applications). Driven from this, startup companies in the zero-knowledge ecosystem, such as Axiom [5], Lagrange Labs [8], and Space&Time [9], are actively developing their own verifiable database solutions.

Despite the above, existing works on verifiable database schemes face some crucial limitations. First, several works in this field are limited to specific database operations, such as only range queries [43, 52], or only join queries [69], or a small subset of SQL queries [19, 24, 55], since they propose specialized authenticated data structures that can not support arbitrary complex queries. In fact, to the best of our knowledge, there are only three works vSQL [75, 76], zkSQL [44], and PoneglyphDB [37] that can handle arbitrary queries. Second, in all three of these systems, the time required to prove the correctness of a SQL query is always *proportional to the sizes of the tables involved in the query*. Simply put, the (possibly complex) query is abstracted as one or more circuits that take the entire table as input. As a simple example, a range query “SELECT * FROM LINEITEM WHERE l_shipdate > ‘d₁’ AND l_shipdate < ‘d₂’ ” will process the table line-by-line, check if it satisfies the query bounds, and include or omit it from the result. This is in *stark contrast* to the massive line of work on database management systems that explores how query performance can be sped up by deploying and dynamically maintaining special data structures called *indexes* [27, 80]. Perhaps the most famous such index is the B+ tree, allowing retrieval and modification of tuples in time *logarithmic* to the table size.

In this work, we propose ROGUEDB, the first implemented database system with explicit support for database indexes (and specifically B+ Trees), enabling prover times sublinear to the table size. As a starting point, we extend the definition of verifiable databases (as proposed in vSQL [75]) by enabling the prover to commit to indexes, prove that they have been constructed or updated correctly, and then use them to efficiently prove SQL queries. To achieve this, we use as a main building block our updatable lookup argument, ROGUE as it enables us to efficiently access and update the leaves of the index within the SNARK. Unfortunately, ROGUE alone is not enough to efficiently prove the correct computation of the index construction and update algorithms—as they are dominated by “SNARK-unfriendly” operations such as conditional if-then-else statements, that would hamper performance. Instead, we propose several novel optimized components which we briefly discuss next. *‘SNARK-friendly’ certification of index operations.* We first introduce novel certification algorithms that validate the correctness of each index operation, without recomputing it, in a SNARK-friendly manner. We then propose specialized sumcheck-based protocols for proving the correctness of these algorithms. Our starting point is the crucial observation that committing to the entire index (which

is originally a B+ tree) is extremely redundant. Instead, we only commit the leaves of the tree, which, in practice, constitute a sorted linked list. This not only makes lookup queries faster but also significantly simplifies the task of proving index construction and updates to merely checking whether the given committed input is a sorted linked list (see Section 4).

Optimized sub-protocols for SQL queries. In addition, we also introduce specialized protocols for proving the correctness of various SQL queries, which significantly reduce the concrete proving costs compared to prior works [37, 75]. Specifically, although we employ similar techniques to design our circuits for computing SQL queries as those used in [37, 75], we deviate from prior works in that we do not make black-box use of an existing proof system to prove their correct computation. Instead, we decompose the circuits into smaller ones that perform fundamental operations such as range checks or set operations and which are proven using specialized sumcheck protocols [16, 62, 64].

Novel protocol for proving non-foreign-key joins. Last but not least, we propose a specialized protocol for proving the correctness of non-primary-foreign-key joins in time *linear to the table sizes*. This is a huge improvement compared to the worst-case quadratic cost incurred by the prior works [37, 75] (which arises from proving the pairwise comparison of all tuples in the tables). We achieve this by constructing an arithmetic circuit that is similar to primary-key joins but makes the following additional checks: (1) the number of tuples with the same attribute in the joined table is equal to the product of the number of tuples with that attribute in both tables, and (2) all elements in the joined table are unique. Then, to prove its correct evaluation, we use a specialized protocol for histogram computation from [57].

Experimental Evaluation. We implemented our schemes in C++ and experimentally evaluated their performance. We extensively benchmarked ROGUE and compared it with (a) the state-of-the-art updatable lookup argument proposed in [28] (denoted with CQ-ULA), (b) the matrix lookup argument **SubIonK** [26] which is not updatable but achieves fast lookup times and (c) the “standard” approach (denoted with MT+SNARK), which commits to the matrix using a Merkle Tree and lookups or updates rows by opening them into the SNARK. Experimental results (section 5.1) show that with the exception of very small values of k , our lookup time is $\times 2.13$ -163, $\times 1.1$ -24 and $\times 2.4$ -45, and our update time is $\times 46$ - $\times 3030$, $\times 185$ -10⁶ and $\times 18$ -40 faster than CQ-ULA, **SubIonK** and MT+SNARK.

We also compared ROGUEDB with existing verifiable database schemes with succinct proofs, namely vSQL and PoneglyphDB. When proving range queries, ROGUEDB is $\times 13.6$ -22516, $\times 8.3$ -13720 and $\times 1.02$ -1593 faster than PoneglyphDB, vSQL, and our implementation without indexes. For non-primary-foreign key joins, ROGUEDB is $\times 301.1$ -505840 and $\times 4.88$ -8624 faster than PoneglyphDB and vSQL. Finally, we tested ROGUEDB on more complex TPC-H benchmark queries, in which we are $\times 84.2$ -181.3 and $\times 1.4$ -12.1 faster than PoneglyphDB and ROGUEDB without indexes.

2 Preliminaries

Notation. We denote $[n] = \{1, \dots, n\}$ and $\mathbb{F}$ a field of prime order p . Furthermore, we denote with $\mathbf{x}_i$ the i -th element of the vector $\mathbf{x} \in \mathbb{F}^n$. Likewise, we let $\mathbf{M}_i$ be the i -th row of the matrix $\mathbf{M} \in$

$\mathbb{F}^{n \times m}$. In addition, for a set $\mathcal{I} \in [n]^k$, we let $\mathbf{T}_{|\mathcal{I}} \in \mathbb{F}^{k \times m}$ be the matrix that contains only the rows of $\mathbf{T}$ with indexes $\mathcal{I}$. Finally, we can encode any vector $\mathbf{x} : \{0, 1\}^{\log N} \rightarrow \mathbb{F}$ as a multi-linear polynomial $f_{\mathbf{x}} : \mathbb{F}^{\log N} \rightarrow \mathbb{F}$ s.t. $f_{\mathbf{x}}(\mathbf{z}) = \sum_{\mathbf{b} \in \{0, 1\}^{\log N}} \beta(\mathbf{z}, \mathbf{b}) \mathbf{x}(\mathbf{b})$ where $\beta(\mathbf{z}, \mathbf{b}) = \prod_{i \in [\log N]} (b_i z_i + (1 - b_i)(1 - z_i))$ is the identity polynomial. We call $f_{\mathbf{x}}$, the **multi-linear extension** of $\mathbf{x}$.

Argument Systems. An *Argument System* [41] for an NP language $\mathcal{L}$ is a protocol involving a computationally bounded *prover* and *verifier*. Both parties share an instance $\mathbf{x}$ while the prover also knows a witness $\mathbf{w}$ and tries to convince the verifier that $\mathbf{x} \in \mathcal{L}$. An argument system is *knowledge sound* when there exists a probabilistic polynomial time (PPT) program, called an extractor, that extracts a valid witness for $\mathbf{x}$. In addition, if the verifier learns nothing besides the validity of $\mathbf{x}$, then the argument system is *zero-knowledge*. Finally, when the interaction is one-way and the communication size and verification time are poly-logarithmic to $|\mathbf{w}|$, then the argument system is *non-interactive* and *succinct*. If all these properties hold, we have a SNARK [36, 58]. For our verifiable databases (Section 4) we use SNARKs based on GKR [67]. We note that our circuits have regular and well-known structure, hence the vulnerability of [40] does not seem to apply (see Remark 3).

Generalized Inner Product Arguments. *Inner pairing product arguments*, first introduced by Bunz et al. [20], enable a prover to generate proofs for inner product relations between a committed vector of group elements (committed using the pairing commitment of Abe et al. [14]) and a committed vector of group or field elements. In this work, we will use such arguments to prove two relations.

The first one, called *TIPP*, enables the prover to show that for the commitments $C_T, C_A, C_B \in \mathbb{G}_T$ it knows $\mathbf{A} \in \mathbb{G}_1^n$ and $\mathbf{B} \in \mathbb{G}_2^n$ such that C_A and C_B are the pairing commitments of $\mathbf{A}$ and $\mathbf{B}$ respectively and that C_T is an inner pairing product of $\mathbf{A}$ and $\mathbf{B}$, namely, $C_T = \prod_{i \in [n]} e(\mathbf{A}_i, \mathbf{B}_i)^{r^i}$, where $r \in \mathbb{F}$ is a random point selected by the verifier. More formally, we define TIPP as follows:

$$\mathcal{R}_{TIPP} = \{(\mathbf{vk}, C_T, C_A, C_B \in \mathbb{G}_T, r \in \mathbb{F}); (\mathbf{A}, \mathbf{g} \in \mathbb{G}_1^n, \mathbf{B}, \mathbf{h} \in \mathbb{G}_2^n) : C_A = \prod_{i \in [n]} e(\mathbf{A}_i, \mathbf{h}_i) \wedge C_B = \prod_{i \in [n]} e(\mathbf{g}_i, \mathbf{B}_i) \wedge C_T = \prod_{i \in [n]} e(\mathbf{A}_i, \mathbf{B}_i)^{r^i}\}$$

Where $\mathbf{vk}, \mathbf{pk} = (\mathbf{g}, \mathbf{h})$ are the public parameters.

The second one, called *MIPP*, enables the prover to show that for commitments $C_A \in \mathbb{G}_T, C_b, C_T \in \mathbb{G}_1$, it knows $\mathbf{A} \in \mathbb{G}_1^n$ and $\mathbf{b} \in \mathbb{F}^n$ such that C_A is the pairing commitment of $\mathbf{A}$, $\mathbf{b}$ the pre-image of C_b and $C = \prod_{i \in [n]} \mathbf{A}^{b_i}$. More formally:

$$\mathcal{R}_{MIPP} = \{(\mathbf{vk}, C, C_A \in \mathbb{G}_T, C_b \in \mathbb{G}_1 \in \mathbb{F}); (\mathbf{A}, \mathbf{g} \in \mathbb{G}_1^n, \mathbf{h} \in \mathbb{G}_2^n, \mathbf{b} \in \mathbb{F}^n) : C_A = \prod_{i \in [n]} e(\mathbf{A}_i, \mathbf{h}_i) \wedge C_b = \prod_{i \in [n]} \mathbf{g}_i^{b_i} \wedge C = \prod_{i \in [n]} \mathbf{A}_i^{b_i}\}$$

Performance-wise, proving both relations requires $O(n)$ pairing operations while the proof size and verification time are $O(\log n)$. Finally, we note that in some cases, when proving an MIPP relation $\mathbf{b}$ might have succinct representation (e.g. a vector of the form $\mathbf{b} = (1, a^1, \dots, a^{n-1})$). In these cases, we do not have to commit to $\mathbf{b}$.

Polynomial Commitment Schemes. A *Polynomial Commitment Scheme* (PCS) [39, 53, 61] enables a prover to commit to an n -variate polynomial of degree d and later generate a proof that it correctly evaluated that polynomial at a point. Formally, a PCS consists of the algorithms: $(Gen, Commit, Eval, Verify)$. In more detail, *Gen*, generates some public parameters while *Commit*, takes a polynomial

f and produces a commitment C_f . Given f and a random point $\mathbf{x}$, $Eval$ outputs a proof showing that $y = f(\mathbf{x})$ and $Verify$ validates its correctness. A PCS is *knowledge sound* if, for any Probabilistic Polynomial Time (PPT) adversary that generates an accepting evaluation proof, there exists a PPT extractor that extracts the committed polynomial f , such that the probability that $f(\mathbf{x}) \neq y$ is negligible with respect to the security parameter λ . A PC scheme is *complete* if the verifier always accepts a valid proof and *zero-knowledge* if it learns nothing more other than $y, \mathbf{x}$.

In this work, we heavily rely on the Kopis PCS introduced by Setty et. al. [61], which can be seen as an improvement over the KZG PCS for multi-linear polynomials [53]. To simplify the explanation of our constructions, we “reformulate” Kopis as a PCS that commits to the multi-linear extension of a matrix $\mathbf{M} \in \mathbb{F}^{n \times m}$. We present a detailed construction of both the KZG and Kopis PCS in Appendix B.2. Performance-wise, the commitment of Kopis requires $O(n)$ pairing operations and $O(mn)$ group operations. Open requires $O(n)$ pairing operations, $O(m)$ group operations and $O(mn)$ field operations. The proof size and verification time are $O(\log m + \log n)$.

3 Our Updatable Matrix Lookup Argument

In this section, we present ROGUE, an updatable matrix lookup argument with the following properties. First, it enables a prover to show that all rows of a committed matrix $\mathbf{F} \in \mathbb{F}^{k \times m}$ are included in a (larger) committed matrix $\mathbf{T} \in \mathbb{F}^{n \times m}$ in time only proportional to k . Furthermore, our scheme enables a prover to show that it honestly modified k rows of $\mathbf{T}$ in time proportional to km .

More formally, ROGUE consists of the following algorithms (*KeyGen*, *PreProcess*, *Lookup*, *VerifyLookup*, *Update*, *VerifyUpdate*). At a high level, *KeyGen* takes as input the dimensions of the matrix n, m and produces proving and verification public parameters. *PreProcess* takes as input the matrix $\mathbf{T} \in \mathbb{F}^{n \times m}$ and outputs its commitment C_T along with auxiliary information needed to efficiently execute the *Lookup* and *Update* algorithms. *Lookup* takes as input the matrixes $\mathbf{F} \in \mathbb{F}^{k \times m}$, $\mathbf{T}$ an index set $I = \{i_1, \dots, i_k\} \in [n]^k$ and returns a proof π_{lkp} showing that $\mathbf{F}_j = \mathbf{T}_{i_j}$, for all $j \in [k]$. *VerifyLookup* uses the proof π_{lkp} along with the commitments of $\mathbf{T}, \mathbf{F}$ and I to validate its correctness. Next, *Update* takes as input the matrixes $\mathbf{F}' \in \mathbb{F}^{k \times m}$, $\mathbf{T}$ and the index set $I \in [n]^k$, and outputs the commitment of the updated matrix $\mathbf{T}'$ along with a proof π_{upd} showing that $\mathbf{T}'$ has been produced after replacing the i_j -th row of $\mathbf{T}$ with the j -th row of $\mathbf{F}'$, for all $j \in [k]$. Furthermore, it also updates the auxiliary information of the original matrix so that it can efficiently perform further lookups or updates on $\mathbf{T}'$. Finally, *VerifyUpdate* uses the proof π_{upd} and commitments of $\mathbf{T}, \mathbf{T}', \mathbf{F}'$ to validate its correctness.

Our updatable matrix lookup argument must satisfy *row linkability*, meaning that whenever a prover generates an accepting lookup proof, then it must know the tables $\mathbf{T}, \mathbf{F}$ and a set of indexes I such that $\mathbf{F} = \mathbf{T}_{|I}$. Furthermore, it must satisfy *update consistency*, i.e., whenever the prover generates an accepting update proof, then it should know the tables $\mathbf{T}, \mathbf{T}', \mathbf{F}'$ and the set of indexes I such that $\mathbf{T}'$ has been produced after replacing all rows of $\mathbf{T}$ at I with the rows in $\mathbf{F}'$. More formally:

DEFINITION 3.1. *An updatable matrix lookup argument is a tuple of six probabilistic polynomial-time algorithms (*KeyGen*, *PreProcess*, *Lookup*, *VerifyLookup*, *Update*, *VerifyUpdate*) defined as follows:*

- $\mathbf{pk}, \mathbf{vk} \leftarrow \text{KeyGen}(\lambda, n, m)$: Takes as input the security parameter λ , the maximum number of rows n columns m and outputs the proving and verification public parameters $\mathbf{pk}, \mathbf{vk}$.
- $C_T, \mathbf{aux} \leftarrow \text{PreProcess}(\mathbf{pk}, \mathbf{T})$: Takes as input the matrix $\mathbf{T} \in \mathbb{F}^{n \times m}$ and outputs its commitment C_T and some auxiliary information $\mathbf{aux}$.
- $0, 1 \leftarrow \left(\text{Lookup}(\mathbf{pk}, \mathbf{T}, \mathbf{F}, I), \text{VerifyLookup}(\mathbf{vk}, C_T, C_F, C_I) \right)$: The *Lookup* algorithm takes as input the matrixes $\mathbf{T}, \mathbf{F} \in \mathbb{F}^{k \times m}$, and an index set I and interacts with *VerifyLookup* showing that $\mathbf{F} = \mathbf{T}_{|I}$. *VerifyLookup* takes as input the commitments C_T, C_F and C_I returns 1 iff the communication transcript is valid.
- $0, 1 \leftarrow \left(\text{Update}(\mathbf{pk}, \mathbf{T}, \mathbf{F}', I), \text{VerifyUpdate}(\mathbf{vk}, C_T, C_{T'}, C_F, C_{I'}) \right)$: *Update* takes as input the matrixes $\mathbf{T}, \mathbf{F}'$ and the index set I and interacts with the verifier showing that $\mathbf{T}'$ is the update of $\mathbf{T}$ after replacing all of its rows in I with $\mathbf{F}'$ along with $\mathbf{T}'$. *VerifyUpdate* takes as input the update proof, the commitments $C_T, C_{T'}, C_F$ and $C_{I'}$ and returns 1 iff the communication transcript is valid.

Furthermore, for these algorithms the following must hold:

- **Lookup Completeness.** An updatable matrix lookup argument is *lookup complete* if for any security parameter λ and matrix dimensions n, m , matrix $\mathbf{T}$ and $\mathbf{F} \subseteq \mathbf{T}$ it holds that:

$$\Pr \left(\begin{array}{l} \mathbf{pk}, \mathbf{vk} \leftarrow \text{KeyGen}(1^\lambda, n, m, k), \\ C_T, \mathbf{aux} \leftarrow \text{PreProcess}(\mathbf{pk}, \mathbf{T}), \\ 1 = \left(\text{Lookup}(\mathbf{pk}, \mathbf{T}, \mathbf{F}, \mathbf{aux}), \right. \\ \left. \text{VerifyLookup}(\mathbf{vk}, C_T, C_F) \right) \end{array} \right) = 1$$

- **Update Completeness.** An updatable matrix lookup is *update complete* if for any security parameter λ and matrix dimensions n, m , matrix $\mathbf{T}$ and update circuit C_u , it holds that:

$$\Pr \left(\begin{array}{l} \mathbf{pk}, \mathbf{vk} \leftarrow \text{KeyGen}(1^\lambda, n, m, k), \\ C_T, \mathbf{aux} \leftarrow \text{PreProcess}(\mathbf{pk}, \mathbf{T}), \\ 1 = \left(\text{Update}(\mathbf{pk}, \mathbf{T}, \mathbf{F}', I), \right. \\ \left. \text{VerifyUpdate}(\mathbf{vk}, C_T, C_{T'}, C_F, C_{I'}) \right) \end{array} \right) = 1$$

- **Vector Linkability.** A updatable matrix lookup is *vector linkable* if for any PPT adversary $\mathcal{A}$ there exists a PPT extractor $\mathcal{E}$ such that:

$$\Pr \left(\begin{array}{l} \mathbf{pk}, \mathbf{vk} \leftarrow \text{KeyGen}(1^\lambda, n, m, k), \\ C_T, C_F, C_I \leftarrow \mathcal{A}(\mathbf{pk}, \mathbf{vk}), \\ 1 = \left(\text{Lookup}(\mathbf{pk}, \mathbf{T}, \mathbf{F}, \mathbf{aux}), \right. \\ \left. \text{VerifyLookup}(\mathbf{vk}, C_T, C_F) \right) : \\ \mathbf{F} \neq \mathbf{T}_{|I} | \mathbf{T}, \mathbf{F}, I \leftarrow \mathcal{E}^{\mathcal{A}}(\mathbf{pk}, \mathbf{vk}), \end{array} \right) < \text{negl}(\lambda)$$

- **Update Consistency.** An updatable matrix lookup argument is *update consistent* if for any PPT adversary $\mathcal{A}$ there exists a

PPT extractor $\mathcal{E}$ such that:

$$Pr \left(\begin{array}{l} \mathbf{pk}, \mathbf{vk} \leftarrow \text{KeyGen}(1^\lambda, n, m, k), \\ C_T, C_T', C_F, C_I \leftarrow \mathcal{A}(\mathbf{pk}, \mathbf{vk}), \\ 1 = \langle \text{Update}(\mathbf{pk}, \mathbf{T}, \mathbf{F}, \mathcal{I}), \\ \text{VerifyUpdate}(\mathbf{vk}, C_T, C_T', C_F, C_I) \rangle : \\ (\exists i_j \in \mathcal{I}, \mathbf{T}_{i_j} \neq \mathbf{F}_{i_j} \vee \exists i \in [n] \setminus \mathcal{I}, \mathbf{T}_i \neq \mathbf{T}_i') \\ \mathbf{T}', \mathbf{T}, \mathbf{F}, \mathcal{I} \leftarrow \mathcal{E}^{\mathcal{A}}(\mathbf{pk}, \mathbf{vk}) \end{array} \right) < \text{negl}(\lambda)$$

3.1 Our Construction

3.1.1 KeyGen & PreProcess Algorithms. Given the maximum number of rows n and columns m of the matrix, *KeyGen* generates the public parameters, consisting of KZG parameters for multi-linear polynomials [53] of maximum size $n \cdot m$ as well as the public parameters needed for the TIPP and MIPP protocols for instances of size n .

Next, given the public parameters and a matrix $\mathbf{T}$, *PreProcess* works as follows. First, it commits to $f_{\mathbf{T}} : \mathbb{F}^{\log mn} \rightarrow \mathbb{F}$, the multi-linear extension of $\mathbf{T}$, using the KZG PC scheme. Then, it computes the membership proofs for all the rows. For this, we use the observation that $\mathbf{T}_i$ is the i -th row of $\mathbf{T}$ if and only if there exist $\log n$ polynomials $Q_{i,j} : \mathbb{F}^{\log nm-j} \rightarrow \mathbb{F}$ such that: $f_{\mathbf{T}}(\mathbf{x}, \mathbf{y}) - f_{\mathbf{T}_i}(\mathbf{y}) = \sum_{j \in [\log n]} (\mathbf{x}_j - \mathbf{i}_j) Q_{i,j}(\mathbf{x}_{j+1}, \dots, \mathbf{x}_{\log n}, \mathbf{y})$ where $f_{\mathbf{T}_i} : \mathbb{F}^{\log m} \rightarrow \mathbb{F}$ is the multi-linear extension of $\mathbf{T}_i$. Hence, for the membership proofs for rows of $\mathbf{T}$, *PreProcess* computes the KZG commitments for $\{Q_{i,j}\}_{i \in [n], j \in [\log n]}$, denoted with $\{\pi_{i,j}\}_{i \in [n], j \in [\log n]}$ and stores them in $\mathbf{aux}$, along with the KZG commitments of all $f_{\mathbf{T}_i}$.

