

TruthTable: A Verifiable Query Engine

Bharath Namboothiry* namboo@upenn.edu UPenn
Alireza Shirzad* alrshir@upenn.edu UPenn
Spencer Solit ssolit44@gmail.com UPenn
Ryan Marcus rcmarcus@upenn.edu UPenn
Pratyush Mishra prat@upenn.edu UPenn

June 14, 2026

Abstract

We present TruthTable, a verifiable database engine that allows a prover to produce a succinct proof that convinces a verifier of the correct execution of the verifier's SQL query over the prover's committed database.

TruthTable supports a large subset of SQL, enabling it to prove 17 out of 22 queries in the standard TPC-H benchmark. To our knowledge, this is the widest support out of all prior work. Moreover, TruthTable's proofs are small, and fast to generate and verify: on the TPC-H benchmark with a database of a million rows, TruthTable's average proving time is 55 seconds, average verification time is 32 ms, and average proof size is 24 kB. Compared to prior work, TruthTable's proving times are between 6.3×-63× better, while the verification times and proof sizes are competitive.

TruthTable achieves these properties via a codesign of cryptography and database techniques. On the cryptographic front, we propose a new polynomial representation of database tables, and design new subprotocols for proving the correct execution of various relational operators on these representations. On the database front, we propose a query planner that optimizes queries for minimal proving time, as opposed to minimal execution time. We also design new optimizations for this planner that reduce proving time by up to 2×.

*Equal contribution, order decided by coin flip.

Contents

1 Introduction **4**
- 1.1 Our contribution: TruthTable . . . 4

2 Overview of TruthTable **6**
- 2.1 Construction strategy for TruthTable . . . 6
  - 2.1.1 Commit . . . 7
  - 2.1.2 Prover and verifier . . . 7
- 2.2 Proof-aware query planner . . . 7
- 2.3 Proof engine . . . 8

3 Preliminaries **9**

4 Table arithmetization **10**

5 PIOP toolbox **11**
- 5.1 Booleanity . . . 11
- 5.2 Data equality . . . 11
- 5.3 Non-Zero check . . . 11
- 5.4 Table fingerprints . . . 12
- 5.5 Logarithmic derivative checks . . . 12
- 5.6 Lookup . . . 12
- 5.7 Non-negativity and positivity . . . 13
- 5.8 Lexicographic sorting . . . 14
- 5.9 No Duplicate check . . . 15
- 5.10 Support check . . . 15

6 PIOPs for query plan nodes **16**
- 6.1 Expressions . . . 16
- 6.2 Filter . . . 17
- 6.3 Project . . . 18
- 6.4 Compaction . . . 18
- 6.5 Order-By . . . 18
- 6.6 Limit . . . 19
- 6.7 Distinct . . . 19
- 6.8 Aggregate . . . 20
- 6.9 Inner join . . . 21
- 6.10 Opening the result commitment . . . 22

7 Query plan optimization **22**

8 Implementation **23**
- 8.1 TruthTable System . . . 23
- 8.2 `ark-piop` infrastructure . . . 23

9 Evaluation **24**
9.1 Benchmarks on TPC-H . . . 25
9.2 The effect of optimizations . . . 26
9.3 Comparison against Poneglyph . . . 26
9.4 Comparison against `PoSQL` and `qedb` . . . 27

10 Related work **28**

Acknowledgements **30**

References **30**

A Extended preliminaries **31**
A.1 s-Contiguous Polynomial Identity . . . 31
A.2 Commit-carrying SNARKs . . . 32
A.3 PIOP security properties . . . 32
A.4 Deferred Low-Level PIOPs . . . 33

B Deferred Protocols **33**
B.1 NoDupcheck without sorting . . . 34
B.2 Keyed sumcheck . . . 35
B.3 Matching pair check . . . 35

C Security of TruthTable's protocol **36**

D Heuristic degree reduction **37**

E Benchmark Queries **37**
E.1 TPC-H Benchmark Queries . . . 38
E.2 Synthetic Benchmark Queries . . . 44

1 Introduction

A fundamental design pattern in modern computer applications is to store data in a database and subsequently issue SQL queries to this database to retrieve (parts of) the data. In many applications, the party that stores the database and the party that issues database queries live in different trust domains. In particular, the query-issuer might not trust that the database's responses are correct. Indeed, in a number of applications, the database server might even be *incentivized* to report incorrect results.

Consider for example a loan-issuing smart contract. To safely determine loan parameters such as duration and interest rate, the smart contract might want to develop a risk profile from the user's transaction history. We can view this risk profile calculation as a query against the database of historical blockchain transactions. Unfortunately, since it is the user that invokes the smart contract, they are also responsible for providing the response to this risk profile query, which means that they can provide an incorrect response. Indeed, high-risk users are incentivized to provide an incorrect response to create the illusion of a better risk profile.

To prevent such incorrect responses, the smart contract needs a mechanism for checking that a claimed query response was correctly generated with respect to a particular database and query. *Verifiable database* systems provide exactly such a mechanism. They enable a database owner to publish a commitment to a database, and to later produce succinctly-verifiable proofs that a claimed result is indeed the result of executing a query against the committed database. In the last few years, there has been a proliferation of verifiable databases [ZKP15; ZGKPP17b; GFN25; Spa25; BBCRT25]. Unfortunately, all of these systems have shortcomings along at least one of the following dimensions:

- *Expressivity:* existing general-purpose verifiable databases tend to support only a limited subset of SQL that does not capture commonplace queries. For instance, the academic state-of-the-art Poneglyph [GFN25] supports only 6 out of the 22 queries in the standard TPC-H [PF00] benchmark, while industrial systems such as `PoSQL` [Spa25] and `qedb` [BBCRT25] support only a very limited subset of SQL operations (for example, `qedb` does not support joins or limit/offset) and consequently do not support TPC-H queries at all.

- *Efficiency:* existing systems incur high costs for generating proofs even for relatively simple queries on moderately large databases. Take one of the simplest operators in SQL: Filter. On a table of just 2^{19} (roughly half a million) rows, `PoSQL` [Spa25] takes over 82 seconds to prove a single filter query, and `qedb` [BBCRT25] takes over 45 seconds with a verifier time exceeding 1.5 seconds.

- *Extensibility:* existing systems are difficult to extend with efficient support for new kinds of SQL operators. For example, Poneglyph [GFN25] hand-crafts a separate arithmetic circuit for each supported query, hardcoding the query plan and operator implementations directly into a low-level constraint system; adding a new operator, or even a small variation on an existing query, requires writing and auditing fresh circuits from scratch, a process that is both labor-intensive and error-prone.

1.1 Our contribution: TruthTable

In this work, we introduce TruthTable, a verifiable database that provides cryptographic guarantees of query-response correctness for a large subset of SQL queries, and which has the following desirable

properties:

TruthTable supports a large subset of SQL, including almost all major relational operators such as filters, aggregations, projections, and inner joins, and operations over key data types such as integer and fixed-point arithmetic. This enables TruthTable to support 17 out of 22 TPC-H [PF00] benchmark queries. This is the most out of any prior work.

TruthTable is able to prove these queries efficiently: on the TPC-H benchmark with scale factor 0.1 (corresponding to roughly 1 million rows), TruthTable's average proving time (with 4 threads) is 55 seconds, average verification time is 32 ms, and average proof size is 24 kB. Furthermore, these costs are heavily skewed by complex queries: single-threaded, the simplest queries (e.g., Q2) prove in only about 38 seconds, whereas the most expensive, Q10, takes roughly 314 seconds. Compared to prior work, TruthTable proves up to 63× faster, and up to 6.3× faster even than the closest competitor, Poneglyph, while its verification times and proof sizes remain competitive.

TruthTable achieves these properties via a careful design of cryptographic protocols and database query planning techniques; we elaborate on these next.

TruthTable's cryptographic protocol is a composition of modular subprotocols for the various relational operators. Underlying this design is a novel encoding of relational data as polynomials, and novel polynomial IOPs [CHMMVW20; BFS20] for efficiently proving properties about this data.

In more detail, TruthTable encodes tables in a column-by-column manner, with each column being represented as a polynomial via its multilinear extension. We accompany each table with an "activator" polynomial which indicates which rows are active; inactive locations include padding rows and those rows rejected by filters or other operations. We then design PIOPs based on multivariate sumcheck [LFKN92] that minimize proving time. We first build PIOPs for proving basic properties of individual columns (e.g., booleanity, sortedness, uniqueness), and then build upon these to develop protocols for high-level relational operators such as filters, joins, aggregates and more.

Our polynomial encodings and IOPs enable efficient compilation to a succinct argument via the standard 'PIOP + PC → SNARK' paradigm [CHMMVW20; BFS20].

TruthTable's query planner takes advantage of the fact that our protocols often exhibit a different cost profile compared to the standard query execution cost profile. While we can (and do) reuse many standard optimizations such as filter and projection pushdown, limit pushdown, common-subexpression elimination, and more, we also introduce new optimizations. For example, reducing the size of polynomial encodings after operations such as filter, can greatly improve proving, and so our query planner selectively "compacts" tables by removing filtered rows when it would be beneficial. By judiciously applying these new optimizations, our query planner improves TruthTable's proving times on the TPC-H benchmark by up to 2.06×, with a geometric-mean speedup of 1.26× across the benchmark.

Remark 1.1 (privacy)**.** We emphasize that TruthTable (like most prior work) focuses on providing succinctly verifiable proofs of query execution, and does not aim to provide any privacy guarantees about the committed database. This lack of privacy still suffices for many applications where the data does not need to be hidden, but the verifier is too weak to store it in its entirety. This includes the blockchain application mentioned in Section 1, but also other systems that query public data, such as audits to transparency logs, and, perhaps speculatively, agentic AI systems where one agent needs to establish correctness of (perhaps hallucinated) query results provided by other agents.

Remark 1.2 (trusting the database commitment)**.** TruthTable guarantees that a query result is consistent with a given database commitment, but not that the commitment faithfully represents

the underlying data; bootstrapping a trustworthy commitment is an orthogonal concern left to the application. Among possible approaches, the commitment could be produced by a trusted party (e.g., trusted hardware) that commits once and then goes offline; recomputed and vouched for by an occasional *auditor* that signs it or posts it to a public ledger, e.g., in certificate transparency applications; or established through a consensus mechanism, e.g., in blockchain applications.

2 Overview of TruthTable

System model for verifiable databases. A verifiable database system involves three parties: a *trusted* database owner $\mathcal{D}$ who holds a database DB, an *untrusted* database server $\mathcal{S}$, and a client $\mathcal{C}$. These parties interact as follows.

1. The database owner $\mathcal{D}$ generates a cryptographic commitment $\mathsf{cm_{DB}}$ to the database DB, and sends $\mathsf{cm_{DB}}$ to both $\mathcal{C}$ and $\mathcal{S}$ (or equivalently, posts it to a public ledger). $\mathcal{D}$ also sends the database DB to $\mathcal{S}$. At this point, $\mathcal{D}$'s job is done, and it can go offline.
2. $\mathcal{C}$ issues a query Q to $\mathcal{S}$. The latter executes Q on DB, and additionally generates a cryptographic proof π_{Q} certifying that the query result is correct with respect to the committed database $\mathsf{cm_{DB}}$.
3. $\mathcal{S}$ sends both res and π_{Q} to $\mathcal{C}$, who verifies the proof π_{Q} to confirm res is indeed the correct query result.

Definition of verifiable databases. Formally, a verifiable database system is a commit-carrying SNARK (cc-SNARK) [CFQ19] for the following ternary relation:

$$\mathcal{R}_{\mathsf{SQL}} = \{\, (\mathbb{x} = (\mathsf{Q}, \mathsf{res}), \mathbb{y} = \mathsf{DB}, \mathbb{w} = \bot) \mid \mathsf{res} \in R_{\mathsf{Q}}(\mathsf{DB}) \,\},$$

where $R_{\mathsf{Q}}(\mathsf{DB})$ denotes the set of valid query results for the query Q on the database DB with respect to SQL semantics.

Note that in a ternary relation, $\mathbb{x}$ denotes the explicit instance, $\mathbb{y}$ denotes an implicit instance, and $\mathbb{w}$ denotes the witness. The distinction between explicit and implicit instances differentiates between those parts of the instance that are read by the cc-SNARK verifier entirely ($\mathbb{x}$) versus those that are read only in committed form ($\mathbb{y}$).

We describe below the syntax of the cc-SNARK algorithms specialized to this relation.

- $\mathsf{Setup}(1^\lambda, N) \to (\mathsf{pk}, \mathsf{vk}, \mathsf{ck})$, on input a database size N and security parameter λ, outputs a prover key pk, verifier key vk, and commitment key ck.
- $\mathsf{Commit}(\mathsf{ck}, \mathsf{DB}) \to \mathsf{cm_{DB}}$, on input a database DB, outputs a commitment $\mathsf{cm_{DB}}$ to the database.
- $\mathsf{Prove}(\mathsf{pk}, \mathsf{DB}, \mathsf{cm_{DB}}, \mathsf{Q}, \mathsf{res}) \to \pi_{\mathsf{Q}}$, on input a database DB, the database commitment $\mathsf{cm_{DB}}$, a query Q, and the query result res, outputs a proof π_{Q} that res is the result of executing Q on DB.
- $\mathsf{Verify}(\mathsf{vk}, \mathsf{cm_{DB}}, \mathsf{Q}, \mathsf{res}, \pi_{\mathsf{Q}}) \to \{0, 1\}$, on input a database commitment $\mathsf{cm_{DB}}$, a query Q, a query result res, and a proof π_{Q}, accepts if the proof is valid.

These algorithms must satisfy standard properties of completeness and knowledge soundness for cc-SNARKs [CFQ19]; see Appendix A for details.

2.1 Construction strategy for TruthTable

We now describe how TruthTable realizes the various verifiable database algorithms. We focus on the commitment, prover, and verifier algorithms.

2.1.1 Commit

TruthTable represents SQL tables as tuples of (multilinear) polynomials, and commits to these via a polynomial commitment scheme [KZG10]. Each column is represented as a separate polynomial, and hence the size of a table commitment is the number of columns times the size of a single commitment. Similarly, the cost of producing commitments is the product of the number of columns and the cost of committing to a single column. We refer to the polynomial encoding of a table T as its *arithmetization*, and denote it by $\hat{\mathsf{T}}$ and we extend the same hat convention to columns and other table-derived objects throughout the paper.

Looking ahead, our polynomial encoding will enable efficient protocols for checking that SQL operators were correctly applied to the encoded tables. We provide details about this encoding in Section 4.

2.1.2 Prover and verifier

To check that a claimed instance $(\mathsf{Q}, \mathsf{res})$ is valid for a committed database DB (i.e., $((\mathsf{Q}, \mathsf{res}), \mathsf{DB}) \in \mathcal{R}_{\mathsf{SQL}}$), TruthTable's prover and verifier proceed in the following step-by-step manner. First, they invoke a parser to transform the query Q into a *query plan* QP, which is a tree representation of the query that makes analysis and optimization easier. Next, both invoke TruthTable's query planner to optimize the query plan. Finally, both invoke TruthTable's proof engine on this query plan. At a high level, in the latter step, both parties traverse the plan node-by-node, and execute subprotocols that prove the correctness of each node.

In the rest of this section, we provide details on each of the two foregoing components. We begin with a description of the core data structure that TruthTable operates over, namely the query plan.

A **query plan** QP for a query Q is an alternate tree-based representation of Q where nodes represent relational operators (e.g., filter, join, limit), and edges represent the flow of data between these operators. Edges into the leaves represent the input database tables, while the edge leaving the root represents the output table. There exist numerous parsers which parse a SQL query into a query plan. Fig. 1b depicts the query plan resulting from such a parsing process for the query in Fig. 1a.

We introduce query plans because their modular nature makes reasoning about protocol construction and analysis much simpler. Indeed, instead of working with $\mathcal{R}_{\mathsf{SQL}}$, we will instead work with an equivalent relation $\mathcal{R}_{\mathsf{QP}}$ that talks about query plans instead:

$$\mathcal{R}_{\mathsf{QP}} = \{ (\mathbb{x} = (\mathsf{QP}, \mathsf{res}), \mathbb{y} = \mathsf{DB}, \mathbb{w} = \bot) \mid \mathsf{res} \in R_{\mathsf{QP}}(\mathsf{DB}) \} \quad .$$

The two relations are equivalent: a claimed query output res is the correct result of executing a query Q on DB (i.e., $((\mathsf{Q}, \mathsf{res}), \mathsf{DB}) \in \mathcal{R}_{\mathsf{SQL}}$) if and only if res is the result of executing the corresponding QP (produced, e.g., by a standard SQL parser) on DB (i.e., $((\mathsf{QP}, \mathsf{res}), \mathsf{DB}) \in \mathcal{R}_{\mathsf{QP}}$).

2.2 Proof-aware query planner

TruthTable's query planner takes as input a query plan QP derived by parsing a query Q and optimizes it into an equivalent plan $\mathsf{QP}_{\mathsf{opt}}$ that minimizes proving cost. It does so via a sequence of semantics-preserving query rewrites. There is a rich literature on such query optimizations, and TruthTable's query planner relies on these optimizations.

However, it also introduces new optimizations by leveraging the different cost model of execution. For example, our table encoding maintains a "bitmap" which specifies active rows in the table. This

minimizes recommitment cost in, e.g., our filter protocol, because the protocol only has to update the commitment to this bitmap and not to the entire table. However, when the filter is selective, only updating the bitmap makes subsequent operations more expensive as they have to operate over the entire table and handle missing rows. To avoid this cost, our query planner notices that when the filter is selective recommitment cost is low, and so it inserts a "compaction" step that recommits to the entire table. This optimization reduces prover time by up to 1.64× for certain queries.

We note that this optimization leverages an optimization opportunity that is available only during proving: it uses exact information about the intermediate results (namely, size of post-filter tables) to optimize the query plan. Standard query planners generally do not make such data-dependent optimizations, as computing the exact cardinalities tends to be as expensive as the original query itself.

However, using such data-dependent optimizations must be done carefully to avoid soundness issues, as some data-dependent optimizations are only soundly applicable if the data satisfies particular conditions. To use such optimizations, the prover has to demonstrate that these preconditions are satisfied. By carefully designing the set of allowed rewrites, we guarantee that $((\mathsf{QP}, \mathsf{res}), \mathsf{DB}) \in \mathcal{L}_{\mathsf{QP}}$ if and only if $((\mathsf{QP}_{\mathsf{opt}}, \mathsf{res}), \mathsf{DB}) \in \mathcal{L}_{\mathsf{QP}}$, allowing us to focus on proving the latter. We provide more details on our query planner in Section 7.

2.3 Proof engine

On input an optimized plan $\mathsf{QP}_{\mathsf{opt}}$ generated by the query planner, TruthTable's proof engine constructs a protocol for proving that $((\mathsf{QP}_{\mathsf{opt}}, \mathsf{res}), \mathsf{DB}) \in \mathcal{L}_{\mathsf{QP}}$ as follows. We first design a Polynomial Interactive Oracle Proof (PIOP) [CHMMVW20; BFS20] for QP, and compile this to a succinct argument using a polynomial commitment scheme [KZG10]. In particular, we use the same PC scheme from the commit phase. The compilation phase is standard, so in this paper we focus on the PIOP construction.

The prover first generates *table assignment* $\{\mathsf{T}_i\}_{i=1}^m$ consisting of intermediate result tables that label the m non-leaf edges of QP. It then encodes these as polynomials, and sends to these as oracles to the verifier. Then, for each node $\nu \in \mathsf{QP}$, the prover and verifier engage in a subPIOP that asserts that the output table oracle O_ν is the result of executing $\nu.\mathsf{op}$ on the input table oracles $\{\mathtt{I}\}_\nu$.[1] Finally, the prover and verifier engage in a *result check* to verify that the root oracle opens to the purported query result res. An example proof strategy is illustrated in Fig. 1c for the query plan in Fig. 1b.

More formally, each of these node-specific subPIOPs proves membership in a "node relation" that is similar to $\mathcal{R}_{\mathsf{QP}}$, except restricted to just the operation for that node. For example, for a filter σ with predicate p, $\mathcal{R}_\sigma$ requires that (i) every row of $\mathtt{O}$ appears in $\mathtt{I}$, (ii) every row of $\mathtt{O}$ satisfies p, and (iii) every row of $\mathtt{I}$ absent from $\mathtt{O}$ fails p.[2] In Section 6, we specify these relations for each supported operator in the query plan.

