ZKSQL: Verifiable and Efficient Query Evaluation with Zero-Knowledge Proofs

Xiling Li
Northwestern University
xiling.li@northwestern.edu

Chenkai Weng
Northwestern University
ckweng@u.northwestern.edu

Yongxin Xu
Northwestern University
yongxinxu2022@u.northwestern.edu

Xiao Wang
Northwestern University
wangxiao@northwestern.edu

Jennie Rogers
Northwestern University
jennie@northwestern.edu

ABSTRACT

Individuals and organizations are using databases to store personal information at an unprecedented rate. This creates a quandary for data providers. They are responsible for protecting the privacy of individuals described in their database. On the other hand, data providers are sometimes required to provide statistics about their data instead of sharing it wholesale with strong assurances that these answers are correct and complete such as in regulatory filings for the US SEC and other goverment organizations.

We introduce a system, *ZKSQL*, that provides authenticated answers to ad-hoc SQL queries with zero-knowledge proofs. Its proofs show that the answers are correct and sound with respect to the database's contents and they do not divulge any information about its input records. This system constructs proofs over the steps in a query's evaluation and it accelerates this process with authenticated set operations. We validate the efficiency of this approach over a suite of TPC-H queries and our results show that ZKSQL achieves two orders of magnitude speedup over the baseline.

PVLDB Reference Format:
Xiling Li, Chenkai Weng, Yongxin Xu, Xiao Wang, and Jennie Rogers. ZKSQL: Verifiable and Efficient Query Evaluation with Zero-Knowledge Proofs. PVLDB, 16(8): 1804 - 1816, 2023.
doi:10.14778/3594512.3594513

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/vaultdb/zksql.

1 INTRODUCTION

Databases are ubiquitous and we entrust them with private data in an ever-increasing diversity of applications. Hence, users are frequently asking questions about a data provider's records in contexts where they are reticent to share their records. On the other hand, when clients are using query answers in mission-critical settings–such as census data or unemployment statistics–they need to be convinced of the *soundness* of their answers, or that they are correct and complete. This issue is also becoming increasingly important for outsourced DBMSs where data owners are delegating more and more of their operations to a cloud service provider.

For example, the US Department of Education (DoE) collects statistics about student outcomes from domestic universities on its College Scorecard [24] for prospective students, which includes graduation rate, average annual cost and median earnings. The DoE wishes to get these statistics with strong assuances of their correctness and completeness. At the same time, colleges are required to protect student confidentiality. A lack of strong assurances can result in incidents that undermine user trust in these publications. For example, in 2022 Columbia University's ranking in US News dropped by several positions when it came to light that some of the statistics they submitted were incorrect [26]. Using conventional a DBMS, it is difficult for the DoE to verify statistics provided by universities. However, if DoE can query these databases directly, there's a risk of unauthorized leakage of student records. In this work, we explore one solution from cryptography to this challenge.

Zero-knowledge (ZK) proofs, proposed in [16, 17], are cryptographic protocols within which a prover, $\mathcal{P}$, convinces a verifier, $\mathcal{V}$, that its results from some computation are correct and complete without revealing any additional information about its inputs except that which can be deduced from its results. Interactive protocols accomplish this via a series of challenge-response rounds. For querying a database, $\mathcal{P}$ is the data owner and $\mathcal{V}$ is the client. $\mathcal{P}$ sends $\mathcal{V}$ a query answer, A, and a proof of its soundness and completeness. Until recently ZK proofs were too inefficient to be practical for all but the simplest applications. In the past few years, the cryptography community has put substantial research effort into making these protocols more concretely efficient thereby reducing their overhead by orders of magnitude [1, 5, 7, 12, 14, 15, 19, 20, 22, 25, 29–31].

In the aforementioned example, a university is the prover and the DoE is the verifier. Using interactive ZK proofs, the DoE would verify those aggregated statistics while learning no individual student records in the university's private database.

In this work, we propose a system named *ZKSQL* or "Zero-Knowledge proofs over SQL" that provides authenticated, ZK proofs for one or more query answers with respect to a private database. ZKSQL enables $\mathcal{V}$ (e.g., the DoE) to get authenticated answers (e.g., graduation rate) efficiently from $\mathcal{P}$ (e.g., a university) while $\mathcal{P}$ may be untrusted. Its proofs are with respect to a public commitment of the private data. They reveal only the schema and cardinality of each table in the private database. Then, ZKSQL executes protocols over the authenticated values of committed data.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 8 ISSN 2150-8097.
doi:10.14778/3594512.3594513

ZKSQL traces the path of transformations the query evaluator takes from committed tuples to query answers. It does so by creating ZK proofs one operator at a time to authenticate its intermediate and final results without leaking information about them. This approach may make verifiable cloud computing more accessible for data owners who want assurances that the cloud is faithfully storing their data and answering their queries. Moreover, this system makes it possible for end users (non-programmers) to prove knowledge of facts about the contents of their database to a third party without divulging their data. It also confirms that a batch of interrelated facts from multiple queries are derived from the same dataset. This research addresses the twin issues of data integrity and data privacy for querying relational databases with SQL by extending efficient, state-of-the-art interactive proving techniques to this setting. $\mathcal{P}$ computes the queries from $\mathcal{V}$ over the committed relations in their database. $\mathcal{P}$ returns authenticated answers to $\mathcal{V}$ with proofs that the answers are faithfully computed over their data. It builds atop state-of-the-art ZK protocols based on *Vector Oblivious Linear Evaluation (VOLE)* [5, 12, 29, 30]. An authenticated query provides authenticated tags of its query answers, thereby making them composable with outside workloads. This system supports a broad class of database operators including select, project, join, aggregate (including group-bys), sort, and set operations. Hence it makes this proving technology accessible to a broader audience.

We offer two ways to construct its ZK proofs: circuit-based and set-based. A given operator uses one or both of them in its proofs. Circuit-based proofs compute the authenticated answer by tracing the operator logic over encrypted, data-independent circuits with inputs from the initial commitment of the provers's tables. Set-based ones prove *properties* about a given operator's results. For example, for sort we prove that its output has the same set of tuple values as its input and that its outputs are monotonically increasing. We do so using polynomials so that we can prove properties about the entire set of rows in lieu of doing so one gate and one tuple at a time. This set equality expressed as a polynomial is a linear-time operation (in the set size), whereas doing the proof in circuits using an oblivious sort would run in $n \log^2 n$ time. The AKS sorting network has complexity $O(n \log n)$, but only when $n \geq 10^{52}$. This set-based optimization will not improve the performance of all ZKSQL operators, in particular, tuple-at-a-time ones like filter.

To the best of our knowledge, ZKSQL is the first to offer ZK verifiable computation of ad-hoc SQL queries for relational DBs. There is limited existing work on ZK proofs for querying relational databases. Since ZK protocols can prove any NP statement, proving SQL queries in ZK has been long known to be feasible. There was prior work [33] that brings ZK to verifiable SQL computation, but they do not support all types of queries, let alone ad-hoc ones. This result is theoretical in nature and has not been made practical for implementation in a DBMS. In this work, we focus on verifying the correctness of answers to ad-hoc SQL queries using ZK proofs.

Our main contributions are:

- First work on ZK proofs for ad-hoc SQL queries
- Operator-at-a-time proofs for steps in a given query plan.
- Set-based protocols for proving properties about the intermediate results of an authenticated query.
- Experimental results verify demonstrate ZKSQL's speedup of up to two orders of magnitude over the naïve baseline.

The rest of this paper is organized as follows. Section 2 provides background on ZK proofs and relates them to conventional RDBMS query execution. Section 3 introduces notations, security model, and workflow of ZKSQL. Section 4 illustrates functionality, authenticated set operations, and protocols of ZKSQL one relational operator at a time. We reveal our experimental results from our prototype on top of EMP Toolkit [28] in Section 5 using the TPC-H benchmark [27]. We survey relevant research on ZK proofs and verifiable query evaluation in Section 6 and conclude in Section 7.

2 BACKGROUND

In this section, we provide fundamentals upon which ZKSQL builds. First, we describe zero-knowledge proofs and protocols we use in this system. Then we review aspects of conventional query execution that are relevant to the new functionality offered by ZKSQL.

2.1 Zero-Knowledge Proofs

A *zero-knowledge proof* allows that a prover convinces a verifier that a statement they made is true without revealing any additional information. A data owner can use this kind of proof to demonstrate it possesses some knowledge without revealing it. In other words, ZK proof enables a prover $\mathcal{P}$ with a private witness w to show a public statement C with respect to w is true to a verifier $\mathcal{V}$ without revealing anything beyond $C(w) = 1$. For example, if a system has a database instance w and the client receives a query answer A accompanied by an interactive proof $C(w)$ that verifies the answer.

There are two common use cases for ZK proofs. First, when a prover wishes to verify a statement C that is a solution to some computationally difficult problem. For example, if we factor a large number n to two primes and wish to prove we know its factors, we can reveal that we hold two w witnesses whose product is equal to n and have no divisors without revealing w. Second, these protocols can prove multiple facts with respect to some private data (say a database D). ZKSQL addresses the latter class of problems using the commit-and-prove paradigm [11, 23].

More specifically, ZK protocols prove any NP-time result in polynomial time. The focus of this paper is to extend these techniques to prove that query answers are correct, sound, and leak no additional information about the source data. The dominant paradigm for expressing ZK proofs is circuits thereby making them easily composable [10] for database operators. Hence, our proof framework is modular enough to support ad-hoc querying.

Our system uses *Vector Oblivious Linear Evaluation (VOLE)* [5, 12] ZK protocols, the state-of-the-art interactive ones but it can use any protocols that instantiate this circuit-based API. In contrast, non-interactive ones like *zk*-SNARKs [8] and Bulletproofs [9], are less able to scale to large commitments and complex query logic with billions of gates because they do everything in a single pass [18]. ZKSQL's prover and verifier interactively compute its proofs with VOLE-based protocols [29, 30] in EMP-Toolkit [28].

Owing to the expressiveness of the circuit-based paradigm, we could verify the evaluation of any SQL statement using them by translating the operator logic directly to circuits. On the other hand, re-executing the entirety of a query's instruction traces using this technique would be prohibitively expensive–taking orders of magnitude longer than running the same query with no verification (see Table 2). To address this, we identify ways to prove *properties* about our query answers and their intermediate results in lieu of

Table 1: ZKSQL Notation.

Symbol	Description	Symbol	Description
$\mathcal{P}, \mathcal{V}$	Prover, Verifier	D	Set of relations in the database
R, S, T	Relations	$r_i, r_{i,j}$	i^{th} tuple in R, j^{th} field of r_i
$r_{i,\text{dummy}}$	Dummy tag of tuple r_i	$\mathcal{W}_R$	Degree or arity of R
C_e	Circuit to verify expr e	$[R]$	Authenticated tags of R
Q	Query to be verified	A	Authenticated answer to Q

fully tracing the data's path through its operators from committed bits to output. For example, if we are proving the correctness of a sort's results, we can show that it has the same tuples as its input and that the output rows are monotonically increasing. We reason about the correctness of many of our intermediate results using our set-based proofs in Section 4.1. Here we construct our proofs by expressing multisets of tuples as polynomials. We use the Schwartz-Zippel lemma to sample points on the corresponding curve to prove relationships such as set equality and intersection.