3.1.2 Lookup Algorithm. Given the matrixes $\mathbf{T} \in \mathbb{F}^{n \times s}$, $\mathbf{F} \in \mathbb{F}^{k \times s}$ and the index set $\mathcal{I} = \{i_1, \dots, i_k\}$, the *Lookup* algorithm generates a SNARK proof for the relation $\mathcal{R}_{lkp} = \{(C_T, C_F, C_I); (\mathcal{I}, \mathbf{F}, \mathbf{T}) : \mathbf{F} = \mathbf{T}_{|\mathcal{I}} \wedge C_T = \text{Com}_1(\mathbf{T}) \wedge C_F = \text{Com}_2(\mathbf{F}) \wedge C_I = \text{Com}_3(\mathcal{I})\}$.

High-Level Idea of our Protocol. Using the auxiliary information $\mathbf{aux}$ provided by the *PreProcess* algorithm, one can directly prove $\mathcal{R}_{lkp}$ by sending a proof that includes $\mathcal{I}$, the KZG commitments C_{F_j} of each row of the matrix $\mathbf{F}$, and their corresponding membership proofs on $\mathbf{T}$. Then, by utilizing these membership proofs, the verifier ensures that the pre-image of each C_{F_j} is the i_j -th row of the committed matrix, by checking whether $e(C_T/C_{F_j}, h) = \prod_{l \in [\log n]} e(\pi_{i_j, l}, h_l h^{-i_j^{(l)}})$ for each $j = 1, \dots, k$, where $i_j^{(l)}$ the l -th bit of i_j . Unfortunately, the proof size and verification time of this approach grow proportionally to k —whereas our goal is to obtain a proof size and verification time that is poly-logarithmic to k . To achieve this, the verifier delegates these checks to the prover as follows. First, it selects a random point $s \in \mathbb{F}$ and sends it to the prover. This enables the prover to batch the membership proofs by checking

$$\prod_{j \in [k]} e(C_T/C_{F_j}, h)^{s^j} = \prod_{l \in [\log n]} \left(\prod_{j \in [k]} e(\pi_{i_j, l}, h_l h^{-i_j^{(l)}})^{s^j} \right).$$

It then proves the correctness of the above equation using generalized inner product arguments [20]. Note that this idea has already been used to aggregate the membership proofs of the Hyperproofs vector commitment scheme [63]. However, their construction assumes that the verifier already holds all C_{F_j} and $\mathcal{I}$, leading to identical verification time and proof size complexities with the earlier approach—albeit, concretely smaller. Instead, we prove the above

batch membership equation using the pre-images of C_F and C_I . That, however, is not enough for proving that $\mathbf{F}$ is a sub-matrix of $\mathbf{T}$ since the verifier has no way of knowing whether the pre-image of C_I is well-formed, namely, its elements belong to $\{1, \dots, n\}$. Hence, we also need to prove the well-formedness of $\mathcal{I}$.

Protocol Description. Based on the above, we construct our *Lookup* algorithm which we present in **Construction 1**. To ease the presentation, we will describe it as an interactive protocol between the prover and verifier. Since it is a public coin protocol, we can make it non-interactive using the Fiat-Shamir heuristic. Having established that, the prover starts by committing to $\mathbf{F}$ and $\mathcal{I}$. To commit $\mathbf{F}$, it uses the KZG PCS, by first retrieving all KZG commitments $C_{F_{i_j}}$, $i_j \in \mathcal{I}$ from $\mathbf{aux}$ and then computing their pairing commitment. As for $\mathcal{I}$, the prover first sets $\mathbf{i}^{(l)} \in \{0, 1\}^k$, to be a vector containing the l -th bit of each index in $\mathcal{I}$. Next, it generates $\log n$ commitments, one for each $\mathbf{i}^{(l)}$ by parsing it as an $k \times 1$ matrix and then invoking the KZG PCS commit algorithm to compute $C_{idx}^{(l)}$. Finally, it sends $C_I \leftarrow \{C_{idx}^{(l)}\}_{l \in [\log n]}$ and C_F to the verifier. It is worth mentioning that both C_F and C_I are polynomial commitments over the multi-linear extensions of $\mathbf{F}$ and $\mathbf{i}$.

Then, we partition our protocol into two phases. First, the prover interacts with the verifier to prove the batch membership equation (steps 1-5). In the second one, it shows the well-formedness of the pre-image of C_I (steps 6,7).

Phase 1. Initially, the prover commits to the membership proofs by computing $C_\pi^{(l)}$, the pairing commitment of $(\pi_{i_1, l}, \dots, \pi_{i_k, l})$ for every $l \in [\log n]$ (step 1). Upon receiving these commitments, the verifier samples a random point $s \in \mathbb{F}$ and sends it to the prover. At this point, observe that we can re-write the above equation as $e(C_T^{\sum_{j \in [k]} s^j} / \prod_{j \in [k]} C_{F_j}^{s^j}, h) = \prod_{l \in [\log n]} C_p^{(l)}$, where $C_p^{(l)} = \prod_{j \in [k]} e(\pi_{i_j, l}, h_l h^{-i_j^{(l)}})^{s^j}$. Based on this, the prover computes and sends to the verifier $C^* = \prod_{j \in [k]} C_{F_j}^{s^j}$ and $C_p^{(l)}$ for all $l \in [\log n]$. The latter computes $S = \sum_{j \in [k]} s^j$ and checks if $e(C_T^S / C^*, h) = \prod_{l \in [\log n]} C_l$. It remains to validate that C^* and all $C_p^{(l)}$ were computed correctly.

For this, observe that C^* is a multi-exponentiation between the vectors $(C_{F_1}, \dots, C_{F_k})$ and $(s, \dots, s^k)$. Hence, to prove the correctness of C^* , both parties interact over the MIPP argument with public statement, the commitment C_F and the point s . Likewise for $C_p^{(l)}$, observe that it is an inner pairing product between $(\pi_{i_1, l}^s, \dots, \pi_{i_k, l}^s)$, $(h_l h^{-i_1^{(l)}}), \dots, (h_l h^{-i_k^{(l)}})$. Consequently, to prove the correctness of $C_p^{(l)}$ both parties interact over the TIPP argument using as public statements the point s and $C_\pi^{(l)}$, $C_h^{(l)} \leftarrow \prod_{j \in [k]} e(g_j, h_l h^{-i_j^{(l)}})$ (step 4). Although the verifier does not hold $C_h^{(l)}$, it can compute it in constant time by dividing the pairing commitment $\prod_{j \in [k]} e(g_j, h)$ with $C_{idx}^{(l)}$ (step 5). The former is independent of the index $\mathcal{I}$ and thus can be included as a verification key.

Phase 2: Then, the prover shows that every element of $\mathcal{I}$ belongs to $\{0, \dots, n-1\}$. This translates to proving that every element $\mathbf{i}^{(l)} \in \{0, 1\}^k$ for all $l \in [\log n]$. To achieve this, for each $l \in [\log n]$ the prover interacts with the verifier over the sumcheck protocol for proving the instance $0 = \sum_{j \in \{0, 1\}^{\log k}} \beta(\mathbf{r}, \mathbf{j})(1 - f_i^{(l)}(\mathbf{j}))f_i^{(l)}(\mathbf{j})$,

Construction 1: ROGUE

- **pp** $\leftarrow$ Generate(n, s, k): Invoke **pp**₁ $\leftarrow$ KZG.Gen($n \cdot s$), **pp**₂ $\leftarrow$ TIPP.Gen(k) and **pp**₃ $\leftarrow$ MIPP.Gen(k). Set **pp** $\leftarrow$ {**pp**₁, **pp**₂, **pp**₃}.
- **aux** $\leftarrow$ PreProcess(**pp**, **T** $\in \mathbb{F}^{n \times s}$): Compute the auxiliary information needed to efficiently generate lookup proofs:
 - (1) Let $f_T : \mathbb{F}^{\log ns} \rightarrow \mathbb{F}$ be the multi-linear extension of **T**. Compute $C_F \leftarrow g^{f_T(t_1, \dots, t_{\log ns})}$ and $C_{f_i} \leftarrow g^{f_i(t_{\log n}, \dots, t_{\log ns})}$, $\forall i \in [n]$.
 - (2) For each $i \in [n]$ and $j \in [\log n]$, compute $\pi_{i,j} \leftarrow g^{Q_{i,j}(t_{j+1}, \dots, t_{\log ns})}$, such that $f_T(x, y) - f_T^{(i)}(y) = \sum_{j \in [\log n]} (x_j - i_j) Q_{i,j}(x_{j+1}, \dots, x_{\log ns})$.
 - (3) Set **aux** $\leftarrow$ { C_T , { C_{f_i} } $_{i \in [n]}$, { $\pi_{i,j}$ } $_{i \in [n], j \in [\log n]}$ }.
- $\langle \mathcal{P}.\text{Lookup}(\mathbf{pk}, \mathcal{I}, \mathbf{T}, \mathbf{F}, \mathbf{aux}), \mathcal{V}.\text{Lookup}(C_T, C_F, C_I) \rangle$: Let C_F is the Kopis commitment of **F** while C_I consists of $\log n$ commitments $C_{idx}^{(l)}$, for $l \in [\log n]$, such that $C_{idx}^{(l)}$ is the Kopis commitment of the matrix $\mathbf{i}^{(l)} \in \{0, 1\}^{k \times 1}$ storing the l -the bit of each element in $\mathcal{I}$.
 - (1) $\mathcal{P}$: Compute $C_\pi^{(l)} \leftarrow \prod_{j \in [k]} e(\pi_{i_j, l}, h_j)$ for all $l \in [\log n]$. Send { $C_\pi^{(l)}$ } $_{l \in [\log n]}$ to $\mathcal{V}$.
 - (2) $\mathcal{V}$: Randomly pick $s \in \mathbb{F}$ and send it to $\mathcal{P}$. Set $C_h^{(l)} \leftarrow (\prod_{j \in [k]} e(g_j, h^{t_j})) / C_{idx}^{(l)} = \prod_{j \in [k]} e(g_j, h^{t_j - i_{j,l}})$.
 - (3) $\mathcal{P}$: Compute and send to $\mathcal{V}$ (1) $C^* \leftarrow \prod_{j \in [k]} C_{f_j}^{s_j}$ and (2) $C_p^{(l)} \leftarrow \prod_{j \in [k]} e(\pi_{i_j, l}, h^{t_j - i_{j,l}})^{s_j}$ for all $l \in [\log n]$.
 - (4) $\mathcal{P} - \mathcal{V}$: Interact over the following protocols:
 - (a) For each $l \in [\log n]$: A TIPP argument to prove that $C_p^{(l)}$ is well formed using $s, C_h^{(l)}$ and $C_\pi^{(l)}$.
 - (b) Invoke the MIPP protocol to prove that C^* is well formed using s and C_F .
 - (5) $\mathcal{V}$: Compute $C_T^{\sum_{i \in [k]} s^i}$, and check if $e(C_T^{\sum_{i \in [k]} s^i} / C^*, h) = \prod_{l \in [\log n]} C_p^{(l)}$. Finally, select a random point $\mathbf{r} \in \mathbb{F}^{\log k}$ and send it to $\mathcal{P}$.
 - (6) $\mathcal{P} - \mathcal{V}$: For each $l \in [\log n]$, interact using the sumcheck protocol for the instance $0 = \sum_{j \in \{0,1\}^{\log k}} \beta(\mathbf{r}, \mathbf{j}) (1 - i_{j,l}) i_{j,l}$. At the end, $\mathcal{V}$ ends up with a claimed evaluation y_l of the multi-linear extension of $(i_{1,l}, \dots, i_{k,l})$ at a random point $\mathbf{r}^{(l)} \in \mathbb{F}^{\log k}$.
 - (7) $\mathcal{P}$: Using $C_{idx}^{(l)}$ generates an evaluation proof π_l for y_l at $\mathbf{r}^{(l)}$. Finally sends the $\log n$ proofs to $\mathcal{V}$ which validates their correctness.
- $\langle \mathcal{P}.\text{Update}(\mathbf{pk}, \mathbf{T}, \mathbf{F}', \mathcal{I}, \mathbf{aux}), \mathcal{V}.\text{Update}(C_T, C_{T'}, C_{F'}, C_I) \rangle$: Let $C_{F'}$ be the Kopis commitment of **F'** and C_I be the commitment of the index set $\mathcal{I}$.
 - (1) $\mathcal{P}$: For all rows $\mathbf{F}'_j$ in **F'**, invoke UpdateProofs(**aux**, $i_j, \mathbf{F}'_j$) to compute the updated auxiliary parameters **aux'**.
 - (2) $\mathcal{P} - \mathcal{V}$: Interact following the lookup protocol, to prove that $\mathbf{F}' = \mathbf{T}'_{|\mathcal{I}}$.
 - (3) $\mathcal{P}$: Let $\delta(\mathbf{x}, \mathbf{y}) = \sum_{j \in \mathcal{I}} \beta(\mathbf{x}, \mathbf{j}) f_j(\mathbf{y})$. Likewise we define $\delta'(\mathbf{x}, \mathbf{y})$ but use the multi-linear extensions $f'_j(\mathbf{y})$ of the rows of **F'**. Compute the KZG commitments C_δ and C'_δ of $\delta(\mathbf{x}, \mathbf{y})$ and $\delta'(\mathbf{x}, \mathbf{y})$ respectively. Send C_δ and C'_δ to $\mathcal{V}$.
 - (4) $\mathcal{V}$: Pick a random point $\mathbf{r} \in \mathbb{F}^{n \times s}$ and send it to $\mathcal{P}$.
 - (5) $\mathcal{P}$: Using C_δ and C'_δ generate evaluation proofs π_δ, π'_δ of y_δ and y'_δ at $\mathbf{r}$. Send $\pi_\delta, \pi'_\delta, y_\delta, y'_\delta$ to $\mathcal{V}$.
 - (6) $\mathcal{P}$: Generate a CP-SNARK proof for $\mathcal{R} = \{(C_F, C_{F'}, C_I, y_\delta, y'_\delta, \mathbf{r}; \mathbf{F}, \mathbf{F}', \mathcal{I}) : y_\delta = \sum_{i \in \mathcal{I}} \beta(\mathbf{r}_1, \mathbf{i}) f_i(\mathbf{r}_2) \wedge y'_\delta = \sum_{i \in \mathcal{I}} \beta(\mathbf{r}_1, \mathbf{i}) f'_i(\mathbf{r}_2)\}$, where f_i, f'_i the multi-linear extensions of $\mathbf{F}_i, \mathbf{F}'_i$, $\mathbf{r} = (\mathbf{r}_1, \mathbf{r}_2) \in \mathbb{F}^{\log n} \times \mathbb{F}^{\log s}$ (see Appendix C.1.1).

where $f_i^{(l)}$ the multi-linear extension of $\mathbf{i}^{(l)}$. In the final round, the verifier ends up with an evaluation claim of $f_i^{(l)}$ at a random point, whose correctness can be proven by treating $C_{idx}^{(l)}$ as a Kopis PC.

3.1.3 Update Algorithm. Given the matrixes $\mathbf{T} \in \mathbb{F}^{n \times m}$, $\mathbf{F}' \in \mathbb{F}^{k \times m}$ and an index set $\mathcal{I} = \{i_1, \dots, i_k\}$, we want to show that **T** has been correctly updated to **T'** after replacing the rows of **T** in $\mathcal{I}$ with the rows of **F'**. In particular, we want to ensure that (1) $\mathbf{T}'_{i_j} = \mathbf{F}'_j$, for all $j \in [k]$ and (2) all other rows remain unchanged, namely, $\mathbf{T}'_i = \mathbf{T}_i$ for all $i \in [n] \setminus \mathcal{I}$. More formally, we want to generate a SNARK proof for the relation $\mathcal{R}_{upd} = \{(C_T, C_{T'}, C_{F'}, C_I; \mathbf{T}, \mathbf{T}', \mathcal{I}) : \mathbf{T}'[i_j] = \mathbf{F}'[j], \forall j \in [k] \wedge \mathbf{T}[j] = \mathbf{T}'[j], \forall j \in [n] \setminus \mathcal{I}\}$.

Protocol Description. To prove the above relation, we partition our protocol into two phases. First, the prover convinces the verifier that **F'** has been correctly included in **T'**. Then, it proves that all other rows of **T'** and **T** are identical.

Proving the Correct Inclusion of $\mathbf{F}'$. In this phase the prover has to show that $\mathbf{F}' = \mathbf{T}'_{|\mathcal{I}}$ which can be done using our *Lookup* algorithm. However, to efficiently utilize the lookup algorithm, the prover must have access to all membership proofs of the new vector **T'** but currently, it only holds those of **T**. Naively, the prover could compute all membership proofs of **T'** from scratch. However, this would blow up the prover time to $O(nm \log n)$ —whereas we opt for a prover complexity proportional to km . Instead, the prover uses the algorithm proposed by [63] which allows it to efficiently update

all membership proofs of **T** to **T'** in a time that is *proportional only to the number of changes between $\mathbf{T}$ and $\mathbf{T}'$* .

Proving Well-Formedness of $\mathbf{T}'$. What remains is to show that **T'** is well-formed, namely, that all rows of **T'** not in $\mathcal{I}$ are the same as those of **T**. For this, we first introduce matrices $\mathbf{T}_\delta, \mathbf{T}'_\delta : \mathbb{F}^{n \times m}$ such that their rows restricted to $\mathcal{I}$ are equal to $\mathbf{T}_{|\mathcal{I}}$ and $\mathbf{T}'_{|\mathcal{I}}$, respectively, while all other rows are filled with zeros. Observe that **T'** is well-formed iff: $\mathbf{T}' = \mathbf{T} - \mathbf{T}_\delta + \mathbf{T}'_\delta$.

To prove the above, the prover first sends the KZG commitments of the multi-linear extensions of $\mathbf{T}_\delta$ and $\mathbf{T}'_\delta$ denoted with $C_\delta, C'_{\delta'}$. Upon receiving them, the verifier checks if $C_{T'} = C_T C_{\delta'} / C_\delta$. Unfortunately, this is not enough to prove the well-formedness of **T'** since the verifier has no clue whether the pre-images of C_δ or $C_{\delta'}$ are computed honestly. To ensure their correctness, the verifier picks a random point $\mathbf{r} = (\mathbf{r}_1, \mathbf{r}_2) \in \mathbb{F}^{\log n + \log m}$ and sends it to the prover. At this point, note that C_δ and $C_{\delta'}$ are well-formed, if and only if they evaluate at y_δ and $y_{\delta'}$ respectively, where $y_\delta = \sum_{i \in \mathcal{I}} \beta(\mathbf{r}_1, \mathbf{i}) f_{\mathbf{F}_i}(\mathbf{r}_2)$ and $f_{\mathbf{F}_i}$ is the multi-linear extension of the i -th row of $\mathbf{F} = \mathbf{T}_{|\mathcal{I}}$ (likewise we define $y_{\delta'}$). Thus, the prover first sends y_δ and $y_{\delta'}$ to the verifier and (1) uses $\mathbf{F}, \mathbf{F}'$ to prove the computation of $y_\delta, y_{\delta'}$ and (2) generates evaluation proofs of $C_\delta, C'_{\delta'}$ at $\mathbf{r}$.

For (1), the prover first computes the Kopis commitment of **F**, sends it to the verifier and then invokes the lookup algorithm to show that the committed **F** equals to $\mathbf{T}_{|\mathcal{I}}$. Then, it produces a SNARK proof showing the correct computation of an arithmetic circuit that takes as (private) input $\mathcal{I}, \mathbf{F}, \mathbf{F}'$ and (public) input $\mathbf{r} = (\mathbf{r}_1, \mathbf{r}_2)$ and outputs y_δ and $y_{\delta'}$. For (2), the prover generates two

KZG evaluation proofs of C_δ and C'_δ at $\mathbf{r}$ and sends them to the verifier which checks their validity. At first glance, this last step seems to prover time linear to the size of $\mathbf{T}, \mathbf{T}'$ since both δ and δ' have nm coefficients. However, note that these polynomials are sparse, namely, they only have km non-zero coefficients. Based on this observation, we slightly modify the evaluation algorithm of the KZG PCS to produce their evaluation proofs in time $O(km \log n)$. We start with the simple case where $k = 1$ and later show how to adapt our algorithm for the general case where $k > 1$.

Case where $k = 1$. Let $\delta(\mathbf{x}, \mathbf{y}) = \beta(\mathbf{x}, \mathbf{i})f(\mathbf{y})$ where $\mathbf{i} \in \{0, 1\}^{\log n}$ and $f : \mathbb{F}^{\log m} \rightarrow \mathbb{F}$. Furthermore, let $\mathbf{r} \in \mathbb{F}^{\log nm}$ be the evaluation point. To generate an opening proof we need to commit to the quotient polynomials $q_i : \mathbb{F}^{\log m + \log n - i} \rightarrow \mathbb{F}$ such that $\delta(\mathbf{t}) - \delta(\mathbf{r}) = \sum_{i \in [\log mn]} (\mathbf{t}_i - \mathbf{r}_i) q_i(\mathbf{t}_{i+1}, \dots, \mathbf{t}_{\log mn})$. For this, observe that

$$\begin{aligned} \delta(\mathbf{x}, \mathbf{y}) &= (\mathbf{x}_1 - \mathbf{i}_1 - \mathbf{r}_1 + \mathbf{r}_1) \beta(\mathbf{x}_{2:\log n}, \mathbf{i}_{2:\log n}) f(\mathbf{y}) = \\ &= (\mathbf{x}_1 - \mathbf{r}_1) \beta(\mathbf{x}_{2:\log n}, \mathbf{i}_{2:\log n}) f(\mathbf{y}) + (\mathbf{r}_1 - \mathbf{i}_1) \beta(\mathbf{x}_{2:\log n}, \mathbf{i}_{2:\log n}) f(\mathbf{y}) = \\ &= (\mathbf{r}_1 - \mathbf{i}_1) q_1(\mathbf{x}_{2:\log n}, \mathbf{y}) + \delta'(\mathbf{x}_{2:\log n}, \mathbf{y}) \end{aligned}$$

We then recursively repeat the same process over $\delta'(\mathbf{x}_{2:\log n}, \mathbf{y}) = \beta(\mathbf{x}_{2:\log n}, \mathbf{i}_{2:\log n}) f'(\mathbf{y})$, where $f'(\mathbf{y}) = (\mathbf{r}_1 - \mathbf{i}_1) f(\mathbf{y})$ until we reach the $\log n$ -th variable, where we use the standard KZG evaluation algorithm. At this point, observe that for $i \in [\log n]$ the quotient polynomials are of the form $q_i(\mathbf{x}_{i:\log n}, \mathbf{y}) = c_i \beta(\mathbf{x}_{i+1:\log n}, \mathbf{r}_{i+1:\log n}) f(\mathbf{y})$, where $c_i \in \mathbb{F}$ a constant. In other words, they have only m non-zero coefficients, and thus computing the KZG commitments of $q_1, \dots, q_{\log n}$ requires only $\log n$ multi-scalar multiplications of size m . Given that the rest of the quotients have size less than m , we end up with a total prover complexity of $O(m \log n)$!

Case where $k > 1$. In this case, the prover computes a KZG opening proof for every sub-polynomial $\beta(\mathbf{x}, \mathbf{i}) f_i(\mathbf{y})$ where $i \in \mathcal{I}$, and then accumulates all different proofs in a single one by computing $q_j = \prod_{i \in \mathcal{I}} q_{j,i}$. It is not hard to see that the total prover time is $O(km \log n)$ since the prover has to generate k opening proofs and to accumulate them takes only $O(k \log n)$ group operations.

