Zero Knowledge on SQL

EE5178 DBMS Final Project Group 10

Meng-Hsueh Lee
Dept. of Economic
National Taiwan University
Taipei, Taiwan
b07303086@ntu.edu.tw

En-Jhih Lo
Dept. of Electrical Engineering
National Taiwan University
Taipei, Taiwan
r11921008@ntu.edu.tw

Chiung-Tao Chen
Dept. of Electrical Engineering
National Taiwan University
Taipei, Taiwan
r11921009@ntu.edu.tw

Hua-Yu Shyu
Dept. of Electrical Engineering
National Taiwan University
Taipei, Taiwan
r11921018@ntu.edu.tw

Sheng-Wei Peng
Dept. of Electrical Engineering
National Taiwan University
Taipei, Taiwan
r11921019@ntu.edu.tw

ABSTRACT

Databases play a crucial role in various applications, handling vast amounts of private data. However, concerns about data privacy and the reliability of query answers have emerged as users seek assurances from data providers without disclosing their own records. Historically, zero-knowledge proofs suffered from significant inefficiencies, limiting their practicality in complex applications. However, recent advancements by the cryptography community, such as Zero-Knowledge Succinct Non-interactive ARguments of Knowledge (ZK-SNARK) protocols, have significantly improved their efficiency. In this project, we employ ZK-SNARK to implement aggregation functions in SQL, specifically focusing on the average, variance, and quantile calculations. Our implementation utilizes the ZK-SNARK library in Python and SQLite, showcasing the feasibility of integrating zero-knowledge proofs into SQL for aggregation functions. By exploring the possibilities offered by ZK-SNARK in SQL, we aim to lay the foundation for secure and privacy-preserving data analysis, enhancing user trust in the reliability and privacy of query results.

KEYWORDS

Zero-knowledge, SQL, Database, Succinct arguments

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 reluctant 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 A lack of strong assurances can result in incidents that undermine user trust in these publications. Therefore, numerous researchers are actively addressing this issue, including Xiling Li et al [1]. They 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 the verifier to get authenticated answers efficiently from the prover while the prover may be untrusted. Its proofs are with respect to a public commitment to 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. They 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' tables. Set-based ones prove properties about a given operator's results. 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. For instance, Ben-Sasson et al [2] introduce ZK-SNARK for program executions, which we will elaborate on in section 2. In this project, we will utilize ZK-SNARK to implement some aggregation functions in SQL, as detailed in sections 3. Although the implementation may seem somewhat removed from the reality of aggregation function in SQL, we believe it provides a foundation for exploring the possibilities.

2 ZK-SNARK

ZK-SNARK are regarded as one of the most popular cryptographic technologies with many potential applications. ZK-SNARK is a concise, non-interactive zero-knowledge proof. It enables the proof of a proposition's correctness without revealing any additional information. Moreover, the resulting proofs possess succinctness, meaning they are small and independent of the computational complexity. In simple terms, ZK-SNARKs allow an individual to prove something to others without disclosing any privacy-related details, while generating compact proofs with low

verification costs that are unrelated to the computational effort required for the proof's content.

The following parts are the explanation of ZK-SNARK principle [3, 4, 5] and its flow chart is shown in **Figure 1**. For example, we try to get the average and need to confirm correctness without access to the raw data. We know a solution to a mean value equation:

$$avg = \frac{\sum_{i=1}^{n} x_i}{n} \quad (1)$$

where we set $n = 5$ as example.

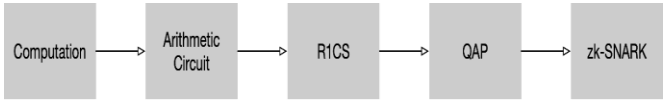

Figure 1: ZK-SNARK flow chart

2.1 Computation to R1CS

Because ZK-SNARK cannot directly be used to solve any computational problem, we need to first convert the problem into the correct "form" to handle it, which is called an R1CS (Rank-1 Constraint System). To be more specific, we use the method called "flattening", which is let the equation only has two term:

1. $x = y$ (where y can be a variable or a number)
2. $x = y\ op\ z$ (where *op* is a binary operator such as +, -, *, or /, and y and z can be variables, numbers, or sub-expressions)