2.2 Oblivious Query Evaluation

In ZKSQL, the prover holds a private database. The verifier only knows the database schema and table cardinalities, which enables the verifier to compose a query and to work interactively with the prover to verify the answer. When a conventional DBMS receives a query, it first confirms that it is syntactically correct by binding its references to its table definitions in the system catalog. In ZKSQL, both parties have the SQL statement and parse it to the agreed-upon schema. They parse the query into a directed acyclic graph (DAG) of database operators, such as filters, joins, and aggregates.

Since we want to reveal no information about our query answers except that which can be inferred from their contents, we make their traces data-independent or *oblivious*. Hence, the observable transcript of the program does not branch or have early termination in loops. This makes it impossible for a malicious verifier to deduce any information about the query's input rows, such as the ones selected by a filter or matched in a join.

To facilitate obliviousness, we introduce dummy tuples into our query evaluation. Hence, when an operator deletes a tuple ZKSQL nulls out the deleted row's physical bits, creating a tombstone. All rows in a ZKSQL query have a bit at the end for this called a *dummy tag* to denote if a tuple should contribute to our query evaluation. When the engine creates the commitments of the prover's data, it initializes this tag to false. When we delete a tuple, we also set its dummy tag to true to communicate this to subsequent operators.

3 OVERVIEW AND QUERY PLANNING

We next provide an overview for the architecture of ZKSQL. We first describe the notations. Then, we describe our security guarantees. Next, we describe the workflow with which the system goes from a SQL statement to providing the client with an authenticated answer.

Table 1 introduces the notation of ZKSQL. First, let us call the prover ($\mathcal{P}$) or *Alice* and the verifier ($\mathcal{V}$) or *Bob*. D is the private database to which $\mathcal{P}$ alone has access. Our unary operators take as input a relation R and output T. In other words, they take the form $T := op(R)$ (e.g. op is Project). Similarly, we denote our binary operators as $T := op(R, S)$ (e.g. op is Join). r_i is the ith tuple in R, and $r_{i,j}$ refers to the jth field in r_i. $\mathcal{W}_R$ denotes the number of columns in R. $r_{i,dummy}$ is the dummy tag of r_i introduced in Section 2.2.

We prove that the system correctly executed each expression in a query using circuits. We show a circuit for verifying an expression

Functionality $\mathcal{F}_{\text{ZK}}$

Inputs: Upon receiving (Input, x) from $\mathcal{P}$, the functionality stores x, generates tags $[x]$, and sends $[x]$ to $\mathcal{P}$ and $\mathcal{V}$.

Constants: $\mathcal{P}$ and $\mathcal{V}$ send (Const, x) to the functionality. It stores x, generates tags $[x]$, and sends them to both parties.

Circuit evaluation: The functionality receives $(\text{Compute}, C, [x_0], ..., [x_{n-1}])$ from both parties, where $x_i \in \{0, 1\}$ and C is a Boolean circuit, the functionality computes $(y_0, ..., y_{m-1}) := C(x_0, ..., x_{n-1})$, generates tags $[y_0], ..., [y_{m-1}]$ and sends them to both parties.

Prove circuit satisfiability: Upon receiving $(\text{Verify}, C, [x_0], ..., [x_{n-1}])$ from both parties, where $x_i \in \{0, 1\}$ and C is a Boolean circuit, the functionality computes $b := C(x_0, ..., x_{n-1})$. If $b = 0$ the functionality aborts; otherwise it sends (satisfied) to $\mathcal{V}$.

Figure 1: Ideal functionality for proofs.

e as C_e. Recall that we compute over a witness (see Section 2.1), and it is comprised of authenticated tags. We refer to the authenticated tags, witness, or commitment of R using $[R]$ and use these terms interchangeably. After the commitment protocol, $\mathcal{P}$ and $\mathcal{V}$ each have a partition of $[R]$ with which they compute proofs, $[R]_{\mathcal{P}}$ and $[R]_{\mathcal{V}}$ respectively. In addition, $\mathcal{V}$ submits a SQL query Q to the engine. After evaluating Q interactively, $\mathcal{P}$ sends Q's answer, A, to $\mathcal{V}$. If our query has input tables R and S, we prove $C([R], [S]) = 1$.

3.1 Security Model

Recall that ZKSQL verifies queries under the commit-and-prove paradigm. Our proofs offer the following properties:

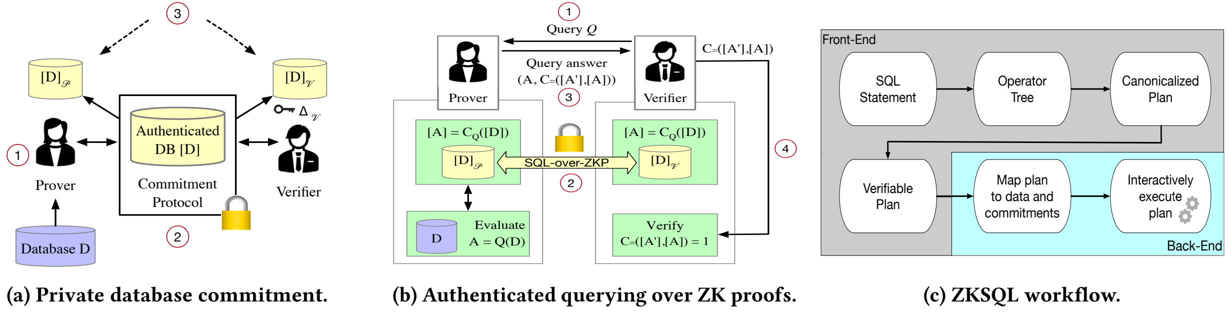
(1) **Correctness.** If both parties participate honestly in the protocol, $\mathcal{V}$ will be convinced of true statements alone.
(2) **Soundness.** If $\mathcal{P}$ tries to prove a false claim about w, $\mathcal{V}$ will catch them with all but negligible probability.
(3) **Zero knowledge.** Even a misbehaving $\mathcal{V}$ learns only the query answer and its truthfulness, but nothing else about w.

Let's consider ZK proofs as an ideal functionality $\mathcal{F}_{\text{ZK}}$ in Figure 1. This is analogous to a trusted third party who convinces $\mathcal{V}$ that a statement is true. It enables $\mathcal{P}$ to construct commitments of its inputs and facilitates provable computation of one or more circuits with respect to input data x. We use these building blocks in our operator-level proofs in Section 4.

The Input method enables $\mathcal{P}$ to generate authenticated tags with which we will construct our proofs or generate the witness w. The prover sends the oracle a bitstring of its input, x. If we query over our entire database, we send D to the functionality. It generates tags for $[x]$ and sends them to both parties. Similarly, the parties prepare public literals as inputs for proofs with the Const method.

The Compute method evaluates circuits over the committed tags, $[x]$. The circuit is comprised of logical gates, such as AND and OR, and the oracle traverses the circuit in topological order. The output of a Compute call is itself an array of authenticated tags. This makes it possible to compose operators one after another by passing the output tags of a child as input to its parent. In the ideal world functionality, the oracle recalls the tags it stored previously, x, and computes C over x to produce y. It then generates commitments for this answer, $[y]$, and sends them to $\mathcal{P}$ and $\mathcal{V}$.

The Verify method confirms that the inputs satisfy the relationship codified in the circuit. We can invoke this method on the query answer, A, or prove properties about a query's intermediate results. For example, after a sort operator, we may use a circuit to iteratively



(a) Private database commitment. **(b) Authenticated querying over ZK proofs.** **(c) ZKSQL workflow.**

Figure 2: ZKSQL workflow in commit-and-prove paradigm from front end to back end

test if two adjacent output tuples are in ascending order by invoking $\mathsf{Verify}(C_{\geq}, t_i, t_{i-1})$. Say we are proving $f(x) = y$. The functionality will return 1 to $\mathcal{V}$ iff $f(x) = y$. For our queries, we prove $[A]$= (Input, A) or that the output of computing our proof is equal to the result we reveal to $\mathcal{V}$.

Guarantees. In ZKSQL, $\mathcal{P}$ and $\mathcal{V}$ instantiate $\mathcal{F}_{\mathsf{ZK}}$ with VOLE-based, interactive ZK protocols [5, 29, 30] although our results generalize to other circuit-based ones. Circuits are the dominant representation for ZK proofs. In terms of security, the universal composability (UC) [10] is a framework to analyze cryptographic protocols. It evaluates whether the security properties are preserved when protocols are composed arbitrarily. It benefits the modular design of our protocols. Owing to UC, we guarantee the soundness, completeness, and zero-knowledge of these protocols against a malicious, static adversary as follows. For any *probabilistic polynomial time (PPT)* adversary, there exists a PPT simulator. For any environment with an arbitrary auxiliary input, if the output distribution of the environment in the *real-world* execution where the parties interact with the adversary and execute a protocol is *computationally indistinguishable* from the output distribution of the environment in the *ideal-world* execution where the parties interact with the simulator and $\mathcal{F}_{\mathsf{ZK}}$, it means the protocol *UC-realizes* $\mathcal{F}_{\mathsf{ZK}}$. Our protocols in Section 4 consists of circuit-based and set-based components. The circuit-based components are direct instantiations of the functionality $\mathcal{F}_{\mathsf{ZK}}$, thus we refer the reader to [30] for detailed security analysis. Our set-based protocols construct novel ZK proofs of set operations over $\mathcal{F}_{\mathsf{ZK}}$. Their security analysis is in Section 4.1.

In this system, $\mathcal{P}$ alone has access to the query inputs, D. From participating in an interactive proof, $\mathcal{V}$ learns the answer to their query, A, accompanied by a proof confirming its correctness and completeness. In order to run a query, both parties need some background information. In particular, the table definitions of D –including column names, types, and schema constraints–are public. In addition, the table cardinalities, the query and its DAG, and the query answer are known to both parties.

Speaking imprecisely, ZKSQL supports two adversaries –a malicious $\mathcal{P}$ and a malicious $\mathcal{V}$. Owing to $\mathcal{F}_{\mathsf{ZK}}$, ZKSQL guarantees the soundness of all query answers. A malicious $\mathcal{P}$ cannot compromise the integrity of query evaluation. $\mathcal{V}$ will abort the protocol if it detects cheating on the part of $\mathcal{P}$. Owing to our commitment protocol, which implements Input and Const in $\mathcal{F}_{\mathsf{ZK}}$, it is impossible for the prover to change the contents of D after this setup step. Our operator-at-a-time proofs provide an oblivious "chain of custody" from our authenticated tags, $[R]$ and $[S]$ to A. Hence, ZKSQL guarantees the zero-knowledge property of our aims above. A malicious $\mathcal{V}$ cannot steal any information about individual tuples during query processing since our proving logic is oblivious. Section 4.2 describes how each proof is data-independent and how we pad intermediate results to protect their cardinalities. For example, a ZKSQL filter operator has an output cardinality equal to that of its input. Users may optionally truncate this output if desired using public constants. For primary key-foreign key joins, their output cardinality is equal to that of the foreign key input since each one can match at most one primary key row. The dummy tags introduced in Section 2.2 enable us to obliviously process them.

3.2 Workflow

Our workflow has two parts. First, the engine sets up the commitments over which we will evaluate our zero-knowledge proofs, $[D]$, as shown in Figure 2(a). After that, $\mathcal{P}$ and $\mathcal{V}$ interactively verify the answer to one or more SQL queries with respect to $[D]$, visualized in Figure 2(b). As the example introduced in Section 1, a university with its private student-record database is $\mathcal{P}$ and DoE is $\mathcal{V}$.