Finally, we claim the following which we prove in Appendix C:

THEOREM 3.1. *For any matrix $\mathbf{T} \in \mathbb{F}^{n \times m}$, ROGUE is an updatable matrix lookup argument with $O(k \log n)$ lookup complexity and $O(\log^2 n)$ proof size/verification time where k the number of row lookups. Furthermore, it has $O(km \log n)$ update complexity, and $O(\log^2 n)$ proof size/verification time where k the changed rows.*

Optimization: Reducing the Prover's Public Parameters. Up to this point, we explained our protocol using the multi-linear KZG scheme. This, however, leads to public parameters size proportional to the table size, which significantly increases the space requirements of the prover. Here, we show how to reduce the size of the public parameters by replacing the multi-linear KZG PCS with the Kopsis PCS [61]. Initially, we change the *PreProcess* algorithm by allowing the prover to take an additional parameter K which it uses to partition the matrix $\mathbf{T} \in \mathbb{F}^{n \times m}$ into K sub-matrixes $\mathbf{T}^{(i)} \in \mathbb{F}^{n/K \times m}$, for all $i \in [K]$. Then, it runs the KZG-based *PreProcess* algorithm to compute all $C_{\mathbf{T}^{(i)}}$ and computes $C_{\mathbf{T}} = \prod_{i \in [K]} e(C_{\mathbf{T}^{(i)}}, \mathbf{h}_i)$. Next, we modify the lookup and update protocol as follows:

Lookup Protocol. This protocol, remains almost the same, with the only difference being that the number of TIPPes we invoke is now n/K and not n . The only part that significantly changes is step 6.

In particular, the verifier can not compute $C_{\mathbf{T}}^{\sum s^i}$, since $C_{\mathbf{T}}$ is now a Kopsis commitment. Instead, it should compute $\prod_{i \in [K]} C_{\mathbf{T}^{(i)}}^{\sum_{j \in \mathcal{J}_i} s^j}$, where $\mathcal{J}_i \in [k]^{k_i}$ contains all indexes j such that $(i-1)n/K \leq l_j < i \cdot n/K$ and $l_j \in \mathcal{I}$. Naively, the verifier could receive $\mathcal{I}$, all $C_{\mathbf{T}^{(i)}}$ and compute the above by itself. However, this would blow up verification time and proof size. Instead, the verifier delegates this check to the prover. This is done by committing to $\mathbf{s} = [\sum_{j \in \mathcal{J}_1} s^j, \sum_{j \in \mathcal{J}_2} s^j, \dots, \sum_{j \in \mathcal{J}_K} s^j]$ and then (1) following the MIPP protocol to prove the correct computation of $\prod_{i \in [K]} C_{\mathbf{T}^{(i)}}^{\sum_{j \in \mathcal{J}_i} s^j}$ and (2) showing the correct computation of $\mathbf{s}$ using the specialized protocol for proving Histograms as described in [57].

Update Protocol: As for the update protocol, all parts can be directly adapted in the setting where we commit the tables using the Kopsis PCS except from those for committing to and opening $C_\delta, C_{\delta'}$. For these, the prover first parses δ and δ' as $(\delta^{(1)}, \dots, \delta^{(K)})$ (and likewise for δ') and commits them using the Kopsis PCS. To generate their evaluation proofs, it uses the Kopsis PCS but modified so that it maintains a $O(K + km/K \log n)$ prover time. In particular, when aggregating $(\delta^{(1)}, \dots, \delta^{(K)})$, it only keeps track of the non-zero sub-polynomials. Then, to generate the KZG evaluation proof of the aggregated polynomial, it uses the evaluation algorithm described in the above section.

Performance-wise, our modified lookup protocol maintains the same proof size and verification time but increases the prover time to $O(K + k \log n)$, where the $O(K)$ factor stems from the proof generation of (1) and (2). Likewise, the verification time and proof size of the update protocol remain the same but the prover time increases to $O(ks \log n + K)$. The additional $O(K)$ factor stems from the modified lookup protocol and the commitment/evaluation algorithm for the polynomials δ, δ' .

3.2 Proving RAM computations with ROGUE

The most common use of lookup arguments in prior works is the execution of RAM programs with read/write memory accesses. One concrete studied use-case is zkRollups [6, 60, 65] that have emerged as a promising solution to mitigate the high on-chain costs of Blockchains by processing transactions off-chain and publishing on-chain only a small and easy-to-verify proof of correct execution. At a very high level, zkRollups require a prover to produce a SNARK proof showing that an initial global state $\mathbf{T}$ has been correctly transitioned into a final global state $\mathbf{T}'$ after running (possibly several) programs (e.g., transactions or smart contracts) where the global states $\mathbf{T}, \mathbf{T}'$ can be represented as matrixes, with each row corresponding to the state of an individual program (typically of size $< 1\text{KB}$ [15]). To achieve this, zkRollups deployed in practice [6, 60] commit to the global state using a Merkle-Tree commitment scheme, and consequently to access any element from the global state they have to validate its membership proof within the SNARK. This makes reads and writes to the global state one of the main bottlenecks in prover time [28, 51]. Using ROGUE instead, we can circumvent that issue by having the prover commit to the global state using the *PreProcess* algorithm and then prove (1) the validity of all read and writes done in the global state using the update algorithm of ROGUE and (2) the correct computation of the arithmetic circuit $C_{\mathcal{U}}$ that encodes the programs been proven.

4 Verifiable Databases with Indexes

In this section, we introduce the first verifiable database scheme that integrates indexes. Specifically, our scheme allows the prover to commit not only to tables, as done in previous works [37, 75], but also to indexes, leveraging them to prove the correctness of specific queries in time sub-linear to the table size. Additionally, when dealing with update queries, our scheme has the following desirable property: It allows the prover to efficiently maintain the committed indexes by enabling it to prove that they have been correctly updated in time, only proportional to the number of modified tuples. First, we propose a definition for *verifiable databases with indexes*, expanding the verifiable databases definition of [75] to also accommodate auxiliary query indexes. Our scheme consists of the algorithms: (*Gen*, *CommitTable*, *VerifyCommit*, *ProveQuery*, *VerifyQuery*, *ProveUpdate*, *VerifyUpdate*). More specifically, *Gen* takes as input the maximum possible table dimensions and uses ROGUE’s *KeyGen* to produce the public parameters necessary for committing tables, indexes, and proving queries. *CommitTable*, takes as input a table $\mathbf{T} \in \mathbb{F}^{n \times m}$, a set of attributes $\mathcal{J}$ for which an index is to be created, and (i) outputs a commitment to the table $C_{\mathbf{T}}$ and (ii) a commitment to each index $C_{\mathbf{T}}^i$ along with a proof π_i that it was computed honestly. For each $i \in \mathcal{J}$, *VerifyCommit* uses $C_{\mathbf{T}}^i$ and $C_{\mathbf{T}}$ to validate the correctness of π_i . After committing to the tables and indexes, the prover can now prove queries and update tables or indexes. In particular, *ProveQuery*, takes as input a SQL query as well as the tables or indexes used to evaluate the query and generates a proof of its correct computation. Meanwhile, *VerifyQuery* uses the index and table commitments along with the result of the query to verify the correctness of the proof. Finally, *ProveUpdate* takes as input a table $\mathbf{T}$, its indexes, and an update query and generates a proof that both the table and the indexes have been updated properly, while *VerifyUpdate* checks the correctness of the proof using the index and table commitments. We give a formal definition of our scheme and a theorem along with its proof in Appendix D.1. Here, we describe each algorithm of our scheme in more detail.

4.1 Committing to Tables and Indexes

For table $\mathbf{T} \in \mathbb{F}^{n \times m}$, the prover first computes its commitment $C_{\mathbf{T}}$ with the commitment algorithm of Kopis PCS over $f_{\mathbf{T}} : \mathbb{F}^{\log(nm)} \rightarrow \mathbb{F}$, the multi-linear extension of $\mathbf{T}$. If there are no indexes to be made (e.g., $\mathcal{J} = \emptyset$), the algorithm just outputs $C_{\mathbf{T}}$. Otherwise, it also constructs the index $\mathcal{T}_i$ for each attribute $i \in \mathcal{J}$. In our case, we consider indexes $\mathcal{T}_i$ that maintain a sorted list of the tuples, according to their i -th attribute.

In practice, this is typically done using B+ Trees—the most commonly used index structure in DB systems [50]. One important observation is that we do not need to commit to all the tree nodes (as has been in other works, e.g., for decision trees or random forests [21, 57, 72]). We simply need to commit to the leaf nodes, which simplifies our task. Intuitively, this stems from the following facts: First, all information needed to answer queries with this index resides on leaf nodes, while the sole purpose of the non-leaf nodes of a B+ Tree is to facilitate the efficient retrieval of leaf nodes. Second, and as we will elaborate next, because the leaf nodes of $\mathcal{T}$ form a sorted linked list, each index operation can be efficiently certified by only having access to the retrieved leaf nodes. Hence, we will

use our matrix lookup argument to commit to the leafs of the B+ tree only. To ease explanation, below we first consider the case where the table has a single attribute (e.g., $m = 1$); in Appendix D.3, we show how to handle the case where $m > 1$.

Committing to an Index. Let $\mathcal{T}$ be the index formed after the insertion of n keys. Recall that $\mathcal{T}$ is a B+ tree and that it consists of $\Theta(n/K)$ nodes, where K is a constant that represents the maximum number of keys each node can store. We parse the ordered list of tree leafs, as a matrix of dimensions $\Theta(n/K) \times \Theta(K)$, where every row corresponds to a leaf node of $\mathcal{T}$, commit to it using our matrix lookup argument, and generate a proof for its correct computation.

In more detail, the prover first converts each leaf node to a $(K+3)$ -sized vector of the form $[id, ptr, k_{min}, \mathbf{k}]$ where id is the identifier of the node, ptr and k_{min} is the identifier and minimum key value of its right neighbor, and $\mathbf{k} \in \mathbb{F}^K$ (padded if necessary) is the vector containing all key values of that node. After converting each leaf node to a vector, the prover forms $\mathbf{T}^{idx}$ and commits to it using the *PreProcess* algorithm of our matrix lookup argument. At this point, it is worth highlighting some conventions used in forming $\mathbf{T}^{idx}$. First, note that the rows of $\mathbf{T}^{idx}$ correspond to a sorted list. We set the head of this list to be a dummy vector containing only zeros except from ptr and k_{min} where it stores the identifier and minimum value of the left-most leaf node of $\mathcal{T}$. Furthermore, the row corresponding to the tail of the list has its ptr and k_{min} values set to $+\infty$ —a pre-agreed field element. Finally, $\mathbf{T}^{idx}$ also maintains multiple dummy rows filled with zeros which will be used for inserting new keys.

Proving Index Correctness. What remains for the prover is to show the well-formedness of the committed index. This is done by generating a SNARK proof showing that (a) all keys in the (committed) $\mathbf{T}^{idx}$ equal to the (committed) $\mathbf{T}$ and (b) all nodes in the (committed) $\mathbf{T}^{idx}$ form a sorted list. Based on this, we will show how to translate these conditions into an arithmetic circuit and efficiently prove its correct computation. Condition (a) corresponds to a set equality between $\mathbf{T}$ and the keys stored in each node of $\mathbf{T}^{idx}$, which can be proved using a circuit that encodes a multiplication tree [64, 75]. Hence, we will focus on checking whether the leaf nodes correspond to a sorted linked list. This holds if and only if (i) each node in $\mathbf{T}^{idx}$ points to a node where its minimum value is larger or equal to all its key values, (ii) the key values of each node are sorted (iii) every node points to another node and (iv) every node is pointed only once (except from the head node) and that each node has a unique identifier. For every node in $\mathbf{T}^{idx}$, conditions (i)-(iii) are validated by checking if $k_{min} - \mathbf{k}[i] \geq 0$, $\mathbf{k}[i] - \mathbf{k}[i-1] \geq 0$ and $ptr \neq id$ respectively for all $i \in [K]$.

As for (iv), we first introduce the set $\mathcal{N}_{curr}$, which, for each node in $\mathbf{T}^{idx}$, includes a pair of its identifier and minimum key value. Likewise, we define $\mathcal{N}_{next}$, but for every node it contains a pair of the node identifier it points to and its minimum key value. More formally, we set $\mathcal{N}_{curr} = \{(\mathbf{n}.id, \mathbf{n}.p[0])\}_{\mathbf{n} \in \mathbf{T}^{idx}}$ and $\mathcal{N}_{next} = \{(\mathbf{n}.ptr, \mathbf{n}.k_{min})\}_{\mathbf{n} \in \mathbf{T}^{idx}}$. Given these sets, we can enforce condition (iv) by checking if $\mathcal{N}_{next} \cup (0, 0) = \mathcal{N}_{curr} \cup (+\infty, +\infty)$, where the pair $(0, 0)$ corresponds to the identifier and minimum value of the dummy node. Finally, to ensure the uniqueness of the identifiers, we check if the set of indexes is equal to $[|\mathbf{T}^{idx}|] \cup 0$. Note that (i)-(iii) consist of range checks, which are proved efficiently via lookup

arguments [56, 62], while (iv) translates to a circuit representing a multiplication tree and is proved via specialized sumchecks [64].

We note that after validating the correctness of all index proofs, the verifier can discard the Kopsis commitment of the table and replace it with a commitment of one of the indexes. This is done to efficiently update the table commitment after an update operation. Crucially, this does not limit the expressiveness of our scheme, since an index commitment is a PC scheme storing the entire table.

4.2 Proving Query Evaluations

After committing to all tables and indexes, the prover can now generate proofs for the correct computation of queries. In this section, we will focus on proving the correctness of elementary queries such as point and range queries (i.e., selections) or joins and also show how to utilize the committed indexes (if any) to significantly improve the prover's time. Subsequently, using these queries as basic building blocks, we can prove the correct execution of complex queries as shown in [75]. Next, we present point, range, and join queries. We defer details for other queries in Appendix D.2. *Proving the correctness of Point & Range Queries.* We first consider queries of the form “SELECT * FROM T WHERE $Attr = v$ ”. If T does not have an index on $Attr$, then the best we can do is to prove the correct execution of a circuit that is given the table, checks every individual tuple, and filters out all unnecessary ones. In our scheme, we follow the techniques of [75] but instead of using the GKR protocol [67], we design the circuit so that we can prove it with specialized sumchecks, which further reduces prover time.

Now, we will show how to use the index commitments along with our matrix lookup argument to make the prover time only proportional to the number of tuples that satisfy the predicate. In more detail, whenever an index for $Attr$ exists, the prover first retrieves all index rows from T^{idx} containing the desirable tuples, stores them in the matrix F , and sends its Kopsis commitment to the verifier. Next, the prover (i) shows that all rows of F belong to the committed index matrix T^{idx} , (ii) proves that F contains all tuples satisfying the predicate, and (iii) filters out all unnecessary tuples from F . To prove (i), it uses our matrix lookup argument. For (ii), the prover first shows that F represents a linked list. By doing so, the verifier also knows that F is sorted since T^{idx} represents a sorted linked list and $F \subseteq T^{idx}$. What remains is to show that F includes all tuples that satisfy the predicate. This is done by showing that there exist two tuples with attribute values strictly greater and smaller than v . We can translate these checks into an arithmetic circuit by using the techniques for proving index correctness. Finally, to filter out all unnecessary tuples, the prover uses the arithmetic circuit for selection queries, but this time over the table residing in F .

To prove general range queries of the form “SELECT * FROM T WHERE $v_1 \leq Attr \leq v_2$ ” we apply similar techniques with selection queries but with the following differences. First, for comparisons and range checks we use lookup arguments [62] instead of bit-decomposition based methods of [46, 75]. Furthermore, to check that F contains all desirable tuples, the prover shows that there exist two tuples with $Attr$ value greater and smaller than v_2 and v_1 . *Proving the Correctness of equi-Joins.* Here, we show how to prove the correctness of join queries, i.e., “SELECT * FROM T_1, T_2 WHERE $T_1.Attr_1 = T_2.Attr_2$ ”. We consider (1) primary and (2) non-primary

foreign key joins. For (1), we use techniques similar to [37, 44]. In particular, we first partition each table T_i , for $i = \{1, 2\}$ into T_i^1 and T_i^2 such that T_i^1 contains all tuples that satisfy the join predicate and vice versa. Furthermore, we parse the joined table T_3 as a set of pairs $(t_{i,1}, t_{i,2}) \in T_1^1 \times T_2^1$ and set $T_3^1 = \{t_{i,1}\}_{i \in [|T_3|]}$ and $T_3^2 = \{t_{i,2}\}_{i \in [|T_3|]}$. We then prove the correctness of T_3 as follows. First, we show that each tuple in T_3 satisfies the join predicate, i.e., $t_{i,1}.Attr_1 = t_{i,2}.Attr_2$ for all $i \in [|T_3|]$. Next, we ensure that all tuples of T_3 derive from T_1^1, T_2^1 . For this, we show that $T_3^1 = T_1^1$, for $i = \{1, 2\}$. Finally, we validate that all tuples from T_1, T_2 satisfying the join predicate are included in T_3 . This is done by showing that $T_{1|Attr_1}^2 \cap T_{2|Attr_2}^2 = \emptyset$ where $T_{i|Attr_i}^2$ is the set of all values of $Attr_i$ from T_i^2 , $T_i^1 \cup T_i^2 = T_i$, for $i \in \{1, 2\}$ and that $T_{1|Attr_1}^1 \cap T_{2|Attr_2}^2 = \emptyset$.

Next, we consider non-primary-foreign-key joins. Prior works [37, 75] achieved that by proving the correct execution of an arithmetic circuit that first performs pairwise comparisons of all tuples in T_1^1 and T_2^1 and then filters all concatenated tuples that satisfy the join predicate. Unfortunately, the size of the circuit and, therefore, the time to prove it is proportional to the size of T_3 times the number of unique predicate values in T_1^1 . In the worst case, this can be quadratic to the tables' size, leading to impractical prover times. Instead, we achieve a prover time that is linear to the size of T_3 !

Let $\mathbf{v} \in \mathbb{F}^N$ be a vector containing all the unique values of $T_{1|Attr_1}^1, T_{2|Attr_2}^1$, and $\mathcal{H} : \mathbb{F}^* \rightarrow [\mathbb{N}]^N$ be the function that takes as input $T_{i|Attr_i}^1$, and computes $\mathbf{h}_i$, such that $\mathbf{h}_i[j]$ is the frequency of $\mathbf{v}[j]$ in $T_{i|Attr_i}^1$. From the definition of the join operation, it must hold that the number of times each element $\mathbf{v}[j]$ appears in the joined table T_3 must be $\mathbf{h}_1[j] \cdot \mathbf{h}_2[j]$. Based on this, the prover performs the same checks with the primary-foreign key join, but also shows that $\mathbf{h}_i = \mathcal{H}(T_{i|Attr_i}^1)$ and that $\mathbf{h}_1 \circ \mathbf{h}_2 = \mathcal{H}(T_{3|Attr_1})$ —namely that T_3 contains the correct number of tuples with the same unique predicate value. Unfortunately, that is not enough because a cheating prover could replace tuples of T_3 with the same predicate value with a single one that still satisfies all the above checks. To overcome this issue, the prover further shows that all tuples in T_3 are unique using known techniques [75]. See Appendix D.3 for details on compiling the above into an arithmetic circuit and proving it via specialized sumchecks [57, 64].

4.3 Proving Database Updates

We finally show how to prove the correctness of updates on a table T . We focus on proving the correct computation of the three basic update operations, namely, insertions and deletions of tuples and updates of existing ones. When a table does not have indexes, we treat updates as [75] did, namely, we prove the correct computation of a circuit that takes as input the original and updated table and validates the correctness of the latter. Consequently, for the rest of this section, we focus on efficiently proving updates in the presence of indexes, and specifically, we will present insertions and deletions. In Appendix D.2, we also show how to extend our techniques in order to prove the correctness of expressive update queries.

Proving Insertions. To prove the correctness of a query of the form “INSERT INTO T VALUES $\mathcal{D} = (t_1, \dots, t_k)$ ”, the prover updates all indexes $\{\mathcal{T}_i\}_{i \in \mathcal{I}}$ and for each of them proves the correct insertion of $\mathcal{D}$. Next, we show how to achieve this for a single index $\mathcal{T}$.

Index Insertions. Let $\mathbf{T}^{idx}$ be the set of leaf nodes and $\mathbf{T}^{idx'}$ the set of leaf nodes after adding the tuples in $\mathcal{D}$. Furthermore, let $C_{\mathbf{T}^{idx}}, C_{\mathbf{T}^{idx'}}$ and C_D be their corresponding commitments. We want to generate a proof for the argument: “ $\mathbf{T}^{idx'}$ is the result of inserting $\mathcal{D}$ to $\mathbf{T}^{idx}$ and that $\mathbf{T}^{idx'}, \mathbf{T}^{idx}, \mathcal{D}$ are the pre-images of $C_{\mathbf{T}^{idx}}, C_{\mathbf{T}^{idx'}}$ and C_D ”. Essentially, we want to prove that $\mathbf{T}^{idx'}$ is a sorted linked list and that it only contains the tuples in D and $\mathbf{T}^{idx}$. The straightforward way of generating such proof would be to prove the well-formedness of $\mathbf{T}^{idx'}$ as we did in Section 4.1. Unfortunately, this would lead to a prover time proportional to the size of the new index. Instead, we construct a protocol that results in a prover time proportional to the number of inserted tuples!

Our key observation is that when inserting $\mathcal{D}$, we have to change at most $|\mathcal{D}|$ nodes from $\mathbf{T}^{idx}$. Based on this, the prover has to ensure that all “affected” nodes have been correctly updated, while the rest of the nodes remained unchanged. Looking ahead, the last check is going to be guaranteed by the update algorithm of our matrix lookup argument. Hence, we will focus on ensuring the first one. For this, let $\mathcal{N}$ be the set of all nodes in $\mathbf{T}^{idx}$ that will change after inserting $\mathcal{D}$ (e.g., every node in $\mathcal{N}$ will either have its tuples updated or split into two or more nodes). Furthermore, let $\mathcal{N}'$ be the set containing all changed nodes and $\mathcal{U} = \mathbf{T}^{idx} \cap \mathcal{N} = \mathbf{T}^{idx'} \cap \mathcal{N}'$ the set of all unchanged nodes. Now, to ensure that $\mathcal{N}'$ is a set of correct updates, we first retrieve from $\mathcal{U}$ the sets of nodes $\mathcal{L}$ and $\mathcal{R}$ consisting of the left and right siblings of nodes in $\mathcal{N}$, that do not belong to $\mathcal{N}$. While this step seems irrelevant at first glance, we observe that because $\mathbf{T}^{idx}$ is well-formed, the set $\mathcal{L} \cup \mathcal{N} \cup \mathcal{R}$ encompasses $|\mathcal{L}|$ lists with heads in $\mathcal{L}$ and tails in $\mathcal{R}$. This observation enables us to efficiently validate the insertion correctness by performing “localized” checks in the following way. Firstly, we check whether $\mathcal{L} \cup \mathcal{N}' \cup \mathcal{R}$ is still a set of sorted lists with heads and tails strictly lying in $\mathcal{L}$ and $\mathcal{R}$ respectively. In this way, we ensure that all initial lists remain sorted after adding $\mathcal{D}$ and, consequently, guarantee that $\mathbf{T}^{idx'}$ is a sorted linked list. Second, we check if all tuples in $\mathcal{N}'$ are equal to all tuples in $\mathcal{N}$ and $\mathcal{D}$. Since $\mathcal{U}$ is common to $\mathbf{T}^{idx}$ and $\mathbf{T}^{idx'}$ it guarantees the consistency of $\mathbf{T}^{idx'}$ ’s tuples.

