qedb: Expressive and Modular Verifiable Databases (without SNARKs)

Vincenzo Botta[1], Simone Bottoni[1], Matteo Campanelli[3], Emanuele Ragnoli[1], and Alberto Trombetta[1]

[1] Provably Technologies
[2] Offchain Labs

Abstract. Verifiable Databases (VDBs) let clients delegate storage to an untrusted provider while maintaining the ability to verify query results. Since databases are foundational and storage delegation is increasingly common, VDBs address a critical need. Existing VDB designs face several limitations: approaches based on general-purpose proof systems (e.g., SNARKs) offer high expressivity but at the cost of cumbersome intermediate representations, heuristic assumptions, and heavy cryptographic machinery, whereas schemes built from specialized authenticated data structures (ADS), such as accumulators, are simpler and rely on well-founded assumptions, yet they support only restricted queries and often incur unacceptable overheads (e.g., a quadratic overhead in the number of the database columns).

We present qedb, a new construction that advances the state of the art on verifiable databases from ADS:it is the most expressive scheme to date and the first with proof length completely independent of the database size; it has no quadratic dependence on the number of columns. qedb is deployment-oriented: it is performant and simple to implement and analyze; it relies on well-founded assumptions; it can be easily made post-quantum secure (using lattice-based instantiations).

One of our primary contribution is a modular, foundational framework separating the information-theoretic logic of a VDB from its cryptographic instantiations. We show how to realize it using pairing-based set accumulators and linear-map vector commitments (which we introduces as a technique), and more generally show that extractable homomorphic polynomial commitments suffice.

Our Rust implementation of qedb scales to DBs with millions of rows. Its proving and verification times are competitive against SNARK-based constructions and improve on ADS-based solutions.

Table of Contents

1 Introduction

Databases are one of the foundations of modern technological infrastructure. They underpin financial systems, enable researchers to analyze scientific data, support social platforms in storing user activity, and allow governments to manage critical citizen records. Given their central role, guaranteeing their integrity features is fundamental. However, organizations increasingly outsource database management to third parties, entrusting them not only with the task of storing data but also with that of executing queries and providing operational support. This trend gives rise to a fundamental security challenge: *how can an organization verify that outsourced queries are executed correctly without having to recompute them locally?*

Verifiable Databases (VDBs) [69] address this challenge by enabling clients to verify both data integrity and query correctness without trusting the database provider. A client can hold a short cryptographic digests encoding the dataset. Whenever it receives a query and a claimed response, it will also receive a proof certifying its correctness. The client can then use this proof and the digest to trust the response without having to access the entire database.

Ideal features of verifiable databases. For VDBs to be practical and widely applicable, they should ideally satisfy some efficiency properties, as well as some security-related ones. We now zoom in on some of the features that will be the focus of this work, at the same time motivating them.

First, verification in a VDB should be efficient, i.e., consuming significantly fewer resources than re-executing queries. For wider applicability, it should also be *public* (not limited to predefined verifiers or requiring secret keys). Finally, it is desirable that they are *non-interactive*: after sending a string certifying the result of a query (a proof), the prover is not required to be online for verification. Lack of interaction reduces latency, simplifies composition with other protocols, and in some applications is essentially required (e.g., smart contracts, where interaction would both more expensive and substantially impact latency).

Any deployed VDB, naturally, should be expected to be secure. For this, solid cryptographic assumptions are essential. Yet, they alone are not a guarantee of real-world security. An additional desirable feature of a trustworthy VDB construction, we argue, is that it should be as simple as possible. In software, simpler designs tend to be less vulnerability-prone, easier to audit, and more maintainable [29, 74]. Yet, many efficient VDB constructions—both in prior literature and industrial efforts—come with considerable complexity, which can undermine these very goals. (We expand in Section 1.1, Section 3 and Section 4)

This work in a nutshell. In this work we show it is possible to design *highly efficient* and *expressive* VBDs (supporting a large subset of SQL) from *simple* building blocks. To this end, we introduce a new construction, qedb, which addresses several limitations of prior approaches. Along the way, we develop novel techniques and establish a solid theoretical foundation for the design of VDBs.

1.1 The current landscape of VDB designs and their limitations

Before presenting our results, we give a brief overview of the existing designs in the VDB literature (which we expand in Section 3). They all fall into two categories[3]. The first is based on *general-purpose* proof systems and, in particular, SNARKs [6]. Informally, a SNARK—*succinct non-interactive arguments of knowledge*—is a proof system that computes a certificate π that guarantees the result of a computation $f(x)$. One of their most important features is that while the computation can be very long, the certificate π can be very short. What makes general-purpose approaches attractive is their natural support for expressive queries. However, this often requires representing the database logic in the form of cumbersome intermediate repre-

[3] In this work, and in these paragraphs, we focus on *succinct* VDBs where both the client's running time and the proof size are sublinear in the size of the database (ideally *independent* of it). In this paper we are not interested in hiding properties and we leave zero-knowledge as an interesting future work.

sentations (such as constraint systems[4]). This approach adds unnecessary complexity, increases the likelihood of bugs, and results in a less accessible developer experience [73] (see also additional limitations in Section 4). Moreover, in order to achieve practical efficiency on the prover side, these solutions often need to rely on recursion [4, 55] (where, informally, a SNARK proves the valid verification of another SNARK proof). This has implications for the security of the system, which must now rely on *heuristic assumptions*[5]. Indeed, the cryptographic community has been suspicious of these heuristics for a while, and recent works exposed concrete ways to attack them [53]. Finally, general-purpose tools may be a "sledgehammer" approach to what is arguably a very structured application setting.

The second category of prior works on VDBs employs instead *specialized* Authenticated Data Structures [58, 61, 71, 78, 86] (which we will often refer to simply as ADS). For an intuition the reader can think of these as a more limited type of proof systems (e.g., specializing in proving exclusively set membership, or interval queries, or inner product, etc.). On the plus side, these approaches tend to have simpler designs, which makes them easy to implement, analyze and audit. They also substantially mitigate some of the common drawbacks of general-purpose approaches, by relying on more well-founded security assumptions and not requiring an explicit intermediate representation through constraint systems. These features make ADS-based schemes excellent candidates for solutions with strong security guarantees and that are easy to deploy. Unfortunately, though, the overall efficiency profile of the current state of the art (IntegriDB [85]) substantially limits their applicability. In general, its proving time is worse than other SNARK-based solutions such as vSQL [84](albeit of the same order of magnitude). While IntegriDB does compensate for this with better verification time and proof size on some specific queries, in order to support JOINs, it introduces a specific runtime and storage overhead that is a dealbreaker in many real-world settings; in particular, its preprocessing depends quadratically on the number of columns in the DB tables[6]. Finally IntegriDB—and all prior ADS-based solutions—features a proof size that depends on the database size (which could potentially be huge).

Current ADS-based design also suffer from an additional limitation, one that is more conceptual in nature: since there is no clear unifying theory common to all these protocols, it is not obvious for researchers how to extend or improve them (e.g., by updating some of their building blocks). In particular, all existing blueprints provide little to no modularity (a feature that the cryptographic proof community has recognized as essential [50]). Rather, they are offered as a "package deal" [50] which couples the ideas behind the constructions and the cryptographic building blocks in it. The result is a protocol of the form "*employ ADS X here and employ ADS Y there; combine them in such a way here*". This makes it unclear to see the implications of changing a building block for another. In such cases, the security and expressivity of the final scheme may need to be proved almost from scratch.

1.2 Our results

We propose new frameworks and designs based on ADS that improve on the state of the art of VDBs, addressing all of the limitations above. Our results include:

- **New techniques** for verifiable databases based on authenticated data structures. We show how to combine vector commitments with inner product features and accumulators to represent queries in a modular way and to efficiently prove their correctness over large databases. Our new techniques play a crucial role in obtaining the efficiency features of our final construction (see next item). Specifically, it is crucial to remove the quadratic dependency on

[4] A constraint system is a way to encode a computation, usually for the purpose of proving it through a cryptographic argument. For simplicity the reader can think of a *circuit representation* whenever we use the phrase *constraint system*.

[5] The heuristic assumptions related to recursion often stem from two (independent) aspects: treating the random oracle as part of the circuit; applying recursion for depths for which we have no formal guarantee of security. See also [16, 53].

[6] It is not uncommon to have 10–15 columns in some tables in practice (which yields an overhead of $\approx$ 100–225$\times$). Another point of reference is the standard benchmarks for database performance, TPC-H, which includes tables containing 62 columns [32].)

Table 1: Comparison of expressive and succinct verifiable databases constructions. We compare qualitative features in the top tables and efficiency metrics in the bottom table. We compare to general-purpose SNARKs in Section 3, in Section 4 and in Appendix A.

Scheme	Setup	Core Building Blocks	Decouples Logic & Instantiations?	Expressivity
IntegriDB [85]	powers of τ	Merkle trees, pairing-based accumulators	✗	Fig. 1 (top part only)
This work	powers of τ	KZG	✓	Fig. 1 (all)

Scheme	Overhead in $\lvert\pi\rvert$, $V_{\mathbf{time}}$ (queries w/o JOINs)	Overhead in $\lvert\pi\rvert$, $V_{\mathbf{time}}$ (JOINs)	Preprocessing & server storage
IntegriDB [85]	$\log(\lvert\mathsf{column}\rvert)$	$\lvert\mathsf{resp}\rvert \cdot \log \lvert\mathsf{column}\rvert$	$\lvert\mathsf{db}\rvert + n_{\text{cols}}^2$
This work	$\lvert\mathsf{qry}\rvert$	$\lvert\mathsf{resp}\rvert$	$\lvert\mathsf{db}\rvert$

(NB: commonly $\lvert\mathsf{resp}\rvert \ll \lvert\mathsf{column}\rvert \ll \lvert\mathsf{db}\rvert$; for aggregate queries, $\lvert\mathsf{qry}\rvert \approx \lvert\mathsf{resp}\rvert$, else $\lvert\mathsf{qry}\rvert \ll \lvert\mathsf{resp}\rvert$).

Other notes: Both schemes have a digest of constant size. The overhead for proof size and verification time below is an *additive* overhead in addition to the query qry and the response resp. For simplicity below we assume JOINs of two tables only. All quantities are implicitly asymptotic. n_{cols} denotes the maximum number of columns in a table.

number of the database columns in the preprocessing phase, resulting in a prohibitive storage overhead even for medium-sized databases.

- **A new VDB construction**, **qedb**[7], that supports a representative subset of SQL. Our construction improves on the state of the art of VDB designs based on authenticated data structures. In particular, qedb:
  - ▷ is the first scheme of this type achieving proof size *completely independent* of the database size;
  - ▷ is the most expressive scheme of its kind at the time of writing;
  - ▷ concretely pushes further the scalability of proving by supporting larger database, both in terms of number of rows and number of columns.

  Our scheme shows improvements over VDBs *from general-purpose proof systems* as well, compared to which it offers: competitive—and often superior—performance (see Table 5 and Appendix A); a *substantially* simpler architecture which employs only few basic building blocks (instead of a stack of complex protocols) and does not require writing any circuits; better security guarantees (by relying on fewer and better understood cryptographic assumptions). We also refer the reader to the discussion in Section 4.
- **The first abstract framework for VDB constructions**. Along the way to our construction, we provide new theoretical foundations for the design of VDBs: a new framework to model idealized (*information-theoretic*) protocols for verifiable databases. In contrast to all prior work, our approach is the first to decouple the *essential ideas* behind a construction from the *cryptographic tools* used to instantiate them. This conceptual angle brings VDBs closer to the established practice in the world of SNARKs, which separates the underlying information-theoretic blueprint from its cryptographic compilation (see [1,5,7,11,17,28,42]). We stress that while our idealized model for VDB is close in spirit to those in the SNARK world—where the idealized objects sent around are *oracles* to *polynomials*—our framework is intentionally specialized to the VDB setting and requires inventing a completely new formalism—which employs vectors and sets as first-class citizens, instead of polynomials.
- **An implementation**[8] **and experimental evaluation** of qedb. Our results confirms the practicality of our scheme and its concrete improvements over prior ADS-based constructions. Our—still very unoptimized—implementation shows that our prover easily scales to

[7] qedb is a recursive acronym standing for "qedb error-checks databases". It is also a shameless pun on it being a *proof* system for DBs.

[8] We are planning to soon make it available as open source.

Queries supported by both this work and IntegriDB:

Predicate types: Multi-dim. range queries, list membership, any AND/OR.

Aggregate queries: MAX, MIN, COUNT, SUM, AVG.

JOINs: Equality-based joins over columns, possibly with duplicates.

Queries supported by this work but not by IntegriDB:

Comparison between columns: Predicates involving more than one column (e.g. "[...] WHERE $c_1 \geq 2c_2 + c_3$").

Aggregation among columns: Expressions involving more than one column in the SELECT clause (e.g. "SELECT $c_1 + 2c_2$ FROM [...]").

Fig. 1: Query types supported in IntegriDB (top) vs our work (top and bottom).

datasets containing millions of rows for many types of queries and that our verifier is very fast independently of the query type. Our verification runs below 25ms on aggregation queries (e.g., `SUM` or `MAX`) and selection queries with a response of $\approx$ 1K rows. On selection queries with very large outputs (tens of thousands of rows), verification runs in only 100ms. The ballpark for our proof size on most queries is 1KB.

Additional impact and features of our work.

- *A stepping stone for practitioners.* Relying on a minimal tech stack and an already deployed trusted setups (a powers-of τ-setup for KZG has for example been produced by the Ethereum Foundation [37]), qedb makes verifiable database more accessible. Its implementation can be easily integrated with off-the-shelf DBMSs (our evaluation uses PostgreSQL). Due to its modularity, practitioners can simply replace building blocks suiting their needs.
- *A stepping stone for future research.* Thanks to our framework, protocol designers can improve on our work without reanalyzing its security from scratch, but focusing on the right building blocks and the underlying approach.
- *Post-quantum security.* Our framework allows one to obtain a VDB with post-quantum security in a plug-and-play manner by simply replacing the appropriate primitives (see Remark 6).

We refer the reader to Section 3 for a discussion of interesting open problems stemming from this paper.

1.3 Motivating applications

Financial data reporting. Consider a financial firm that curates a database market transactions, upon which a third-party reporting system runs analytics queries on behalf of external clients, such as regulatory entities[9]. We provide a simple representative example of queries for this scenario supported by qedb in Fig. 2. The database assumed in the figure contains at least three tables `Account`, `Asset`, `Transaction` that store data about clients' accounts, traded assets and transactions between accounts involving such assets. The setting of financial data reporting showcases some of the features of qedb: in this scenario it crucial that such aggregate data reporting is trustworthy and compliant with regulation [38] [35]. Simpler constructions such as qedb are easier and cheaper to audit, and thus easier to demonstrate as compliant.

Data market aggregator. An untrusted data aggregator analyzes data from multiple data providers. The analyses take may take into account several metrics and can be accessed by data consumers, certifying the validity of Data Service Level Agreements, e.g. *all the records in the reporting data sets must have been updated within the prior 24 hours* [30, 67].

[9] As a concrete instance of this scenario: in the United States, firms such such as JPMorganChase and HSBC report annually to the US Securities and Exchange Commission regarding their financial conditions, regulatory capital, risk metrics and compliance status [51] [47].

Q_{Tot} || `SELECT SUM(price) FROM Transaction`
|| `WHERE account_id = '5938' AND trade_date = '2025-01-01'`

↳ Computes total price of transactions executed by an account on a given date

Q_{CntTx} || `SELECT COUNT(*) FROM Transaction`
|| `WHERE trade_date BETWEEN '2025-01-01' AND '2025-03-31'`

↳ Computes the number of transactions executed within the first quarter

Q_{MatchExp} || `SELECT tx_id, price, expected_price, price = expected_price`
|| `FROM Transaction WHERE trade_date = '2025-04-05'`

↳ Retrieves the transactions whose executed price equals their expected price

Fig. 2: Example queries for financial data reporting.

Blockchain oracles and blockchain analytics systems. A blockchain oracle [46] feeds data coming from off-chain sources to smart contracts operating on a blockchain. Assuming that data source are trusted, it is crucial that an oracle proves to the blockchain that data has not tampered with [68]. As an additional viewpoint on the last setting, consider a datastore containing a curated list of transactions executed on a blockchain. Clients are interested in executing analytics-based queries on such data. Here again, the client can be convinced of the validity of the responses without having to process the data themselves [39].

1.4 Outline

We will describe our techniques and general approach in Section 2. We discuss additional related literature and interesting future work in Section 3. Section 4 expands on the merits of the methodology behind qedb vs general-purpose and/or recursive cryptographic proof systems. In Section 5 we present the concrete performance of qedb. The bulk of the formal treatment in the paper is contained in the remainder of the document: Sections 6 and 7 are background sections, respectively on the type of ADS we will employ and on verifiable databases as a cryptographic primitive; Section 8 contains our new information-theoretic abstraction (idealized VDBs); in Section 9 we formalize our new property for vector commitments and accumulators (zero-testing on accumulated sets), show how to instantiate it and we finally describe our compiler; the bulk of Section 10 contains the idealized protocol behind qedb.

2 Technical Overview

A stepping stone: idealized protocols. Although a VDB protocols are cryptographic in nature, our approach starts designing such a scheme by focusing on the *essential* and *non-cryptographic* behavior at its core. The final result will be an astonishingly simple protocol with obvious correctness and soundness guarantees. This simplicity (together with the efficiency features stemming from it) constitutes one of the strengths of qedb compared to existing constructions in the space. To exemplify our approach, let us consider a simple query template such as this (we assume the reader is familiar with the basics of SQL):

|| `Q: SELECT C FROM T WHERE SomeCondition`

Above, `C` and `T` are respectively a column and a table and `SomeCondition` is some abstract property through which we want to filter `C` (e.g., it could stand for $\mathtt{C}' \geq 2$ where $\mathtt{C}'$ is some other column).

The output of such queries will be a tuple of rows in `C`. If the prover (the server) claims as output a tuple $\boldsymbol{y}$, what are the minimal checks for the verifier (the client) to have the guarantee that the result is correct? For example, it should check that the $\boldsymbol{y}$ is actually derived by reading `C` at some subset of indices X *and* that `SomeCondition` is satisfied at *every* index in X *and nowhere else*.

Clearly we want our verifier to be efficient. It should certainly not run in time linear in the database; ideally its overhead should be linear in the size of the response alone (which is the case for qedb). As a stepping stone in that direction, let us consider some "idealized" version of a VDB protocol where we assume that the prover and verifier can "magically" perform some steps in constant time. To explain this concept, we will jump directly to an example of a simple idealized protocol for the query above, which we will break down for the reader in the following paragraphs. Below, we assume that the prover can send to the verifier what we will call *handles* to vectors or sets. The reader can think of them as "immaterial pointers"—to vectors or sets—and whose size is constant, that is completely independent of the size of the object they refer to[10]. To distinguish them from the actual object they refer to, we denote a handle to a set X (resp. vector $\boldsymbol{v}$) as $\boxed{X}$ (resp. $\boxed{\boldsymbol{v}}$). These handles can be useful if the verifier is able to verify some properties on the underlying object. For instance, a basic test we will provide is a *read* check: through $\mathbf{read}?(\boldsymbol{u}, \boxed{X}, \boldsymbol{v})$, the verifier can check whether $\boldsymbol{v}$ is actually the subvector obtained by reading the set of indices X in $\boldsymbol{u}$, i.e., whether $\boldsymbol{u}_X = \boldsymbol{v}$[11]. Notice that the $\boldsymbol{v}$ does not need to be a handle in $\mathbf{read}?$; in contrast, $\boldsymbol{u}$ does (the verifier should not run in time $|\boldsymbol{u}|$ in order to read $\boldsymbol{v}$ since it could be the case that $|\boldsymbol{u}| \gg |\boldsymbol{v}|$). Using this basic vocabulary on handles, here is the sketch of an idealized protocol for query `Q`:

- During an offline stage, let the verifier hold a handle to `C`.
- After receiving the query, the prover provides the result $\boldsymbol{y}$ as well $\boxed{X}$, a handle to X (the set of indices to be read from `C` where `SomeCondition` holds).
- The verifier runs $\mathbf{read}?(\boldsymbol{u}, \boxed{X}, \boldsymbol{v})$.
- The verifier and the prover run a subprotocol (potentially in parallel) to check `SomeCondition` is satisfied at all and only the indices in $\boxed{X}$.

The correctness and soundness of the protocol above is immediate (under the assumption of the subprotocol for `SomeCondition` also being correct and sound). Before we discuss more in detail how to more generally adopt the approach above for more complex queries—and before we discuss how we can instantiate the above subprotocol for `SomeCondition` in the first place—we shall now clarify where idealized protocols fit into the larger picture of how to construct concrete VDB schemes.

Our framework for modular VDBs. We develop a framework to design modular VDBs from the following recipe (see also bottom half of Fig. 3):

(a) First, define an *idealized* protocol (fully information-theoretic) capturing the essence of your approach to query verification; then prove its security.
(b) Identify a set of cryptographic building blocks to instantiate the client's abstract operations in the idealized scheme from (a).
(c) Invoke one of our compilation results—e.g., Theorem 1—to obtain a concrete VDB protocol from (a) and (b) (automatically guaranteed to be secure).

Simple, yet powerful idealized protocols. We formalize a class of idealized protocols where the verifier supports a small set of abstract operations on handles. Through our construction, qedb we will show how this is sufficient to support a large class of SQL. We stress that this formalism can be used independently of our specific scheme: it can be used as it is for different constructions and/or tailored to change the set of supported operations (in the last case, the compilation theorems should be appropriately modified as well).

Our notion of idealized protocols is inspired from algebraic idealized protocols (e.g., Polynomial IOPs) employed in the SNARK literature, where the prover can send *oracle* polynomials and the verifier is endowed with an abstract *polynomial evaluation* operation on those oracles [1, 7, 11, 17, 28, 42]. Our notion departs from these formalisms in a few ways: our prover can

[10] The reader familiar with the SNARKs literature can think of them as *oracles*.

[11] Here we are using the common notation $\boldsymbol{u}_X := (u_{i_1}, \ldots, u_{i_\ell})$ whenever $X \subseteq [|\boldsymbol{u}|]$ and $X = \{i_1, \ldots, i_\ell\}$. We also remark that, for sake of clarity, we are presenting $\mathbf{read}?$ as a predicate and using a different notation from the one in our formal treatment, in two ways: a read will have a slightly special role there and will be applied as a function rather than a check; the set X will not be passed as a handle.