Before ZKSQL may process its first query, the parties complete a setup phase with which $\mathcal{P}$ commits its private database, D. Figure 2(a) illustrates how we bootstrap our query verification by generating these authenticated tags. The tags serve as an immutable, agreed-upon starting point from which our authenticated queries begin. This makes it possible for $\mathcal{V}$ to confirm multiple query answers referring to the same dataset. $\mathcal{P}$ starts by inputting its private database D to this cryptographic protocol (Step ①). This is equivalent to invoking (Input, D) in $\mathcal{F}_{\mathsf{ZK}}$. $\mathcal{P}$ and $\mathcal{V}$ generate the tags one for each bit in D (Step ②). At the end of the protocol, each party knows only a partition of $[D]$, where $\mathcal{P}$ alone holds $[D]_{\mathcal{P}}$, and $\mathcal{V}$ alone holds $[D]_{\mathcal{V}}$ and an authentication key, $\Delta_{\mathcal{V}}$ respectively (Step ③). The relationship of $[D]_{\mathcal{P}}$ and $[D]_{\mathcal{V}}$ is:

$$[D]_{\mathcal{P}} = [D]_{\mathcal{V}} + D \cdot \Delta_{\mathcal{V}}.$$

We do this process once for any number of queries over D. The entire set of tags, $[D]$, is known to no one. In the example of student statistics, after commitment, the university knows $[D]_{\mathcal{P}}$, and DoE only knows $[D]_{\mathcal{V}}$ and $\Delta_{\mathcal{V}}$, but only the university knows plaintext database D with student records.

Now, this system is ready to accept its first query for verification. Figure 2(b) outlines this process. First, $\mathcal{V}$ sends a SQL statement over the schema of D (Step ①). When ZKSQL processes a query, the prover maintains a dual representation of each table or intermediate result. $\mathcal{V}$ works with only their partition of the committed tags (Step ②). Hence, if we are computing over an intermediate result R, $\mathcal{P}$ has R and $[R]_{\mathcal{P}}$ and the $\mathcal{V}$ starts with $[R]_{\mathcal{V}}$. With

each operator in the query evaluation pipeline, $\mathcal{P}$ and $\mathcal{V}$ work in lockstep to compute each one. Each operator produces an array of authenticated tags as its output. Speaking imprecisely, in $\mathcal{F}_{\mathsf{ZK}}$ evaluating an unary operator op is implementing the functionality $[T]$:=Compute(C_{op}, $[R]$). $\mathcal{P}$ computes T with conventional means.

When an operator proves properties about the output of their computation, such as verifying that a sort is in ascending order, the parties create new committed bits for the operator output T and they prove that its contents have the expected relationship with the operator's inputs. When we finish evaluating the root operator of the query tree, we call its output A. $\mathcal{P}$ and $\mathcal{V}$ each have their partitions of $[A]$. Both parties know A and they commit it using the protocol $[\mathrm{A}']$:=Const(A) in $\mathcal{F}_{\mathsf{ZK}}$ (Step ③). They then create a circuit, $C_=$ that checks the bitwise equality between two sets of tags. $\mathcal{P}$ and $\mathcal{V}$ invoke (Verify, $C_=$,[A], $[\mathrm{A}']$). If this returns 1 to $\mathcal{V}$, then the proof is successful (Step ④).

For example, the DoE wishes to compute the average annual costs by family income for a student at university U. The university runs ZKSQL on top of their DBMS. Before querying, U and the DoE work together to commit their database, D_U using the steps in Figure 2(a). The DoE submits their query: `SELECT fi.income_bracket, avg(s.cost) FROM student_cost s JOIN family_income fi ON s.sid = fi.sid AND s.year = fi.year GROUP BY fi.income_bracket ORDER BY fi.income_bracket` to ZKSQL. The two parties follow the steps in Figure 2(b) to answer the DoE's query. ZKSQL parses the SQL statement into a DAG of ZKSQL operators, described in Section 4 by invoking the functionality described in Figure 3. It first invokes the Join operator to bring together its input tables to match the annual student costs with their corresonding family incomes. ZKSQL interactively proves to the DoE that the committed matching tuples alone are passed on to its parent operator. The Aggregate operator then obliviously calculates the average family contribution for each income bracket and sorts its results. Last, the DoE receives the query answer, A, and its tags. The parties invoke the Verify protocol to confirm the answer is correct and sound.

SQL to ZK Proof Processing. Now that we've covered the cryptographic workflow that powers our authenticated query evaluation, let's zoom into the techniques we use to translate SQL queries into this circuit-based framework. Figure 2(c) outlines this process.

ZKSQL has a two-part system for its query evaluation. The front end checks that the query is syntactically correct with respect to D's schema, extracts the DAG, and regularizes this execution plan. It is written in Java. We parse queries using Apache Calcite [6] rather than creating our own SQL parser. We canonicalize the plan by applying a series of transformations that make it run more efficiently in circuits. Our transformations include projecting out all unnecessary attributes eagerly, rewriting to eliminate common table expressions, and combining filters to minimize data passes.

Each operator type (e.g., Join, Filter) in ZKSQL has at least one proof template, described in Section 4.2. Proof templates are protocols for constructing ZK proofs for the transformations an operator performs. We parameterize them with the expressions and fields specified by the DAG. This constitutes our verifiable plan. The front end outputs this plan as a JSON file for the back end. $\mathcal{P}$ and $\mathcal{V}$ both have proceeded from the same JSON file.

Table 2: Baseline ZK proof overhead. Runtime (in secs) of plain query evaluation vs naïve ZK baseline on TPC-H.

	*Q*1	*Q*3	*Q*5	*Q*8	*Q*9	*Q*18
Plaintext	0.05	0.02	0.03	0.01	0.13	0.05
Circuit-Only	777	24,130	38,106	32,040	43,562	126,168
Slowdown	15,540×	1,206,500×	1,270,200×	3,204,000×	335,092×	2,523,360×

The back end parses this query execution plan and instantiates the operators it specifies in a query tree. It then sets up $\mathcal{P}$'s dual representation of its inputs (e.g., R and $[R]$) while the verifier reads their corresponding tags. Both parties work interactively to evaluate the query tree bottom up to produce A and $[A]$. They validate the answer as described above.

4 ZKSQL

In this work, we propose efficient ZK proofs for query evaluation in database, named *ZKSQL*. More specifically, this system does:

Given a query Q, a prover $\mathcal{P}$ with access to a set of relations D works interactively to produce and verify an answer A to a verifier $\mathcal{V}$.

$\mathcal{P}$ accepts Q and commits the authenticated tables $[D]$ to $\mathcal{V}$. $\mathcal{P}$ and $\mathcal{V}$ work interactively to generate authenticated A. This engine provides a generic method for generating proofs for each operator in this tree. In this work, we focus on Project, Filter, Join, Sort and Aggregate operators and describe them in Section 4.2. One key optimization here is decoupling operator evaluation from proving. Doing so reduces the footprint of the query that it evaluates with expensive ZK circuits.

Beyond pure circuit implementation for each operator, some operators allow $\mathcal{P}$ to compute the intermediate result by its local computation on plaintext table and $\mathcal{V}$ to interactively verify the correctness of the result with $\mathcal{P}$. For example, in Sort, it has a table of the tags of sorted tuples, $[T]$, and verifies that adjacent ones satisfy the sort definition S. In other words, $\forall i \in [|T|-1]$: (Verify, $C_<$, $[t_i]$, $[t_{i+1}]$) with respect to S. We further propose set operations (see Section 4.1) to assist with the verification. For example, we also need to verify $[T]$ contains exactly same rows as input of sort, $[R]$, where $\forall [t_i], \exists [r_j] : [t_i] = [r_j]$ and vice versa by (Equality, [T],[R]) shown in Figure 3.

To motivate this work, Table 2 compares unauthenticated and insecure query evaluation with naïvely constructing proofs using circuits alone, called *Circuit-Only*. This baseline is analogous to translating the logic of a conventional DBMS operator directly into circuits and then using `EMP Toolkit`'s VOLE-ZK protocols to trace its gates for the proof. It is clear that simply doing this one-to-one substitution creates a slowdown of 5-6 orders of magnitude. As we will confirm in Section 5.4 proving properties about a query's intermediate results instead of verifying its execution traces cuts down substantially on the costs of these proofs.

4.1 Set Operations

One key insight in this work is that although queries often have numerous compute-intensive operations like sorting and joining (and this task is expensive to compute a ZK proof by following the computation as a circuit), we can efficiently verify the outputs by expressing them as polynomials. More specifically, we show protocols for efficiently performing set operations over tuples. To formally describe our ZK operations on set, Figure 3 describes the ideal functionality $\mathcal{F}_{\mathsf{ZKSET}}$ for these building blocks.

Functionality $\mathcal{F}_{\text{ZKSET}}$

Equality: Upon receiving (Equality, $[R]$, $[S]$) from both parties, the functionality fetches R and S using $[R]$ and $[S]$, computes $b := (R = S)$ and sends b to $\mathcal{V}$.

Disjoint: Upon receiving (Disjoint, $[R]$, $[S]$) from both parties, the functionality fetches R and S from $[R]$ and $[S]$, computes $b := (R \cap S = \emptyset)$ and sends b to $\mathcal{V}$.

Intersection: Upon receiving (Intersection, $[R]$, $[S]$, $[T]$) from both parties, the functionality fetches R, S and T from $[R]$, $[S]$ and $[T]$, computes $b := (R \cap S = T)$ and sends b to $\mathcal{V}$.

Union: Upon receiving (Union, $[R]$, $[S]$, $[T]$) from both parties, the functionality fetches R, S and T from $[R]$, $[S]$ and $[T]$, computes $b := (R \cup S = T)$ and sends b to $\mathcal{V}$.

Figure 3: Ideal functionality for set operations in ZKSQL.

Figure 4 describes the protocol Π_{ZKSET} following theorem:

THEOREM 4.1. *The protocol Π_{ZKSET} securely realizes the functionality $\mathcal{F}_{\text{ZKSET}}$ in the $(\mathcal{F}_{\text{ZK}}, \mathcal{F}_{\text{ZKSET}})$-hybrid model.*

The protocol regards any input tuples as binary strings and assumes the tuples in the input tables are of the same length m. Note that all four set operations have a complexity of $O(m)$. We show specialized protocols in our design which outperform the generic circuit-based approach, which has a complexity of at least $\Omega(m \log m)$ and practical ones often run in time $O(m \log^2 m)$.

For the purpose of security and efficiency, the set operations manipulate on κ bit tuples where the security parameter κ usually equals to 128. The function *Packing* takes a table consisting of m-bit ($m > \kappa$) tuples and converts them into κ-bit tuples via universal hashing. This packing step compresses large tuples into κ bits apiece before we process them in the set operations.