Based on the above, we have everything we need to construct our circuit to validate the correctness of $\mathcal{N}'$. Initially, the circuit takes as input the nodes in $\mathcal{L}, \mathcal{R}, \mathcal{N}, \mathcal{N}'$ and $\mathcal{D}$ and shows that (i) $\mathcal{L} \cup \mathcal{R} \cup \mathcal{N}$ and $\mathcal{L} \cup \mathcal{R} \cup \mathcal{N}'$ contain sorted lists with heads in $\mathcal{L}$ and tails in $\mathcal{R}$ and (ii) $\mathcal{D} \cup \mathcal{N}_{|keys} = \mathcal{N}'_{|keys}$. To translate (i) into an arithmetic circuit we follow the same techniques used for proving the well-formedness of the tree. For (ii), we utilize the techniques of [75]. Finally, to prove the correctness of $\mathcal{T}'$, we invoke the update algorithm of our lookup using as $C_{\mathcal{U}}$ the described circuit.

Deletions are handled identically but in “reversed order”.

5 Experimental Evaluation

We implemented and experimentally evaluated ROGUE and ROGUEDB and in this section we show our results. We implemented our schemes in C++ and roughly 8000 lines of code. For field and elliptic-curve operations we used the MCL library [12]. In all our experiments, we utilized the BLS12_381 elliptic curve that supports approximately 120-bits of security. We run our benchmarks using a single thread on a dedicated machine running Linux Ubuntu 20.04.6

LTS with 131GB of RAM and Intel(R) Xeon(R) E-2174G CPU, with 8 cores at 3.80GHz. In all our experiments, we utilized a single thread.

5.1 Benchmarking ROGUE

In this section, we benchmark ROGUE by measuring the prover, verification time and proof size when generating a lookup and update proof in the following scenarios: a matrix with $n = 2^{20}$ rows and (1) variable columns ranging between $m = 1-2^8$ and constant row lookups/updates and (2) constant number of columns, and increasing row lookups/updates, ranging between $k = 1-2^{12}$. We also compare ROGUE with the (non-updatable) matrix lookup argument **SublonK** [26] and the updatable lookup argument proposed in [28] (which we denote with CQ-ULA) using their open-sourced implementations [1, 2]. For CQ-ULA, we measure average lookup and update costs with the optimal parameter configuration proposed in [28] which achieve an amortized lookup and update complexity of $\tilde{O}(m\sqrt{nk})$ where n, m are the rows and columns of the underlying matrix, and k is the number of row lookups/updates. We also compare ROGUE with the “folklore” approach that commits the matrix using a Merkle tree and “opens” the desired rows inside the SNARK. We implement this, called MT+SNARK, using the SNARK-friendly Poseidon hash [35] and a GKR-based SNARK with the implementation of [73], but replace its PCS with the faster Kopis. Finally, we run ROGUE with the optimization described in 3 which decomposes the table into L equal-sized sub-tables and commits them with Kopis. We set $L = 1024$, which significantly reduces space requirements, allowing ROGUE to scale to large matrix sizes. For instance, for $n = 2^{20}$ and $m = 64$, the space requirements are reduced from 110GB to 5.74GB. Unfortunately, it is unclear how to apply this optimization for CQ-ULA or **SublonK**.

Benchmarking the Lookup Algorithm. The first two plots in Figure 1 (top) show the time required to lookup 16 and 1024 rows, from a table with constant number of rows and increasing number of columns. In both cases ROGUE’s prover time remains unaffected by the number of columns, which aligns with our asymptotic analysis. In contrast, the prover times for **SublonK**, CQ-ULA, and MT+SNARK increase proportionally with the number of columns. Eventually, ROGUE becomes $\times 21.43$, $\times 450-3257$ and $\times 18.95-68.9$ faster than **SublonK**, CQ-ULA, and MT+SNARK. Concretely, to lookup 1024 rows of size 256, ROGUE needs 9.93sec, while **SublonK**, CQ-ULA, and MT+SNARK take 427, 32340, and 684sec.

The third, fourth and fifth plots of Figure 1 (top) show the time needed to lookup $k = 1-2^{14}$ rows, across three different matrices with 4, 32 and 128 columns respectively. First, observe that our prover time increases the same in all three matrices, ranging from 0.67-147sec for $k = 1-2^{14}$. As k increases, ROGUE becomes significantly faster than all other schemes. In particular, it is $\times 2.13$ - $\times 163$ and $\times 2.4$ - $\times 45$ faster than CQ-ULA and MT+SNARK when $k \geq 16$ and $\times 1.1$ - $\times 22.4$ faster than **SublonK** when $k \geq 64$. At the same time, for very small k , our prover time stays almost constant. This derives from the complexity of the prover when applying the optimization of Section 3, which, for small k is dominated by $O(L)$.

Benchmarking the Update Algorithm. The first and second plots of Figure 1 (bottom) report the time required to update 16 and 1024 rows, respectively, from a table with increasing number of columns. Unlike lookup, ROGUE’s update time increases linearly

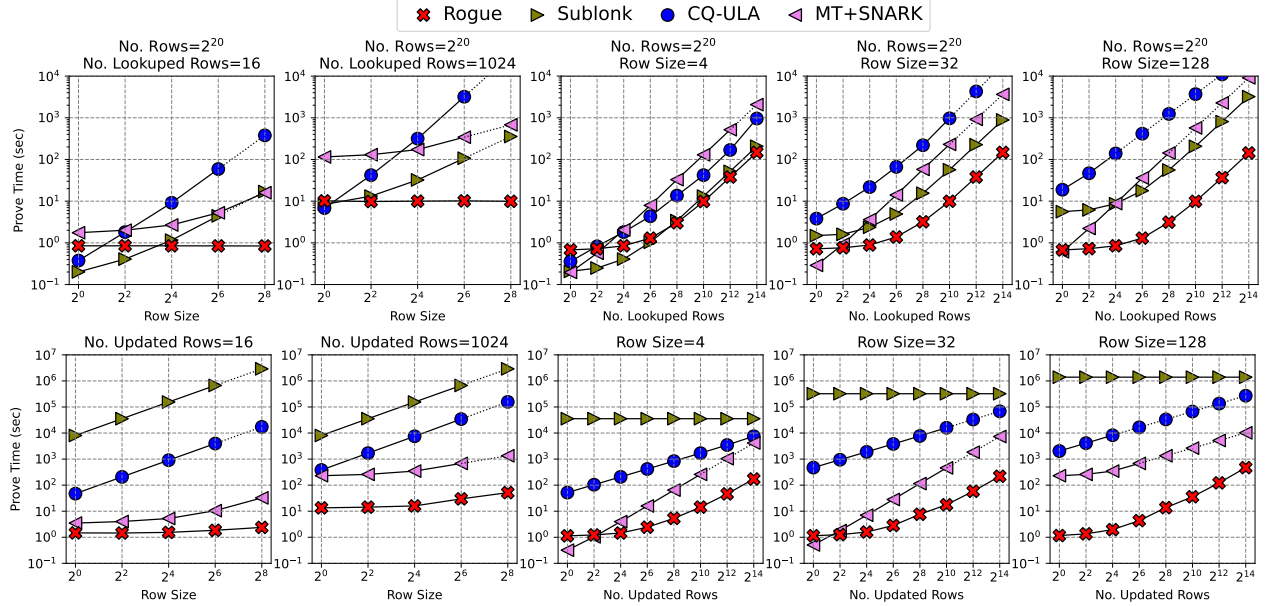


Figure 1: Lookup (top) and Update (bottom) prove time for a table with variable row size, a constant number of rows $n = 2^{20}$ and constant batch sizes $k = 16$ (Left 1) and $k = 1024$ (Left 2). Update prove time for a table with variable batch size, a constant number of rows $n = 2^{20}$ and constant row sizes $m = 4$ (Right 3), $m = 32$ (Right 2) and $m = 128$ (Right 1).

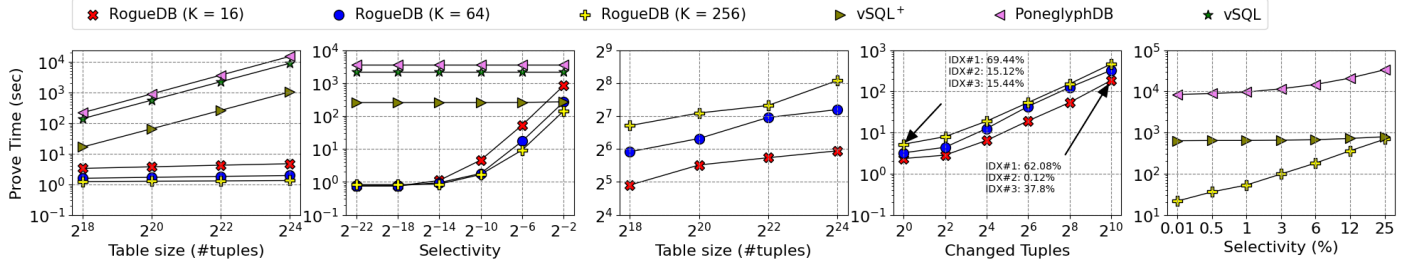


Figure 2: ROGUEDB benchmarks. Left to right: proving time for a selection query with (1) variable table size but fixed number of selected tuples (2) variable selectivity but fixed table size, and expressive update query with (3) variable table size and fixed number of changes, (4) variable number of changes but fixed table size, and (5) TPC-H vs. variable selectivity.

with the number of columns. Nonetheless, our scheme outperforms CQ-ULA, Sublonk and MT+SNARK by $\times 28$ - $\times 7206$, $\times 595$ - $\times 8 \cdot 10^5$ and $\times 2.38$ - $\times 27.7$ respectively. For instance, for $m = 256$ columns and $k = 1024$, the update time of ROGUE is and 51.6sec compared to the 43.6, 573.7 and 0.4 hours of CQ-ULA, Sublonk and MT+SNARK! The rest of the plots present the time needed to update $k = 1$ - 2^{14} rows on three different matrixes with 4, 32 and 128 columns. All schemes scale proportionally to the k except from Sublonk, which is not updatable and hence its update time equals to the time needed to pre-process the table from scratch. Overall, ROGUE is faster than all other schemes except when comparing with MT+SNARK at very small m and k . Specifically, ROGUE is $\times 185$ - $\times 10^6$ faster than Sublonk and $\times 46$ - $\times 3030$ faster than CQ-ULA. When $k \geq 256$, ROGUE is $\times 18$ - $\times 40$ faster than MT+SNARK.

Proof size & Verification Time. ROGUE's lookup and update proof size ranges from 44.74-183.33KB and 50.83-209.43KB. This is $\times 3.2$ - $\times 5$ smaller compared to MT+SNARK but significantly larger than those of CQ-ULA and Sublonk, which achieve a constant proof

size of 4.4KB and 2.4KB, respectively. A similar trend appears in verification time which ranges from 22-92ms for lookups and 35-106ms for updates.

Comparison with Prior Works when Proving RAM Computations. In this section, we also compare the performance of ROGUE with CQ-ULA and MT+SNARK when proving the correctness of a program that performs k reads and writes from a large RAM. This requires the prover to generate a proof showing that (1) the RAM has been correctly updated after performing k reads and writes and (2) the underlying program has been correctly computed. Here, we will particularly focus on benchmarking (1), which in practice corresponds to the update algorithm of our lookup argument. With that in mind, for a RAM of size 2^{20} and a number of reads/writes ranging from $k = 16$ - 2^{12} , it takes 1.46-42.09sec to generate a proof for (1) using ROGUE. In contrast, MT+SNARK takes 3.3-866sec while CQ-ULA has an average cost of 47.1-771.1sec which is $\times 2.2$ - $\times 32$ slower than ROGUE. Most importantly, this speedup increases even further when the program reads and writes to consecutive elements

of the RAM, which is the case for zkRollups (see discussion in Section 3). For instance, when reading k batches of 32 consecutive elements where k ranges between $16 \cdot 2^{12}$, the prover time of ROGUE will only slightly increase to 1.59-57.9sec vs the 1894-31899sec and 7.03-1837sec of CQ-ULA and MT+SNARK respectively.

5.2 Benchmarking ROGUEDB

Here, we benchmark ROGUEDB when proving the correct computation of range, update, join, and more complex SQL queries. Unless specified otherwise, we carried out all our experiments using the database provided by the widely used (in the database community) TPC-H [3] benchmark (see Appendix E for more details on the queries and the dimensions of each table). Furthermore, we compare our construction with all existing *succinct* verifiable database schemes, namely, PoneglyphDB [37] using its publicly available code provided in [4] and vSQL [75]. Because vSQL is not open-sourced, we implement it from scratch using the implementation of an optimal-time GKR provided in [74] but replacing the KZG PCS with the faster Kopis PCS, thus improving the performance of the original vSQL by $\times 3$. Finally, we also test the performance of our scheme without indexes (vSQL⁺).

Benchmarking Range Queries. We begin by evaluating the performance of our scheme when proving the correct computation of the query that selects all tuples from the LINEITEM table with shipment date attribute (L_SHIPDATE) ranging from d_1 and d_2 (see Appendix E for a detailed description of all queries we prove throughout this section). We test ROGUEDB on: (a) varying table size, ranging from $n = 2^{18}$ - 2^{24} tuples and a constant number of selected tuples $n' = 2^{12}$, and (b) varying selectivity ratio (i.e., the fraction of selected tuples with the table size ($\frac{n'}{n}$)) ranging from 2^{-22} - 2^{-2} and constant table size of $n = 2^{22}$ tuples. Recall that for the index (on the L_SHIPDATE), we need to specify the value K , i.e., the number of tuples each index node stores. Here we experiment with three different values $K = 16, 64$ and 256 and compare ROGUEDB against vSQL, vSQL⁺, and PoneglyphDB.

The first plot of Figure 2 reports the prover time of all schemes on variable table sizes and a constant number of selected tuples, while the second one presents the prover time for the same query but for constant table size and varying selectivity ratio. Overall, ROGUEDB's prover time remains almost unaffected by the table size and depends only on the selectivity ratio, which aligns with its asymptotic complexity. Furthermore, observe that the opposite is true for PoneglyphDB, vSQL, and vSQL⁺ since the prover complexity of these schemes is proportional not only to selectivity, but also to table size, which becomes the main bottleneck of the prover time. Concretely, ROGUEDB is $\times 13.6 \cdot \times 22 \cdot 10^3$ and $\times 8.3 \cdot \times 13 \cdot 10^3$ faster than PoneglyphDB and vSQL. It is also $\times 1.02 \cdot \times 1593$ faster than vSQL⁺ except for the case $K = 16$ and with selectivity ratio $> 8\%$. Proof size & Verification Time. Overall, the proof size of both vSQL⁺ and ROGUEDB ranges from 40-120KB, and is almost identical to vSQL as both schemes rely on GKR. In contrast, the proof size of the PoneglyphDB remains constant at approximately 4.2KB. Lastly, verification times are extremely low, e.g., < 60 ms in all experiments.

Benchmarking Update Queries. Here, we test the performance of ROGUEDB when proving the correct computation of an update query that increases by 0.01 the tax attribute (L_TAX) of all tuples from

the LINEITEM table with shipment date (L_SHIPDATE) between d_1 and d_2 . Furthermore, we assume that the prover maintains indexes on the attributes L_TAX, L_SHIPDATE, and L_ORDERKEY (denoted with IDX#1, IDX#2, and IDX#3, respectively). As we will show later, we select these indexes to capture all possible update operations and edge cases. With that in mind, we test ROGUEDB's performance when proving the above query for: (1) varying table size, ranging from $n = 2^{18}$ - 2^{24} tuples and a constant number of changed tuples $\delta = 256$ and (2) varying number of changed tuples ranging from $\delta = 1$ - 2^{10} and a constant table size of $n = 2^{22}$ tuples.

The third plot of Figure 2 shows update prover time vs. increasing table size and constant number of changed tuples, while the fourth shows prove time for constant table size vs. increasing number of changes. Observe that increasing the table size from 2^{18} to 2^{24} tuples only increases the total prover time by $\times 2.3$. At the same time, when increasing δ , the prover time grows from 2.3-182.6sec for $K = 16$, 3.2-322.9sec for $K = 64$ and 5.1-457.9sec for $K = 256$. Furthermore, note that more than 60% of the prover's time is spent on generating a proof for the correct update of IDX#1. This is expected because it is an index on L_TAX—the attribute being changed by the query. Hence, proving its correct update requires generating two separate proofs: one for the deletion and one for the insertion of $O(\delta)$ tuples. As for IDX#2 and IDX#3, while we use the optimization described in Appendix D.2 to prove their correct update, observe that updating IDX#2 requires significantly less time compared to IDX#3. That is because IDX#2 is an index on L_SHIPDATE — the predicate value of the update query — meaning that all tuples to be changed lie sorted on $O(\delta/K)$ nodes. Thus, we only need to update $O(\delta/K)$ nodes compared to IDX#3 where we need to update $O(\delta)$ nodes.

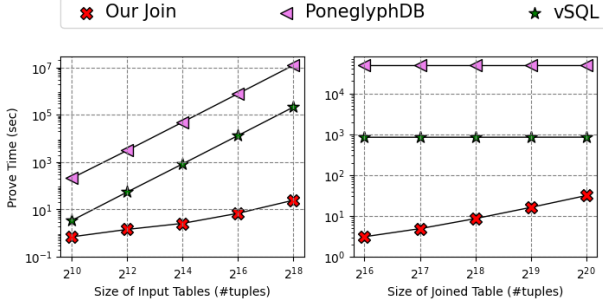
Proof size & Verification Time. In summary, the total proof size varies between 112.2-320.1KB where roughly the 55% is attributed to the proof of the update for IDX#1. A similar trend appears in the verification time, spanning from 12-65ms.

Benchmarking Join Queries. Next, we proceed to test the performance of ROGUEDB on join queries for non-primary-foreign key joins in Section 5.3 we also test primary-foreign key joins. In more detail, we measure the time needed to prove a join when: (1) the two input tables of size n tuples produce a joined table of $2n$ tuples for n ranging between 2^{10} - 2^{18} and (2) the number of tuples of each input table is fixed to $n = 2^{14}$ and the tuples of the joined table ranges from $n' = 2^{16}$ - 2^{20} . We use a synthetic database generated in the same way as described in [42], and set 8 attributes in both tables. Overall, our scheme outperforms vSQL by $\times 4.88 \cdot \times 8624$ and PoneglyphDB by $\times 301.1 \cdot \times 50 \cdot 10^4$. Concretely, to prove the join with input tables of size 2^{18} that produces a 2^{19} -sized table is expected to take 2.45 days on vSQL, 4.8 months on PoneglyphDB, but only 34.4sec on ROGUEDB!

Benchmarks on TPC-H Queries. In this benchmark, we tested our scheme on five different TPC-H Queries: Query #3, #5, #6, #15 and #19, which capture all basic SQL operations such as selections, joins, group and order by, and aggregations. We measured the prover time, proof size, and verification time when proving the above queries on a TPC-H database generated using a scaling factor of 1.32 corresponding to a database of size 1.1GB and a selectivity ratio of $\sigma = 1\%$. We also measured the prover time for query #3 with

Table 2: Prover time (left), proof size (middle), and verification time (right) of ROGUEDB, vSQL⁺ and PoneglyphDB when proving five TPC-H queries with selectivity ratio $\sigma = 1\%$ for a database generated using a scaling factor of 1.32.

	Prover Time (sec)			Proof size (KB)			Verification time (ms)		
	ROGUEDB	vSQL ⁺	PoneglyphDB	ROGUEDB	vSQL ⁺	PoneglyphDB	ROGUEDB	vSQL ⁺	PoneglyphDB
TPCH-3	52.7	638	9554	813.2	867.84	4.5	167	114	10
TPCH-5	259.5	366.2	38826.8	1126.37	1206.18	4.5	150	129	11
TPCH-6	50.4	541.9	6952.0	261.71	303.65	4.5	93	56	12
TPCH-15	47.2	528.6	6909	236.5	263.71	4.5	92	60	10
TPCH-19	306	1714.8	25777	1498.75	1523.2	4.5	300	276	10

**Figure 3: Time required to prove a non-primary-foreign key join for increasing input table sizes ranging between $n = 2^{10}$ - 2^{18} and a joined table of size $2n$ tuples (Left) and input tables with a constant size of $n = 2^{22}$ tuples and variable size of joined table (Right).**

variable selectivity ranging from 0.1%-25%. Lastly, for ROGUEDB we set $K = 256$ and compare it with vSQL⁺ and PoneglyphDB.

Table 2 shows the performance of all schemes when proving the five selected queries. In total, ROGUEDB outperforms vSQL⁺ and PoneglyphDB by $\times 1.4$ - $\times 12.1$ and $\times 84.2$ - $\times 181.3$ respectively. Note that while ROGUEDB consistently outperforms the other schemes, it has comparable performance with vSQL⁺ on queries #5 and #19. For query #5, this is because it requires a join on the entire LINEITEM table, which contains 8million tuples and naturally dominates the prover time. As for query #19, the minimum possible selectivity that it can be set to is 6.25%, which limits the ability to fully utilize the underlying indexes as effectively as possible. Finally, the 5th plot of Figure 2 presents the time it takes for ROGUEDB, vSQL⁺ and PoneglyphDB to prove query #3 for variable selectivity ratio. Overall, ROGUEDB is $\times 1.14$ - $\times 29.2$ and $\times 48.5$ - $\times 388.4$ faster than vSQL⁺ and PoneglyphDB.

5.3 Additional Experiments

Primary-Foreign Key Joins. We also measured the time needed to generate a proof for a primary-foreign key join query on increasing input table size ranging from $n = 2^{10}$ - 2^{18} tuples and joined table size of $2n$ tuples. Compared to the non-primary-foreign key join our prover time only slightly decreased to 0.7sec for $n = 2^{10}$ and to 24.6sec for $n = 2^{22}$. In contrast, the prover time PoneglyphDB significantly reduced, now ranging from 20.8 to 6882.2sec for $n = 2^{10}$ - 2^{18} . Despite this improvement, our prover remains $\times 29.7$ - $\times 279.7$

faster than PoneglyphDB. Finally, since the join we test is not one-to-one, vSQL will have the same prover time as in non-primary-foreign key join.

ROGUEDB vs vSQL⁺ with Hash-Based Indexes. While there is no verifiable database system that supports the proof generation of arbitrary SQL queries and indexes at the same time, one could directly construct such a system, by committing the indexes using a Merkle tree and later opening the desired tuples within the SNARK. In practice, we distinguish two main approaches for instantiating such a scheme. The first approach is to commit to indexes the same way we do in ROGUEDB but replacing ROGUE as an underlying matrix lookup argument with a hash-based one. The second approach would be to Merkle commit an index while keeping all tuples sorted by the relevant attribute. Overall, we expect this approach to obtain a significantly faster prover time when proving the correctness of a range query, as it directly filters out all unnecessary tuples and consequently the prover has to compute a smaller number of hashes within the SNARK. On the other hand, it is unclear how efficiently one can update that committed index since one insertion might result in recomputing the entire Merkle tree within the SNARK.

