vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases

Yupeng Zhang*, Daniel Genkin†,*, Jonathan Katz*, Dimitrios Papadopoulos‡,* and Charalampos Papamanthou*
*University of Maryland †University of Pennsylvania ‡Hong Kong University of Science and Technology
Email: {zhangyp,cpap}@umd.edu, danielg3@cis.upenn.edu, jkatz@cs.umd.edu, dipapado@cse.ust.hk

***Abstract*—Cloud database systems such as Amazon RDS or Google Cloud SQL enable the outsourcing of a large database to a server who then responds to SQL queries. A natural problem here is to efficiently verify the correctness of responses returned by the (untrusted) server. In this paper we present vSQL, a novel cryptographic protocol for publicly verifiable SQL queries on dynamic databases. At a high level, our construction relies on two extensions of the CMT interactive-proof protocol [Cormode et al., 2012]: (i) supporting outsourced input via the use of a polynomial-delegation protocol with succinct proofs, and (ii) supporting auxiliary input (i.e., non-deterministic computation) efficiently. Compared to previous verifiable-computation systems based on interactive proofs, our construction has verification cost polylogarithmic in the auxiliary input (which for SQL queries can be as large as the database) rather than linear.**

In order to evaluate the performance and expressiveness of our scheme, we tested it on SQL queries based on the TPC-H benchmark on a database with 6×10^6 rows and 13 columns. The server overhead in our scheme (which is typically the main bottleneck) is up to 120× lower than previous approaches based on succinct arguments of knowledge (SNARKs), and moreover we avoid the need for query-dependent pre-processing which is required by optimized SNARK-based schemes. In our construction, the server/client time and the communication cost are comparable to, and sometimes smaller than, those of existing customized solutions which only support specific queries.

I. Introduction

All major cloud providers offer Database-as-a-Service solutions that allow companies and individuals (*clients*) to alleviate storage costs and achieve resource elasticity by delegating storage and maintenance of their data to a cloud server. A client can then query and/or update its data using, e.g., standard SQL queries. Outsourcing data in this way, however, introduces new security challenges: in particular, the client may need to ensure the *integrity* of the results returned by the server. Providing such a guarantee is important if the client does not trust the server, or even if the client is concerned about the possibility of server errors or external compromise.

Prior works[1] on *verifiable computation/outsourcing* and *authenticated data structures* address exactly this problem (see Section I-D for a detailed discussion), but have significant drawbacks. Generic solutions (e.g., SNARKs) can be used to verify arbitrary computations, but impose an unacceptable overhead at the server. Function-specific schemes (e.g., authenticated data structures) target specific classes of computations and can be much more efficient than generic solutions; however, they suffer from limited expressiveness, and in particular they cannot handle a wide range of SQL queries.

A. Our Results

In this work we present vSQL, a system for verifiable SQL queries. vSQL allows a client who owns a relational database to outsource it to an untrusted server while storing only a small *digest* locally. Later, the client can issue arbitrary SQL queries to the server, who returns the query's result. (In the case of an update query, the result is an updated digest.) The client can then verify the validity of the result using an interactive protocol with the server; if the result returned by the server is incorrect, the client will reject with overwhelming probability.

vSQL overcomes the drawbacks of existing works. It is highly expressive, supporting any computation expressed as an arithmetic circuit (which in particular means arbitrary SQL queries, including updates) efficiently. We empirically demonstrate vSQL's concrete performance and expressiveness using the TPC-H [7] benchmark, and find that the server-side computation (which is usually the limiting factor in verifiable-computation schemes) is 5–120× better for vSQL than it is in highly optimized libsnark-based constructions [4] (that further require query-dependent preprocessing), and comparable to or better than a state-of-the-art database-delegation scheme [60] that only supports a limited subset of SQL.

At a high level, vSQL gains efficiency by combining two different approaches. First, vSQL uses a highly efficient information-theoretic interactive proof system [25] for delegating computations expressed as arithmetic circuits. This reduces the number of cryptographic operations performed by the server from linear *in the circuit size* to linear *in the lengths of the circuit's inputs and outputs*. Second, vSQL supports non-deterministic computation by allowing the server to provide auxiliary inputs to the circuit. We achieve this by using a novel scheme for verifiable polynomial delegation with knowledge-extraction properties, improving previous results [56]. To the best of our knowledge, our protocol is the first implementation of an argument system that supports general non-deterministic computations and simultaneously achieves the following two properties: (1) the number of cryptographic operations performed by the prover depends linearly on the circuit's input and output size, and (2) the total prover time is quasilinear in the size of the evaluated circuit. In addition, vSQL benefits from several performance optimizations that improve both the server's and client's concrete efficiency (see Section VI). We describe our techniques in further detail next.

[1] We assume no trusted hardware, nor are we willing to assume multiple, non-colluding servers (as required by [22]).

B. Our Techniques

Reducing Cryptographic Overhead. A key building block of vSQL is the information-theoretic interactive proof system due to Cormode et al. [25] (the *CMT protocol*) that allows a client to verify that $y = C(x)$ for some circuit C and input x known to the client. It is natural in our setting to let x be the client's data, and to let C be a circuit corresponding to the client's query. While this is a good starting point for reducing the number of cryptographic operations (since the CMT protocol is information-theoretic), in our setting we cannot directly apply the CMT protocol since our goal is to avoid requiring the client to store its data x.

Supporting Delegated Inputs. We observe that at the last step of the CMT protocol it is sufficient for the client to be able to evaluate a certain multivariate polynomial p_x that depends on x (but not on C) at a random point. We develop a new *polynomial-delegation scheme* that allows the client to outsource p_x itself; i.e., it enables the client to store a short digest com_x of p_x that can be used to verify claimed results about the evaluation of p_x on points of the client's choice. This allows the client to delegate its input to the untrusted server while still being able to execute the CMT verifier.

Supporting Auxiliary Inputs. In order to leverage the efficiency improvements provided by non-determinism, we extend the CMT protocol to support server-provided *auxiliary inputs*. (This allows the server to prove that there exists w such that $y = C(x, w)$.) A straightforward approach for achieving this [56] is to have the server provide the entire auxiliary input to the client. However, we show that when evaluating SQL queries it is possible that the auxiliary input that is necessary in order to reduce the size of the circuit being evaluated is as large as the database itself; in that case, the naive approach of sending the auxiliary input to the client is inefficient. Instead, we have the server provide a succinct commitment to the auxiliary input which will be used by the CMT verifier. To instantiate this approach efficiently, we use the polynomial-delegation scheme described above, augmented with a knowledge-extraction property (which is required for extracting a cheating server's auxiliary input).

Supporting Efficient Updates. We also build on the above techniques to handle arbitrary updates efficiently, something that is notoriously hard for previous approaches to verifiable computation. Say the client wants to apply an update given by a circuit C, and let $x' = C(x)$. Obviously, having the server send the result x' to the client is impractical. To avoid this, we first observe that the client need not learn x' in order to verify future queries about x'; rather, it is sufficient for the client to learn $\mathsf{com}_{x'}$. At this point, we could just extend C to a circuit C' that also includes the computation of $\mathsf{com}_{x'}$ and handle updates the same way we handle queries but this would impose an additional cost as it would result in a increased circuit size. Instead, we further observe that this is not necessary: in order to run the CMT verifier for circuit C, the client need not know the output (x' in this case) explicitly; as with the input, it is enough to be able to evaluate the corresponding multivariate polynomial $p_{x'}$ at a random point. For this, we can again use our polynomial-delegation scheme to have the server first provide the commitment $\mathsf{com}_{x'}$ to the client and then provably evaluate $p_{x'}$ at a point chosen by the latter, which is sufficient to prove that $\mathsf{com}_{x'}$ is the new digest.

Eliminating Interaction. Our approach uses the CMT protocol and thus inherits its need for interaction. While this is not a major barrier (as demonstrated by our experimental evaluation in Section VII), we remark that interaction can be avoided using the Fiat-Shamir approach in the random-oracle model.

C. Outline

In Section II we establish notation and introduce the necessary background. Section III presents our verifiable polynomial-delegation protocol, which is used as a building block for our main construction. We introduce our security definition in Section IV, and the vSQL construction satisfying this definition in Section V. We discuss a series of optimizations that improve the performance of our construction in Section VI. In Section VII we provide a detailed experimental evaluation of our system. We conclude in Section VIII.

D. Related Work

Verifiable outsourcing of data has been studied from multiple perspectives, and under various names. Work on authenticated data structures typically focuses on handling only a specific class of computations on outsourced data, e.g., range queries [43], [47], joins [59], [62], [30], pattern matching [27], [48], set operations [50], [21], [41], polynomial evaluation [9], graph queries [35], [61], and search problems [45], [46]. The most relevant point of comparison to our work is IntegriDB [60], which supports a subset of SQL. In Section VII, we show that vSQL is significantly more expressive than IntegriDB while enjoying comparable efficiency.

Arbitrary computations on outsourced data can be handled by schemes for verifiable delegation of computation [31], [24]. State-of-the-art systems [10], [51], [26], [52], [57] can handle arbitrary non-deterministic arithmetic circuits by relying on succinct arguments of knowledge (SNARKs) [14], [32]. SNARKs provide an efficiently verifiable, constant-size proof for the correct evaluation of a circuit. The major disadvantage of SNARK-based approaches is the extremely high prover time they currently impose (cf. Section VII). In addition, the fastest existing implementations of SNARKs assume a circuit-specific preprocessing step, something that is not practical (and may be impossible) in a scenario where multiple queries that cannot be predicted in advance will be made on a given database. Finally, we remark that the systems mentioned above are all "natively" designed to support verification only when the input is known to the client. Support for outsourced data can be handled by having the client compute a succinct hash of its data, and then verifying the hash computation along with verification of the result. However, this adds additional overhead as the hash computation needs to either be computed as part of the arithmetic circuit [19], or checked by an external mechanism [8], [28]. Alternatively, one could hard-code the database into the circuit being evaluated, but then the circuit-specific preprocessing needs to be executed after each database update. In Section VII, we show that vSQL has significantly better prover time than SNARK-based approaches.

While some other works also aim at verifying arbitrary computations over remotely stored data [23], [17], [38], [20], these approaches are only of theoretical interest at this point.

Interactive proofs were introduced by Goldwasser et al. [34], and have been studied extensively in complexity theory. More recently, prover-efficient interactive proofs for log-depth circuits were introduced in [33]. Subsequent works have optimized and implemented this protocol, demonstrating the potential of interactive proofs for practical verifiable computation [25], [54], [56].

II. Preliminaries

A. SQL Queries

Structured Query Language (SQL) is a very popular programming language designed for querying and managing relational database systems. It operates on databases that consist of collections of two-dimensional matrices called *tables*. In the following, we briefly present the general structure of such queries and provide concrete examples for common types.

In SQL, a simple query begins with the keyword `SELECT` followed by a function $A(col_1, \ldots)$ and then the keyword `FROM` followed by a number of tables, where A is defined over (a subset of) the columns of the specified tables. This sequence of clauses and expressions dictates the output of the query. Following these, there is a `WHERE` clause followed by a sequence of predicates connected by logical operators (e.g, `AND`, `OR`, `NOT`) that restrict the rows used when computing the output. The above is best illustrated by a series of examples. Consider a database consisting of tables $\mathbf{T_1}$ and $\mathbf{T_2}$:

$\mathbf{T_1}$:

row_id	employee_id	name	age	salary
1	2019	John	28	45,000
2	1905	Kate	31	55,000
3	1908	Lisa	44	70,000
4	2117	Leo	23	39,000
5	2003	Alice	29	34,000

$\mathbf{T_2}$:

row_id	employee_id	department
1	1905	Sales
2	1906	Sales
3	1908	HR
4	2003	R&D
5	2022	HR
6	2117	R&D

The first example we provide is a SQL range query which is used to select rows for which particular values fall within a set of specified ranges. The conditions may be defined over multiple columns, in which case we refer to it as a *multi-dimensional range query*. For example, the query "`SELECT * FROM` $\mathbf{T_1}$ `WHERE` age < 35 `AND` salary $> 40,000$" is a two-dimensional range query that returns the following table.

row_id	employee_id	name	age	salary
1	2019	John	28	45,000
2	1905	Kate	31	55,000

A `FROM` clause can be followed by `JOIN` sub-clauses that are used to combine multiple tables based on common values in specific columns. An example of such a `JOIN` query is "`SELECT` $\mathbf{T_1}$.name, $\mathbf{T_2}$.department `FROM` $\mathbf{T_1}$ `JOIN` $\mathbf{T_2}$ `ON` $\mathbf{T_1}$.employee_id = $\mathbf{T_2}$.employee_id," which returns:

name	department
Kate	Sales
Lisa	HR
Alice	R&D

The result of any SQL query is itself a table to which another SQL query can be applied. In other words, a SQL query may be composed of several sub-queries. SQL also provides queries for adding, updating, and deleting data from a SQL database. Data-manipulation queries start with an `INSERT`, `DELETE`, or `UPDATE` clause followed by a table identifier, a series of values, and (optionally) a sequence of `WHERE` clauses. For example, the query "`DELETE FROM` $\mathbf{T_2}$ `WHERE` department = Sales" deletes the first two rows from $\mathbf{T_2}$. Finally, there are queries that manipulate the database structure, e.g, by adding new columns or creating a new table.

Note that a common theme of the examples presented above is that they process each row of some table independently, performing a specific operation (e.g., comparing values from given columns with a specified range) on each row. In Section V, we discuss how this structure can be leveraged to improve efficiency in our setting.