Therefore, the flattening result in our example expressed as:

$$sym_1 = x_1 + x_2$$
$$sym_2 = sym_1 + x_3$$
$$sym_3 = sym_2 + x_4$$
$$sym_4 = sym_3 + x_5$$
$$\sim out = \frac{sym_4}{5}$$

We can see that each of the statements above is a logic gate in a circuit. In comparison to the original code (1), we introduce 4 intermediate variables, sym_1, sym_2, sym_3 and sym_4, and an additional redundant variable $\sim out$ to represent the output. Therefore, this set of statements are R1CS form, which each statement is a constraint and has only one rank.

2.2 R1CS to QAP

"Quadratic arithmetic problem" (QAP) is a form of problem and it is easy to verify but hard to solve, which means it is not NP complete question. We need to use QAP to do further ZK-SNARK method. During the process of QAP conversion, if you have the input, you can create a corresponding solution referred to as the witness of the QAP. Then, following step is required to create an actual "zero-knowledge proof" for the witness.

Next, we need to convert the R1CS to a sequence composed of three vectors (*A, B, C*). There is a solution vector *s*, which must satisfy the inner product operation:

$$s \cdot A \times s \cdot B = s \cdot C \quad (2)$$

In this case, the solution vector *s* corresponds to the witness. Then, we transform each constraint statement into the form as (2). For our average example, apart from the 10 variables after flattening $(sym_1, sym_2, sym_3, sym_4, x_1, x_2, x_3, x_4, x_5, \sim out)$, we also need to introduce the redundant variable ~one in the first component position to represent the number 1. In our average example, one possible arrangement of these 11 components corresponding to a vector is as follows:

$$s = [\sim one, sym_1, sym_2, sym_3, sym_4, x_1, x_2, x_3, x_4, x_5, \sim out] \quad (3)$$

Assume the input x_n are [5,8,1,19,3], and then we can easily get the *s* value:

$$s = [1, 13, 14, 33, 36, 5, 8, 1, 19, 3, 7.2]$$

Take first sequence $sym_1 = x_1 + x_2 \Rightarrow (x_1 + x_2) * 1 = sym_1$ as example, the vectors (A, B, C) like:

$$\begin{cases} A_1 = [0,0,0,0,0,1,1,0,0,0,0] \\ B_1 = [1,0,0,0,0,0,0,0,0,0,0] \\ C_1 = [0,1,0,0,0,0,0,0,0,0,0] \end{cases} \quad (4)$$

As for the second sequence $sym_2 = sym_1 + x_3 \Rightarrow (sym_1 + x_3) * 1 = sym_2$, the vectors (*A, B, C*) become this:

$$\begin{cases} A_2 = [0,1,0,0,0,0,0,1,0,0,0] \\ B_2 = [1,0,0,0,0,0,0,0,0,0,0] \\ C_2 = [0,0,1,0,0,0,0,0,0,0,0] \end{cases} \quad (5)$$

The other sequences do the same things, then we can get 5 sets of QAP because we have 5 R1CSs. After that, we need to convert the QAP form to polynomial instead of the dot product of the matrix. Lagrange interpolation is applied in this process in order to get the coefficient of the constraints which come from the polynomial pass some specific point.

To be more specific, we convert 5 three-vector-sets of length 11, which like (4) and (5) into 11 sets of polynomials. Each set of polynomials consists of three quartic polynomials.

First, we obtain the polynomials for the first value of *A* vector corresponding to the 5 constraints. This means we use the Lagrange interpolation theorem to find the polynomials passing through points $(1, A_1(1))$, $(2, A_2(1))$, $(3, A_3(1))$, $(4, A_4(1))$ and $(5, A_5(1))$. Then, get the coefficients of this polynomial. Because of the polynomials passing through 5 points, it is a quartic polynomial and correspondingly it has 5 coefficients for the new first value of *A* vector corresponding to the 5 constraints.