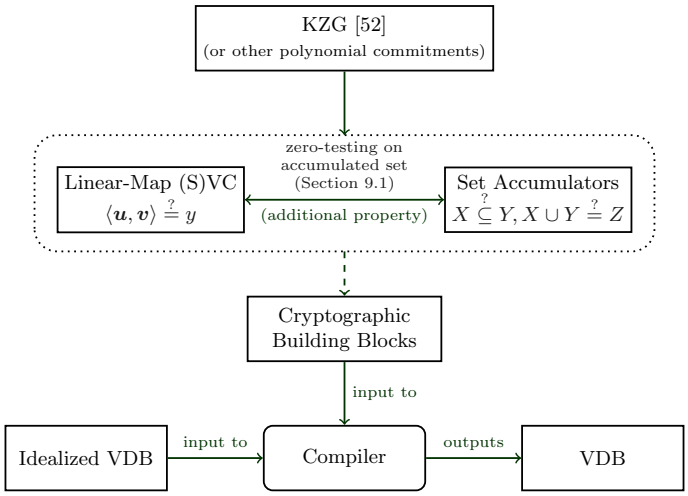

Fig. 3: Overview of our general results on VDBs through the framework we introduce and from common authenticated data structures. "(S)VC" stands for "Vector Commitment with Subvector opening". KZG can be replaced by other schemes (Theorems 6 and 7).

send different types of oracles (our handles) to both vectors ("slice" handles[12]) and sets (set handles); the abstract operations allowed to our verifier are heterogenous and may refer to more than one handle at once; for sake of simplicity our idealized protocol non-interactive, that is the prover is allowed to send a single batch of handles and then it remains silent (our final scheme is evidence that this is already sufficiently powerful). In addition to the handles sent by the prover, the verifier can also access a set of handles honestly generated during a preprocessing (indexing) stage of the database. Our final formalism—which we describe in full in Section 8—endows the verifier with the set of primitive operations below (to which we summarily refer to as *idealized interface*).

$$\boldsymbol{u} \stackrel{?}{=} \alpha \boldsymbol{v} + \boldsymbol{w} \quad \textit{(homomorphism)} \qquad \langle \boldsymbol{u}, \boldsymbol{v} \rangle \stackrel{?}{=} y \quad \textit{(inner product)}$$
$$X \stackrel{?}{\subseteq} Y \qquad Z \stackrel{?}{=} X \cup Y \qquad Z \stackrel{?}{=} X \cap Y \quad \textit{(set ops)}$$
$$\textbf{read}?(\boldsymbol{u}, X, \boldsymbol{v}) \ \textit{(read)} \qquad \boldsymbol{v}\left[X\right] \stackrel{?}{=} \boldsymbol{0} \quad \textit{(zero test)}$$

This interface is simple enough to be captured by relatively basic cryptographic primitives. At the same time, it is extremely expressive, permitting to describe range checks and other complex predicates (see Fig. 5 and Table 3). Later, when describing `qedb`, we will return to how this interface alone can capture complex queries. We will now discuss how an idealized protocol can be transformed into a full-fledged cryptographic VDB.

Our main compilation results. The reader familiar with authenticated data structures may have noticed that our idealized interface resembles (to an extent) the functionalities offered by *vector commitments* [22] and *set accumulators* [66]. Vector commitments allow one to commit to a vector of values and later open a subset J of positions via succinct proofs—a property known as *subvector opening* if the proofs are of length sublinear in $|J|$—while set accumulator allows one to compute a digest of a set of values and to succinctly prove set membership; occasionally (as well as in this work) accumulators can prove relations about accumulated sets succinctly, such as subset or intersection.

Standard accumulators and vectors commitments *almost* match the operations in the figure above. As we will soon explain, we can extend the functionalities of these cryptographic primitives (to which we hereby refer for brevity as ADS) to obtain an *exact* match the ones required in idealized scheme. First though, let us see how, with the right primitives under one's belt, there is an intuitive compilation strategy to transform an idealized VDB into a cryptographic one.

[12] We refer to them as "slices" because intuitively they are often used to point to "slices" of data, e.g. a column in a table.

Our compiler: consider an idealized protocol that captures verification for a set of queries, i.e., that is complete and sound on them (such as the sketched one at the beginning of this overview); for each set handle (resp. slice handle) sent by the idealized prover, let the cryptographic prover send a corresponding set accumulator (resp. vector commitment) during VDB execution; whenever the idealized verifier performs one of the abstract operations, the prover will send a related proof for it using the related ADS interface (on which we now expand).

Matching the idealized interface and new types of ADS. Let us look more closely at how we can use cryptographic ADS to handle all the idealized operations.

Instantiations. Given the discussion above, the final minimal requirements for our compiler are two primitives Π_{LVC} and Π_{Acc}, respectively a homomorphic LVC scheme (supporting our flavor of inner product test) and a set accumulator supporting set relations, and such that *together* they support zero-testing proofs. We show how to instantiate them in two ways—a more concrete and efficient instantiation and a more generic one (see also Fig. 3):

- *From pairings:* we show how the pairing-based LVC from [21] and set accumulators from [71] can, with a few modifications, match our interface. These two primitives are good candidates not only because of their efficient profile, but also from a technical standpoint in order to obtain our special properties: both primitives can be seen as committing (under the hood) to a vector/set through a KZG-style commitment [52], a popular polynomial commitment.
- *From any extractable homomorphic polynomial commitment:* we observe that the patterns that allow one to obtain our ADSs through KZG-style primitives are actually more general. Thus, at the price of a slightly less inefficient instantiation, we can show that a homomorphic polynomial commitment suffices to build our special primitives[13]. This result allows us to obtain instantiations from other assumptions (for example, post-quantum secure ones through lattices; see also Remark 6).

A concrete compilation theorem: In our formal sections, we show two compilations theorems (Theorem 6 and Theorem 7) which intuitively correspond to each of the instantiation classes described above. Our theorems automatically preserve not only *security* but also *succinctness* (a secure succinct idealized scheme yields a secure succinct VDB). For sake of concreteness below we state an informal compilation theorem, which can be seen a corollary of our general compilation theorem Theorem 6 and our instantiations from pairing.

Theorem 1 (informal). *Let Π_{ideal} be an idealized protocol for query family $\mathcal{Q}$. Then it is possible to compile Π_{ideal} into a VDB protocol Π_{VDB} for query family $\mathcal{Q}$ secure in the AGM where: (i) the client keeps a KZG commitment for each of the handles in Π_{ideal}[14]; (ii) the setup is the standard KZG setup.*

The AGM in the statement is only required for the pairing case; it is not required in our compilation theorem from polynomial commitments alone (Theorem 7). We also believe that it should be possible to remove the AGM altogether through a different analysis of our building block (left as future work).

Zooming in on our main construction. Our final construction is obtained by describing an idealized VDB in the sense above specialized to the SQL setting, applying our compiler and then finally some standard optimizations for concrete performance (e.g,, batching pairing checks). We now provide only a high-level view of its design (see Section 10 for details):

[13] The resulting scheme is slightly less efficient than the direct instantiation with pairings because pairing operations offer some shortcuts that are not available with an abstract polynomial commitment interface. In the latter case, we need to occasionally provide additional evaluation proofs and have the verifier provide random challenges (the resulting ADS schemes are secure in the random oracle model). Finally, we remark that, while it is a folklore result that one can obtain vector commitments and accumulators from polynomial commitments, our technical contribution is to show how *our new properties* can be obtained from polynomial commitments alone.

[14] NB: We then adopt a "meta-commitment" to all handles to obtain a $O(1)$ size digest.

- *The role of handles:* roughly (with exceptions) our construction mainly adopts the mappings: $\boldsymbol{v}$ ⇜⇝ [columns] and X ⇜⇝ [rows satisfying a predicate] [15]. This is a natural choice in SQL (and may vary slightly in other types of databases) because slices now offers us a natural way to filter rows out of columns, one of the the most common operations in databases.
- *Composition and invariants:* Leveraging the mapping above, *set* handles have the primary role of supporting easy *composition* of different parts of the queries: every time a `SELECT` or a `JOIN` query requires filtering a subset of the rows, we reuse the set handle to "reason" on them for additional claims (we did this in our sketched idealized protocol earlier by delegating the check of `SomeCondition` on X to a subprotocol).

We also adopt other important optimizations at preprocessing time, such as precomputing, for small domains, which rows in a column contain which value.

Other aspects and technical divergence from IntegriDB. Our compositional approach above stems from the delicate interplay of slice handles (vector commitments) and set handles (accumulators). Our introduction of vector commitments as main "receptacles for value" is one of the key differences from the approach in IntegriDB [85], which employs accumulators to represent values instead of vector commitments. This has two implications: handling duplicated values can be cumbersome in IntegriDB and cause a loss of succinctness; it also leads to the quadratic factor in their construction, since IntegriDB must "emulate" some type of vector opening by explicitly preprocessing how sets of values from one column are linked to those in others (thanks to our use of set as pointers to rows we do not run into this issue). Our employment of homomorphic vector commitments also allows us to apply different techniques for aggregation queries (`SUM`, `MAX`, `MIN`, `COUNT`) and to obtain a natural form of *range proofs* that avoid a logarithmic dependency on column size (which instead IntegriDB incurs).

On updatability. In our exposition so far we have described our framework considering a static database. Of course, many applications require updating the data and hence the corresponding vector commitments and accumulators. Fortunately, it is not necessary to recompute them from scratch as data change. This is mainly thanks to the homomorphic properties of the LVC scheme (see [21, Section 4.4]). Updating the accumulators in the preprocessing cannot be done in constant-time in a publicly verifiable setting, but our experiments show it to be practical for several common updates.

3 Related and Future Work

Prior schemes from authenticated data structures (ADS). To the best of our knowledge, after almost a decade, IntegriDB [85]still essentially represents the state of the art for efficient and expressive approaches based on ADS with succinct proofs. The reader can find a comparison of `qedb` and IntegriDB in the rest of the main text (including Fig. 1 and Table 1) and in Appendix A.

Other relevant works can be categorized as tree-based (e.g. [58] shows a Merkle tree-like data structure supporting efficient updates to an authenticated dataset) or signature-based (e.g. [65]). None of these works supports efficient verification for `JOIN` queries.

In [64] the authors present a general method for compiling a description of a data structure into its authenticated counterpart. Their final security relies only on collision-resistant hashing. As such, this approach cannot take advantage of the optimizations that are possible using algebraic commitments. Our work can be seen as adopting a similar abstract angle as that in [64] but with a slightly more specialized, algebraic-flavored approach (instead of λ-calculus) and resulting in more succinct schemes.

Several additional references are contained in the survey [75]. We finally cite representative works on functional commitments, a primitive close in spirit to ADS [7, 15, 18, 23, 24, 34, 52, 56, 61, 62, 80, 81].

Prior schemes from general cryptographic proof systems. vSQL [84] is a system that provides verifiability for a large subset of SQL queries and it is based on the CMT protocol [31]. In essence, it can be considered the first example of SNARK based on (an optimization of) the

[15] Set handles also occasionally represent *values*; we exploit this property for `JOINs`.

GKR protocol [43], and it incurs in large time and memory overheads when the database grows to millions of rows. The reader can find an additional comparison to vSQL in Appendix A.

We mention two recent works, that in addition to query verifiability, support zero-knowledge. ZKSQL [59] is VOLE-based [36] (thus inherently interactive and supporting non-public verification only). It supports a large subset of SQL queries, testing over the TPC-H benchmark. Its performances incur in large overheads as the dataset grows over a half million rows. This scheme is not succinct. PoneglyphDB [44] is based on the Halo2 proof system [83] and supports arbitrary SQL queries. It shows better performance than ZKSQL with an asymptotically logarithmic verifier and proof size (improving on the polylogarithmic dependency in vSQL). From preliminary estimates, PoneglyphDB shows typical proving times and proof sizes comparable to ours; on the other hand, it shows verification times at least an order of magnitude worse due to their use of inner product arguments [8] even on relatively small table sizes. While overall a performant proof system supporting zero-knowledge, PoneglyphDB requires writing SQL logic as constraints and its inner workings are quite complex, involving for example polynomial constraints verification within the circuit (see discussion in Section 5.5. in [44]). Being based on Halo2, PoneglyphDB relies on recursion. The recent work in [72] provides a VDB scheme leveraging several tech stacks including Jolt, Halo2 and RiscZero. Other approaches, tracing back to [63], provide zero-knowledge to very simple queries over key-value datasets [9, 25, 60] (these last works can be seen as ADS-based ones but we cite them here in the larger context of zero-knowledge database-like schemes).

Verifiable databases have also received recent attention from commercial efforts [4, 55, 77]. Since these systems offer limited documentation, here we present a preliminary comparison from the publicly available information. The approaches from Lagrange Labs [55] and Axiom [4] employ recursion and/or virtual machines under the hood; therefore, the security caveats we discuss in Section 4 apply; we refer the reader to it for additional discussion. The approach used by [77] seems philosophically closer to ours and attempts to leverage SQL-specific features as much as possible, combining it with techniques from multivariate sumchecks [43]. The building blocks in their code—Dory [57] and HyperPlonk [26]—have a logarithmic overhead for proof size and verification time. Along these two metrics, their system is likely to offer concretely worse performance than ours (especially in the case of Dory, due to the heavy use target group operations). Their HyperPlonk instantiation is likely to have concretely better performance in general compared to the one we experimentally evaluate in this work. This is due to their use of multivariate—rather than univariate—building blocks. We stress that in principle one can instantiate our compiler with similar building blocks (we leave this and the related experimental evaluation as future work). To the best of our knowledge, despite the phrasing in their documents, none of these three projects [4, 55, 77] offers *actual* zero-knowledge (in the sense of hiding parts of the database that are not part of the response).

Future work. A first interesting future line of inquiry could be the application of lookup relations and lookup arguments (see, e.g., [10, 13, 14, 41, 76, 82]), as a way to respectively model and instantiate a variant of our idealized protocols. Lookups seem intuitively close in spirit to some of the operations we perform in our model. At the same time, they may provide worse performance (lookups are optimized for *multisets*, which may be unnecessary for SQL) and they do not provide a perfect match to our interface. We are currently exploring this avenue.

Another interesting problem is how to add zero-knowledge (hiding features) in a simple manner to ADS building blocks. Potential approaches may involve equivocable commitments and some of the techniques used in [82] and in the Curve Trees/Forests lines of work in the transparent setting under DLOG [19, 20].

A final open question is: are there inherent limits to the tradeoffs succinctness–expressivity of ADS-based approaches? That is, can we prove bounds on, e.g., communication complexity for ADS-based approaches for expressive queries? The fact that ADS closely follow the query patterns of data structure for proving/verification suggests we may find such results from the literature on data structures (both authenticated and not).

4 Discussion: On Methodologies to Build VDBs

The methodology in qedb vs VDBs from general-purpose SNARKs In the introduction we motivated a modular approach based on authenticated structures by pointing out two drawbacks in general-purpose proof systems (and in circuit- or constraint- based representations): we stated that VDBs are more prone to bugs and offer a poorer developer experience. We now expand on these points and on why qedb proposes improvements in that respects. We argue that qedb mitigates these problems in at least two ways: offering a simpler *representation* and simpler *protocols*.

As a premise, recall that both qedb and several families of SNARKs—those based on (P)IOPs—can be seen, from 5000 feet, as consisting of a pipeline of this form:

$$\text{representation} \xrightarrow{\text{fed into}} \text{(idealized) protocol} \xrightarrow{\text{compiled into}} \text{argument system}$$

Both in general purpose SNARKs and in this paper, the final compilation process is very simple; we thus focus our comparison on the two ends of the first arrow.

A "representation" of a query qry in qedb directly *reflects* the SQL operations in qry: it consists of a series of steps like the ones in Fig. 5, each a set/vector operation. In contrast, a circuit needs to *emulate* the relational logic through gates or constraints. The result is usually a more complicated object. For example, consider a core query operation such as set intersection. When expressed as a circuit, its most straightforward implementation is not only inefficient—it involves a quadratic number of steps—but also unnecessarily complicated. More efficient approaches add even more complexity; they may involve sorting networks and/or handling duplicates (see vSQL, Sec. VI.C [84]). An auditor or automatic tool will plausibly have an easier time spotting bugs in code for set intersection represented in the style of Fig. 5 (one constraint in our idealized VDB model) than, say, in a circuit implementing (at the very least) a sorting network. The same is arguably true for the developers taking up the task of writing the related code.

The final protocols in qedb are also simpler compared to those in common SNARKs. An idealized protocol for us consists of operations that almost directly map to the operations in a query (see Section 10). Given a query, the steps of the protocol boil down to a handful of simple manipulations of vectors and sets. No algebraic fact is ever invoked in the protocol, which is, at its core, non-interactive. In contrast a (P)IOP typically involves running several rounds of interaction, which are usually "compressed" into one through the Fiat-Shamir heuristic (which requires particular care when implemented; see examples of documented attacks in [33]).

The checks in these idealized protocols typically involve equations over polynomials of various complexity. Protocols of this type arguably require more expertise to write, read, understand, analyze or audit (for an intuition of what we mean here, see for example the *relatively simple* idealized protocol in Figure 5 of [27]). It is our view that the (substantially structured) setting of SQL computations can leverage simpler and more easily auditable schemes without major sacrifices in efficiency.

Comparing qedb and approaches based on recursion or "zk"-VMs Here we compare qedb to alternative designs for general-purpose proof systems, in particular those relying on recursion and/or a "ZK"-VMs[16] While they are at times used as distinct approaches, we discuss them together because it is very common for them to come hand in hand. In particular we use recursive proofs to obtain a cryptographically provable VM[17].

These designs offers compelling advantages, such as a promising proving performance (albeit not on consumer hardware and often requiring networks of GPUs) and high expressivity (being general-purpose they can potentially express any SQL query). Another advantage is a better developer experience compared to constraint systems or circuits: computation to be proven can be specified in an arbitrary programming language, as long as one can compile it into the instruction set of the virtual machine.

[16] A virtual machine that lends itself to cryptographic proofs. We stress that, almost without exception, the "ZK" is a misnomer—these systems do not offer zero-knowledge (hence the quotation marks).

[17] There are notable exceptions to this template, e.g., the *lookup singularity* approach used in Jolt [3,12].

Query	Prover Time	Verifier Time	Proof Size
Q_{Tot}	1.21 s	13.00 ms	0.66 KB
Q_{CntTx}	15.59 s	21.81 ms	5.13 KB
Q_{MatchExp}	6.15 s	25.17 ms	0.98 KB

Table 2: Experimental evaluation over queries from Fig. 2 on a DB with 100K rows.

On the other hand these systems have a few disadvantages: they are very complex and the security of some of their building blocks is not well understood. Regarding the complexity—and hence auditability and maintanibility of the stack, etc.—these designs rely on even more layers and moving parts than the SNARKs we described earlier: a general proof system—typically a STARK—is used in a recursive fashion to prove statements about a VM execution (which involves modeling registers, memory, and so on). SQL query execution is then described *on top* of this last layer. From a security standpoint, recursive proof systems—especially those including Fiat-Shamir challenges generation in the recursion step, a common heuristic underlying "ZK"-VM implementations—have been mentioned as being potentially sensitive to recently documented attacks on Fiat-Shamir (see for example [54]).

5 Implementation and Experimental Evaluation

5.1 Implementation and experimental setup

We have implemented our construction using the Rust programming language and the Arkworks library [2] for efficient cryptographic operations. We use the BLS12-381 as a pairing-friendly elliptic curve. We run our experiments in a multi-threaded setting on a commodity laptop: a MacBook Pro featuring 11 cores and 36 GB of RAM. (Note that our prover has very low memory requirements: less than 3 GB of RAM to handle a database of 100,000 rows.)

Experimental queries and dataset. We performed our experiments using the queries described in Section 1.3 (see also Fig. 2). These queries showcase some of the expressivity features of our system. Overall, they involve different types of queries—aggregate queries (`SUM` and `COUNT`) as well as `SELECT`—and different types of filtering conditions—conjunctions of predicates, range queries, simple equality tests on target values, comparison among columns[18].

The test database consists of a single table with columns populated through synthetic data (generated by sampling random values in the appropriate domain and converting them into scalars). Our main experiments in this section will be on a table with 100,000 rows. This number is chosen both because it is representative of a medium-sized DB and because it roughly corresponds to the cut-off point where previous state-of-the-art solutions (IntegriDB [85]) experimentally fails to be able to perform proving due to its memory usage[19]. In contrast, qedb can run on DBs of millions of rows. We performed experiments on these larger tables too to see how our scheme scales (we will provide details on the related performance in this section). The `SELECT` in Q_{MatchExp} has a response of 1K rows.

5.2 Experimental Results

We now discuss our main experimental results, which are reported in Table 2.

Proof Size. The proof sizes for Q_{Tot} and Q_{MatchExp} reflect a typical ballpark for the proof sizes in qedb, i.e., 0.5-1KB. These two are distinct types of queries, respectively a `SUM` and a `SELECT`.

[18] We defer a full experimental evaluation including `JOIN` queries to the next revision of this article. We stress that both our approach to `JOIN`s and the queries evaluated in this section share the same set of essential operations underlying them. Details on our approach are on page 36.

[19] This was tested on a 128GB RAM machine; the IntegriDB paper also documents this phenomenon with similar numbers.

The larger proof in Q_{CntTx} is due to our range query; this can be reduced by at least 25% through simple optimizations. As mentioned, our proof size is independent of the sizes of the response and the DB.

Verification time. Our verifier runs in 15–25 ms on all three of the queries. This is also a common ballpark which we found in other experiments we performed. The verification time for Q_{CntTx} does not grow substantially despite the larger proof size because it mostly consists of pairings equations which can easily be checked in batch. The only query that admits different response sizes—and where, as a consequence, the verifier may need to perform more work—is Q_{MatchExp}. We tested this query on larger responses and observed that even increasing its size by a factor $5\times$ at most doubles the verification time (which is still ≈ 50 ms).

Proving time. Our prover runs is able to run in one second on our query Q_{Tot}; the other proving times are both in the ballpark of ten seconds. The higher proving time for Q_{CntTx} is due to the range proofs, whereas that for Q_{MatchExp} is mainly due to the type of query, a `SELECT` (in which the proving also has some dependency on the response size). From our additional experiments, our prover scales linearly (with a constant < 1) in the size of the database for aggregation queries such as Q_{Tot} and Q_{CntTx}. For example, on a DB of $\approx 1M$ rows ($10\times$ larger), both queries run in time $\approx 8\times$ that reported in Table 2. Q_{Tot} in particular still runs in only approximately ten seconds. For `SELECT` queries, increasing the DB size has a quasilinear behavior and thus a slightly larger overhead compared to other two (the concrete factor depending also on the response size).

Pre-processing. We now report on the time required to generate the cryptographic material during the DB preprocessing (not in Table 2). Generally this number grows linearly in the total number of rows and columns in the database. For example, for a table with 100K rows it takes approximately 10s for 5 columns and around 16s for 9 columns. For a larger dataset of 1M rows, the generation time increases to roughly 80s for 5 columns and about 130 s for 9 columns.