This strategy is complete and sound as long as the per-node checks are complete and sound. We provide formal details of the foregoing construction sketch in Appendix C

Remark 2.1 (alternate backends for proving)**.** TruthTable picks one strategy for proving the query plan execution, but other strategies are possible. For example, the per-node checks can be written

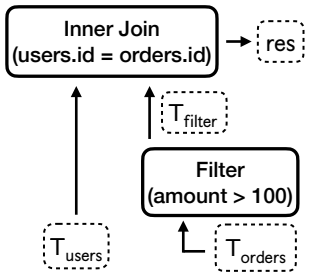
[1] More precisely, since there might be a variety of possible correct execution results for each operation, we require the output table O_ν to be in the set of possible results $R_{\nu.\mathsf{op}}(\{\mathtt{I}\})$.

[2] Precisely, the subPIOPs prove "arithmetized" versions of the node relations, where each table is replaced with its arithmetization. See Section 4.

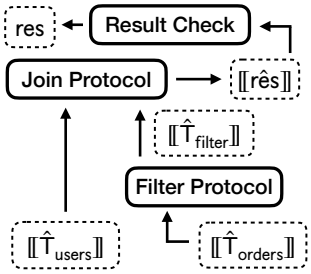
as arithmetic circuits, allowing the use of existing optimized SNARKs for circuits to prove these circuits. However, as we show in our experiments, using these "generic" proving backends tends to incur higher overhead than our specialized protocols.

```
SELECT * FROM users
 JOIN orders
   ON users.id = orders.id
WHERE orders.amount > 100;
```

(a) Example SQL query.



(b) Query plan



(c) Table assignment and PIOPs

Figure 1: Query plan (b) and proof strategy (c) for query (a)

3 Preliminaries

We introduce notation and cryptographic background required to understand TruthTable's protocols.

Boolean hypercube. The μ-dimensional Boolean hypercube $\mathscr{H}_\mu$ is the set $\{0,1\}^\mu$. It has size $n = 2^\mu$. We will commonly interpret elements of $\mathscr{H}_\mu$ as integers in the range $[0, n-1]$.

Polynomials. We write $p \in \mathbb{F}^{\leq d}[X_1, \ldots, X_\mu]$ (or, concisely, $p \in \mathbb{F}^{\leq d}[\mathbf{X}]$) to denote a polynomial in μ variables over the field $\mathbb{F}$, where each variable has degree at most d. We will work primarily with multilinear polynomials (i.e. $d = 1$). We use $*$ to denote multiplication of two polynomials or multiplication of a polynomial by a scalar.

Let $v \in \mathbb{F}^N$ be a vector of field elements with $N = 2^\mu$ for some $\mu \in \mathbb{N}$. The *multilinear extension* $\tilde{v}$ of v is the unique multilinear polynomial whose evaluations at the i-th point in the μ-dimensional boolean hypercube equal the i-th element of v. We recall some useful polynomials.

- *Index polynomial* $\mathcal{I}_\mu(X) := \sum_{i=1}^{\mu} 2^{i-1} X_i$. It maps the i-th point in $\mathscr{H}_\mu$ to the value $i - 1 \in \mathbb{F}$.
- *Equality* polynomial $\mathrm{eq}(X, Y) := \prod_{i=1}^{\mu}((1 - X_i)(1 - Y_i) + X_i Y_i)$. For $x, y \in \mathscr{H}_\mu$, $\mathrm{eq}(x, y)$ equals 1 if $x = y$, and equals 0 otherwise.
- The *s-contiguous* polynomial $\mathcal{C}_s(X) := \sum_{i=1}^{\mu} s_i(1 - X_i) \prod_{j=1}^{i-1} (s_j X_j + (1 - s_j)(1 - X_j))$ is 1 at the first s points in $\mathscr{H}_\mu$ and 0 elsewhere, where $s \in [2^\mu - 1]$, and $\{s_i\}_{i=1}^{\mu}$ is its base-2 representation (see Section A.1).

SNARKs via polynomials. TruthTable follows the now-standard methodology for constructing SNARKs by combining PIOPs with PC schemes [BFS20; CHMMVW20].

A PIOP is an interactive proof between a prover $\mathbf{P}$ and a verifier $\mathbf{V}$, in which the prover sends polynomial oracles and the verifier queries these oracles before deciding acceptance. An oracle to a polynomial p supporting such 'evaluation query' access is denoted by $[\![p]\!]$.

A polynomial commitment [KZG10] enables committing to a polynomial and later producing evaluations proofs that assert correct evaluation of the polynomial at particular points.

The standard 'PIOP + PC $\rightarrow$ SNARK' methodology combines PIOPs and PC schemes to produce a SNARK by replacing each polynomial oracle with a polynomial commitment to it. Oracle query access is emulated via evaluation proofs.

Common PIOPs. TruthTable's PIOPs invoke PIOPs for certain "core" claims, such as sumcheck, zerocheck, and rational sumcheck claims. Sumcheck is standard [LFKN92], and we provide a description of the other two in Section A.4.

4 Table arithmetization

We now describe how we represent columns and tables as polynomials for use with our PIOPs in Sections 5 and 6.

TruthTable represents tables and columns as multilinear polynomials over a Boolean hypercube. Let $\mathbb{F}$ be a finite field of sufficiently large characteristic.

Notation: arithmetized objects. We use bare symbols such as T and c for standard tables and columns, and the hatted forms $\hat{\mathsf{T}}$ and $\hat{\mathsf{c}}$ for their *arithmetized* polynomial representations. Standard tables are accessed with square brackets at integer indices, as in $\mathsf{T.k}[i]$ or $\mathsf{c}[i]$ for $i \in [|\mathsf{T}|]$. Arithmetized entries are evaluated with parentheses at hypercube points, as in $\hat{\mathsf{T}}.\mathsf{k}(x)$ or $\hat{\mathsf{c}}(x)$ for $x \in \mathscr{H}_\mu$. We extend the same hat convention to all arithmetized table-derived objects (such as the source columns of a join) introduced later in the paper.

Converting data to $\mathbb{F}$ elements. We map each column into the prime field $\mathbb{F}_p$ according to its declared type: we embed unsigned integers directly, reduce signed integers and the 128-bit encodings of durations and intervals modulo p, encode booleans as 0/1, represent dates and timestamps by their Unix epoch, and hash strings into $\mathbb{F}_p$.

Columns. A column c with n entries is arithmetized by embedding it into $\mathscr{H}_\mu$ for some $\mu \geq \lceil \log_2 n \rceil$. The arithmetization $\hat{\mathsf{c}}$ is a pair of multilinear polynomials $\hat{\mathsf{c}} := (\hat{\mathsf{c}}.\mathsf{a}, \hat{\mathsf{c}}.\mathsf{d})$, where $\hat{\mathsf{c}}.\mathsf{a} : \mathscr{H}_\mu \to \{0,1\}$ is the *activator* polynomial and $\hat{\mathsf{c}}.\mathsf{d} : \mathscr{H}_\mu \to \mathbb{F}$ is the *data* polynomial. The activator specifies which n hypercube indices are *active* (i.e., correspond to real rows rather than padding), and the data polynomial encodes the column values on those active indices. The value of the arithmetized column at index x is $\hat{\mathsf{c}}(x) := \hat{\mathsf{c}}.\mathsf{a}(x) * \hat{\mathsf{c}}.\mathsf{d}(x)$, so padding indices contribute zero by construction. When the column identifier is clear from context, we drop it and write a and d in place of $\hat{\mathsf{c}}.\mathsf{a}$ and $\hat{\mathsf{c}}.\mathsf{d}$.

Tables. A table T with n rows and k columns is arithmetized as $\hat{\mathsf{T}} := (\hat{\mathsf{T}}.\mathsf{a}, [\hat{\mathsf{T}}.\mathsf{d}_1, \ldots, \hat{\mathsf{T}}.\mathsf{d}_k])$, where all columns share a common activator $\hat{\mathsf{T}}.\mathsf{a}$ that is active at n points in $\mathscr{H}_\mu$. For any $x \in \mathscr{H}_\mu$ with $\hat{\mathsf{T}}.\mathsf{a}(x) = 1$, the tuple $(\hat{\mathsf{T}}.\mathsf{d}_1(x), \ldots, \hat{\mathsf{T}}.\mathsf{d}_k(x))$ represents a row of the T. Single columns may be viewed as single-column tables under this representation. As with columns, we write $\hat{\mathsf{T}}.\mathsf{a}$ and $\hat{\mathsf{T}}.\mathsf{d}_i$ for the activator and data polynomials.

Active indices and layouts. Arithmetizing a column or table with n rows requires choosing which n hypercube indices are *active*. We collect these into the active set $A(\hat{\mathsf{c}}) := \{x \in \mathscr{H}_\mu \mid \hat{\mathsf{c}}.\mathsf{a}(x) = 1\}$ of an arithmetized column $\hat{\mathsf{c}}$ (or table $\hat{\mathsf{T}}$). For columns, we additionally define the active set for a specific value. For a field element $k \in \mathbb{F}$ we further define $A_k(\hat{\mathsf{c}}) := \{x \in A(\hat{\mathsf{c}}) \mid \hat{\mathsf{c}}.\mathsf{d}(x) = k\}$. Many relational properties, such as multiplicities, grouping behavior, and join cardinalities, are naturally expressed as algebraic statements about these sets and their sizes.

We use one of two layouts for the active set. By default we use a *contiguous* layout, setting the activator to the n-contiguous polynomial $\mathcal{C}_{\mu,n}$, which evaluates to 1 on exactly the first n indices and to 0 elsewhere. This layout has two advantages: it places all active indices at the start of the hypercube, and $\mathcal{C}_{\mu,n}$ has a concise closed form (see Section 3) that the verifier can evaluate on its own. The verifier therefore need not receive the activator explicitly or check its booleanity. Some settings (e.g., Section 6.2) instead require a *sparse* layout, in which the active indices are arbitrarily

scattered throughout the hypercube. In such cases the prover must send the activator explicitly and prove its booleanity within the associated node protocol.

Arithmetized relations. Our PIOPs for relational operators prove membership in *table languages*, which are NP languages whose explicit instance contains input and output tables that are related via the operator. TruthTable's PIOPs, however, do not prove table languages directly; this is because the PIOP verifier cannot read the tables in the input explicitly, and must instead obtain polynomial oracle access to them. Hence our PIOPs actually prove *arithmetized languages*. In more detail, an arithmetized language associated with a table language $\mathcal{L}_{\mathsf{op}}$ is $\hat{\mathcal{L}}_{\mathsf{op}} := \left\{(\mathbb{x} = x, \mathbb{y} = (\{\hat{\mathsf{I}}\}, \hat{\mathsf{O}})) \;\middle|\; ((x, \{\mathsf{I}\}, \mathsf{O}), \bot) \in \mathcal{L}_{\mathsf{op}}\right\}$, where each $\hat{\mathsf{T}}$ is the arithmetization of the corresponding concrete table T. While we will present the relations $\mathcal{R}_{\mathsf{op}}$ in the context of concrete tables for clarity, the corresponding arithmetized relations $\hat{\mathcal{R}_{\mathsf{op}}}$ are what we will actually prove in our PIOPs.

5 PIOP toolbox

We present a toolbox of PIOPs that will be used in Section 6 to construct PIOPs for database operators and expressions. All PIOPs that operate over columns/tables can also operate directly over polynomials by setting the activator to be the constant 1.

5.1 Booleanity

Boolcheck proves that all active entries in a column $\hat{\mathsf{c}} = (\hat{\mathsf{a}}, \hat{\mathsf{d}})$ are either 0 or 1 using the trick that a value v lies in $\{0, 1\}$ if and only if $v(1 - v) = 0$. Specifically, we prove that $p = \hat{\mathsf{a}} * \hat{\mathsf{d}} * (1 - \hat{\mathsf{d}}) = 0$. If x is active then $\hat{\mathsf{a}}(x) = 1$ so $p(x) = 0$ if and only if $\hat{\mathsf{d}}(x) \in \{0, 1\}$. Inactive points are zeroed out automatically by $\hat{\mathsf{a}}$

PIOP 1: Boolcheck. Inputs: $[\![\hat{\mathsf{c}} = (\hat{\mathsf{a}}, \hat{\mathsf{d}})]\!]$

1. Invoke Zerocheck to prove that $\hat{\mathsf{a}} * (1 - \hat{\mathsf{d}}) * \hat{\mathsf{d}} = 0$.

5.2 Data equality

DataEqcheck proves that two data polynomials $\hat{\mathsf{d}_{\mathsf{L}}}$ and $\hat{\mathsf{d}_{\mathsf{R}}}$ are equal at all points activated by $\hat{\mathsf{a}}$ by proving that $p = \hat{\mathsf{a}} * (\hat{\mathsf{d}_{\mathsf{L}}} - \hat{\mathsf{d}_{\mathsf{R}}}) = 0$ over $\mathscr{H}_\mu$. If x is active then $\hat{\mathsf{a}}(x) = 1$ so $p(x) = 0$ if and only if $\hat{\mathsf{d}_{\mathsf{L}}}(x) = \hat{\mathsf{d}_{\mathsf{R}}}(x)$ as desired. Inactive points are zeroed out automatically by $\hat{\mathsf{a}}$

PIOP 2: DataEqcheck. Inputs: $[\![\hat{\mathsf{d}_{\mathsf{L}}}]\!], [\![\hat{\mathsf{d}_{\mathsf{R}}}]\!], [\![\hat{\mathsf{a}}]\!]$

1. Invoke Zerocheck to prove that $\hat{\mathsf{a}} * (\hat{\mathsf{d}_{\mathsf{L}}} - \hat{\mathsf{d}_{\mathsf{R}}}) = 0$.

5.3 Non-Zero check

NonZerocheck enforces that every active entry of a column is non-zero. It leverages the idea that a field element is non-zero if and only if it has an inverse in the field. $\mathbf{P}$ computes a polynomial $\hat{\mathsf{inv}}$ whose evaluation at the i-th active index is $\hat{\mathsf{d}}(i)^{-1}$, and is zero at inactive indices. Now, if $\hat{\mathsf{inv}}$ is correctly constructed, then $\hat{\mathsf{d}} * \hat{\mathsf{inv}} = \hat{\mathsf{a}}$ on the entire hypercube: at active indices both sides equal 1,

and at inactive indices both sides equal 0 (since $\hat{\mathsf{inv}}$ is set to 0 there). Our PIOP proves this with Zerocheck.

PIOP 3: NonZerocheck. Inputs: $[\![\hat{\mathsf{c}} = (\hat{\mathsf{a}}, \hat{\mathsf{d}})]\!]$

1. **P** sends **V** the oracle $[\![\hat{\mathsf{inv}}]\!]$ as defined above, and then the two invoke Zerocheck to prove $\hat{\mathsf{d}} * \hat{\mathsf{inv}} - \hat{\mathsf{a}} = 0$.

5.4 Table fingerprints

The fingerprint of an arithmetized table $\hat{\mathsf{T}} = (\hat{\mathsf{a}}, [\hat{\mathsf{d}}_1, \ldots, \hat{\mathsf{d}}_k])$ is a single column whose evaluation at x hashes the corresponding table row. We create table fingerprints via a public-coin hash based on Reed–Solomon fingerprinting [Lip89] as follows. **V** samples random coefficients $r_1, \ldots, r_k \xleftarrow{\$} \mathbb{F}$ and sends them to **P**. Both parties then locally compute the fingerprint column $\hat{\mathsf{fp}} = (\hat{\mathsf{a}}, \sum_{i=1}^{k} r_i \hat{\mathsf{d}}_i)$. The hashes are collision-resistant as long as the random coefficients are sampled *after* the table oracles have been specified. This mechanism allows us to extend certain protocols that operate over columns to also operate over tables.

5.5 Logarithmic derivative checks

In this section we apply the techniques of LogUp [Hab22] to our arithmetized columns/tables, recovering PIOPs for lookups and multiset equality.

5.6 Lookup

We say a column c_1 *looks up* into another column c_2 of if every element in c_1 is also in c_2, which we denote $\mathsf{c}_1 \sqsubset \mathsf{c}_2$. [Hab22] shows that $\mathsf{c}_1 \sqsubset \mathsf{c}_2$ if and only if there is a column m of the same size as c_2 such that the following rational function equality holds:

$$\sum_{i \in |\mathsf{c}_1|} \frac{1}{X - \mathsf{c}_1[i]} = \sum_{j \in |\mathsf{c}_2|} \frac{m[j]}{X - \mathsf{c}_2[j]} \tag{1}$$

We can easily reinterpret this check in our arithmetized setting. Suppose $\hat{\mathsf{c}}_1$ is defined over a hypercube $\mathscr{H}_\mu$, and $\hat{\mathsf{c}}_2$ and $\hat{m}$ are defined over a hypercube $\mathscr{H}_\nu$. Then, we can use their respective activators to zero-out non-contributing terms and express Eq. (1) as sums over the hypercubes:

$$\sum_{i \in \mathscr{H}_\mu} \frac{\hat{\mathsf{a}}_1(i)}{X - \hat{\mathsf{c}}_1(i)} = \sum_{j \in \mathscr{H}_\nu} \frac{\hat{\mathsf{a}}_2(j) * \hat{m}(j)}{X - \hat{\mathsf{c}}_2(j)} \tag{2}$$

To prove $\hat{\mathsf{c}}_1 \sqsubset \hat{\mathsf{c}}_2$, **P** first computes and sends $\hat{m}$. Then, as in LogUp, we utlilize Schwartz–Zippel to reduce Eq. (2) to equality at a random $r \xleftarrow{\$} \mathbb{F}$ sampled by **V**. Specifically, **P** sends a value s, and then invokes rational sumcheck to prove that both sides of Eq. (2) equal s at r.

When $\hat{\mathsf{c}}_2$ is known a priori to contain no duplicates, $\hat{m}$ is uniquely determined as the *multiplicity* of each element in $\hat{\mathsf{c}}_1$. That is, Eq. (2) ensures that $\forall j \in A(\hat{\mathsf{c}}_2)$, $\hat{m}(j)$ is the number of times $\hat{\mathsf{c}}_2(j)$ appears in $\hat{\mathsf{c}}_1$. In such cases, $\hat{m}$ is useful for higher-level protocols (e.g. [Suppcheck]), so we designate it as a protocol output to be stored by **V**.

PIOP 4: Lookup. Inputs: $[\![\hat{\mathsf{c}}_1]\!], [\![\hat{\mathsf{c}}_2]\!]$

1. **P** sends $\hat{m}$ to **V**, who stores it as protocol output.
2. **V** sends **P** a random point $r \xleftarrow{\$} \mathbb{F}$, who responds with a value s.
3. Invoke RatSumcheck twice to prove that both sides of Eq. (2) equal s at r.

Prescribed lookups and permutations. Sometimes we need to prove not only that $\hat{\mathsf{c}_1} \sqsubset \hat{\mathsf{c}_2}$, but that there is a specific mapping $\hat{\mathsf{sfc}}$ which produces this lookup relation. That is, for each $\hat{\mathsf{c}_1}$-index i, $\hat{\mathsf{c}_1}(i) = \hat{\mathsf{c}_2}(\hat{\mathsf{sfc}}(i))$. We denote this property as $\hat{\mathsf{c}_1} \sqsubset_{\hat{\mathsf{sfc}}} \hat{\mathsf{c}_2}$.

Whenever $\hat{\mathsf{c}_1}$ is in a contiguous layout, such a *prescribed lookup* can be proven as follows: **P** sends the multilinear extension of $\hat{\mathsf{sfc}}$, and then proves $(\hat{\mathsf{sfc}}, \hat{\mathsf{c}_1}) \sqsubset (\hat{\mathcal{I}}, \hat{\mathsf{c}_2})$. This is a by-now standard transformation; see, e.g., [GPS25] for a soundness sketch. The reason why we are restricted to contiguous layout columns is because in sparse layout, the index polynomial no longer matches the row indices.

Multiset equality. LogUp additionally tells us that c_1 and c_2 are equal as multisets if and only if Eq. (1) hold when $m[j] = 1$ for all j [Hab22]. We can prove this straightforwardly in our arithmetized setting by simply running the lookup check with $\hat{m} = 1$.

PIOP 5: MultisetEq. Inputs: $[\![\hat{\mathsf{c}_1}]\!], [\![\hat{\mathsf{c}_2}]\!]$

1. Invoke [Lookup] with $\hat{m} = 1$.

Extension to tables. To extend the multiset equality and lookup to tables, we can simply fingerprint the relevant columns down to a single column and then invoke the foregoing PIOPs.