B. Interactive Proofs

An interactive proof [34] is a protocol that allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ of the validity of some statement. We phrase this in terms of $\mathcal{P}$ trying to convince $\mathcal{V}$ that $f(x) = 1$, where f is fixed and x is the common input. Of course, an interactive proof in this sense is only interesting if the running time of $\mathcal{V}$ is less than the time to compute f.

Definition 1. *Let f be a boolean function. A pair of interactive algorithms $(\mathcal{P}, \mathcal{V})$ is an* interactive proof *for f with soundness ϵ if the following holds.*

- ***Completeness.*** *For every x such that $f(x) = 1$ it holds that* $\Pr[\langle\mathcal{P}, \mathcal{V}\rangle(x) = \mathsf{accept}] = 1$.
- ***ϵ-Soundness.*** *For any x with $f(x) \neq 1$ and any $\mathcal{P}^*$ it holds that* $\Pr[\langle\mathcal{P}^*, \mathcal{V}\rangle(x) = \mathsf{accept}] \leq \epsilon$.

Note that the above can be easily extended to prove that $g(x) = y$ (where x, y are common input) by considering the function f defined as $f(x, y) = 1$ iff $g(x) = y$.

The Sum-Check Protocol. A fundamental interactive protocol that serves as an important building block for our work is the *sum-check protocol* [44]. Here, the common input of the prover and verifier is an ℓ-variate polynomial $g(x_1, \ldots, x_\ell)$ over a field $\mathbb{F}$; the prover's goal is to convince the verifier that

$$H = \sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_\ell \in \{0,1\}} g(b_1, b_2, \ldots, b_\ell)\,.$$

Note that direct computation of H by $\mathcal{V}$ requires at least 2^ℓ work. Using the sum-check protocol, the verifier's computation is exponentially smaller. The protocol proceeds in ℓ rounds, as follows. In the first round, the prover sends the univariate polynomial $g_1(x_1) \overset{\text{def}}{=} \sum_{b_2,\ldots,b_\ell \in \{0,1\}} g(x_1, b_2, \ldots, b_\ell)$; the verifier checks that the degree of g_1 is at most the degree of x_1 in g, and that $H = g_1(0) + g_1(1)$; it rejects if these do not hold. Next, $\mathcal{V}$ sends a uniform challenge $r_1 \in \mathbb{F}$. In the ith round $\mathcal{P}$ sends the polynomial $g_i(x_i) \overset{\text{def}}{=} \sum_{b_{i+1},\ldots,b_\ell \in \{0,1\}} g(r_1, \ldots, r_{i-1}, x_i, b_{i+1}, \ldots, b_\ell)$. The verifier checks the degree of g_i and verifies that $g_{i-1}(r_{i-1}) = g_i(0) + g_i(1)$; if so, it sends a uniform $r_i \in \mathbb{F}$ to the prover. After the final round, $\mathcal{V}$ accepts only if $g(r_1, \ldots, r_\ell) = g_\ell(r_\ell)$.

The degree of each monomial in g is the sum of the powers of its variables; the total degree of g is the maximum degree of any of its monomials. We have [44]:

Theorem 1. *For any ℓ-variate, degree-d polynomial g over $\mathbb{F}$, the sum-check protocol is an interactive proof for the (no-input) function $\sum_{b_1 \in \{0,1\}} \cdots \sum_{b_\ell \in \{0,1\}} g(b_1, \ldots, b_\ell)$ with soundness $d \cdot \ell / |\mathbb{F}|$.*

C. Multilinear Extensions

Let $V : \{0,1\}^\ell \to \mathbb{F}$ be a function. Then there exists a unique ℓ-variate polynomial $\tilde{V} : \mathbb{F}^\ell \to \mathbb{F}$, called the *multilinear extension of* V, with the properties that (1) $\tilde{V}$ has degree at most 1 in each variable and (2) $\tilde{V}(x) = V(x)$ for all $x \in \{0,1\}^\ell$. Note that $\tilde{V}$ can be defined as

$$\tilde{V}(x_1, \ldots, x_\ell) = \sum_{b \in \{0,1\}^\ell} \prod_{i=1}^{\ell} \mathcal{X}_{b_i}(x_i) \cdot V(b), \tag{1}$$

where b_i is the ith bit of b, and we set $\mathcal{X}_1(x_i) = x_i$ and $\mathcal{X}_0(x) = 1 - x_i$.

Multilinear Extensions of Arrays. An array $A = (a_0, \ldots, a_{n-1}) \in \mathbb{F}^n$, where for simplicity we assume n is a power of 2, can be viewed as the function $A : \{0,1\}^{\log n} \to \mathbb{F}$ such that $A(i) = a_i$ for $0 \leq i \leq n-1$ (where i is expressed in binary). In the sequel, abuse this terminology and use a multilinear extension $\tilde{A}$ of an array A. A useful property of multilinear extensions of arrays is the ability to efficiently combine two multilinear extensions. That is, let A_1, A_2 be two equal-length arrays, and let $A = A_1 || A_2$ be their concatenation. Then $\tilde{A}(x_1, \ldots, x_{\log n+1})$ can be computed as $(1 - x_1)\tilde{A}_1(x_2, \ldots, x_{\log n+1}) + x_1 \tilde{A}_2(x_2, \ldots, x_{\log n+1})$, i.e., as a linear combination of the evaluations of the multilinear extensions of the smaller arrays. More generally, in the case of m equal-length arrays $A_0, \ldots, A_{m-1}$ (where m is a power of 2) the multilinear extension of $\tilde{A} = A_0 || \ldots || A_{m-1}$ can be evaluated on a point $(x_1, \ldots, x_{\log(nm)})$ as

$$\sum_{i=0}^{m-1} \prod_{j=1}^{\log m} \mathcal{X}_{i_j}(x_j) \tilde{A}_i(x_{\log m+1}, \ldots, x_{\log(nm)}) \tag{2}$$

where i_j is the jth bit of i and $\mathcal{X}_{i_j}(x_j)$ is defined as above.

D. The CMT Protocol

Cormode et al. [25], building on work of Goldwasser et al. [33], show an efficient interactive proof (that we call the *CMT protocol*) for a certain class of functions. The CMT protocol serves as the starting point for our scheme.

High-Level Overview. Let C be a depth-d arithmetic circuit over a finite field $\mathbb{F}$ that is *layered*, i.e., for which each gate of C is associated with a layer, and the output wire from a gate at layer i can only be an input wire to a gate at level $i - 1$. The CMT protocol processes the circuit one layer at a time, starting from layer 0 (that contains the output wires) and ending at layer d (that contains the input wires). The prover $\mathcal{P}$ starts by proposing a value y for the output of the circuit on input x. Then, in the ith round, $\mathcal{P}$ reduces a claim (i.e., an algebraic statement) about the values of the wires in layer i to a claim about the values of the wires in layer $i + 1$. The protocol terminates with a claim about the wire values at layer d (i.e., the input wires) that can be checked directly by the verifier $\mathcal{V}$ who knows the input x. If that check succeeds, then $\mathcal{V}$ accepts.

Notation. Before describing the protocol more formally we introduce some additional notation. Let S_i be the number of gates in the ith layer and set $s_i = \lceil \log S_i \rceil$ so s_i bits suffice to identify each gate at the ith layer. The evaluation of C on an input x assigns in a natural way a value in $\mathbb{F}$ to each gate in the circuit. Thus, for each layer i we can define a function $V_i : \{0,1\}^{s_i} \to \mathbb{F}$ that takes as input a gate g and returns its value (and returns 0 if g does not correspond to a valid gate). Using this notation, V_d corresponds to the input of the circuit, i.e., x. Finally, we define for each layer i two boolean functions $\mathsf{add}_i, \mathsf{mult}_i$, which we refer to as *wiring predicates*, as follows: $\mathsf{add}_i : \{0,1\}^{s_{i-1}+2s_i} \to \{0,1\}$ takes as input three gates g_1, g_2, g_3, where g_1 is at layer $i - 1$ and g_2, g_3 are at layer i, and returns 1 if and only if g_1 is an addition gate whose input wires are the output wires of gates g_2 and g_3. (We define mult_i for multiplication gates analogously.) The value of a gate g at layer $i < d$ can thus be recursively computed as

$$V_i(g) = \sum_{u,v \in \{0,1\}^{s_{i+1}}} \Big(\mathsf{add}_{i+1}(g, u, v) \cdot (V_{i+1}(u) + V_{i+1}(v)) + \mathsf{mult}_{i+1}(g, u, v) \cdot (V_{i+1}(u) \cdot V_{i+1}(v)) \Big).$$

Protocol Details. One idea is for $\mathcal{V}$ to verify that $y = C(x)$ by checking that $V_i(g)$ is computed correctly for each gate g in each layer i. Since $V_i(g)$ can be expressed as a summation, this could be done using the sum-check protocol from Section II-B. However, the sum-check protocol operates on polynomials defined over $\mathbb{F}$ and therefore we need to replace terms with their multilinear extensions. That is:

$$\begin{aligned} \tilde{V}_i(z) = & \sum_{\substack{g \in \{0,1\}^{s_i} \\ u,v \in \{0,1\}^{s_{i+1}}}} f_{i,z}(g, u, v) \qquad (3) \\ \overset{\text{def}}{=} & \sum_{\substack{g \in \{0,1\}^{s_i} \\ u,v \in \{0,1\}^{s_{i+1}}}} \tilde{\beta}_i(z, g) \cdot \Big(\tilde{\mathsf{add}}_{i+1}(g, u, v) \cdot (\tilde{V}_{i+1}(u) \\ & + \tilde{V}_{i+1}(v)) + \tilde{\mathsf{mult}}_{i+1}(g, u, v) \cdot (\tilde{V}_{i+1}(u) \cdot \tilde{V}_{i+1}(v)) \Big), \end{aligned}$$

where $\tilde{\mathsf{add}}_i$ (resp., $\tilde{\mathsf{mult}}_i$) is the multilinear extension of add_i (resp., mult_i) and $\tilde{\beta}_i$ is the multilinear extension of the selector function that takes two s_i-bit inputs a, b and outputs 1 if $a = b$

Construction 1 (CMT protocol). *Let $\mathbb{F}$ be a prime-order field, and let $C : \mathbb{F}^n \to \mathbb{F}^k$ be a depth-d layered arithmetic circuit. $\mathcal{P}$ and $\mathcal{V}$ hold x, y, and $\mathcal{P}$ wants to convince $\mathcal{V}$ that $y = C(x)$. To do so:*

1) *Let $V_0 : \{0,1\}^{\lceil \log k \rceil} \to \mathbb{F}$ be such that $V_0(j)$ equals the jth element of y. Verifier $\mathcal{V}$ chooses uniform $r_0 \in \mathbb{F}^{\lceil \log k \rceil}$ and sends it to $\mathcal{P}$. Both parties set $a_0 = \tilde{V}_0(r_0)$.*
2) *For $i = 1, \ldots, d$:*
   a) *$\mathcal{P}$ and $\mathcal{V}$ run the sum-check protocol for value a_{i-1} and polynomial $f_{i-1,r_{i-1}}$ as per Equation (3). In the last step of that protocol, $\mathcal{P}$ provides (v_1, v_2) for which it claims $v_1 = \tilde{V}_i(q_1)$ and $v_2 = \tilde{V}_i(q_2)$.*
   b) *Let $\gamma : \mathbb{F} \to \mathbb{F}^{s_i}$ be the line with $\gamma(0) = q_1$ and $\gamma(1) = q_2$. Then $\mathcal{P}$ sends the degree-s_i polynomial $h(x) = \tilde{V}_i(\gamma(x))$. Next, $\mathcal{V}$ verifies that $h(0) = v_1$ and $h(1) = v_2$, and rejects if not. Then $\mathcal{V}$ chooses uniformly at random $r'_i \in \mathbb{F}$, sets $r_i = \gamma(r'_i)$, $a_i = h(r_i)$ and sends them to $\mathcal{P}$.*
3) *$\mathcal{V}$ accepts iff $a_d = \tilde{V}_d(r_d)$, where $\tilde{V}_d$ is the multilinear extension of the polynomial representing the input x.*

and 0 otherwise.[2] However, this approach would incur a cost to the verifier larger than the cost of evaluating C, as it requires one execution of the sum-check protocol per gate.

Instead, by leveraging the recursive form of $\tilde{V}_i$, correctness of the circuit evaluation can be checked with a single execution of the sum-check protocol for each layer i, as follows. Assume for simplicity that the output of the circuit is a single value. The interaction begins at level 0, with the prover claiming that $y = \tilde{V}_0(0)$ (i.e., the circuit's output) for some value y. The two parties then execute the sum-check protocol for the polynomial $f_{0,0}$ in order to check this claim. Recall that, at the end of this execution, $\mathcal{V}$ is supposed to evaluate $f_{0,0}$ at a random point $\rho \in \mathbb{F}^{s_0+2s_1}$ (the randomness generated by the sum-check verifier). Since $f_{0,0}$ depends on $\tilde{V}_1(u)$ and $\tilde{V}_1(v)$, in this case $\mathcal{V}$ has to evaluate $\tilde{V}_1$ on the random points $q_1, q_2 \in \mathbb{F}^{s_1}$ where q_2 consists of the last s_1 entries of ρ, and q_1 from the previous s_1 ones. If the verifier had access to all the correct gate values at layer 1, he could compute these evaluations himself. Since he does not, however, he must rely on the prover to provide him with these evaluations, say v_1, v_2. This effectively reduces the validity of the original claim that $y = \tilde{V}_0(0)$ to the validity of the two claims that $\tilde{V}_1(q_1) = v_1$ and $\tilde{V}_1(q_2) = v_2$. The two parties can now execute the sum-check protocol for these two claims. By repeatedly applying this idea, the final claim by the prover will be stated with respect to $\tilde{V}_d$ (i.e., the multilinear extension of the circuit's input), which can be checked locally by the verifier who has the input x.