6 Background: Authenticated Data Structures

6.1 Notation and cryptographic assumptions

We assume basic familiarity with bilinear groups and related assumptions:

Bilinear groups. A bilinear group is given by a description $\mathsf{gk} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ with additive notation such that p is prime, so $\mathbb{F} = \mathbb{F}_p$ is a field. $\mathbb{G}_1, \mathbb{G}_2$ are cyclic (additive) groups of prime order p. We use the notation $[a]_1, [b]_2, [c]_t$ for elements in $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ respectively. $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear asymmetric map (pairing), which means that $\forall a, b \in \mathbb{Z}_p, e([a]_1, [b]_2) := [ab]_t$. We implicitly have that $[1]_t := e([1]_1, [1]_2)$ generates $\mathbb{G}_T$. We use $[a]_{1,2}$ to refer to two group elements $[a]_1 \in \mathbb{G}_1, [a]_2 \in \mathbb{G}_2$. In our constructions, we denote by $\mathcal{G}(p)$ the algorithm that, given as input the prime value p, outputs a description $\mathsf{gk} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$.

Lagrange polynomials. Given n distinct nonzero field elements $x_1, \ldots, x_n \in \mathbb{F}$ and corresponding points $P_1, \ldots, P_n$, Lagrange interpolation computes a linear combination of these points to reconstruct a target polynomial P defined as:

$$P(x) = \sum_{i=1}^{n} \lambda_i \cdot P_i(x),$$

where the *Lagrange coefficients* λ_i are given by

$$\lambda_i = \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x_j}{x_j - x_i} \in \mathbb{F}.$$

In our instantiations we assume fast FFTs to compute the above.

Assumptions. We state the computational assumptions used in this work.

Definition 1. *The q-DLOG assumption holds relative to $\mathcal{G}(1^\lambda)$ if for all PPT adversaries $\mathcal{A}$, the following holds:*

$$\Pr\left[\tau \leftarrow \mathcal{A}(\mathsf{gk}, \{[\tau^i]_1\}_{i=0}^q, \{[\tau^i]_2\}_{i=0}^q) \mid \mathsf{gk} \leftarrow \mathcal{G}(1^\lambda); \tau \leftarrow \mathbb{F}\right] \leq \mathsf{negl}(\lambda).$$

The

Definition 2 ([45]). *The q-BSDH assumption holds relative to $\mathcal{G}(1^\lambda)$ if for all PPT adversaries $\mathcal{A}$, the following holds:*

$$\Pr\left[(c, \frac{1}{(\tau - c)} e([1]_1, [1]_2)) \leftarrow \mathcal{A}(\mathsf{gk}, \{[\tau^i]_1\}_{i=0}^q, \{[\tau^i]_2\}_{i=0}^q) \mid \mathsf{gk} \leftarrow \mathcal{G}(1^\lambda); \tau \leftarrow \mathbb{F}\right] \leq \mathsf{negl}(\lambda).$$

The AGM. The algebraic group model (AGM) [40] is an idealized model, where the adversary is modeled as an algebraic algorithm. Algebraic algorithms may only compute group elements as linear combinations of group elements observed so far. Therefore, whenever they output a group element $X \in \mathbb{G}_1$, they also provide a representation $\{\alpha_i\}_{i=1}^N$ of $X = \sum_{i=1}^N [\alpha_i g_i]_1$ as a function of previously seen elements $[g_1], \ldots, [g_N] \in \mathbb{G}_1$ of the same group.

6.2 Functional commitments with unique setup

A functional commitment is a commitment scheme that supports special ("functional") openings. Set accumulators, polynomial and vector commitments are all special cases of this notion. All the underlying building blocks we will define will rely on the a single setup. Therefore in our construction we instantiate these blocks with tools that use a compatible setup procedure. The setup will have the following syntax:

Definition 3. *A setup $\mathsf{Setup}(1^\lambda) \to (\mathsf{prk}, \mathsf{vrk})$ is a probabilistic algorithm that takes as input the security parameter λ, a size bound, and outputs a pair of keys $(\mathsf{prk}, \mathsf{vrk})$ (for proving and verification respectively). We assume λ implicitly provides a size parameter (e.g., vector size) for all our schemes.*

Since all functional commitments are commitments, they will all have binding properties and a common syntax, which we now define:

Definition 4. ***(Commitment scheme).*** *A commitment scheme $\mathsf{Com} = (\mathsf{Setup}, \mathsf{AlgoCom})$ is composed by a message space $\mathcal{M}$ and a tuple of the following polynomial-time algorithm (we recall that the Setup algorithm is defined in Definition 3):*

- $\mathsf{AlgoCom}(\mathsf{prk}, m) \to (\mathsf{cm}, \mathsf{aux})$ *is a deterministic algorithm that takes as inputs* prk*, a element $m \in \mathcal{M}$ and outputs the commitment* cm *and an auxiliary value* aux*.*

A commitment scheme satisfies the following binding property.

Binding: *For any security parameter $\lambda \in \mathbb{N}$ and any PPT adversary $\mathcal{A}$, it holds that:*

$$\Pr\left[\begin{array}{c|c} \mathsf{cm}_{m_0} \leftarrow \mathsf{AlgoCom}(1^\lambda, \mathsf{prk}, m_0) \wedge & (\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{cm}_{m_1} \leftarrow \mathsf{AlgoCom}(1^\lambda, \mathsf{prk}, m_1) \wedge & (m_0, m_1) \leftarrow \mathcal{A}(1^\lambda) \\ \mathsf{cm}_{m_0} = \mathsf{cm}_{m_1} & m_0, m_1 \in \mathcal{M} \end{array}\right] \leq \mathsf{negl}(\lambda).$$

6.3 Linear-map vector commitments

A linear-map vector commitment[20] (LVC) has these core capabilities: the ability to commit to a vector succinctly; the ability, given two commitments to vectors $\mathsf{cm}_{\boldsymbol{u}}$ and $\mathsf{cm}_{\boldsymbol{v}}$ to verify (and—on the prover side—prove) that $\langle \boldsymbol{u}, \boldsymbol{v} \rangle$ equals some claimed value y.

Definition 5. ***Linear-map Vector Commitment.*** *A Linear-map Vector Commitment (LVC) scheme for vectors in $\mathbb{F}^n$ is a tuple of probabilistic polynomial time algorithms* $(\mathsf{CommitVec}, \mathsf{OpenLin}, \mathsf{VerifyLin})$ *(with Setup algorithm defined in Definition 3) that work as follows:*

- $\mathsf{CommitVec}(\mathsf{prk}, \mathbf{v}) \to (\mathsf{cm}, \mathit{aux})$ *is a deterministic algorithm that on input the proving key* prk *and a vector* $\mathbf{v} = (v_1, v_2, \ldots, v_m) \in M^m$*, returns a commitment* cm *and auxiliary information* aux*.*
- $\mathsf{OpenLin}(\mathsf{prk}, \mathit{aux}_{\boldsymbol{u}}, \mathit{aux}_{\boldsymbol{v}}, y) \to \pi_y$ *is a deterministic algorithm that takes as input* prk*, auxiliary information about two (committed) vectors and outputs a proof π that $y = \langle \boldsymbol{u}, \boldsymbol{v} \rangle$.*
- $\mathsf{VerifyLin}(\mathsf{vrk}, \mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}, y, \pi) \to 0/1$ *is a deterministic algorithm (running in time sublinear in the size of the committed vectors) that takes as input the verification key* vrk*, commitments* $\mathsf{cm}_{\boldsymbol{u}}$ *and* $\mathsf{cm}_{\boldsymbol{v}}$*, a proof π and accepts or rejects.*

An LVC scheme satisfies the following properties: *Correctness* requires that if a prover commits to two vectors and computes their inner product honestly, then the proof they generate will always convince the verifier of the correct result; *binding* property means that it is infeasible for an adversarial committer to produce a commitment cm and two valid openings $\boldsymbol{u}$ and $\boldsymbol{v}$; *Homomorphic* property means that given commitments cm_u to $\boldsymbol{u}$ and cm_v to $\boldsymbol{v}$, it must be efficient to compute a commitment cm_{u+v} by computing $\mathsf{cm}_{u+v} = a \cdot \mathsf{cm}_u + b \cdot \mathsf{cm}_v$; finally, *extractability* property means that there exists an efficient extractor $\mathcal{E}$ such that, given a commitment cm_u and oracle access to an adversarial prover P^* that produces a proof π accepted by the verification algorithm $\mathsf{VerifyLin}$ for some commitment cm_v and scalar y, the extractor can recover a vector $\boldsymbol{u}'$ such that $\mathsf{CommitVec}(\boldsymbol{u}') = \mathsf{cm}_u \wedge \langle \boldsymbol{u}', \boldsymbol{v} \rangle = y$ with all but negligible probability. More formally:

LVC Correctness: An LVC scheme is perfectly correct if for all $\lambda \in \mathbb{N}$, and any $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{F}^n$,

$$\Pr\left[\mathsf{VerifyLin}(\mathsf{vrk}, \mathsf{cm}_u, \mathsf{cm}_v, y, \pi_y) = 1 \;\middle|\; \begin{array}{c} (\mathsf{vrk}, \mathsf{prk}) \leftarrow \mathsf{Setup}(1^\lambda, F) \\ (\mathsf{cm_u}, \mathsf{aux_u}) \leftarrow \mathsf{Commit}(\mathsf{prk}, \mathbf{u}) \\ (\mathsf{cm_v}, \mathsf{aux}) \leftarrow \mathsf{Commit}(\mathsf{prk}, \mathbf{v}) \\ \pi_y \leftarrow \mathsf{OpenLin}(\mathsf{prk}, \mathsf{aux_u}, \mathsf{aux_v}, y) \end{array}\right] = 1.$$

Extractability: For all $\lambda \in \mathbb{N}$, for all PPT $\mathcal{A}$ there exists an efficient extractor $\mathcal{E}_{\mathcal{A}}$ such that:

$$\Pr\left[\begin{array}{l} \mathsf{VerifyLin}(\mathsf{vrk}, \mathsf{cm}_u, \mathsf{cm}_v, y, \pi) = 1 \wedge \\ \langle \boldsymbol{u}^*, \boldsymbol{v}^* \rangle \neq y \vee \\ (\mathsf{cm}_u, \cdot) \neq \mathsf{Commit}(\mathsf{prk}, \boldsymbol{u}^*) \vee \\ (\mathsf{cm}_v, \cdot) \neq \mathsf{Commit}(\mathsf{prk}, \boldsymbol{v}^*) \end{array} : \begin{array}{c} (\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{Setup}(1^\lambda, F) \\ (\mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}, y, \pi) \leftarrow \mathcal{A}(\mathsf{prk}) \\ (\boldsymbol{u}^*, \boldsymbol{v}^*) \leftarrow \mathcal{E}_{\mathcal{A}}(\mathsf{prk}, \mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Homomorphism of Commitments: Given two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ and any scalars a, b it holds that if $\boldsymbol{w} = a \cdot \boldsymbol{u} + b \cdot \boldsymbol{v}$ then $\mathsf{Commit}(\mathsf{prk}, \boldsymbol{w}) = a \cdot \mathsf{cm}_{\boldsymbol{u}} + b \cdot \mathsf{cm}_{\boldsymbol{v}}$, where $\mathsf{cm}_{\boldsymbol{u}}$ and $\mathsf{cm}_{\boldsymbol{v}}$ are respectively the commitments of $\boldsymbol{u}$ and $\boldsymbol{v}$.

Remark 1 (Difference with [21]). We observe that our interface is slightly different than the one in the original presentation of linear-map vector commitments in [21]. There the verifier holds a *commitment* $\mathsf{cm}_{\boldsymbol{u}}$ and a plaintext $\boldsymbol{v}$ (the linear function). In our case both are committed. This *symmetric* interface is more versatile in our setting; we describe how to realize it in Section 6.6.

[20] Note to the reader: in our syntax we consider a restricted version of linear-map (inner product). We keep the terminology "linear map" because vector commitments that can prove inner products can be immediately lifted to ones proving general linear maps [21].

6.4 Subvector-opening vector commitments

A Subvector Opening Vector Commitment (VC) scheme is a tuple of probabilistic polynomial time algorithms $(\mathsf{CommitVec}, \mathsf{OpenSub}, \mathsf{VerifySub})$.

Definition 6. ***Subvector Opening Vector Commitment.*** *A Subvector Opening Vector Commitment (VC) scheme for vectors in $\mathbb{F}^n$ is a tuple of probabilistic polynomial time algorithms $(\mathsf{CommitVec}, \mathsf{OpenSub}, \mathsf{VerifySub})$ (with Setup algorithm defined in Definition 3) that work as follows:*

- $\mathsf{OpenSub}(\mathsf{prk}, \mathsf{aux}_{\boldsymbol{u}}, I) \to \pi$ *is a deterministic algorithm that takes as input* prk*, a vector* $\boldsymbol{u}$*, a set of indices* I *and outputs a proof* π*.*
- $\mathsf{VerifySub}(\mathsf{vrk}, \mathsf{cm}_u, \boldsymbol{v}, I, \pi) \to 0/1$ *is a deterministic algorithm (running in time sublinear in the size of the committed vector* $\boldsymbol{u}$*) that takes as input the verification key* vrk*, commitments* cm_u*, a subvector* $\boldsymbol{v}$*, a set of indices* I*, a proof* π *and accepts or rejects.*

The commitment algorithm $\mathsf{CommitVec}$ is the same used by LVC(see Def.5). VC scheme satisfies all the properties of LVC (see Def.5) where only correctness and extractability change as follows:

The properties of a Vector Commitment (VC) scheme (see Section 6.6 for details) match those of an LVC scheme, except that: (i) correctness requires that if the prover commits to a vector $\boldsymbol{v}$ and opens a subvector $\boldsymbol{u}$ at indices $I \subseteq [\bar{n}]$, then the proof from $\mathsf{OpenSub}$ always convinces the verifier that $\boldsymbol{u}$ is correct; (ii) extractability requires an efficient extractor $\mathcal{E}$ such that, given a commitment cm_u and oracle access to an adversary P^* outputting a proof π accepted by $\mathsf{VerifySub}$ for some I, it can recover $\boldsymbol{u}'$ satisfying $\mathsf{CommitVec}(\mathsf{prk}, \boldsymbol{u}') = \mathsf{cm}_u$ with all but negligible probability.

Correctness For all $\lambda \in \mathbb{N}, \boldsymbol{v} \in \mathbb{F}^n, X := \{j_1, \ldots, j_\ell\} \subseteq [n], \boldsymbol{w} \in \mathbb{F}^\ell$ such that $(v_{j_1}, \ldots, v_{j_\ell}) = (w_1, \ldots, w_\ell)$, it holds that:

$$\Pr\left[\mathsf{VerifySub}\left(\mathsf{vrk}, \mathsf{cm}_{\boldsymbol{v}}, \boldsymbol{w}, X, \pi\right) = 1 \;:\; \begin{array}{l}(\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathsf{cm}_{\boldsymbol{v}}, \mathsf{aux}_{\boldsymbol{v}}) \leftarrow \mathsf{CommitVec}(\mathsf{prk}, \boldsymbol{v}) \\ \pi \leftarrow \mathsf{OpenSub}\left(\mathsf{prk}, \mathsf{aux}_{\boldsymbol{v}}, X\right)\end{array}\right] \geq 1 - \mathsf{negl}(\lambda).$$

Extractability: For all $\lambda \in \mathbb{N}$, for all PPT $\mathcal{A}$ there exists an efficient extractor $\mathcal{E}_{\mathcal{A}}$ such that:

$$\Pr\left[\begin{array}{l}\mathsf{VerifySub}\left(\mathsf{vrk}, \mathsf{cm}_{\boldsymbol{v}}, \boldsymbol{w}, X, \pi\right) = 1 \wedge \\ \left(\mathsf{cm}_{\boldsymbol{v}} \neq \mathsf{CommitVec}(\mathsf{prk}, \boldsymbol{v}) \vee\right. \\ \left.(v_{j_1}, \ldots, v_{j_\ell}) \neq (w_1, \ldots, w_\ell)\right)\end{array} : \begin{array}{l}(\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathsf{cm}_{\boldsymbol{v}}, X := \{j_1, \ldots, j_\ell\}, \pi) \leftarrow \mathcal{A}\left(\mathsf{prk}\right) \\ \boldsymbol{v} \leftarrow \mathcal{E}_{\mathcal{A}}(\mathsf{prk}, \mathsf{cm}_{\boldsymbol{v}})\end{array}\right] \leq \mathsf{negl}(\lambda).$$

6.5 Polynomial commitments

A polynomial commitment scheme (PCS) $\mathsf{PCom} = (\mathsf{Commit}, \mathsf{OpenPos}, \mathsf{VerifyPos})$ (with Setup algorithm defined in Definition 3) is defined as follows:

- $\mathsf{Commit}(\mathsf{prk}, \phi(x)) \to (\mathsf{cm}, d)$ is a probabilistic polynomial time algorithm that takes in input the public key and a polynomial $\phi(x)$ returns a commitment cm of $\phi(x)$ and a decommitment d.
- $\mathsf{OpenPos}(\mathsf{prk}, \phi(i), i, d) \to (i, \phi(i), \pi_i)$ is a probabilistic polynomial time algorithm that takes in input the public key, the polynomial evaluated in i, the value of i and the decommitment value and outputs a proof π_i for the evaluation $\phi(i)$ of $\phi(x)$ at the index i, together with i and $\phi(i)$.
- $\mathsf{VerifyPos}(\mathsf{vrk}, \mathsf{cm}, i, \phi(i), \pi_i) \to 0/1$ is a polynomial time algorithm (running in time sublinear in the size of the committed polynomial ϕ) that takes in input the public key, the commitment, the evaluation at index i of the polynomial, and the proof π_i and outputs 1 in case $\phi(i)$ is indeed the evaluation of the polynomial at index i and 0 otherwise.

A PCS is *succinct* if the size of commitments and evaluation proofs grows at most logarithmically with the degree of committed polynomials. We require that a PCS satisfies the binding property meaning that it is infeasible for an adversary to find an opening for the commitment to two different polynomials ϕ and ϕ'. The PCS must satisfy evaluation binding, that captures the infeasibility for an adversary to prove two distinct evaluations of a committed polynomial on the same input. We also require that PCS achieves extractability, meaning that whenever an adversary outputs a valid commitment cm and a proof π for the evaluation in the point (x, y), there exists an efficient extractor that can recover the polynomial ϕ such that $\phi(x) = y$ and cm is the commitment of ϕ. The last property we require for a PCS is the homomorphic property that guarantees that given commitments cm_ϕ and cm_ψ of polynomials ϕ and ψ respectively, it is efficient to compute a new commitment $\mathsf{cm}_{\phi+\psi}$ that is a valid commitment to the corresponding operation on the polynomials, without requiring access to the underlying coefficients.

Correctness For all $\lambda \in \mathbb{N}, \phi(x) \in \mathbb{F}[x], i \in [n]$, it holds that:

$$\Pr\left[\begin{array}{r} \mathsf{VerifyPos}(\mathsf{vrk}, \mathsf{cm}, i, \\ \phi(i), w_i) = 1 \end{array} : \begin{array}{l} (\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathsf{cm}, d) \leftarrow \mathsf{Commit}(\mathsf{prk}, \phi(x)) \\ (i, \phi(i), w_i) \leftarrow \mathsf{OpenPos}(\mathsf{prk}, \phi(i), i, d) \end{array}\right] \geq 1 - \mathsf{negl}(\lambda).$$

Extractability: For all $\lambda \in \mathbb{N}$, for all PPT $\mathcal{A}$ there exists an efficient extractor $\mathcal{E}_\mathcal{A}$ such that:

$$\Pr\left[\begin{array}{l} \mathsf{VerifyPos}(\mathsf{vrk}, \mathsf{cm}, i, \phi(i), w_i) = 1 \wedge \\ (\mathsf{cm}, \cdot) \neq \mathsf{Commit}(\mathsf{prk}, \phi'(x)) \vee \\ \phi'(i) \neq \phi(i) \end{array} : \begin{array}{l} (\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathsf{cm}, i, \phi(i), w_i) \leftarrow \mathcal{A}\,(\mathsf{prk}) \\ \phi'(x) \leftarrow \mathcal{E}_\mathcal{A}(\mathsf{prk}, \mathsf{cm}) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

6.6 Building blocks (Instantiations)

We call $\bar{n}$ the size of the vectors; for simplicity we assume it is some well-defined function (a polynomial) of the security parameter λ.

Setup. The setup procedure we require is the standard one for KZG:

$\mathsf{Setup}(1^\lambda, \bar{n}) \to (\mathsf{prk}, \mathsf{vrk})$**:** Given the security parameter and the size bound, Setup works as follows:

- Generate the group description $gk = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathcal{G}(p)$; Define the multiplicative group $\mathbb{H} = \{\mathsf{h}_1, \ldots, \mathsf{h}_{\bar{n}}\}$ in $\mathbb{F}$: Compute the Lagrange polynomials $\{\lambda_j(X)\}_{j=1}^{\bar{n}}$ over $\mathbb{H}$; Sample $\tau \leftarrow_\$ \mathbb{F}$.
- Output $\mathsf{prk} = \left(\{[\tau^i]_{1,2}, [\lambda_i(\tau)]_1\}_{i=1}^{\bar{n}}\right)$ and $\mathsf{vrk} = \left([1]_{1,2}, \left\{[\tau^i]_2, [\lambda_i(\tau)]_2\right\}_{i=1}^{\bar{n}}\right)$.

Linear-map Vector Commitment. This is a linear-map vector commitment scheme for a Lagrange basis $\{\lambda_i(X)\}_{i=1}^{\bar{n}}$ over a multiplicative group $\mathbb{H} = \{\mathsf{h}_1, \ldots, \mathsf{h}_{\bar{n}}\}$ of size $\bar{n}$ in $\mathbb{F}$. We encode a vector $\boldsymbol{a} \in \mathbb{F}^{\bar{n}}$ as a polynomial $a(X) = \sum_{i=1}^{\bar{n}} a_i \lambda_i(X)$.

$\mathsf{CommitVec}(\mathsf{prk}, \boldsymbol{a}) \to (\mathsf{cm}, \mathbf{aux})$**:** Compute $\mathsf{cm} = \sum_{i=1}^{\bar{n}} a_i [\lambda_i(\tau)]_1$ and output $(\mathsf{cm}, \boldsymbol{a})$. Notice that $\mathsf{CommitVec}$ is the instantiation of $\mathsf{AlgoCom}$.

$\mathsf{OpenLin}(\mathsf{prk}, \boldsymbol{u}, \boldsymbol{v}, y) \to \pi_y$**:** Find $R(X), H(X)$ such that $\deg(R) < \bar{n} - 1$ and

$$\left(\sum_{i=1}^{\bar{n}} u_i \lambda_i(X)\right)\left(\sum_{i=1}^{\bar{n}} v_i \lambda_i(X)\right) - \bar{n}^{-1} y = XR(X) + t(X)H(X).$$

Define $\hat{R}(X) = X^2 R(X)$ and $\widetilde{\mathsf{cm}}_v = \sum_{i=1}^{\bar{n}} v_i [\lambda_i(\tau)]_2$. Output $\pi = \left([R(\tau)]_1, [H(\tau)]_1, [\hat{R}(\tau)]_1, \widetilde{\mathsf{cm}}_v\right)$.