Equality. The ZK proof of set *equality* efficiently proves that the unordered tables $R = \{[r_i]\}_{i \in [n]}$ and $S = \{[s_i]\}_{i \in [n]}$ are equal. The straw-man approach executes a generic circuit computation to first sort two tables and then compare the tuples at each entry. It incurs $O(m \log m)$ computation and communication overhead. Our key idea is to convert the function from the circuit representation to a polynomial representation. In detail, $\mathcal{V}$ samples a uniform $a \leftarrow \mathbb{F}_{2^\kappa}$ and sends it to $\mathcal{P}$. $\mathcal{P}$ and $\mathcal{V}$ compute and open the value $\prod_{i=1}^{n}([r_i] - a) - \prod_{i=1}^{n}([s_i] - a)$, which equals to 0 if two tables are equal. The online communication cost for this method is $2m + 2$ elements in $\mathbb{F}_{2^\kappa}$. The idea comes from the work of Franzese et al. [13]. Namely, the authenticated tables R and S constitute a degree-n polynomial $f(X) := \prod_{i=1}^{n}(r_i - X) - \prod_{i=1}^{n}(s_i - X)$. If $R = S$, $f(X)$ always equals to 0. Otherwise, for any evaluation point a, $f(a)$ equals to 0 only if a happens to be a root of $f(X)$. Because a is sampled randomly, this incident happens with probability $p \leq n/2^\kappa$, which is negligible in our instances when $\kappa = 128$.

Disjoint. The ZK proof of set *disjoint* relies on the set equality check and the generic comparison circuits. Along with the input tables (R, S), $\mathcal{P}$ additionally defines and commits to a table T, which is generated by first concatenating the tables R and S, and then sorting the combined table. Two parties invoke the set *equality* protocol to check if T equals to $R||S$, then invoke the generic comparison circuit $|T| - 1$ times to check if for any two consecutive tuple $[t_i]$ and $[t_{i+1}]$ in table T, it satisfy that $t_i < t_{i+1}$. If two tables are not disjoint, a malicious prover may either replace the overlapped tuples in one table when constructing T or place the overlapped tuples apart after sorting. The former would be caught by the equality check and the latter would be caught by comparison checks.

ZK Set Protocols Π_{ZKSET}

Parameters: $\mathbb{F}_{2^m}$ and $\mathbb{F}_{2^\kappa}$ denote binary extension fields of degree m and κ. κ is the security parameter and is instantiated as 128 in practice. All set elements are m bits in length and $m > k$.

Packing(T, m):

(1) Denote $d = \lceil m/\kappa \rceil$, $\mathcal{V}$ uniformly samples coefficients $\{c_i\}_{i \in [d]} \leftarrow \mathbb{F}_{2^\kappa}^d$ and sends them to $\mathcal{P}$.
(2) Let $n = |\mathbf{T}|$, and parse $\mathbf{T} \leftarrow \{[t_1], \cdots, [t_n]\}$, where for $i \in [n]$, $[t_i] \leftarrow \{[t_i^1], \cdots, [t_i^m]\}$ contains m authenticated bits.
(3) $\mathcal{P}$ and $\mathcal{V}$ compute: $[t'_i] \leftarrow \sum_{k=1}^{d} c_k \cdot (\sum_{j=1}^{\kappa} [t_i^{k \cdot \kappa + j}] \cdot X^{j-1})$ for $i \in [n]$ and output $T \leftarrow \{[t'_i]\}_{i \in [n]}$.

Equality(R, S):

(1) Let $n = |R|$; if $n \neq |S|$, two parties abort.
(2) Compute $R \leftarrow \text{Packing}(R, m)$ and $S \leftarrow \text{Packing}(S, m)$.
(3) $\mathcal{V}$ samples a uniformly from $\mathbb{F}_{2^\kappa}$ and sends it to $\mathcal{P}$.
(4) $\mathcal{P}$ proves to $\mathcal{V}$ that $\prod_{i=1}^{n}([r_i] - a) = \prod_{i=1}^{n}([s_i] - a)$.

Disjoint(R, S):

(1) Compute $R \leftarrow \text{Packing}(R, m)$ and $S \leftarrow \text{Packing}(S, m)$.
(2) $\mathcal{P}$ locally computes and authenticates $T \leftarrow \text{Sort}(R||S)$.
(3) Invoke $\text{Equality}(R||S, T)$.
(4) $\mathcal{P}$ proves to $\mathcal{V}$ in zero-knowledge that T is sorted.

Intersection(R, S, T):

(1) Compute $R \leftarrow \text{Packing}(R, m)$, $S \leftarrow \text{Packing}(S, m)$ and $T \leftarrow \text{Packing}(T, m)$.
(2) $\mathcal{P}$ inputs the sets $P \leftarrow R \backslash T$ and $Q \leftarrow S \backslash T$.
(3) Invoke $\text{Equality}(P||T, R)$ and $\text{Equality}(Q||T, S)$.
(4) Invoke $\text{Disjoint}(P, Q)$.

Union(R, S, T):

(1) Compute $R \leftarrow \text{Packing}(R, m)$, $S \leftarrow \text{Packing}(S, m)$ and $T \leftarrow \text{Packing}(T, m)$.
(2) $\mathcal{P}$ inputs the set $P \leftarrow R \cap S$.
(3) Invoke $\text{Intersection}(R, S, P)$ and $\text{Equality}(R||S, P||T)$.

Figure 4: Set verification protocols for Π_{ZKSET}.

Intersection. The ZK proof of set *intersection* is done by invoking the proof of *equality*. Along with the input tables ($[R]$, $[S]$, $[T]$), $\mathcal{P}$ additional define and commits to two tables $P := R \backslash T$ and $Q := S \backslash T$. It first proves that the tables P and Q are honestly constructed by two invocations of the set *equality* protocols. A malicious prover will also be caught at this step if it does not satisfy that $T \subseteq R$ and $T \subseteq S$. Then it proves that T and S are disjoint. A malicious prover will be caught at this step if T does not contain all tuples that R and S have in common.

Union. Though it is not used in our ZKSQL protocol, we propose a set *union* protocol to complete our ZKSET tool-kits. $\mathcal{P}$ locally computes the intersection of R and S and denotes it as P. $\mathcal{P}$ and $\mathcal{V}$ invoke the set *intersection* protocol to check the honesty of $\mathcal{P}$. Then they invoke a set *equality* protocol to check if $(R||S) = (P||T)$. If T contains tuples that do not belong to $R||S$, the malicious prover needs to cheat in the equality check. If T does not contain a tuple that belongs to $R||S$, this tuple must be replaced by another tuple to make the total size of $(R||S)$ and $(P||T)$ equal. Again, this action will be caught in the equality check.

The above protocol allows proving in ZK relationship between sets with cost linear to the set size but it comes with some caveats. In particular, we keep obliviousness by treating dummy tuples

as tombstones (see Section 2.2), equivalent to pad intermediate results up to its maximum bound. Section 5.3 shows that no padding approach is feasible while it leaks information of cardinalities.

4.1.1 Security Analysis. The simulation-based proof is provided for analyzing the zero-knowledge property of the set-based protocols. We construct a simulator $\mathcal{S}$ who interacts with a malicious verifier $\mathcal{A}$ in the ideal world and simulate a view for $\mathcal{A}$, such that there is only a negligible probability for $\mathcal{A}$ to distinguish between the simulated views in the ideal world and the views when interacting with a real-world $\mathcal{P}$.

$\mathcal{S}$ emulates the functionality $\mathcal{F}_{\mathsf{ZK}}$, which samples the global key Δ for $\mathcal{A}$. $\mathcal{F}_{\mathsf{ZK}}$ also handles the authentication of inputs and the computation of circuits. On simulating the function Equality, $\mathcal{S}$ acts as an honest $\mathcal{P}$ until it receives the challenge a from $\mathcal{A}$. $\mathcal{S}$ computes $\delta := \prod_{i=1}^{n}([r_i] - a) - \prod_{i=1}^{n}([s_i] - a)$. If $\delta = 0$, it follows the protocol instructions. Otherwise it fetches the MAC m_δ of δ recorded in $\mathcal{F}_{\mathsf{ZK}}$ and sends $m'_\delta := m_\delta + \delta \cdot \Delta$ to $\mathcal{A}$. On simulating the function Disjoint, $\mathcal{S}$ constructs a sorted set T which contains distinct values. $\mathcal{S}$ simulates the equality check between $R||S$ and T, and proves in ZK that T is sorted. On simulating the functions Intersection and Union, $\mathcal{S}$ acts as an honest $\mathcal{P}$. Whenever invoking the functions Equality and Disjoint, it simulates the view of $\mathcal{A}$ as described above. This concludes the simulation for any $\mathcal{V}$ that is corrupted by a malicious adversary.

The view of a malicious prover can also be simulated by a simulator $\mathcal{S}$ who emulates $\mathcal{F}_{\mathsf{ZK}}$, extracts the input of $\mathcal{A}$, and sends it to $\mathcal{F}_{\mathsf{ZKSET}}$. The soundness error, in this case, is the probability that $\mathcal{A}$ who holds invalid inputs convinces $\mathcal{V}$ to accept the proof in the real world. As analyzed earlier in this section, the soundness error for the Equality function is bounded by $n/2^\kappa$ for table size n. We instantiate the circuit computation of $\mathcal{F}_{\mathsf{ZK}}$ by the protocol of Yang et al. [30]. The soundness error of the circuit computation is $(t+3)/2^\kappa$ for a circuit consisting of t non-linear gates (e.g. multiplication gates or logic-AND gates). The function Disjoint invokes a Equality check and $O(n)$ comparison circuits. So its soundness error is bounded by $O(n/2^\kappa)$. Since the functions Intersection and Union are solely constructed by invoking Equality and Disjoint checks for constant number of times, their soundness errors are also bounded by $O(n/2^\kappa)$.

4.2 Operators

In this section, we describe ZK proofs of the query Q in the operator level since Q can be decomposed into a tree of operators including Project, Filter, Join, Sort, and Aggregate. For each operator, we design ZK protocols to prove the correctness and soundness of its intermediate result. After executing the last operator in the parse tree, $\mathcal{P}$ with access to the database D only provides an authenticated answer A to $\mathcal{V}$ without leaking any information about those intermediate results during query evaluation. Figure 5 shows the ideal functionality, $\mathcal{F}_{\mathsf{ZKSQL}}$ for operators. Figures 6 shows the protocol $\Pi_{CIRCUIT}$ using pure ZK circuits and Figure 7 shows the protocol Π_{SET_OP} integrating set operations introduced in Section 4.1.

We have the following theorem:

THEOREM 4.2. The protocols $\Pi_{CIRCUIT}$ and Π_{SET_OP} securely realizes the functionality $\mathcal{F}_{\mathsf{ZKSQL}}$ in the $(\mathcal{F}_{\mathsf{ZK}}, \mathcal{F}_{\mathsf{ZKSQL}})$-hybrid model.

We now detail these operator-level protocols.

Functionality $\mathcal{F}_{\mathsf{ZKSQL}}$

Query: On input (Query, Q, $[D]$) from both parties, the functionality fetches the set of relations D using $[D]$, computes $A \leftarrow Q(D)$ and sends A to the verifier $\mathcal{V}$.

Project: On input (Project, $[R]$, $\mathcal{E}$) from both parties, where the expressions $\mathcal{E}$ specify its projections, the functionality fetches the relation R using $[R]$, computes $T \leftarrow \pi_{\mathcal{E}}(R)$, generates the tags $[T]$ and sends them to both parties.

Sort: On input (Sort, $[R]$, S) from both parties, where S specifies the attributes and directionality of the sort, the functionality fetches the relation R using $[R]$, computes $T \leftarrow \tau_{\mathrm{S}}(R)$, generates the tags $[T]$ and sends them to both parties.