Unfortunately, this approach still potentially requires 2^d executions of the sum-check protocol, since the number of claims being verified doubles with each level.

Condensing to a Single Evaluation Per Layer. Efficiency can be improved by reducing the proof that $v_1 = \tilde{V}_1(q_1)$ and $v_2 = \tilde{V}_1(q_2)$ to a *single* sum-check execution, as follows. Let $\gamma : \mathbb{F} \to \mathbb{F}^{s_1}$ be the unique line with $\gamma(0) = q_1$ and $\gamma(1) = q_2$. The prover sends a degree-s_1 polynomial h that is supposed to be $\tilde{V}_1(\gamma(x))$, i.e., the restriction of $\tilde{V}_1$ to the line γ. The verifier checks that $h(0) = v_1$ and $h(1) = v_2$, and then picks a new random point $r'_1 \in \mathbb{F}$ and initiates a *single* invocation of the sum-check protocol to verify that $\tilde{V}_1(\gamma(r'_1)) = h(r'_1)$. Proceeding in this way, it is possible to obtain a protocol that uses only $O(d)$ executions of the sum-check protocol.

We assumed so far that there is a single output value y. Larger outputs can be handled efficiently [56] by adapting the above approach so that the initial claim by the prover is stated directly about the multilinear extension of the claimed output.

The CMT protocol is formally described in Construction 1.[3]

Theorem 2 ([33], [25], [56], [54])**.** *Let $C : \mathbb{F}^n \to \mathbb{F}^k$ be a depth-d layered arithmetic circuit. Construction 1 is an interactive proof for the function computed by C with soundness $O(d \cdot \log S/|\mathbb{F}|)$, where S is the maximal number of gates per circuit layer. It uses $O(d \log S)$ rounds of interaction, and the running time of $\mathcal{P}$ is $O(|C| \log S)$. If $\tilde{\mathsf{add}}_i$ and $\tilde{\mathsf{mult}}_i$ are computable in time $O(\text{polylog}\, S)$ for all layers $i \leq d$, then the running time of the verifier $\mathcal{V}$ is $O(n + k + d \cdot \text{polylog}\, S)$.*

The following remark will be particularly useful for us in the context of evaluating circuits representing SQL queries.

Remark 1 ([54])**.** *If C can be expressed as a composition of (i) parallel copies of a layered circuit C' whose maximum number of gates at any layer is S', and (ii) a subsequent "aggregation" layered circuit C'' of size $O(|C|/\log |C|)$, the running time of $\mathcal{P}$ is reduced to $O(|C| \log |S'|)$.*[4]

E. Bilinear Pairings

We denote by $\mathsf{bp} := (p, \mathbb{G}, \mathbb{G}_T, e, g) \leftarrow \mathsf{BilGen}(1^\lambda)$ the generation of parameters for a bilinear map,[5] where λ is the security parameter, $\mathbb{G}, \mathbb{G}_T$ are two groups of order p (with p a λ-bit prime), $g \in \mathbb{G}$ is a generator, and $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ is a bilinear map. In Appendix A we present the assumptions necessary for proving the security of our scheme.

III. Verifiable Polynomial Delegation

As a key part of our main construction, we use a new scheme for *verifiable polynomial delegation* that allows a client to outsource storage of a multivariate polynomial f to a server while retaining only a short commitment com. The server can

[2] Although using $\tilde{\beta}$ is not strictly necessary [55], we use it in our implementation because it improves efficiency when C is composed of many parallel copies of a smaller circuit C' (as is the case for standard SQL queries).

[3] Throughout the paper, when reporting asymptotic complexities we omit a factor that is polylogarithmic in the field/blinear group size, implicitly assuming all operations take constant time.

[4] Note that the bound on the running time of $\mathcal{V}$ can also be improved if one makes stronger assumptions about the "regularity" of C''. Many common SQL queries satisfy this condition (e.g., return the average of a range query).

[5] For simplicity of exposition we assume symmetric (Type I) pairings. Our treatment can be extended to asymmetric pairings, which are what we use in our implementation for better efficiency.

Construction 2 (Verifiable Polynomial Delegation). *Let $\mathbb{F}$ be a prime-order finite field, ℓ be a variable parameter, and d be a degree parameter such that $O(\binom{\ell+\ell d}{\ell d})$ is polynomial in λ. Consider the following verifiable delegation protocol that supports the family $\mathcal{F}$ of all ℓ-variate polynomials of variable-degree d over $\mathbb{F}$.*

1) $\mathsf{KeyGen}(1^\lambda, \ell, d)$*: Run* $\mathsf{bp} \leftarrow \mathsf{BilGen}(1^\lambda)$*. Select uniform $\alpha, s_1, \ldots, s_\ell \in \mathbb{F}$ and compute $\mathbb{P} = \{g^{\prod_{i \in W} s_i},\ g^{\alpha \cdot \prod_{i \in W} s_i}\}_{W \in \mathcal{W}_{\ell,d}}$. The public parameters are* $\mathsf{pp} = (\mathsf{bp}, \mathbb{P}, g^\alpha)$.
2) $\mathsf{Commit}(f, \mathsf{pp})$*: Compute $c_1 = g^{f(s_1,\ldots,s_\ell)}$ and $c_2 = g^{\alpha \cdot f(s_1,\ldots,s_\ell)}$, and output the commitment* $\mathsf{com} = (c_1, c_2)$.
3) $\mathsf{Evaluate}(f, t, r, \mathsf{pp})$*: On input $t = (t_1, \ldots, t_\ell)$ and random challenge $r = (r_1, \ldots, r_{\ell-1})$, compute $y = f(t)$. Using Lemma 1 compute polynomial $q_\ell(x_\ell)$ and polynomials $q_i(x_i, \ldots, x_\ell)$ for $i = 1, \ldots, \ell-1$, such that*
$$f(x_1, \ldots, x_\ell) - f(t_1, \ldots, t_\ell) = (x_\ell - t_\ell) \cdot q_\ell(x_\ell) + \sum_{i=1}^{\ell-1} (r_i \cdot (x_i - t_i) + x_{i+1} - t_{i+1}) \cdot q_i(x_i, \ldots, x_\ell).$$
Output y and the proof $\pi := (g^{q_1(s_1,\ldots,s_\ell)}, \ldots, g^{q_{\ell-1}(s_1,\ldots,s_\ell)}, q_\ell)$.
4) $\mathsf{Ver}(\mathsf{com}, y, t, \pi, r, \mathsf{pp})$*: Parse the proof π as $(\pi_1, \ldots, \pi_{\ell-1}, q_n)$. If $e(c_1/g^y, g) \overset{?}{=} e(g^{s_\ell - t_\ell}, g^{q_\ell(s_\ell)}) \cdot \prod_{i=1}^{\ell-1} e(g^{r_i(s_i-t_i)+s_{i+1}-t_{i+1}}, \pi_i)$ and $e(c_1, g^\alpha) = e(c_2, g)$, output* accept *else, output* reject.

Definition 2. *Let $\mathbb{F}$ be a finite field, $\mathcal{F}$ be a family of ℓ-variate polynomials over $\mathbb{F}$, and d be a variable-degree parameter.* $(\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{Evaluate}, \mathsf{Ver})$ *constitute a* verifiable polynomial-delegation protocol *for $\mathcal{F}$ if:*

- ***Perfect Completeness.*** *For any polynomial $f \in \mathcal{F}$, if* $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, \ell, d)$ *and* $\mathsf{com} \leftarrow \mathsf{Commit}(f, \mathsf{pp})$*, then for any $t \in \mathbb{F}^\ell$ and $r \in \mathbb{F}^\ell$ if* $(y, \pi) \leftarrow \mathsf{Evaluate}(f, t, r, \mathsf{pp})$ *then (1) $y = f(t)$ and (2)* $\mathsf{Ver}(\mathsf{com}, t, y, \pi, r, \mathsf{pp})$ *=accept.*
- ***Soundness.*** *For any* PPT *adversary $\mathcal{A}$ the following is negligible:*
$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, \ell, d); (f^*, t^*) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}); \\ \mathsf{com} \leftarrow \mathsf{Commit}(f^*, \mathsf{pp}); r \leftarrow \mathbb{F}^\ell; (y^*, \pi^*) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}, f^*, r) \end{array} : \begin{array}{c} \mathsf{Ver}(\mathsf{com}, t^*, y^*, \pi^*, r, \mathsf{pp}) = \mathsf{accept} \\ \wedge y^* \neq f^*(t^*) \wedge f^* \in \mathcal{F} \end{array}\right]$$

$(\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{Evaluate}, \mathsf{Ver})$ *is an* extractable, verifiable polynomial-delegation protocol *if it additionally satisfies the following:*

- ***Knowledge Soundness.*** *For any* PPT *adversary $\mathcal{A}$ there exists a polynomial-time algorithm $\mathcal{E}$ with access to $\mathcal{A}'s$ random tape, such that the following probability is negligible:*
$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, \ell, d); (\mathsf{com}^*, t^*) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}); f' \leftarrow \mathcal{E}(1^\lambda, \mathsf{pp}); \\ r \leftarrow \mathbb{F}^\ell; (y^*, \pi^*) \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}, r); \mathsf{com} \leftarrow \mathsf{Commit}(f', \mathsf{pp}) \end{array} : \begin{array}{c} \mathsf{Ver}(\mathsf{com}^*, t^*, y^*, \pi^*, r, \mathsf{pp}) = \mathsf{accept} \wedge \\ (\mathsf{com}^* \neq \mathsf{com}\ \vee y^* \neq f'(t^*) \vee f' \notin \mathcal{F}) \end{array}\right]$$

then respond to requests for the correct evaluation of f on various points, along with a proof of correctness of the result.

There are several works in the literature addressing this problem [39], [13], [29], [49]. Our construction extends the scheme of Papamanthou et al. [49] (which itself extends prior work [39] to the multivariate case) in order to achieve a "knowledge" property, i.e., to ensure that if the server can correctly prove that y is the correct output relative to com for some input t, then the server in fact knows a polynomial f of the correct degree for which $f(t) = y$. Thus, our construction can be viewed as a special-purpose SNARK for polynomial evaluation. The modifications to the prior scheme are relatively small: we change the commitment to contain two group elements with "related" exponents (instead of containing one group element), and change the verification algorithm correspondingly. In the following, we define the *variable-degree* of a multivariate polynomial f be the maximum degree of f in any of its variables, and use $\mathcal{W}_{\ell,d}$ to denote the collection of all multisets of $\{1, \ldots, \ell\}$ for which the multiplicity of any element is at most d. Our polynomial-delegation protocol is presented in Construction 2 and relies on the following lemma.

Lemma 1 ([49]). *Let $f : \mathbb{F}^\ell \to \mathbb{F}$ be a polynomial of variable degree d. For all $t \in \mathbb{F}^\ell$ and all $r_1, \ldots, r_{\ell-1} \in \mathbb{F} \setminus \{0\}$, there exist efficiently computable polynomials $q_1, \ldots, q_\ell$ such that:*
$$f(x) - f(t) = \sum_{i=1}^{\ell-1} [r_i(x_i - t_i) + x_{i+1} - t_{i+1}] q_i(x) + (x_\ell - t_\ell) q_\ell(x_\ell)$$
where q_n is a univariate polynomial of degree at most d, and t_i is the ith element of t.

We define (extractable) verifiable polynomial delegation in Definition 2. A proof of the following appears in the full version of the paper.

Theorem 3. *Under Assumption 1, Construction 2 is a verifiable polynomial-delegation protocol. Moreover, under Assumptions 1 and 2, it is an* **extractable***, verifiable polynomial-delegation protocol. Algorithms* $\mathsf{KeyGen}, \mathsf{Commit}$ *run in time $O(\binom{\ell+\ell d}{\ell d})$,* $\mathsf{Evaluate}$ *in time $O(\ell d \binom{\ell+\ell d}{\ell d})$, and* Ver *in time $O(\ell + d)$. The commitment produced by* Commit *consists of $O(1)$ group elements, and the proof produced by* $\mathsf{Evaluate}$ *consists of $O(\ell)$ elements of $\mathbb{G}$ and $O(d)$ elements of $\mathbb{F}$.*

In the context of our verifiable database system (Section V), our verifiable polynomial-delegation protocol will be used in two ways. First, the database owner will use it to commit to its database; all subsequent proofs will be formulated with respect to that commitment. Queries may possibly be evaluated on additional auxiliary inputs (beyond the client's database) generated by the server. In this case the server will produce commitments to these inputs and the proof will be formulated with respect to both the original database commitment and these additional commitments. In the former case, the necessary security property is soundness alone; in the second case (since the commitment comes from the untrusted server), we need the stronger notion of knowledge soundness.