$\mathsf{VerifyLin}(\mathsf{vrk}, \mathsf{cm}_u, \mathsf{cm}_v, y, \pi) \to 0/1$**:** Parse $\pi = \Big([R]_1, [H]_1, [\hat{R}]_1, \widetilde{\mathsf{cm}}_v\Big)$ and output 1 if and only if $e(\mathsf{cm}_u, \widetilde{\mathsf{cm}}_v) - e(\bar{n}^{-1} y[1]_1, [1]_2) = e([R]_1, [\tau]_2) + e([H]_1, [t(\tau)]_2)$, $e([R]_1, [\tau^2]_2) = e\big([\hat{R}]_1, [1]_2\big)$, and $e\big(\mathsf{cm}_v, [1]_2\big) = e\big([1]_1, \widetilde{\mathsf{cm}}_v\big)$.

Theorem 2 (Implicit in [21]). *The scheme above is a correct, binding and extractable homomorphic LVC (Section 6.3) in the AGM under the q-BSDH assumption.*

Proof (Proof of Theorem 2).

- Completeness follows by inspection.
- The homomorphic property follows from the following observation. Given the commitments $\mathsf{cm}_u = \sum_{i=1}^{N} u_i [\lambda_i(\tau)]_1$ $\mathsf{cm}_v = \sum_{i=1}^{N} v_i [\lambda_i(\tau)]_1$ respectively of $\boldsymbol{u}$ and $\boldsymbol{v}$, let $\boldsymbol{w} = a \cdot \boldsymbol{u} + b \cdot \boldsymbol{v}$, the commitment cm_w of $\boldsymbol{w}$ is $a \cdot \mathsf{cm}_u + b \cdot \mathsf{cm}_v$.
- The scheme satisfies binding under q-DLOG assumption in the AGM (Definition 2).
- The scheme satisfies extractability in the AGM model under the q-BSDH assumption. We follow and expand the proof in [21, Thm. 6]. The proof proceeds trough a sequence of games. The first game G_0 is the original game of the binding game (Def. 4). $\mathcal{A}$ outputs $(\mathsf{cm}_u, \mathsf{cm}_v, y, \pi)$ and $(\mathsf{cm}_u, \mathsf{cm}_{v'}, y', \pi')$ with $y \neq y'$ but both verify it is possible to extract $\boldsymbol{v}$ and $\boldsymbol{v}'$ from cm_v and $\mathsf{cm}_{v'}$. G_1 is the same as G_0 but aborts if $\deg(R) > m - 2$ where $R(X)$ is the algebraic representation of $[R]_1$ from π. If G_0 and G_1 are distinguishable we can construct an adversary $\mathcal{B}$ that solves qSDH by extracting high-degree terms from $R(X)$. The advantage of $\mathcal{A}$ in G_1 is negligible, indeed, given $\mathsf{cm}_u(X) = \sum_{j=1}^{m} u_j \lambda_j(X) + X^m \hat{u}$ and $P(X) = \mathsf{cm}_u(X)\mathsf{cm}_v(X) - m^{-1} y - XR(X) - t(X)Q(X)$, the verification equation implies $P(\tau) = 0$. Therefore there are two cases:
  Case 1: $P(X) = 0$. Then:

$$\begin{aligned} \sum_{j=1}^{m} u_j v_j \lambda_j(X) + X^m \hat{u} \sum_{j=1}^{m} v_j \lambda_j(X) &= m^{-1} y + XR(X) \\ \implies \sum_{j=1}^{m} u_j v_j \lambda_j(0) &= m^{-1} y \quad \text{(evaluate at 0)} \\ \implies \sum_{j=1}^{m} u_j v_j &= y \quad (\text{since } \lambda_j(0) = m^{-1}) \end{aligned}$$

  Thus there exists $\boldsymbol{u}$ with $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = y$ and $\mathcal{A}$ loses.
  Case 2: $P(X) \neq 0$ but $P(\tau) = 0$. We build $\mathcal{B}$ against dlog:
  1. Given $[\tau]_1$, $\mathcal{B}$ computes roots of $P(X)$;
  2. Checks which root τ' of $P(X)$ satisfies $[\tau']_1 = [\tau]_1$.

  The only point that remains to prove is that the value $\widetilde{\mathsf{cm}}_v$ in π and the value $\widetilde{\mathsf{cm}}_{v'}$ in π' are equal to the values cm_v and $\mathsf{cm}_{v'}$ received in input from $\mathsf{VerifyLin}$, but it follows from the last check $e\big(\mathsf{cm}_v, [1]_2\big) = e\big([1]_1, \widetilde{\mathsf{cm}}_v\big)$ of $\mathsf{VerifyLin}$.

Subvector Opening Vector Commitment A Subvector Opening Vector Commitment (VC) scheme for vectors in $\mathbb{F}^{\bar{n}}$ is a tuple of probabilistic polynomial time algorithms (Setup, $\mathsf{CommitVec}$, $\mathsf{OpenSub}$, $\mathsf{VerifySub}$). $\mathsf{CommitVec}$ is equal to the commitment algorithm of LVC. The others algorithms work as follows:

- $\mathsf{OpenSub}(\mathsf{prk}, \mathsf{aux}_{\boldsymbol{u}}, I) \to \pi$: Let $I = \{j_1, \ldots, j_\ell\}$, and $u(x)$ be the interpolation of $\boldsymbol{u}$, the vector $\boldsymbol{v} = (v_1, \ldots, v_\ell)$ is the subvector of $\boldsymbol{u}$ in I, i.e., $v_i := u_{j_i}$ for $i = 1, \ldots, \ell$. Compute $v(X)$, the polynomial obtained by interpolation so that $v(j) = v_j, \forall j \in I$. Find $H(X)$ such that for $t_I(X) := \prod_{i \in I}(X - h_i)$ it holds that $u(X) - v(X) = t_I(X)H(X)$. Output $\pi = [H]_1 = [H(\tau)]_1$.
- $\mathsf{VerifySub}(\mathsf{vrk}, \mathsf{cm}_u, \boldsymbol{v}, I, \pi) \to 0/1$: Compute $[t_I]_2 = [t_I(\tau)]_2$, and $v(X)$ as above, and output 1 if and only if $e(\mathsf{cm}_u - [v(\tau)]_1, [1]_2) = e([H]_1, [t_I]_2)$.

This construction is described in [21, Section 6.3, Appendix C] and is originally from [79].

Theorem 3 (Implicit in [21]). *The scheme above is a correct, binding and extractable homomorphic VC (Section 6.4) in the AGM.*

The theorem follows [21, Sections 5.2 and 6.3; Appendix C].

KZG Polynomial Commitment. The polynomial commitment scheme used in this paper is the construction defined in [52] with polynomials in $\mathbb{F}[x]$.

$\mathsf{Commit}(\mathsf{prk}, \phi(x)) \to (\mathsf{cm}, \mathsf{aux})$**:** Compute $\mathsf{cm} = \sum_{i=1}^{\bar{n}} \phi_i[\tau^i]_1$ and output $(\mathsf{cm}, \phi(x))$. Notice that Commit is the instantiation of AlgoCom.

$\mathsf{OpenPos}(\mathsf{prk}, \phi(x), i) \to (i, \phi(i), \pi_i)$**:** Compute $Q(X) = \frac{\phi(x)-\phi(i)}{x-i}$ and $\pi_i = \sum_{i=1}^{\bar{n}} Q_i[\tau^i]_1$ and output $(i, \phi(i), \pi_i)$.

$\mathsf{VerifyPos}(\mathsf{vrk}, \mathsf{cm}, i, \phi(i), \pi_i) \to 0/1$**:** If $e(\pi_i, [\tau]_2 - i[1]_2) = e([1]_1, \mathsf{cm} - \phi(i)[1]_2)$ output 1 and 0 otherwise.

Theorem 4. *The KZG scheme is a secure PCS (Section 6.5) in the AGM.*

This theorem follows from [28, Appendix B.3] and [11, Section 7].

Set Accumulator. We note that the KZG polynomial commitment scheme can be used as a set accumulator by committing to the characteristic polynomial of the set itself. Given a set S, the *characteristic polynomial* $P_S(x)$ is defined as $P_S(x) = \prod_{s \in S} (x - s)$. Following [71], we remark that the KZG-based set accumulator can be used to prove that a set I is the intersection of two accumulated sets S_1, S_2, that a set Z is the union of two accumulated sets S_1, S_2 and that a set T is the subset of an accumulated set S.

$\mathsf{Accum}(\mathsf{prk}, S) \to \mathsf{acc}$**:** Given S, compute $P_S(x)$ and $(\mathsf{cm}, d) \leftarrow \mathsf{KZG.Commit}(\mathsf{prk}, P_S(x))$. Return cm.

A Set Accumulator is a tuple of PPT algorithms $(\mathsf{Setup}, \mathsf{Accum}, \mathsf{OpenOp}, \mathsf{VerifyOp})$ such that:

- $\mathsf{Accum}(\mathsf{prk}, S) \to \mathsf{acc}$ is a deterministic algorithm that takes as inputs a set S and the public key prk and outputs the accumulator value acc.
- $\mathsf{OpenOp}(\mathsf{prk}, S_1, S_2, \mathsf{op}) \to (S = \mathsf{op}(S_1, S_2), \pi_{\mathsf{op}})$ is defined as follows for the different operations:
  - **Subset:** is a deterministic algorithm that takes as inputs the sets S_1 and S_2 such that $S_1 \subseteq S_2$, the operation $\subseteq$, prk and outputs S_1 and a proof $\pi_\subseteq$ that S_1 is a subset of S_2;
  - **Union:** is a deterministic algorithm that takes as inputs sets S_1 and S_2, prk and outputs $S = S_1 \cup S_2$ and a proof $\pi_\cup$ that S is the union of S_1 and S_2;
  - **Intersection:** is a deterministic algorithm that takes as inputs sets S_1 and S_2, prk and output $S = S_1 \cap S_2$ and a proof $\pi_\cap$ that S is the intersection of S_1 and S_2.
  - **Set Difference:** is a deterministic algorithm that takes as inputs sets S_1 and S_2, prk and output $S = S_1 \setminus S_2$ and a proof $\pi_\setminus$ that S is the intersection of S_1 and S_2.
- $\mathsf{VerifyOp}(\mathsf{vrk}, \mathsf{acc}_S, \mathsf{acc}_{S_1}, \mathsf{acc}_{S_2}, \mathsf{op}, \pi_{\mathsf{op}}) \to 0/1$ is defined as follows for the different operations:
  - **Subset:** is a deterministic algorithm that takes as inputs accumulators acc_S, acc_{S_1}, acc_{S_2}, the operation $\subseteq$, a proof $\pi_\subseteq$ and accepts or rejects, for simplicity we are taking the same interface for this verification, but the verifier ignores the value acc_S and checks that S_1 is a subset of S_2;
  - **Union:** is a deterministic algorithm that takes as inputs accumulators acc_S, acc_{S_1}, acc_{S_2}, vrk, the operation $\cup$, a union proof $\pi_\cup$ and accepts or rejects;
  - **Intersection:** is a deterministic algorithm that takes as inputs accumulators acc_S, acc_{S_1}, acc_{S_2}, vrk, the operation $\cap$, a union proof $\pi_\cap$ and accepts or rejects.
  - **Set Difference:** is a deterministic algorithm that takes as inputs accumulators acc_S, acc_{S_1}, acc_{S_2}, vrk, the operation $\setminus$, a difference proof $\pi_\setminus$ and accepts or rejects.

A set accumulator scheme satisfies the properties of correctness, binding, extractability and homomorphism as defined for the LVC, where $\mathsf{AlgoCom}$ is instantiated with Accum and is the setup algorithm presented in Def.4. The message space $\mathcal{M}$ is $\mathbb{F}^n$.

For the opening and verifications of $\subseteq$, $\cup$, $\cap$, and $\setminus$ we describe them using the following subprotocols:

Quotient: Note that to prove that a given polynomial $P(X)$ is 0 in points $T = (t_1, \ldots, t_\ell)$, it is enough to divide $P(X)$ for the vanishing polynomial $Z_t(X) := \prod_{t \in T}(X - t)$. If the reminder is 0 $Z_t(X) \mid P(X)$ and $P(X)$ is 0 in the points in T. This is equivalent to finding a polynomial $q(X)$ such that $q(X) \cdot Z_t(X) = P(X)$. Given a set $S = \{s_1, \ldots, s_\ell\}$, the commitment to S is the commitment to the polynomial $P_S(X) = \prod_{s \in S}(x - s)$. To prove that S is 0 in T, the prover computes $q(X) = P_S(X)/Z_t(X)$. The prover computes $[P_S(\tau)]_1$, $[Z_t(\tau)]_1$, and $[q(\tau)]_1$. Moreover, using the Fiat-Shamir transformation computes a random value r and the evaluation $y_p = P_S(r)$, $y_q = q(r)$ $y_z = Z_t(r)$ and the respective opening proofs π_p, π_q, π_z. The prover sends $([P_S(\tau)]_1, [Z_t(\tau)]_1, [q(\tau)]_1, y_p, y_q, y_z, \pi_p, \pi_q, \pi_z)$ to the verifier. The verifier recomputes r, verifies the proofs π_p, π_q, π_z and checks that $y_p = y_q \cdot y_z$. For the Schwartz-Zippel lemma this procedure is sound.

Subset: Given the Quotient procedure, the Subset procedure can be implemented by observing that given sets S and T, proving that $T \subseteq S$ is equivalent to say that $P_T(X) = \prod_{t \in T}(X - t)$ divides $P_S(X) = \prod_{s \in S}(X - s)$. Therefore the prover can send to the verifier $([P_S(\tau)]_1, [P_T(\tau)]_1, \pi = ([q(\tau)]_1, y_s, y_q, y_t, \pi_s, \pi_q, \pi_t))$ computed as before and the verifier can perform the same check done in the case of Quotient.

Set difference: Intuitively given sets S and T we can prove their difference through the approach for subsets above: given $P_S(X)$ and $P_T(X)$ the respective polynomials encoding S and T, $P_{S \setminus T} = P_S(X)/P_T(X)$ encodes $S \setminus T$.

Intersection: The Intersection protocol can be computed using the previous protocols, indeed given sets S and T, the intersection $I = S \cap T$ is correct if (i) $I \subseteq S \wedge I \subseteq T$ and (ii) $(S \setminus I) \cap (T \setminus I) = \emptyset$. Condition (i) can be satisfied using the Subset protocol. Condition (ii) can be proved observing that I contains all the elements only if exists polynomials $q_1(X)$ and $q_2(X)$ such that $q_1(X) \cdot P_{S \setminus I}(X) + q_2(X) \cdot P_{T \setminus I}(X) = 1$ (see [71, Lem. 6]). Therefore, for condition (ii) the prover computes $[q_1(\tau)]_1$, $[q_2(\tau)]_1$, computes a random r and sends in addition to the values for the verification of condition (i) also $([q_1(\tau)]_1, [q_2(\tau)]_1, P_{S \setminus I}(r), P_{T \setminus I}(r), q_1(r), q_2(r), \pi_s, \pi_t, \pi_{q_1}, \pi_{q_2})$, where $\pi_s, \pi_t, \pi_{q_1}, \pi_{q_2}$ are the proof of opening of the polynomials in r. The verifier checks the subset condition as done in the Subset procedure and in addition checks the proofs $\pi_s, \pi_t, \pi_{q_1}, \pi_{q_2}$ and the equation $q_1(r) \cdot P_{S \setminus I}(r) + q_2(r) \cdot P_{T \setminus I}(r) = 1$.

Complement (only for known sets of polynomial size $\bar{n}$): [21] If the maximum dimension of the sets is known and is $\bar{n}$, it is possible compute the complement of a set S as $U - S$, where $U = [\bar{n}]$ is the set containing all the $\bar{n}$ elements. Since the verifier knows $\bar{n}$, this protocol is equivalent to the Set difference protocol.

Union: Intuition: to implement a union protocol on sets (of maximum size $\bar{n}$), we can exploit De Morgan's law stating that $S \cup T = \neg(\neg S \cap \neg T)$, where the prover implements $\neg$ with the complement protocol.

Theorem 5. *The set accumulator above is a secure set accumulator scheme (**??**) in the AGM under the q-BSDH assumption.*

7 Background: Cryptographic Verifiable Databases

The following definitions are a minor variation of those from [69]. Below, we assume the existence of an algorithm $\mathsf{ValidDB}(\mathsf{db})$ that on input an alleged database string db and (implicitly) a size parameter n checks whether db is a valid database of size n.

[21] We will use this building block in our construction only for the case of sets of rows, which satisfy this requirement. Wherever we apply sets on *objects* (rather than rows) we will not use the complement as a subprotocol.

Definition 7 (VDB). *A (non-interactive) VDB consists of four probabilistic polynomial-time (PPT) algorithms: a setup algorithm* Setup *generating the public parameters of the scheme, an algorithm* PreProc *to produce a digest from the database, an answering algorithm for the server* AnsQry*, and a client algorithm* VerQry*.*

$\mathsf{Setup}(1^\lambda) \to (\mathsf{prk}, \mathsf{vrk})$**:** *is a probabilistic algorithm that takes as input the security parameter λ and outputs a pair of keys* $(\mathsf{prk}, \mathsf{vrk})$ *(for proving and verification respectively).*

$\mathsf{PreProc}(\mathsf{prk}, \mathsf{db}) \to (c, \mathsf{state})$**:** *is a deterministic algorithm that takes in input* prk*, and a valid database* db*. It produces a (public) commitment c and some internal state information* state*.*

$\mathsf{AnsQry}(\mathsf{prk}, q) \to (a, \pi)$**:** *is a deterministic algorithm that can be invoked with a query $q \in \mathcal{Q}$ and the setup information state as input. The corresponding output is an answer/proof pair (a, π), where $a = Q(q, \mathsf{db})$.*

$\mathsf{VerQry}(\mathsf{V}, q, a, c, \pi)$**:** *is a deterministic algorithm that receives as input* V*, the commitment c, a query q and an answer/proof pair (a, π).* VerQry *"accepts" or "rejects" the proof π.*

Definition 8. *A VDB is consistent if it is complete and sound:*

- ***Completeness**: For every valid database* db *and query $q \in \mathcal{Q}$, if the setup is performed correctly then with overwhelming probability,* AnsQry *outputs both the correct answer and a proof which is accepted by* VerQry*. Formally, for all $\lambda \in \mathbb{N}$, $q \in \mathcal{Q}$,* db *s.t.* $\mathsf{ValidDB}(\mathsf{db}) = 1$*, the following holds:*

$$\Pr\left[\begin{array}{l|l} \mathsf{VerQry}(\mathsf{V}, q, a, c, \pi) = \text{"accept"} & (\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{Setup}\left(1^\lambda\right) \\ \wedge\, a = Q(q, \mathsf{db}) & : \ (c, \mathsf{state}) \leftarrow \mathsf{PreProc}(\mathsf{pp}, \mathsf{db}) \\ & (a, \pi) \leftarrow \mathsf{AnsQry}(q, \mathsf{state}) \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

- ***(Computational) Soundness**: For every PPT adversary $\mathcal{A}$, run $\mathcal{A}$ to obtain database* db*, a commitment c and a list of triples (q_i, a_i, π_i). We say $\mathcal{A}$ acts consistently if $a_i = Q\left(q_i, \mathsf{db}\right)$ for all i for which π_i is a valid proof. The protocol is sound if all PPT adversaries $\mathcal{A}$ act consistently. Formally, for all $\lambda \in \mathbb{N}$, PPT $\mathcal{A}$, the following holds:*

$$\Pr\left[\begin{array}{l|l} \mathsf{ValidDB}(\mathsf{db}) = 1 \wedge & (\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{Setup}\left(1^\lambda\right) \\ \forall i \in [t]\, b_i = 1 \Rightarrow a_i = Q\left(q_i, \mathsf{db}\right) & : \ \left(\mathsf{db}, c, \{(q_i, a_i, \pi_i)\}_{i \in [t]}\right) \leftarrow \mathcal{A}(\mathsf{prk}) \\ & b_i \leftarrow \mathsf{VerQry}\left(\mathsf{vrk}, q_i, a_i, c, \pi_i\right), i \in [t] \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

We say that a VDB is succinct if its proof adds no overhead to the substantial overhead to the verifier.

Definition 9 (Succinctness of a VDB). *We say that a VDB is succinct if for any database and query* qry *we have $|\pi| = O_\lambda(|\mathsf{resp}| + |\mathsf{qry}|)$ (where* resp *is a correct response to the query and π is the proof from an honest prover) and so is the verification running time.*

Above, the size of a query consists of the number of columns referenced in it (even implicitly through a $\mathsf{SELECT}\,\star$).

8 A New Information-Theoretic Model for Idealized VDB

8.1 The idealized model

In this section we present an intermediate notion of the VDB primitive described in Section 7. We will *idealize* this notion a bit to make it simpler to construct and to reason about it. For example, while the notion from Section 7 is a *cryptographic* one—it is secure against a computationally bounded adversary—here we will consider an information-theoretic notion against an adversary that is limited in other ways. This adversary will be in particular able to manipulate two basics

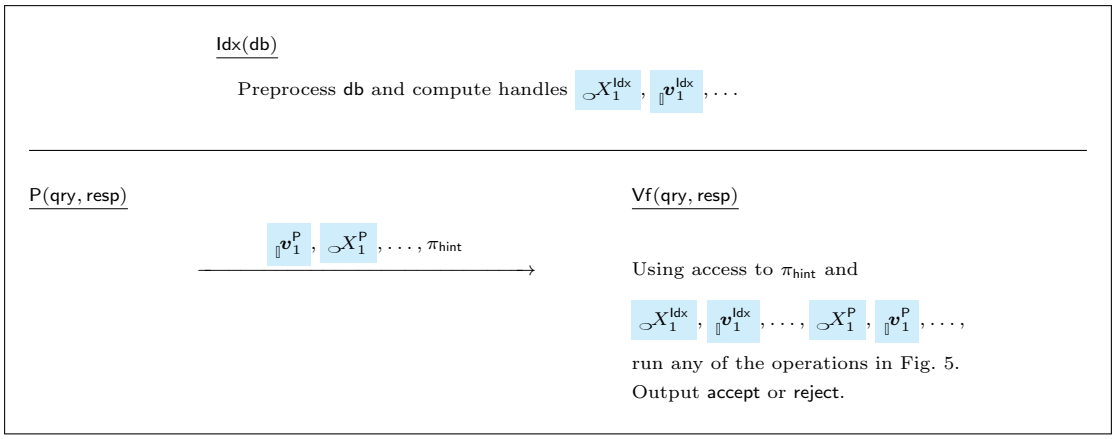

Fig. 4: Interaction in our idealized protocols (informal). Formal details in Fig. 6.

types of objects, *sets* and *vectors*, so that the verifier can check these manipulations without reading the whole content of these sets and vectors. In particular the verifier will access only *pointers* ("handles") to these objects and check these requirements through them. Surprisingly, a very limited vocabulary of core "checks" will allow a broad range of queries in an actual relational database later. We expand on our approach in the remainder of this section and in Fig. 5.