Filter: On input (Filter, $[R]$, P, m) from both parties, where p is the filter predicate and m is the desired output cardinality, the functionality fetches the relation R using $[R]$ and computes $T \leftarrow \sigma_p(R)$. It generates the tags $[T]$ and sends them to both parties. If $|T| > m$, it truncates T to length m by sorting it on the dummy tag and deleting the last $|T| - m$ tuples.

Join: On input (Join, $[R]$, $[S]$, P, m) from both parties, where p is the join predicate and m is desired output cardinality, the functionality fetches the relations R, S using $[R]$, $[S]$ and computes $T \leftarrow R \bowtie_p S$. It generates the tags $[T]$ and sends to both parties.

Aggregate: On input (Aggregate, $[R]$, $\mathcal{G}$, **A**) from both parties, where $\mathcal{G}$ specifies the group-by expressions and **A** is the aggregators, the functionality fetches the relation R using $[R]$, computes $T \leftarrow \mathrm{A}_{\mathcal{G}}(R)$, generates the tags $[T]$, and sends them to both parties.

Figure 5: Ideal functionality for querying in ZKSQL.

4.2.1 Circuit-Based Protocols. We will first examine the protocols with which we prove the correctness and completeness of their outputs purely in circuits shown in Figure 6. We do this one tuple at a time. The logic of these proofs very closely follows that of unauthenticated implementations of these operators.

Projection. This protocol begins when $\mathcal{F}_{\mathsf{ZKSQL}}$ receives the command (Project, $[R]$, $\mathcal{E}$) from both parties. $[R]$ serves as the input to this operator. Like all the rest operators, it may either be a table in the database or the output of another operator. An expression, $e_i \in \mathcal{E}$, either refers to a column in the source table or it applies a transformation, either arithmetic or boolean, to its input columns once per tuple. Locally, $\mathcal{P}$ computes $T \leftarrow \pi_{\mathcal{E}}(R)$.

We perform a ZK proof for this process to compute $[T] \leftarrow$ Project$(R, \mathcal{E})$ in Figure 6. Both parties parse the authenticated relation $[R]$ into authenticated tuples $([r_1], \ldots, [r_n])$ on the first line of the protocol. Next, for each tuple $[r_i] \in [R]$, both parties evaluate $\mathcal{E}$ to produce $[t_i]$ on Line 2. If the projection evaluates any expression, it does so using a circuit. If $e_j \in \mathcal{E}$ is an arithmetic or boolean expression, we invoke (Compute, C_{e_j}, $[r_i]$) in $\mathcal{F}_{\mathsf{ZK}}$, and receive an authenticated field $[t_{i,j}]$. Otherwise, we copy the authenticated bits for $[t_{i,j}]$ from the source field in $[r_i]$. The protocol outputs all tuples from the circuit as the authenticated output relation $[T]$ to both parties.

For example, consider (Project, $[R]$, {\$2, \$1 + \$3}). Its output is T, and T's first column contains the second column of R. For this, ZKSQL simply copies the committed bits for this tuple from R to T thereby incurring minor overhead. These copies are trivially provably correct by checking the equality of their bits. The second column of T is the sum of the values in the first and third fields of each tuple in R. We apply a circuit, C_{e_2} to verify this transformation

Circuit-Based Operator Protocols $\Pi_{CIRCUIT}$

Project$(R, \mathcal{E})$:

(1) Both parties parse $\mathbf{R} \rightarrow \{[r_1], \cdots, [r_n]\}$
(2) $\forall i \in [n]$, $\mathcal{P}$ and $\mathcal{V}$ send (Compute, $C_{\mathcal{E}}$, $[r_i]$) to $\mathcal{F}_{\mathsf{ZK}}$ and receive $[t_i]$, where $C_{\mathcal{E}}$ is a circuit for the projections $\mathcal{E}$. Both parties verify the result of any non-trivial expression $e_j \in \mathcal{E}$ by calling $b :=$ (verify, C_{e_j}, $\{[r_i], [t_i]\}$) returning b to $\mathcal{V}$. The verifier aborts of b = 0.
(3) Output $\mathbf{T} \leftarrow \{[t_{i \in [n]}]\}$.

Filter(R, p, m):

(1) Both parties parse $\mathbf{R} \rightarrow \{[r_1], \cdots, [r_n]\}$
(2) **Predicate check:** $\forall i \in [n]$ both parties send (Compute, C_p, $[r_i]$) to $\mathcal{F}_{\mathsf{ZK}}$ and receive $[t_i]$. Both parties verify the selection result by invoking $b :=$ (Verify, C_p, $\{[r_i], [t_{i,dummy}]\}$) returning b to $\mathcal{V}$. C_p confirms that $[t_{i,dummy}] = 0$ (not dummy) iff $[r_i]$ satisfies p and $\neg[r_{i,dummy}]$. The verifier aborts of b = 0.
(3) If $m < |T|$, both parties compute $T' \leftarrow \mathsf{Sort}(T, \mathsf{S}_{dummy})$, where S_{dummy} sorts T in *ascending* order on T's dummy tags. Otherwise, output $\mathbf{T} \leftarrow \{[t_{i \in [n]}]\}$.
(4) Output $\mathbf{T} \leftarrow \{[t_{i \in [m]}]\}$.

Aggregate$(R, \mathcal{G}, A)$:

(1) Both parties compute $R_{\mathcal{G}} \leftarrow \mathsf{Sort}(R, \mathsf{S}_{\mathcal{G}})$ where $\mathsf{S}_{\mathcal{G}}$ sorts R with respect to the group-by attributes $\mathcal{G}$. It returns $[R_{\mathcal{G}}]$ to both parties.
(2) Parse $[R_{\mathcal{G}}] \rightarrow \{[r_1], \cdots, [r_n]\}$
(3) $\forall i \in [n]$, create an output tuple $[t_i]$ and initialize its first $|\mathcal{G}|$ fields with $[r_i]$'s group-by column values. $\forall j \in [|\mathbf{A}|]$, both parties send (Compute, $C_{\mathbf{A}_j}$, $[r_i]$) and receive $[t_{i+|\mathcal{G}|,j}]$, where $C_{\mathbf{A}_j}$ is a circuit for the jth aggregator.
(4) Output $\mathbf{T} \leftarrow \{[t_{i \in [n]}]\}$.

Figure 6: ZKSQL boolean circuit protocols.

for each row of T. Hence, after computing $[t_{i,2}]$, both parties call (Verify, C_{e_2}, $[r_i]$, $[t_{i,2}]$) in $\mathcal{F}_{\mathsf{ZK}}$ to confirm the soundness of this field. $\mathcal{V}$ aborts if it receives $b = 0$. Naturally, the output of this operator has the same tuple count as its input table, $|T| = |R|$.

Filter. When $\mathcal{F}_{\mathsf{ZKSQL}}$ receives the input (Filter, $[R]$, P, m) from both parties, it begins the filter protocol wherein it constructs an output relation $[T]$ that eliminates tuples in $[R]$ that don't satisfy the selection criteria specified in predicate p, a boolean expression condition related to one or more columns in R .

Recall from Section 2.2 that in order to remain oblivious, filter must not reveal its selectivity nor the specific tuples it selects. Hence, in the absence of a low cardinality bound (m), the output table has the same cardinality as its input, i.e., $|T| = |R|$. If the parties input $m = -1$, the operator does not truncate its output. Without truncation, the two tables are identical in the values in their non-dummy rows.

If the filter truncates its output, $m < |R|$, it minimizes the number of non-dummy tuples eliminated by this step. It first sorts T's outputs by their dummy tags in ascending order. If $t_{i,dummy} = 0$ a tuple is real, not a tombstone. Thus this puts all of the dummies at the end. The protocol then deletes the last $|T| - m$ tuples.

In the filter protocol, as shown in Figure 6, both parties first parse the authenticated relation $[R]$ into authenticated tuples $([r_1], \ldots, [r_n])$. For each tuple $[r_i]$, both parties evaluate the predicate circuit, C_p, to produce $[t_i]$ by invoking Compute in $\mathcal{F}_{\mathsf{ZK}}$ as shown on Line 2. Since $[R]$ itself may be the intermediate result of some other operator, if $r_{i,dummy} = 1$ (r_i is already a dummy) it remains a dummy regardless of the result of C_p. Therefore, when we invoke (Verify, C_p, $[r_i]$, $[t_i]$) in $\mathcal{F}_{\mathsf{ZK}}$, it confirms that $[t_{i,dummy}] = 0$, or $[t_i]$ is not a dummy, iff both its input tuple was non-dummy ($\neg r_{i,dummy}$) *and* r_i satisfies p ($C_p([r_i]) = 1$). If $|T| < m$ or $m = -1$ the protocol terminates returning $[T]$ on Line 4. Otherwise, we sort on $[T]$ on its dummy tag, write the ordered output to $[T]'$ and truncate it to m tuples on Line 3. It does so by invoking the Sort protocol in Figure 7 and more in Section 4.2.2.

Aggregation. The protocol starts when $\mathcal{F}_{\mathsf{ZKSQL}}$ receives the input (Aggregate, $[R]$, $\mathcal{G}$, $\mathbf{A}$) from both parties, the protocol begins to evaluate aggregators $\mathbf{A}$ (e.g., SUM, AVG, COUNT) based on group-by attributes $\mathcal{G}$. If it receives an empty set of group-by columns, ZKSQL implements this as a scalar aggregation with one output row. This protocol computes $[T] \leftarrow \mathsf{Aggregate}([R], \mathcal{G}, \mathbf{A})$.

The engine sorts the input table, $[R]$, first with respect to the group-by attributes $\mathcal{G}$ using the sort protocol in Figure 7, and $\mathcal{P}$ makes a linear pass over all sorted table, $[R_{\mathcal{G}}]$, aggregating them one group-by bin at a time, as shown in the first line of Figure 6. We do this so that rows that will be aggregated in the same group-by bin will be adjacent to one another for the aggregation pass.

Next, $\mathcal{P}$ and $\mathcal{V}$ parse $[R_{\mathcal{G}}]$ into a set of tuples $\{[r_1], \ldots, [r_n]\}$. The output of aggregate has the same cardinality as its input ($|R| = |T|$) to avoid leaking the size of the domain of $[R]$'s group-by columns. For each tuple, $[r_i]$, we start by constructing its corresponding output tuple, $[t_i]$. For $[t_i]$, we iterate over each aggregator $\mathbf{A}_j \in \mathbf{A}$ and add its contribution to the current group-by bin on Line 3. If $[r_i]$ is a dummy ($[r_{i,dummy}] = 1$), the aggregator's state is obliviously left unchanged. Otherwise, we update the state according to the aggregator's logic. For example, if we are computing `SUM($2)`, and $[r_i]$ is not a dummy, then we add the second field in $[r_i]$ to the sum's running total. We then output a non-dummy tuple with the current partial aggregates. When we visit the next tuple, $[r_{i+1}]$, if it belongs to the same group-by bin as $[r_i]$, we mark $[t_i]$ as a dummy. Hence, each group-by bin has exactly one non-dummy row in $[T]$.