When proving the correct computation of the range query described in section 5.2, for selectivity ratio ranging between 2^{-22} - 2^{-2} and table size of 2^{22} tuples, the first approach takes 0.32-66323.9sec for $K = 16$, 0.43-59156.4sec for $K = 64$ and 0.45-58321sec for $K = 256$. In contrast, the second approach takes 0.11-4922.8sec. Although ROGUEDB has comparable but faster prover times for extremely small selectivity ratios (e.g., $< 2^{-18}$), it eventually becomes $\times 79.9$ - $\times 423.4$ faster than the first approach and $\times 6$ - $\times 35$ faster than the second approach.

Regarding updates, when proving the correctness of the expressive update query discussed in section 5.2, the first approach eventually becomes $\times 3.75$, $\times 7.7$ and $\times 21.7$ slower than ROGUEDB for $K = 16$, 64 and 256 respectively.

Performance of ROGUEDB vs. vSQL⁺ on Updates. Here, we compare the performance of ROGUEDB and vSQL⁺ when proving the correct computation of the update query of section 5.2. Since vSQL⁺ does not use any indexes, the prover only has to show that the table has been correctly updated following the same techniques proposed by [75]. For a table with 2^{22} tuples and an increasing number of changes between $\delta = 1$ - 2^{10} , the prover time of vSQL⁺ remains almost constant, ranging between 366.76-369.1sec. This stems from the fact that the prover complexity is proportional to both the table size and δ . Likewise, when keeping the number of changed tuples constant and varying the table size from 2^{18} to 2^{24} , the prover time ranges from 27.83 to 1467 seconds. This is $\times 1.1$ - $\times 157.6$ slower than

ROGUEDB with the only exception when $K = 256$ and $\delta \geq 2^9$, where $vSQL^+$ only marginally outperforms ROGUEDB by $\times 1.24$.

Acknowledgements

We thank the anonymous reviewers for their feedback. This work was supported in part by the Hong Kong Research Grants Council under grant GRF-16213625.

Ethics considerations

In this work, we construct the first updatable matrix lookup argument. Our work significantly improves the time required to prove computations that read and write only a small number of rows from a committed matrix, which can be beneficial for securing various systems. We did not use personal data for our experiments, and all our experiments are conducted on dummy inputs or public datasets. Furthermore, all cryptographic tools used comply with current laws and regulations.

Open Science

We fully comply with the ACM CCS 2026 Open Science policy by open-sourcing our entire codebase and providing the benchmark scripts in the repository <https://github.com/ChristodoulosPappas/Updatable-Matrix-Lookup-Arguments-and-Applications-to-Verifiable-Databases>.

References

- [1] [n. d.]. ([n. d.]). ark-segmentlookup, <https://github.com/txaty/ark-segmentlookup/>.
- [2] [n. d.]. ([n. d.]). Batching-Efficient RAM using Updatable Lookup Arguments, <https://github.com/nitsatisc/caulk/tree/updateable-ram>.
- [3] [n. d.]. ([n. d.]). TPC-H Benchmark, <https://www.tpc.org/tpch/>.
- [4] [n. d.]. ([n. d.]). <https://github.com/tuzijun111/halo2-TPCH>.
- [5] [n. d.]. Axiom OpenVm. ([n. d.]). <https://www.axiom.xyz/>.
- [6] [n. d.]. barryWhiteHat. https://github.com/barryWhiteHat/roll_up/.
- [7] [n. d.]. Basel Committee on Banking Supervision Principles for effective risk data aggregation and risk reporting. ([n. d.]). <https://www.bis.org/publ/bcbs239.pdf>.
- [8] [n. d.]. Lagrange Labs. ([n. d.]). www.lagrange.dev/.
- [9] [n. d.]. Space & Time. ([n. d.]). <https://www.spaceandtime.io/>.
- [10] [n. d.]. Space And Time White Paper. ([n. d.]). <https://sxtdocpub.blob.core.windows.net/docs/sxt-whitepaper.pdf>.
- [11] [n. d.]. ZK Coprocessor. ([n. d.]). <https://www.lagrange.dev/blog/zk-coprocessor-euclid-testnet-phase-1-recap>.
- [12] 2015. <https://github.com/herumi/mcl>. (2015).
- [13] Kasra Abbaszadeh, Christodoulos Pappas, Jonathan Katz, and Dimitrios Papadopoulos. 2024. Zero-Knowledge Proofs of Training for Deep Neural Networks. In *ACM CCS*. 4316–4330.
- [14] Masayuki Abe, Georg Fuchsbaauer, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. 2010. Structure-Preserving Signatures and Commitments to Group Elements. In *CRYPTO*. 209–236.
- [15] Maha Ayub, Tania Saleem, Muhammad Umar Janjua, and Talha Ahmad. 2023. Storage State Analysis and Extraction of Ethereum Blockchain Smart Contracts. *ACM Trans. Softw. Eng. Methodol.* (2023), 57:1–57:32.
- [16] David Balbás, Dario Fiore, María Isabel González Vasco, Damien Robissout, and Claudio Soriente. 2023. Modular Sumcheck Proofs with Applications to Machine Learning and Image Processing. In *ACM CCS*. 1437–1451.
- [17] Josh Benaloh and Michael De Mare. 1993. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application of Cryptographic Techniques*. Springer, 274–285.
- [18] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. 2021. Time- and Space-Efficient Arguments from Groups of Unknown Order. In *CRYPTO (4) (Lecture Notes in Computer Science, Vol. 12828)*. Springer, 123–152.
- [19] Vincenzo Botta, Simone Bottoni, Matteo Campanelli, Emanuele Ragnoli, and Alberto Trombetta. 2025. *qedb*: Expressive and Modular Verifiable Databases (without SNARKs). *Cryptology ePrint Archive* (2025).
- [20] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. 2021. Proofs for Inner Pairing Products and Applications. In *ASIACRYPT*. 65–97.
- [21] Matteo Campanelli, Antonio Faonio, Dario Fiore, Tianyu Li, and Helger Lipmaa. 2024. Lookup Arguments: Improvements, Extensions and Applications to Zero-Knowledge Decision Trees. In *PKC*. 337–369.
- [22] Matteo Campanelli, Dario Fiore, and Rosario Gennaro. 2025. Natively Compatible Super-Efficient Lookup Arguments and How to Apply Them. *J. Cryptol.* (2025), 14.
- [23] Matteo Campanelli, Dario Fiore, Semin Han, Jihye Kim, Dimitris Kolonelos, and Hyunok Oh. 2022. Succinct Zero-Knowledge Batch Proofs for Set Accumulators. In *ACM CCS*. 455–469.
- [24] Ran Canetti, Omer Paneth, Dimitrios Papadopoulos, and Nikos Triandopoulos. 2014. Verifiable Set Operations over Outsourced Databases. In *PKC*. 113–130.
- [25] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. 2023. HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates. In *EUROCRYPT*. 499–530.
- [26] Arka Rai Choudhuri, Sanjam Garg, Aarushi Goel, Sruthi Sekar, and Rohit Sinha. 2024. SublonK: Sublinear Prover PlonK. *PETs* (2024), 314–335.
- [27] Chris Clifton and Hector Garcia-Molina. 1990. Indexing in a Hypertext Database. In *Vldb*. Morgan Kaufmann, 36–49.
- [28] Moumita Dutta, Chaya Ganesh, Sikhar Patranabis, Shubh Prakash, and Nitin Singh. 2024. Batching-Efficient RAM using Updatable Lookup Arguments. In *CCS*. ACM, 4077–4091.
- [29] Liam Eagen, Dario Fiore, and Ariel Gabizon. 2022. cq: Cached quotients for fast lookups. *IACR Cryptol. ePrint Arch.* (2022), 1763.
- [30] Georg Fuchsbaauer, Eike Kiltz, and Julian Loss. 2018. The Algebraic Group Model and its Applications. In *CRYPTO 2018*. 33–62.
- [31] Ariel Gabizon and Dmitry Khovratovich. 2022. flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size. *IACR Cryptol. ePrint Arch.* (2022).
- [32] Ariel Gabizon and Zachary J. Williamson. 2020. plookup: A simplified polynomial protocol for lookup tables. *IACR Cryptol. ePrint Arch.* (2020), 315. <https://eprint.iacr.org/2020/315>
- [33] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. 2008. Delegating computation: interactive proofs for muggles. In *ACM STOC*, Cynthia Dwork (Ed.). 113–122.
- [34] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract). In *ACM STOC*. 291–304.
- [35] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schafneggger. 2021. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *USENIX Sec*. 519–535.
- [36] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *EUROCRYPT*. 305–326.
- [37] Binbin Gu, Juncheng Fang, and Faisal Nawab. 2025. PoneglyphDB: Efficient Non-interactive Zero-Knowledge Proofs for Arbitrary SQL-Query Verification. *Proc. ACM Manag. Data* (2025), 63:1–63:27.
- [38] Ulrich Haböck. 2022. Multivariate lookups based on logarithmic derivatives. *IACR Cryptol. ePrint Arch.* (2022), 1530.
- [39] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. 2010. Constant-Size Commitments to Polynomials and Their Applications. In *ASIACRYPT 2010*. 177–194.
- [40] Dmitry Khovratovich, Ron D Rothblum, and Lev Soukhanov. 2025. How to prove false statements: Practical attacks on fiat-shamir. In *Annual International Cryptology Conference*. Springer, 3–26.
- [41] Joe Kilian. 1992. A Note on Efficient Zero-Knowledge Proofs and Arguments (Extended Abstract). In *ACM STOC*. 723–732.
- [42] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient Oblivious Database Joins. *Vldb* (2020), 2132–2145.
- [43] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. 2006. Dynamic authenticated index structures for outsourced databases. In *ACM SIGMOD*. 121–132.
- [44] Xiling Li, Chenkai Weng, Yongxin Xu, Xiao Wang, and Jennie Rogers. 2023. ZKSQL: Verifiable and Efficient Query Evaluation with Zero-Knowledge Proofs. *Vldb* (2023), 1804–1816.
- [45] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. 2023. Pianist: Scalable zkRollups via Fully Distributed Zero-Knowledge Proofs. *IACR Cryptol. ePrint Arch.* (2023).
- [46] Tianyi Liu, Xiang Xie, and Yupeng Zhang. 2021. zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy. In *ACM CCS*. 2968–2985.
- [47] Tianyi Liu, Zhenfei Zhang, Yuncong Zhang, Wenqing Hu, and Ye Zhang. 2025. Ceno: Non-uniform, Segment and Parallel Zero-Knowledge Virtual Machine. *J. Cryptol.* 38, 2 (2025), 17.
- [48] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. 1992. Algebraic Methods for Interactive Proof Systems. *J. ACM* (1992), 859–868.
- [49] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, 369–378.

- [50] Yehudit Mond and Yoav Raz. 1985. Concurrency Control in B+-Trees Databases Using Preparatory Operations. In *VLDB*. Morgan Kaufmann, 331–334.
- [51] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. 2020. Scaling Verifiable Computation Using Efficient Set Accumulators. In *USENIX Sec*. 2075–2092.
- [52] Dimitrios Papadopoulos, Stavros Papadopoulos, and Nikos Triandopoulos. 2014. Taking Authenticated Range Queries to Arbitrary Dimensions. In *ACM CCS*. 819–830.
- [53] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. 2013. Signatures of Correct Computation. In *TCC*. 222–242.
- [54] Charalampos Papamanthou, Shrawan Srinivasan, Nicolas Gailly, Ismael Hishon-Rezaizadeh, Andrus Salumets, and Stjepan Golemac. 2024. Reckle Trees: Updatable Merkle Batch Proofs with Applications. In *ACM CCS*. 1538–1551.
- [55] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. 2011. Optimal Verification of Operations on Dynamic Sets. In *CRYPTO*. 91–110.
- [56] Shahar Papini and Ulrich Haböck. 2023. Improving logarithmic derivative lookups using GKR. *IACR Cryptol. ePrint Arch.* (2023), 1284.
- [57] Christodoulos Pappas and Dimitrios Papadopoulos. 2024. Sparrow: Space-Efficient zkSNARK for Data-Parallel Circuits and Applications to Zero-Knowledge Decision Trees. In *ACM CCS*. 3110–3124.
- [58] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2016. Pinocchio: nearly practical verifiable computation. *Commun. ACM* (2016), 103–112.
- [59] Jim Posen and Assimakis A. Kattis. 2022. Caulk+: Table-independent lookup arguments. *IACR Cryptol. ePrint Arch.* (2022), 957. <https://eprint.iacr.org/2022/957>
- [60] Scroll Architecture. 2023. Scroll Architecture. <https://docs.scroll.io/en/technology/>.
- [61] Srinath T. V. Setty and Jonathan Lee. 2020. Quarks: Quadruple-efficient transparent zkSNARKs. *IACR Cryptol. ePrint Arch.* (2020).
- [62] Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. 2024. Unlocking the Lookup Singularity with Lasso. In *EUROCRYPT* 2024. 180–209.
- [63] Shrawan Srinivasan, Alexander Chepurnoy, Charalampos Papamanthou, Alin Tomescu, and Yupeng Zhang. 2022. Hyperproofs: Aggregating and Maintaining Proofs in Vector Commitments. In *USENIX Sec*. 3001–3018.
- [64] Justin Thaler. 2013. Time-Optimal Interactive Proofs for Circuit Evaluation. In *CRYPTO*. 71–89.
- [65] Louis Tremblay Thibault, Tom Saray, and Abdelhakim Senhaji Hafid. 2022. Blockchain scaling using rollups: A comprehensive survey. *IEEE Access* (2022).
- [66] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. 2022. zkbridge: Trustless cross-chain bridges made practical. In *ACM CCS*. 3003–3017.
- [67] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. 2019. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. In *CRYPTO*. 733–764.
- [68] Jiajun Xin, Arman Haghighi, Xiangnan Tian, and Dimitrios Papadopoulos. 2024. Notus: Dynamic Proofs of Liabilities from Zero-knowledge RSA Accumulators. In *USENIX Sec*.
- [69] Yin Yang, Dimitris Papadias, Stavros Papadopoulos, and Panos Kalnis. 2009. Authenticated join processing in outsourced databases. In *ACM SIGMOD*. 5–18.
- [70] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. 2022. Caulk: Lookup arguments in sublinear time. In *CCS*. 3121–3134.
- [71] Arantxa Zapico, Ariel Gabizon, Dmitry Khovratovich, Mary Maller, and Carla Ràfols. 2022. Baloo: Nearly Optimal Lookup Arguments. *IACR Cryptol. ePrint Arch.* (2022), 1565. <https://eprint.iacr.org/2022/1565>
- [72] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. 2020. Zero Knowledge Proofs for Decision Tree Predictions and Accuracy. In *ACM CCS*. 2039–2053.
- [73] Jiaheng Zhang, Tianyi Liu, Weijie Wang, Yinyu Zhang, Dawn Song, Xiang Xie, and Yupeng Zhang. 2021. Doubly Efficient Interactive Proofs for General Arithmetic Circuits with Linear Prover Time. In *CCS*. ACM, 159–177.
- [74] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. 2020. Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof. In *IEEE SP*. 859–876.
- [75] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases. In *IEEE SP*. 863–880.
- [76] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. A Zero-Knowledge Version of vSQL. *IACR Cryptol. ePrint Arch.* (2017).
- [77] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2015. IntegriDB: Verifiable SQL for Outsourced Databases. In *ACM CCS*. 1480–1491.
- [78] Yuncong Zhang, Shifeng Sun, and Dawu Gu. 2024. Efficient KZG-Based Univariate Sum-Check and Lookup Argument. In *PKC* (2). Springer, 400–425.
- [79] Yuncong Zhang, Shifeng Sun, Ren Zhang, and Dawu Gu. 2023. Polynomial IOPs for Memory Consistency Checks in Zero-Knowledge Virtual Machines. In *ASIACRYPT* (2) (Lecture Notes in Computer Science, Vol. 14439). Springer, 111–141.
- [80] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. 1992. An Efficient Indexing Technique for Full Text Databases. In *VLDB*. Morgan Kaufmann, 352–362.

A Definitions

DEFINITION A.1 (ARGUMENTS OF KNOWLEDGE). For any NP relation $\mathcal{R}$, a tuple of PPT algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is an Argument of Knowledge if it satisfies the following:

- **Completeness.** For any security parameter λ , $(pk, vk) \leftarrow \mathcal{G}(1^\lambda)$ and $(x, w) \in \mathcal{R}$, $\Pr(\mathcal{V}(vk, x, t) = \text{accept}) = 1$, where $t = \langle \mathcal{P}(pk, w), \mathcal{V}(vk) \rangle(x)$ the communication transcript.
- **Knowledge Soundness.** For every PPT prover $\mathcal{P}^*$ there exists a PPT extractor $\mathcal{E}$ that have access to $\mathcal{P}^*$'s random tape, such that for any security parameter λ , $(pk, vk) \leftarrow \mathcal{G}(1^\lambda)$ and x it holds that:

$$\Pr \left(\begin{array}{l} t = \langle \mathcal{P}^*(pk), \mathcal{V}(vk) \rangle(x), \\ \mathcal{V}(vk, t, x) = \text{accept} \wedge \\ w \leftarrow \mathcal{E}(pk, x) \wedge \\ (x, w) \notin \mathcal{R} \end{array} \right) \leq \text{negl}(\lambda)$$

DEFINITION A.2 (POLYNOMIAL COMMITMENT). A polynomial commitment (PC) scheme is a tuple of four algorithms $(\text{Gen}, \text{Commit}, \text{Eval}, \text{Verify})$ defined as follows:

- $pk, vk \leftarrow \text{PC.Gen}(1^\lambda, l, d)$. Given the security parameter λ , number of variables l and degree d , generates the public parameters pk, vk .
- $C_f \leftarrow \text{PC.Commit}(pk, f, r)$. Given pk and a l -variate polynomial f of degree d and randomness $r \in \mathbb{F}$, returns a polynomial commitment C_f .
- $y, \pi \leftarrow \text{PC.Eval}(pk, f, x)$. Given pk, C_f, f and a point $x \in \mathbb{F}^l$, returns $y = f(x)$ and a proof π that $f(x) = y$.
- $0, 1 \leftarrow \text{PC.Verify}(vk, \pi, C_f, x, y)$. Given vk , the commitment C_f , the proof π , the y and x , returns 1 if $f(x) = y$, 0 otherwise.

Furthermore, any PCS must satisfy the following:

- **Completeness.** A polynomial commitment scheme is complete, if for any λ , and number of variables l , we have:

$$\Pr \left(\begin{array}{l} pk, vk \leftarrow \text{PC.Gen}(1^\lambda, l, d), \\ C_f \leftarrow \text{PC.Commit}(pk, f, r) \\ y, \pi \leftarrow \text{PC.Eval}(pk, f, x); \\ \text{PC.Verify}(vk, C, \pi, y, x) = 1 \end{array} \right) = 1$$

- **Knowledge Soundness.** A polynomial commitment scheme is knowledge sound, if for any λ, l, d and any probabilistic polynomial time adversaries $\mathcal{A}_{PC}$ there exists an extractor $\mathcal{E}_{PC}$ that has access to the random tape of $\mathcal{A}_{PC}$ such that:

$$\Pr \left(\begin{array}{l} pk, vk \leftarrow \text{PC.Gen}(1^\lambda, l, d), \\ C_f, y, \pi, x \leftarrow \mathcal{A}(pk, vk) : \\ f, r \leftarrow \mathcal{E}_{PC}(pk, vk) \wedge \\ \text{PC.Verify}(vk, C_f, \pi, y, x) = 1 \wedge \\ (f(x) \neq y \vee C_f \neq \text{PC.Commit}(pk, f, r)) \end{array} \right) \leq \text{negl}(\lambda)$$

B Additional Preliminaries

B.1 SNARK from the GKR Protocol

First introduced by Goldwasser et al. [33], the GKR protocol is an interactive proof system [34] that enables the prover to show the correct execution of a d -layered arithmetic circuit $C : \mathbb{F}^{N_0} \rightarrow \mathbb{F}^{N_d}$. Assuming that N_i is the number of gates of the i -th layer of C , the GKR represents each layer $i \in [d]$ of the circuit using

the functions $add_i, mul_i : \{0, 1\}^{2 \log N_{i-1} + \log N_i} \rightarrow \{0, 1\}$, such that $add_i(x, y, z) = 1$ (and $mul_i(x, y, z) = 1$ resp.) if the z -th gate is an addition (and multiplication resp.) gate and takes as input the gates with labels x, y (from the $(i-1)$ -th layer). Finally, we denote with $f_{add}^i, f_{mul}^i : \mathbb{F}^{2 \log N_{i-1} + \log N_i} \rightarrow \mathbb{F}$ and $f_i : \mathbb{F}^{\log N_i} \rightarrow \mathbb{F}$ the multi-linear extensions of add_i, mul_i and V_i respectively, where $V_i \in \mathbb{F}^{N_i}$ is a vector consisting of the output values of every gate in that layer.

To validate that V_d is the correct output of the circuit C on input V_d , the verifier first selects a random point $r_d \in \mathbb{F}^{\log N_d}$, computes $y_d = f_d(r_d)$ and sends r_d to the prover. Next, both parties interact to prove the correct computation of the d -th layer by following the *sumcheck protocol* [48] over the instance:

$$y_d = \sum_{x, y \in \{0, 1\}^{\log N_{d-1}}} f_{add_{d-1}}(x, y, r_d) (f_{d-1}(x) + f_{d-1}(y)) + f_{mul_{d-1}}(x, y, r_d) f_{d-1}(x) f_{d-1}(y)$$

At the end of the sumcheck protocol, the verifier will hold evaluation claims of f_{d-1} at r_{d-1}^1, r_{d-1}^2 and claims of $f_{add_{d-1}}, f_{mul_{d-1}}$ at $(r_{d-1}^1, r_{d-1}^2, r_d)$. It validates the correctness of the last two locally, while for the first two, it reduces the two claims into one following the techniques of [33, 67]. Next, proceeds with the same sumcheck but for the $d-1$ -th layer and the reduced claim y_{d-1} . This process repeats until reaching a claim for the input, which, can be verified locally.

SNARKs from GKR. Using a *polynomial commitment scheme*, we can turn the GKR protocol into a SNARK in the following way [67, 75]. First, the prover commits to f_w , the multi-linear extension of the witness w , and sends it to the verifier. Then, runs the GKR protocol to prove that $C(x, w) = 1$. Finally, the verifier ends up with two evaluation claims of f_w at the points r_d^1, r_d^2 , the validity of which is proven by “opening” the polynomial commitment at these points.

Performance wise, SNARKs based on GKR have $O(|C|)$ prover complexity and $\text{polylog}(|C|)$ proof size. Furthermore, if the underlying circuit is log-space uniform then the verification time is $\text{polylog}(|C|)$.

B.2 The KZG Commitment Scheme

In this section, we first present the KZG PCS for multi-linear polynomials first introduced in [53] and later we show how to use it in order to construct the Kopis PCS.