From databases to handles. From a general perspective, a database db contains *values* indexed via *keys* (e.g., primary keys in a relational table or row numbers in a spreadsheet table). The concept of set and slice handles introduced in Section 2 are particularly apt for denoting the data stored inside a database and for representing queries over the database. Here we briefly recall the intuition behind the concepts of set and slice handles:

- Given a set X of values, the *set handle* of X is denoted as $\subset X$. This kind of handle denotes the values contained in X. Two special set handles denote the set of all row indices, written as $\subset *$, and the empty set, written as $\subset \emptyset$. The handle denoting the set of all rows is kept implicit from the context.
- Given a vector $\boldsymbol{v}$ of values, the *slice handle* of $\boldsymbol{v}$ is denoted as $[\boldsymbol{v}$. This kind of handle denote the list of values contained in $\boldsymbol{v}$.

The flow of an idealized VDB. At the high-level an interaction during an idealized VDB execution proceeds as follows (see Fig. 4):

- *Pre-processing*: from the database we produce several sets and slices as functions of the database, as well as their respective handles. We refer to them as *database* or *indexing* handles; the verifier will have access to these through the next stages.
- *Proving*: on input a query and the database, the prover produces a set of new handles and sends them to the verifier together with a hint, a proper string message (not a handle) whose size will usually be sublinear in the response size and that acts as auxiliary information.
- *Verifying/Retrieving*: The verifier manipulates and performs checks on the handles (both indexing and response ones) using the hint. It can also perform and retrieve information from the slice handles (see remainder of section). It finally reconstructs the response (and accept), or rejects.

While we focus on a non-interactive solution where the idealized prover sends a single batch of handles/messages, our framework can be immediately extended to additional rounds. Also, if the building blocks used to compile (see remainder of section) an idealized VDB are non-interactive, the resulting cryptographic VDB will be non-interactive as well.

Performing tests and reads over handles. We support a small core set of tests and operations that can be performed over handles (see Fig. 5): given set handles, the verifier can check

whether one "denotes" the union/intersection of the sets corresponding to two other handles; similarly for set containment. As for slice handles, one can test whether the inner product of the vectors behind the two handles is equal to a given value. A test involving both types of handles is a zero-test: *does a slice handle "contains" the value* 0 *in the positions denoted by a set handle?* New slice handles may be created via homomorphic operations like summation and scalar multiplication. From these primitive/core tests, one can derive more complex ones. In Table 3 we describe how to obtain the derived operations mentioned in Fig. 5 and others that we will use in our final construction (and that are generally support by the core ones). The notation $\supset^*$ denotes the handle for the set *of all rows*.

Remark 2 (A note on numerical types and range checks). We assume that sets and slices are defined over a field. Nonetheless when performing range queries or other additions we will be working on bounded integers. We will make two assumptions (which are easy to instantiate and have also been used in prior work on VDBs) that will make our approaches based on homomorphism work: the fact that at preprocessing time commitments to integer values are generated honestly and will be limited to a certain range; the fact that the number of homomorphic operations required by any given query will not go beyond that range even in the worst case.

Finally, we remind the reader that given a test of the type $\boldsymbol{v} \overset{?}{\geq} 0$[22], it is easy, through homomorphism, to obtain tests for $\boldsymbol{v} \overset{?}{\geq} \alpha$ (which becomes $\boldsymbol{v} - \alpha \cdot \mathbf{1} \overset{?}{\geq} 0$) and for $\boldsymbol{v} \overset{?}{\leq} \alpha$ (which becomes $-\boldsymbol{v} \overset{?}{\geq} -\alpha$).

Remark 3 (On security, batching and efficiency). For simplicity, we present simple instantiations that are correct and sound by immediate inspection. Some of the instantiations will perform redundant checks in order not to overcomplicate their description. However, it is very easy to improve the concrete efficiency of many of these building blocks (e.g., eqSet or our range checks) by simple batching techniques due to homomorphism and basic additional observations.

Finally, one can *read* values at specific positions in a slice. Notice that the Verifier actually accesses data is through the retrieval operation. It is reasonable for retrieval to contain *at least* enough information to reconstruct the response to the query. What we will show in our construction is that the retrieved data may include *no more* than that amount of information and still provide verifiability.

Semantics We require two properties from such an idealized protocol:

- *completeness*: there is always a way to use our "handle" language (the one in Fig. 5) to convince the verifier of the correct response.
- *soundness*: for any supported query/database, there is no set of handles a malicious prover can "send" to make the client accept a wrong response[23].

8.2 The full formalization of idealized protocols

We now formalize the semantics of idealized VDBs.

Definition 10 (Idealized Prover/Verifier and their interaction). *We denote by an idealized algorithm (prover or verifier) a randomized* [24] *machine with access to special oracles, described in Fig. 6 (and more informally in Fig. 4). We denote the verifier's decision at the*

[22] Here we abuse notation to denote the AND of element-wise comparison, e.g., $\boldsymbol{v} \overset{?}{\geq} 0$ tests whether all elements in $\boldsymbol{v}$ are non-negative.

[23] This is under the guarantee that all tests on these handles work as expected. This will be enforced through cryptographic compilation in the next sections.

[24] The honest *idealized* prover/verifier should operate in polynomial time. However, our notion of soundness will be quite strong and state that the protocol is secure even against non polynomial time machines. This is similar to what is a achieved in other idealized models such as [1, 7, 11, 17, 28, 42].

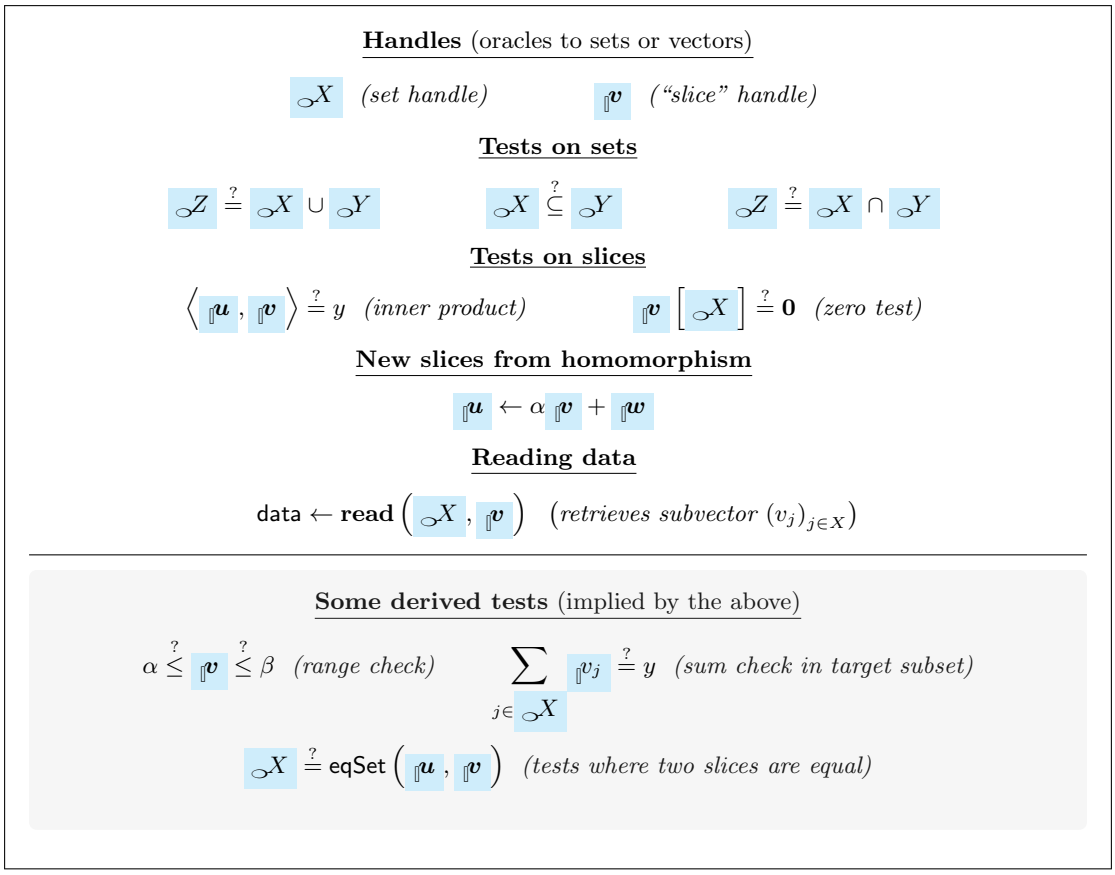

Fig. 5: Oracles and operations supported by the verifier in our idealized model. Only the "core" operations (above the line) are the ones we are actually *assuming* in the model. The remaining tests (below the line) are *implied* by the core ones (with overhead independent of $|\mathsf{db}|$); we stress these are just derived operations that are useful shortcuts when describing a protocol and that the actual core model only includes the part not in gray. We implicitly assume that if the verifier holds a set X (usually small), this can be treated as a handle $\mathcal{O}X$ and used as such in the tests above. See Fig. 6

end of the interaction flow (composed of local oracle invocations and a single round of message passing) in Fig. 6 as

$$\mathsf{b} \leftarrow \left\langle \mathsf{P}^{\mathcal{O}_\mathsf{P}}, \mathsf{Vf}^{\mathcal{O}_\mathsf{Vf}} \right\rangle (\mathsf{qry}, \mathsf{resp})$$

where $\mathsf{b} \in \{0,1\}$ *denotes rejection or acceptance.*

Remark 4. We assume all algorithms implicitly takes as input a security parameter λ. For this idealized model this will play uniquely the role of a *statistical* security parameter (see Footnote 24). We will also implicitly assume it to be useful as offering some size bound parameter (polynomial in λ for some fixed polynomial) to the honest algorithms (e.g., maximum size of data structures to allocate internally, etc).

Completeness For all $\mathsf{db} \in \mathcal{DB}$, for all queries qry and responses resp s.t. $\mathsf{SatisfiesQry}(\mathsf{db}, \mathsf{qry}, \mathsf{resp}) = \mathsf{true}$, then the following holds:

$$\Pr\left[\left\langle \mathsf{P}^{\mathcal{O}_\mathsf{P}}, \mathsf{Vf}^{\mathcal{O}_\mathsf{Vf}} \right\rangle (\mathsf{qry}, \mathsf{resp}) = 1 \;:\; \begin{array}{c} (\mathsf{T}_{[\![]\!]}^{\mathrm{handles}}, \mathsf{T}_{\mathcal{O}}^{\mathrm{handles}}) \leftarrow \mathsf{Idx}(\mathsf{db}) \\ \mathcal{O}_\mathsf{P}, \mathcal{O}_\mathsf{Vf} \text{ as in Fig. 6} \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Soundness For all idealized provers $\tilde{\mathsf{P}}$, for all $\mathsf{db} \in \mathcal{DB}$, for all queries qry and responses resp s.t. $\mathsf{SatisfiesQry}(\mathsf{db}, \mathsf{qry}, \mathsf{resp}) = \mathsf{false}$, then the following holds:

$$\Pr\left[\left\langle \tilde{\mathsf{P}}^{\mathcal{O}_{\mathsf{P}}}, \mathsf{Vf}^{\mathcal{O}_{\mathsf{Vf}}} \right\rangle(\mathsf{qry}, \mathsf{resp}) = 1 \ : \ \begin{array}{c} (\mathsf{T}_{\mathbb{I}}^{\mathrm{handles}}, \mathsf{T}_{\mathcal{O}}^{\mathrm{handles}}) \leftarrow \mathsf{Idx}(\mathsf{db}) \\ \mathcal{O}_{\mathsf{P}}, \mathcal{O}_{\mathsf{Vf}} \text{ as in Fig. 6} \end{array}\right] \leq \mathsf{negl}(\lambda)$$

8.3 From core to derived operations in idealized VDBs

We now describe how the limited sets of core operations in Fig. 5 can be used to obtain more "complex" operations that can be useful in a verifiable database setting. By analogy, our approach is tantamount to defining a "RISC-like" set of primitive instructions (the core operations in our model) and then deriving more complex ones as an aggregate of the former.

In producing our derived operations we use the following core techniques:

- *homomorphism*: this is often used to check whether two vectors are equal in a set of positions by substracting them; we also apply homomorphism to produce a vector of values all equal to the same value α by, e.g., multiplying α by the slice handle of all ones.
- *complement set and "partition trick"*: when checking that a property is true in a target set X and only in that set, we let the prover send (a handle to) the set complement set $\bar{X}$ and check the property is true in X but false in $\bar{X}$.
- *incrementality*: we occasionally use each of the new derived tests as a building blocks in others.

The complete set of derived operations is contained in Table 3.

Nullifying test The key insight is that by checking the difference vector Δ is zero on the complement of the target subset X_0, combined with checking that $\bar{u}$ is zero on X_0, the verifier ensures the property without needing to explicitly compute the indicator function. Notice that the checks correspond exactly to the property on the left and correctness and soundness hold unconditionally.

Range check Intuitively, this test works by letting the prover send slices of bits that, when appropriately scaled (by respective powers of two) and summed, can be seen to provide the target slice $\boldsymbol{v}$. The verifier can check the "bit-ness" of the received slices by having the prover send the handle sets of positions where these slices are equal to 0; the verifier then applies zeroing (the instruction we described in the previous bullet) in these claimed positions and checks equality to the claimed bit slices.

Strict sign check Here we apply this observation: the subvector $\boldsymbol{v}_{X_>}$ is positive everywhere if and only if $\boldsymbol{v}_{X_>} - \mathbf{1}$ is non-negative everywhere. We emulate the "subvectoring" operation by nullifying anything outside $X_>$. The verifier then applies the range check from the previous item.

Positions where two slices are equal First observe that two slices are equal in the set of positions X_0 if and only there if and only *i)* if their difference is zero in X_0 and *ii)* non-zero everywhere else. Testing *i)* is straightforward (zero-testing is a core operation). To check *ii)* we first let the prover "declare" what positions are non-zero and positive and which positions are non-zero and negative. We then apply the test from the last item twice (to check that a set of positions is strictly negative we first negate the slice through homomorphism and then check it is strictly positive).

Sumcheck within a target subset X We let the prover send the indicator vector of X (which is appropriately checked by the verifier) and then we apply an inner product between slices (a core operation).

Pre-image check Here, given a value α we want to test whether it is true that the position in which a slice $\boldsymbol{v}$ assumes value α corresponds exactly to a claimed set of positions X_α (we call it a *preimage* check because this is the preimage of the value α when seeing the slice as a function $\boldsymbol{v} : [n] \to \mathbb{F}$). This check will be useful in JOINs and is a natural tool in other SQL queries. Its implementation is straightforward given one the derived check for positions where two slices are equal: the verifier produces a slice $(\alpha, \alpha, \ldots, \alpha)$ and then checks that the latter and $\boldsymbol{v}$ assume equal values in all and only the positions in X_α.

Table 3: How to obtain derived operations in our model (see also Fig. 5). Above we assume that the universe of all indices ($\circlearrowright *$) and the slice comprising of all ones ($\mathbb{1}$) are available to the verifier from the indexing process. (Table continues on next page)

"Nullifying" test $\mathbb{u} \stackrel{?}{=} \mathbb{v}\left[\circlearrowright X_0 \to 0\right]$ i.e., $\forall j\ u_j \stackrel{?}{=} v_j \cdot (1 - \mathbb{1}_{X_0}(j))$	Prover sends: $\circlearrowright \bar{X}_0 := \circlearrowright * \setminus \circlearrowright X_0$ Verifier defines $\mathbb{\Delta} \leftarrow \mathbb{u} - \mathbb{v}$ and then checks: $\circlearrowright \bar{X}_0 \stackrel{?}{=} \circlearrowright * \setminus \circlearrowright X_0$ $\mathbb{u}\left[\circlearrowright X_0\right] \stackrel{?}{=} 0$ *("is $u_j = 0$ for each $j \in X_0$?")* $\mathbb{\Delta}\left[\circlearrowright \bar{X}_0\right] \stackrel{?}{=} 0$ *("is $u_j = v_j$ for each $j \notin X_0$?")*
Range check $\mathbb{v} \stackrel{?}{\in} [0, 2^\ell)$	Let $v_j^{(i)}$ denote the i-th bit of v_j, i.e., for each j, $v_j = \sum_i 2^{i-1} v_j^{(i)}$ Let $\boldsymbol{v}^{(i)} := \left(v_1^{(i)}, \ldots, v_m^{(i)}\right)$ for $i \in [\ell]$, with $m := \lvert\boldsymbol{v}\rvert$ Let $X_0^{(i)} := \{j : v_j^{(i)} = 0\}$ for $i \in [\ell]$ *(NB: $v_j^{(i)} = 1$ for all $j \notin X_0^{(i)}$)* Prover sends: $\mathbb{v}^{(1)}, \ldots, \mathbb{v}^{(\ell)}$ $\circlearrowright X_0^{(1)}, \ldots, \circlearrowright X_0^{(\ell)}$ Verifier defines $\mathbb{\Delta} \leftarrow \sum_i \left(2^{i-1}\ \mathbb{v}^{(i)}\right) - \mathbb{v}$ and then checks: $\mathbb{\Delta}[\circlearrowright *] \stackrel{?}{=} 0$ *(equivalent to $\sum_i \left(2^{i-1}\boldsymbol{v}^{(i)}\right) \stackrel{?}{=} \boldsymbol{v}$)* $\mathbb{v}^{(i)} \stackrel{?}{=} \mathbb{1}\left[\circlearrowright X_0^{(i)} \to 0\right]$ for all $i \in [\ell]$ *("are these bits?")*
Strict sign check within *target subset* $\mathbb{v}\left[\circlearrowright X_>\right] \stackrel{?}{>} 0$	Let $\boldsymbol{u}_1 := \left(\mathbb{1}_{X_>}(1), \ldots, \mathbb{1}_{X_>}(m)\right)$, with $m := \lvert\boldsymbol{v}\rvert$ *(indicator vector for $X_>$)* Let $\boldsymbol{u}_{\text{zero}}$ be such that $u_{\text{zero},j} = \begin{cases} v_j & \text{if } j \in X_> \\ 0 & \text{if } j \notin X_> \end{cases}$ Prover sends: $\mathbb{u}_1, \mathbb{u}_{\text{zero}}, \circlearrowright \bar{X}_> := \circlearrowright * \setminus \circlearrowright X_>$ Verifier defines $\mathbb{v}_{\geq 0} := \mathbb{u}_{\text{zero}} - \mathbb{u}_1$ and then checks: $\mathbb{u}_1 \stackrel{?}{=} \mathbb{1}\left[\circlearrowright X_> \to 0\right]$ *("is this the indicator vector?")* $\mathbb{u}_{\text{zero}} \stackrel{?}{=} \mathbb{v}\left[\circlearrowright \bar{X}_> \to 0\right]$ *("does this satisfy the def. of $\boldsymbol{u}_{\text{zero}}$?")* $\mathbb{v}_{\geq 0} \stackrel{?}{\geq} 0$ $\quad$ $\circlearrowright \bar{X}_> \stackrel{?}{=} \circlearrowright * \setminus \circlearrowright X_>$
Tests where two slices are equal $\circlearrowright X_0 \stackrel{?}{=} \mathsf{eqSet}\left(\mathbb{u}, \mathbb{v}\right)$ i.e., we test: $X_0 \stackrel{?}{=} \{j : u_j = v_j\}$	Let $X_+ := \{j : u_j > v_j\}, X_- := \{j : u_j < v_j\}$. Prover sends: $\circlearrowright X_+, \circlearrowright X_-$ Verifier defines $\mathbb{\Delta} \leftarrow \mathbb{u} - \mathbb{v}$ and then checks: That $\mathbb{X}_0, \mathbb{X}_+, \mathbb{X}_-$ partition $\circlearrowright *$ (via basic set handle tests) $\mathbb{\Delta}\left[\circlearrowright X_0\right] \stackrel{?}{=} 0$ $\quad$ $\mathbb{\Delta}\left[\circlearrowright X_+\right] \stackrel{?}{>} 0$ $\quad$ $-\mathbb{\Delta}\left[\circlearrowright X_-\right] \stackrel{?}{>} 0$
Sum check within target subset $\sum_{j \in \circlearrowright X} \mathbb{v}_j \stackrel{?}{=} y$	Let $\boldsymbol{u}_1 := \left(\mathbb{1}_X(1), \ldots, \mathbb{1}_X(m)\right)$, $m := \lvert\boldsymbol{v}\rvert$ *(indicator vector for X)* Prover sends: $\mathbb{u}_1, \circlearrowright \bar{X} := \circlearrowright * \setminus \circlearrowright X$ Verifier checks: $\mathbb{u}_1 \stackrel{?}{=} \mathbb{1}\left[\circlearrowright \bar{X} \to 0\right]$ *("is it the indicator vector?")* $\left\langle \mathbb{v}, \mathbb{u}_1 \right\rangle \stackrel{?}{=} y$ *(checks actual sum)* $\circlearrowright \bar{X} \stackrel{?}{=} \circlearrowright * \setminus \circlearrowright X$

Pre-image check	Verifier defines $\llbracket \boldsymbol{u}_\alpha \rrbracket := \alpha \llbracket \mathbf{1} \rrbracket$ and checks:
$\subset\!\!\cdot X_\alpha \stackrel{?}{=} \alpha^{-1}(\llbracket \boldsymbol{v} \rrbracket)$	$\subset\!\!\cdot X_\alpha \stackrel{?}{=} \mathsf{eqSet}(\llbracket \boldsymbol{u}_\alpha \rrbracket, \llbracket \boldsymbol{v} \rrbracket)$
where: $\alpha^{-1}(\boldsymbol{v}) := \{j : v_j = \alpha\}, \alpha \in \mathbb{F}$	

9 Our Compilation Results

9.1 Accumulators and LVC with Zero-Testing: Definition

Below we assume a commitment scheme to vectors $\mathsf{Setup}, \mathsf{CommitVec}$ and a commitment to sets (an accumulator) $\mathsf{Setup}, \mathsf{Accum}$ (the setup algorithm is the same) such that they are both binding in their respective domains. Below, $\bar{n}$ is the size of the vectors; for simplicity we assume it is some well-defined function (a polynomial) of λ.

The property we are interested in can be thought of as an "extended" primitive of both accumulators and (subvector-opening) vector commitments: intuitively, this primitive is such that given $\mathsf{cm}_{\boldsymbol{v}}$ and acc_X where $X \subseteq [|\boldsymbol{v}|]$, then a prover is able to convince a verifier that $v_j = 0$ for all $j \in X$.