IV. Model

In this section, we present our security definition for a verifiable database system, viewed as a two-party protocol run

Definition 3. *A verifiable database system for database class $\mathcal{D}$ and query class $\mathcal{Q} = \mathcal{U} \cup \mathcal{S}$ (where $\mathcal{U}$ denotes update queries and $\mathcal{S}$ denotes selection queries), is a tuple of algorithms defined as follows:*

1) Setup *takes as input 1^λ, a database $D \in \mathcal{D}$ and outputs a digest δ and public parameters* pp.
2) $\mathsf{Evaluate}$ *is an interactive protocol run between two probabilistic polynomial-time algorithms C and S on common input a digest δ, a query $Q \in \mathcal{Q}$, and public parameters pp. Moreover, S holds database D. If $Q \in \mathcal{S}$, then at the end of the protocol C either outputs a result y (and accepts) or rejects. If $Q \in \mathcal{U}$, then at the end of the protocol C outputs a new digest δ' (and accepts), or rejects.*

Denote by $Q(D)$ the evaluation of query Q on database D. We require that Setup *and* $\mathsf{Evaluate}$ *have the following properties.*

- ***Perfect Completeness.*** *For any λ, any $D_0 \in \mathcal{D}$, any $t \geq 0$, and any queries $Q_1, \ldots, Q_t \in \mathcal{Q}$ and $Q^* \in \mathcal{S}$, we require that $y = Q^*(D_t)$ in the following experiment:*
  - Setup *is invoked on the input $(1^\lambda, D_0)$ and outputs (δ_0, pp).*
  - *For $1 \leq i \leq t$, do: S and C run $\mathsf{Evaluate}$ on inputs $(Q_i, \delta_{i-1}, D_{i-1}, \mathsf{pp})$ and $(Q_i, \delta_{i-1}, \mathsf{pp})$, respectively. If $Q_i \in \mathcal{U}$, let δ_i denote the output of C and set $D_i = Q_i(D_{i-1})$; otherwise, set $\delta_i = \delta_{i-1}$ and $D_i = D_{i-1}$.*
  - *S and C run $\mathsf{Evaluate}$ on inputs $(Q^*, \delta_t, D_t, \mathsf{pp})$ and $(Q^*, \delta_t, \mathsf{pp})$, respectively. Let y denote the output of C.*
- ***Soundness.*** *For any t and polynomial-time attacker S^*, the probability that S^* succeeds in the following experiment is negligible:*
  1) *$\mathsf{S}^*(1^\lambda)$ outputs $D_0 \in \mathcal{D}$.*
  2) *$\mathsf{Setup}(1^\lambda, D_0)$ outputs (δ_0, pp).*
  3) *For $1 \leq i \leq t$, do: S^* outputs Q_i. Then S^* and C run $\mathsf{Evaluate}$ on inputs $(Q_i, \delta_{i-1}, D_{i-1}, \mathsf{pp})$ and $(Q_i, \delta_{i-1}, \mathsf{pp})$, respectively. If C rejects, the experiment ends. If C accepts and $Q_i \in \mathcal{U}$, let δ_i denote the output of C and set $D_i = Q_i(D_{i-1})$; otherwise, set $\delta_i = \delta_{i-1}$ and $D_i = D_{i-1}$.*
  4) *S^* outputs $Q^* \in \mathcal{S}$. Then S^* and C run $\mathsf{Evaluate}$ on inputs $(Q^*, \delta_t, D_t, \mathsf{pp})$ and $(Q^*, \delta_t, \mathsf{pp})$, respectively. Let y denote the output of C. We say that S^* succeeds if C accepts with output y, but $y \neq Q^*(D_t)$.*

between a *client* that owns a database D which it wishes to outsource to a remote *server*. In a setup phase, the client computes a short digest of D, which it stores locally, and uploads D to the server. Subsequently, he issues queries about the data or requests to update the data, which are processed by the server. Each query evaluation is executed by an interactive protocol between the two parties, at the end of which the client either accepts the returned output or rejects it. Informally, the required security property is that no computationally bounded adversarial server can convince the client into accepting a false result. This is defined formally in Definition 3. To simplify notation, we do not distinguish between verification parameters (that are stored by the client and should be succinct) and proof-computation parameters (stored by the server).

Supporting Database Size Increases. For some constructions (including ours), the size of the public parameters pp may depend on the database size. If the database size increases (as a result of updates), it may be necessary to extend pp; there are various ways this can be done. For instance, the database owner can choose an upper bound for the database size, and generate a long-enough pp during the setup phase. Alternatively, the owner may maintain some (succinct) trapdoor information that allows it to extend pp as needed.

Efficiency Considerations. One important aspect of a verifiable database system is efficiency; a trivial approach is to transmit D for each query and have the client evaluate it himself. Therefore, a basic efficiency requirement is that the communication between client and server for query evaluation should be sublinear in the database size $|D|$. Also important is the client's computational cost for, which should ideally be smaller than evaluating the query (so the client can benefit not only from delegation of its storage but also from delegation of its computation). A final efficiency metric is the computational overhead of the server, which should ideally be asymptotically the same as the cost of evaluating the query.

V. The vSQL Protocol

A. High-Level Description

In this section, we present our construction of a verifiable database system. As mentioned above, our protocol uses the CMT protocol [25] as presented in Section II-D, as well as our verifiable polynomial-delegation protocol from Section III. In the sequel we refer to the prover and verifier of the CMT protocol as $(\mathcal{P}^{cmt}, \mathcal{V}^{cmt})$, and we refer to the algorithms of our polynomial-delegation protocol as $(\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{Evaluate}, \mathsf{Ver})$.

Preprocessing Phase. At a high level, we combine the CMT protocol and our polynomial delegation scheme as follows. Initially, the client views its database D as an array of $|D|$ elements (where $|D|$ is equal to number of rows times number of columns) and computes the multilinear extension $\tilde{D}$ as in Section II-C.[6] Note that the number of variables in $\tilde{D}$ is logarithmic in the total size of D. Next, the client generates a commitment com to $\tilde{D}$ using our polynomial-delegation protocol, stores com locally, and uploads D to the untrusted server. We stress that this phase does not depend on any specific queries the client may choose to issue later.

Query Evaluation Phase. All subsequent queries are validated by running a modified version of the CMT protocol between the client and the server. Our main observation is that in the last step of the CMT protocol, $\mathcal{V}^{cmt}$ needs to evaluate $\tilde{D}$ at a random point. Since the client no longer knows $\tilde{D}$, the client needs to rely on the untrusted server to provide this value. In order to ensure the correctness of the value provided by the server, the client and server use our polynomial-delegation protocol relative to the commitment com that the client holds. The client accepts the answer returned by the server to its original query only if both Ver and $\mathcal{V}^{cmt}$ accept.

[6] This can be done by concatenating the $|D|$ elements of the database into a single array of length $|D|$, by sorting them first by row and then by column.

Construction 3. *Let λ be a security parameter, let D be a database and let $\mathbb{F}$ be a prime-order field with $|\mathbb{F}|$ exponential in λ.*

Setup Phase. *On input 1^λ and a database $D \in \mathcal{D}$, the client picks a parameter $N \geq |D|$ such that $N \in O(|D|)$, which denotes an upper bound on the size of databases (in terms of values in the database) that can be supported, and sets $n = \lceil \log N \rceil$. Let $\tilde{D}$ denote the multilinear extension of D. The client runs $\mathsf{KeyGen}(1^\lambda, n, 1)$ to compute public parameters pp, and $\mathsf{Commit}(\tilde{D}, \mathsf{pp})$ to compute commitment com on $\tilde{D}$. It then sends $(D, \mathsf{pp}, \mathsf{com})$ to the server and stores $(\mathsf{pp}, \mathsf{com})$.*

Evaluation Phase. *Let $(x_0, \ldots, x_{N-1})$ be the current version of the database D stored by the server and let com be the commitment stored by both client and server. Given a query $Q \in \mathcal{Q}$, let C be a depth-d circuit over $\mathbb{F}$ that evaluates Q on input D and (possibly empty) auxiliary input $B \in \mathbb{F}^{|B|}$. Assume w.l.o.g. that $|B| = (2^m - 1) \cdot N$ for some integer m. Partition the input of C into 2^m arrays $(B_1, \ldots, B_{2^m})$ each of size N with B_1 corresponding to D and the rest corresponding to the auxiliary input. Finally, let $\tilde{B}_1, \ldots, \tilde{B}_{2^m}$ denote the corresponding multilinear extensions of $B_1, \ldots, B_{2^m}$ where $\tilde{B}_1 = \tilde{D}$.*

- *If Q is a selection query, the two parties then interact as follows:*
  1) S *computes the necessary auxiliary input $B_2, \ldots, B_{2^m}$, and runs $\mathsf{Commit}(\tilde{B}_i, \mathsf{pp})$ for $2 \leq i \leq 2^m$ to obtain values $\mathsf{com}_2, \ldots, \mathsf{com}_{2^m}$, which it sends to* C.
  2) C *runs $\mathcal{V}^{cmt,1+2}$ and* S *runs $\mathcal{P}^{cmt}$ to evaluate $C(B_1, \ldots, B_{2^m})$. If $\mathcal{V}^{cmt,1+2}$ rejects at any point,* C *outputs* reject. *Otherwise, let r_d, a_d be the final values returned by $\mathcal{V}^{cmt,1+2}$. Let $\tilde{V}_d$ be the multilinear extension of the input layer of C. At this point,* C *must verify that $\tilde{V}_d(r_d) = a_d$, which is done as follows.*
  3) C *sends to* S *values $\rho^{(1)}, \ldots, \rho^{(2^m)} \in \mathbb{F}^{n-1}$ chosen uniformly at random.*
  4) S *parses r_d as $r_d := (\kappa_1, \ldots, \kappa_{m+n})$ and defines $r'_d := (\kappa_{m+1}, \ldots, \kappa_{m+n})$.* S *then sends to* C *the evaluations $(v_1, \ldots, v_{2^m})$ of polynomials $\tilde{B}_1(r'_d), \ldots, \tilde{B}_{2^m}(r'_d)$ along with corresponding proofs π_i computed by $\mathsf{Evaluate}(\tilde{B}_i, r'_d, \rho^{(i)}, \mathsf{pp})$, for all $1 \leq i \leq 2^m$.*
  5) C *runs $\mathsf{Ver}(\mathsf{com}_i, r'_d, v_i, \pi_i, \rho^{(i)}, \mathsf{pp})$ for $1 \leq i \leq 2^m$. If any execution outputs* reject, C *outputs* reject. *Otherwise,* C *defines $r''_d := (\kappa_1, \ldots, \kappa_m)$ and computes $\tilde{V}_d(r_d)$ by combining values $v_1, \ldots, v_{2^m}$ as per Equation II-C-(2). If $\tilde{V}_d(r_d) \neq a_d$,* C *outputs* reject, *otherwise* accept.
  6) *The output of* S *is set to $C(B_1, \ldots, B_{2^m})$.*
- *If Q is an update query, the two parties then interact as follows:*
  1) S *computes the necessary auxiliary input $B_2, \ldots, B_{2^m}$, and runs $\mathsf{Commit}(\tilde{B}_i, \mathsf{pp})$ for $2 \leq i \leq 2^m$ computing values $\mathsf{com}_2, \ldots, \mathsf{com}_{2^m}$. Moreover, it computes the multilinear extension $\tilde{V}_{out}$ of the output of $C(B_1, \ldots, B_{2^m})$ and runs $\mathsf{Commit}(\tilde{V}_{out}, \mathsf{pp})$ to compute output commitment com_{out}. Finally, it sends $\mathsf{com}_{out}, \mathsf{com}_2, \ldots, \mathsf{com}_{2^m}$ to* C.
  2) C *chooses $r_0 \in \mathbb{F}^n$, (the output of C is the entire new database which by assumption is at most N therefore its multilinear extension operates on $n = \log N$ elements), and sends it to the server along with a uniform value $\rho_{out} \in \mathbb{F}^{n-1}$.*
  3) S *responds with $a_0 = \tilde{V}_{out}(r_0)$ and corresponding proof π_{out} computed with $\mathsf{Evaluate}(\tilde{V}_{out}, r_0, \rho_{out}, \mathsf{pp})$.*
  4) C *runs $\mathsf{Ver}(\mathsf{com}_{out}, r_0, a_0, \pi_{out}, \rho_{out}, \mathsf{pp})$ and rejects if it outputs* reject. *Otherwise,* C *runs $\mathcal{V}^{cmt,2}$ while* S *runs $\mathcal{P}^{cmt,2}$ on common input r_0, a_0. If $\mathcal{V}^{cmt,2}$ rejects at any point,* C *outputs* reject. *Otherwise, let r_d, a_d be the final values returned by $\mathcal{V}^{cmt,2}$. Let $\tilde{V}_d$ be the multilinear extension of the input layer of C. At this point,* C *must verify that $\tilde{V}_d(r_d) = a_d$. This is achieved by having* C *and* S *perform steps 3–5 from above.*
  5) *The output of* S *is set to $C(B_1, \ldots, B_{2^m})$ and com_{out}. If* C *accepts, it sets $\mathsf{com} \leftarrow \mathsf{com}_{out}$.*

B. Adapting the CMT protocol to SQL queries

In this section we discuss how we address various difficulties that arise when applying our protocol to SQL queries.

Supporting Comparisons. Since our approach utilizes the CMT scheme, queries need to be encoded as arithmetic circuits. One side effect of this is that non-arithmetic gates, such as a comparisons, cannot be handled directly. One way to handle comparisons in arithmetic circuits is to decompose the inputs into their bit-level representations, perform the comparison in binary, and then "glue" the results back together into a single element. This is inefficient since it requires many bit-decomposition operations as well as a complicated binary comparison circuit.

One way to reduce the overhead induced by arithmetic circuits is to allow the circuit to use additional "advice" provided by the server in the form of auxiliary inputs, i.e., to add support for non-deterministic computations. (So, for example, rather than compute the bit decomposition of an input directly, we can instead have the server provide the bit decomposition and then only use the circuit to verify that the given bit decomposition is correct. We highlight other efficiency benefits of this approach in Section VI-A.) Unfortunately, the CMT protocol does not naturally support such auxiliary input.

Supporting Auxiliary Inputs. Previous CMT-based works [56] addressed this issue by having the CMT prover provide the entire auxiliary input to the verifier, effectively treating the witness as part of the verifier's input. While this works, it is only effective when the size of the auxiliary input is small compared to the input size. For vSQL, however, we would like to support auxiliary input whose size is comparable to the size of the entire database.