4.2.2 *Set-Based Protocols.* Pure circuit-based protocol for operators like Sort and Join is not efficient, and it can be optimized by set operations (see Section 4.1) while correctness, soundness and ZK properties hold. Figure 7 introduces set-based protocols for Sort and Join operators. Here rather than tracing the execution of an operator as we did in $\Pi_{CIRCUIT}$, $\mathcal{P}$ computes the operator locally and then uses the public commitment protocol to generate tags for the output ($[T]$). It then constructs ZK proofs over these tags to verify properties about $[T]$. We use $\mathcal{F}_{\mathsf{ZKSET}}$ realized by Π_{ZKSET} for this table-at-a-time verification.

Sort. $\mathcal{F}_{\mathsf{ZKSQL}}$ takes an authenticated relation $[R]$ and a sort definition S from both parties, and produces an authenticated relation $[T]$, which can be expressed as $T \leftarrow \tau_{\mathsf{S}}(R)$. The sort definition, S consists of pairs of expressions (usually column ordinals) and their desired directionality (ascending or descending). For example, $\tau_{\{\$2\uparrow,\$1\downarrow\}}(R)$ has sort criteria by the second column of R in ascending order followed by its first column in descending order.

The protocol begins with $\mathcal{P}$ parsing its private copy of R into tuples and sorting them using conventional techniques shown in Figure 7. $\mathcal{P}$ stores the sorted output as R_{S}. $\mathcal{P}$ and $\mathcal{V}$ then execute

Set-Based Operator Protocols Π_{SET_OP}

Sort(R, S):

(1) $\mathcal{P}$ parses $R \rightarrow \{r_1, \ldots, r_n\}$ and locally sorts it with respect to sort definition S as R_S
(2) $\mathcal{P}$ sends $\mathcal{F}_{\mathsf{ZK}}$ (Input, R_S) and both parties receive $[T]$.
(3) Both parties parse $[T] \rightarrow ([t_1], \ldots, [t_n])$
(4) **Equality check:** $\mathcal{P}$ and $\mathcal{V}$ send (Equality, $[R]$, $[T]$) to $\mathcal{F}_{\mathsf{ZKSET}}$ and $\mathcal{V}$ receives b from $\mathcal{F}_{\mathsf{ZK}}$. If $b = 0$, $\mathcal{V}$ aborts.
(5) **Order check:** $\forall i \in [n-1]$, both parties send (Verify, C_S, $[t_i]$, $[t_{i+1}]$) to $\mathcal{F}_{\mathsf{ZK}}$ and $\mathcal{V}$ receives b, where C_S is a circuit to verify $[t_i] < [t_{i+1}]$'s with respect to $\mathcal{S}$. If $b = 0$, then $\mathcal{V}$ aborts.
(6) Output $[T]$.

Join(R, S, p, m):

(1) $\mathcal{P}$ parses the relations R and S into a set of tuples, $(r_1, \ldots, r_n)$ and $(s_1, \ldots, s_n)$, respectively.
(2) $\mathcal{P}$ locally computes $T \leftarrow R \bowtie_p S$, where p is the join predicate.
(3) $\mathcal{P}$ sends $\mathcal{F}_{\mathsf{ZK}}$ (Input, T) and both parties receive $[T]$.
(4) Both parties parse $[T] \rightarrow ([t_1], \ldots, [t_m])$
(5) **Predicate check:** $\forall i \in [m]$ both parties send (Verify, C_p, $[t_i]$) to $\mathcal{F}_{\mathsf{ZK}}$ and receive b, where $[t_i]$ is (not dummy) iff $b = 1 \wedge \neg r_{i,dummy} \wedge \neg s_{i,dummy}$.
(6) Both parties evaluate $[U] \leftarrow$ (Project, $[T]$, $\{\$1, \ldots, \$\mathcal{W}_R\}$) and $[V] \leftarrow$ (Project, $[T]$, $\{\$(\mathcal{W}_R + 1), \ldots, \$(\mathcal{W}_R + \mathcal{W}_S)\}$). $\mathcal{P}$ locally computes U and V similarly from T.
(7) $\mathcal{P}$ locally computes the set differences and generates tags with $\mathcal{V}$ by calling: $[\Delta_R] \leftarrow$ (Input, $R - U$) and $[\Delta_S] \leftarrow$ (Input, $S - V$).
(8) **Set difference check:** Both parties send (Equality, $([\Delta_R] \mid\mid [U])$, $[R]$) and (Equality, $([\Delta_S] \mid\mid [V])$, $[S]$) to $\mathcal{F}_{\mathsf{ZKSET}}$ and $\mathcal{V}$ receives b_r and b_s. $\mathcal{V}$ aborts if $b_r = 0$ or $b_s = 0$.
(9) Both parties evaluate $[K_R] \leftarrow$ (Project, $[\Delta_R]$, p_R) and $[K_S] \leftarrow$ (Project, $[\Delta_S]$, p_S), where p_R and p_S are R's and S's inputs to p, resp.
(10) **Disjoint check:** Both parties send (Disjoint, $[K_R]$, $[K_S]$) to $\mathcal{F}_{\mathsf{ZKSET}}$ and $\mathcal{V}$ receives b. $\mathcal{V}$ aborts if $b = 0$.
(11) Output $[T]$.

Figure 7: ZKSQL protocols based on set operations.

the Input functionality to generate $[T]$ on Line 2 and both parties parse $[T]$ into rows $\{[t_1], \ldots, [t_n]\}$ on Line 3.

We are now ready to verify the sorted table $[T]$. We need to prove two properties of this table. First, we check set equality to confirm $[T]$ has the same records as the input tags, $[R]$. This ensures that $\mathcal{P}$ does not delete any record, add a spurious one, or modify any one from its local computation. As shown on Line 4, we complete this step by performing the Equality protocol in Figure 4. If $\mathcal{V}$ receives the verification bit $b = 0$, it aborts. Otherwise, we progress on to check the order of the sorted tuples. For each tuple $i \in 1 \ldots n-1$, we verify $[t_i] \leq [t_{i+1}]$ with respect to the sort definition S. Hence, in the next step, we invoke (Verify, C_S, $\{t_i, t_{i+1}\}$) in $\mathcal{F}_{\mathsf{ZK}}$, where C_S returns true if the tuples are in increasing order when sorted by S. If $\mathcal{V}$ receives the verification bit $b = 0$, it aborts. This concludes our proof of the correctness and soundness of this operator.

This optimized sort operator is very important in ZKSQL because this protocol supports the proofs of other types, namely Filter, Join, and Aggregate. In addition, we use this authenticated Sort for the Disjoint proofs in Π_{ZKSET}.

Join. $\mathcal{F}_{\mathsf{ZKSQL}}$ takes two authenticated relations $[R]$ and $[S]$, a join predicate p and an output cardinality m. Then, it produces an authenticated relation $[T]$. The protocol logic is equivalent to $T \leftarrow R \bowtie_p S$ and supports equi-joins alone, although more complex predicates would be possible if we decompose this into a cross product (using `memcpy` to create the cartesian product and filter, i.e., $T \leftarrow \sigma_p(R \times S)$. As in conventional joins, the output schema of T is a concatenation of the columns in R and S such that $\mathcal{W}_T = \mathcal{W}_R + \mathcal{W}_S$

The join protocol in Figure 7 begins with Alice locally parsing R into its tuples $\{r_1, \ldots, r_n\}$. It then locally computes $T \leftarrow R \bowtie_p S$ on Line 2. In ZKSQL we do this with a hash join. T starts out with the true output cardinality that is only visible to $\mathcal{P}$.

We pad T with dummies to protect the selectivity of the join. Recall that for primary key-foreign key joins, the engine automatically truncates them to the length of the foreign key table since primary keys admit no duplicates and therefore each foreign key can match at most one row in the other relation. The protocol uses this public schema information to set T's cardinality accordingly, called keyed join. Other joins follow our *basic* join protocol and for them, T's output cardinality is the size of the cartesian product, $|R| * |S|$. The prover locally pads T with dummies until its size matches this cardinality bound. Next, $\mathcal{P}$ and $\mathcal{V}$ perform the Input protocol on T and both receive $[T]$. In the final step, both parties parse it into $\{[t_1], \ldots, [t_m]\}$.

There are three properties we need to prove to authenticate the join. First, the *predicate check* confirms that each non-dummy tuple in $[T]$ satisfies the join criteria p on Line 5. This is similar to how we verified the filter in the previous section. The *set difference* check proves to Bob that all of our tuples are derived from real rows from the input relations. The *disjoint check* confirms that $\mathcal{P}$ omitted no rows from the join output that should have produced matches. We describe the mechanics of each below.

Before we can prove $[T] = ([R] \bowtie_p [S])$, we need to set up a few more supporting tables. First, we isolate the input tuples that contributed to the join's output using projection. By simply copying the first $\mathcal{W}_R$ columns from each tuple in $[T]$ the join gets $[U] \leftarrow \pi_{\{\$1, \$\mathcal{W}_R\}}([T])$. Likewise, we get the selected tuples from $[S]$ with $[V] \leftarrow \pi_{\{\$(\mathcal{W}_R+1), \$(\mathcal{W}_R+\mathcal{W}_S)\}}([T])$. $\mathcal{P}$ locally computes U and V with respect to T on Line 6.

Next, we need the list of tuples *not* selected by the join. $\mathcal{P}$ locally computes $\Delta_R \leftarrow R - U$ and uses the Input functionality so that both parties have $[\Delta_R]$, which is shown on Line 7. It follows the same procedure to generate tags for $[\Delta_S]$ where $\Delta_S \leftarrow S - V$. We confirm the correctness of Δ_R and Δ_S in the next step.

The set difference check confirms two properties of the dataset on Line 8. First, it shows $U \subseteq R$. If $\mathcal{P}$ inserted tuples into $[T]$ where their columns from $[R]$ were not in this input the protocol aborts. Second, it verifies that $\Delta_R = R - U$. We do so by concatenating Δ_R and U as tags and then Alice and Bob perform the *Equality* protocol in $\mathcal{F}_{\mathsf{ZKSET}}$ to prove $([\Delta_R] || [U]) = [R]$. We compute the same proof $[S]$, $[V]$ and $[\Delta_S]$ to verify that no new values were added.

The disjoint check demonstrates the prover does not omit any output tuples. First, we project the join key columns from the table of tuples that did not contribute to the join's output to generate $[K_R]$ and $[K_S]$. For example, consider a join predicate p is \$1 = \$6 *AND* \$2 = \$7. Moreover, $\mathcal{W}_R = 4$ or R has four columns. The source for our join keys for Δ_R, or p_R, will be {\$1, \$2}. Similarly, p_s

Table 3: Performance of commitment over TPC-H workload.

	P	PS	L	O	S	R	N	C
Runtime (s)	0.59	2.01	11.40	2.51	0.36	0.33	0.33	0.58
Memory (MB)	764	960	1,916	997	723	719	720	764
Comm. Cost (MB)	3.75	4.31	8.06	4.31	3.75	3.75	3.75	3.75

is {\$2, \$3}, subtracting the ordinals for R's four columns. To isolate the unselected join keys, the parties compute $[K_R] \leftarrow \pi_{p_r}(\Delta_R)$ and $[K_S] \leftarrow \pi_{p_s}(\Delta_S)$, shown on Line 9. $\mathcal{P}$ and $\mathcal{V}$ then complete the Disjoint protocol over $[K_R]$ and $[K_S]$ to prove $K_R \cap K_S = \emptyset$, shown on Line 10. In other words, for each join key in R that does not contribute to an output row in T, it has no matches in similarly eliminated rows from S.