Definition 11. *Formally, we say that an LVC (Section 6.3) and accumulation (**??**) scheme (jointly) support zero-testing on accumulated sets if there exist the following two efficient algorithms satisfying the properties below (the $\star$ is just to stress that the algorithms are new and not standard in this type of authenticated data structures):*

- $\mathsf{PrvSubvecIsZero}^\star\,(\mathsf{prk}, \mathsf{aux}_{\boldsymbol{v}}, \mathsf{aux}_X) \to \pi$
- $\mathsf{VfySubvecIsZero}^\star\,(\mathsf{vrk}, \mathsf{cm}_{\boldsymbol{v}}, \mathsf{acc}_X, \pi) \to 0/1$

Correctness *For all $\lambda \in \mathbb{N}, \boldsymbol{v} \in \mathbb{F}^{\bar{n}}, X \subseteq [\bar{n}]$ such that $\boldsymbol{v}_j = 0$ for all $j \in X$, it holds that the following inequality is true:*

$$\Pr\left[\mathsf{VfySubvecIsZero}^\star(\mathsf{vrk},\ \mathsf{cm}_{\boldsymbol{v}}, \mathsf{acc}_X, \pi) = 1\ :\ \begin{array}{l}(\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{Setup}(1^\lambda)\\ (\mathsf{cm}_{\boldsymbol{v}}, \mathsf{aux}_{\boldsymbol{v}}) \leftarrow \mathsf{VC.Com}(\mathsf{prk}, \boldsymbol{v})\\ (\mathsf{acc}_X, \mathsf{aux}_X) \leftarrow \mathsf{Acc.Com}(\mathsf{prk}, X)\\ \pi \leftarrow \mathsf{PrvSubvecIsZero}^\star\,(\mathsf{prk}, \mathsf{aux}_{\boldsymbol{v}}, \mathsf{aux}_X)\end{array}\right] \geq 1 - \mathsf{negl}(\lambda).$$

Extractability *For all $\lambda \in \mathbb{N}$, for all PPT $\mathcal{A}$ there exists an efficient extractor $\mathcal{E}_{\mathcal{A}}$ such that the following holds:*

$$\Pr\left[\begin{array}{l}\mathsf{VfySubvecIsZero}^\star\,(\mathsf{vrk}, \mathsf{cm}_{\boldsymbol{v}}, \mathsf{acc}_X, \pi) = 1 \wedge\\ \big(\mathsf{acc}_X \neq \mathsf{Acc.Com}(\mathsf{prk}, X) \vee\\ \mathsf{cm}_{\boldsymbol{v}} \neq \mathsf{VC.Com}(\mathsf{prk}, \boldsymbol{v}) \vee\\ \exists j \in X\ : v_j \neq 0\big)\end{array}\ :\ \begin{array}{l}(\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{Setup}(1^\lambda)\\ (\mathsf{cm}_{\boldsymbol{v}}, \mathsf{acc}_X, \pi) \leftarrow \mathcal{A}\,(\mathsf{prk})\\ (\boldsymbol{v}, X) \leftarrow \mathcal{E}_{\mathcal{A}}(\mathsf{prk})\end{array}\right] \leq \mathsf{negl}(\lambda).$$

9.2 Instantiating the zero-testing property with polynomial commitments

We present a general construction based on polynomial commitments.

Intermezzo: LVC and accumulators from polynomial commitments Consider a linear-map vector commitment where, in order to commit to a vector $\boldsymbol{v}$ we commit to $p_{\boldsymbol{v}}(X) := \sum_{i \in [|\boldsymbol{v}|]} v_i \lambda_i(X)$. Consider also an accumulator where we commit to a set S by committing to $p_S(X) := \prod_{s \in S}(X - s)$. Notice that the ADSs we described in Section 6.6 satisfy this property immediately (they are implicitly polynomial commitments—KZG commitments in particular—to the polynomials above).

Intermezzo: the polynomial remainder theorem as key tool Recall from standard algebra that for any polynomial p, set $S = \{s_1, \ldots, s_\ell\}$ of size ℓ, we have that $p(s_i) = 0$ for all $i \in [\ell]$ iff $Z_S(X) \mid p(X)$ where $Z_S(X) := \prod_{s \in S}(X - s)$ is the vanishing polynomial in S. This is equivalent to the existence of a polynomial $q(X)$, such that $q(X) \cdot Z_S(X) = p(X)$. This suggests the following protocol that applies standard techniques including Schwartz-Zippel to the setting above (observing that our p of interest is $p_{\boldsymbol{v}}$ and that the polynomial p_S encoding the set *is* the vanishing polynomial Z_S).

A protocol for zero-testing in accumulated sets:

- To prove that $\boldsymbol{v}$ is zero in all indices in set S (for $\boldsymbol{v}$ and S respectively committed and accumulated as above, i.e. with underlying polynomial $p_{\boldsymbol{v}} := \sum_{i \in [|\boldsymbol{v}|]} v_i \lambda_i(X)$ and $p_S := Z_S(X)$): compute $q(X) := p_{\boldsymbol{v}}(X)/p_S(X)$; send polynomial commitment cm_q; verifier samples a random r; prover sends $(y_q := q(r), y_v := p_{\boldsymbol{v}}(r), y_S := p_S(r), \pi_q, \pi_v, \pi_S)$ where each $\pi_\star$ is a polynomial evaluation proof for the respective values/polynomials (and might be batched in principle).
- The verifier checks that $y_v = y_q \cdot y_S$ and checks all polynomial evaluation proofs.

The protocol we just presented has proofs consisting of a constant number of polynomial opening (and a constant number of field elements); it is non-interactive and extractable in the ROM using the Fiat-Shamir heuristic if the underlying polynomial commitment is extractable and homomorphic.

The protocol above in principle works immediately on the (vector and set) commitments we are interested in (those described in Section 6.6). However, in the next paragraphs we will also present another instantiation directly from pairings and without use of the random oracle.

9.3 An instantiation of zero-testing directly from pairings

The scheme we present now represent a way to perform the same checks as those in Section 9.2, but through the "shortcuts" that pairings allow us. Recall that, in this particular case, the commitment to $\boldsymbol{v}$ and the accumulator to S are KZG commitments, that is they are respectively $\mathsf{cm}_{\boldsymbol{v}} := [p_{\boldsymbol{v}}(\tau)]_1$ and $\mathsf{acc}_S := [p_S(\tau)]_1$ (with τ being the secret element in the setup).

- The prover computes $\pi_{2,S} := [p_S(\tau)]_2$ and $\pi_q := [q(\tau)]_1$ with $q(X) := p_{\boldsymbol{v}}(X)/p_S(X)$.
- The verifier performs two checks:
  - $e(\pi_q, \pi_{2,S}) \stackrel{?}{=} e(\mathsf{cm}_{\boldsymbol{v}}, [1]_2)$
  - $e([1]_1, \pi_{2,S}) \stackrel{?}{=} e(\mathsf{acc}_S, [1]_2)$

The security of the protocol follows immediately from observations similar to those in the proof of Theorem 2.

9.4 The actual compiler

The executed $(\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{Setup}(1^\lambda)$, is the algorithm defined in Section 6.6. Notice that this setup is used by all cryptographic tools in our system and can be reused among multiple executions.

$\mathsf{PreProc}(\mathsf{prk}, \mathsf{db})$ is implemented as follows: for each column $\boldsymbol{u_i}$ in input to PreProc computes $\mathsf{cm}_i \leftarrow \mathsf{LVC.CommitVec}(\mathsf{prk}, \boldsymbol{u_i})$. For each set of indices X_i in input computes $\mathsf{acc}_i \leftarrow \mathsf{SA.Accum}(\mathsf{prk}, X_i)$. The output (c, state) is defined as follows: c contains all the cm_i computed

together with all the acc_i, state contains the additional information for the internal computation. Notice that to have a constant size digest of db, it is possible to create a vector commitment with LVC of all vector commitments and set accumulators.

The algorithms $\mathsf{AnsQry}(\mathsf{prk}, q)$ and $\mathsf{VerQry}(\mathsf{V}, q, a, c, \pi)$ are implicitly reported in the compiler that can be found in Table 4.

Table 4: Compilation of idealized operations through cryptographic building blocks.

Idealized Operation	**Cryptographic Implementation**
Produce and send new slice handle $[\![\boldsymbol{v}]\!]$	Send $\mathsf{cm}_{\boldsymbol{v}} \leftarrow \mathsf{LVC.CommitVec}(\mathsf{prk}, \boldsymbol{v})$
Produce and send new set handle $\bigcirc X$	Send $\mathsf{acc}_X \leftarrow \mathsf{SA.Accum}(\mathsf{prk}, X)$
$\bigcirc Z \stackrel{?}{=} \bigcirc X \cap \bigcirc Y$	Prover computes $\mathsf{SA.OpenOp}(\mathsf{prk}, X, Y, \cap) \to (Z, \pi)$. Verifier checks $\mathsf{SA.VerifyOp}(\mathsf{vrk}, \mathsf{acc}_Z, \mathsf{acc}_X, \mathsf{acc}_Y, \cap, \pi)$
$\bigcirc Z \stackrel{?}{=} \bigcirc X \cup \bigcirc Y$	Same as $\cap$, using $\cup$ operator
$\bigcirc X \stackrel{?}{\subseteq} \bigcirc Y$	Prover computes $\mathsf{SA.OpenOp}(\mathsf{prk}, X, Y, \subseteq) \to \pi$ Verifier checks $\mathsf{SA.VerifyOp}(\mathsf{vrk}, \mathsf{acc}_X, \mathsf{acc}_Y, \subseteq, \pi)$
$\langle [\![\boldsymbol{u}]\!], [\![\boldsymbol{v}]\!] \rangle \stackrel{?}{=} y$	Prover computes $\pi \leftarrow \mathsf{LVC.OpenLin}(\mathsf{prk}, \boldsymbol{u}, \boldsymbol{v}, y)$ Verifier checks $\mathsf{LVC.VerifyLin}(\mathsf{vrk}, \mathsf{cm}_u, \mathsf{cm}_v, y, \pi)$
$[\![\boldsymbol{u}]\!] \leftarrow \alpha [\![\boldsymbol{v}]\!] + [\![\boldsymbol{w}]\!]$	Uses homomorphism of LVC
$[\![\boldsymbol{v}]\!] \leftarrow (v_1, \ldots, v_n)$	$\mathsf{LVC.CommitVec}(\mathsf{prk}, (v_1, \ldots, v_n))$
$[\![\boldsymbol{u}]\!][\bigcirc X] \stackrel{?}{=} 0$	Prover computes $\pi \leftarrow \mathsf{LVC.PrvSubvecIsZero}^{\star}(\mathsf{prk}, \boldsymbol{u}, X)$ Verifier checks $\mathsf{LVC.VfySubvecIsZero}^{\star}(\mathsf{vrk}, \mathsf{cm}_{\boldsymbol{u}}, \mathsf{acc}_X, \pi)$
$\mathsf{data} \leftarrow \textbf{read}(\bigcirc X, [\![\boldsymbol{v}]\!])$	Prover sends X, $\pi \leftarrow \mathsf{LVC.OpenSub}(\mathsf{prk}, C, X, \mathsf{data})$ Verifier checks $\mathsf{LVC.VerifySub}(\mathsf{vrk}, C, X, \mathsf{data}, \pi)$ $\mathsf{acc}_X = \mathsf{SA.Accum}(\mathsf{prk}, X)$

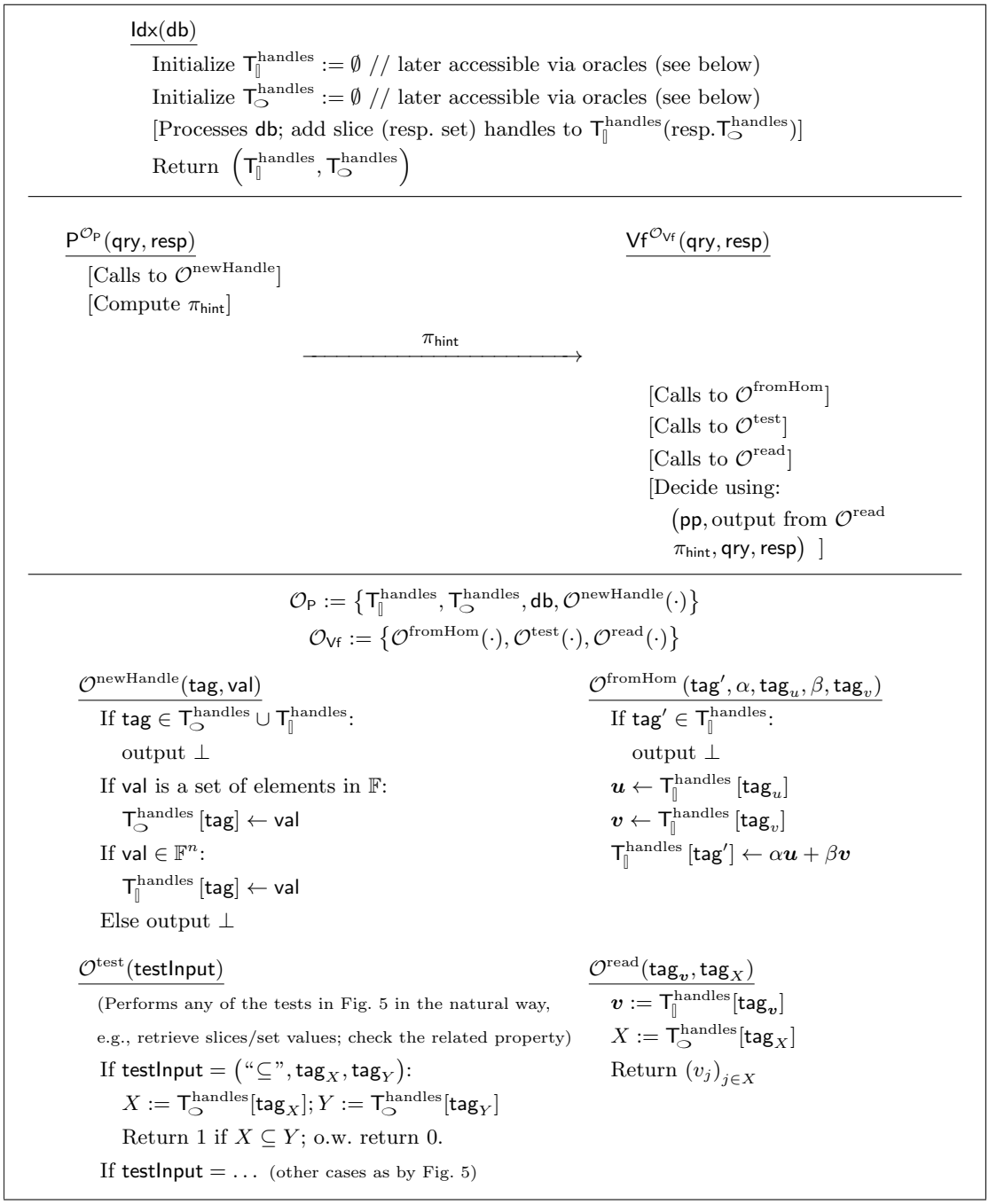

Fig. 6: Formal details for the idealized protocol interaction. We assume that any of the oracles returns $\perp$ whenever trying to access a non-existing tag in the tables. Notice that, by placing them among its oracles, the prover has RAM access to db and the preindexed tables $\mathsf{T}_{[]}^{\text{handles}}$ and $\mathsf{T}_{\bigcirc}^{\text{handles}}$. The implementation of $\mathcal{O}^{\text{test}}$ is straightforward and above we just provide an example that is immediate to extend.

9.5 Instantiating our compiler

Here we describe our formal results. The results follows immediately and their proofs follow the common proofs in compilers of this type (e.g., [17,28]);

Remark 5 (On security requirements of the underlying building blocks). While we require extractability as a form of security for our underlying primitives, we do not believe this requirement to be necessary because of the reliance on at least *some* honestly computed commitments. This may allow us to rely on some form of less strong evaluation binding. We choose to describe our framework in terms of extractability because it makes extremely easy (almost immediate) to prove security of our compilers. Moreover, the extractability properties of KZG—essentially, our core instantiation—have been widely discussed in literature. We leave as future work to a more fine-grained security treatment of our framework.

An instantiation from (augmented) vector commitments & accumulators

Theorem 6. *Let Π_{LVC} and Π_{Acc} be respectively:*

- *an homomorphic linear-map vector commitment scheme (Section 6.3) with subvector opening (Section 6.4);*
- *an accumulator with set relation opening (**??**).*

Let Π_{LVC} and Π_{Acc} also jointly support zero-testing in accumulated sets (defined on page 29). Then our compiler (page 30 and Table 4) on input a succinct idealized VDB (Section 7) produces a succinct VDB (Definition 7, Definition 8 and Definition 9) supporting the same family of queries.

Proof. Let V be a succinct idealized VDB as described in Section 7, the protocol V' obtained by the compiler described in Section 9.4 on input V is a succinct VDB:

Completeness: Intuitively, completeness follows from the completeness of the underlying cryptographic tools. In particular each set handle $\llcorner X$ is compiled in a set accumulator containing X and each slice handle $[\![\boldsymbol{v}$ is compiled in a vector commitment. Every test on set operations is complete for the completeness of the opening operations of set accumulators. The inner product completeness holds for the LVC correctness. The zero test correctness follows from the correctness of the zero-testing in accumulated sets. The new slices from homomorphisms are complete due to homomorphic properties of the LVC commitment. The completeness of the derived operations in Table 3 follows from the correctness of these basic compiled operations.

Soundness: Soundness follows from the extractability of the cryptographic schemes. Since the analysis is extremely standard (e.g., same as in similar compiler approaches as in [17,28]) we simply provide an intuition. Each proof in the idealize VDB is of the form $[\![\boldsymbol{v}_1^{\mathsf{P}}, \llcorner X_1^{\mathsf{P}}, \ldots, \pi_{\mathsf{hint}}$. We assume this proof is correctly computed while the compiled proof does not satisfy the computational soundness of VDB, i.e., the verification of the proof returns 1 but the response is not correct given the committed database. Let $\mathcal{A}$ be the adversary that breaks the soundness (Definition 8) with probability greater than negligible. By invoking the extractability of the underlying building block we are able to extract the committed values from $[\![\boldsymbol{v}_1^{\mathsf{P}}, \llcorner X_1^{\mathsf{P}}, \ldots$ sent at proving time (the preprocessing handles are generated honestly and we can directly point to the respective commitment openings in the reduction). That is, we can use the LVC extractors for each slice handle and the set accumulator extractor for each set handle. By the security of the underlying idealized protocol, it holds that the extracted values justify the response with overwhelming probability contradicting the hypothesis that $\mathcal{A}$ can break the soundness.

Succinctness: This follows immediately by inspection: a succinct transcript from an idealized VDB is compiled into a tuple of vector commitments, set accumulator and respective proofs of the same size up to a $O_\lambda(1)$ factor.

A general instantiation from homomorphic polynomial commitments When we discussed how to build zero-testing on page 29, we hinted at a connection between polynomials and vector/set commitments. This connection goes beyond zero-testing and can be show to generalize quite easily to the other properties we require in Section 7. To prove the theorem below all we need is to show that we can obtain the ingredients of Theorem 6 from any polynomial commitment scheme because we can generalize the constructions in the last subsection. This can be observed by inspecting a generalized version of common constructions (Section 6.6) and observing the same type of patterns we noticed on page 29. (This holds in particular for the specific pairing-based building blocks described in Section 6.6, which can be easily satisfy the requirements of Theorem 6. We will apply this observation in the next section and in Corollary 1).

Theorem 7. *Let Π_{pc} be an homomorphic extractable polynomial commitment scheme. Then our compiler on input a succinct idealized VDB (Section 8) produces a succinct VDB (Section 9) supporting the same family of queries in the random oracle model (ROM).*

Proof. We have to show that all the operations guaranteed by Π_{LVC} and Π_{Acc} are implied by any Π_{pc}.

Therefore, we can show how to perform each operation of Π_{LVC} and Π_{Acc} using polynomial evaluations. For Π_{LVC} :

- For the operations $\mathsf{OpenLin}(\mathsf{prk}, \boldsymbol{u}, \boldsymbol{v}, y)$ and $\mathsf{VerifyLin}(\mathsf{vrk}, \mathsf{cm}_u, \mathsf{cm}_v, y, \pi)$, the prover and the verifier interact as follows: the verifier receives from the prover the commitments $\mathsf{cm}_{\boldsymbol{u}}$, $\mathsf{cm}_{\boldsymbol{v}}$, $\mathsf{cm}_{R(X)}$, $\mathsf{cm}_{H(X)}$, $\mathsf{cm}_{\hat{R}(X)}$ such that $\hat{R}(X) = X^2 R(X)$, $\deg(R) < \bar{n} - 1$ and

$$\left(\sum_{i=1}^{\bar{n}} u_i \lambda_i(X)\right)\left(\sum_{i=1}^{\bar{n}} v_i \lambda_i(X)\right) - \bar{n}^{-1} y = XR(X) + t(X)H(X). \tag{1}$$

  The verifier sends a random value k to the prover. The prover sends $\pi_{\phi_j} \leftarrow \mathsf{OpenPos}(\mathsf{prk}, \phi_j(k), k, d)$, where $\phi_j \in \{\boldsymbol{u}, \boldsymbol{v}, R(X), H(X), \hat{R}(X)\}$. The verifier checks that $\mathsf{VerifyPos}(\mathsf{vrk}, \mathsf{cm}_j, k, \phi(k), \pi_{\phi_j}) = 1$ for all $\mathsf{cm}_j \in \{\mathsf{cm}_{\boldsymbol{u}}, \mathsf{cm}_{\boldsymbol{v}}, \mathsf{cm}_{R(X)}, \mathsf{cm}_{H(X)}, \mathsf{cm}_{\hat{R}(X)}\}$ and evaluates Eq. (1) at $X = k$. If the opening proofs are valid, the equations $\hat{R}(k) - k^2 R(k) = 0$ and Eq. (1) hold, the Schwartz-Zippel lemma guarantees that the proof holds.
- $\mathsf{OpenSub}(\mathsf{prk}, \mathsf{aux}_{\boldsymbol{u}}, I)$ and $\mathsf{VerifySub}(\mathsf{vrk}, \mathsf{cm}_u, \boldsymbol{v}, I, \pi)$ can be computed using the same strategy of $\mathsf{OpenLin}(\mathsf{prk}, \boldsymbol{u}, \boldsymbol{v}, y)$ and $\mathsf{VerifyLin}(\mathsf{vrk}, \mathsf{cm}_u, \mathsf{cm}_v, y, \pi)$.

For the set accumulator, the fact that we can construct them from polynomial commitments has already been shown in Section 6.6. The zero-testing in accumulated sets follows the blueprint described in Section 9.2. To make the protocol non interactive the prover can use the Fiat-Shamir heuristic producing the challenges using a RO. □

Remark 6 (Implication: post-quantum VDBs (for free!)). An implication of our work and specifically of Theorem 7 is that we can plug in *any extractable homomorphic polynomial commitment* and obtain a construction of verifiable databases. For example, using the polynomial commitment based on standard lattice assumptions from [48] to immediately obtain a plausibly post-quantum secure VDB construction[25].