Our main observation is that in order to successfully execute the CMT protocol, all the verifier needs for the input gates (including both the real input and the auxiliary input) is to be able to evaluate the multilinear extension polynomial of the inputs on a random point. Thus, instead of transmitting the entire auxiliary input to the verifier, the prover can commit to the multilinear extension of the auxiliary input using our verifiable polynomial-delegation protocol. To see how this works, assume the auxiliary input is the same size as the database. At the last step of the interactive proof protocol, the client will request the evaluations of the two polynomials (one for the database and one for the auxiliary input) at the same random point. The server then responds with the values and their corresponding proofs. Due to the additive property of multilinear extensions, the client can then combine these two values in order to reconstruct the evaluation of the multilinear extension of the entire input to the circuit, as described in Section II-C. Moreover, since the polynomial commitment is binding, the client can be sure that the server's response does not depend on the random point chosen by the client, thus preserving the soundness of the interactive-proof protocol.

More generally, this can be applied to any query (even those that require auxiliary inputs larger than the database size) by adding sufficient "padding" such that the total circuit input size (database plus auxiliary input) is a power-of-two multiple of the database size (e.g., if auxiliary input is $2 \cdot |D|$, it should be padded with $|D|$ dummy values, thus making the total circuit input size $4 \cdot |D|$). The server provides a separate commitment, evaluation and proof for each database-length "chunk" of the auxiliary input and the evaluation of the multilinear extension of the entire input is calculated by the client by applying Equation II-C-(2).

Supporting Expressive Updates. A common problem of existing dynamic authenticated data structures (e.g., [47], [60]) is that they support limited types of updates: element insertions and deletions. Thus, they cannot handle general updates that can be expressed as SQL queries themselves, e.g., the query `UPDATE` **Employees**; `SET` **Salary** $= 45000$; `WHERE` **Age** $= 33$.

The main reason such update queries are hard to handle is that the client must eventually compute the corresponding updated database commitment. Without access to the database, it must again rely on the untrusted server to provide this new commitment. SNARK-based constructions can support expressive updates by including the commitment computation in the circuit. However, this would considerably increase the prover's overhead. Our approach avoids this cost by separating the computation of the update from its verification. First, the server computes the updated database normally, and commits to the multilinear extension of the result using our verifiable polynomial delegation scheme. The client and server then verify that the update was performed correctly by running the CMT protocol on the circuit that performs the update. In order to initiate the CMT protocol, the client needs to compute the multilinear extension of the updated database (which is here the circuit's output) and evaluate it on a random point. This would naively require transmitting the entire updated database back to the client. Instead, we rely on the server to compute the evaluation for the client, and verify this value using our verifiable polynomial-delegation scheme. Once this is done, the remainder of the CMT evaluation proceeds normally.

Exploiting SQL Query Structure. Our construction can be applied to verify the computation of any arithmetic circuit, which clearly includes SQL queries. But the specific structure of SQL queries allows for additional efficiency improvements. Concretely, most "natural" SQL queries specify some computation that is applied independently to every database row, followed by a final aggregation/post-processing phase. Thus, the arithmetic circuit C that corresponds to the entire SQL query can be written as a sequence of parallel copies of a smaller circuit C' corresponding to the single-row logic, where inputs to C are wired directly to the appropriate copy of C', and the outputs of the copies of C' are wired into a (small) post-processing circuit C''. We can thus rely on Remark 1 to improve the prover's efficiency.

C. Our Construction

Before describing our construction, we introduce some notation. The client in our construction will be running a modified version of $\mathcal{V}^{cmt}$ that selectively executes some of the three steps of Construction 1. Let $\mathcal{V}^{cmt,1+2}$ denote a version of the verifier $\mathcal{V}^{cmt}$ that only runs the first two steps in Construction 1 and then outputs r_d, a_d, omitting step 3. Likewise, let $\mathcal{V}^{cmt,2}$ denote the restricted version of $\mathcal{V}^{cmt}$ that on input r_0, a_0 runs only the second step, again outputting r_d, a_d, and $\mathcal{P}^{cmt,2}$ the similar restricted version of $\mathcal{P}^{cmt}$ that runs the second step on input r_0, a_0.

Our main construction is given as Construction 3. A proof of the following appears in the full version of the paper.

Theorem 4. *If Construction 2 is an extractable, verifiable polynomial-delegation protocol, then Construction 3 is a verifiable database system for SQL queries.*

If Construction 3 is executed on a database D with $|D|$ values, to evaluate a query expressed as a non-deterministic, depth-d arithmetic circuit C with at most S gates per layer, that consists of parallel copies of a circuit C' with at most S' gates per layer, followed by a post-processing circuit C'' of size $O(|C|/\log|C|)$, and with auxiliary input B, then

1) *The running time of* Setup *is $O(|D|)$.*
2) $\mathsf{Evaluate}$ *requires $O(d \log S)$ rounds of interaction.*
3) *The running time of* C *is $O(k + d \cdot \mathrm{polylog}(S) + \log(|B| + |D|))$, where k is the size of the result for selection queries and k is $O(\log|D'|)$ for updates (D' is the output size).*
4) *The running time of* S *is $O(|C| \cdot \log S' + (|B| + |D|) \cdot \log(|B| + |D|))$.*

Local State at the Client. For simplicity, in our description we assume the client stores the entire public parameters pp, which are as large as D. However, in practice the client only needs to store n terms from pp, specifically all terms g^{τ_i} for $i \leq 1 \leq n$, that are necessary for verifying the evaluation of the multilinear extensions it receives from the server using our verifiable polynomial-delegation protocol. We also note that verification does not require any trapdoor information, and therefore our scheme has *public verifiability*: namely, anyone with access to the database commitment produced by the client and the public parameters can issue and verify queries.

Handling Unbounded Database Size. For simplicity, we assume that the size of the database will never exceed the bound N. In practice, this can achieved by picking N large enough. Note however, that this is not a limitation of our construction: a client that stores the n trapdoor values $s_1, \ldots, s_n$ of the verifiable polynomial-delegation scheme can compute additional elements, as needed, in case the database size exceeds N. Moreover, our construction has the nice feature that the work required by the client and the server at any given time only depends on the size of the actual database and not the upper bound N.

VI. Performance Optimizations

In this section, we present a number of optimizations that we apply to the evaluation phase. In particular, we leverage the ability of our scheme to efficiently handle auxiliary inputs in order to: (i) achieve faster equality testing (which is useful for selection queries), (ii) allow for input/output gates at arbitrary layers of the circuit with minimal overhead, and (iii) verify the results of set intersections using a smaller number of gates (which is useful for join queries). Finally, we discuss how simple updates (that consist of assigning values to unused table cells) can be verified using one round of interaction.

Most of the optimizations discussed below exploit various techniques for constructing efficient representations of computations commonly when answering SQL queries. These techniques include modifying the queries' circuit representations in order to utilize auxiliary inputs, encoding some of the query computations directly as polynomials, and utilizing interaction in order to reduce the circuit size. Since these modifications are applied directly to the underlying circuit being computed, security when using these optimizations follows readily from security of our protocol.

A. *Optimizing Equality Testing*

A very common subroutine used in both selection and join queries is testing whether two values are equal, which can be reduced to testing whether their difference is 0. Here we show how we can efficiently perform such zero tests using auxiliary input provided by the prover.

Optimized Zero Testing. Ideally, we would like a small arithmetic circuit that takes as input a field element x and outputs $x' = 0$ if $x = 0$ and $x' = 1$ otherwise. It is well known [25] that, by relying on Fermat's little theorem, this can be done by computing $x' = x^{p-1}$ (where p is the field size). This approach is relatively expensive, however, since it requires a circuit of size and depth $O(\log p)$. Instead, we will construct a non-deterministic circuit for this task that has two outputs x', z and satisfies the following: $x = 0$ iff there is an auxiliary input y such that $x' = 0$ and $z = 0$; also, $x \neq 0$ iff there is an auxiliary input y such that $x' = 1$ and $z = 0$. Thus, the rest of the computation can use x', and the client will additionally verify that $z = 0$.

We can achieve the above by computing $x' = xy$ and $z = x \cdot (1 - xy)$. Note that setting $y = x^{-1}$ if $x \neq 0$ (and setting y arbitrarily otherwise) yields correct values for x' and z. Moreover, if $x = 0$ then $x' = z = 0$ for any choice of y, and if $x \neq 0$ then the only way to force $z = 0$ is to set $x' = 1$. We note that the same high-level idea has appeared before (e.g., [53], [51]) in the context of SNARKs that are defined based on constraint systems. In our case, the CMT protocol only supports the evaluation of arithmetic circuits (and not constraint systems), and so we need a slightly different technique.

Enforcing Zero Values. A trivial implementation of the above would require the server to send all the x', y values to the client, resulting in the client performing work linear in the number of zero tests. Since zero testing may be done at least once per database row, this will lead to large overheads.

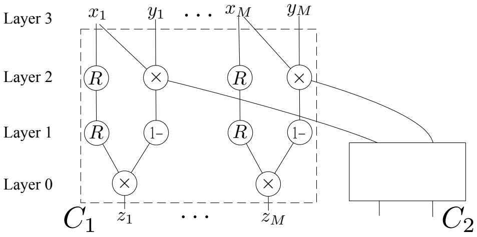

Fig. 1. Zero testing. If $z_i = 0$ then the input to C_2 is a 0/1 value indicating whether x_i is zero.

Instead (cf. Figure 1), we split the computation into two parts: (i) a circuit C_1 that computes $z = x(1 - xy)$, and (ii) a circuit C_2 that evaluates the SQL query using the result of the zero test (i.e., $x' = xy$). Without loss of generality, we assume the result of the zero test is used at the input layer of C_2, as shown in Figure 1. The client and the server will run two separate interactive proof protocols for C_1 and C_2. First, the protocol for C_2 is executed up to one layer before its input layer (i.e., the client and server pause before proceeding to its input layer). After that, the protocol for C_1 is initiated. Note that the honest prover does not need to send any of the outputs of C_1 to the verifier since the verifier knows all of them are supposed to be 0. Moreover, in order to initiate the execution of this protocol, the verifier needs to compute the multilinear extension of the outputs of C_1 evaluated at a random point. Since the multilinear extension of the 0-vector is the 0-polynomial, this step is free. Once the interactive protocol for C_1 finishes layer 1, the verifier uses *the same randomness* for the next layer of both circuits (layer 2 of C_1 and the input layer of C_2, which have the same values).[7] This reduces the claims in both executions to a single evaluation of the multilinear extension of the joint input for that layer. Finally, layer 3 (the input layer) of C_1 is verified normally. In this way, the prover's overhead for zero testing is only linear in the size of C_1, which only has 3 layers. The verifier's overhead is only polylogarithmic in the size of C_1.

In our experiments (where $\lceil \log p \rceil = 254$), the above zero testing and enforcement method yield an 80× speedup for both prover and verifier compared to the deterministic approach using Fermat's little theorem.

Handling Conjunctions and Disjunctions. In multi-dimensional SQL selection queries, AND or OR operators are applied on the results of multiple selection clauses over different columns, and thus the number of zero tests required potentially grows with the number of columns. But note that OR clauses can be trivially reduced to a single zero test; e.g., testing $x_1 = 0 \vee x_2 = 0$ reduces to testing $x_1 x_2 = 0$. We further observe that AND clauses can also be reduced to a single zero test if the input values are known to be in a bounded range. For example, if it is known that $-\sqrt{p/2} < x_1, x_2 < \sqrt{p/2}$ then we may reduce evaluating the conjunction $x_1 = 0 \wedge x_2 = 0$ to evaluating whether $x_1^2 + x_2^2 = 0$. In particular, if all values in question are 32 bits long and p is a 254-bit value, then we can test conjunctions involving up to 2^{189} values using just a single zero test. Alternatively, we can handle conjunctions using

[7] Using the same randomness for both C_1 and C_2 does not affect the soundness of the CMT protocol here.

packing: e.g., if x_1, x_2 are 32-bit values (and $|p| > 64$) then testing whether $x_1 = 0 \wedge x_2 = 0$ is equivalent to testing whether $2^{32}x_1 + x_2 = 0$. These approaches ensure the number of required auxiliary inputs (as well as the size of the zero-test circuit) for a multi-dimensional selection query depends linearly on the number of rows in the table and is almost independent of the number of columns involved in the query.

B. Supporting Inputs/Outputs at Arbitrary Circuit Layers

So far, we have assumed that the circuit being computed takes all its inputs at the same layer, and produces all its outputs at the same layer. This is without loss of generality since one can always define a "relay" gate that simply passes its input to the next layer. In practice however, such relay gates will contribute some cost to the execution of the interactive-proof protocol [56]. For many natural SQL queries, this might even result in a highly inefficient circuit where most gates are relay gates. For example, consider an SQL query of the form `SELECT * FROM` **T** `WHERE` $\text{col}_i = x$. A circuit for evaluating this query takes the entire table as input, but only values from the ith column are involved in the selection process. All the other values, from all other columns, are simply relayed between the various circuit layers.

Avoiding Relaying the Inputs. We now describe a technique that avoids relay gates by leveraging the property of the multilinear extension described in Section II-C. Concretely, consider a circuit C such that some internal layer k operates on $2M$ values with $m = \lceil \log M \rceil$. Assume the second half (denoted by B) of the $2M$ values are "fresh inputs" (these may either be from the database itself, or auxiliary input from the prover), while the first half (denoted by A) come from layer $k+1$. Before running the CMT protocol for C, the verifier holds the commitment (either obtained from the preprocessing or received from the server) to the multilinear extension, $\tilde{V}_k^B$, of the fresh inputs to the kth layer. Next, during the execution of the CMT protocol, the client receives the evaluation of the multilinear extension of the values at layer k, i.e., $\tilde{V}_k(r_1, \ldots, r_{m+1})$, at some random point $(r_1, \ldots, r_{m+1})$ as before. As only the first M wires (corresponding to A) are connected to layer $k+1$, the client needs to obtain the evaluation of the multilinear extension (denoted by $\tilde{V}_k^A$) of the first M values, at a random point and use it to continue the CMT protocol for layer $k+1$.