5.7 Non-negativity and positivity

Various subprotocols require enforcing that a column $\hat{\mathsf{c}}$'s entries lie in the range $\{0, \ldots, 2^\alpha - 1\}$, i.e., that $\hat{\mathsf{c}}$'s entries are non-negative integers of bitwidth $\alpha < \log_2 |\mathbb{F}|$.

A standard approach is to prove that $\hat{\mathsf{c}} \sqsubset \mathcal{I}_\alpha$, as $\mathcal{I}_\alpha$ contains exactly the target range. However, this approach becomes expensive when α increases, as the lookup cost scales with $|\mathcal{I}_\alpha| = 2^\alpha$. To handle this, we use a standard technique which witnesses the 16-bit limb decomposition of the column values (denoted $\hat{\mathsf{d}}_1, \ldots, \hat{\mathsf{d}}_{k+1}$), asserts that combining these recovers the actual column values, and performs a lookup for each limb.

Enforcing positivity. We can add a NonZerocheck to assert that $\hat{\mathsf{c}}$'s entries are in the *strictly positive* range $\{1, \ldots, 2^\alpha - 1\}$.

The full PIOP is described below, and handles the edge case where the last limb has bitwidth smaller than α.

PIOP 6: NonNegcheck/Poscheck. Inputs: $[\![\hat{\mathsf{c}} = (\hat{\mathsf{a}}, \hat{\mathsf{d}})]\!], \alpha$

1. **P** and **V** compute β, k such that $16k + \beta = \alpha$ and $\beta \leq 16$.
2. **P** constructs and sends $[\![\hat{\mathsf{d}}_1]\!], \ldots, [\![\hat{\mathsf{d}}_{k+1}]\!]$ as defined above.
3. Invoke Zerocheck to prove that $\hat{\mathsf{a}} * \hat{\mathsf{d}} = \sum_{i=1}^{k+1} 2^{16(i-1)} * \hat{\mathsf{d}}_i$.
4. For each $i \in [k]$, invoke Lookup to prove $(1, \hat{\mathsf{d}}_i) \sqsubset (1, \mathcal{I}_{16})$.
5. Invoke Lookup to prove $(1, \hat{\mathsf{d}}_{k+1}) \sqsubset (1, \mathcal{I}_\beta)$.
6. (Poscheck only) Invoke NonZerocheck on $\hat{\mathsf{c}}$.

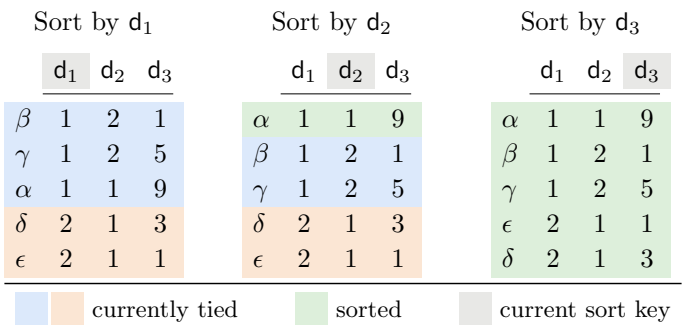


Figure 2: Example execution of lexicographic sort.

5.8 Lexicographic sorting

The *lexicographic sorting check* proves that the rows of a $\mathsf{T} = (\mathsf{d}_1, \ldots, \mathsf{d}_n)$ are in non-decreasing lexicographic order, i.e., for every row index k, $(\mathsf{d}_1[k], \ldots, \mathsf{d}_n[k]) \leq_{\mathsf{lex}} (\mathsf{d}_1[k+1], \ldots, \mathsf{d}_n[k+1])$. The following algorithm sorts the table T lexicographically. First, sort the table according to d_1. Whenever there are ties in d_1, sort each run of ties by d_2. For those rows that are still tied after sorting by d_2, sort by d_3, and so on. An example execution is illustrated in Fig. 2.

We now describe how to use the idea behind this algorithm to check that a T is indeed lexicographically sorted. Let us introduce some notation. We say that a row index j *tied at column* i if the partial rows $(\mathsf{d}_1[j], \ldots, \mathsf{d}_i[j])$ and $(\mathsf{d}_1[j+1], \ldots, \mathsf{d}_i[j+1])$ are equal. The tie-set $\mathcal{T}_i$ is the set of all row indices that are tied at column i. Observe that T is lexicographically sorted if and only if, for every tied row index j in $\mathcal{T}_i$, it holds that $\mathsf{d}_{i+1}[j] \leq \mathsf{d}_{i+1}[j+1]$. In other words, for every row that is tied at column i, the tie is either maintained or correctly broken at column $i+1$. This implies that $\mathcal{T}_i \subseteq \mathcal{T}_{i-1}$.

Our checker. This leads to the following checker given tie-sets $\mathcal{T}_1, \ldots, \mathcal{T}_n$. Set $\mathcal{T}_0$ to contain every row index except the last. For each column $i = 1, \ldots, n$, first perform checks asserting the correct construction of $\mathcal{T}_i$: (i) check that $\mathcal{T}_i \subseteq \mathcal{T}_{i-1}$; (ii) check that column d_i is indeed tied at each location j in $\mathcal{T}_i$ by checking that $\mathsf{d}_i[j] = \mathsf{d}_i[j+1]$; (iii) check that for all locations $j \in \mathcal{T}_{i-1} \setminus \mathcal{T}_i$ (i.e., locations where d_i is supposed to not be tied), $\mathsf{d}_i[j] \neq \mathsf{d}_i[j+1]$. Finally, check that the next column either maintains or breaks the current ties: for all locations $j \in \mathcal{T}_i$, $\mathsf{d}_{i+1}[j] \leq \mathsf{d}_{i+1}[j+1]$.

We describe how to implement these last three checks in a way that will make the PIOP construction easier. To check conditions (ii) and (iii), the checker gets, for each i, as additional hints, a column d'_i that is the cyclic left-rotation by one position of d_i. That is, for each j, $\mathsf{d}'_i[j] = \mathsf{d}_i[j-1 \bmod s]$.

This enables checking condition (ii) by asserting that for each j in $\mathcal{T}_i$, $\mathsf{d}'_i[j] = \mathsf{d}_i[j]$. Similarly, condition (iii) is checked by asserting that for each $j \in \mathcal{T}_{i-1} \setminus \mathcal{T}_i$, $\mathsf{d}'_i[j] - \mathsf{d}_i[j] \neq 0$ Finally, the sortedness check is done by asserting that $j \in \mathcal{T}_i$, $\mathsf{d}'_{i+1}[j] - \mathsf{d}_{i+1}[j] \geq 0$. The final thing the checker does is ensure that each d'_i is indeed the rotation of d_i.

Our PIOP. To arithmetize this checker into a PIOP, we first have to first arithmetize each $\mathcal{T}_i$, which we do by encoding it as the MLE of an indicator vector ties_i, such that $\mathsf{ties}_i[j] = 1$ if and only if $j \in \mathcal{T}_i$. Our PIOP will work only for tables in contiguous layout, so $\mathbf{P}$ begins by sending $s = |\mathsf{T}|$, and $\mathbf{V}$ asserts $\mathsf{T.a} = \mathcal{C}_{\mu,s}$. $\mathbf{V}$ can then set ties_0 to be $\mathcal{C}_{\mu,s-1}$.

Now we can describe the various checks. First, the PIOP ensures that for each $i > 0$, $\hat{\mathsf{ties}}_i$ is

indeed boolean via Boolcheck. Next, to enforce condition (i), the PIOP invokes a Zerocheck on $\hat{\mathsf{ties}}_i * (1 - \hat{\mathsf{ties}}_{i-1})$. Condition (ii) is data-equality between $\hat{\mathsf{d}}_i$ $\hat{\mathsf{d}}'_i$ with activator $\hat{\mathsf{ties}}_i$, so the PIOP proves it via a DataEqCheck. Condition (iii) is enforced via a NonZerocheck on $\mathsf{d}'_i - \mathsf{d}_i$ with activator $\hat{\mathsf{ties}}_{i-1}(1 - \hat{\mathsf{ties}}i)$. Finally, sortedness is proved via a NonNegCheck.

We are left to enforce that each $\hat{\mathsf{d}}'_i$ is the correct cyclic rotation of $\hat{\mathsf{d}}_i$. We do this via a prescribed permutation lookup check, with the permutation $\circlearrowleft(x) = \mathcal{I}(x+1) * (1 - \mathrm{eq}(x, s-1))$ mapping $i \mapsto i + 1 \mod s$.

Strict ordering. To check that $\hat{\mathsf{T}}$ is *strictly sorted*, meaning that no ties remain after all columns have been considered, $\mathbf{V}$ simply asserts that $\hat{\mathsf{ties}}_n = 0$.

PIOP 7: (Strict)Sortcheck. Inputs: $[\![\hat{\mathsf{T}} = (\hat{\mathsf{a}}, [\hat{\mathsf{d}}_1, \ldots, \hat{\mathsf{d}}_n])]\!]$

1. $\mathbf{P}$ sends s to $\mathbf{V}$, who asserts $\hat{\mathsf{a}} = \mathcal{C}_{\mu,s}$.
2. $\mathbf{P}$ sends $\mathbf{V}$ data rotations $[\![\hat{\mathsf{d}'}_1]\!], \ldots, [\![\hat{\mathsf{d}'}_n]\!]$ and tie indicators $[\![\hat{\mathsf{ties}}_1]\!], \ldots, [\![\hat{\mathsf{ties}}_n]\!]$. $\mathbf{V}$ sets $\hat{\mathsf{ties}}_0 := \mathcal{C}_{\mu,s-1}$.
3. For each $i \in [n]$:
(a) Invoke Lookup to prove $\hat{\mathsf{d}}_i \sqsubset_{\circlearrowleft} \hat{\mathsf{d}'}_i$.
(b) Invoke Boolcheck on $\hat{\mathsf{ties}}_i$.
(c) Invoke Zerocheck to prove $\hat{\mathsf{ties}}_{i-1} * \hat{\mathsf{ties}}_i - \hat{\mathsf{ties}}_i = 0$.
(d) Invoke DataEqcheck to prove that $\hat{\mathsf{d}}_i$ and $\hat{\mathsf{d}'}_i$ are equal at $\hat{\mathsf{ties}}_i$.
(e) Invoke NonZerocheck on $(\hat{\mathsf{ties}}_{i-1} * (1 - \hat{\mathsf{ties}}_i),\ \hat{\mathsf{d}'}_i - \hat{\mathsf{d}}_i)$.
(f) Invoke NonNegcheck on $(\hat{\mathsf{ties}}_i, \hat{\mathsf{d}'}_{i+1} - \hat{\mathsf{d}}_{i+1})$.
4. (*StrictSortcheck only*) $\mathbf{V}$ asserts that $\hat{\mathsf{ties}}_n = 0$.

5.9 No Duplicate check

To prove that a table $\hat{\mathsf{T}}$ has no duplicate rows (i.e., the rows of $\hat{\mathsf{T}}$ at positions i and j differ for all $i \neq j$), $\mathbf{P}$ sends a strictly sorted version $\hat{\mathsf{T}}'$ of $\hat{\mathsf{T}}$, and both parties assert that (a) $\hat{\mathsf{T}}'$ and $\hat{\mathsf{T}}$ are equal as multisets of rows (via MultisetEq), and (b) $\hat{\mathsf{T}}'$ is strictly lexicographically sorted (via StrictSortCheck).

Soundness. Strict sortedness forces every row of $\hat{\mathsf{T}}'$ to be distinct, and if the latter is equal as a multiset to $\hat{\mathsf{T}}$, then every row of $\hat{\mathsf{T}}$ must be distinct as well.

PIOP 8: NoDupcheck. Inputs: $[\![\hat{\mathsf{T}}]\!]$

1. $\mathbf{P}$ sends $[\![\hat{\mathsf{T}}']\!]$ to $\mathbf{V}$.
2. Invoke MultisetEq to prove that $\hat{\mathsf{T}}$ and $\hat{\mathsf{T}}'$ are equal as multisets of rows.
3. Invoke StrictSortcheck to prove that $\hat{\mathsf{T}}'$ is strictly lexicographically sorted.

Primary keys. If any column of $\hat{\mathsf{T}}$ is a primary key, then by definition $\hat{\mathsf{T}}$ has no duplicate rows, so we can skip the foregoing protocol.

NoDupcheck Without Sorting. Certain tables may contain large values or arbitrary field elements that are untenable to sort. To address this, we also present an alternative Bézout's NoDupcheck protocol in Section B.1 that does not require sorting.

5.10 Support check

To prove that table $\hat{\mathsf{R}} = (\hat{\mathsf{a}}_{\mathsf{R}}, [\hat{\mathsf{d}}_{\mathsf{R},1}, \ldots, \hat{\mathsf{d}}_{\mathsf{R},n}])$ is the support of another table $\hat{\mathsf{L}}$, (i.e., $\hat{\mathsf{R}}$ contains exactly the distinct rows of $\hat{\mathsf{L}}$), our PIOP enforces three conditions. First, it asserts that $\hat{\mathsf{R}}$'s rows

are unique via NoDupCheck. Next, it asserts that every row in $\hat{\mathsf{L}}$ indeed appears in $\hat{\mathsf{R}}$ via a lookup $\hat{\mathsf{L}} \sqsubset \hat{\mathsf{R}}$. Finally, it needs to assert that $\hat{\mathsf{R}}$ does not contain extraneous rows that do not appear in $\hat{\mathsf{L}}$. We could do this via another lookup $\hat{\mathsf{R}} \sqsubset \hat{\mathsf{L}}$, but our protocol uses a clever trick to reduce costs. Specifically, since we know that $\hat{\mathsf{R}}$ has no duplicate rows, the multiplicity oracle $\hat{m}$ in the Lookup for $\hat{\mathsf{L}} \sqsubset \hat{\mathsf{R}}$ actually records multiplicities, i.e., the number of times the row of $\hat{\mathsf{R}}$ at position i appears in $\hat{\mathsf{L}}$. All we have to do is assert that there are no entries with multiplicity 0, which our PIOP does via a NonZerocheck on $(\hat{\mathsf{a}}_{\mathsf{R}}, \hat{m})$.

Looking ahead, Aggregate uses this PIOP as a sub-protocol and requires access to $\hat{m}$, so we expose it as a protocol output.

PIOP 9: Suppcheck. Inputs: $[\![\hat{\mathsf{L}}]\!], [\![\hat{\mathsf{R}}]\!]$

1. Invoke NoDupcheck on $\hat{\mathsf{R}}$.
2. Invoke Lookup to prove that $\hat{\mathsf{L}} \sqsubset \hat{\mathsf{R}}$. $\mathbf{V}$ receives and stores $[\![\hat{m}]\!]$ as output.
3. Invoke NonZerocheck on $(\hat{\mathsf{a}}_{\mathsf{R}}, \hat{m})$.

6 PIOPs for query plan nodes

6.1 Expressions

An expression e, on input columns l and r, outputs a column $e(\mathsf{l}, \mathsf{r})$ whose entry at row r is e applied to $\mathsf{l}[r]$ and $\mathsf{r}[r]$. TruthTable supports three categories of expressions: arithmetic ($+$, $\times$), comparison ($=, \neq, >, <, \geq, \leq$), and logical ($\wedge$, $\vee$, $\neg$). In the following, let $\hat{\mathsf{d}}_{\mathsf{L}}$ denote l's data oracle, and $\hat{\mathsf{d}}_{\mathsf{R}}$ denote r's data oracle. Additionally, our PIOP assumes that $\hat{\mathsf{d}}_{\mathsf{L}}$ and $\hat{\mathsf{d}}_{\mathsf{R}}$ have the same activator $\hat{\mathsf{a}}$; this is a mild assumption as generally l and r are from the same table.

Algebraic expressions. $\mathbf{V}$ can obtain oracle access to the sum of l and r as a "virtual" column $(\hat{\mathsf{a}}, \hat{\mathsf{d}}_{\mathsf{L}} + \hat{\mathsf{d}}_{\mathsf{R}})$. To query the data polynomial of this oracle, it queries $\hat{\mathsf{d}}_{\mathsf{L}}$ and $\hat{\mathsf{d}}_{\mathsf{R}}$, and sums the responses. This works because the virtual oracle $\hat{\mathsf{l}} + \hat{\mathsf{r}}$ agrees with the oracle corresponding to the sum at all points in the hypercube. The same holds for multiplication, except the data column becomes $\hat{\mathsf{d}}_{\mathsf{L}} * \hat{\mathsf{d}}_{\mathsf{R}}$.[3]

Logical expressions. Similarly, when l and r are boolean, $\mathbf{V}$ can obtain oracle access to the result of applying logical operations to these columns via a virtual column whose data polynomial is constructed as follows: `NOT`: $1 - \hat{\mathsf{d}}_{\mathsf{L}}$, `AND`: $\hat{\mathsf{d}}_{\mathsf{L}} * \hat{\mathsf{d}}_{\mathsf{R}}$; `OR`: $1 - (1 - \hat{\mathsf{d}}_{\mathsf{L}}) * (1 - \hat{\mathsf{d}}_{\mathsf{R}})$. Some other SQL operators such as `BETWEEN` (`AND` of two comparisons) and `IN` (`OR` of multiple equalities) can be implemented using the foregoing.

Comparison expressions. Recall that a comparison expression with operator $\diamond \in \{=, \neq, >, <, \geq, \leq\}$, on input l and r, produces a boolean output column b whose r-th row equals $\mathsf{l}[r] \diamond \mathsf{r}[r]$. This is captured via the language $\mathcal{L}_\diamond = \{(\mathtt{I}, \mathsf{b}) \mid \forall r, \mathsf{b}[r] = \mathsf{l}[r] \diamond \mathsf{r}[r]\}$.

Unlike the other expressions, comparisons require a non-trivial PIOP that works as follows. First, it enforces basic correctness checks, namely that $\hat{\mathsf{b}}$ is boolean (via Boolcheck) and is not active outside of $\hat{\mathsf{a}}$ (via Zerocheck on $(1 - \hat{\mathsf{a}}) * \hat{\mathsf{b}}$). Next, it enforces that $\hat{\mathsf{b}}$ is correctly constructed by

[3] This technique does *not* handle overflows where the result exceeds the designated bitwidth or the field $\mathbb{F}$. Overflows can be checked via range checks, but this incurs significant overhead, so following prior work [GFN25; BBCRT25] we leave to future work techniques for amortizing such range checks to minimize costs, e.g. by tracking the maximum bitwidth of a column.

enforcing that there are neither false negatives (i.e., active indices r where $\hat{\mathsf{b}}(r) = 0$ but $\hat{\mathsf{l}}(r) \diamond \hat{\mathsf{r}}(r) = 1$) nor false positives (i.e., indices r where $\hat{\mathsf{b}}(r) = 1$ but $\hat{\mathsf{l}}(r) \diamond \hat{\mathsf{r}}(r) = 0$). We illustrate how to do this for two operators, as checks for the rest can be obtained from these.

Equality (=): Our PIOP prevents false positive via DataEqcheck to enforce data equality between $\hat{\mathsf{l}}$ and $\hat{\mathsf{r}}$ with activator $\hat{\mathsf{b}}$. It prevents false negatives by asserting that the active locations not selected by $\hat{\mathsf{b}}$ are not equal, i.e. their difference is not zero. It does this via NonZerocheck on $(\hat{\mathsf{a}} - \hat{\mathsf{b}}, \hat{\mathsf{d}}_{\mathsf{L}} - \hat{\mathsf{d}}_{\mathsf{R}})$.

Greater than (>): To deal with false positives, we run Poscheck on the column $(\hat{\mathsf{b}}, \hat{\mathsf{d}}_{\mathsf{L}} - \hat{\mathsf{d}}_{\mathsf{R}})$ to assert that locations selected by $\hat{\mathsf{b}}$ satisfy $\hat{\mathsf{d}}_{\mathsf{L}} > \hat{\mathsf{d}}_{\mathsf{R}}$. For false negatives, we run NonNegcheck on $(\hat{\mathsf{a}} - \hat{\mathsf{b}}, \hat{\mathsf{d}}_{\mathsf{R}} - \hat{\mathsf{d}}_{\mathsf{L}})$ to assert that $\hat{\mathsf{d}}_{\mathsf{L}} \leq \hat{\mathsf{d}}_{\mathsf{R}}$ at every active index not selected by $\hat{\mathsf{b}}$.