10 Our Final Construction: qedb

Here we first describe our construction in terms of the operations—core and derived (see Fig. 1)—for our idealized protocol. Our final construction follows from compiling and optimizing it our idealized VDB construction.

We now list few relevant queries that are expressible in our intermediate (idealized) representation formalism.

[25] The resulting construction has superconstant overhead for the verifier and proof size unlike the one in this work.

Simple selection queries. We provide some examples of simple queries over tabular data. We start considering a single table T composed of several columns $\mathsf{col}_1, \mathsf{col}_2, \ldots, \mathsf{col}_u$.

The simplest query returns some column from T, without any selection condition (in this case the prover does not have to check any selection condition):

$$\mathsf{Q1}: \quad \mathsf{SELECT}\ \mathsf{col}_1, \mathsf{col}_2\ \mathsf{FROM}\ \mathsf{T} \tag{2}$$

Pre-processing: The Data Owner defines slice handles $\llbracket \mathsf{col}_1 \rrbracket$, $\llbracket \mathsf{col}_2 \rrbracket$
Proof computation: Prover does nothing
Proof verification: Verifier retrieves the data in columns col_1, col_2 with $\mathbf{read}(\bigcirc\!\!*, \llbracket \mathsf{col}_1 \rrbracket)$ and $\mathbf{read}(\bigcirc\!\!*, \llbracket \mathsf{col}_2 \rrbracket)$.

Completeness holds since the verifier collects the handles $\llbracket \mathsf{col}_1 \rrbracket$ and $\llbracket \mathsf{col}_2 \rrbracket$ to col_1 and col_2 from $\mathsf{T}^{\mathsf{handles}}_{\llbracket\rrbracket}$ and $\mathbf{read}(\bigcirc\!\!*, \llbracket \mathsf{col}_1 \rrbracket)$ and $\mathbf{read}(\bigcirc\!\!*, \llbracket \mathsf{col}_2 \rrbracket)$ return exactly what prescribed by the calls to $\mathcal{O}^{\mathrm{read}}(\llbracket \mathsf{col}_1 \rrbracket, \bigcirc\!\!*)$ and $\mathcal{O}^{\mathrm{read}}(\llbracket \mathsf{col}_2 \rrbracket, \bigcirc\!\!*)$. Soundness holds since the verifier is not taking anything from the prover and only computes $\mathbf{read}(\bigcirc\!\!*, \llbracket \mathsf{col}_1 \rrbracket)$ and $\mathbf{read}(\bigcirc\!\!*, \llbracket \mathsf{col}_2 \rrbracket)$. These commands return exactly the values contained in col_1 and col_2, therefore $\mathsf{SatisfiesQry}\,(\mathsf{db}, \mathsf{qry}, \mathsf{resp}) = \mathsf{true}$.

The following query contains an equality-based selection condition:

$$\mathsf{Q2}: \quad \mathsf{SELECT}\ \mathsf{col}_1\ \mathsf{FROM}\ \mathsf{T}\ \mathsf{WHERE}\ \mathsf{col}_2 = u \tag{3}$$

Pre-processing: we assume that the Data Owner knows the set V to which the value u belongs to. The Data Owner computes the slice handle $\llbracket col_1 \rrbracket$; then it computes an *inverted index* associated to the set V: (i) for all the values of V appearing in col_2, take note of all their positions in such column; (ii) compute set handles of such lists of positions. More formally, for each $v \in V$ define the inverted index $X_v = \{j : \mathsf{col}_2[j] = v\}$. Then, define the set handle $\bigcirc X_v$.
Proof computation: As in the previous query, the Prover does nothing
Proof verification: The Verifier picks $\bigcirc X_u$ and outputs $\mathbf{read}(\bigcirc X_u, \llbracket col_1 \rrbracket)$.

Completeness holds since the verifier collects the handles $\llbracket \mathsf{col}_1 \rrbracket$ to col_1 from $\mathsf{T}^{\mathsf{handles}}_{\llbracket\rrbracket}$ and $\bigcirc u$ from $\mathsf{T}^{\mathsf{handles}}_{\bigcirc}$, therefore the verifier computes $\mathbf{read}(\bigcirc u, \llbracket \mathsf{col}_1 \rrbracket)$ that returns exactly the positions in col_1 where $\mathsf{col}_2 = u$ that is what prescribed by the calls to $\mathcal{O}^{\mathrm{read}}(\llbracket \mathsf{col}_1 \rrbracket, \bigcirc u)$. Soundness holds since the verifier is not taking anything from the prover and only computes $\mathbf{read}(\bigcirc u, \llbracket \mathsf{col}_1 \rrbracket)$. This command returns exactly the values contained in col_1 required by the query, therefore $\mathsf{SatisfiesQry}\,(\mathsf{db}, \mathsf{qry}, \mathsf{resp}) = \mathsf{true}$.

Range queries and conjunctive queries. Consider the following range query:

$$\mathsf{Q3}: \quad \mathsf{SELECT}\ \mathsf{col}_1\ \mathsf{FROM}\ \mathsf{T}\ \mathsf{WHERE}\ \mathsf{col}_2 \in [\alpha, \beta] \tag{4}$$

Pre-processing: as before.
To show that X_{in} is the set of all indices j such that $\alpha \leq \boldsymbol{v}[j] \leq \beta$, prover and verifier execute the following steps (we have renamed col_2 as $\boldsymbol{v}$):

- prover sends $\bigcirc X_{out}$
- verifier checks it is as claimed

// *check that all the values indexed by X_{in} are in the range*:

- prover sends $\llbracket \boldsymbol{u}_{\mathbb{1}} \rrbracket$, where $\boldsymbol{u}_{\mathbb{1}}$ is equal to 1 for each index in X_{in} and 0 elsewhere
- verifier checks that $\llbracket \boldsymbol{u}_{\mathbb{1}} \rrbracket \stackrel{?}{=} \llbracket \mathbf{1} \rrbracket \left[\bigcirc X_{out} \to 0 \right]$
- let $\boldsymbol{u}_{\geq \alpha} := \boldsymbol{v} - \alpha \boldsymbol{u}_{\mathbb{1}}$

– let $\boldsymbol{u}_{\leq\beta} := \beta\boldsymbol{u}_{\mathbb{1}} - \boldsymbol{v}$
– prover and verifier run the protocol to check that $\mathbb{P}\boldsymbol{u}_{\geq\alpha} \geq 0$
– prover and verifier run the protocol to check that $\mathbb{P}\boldsymbol{u}_{\leq\beta} \geq 0$

// *check that no value out of those indexed by* X_{in} *is in the range*:

– prover sends two additional set handles $\bigcirc X_{<\alpha}$, $\bigcirc X_{>\beta}$
– verifier checks that $\bigcirc X_{in}$, $\bigcirc X_{<\alpha}$, $\bigcirc X_{>\beta}$ form a partition of $\bigcirc X_{out}$
– let $\boldsymbol{u}_{<} := \alpha\boldsymbol{u}_{\mathbb{1}} - \boldsymbol{v}$
– let $\boldsymbol{u}_{>} := \boldsymbol{v} - \beta\boldsymbol{u}_{\mathbb{1}}$
– prover and verifier run the protocol to check that $\mathbb{P}\boldsymbol{u}_{<}[\bigcirc X_{<\alpha}] \geq 0$
– prover and verifier run the protocol to check that $\mathbb{P}\boldsymbol{u}_{>}[\bigcirc X_{>\beta}] \geq 0$
– verifier performs **read**($\mathbb{P}X_{in}$, $\bigcirc\mathsf{col}_1$)

Completeness holds since for each response that is correct, thanks to the correctness of the underlying scheme, the prover can correctly show that elements indexed in X_{in} are in the range, while the others values are outside the range. Let us assume that soundness does not hold, therefore there is a prover that generates accepting proof for a response resp such that $\mathsf{SatisfiesQry}\,(\mathsf{db}, \mathsf{qry}, \mathsf{resp}) = \mathsf{false}$. Let π_α, π_β, $\pi_<$, $\pi_>$ the proofs of – respectively – the chosen values in $mathcol_2$ are contained in the range $[\alpha, beta]$, and no other values have been chosen. Since π_α guarantees that all the element in col_2 in positions contained in X_{in} are greater or equal to α, π_β guarantees that that values are also lower than or equal to β. The first verifier check guarantees that there is no intersection among the two sets X_{in} and X_{out}. The proof $\pi_<$ guarantees that the elements in $X_<$ are all small than α while $\pi_>$ shows that elements in $X_>$ are all greater than β. Therefore **read**($\mathbb{P}X_{in}$, $\bigcirc\mathsf{col}_1$) is equal to resp and $\mathsf{SatisfiesQry}\,(\mathsf{db}, \mathsf{qry}, \mathsf{resp}) = \mathsf{true}$.

As for general conjunctive query, consider the query:

$$\mathsf{Q4}: \quad \mathsf{SELECT}\ \mathsf{col}_1\ \mathsf{FROM}\ \mathsf{T}\ \mathsf{WHERE}\ Test_1(\mathsf{col}_2)\ \mathsf{AND}\ Test_2(\mathsf{col}_2) \tag{5}$$

Pre-processing: as before.
Proof computation: the Prover computes the set handles $\bigcirc X_{Test_1}$, $\bigcirc X_{Test_2}$ and sends them to the Verifier.
Proof Verification: the Verifier performs the following steps:

– after renaming the set handles received from the Prover to $\bigcirc X_{in,1}$, $\bigcirc X_{in,2}$ respectively, define the set handle $\bigcirc X_{in,\wedge} = \bigcirc X_{in,1} \cap \bigcirc X_{in,2}$
– output **read**($\bigcirc X_{in,\wedge}$, $\mathbb{P}\mathsf{col}_1$)

Completeness follows from the the fact that $Test_1(\mathsf{col}_2)\ \mathsf{AND}\ Test_2(\mathsf{col}_2)$ will return the correct set handles $\bigcirc X_{Test_1}$ and $\bigcirc X_{Test_2}$. Indeed the verifier can correctly compute $\bigcirc X_{Test_1} \cup \bigcirc X_{Test_2}$. Since the only data sent by the prover are $\bigcirc X_{Test_1}$ and $\bigcirc X_{Test_2}$, we assume that there exists an adversarial prover that is able to return values $\bigcirc X_{Test_1}$ and $\bigcirc X_{Test_2}$ such that the intersection among these indexes contains wrong elements. Let $\bigcirc X_{Test_1}$ and $\bigcirc X_{Test_2}$ be correct values for the soundness of $Test_1(\mathsf{col}_2)$ and $Test_2(\mathsf{col}_2)$. Since the read is computing the elements obtained by the intersection of $\bigcirc X_{Test_1}$ and $\bigcirc X_{Test_2}$ and the intersection algorithm is sound, it holds that $\mathsf{SatisfiesQry}\,(\mathsf{db}, \mathsf{qry}, \mathsf{resp}) = \mathsf{true}$.

Join queries. Consider tables T_1, T_2 with respective columns named pk, col_1 and fk, col_2. As their names suggest pk is primary key of table T_1, and fk is a foreign key in T_2 referencing values from pk. Consider the query:

$$\mathsf{Q5}: \quad \mathsf{SELECT} * \mathsf{FROM}\, T_1 \,\mathsf{JOIN}\, T_2 \,\mathsf{ON}\, pk = fk \tag{6}$$

Pre-processing: as before.
Proof computation: the Prover performs the following steps:

- retrieves the set handle $\circlearrowright fk$ referring the inverse lookup $\{j : fk[j] = v\}$, for each $v \in V_{pk}$.
- retrieves the set handle $\circlearrowright pk$ referring the inverse lookup $\{j : pk[j] = v\}$, for each $v \in V_{fk}$.

The Prover sends $\circlearrowright pk$, $\circlearrowright fk$ to the Verifier.
Proof verification: The Verifier performs the following steps:

- compute $\widehat{pk} \leftarrow \mathbf{read}(\circlearrowright pk, \llbracket pk \rrbracket)$
- compute $\widehat{fk} \leftarrow \mathbf{read}(\circlearrowright fk, \llbracket fk \rrbracket)$
- check that $\widehat{fk} = \widehat{pk}$
- $\widehat{rst_1} \leftarrow \mathbf{read}(\circlearrowright pk, \llbracket T_1.rst \rrbracket)$
- compute $\widehat{rst_2} \leftarrow \mathbf{read}(\circlearrowright fk, \llbracket T_2.rst \rrbracket)$

Subsequently, the Verifier concatenates the data retrieved in $\widehat{pk}$, $\widehat{rst_1}$ and $\widehat{rst_2}$. To prove that the query result contains all the valid tuples, prover and verifier engage in a protocol similar to the second part of the one defined for Query .

To join two tables T and T' on equality on column C with duplicates in both tables, we do the following:

Invariant (initially enforced through indexing):

- For each table T and column C we keep a set of values $V(T, C)$ in that column

Observation: let $V_\cap := V(T, C) \cap V(T', C) = \{\alpha_1, \ldots, \alpha_\ell\}$, the JOIN will be given by the cross product of the rows from each table as follows:

$$\alpha_i^{-1}(T.C) \times \alpha_i^{-1}(T'.C) \text{ for } i = 1, \ldots, \ell$$

where recall that $\alpha^{-1}(\boldsymbol{v}) := \{j : v_j = \alpha\}$ denotes the set of preimages of the value α in the vector $\boldsymbol{v}$.

Our protocol is then as follows:

- the prover sends $V_\cap$ as defined above to the verifier (in the clear—this is not a handle)
- it also sends for each $\alpha \in V_\cap$, the respective preimage set (as a handle) in each table. I.e., it sends $\circlearrowright X_{\alpha,\mathsf{col}} := \alpha^{-1}(\mathsf{col})$ for $\mathsf{col} \in \{T.C, T'.C\}$ and $\alpha \in V_\cap$.
- Then, for each $\llbracket \mathsf{col} \rrbracket \in \{\llbracket T.C \rrbracket, \llbracket T'.C \rrbracket\}$ and $\alpha \in V_\cap$, the prover and verifier run a sub-protocol $X_{\alpha,\mathsf{col}} \stackrel{?}{=} \alpha^{-1}(\llbracket \mathsf{col} \rrbracket)$.
- They also run a sub-protocol to show $\circlearrowright V_\cap \stackrel{?}{=} \circlearrowright V(T, C) \cap \circlearrowright V(T', C)$ (recall that the last two handles are already held by the verifier as handles from the invariant).
- Finally the verifier run $\mathbf{read}(\bigcup_\alpha \circlearrowright X_{\alpha,\mathsf{T.C}}, \llbracket T.\star \rrbracket)$ and $\mathbf{read}(\bigcup_\alpha \circlearrowright X_{\alpha,\mathsf{T'.C}}, \llbracket T'.\star \rrbracket)$ (i.e., it reads any column selected for the JOIN) and performs the cartesian product appropriately.

The protocol above directly extends to multi-way JOINs.

Aggregate queries. We now present the protocols for min, sum, count, and average queries.

– MIN query: consider the query:

$$\mathsf{Q6}: \quad \mathsf{SELECT\ MIN}(\mathsf{col}_{tgt})\ \mathsf{FROM}\ T \tag{7}$$

Pre-processing: we assume that the Verifier has the slice handle $\llbracket tgt \rrbracket$ corresponding to col_{tgt}.
Proof computation: the Prover does as follows:
  – compute the set of positions $argmin(tgt)$ in col_{tgt} of the minimum values: $argmin(tgt) = \{j : \mathsf{col}_{tgt}[j] \leq v \quad \text{for all } v \in \mathsf{col}_{tgt}\}$ (here we denote with $\mathsf{col}_{tgt}[j]$ the j-th element in the slice referred by $\llbracket tgt \rrbracket$.
  – compute $\bigcirc X_{argmin}$ and sends it to the Verifier
Proof verification: the Verifier performs the following steps:
  – get $\bigcirc X_{argmin}$
  – retrieve $v_{\mathsf{min}} \leftarrow \mathbf{read}(\bigcirc X_{argmin}, \llbracket tgt \rrbracket)$
  – define $\llbracket tgt' \rrbracket = \llbracket tgt \rrbracket - \llbracket (v_{\mathsf{min}}, \ldots, v_{min}) \rrbracket$
  – check that every value in the slice handle $\llbracket tgt' \rrbracket$ lies in the interval $[0, 2^l)$

Completeness follows from the fact that $\bigcirc X_{argmin}$ actually contains only the indices with the minimum value in the column (it can contain more than one element if the minimum is repeated multiple times). Therefore $v_{\mathsf{min}} \leftarrow \mathbf{read}(\bigcirc X_{argmin}, \llbracket tgt \rrbracket)$ outputs a vector containing only the minimum value in the column col_{tgt}. The correctness is also enforced by the range proof proving that after removing from col_{tgt} the vector $v_{\mathsf{min}}[0]\boldsymbol{u}_{\mathbb{1}}$, where $\boldsymbol{u}_{\mathbb{1}}$ is the slice comprising of all ones, it is contained in the range $[0, 2^\ell)$ meaning that these values are all greater than $v_{\mathsf{min}}[0]$. The adversarial prover sends $\bigcirc X_{argmin}$, therefore if the verification returns 1 while $\mathsf{SatisfiesQry}\,(\mathsf{db}, \mathsf{qry}, \mathsf{resp}) = \mathsf{false}$, it means that $\bigcirc X_{argmin}$ are not the indexes containing the minimum value. If this is the case, the range proof fails since there is at least a value in $\llbracket tgt' \rrbracket$ that is less then 0, since the range proof is sound it can happen only with negligible probability.
The protocol for max is a trivial adaptation of the above.

– SUM query: consider the query:

$$\mathsf{Q7}: \quad \mathsf{SELECT\ SUM}(\mathsf{col}_{tgt})\ \mathsf{FROM}\ T \tag{8}$$

Pre-processing: as before.
Proof computation:
  – the Prover computes the set handle $\bigcirc *$ corresponding to the rows of T and the value $s_{tgt} = \sum_{v \in \mathsf{col}_{tgt}} v$
  – The Prover sends $\bigcirc *$ and s_{tgt} to the Verifier
Proof Verification:
  – the Verifier gets $\bigcirc *$ and s_{tgt} from the Prover
  – the Verifier checks that $\sum_{\bigcirc *} \llbracket tgt \rrbracket$ is equal to s_{tgt}

Completeness follows from the correctness of the sum check within target subset operation, indeed the prover is sending $\bigcirc *$ together with $\boldsymbol{u}_{\mathbb{1}}$, I.e., the vector that contains all ones, the verifier checks that $\boldsymbol{u}_{\mathbb{1}}$ is actually one in all positions and then performs the inner product between col_{tgt} and $\boldsymbol{u}_{\mathbb{1}}$ checking if it is equal to the response. To prove soundness let us assume that there exists an adversarial prover that will cause the verifier to return 1 but such that $\mathsf{SatisfiesQry}\,(\mathsf{db}, \mathsf{qry}, \mathsf{resp}) = \mathsf{false}$. Therefore resp does not contain the sum of the elements in col_{tgt}. The probability that it happens is negligible indeed the verifier can check that $\boldsymbol{u}_{\mathbb{1}}$ is a vector of all ones, that $\bigcirc *$ is indeed a set handle to all indices and that the the inner product of the two handles is actually the expected sum.

– COUNT query: consider the query:

$$\mathsf{Q8}: \quad \mathsf{SELECT\ COUNT}(col_{tgt})\ \mathsf{FROM}\ T \tag{9}$$

Pre-processing: as before.
Proof computation:
– the Prover computes the set handle $\circlearrowright *$ referring to all rows of T
– the Prover sends $\circlearrowright *$ and the value n to the Verifier
Proof Verification: The Verifier performs the following steps:
– get $\circlearrowright *$ and the value n from the Prover,
– define slice handle $[\![ones]\!] \leftarrow (\underbrace{1, \ldots, 1}_{n})$
– check that $\sum_{\circlearrowright *} [\![ones]\!]$ is equal to n

Completeness and soundness discussion are equal to the one done for Q8.

Nested queries. Consider the query:

$$\begin{array}{l}\mathsf{Q9}: \quad \mathsf{SELECT}\ col_1\ \mathsf{FROM}\ T_1\ \mathsf{WHERE}\ col_2\ \mathsf{IN} \\ \quad (\mathsf{SELECT}\ col_3\ \mathsf{FROM}\ T_2\ \mathsf{WHERE}\ col_4 = u)\end{array} \tag{10}$$

Pre-processing: as before.
Proof computation: Prover sends to Verifier the following set handles: $\circlearrowright col_{1_u}$ (the entry of set handle $\circlearrowright col_4$ corresponding to value v) and $\circlearrowright col_{2_{v_1}}$, $\circlearrowright col_{2_{v_2}}, \ldots, \circlearrowright col_{2_{v_n}}$, that are the set handles entries in $\circlearrowright col_2$ of the values $\{v_1, v_2, \ldots, v_n\}$ contained in the answer to the sub-query $\mathsf{SELECT}\ col_3\ \mathsf{FROM}\ T_2\ \mathsf{WHERE}\ col_4 = u$.
Proof verification: Verifier creates the new set handle $\circlearrowright col_{2_{OR}}$ equal to $\circlearrowright col_{2_{v_1}} \cup \circlearrowright col_{2_{v_2}} \cup \cdots \cup \circlearrowright col_{2_{v_n}}$; finally, Verifier retrieves the answer of the nested query via **read**($\circlearrowright col_{2_{OR}}$, $[\![col_4]\!]$).

Again, to prove that the query result contains all the valid tuples, prover and verifier engage in a protocol similar to the second part of the one defined for Query . In general, we can handle nested queries through techniques analogous to those in [85] but without having to rely on a preprocessing containing auxiliary info on every possible pair of columns in the same table (which leads to its quadratic blowup).

Group by queries. Consider the query, where the condition on $\mathsf{agg}(\mathsf{col}_{tgt})$ can be any aggregation expression.

$$\mathsf{Q10}: \quad \mathsf{SELECT\ col}_1, \mathsf{\ agg}(\mathsf{col}_{tgt})\ \mathsf{FROM}\ T_1\ \mathsf{GROUP\ BY\ col}_1 \tag{11}$$

Pre-processing: We assume that the Verifier has the slice handle $[\![tgt]\!]$ corresponding to col_{tgt} together with the set handles for column col_1.
Proof computation: the Prover performs the following steps:

– select the elements set $\{\alpha_1, \ldots, \alpha_n\}$ in col_1 corresponding to the indices in X_{col_2}
– given the values $\{\alpha_1, \ldots, \alpha_n\}$ in col_1, compute the sets of indices X_1, …, X_n where $X_i = \{j : \mathsf{col}_1[j] = \alpha_i\}$
– compute the aggregations a_1, …, a_n on col_{tgt} on indices X_1, …, X_n
– send $\circlearrowright X_1, \ldots, \circlearrowright X_n$ to the Verifier
– the Prover and the Verifier run a sub-protocol to prove that $\circlearrowright X_1 \cup \ldots \cup \circlearrowright X_n = \circlearrowright *$
– the Verifier checks that each $\circlearrowright X_i$ is contained in the set accumulator for column col_1

– the Prover and the Verifier run a sub-protocol to prove that each aggregation a_i, for $i \in [n]$ is correct in $\mathbb{tgt}$ under the indices $\supset X_i$.