This is done as follows. By Equation 2 in Section II-C, we have $\tilde{V}_k(r_1, \ldots, r_{m+1}) = (1 - r_1)\tilde{V}_k^A(r_2, \ldots, r_{m+1}) + r_1 \cdot \tilde{V}_k^B(r_2, \ldots, r_{m+1})$. Since B are all input gates, the client can request the evaluation of $\tilde{V}_k^B$ at point $(r_2, \ldots, r_{m+1})$ along with a corresponding proof (using the verifiable polynomial-delegation protocol). Next, the client computes $\tilde{V}_k^A(r_2, \ldots, r_{m+1}) = \left(\tilde{V}_k(r_1, \ldots, r_{m+1}) - r_1 \cdot \tilde{V}_k^B(r_2, \ldots, r_{m+1})\right)/(1 - r_1)$, obtaining an evaluation of $\tilde{V}_k^A$ at the random point $(r_2, \ldots, r_{m+1})$. The client then uses it to continue the execution of the CMT protocol for layer $k+1$ as usual.

We note that similar optimizations can be performed in order to avoid relying output gates as well.

Generalizations. For both inputs and outputs, using Equation 2 allows us to avoid relaying a number of input and the output gates. We notice that the number of input (resp. output) gates does not have to be half of the total number of gates in the layer, but can be any fraction $1/m'$ such that m' is a power of 2. Moreover, while we described the solution assuming that the "fresh" inputs at some layer are all in the second half of the inputs to that layer, this is not required. With small modifications we can accommodate more complicated wiring patterns, e.g., the case where odd wires are routed from the previous layer and even wires are fresh inputs to the circuit.

C. Verifying Set Intersections

A join operation requires computing the intersection of two large sets of column values (assuming for now there are no duplicates). The naive way to compute the intersection of two N-element sets, where each element is represents using z bits, requires a circuit that performs N^2 equality tests on z-bit inputs. We describe here several ways this can be improved.

A Sorting-Based $O(zN \log^2 N)$ Solution. An asymptotic improvement can be obtained by first sorting the $2N$ elements, and then comparing consecutive elements in the sorted result. Sorting can be done using $O(N \log^2 N)$ comparator gadgets of width z, resulting in a circuit of size $O(zN \log^2 N)$ overall. The concrete overhead of this approach is high, as each comparator must be implemented by decomposing the inputs to their bit-level representations.

A Routing-Based $O(zN + N \log N)$ Solution. Prior literature on SNARKs [10] improves the above by relying on auxiliary input from the prover to replace sorting networks with switching networks that can induce arbitrary permutations on N elements. Using this approach, the server will simply specify the permutation that sorts the elements; the client can verify that the elements are sorted in linear time. Switching networks can be built using $O(N \log N)$ gadgets that swap their inputs if an auxiliary bit is set to 1. The total complexity of this approach is $O(zN + N \log N)$.

An $O(zN)$ Interactive Solution. In our setting, where we have interaction, we can do better. We simply have the server provide the sorted list $x'_1, \ldots, x'_{2N}$ corresponding to the original items $x_1, \ldots, x_{2N}$. The client can verify that the new list is sorted in $O(N)$ time, so all that remains is for the client to verify that it is a permuted version of the original list. This can be done by having the server commit to the new values (as part of the auxiliary input he computes) using our verifiable polynomial-delegation scheme. The client then chooses and sends to the server a uniform value r, and both parties then run an interactive proof protocol to verify that $\prod_{i=1}^{N}(x_i - r) - \prod_{i=1}^{N}(x'_i - r) = 0$. Overall, this approach requires $O(zN)$ auxiliary inputs and gates.

Sorting 0 Values. The concrete cost can be further reduced as follows. In case many of the elements are 0, after the sorting step they will be pushed to the front of the auxiliary-input array (assuming, for simplicity, that all values are non-negative). Instead of providing one auxiliary input per element, it suffices for the prover to tell the verifier the number of non-zero elements, and only provide auxiliary inputs for those. For example, assume only the last $1/m$ of elements are non-zero (where m is a power of 2), using Equation 2 in Section II-C, the evaluation of the multilinear extension for all elements at point $r = (r_1, \dots, r_{\log(mn)})$ is $\tilde{V}(r) = r_1 \dots r_{\log m} \tilde{V}_m(r_{\log m+1}, \dots, r_{\log mn})$, where $\tilde{V}_m$ is the multilinear extension of the non-zero elements. Thus, the size of the auxiliary input and the number of necessary comparisons only depend on the number of non-zero elements (as opposed to the total number of elements).

In the context of SQL queries, the scenario above is very common. Consider a query where a join clause is applied on the result of two range queries. It is often the case that only a small portion of rows in the table fall within the bounds imposed by the latter. Therefore, after evaluating the range selection, the values in these rows will be propagated through the circuit, while the values in all other rows will effectively be set to 0. The join query (and therefore the sorting) will then be applied on this result which has the property that many of its elements are 0. Thus the above optimization can significantly lower the join evaluation cost in this case.

Sorting Multiple Columns. Another challenge arises when the output of a join query includes more than just the reference column, e.g., `SELECT * FROM` $\mathbf{T}_1$,$\mathbf{T}_2$, `WHERE` $\mathbf{T}_1.col_i = \mathbf{T}_2.col_j$. In this case, in order to compute the set intersection using the above interactive method, the verifier must make sure that the prover permuted all of the columns of $\mathbf{T}_1$ (resp., $\mathbf{T}_2$) with the same permutation used for col_i (resp., col_j).

We achieve this using the following *packing* technique. Assume for simplicity that each database row has two columns with values x_i, y_i respectively, and that the elements are arranged as tuples $(x_1, y_1), \cdots, (x_N, y_N)$. Suppose the elements x_i, y_i have length at most z bits, with $z < \lfloor \log p \rfloor / 2$. To sort both columns based on the x_i values, we ask the server to provide auxiliary inputs $(a_1, \dots, a_{2N}) = (x_{\pi(1)}, y_{\pi(1)}, \cdots, x_{\pi(N)}, y_{\pi(N)})$, such that the $\{x_{\pi(i)}\}$ are sorted and the $\{y_{\pi(i)}\}$ are permuted by the same permutation. The client then chooses and sends to the server two random values r_1, r_2, and both parties run the interactive proof protocol described above for the following three checks:

1) $\prod_{i=1}^{N}(x_i - r_1)(y_i - r_1) - \prod_{i=1}^{2N}(a_i - r_1) = 0$;
2) $\prod_{i=1}^{N}(b_i - r_2) - \prod_{i=1}^{N}(b'_i - r_2) = 0$, where $b_i = x_i + y_i 2^z$ and $b'_i = a_{2i-1} + a_{2i} 2^z$;
3) $(a_1, a_3, \dots, a_{2N-1})$ are sorted.

The first check guarantees that a_is are a permutation of x_i, y_is, which also implies that a_is have length at most z bits. Now as x_i, y_i, a_is all have length at most z bits, the second check guarantees that $\exists \pi : a_{2i-1} = x_{\pi(i)}$ and $a_{2i} = y_{\pi(i)}$ (note that we cannot omit the first check as there exist a_is with more than z bits that can pass the second check). This, together with the last check, guarantees $x_{\pi(i)}$s are sorted and $y_{\pi(i)}$s are permuted by the same permutation.

The technique generalizes naturally to sort multiple columns based on a reference column. As long as the packing result does not overflow in $\mathbb{F}_p$, we can pack all the columns. Otherwise, we can duplicate the reference column, perform a separate packing of subsets of columns, and sort them separately. In particular, assuming $z = 32$ and p is 254 bits long, we can pack up to 7 columns in a single field element.

Handling Duplicate Values. Finally, if there are duplicate values in the reference columns, the result of a join query can no longer be described as a set intersection. In this case, a pairwise comparison of the elements of the two columns, viewed as multisets, provides the correct result but the cost is quadratic in the number of database rows. Instead we can do the following. First, we extract the unique values from each multiset (using a linear-size circuit as described in [54]). Then we compute the intersection of the resulting sets with our previous technique for the case of no duplicates. Following this, we apply again the same technique to intersect this intersection with each of the original multisets. This returns two multisets such that: (i) each of them contains exactly those elements that appear in both original multisets, and (ii) every element appears in each multiset exactly the same amount of times as it appeared in the the corresponding original multiset. Finally, the join result can be computed with a pair-wise comparison of the elements of these two multisets. Note that the cost for this final step is asymptotically optimal as it is exactly the same as simply parsing the join's output.

D. Efficient Value Insertions

As explained above, our construction can handle any update query by having the server evaluate the update-query circuit and then commit to the output as the new digest. For simple updates such as adding/subtracting a constant from an element, we have a much simpler mechanism. By utilizing the closed form of the multilinear extension, in order to add a constant v to the bth entry in the database, the multilinear extension of the database is increased by $\mathcal{X}_b(x_1, \cdots, x_n)v$ (as defined in Equation 1). Therefore, the client only needs to multiply the commitment of the database by $g^{\mathcal{X}_b(x_1,\cdots,x_n)v} = p_b^v$, where p_b is the bth element of the public key $\mathbb{P}$. In practice, as the size of $\mathbb{P}$ is linear in the size of the database, the client can outsource its storage to the server and obtain an authenticated value of p_b using a Merkle hash tree or digital signatures. Thus, simple updates of this form can be handled with one round of interaction, and the running time for both parties is logarithmic in the database size using a Merkle tree, or constant using a digital signature scheme. The update above also captures inserting a new element/row to the database, which is adding their values to previously unused cells.

VII. Empirical Evaluation

A. Experimental Setup

Software. We implemented our constructions (including the circuit generator, CMT protocol, and polynomial-delegation

```
1. SELECT n_name, SUM(l_extendedprice*(1-l_discount))
2. AS revenue
3. FROM customer, orders, lineitem, supplier, nation, region
4. WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey
5.     AND l_suppkey = s_suppkey   AND c_nationkey = n_nationkey
6.    AND n_regionkey = r_regionkey AND r_name = 'MIDDLE EAST'
7.     AND o_orderdate >= date '1997-01-01'
8.     AND o_orderdate < date '1997-01-01'+interval '1' year
9. GROUP BY n_name
10. ORDER BY (revenue) DESC;
```

Fig. 2. Query #5 of the TPC-H benchmark.

protocol) in C++, and compiled it with g++ 4.8.4. We use the NTL library [5] for number-theoretic operations, and SHA-256 in OpenSSL [6] for hashing (in order to instantiate a random oracle). For the bilinear pairing we use the ate-paring library [1] on a 254-bit elliptic curve. The EMP toolkit [58] was used for the network I/O between the server and the client.

Hardware and Network. Our experiments were executed on two Amazon EC2 c4.8xlarge machines running Linux Ubuntu 14.04, with 60GB of RAM and Intel Xeon E5-2666v3 CPUs with 36 virtual cores running at 2.9 GHz. For the WAN experiments, we used machines hosted in two different regions, one in the US East and the other in the US West. The average network delay was measured at 72ms and the bandwidth was 9MB/s. For each data point, we collected 10 experimental results and report their average.

B. Benchmark Dataset

Database Setup. We evaluate performance using the TPC-H benchmark [7], which contains 8 synthetic tables and 22 SQL queries and is widely used by the database community for performance evaluation. We represented decimal numbers, dates, and categorical strings in the tables as elements in the field used by our constructions. In our experimental evaluations, we do not consider substring or wildcard queries, and the corresponding columns were discarded. The TPC-H database contains tables of various sizes. The two largest tables used in our experiments contained 6 million rows and 13 columns and 0.8 million rows and 4 columns, respectively.

TPC-H Queries. We tested five TPC-H queries: query #2, #5, #6, #15, and #19. As a representative example, query #5 is shown in Figure 2. It gives an example of multi-way join queries on different columns of different tables. sub-query in line 6 is a selection query on table region, and the query in lines 7–8 is a range query on table order. Lines 4–6 consist of join queries among tables customer, order, lineitem, supplier, nation, region. In line 1, the result is projected to three columns, two of which are aggregated. Finally, in lines 9–10, the aggregated values are summed for each unique value of n_name, and sorted based on n_name in descending order.

Query #2 is a nested query. The inner query consists of a 4-way join followed by a MIN query, resulting in a single value. The outer query consists of selection queries, where the result of the inner query is used as a constraint, followed by a 4-way join and projections. Query #6 is a simple 3-dimensional range query followed by an aggregation. Query #19 consists of range and selection queries on two tables, followed by a single join query and an aggregation. Query #15 creates a new table that is the result of a one-dimensional range query and a SUM query. All the other queries in TPC-H are variants of these five queries with different dimensions and constraints.

Query Representation and Field Sizes. For every TPC-H query we implemented a circuit generator that takes as input the database size and outputs an arithmetic circuit for evaluating the specified query on a database of that size, using the optimizations described in Section VI (when possible). We implemented both the CMT protocol (Construction 1) and the verifiable polynomial-delegation protocol (Construction 2) using a prime-order field with a 254-bit prime.

C. Performance Comparison: Selection Queries

We compare the performance of our construction with prior work, including IntegriDB [60], a special purpose system optimized for a class of SQL queries, and libsnark [4], the state-of-the-art general-purpose SNARK implementation. We also against (non-verifiable) SQL, based on MySQL. Below, we report the results on queries #2, #5, #6, and #19.

