Towards Efficient Verifiable SQL Query for Outsourced Dynamic Databases in Cloud

Jiawei Yuan and Shucheng Yu

University of Arkansas at Little Rock, USA
jxyuan, sxyu1@ualr.edu

Abstract. With the rising trend of outsourcing databases to the cloud, it is important to allow clients to securely verify that their queries on the outsourced databases are correctly executed by the cloud. Existing solutions on this issue either suffer from a high communication cost, or introduce too much computational cost on the client side. Besides, so far only four types of SQL queries (i.e., selection query, projection query, join query and weighted sum query) are supported in existing solutions. It still remains challenging to design a verifiable SQL query scheme that introduces affordable storage overhead, communication and computational cost, and supports more SQL queries used in practice.
This paper investigates this problem and proposes an efficient verifiable SQL query scheme for dynamic databases outsourced to the cloud. Our proposed scheme makes several major progresses: 1) it reduces the communication complexity (excluding the query results) to a logarithmic level (i.e., $O(log\ n)$, where n is the number of tuples in a table), while existing schemes all have a linear or quadratic complexity; 2) it constrains the computational complexity on the client side, in terms of expensive operations such as exponentiation, to a constant level, which is of linear level in existing schemes; 3) in addition to the queries supported by existing schemes, our proposed scheme also supports more practical query types including polynomial queries of any degrees, variance query and many other linear queries. Our design exploits techniques such as Merkle hash tree and constant size polynomial commitment. We shows the efficiency and scalability of our scheme through extensive numerical analysis. Based on the Strong Diffie-Hellman assumption, the Bilinear Strong Diffie-Hellman assumption and the Computational Diffie-Hellman problem, we show that our scheme is provably secure.

Keywords: Integrity Check, Database Outsourcing, SQL Query, Authenticated Data Structure, Cloud Storage

1 Introduction

Database outsourcing is becoming a trend due to the surge of remote storage techniques such as cloud storage. By outsourcing databases to the cloud, data owners can enjoy data sharing across geographical boundaries in addition to other benefits such as cost saving, on-demand self-service, resource elas-

ticity, etc[18]. Commercial cloud infrastructures, including Amazon EC2, Microsoft Azure, Google Cloud, etc., have become prevalent choices as platforms for database outsourcing.

Despite the appealing advantages, outsourcing databases to the cloud also causes security concerns even for non-confidential databases. In particular, clients need to verify the correct execution of database queries to avoid any deliberate or inadvertent misbehavior of the cloud. To specify, clients need to verify the *integrity*, *completeness* and *freshness* of their queries over the outsourced databases [8]: for any query request, we need to assure that the query is executed by the cloud on correct data and the returned results have not been modified (integrity); the results shall include the complete data set (completeness), e.g., if the query is on range $[a, b]$, the results shall not be on other ranges $[a', b']$, where $a' \neq a$ and $b' \neq b$; and the query must be executed on the database of latest version considering frequent updates to the database (freshness).

Related Work. To ensure the users' confidence on the integrity, completeness and freshness of their queries, a series of schemes have been proposed[13, 15, 8, 11, 12, 16, 10, 14, 19] for query verification, which can be roughly categorized into two branches: the tree-based approaches[13, 8, 10, 14] and the signature-based ones[15, 11, 12, 16]. In the tree-based approaches, a Merkle hash tree[9] or its variants[13, 8, 10, 14] are used to assure the integrity of query. In Ref.[8, 10] integrity, completeness and freshness of the query results can be all verified with simple hash operations. However, their communication complexity is proportional to the number of tuples and attributes associated with the query results, which represents a significant overhead in practice, especially for queries on large ranges. In addition, Ref.[8, 10] only support selection query, projection query and joint query with pre-defined keyword attributes. The signature-based approaches[15, 11, 12, 16] utilize the signature aggregation technique[2] and its variants to aggregate the proof information of the query results. Compared with the tree-based approaches, signature-based approaches greatly reduce the communication complexity for query verification at the price of more computational cost, especially for projection queries and joint queries. In the latest signature-based approach[16], the computational complexity on the client side, in terms of the number of expensive operations such as exponentiation, is proportional to the number of tuples and attributes associated with the query results. Moreover, this approach only supports the same queries as Ref.[8, 10]. To improve the previous approaches, Zheng et. al.[19] proposed a SQL query integrity check scheme based on the Merkle hash tree and homomorphic linear tag (HLT). Compared with previous schemes[8, 16], Ref.[19] supports flexible joint query and weighted sum query. However, their scheme requires the transmission of authentication tags, the number of which is proportional to the number of tuples associated with the query results, and such number can be even larger in their weighted sum query. Moreover, in Ref.[19] the client has to perform expensive exponentiation operations, the number of which is also proportional to the number of tuples associated with the result. In practice, such a linear (or quadratic) communication and/or computational complexity can cause a significant communication

and/or computational overhead especially for queries on large ranges. Moreover, to better fulfill the requirements of many practical systems, more SQL queries shall be supported.