5 EXPERIMENTAL RESULTS

In this section, we first describe the implementation of ZKSQL and the setup for experimental evaluation. We then verify the performance characteristics of our ZK proofs over SQL, examining the impact of our set-based proof protocols and operator-at-a-time costs. We examine the performance in a limited-bandwidth environment, scalability with larger data sizes, and the monetary cost.

5.1 Experimental Setup

We implement ZKSQL on top of `EMP Toolkit` [28]. Our prototype uses PostgreSQL as its private database back-end. We evaluate this work on a subset of TPC-H [27] queries, namely Q1, Q3, Q5, Q8, Q9 and Q18. These queries demonstrate our performance under varying degrees of complexity wrt their operator count and the amount of data they access. Authenticated query answers run several orders of magnitude slower than their unverified equivalents (see Table 2). Therefore, our experiments use small instances of the TPC-H database. We measure our scale of the database by the size of the fact table, lineitem, with its dimension tables scaled proportionally as described in the benchmark specification. Our results feature 3 database sizes: *60k Rows*, *120k Rows*, and *240k Rows*, and they have 60k, 120k, and 240k rows in lineitem respectively. We use the larger instances to probe the scalability and pragmatism of this new approach to verifiable databases.

Although ZKSQL supports floats, we converted all floating point operations to 64-bit integer ones in our experiments. Otherwise the floating point operations in projections and aggregates were the dominant cost in our workload. Improving the performance of floats in ZK proofs is orthogonal to this research, so this enabled us to get a clearer picture of the strengths and weaknesses of the entire query lifecycle. Also, we omitted the string pattern matching predicate in Q9 because these complex selection criteria are beyond the scope of this work. In the query execution plan, this eliminates the filter. Since query evaluation is oblivious, the filter's selectivity would not impact the performance of its parent operators.

Each experiment had one host for $\mathcal{P}$ and another for $\mathcal{V}$. We store the input database locally on $\mathcal{P}$ and an empty database instance with the schema alone on the $\mathcal{V}$ host. We deployed on two `AWS EC2 r6i.4xlarge` instances with Ubuntu Server 22.04 LTS, 128 GiB memory, 16 vCPUs and up to 12.5 Gbps connectivity between the parties. Unless otherwise specified, our experiments run on *60k Rows*.

ZKSQL ran in Docker containers to simplify its setup. For query duration experiments, we report the runtime of the verifier since $\mathcal{V}$ finishes last and both parties start at the same time. We also report memory usage and communication cost between parties.

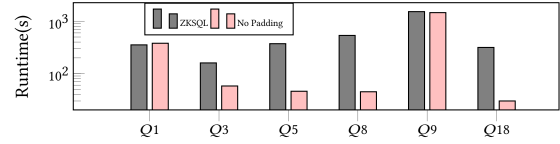


Figure 8: Cost of dummy padding for oblivious querying.

5.2 Setup Costs

We first probe the overhead associated with ZKSQL's authenticated query answers. Recall that the engine commits once for all of its proofs to confirm their answers are all derived from the same data. We measured the performance of committing the TPC-H tables *PART (P), PARTSUPP (PS), LINEITEM (L), ORDERS (O), SUPPLIER (S), REGION (R), NATION (N)* and *CUSTOMER (C)*. The runtime associated wtih our commitments is shown in Table 3.

In the context of our end-to-end workload runtime, this setup cost is 0.6% of our duration and 1.1% of our bandwidth. This one-time setup step is a minor part of our end-to-end cost. We measure our communication costs as the data sent from $\mathcal{P}$ to $\mathcal{V}$ since $\mathcal{V}$ will always send fewer data. With the parties communicating over a gigabit link, this isn't a limiting factor in our performance.

During the commitment protocol, we use at most 2 GB of RAM for lineitem table. Our queries in *60k Rows* used at most 18 GB of RAM in our experiments. Therefore, the memory footprint is not a bottleneck in this system. Taken together, this one-time setup cost of committing the input data works efficiently in ZKSQL.

5.3 Oblivious Proving Overhead

In order to not leak information about data-dependent changes in the control flow of ZKSQL's operators, when an operator deletes a tuple, we replace it with a tombstone that does not contribute to the results of subsequent operators. We maintain a dummy tag for each tuple to keep track of this. In our next experiment, we quantify the overhead associated with this padding.

In this experiment, we compare ZKSQL's oblivious query processing with one that reveals our intermediate result sizes, labeled "No Padding" in Figure 8. To not leak information about which tuples are dummies, it sorts the intermediate results of each operator to put all dummies at the end and then truncates the set as in [3].

Naturally, if our true intermediate result cardinalities are small, our runtime with no padding is faster significantly and we amortize the cost of obliviously deleting the dummies. Our queries with the biggest slowdowns owing to dummy padding are Q8 (12×) and Q18 (10.5×). This makes sense because these queries have many joins and fewer fixed costs like expressions that always require verification with circuits. On the other hand, Q1's performance is on par with its non-oblivious counterpart. Somewhat surprisingly *Q9* has similar performance in both settings despite it having several joins. This is because the Q9 has a cascade of primary key-foreign key joins without filters on their source tables. Hence, the true intermediate cardinalities are quite close to their oblivious versions.

5.4 Set-Based Operators

We now turn our attention to the set-based operators described in Section 4. Recall that they construct their proofs based on the

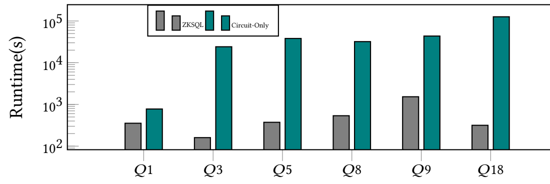


Figure 9: Runtime of ZKSQL vs Circuit-Only baseline.

properties of an operator's results in lieu of following its execution flow in circuits. In particular, our sorts and joins benefited from this technique. Figure 9 shows the runtime of ZKSQL in comparison to the Circuit-Only baseline we introduced in Section 4.

All of our queries, except Q1, realized at least two orders of magnitude over the baseline. Q1, like any query with few set operations, has a performance comparable to the Circuit-Only approach. Its two sort operations give us a speedup of 2×.

Queries with several joins and few expressions realized the biggest gains in performance. The queries with many sorts and aggregates–they sort for grouping–benefit disproportionately from Π_{SET_OP}. Our biggest speedup came from Q18 at 410×. It joins four tables and has two aggregates owing to a nested subquery. Also, it has no expressions from filters or projections. Shown in Section 5.5, circuit-based operators can get costly for these operators.

This confirms our hypothesis that proving properties about the results of our operators is more efficient than tracing their execution in ZK. Sorting in particular produces substantial end-to-end gains because group-by aggregation uses it. Our set-based join operators also significantly speed up this authenticated query processing.

That said, our performance is still several orders of magnitude slower than the plaintext ones in Table 2. For high-stakes scenarios, such as guaranteeing the privacy of records (by not releasing them) while upholding proof of regulatory compliance, this technology is practical since our queries complete in minutes. Nevertheless, its overhead is likely too substantial for very large datasets.

5.5 Operator Performance

We now examine the operator-at-a-time performance of Q3 in Figure 10. We choose this one because it is of medium complexity and includes a mix of circuit-based and set-based operators.

Projection without parameters denotes it only deletes or reorders columns from its input. If a projection proves an expression, we note it in the label such as *Project(revenue)*. This operator computes the field revenue from `l.l_extendedprice * (1 - l.l_discount)`. Since the former projections are simply performing `memcpy` operations, their runtimes are naturally minor compared to ones that require proofs. It is quite costly to prove arithmetic expressions in ZK. Hence, the Project(revenue) operator made up slightly less than a third of our query runtime. More work is needed to optimize how we select the protocols for these expressions.

In contrast, boolean expressions are cheaper in ZK, and thus Filter makes up 3% of our runtime. Despite all of the optimizations we made to joins in ZKSQL, they were still the most time-consuming operators in this query. The two joins make up 57% of our time. This is partially because our prover verifies the join predicate on every output tuple. These boolean expressions are cheaper than math ones, but they still take time. Our sorts and aggregate have

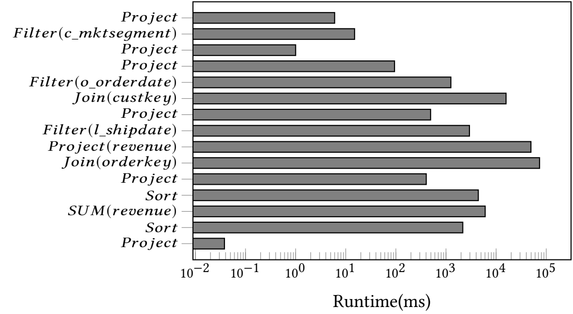


Figure 10: Operator-level performance on TPC-H Q3.

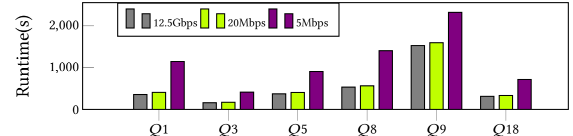


Figure 11: Runtime with decreasing network speeds.

comparatively minor runtimes at 4% and 3% respectively. The aggregate is a simple sum and this addition is comparatively cheaper than the division we saw in the revenue expression.

5.6 Network Throughput

ZKSQL's proofs are interactive, meaning $\mathcal{P}$ and $\mathcal{V}$ go through challenge-response rounds. In this experiment, we quantify the cost of this communication. We use the 12.5Gbps network of our `r6i.4xlarge` instances as our baseline. See how our performance is impacted with constrained bandwidth, such as if our instances were geographically distributed or sharing resources with others, we throttle our network to 20*Mbps* and then 5*Mbps* and measure the runtimes of our query workload.

Figure 11 shows that our protocols largely don't hit a network bottleneck until they reach 5*Mbps*. Moreover, reducing our bandwidth to 5*Mbps* is only about 2× slower than the gigabit baseline. These results indicate that our performance is largely not dependent on variations in network availability.

5.7 Scalability

Although we conduct most of our experiments on the *60k Rows* instance, it does not mean that ZKSQL is only efficient for databases in this size range. To confirm this hypothesis, we also ran this workload with TPC-H's lineitem table at 120k and 240k rows. Figure 12 illustrates our performance in terms of runtime, CPU time, memory, and communication cost. We do so to get a comprehensive view of factors in our performance. We measure CPU time using the Linux `clock()` facility and peak memory utilization with `rusage`.

All of our queries except Q9 have their runtimes double in proportion to their input sizes. Q9 has a 3× slowdown because it has wider rows and this means that we spend more time paging data into the CPU cache. Our optimization in the parser of projecting out unused fields eagerly between each operator prevents other queries from exhibiting similar behavior.