For IntegriDB, we downloaded the implementation from [2] and executed it on our machine. For libsnark, we estimated the performance as follows. For each query, we first produced its circuit representation the jSNARK compiler [3], hardcodeding the TPC-H dataset in the circuit. This resulted in a circuit which takes as inputs the values used in selection and range queries. We then constructed a SNARK using libsnark for this circuit, and report its performance. We note that this approach of hardcoding the database and the query into the circuit yields a preprocessing phase whose results are only useful for that specific query and database. In particular, the results of the preprocessing phase *cannot* be reused for other queries or databases, or even an updated version of the database. Although clearly unrealistic, this approach gives a lower bound on the server time when using a SNARK-based approach.[8] Even with this more efficient approach, we were not able to generate SNARKs for circuits containing more than 2^{20} multiplication gates (see Table 7 for the circuit sizes of the queries we used in our evaluation). Therefore, for experiments requiring larger circuits, we estimated the cost assuming the prover time grows linearly in the circuit size (this is, again, an underestimate since the prover time actually grows quasilinearly in the circuit size).

Setup Phase. The setup phases in both IntegriDB and vSQL are query independent and thus need to be executed only once, after which any supported queries can be handled. We run the setup phases of both IntegriDB and vSQL on all eight TPC-H tables. The setup for vSQL took about 2,467 seconds. For IntegriDB, the setup phase could not be completed on the entire TPC-H database due to excessive memory consumption. Our estimate for the setup phase of IntegriDB was about

[8] It is possible to use SNARKs that support arbitrary queries by constructing a SNARK for a universal circuit [12] and supporting delegation of storage [28], [19]. However, these approaches introduce additional overhead.

	IntegriDB		SNARKs		vSQL (ours)				MySQL
Query	Server	Client	Server*	Client*	Server	Client	Total (WAN)	Total (NI)	
#19	6,376s	232ms	196,000s	6ms	4,892s	162ms	4,957s	4,892s	0.67s
#6	1,818s	74ms	19,000s	6ms	3,851s	129ms	3,869s	3,851s	3.92s
#5	✗	✗	615,000s	110ms	5,069s	398ms	5,278s	5,069s	4.16s
#2	✗	✗	58,000s	40ms	2,346s	508ms	2,559s	2,346s	2.96s

Fig. 3. Comparison of server and client times for evaluating queries using IntegriDB, a SNARK-based approach, our construction, and plain SQL (based on MySQL). (See text for details.) The numbers in columns marked by * are estimated. ✗ denotes an unsupported query.

350,000 seconds. Our construction is about 121× faster than IntegriDB because the complexity of our setup phase is linear in the number of columns compared to quadratic in IntegriDB.

For libsnark, the setup time depends on the query. The fastest setup time, for query #6, is estimated to take 36,000 seconds, which is an order-of-magnitude slower than vSQL. Running setup for all four queries is estimated to require about $1.7 \cdot 10^7$ seconds (roughly 197 days).

Evaluation Phase. The results of the evaluation phase are summarized in Table 3. The numbers reported in the "Server" column reflect the computation time required for the server to evaluate the SQL query and produce a valid proof; those in the "Client" column reflect the time for the client to verify the proof. For vSQL, in the "Total (WAN)" column we also report the total end-to-end time which includes the overhead due to communication between the client and server over a WAN network. For comparison, the total time for IntegriDB and SNARKs (which are non-interactive) is essentially the same as the server time, since the client time is negligible. Note, however, that vSQL can be made non-interactive in the random oracle model, virtually eliminating the cost of interaction at the negligible expense of a small number of SHA-256 computations. We report the performance of this non-interactive mode, including the SHA-256 computation time, under "Total (NI)."

Evaluation Phase: vSQL vs. IntegriDB. As shown in Table 3, IntegriDB can only support queries #19 and #6, compared with vSQL which can support all TPC-H queries. While being more expressive, the running times of vSQL's client and server are on the same order of magnitude as those of IntegriDB; in fact, for query #19, vSQL's server (resp., client) outperforms that of IntegriDB by about 23% (resp., 30%). We observe also that the cost of interaction for vSQL (even over a WAN) is small, mainly because prover time is by far the dominating cost.

Evaluation Phase: vSQL vs. SNARKs. Compared to libsnark, the server time of our constructions is significantly faster (ranging from 5× for query #6 to 120× for query #5). At a high level, the better performance of vSQL is a consequence of two features. First, our construction is mostly information-theoretic and the number of (relatively slow) cryptographic operations it requires is linear in the *input and output length*, whereas SNARK-based approaches require a number of cryptographic operations linear in the *circuit size*. In addition, as described in Section VI, our construction leverages interaction and auxiliary input to reduce the size of a query's circuit representation. Verification when using a SNARK-based approach is faster than in vSQL since it requires only a number of cryptographic operations linear in the output length. In practice, however, the difference is at most 0.5sec which we consider negligible for most applications. We stress that all numbers reported for libsnark are underestimations since they assume the database and queries are fixed in advance. We expect our construction's improvement to be even more significant compared to more general SNARK-based systems that support arbitrary queries and dynamic outsourced databases (see discussion below).

Communication. For libsnark-based systems, the additional communication required for the proof is always constant (e.g., 288 bytes). For IntegriDB and vSQL the communication required in all experiments was under half a megabyte. We consider this to be negligible in practice for modern networks and thus omit additional details.

Comparison with Other SNARK-based Systems. SNARKs can be used for verifiable computation in various ways other than the one we used for our comparison.

Exploiting Structured Computations via Bootstrapping. Geppetto [26] takes a complex computation and splits it into smaller building blocks, each represented as a "small" circuit. Each such circuit can then be pre-processed with a SNARK separately. In the context of SQL queries, the natural way to split the computation is by having one small circuit that operates on a single row, and then applying that circuit iteratively to each row in the database. An additional SNARK is then needed to aggregate and verify the outputs of all the smaller SNARKs into a single succinct proof, in a "bootstrapping" step. In practice, Geppetto has the potential to significantly reduce the preprocessing time and memory consumption since the same small circuit is used throughout the query evaluation. However, the total prover time to execute the smaller SNARK on every row is similar to that of the single large SNARK we used as our benchmark, as the total number of exponentiations is linear in the total number of multiplication gates and breaking a large circuit into multiple smaller ones does not reduce that. Our results already show that the prover time in that case is up to 2 orders of magnitude slower than for vSQL. Additionally, the bootstrapping phase requires approximately 30,000–100,000 gates per instance of the smaller circuit [26, Section 7.3.1]. Applying this to a table with 6 million rows (as in the TPC-H dataset) will thus introduce an additional overhead of 1.4–4.8 × 10^7s for the prover, based on our estimations with libsnark (which is itself an underestimate as the bootstrapping phase operates over a larger and less efficient elliptic curve).

Memory Delegation via Hash Functions. Pantry [19] can

Server	Client	Total (WAN)	Total (RO)	Comm.
2,049.9s	85ms	2,106.9s	2,050.8s	85.7KB

Fig. 4. Performance of our construction on TPC-H query #15, creating a new table from table lineitem.

be used to outsource memory by implementing a Merkle hash tree on top of Pinocchio [51]. The consistency of each memory read/write access must be proven by checking the corresponding Merkle path as part of the SNARK that evaluates the query. This approach has the benefit of allowing the verifiable evaluation of RAM programs on outsourced memory (as opposed to expressing the computation directly as a circuit). For SQL queries, assuming the existence of pre-built database indices (as is typically the case with modern database management systems), there are programs that can evaluate certain queries in time sublinear in the database size. (E.g., assuming the existence of a search tree that stores the ordered element values at its connected leaves, a simple 1-D range query can be evaluated in time logarithmic in the database size and linear in the result.) Thus, for specific queries for which such indices can be built, Pantry can in theory outperform vSQL. Regarding the specific queries we evaluated here, we note that the number of memory accesses would still be very large even with the help of pre-built indices. The simplest TPC-H query we tested is query #6, which is a 3-dimensional range query followed by a summation. Assuming a search tree is built for each dimension and each 1-D range query has a 1% selectivity (which is well below the selectivity in our experiments), getting the result of each dimension requires 60,000 memory accesses. In practice, the concrete cost of proving the correctness of each memory access would be approximately 2.5s, using a SNARK-friendly algebraic hash function [40] for 10^6 4-byte memory blocks. Therefore, just verifying the memory accesses for query #6 would take around 450,000s (5 days) in this case.

Finally, in contrast to vSQL, both approaches require a query-specific setup phase that can only be avoided if one uses a universal circuit [12] or proof-bootstrapping [11], but these techniques incur considerable additional overheads.

D. Performance Comparison: Update Queries

We test the performance of vSQL on a CREATE query (query #15 in TPC-H). As shown in Table 4, the communication required is only 85.7KB, even though the newly created table is itself more than 1MB in size. IntegriDB cannot directly support such expressive updates. The only solution for IntegriDB would be for the client to download the entire new table, verify its correctness, and preprocess it from scratch.

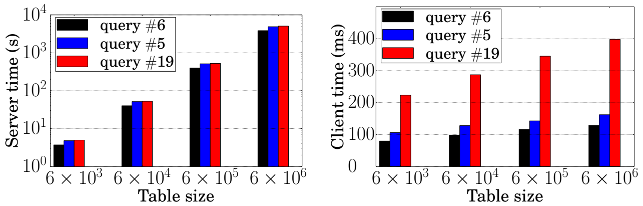

Fig. 5. Performance of vSQL for three TPC-H queries as a function of the number of rows in the largest table involved in the query.

Query	# of Inputs	Time (sec)	Time/Input (ms)
#15	12,002,430	1,405	0.1171
#2	17,840,340	1,990	0.1115
#6	24,004,860	2,715	0.1130
#5	31,397,075	3,509	0.1118
#19	32,406,075	3,695	0.1140

Fig. 6. Prover time for our polynomial-delegation scheme. The number of inputs includes both the database and the prover's auxiliary inputs.

Next we look at updates that can be supported by IntegriDB, in particular inserting a new row. In this case, vSQL outperforms IntegriDB since the total time for inserting a row into the lineitem table in TPC-H is only 5.2ms using vSQL vs. 1.46s using IntegriDB. This is because the vSQL client only needs to verify the corresponding elements of the public parameters using a Merkle-tree proof and then perform one exponentiation per column. For IntegriDB, the required number of exponentiations is quadratic in the number of columns and logarithmic in the number of rows.

In order for a SNARK-based system to support updates, it must offer a way to check the validity of a new database digest returned by the prover. The standard way of doing this is by incorporating the digest computation in the circuit that is evaluated, which introduces a huge cost in practice. More recent approaches can achieve this via an external mechanism that is not part of the circuit of the SNARK (e.g., [8], [28]). Note however, that at the very least the update circuit must be evaluated and the SNARK proof must be computed by the prover. According to our performance comparison in the previous section, this already takes more time than vSQL.

E. Scalability of Our Construction

In this section, we evaluate the performance of our constructions as a function of the database size. To that end, we run our construction on the largest three of the previous queries and scale the number of rows in the largest participating table from 6,000 to 6,000,000.

Server Time. As shown in Figure 5, the server performance for query evaluation scales almost linearly with the size of the largest table, matching the theoretical analysis of Theorem 4.

Client Time. Figure 5 shows that the client's verification time grows logarithmically with the number of rows in the largest table participating in a query (note the logarithmic scale of the horizontal axis). This again matches Theorem 4.

F. Microbenchmarks

In addition to evaluating vSQL's end-to-end performance, we also report the performance of vSQL's main components.

Performance of the Polynomial-Delegation Scheme. Table 6 shows the prover time for our implementation of the polynomial-delegation scheme (Construction 2). The prover spends about 0.11ms per input, which is the same order of magnitude as SNARK-based schemes. Preprocessing for our polynomial-delegation scheme (which is the only part of our construction that requires preprocessing) took only 2,467 seconds for 95,525,880 inputs (25.8μs per gate).

Query	# of Gates	Time (sec)	Time/Gate (μs)
#2	198,646,335	356	1.79
#15	367,495,719	646	1.76
#6	704,643,060	1,137	1.61
#19	801,374,196	1,198	1.49
#5	945,828,996	1,560	1.65

Fig. 7. Prover time for our implementation of the CMT protocol.

Performance of the CMT Protocol. Table 7 shows the performance of our implementation of the CMT protocol. As can be seen, the average time required per gate is about 1.7μs.

VIII. Conclusion

In this work we show how to extend the CMT protocol using a polynomial-delegation protocol to efficiently support outsourced data as well as auxiliary inputs (i.e., non-deterministic computations). We then incorporate this new construction into vSQL, a verifiable database system that can support arbitrary SQL queries, including updates. Compared to previous general approaches, vSQL offers significantly faster evaluation time for the server as it requires cryptographic work linear in the size of the query's inputs and outputs (and independent of the size of the query's circuit representation). Our experimental evaluation demonstrates that in practice this results in a concrete speedup for the server of up to $120\times$ compared to a SNARK-based approach, with negligibly larger overhead for the client. The overall performance of vSQL is comparable to, and sometimes better than, that of IntegriDB, a state-of-the-art scheme that only supports (and is optimized for) a restricted subclass of SQL.

Limitations and Future Work. Our main construction can support the delegation of any computation that can be represented as a non-deterministic arithmetic circuit, which theoretically includes all possible SQL queries. For many common operations (such as joins and comparisons), vSQL includes specific optimizations which reduce the overall circuit size. vSQL also benefits from the structure of most SQL queries, which can be viewed as applying the same computation to every row in a table. In general, however, our optimizations do not apply to all possible SQL queries. In particular, since some variants of the SQL language are Turing complete [42], general (and relatively inefficient) reductions from Turing machines to circuits are required to support them. Moreover, additional work is needed to handle some operations that arise in many "natural" SQL queries, most prominent of which are substring and wildcard queries. We leave the task of handling such queries efficiently for future work.