- $\mathbf{pk}, \mathbf{vk} \leftarrow \text{KZG.Gen}(\lambda, n)$: Given the security parameter λ and the number of coefficients n , the generation algorithm selects trapdoor variables $\mathbf{t} \in \mathbb{F}^{\log n}$, bases $g \in \mathbb{G}_1$ and $h \in \mathbb{G}_2$ and sets $\mathbf{pk} = \{g, h, \mathbf{G} = \{g^{\beta(\mathbf{i}, \mathbf{t})}\}_{\mathbf{i} \in \{0, 1\}^{\log n}}\}$ and $\mathbf{vk} = \{g, h, \mathbf{H} = \{h^{\mathbf{t}_i}\}_{i \in [\log n]}\}$. Finally, outputs $\mathbf{pk}, \mathbf{vk}$.
- $C \leftarrow \text{KZG.Commit}(\mathbf{pk}, f)$: Assuming that f_i is the i -th coefficient of f , commit computes and outputs $C \leftarrow \prod_{i \in [n]} \mathbf{G}_i^{f_i} (= g^{f(\mathbf{t})})$.
- $\pi, y \leftarrow \text{KZG.Open}(\mathbf{pp}, f, \mathbf{r})$: First, Open generates the polynomials $Q_i \in \mathbb{F}^{\log n - i} \rightarrow \mathbb{F}$, such that $f(\mathbf{x}) - f(\mathbf{r}) = \sum_{i \in [\log n]} (x_i - r_i) Q_i(x_{i+1}, \dots, x_{\log n})$ and calculates the proof $\pi = \{g^{Q_i(\mathbf{t}_{i+1}, \dots, \mathbf{t}_{\log n})}\}_{i \in [\log n]}$ (using $\mathbf{pk}$) and $y = f(\mathbf{r})$.

- $0, 1 \leftarrow \text{KZG.Verify}(\mathbf{vk}, \pi, y, \mathbf{r})$: Return 1 if and only if $e(C/g^y, h) = \prod_{i \in [\log n]} e(\pi_i, \mathbf{H}_i/h^{\mathbf{r}_i})$.

Next, we show how Kopis PCS is adapted to commit to the multi-linear extension of a matrix $\mathbf{M} \in \mathbb{F}^{n \times m}$ and subsequently open the commitment at a random evaluation point.

- $\mathbf{pp} \leftarrow \text{Kopis.Gen}(\lambda, n, m)$: Invokes and outputs the public parameters of the KZG algorithm for polynomials of size m and the generalized inner pairing product algorithm for instances of size n .
- $C \leftarrow \text{Kopis.Commit}(\mathbf{pp}, \mathbf{M})$: Assume that $\mathbf{M}_i \in \mathbb{F}^m$ is the i -th row of $\mathbf{M}$ and $f_i : \mathbb{F}^{\log m} \rightarrow \mathbb{F}$ its multi-linear extension. The prover computes $C_{f_i} \leftarrow \text{KZG.Commit}(\mathbf{pp}_{\text{KZG}}, f_i)$ for all $i \in [n]$. Finally sets $C \leftarrow \prod_{i \in [n]} e(C_{f_i}, \mathbf{h}_i)$.
- $\pi_{\text{Kopis}}, y \leftarrow \text{Kopis.Open}(\mathbf{pp}, \mathbf{M}, \mathbf{r})$: Let $(\mathbf{r}_1, \mathbf{r}_2) \in \mathbb{F}^{\log n} \times \mathbb{F}^{\log m}$ and $y \leftarrow f_{\mathbf{M}}(\mathbf{r})$, where $f_{\mathbf{M}}$ is the multi-linear extension of $\mathbf{M}$. The prover generates a proof π_{MIPP} for the MIPP relation showing that $C' = \prod_{i \in [n]} C_{f_i}^{\beta(\mathbf{i}, \mathbf{r}_1)}$ and then uses the *Open* algorithm of the KZG to prove that C' evaluates to y at the point $\mathbf{r}_2$. Finally, set $\pi_{\text{Kopis}} \leftarrow \{C', \pi_{\text{MIPP}}, \pi_{\text{KZG}}\}$.
- $0, 1 \leftarrow \text{Kopis.Verify}(\mathbf{pp}, \pi, y, \mathbf{r})$: Return 1 if π_{MIPP} and π_{KZG} are valid proofs.

Interestingly, observe that if the matrix has a single row, then Kopis PCS corresponds to the KZG PCS with base $\mathbb{G}_1$. Likewise, when the matrix has a single column, then the Kopis PCS corresponds to the KZG PCS with base $\mathbb{G}_2$.

C Details of ROGUE

In this section, we give more details of our matrix lookup argument. We begin by providing a detailed definition, then provide more information on how to instantiate and optimize specific parts of the protocol and finally prove Theorem C.1.

C.1 Algorithm Details

C.1.1 Proving Step 6 of the Update Algorithm. Here we show how to generate proof for the relation $\mathcal{R}$ as shown in **Construction 1**. To ease explanation, we will show how to generate a proof for the simpler relation $\mathcal{R}' = \{(C_F, y_\delta, C_I, \mathbf{r}); (\mathbf{F}, \mathcal{I}) : y_\delta = \sum_{j \in [k]} \beta(\mathbf{r}_1, \mathbf{i}_j) f_j(\mathbf{r}_2) \wedge C_F = \text{Kopis.Commit}(\mathbf{F}) \wedge C_I = \text{Commit}(\mathcal{I})\}$. For this, we first create a circuit that takes as (private) input the vector $\mathbf{v} = (f_1(\mathbf{r}_2), \dots, f_k(\mathbf{r}_2))$, all log n bit vectors $\mathbf{i}^{(l)} \in \{0, 1\}^k$ and public input the evaluation point $\mathbf{r}$ and outputs y_δ . In particular, the circuit (1) uses the bit vectors computes $\beta(\mathbf{r}_1, \mathbf{i}_j)$, for all $i_j \in \mathcal{I}$ and (2) outputs $y_\delta = \sum_{j \in [k]} \beta(\mathbf{r}_1, \mathbf{i}_j) \mathbf{v}_j$. Note that (1) corresponds to k multiplication trees of input size $\log n$ which can be proven using the specialized sumcheck protocol of [64]. At the same time (2) corresponds into an inner product which we can prove directly using the sumcheck protocol. At the end, the verifier ends up with evaluation claims of the multi-linear extensions of $\mathbf{i}^{(l)}$ at a random point $\mathbf{r}'$. To validate their correctness, the prover generates an evaluation proof for each one of them using $C_{idx}^{(l)}$. As for the evaluation claim of the multi-linear extension of $\mathbf{v}$ at $\mathbf{r}'$, we observe that $y_v = f_{\mathbf{v}}(\mathbf{r}') = f_{\mathbf{F}}(\mathbf{r}_2, \mathbf{r}')$ —where $f_{\mathbf{F}}$ the multilinear extension of $\mathbf{F}$. Consequently, to show the correctness of y_v , the prover generates an evaluation proof on C_F at $(\mathbf{r}_2, \mathbf{r}')$.

C.2 Proof of Theorem C.1

THEOREM C.1. *For any matrix $T \in \mathbb{F}^{n \times m}$, ROGUE is an updatable matrix lookup argument with $O(k \log n)$ lookup complexity and $O(\log^2 n)$ proof size/verification time where k the number of row lookups. Furthermore, it has $O(km \log n)$ update complexity, and $O(\log^2 n)$ proof size/verification time where k the changed rows.*

C.2.1 Complexity Analysis. In this section, we analyze the performance ROGUE, in terms of prover time, proof size and verification time.

PreProcess Algorithm: Committing to the table $T \in \mathbb{F}^{n \times m}$ requires an MSM of size nm . To compute the membership proofs for each row, the prover requires $n \log n$ MSMs of size m . Hence, the total prover complexity is $O(nm \log n)$.

Lookup Algorithm: Initially, to compute the commitment of F we need $O(k)$ pairing operations, while computing $C_{idx}^{(l)} \forall l \in [\log n]$ requires $O(k \log n)$ pairing operations. Step 1 requires $O(k \log n)$ pairing operations. Likewise for step 3, but with an additional step of computing C^* which requires MSM of size k . Step 4, requires proving $\log n$ TIPP instances and a single MIPP instance of size k leading to a proving complexity of $O(k \log n)$ pairing operations. As for Step 6, the prover has to use the sumcheck protocol for $\log n$ instances of size k which requires $O(k \log n)$ field operations. Furthermore, step 7 consists of $\log n$ openings of k -sized polynomials requiring $O(k \log n)$ pairing operations in total. Consequently, the total complexity is $O(k \log n)$ —dominated by pairing operations.

Verification Complexity & Proof Size. Regarding verification time, step 2 requires $\log n$ pairing operations while steps 4 and 7 are dominated by $O(\log n \log k) \approx O(\log^2 n)$ pairing operations. Step 5 requires $O(\log k)$ field operations and $O(\log n)$ pairing operations. Finally, step 6 requires $O(\log n \log k) = O(\log^2 n)$. Consequently, the total proof size for the lookup is dominated by $O(\log^2 n)$ pairing operations. Hence overall the verification complexity is $O(\log^2 n)$. We follow a similar argument for the proof size to show that it is $O(\log^2 n)$.

Update Algorithm: Regarding proving complexity, steps 1 requires $O(k \log n)$ group operations (using the algorithm of [63]) while step 2 requires $O(k \log n)$ pairing operations. Next, we can efficiently compute C_δ and $C_{\delta'}$ on step 3 using two MSMs of size ks and their evaluation proofs at r (step 5) in time $O(km \log n)$ (see Section 3). As for step 6, we use a SNARK based on the GKR protocol to prove $\mathcal{R}$, using a circuit similar to the one proposed in [74] that has size $O(km)$ leading to a prover complexity of $O(km)$. In total, update time is $O(km \log n)$.

Verification Complexity & Proof Size. As for verification complexity, step 2 requires $O(\log^2 n)$ pairing operations, step 5 requires $O(\log km)$ group operations while step 6 is dominated by $O(\log^2 k)$ field operations. Hence, the overall verification time is $O(\log^2 n)$. We follow a similar argument for the proof size to show that it is $O(\log^2 n)$.

C.2.2 Security proofs. Completeness comes directly from the completeness of the KZG and Kopis PCS but also the MIPP and TIPP protocols. Here we will focus on proving vector linkability and update consistency on the AGM model [30].

Vector Linkability. To prove vector linkability, we will show that for any adversary $\mathcal{A}$ against the vector linkability game, there exists an extractor $\mathcal{E}$ that outputs T, F and $\mathcal{I}$ such that $F = T|_{\mathcal{I}}$ with overwhelming probability. Our extractor first constructs $\log n$ adversaries against the TIPP extractability game and one adversary against the MIPP game and then invokes their corresponding extractors (as defined in [20]) to extract $\{\pi_{i_j, l}, h_{i_j, l}\}$ and $\{C_{f_{i_j}}\}$ for every $l \in [\log n], i_j \in \mathcal{I}$. What remains is to show that $e(C_T/C_F, h) = \prod_{l \in [\log n]} e(\pi_{i_j, l}, h_{i_j, l})$ for every $i_j \in \mathcal{I}$. If this was not the case, then the adversary could use the extracted information to break the q-SDH assumption [14] (the proof for this is identical to the one of [63] Theorem B.2). Furthermore, because we work in the AGM model, the extractor can directly obtain $F_1, \dots, F_k$ from $C_{F_1}, \dots, C_{F_k}$ and T from C_T . What remains is to show that $T|_{\mathcal{I}} = F$. For this, the extractor creates an adversary against the extractability game of SNARKs based on the GKR protocol and invokes the extractor (as defined in [67, 75]) to extract the bits i_l , for all $l \in [\log n]$. By the knowledge soundness definition of SNARKs, it should hold that $i_l \in \{0, 1\}^k$ for all $l \in [\log n]$ with overwhelming probability. By the binding property of the pairing commitment, it should also hold that $e(C_T/C_F, h) = \prod_{l \in [\log n]} e(\pi_{i_j, l}, h^{t_l - i_j, l})$, which means that $f_T(y, x) - f_j(x) = \sum_{k \in [\log n]} (y_k - i_{j, k}) Q_{j, k}(y, x)$ and thus $T_{i_j} = F_j$, for all $i_j \in \mathcal{I}$ since otherwise we could use the adversary to break the l-SDH [14] as we could find two different multi-linear KZG openings which happens only with negligible probability.

Update Consistency What remains is to show that our construction is update consistent. For this we will show how to build from any PPT adversary $\mathcal{A}$ a PPT extractor $\mathcal{E}$ that extracts $T, T', F' \in \mathbb{F}^{n \times m}$, $F' \in \mathbb{F}^{k \times m}$ and $\mathcal{I}$ such that $T'_j = F'_j$ for all $j \in [k]$ and $T'_i = T_i$ for all $i \in [n] \setminus \mathcal{I}$ with overwhelming probability. Initially, $\mathcal{E}$ constructs two adversaries against the lookup consistency and then invokes their corresponding extractors (as defined in the previous section), to extract F, F', T and T' along with the indexes $\mathcal{I}$. Due to the lookup consistency of our construction, it would hold that $F = T|_{\mathcal{I}}$, $F' = T'|_{\mathcal{I}}$ with overwhelming probability. Then, the extractor creates an adversary against the knowledge soundness game of the KZG PCS, and invokes its extractor to get the polynomials $\delta(x, y)$ and $\delta'(x, y)$. Finally, it creates an adversary against the knowledge soundness game of the SNARK for $\mathcal{R}$ (based on the GKR protocol) and invokes its extractor (as defined in [67, 75]) to get I^*, F'^*, F^* . Due to the knowledge soundness of SNARKs, it should hold that $y_\delta = \sum_{i \in \mathcal{I}^*} \beta(r_1, i) f_i^*(r_2)$ and $y'_{\delta} = \sum_{i \in \mathcal{I}^*} \beta(r_1, i) f_i'^*(r_2)$.

Given all extracted information, it remains to show that T' is the correct update from T . First, due to the Schwartz-Zippel lemma, it holds that $\delta(x, y) = \sum_{i \in \mathcal{I}} \beta(i, x) f_i^*(y)$ with $\log(nm)/|\mathbb{F}|$ probability (and likewise for δ'). Furthermore, due to the binding property of the Kopis PCS, $F^* = F$ and $F'^* = F'$ with probability $1 - \text{negl}(\lambda)$ since otherwise we could use $\mathcal{A}$ to construct an adversary that breaks the q-ASDBP assumption [20]. Finally, it should hold that $f_T(x, y) - \delta(x, y) + \delta'(x, y) = f_{T'}(x, y)$ (and consequently all vectors other than those in $\mathcal{I}$ remained unchanged), with overwhelming probability since otherwise we could construct an adversary that breaks the binding of the multi-variate KZG commitment which happens only with negligible probability.

D Details of our Verifiable Database Scheme

D.1 Definition of our Verifiable Database Scheme

DEFINITION D.1. A verifiable database scheme is a tuple of seven algorithms (*Gen*, *CommitTable*, *VerifyCommit*, *ProveQuery*, *VerifyQuery*, *ProveUpdate*, *VerifyUpdate*) defined as follows:

- $\mathbf{vk}, \mathbf{pk} \leftarrow \text{Gen}(\lambda, n, m)$: Given the security parameter λ , the maximum possible number of tuples $n_{\max}$ and attributes $m_{\max}$ of a database table, outputs verification and proving public parameters.
- $C_T, \{C_T^i, \pi_i\}_{i \in \mathcal{J}} \leftarrow \text{CommitTable}(\mathbf{pk}, T, \mathcal{J})$: Given $\mathbf{pk}$, a table $T \in \mathbb{F}^{n \times m}$ and a set of attributes for which we want to construct indexes, return the table commitment C_T , and index commitment C_T^i and a proof π_i that the index was computed correctly for each $i \in \mathcal{J}$.
- $0, 1 \leftarrow \text{VerifyCommit}(\mathbf{vk}, C_T, \{C_T^i, \pi_i\}_{i \in \mathcal{J}}, \mathcal{J})$: Return 1 if all proofs π_i are valid with respect to C_T and C_T^i .
- $\pi, y \leftarrow \text{ProveQuery}(\mathbf{pk}, \mathcal{DB}, Q)$: Given the database $\mathcal{DB}$ containing all tables along with their indexes and a query description Q , output a proof π showing that $y = \text{QUERY}(\mathcal{DB}, Q)$.
- $0, 1 \leftarrow \text{VerifyQuery}(\mathbf{pk}, C_{\mathcal{DB}}, Q, \pi, y)$: Given $\mathbf{pk}$, the commitments of all tables and indexes, the query and the output y return 1 if π is valid.
- $\pi, C_{T'}, \{C_{T'}^i\}_{i \in \mathcal{J}} \leftarrow \text{ProveUpdate}(\mathbf{pk}, T, \{\mathcal{T}_i\}_{i \in \mathcal{J}}, \mathcal{J}, Q)$: Given $\mathbf{pk}$, a table T , its indexes and an update query, return a proof π showing that $T', \{\mathcal{T}_i'\}_{i \in \mathcal{J}} \leftarrow \text{UPDATE}(T, \{\mathcal{T}_i\}_{i \in \mathcal{J}}, Q)$ along with the commitments of $C_{T'}, \{C_{T'}^i\}_{i \in \mathcal{J}}$.
- $0, 1 \leftarrow \text{VerifyUpdate}(\mathbf{vk}, C_T, C_{T'}, \{C_T^i\}_{i \in \mathcal{J}}, \{C_{T'}^i\}_{i \in \mathcal{J}}, \mathcal{J}, Q)$: Use $C_T, C_{T'}, \{C_T^i\}_{i \in \mathcal{J}}, \{C_{T'}^i\}_{i \in \mathcal{J}}$ to validate the correctness of π . If π is valid then set $C_{T'} \leftarrow C_T$ and $\{C_{T'}^i\}_{i \in \mathcal{J}} \leftarrow \{C_T^i\}_{i \in \mathcal{J}}$ and return 1.

And has the following properties:

- **Index Building Completeness.** For any security parameter λ , $\mathbf{pk}, \mathbf{vk}$, table T and set of attributes $\mathcal{J}$ it holds that:

$$\Pr \left(\begin{array}{l} \mathbf{pk}, \mathbf{vk} \leftarrow \text{Gen}(1^\lambda, n_{\max}, m_{\max}), \\ C_T, \mathcal{J}, \{C_T^i, \pi_i\}_{i \in \mathcal{J}} \leftarrow \text{CommitTable}(\mathbf{pk}, T, \mathcal{J}), \\ 1 = \text{VerifyTable}(\mathbf{vk}, C_T, \{C_T^i, \pi_i\}_{i \in \mathcal{J}}, \mathcal{J}) \end{array} \right) = 1$$

- **Table & Index Extractability.** For any PPT adversary $\mathcal{A}$ there exists a PPT extractor $\mathcal{E}$ such that:

$$\Pr \left(\begin{array}{l} \mathbf{pk}, \mathbf{vk} \leftarrow \text{Gen}(1^\lambda, n_{\max}, m_{\max}), \\ C_T, \mathcal{J}, \{C_T^i, \pi_i\}_{i \in \mathcal{J}} \leftarrow \mathcal{A}(\mathbf{pk}, \mathbf{vk}), \\ T, \{\mathcal{T}_i^{dx}\}_{i \in \mathcal{J}} \leftarrow \mathcal{E}^{\mathcal{A}}(\mathbf{pk}, \mathbf{vk}), \\ 1 = \text{VerifyTable}(\mathbf{vk}, C_T, \{C_T^i, \pi_i\}_{i \in \mathcal{J}}, \mathcal{J}) = 1 \wedge \\ \exists i \in \mathcal{J} : 1 \neq \text{Valid}(T, \mathcal{T}_i^{dx}) \end{array} \right) < \text{negl}(\lambda)$$

Where the function $\text{Valid}(T, \mathcal{T}_i^{dx})$ checks whether all rows of $\mathcal{T}_i^{dx}$ correspond to the leaves of a valid B+ tree constructed from T .

- **Query Completeness.** For any $\mathbf{pk}, \mathbf{vk}$, set of involved tables $\mathcal{DB}$ and query Q it holds that:

$$\Pr \left(\begin{array}{l} \pi_Q, y \leftarrow \text{ProveQuery}(\mathbf{pk}, \mathcal{DB}, Q), \\ y = \text{ANSWER}(Q, \mathcal{DB}) \wedge \\ 1 = \text{VerifyQuery}(\mathbf{vk}, C_{\mathcal{DB}}, Q, \pi, y) \end{array} \right) = 1$$

- **Database Extractability.** For any PPT adversary $\mathcal{A}$ there exists a PPT extractor $\mathcal{E}$ such that:

$$\Pr \left(\begin{array}{l} Q, C_{\mathcal{DB}}, \pi_Q, y \leftarrow \mathcal{A}(\mathbf{pk}, \mathbf{vk}), \\ \mathcal{DB} \leftarrow \mathcal{E}^{\mathcal{A}}(\mathbf{pk}, \mathbf{vk}), \\ y \neq \text{ANSWER}(Q, \mathcal{DB}) \wedge \\ 1 = \text{VerifyQuery}(\mathbf{vk}, C_{\mathcal{DB}}, Q, \pi, y) \end{array} \right) < \text{negl}(\lambda)$$

- **Update Completeness** For any $\mathbf{pk}, \mathbf{vk}$, table T , set of indexes $\{\mathcal{T}_i\}_{i \in \mathcal{J}}$ and query Q it holds that:

$$\Pr \left(\begin{array}{l} C_T, \{C_T^i\}_{i \in \mathcal{J}} \leftarrow \text{CommitTable}(\mathbf{pk}, T, \mathcal{J}) \\ \pi_{\text{upd}}, C_{T'}, \{C_{T'}^i\}_{i \in \mathcal{J}} \leftarrow \text{ProveUpdate}(\mathbf{pk}, T, \{\mathcal{T}_i\}_{i \in \mathcal{J}}, Q), \\ T', \{\mathcal{T}_i'\}_{i \in \mathcal{J}} = \text{UPDATE}(T, \{\mathcal{T}_i\}_{i \in \mathcal{J}}) \wedge \\ 1 = \text{VerifyUpdate}(\mathbf{vk}, C_T, C_{T'}, \{C_T^i, C_{T'}^i\}_{i \in \mathcal{J}}, Q, \pi) \end{array} \right) < \text{negl}(\lambda)$$

- **Update Extractability** For any PPT adversary $\mathcal{A}$ there exists a PPT extractor $\mathcal{E}$ such that:

$$\Pr \left(\begin{array}{l} C_T, \{C_T^i, \pi_i\}_{i \in \mathcal{J}}, C_{T'}, \{C_{T'}^i\}_{i \in \mathcal{J}}, Q, \mathcal{J} \leftarrow \mathcal{A}(\mathbf{pk}, \mathbf{vk}) \\ T, T', \{\mathcal{T}_i^{dx}, \mathcal{T}_i^{dx'}\}_{i \in \mathcal{J}} \leftarrow \mathcal{E}^{\mathcal{A}}(\mathbf{pk}), \\ 1 = \text{VerifyTable}(\mathbf{vk}, C_T, \{C_T^i, \pi_i\}_{i \in \mathcal{J}}, \mathcal{J}) \wedge \\ 1 = \text{VerifyUpdate}(\mathbf{vk}, C_T, C_{T'}, \{C_T^i, C_{T'}^i\}_{i \in \mathcal{J}}, Q, \pi) \wedge \\ (T' \neq \text{UPDATE}(T, Q) \vee \exists i \in \mathcal{J} : \text{s.t.} : \\ 1 \neq \text{Valid}(T, \mathcal{T}_i^{dx}) \vee 1 \neq \text{Valid}(T', \mathcal{T}_i^{dx'})) \end{array} \right) < \text{negl}(\lambda)$$

D.2 Proving Other Queries

Proving Other Query Types. For other query types such as group-by or aggregation queries we use the techniques presented in [37] but adapt them to work over SNARKs based on the GKR protocol.