Similarly, we can obtain the polynomials for each vector corresponding to the remaining constraints. After we get these 11 polynomials, the equation (2) can become:

$$[s \cdot A(n)] \times [s \cdot B(n)] - [s \cdot C(n)] = H(n) \times Z(n) \quad (6)$$

where $Z(n) = (n-1)(n-2)\ldots(n-5)$.

Therefore, the formula (6) is a standard QAP. Then, the following things is that using this QAP do the ZK-SNARK.

2.3 QAP to ZK-SNARK

By converting computational problem into QAP, we can simultaneously check all the constraints using polynomial inner products, rather than individually checking each constraint like in R1CS.

To verify the result polynomial $[s \cdot A(n)] \times [s \cdot B(n)] - [s \cdot C(n)]$ at $n = 1, 2, 3, 4$ and 5, we need to check if it equals 0 at all these points. If the polynomial is non-zero at any of these 5 points,

the verification fails. Otherwise, if it is zero at all 5 points, the verification is successful.

According the QAP is easy to verify but hard to solve, so if prover have the solution, it is easy to verify. To be specific, verifier randomly give the n' to the prover, then the prover can calculate $\bar{A}(n'), \bar{B}(n'), \bar{C}(n')$ and $H(n')$ from (6) to verifier where $\bar{A}(n') = s \cdot A(n')$ and so on. Then, the verifier can calculate the $\bar{A}(n') \times \bar{B}(n') - \bar{C}(n')$ is equal to $H(n') \times Z(n')$ or not to verify that the prover has the knowledge or not. So far, the briefly proof of ZK-SNARK has been explained. The real implementation on ZK-SNZRK is shown in **Figure 2**.

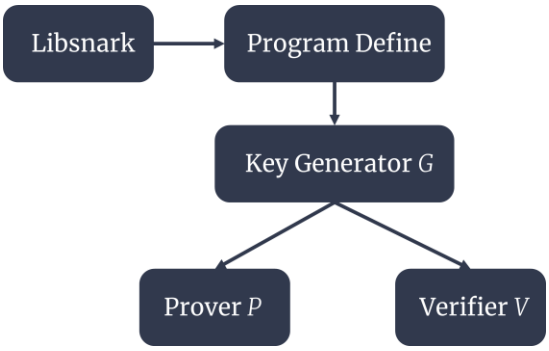

Figure 2: Real implementation on ZK-SNZRK

3 Implementation

In this section, we utilize python for implementing some aggregation functions such as sum, average, standard deviation and quantile calculations, and showcasing the feasibility of integrating ZK-SNARK on these functions. We reference and modify the codes from python-libsnark [6] repository on GitHub to finish our tasks. The implementation details are shown in the following passages.

3.1 Sum

In this section, a protoboard hierarchy is established based on the input, as shown in **Figure 3**. Each addition of pair of data on the same level represents one of the data on the level above. The inputs are accumulated layer by layer, from bottom to top, with the total sum of all inputs at the top. Additionally, appropriate R1CS addition constraints are set between each hierarchy. By doing so, the implementation of *sum_zk* can be achieved.

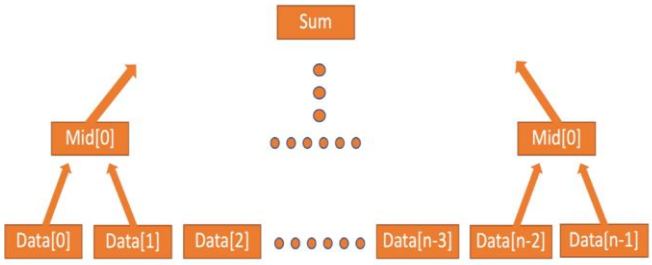

Figure 3: Protoboard hierarchy structure of *sum_zk*

3.2 Average

In this section, we implement average based on previous calculated sum of one input list. Due to the fact that there are 2 main restrictions of R1CS constraint, the implementation of calculating average with ZK-SNARK became more challenging. These two restrictions include, first, R1CS constraint does not support division, and the second is that R1CS only supports integer calculation. And this implies that we can only implement integer addition, subtraction and multiplication with R1CS.