PIOP 10: ComparisonExpr. Inputs: $[\![\hat{\mathsf{l}}]\!], [\![\hat{\mathsf{r}}]\!], [\![\hat{\mathsf{b}}]\!], \diamond$

1. Invoke Boolcheck on $\hat{\mathsf{b}}$.
2. Invoke Zerocheck on $(1 - \hat{\mathsf{a}}) * \hat{\mathsf{b}}$.
3. If $\diamond$ is $=$:
(a) Invoke DataEqcheck to prove that $\hat{\mathsf{d}}_{\mathsf{L}}$ and $\hat{\mathsf{d}}_{\mathsf{R}}$ are equal at $\hat{\mathsf{b}}$.
(b) Invoke NonZerocheck to prove $(\hat{\mathsf{a}} - \hat{\mathsf{b}}, \hat{\mathsf{d}}_{\mathsf{L}} - \hat{\mathsf{d}}_{\mathsf{R}})$ contains no zero elements.
4. If $\diamond$ is $\neq$, run Step 3, but using $\hat{\mathsf{b}}' = \hat{\mathsf{a}} * (1 - \hat{\mathsf{b}})$.
5. If $\diamond$ is $>$:
(a) Invoke Poscheck to prove that $(\hat{\mathsf{b}}, \hat{\mathsf{d}}_{\mathsf{L}} - \hat{\mathsf{d}}_{\mathsf{R}})$ contains only positive elements.
(b) Invoke NonNegcheck to prove that $(\hat{\mathsf{a}} - \hat{\mathsf{b}}, \hat{\mathsf{d}}_{\mathsf{R}} - \hat{\mathsf{d}}_{\mathsf{L}})$ contains no negative elements.
6. If $\diamond$ is $\leq$: run Step 5 but using $\hat{\mathsf{b}}' = \hat{\mathsf{a}} * (1 - \hat{\mathsf{b}})$.
7. If $\diamond$ is $<$: run Step 5, but with $\hat{\mathsf{d}}_{\mathsf{L}}$ and $\hat{\mathsf{d}}_{\mathsf{R}}$ switched.
8. If $\diamond$ is $\geq$: run Step 7, but using $\hat{\mathsf{b}}' = \hat{\mathsf{a}} * (1 - \hat{\mathsf{b}})$.

6.2 Filter

On input table $\mathtt{I}$ and a boolean expression f, `filter` produces an output table $\mathtt{O}$ comprising those of $\mathtt{I}$'s rows on which f evaluates to 1. This is captured via the language $\mathcal{L}_\sigma = \{ (\mathtt{I}, \mathtt{O}, f) \mid \{\!\{\mathtt{O}\}\!\} = \{\!\{\mathtt{I}[r] \mid f(\mathtt{I}[r]) = 1\}\!\}\}$, which says that $\mathtt{O}$ must be equal as a multiset to $\mathtt{I}$ restricted to rows selected by f.

In proving membership in the arithmetized language, our PIOP minimizes the number of new oracles sent by relying on the observation that filtering only needs to update the active rows in $\hat{\mathsf{O}}$. That is, we only need to produce a new activator and prove that it is correct, instead of producing new data columns.

This is achieved as follows. $\mathbf{P}$ sends the new activator $\hat{\mathsf{b}}$ whose i-th entry is the result of evaluating f on $\hat{\mathsf{I}}(i)$. Then, $\mathbf{P}$ and $\mathbf{V}$:

1. Prove that $\hat{\mathsf{b}}$ is computed correctly, i.e. for each $i \in \mathscr{H}_\mu$, $\hat{\mathsf{b}}(i) = f(\hat{\mathsf{I}}(i))$ if $i \in A(\hat{\mathsf{T}})$ and 0 otherwise. This is accomplished with the appropriate expression protocols from Section 6.1.
2. Assert that the data columns of $\hat{\mathsf{O}}$ are unchanged from $\hat{\mathsf{I}}$.
3. Assert that the activator of $\hat{\mathsf{O}}$ is $\hat{\mathsf{b}}$.

Soundness: Since $\hat{\mathsf{O}}$'s activator is known to be boolean on $A(\hat{\mathsf{O}})$ and 0 elsewhere, $\hat{\mathsf{O}}$'s active rows are at most a subset of those of $\hat{\mathsf{I}}$. Moreover, since the PIOP for f proves that $\hat{\mathsf{b}}$ is correctly computed, $\mathbf{P}$ can neither omit selected rows nor include unselected ones.

PIOP 11: Filter. Inputs: $[\![\hat{\mathtt{I}}]\!], [\![\hat{\mathtt{O}}]\!], f$

1. **P** sends the oracle $[\![\hat{\mathtt{b}}]\!]$ computed as above.
2. Invoke PIOP for f from §6.1 to prove $\hat{\mathtt{b}}$'s correctness.
3. **V** asserts that $\hat{\mathtt{O}}.\mathtt{d}_i = \hat{\mathtt{I}}.\mathtt{d}_i$ for every data column i, and that $\hat{\mathtt{O}}.\mathtt{a} = \hat{\mathtt{b}}$.

6.3 Project

On input table `I` and a list of expressions $e_1, \ldots, e_m$, `project` produces an output table `O` whose i-th column is $e_i(\mathtt{I})$. This is captured by the language $\mathcal{L}_\pi = \{(\mathtt{I}, \mathtt{O}, e_1, \ldots, e_m) \mid \mathtt{O}.\mathtt{d}_i[r] = e_i(\mathtt{I}[r])\}$.

Our PIOP for this is straightforward: it simply invokes the appropriate PIOPs from Section 6.1 to show that each column is correctly computed. Note that sometimes the expression is the identity; in this case an equality assertion will suffice. **V** also asserts that $\hat{\mathtt{O}}$ and $\hat{\mathtt{I}}$ share the same activator.

PIOP 12: Project. Inputs: $[\![\hat{\mathtt{I}}]\!], [\![\hat{\mathtt{O}}]\!], (e_1, \ldots, e_m)$

1. **V** asserts $\hat{\mathtt{O}}.\mathtt{a} = \hat{\mathtt{I}}.\mathtt{a}$.
2. For each $i \in [m]$, invoke e_i's PIOP to prove $\hat{\mathtt{O}}.\mathtt{d}_i = e_i(\hat{\mathtt{I}})$.

6.4 Compaction

After a highly selective filter, the activator could be highly sparse. To avoid subsequent PIOPs incurring a cost for processing these inactive rows, TruthTable introduces a new node called *compaction* that asserts that an output table `O` is equal as a multiset to an input table `I`. While semantically this is just enforcing identity between the tables, it helps with efficiency because we enforce that $\hat{\mathtt{O}}$ is in contiguous layout over a potentially much smaller hypercube.

PIOP 13: Compact. Inputs: $[\![\hat{\mathtt{I}}]\!], [\![\hat{\mathtt{O}}]\!]$

1. **P** sends s to **V**, who asserts that $\hat{\mathtt{O}}.\mathtt{a} = \mathcal{C}_{\mu,s}$.
2. Invoke MultisetEq on $\hat{\mathtt{I}}$ and $\hat{\mathtt{O}}$.

6.5 Order-By

`ORDER BY` asserts that an output table `O` is the sorted version of an input table `I`, according to some number $m \leq n$ of *sort* columns. This is captured by the language $\mathcal{L}_\tau = \{(\mathtt{I}, \mathtt{O}, m) | \{\!\{\mathtt{O}\}\!\} = \{\!\{\mathtt{I}\}\!\}$ and $\kappa_m(\mathtt{O})$ is sorted$\}$, where $\kappa_m(\mathtt{T})$ is the restriction of `T` to the first m data columns.

Our PIOP for this language just invokes MultisetEq on $\hat{\mathtt{I}}$ and $\hat{\mathtt{O}}$, and invokes Sortcheck on $(\hat{\mathtt{O}}.\mathtt{a}, [\hat{\mathtt{O}}.\mathtt{d}_1, \ldots, \hat{\mathtt{O}}.\mathtt{d}_m])$. Note that because $\hat{\mathtt{O}}$ is a fresh table, it is in a contiguous layout (which we assert), and so it is sound to invoke Sortcheck.

PIOP 14: Order By. Inputs: $[\![\hat{\mathtt{I}}]\!], [\![\hat{\mathtt{O}}]\!], m$

1. **P** sends s to **V**, who asserts that $\hat{\mathtt{O}}.\mathtt{a} = \mathcal{C}_{\mu,s}$.
2. Invoke MultisetEq on $\hat{\mathtt{I}}$ and $\hat{\mathtt{O}}$.
3. Invoke Sortcheck to prove that $(\hat{\mathtt{O}}.\mathtt{a}, [\hat{\mathtt{O}}.\mathtt{d}_1, \ldots, \hat{\mathtt{O}}.\mathtt{d}_m])$ is lexicographically sorted.

Sorting by expressions. Some queries sort by *expressions* evaluated over columns. Our PIOP can be extended to support this by creating new columns that contain the evaluations of the expressions, proving their correctness via PIOPs from Section 6.1, and then modifying the Order By PIOP to sort $\hat{\mathsf{O}}$ by these columns.

6.6 Limit

`LIMIT` i, on input a table $\mathtt{I}$ and integer $i \leq |\mathtt{I}|$, produces an output table $\mathtt{O}$ over the same columns whose rows are the first i rows of $\mathtt{I}$. This is captured via the language $\mathcal{L}_{\mathsf{limit}} = \{(\mathtt{I}, \mathtt{O}, i) || \mathtt{O}| = i \text{ and } \forall r \in [i] : \mathtt{O}[r] = \mathtt{I}[r]\}$. In proving this, our PIOP again minimizes costs by only sending a new activator $\hat{\mathsf{O}}.\mathsf{a}$ that selects the first i active rows of $\hat{\mathsf{I}}$. As long as this activator is correctly constructed, $\mathbf{V}$ only needs to assert that the data columns of $\hat{\mathsf{I}}$ and $\hat{\mathsf{O}}$ are equal as oracles. To prove correctness of the activator, our PIOP proceeds as follows.

$\mathbf{P}$ sends an integer j such that the first i active rows of $\hat{\mathsf{I}}$ live in the first j hypercube indices. If this is indeed the case, then $\hat{\mathsf{O}}.\mathsf{a}$ is $\hat{\mathsf{I}}.\mathsf{a}$ `AND` $\mathcal{C}_{\mu,j}$ (i.e., it is the virtual oracle $\hat{\mathsf{I}}.\mathsf{a} * \mathcal{C}_{\mu,j}$), which is asserted by $\mathbf{V}$. Then to prove that $\hat{\mathsf{O}}.\mathsf{a}$ indeed activates only i active locations; the PIOP invokes sumcheck proving $\sum_{x \in \mathscr{H}_\mu} \hat{\mathsf{O}}.\mathsf{a}(x) = i$.

Soundness: Asserting $\hat{\mathsf{O}}.\mathsf{a} = \hat{\mathsf{I}}.\mathsf{a} * \mathcal{C}_{\mu,j}$ forces $\hat{\mathsf{O}}.\mathsf{a}$ to activate rows of $\hat{\mathsf{I}}$ in order starting from the first active row and without skipping. By construction, $\hat{\mathsf{O}}.\mathsf{a}$ is boolean, so the resulting sumcheck proves that there are exactly i active locations. Together these properties imply $\hat{\mathsf{O}}.\mathsf{a}$ activates the first i active locations of $\hat{\mathsf{I}}$.

PIOP 15: Limit. Inputs: $[\![\hat{\mathsf{I}}]\!], [\![\hat{\mathsf{O}}]\!], i$

1. $\mathbf{V}$ asserts that $\hat{\mathsf{O}}.\mathsf{d}_i = \hat{\mathsf{I}}.\mathsf{d}_i$ for every data column i.
2. $\mathbf{P}$ sends j to $\mathbf{V}$, who asserts that $\hat{\mathsf{O}}.\mathsf{a} = \hat{\mathsf{I}}.\mathsf{a} * \mathcal{C}_{\mu,j}$.
3. Invoke Sumcheck to prove that $\sum_{x \in \mathscr{H}_\mu} \hat{\mathsf{O}}.\mathsf{a}(x) = i$.

6.7 Distinct

`DISTINCT`, on input a table $\mathtt{I}$, produces an output table $\mathtt{O}$ containing exactly the distinct rows of $\mathtt{I}$. This is captured via the language $\mathcal{L}_\delta = \{(\mathtt{I}, \mathtt{O}) \mid \forall, i \neq i', \mathtt{O}[i] \neq \mathtt{O}[i'] \text{ and } \{\mathtt{O}\} = \{\mathtt{I}\}\}$ which asserts that no two rows of $\mathtt{O}$ are equal, and that $\mathtt{O}$ and $\mathtt{I}$ are equal as sets.

To prove membership in this PIOP, we again have $\mathbf{P}$ send only a new activator and preserve the other oracles. Specifically, $\mathbf{P}$ sends $\hat{\mathsf{O}}.\mathsf{a}$ that is 1 at every distinct row in $A(\hat{\mathsf{I}})$. The PIOP enforces that this is boolean. Finally, it asserts that $\hat{\mathsf{O}}$ is the support of $\hat{\mathsf{I}}$ (i.e. contains the distinct columns of $\hat{\mathsf{I}}$) via a Suppcheck. The resulting multiplicity $\hat{m}$ is stored as a protocol output.

PIOP 16: Distinct. Inputs: $[\![\hat{\mathsf{I}}]\!], [\![\hat{\mathsf{O}}]\!]$

1. $\mathbf{V}$ asserts that $\hat{\mathsf{O}}.\mathsf{d}_i = \hat{\mathsf{I}}.\mathsf{d}_i$ for every data column i.
2. Invoke Boolcheck on $\hat{\mathsf{O}}.\mathsf{a}$.
3. Invoke Suppcheck to prove that $\hat{\mathsf{O}}$ contains exactly the distinct rows of $\hat{\mathsf{I}}$, $\mathbf{V}$ receives and stores $[\![\hat{m}]\!]$ as output.

6.8 Aggregate

Let $\mathtt{I}$ and $\mathtt{O}$ each have a key column k and a data column d. For each distinct key k in the input table, `Aggregate` produces a new row i in $\mathtt{O}$, where $\mathtt{O}.\mathsf{k}[i] = k$ and $\mathtt{O}.\mathsf{d}[i]$ is the result of applying an aggregate function $\mathsf{agg} \in \{\mathsf{count}, \mathsf{sum}, \mathsf{max}, \mathsf{min}, \mathsf{mean}\}$ to all data values in $\mathtt{I}$ with key k.

Let the "key group" W_k be the set of indices j such that $\mathtt{I}.\mathsf{k}[j] = k$. Aggregate is then captured by the language: $\mathcal{L}_\gamma = \{(\mathtt{I}, \mathtt{O}) \mid (\mathtt{I}.\mathsf{k}, \mathtt{O}.\mathsf{k}) \in \mathcal{L}_\delta \text{ and } \mathtt{O}.\mathsf{d}[i] = \mathsf{agg}(\mathtt{I}.\mathsf{d}[W_k])\}$

Since aggregation maintains one representative row per key, we use an input-aligned layout: set $\hat{\mathtt{O}}.\mathsf{k} = \hat{\mathtt{I}}.\mathsf{k}$ and let $\hat{\mathtt{O}}.\mathsf{a}$ select one representative per key. This formats the key columns as inputs to Distinct certify that the output keys are exactly the distinct input keys. It remains to certify the aggregate value at each selected representative.

Count aggregation. For count, the multiplicity oracle returned by Distinct gives the count for each key, so $\mathbf{V}$ asserts $[\![\hat{\mathtt{O}}.\mathsf{d}]\!] = [\![\hat{m}]\!]$.

Sum aggregation. For sum, we designate a helper PIOP KeyedSumcheck which proves that for each key k, the sum of the data values in $\hat{\mathtt{I}}$ with key k equals the sum of the data values in $\hat{\mathtt{O}}$ with key k. As we have already shown there is exactly one active output row per key, this is sufficient to prove the aggregated sums.

on $\hat{\mathtt{I}}$ and $\hat{\mathtt{O}}$. Since there is one active output row per key, equality of keyed sums forces $\hat{\mathtt{O}}.\mathsf{d}$ to contain the aggregated sums of $\hat{\mathtt{I}}$.

Maximum aggregation. For max, $\mathbf{P}$ formats $\hat{\mathtt{O}}.\mathsf{d}$ as a "broadcast" column. Let M_k be the maximum value of $\hat{\mathtt{I}}.\mathsf{d}$ in accross key group W_k. At every $x \in A(\hat{\mathtt{I}})$, $\hat{\mathtt{O}}.\mathsf{d}$ should equal $M_{\hat{\mathtt{I}}.\mathsf{k}(x)}$. Additionally, $\hat{\mathtt{O}}.\mathsf{a}$ should select representative rows that actually attain M_k. With this formatting, the PIOP proves the following:

1. $\hat{\mathtt{O}}.\mathsf{d}$ is uniform accross each input key group W_k. This is equivalent to $\hat{\mathtt{O}}$ being the support of $(\hat{\mathtt{I}}.\mathsf{a}, [\hat{\mathtt{I}}.\mathsf{k}, \hat{\mathtt{O}}.\mathsf{d}])$, which is proven via Suppcheck.
2. $\hat{\mathtt{O}}.\mathsf{d}$ upper-bounds $\hat{\mathtt{I}}.\mathsf{d}$ on all active input rows, which is proven via NonNegcheck on their difference.
3. $\hat{\mathtt{O}}.\mathsf{d} = \hat{\mathtt{I}}.\mathsf{d}$ at the selected representative rows, which is proven via DataEqcheck.

Soundness. If $\hat{\mathtt{O}}.\mathsf{d}$ is uniform across W_k, upperbounds $\hat{\mathtt{I}}.\mathsf{d}$ accross W_k, and is equal to $\hat{\mathtt{I}}.\mathsf{d}$ at the representative row of k, then it MUST be broadcasting M_k to all points in W_k, including the representative row of k. Our checks enforce these conditions for all keys simultaneously, proving that $\hat{\mathtt{O}}.\mathsf{d}$ contains the aggregated maximums of $\hat{\mathtt{I}}$.

PIOP 17: Aggregate. Inputs: $[\![\hat{\mathtt{I}}]\!], [\![\hat{\mathtt{O}}]\!], \mathsf{agg}$

1. Invoke Distinct on $(\hat{\mathtt{I}}.\mathsf{a}, \hat{\mathtt{I}}.\mathsf{k})$ and $(\hat{\mathtt{O}}.\mathsf{a}, \hat{\mathtt{O}}.\mathsf{k})$. $\mathbf{V}$ receives $[\![\hat{m}]\!]$ as output.
2. If $\mathsf{agg} = \mathsf{count}$, $\mathbf{V}$ sets $[\![\hat{\mathtt{O}}.\mathsf{d}]\!] := [\![\hat{m}]\!]$.
3. If $\mathsf{agg} = \mathsf{sum}$, invoke KeyedSumcheck on $\hat{\mathtt{I}}$ and $\hat{\mathtt{O}}$ with respect to $\hat{\mathtt{I}}.\mathsf{k}$ and $\hat{\mathtt{O}}.\mathsf{k}$.
4. If $\mathsf{agg} = \mathsf{max}$:
   (a) Invoke Suppcheck to prove that $\hat{\mathtt{O}}$ is the support of $(\hat{\mathtt{I}}.\mathsf{a}, [\hat{\mathtt{I}}.\mathsf{k}, \hat{\mathtt{O}}.\mathsf{d}])$.
   (b) Invoke NonNegcheck to prove that $(\hat{\mathtt{I}}.\mathsf{a}, \hat{\mathtt{O}}.\mathsf{d} - \hat{\mathtt{I}}.\mathsf{d})$ is non-negative.
   (c) Invoke DataEqcheck to prove that $\hat{\mathtt{O}}.\mathsf{d}$ and $\hat{\mathtt{I}}.\mathsf{d}$ are equal at $\hat{\mathtt{O}}.\mathsf{a}$.

Aggregate min and mean. Min and mean require no new primitives. For min, run the max protocol but replace its NonNegcheck with one proving that $(\hat{\mathtt{I}}.\mathsf{a}, \hat{\mathtt{I}}.\mathsf{d} - \hat{\mathtt{O}}.\mathsf{d})$ is non-negative. For mean, run the sum and count protocols separately; $\mathbf{V}$ computes the mean from the two outputs using the desired rounding rule.