Acknowledgments

We thank the anonymous reviewers for their comments, and Andrew Miller for shepherding the paper. This work was supported in part by NSF awards #1514261 and #1526950, financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology, the Rothschild foundation, and the Warren Center for Network and Data Sciences.

References

[1] Ate pairing. https://github.com/herumi/ate-pairing.

[2] Integridb. https://github.com/integridb/Code.

[3] jsnark. https://github.com/akosba/jsnark.

[4] libsnark. https://github.com/scipr-lab/libsnark.

[5] NTL library. http://www.shoup.net/ntl/.

[6] OpenSSL toolkit. https://www.openssl.org/.

[7] TPC-H benchmark. http://www.tpc.org/tpch/.

[8] Backes, M., Barbosa, M., Fiore, D., and Reischuk, R. M. ADSNARK: Nearly practical and privacy-preserving proofs on authenticated data. In *S&P 2015*, pp. 271–286.

[9] Backes, M., Fiore, D., and Reischuk, R. M. Verifiable delegation of computation on outsourced data. In *CCS 2013*, pp. 863–874.

[10] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., and Virza, M. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO 2013*. pp. 90–108.

[11] Ben-Sasson, E., Chiesa, A., Tromer, E., and Virza, M. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO 2014*. pp. 276–294.

[12] Ben-Sasson, E., Chiesa, A., Tromer, E., and Virza, M. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security 2014*.

[13] Benabbas, S., Gennaro, R., and Vahlis, Y. Verifiable delegation of computation over large datasets. In *CRYPTO 2011*, pp. 111–131.

[14] Bitansky, N., Canetti, R., Chiesa, A., and Tromer, E. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS 2012*, pp. 326–349.

[15] Bitansky, N., Canetti, R., Paneth, O., and Rosen, A. On the existence of extractable one-way functions. STOC 2014, pp. 505–514.

[16] Boneh, D., and Boyen, X. Short signatures without random oracles. In *EUROCRYPT 2004*, pp. 56–73.

[17] Boyle, E., Goldwasser, S., and Ivan, I. Functional signatures and pseudorandom functions. In *PKC 2014*, pp. 501–519.

[18] Boyle, E., and Pass, R. Limits of extractability assumptions with distributional auxiliary input. In *ASIACRYPT 2015*, pp. 236–261.

[19] Braun, B., Feldman, A. J., Ren, Z., Setty, S., Blumberg, A. J., and Walfish, M. Verifying computations with state. In *SOSP 2013*, pp. 341–357.

[20] Canetti, R., Chen, Y., Holmgren, J., and Raykova, M. Adaptive succinct garbled RAM or: How to delegate your database. In *TCC 2016-B*, pp. 61–90.

[21] Canetti, R., Paneth, O., Papadopoulos, D., and Triandopoulos, N. Verifiable set operations over outsourced databases. In *PKC 2014*, pp. 113–130.

[22] Canetti, R., Riva, B., and Rothblum, G. N. Practical delegation of computation using multiple servers. In *CCS 2011*, pp. 445–454.

[23] Catalano, D., Fiore, D., Gennaro, R., and Nizzardo, L. Generalizing homomorphic MACs for arithmetic circuits. In *PKC 2014*, pp. 538–555.

[24] Chung, K., Kalai, Y. T., Liu, F.-H., and Raz, R. Memory delegation. In *CRYPTO 2011*, pp. 151–168.

[25] Cormode, G., Mitzenmacher, M., and Thaler, J. Practical verified computation with streaming interactive proofs. In *ITCS 2012*, pp. 90–112.

[26] Costello, C., Fournet, C., Howell, J., Kohlweiss, M., Kreuter, B., Naehrig, M., Parno, B., and Zahur, S. Geppetto: Versatile verifiable computation. In *S&P 2015*, pp. 253–270.

[27] Devanbu, P., Gertz, M., Kwong, A., Martel, C., Nuckolls, G., and Stubblebine, S. G. Flexible authentication of XML documents. In *CCS 2001*, pp. 136–145.

[28] Fiore, D., Fournet, C., Ghosh, E., Kohlweiss, M., Ohrimenko, O., and Parno, B. Hash first, argue later: Adaptive verifiable computations on outsourced data. Cryptology ePrint Archive, 2016.

[29] Fiore, D., and Gennaro, R. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *CCS 2012*, pp. 501–512.

[30] Fournet, C., Kohlweiss, M., Danezis, G., and Luo, Z. ZQL: A compiler for privacy-preserving data processing. In *USENIX Security 2013*, pp. 163–178.

[31] Gennaro, R., Gentry, C., and Parno, B. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO 2010*, pp. 465–482.

[32] Gennaro, R., Gentry, C., Parno, B., and Raykova, M. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT 2013*, pp. 626–645.

[33] Goldwasser, S., Kalai, Y. T., and Rothblum, G. Delegating computation: interactive proofs for muggles. In *STOC 2008*, pp. 113–122.

[34] Goldwasser, S., Micali, S., and Rackoff, C. The knowledge complexity of interactive proof-systems. In *STOC 1985*, pp. 291–304.

[35] Goodrich, M. T., Tamassia, R., and Triandopoulos, N. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica 60*, 3 (2011), 505–552.

[36] Groth, J. On the size of pairing-based non-interactive arguments. In *EUROCRYPT 2016*, pp. 305–326.

[37] Groth, J. Short pairing-based non-interactive zero-knowledge arguments. In *ASIACRYPT 2010*, pp. 321–340.

[38] Kalai, Y. T., and Paneth, O. Delegating RAM computations. In *TCC 2016-B,*, pp. 91–118.

[39] Kate, A., Zaverucha, G. M., and Goldberg, I. Constant-size commitments to polynomials and their applications. In *ASIACRYPT 2010*, pp. 177–194.

[40] Kosba, A., Zhao, Z., Miller, A., Qian, Y., Chan, H., Papamanthou, C., Pass, R., abhi shelat, and Shi, E. C∅c∅: A framework for building composable zero-knowledge proofs. Cryptology ePrint Archive, Report 2015/1093, 2015. http://eprint.iacr.org/2015/1093.

[41] Kosba, A. E., Papadopoulos, D., Papamanthou, C., Sayed, M. F., Shi, E., and Triandopoulos, N. TRUESET: Faster verifiable set computations. In *USENIX Security 2014*, pp. 765–780.

[42] Law, Y.-N., Wang, H., and Zaniolo, C. Query languages and data models for database sequences and data streams. In *VLDB 2004*, pp. 492–503.

[43] Li, F., Hadjieleftheriou, M., Kollios, G., and Reyzin, L. Dynamic authenticated index structures for outsourced databases. In *SIGMOD 2006*, pp. 121–132.

[44] Lund, C., Fortnow, L., Karloff, H., and Nisan, N. Algebraic methods for interactive proof systems. *J. ACM 39*, 4 (1992), 859–868.

[45] Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., and Stubblebine, S. G. A general model for authenticated data structures. *Algorithmica 39*, 1 (2004), 21–41.

[46] Miller, A., Hicks, M., Katz, J., and Shi, E. Authenticated data structures, generically. In *POPL 2014*, pp. 411–423.

[47] Papadopoulos, D., Papadopoulos, S., and Triandopoulos, N. Taking authenticated range queries to arbitrary dimensions. In *CCS 2014*, pp. 819–830.

[48] Papadopoulos, D., Papamanthou, C., Tamassia, R., and Triandopoulos, N. Practical authenticated pattern matching with optimal proof size. *VLDB 2015*, 750–761.

[49] Papamanthou, C., Shi, E., and Tamassia, R. Signatures of correct computation. In *TCC 2013*, pp. 222–242.

[50] Papamanthou, C., Tamassia, R., and Triandopoulos, N. Optimal verification of operations on dynamic sets. In *CRYPTO 2011*, pp. 91–110.

[51] Parno, B., Howell, J., Gentry, C., and Raykova, M. Pinocchio: Nearly practical verifiable computation. In *S&P 2013*, pp. 238–252.

[52] Setty, S., Braun, B., Vu, V., Blumberg, A. J., Parno, B., and Walfish, M. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys 2013*, pp. 71–84.

[53] Setty, S. T. V., Vu, V., Panpalia, N., Braun, B., Blumberg, A. J., and Walfish, M. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security Symposium 2012*, pp. 253–268.

[54] Thaler, J. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO 2013*, pp. 71–89.

[55] Thaler, J. A note on the GKR protocol, 2015. Available at http://people.cs.georgetown.edu/jthaler/GKRNote.pdf.

[56] Vu, V., Setty, S., Blumberg, A. J., and Walfish, M. A hybrid architecture for interactive verifiable computation. In *S&P 2013*, pp. 223–237.

[57] Wahby, R. S., Setty, S. T. V., Ren, Z., Blumberg, A. J., and Walfish, M. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS 2015*.

[58] Wang, X., Malozemoff, A. J., and Katz, J. EMP-toolkit: Efficient multiparty computation toolkit. https://github.com/emp-toolkit.

[59] Yang, Y., Papadias, D., Papadopoulos, S., and Kalnis, P. Authenticated join processing in outsourced databases. In *SIGMOD 2009*, pp. 5–18.

[60] Zhang, Y., Katz, J., and Papamanthou, C. IntegriDB: Verifiable SQL for outsourced databases. In *CCS 2015*, pp. 1480–1491.

[61] Zhang, Y., Papamanthou, C., and Katz, J. Alitheia: Towards practical verifiable graph processing. In *CCS 2014*, pp. 856–867.

[62] Zheng, Q., Xu, S., and Ateniese, G. Efficient query integrity for outsourced dynamic databases. In *CCSW 2012*, pp. 71–82.

Appendix A
Bilinear Assumptions

We rely on the following assumptions. Let PPT stand for "probabilistic polynomial-time." To prove security of our construction, we use the following assumptions:

Assumption 1 ([16] (q-Strong Diffie-Hellman))**.** *For any* PPT *adversary $\mathcal{A}$, the following probability is negligible:*

$$\Pr\left[\begin{array}{c} \mathsf{bp} \leftarrow \mathsf{BilGen}(1^\lambda); \\ \tau \xleftarrow{R} \mathbb{Z}_p^*; \\ \sigma = (\mathsf{bp}, g^{\tau}, \ldots, g^{\tau^q}) \end{array} : (x, e(g,g)^{\frac{1}{\tau+x}}) \leftarrow \mathcal{A}(1^\lambda, \sigma)\right]$$

Let $\mathcal{W}_{\ell,d}$ denote the set of all multisets of $\{1, \ldots, \ell\}$ in which the multiplicity of any element is at most d.

The next assumption states the following. Assume a polynomial time algorithm that receives as input two ordered sequences of elements of $\mathbb{G}$ such that each element contains in the exponent a multivariate monomial with at most ℓ variables and of degree at most $\ell \cdot d$, for some d, and for every ordered pair of elements (across the two sequences) it holds that the elements differ in the exponent by a fixed multiplicative factor α. Then, if the party outputs a new such pair of elements that differ in the exponent by α, then it must hold that the first of these two elements was computed as a linear combination of the elements of the first sequence (and likewise for the second and the same linear combination). This fact is captured by the existence of a polynomial-time extractor that, upon the same input outputs this linear combination.

Assumption 2 (((d, ℓ)-Power Knowledge of Exponent)**.** *For any* PPT *adversary $\mathcal{A}$ there is a polynomial-time algorithm $\mathcal{E}$ (running on the same random tape) such that for all benign auxiliary inputs $z \in \{0,1\}^{poly\lambda}$ the following probability is negligible:*

$$\Pr\left[\begin{array}{c} \mathsf{bp} \leftarrow \mathsf{BilGen}(1^\lambda); \\ \tau_1, \ldots, \tau_\ell, \alpha \xleftarrow{R} \mathbb{Z}_p^*, \tau_0 = 1; \\ \sigma_1 = \{g^{\prod_{i \in W} \tau_i}\}_{W \in \mathcal{W}_{\ell,d}}; \\ \sigma_2 = \{g^{\alpha \cdot \prod_{i \in W} \tau_i}\}_{W \in \mathcal{W}_{\ell,d}}; \\ \sigma = (\mathsf{bp}, \sigma_1, \sigma_2, g^\alpha); \\ \mathbb{G} \times \mathbb{G} \ni (h, \tilde{h}) \leftarrow \mathcal{A}(1^\lambda, \sigma, z); \\ (a_0, \ldots, a_{|\mathcal{W}_{\ell,d}|}) \leftarrow \mathcal{E}(1^\lambda, \sigma, z) \end{array} : \begin{array}{c} e(h, g^\alpha) = e(\tilde{h}, g) \\ \wedge \\ \displaystyle\prod_{W \in \mathcal{W}_{\ell,d}} g^{a_W \prod_{i \in W} \tau_i} \\ \neq h \end{array}\right]$$

The above is a knowledge-type assumption that is a direct generalization of Groth's q-PKE assumption [37] for the case of multivariate polynomials. In fact, q-PKE is by definition the same as $(1, q)$-PKE, using our notation. Note that $\mathcal{W}_{\ell,d}$ has size $O(\binom{\ell+\ell d}{\ell d})$. In our construction, we will be using this assumption for the case where d is constant and ℓ is logarithmic in the database size. The results of [18], [15] show the impossibility of knowledge assumptions with respect to arbitrary auxiliary inputs. In the above definition we use the notion of a benign auxiliary input (or, alternatively, a benign state generator), similar to [26], [36], [28], to refer to auxiliary inputs that make extraction possible, avoiding these negative results. Concretely, our proofs hold assuming the auxiliary input of the extractor comes from a benign distribution.