Thus, we design the average function without the usage of division. The algorithm is shown on **Figure 4**. Instead of the directly division, we implement a while loop to repeatedly subtract the sum with the value of length of list until the sum value is no longer larger than 0, and also record the number of subtractions simultaneously. To this end, the final answer would be the number of subtractions minus 1. We construct R1CS linear combination constraint on each intermediate result in the while loop and number of subtractions to achieve ZK-SNARK. Notice that R1CS constraint does not support decimal points, so the final average value of the list would be rounded down. **Figure 5** shows a clearer example of our algorithm. From **Figure 5**, if length of the list is 7 and the sum of the list is 22, then we can get correct average value 3 by our algorithm.

Algorithm 1 Average of a list (Round Down)

```
l ← length of input list        ▷ 'l' denotes the length of the list
total ← sum                     ▷ Initialize 'total' as the sum of the list
ans ← 0
while total ≥ 0 do
    total ← total − l           ▷ Total sum minus the length of list
    ans ← ans + 1               ▷ Increment the counts of subtraction
end while
return ans - 1
```

Figure 4: Algorithm of calculating average of a list

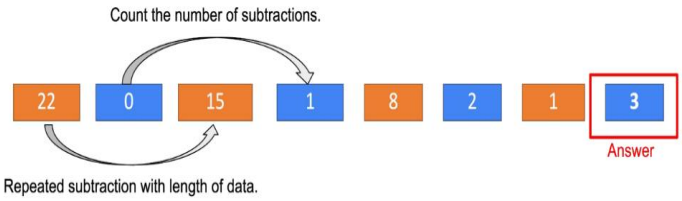

Figure 5: One example of the implementation of calculating average with our methods with input sum = 22

3.3 Variance and Standard Deviation

We also implement calculation of getting variance and standard deviation of the input list. The implementation details are shown in **Figure 6**.

To calculate the variance of the list, we first get the sum and average of the list. After that, calculate the square error of each value via a for loop and use a variable *tempSum* to store the total sum of square error. The sum function is implemented via a for loop. The variance of a list would be the sum of square error divided by

the length of list. We implement the same method as calculating the average of the list as **Section 3.2**.

As of standard deviation, we need to apply square root of variance to get the standard deviation. To this end, we use a while loop to handle this problem. We initialize a variable n as 0 and repeatedly increment n when its square value is lower than variance.

During the calculation, we all construct R1CS linear combination constraint on all intermediate result to achieve ZK-SNARK. However, due to the fact that there exists some for loops and while loops, the efficiency would be quite low when the input list is with large variance. This efficiency problem would be our future work.

Algorithm 2 Variance and Standard deviation of a list (Round Down)

```
l ← length of input list          ▷ 'l' denotes the length of the list
total ← sum                       ▷ Initialize 'total' as the sum of the list
avg ← average                     ▷ Initialize 'avg' as the average of the list
varlist ← []
for each l in list do
    varlist[i] ← (list[i] − avg)^2   ▷ Calculate the square error of each value
end for

tempSum = 0
for each l in list do             ▷ Calculate the sum of square error
    if tempSum = 0 then
        tempSum ← varlist[i − 1] + varlist[i]
    else
        tempSum ← tempSum + varlist[i]
    end if
end for

m = 0                             ▷ Calculate the variance
while tempSum ≥ 0 do
    tempSum ← tempSum − l
    m ← m + 1
end while
var = m - 1                       ▷ Variance of the list

n = 0                             ▷ Calculate the standard deviation
while n^2 ≤ var do                ▷ Implement square root with while loop
    n ← n + 1                     ▷ Increment the counts
end while
std = n - 1                       ▷ Standard deviation of the list
```

Figure 6: Algorithm of calculating variance and standard deviation of a list

3.4 Quantiles

The input of *PR_zk(PR, Data)* consists of the desired PR value and the input data. Assuming the length of the data is L, the output is the data from the lowest to the $\left\lceil \frac{PR}{100} \times L \right\rceil$-th element. Therefore, if the goal is to obtain the median, the input PR would be 50. If the goal is to obtain the first quartile Q1, the input PR would be 25. Any data greater than the returned value would be greater than the first quartile.