Multiple key and data columns. For multiple key columns, fingerprint them into one composite key column per table and run the same protocol. For multiple data columns, run the chosen `agg` step independently on each.

6.9 Inner join

An `Inner Join` takes two input tables L and R and produces an output table O containing the cross product of the two tables, filtered to retain only rows with matching keys. We first describe the protocol in the setting where each input has one key column and one data column: L and R each expose a key column k and a data column d, and O exposes a single key column $\mathsf{O.k}$, along with left and right data columns $\mathsf{O.d_L}$, and $\mathsf{O.d_R}$. Each output row copies the common key into $\mathsf{O.k}$, the corresponding data values into $\mathsf{O.d_L}$ and $\mathsf{O.d_R}$. This is captured by the language: $\mathcal{L}_{\bowtie} = \{(\mathsf{L}, \mathsf{R}, \mathsf{O}) \mid \{\!\{\mathsf{O}\}\!\} = \{\!\{(k, \mathsf{L.d}[i], \mathsf{R.d}[j]) | \mathsf{L.k}[i] = \mathsf{R.k}[j] = k\}\!\}\}$

Reasoning about the cross product directly is wasteful, since it can be far larger than O. We avoid it by checking the following conditions instead:

1. *Row count.* The number of rows in O equals the number of matching key-pairs across L and R.
2. *Row correctness.* For $\mathsf{T} \in \{\mathsf{L}, \mathsf{R}\}$: the partial rows $(\mathsf{O.k}[i], \mathsf{O.d_T}[i])$ are contained as full rows in the corresponding table T.
3. *Row uniqueness.* Every row in O is sourced from a unique pair of rows in L and R.

Soundness. Row count certifies that O has the expected number of rows. As O only has one key column, row correctness implies that every row in O is a member of the filtered cross product. So, the only way for O to be incorrect is if it contains duplicates of correct rows, which is caught by row uniqueness.

The protocol begins with $\mathbf{P}$ sending the row count $s = |\mathsf{O}|$ to $\mathbf{V}$, who confirms that $\hat{\mathsf{O}}.\mathsf{a} = \mathcal{C}_{\mu,s}$. Each condition is then straightforward to prove. We designate a helper PIOP MatchPaircheck to prove there are exactly s matching pairs between the key columns of L and R. For row correctness, $\mathbf{P}$ sends witness columns $\mathsf{src_L}$ and $\mathsf{src_R}$, where $\mathsf{src_T}(i)$ maps output row i to its source row in T. Then, the condition reduces to $(\mathsf{O.a}, [\mathsf{O.k_T}, \mathsf{O.d_T}]) \sqsubset_{\mathsf{src_T}} \mathsf{T}$, which is proven with a prescribed lookup. Opting for the prescribed lookup proves the correctness of the $\mathsf{src_T}$ maps, so row uniqueness reduces to checking that there are no duplicate pairs in $(\mathsf{src_L}, \mathsf{src_R})$.

PIOP 18: Inner Join. Inputs: $[\![\hat{\mathsf{L}}]\!], [\![\hat{\mathsf{R}}]\!], [\![\hat{\mathsf{O}}]\!]$

1. $\mathbf{P}$ sends s to $\mathbf{V}$, who asserts that $\hat{\mathsf{O}}.\mathsf{a} = \mathcal{C}_{\mu,s}$.
2. Invoke MatchPaircheck to prove that there are exactly s matching pairs between $(\hat{\mathsf{L}}.\mathsf{a}, \hat{\mathsf{L}}.\mathsf{k})$ and $(\hat{\mathsf{R}}.\mathsf{a}, \hat{\mathsf{R}}.\mathsf{k})$.
3. $\mathbf{P}$ sends $\mathbf{V}$ $[\![\widehat{\mathsf{src}}_{\mathsf{L}}]\!]$ and $[\![\widehat{\mathsf{src}}_{\mathsf{R}}]\!]$.
4. For $\mathsf{T} \in \{\mathsf{L}, \mathsf{R}\}$, invoke Lookup to prove that
$$(\hat{\mathsf{O}}.\mathsf{a}, [\hat{\mathsf{O}}.\mathsf{k}, \hat{\mathsf{O}}.\mathsf{d_T}]) \sqsubset_{\mathsf{src_T}} (\hat{\mathsf{T}}.\mathsf{a}, [\hat{\mathsf{T}}.\mathsf{k}, \hat{\mathsf{T}}.\mathsf{d})])$$
5. Invoke NoDupcheck on the table $(\hat{\mathsf{O}}.\mathsf{a}, [\widehat{\mathsf{src}}_{\mathsf{L}}, \widehat{\mathsf{src}}_{\mathsf{R}}])$.

Multiple key and data columns. For multiple data columns, we include all the data columns of L and R in the row-correctness lookup checks in step 3. For multiple key columns, we fingerprint them into a single composite key column on each side and run the standard protocol.

Primary-key to foreign-key. A common special case of inner join is a *primary-key to foreign-key join*. In this setting, two constraints on the inputs come from the database schema itself: $\mathsf{L.k}$ has no duplicate entries, and every entry of $\mathsf{R.k}$ appears in $\mathsf{L.k}$.[4] [5]

Under this schema, every row of $\hat{\mathsf{R}}$ matches exactly one row of $\hat{\mathsf{L}}$, so $\hat{\mathsf{O}}$ has exactly one row per row of $\hat{\mathsf{R}}$. We exploit this by setting $\hat{\mathsf{O}}.\mathsf{a}, \hat{\mathsf{O}}.\mathsf{k}$, and $\hat{\mathsf{O}}.\mathsf{d_R}$ to be the corresponding columns from $\hat{\mathsf{R}}$. Then, **P** need only provide the joined left data $\hat{\mathsf{O}}.\mathsf{d_L}$, and prove that each pair $(\hat{\mathsf{O}}.\mathsf{k}, \hat{\mathsf{O}}.\mathsf{d_L})$ is a row of $\hat{\mathsf{L}}$. This can be done with a single lookup check.

PIOP 19: PKFKJoin. Inputs: $[\![\hat{\mathsf{L}}]\!], [\![\hat{\mathsf{R}}]\!], [\![\hat{\mathsf{O}}]\!]$

1. **V** asserts $\hat{\mathsf{O}}.\mathsf{a} = \hat{\mathsf{R}}.\mathsf{a}$, $\hat{\mathsf{O}}.\mathsf{k} = \hat{\mathsf{R}}.\mathsf{k}$, and $\hat{\mathsf{O}}.\mathsf{d_R} = \hat{\mathsf{R}}.\mathsf{d}$.
2. Invoke Lookup to prove $(\hat{\mathsf{O}}.\mathsf{a}, [\hat{\mathsf{O}}.\mathsf{k}, \hat{\mathsf{O}}.\mathsf{d_L}]) \sqsubset \hat{\mathsf{L}}$.

6.10 Opening the result commitment

The query plan node PIOPs establish that the root oracle $[\![\hat{\mathsf{T}}]\!]$ carries the correct query result, but the reported result res is given as a *plaintext array*. This PIOP proves that the two are consistent. **V** begins by locally converting res into the contiguous arithmetized table $\hat{\mathsf{R}}$. If $\hat{\mathsf{T}}$ is also contiguous, then result check is straightforward: we simply prove that the active rows of $\hat{\mathsf{T}}$ match the corresponding rows in res via fingerprinting. Else, we prove that the two tables are equal as multisets.

Soundness. SQL typically regards two tables as equal if they contain the same rows, regardless of their order, so multiset equality is a sound notion of equality. The exception is when an `ORDER BY` clause is present. In this case, the OrderBy protocol already asserts contiguity, so the order-preserving first approach will be used.

PIOP 20: ResultCheck. Inputs: $[\![\hat{\mathsf{T}}]\!], \mathsf{res}$

1. **V** converts res into the contiguous arithmetized table $\hat{\mathsf{R}}$.
2. If $\mathsf{T.a} = \mathcal{C}_{\mu,|\mathsf{res}|}$: Fingerprint $\hat{\mathsf{T}}$ and $\hat{\mathsf{R}}$ into columns $\hat{\mathsf{fp}}_\mathsf{T}, \hat{\mathsf{fp}}_\mathsf{R}$ and invoke DataEqcheck to prove that $\hat{\mathsf{fp}}_\mathsf{T} = \hat{\mathsf{fp}}_\mathsf{R}$ at $\hat{\mathsf{T}}.\mathsf{a}$.
3. Else: Invoke MultisetEq to prove $\hat{\mathsf{T}}$ and $\hat{\mathsf{R}}$ are equal as multisets

7 Query plan optimization

As noted in Section 2.2, the query planner rewrites the query plan to minimize prover time. Also noted there was a caveat: all optimizations must preserve soundness, i.e., they must preserve the semantics of the original query regardless of the data. In particular, we are concerned about the case where the planner makes optimizations based on information about the data.

There are two cases here. In the first case, the data-dependence only affects efficiency, and not soundness. For example, reordering joins is always valid, but only profitable when it reduces the size of intermediate results. In the second case, when an optimization is only valid to apply if the data satisfies particular conditions, then the planner must insert checks to assert the satisfaction of those conditions. For example, a `LIMIT 10` operator can be removed if its input contains 8 rows,

[4] The schema is recorded in $\mathsf{cm_{DB}}$, and needs no re-proof.

[5] This optimization only applies when $\mathsf{R.k}$ contains no orphaned keys. If filtering L might leave orphans, we revert to the standard protocol.

but not if it contains 15 rows. Thus, to soundly remove the operator, the query planner would need to issue a proof that the number of active rows is at most 10 (e.g., via a sumcheck on the activator).

For simplicity, beyond the standard data-independent query optimizations, TruthTable's query planner conservatively applies two optimizations that fall into the first category of data-dependent optimizations.

Compaction: As mentioned in Section 6.4, the Compaction PIOP proves the identity of its input and output tables. Hence an identity node inserted into the query plan cannot change the semantics of the plan. [6] Our planner inserts this PIOP whenever it detects that the number of active rows of an intermediate table is at most half the hypercube size. Since the output table is smaller, subsequent PIOPs are faster to prove.

Switching inner joins. Whenever the query performs an inner join where the join key is the primary key of the first table and a foreign key of the second table, our planner switches from the generic inner join protocol to our specialized protocol that is optimized for this case. Because the verifier has access to the database schema, it can check when this rewrite is applicable soundly.

8 Implementation

We implement our system as two production-ready Rust libraries: (i) TruthTable[7], which realizes the end-to-end verifiable query execution system described in this paper, and (ii) `ark-piop`[8], the underlying infrastructure on which TruthTable relies and which may be of independent interest to implementers of other PIOP-based systems.

8.1 TruthTable System

We implement TruthTable as an end-to-end system for verifiable query execution in 60K lines of Rust code. Our implementation follows the outline laid out in Section 2, but performs some optimizations such as computing all table commitments in parallel whenever possible. Our query planner extends the industry-grade planner of Datafusion by adding the two optimizations we mentioned in Section 7. In doing so, it gains access to Datafusion's battle-tested suite of optimizations.

8.2 `ark-piop` infrastructure

Implementing complex PIOPs like TruthTable's requires care to ensure that **P** and **V** proceed in lockstep by updating the protocol transcript in the same order (even under prover parallelism), and these steps must be deterministic across executions of the same query. Moreover, such PIOPs give rise to hundreds of intermediate claims (sumcheck, zerocheck, evaluation, lookup, and others). For example, proving TPC-H Query 19 leads to over 700 claims (264 zerocheck, 152 sumcheck, 31 nonzerocheck, and 260 lookup). Managing this by hand is error-prone and hard to audit.

To address this, TruthTable introduces `ark-piop`, a new library (14k lines of Rust) that handles these concerns for SNARK implementers. It is a generic tool of likely interest to implementers of other PIOP-based systems, and it provides several PIOP-level optimizations.

[6] The exception is after a groupby operation, but this scenario depends only on the query and not the data, so it can be detected by the verifier and is not a soundness issue.

[7] `https://github.com/arkworks-rs/truth-table`

[8] `https://github.com/alireza-shirzad/ark-piop`

Batching fundamental claims. TruthTable's PIOPs reduce their checks to a large set of "fundamental claims", and proving each individually would be prohibitively expensive. `ark-piop` instead batches them: lookup claims using the ideas of Haböck [Hab22], zerocheck and sumcheck claims using those of HyperPlonk [CBBZ23], and nonzerocheck claims using a new technique: We batch t nonzerocheck claims on arithmetized columns $\hat{\mathsf{c}}_1, \ldots, \hat{\mathsf{c}}_t$ that may have different active sets, so standard random-linear-combination batching does not apply. The verifier samples $r_1, \ldots, r_t \in \mathbb{F}$, and both parties locally form the virtual columns $\hat{\mathsf{c}}'_i := (1,\ \hat{\mathsf{c}}_i.\mathsf{d} + (1 - \hat{\mathsf{c}}_i.\mathsf{a}) * r_i)$, each active on the entire hypercube. With high probability over r_i, $\hat{\mathsf{c}}'_i$ is non-zero everywhere if and only if $\hat{\mathsf{c}}_i$ is non-zero on its active set: a zero of $\hat{\mathsf{c}}_i$ on its active set ($\hat{\mathsf{c}}_i.\mathsf{a} = 1$) forces $\hat{\mathsf{c}}'_i$ to vanish there, while if $\hat{\mathsf{c}}_i$ is non-zero throughout its active set then $\hat{\mathsf{c}}'_i$ vanishes nowhere. Hence $\prod_{i=1}^{t} \hat{\mathsf{c}}'_i$ is non-zero iff every $\hat{\mathsf{c}}_i$ is, and the parties invoke Nonzerocheck on this single product to discharge all t claims at once.

Normalizing number of variables. Sumcheck requires all involved polynomials to share a common number of variables, but TruthTable generates polynomials with *different* variable counts (e.g., after compaction). To avoid the overhead of padding out to the worst-case number of variables μ_{max}, `ark-piop` tracks both μ and μ_{max} for each polynomial, optimizes operations such as evaluation and variable elimination to minimize time and space overheads.

Leveraging sparsity. Many polynomials in the sumcheck claims generated by TruthTable are sparse: only a few evaluations over the Boolean hypercube are non-zero. For example, the output $f = \hat{\mathsf{a}} * \hat{\mathsf{d}}$ of a selective filter is non-zero only in the selected locations. `ark-piop` exploits this by skipping field operations at zero evaluations. This optimization is beneficial in key expensive components such as polynomial commitments, sumcheck prover operations, and more.

Handling run-time constants. `ark-piop` represents *constant* polynomials with a single field element which enables it to (i) skip commitments to them, saving commitment time and proof size, and (ii) treat them as constants in virtual polynomials, lowering the degree and speeding up sumcheck.

Degree reduction. The virtual polynomial V we sumcheck over often has high-degree terms (frequently above 10 on TPC-H queries), and the prover's per-round work grows superlinearly in this degree. `ark-piop` mitigates this with a heuristic degree-reduction pass: given a target bound (default 6), it repeatedly commits once to a sub-product shared by many oversized terms of V and substitutes that commitment for every occurrence, lowering V's degree at the cost of a few extra commitments and zero-check constraints. We defer the full algorithm to Appendix D.

9 Evaluation

We evaluate the prover time, verifier time, and proof size of our implementation of TruthTable (TT) and compare it against state-of-the-art baselines.

Baselines. We compare TT against three state-of-the-art verifiable query processing systems: Poneglyph[9] [GFN25], `qedb`[10] [BBCRT25], and SpaceXTime's Proof of SQL[11] (`PoSQL`) [Spa25]. As noted in Section 10, each baseline represents the state-of-the-art in its respective design paradigm, i.e. circuit-based (Poneglyph), ADS-based (`qedb`), and PIOP-based (`PoSQL`). We provide a qualitative discussion of related work in Section 10.

[9] `https://github.com/tuzijun111/PoneglyphDB`

[10] We obtained `qedb`'s implementation directly from its authors via private communication.

[11] `https://github.com/spaceandtimefdn/sxt-proof-of-sql`

Experimental setup. We instantiate TT with the PST PC scheme [PST13] over the BN254 elliptic curve. We conduct our experiments on a machine with a 13th Gen Intel Core i7-13700 CPU with 16 hardware threads and 64 GB of RAM.

9.1 Benchmarks on TPC-H

TPC-H [PF00] is the standard benchmark for relational databases, consisting of 22 analytical SQL queries over a synthetic business dataset whose size is controlled by a *scale factor* (SF). We provide a sample query in Appendix E.

Setup. We evaluate TT with TPC-H scale factor 0.1. This corresponds to roughly 1 million rows in the largest table (`lineitem`). We prove 17 out of 22 queries in the benchmark; the remaining 5 queries, Q11, Q13, Q16, Q21, and Q22, rely on semi-joins or left/right joins that we do not yet support. As in previous work [LWXWR23; GFN25], we omit all string operations (e.g. `LIKE`) except string equality. We report TT's performance on these queries in Fig. 3.

Summary. At scale factor 0.1, table commitment time ranges from about 0.4 to 8.9 seconds, with only the two largest tables, `lineitem` and `orders`, exceeding one second. With one thread, TT's prover time ranges from 38 to 314 seconds, with the maximum on Q10, the most expensive query in the suite. While not reported in Fig. 3, using 4 threads reduces all times by roughly 3×. Verifier time ranges from 14 to 57 ms, with roughly half the verifier time spent on cryptographic work. Proof size ranges from 14 to 39 kB, A commitment's size ranges from 0.8 KB for the narrowest table (`region`, 3 columns) to 5.0 KB for the widest (`lineitem`, 16 columns) and averages about 2.0 KB across the tables (roughly 0.25 KB per column).

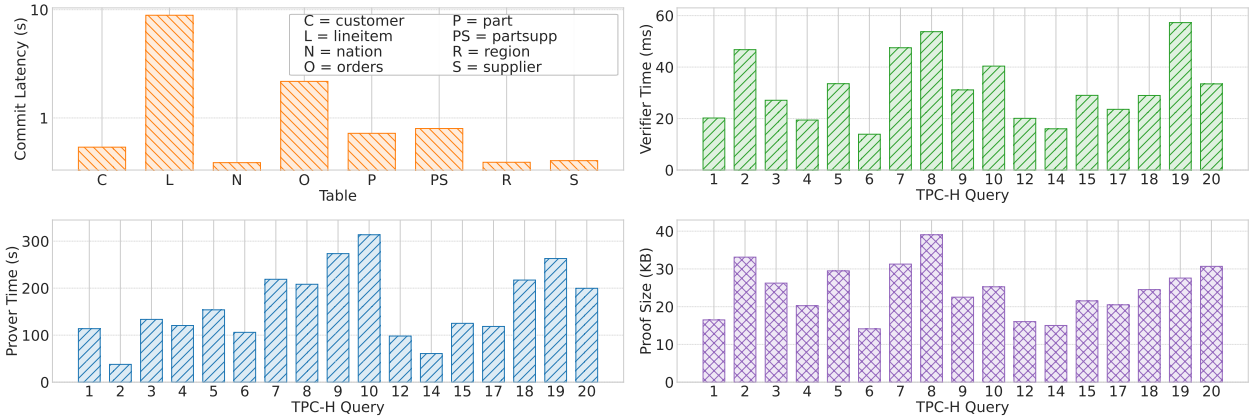


Figure 3: TT's standalone performance on the supported TPC-H queries, under the setup of Section 9.1.

As Table 1 shows, TT's prover scales approximately linearly: halving the scale factor to 0.05 roughly halves both commitment and prover time on every supported query, while leaving verifier time and proof and commitment size largely untouched, matching their expected asymptotic costs.

query	prover time (s)		query	prover time (s)	
	0.05	0.1		0.05	0.1
1	56.86	113.82	10	152.70	313.62
2	22.15	37.86	12	49.42	98.16
3	67.92	133.56	14	30.73	60.90
4	60.66	120.54	15	62.88	125.16
5	77.40	153.84	17	61.26	118.56
6	52.68	106.02	18	106.92	217.26
7	109.74	218.88	19	129.72	263.04
8	107.34	208.26	20	94.98	199.68
9	126.12	273.48			

Table 1: TruthTable's single-threaded prover time on TPC-H queries at scale factors (SF) 0.05 and 0.1.

9.2 The effect of optimizations

We evaluate the benefits of the optimizations introduced by TruthTable's query planner in reducing prover, verifier, and proof size costs. Our hypothesis is that since the two optimizations are complementary, their benefits largely compose.