Let us assume that completeness does not hold, in this case either the prover did not send a pair (α_j, a_j) or there exists at least a value $\alpha_j \in \{\alpha_1, \ldots, \alpha_n\}$ such that the aggregator is not aggregating all values. In the first case it implies that X_1, ..., X_n is not a partition of $*$ (i.e., proving that each X_i is in a set handle for col_1 and that the union of all X_i is equal to $\supset *$) that is a contradiction. The second case implies that the aggregation proof is not complete statement that is a contradiction. Let us assume that soundness does not hold. In this case, either in (α_j, a_j), a_j does not correspond to the indices in X_j or there exists an α_j such that is not in the positions X_{col_2}. The first case contradicts the soundness of the aggregation proof. The second case instead violates one of the checks performed by the Verifier in the containment of $\supset X_i$ in the set accumulator for column col_1.

Other queries. Our construction takes into account queries that contain predicates and expressions involving two or more columns. Consider the query:

$$\mathsf{Q11}: \quad \mathsf{SELECT}\ c_1\ \mathsf{FROM}\ T\ \mathsf{WHERE}\ c_2 \geq 2c_3 + c_4 \tag{12}$$

This can be handled immediately thanks to the homomorphic properties of our framework (for simplicity here we omit the usual color coding for handles). For example, in the above the prover can first homomorphically compute a "virtual" column (a slice) $c' := c_2 - 2c_3 - c_4$ (and so can the verifier) and then provide a set handle of rows in which such a column would have a value greater than 0 (as we did for the simple case of range queries). The verifier can then read that set from c_1. Completeness and soundness discussion are equal to the one done for Q3.

Similarly, our system can handles queries that involve, e.g., $\mathsf{SELECT}\ 3c_1 + c_5\ \mathsf{WHERE}\ldots$. Here the prover and verifier can produce a new column c' (in this case $c' := 3c_1 + c_5$). The read will occur on this specific column. For more details on homomorphisms in our framework we also refer the reader to Remark 2.

Putting it all together: VDBs from pairing techniques (and essentially KZG) We instantiate our compiler using the building blocks described in Section 6.6 and on page 29 and combining it with the construction in this section:

Corollary 1. *There exists a succinct verifiable DB with the efficiency metrics in Table 1 (last row) supporting the set of queries in Fig. 1 whose setup is the standard KZG setup.*

Acknowledgments and disclosure

Matteo Campanelli would like to thank Nicola Greco who, around 2022, first suggested verifiable databases as an interesting topic to work on. Matteo has no financial interest in Provably Technologies and has pursued this research work on a voluntary basis, motivated by the potential relevance to the cryptographic proofs community.

References

1. D. F. Aranha, E. M. Bennedsen, M. Campanelli, C. Ganesh, C. Orlandi, and A. Takahashi. Eclipse: Enhanced compiling method for pedersen-committed zksnark engines. In *IACR International Conference on Public-Key Cryptography*, pages 584–614. Springer, 2022.
2. arkworks contributors. `arkworks` zksnark ecosystem, 2022.
3. A. Arun, S. T. V. Setty, and J. Thaler. Jolt: Snarks for virtual machines via lookups. In *Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024*, 2024.
4. Axiom. Axiom openvm, 2025. `https://www.axiom.xyz/`.

5. E. Ben-Sasson, A. Chiesa, and N. Spooner. Interactive oracle proofs. In *Theory of Cryptography: 14th International Conference, TCC 2016-B, Beijing, China, October 31-November 3, 2016, Proceedings, Part II 14*, pages 31–60. Springer, 2016.
6. N. Bitansky, R. Canetti, A. Chiesa, S. Goldwasser, H. Lin, A. Rubinstein, and E. Tromer. The hunting of the snark. *Journal of Cryptology*, 30(4):989–1066, 2017.
7. B. Bünz, B. Fisch, and A. Szepieniec. Transparent snarks from dark compilers. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*, pages 677–706. Springer, 2020.
8. B. Bünz, M. Maller, P. Mishra, N. Tyagi, and P. Vesely. Proofs for inner pairing products and applications. Cryptology ePrint Archive, Paper 2019/1177, 2019.
9. M. Campanelli, F. Engelmann, and C. Orlandi. Zero-knowledge for homomorphic key-value commitments with applications to privacy-preserving ledgers. In *International Conference on Security and Cryptography for Networks*, pages 761–784. Springer, 2022.
10. M. Campanelli, A. Faonio, D. Fiore, T. Li, and H. Lipmaa. Lookup arguments: Improvements, extensions and applications to zero-knowledge decision trees. In *Public-Key Cryptography - PKC 2024 - 27th IACR International Conference on Practice and Theory of Public-Key Cryptography*, 2024.
11. M. Campanelli, A. Faonio, D. Fiore, A. Querol, and H. Rodríguez. Lunar: A toolbox for more efficient universal and updatable zksnarks and commit-and-prove extensions. In *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part III*, 2021.
12. M. Campanelli, A. Faonio, and L. Russo. Snarks for virtual machines are non-malleable. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 153–183. Springer, 2025.
13. M. Campanelli, D. Fiore, and R. Gennaro. Natively compatible super-efficient lookup arguments and how to apply them. *Journal of Cryptology*, 38(1):14, 2025.
14. M. Campanelli, D. Fiore, S. Han, J. Kim, D. Kolonelos, and H. Oh. Succinct zero-knowledge batch proofs for set accumulators. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 455–469, 2022.
15. M. Campanelli, D. Fiore, and H. Khoshakhlagh. Witness encryption for succinct functional commitments and applications. In *Public-Key Cryptography - PKC 2024 - 27th IACR International Conference on Practice and Theory of Public-Key Cryptography, Sydney, Australia, April 15-17, 2024, Proceedings, Part II*, volume 14602 of *Lecture Notes in Computer Science*, pages 132–167. Springer, 2024.
16. M. Campanelli, D. Fiore, and M. Pancholi. When can we incrementally prove computations of arbitrary depth? *Cryptology ePrint Archive*, 2025.
17. M. Campanelli and M. Hall-Andersen. Fully succinct arguments over the integers from first principles. Cryptology ePrint Archive, Paper 2024/1548, 2024.
18. M. Campanelli and M. Hall-Andersen. Fully succinct arguments over the integers from first principles. Cryptology ePrint Archive, Paper 2024/1548, 2024.
19. M. Campanelli, M. Hall-Andersen, and S. H. Kamp. Curve trees: Practical and transparent {Zero-Knowledge} accumulators. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4391–4408, 2023.
20. M. Campanelli, M. Hall-Andersen, and S. H. Kamp. Curve forests: Transparent zero-knowledge set membership with batching and strong security. *Cryptology ePrint Archive*, 2024.
21. M. Campanelli, A. Nitulescu, C. Ràfols, A. Zacharakis, and A. Zapico. Linear-map vector commitments and their practical applications. In *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022*, 2022.
22. D. Catalano and D. Fiore. Vector commitments and their applications. In *Public-Key Cryptography - PKC 2013 - 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26 - March 1, 2013*, 2013.
23. D. Catalano and D. Fiore. Vector commitments and their applications. In T. Johansson and P. Q. Nguyen, editors, *Public-Key Cryptography – PKC 2013, 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26 – March 1, 2013. Proceedings*, volume 7778 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2013.
24. D. Catalano, D. Fiore, and I. Tucker. Additive-homomorphic functional commitments and applications to homomorphic signatures. In S. Agrawal and D. Lin, editors, *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology*

and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part IV, volume 13794 of *Lecture Notes in Computer Science*, pages 159–188. Springer, 2022.
25. M. Chase, A. Healy, A. Lysyanskaya, T. Malkin, and L. Reyzin. Mercurial commitments with applications to zero-knowledge sets. In R. Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*. Springer, 2005.
26. B. Chen, B. Bünz, D. Boneh, and Z. Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part II*, 2023.
27. A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, and N. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. Cryptology ePrint Archive, Paper 2019/1047, 2019.
28. A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, and N. P. Ward. Marlin: Preprocessing zksnarks with universal and updatable SRS. In *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*, 2020.
29. I. Chowdhury and M. Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.
30. E. Commission. Data act explained, 2025.
31. G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In S. Goldwasser, editor, *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 90–112. ACM, 2012.
32. T. P. P. Council. Tpc benchmark h standard specification. Technical report, Transaction Processing Performance Council (TPC), 2022.
33. Q. Dao, J. Miller, O. Wright, and P. Grubbs. Weak fiat-shamir attacks on modern proof systems. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 199–216. IEEE, 2023.
34. L. de Castro and C. Peikert. Functional commitments for all functions, with transparent setup and from SIS. In C. Hazay and M. Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part III*, volume 14006 of *Lecture Notes in Computer Science*, pages 287–320. Springer, 2023.
35. Deloitte. It's time to reprioritize bcbs239 compliance, 2023.
36. S. Dittmer, Y. Ishai, and R. Ostrovsky. Line-point zero knowledge and its applications. *IACR Cryptol. ePrint Arch.*, page 1446, 2020.
37. Ethereum. Powers of tau specification, 2022.
38. B. for International Settlements. Principles for effective riskdata aggregation and risk reporting. BIS report BCBS239, 2013.
39. W. E. Forum. Data integrity (blockchain toolkit), 2025.
40. G. Fuchsbauer, E. Kiltz, and J. Loss. The algebraic group model and its applications. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, 2018.
41. A. Gabizon and Z. J. Williamson. plookup: A simplified polynomial protocol for lookup tables. *Cryptology ePrint Archive*, 2020.
42. A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, page 953, 2019.
43. S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4):27:1–27:64, 2015.
44. B. Gu, J. Fang, and F. Nawab. PoneglyphDB: Efficient non-interactive zero-knowledge proofs for arbitrary SQL-query verification. *Proc. ACM Manag. Data*, 3(1):63:1–63:27, 2025.
45. F. Guo, Y. Mu, and Z. Chen. Identity-based encryption: How to decrypt multiple ciphertexts using a single decryption key. In *International Conference on Pairing-Based Cryptography*, pages 392–406. Springer, 2007.
46. Hacken. https://hacken.io/discover/blockchain-oracles/, 2025.
47. HSBC. Form 20-f, 2025.
48. I. Hwang, J. Seo, and Y. Song. Concretely efficient lattice-based polynomial commitment from standard assumptions. In *Annual International Cryptology Conference*, pages 414–448. Springer, 2024.
49. IntegriDB. Integridb code repository. `https://github.com/integridb/Code`.
50. Y. Ishai. Zero knowledge proofs from information-theoretic proof systems i, 2025. `https://zkproof.org/2020/08/12/information-theoretic-proof-systems/`.
51. JPMorganChase. Form 10-k, 2024.

52. A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore*, 2010.
53. D. Khovratovich, R. D. Rothblum, and L. Soukhanov. How to prove false statements: Practical attacks on fiat-shamir. *IACR Cryptol. ePrint Arch.*, 2025.
54. D. Khovratovich, R. D. Rothblum, and L. Soukhanov. How to prove false statements: Practical attacks on fiat-shamir. Cryptology ePrint Archive, Paper 2025/118, 2025.
55. L. Labs. Zk co-processor, 2025. `https://www.lagrange.dev/zk-coprocessor`.
56. R. W. F. Lai and G. Malavolta. Subvector commitments with application to succinct arguments. In A. Boldyreva and D. Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part I*, volume 11692 of *Lecture Notes in Computer Science*, pages 530–560. Springer, 2019.
57. J. Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. *IACR Cryptol. ePrint Arch.*, page 1274, 2020.
58. F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 121–132. ACM, 2006.
59. X. Li, C. Weng, Y. Xu, X. Wang, and J. Rogers. ZKSQL: verifiable and efficient query evaluation with zero-knowledge proofs. *Proc. VLDB Endow.*, 16(8), 2023.
60. B. Libert, K. Nguyen, B. H. M. Tan, and H. Wang. Zero-knowledge elementary databases with more expressive queries. In D. Lin and K. Sako, editors, *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part I*, Lecture Notes in Computer Science. Springer, 2019.
61. B. Libert, S. C. Ramanna, and M. Yung. Functional commitment schemes: From polynomial commitments to pairing-based accumulators from simple assumptions. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy, Proceedings*, 2016.
62. B. Libert and M. Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In D. Micciancio, editor, *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings*, volume 5978 of *Lecture Notes in Computer Science*, pages 499–517. Springer, 2010.
63. S. Micali, M. O. Rabin, and J. Kilian. Zero-knowledge sets. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*. IEEE Computer Society, 2003.
64. A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 411–424. ACM, 2014.
65. E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *ACM Trans. Storage*, 2(2):107–138, 2006.
66. L. Nguyen. Accumulators from bilinear pairings and applications. In *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, pages 275–292. Springer, 2005.
67. U. T. O. of Financial Research. Developing best practices for regulatory data collections, 2016.
68. B. of International Settlements. The oracle problem and the future of defi, 2023.
69. R. Ostrovsky, C. Rackoff, and A. D. Smith. Efficient consistency proofs for generalized queries on a committed database. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, volume 3142 of *Lecture Notes in Computer Science*, pages 1041–1053. Springer, 2004.
70. C. Papamanthou, E. Shi, and R. Tamassia. Signatures of correct computation. In *Theory of Cryptography Conference*, pages 222–242. Springer, 2013.
71. C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011*, 2011.
72. G. Ramezan, E. R. Casas, B. Beath, and J. Godfrey. zk-database: Privacy-enabled databases using zero-knowledge proof. In *Proceedings of the 2024 7th International Conference on Blockchain Technology and Applications, ICBTA 2024, Xi'an, China, December 6-8, 2024*. ACM, 2024.
73. M. Z. Sam Ragsdale and J. Thaler. Understanding lasso and jolt, from theory to code. `https://a16zcrypto.com/posts/article/building-on-lasso-and-jolt/`.
74. Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering*, 37(6):772–787, 2010.

75. D. Sohn, X. Li, and J. Rogers. Everything you always wanted to know about secure and private database systems (but were afraid to ask). *IEEE Data Eng. Bull.*, 47(2), 2024.
76. T. Solberg. A brief history of lookup arguments. https://github.com/ingonyama-zk/papers/blob/main/lookups.pdf, 2023.
77. Space and Time. Proof-of-sql co-processor, 2025. `https://github.com/spaceandtimefdn/sxt-proof-of-sql`.
78. R. Tamassia. Authenticated data structures. In *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, 1997.
79. A. Tomescu, I. Abraham, V. Buterin, J. Drake, D. Feist, and D. Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. In *International Conference on Security and Cryptography for Networks*, pages 45–64. Springer, 2020.
80. H. Wee and D. J. Wu. Succinct vector, polynomial, and functional commitments from lattices. In C. Hazay and M. Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part III*, volume 14006 of *Lecture Notes in Computer Science*, pages 385–416. Springer, 2023.
81. H. Wee and D. J. Wu. Succinct functional commitments for circuits from k-Lin. In *Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part II*, volume 14652 of *Lecture Notes in Computer Science*, pages 280–310. Springer, 2024.
82. A. Zapico, V. Buterin, D. Khovratovich, M. Maller, A. Nitulescu, and M. Simkin. Caulk: Lookup arguments in sublinear time. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3121–3134, 2022.
83. Zcash. The halo2 book. `https://github.com/zcash/halo2`.
84. Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, 2017.
85. Y. Zhang, J. Katz, and C. Papamanthou. IntegriDB: Verifiable SQL for outsourced databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, 2015.
86. Y. Zhang, J. Katz, and C. Papamanthou. An expressive (zero-knowledge) set accumulator. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, 2017.

A Concrete Efficiency Comparison to IntegriDB and vSQL

Below we discuss how our scheme concretely compares to IntegriDB and vSQL (the state of the art of VDBs based on non-recursive SNARKs). We also present a comparison summary (including asymptotics) in Table 5.

Concrete efficiency comparison to the state of the art on ADS-based VDB (IntegriDB):[26]

- *Verification time*: our experiments show a consistently efficient verifier for qedb: for aggregation queries such as SUM or MIN (independently of the database size) it runs in the order of milliseconds on our evaluation machine. When verifying results of the order of hundreds of thousands of rows, it runs in the order of the tenth of a second. Like our instantiation, IntegriDB's verifier also requires pairings. The number of pairings in IntegriDB, however, is logarithmic in the table size (in our case it is completely independent from it). We were able to replicate the benchmarks presented in Table 4 in [85] and conclude that, for the same query and on the same machine, IntegriDB's verifier is generally at least an order of magnitude slower than qedb's[27].
- *Proof size*: Our proof sizes shows an order of magnitude improvement compared to running IntegriDB on the same queries. On common queries with simple equality checks our proof size is around 0.5–1 KB independently of the number of rows in the table; IntegriDB's proof sizes, on the other hand, grow with the number of rows and are of approximately of size 5KB

[26] Note: some of the experimental results mentioned in this section refer to additional experiments to those reported in Section 5.

[27] The synthetic data used for this comparison were generated following the same approach as IntegriDB [49], with tables populated randomly.

for this type of queries even for the case of small tables[28]. Other types of queries show a substantially worse proof size than IntegriDB's. Queries with range checks occasionally show a proof size in IntegriDB close to the ballpark of those in qedb.
– *Preprocessing and proving time:* The asymptotic improvements for preprocessing times in qedb vs IntegriDB are confirmed by experiments, requiring roughly two orders of magnitude less time in qedb even on small tables (and become more prominent with larger tables). An end-to-end experimental comparison of proving times is non-trivial (see discussion in Remark 7), but simple experiments confirm that qedb performs competitively or better than IntegriDB for different queries/table sizes and without paying the price in proof size or verification time. For medium-sized tables ($\geq 100K$ rows), IntegriDB's prover was unable to run on our machine due to the memory overhead involved.

Concrete comparison for qedb *vs vSQL:* Above we compared against IntegriDB, the state of the art for VDBs based on authenticated data structures. While our main point of advantage against general-purpose proof systems is due to simplicity and modularity (at the price of generality), here we now discuss differences in performance between our system and vSQL [84], arguably the most prominent and best documented SNARK-like system tailored to database queries in literature with succinct proofs. We could not perform end-to-end comparisons against vSQL (see Remark 7), but we can nonetheless provide evidence of concrete improvements (on top of the asymptotic ones described in Table 5). Depending on the metric and the query, qedb's verifier and proof size show substantial improvements. For example, vSQL's verifier involves a number of pairings that grows with the logarithm of the proved circuit (which is *at least* as large as the database DB); in contrast, the number of pairings in qedb are independent of the size of DB. From the values reported in [84] we conclude that the proof size in vSQL is comparable to IntegriDB's and therefore substantially larger than ours. Comparing the proving time in qedb and vSQL is substantially harder due to the lack of a public implementation of the latter and the difference in expressivity between the two systems; however, on the queries supported by us, it is plausible that vSQL's prover would perform substantially worse: its running time grows at least linearly with the size of *the circuit* performing the query rather than *the table size* itself (see also discussion in the related work section and in Section 4)[29].

Remark 7 (On replicating results from prior works). While it was our initial intention to provide an apple-to-apple comparison, we encountered a few challenges in the process, in particular in the case of the prominent works of IntegriDB and vSQL. Specifically, vSQL's code is not publicly available, and IntegriDB's code is incomplete—e.g., it lacks the TPC-H benchmarks—and is fragile (crashing with even minor modifications due to seemingly subtle bugs). An end-to-end comparison on complex benchmarks against these systems was not easy without reimplementing them from scratch. We thought this would be out of the scope of this paper given the already stated benefits of our approach in terms of modeling, simplicity, asymptotic behavior. At the same time, we strived to still provide strong evidence that qedb does have concrete improvements in practice; see Section 5.

[28] More details on our experimental findings for IntegriDB obtained running the code provided in their official repository: SELECT queries filtered by value equality on tables with approximately 1000 rows IntegriDB gives a proof of 5KB (it is not a fixed number because it is sensitive to the distribution of the values in the database); for aggregate SUM queries on tables of the same size it provides proofs of roughly 10KB.

[29] We leave as future work to extend this section with additional comparison between qedb and another fairly well documented VDB from general-purpose (recursive) SNARKs, the recent work in PoneglyphDB [44]. The reader can find a preliminary comparison in the related work section.

Table 5: Comparison of expressive and succinct verifiable databases constructions. We compare qualitative features in the top tables and efficiency metrics in the bottom table. In the table, we focus on state of the art works based on authenticated data structures and non-recursive SNARKs. See related work section for additional comparison and Section 4 for further discussion.

Scheme	**Setup**	**Core Building Blocks**	**Decouples Logic & Instantiations?**	**"Circuit-less"**	**Expressivity**
IntegriDB [85]	powers of τ	Merkle trees, pairing-based accumulators	✗	✓	Fig. 1 (top part only)
vSQL [84]	PST [70]	multivar. poly commitments, GKR [43]	✗	✗	anything expressible as an arithmetic circuit
This work	powers of τ	KZG	✓	✓	Fig. 1 (all)

Scheme	**Overhead in $\lvert\pi\rvert$, $V_{\mathbf{time}}$** (queries w/o JOINs)	**Overhead in $\lvert\pi\rvert$, $V_{\mathbf{time}}$** (JOINs)	**Preprocessing & server storage**
IntegriDB [85]	$\log(\lvert\mathsf{column}\rvert)$	$\lvert\mathsf{resp}\rvert \cdot \log \lvert\mathsf{column}\rvert$	$\lvert\mathsf{db}\rvert + n_{\text{cols}}^2$
vSQL [84]	polylog$\lvert\mathsf{db}\rvert$	polylog$\lvert\mathsf{db}\rvert$	$\lvert\mathsf{db}\rvert$
This work	$\lvert\mathsf{qry}\rvert$	$\lvert\mathsf{resp}\rvert$	$\lvert\mathsf{db}\rvert$

(NB: commonly $\lvert\mathsf{resp}\rvert \ll \lvert\mathsf{column}\rvert \ll \lvert\mathsf{db}\rvert$; for aggregate queries, $\lvert\mathsf{qry}\rvert \approx \lvert\mathsf{resp}\rvert$, else $\lvert\mathsf{qry}\rvert \ll \lvert\mathsf{resp}\rvert$).

Other notes: All schemes have a digest of constant size. The overhead for proof size and verification time below is an *additive* overhead in addition to the query qry and the response resp. For simplicity below we assume JOINs of two tables only. All quantities are implicitly asymptotic. n_{cols} denotes the maximum number of columns in a table.