In terms of implementation, the input data is first sorted, and then a data structure diagram, as shown in **Figure 7**, is established. The blue blocks start from 1. Next, R1CS addition constraints are set for the data, incrementing by 1. This allows for the numbering of the data using R1CS. Finally, the second-to-last data point (i.e., the last orange data point) is returned.

Figure 7: Protoboard structure of PR_zk

4 Conclusion

In today's modern era, numerous organizations have embraced the utilization of databases as a means to store vast amounts of personal information. However, with the increasing concern for privacy and data security, it has become imperative to safeguard this sensitive data. This is where zero-knowledge (ZK) proof comes into play, as it assumes a pivotal role in providing statistical results derived from data without the need to expose the raw information itself. In our final project, we aim to explore and demonstrate the efficacy of ZK-SNARK in maintaining privacy and security. Our focus lies in implementing various aggregate functions, such as computing averages, using ZK-SNARK on real National Basketball Association (NBA) data. By employing ZK-SNARK, we can perform computations and generate statistical insights on the NBA data while preserving the confidentiality of the underlying information. This means that we can provide meaningful and accurate statistics without compromising the privacy of individual records. Ultimately, our project aims to showcase the potential of ZK-SNARK as a powerful tool for preserving privacy in data-driven organizations like the NBA, ensuring that valuable insights can be derived without infringing upon individuals' personal information.

5 Future Work

The application of ZK-SNARK on database will revolutionized the data privacy and security. ZK-SNARK is provided the ability for one party to prove to another that they know a secret without revealing the secret itself. This technology has been extensively used in blockchain systems to ensure transaction privacy, but its application to databases has opened up a whole new realm of possibilities.

In our project, we use this tool to write an aggregate function to verify some data characteristics like average, variance, and standard deviation. According to the traditional reading method, all data would be transmitted to the user for computation, and the user would also receive this individual data. However, if our aggregate function is used, only the final computed result can be provided, while simultaneously proving to the user that the provided result is correct. By providing accurate data, we also ensure the privacy of the original data.

Nevertheless, because ZK-SNARK is used to apply to blockchain systems, if we want to take advantage of them, there are

several problems that still need to be solved. First, in the world of money transactions, only addition, subtraction, and multiplication are used, but there are many other calculation methods in data processing, such as division, power and square root, etc., which were not provided in the original tools. Need to find another way to calculate. Second, because the money transaction will not appear the decimal, the flexibility and practicality that can be used in the database of real-world is relatively insufficient. At last but not least, the data in database not only include the numbers but also words. How to verify the correctness of words can be the challenging work.

ACKNOWLEDGMENTS

In this project, we are grateful for lectures by Professor Ming-Ling Lo, and his advice and support. We have also got a lot of advice from teaching assistants.

REFERENCES

[1] LI, Xiling, et al. ZKSQL: Verifiable and E icient Query Evaluation with Zero-Knowledge Proofs. Proc. VLDB Endow. 16(8): 1804-1816 (2023)
[2] Ben-Sasson, Eli, et al. "Succinct non-interactive zero knowledge for a von Neumann architecture." 23rd Security Symposium (Security 14). 2014.
[3] Chen, Thomas, et al. "A review of zk-snarks." arXiv preprint arXiv:2202.06877 (2022).
[4] Buterin, V. (2017, February 3). Zk-SNARKs: Under the Hood. https://medium.com/@VitalikButerin/zk-snarks-under-the-hood-b33151a013f6
[5] Shen, G. (2022, July 17). 圖解 Zk-SNARK 思維. https://medium.com/@gregshen0925/%E8%BC%95%E9%AC%86%E4%BA%86%E8%A7%A3-zk-snark-%E7%AC%AC%E4%BA%8C%E7%AF%87-94f6e3483290
[6] Meilof. (2022, May 22). Python-Libsnark. https://github.com/meilof/python-libsnark

VIDEO LINK & WORK DIVISION

1. Video link:
   https://www.youtube.com/watch?v=-QkWIDQqHU4
2. Work division:
   Meng-Hsueh Lee: SQL code, video, report
   En-Jhih Lo: ZK code, video, report
   Chiung-Tao Chen: ZK code, video, report
   Hua-Yu Shyu: ZK proof, report
   Sheng-Wei Peng: ZK proof, video, report