Prover time. On scale factor 0.05 with a single thread, The PK/FK optimization accelerates join-heavy queries (geometric-mean speedup 1.14×, up to 2.07× on Q9) and leaves join-free queries unaffected, while rematerialization helps queries that thread wide intermediate results through several operators (geometric-mean 1.09×, up to 1.64× on Q15). Enabling both yields a geometric-mean speedup of 1.26× (up to 2.06×). The PK/FK optimization also shrinks the proof and the verifier's work, since the un-materialized join need not be committed: on Q9 it cuts proof size from 32.5 to 21.6 KB and verifier time from 42.3 to 30.0 ms.

As observed in Section 7, however, neither optimization is universally beneficial: each carries an overhead that the downstream savings do not always recoup, yielding slight regressions on a few queries (down to 0.92×). These cases are largely complementary, so the regressions tend to vanish once both are enabled (e.g., Q12 recovers from 0.92× to 1.45×). The right placement thus depends on the data distribution at hand; we leave an automated proof-aware planner to future work.

9.3 Comparison against Poneglyph

Benchmark setup. We compare with Poneglyph on TPC-H, as this is the only benchmark their system supports. We use the largest scale factor supported by Poneglyph, namely 0.04; at this scale factor, the `lineitem` table contains 240K rows. Both Poneglyph and TT use BN254 for underlying curve arithmetic. We compare against Poneglyph on Q1, Q3, Q5, Q8, Q9, and Q18.[12]

Summary. Fig. 4 reports the comparison result. The takeaway is that TT achieves 1. lower prover latency on all queries, ranging from 2.3× to 6.3× faster than Poneglyph, 2. 2.2×–3.9× higher verifier times, and 3. comparable proof sizes (0.7× to 1.8× larger) We could not compare commitment times or sizes as their implementation does not support generating table commitments.

[12]Poneglyph's benchmarks simplify Q1, Q3, and Q18 by removing some operators; we benchmark both systems on these simplified queries.

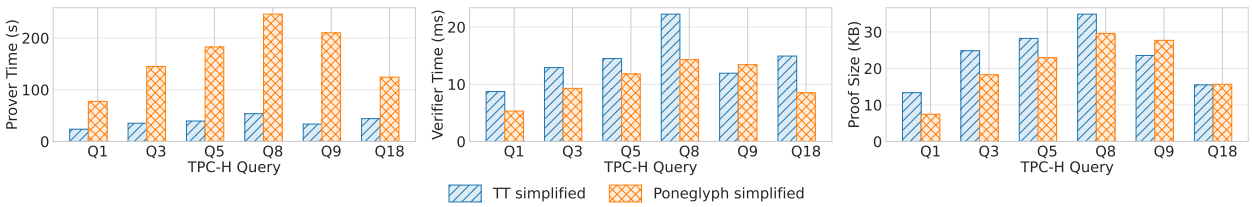


Figure 4: TruthTable versus Poneglyph on the abridged TPC-H queries, under the setup of Section 9.3.

To test how these costs change with the scale factor, we ran both systems at scale factors 0.01, 0.02, and 0.04, reporting the prover-time speedup of TT over Poneglyph in Table 2. TT scales better on every query: the speedup *grows* with the scale factor (for example, from 2.8× to 6.3× on Q9). We believe this is because TT's prover time is linear, while that of Poneglyph is quasilinear.

query	speedup over Poneglyph (×)		
	SF=0.01	SF=0.02	SF=0.04
Q1	3.1	3.1	3.3
Q3	3.6	3.8	4.1
Q5	3.7	4.1	4.6
Q8	3.1	3.6	4.6
Q9	2.8	4.6	6.3
Q18	2.3	2.6	2.8

Table 2: Single-threaded prover-time speedup of TruthTable over Poneglyph across scale factors.

9.4 Comparison against PoSQL and qedb

Neither PoSQL nor qedb support TPC-H queries, so we compare against them on synthetic workloads instead.

Setup. We benchmark five representative query types covering core relational operators: Filter, Aggregate, Join, and Limit/Offset (See Appendix E for a sample query). qedb supports neither Join nor Limit/Offset, so we do not compare against it on those. PoSQL uses HyperKZG over BN254 and qedb uses BLS12-381, and both systems hardcode their curves. To enable fair comparisons, when comparing against PoSQL, we set up TT to BN254, while when comparing against qedb, we set it up to use BLS12-381.

We generate tables for the queries via the PoSQL benchmark harness, and use them to evaluate all three systems. The generated tables have 2^{19} rows, with columns being bounded 64-bit or 32-bit integers. The key used for joining is a PK in one table and a FK in the other.

Summary. TT's prover time is consistently faster than both baselines across every operator. Compared to PoSQL (Fig. 5), TT proves Filter 11.4× faster, Aggregate 45.8× faster, Join 2.7× faster with the generic protocol, and 50× with the PK/FK protocol, and on Limit/Offset 23.2× faster. Compared to qedb (Fig. 6), TT proves Filter 4.2× faster and Aggregate 50.9× faster. These advantages are preserved as the database grows, as we confirm by sweeping the table size from 2^{16} to 2^{19} rows (Tables 3a and 3b). Against PoSQL, TT scales better overall: the speedup grows with table size on Filter, Join, and Limit/Offset, and is roughly constant (a tie in scaling) on Aggregate.

Against qedb, the behavior is operator-dependent: on Filter the two systems scale at the same rate, so the speedup stays flat at about 4×, while on Aggregate qedb scales sub-linearly and TT remains linear, so our speedup narrows with size but stays large (from 63× down to 50.9×).

Compared to PoSQL, TT's verifier time is 4–5× worse on Filter, Aggregate, and Limit/Offset, but is 3.4× better on Join; its proof sizes are essentially on par, ranging from 0.86× to 1.07× those of PoSQL. Compared to qedb, TT's verifier is far cheaper (122× faster on Filter and 36× faster on Aggregate), at the cost of larger proofs (3.1× on Filter and 12.7× on Aggregate).

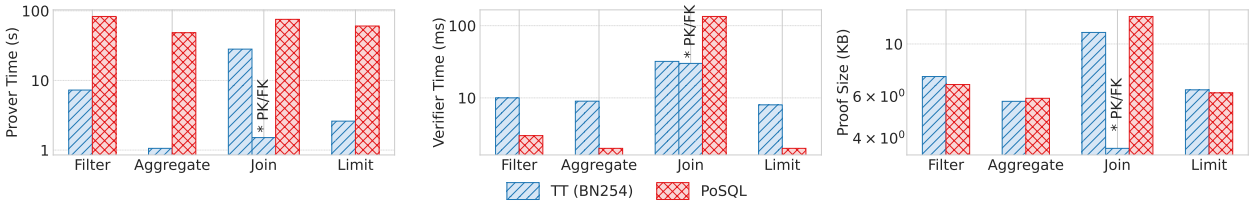


Figure 5: TruthTable vs. PoSQL on synthetic microbenchmarks, both on BN254, under the setup of Section 9.4.

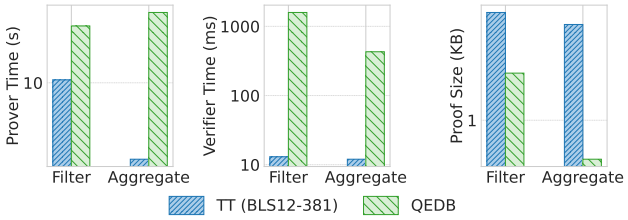


Figure 6: TruthTable vs. qedb on synthetic microbenchmarks, both on BLS12-381

operator	speedup (×)			
	2^{16}	2^{17}	2^{18}	2^{19}
Filter	9.0	9.8	10.5	11.4
Aggregate	43.0	46.1	44.6	45.8
Join	1.6	2.2	2.5	2.7
Limit	18.3	20.2	21.7	23.2

(a) over PoSQL

operator	speedup (×)			
	2^{16}	2^{17}	2^{18}	2^{19}
Filter	4.2	4.0	4.1	4.2
Aggregate	63.2	59.3	57.0	50.9

(b) over qedb

Table 3: Single-threaded prover-time speedup of TruthTable over PoSQL and qedb across table sizes.

10 Related work

Prior work on verifiable query processing broadly falls into three categories: systems for specific queries, systems for general queries that use general-purpose verifiable computation, and systems for general queries that modularly compose specialized subprotocols. We compare TruthTable against each kind of protocol next.

Query-specific systems usually design specialized Authenticated Data Structures (ADSes) that can prove a specific kind of query over a committed database extremely efficiently [LHKR06; DGK04; PPT14; PPTT15; YPPK09; BS13; ZXA12; HSBB25]. Unfortunately, it has proven difficult to extend these per-query ADSes to support complex SQL operations like joins, aggregates, or nested subqueries. Additionally, it is difficult to compose different ADSes together to prove complex SQL queries efficiently (but see `qedb` below for the latest stab at this).

A second class of systems supports **complex queries via verifiable computation** by compiling the queries into optimized circuits, and then proving the satisfaction of these circuits via general-purpose cryptographic proof systems. The foundational work in this line, vSQL [ZGKPP17b; ZGKPP17a], designs efficient circuits for use with the GKR protocol. ZKSQL [LWXWR23], uses VOLE-based interactive proofs, while PoneglyphDB [GFN25], the state-of-the-art in this approach, uses the halo2 proof system with PLONKish arithmetization.

By using existing proof systems as a building block, these systems inherit their benefits, such as zero-knowledge, existing security proofs, and implementation optimizations. However, they also incur two key downsides. First, as we show in Section 9.3, the use of off-the-shelf tools leads to worse prover times than in tailored protocols. Second, and more importantly, query circuits are specialized both for specific queries *and* for specific database sizes; changing either one requires constructing a new circuit and, for preprocessing SNARKs such as those used by Poneglyph, recomputing its verifier key. This latter step requires (super)linear verifier work as circuit size can increase (super)linearly with database size.[13]

Finally, the third class of systems supports **complex queries** via **composition of subprotocols** for relational operators. TruthTable falls into this category. Other systems that use this approach are `qedb` [BBCRT25] and the industrial Proof-of-SQL (`PoSQL`) system [Spa25]. We have provided quantitative comparisons against these in Section 9, so here we focus on qualitative differences.

`qedb`'s fundamental algebraic building block is vectors, and so its cryptographic protocols are designed by combining specialized "vector IOPs" with homomorphic vector commitments and accumulators based on bilinear pairings. These protocols are of a much different flavour to those of TruthTable, and provide different trade-offs.

Beyond differing building blocks, a key difference between `qedb` and TruthTable is that in `qedb`, tables must be preprocessed to support different queries and operators. For example, joins require creating and committing to specialized indices. This means that when queries change, so does the required preprocessed commitment. TruthTable, on the other hand, does not require any additional preprocessing or commitments beyond the initial table commitments, and hence does not suffer from these shortcomings.

`PoSQL` [Spa25] is closer to our design in both theory and practice. From the protocol side, `PoSQL`, like TruthTable, employs sumcheck-based protocols for SQL operations. Indeed, their protocols for some operators, such as `JOIN`s, are very similar to ours. From the implementation side, their implementation, like ours, is also closely integrated with Apache DataFusion.

However, there are also differences on both fronts. On the protocol design front, unlike TruthTable, `PoSQL`'s operators do not have activators. This makes protocols such as filters more expensive, as one now has to materialize the filtered table, as opposed to the updated activator. They also do not implement optimized PK/FK joins. Even the protocols that *are* similar, such as joins and aggregates, are simpler in that they do not support multi-column keys. Finally, unlike TruthTable

[13]This can be avoided if the underlying proof system supports uniform computation; vSQL's GKR does, but Poneglyph's halo2 does not.

and other academic works, PoSQL provides no formal protocol specifications or soundness analysis.

Acknowledgements

We thank the anonymous reviewers for their helpful feedback. We thank Binbin Gu for clarifications and help regarding the Poneglyph paper, and Alberto Trombetta and Emanuele Ragnoli for providing access to the qedb codebase.

References

[BBCRT25] V. Botta, S. Bottoni, M. Campanelli, E. Ragnoli, and A. Trombetta. "qedb: Expressive and Modular Verifiable Databases (without SNARKs)". Cryptology ePrint Archive, Paper 2025/1408. 2025.

[BFS20] B. Bünz, B. Fisch, and A. Szepieniec. "Transparent SNARKs from DARK Compilers". In: *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT '20. 2020, pp. 677–706.

[BS13] S. Bajaj and R. Sion. "CorrectDB: SQL Engine with Practical Query Authentication". In: *Proc. VLDB Endow.* 6.7 (2013), pp. 529–540.

[CBBZ23] B. Chen, B. Bünz, D. Boneh, and Z. Zhang. "HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates". In: *Proceedings of the 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT '23. 2023, pp. 499–530.

[CFQ19] M. Campanelli, D. Fiore, and A. Querol. "LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs". In: *Proceedings of the 26 ACM Conference on Computer and Communications Security*. CCS '19. 2019, pp. 2075–2092.

[CHMMVW20] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. "Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS". In: *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT '20. 2020.

[DGK04] P. T. Devanbu, M. Gertz, and A. Kwong. "Flexible authentication of XML documents". In: *J. Comput. Secur.* 12.6 (2004), pp. 841–864.

[GFN25] B. Gu, J. Fang, and F. Nawab. "PoneglyphDB: Efficient Non-interactive Zero-Knowledge Proofs for Arbitrary SQL-Query Verification". In: *Proc. ACM Manag. Data* 3.1 (2025), 63:1–63:27.

[GPS25] C. Ganesh, S. Patranabis, and N. Singh. "Samaritan: Linear-Time Prover SNARK from New Multilinear Polynomial Commitments". In: *Proceedings of the 31st International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT '25. 2025.

[Hab22] U. Haböck. "Multivariate lookups based on logarithmic derivatives". Cryptology ePrint Archive, Paper 2022/1530. https://eprint.iacr.org/2022/1530. 2022.

[HSBB25] H. Hafezi, A. Shirzad, B. Bünz, and J. Bonneau. "IronDict: Transparent Dictionaries from Polynomial Commitments". Cryptology ePrint Archive, Paper 2025/1580. 2025.

[KZG10] A. Kate, G. M. Zaverucha, and I. Goldberg. "Constant-Size Commitments to Polynomials and Their Applications". In: *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT '10. 2010, pp. 177–194.

[LFKN92] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. "Algebraic Methods for Interactive Proof Systems". In: *Journal of the ACM* 39.4 (1992), pp. 859–868.

[LHKR06] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. "Dynamic authenticated index structures for outsourced databases". In: *Proceedings of the ACM International Conference on Management of Data*. SIGMOD '06. 2006, pp. 121–132.

[Lip89] R. J. Lipton. "New Directions In Testing". In: *Proceedings of a DIMACS Workshop in Distributed Computing And Cryptography*. 1989, pp. 191–202.

[LWXWR23] X. Li, C. Weng, Y. Xu, X. Wang, and J. Rogers. "ZKSQL: Verifiable and Efficient Query Evaluation with Zero-Knowledge Proofs". In: *Proc. VLDB Endow.* 16.8 (2023), pp. 1804–1816.

[OLB25] A. Ozdemir, E. Laufer, and D. Boneh. "Volatile and Persistent Memory for zkSNARKs via Algebraic Interactive Proofs". In: *Proceedings of the 46th IEEE Symposium on Security and Privacy*. IEEE S&P '25. 2025.

[PF00] M. Poess and C. Floyd. "New TPC Benchmarks for Decision Support and Web Commerce". In: *SIGMOD Records*. SIGMOD '00 29.4 (2000), pp. 64–71.

[PPT14] D. Papadopoulos, S. Papadopoulos, and N. Triandopoulos. "Taking Authenticated Range Queries to Arbitrary Dimensions". In: *Proceedings of the 2014 ACM Conference on Computer and Communications Security*. CCS '14. ACM, 2014, pp. 819–830.

[PPTT15] D. Papadopoulos, C. Papamanthou, R. Tamassia, and N. Triandopoulos. "Practical Authenticated Pattern Matching with Optimal Proof Size". In: *Proc. VLDB Endow.* 8.7 (2015), pp. 750–761.

[PST13] C. Papamanthou, E. Shi, and R. Tamassia. "Signatures of Correct Computation". In: *Proceedings of the 10th Theory of Cryptography Conference*. TCC '13. 2013, pp. 222–242.

[Spa25] Space and Time. *Space and Time — Proof of SQL (sxt-proof-of-sql)*. Version v0.100.0. GitHub repository (tag v0.100.0, commit 1c27b7f). Accessed 2026-02-06. May 7, 2025.

[YPPK09] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. "Authenticated join processing in outsourced databases". In: *Proceedings of the ACM International Conference on Management of Data*. SIGMOD '09. 2009, pp. 5–18.

[ZGKPP17a] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. "A Zero-Knowledge Version of vSQL". Cryptology ePrint Archive, Report 2017/1146. 2017.

[ZGKPP17b] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. "vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases". In: *Proceedings of the 38th IEEE Symposium on Security and Privacy*. S&P '17. 2017, pp. 863–880.

[ZKP15] Y. Zhang, J. Katz, and C. Papamanthou. "IntegriDB: Verifiable SQL for Outsourced Databases". In: *Proceedings of the 22nd ACM Conference on Computer and Communications Security*. CCS '15. 2015, pp. 1480–1491.

[ZXA12] Q. Zheng, S. Xu, and G. Ateniese. "Efficient query integrity for outsourced dynamic databases". In: *Proceedings of the ACM Workshop on Cloud Computing Security*. CCSW '12. 2012, pp. 71–82.

A Extended preliminaries

A.1 s-Contiguous Polynomial Identity

Here we show that the s-contiguous polynomial, defined in Section 3 as: $\mathcal{C}_s(X) := \sum_{i=1}^{\mu} s_i(1 - X_i) \prod_{j=1}^{i-1} (s_j X_j + (1 - s_j)(1 - X_j))$ is 1 for the first s points in $\mathscr{H}_\mu$ and 0 elsewhere. Here, $s_i =$

$\lfloor \frac{s}{2^{\mu-i}} \rfloor \bmod 2$ is the i-th most significant bit of s. Equivalently, $\mathcal{C}_s(X) = 1$ if and only if $\mathcal{I}_\mu(X) < s$, and $\mathcal{C}_s(X) = 0$ otherwise. Here, we will provide a short derivation of this identity.

Observe that the condition $\mathcal{I}_\mu(X) < s$ is satisfied if and only if there is some position i such that:

1. **Prefix Match:** For all indices $j < i$ (bits more significant than i), the bit X_j is identical to s_j.
2. **Current Bit Lower:** At index i, the bit X_i is strictly less than s_i. Since $X_i, s_i \in \{0, 1\}$, this forces $s_i = 1$ and $X_i = 0$.

A.2 Commit-carrying SNARKs

We recall the definitions of completeness and knowledge soundness for cc-SNARKs.

Completeness. For every efficient adversary $\mathcal{A}$, the following probability is 1.

$$\Pr\left[\begin{array}{c|r} (\mathbb{x}, \mathbb{y}, \mathbb{w}) \in \mathscr{R} & (\mathsf{pk}, \mathsf{ck}, \mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda, N) \\ \Downarrow & (\mathbb{xy}, \mathbb{w}) \leftarrow \mathcal{A}(\mathsf{pk}) \\ \mathsf{Verify}(\mathsf{vk}, \mathbb{x}, \mathsf{cm}, \pi) = 1 & \mathsf{cm} \leftarrow \mathsf{Commit}(\mathsf{ck}, \mathbb{y}) \\ & \pi \leftarrow \mathsf{Prove}(\mathsf{pk}, \mathbb{x}, \mathbb{y}, \mathbb{w}) \end{array}\right]$$

Soundness. For every efficient adversary $\mathcal{A}$, there exists an efficient extractor $\mathcal{E}$ such that the following probability is negligible.

$$\Pr\left[\begin{array}{c|r} (\mathbb{x}, \mathbb{y}, \mathbb{w}) \notin \mathscr{R} & (\mathsf{pk}, \mathsf{ck}, \mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda, N) \\ \vee & (\mathbb{x}, \mathsf{cm}, \pi) \leftarrow \mathcal{A}(\mathsf{pk}) \\ \mathsf{Commit}(\mathsf{ck}, \mathbb{y}) \neq \mathsf{cm} \wedge & (\mathbb{y}, \mathbb{w}) \leftarrow \mathcal{E}_{\mathcal{A}}(\mathsf{pk}) \\ \mathcal{V}(\mathsf{vk}, \mathbb{x}, \mathsf{cm}, \pi) = 1 & \end{array}\right]$$

The polynomial captures this logic by summing over all possible positions i where this divergence could occur. The inner product, $\prod_{j=1}^{i-1}(s_j X_j + (1 - s_j)(1 - X_j))$, acts as an equality indicator for the prefix, evaluating to 1 if and only if $X_j = s_j$ for all $j < i$. The term $s_i(1 - X_i)$ enforces the strict inequality at the current bit. Because the "first position of difference" is unique for any distinct pair of integers, these conditions are mutually exclusive across different i. Thus, the summation yields exactly 1 if such an i exists and 0 otherwise.