Proving Expressive Update Queries. Here we will show how to prove expressive update queries, e.g., queries of the form “UPDATE T ; SET $\text{Attr}_1 = v_1$; WHERE $\text{Attr}_2 = v_2$ ”. Initially, the prover generates two SNARK proofs. In the first one, it shows the correct computation of T_u , which is the table satisfying the update predicate (e.g., the query “UPDATE T ; SET $\text{Attr}_1 = v_1$; WHERE $\text{Attr}_2 = v_2$ ”). In the second one, it proves the correct computation of T'_u which is the table after applying all changes to T_u . Then, using T_u and T'_u it proves that all index commitments have been correctly updated. For each index, we can achieve this by first proving that all tuples in T_u have been removed, and then, prove the correct insertion of all tuples in T'_u by using the insertion and deletion algorithms described above. Note that with this approach, the prover has to invoke the update algorithm of the lookup protocol twice. While this is unavoidable for the index on Attr_1 , we can optimize the above process for all other indexes. In particular, because we only change the contents of the tuples in these indexes and do not rearrange them, we can construct a single update circuit that takes as input the nodes containing all tuples in T_u and T'_u from the old and new index, respectively, and checks whether the updates have been performed correctly. In this way, we only invoke the update algorithm once, thus further improving prover time.

In more detail, assume a single index $\mathcal{T} \in \mathbb{F}^{n \times m}$ and its updated version $\mathcal{T}' \in \mathbb{F}^{n \times m}$. The prover first retrieves all nodes $N, N' \in \mathbb{F}^{k \times \mathcal{O}(Km)}$ from $\mathcal{T}$ and $\mathcal{T}'$ containing at least one tuple. Then, the prover generates a proof for the correct computation of a circuit

$C_{\mathcal{U}}$ that takes as input $\mathbf{N}, \mathbf{N}', \mathbf{T}, \mathbf{T}'$ and outputs 1 if and only if (1) all tuples of $\mathbf{T}$ and $\mathbf{T}'$ belong to $\mathbf{N}$ and $\mathbf{N}'$ respectively, and (2) $\mathbf{N}$ and $\mathbf{N}'$ are identical in all other tuples. Both (1)-(2) can be checked in a straightforward way by $C_{\mathcal{U}}$ using similar techniques with the ones described above. Note that the size of the circuit is $O(k \cdot Km + nm) = O(nKm)$ since $k = O(n)$. Furthermore, the depth of the circuit is $O(\log(nKm))$.

D.3 Description of our Verifiable Database Scheme

Construction 2 gives a detailed description of our verifiable database scheme. In this section, we emphasize the specifics of each algorithm that were not included in the main text and **Construction 2**, particularly regarding the instantiation of circuits for each operation.

Committing indexes for $m > 1$. Whenever the table $\mathbf{T}$ has more than one attributes, the prover stores the entire tuples in $\mathbf{T}^{idx}$ and not only the values in the corresponding attributes. Likewise, we adapt our proof generation algorithm to work over tuples instead of single elements which can be achieved in a straightforward way. We note that this does not blow up prover time since the prover can generate every row of $\mathbf{T}^{idx}$ “on the fly” using access to $\mathbf{T}$. **CommitTable**.

As explained in the main body, **CommitTable** commits to a table and its indexes and generates a SNARK proof for the correctness of the latter. To achieve this, we construct a circuit C_{idx} that takes as input $\mathbf{T} \in \mathbb{F}^{n \times m}$, $\mathbf{T}_i^{idx} \in \mathbb{F}^{n/K \times O(Km)}$ along with some additional auxiliary information and returns one if $\mathbf{T}_i^{idx}$ is well formed, namely, it represents a sorted linked list. Recall from the main body that C_{idx} checks the following. First, it ensures that (i) $\mathbf{t}.k_{min} - \mathbf{t}.k[i] \geq 0$, (ii) $\mathbf{t}.k[i] - \mathbf{t}.k[i-1] \geq 0$, and (iii) $\mathbf{t}.id - \mathbf{t}.ptr \neq 0$ for all $i \in [K]$ (where K the maximum tuples stored in a node) and $\mathbf{t} \in \mathbf{T}_i^{idx}$. These checks are validated through two arithmetic circuits such that their first layer computes the desired differences, and subsequent layers perform the range checks. For the latter, we use lookup arguments based on lasso [62], which correspond to circuits representing a multiplication tree. Furthermore, the circuit has to ensure that $\mathcal{N}_{next} \cup (0, 0) = \mathcal{N}_{curr} \cup (+\infty, +\infty)$. For this, it is enough to construct a circuit showing that:

$$\prod_{i \in [n/K]} (a\mathbf{T}^{idx}[i].ptr + b\mathbf{T}^{idx}[i].k_{min} + c) = C_{inf} \prod_{i \in [n/K]} (a\mathbf{T}^{idx}[i].id + b\mathbf{T}^{idx}[i].k[0] + c)$$

where C_{inf} is the dot product of $(+\infty, +\infty, 1)$ and (a, b, c) . Note that this circuit consists of an aggregation layer followed by a multiplication tree. Similarly, we prove that $\{\mathbf{t}.idx\}_{\mathbf{t} \in \mathbf{T}^{idx}} = [n/K]$ and also that $\mathbf{T}^{idx}$ contains all the tuples of $\mathbf{T}$ as it translates to a set equality, e.g., $\{\mathbf{T}^{idx}[j].k[i]\}_{i,j \in [K] \times [n/K]} = \mathbf{T}$.

Observe that the circuits used to ensure (i),(iii), pointer and identifier consistency have size $O(n/K)$. The circuit used to prove (ii) has size $O(n)$ while the circuit used to prove that all tuples are included in the index has size $O(nm)$. Consequently, the size of the circuit is $O(nm)$. Furthermore, because almost all circuits correspond to multiplication trees, its depth will be $O(\log(nm))$.

ProveQuery. As shown in **Construction 2**, at the beginning of the *ProveQuery* algorithm, the prover decomposes Q into multiple basic queries (step 1). Then it generates a SNARK proof for each basic query Q_j by first converting it into an arithmetic circuit C_{Q_j} and then proving its correct computation using a SNARK based on the GKR protocol. For most queries, we use the circuits presented in [37, 75]. Here, we will provide more details on constructing the circuits that check the correctness of selection/range and join queries.

Arithmetic Circuit For Selection/Range Queries (with indexes). Let C_s be an arithmetic circuit (similar to that of [75]) that takes as input the table $\mathbf{T} \in \mathbb{F}^{n \times m}$ and the table formed from the tuples selected by the query $\mathbf{T}' \in \mathbb{F}^{n' \times m}$ and returns 1 if and only if $\mathbf{T}'$ has been computed honestly. When a table has an index on the predicate attribute, we construct a circuit C'_s as described in the main body. First, C'_s takes as input a subset of nodes of $\mathbf{F} \subseteq \mathbf{T}^{idx}$ the output table $\mathbf{T}'$ and works as follows. It constructs the sub-table $\tilde{\mathbf{T}}$ from $\mathbf{F}$ by filtering everything except from its tuples and uses C_s on $\tilde{\mathbf{T}}, \mathbf{T}'$ to ensure the correctness of $\mathbf{T}'$. Next, it validates that $\mathbf{F}$ encompasses a single list. For this, it arranges $\mathbf{F}$ in such a way that $\mathbf{F}[0]$ is the head node and $\mathbf{F}[1]$ the tail and checks if

$$(\mathbf{F}[0].id, \mathbf{F}[0].p[0]) \cup \{(\mathbf{F}[i].ptr, \mathbf{F}[i].k_{min})\}_{i \in [|\mathbf{F}|]} = (\mathbf{F}[1].ptr, \mathbf{F}[1].k_{min}) \cup \{(\mathbf{F}[i].id, \mathbf{F}[i].k[0])\}_{i \in [|\mathbf{F}|]}$$

using the same circuit with the one described in the **CommitTable** algorithm. Finally, it ensures that $\mathbf{F}[0].k[0]$ and $\mathbf{F}[1].k_{min}$ do not satisfy the selection predicate using two range checks.

It is not hard to see that the size of $|C'_s|$ is proportional to the size of $\mathbf{F}$ which in practice is $O(n'm)$. Furthermore, because it encompasses two multiplication trees (one for C_s and one for proving that $\mathbf{F}$ is a single linked list), its depth should be $O(\log(n'm))$.

Arithmetic Circuit For Joins. Here, we will present how to construct a circuit that proves the correctness of the join operation. As a starting point, we show how to construct a circuit C_{join} for proving primary-foreign key joins. Next, we show how to modify it for non-primary-foreign key ones. At a high-level, C_{join} takes as input the tables $\mathbf{T}_3 = (\mathbf{T}_3^1, \mathbf{T}_3^2)$, $\mathbf{T}_1, \mathbf{T}_2$ and $\mathbf{T}_1^1, \mathbf{T}_1^2, \mathbf{T}_2^1, \mathbf{T}_2^2$ as defined in the main body. To confirm that $\mathbf{T}_i^1 \cup \mathbf{T}_i^2 = \mathbf{T}_i$, for every $i \in \{1, 2\}$, it computes:

$$\prod_{i \in [n_i]} \left(\sum_{j \in [m_i]} a^j \mathbf{T}_i[i][j] \right) = \prod_{i \in [n_i^{(1)}]} \left(\sum_{j \in [m_i^{(1)}]} a^j \mathbf{T}_i^1[i][j] \right) \cdot \prod_{i \in [n_i^{(2)}]} \left(\sum_{j \in [m_i^{(2)}]} a^j \mathbf{T}_i^2[i][j] \right)$$

for a random point a selected by the verifier. Naturally, this translates into 3 arithmetic circuits representing an inner product followed by a multiplication tree. Now, to prove that $\mathbf{T}_{1|Attr_1}^2 \cap \mathbf{T}_{2|Attr_2}^2 = \emptyset$ and $\mathbf{T}_{1|Attr_1}^1 \cap \mathbf{T}_{2|Attr_2}^1 = \emptyset$ we follow the techniques of [75]. Namely, to prove that each intersection is empty, the circuit takes additional input an auxiliary vector $\mathbf{v} \in \mathbb{F}^{n_1^{(2)} + n_2^{(2)}}$ containing $(\mathbf{T}_{1|Attr_1}^2 || \mathbf{T}_{2|Attr_2}^2)$ but in sorted order. Then, using a lookup argument, it shows that $\mathbf{v}[i] - \mathbf{v}[i-1] > 0$, for all $i \in [n_1^{(2)} + n_2^{(2)}]$ and using the set equality circuit as described above it shows that $\mathbf{v}$ is a permutation of $(\mathbf{T}_{1|Attr_1}^2 || \mathbf{T}_{2|Attr_2}^2)$. For simplicity, we assumed

that sets have unique values which not the case in practice. To de-duplicate the sets we use the approach of [37].

Next, assuming that T_1 is the table with the primary key, the circuit ensures that $T_1^1 = T_3^1$ using the set equality check. However, it can not do the same for T_2^1 and T_3^2 as the latter possibly contains multiple copies of the same tuple from T_2^1 . Instead, our circuit uses a single lookup argument such as the LogUp-GKR [56], to show that every tuple of T_3^2 belongs to T_2^1 . Last but not least, to prove that T_3 satisfies the correctness of the predicate, it checks if $T_3^1[i][Addr_1] - T_3^2[i][Addr_2] = 0$, for every $i \in [n_3]$.

Adapting C_{join} for non-primary-foreign key joins. First, to confirm that “ $T_i^1 = T_i^2$ ” the circuit uses a lookup argument for both tables since T_3^1 might contain multiple copies of the same tuple in T_i^1 . Next, it also includes three copies (of different sizes) of the circuit C_{Hist} that takes as input the vector $T_{i|Attr_i}$, for $i \in [3]$, and outputs the histogram $\mathbf{h}_i$ of the unique keys. Then checks if $\mathbf{h}_1 \circ \mathbf{h}_2 = \mathbf{h}_3$. What remains is to show that every tuple of T_3 is distinct. First, C_{join} computes a vector $\mathbf{v} \in \mathbb{F}^{n_3}$ accumulating the primary keys of the sub-tables T_3^1 and T_3^2 such that if two pairs of keys are different, then their accumulated value would also be different (e.g., by multiplying one key value by a power of two and then aggregating it with the other one). Then, provided an additional input $\mathbf{v}' \in \mathbb{F}^{n_3}$, claimed to be the sorted variant of $\mathbf{v}$, it tests whether $\mathbf{v}'[i] - \mathbf{v}'[i-1] > 0$ and also that $\mathbf{v}'$ is a permutation of $\mathbf{v}$.

For primary-foreign key joins, the size of the circuit is $O(n_1m_1 + n_2m_2 + n_3m_3)$. In particular, checking that $T_i^1 \cup T_i^2 = T_i$ requires $O(n_im_i)$ gates, $T_i^2 \cap T_i^1 = \emptyset$ requires $O(n_1 + n_2)$ gates, $T_i^1 = T_i^2$ requires $O(n_im_i + n_3m_3)$ gates.

Our new circuit has asymptotically the same size with the one for non-primary-foreign key joins since each histogram circuit requires $O(n_i + \min(n_1, n_2))$ gates, while the circuit ensures that each tuple in T_3 is unique needs $O(n_3)$ additional gates.

ProveUpdate. In this paragraph, we give more details on how we prove the correctness of update queries and specifically insertions (which are identical to deletions).

Proving Insertions. Upon receiving the query, the prover parses Q to extract all tuples $\mathcal{D}$ to be inserted. Then, it inserts $\mathcal{D}$ to the original index $\mathcal{T}$ and forms the updated index $\mathcal{T}'$. Using $\mathcal{T}$ and $\mathcal{T}'$, it retrieves the set of nodes $\mathbf{N} \in \mathbb{F}^{k \times O(K)}$, $\mathbf{N}' \in \mathbb{F}^{k \times O(K)}$, $\mathbf{L} \in \mathbb{F}^{l \times O(K)}$ and $\mathbf{R} \in \mathbb{F}^{r \times O(K)}$. The prover computes these nodes the same way described in the main body with a minor difference that $\mathbf{N}$ is padded with the same number of rows as $\mathbf{N}'$. That is done by appending dummy nodes (e.g., zero vectors) from $\mathbf{T}^{idx}$ into $\mathbf{N}$. In this way, the prover also commits to a single index $\mathcal{I}$ containing the row indexes of $\mathbf{N}$.

Now we have everything we need to show how to construct the circuit $C_{\mathcal{U}}$ that validates the correctness of $\mathbf{N}'$. In particular, $C_{\mathcal{U}}$ takes as input $\mathbf{N}, \mathbf{N}', \mathbf{L}, \mathbf{R}$ and $\mathcal{I}$, and works as follows. First it validates the correctness of $\mathbf{L}, \mathbf{R}$, namely that $\mathbf{L} \cup \mathbf{R} \cup \mathbf{N}$ represents l lists starting from $\mathbf{L}$ and ending up in $\mathbf{R}$. To ensure this, it checks if $\{(\mathbf{L}[i].ptr, \mathbf{L}[i].k_{min})\}_{i \in [l]} \cup \{\mathbf{N}[i].ptr, \mathbf{N}[i].k_{min}\}_{i \in [k]}$ equals to $\{(\mathbf{R}[i].id, \mathbf{R}[i].k[0])\}_{i \in [l]} \cup \{\mathbf{N}[i].id, \mathbf{N}[i].k[0]\}_{i \in [k]}$ which translates to a set equality circuit.

Next, it ensures the correctness of $\mathbf{N}'$ by checking that the set of lists in $\mathbf{L} \cup \mathbf{R} \cup \mathbf{N}'$ start from $\mathbf{L}$, end up in $\mathbf{R}$ and are sorted. It does so by using the same check as with ensuring the correctness of $\mathbf{L}$ and

$\mathbf{R}$ but also the consistency checks used to show the well-formedness of the tree. Recall that the circuit ensuring the correctness of index construction also ensured that the identifiers of each node were assigned properly. To guarantee this when proving correctness of insertions, $C_{\mathcal{U}}$ first checks that (i) for each $i_j \in \mathcal{I}$, $i_j = \mathbf{N}'[j].id$ and also that (ii) all indexes in $\mathcal{I}$ are unique using the same circuit with the one used to check the uniqueness of tuples in the joined result. What remains is to show that $\mathbf{N}'$ contains all tuples of $\mathbf{N}$ but also all tuples in $\mathcal{D}$. $C_{\mathcal{U}}$ ensures this by checking where the set $\mathcal{D} \cup \{\mathbf{N}[i].k[j]\}_{i,j \in [k] \times [K]}$ equals to $\{\mathbf{N}'[i].k[j]\}_{i,j \in [k] \times [K]}$.

Assuming that n is the number of inserted tuples, $C_{\mathcal{U}}$ has $O(k \cdot Km + nm)$ gates. In more detail, the circuit checking the correctness of $\mathbf{L}$ and $\mathbf{R}$ requires $O(k)$ gates, the one that checks the well-formedness of $\mathbf{N}'$ requires $O(k \cdot K)$ gates while the circuit ensuring that tuples remain consistent has size $O(k \cdot Km + nm)$ gates. Because $k = O(n)$, the total size of the circuit becomes $O(nKm)$. Due to the use of multiplication trees for lookups and set equality checks, the circuit has $O(\log nK)$ depth.

D.4 Proof of Theorem D.1

THEOREM D.1. **Construction 2** is a verifiable database scheme. For a table $T \in \mathbb{F}^{n \times m}$, and index set $\mathcal{J}$, *CommitTable* has $O(nm(1 + |\mathcal{J}| \log(n/K)))$ prover complexity and $O(|\mathcal{J}| \log^2(nm))$ proof size and verification time where K is the number of tuples that reside in a single node of the index. When using indexes, a point & range query has prover complexity of $O((k/K) \cdot \log(n/K))$ where k the number of selected tuples and $O(\log^2(km))$ verification time and proof size. Furthermore, equi-joins have prover complexity of $O(n_1m_1 + n_2m_2 + n_3m_3)$ and $O(\log^2(\max\{n_im_i\}_{i \in [3]}))$ verification time and proof size. Finally, when using indexes, *ProveUpdate* has prover complexity of $O(|\mathcal{J}| kKm \log(n/K))$ and $O(|\mathcal{J}| \log^2(kKm))$ verification time and proof size where k the number of changed nodes.

D.4.1 Complexity Analysis. In this section, we analyse the performance of our verifiable database scheme.

CommitTable. If the table has no index, then the prover complexity of *CommitTable* for a table $T \in \mathbb{F}^{n \times m}$ is $O(nm)$. Otherwise, the prover complexity is $O(|\mathcal{J}| \cdot (|\mathcal{C}_{idx}| + nm \log(n/K))) = O(|\mathcal{J}| nm \log(n/K))$. Finally, the size of each individual proof showing the correct computation of the index is $O(\log^2(nm))$ leading to a total proof size of $O(|\mathcal{J}| \log^2(nm))$. Likewise we argue for verification time.

ProveQuery. Because we use a SNARK based on the GKR protocol, the prover complexity will be linear to the size of the circuit used to validate the correctness of a given circuit. Here we will focus on analysing the prover complexity of selection and join queries. For selection queries, it takes $O((k/K) \cdot \log(n/K))$ time to lookup all desired nodes and $O(|\mathcal{C}_s|) = O(km)$ to prove the correct computation of the circuit C_s . Consequently, the prover time for point and range queries is $O((k/K) \cdot \log(n/K) + km)$. Furthermore, because the depth of C_s is $O(\log(km))$, the proof size and verification time will be $O(\log^2(km))$. Regarding join queries, as we presented in the above section, C_{join} has size $O(n_1m_1 + n_2m_2 + n_3m_3)$ and $O(\log(\max\{n_im_i\}))$ depth where $n_1 \times m_1, n_2 \times m_2$ are the dimensions of the input tables and $n_3 \times m_3$ the dimensions of the output table. Consequently, the prover complexity will be