If we consider our CPU utilization as a percentage of our runtime, this measurement is inversely correlated to our memory footprint. Queries with a small memory footprint, such as Q1, consistently run with 90% CPU utilization or greater. Its largest memory footprint is 15 GB. In contrast, Q9 has a much larger memory footprint,

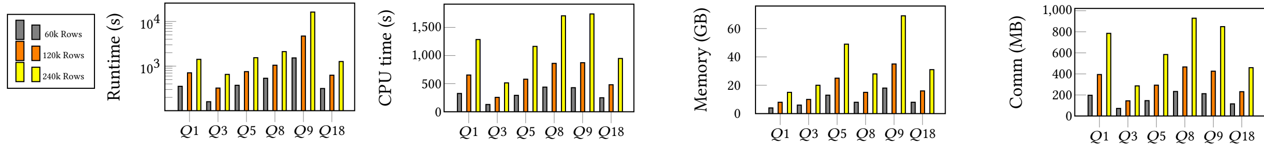


Figure 12: Query performance over data of increasing sizes.

ranging from 18 to 69 GB, and its CPU utilization is 28% in *60k Rows* and gradually reduces to 11% when we reach *240k Rows*. Our communication costs scale linearly with our input sizes. As we saw in Figure 11, our network bandwidth is not substantial enough to be a limiting factor in the current cloud-based infrastructure.

5.8 Financial Cost

So far we've examined ZKSQL's performance through the lens of system performance. Finally, we quantify the financial cost of these authenticated queries in the cloud. Table 4 shows the monetary cost per query of the Circuit-Only approach and ZKSQL over increasing database sizes. Our instances cost \$1.01/hr per instance.

For *60k Rows*, we can see that the Circuit-Only approach takes 1.2× to 392× more monetary cost than ZKSQL, so optimization using set operations is a big step to make the system in practice.

For ZKSQL, the cost of proving these queries grows linearly in the size of `lineitem` table doubles (and the additional tables grow accordingly to uphold key constraints). The one exception to this is Q9, as in Section 5.7. Our reasons for this additional slowdown and cost are the same. More specifically, in Q9 the join with `l_orderkey == o_orderkey` for *60k Rows* has a slowdown of 3× when we scale up to *120k Rows*. Scaling to *240k Rows* follows the same trajectory. In total, *120k Rows* costs 2.5× more than *60k Rows*, and *240k Rows* incurs 2.8× higher charges than *120k Rows* for all queries. This financial cost is reasonable for high-stakes settings.

Table 4: In cloud monetary cost of Circuit-Only vs ZKSQL over data of increasing size.

Query	*Q*1	*Q*3	*Q*5	*Q*8	*Q*9	*Q*18	Total
60k Rows (Circuit-Only)	\$0.22	\$6.77	\$10.69	\$8.99	\$12.22	\$35.39	\$74.28
60k Rows (ZKSQL)	\$0.10	\$0.04	\$0.10	\$0.14	\$0.42	\$0.09	\$0.89
120k Rows (ZKSQL)	\$0.20	\$0.09	\$0.21	\$0.29	\$1.27	\$0.17	\$2.23
240k Rows (ZKSQL)	\$0.39	\$0.18	\$0.43	\$0.57	\$4.39	\$0.34	\$6.30

6 RELATED WORK

There is robust research on verifiable SQL querying using a variety of techniques. CorrectDB [2] and VeriDB [35] offer authenticated SQL querying over trusted hardware. They are efficient because they delegate some of their operations to specialized hardware, but they are not oblivious so they leak some information in their program traces. In contrast, ZKSQL works on generic hardware and builds trust from cryptographic protocols, and prevents information leakage during the proving process.

IntegriDB [34] and vSQL [32] use cryptographic verifiable computation protocols to prove the correctness of a broad class of SQL queries. However, they work in the outsourcing setting, thus only providing integrity but not privacy (i.e., ZK property).

vSQL has an extension for ZK proofs [33], but it does not support ad-hoc queries. This work formalizes that vSQL could be made zero-knowledge but does not address how to translate SQL statements into the cryptographic protocols needed to prove arbitrary, ad-hoc queries, nor the practical efficiency of such an approach. The focus of this work is on making this authenticated, zero-knowledge querying efficient and accessible to a broad audience by generalizing the architecture of relational DBMSs for ZK proofs.

Recently, there has been substantial cryptography research in making ZK proofs more efficient [1, 4, 5, 12, 20, 25, 29, 30]. Our work builds from these results by using EMP Toolkit [28] as ZKSQL's cryptographic back-end. Rather than creating cryptographic primitives for functionality, we introduce protocols for efficient, pragmatic proofs for SQL queries.

7 CONCLUSIONS AND FUTURE WORK

In this work, we propose *ZKSQL*, the first system of its kind for automatically proving the completeness and soundness of answers to ad-hoc SQL queries in ZK. ZKSQL evaluates each query interactively between a prover and a verifier at the operator level. It supports `Project`, `Filter`, `Sort`, `Join` and `Aggregate` operators and these operators seamlessly compose for open-ended querying. We formalized the functionality for these operators in ZK with $\mathcal{F}_{\mathsf{ZKSQL}}$ and detailed protocols for each of them. Beyond directly applying circuit-based protocols once per operator, we introduce set operations over polynomials to optimize our proving of some operators by reasoning over entire sets of tuples rather than doing so one at a time. We then prototyped ZKSQL using a state-of-the-art cryptographic back-end, EMP Toolkit, and evaluate it with `TPC-H`.

The operator optimizations in ZKSQL yielded approximately two orders of magnitude performance improvement on average. Our emphasis was on building blocks within database operators. Future work may include applying these optimizations on top of other ZK paradigms, e.g., based on *MPC-in-the-head* [21]. It is also an interesting direction to explore if it is possible to prove SQL query results without proving all operators individually or for what query verification can be performed much cheaper than with this fine-grained proving. On the other hand, since ZKSQL evaluates each query in an operator-at-a-time fashion sequentially, execution order optimization of operators should be the crucial step along with other query optimization techniques to improve the system further. In addition, `Join` operator of ZKSQL supports two-way equi-join alone, so supporting a larger class of joins would be an interesting future direction for this work.

ACKNOWLEDGMENTS

This work is supported in part by DARPA under Contract No. HR001120C0087, NSF awards #2016240, #1846447, #2236819, and research awards from Meta and Google. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

[1] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, Dallas, TX, USA, 2087–2104.

[2] Sumeet Bajaj and Radu Sion. 2013. CorrectDB: SQL engine with practical query authentication. *Proceedings of the VLDB Endowment* 6, 7 (2013), 529–540.

[3] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment* 12, 3 (2018).

[4] Carsten Baum, Alex J. Malozemoff, Marc Rosen, and Peter Scholl. 2020. Mac'n'Cheese: Zero-Knowledge Proofs for Arithmetic Circuits with Nested Disjunctions. Cryptology ePrint Archive, Report 2020/1410. https://eprint.iacr.org/2020/1410.

[5] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. 2021. Mac'n'Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions. In *CRYPTO 2021, Part IV (LNCS)*, Tal Malkin and Chris Peikert (Eds.), Vol. 12828. Springer, Heidelberg, Germany, Virtual Event, 92–122.

[6] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/3183713.3190662

[7] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *CRYPTO 2013, Part II (LNCS)*, Ran Canetti and Juan A. Garay (Eds.), Vol. 8043. Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 90–108.

[8] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 781–796. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/ben-sasson

[9] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *2018 IEEE Symposium on Security and Privacy (SP)*. 315–334. https://doi.org/10.1109/SP.2018.00020

[10] R. Canetti. 2001. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. 136–145. https://doi.org/10.1109/SFCS.2001.959888

[11] Ronald Cramer and Ivan Damgård. 1997. Linear Zero-Knowledge—a Note on Efficient Zero-Knowledge Proofs and Arguments. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing* (El Paso, Texas, USA) *(STOC '97)*. Association for Computing Machinery, New York, NY, USA, 436–445. https://doi.org/10.1145/258533.258635

[12] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. 2021. Line-Point Zero Knowledge and Its Applications. In *2nd Conference on Information-Theoretic Cryptography (ITC 2021) (Leibniz International Proceedings in Informatics (LIPIcs))*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany.

[13] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. 2021. Constant-Overhead Zero-Knowledge for RAM Programs. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, Virtual Event, Republic of Korea, 178–191.

[14] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic Span Programs and Succinct NIZKs without PCPs. In *EUROCRYPT 2013 (LNCS)*, Thomas Johansson and Phong Q. Nguyen (Eds.), Vol. 7881. Springer, Heidelberg, Germany, Athens, Greece, 626–645.

[15] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. 2016. ZKBoo: Faster Zero-Knowledge for Boolean Circuits. In *USENIX Security 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, Austin, TX, USA, 1069–1083.

[16] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1991. Proofs That Yield Nothing But Their Validity Or All Languages in NP Have Zero-Knowledge Proof Systems. *Journal of the ACM* 38, 3 (1991), 691–729.

[17] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract). In *17th ACM STOC*. ACM Press, Providence, RI, USA, 291–304.

[18] Matthew Green, Mathias Hall-Andersen, Eric Hennenfent, Gabriel Kaptchuk, Benjamin Perez, and Gijs Van Laer. 2022. Efficient Proofs of Software Exploitability for Real-world Processors. Cryptology ePrint Archive, Paper 2022/1223. https://eprint.iacr.org/2022/1223 https://eprint.iacr.org/2022/1223.

[19] Jens Groth. 2010. Short Pairing-Based Non-interactive Zero-Knowledge Arguments. In *ASIACRYPT 2010 (LNCS)*, Masayuki Abe (Ed.), Vol. 6477. Springer, Heidelberg, Germany, Singapore, 321–340.

[20] David Heath and Vladimir Kolesnikov. 2020. Stacked Garbling for Disjunctive Zero-Knowledge Proofs. In *EUROCRYPT 2020, Part III (LNCS)*, Anne Canteaut and Yuval Ishai (Eds.), Vol. 12107. Springer, Heidelberg, Germany, Zagreb, Croatia, 569–598.

[21] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2007. Zero-knowledge from secure multiparty computation. In *39th ACM STOC*, David S. Johnson and Uriel Feige (Eds.). ACM Press, San Diego, CA, USA, 21–30.

[22] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 2013. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *ACM CCS 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, Berlin, Germany, 955–966.

[23] Dakshita Khurana, Rafail Ostrovsky, and Akshayaram Srinivasan. 2018. Round Optimal Black-Box "Commit-and-Prove". Cryptology ePrint Archive, Paper 2018/921. https://eprint.iacr.org/2018/921 https://eprint.iacr.org/2018/921.

[24] United States Department of Education. 2022. *College Scorecard.* https://collegescorecard.ed.gov

[25] Srinath Setty. 2020. Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In *CRYPTO 2020, Part III (LNCS)*, Daniele Micciancio and Thomas Ristenpart (Eds.), Vol. 12172. Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 704–737.

[26] The New York Times. 2022. *U.S. News Dropped Columbia's Ranking, but Its Own Methods Are Now Questioned.* https://www.nytimes.com/2022/09/12/us/columbia-university-us-news-ranking.html

[27] Transaction Processing Council. 2023. *TPC-H Benchmark.* http://www.tpc.org/tpch/

[28] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.

[29] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. 2021. Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1074–1091.

[30] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. 2021. QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, Virtual Event, Republic of Korea, 2986–3001.

[31] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. 2020. Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof. In *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 859–876.

[32] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases. In *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 863–880.

[33] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. A Zero-Knowledge Version of vSQL. Cryptology ePrint Archive, Report 2017/1146. https://eprint.iacr.org/2017/1146.

[34] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2015. IntegriDB: Verifiable SQL for Outsourced Databases. In *ACM CCS 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM Press, Denver, CO, USA, 1480–1491.

[35] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. Veridb: An sgx-based verifiable database. In *Proceedings of the 2021 International Conference on Management of Data*. 2182–2194.