A.3 PIOP security properties

Here $\mathbb{x}$ denotes the explicit public input, while $\mathbb{y}$ denotes any implicit input exposed to the verifier through polynomial oracles, as with the arithmetized tables in TruthTable. A PIOP $\Pi = (\mathbf{P}, \mathbf{V})$ for $\mathscr{R}$ fixes, for every prover oracle, an arity and degree bound. A possibly malicious prover $\tilde{\mathbf{P}}$ is *admissible* if every oracle it sends in every round is a polynomial of the prescribed arity and degree. This admissibility condition is part of the ideal PIOP model; after compilation, the polynomial commitment scheme is responsible for enforcing the corresponding degree and evaluation claims.

Definition A.1 (PIOP completeness)**.** A PIOP $\Pi = (\mathbf{P}, \mathbf{V})$ has perfect completeness for $\mathscr{R}$ if, for every $(\mathbb{x}, \mathbb{y}, \mathbb{w}) \in \mathscr{R}$, the honest prover convinces the verifier with probability 1:

$$\Pr\left[\mathbf{V}^{[\![\mathbb{y}]\!], \mathbf{P}(\mathbb{x}, \mathbb{y}, \mathbb{w})}(\mathbb{x}) = \mathbf{1}\right] = 1.$$

Definition A.2 (PIOP soundness)**.** A PIOP $\Pi = (\mathbf{P}, \mathbf{V})$ has soundness error ϵ_s for $\mathscr{R}$ if, for every false instance $(\mathbb{x}, \mathbb{y}) \notin \mathcal{L}_{\mathscr{R}}$ and every admissible prover $\tilde{\mathbf{P}}$,

$$\Pr\left[\mathbf{V}^{[\![\mathbb{y}]\!], \tilde{\mathbf{P}}}(\mathbb{x}) = \mathbf{1}\right] \leq \epsilon_s.$$

A.4 Deferred Low-Level PIOPs

Sumcheck. Sumcheck reduces an exponential sum to a sequence of univariate sums, each involving one fewer variable. At each step, the prover sends a univariate polynomial that partially sums over the remaining variables. The verifier checks consistency between rounds by evaluating each polynomial at a random point and verifying the sum matches the previous claim. On the last round, the verifier queries the original oracle to confirm the final value.

PIOP 21: Sumcheck. Inputs: $[\![p]\!], s$

1. $\mathbf{P}$ sends to $\mathbf{V}$ a univariate polynomial oracle p_1 such that $p_1(X_1) = \sum_{\{x_2,\ldots,x_\mu\} \in \mathscr{H}_{\mu-1}} p(X_1, x_2, \ldots, x_\mu)$.
2. $\mathbf{V}$ checks that the degree of $p_1(x_1)$ is at most the degree of p in x_1. It computes $s_1 = p_1(0) + p_1(1)$ and checks that $s_1 = s$. If not, $\mathbf{V}$ rejects. It then picks a random $r_1 \in \mathbb{F}$ and sends it to the Prover ($\mathbf{P}$).
3. For each i from 2 to μ (subsequent rounds).
   (a) $\mathbf{P}$ sends to $\mathbf{V}$ the univariate polynomial $p_i(x_i) = \sum_{\{x_{i+1},\ldots,x_\mu\} \in \mathscr{H}_{\mu-i}} p(r_1, \ldots, r_{i-1}, x_i, \ldots, x_\mu)$.
   (b) $\mathbf{V}$ checks that the degree of $p_i(x_i)$ is at most the degree of p in x_i. It computes $s_i = p_i(0) + p_i(1)$ and checks that $s_i = p_{i-1}(r_{i-1})$. If not, $\mathbf{V}$ rejects. It picks a random $r_i \in \mathbb{F}$ and sends it to $\mathbf{P}$.
4. $\mathbf{P}$ sends the value $p(r_1, r_2, \ldots, r_{\mu-1}, r_\mu)$ for the random values $r_1, r_2, \ldots, r_\mu$ chosen by $\mathbf{V}$.
5. $\mathbf{V}$ checks that this value is correct by querying p at $(r_1, r_2, \ldots, r_{\mu-1}, r_\mu)$.

PIOP 22: Zerocheck [CBBZ23]. Inputs: $[\![p]\!]$

1. $\mathbf{V}$ samples a vector $r \xleftarrow{\$} \mathbb{F}^\mu$ and sends it to $\mathbf{P}$.
2. Invoke the Sumcheck PIOP for the claim that $\sum_{x \in \mathscr{H}_\mu} p(x) * \mathrm{eq}(x, r) = 0$.

Rational Sumcheck. The prover constructs a polynomial $h(x)$ that equals $p(x)/q(x)$ on the $\mathscr{H}_\mu$ and sends its oracle to the verifier, who can then be convinced that h behaves like the intended rational function via a Zerocheck. With a verified h in hand, the parties reduce the problem to a standard Sumcheck over h.

PIOP 23: RatSumcheck. Inputs: $[\![p]\!], [\![q]\!], s$

1. $\mathbf{P}$ constructs the polynomial h such that for all $x \in \mathscr{H}_\mu$, $h(x) = \frac{p(x)}{q(x)}$.
2. $\mathbf{P}$ sends $|q|$ to $\mathbf{V}$.
3. Invoke Zerocheck for the claim that $\forall x \in \mathscr{H}_\mu$, $h(x)q(x) - p(x) = 0$.
4. Invoke Sumcheck for the claim $\sum_{x \in \mathscr{H}_\mu} h(x) = s$.

B Deferred Protocols

In this appendix, we present the formal definitions of several PIOP protocols that were deferred from Sections 5 and 6.

B.1 NoDupcheck without sorting

This appendix presents an alternative NoDupcheck that proves a table has no duplicate rows without sorting, using Bézout's identity instead. It is useful when the table contains large values or arbitrary field elements that are untenable to sort.

The Bézout-based PIOP. Bézout's NoDupcheck takes oracle $[\![\hat{\mathsf{T}}]\!]$. The parties first fingerprint $\hat{\mathsf{T}}$ into a single column $\hat{\mathsf{c}}$, reducing the question to whether $\hat{\mathsf{c}}.\mathsf{d}$ has duplicate active entries. With high probability, $\hat{\mathsf{c}}.\mathsf{d}$ has duplicate active entries if and only if $\hat{\mathsf{T}}$ has duplicate rows, so the rest of the protocol works on $\hat{\mathsf{c}}$.

To allow randomization at inactive indices, we lift $\hat{\mathsf{c}}.\mathsf{d}$ to a polynomial p that agrees with $\hat{\mathsf{c}}.\mathsf{d}$ on active indices and is random elsewhere. Following [OLB25], the question of whether p has duplicate entries reduces to whether the vanishing polynomial $z(Y) = \prod_{b \in \mathscr{H}_\mu}(Y - p(b))$ has a repeated root. By Bézout's identity, $z(Y)$ has no repeated roots if and only if it is coprime with its derivative $z'(Y)$, that is, there exist univariate polynomials s, t such that

$$z(Y)s(Y) - z'(Y)t(Y) = 1.$$

We verify this identity at a random point $r \in \mathbb{F}$ via Schwartz–Zippel.

To compute $z(r)$ and $z'(r)$, the prover sends a number of auxiliary oracles. First, it sends an oracle $[\![\hat{p}_r]\!]$ for $p_r(X) := p(X) - r$, and the parties invoke DataEqcheck to prove $\hat{\mathsf{c}}.\mathsf{d} = \hat{p}_r + r$ at $\hat{\mathsf{c}}.\mathsf{a}$, ensuring $\hat{p}_r$ is correctly formed. Next, the prover sends univariate oracles $[\![s]\!], [\![t]\!]$, which need no verification since any valid certificate works. Finally, it sends multivariate oracles $[\![f]\!], [\![f^*]\!] : \mathbb{F}^\mu \to \mathbb{F}$ satisfying, for all $a \in \{0,1\}$ and $b \in \mathscr{H}_{\mu-1}$,

$$\begin{aligned} f(a,b) - a[f(b,0)f(b,1)] \\ - (1-a)[p_r(b,0)p_r(b,1)] = 0, \end{aligned} \tag{3}$$

$$\begin{aligned} f^*(a,b) - a[f^*(b,0)f(b,1) + f^*(b,1)f(b,0)] \\ - (1-a)[p_r(b,0) + p_r(b,1)] = 0. \end{aligned} \tag{4}$$

These identities make $f(1,\ldots,1,0) = z(r)$ and $f^*(1,\ldots,1,0) = z'(r)$. To see why, view the evaluations of f as a flattened binary tree whose leaves are $p_r(x)$ for $x \in \mathscr{H}_\mu$. When $a = 0$ (the first half of the hypercube) each entry of f encodes the product of two adjacent entries of p_r. When $a = 1$, each entry of f encodes the product of two adjacent entries from the previous level of f. This pattern compounds until we reach the root $f(1,\ldots,1,0) = z(r)$. A similar argument shows that f^* encodes the derivative tree with root $f^*(1,\ldots,1,0) = z'(r)$.

After verifying Eqs. (3) and (4) via Zerochecks, the verifier directly checks that $s(r)z(r) - t(r)z'(r) = 1$, which by Bézout's identity implies z has no repeated roots. Therefore $\hat{\mathsf{c}}.\mathsf{d}$ has no duplicate active entries and, by extension, $\hat{\mathsf{T}}$ has no duplicate rows.

PIOP 24: Bézout's NoDupcheck. Inputs: $[\![\hat{\mathsf{T}}]\!]$

1. **P** and **V** fingerprint $\hat{\mathsf{T}}$ to obtain a single column $\hat{\mathsf{c}}$.
2. **V** samples a random point $r \xleftarrow{\$} \mathbb{F}$ and sends it to **P**.
3. **P** computes and sends $[\![\hat{p}_r]\!]$ as defined above.
4. Invoke DataEqcheck for the claim $\hat{\mathsf{c}}.\mathsf{d} = \hat{p}_r + r$ at $\hat{\mathsf{c}}.\mathsf{a}$.
5. **P** computes and sends **V** univariate oracles $[\![s]\!], [\![t]\!]$ and multivariate oracles $[\![f]\!], [\![f^*]\!]$.
6. Invoke Zerocheck twice to verify Eqs. (3) and (4).
7. **V** checks that $s(r)f(1,\ldots,1,0) - t(r)f^*(1,\ldots,1,0) = 1$.

B.2 Keyed sumcheck

KeyedSumcheck takes two arithmetized key–value tables $\hat{\mathsf{L}}$ and $\hat{\mathsf{R}}$, defined over $\mathscr{H}_\mu$ and $\mathscr{H}_\nu$ respectively, and proves that their values have the same sum within every key group. For $\mathsf{T} \in \{\mathsf{L}, \mathsf{R}\}$ and $k \in \mathbb{F}$, write

$$A_k(\hat{\mathsf{T}}) := \{x \in A(\hat{\mathsf{T}}) \mid \hat{\mathsf{T}}.\mathsf{k}(x) = k\}, \qquad S_{\mathsf{T}}(k) := \sum_{x \in A_k(\hat{\mathsf{T}})} \hat{\mathsf{T}}.\mathsf{d}(x).$$

The desired claim is precisely that $S_{\mathsf{L}} = S_{\mathsf{R}}$ as functions $\mathbb{F} \to \mathbb{F}$. We use the following logarithmic-derivative identity from Haböck, which is the same principle underlying Lookup.

Function equality lemma [Hab22]. Let $S_{\mathsf{L}}, S_{\mathsf{R}} : \mathbb{F} \to \mathbb{F}$ be any functions. Then $S_{\mathsf{L}}(k) = S_{\mathsf{R}}(k)$ for all $k \in \mathbb{F}$ if and only if

$$\sum_{k \in \mathbb{F}} \frac{S_{\mathsf{L}}(k)}{X - k} = \sum_{k \in \mathbb{F}} \frac{S_{\mathsf{R}}(k)}{X - k}. \tag{5}$$

For each $\mathsf{T} \in \{\mathsf{L}, \mathsf{R}\}$, as the sets $A_k(\hat{\mathsf{T}})$ partition the active indicies $A(\hat{\mathsf{T}})$, we can rewrite each side of Eq. (5) as follows:

$$\sum_{k \in \mathbb{F}} \frac{S_{\mathsf{T}}(k)}{X - k} = \sum_{k \in \mathbb{F}} \frac{\sum_{x \in A_k(\hat{\mathsf{T}})} \hat{\mathsf{T}}.\mathsf{d}(x)}{X - k} = \sum_{x \in A(\hat{\mathsf{T}})} \frac{\hat{\mathsf{T}}.\mathsf{d}(x)}{X - \hat{\mathsf{T}}.\mathsf{k}(x)}$$

Applying this transformation to both sides of Eq. (5) and using the respective activators to zero-out inactive terms, we can rewrite it to sums over the boolean hypercubes:

$$\sum_{x \in \mathscr{H}_\mu} \frac{\hat{\mathsf{L}}.\mathsf{a}(x) * \hat{\mathsf{L}}.\mathsf{d}(x)}{X - \hat{\mathsf{L}}.\mathsf{k}(x)} = \sum_{x \in \mathscr{H}_\nu} \frac{\hat{\mathsf{R}}.\mathsf{a}(x) * \hat{\mathsf{R}}.\mathsf{d}(x)}{X - \hat{\mathsf{R}}.\mathsf{k}(x)}. \tag{6}$$

As in Lookup, Schwartz–Zippel reduces Eq. (6) to a random evaluation: $\mathbf{V}$ samples $r \xleftarrow{\$} \mathbb{F}$, $\mathbf{P}$ sends the claimed common value s, and two RatSumcheck invocations prove that the two sides equal s at $X = r$.

PIOP 25: KeyedSumcheck. Inputs: $[\![\hat{\mathsf{L}}]\!], [\![\hat{\mathsf{R}}]\!]$

1. $\mathbf{V}$ samples a random point $r \xleftarrow{\$} \mathbb{F}$ and sends it to $\mathbf{P}$.
2. $\mathbf{P}$ sends the claimed common value s of Eq. (6) at $X = r$.
3. Invoke RatSumcheck twice to prove that both sides of Eq. (6) equal s at $X = r$.

B.3 Matching pair check

The *matching pair check* takes two arithmetized columns $\hat{\mathsf{l}}$ and $\hat{\mathsf{r}}$ together with an integer s. Let $\mathcal{J} = \{(i, j) \in A(\hat{\mathsf{l}}) \times A(\hat{\mathsf{r}}) : \hat{\mathsf{l}}(i) = \hat{\mathsf{r}}(j)\}$ be the set of all pairs of active indices on which the two columns are equal. The goal of this protocol is to prove that $|\mathcal{J}| = s$.

To recast this counting property as a sumcheck claim, we introduce a duplicate-free witness column $\hat{\mathsf{c}^{\cup}}$ that contains every value occurring in $\hat{\mathsf{l}}$ or $\hat{\mathsf{r}}$ exactly once. Let $m^{\mathsf{L}}(k) = \left|\{i \in A(\hat{\mathsf{l}}) : \hat{\mathsf{l}}(i) = \hat{\mathsf{c}^{\cup}}(k)\}\right|$ and $m^{\mathsf{R}}(k) = \left|\{j \in A(\hat{\mathsf{r}}) : \hat{\mathsf{r}}(j) = \hat{\mathsf{c}^{\cup}}(k)\}\right|$ be the multiplicities of $\hat{\mathsf{c}^{\cup}}(k)$ in $\hat{\mathsf{l}}$ and $\hat{\mathsf{r}}$. Then, we can prove matchpair check by showing the following 3 properties:

1. *Distinctness.* $\hat{\mathsf{c}^{\cup}}$ contains no duplicate values.

2. *Superset.* Every active entry of $\hat{\mathsf{l}}$ and $\hat{\mathsf{r}}$ appears in $\hat{\mathsf{c}^{\cup}}$.
3. *Match count.* $\sum_{k \in A(\hat{\mathsf{c}^{\cup}})} m^{\mathsf{L}}(k) * m^{\mathsf{R}}(k) = s$.

Soundness. For any point $k \in A(\hat{\mathsf{c}^{\cup}})$, we have $m^{\mathsf{L}}(k) * m^{\mathsf{R}}(k)$ as the number of pairs in $\mathcal{J}$ that share the value $\hat{\mathsf{c}^{\cup}}(k)$. And, as $\hat{\mathsf{c}^{\cup}}$ contains every point in $\hat{\mathsf{l}}$ and $\hat{\mathsf{r}}$ and is duplicate-free, the sum of these products over all $k \in A(\hat{\mathsf{c}^{\cup}})$ has one term for every possible shared entry, and therefore equals the number of matching pairs.

Distinctness is proven straightforwardly by a NoDupcheck on $\hat{\mathsf{c}^{\cup}}$, and the superset property is proven by two separate Lookup invocations. As $\hat{\mathsf{c}^{\cup}}$ is proven to be duplicate-free, Lookup will return multiplicity polynomials $\hat{m^{\mathsf{L}}}$ and $\hat{m^{\mathsf{R}}}$ in the exact form we desire. So, we may check the match count condition using a Sumcheck.

PIOP 26: MatchPaircheck. Inputs: $[\![\hat{\mathsf{l}}]\!], [\![\hat{\mathsf{r}}]\!], s$

1. **P** sends $[\![\hat{\mathsf{c}^{\cup}}]\!]$ to **V**.
2. Invoke NoDupcheck on $\hat{\mathsf{c}^{\cup}}$.
3. Invoke Lookup twice to prove $\hat{\mathsf{l}} \sqsubset \hat{\mathsf{c}^{\cup}}$ and $\hat{\mathsf{r}} \sqsubset \hat{\mathsf{c}^{\cup}}$. **V** receives $[\![\hat{m^{\mathsf{L}}}]\!]$ and $[\![\hat{m^{\mathsf{R}}}]\!]$ as outputs.
4. Invoke Sumcheck to prove $\sum_{k \in A(\hat{\mathsf{c}^{\cup}})} m^{\mathsf{L}}(k) * m^{\mathsf{R}}(k) = s$

C Security of TruthTable's protocol

TruthTable is a non-interactive cc-SNARK for $\mathcal{R}_{\mathsf{SQL}}$ that is constructed from a PIOP + PC scheme. Because this transformation is standard, and because the knowledge soundness of the SNARK follows from the soundness of the PIOP (and extractability of the PC scheme), we focus on describing and proving soundness of our PIOP.

Recall that our goal is to demonstrate a PIOP for proving membership in (the arithmetized version of) $\mathcal{L}_{\mathsf{QP}}$.We now describe this PIOP.

The PIOP prover and verifier takes as explicit input a query plan QP and a claimed query result res. The prover receives the database DB explicitly, while the verifier receives oracle access to its arithmetization.

The protocol is described below.

PIOP 27: TTPIOP. Inputs: $\mathsf{QP}, \mathsf{res}, [\![\hat{\mathsf{DB}}]\!]$

1. **P** computes the table assignment and sends its arithmetization $\{[\![\hat{\mathsf{T}}_i]\!]\}_{i=1}^{m}$ to **V**.
2. For each node $\nu \in \mathsf{QP}$, Invoke the operator PIOP for $\nu.\mathsf{op}$ on oracle access to $\{\hat{\mathsf{T}}_{\mathsf{in}}\}_\nu$ and $\hat{\mathsf{O}}_\nu$, proving $(\{\mathtt{I}\}_\nu, \mathsf{O}_\nu) \in \mathcal{L}_{\nu.\mathsf{op}}$.
3. Invoke ResultCheck on $[\![\hat{\mathsf{T}}_m]\!]$ and res, proving $\mathsf{T}_m = \mathsf{res}$.
4. **V** accepts if and only if every invoked subprotocol accepted.