Construction 2: Verifiable Database Scheme with Indexes

- $\mathbf{pk}, \mathbf{vk} \leftarrow \text{Gen}(n_{\max}, m_{\max})$: Return $\text{ROGUE.Gen}(n_{\max}, m_{\max}, k_{\max})$.
- $C_T, \{C_T^i, \pi_i\}_{i \in \mathcal{J}} \leftarrow \text{CommitTable}(\mathbf{pk}, \mathbf{T} \in \mathbb{F}^{n \times m}, \mathcal{J})$: Compute the commitment of the table and indexes and also prove their well formedness:
 - (1) Compute $C_T \leftarrow \text{Kopis.Commit}(\mathbf{pk}, \mathbf{T})$. If $\mathcal{J} = \emptyset$ return C_T .
 - (2) For every $i \in \mathcal{J}$, compute the B+ tree for the i -th attribute $\mathcal{T}^i$, parse its leaf nodes as a matrix $\mathbf{T}_i^{\text{idx}}$ and invoke $C_T^i, \mathbf{aux}^i \leftarrow \text{PreProcess}(\mathbf{pk}, \mathbf{T}_i^{\text{idx}})$.
 - (3) Let C_{idx} be the arithmetic circuit that validates the well-formedness of the leaf nodes (as described in the main body and Appendix D.3) of each committed index. For each $i \in \mathcal{J}$, generate a SNARK proof for the relation $\mathcal{R}_{\text{idx}} = \{(C_T, C_T^i); (\mathbf{T}, \mathbf{T}_i^{\text{idx}}, \mathbf{w}) : C_{\text{idx}}(\mathbf{T}, \mathbf{T}_i^{\text{idx}}, \mathbf{w}) = 1 \wedge C_T = \text{Kopis.Commit}(\mathbf{pk}, \mathbf{T}) \wedge C_T^i \leftarrow \text{PCS.Commit}(\mathbf{pk}, \mathbf{T}_i^{\text{idx}})\}$ using the techniques of [57, 75]. Here PCS is either the KZG or Kopis commitment.
 - (4) Internally store $\{\mathbf{aux}^i\}_{i \in \mathcal{J}}$ and output $C_T, \{C_T^i, \pi_i\}_{i \in \mathcal{J}}$. Finally select an attribute $t \in \mathcal{J}$, and set $C_T \leftarrow C_T^t$.
- $0, 1 \leftarrow \text{VerifyTable}(\mathbf{vk}, C_T, \{C_T^i, \pi_i\}_{i \in \mathcal{J}}, \mathcal{J})$: If $\mathcal{J} \neq \emptyset$ return 1 if and only if $1 = \text{SNARK.Verify}(\mathbf{vk}, C_T, C_T^i, \pi_i)$, for all $i \in \mathcal{J}$. Finally, select $t \in \mathcal{J}$, and set $C_T \leftarrow C_T^t$.
- $\pi_Q, y \leftarrow \text{ProveQuery}(\mathbf{pk}, \mathcal{DB}, Q)$: Generate a proof showing that $y = \text{ANSWER}(\mathcal{DB}, Q)$.
 - (1) Parse Q decomposing it in basic queries $\{Q_1, \dots, Q_l\}$. For every Q_j work as follows:
 - (a) If Q_j is not a selection query, or its input table does not have an index then generate a SNARK proof for the relation $\mathcal{R}_{Q_j} = \{(C_{T'_j}, \{C_{T_i}\}_{i \in \mathcal{I}}); (\mathbf{T}'_j, \{\mathbf{T}_i\}_{i \in \mathcal{I}}, \mathbf{w}) : C_{Q_j}(\mathbf{T}'_j, \{\mathbf{T}_i\}_{i \in \mathcal{I}}, \mathbf{w}) = 1 \wedge C_{T'_j} = \text{PCS.Commit}(\mathbf{pk}, \mathbf{T}'_j) \wedge C_{T_i} = \text{PCS.Commit}(\mathbf{pk}, \mathbf{T}_i), \forall i \in \mathcal{J}\}$, where $\mathcal{I}$ is the set of indexes of all tables involved in that query, $\mathbf{T}'_j$ the output table produced by the query and C_{Q_j} is a circuit that returns 1 if and only if $\mathbf{T}'_j$ has been computed correctly.
 - (b) If Q_j is selection query and an index for its predicate (denoted with addr) exists, the prover first uses the index $\mathcal{T}_{\text{addr}}$ to compute the sub-matrix $\tilde{\mathbf{T}}_j \subseteq \mathbf{T}_j^{\text{idx}}$ that contains all tuples satisfying the query predicate expression and use it to compute the output table $\mathbf{T}'_j$. Then, using the *Lookup* algorithm of ROGUE, generates a SNARK proof for the relation $\mathcal{R}_{Q_j} = \{(C_{T'_j}, C_{\tilde{\mathbf{T}}_j}, C_{\mathbf{T}_j^{\text{idx}}}); (\mathbf{T}'_j, \tilde{\mathbf{T}}_j, \mathbf{T}_j^{\text{idx}}, \mathbf{w}) : C_s(\tilde{\mathbf{T}}_j, \mathbf{T}'_j, \mathbf{w}) = 1 \wedge \tilde{\mathbf{T}}_j \subseteq \mathbf{T}_j^{\text{idx}} \wedge C_{T'_j} = \text{Kopis.Commit}(\mathbf{pk}, \mathbf{T}'_j) \wedge C_{\tilde{\mathbf{T}}_j} = \text{Kopis.Commit}(\mathbf{pk}, \tilde{\mathbf{T}}_j)\}$, where C_s is the circuit as defined in Appendix D.3.
 - (2) Set and output $\pi = \{(\pi_1, C_{T'_1}), \dots, (\pi_l, C_{T'_l})\}$.
- $0, 1 \leftarrow \text{VerifyQuery}(\mathbf{vk}, C_{\mathcal{DB}}, Q, \pi, y)$: Parse the query Q into the corresponding sub-queries. Likewise parse π as proofs of these sub-queries, validate their correctness using $C_{\mathcal{DB}}$. Also check consistency between commitments $(C_{T'_1}, \dots, C_{T'_l})$ whenever their pre-images are used as input for another sub-query.
- $\pi_{\text{upd}}, C'_T, \{C_T^i\}_{i \in \mathcal{J}} \leftarrow \text{ProveUpdate}(\mathbf{pk}, \mathbf{T}, \{\mathcal{T}_i\}_{i \in \mathcal{J}}, Q)$: On an update query Q prove that $\mathbf{T}$ and all indexes have been properly updated.
 - (1) If Q is insertion (or deletion respectively) query first update the indexes $\{\mathcal{T}_i\}_{i \in \mathcal{J}}$ to $\{\mathcal{T}'_i\}_{i \in \mathcal{J}}$ by inserting all tuples in $\mathcal{D}$. Then, for each $i \in \mathcal{J}$, use $\mathcal{T}'_i$ to retrieve the matrix $\mathbf{N}$ consisting of all leaf nodes that are going to be updated and $\mathbf{L}, \mathbf{R}$ consisting of the left and right neighbor nodes of $\mathbf{N}$ that do not belong to $\mathbf{N}$. Let $\mathcal{I}$ be the index set of these nodes. Finally, let $\mathbf{N}'$ be the set of all updated nodes retrieved by $\mathcal{T}'_i$ and $\mathcal{J}$ their corresponding indexes. Provided $(\mathbf{N}, \mathbf{L}, \mathbf{B}, \mathbf{N}', \mathcal{I}, \mathcal{J})$ and $\mathcal{D}$ as an additional witness, the prover uses the *Update* algorithm of ROGUE, to generate a proof for the correct update of each index using as C_U the circuit presented in the main body and Appendix D.3.
 - (2) If Q is an update query then perform the following steps:
 - (a) Decompose Q into two parts: Q^1 is the query that outputs all tuples to be updated and Q^2 is the query performs the desired updates. Use Q^1 to compute $\mathbf{T}_u$ and then apply Q^2 on input $\mathbf{T}_u$ to compute the updated tuples $\mathbf{T}'_u$. Then, generate generate two SNARK proofs π_1, π_2 for the relations $\mathcal{R}_1 = \{(C_{\mathbf{T}_u}, C_T); (\mathbf{T}_u, \mathbf{T}) : C_{Q^1}(\mathbf{T}_u, \mathbf{T}) = 1 \wedge C_{\mathbf{T}_u} = \text{Kopis.Commit}(\mathbf{pk}, \mathbf{T}_u) \wedge C_T = \text{Kopis.Commit}(\mathbf{pk}, \mathbf{T})\}$ and $\mathcal{R}_2 = \{(C_{\mathbf{T}_u}, C_{\mathbf{T}'_u}); (\mathbf{T}_u, \mathbf{T}'_u) : C_{Q^2}(\mathbf{T}_u, \mathbf{T}'_u) = 1 \wedge C_{\mathbf{T}_u} = \text{Kopis.Commit}(\mathbf{pk}, \mathbf{T}_u) \wedge C_{\mathbf{T}'_u} = \text{Kopis.Commit}(\mathbf{pk}, \mathbf{T}'_u)\}$.
 - (b) If $\mathcal{J} = \emptyset$, then compute the updated table $\mathbf{T}'$ by replacing all the tuples $\mathbf{T}_u$ of $\mathbf{T}$ with $\mathbf{T}'_u$ and compute its Kopis commitment. Finally generate a SNARK proof π_3 showing that $\mathbf{T}' = (\mathbf{T} \cap \mathbf{T}_u) \cup \mathbf{T}'_u$ using the techniques of [75]. Return $\pi_{\text{upd}} \leftarrow \{\pi_1, \pi_2, \pi_3\}$.
 - (c) If $\mathcal{J} \neq \emptyset$ work as follows:
 - (i) For all indexes on attributes $i \in \mathcal{J}_1$ that have been changed by the update query generate two proofs $\pi_{i,1}^3$ and $\pi_{i,2}^3$. The first proof shows that all tuples in the pre-image of $C_{\mathbf{T}_u}$ have been correctly removed from $\mathcal{T}_i$ (leading to a temporary index $\mathcal{T}_{i,\text{temp}}$). The second proof shows that all tuples in the pre-image of $C_{\mathbf{T}'_u}$ have been correctly inserted to $\mathcal{T}_{i,\text{temp}}$ (leading to the final index $\mathcal{T}'_i$). For both we use the insertion and deletion algorithms of step (1). Finally set $\pi_i^3 \leftarrow \{\pi_{i,1}^3, \pi_{i,2}^3\}$.
 - (ii) For all indexes on attributes $i \in \mathcal{J} \setminus \mathcal{J}_1$ generate a proof π_i^3 by invoking the *Update* algorithm of ROGUE over the circuit C_U as described in the last paragraph of Appendix D.3.
 - (iii) Return $\pi_{\text{upd}} \leftarrow \{\pi_1, \pi_2, \{\pi_i^3\}_{i \in \mathcal{J}}\}$ along with the new index commitments. Finally, replace the old index commitments with the new ones.
 - $0, 1 \leftarrow \text{VerifyUpdate}(\mathbf{vk}, C_T, C'_T, \{C_T^i\}_{i \in \mathcal{J}}, \{C_{T'}^i\}_{i \in \mathcal{J}}, \mathcal{J}, Q, \pi_{\text{upd}})$: If $\mathcal{J} = \emptyset$, validate π_1, π_2, π_3 using the SNARK verification algorithm. If all proofs are valid return 1 and set $C_T \leftarrow C'_T$. If $\mathcal{J} \neq \emptyset$, use the SNARK verification algorithm to validate π_1, π_2 and the *VerifyUpdate* of ROGUE to check the correctness of $\{\pi_i^3\}_{i \in \mathcal{J}}$. If all verifications pass, return 1, replace all old index commitments with the new ones, select $t \in \mathcal{J}$ and set $C_T \leftarrow C_T^t$.

$O(n_1m_1 + n_2m_2 + n_3m_3)$ while the proof size and verification time will be $O(\log^2(\max(n_i m_i)))$.

ProveUpdate. From Theorem C.1 and from the circuit sizes of update queries described in Appendix D.3, we obtain a prover complexity of $O(nKm \log(n/K) + |C_{\mathcal{U}}|) = O(nKm \log(n/K) + nKm) = O(nKm \log(n/K))$ for inserting (or deleting) n tuples on a single index while the proof size and verification time are $O(\log^2(nKm))$. If the table has $|\mathcal{J}|$ indexes, then the total prover complexity, proof size and verification time become $O(|\mathcal{J}|Km \log(n/K))$, $O(|\mathcal{J}| \log^2(kKm))$ and $O(|\mathcal{J}| \log^2(kKm))$ respectively. Note that the same complexities hold for proving the correctness of expressive update queries, but also we need to include the prover complexity for proving the correct computation of the tables $\mathbf{T}_u, \mathbf{T}'_u$.

D.4.2 Security Proofs. Because completeness comes directly from the completeness of our updatable matrix lookup argument and the completeness of SNARKs based on GKR, we will focus on proving (1) table and index extractability (2) database extractability and (3) update extractability.

Proof of Table & Index Extractability. Given any PPT adversary $\mathcal{A}$ against the Table & Index Extractability game, we will show how to construct a PPT extractor $\mathcal{E}$ that extracts $\mathbf{T}, \{\mathbf{T}_i^{idx}\}_{i \in \mathcal{J}}$ such that $1 = \text{Valid}(\mathbf{T}, \mathbf{T}_i^{idx})$, for every $i \in \mathcal{J}$. For simplicity, we will focus on a single index $\mathbf{T}^{idx}$.

With that in mind, our extractor first constructs an adversary against the knowledge soundness game of the underlying SNARK scheme and invokes its extractor to obtain $\mathbf{T}, \mathbf{T}_{idx}, \mathbf{w}$ such that $C_{idx}(\mathbf{T}, \mathbf{T}_{idx}, \mathbf{w}) = 1$ with overwhelming probability. What remains is to show that $C_{idx}(\mathbf{T}, \mathbf{T}_{idx}, \mathbf{w}) = 1$ if and only if $\text{Valid}(\mathbf{T}, \mathbf{T}^{idx}) = 1$, namely that $\mathbf{T}^{idx}$ represents a sorted linked list and that it contains all tuples of $\mathbf{T}$. The second one, is ensured by the check $\{\mathbf{T}^{idx}[j].\mathbf{k}[i]\}_{i,j \in [K] \times |\mathbf{T}^{idx}|} = \mathbf{T}$ done by C_{idx} .

For the first one, observe that check (ii) (as defined in the main body) ensures that all nodes of $\mathbf{T}^{idx}$ are sorted. Furthermore, check (iii) directly guarantees that every node points to another node and that the identifiers of every node are unique. In addition, the set equality $(+\infty, +\infty) \cup \mathcal{N}_{curr} \neq \mathcal{N}_{next} \cup (0, 0)$ ensures that two nodes cannot point to the same node because. That is because if this was the case, then $\mathcal{N}_{next}$ would contain two pairs of the form $(id, \cdot)$ while $\mathcal{N}_{curr}$, due to the uniqueness of the identifiers, only one and thus the check done by C_{idx} would have to fail. Consequently, because every node points to another one, and a node is pointed only once $\mathbf{T}^{idx}$ represents a linked list. Finally, to argue that $\mathbf{T}^{idx}$ is a sorted list, we need to ensure that each node points to its direct neighbor.

First, observe that by check $(+\infty, +\infty) \cup \mathcal{N}_{curr} \neq \mathcal{N}_{next} \cup (0, 0)$ we know that each pair (ptr, k_{min}) must be equal to a pair $(id, \mathbf{t}[0])$. Hence, due to the uniqueness of the identifiers we know that every k_{min} corresponds to the minimum value of an existing node. Based on this, due to check (i) (but also (ii)) C_{idx} guarantees that a node points to another node with larger key values. Lastly, because $\mathbf{T}^{idx}$ is a linked list, it should hold that every node points to his direct neighbor since otherwise there would be a case where test (i) would fail.

Proof of Database Extractability. Given any PPT adversary against the database extractability game, we will construct an extractor $\mathcal{E}$ that extracts all tables in $\mathcal{DB}$, such that $y = \text{ANSWER}(Q, \mathcal{D})$ with overwhelming probability. In particular, for each sub-query Q_j , $\mathcal{E}$ will construct an adversary against the knowledge soundness of the SNARK scheme or vector linkability of our matrix lookup argument, and then invoke the corresponding extractor to retrieve all involved tables $\mathbf{T}'_j, \{\mathbf{T}_i\}_{i \in \mathcal{I}}, \mathbf{w}$ such that $C_{Q_j}(\mathbf{T}'_j, \{\mathbf{T}_i\}_{i \in \mathcal{I}}, \mathbf{w}) = 1$ with overwhelming probability. Note that extracted output tables of a query that serve as input tables of another query should be the same with overwhelming probability due to the binding property of the underlying PCS.

For the remainder of this section, we will focus on showing that our circuit for joins and index supported selection queries output one if and only if the output table has been computed correctly. For the selection query, observe that because (i) the committed index represents a sorted linked list, (ii) the circuit validates that it reads a sub-list of that list and (iii) it checks if edge cases do not satisfy the query predicate, then it is guaranteed that all tuples that satisfy the predicate of the query must lie inside that sub-list.

For the circuit that validates the correctness of joins, we will specifically focus on the non-primary-foreign key joins since we can directly use the same arguments for the other case. With that in mind, we first show that the underlying circuit does not exclude any tuple that satisfies the predicate. This is ensured by checks $\mathbf{T}_1^{Attr_1} \cap \mathbf{T}_2^{Attr_2} = \emptyset$ and $\mathbf{T}_1^{Attr_1} \cap \mathbf{T}_2^{Attr_2} = \emptyset$. The first one guarantees that a malicious prover does not include a tuple with the same key value in the two different tables while the second one ensures that every tuple of $\mathbf{T}_1$ has a different predicate value from $\mathbf{T}_2$. Next, enforcing that $\mathbf{T}_3^i = \mathbf{T}_1^i$, we know that $\mathbf{T}_3^i$ contains only tuples that belong to $\mathbf{T}_1^i$ and by computing the histograms of the three tables we know that $\mathbf{T}_3$ contains the correct number of tuples that satisfy a specific predicate value. By ensuring that all tuples are distinct, then it should hold that these tuples were formed by the cross product of tuples from both tables with the same predicate values.

Update Extractability In this section, we will prove update extractability. In particular, we will show how to construct an extractor $\mathcal{E}$ from any adversary against the update extractability game that extracts old and new tables/indexes such that (i) $\mathbf{T} = \text{UPDATE}(\mathbf{T}, Q)$ and (ii) $1 = \text{Valid}(\mathbf{T}, \mathbf{T}_i^{idx}), 1 = \text{Valid}(\mathbf{T}', \mathbf{T}_i^{idx'})$, for all $i \in \mathcal{J}$. For simplicity, we will focus on the case where we have a single index and an insertion query. We stress however that we can follow the same techniques to prove update extractability when having more than one indexes and expressive update queries.

Initially, the prover constructs two adversaries, one for the Table & Index extractability game and one for the update consistency game of our matrix lookup argument, and invokes their corresponding extractors. From them it extracts the old and new indexes $\mathbf{T}^{idx}, \mathbf{T}^{idx'}$ and from them it also obtains the tables $\mathbf{T}, \mathbf{T}'$. What remains is to show that conditions of (i),(ii) hold with overwhelming probability.

Due to the Table & Index extractability of our scheme, it holds that $1 = \text{Valid}(\mathbf{T}, \mathbf{T}^{idx}), 1 = C_{\mathcal{U}}(\mathbf{T}_I^{idx}, \mathbf{T}_I^{idx'}, \mathbf{w})$ and $\mathbf{T}^{idx}[i] = \mathbf{T}^{idx'}[i] \forall i \in [n/K] \setminus \mathcal{J}$ with overwhelming probability. To complete the proof we need to show that $\mathbf{T}' = \text{UPDATE}(\mathbf{T}, Q) = \mathbf{T} \cup \mathcal{D}$ and

that $1 = \text{Valid}(\mathbf{T}', \mathbf{T}_i^{\text{idx}'})$. Observe that since $1 = C_{\mathcal{U}}(\mathbf{T}_{|I}^{\text{idx}}, \mathbf{T}_{|I}^{\text{idx}'}, \mathbf{w})$, it will hold that $\mathbf{T}_{|I}^{\text{idx}'} = \mathbf{T}_{|I}^{\text{idx}} \cup \mathcal{D}$. Since all other nodes of $\mathbf{T}^{\text{idx}}, \mathbf{T}^{\text{idx}'}$ remain unchanged, we get that $\mathbf{T}^{\text{idx}'} = \mathbf{T}_{|[n/K] \setminus I}^{\text{idx}} \cup \mathbf{T}_{|I}^{\text{idx}'} = \mathbf{T}_{|[n/K] \setminus I}^{\text{idx}} \cup (\mathbf{T}_{|I}^{\text{idx}} \cup \mathcal{D}) = \mathbf{T}^{\text{idx}} \cup \mathcal{D}$. Hence $\mathbf{T}' = \text{UPDATE}(\mathbf{T}, \mathcal{Q})$. As for $1 = \text{Valid}(\mathbf{T}', \mathbf{T}_i^{\text{idx}'})$, recall that $C_{\mathcal{U}}$ guarantees that every every sublist formed by $\mathbf{T}_{|I}^{\text{idx}}$ remains sorted in $\mathbf{T}^{\text{idx}'}$. Because all other nodes remain unchanged, it will hold that $\mathbf{T}^{\text{idx}'}$ will remain sorted and hence valid.

E Details on Experimental Evaluation

E.1 Details on TPC-H & Description of Queries

The TPC-H benchmark consists of eight tables. These are PART, consisting of 9 attributes, SUPPLIER consisting of 7 attributes, PARTSUPP consisting of 5 attributes, CUSTOMER consisting of 8 attributes, ORDERS consisting of 9 attributes, LINEITEM consisting of 16 attributes and NATION and REGION consisting of 4 and 3 attributes respectively. To generate a TPC-H database we need to use a scaling factor sf which increases the size of the tables by $sf \cdot 200,000$ for PART, $sf \cdot 10,000$ for SUPPLIER, $sf \cdot 800,000$ for PARTSUPP, $sf \cdot 150,000$ for CUSTOMER, $sf \cdot 1,500,000$ for ORDERS and $sf \cdot 8,000,000$ for LINEITEM. Note that the sizes of NATION and REGION remain the same and are equal to 25 and 5 respectively.

To benchmark our range query we use the query described in Figure 4. As for the benchmarks regarding updates, we use an expressive update query as described in Figure 5. Finally, the TPC-H queries #3,#5,#6,#15 and #19 are provided in Figures 6, 7, 8, 9, 10.

```
SELECT *,
FROM LINEITEM
WHERE l_shipdate > '[DATE1]'
AND l_shipdate < '[DATE1]'
```

Figure 4: Query used for the selection benchmarks.

```
UPDATE LINEITEM,
SET l_tax = l_tax + 0.01
WHERE l_shipdate > '[DATE1]'
AND l_shipdate < '[DATE1]'
```

Figure 5: Query used for the update benchmarks.

```
SELECT l_orderkey,
      sum(l_extendedprice*(1-l_discount)) as revenue,
      o_orderdate, o_shippriority
FROM CUSTOMER, ORDERS, LINEITEM
WHERE
c_mktsegment = '[SEGMENT]'
and c_custkey = o_custkey
and l_orderkey = o_orderkey
and o_orderdate < date '[DATE]'
and l_shipdate > date '[DATE]'
GROUP BY
l_orderkey,
o_orderdate,
o_shippriority
ORDER BY
```

```
revenue desc,
o_orderdate;
```

Figure 6: Query #3 of the TPC-H benchmark.

```
SELECT n_name, SUM(l_extendedprice*(1-l_discount))
AS revenue
FROM COSTUMER, ORDERS, LINEITEM, SUPPLIER, NATION, REGION
WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey
AND l_suppkey = s_suppkey AND c_nationkey = n_nationkey
AND n_regionkey = r_regionkey AND r_name = '[REGION]'
AND o_orderdate >= '[DATE1]'
AND o_orderdate < '[DATE2]'
GROUP BY n_name
ORDER BY (revenue) DESC;
```

Figure 7: Query #5 of the TPC-H benchmark.

```
SELECT sum(l_extendedprice*l_discount) as revenue
FROM LINEITEM
WHERE
l_shipdate >= date '[DATE1]'
AND l_shipdate < date '[DATE2]'
AND l_discount between [DISCOUNT] - 0.01
and [DISCOUNT] + 0.01
AND l_quantity < [QUANTITY];
```

Figure 8: Query #6 of the TPC-H benchmark.

```
CREATE view REVENIEW[STREAM_ID] (supplier_no,
total_revenue) as
SELECT l_suppkey, sum(l_extendedprice * (1 - l_discount))
FROM LINEITEM
WHERE l_shipdate >= date '[DATE]'
AND l_shipdate < date '[DATE1]'
AND l_shipdate > '[DATE2]'
GROUP BY l_suppkey;

SELECT s_suppkey,s_name,s_address,s_phone,total_revenue
FROM SUPPLIER, REVENIEW[STREAM_ID]
WHERE s_suppkey = supplier_no
AND total_revenue = (
SELECT max(total_revenue)
FROM REVENIEW[STREAM_ID])
ORDER BY s_suppkey;
```

Figure 9: Query #15 of the TPC-H benchmark.

```
SELECT
sum(l_extendedprice * (1 - l_discount) ) as revenue
FROM LINEITEM, PART
WHERE ( p_partkey = l_partkey
AND p_brand = '[BRAND1]'
AND p_container in ('SM CASE','SM BOX','SM PACK','SM PKG')
AND l_quantity >= [QUANTITY1]
and l_quantity <= [QUANTITY1] + 1
AND p_size between 1 and 5
AND l_shipmode in ('AIR','AIR REG')
AND l_shipinstruct = 'DELIVER IN PERSON')
OR
(p_partkey = l_partkey
AND p_brand = '[BRAND2]'
AND p_container in
('MED BAG','MED BOX','MED PKG','MED PACK')
AND l_quantity >= [QUANTITY2]
and l_quantity <= [QUANTITY2] + 1
AND p_size between 1 and 10
AND l_shipmode in ('AIR','AIR REG')
AND l_shipinstruct = 'DELIVER IN PERSON')
OR
```

```
(p_partkey = l_partkey
AND p_brand = '[BRAND3]'
AND p_container in
    ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
AND l_quantity >= [QUANTITY3]
    and l_quantity <= [QUANTITY3] + 1
AND p_size between 1 and 15
AND l_shipmode in ('AIR', 'AIR REG')
AND l_shipinstruct = 'DELIVER IN PERSON');
```

Figure 10: Query #19 of the TPC-H benchmark.