Lemma C.1 (completeness and soundness (informal))**.** If each node-specific PIOP is complete and sound for its operator language, and if ResultCheck is complete and sound, TTPIOP is a complete and sound PIOP for $\mathcal{R}_{\mathsf{QP}}$.

Proof sketch. **Completeness.** If $\mathsf{res} = \mathsf{Q}(\mathsf{DB})$, then the prover can produce a valid (arithmetized) table assignment. Moreover, each node PIOP will lead to verifier acceptance. ResultCheck will also accept as the output arithmetized table will agree with res. Finally, the honest prover strategy

does indeed only produce polynomials of the prescribed number of variables. Hence, the PIOP is complete.

Soundness. Assume for contradiction that $\mathsf{res} \neq \mathsf{Q}(\mathsf{DB})$, yet $\mathbf{V}$ accepts with non-negligible probability. This means that all the node PIOPs and ResultCheck accepted.

Since $\mathsf{res} \neq \mathsf{Q}(\mathsf{DB})$, this means that no valid table assignment exists. This in turn means that either (a) all intermediate tables assigned to the non-leaf edges are correct inputs and outputs for their nodes, or (b) at least one of the intermediate tables is either an incorrect input or incorrect output for one of its incident nodes. That is, the table is not in the corresponding language for that node.

In case (a), since all tables are correct, completeness tells us that the output edge is a valid result of executing QP on DB However, since $\mathsf{res} \neq \mathsf{Q}(\mathsf{DB})$, it means this output edge table is not equal to res. However, ResultCheck still passes. This contradicts the soundness of ResultCheck.

This means that we are left with case (b), that is, there is some node $\nu^\star$ whose input-output tables are not in $\nu^\star$'s language. However, since all node PIOPs accepted, we know that in particular $\nu^\star$'s PIOP accepted. This contradicts the soundness of that PIOP.

So overall it cannot be the case that $\mathsf{res} \neq \mathsf{Q}(\mathsf{DB})$ yet TTPIOP's verifier accepts. □

D Heuristic degree reduction

We assume $V = \sum_i c_i \prod_j f_{i,j}$ is in *atomized form*, i.e. every factor $f_{i,j}$ is a committed multilinear polynomial or a linear combination of such. (Any V can be put in this form by a one-time pass that distributes products over sums at every factor of degree ≥ 2.) A naive reduction commits, for each oversized term, the product of d of its factors, but this wastes commitments when terms share a sub-product. Our heuristic instead repeatedly commits once to the *most frequent* contiguous length-d sub-product across all oversized terms, binds it with a zero-check, and substitutes it everywhere, so the number of commitments scales with the distinct high-frequency sub-products rather than the number of oversized terms. The full procedure, which outputs a polynomial V' whose terms each have at most d factors together with the zero-check set $\mathcal{Z}$, is given below.

1. Initialize $\mathcal{Z} \leftarrow \emptyset$ and $\mathsf{cache} \leftarrow \emptyset$.
2. While some term $(c,\, f_1 \cdots f_k)$ of V has $k > d$:
   (a) Let F be the multiset of contiguous length-d subsequences of the oversized terms.
   (b) Set $\chi \leftarrow \arg\max_{w \in F} \big(F[w],\ -\mathrm{lex}(w)\big)$ (most frequent, with lexicographic tie-break).
   (c) If $\chi \notin \mathsf{cache}$:
      i. Compute $C \leftarrow \mathsf{Commit}\big(\prod_{f \in \chi} f\big)$ and set $\mathsf{cache}[\chi] \leftarrow C$.
      ii. Update $\mathcal{Z} \leftarrow \mathcal{Z} \cup \big\{ \prod_{f \in \chi} f - C \big\}$.
   (d) Substitute $\mathsf{cache}[\chi]$ for contiguous occurrences of χ in the oversized terms of V.
3. Return $(V,\, \mathcal{Z})$.

E Benchmark Queries

We list below the exact SQL text of every TPC-H and synthetic query used in our evaluation.

E.1 TPC-H Benchmark Queries

The TPC-H benchmark [PF00] is a standard decision-support workload used widely in database design and systems research. It distills business-oriented analytics into a collection of parameterized SQL queries that stress scans, joins, aggregations, and nested subqueries. To ground our evaluation in a familiar setting, we follow prior work by selecting a subset of representative queries and executing them atop the canonical schema and data-generation procedures.

For completeness, we list the exact SQL text of the queries we benchmark. For the queries used in the Poneglyph[GFN25] comparison (Q1, Q3, Q5, Q8, Q9, Q18), we additionally list the abridged variant supported by Poneglyph's implementation, except for Q5, whose abridged form differs only in the table names.

We measure each query against the canonical TPC-H text (specifically, the version returned by DuckDB's `tpch_queries()`). A few modifications recur and are explained here once: (i) `extract(year from ...)` is replaced by a reference to a precomputed `*_year` column, because DataFusion does not yet plan `EXTRACT(YEAR FROM ...)`; (ii) decimal casts of the form `CAST(0.2 AS DECIMAL(15,2))` are added because our bench tables expose decimal-typed columns, so float literals require an explicit cast; (iii) some correlated subqueries are flattened to inner joins so that the benchmark exercises the same join gadget path used elsewhere, rather than planner-generated semi-joins with null right-side provenance. The abridged Poneglyph variants additionally rename every base table to `<name>_poneglyph` and drop trailing semicolons; only the further Poneglyph-specific abridgements are listed beneath each abridged query.

We run queries Q1, Q3, Q5, Q6, Q10, Q12, Q18, and Q19 exactly as specified by the canonical TPC-H text, so we omit their SQL here and refer the reader to the TPC-H specification [PF00]. Below we list only the queries whose text we modified, together with the abridged Poneglyph variants used in the Poneglyph comparison; the modifications for each are noted beneath its SQL.

Q1 (Poneglyph abridged): Pricing Summary Report

```
SELECT
    l_returnflag,
    sum(l_quantity) AS sum_qty,
    sum(l_extendedprice) AS sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
    count(*) AS count_order
FROM lineitem_poneglyph
WHERE l_shipdate <= CAST('1998-09-02' AS date)
GROUP BY l_returnflag
ORDER BY l_returnflag
```

Modifications: Drops `l_linestatus` from `SELECT`/`GROUP BY`/`ORDER BY`; drops all three `avg(...)` aggregates (`avg_qty`, `avg_price`, `avg_disc`).

Q2: Minimum Cost Supplier

```
SELECT s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone, s_comment
FROM part, supplier, partsupp, nation, region
WHERE p_partkey = ps_partkey
  AND s_suppkey = ps_suppkey
```

```
  AND p_size = 15
  AND s_nationkey = n_nationkey
  AND n_regionkey = r_regionkey
  AND r_name = 'EUROPE'
  AND ps_supplycost = (
      SELECT min(ps_supplycost)
      FROM partsupp, supplier, nation, region
      WHERE p_partkey = ps_partkey
        AND s_suppkey = ps_suppkey
        AND s_nationkey = n_nationkey
        AND n_regionkey = r_regionkey
        AND r_name = 'EUROPE')
ORDER BY s_acctbal DESC, n_name, s_name, p_partkey
LIMIT 100;
```

Modifications: Removes the `p_type LIKE '%BRASS'` filter.

Q3 (Poneglyph abridged): Shipping Priority

```
SELECT l_orderkey,
       sum(l_extendedprice * (1 - l_discount)) AS revenue,
       o_orderdate,
       o_shippriority
FROM customer_poneglyph, orders_poneglyph, lineitem_poneglyph
WHERE c_mktsegment = 'BUILDING'
  AND c_custkey = o_custkey
  AND l_orderkey = o_orderkey
  AND o_orderdate < CAST('1995-03-15' AS date)
  AND l_shipdate > CAST('1995-03-15' AS date)
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY revenue DESC, o_orderdate
```

Modifications: Removes `LIMIT 10`.

Q4: Order Priority Checking

```
SELECT o_orderpriority, count(*) AS order_count
FROM orders
INNER JOIN (
    SELECT l_orderkey
    FROM lineitem
    WHERE l_receiptdate > l_commitdate
    GROUP BY l_orderkey
) AS late_orders ON o_orderkey = l_orderkey
WHERE o_orderdate >= CAST('1993-07-01' AS date)
  AND o_orderdate < CAST('1993-10-01' AS date)
GROUP BY o_orderpriority
ORDER BY o_orderpriority;
```

Modifications: Replaces the correlated `EXISTS (SELECT * FROM lineitem WHERE l_orderkey = o_orderkey AND l_commitdate < l_receiptdate)` with an `INNER JOIN` against a derived table; functionally a semi-join → inner-join rewrite.

Q7: Volume Shipping

```
SELECT supp_nation, cust_nation, l_year, sum(volume) AS revenue
FROM (
    SELECT n1.n_name AS supp_nation,
           n2.n_name AS cust_nation,
           l_shipdate_year AS l_year,
           l_extendedprice * (1 - l_discount) AS volume
    FROM supplier, lineitem, orders, customer, nation n1, nation n2
    WHERE s_suppkey = l_suppkey
      AND o_orderkey = l_orderkey
      AND c_custkey = o_custkey
      AND s_nationkey = n1.n_nationkey
      AND c_nationkey = n2.n_nationkey
      AND ((n1.n_name = 'FRANCE' AND n2.n_name = 'GERMANY')
        OR (n1.n_name = 'GERMANY' AND n2.n_name = 'FRANCE'))
      AND l_shipdate BETWEEN CAST('1995-01-01' AS date)
                         AND CAST('1996-12-31' AS date)
) AS shipping
GROUP BY supp_nation, cust_nation, l_year
ORDER BY supp_nation, cust_nation, l_year;
```

Modifications: `extract(year FROM l_shipdate)` → reference to precomputed column `l_shipdate_year`.

Q8: National Market Share

```
SELECT o_orderdate_year,
       sum(CASE WHEN nation = 'BRAZIL' THEN volume ELSE 0 END) AS mkt_share_num,
       sum(volume) AS mkt_share_denom
FROM (
    SELECT o_orderdate_year,
           l_extendedprice * (1 - l_discount) AS volume,
           n2.n_name AS nation
    FROM part, supplier, lineitem, orders, customer, nation n1, nation n2, region
    WHERE p_partkey = l_partkey
      AND s_suppkey = l_suppkey
      AND l_orderkey = o_orderkey
      AND o_custkey = c_custkey
      AND c_nationkey = n1.n_nationkey
      AND n1.n_regionkey = r_regionkey
      AND r_name = 'AMERICA'
      AND s_nationkey = n2.n_nationkey
      AND o_orderdate BETWEEN CAST('1995-01-01' AS date)
                          AND CAST('1996-12-31' AS date)
      AND p_type = 'ECONOMY ANODIZED STEEL'
) AS all_nations
GROUP BY o_orderdate_year
ORDER BY o_orderdate_year;
```

Modifications: `extract(year FROM o_orderdate) AS o_year` → reference to precomputed column `o_orderdate_year` (output column renamed accordingly); the original `sum(CASE...) / sum(volume) AS mkt_share` ratio is split into two output columns (`mkt_share_num`, `mkt_share_denom`), so the engine no longer has to compute the division.

Q8 (Poneglyph abridged): National Market Share

```
SELECT o_orderdate_year,
       sum(CASE WHEN nation = 'BRAZIL' THEN volume ELSE 0 END) AS mkt_share_num,
       sum(volume) AS mkt_share_denom
FROM (
    SELECT o_orderdate_year,
           l_extendedprice * (1 - l_discount) AS volume,
           n2.n_name AS nation
    FROM part_poneglyph, supplier_poneglyph, lineitem_poneglyph,
         orders_poneglyph, customer_poneglyph,
         nation_poneglyph n1, nation_poneglyph n2, region_poneglyph
    WHERE p_partkey = l_partkey
      AND s_suppkey = l_suppkey
      AND l_orderkey = o_orderkey
      AND o_custkey = c_custkey
      AND c_nationkey = n1.n_nationkey
      AND n1.n_regionkey = r_regionkey
      AND r_name = 'AMERICA'
      AND s_nationkey = n2.n_nationkey
      AND o_orderdate BETWEEN CAST('1995-01-01' AS date)
                          AND CAST('1996-12-31' AS date)
      AND p_type = 'ECONOMY ANODIZED STEEL'
) AS all_nations
GROUP BY o_orderdate_year
ORDER BY o_orderdate_year;
```

Modifications: Same rewrites as Q8 above (precomputed `o_orderdate_year`; ratio split into `mkt_share_num` / `mkt_share_denom`).

Q9: Product Type Profit Measure

```
SELECT nation, o_orderdate_year, sum(amount) AS sum_profit
FROM (
    SELECT n_name AS nation,
           o_orderdate_year,
           l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity AS amount
    FROM part, supplier, lineitem, partsupp, orders, nation
    WHERE s_suppkey = l_suppkey
      AND ps_suppkey = l_suppkey
      AND ps_partkey = l_partkey
      AND p_partkey = l_partkey
      AND o_orderkey = l_orderkey
      AND s_nationkey = n_nationkey
) AS profit
GROUP BY nation, o_orderdate_year
ORDER BY nation, o_orderdate_year DESC;
```

Modifications: Removes the `p_name LIKE '%green%'` filter; `extract(year FROM o_orderdate) AS o_year` → reference to precomputed column `o_orderdate_year` (output column renamed accordingly).

Q9 (Poneglyph abridged): Product Type Profit Measure

```
SELECT nation, o_orderdate_year, sum(amount) AS sum_profit
FROM (
    SELECT n_name AS nation,
           o_orderdate_year,
           l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity AS amount
    FROM part_poneglyph, supplier_poneglyph, lineitem_poneglyph,
         partsupp_poneglyph, orders_poneglyph, nation_poneglyph
    WHERE s_suppkey = l_suppkey
      AND ps_suppkey = l_suppkey
      AND ps_partkey = l_partkey
      AND p_partkey = l_partkey
      AND o_orderkey = l_orderkey
      AND s_nationkey = n_nationkey
) AS profit
GROUP BY nation, o_orderdate_year
ORDER BY nation, o_orderdate_year DESC;
```

Modifications: Same rewrites as Q9 above (drop `p_name LIKE '%green%'`; precomputed `o_orderdate_year`).

Q14: Promotion Effect

```
SELECT CAST(100.00 AS DECIMAL(15, 2)) * sum(l_extendedprice * (1 - l_discount))
       / sum(l_extendedprice * (1 - l_discount)) AS promo_revenue
FROM lineitem, part
WHERE l_partkey = p_partkey
  AND l_shipdate >= date '1995-09-01'
  AND l_shipdate < CAST('1995-10-01' AS date);
```

Modifications: Drops the `CASE WHEN p_type LIKE 'PROMO%' THEN ... ELSE 0 END` numerator entirely (so the `LIKE` filter is removed and the ratio becomes degenerate, with `sum(l_extendedprice * (1 - l_discount))` in both numerator and denominator); `100.00 *` → `CAST(100.00 AS DECIMAL(15, 2)) *`.

Q15: Top Supplier

```
WITH revenue AS (
    SELECT l_suppkey AS supplier_no,
           sum(l_extendedprice * (1 - l_discount)) AS total_revenue
    FROM lineitem
    WHERE l_shipdate >= CAST('1996-01-01' AS date)
      AND l_shipdate < CAST('1996-04-01' AS date)
    GROUP BY supplier_no
),
max_revenue AS (
    SELECT max(total_revenue) AS highest_val FROM revenue
)
```

```
SELECT s_suppkey, s_name, s_address, s_phone, total_revenue
FROM supplier
INNER JOIN revenue     ON s_suppkey = supplier_no
INNER JOIN max_revenue ON total_revenue = highest_val
ORDER BY s_suppkey;
```

Modifications: Adds a second CTE `max_revenue`; the canonical version's cross-join with a `WHERE`-equality and inline scalar subquery (`SELECT max(total_revenue) FROM revenue`) is replaced with two explicit `INNER JOIN`s (`ON s_suppkey = supplier_no` and `ON total_revenue = highest_val`).

Q17: Small-Quantity-Order Revenue

```
SELECT sum(l_extendedprice) / CAST(7.0 AS DECIMAL(15, 2)) AS avg_yearly
FROM lineitem, part
WHERE p_partkey = l_partkey
  AND p_brand = 'Brand#23'
  AND p_container = 'MED BOX'
  AND l_quantity < (
      SELECT CAST(0.2 AS DECIMAL(15, 2)) * avg(l_quantity)
      FROM lineitem
      WHERE l_partkey = p_partkey);
```

Modifications: Decimal casts on the literals `7.0` and `0.2` (`CAST(... AS DECIMAL(15, 2))`); otherwise structurally identical (the correlated subquery is preserved).

Q18 (Poneglyph abridged): Large Volume Customer

```
SELECT c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice,
       SUM(l_quantity) AS sum_l_quantity
FROM customer_poneglyph
JOIN orders_poneglyph   ON c_custkey = o_custkey
JOIN lineitem_poneglyph ON o_orderkey = l_orderkey
GROUP BY c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice
ORDER BY o_totalprice DESC, o_orderdate ASC;
```

Modifications: Drops the `o_orderkey IN (SELECT l_orderkey ... HAVING sum(l_quantity) > 300)` pre-filter entirely (this is the substantive abridgement: there is no big-order pre-filter, and the `GROUP BY` now sees all orders); cross joins rewritten to explicit `JOIN ... ON` syntax; drops `LIMIT 100`; adds `sum_l_quantity` alias on `SUM(l_quantity)`.

Q20: Potential Part Promotion

```
SELECT s_name, s_address
FROM supplier
INNER JOIN nation ON s_nationkey = n_nationkey
INNER JOIN (
    SELECT ps_suppkey
    FROM partsupp
    INNER JOIN part ON ps_partkey = p_partkey
    INNER JOIN (
```

```
        SELECT l_partkey, l_suppkey,
               CAST(0.5 AS DECIMAL(15, 2)) * sum(l_quantity) AS half_sum_qty
        FROM lineitem
        WHERE l_shipdate >= CAST('1994-01-01' AS date)
          AND l_shipdate < CAST('1995-01-01' AS date)
        GROUP BY l_partkey, l_suppkey
    ) AS lineitem_half_sum
        ON ps_partkey = l_partkey
       AND ps_suppkey = l_suppkey
    WHERE ps_availqty > half_sum_qty
    GROUP BY ps_suppkey
) AS candidate_suppliers ON s_suppkey = ps_suppkey
WHERE n_name = 'CANADA'
ORDER BY s_name;
```

Modifications: Flattens the canonical triple-nested correlated subquery (`s_suppkey IN (SELECT ps_suppkey FROM partsupp WHERE ps_partkey IN (...) AND ps_availqty > (SELECT 0.5 * sum(l_quantity) FROM lineitem WHERE ...))`) into all `INNER JOIN`s, with the `lineitem` aggregate pulled out into a derived table joined on `(l_partkey, l_suppkey)`; removes the `p_name LIKE 'forest%'` filter (`part` is joined unfiltered); `0.5 *` → `CAST(0.5 AS DECIMAL(15, 2)) *`.

Execution notes. The queries we benchmark span scan-heavy aggregates (Q1, Q6), nested-subquery selections (Q2, Q17), join-and-sort pipelines (Q3, Q10), order-priority and shipping-mode analytics (Q4, Q12), geography-aware joins (Q5, Q7), nested aggregation queries (Q8), profit-attribution workloads (Q9), single-pass arithmetic aggregates (Q14), CTE-based top-supplier ranking (Q15), large-volume customer reporting (Q18), highly selective disjunctive predicates (Q19), and supplier-screening pipelines (Q20). Collectively, they exercise every proving gadget in TruthTable, providing a realistic baseline for system-wide tuning.

E.2 Synthetic Benchmark Queries

To compare TruthTable on workloads where PoSQL [Spa25] and qedb [BBCRT25] baselines are applicable, we benchmark a set of synthetic queries that exercise the core relational operators in isolation: Filter, Aggregate, Join, Join (PK/FK), and Limit/Offset.

Filter

```
SELECT * FROM {table} WHERE a = 1 AND b = 2;
```

Aggregate

```
SELECT count(*) FROM {table};
```

Join

```
SELECT {table1}.a, {table2}.a FROM {table1} JOIN {table2} ON {table1}.a = {table2}.a;
```

Join (PK/FK)

```
SELECT {table1}.a, {table2}.a FROM {table1} JOIN {table2} ON {table1}.a = {table2}.a;
```

Limit/Offset

```
SELECT * FROM {table} LIMIT 10;
```