Our Scheme. In this paper, we design an efficient verifiable SQL query scheme for outsourced dynamic databases. By uniquely incorporating our proposed polynomial based authentication tag and the Merkle hash tree[9], our scheme can efficiently and simultaneously check the integrity, completeness and freshness of SQL queries in cloud. *The communication cost for query verification is constrained to* $O(logn)$, where n is the number of tuples in a table. On the client side, *expensive operations, i.e., exponentiation and pairing, are limited to a constant number*. Moreover, *our scheme supports more SQL queries including polynomial queries of any degree, variance query and many other linear queries*, in addition to the queries supported by existing work. The main idea of our proposed scheme can be summarized as follows: a database owner first generates a Merkle hash tree for the database and an authentication tag σ_i for each tuple $r_i, 1 \leq i \leq n$, in the database. Then, the database, the tags $\{\sigma_i\}$ and the auxiliary information AU of the Merkle hash tree are outsourced to the cloud server and root state information $State_R$ of the tree is published. When a client queries the databases in cloud, the cloud replies with the query result and the corresponding auxiliary information. The client then checks the completeness and freshness of the result based on AU and $State_R$. After that, the client generates a challenge message and sends it to the cloud for integrity check. Based on the received message, the cloud server produces the proof information to show that it actually executed the query correctly and sends the proof to the client. On receiving the proof information, the client verify the proof with our verification algorithm. In our proposed scheme, we tailor a constant size polynomial commitment technique[7] and allow the cloud server to aggregate all the proof information into three elements to reduce communication cost. In addition, our unique design enables the client to offload most computational tasks to the cloud server to keep the computational cost on the client side minimal. With our proposed scheme any client can perform query verification without the help of the database owner, who is only responsible for database update after outsourcing. Extensive analysis shows that our proposed scheme is efficient and scalable. Our proposed scheme is provably secure based on the Computational Diffie-Hellman (CDH) problem, the Strong Diffie-Hellman (SDH) assumption and the Bilinear Strong DiDiffiee-Hellman (BSDH) assumption.

We summarize the main contributions of this paper as below.

- We proposed an efficient verifiable SQL query scheme for outsourced dynamic databases in cloud. For the first time, we achieve not only a logarithmic communication complexity but also a constant computational cost in terms of the number of expensive operations, e.g., exponentiation.
- Our proposed scheme allows verification of polynomial queries of any degree, variance query and many other linear queries, in additional to all the queries supported by existing schemes.
- Our proposed scheme is provably secure under standard assumptions. Its performance advantages are validated via extensive numerical analysis.

– Our proposed polynomial based authentication tag can be used as an independent solution for other related application, such as database auditing, encrypted key word search, etc.

The rest of this paper is organized as follows: Section 2 describes the models and assumptions of our scheme. In Section 3, we introduce the technique preliminaries of this work, which is followed by our scheme description in Section4. In Section 5, we analyze our proposed scheme in terms of security and performance. We conclude our paper in Section 6.

2 Model and Assumption

2.1 System Model

In this work, we consider a system consists of three major entities: *Database Owner*, *Cloud Server* and *Client*. The database owner has a relational database with multiple tables, each of which consists of multiple tuples and multiples attributes. The owner outsources his databases to the cloud server together with the corresponding authentication tags as well as the auxiliary information of Merkle hash tree, and publishes the root state information of the tree. The client who shares the database with the owner can perform verifiable SQL query on it without help of the owner. To check the integrity, completeness and freshness of the query result, the client requests the proper tags and auxiliary information from the cloud server, and then challenges it with a random message. On receiving the message, the cloud server generates the proof information and returns it to the client. Based on the proof information, the client verifies the query results with the verification algorithm.

2.2 Security Model

We consider the cloud server as untrusted and potentially malicious, which is consistent with previous schemes[10, 16, 19]. In our model, we need to assure that our construction is sound and correct. With regard to the soundness, if any malicious cloud server can generate the proof information and pass the verification of our construction, it must execute the query correctly on the right query range and up-to-date database. For the correctness, we require our construction to accept any valid proof information produced from all key pairs (PK, SK), all up-to-date database, all authentication tag σ, all auxiliary information AU and root state information $State_R$. W.l.o.g, we define the following security game for the soundness of our proposed scheme.

Definition 1. *Let* $\nabla = (KeyGen, Setup, Update, Prove, QueryVerify)$ *be a verifiable SQL query scheme and Adv be a probabilistic polynomial-time adversary. Consider the following security game among an Adv, a trust authority (TA), a challenger (C).*

– TA runs $KeyGen(1^\lambda) \rightarrow (PK, SK)$ and sends the public key PK to Adv.

- Adv chooses a database (DTB) and gives it to TA. TA runs $Setup(DTB, SK, PK) \rightarrow (\sigma, AU, State_R)$ to produce σ, AU and sends them back to Adv. TA also publishes $State_R$.
- Adv chooses some data D in DTB and modifies it to D'. Adv sends D' to TA. TA runs $Update$ algorithm to generate the updated state information $State_R^{up}$ with verification and outputs result as $URst$.
- With regard to DTB, the challenger C sends a query request Qry to Adv. Adv returns the query result Rst together with the corresponding number of AU. C then challenges Adv with a random message $Chall$. Adv responses C with the proof information Prf generated by running an arbitrary algorithm instead of the $Prove$ algorithm.
- C checks Prf by running $QueryVerify(Rst, Prf, PK, State_R, Au) \rightarrow (VRst)$.
- Adv wins the game if and only if it can produce AU, Prf without executing the $Update$ algorithm and query correctly, but make TA and C output both $URst$ and $VRst$ as *accept*.

We can consider ∇ is sound if any probabilistic polynomial-time adversary Adv has at most negligible probability to win the above game.

2.3 Assumption

Definition 2. ***Computational Diffie-Hellman (CDH) Problem****[4]*

Let $a, b \overset{R}{\leftarrow} Z_p^*$. Given input as (g, g^a, g^b), it is computationally hard to calculate the value g^{ab}, where g is a generator of a cyclic group G of order p.

Definition 3. ***t-Strong Diffie-Hellman (t-SDH) Assumption****[1]*

Let $\alpha \overset{R}{\leftarrow} Z_p^*$. For any probabilistic polynomial time adversary(Adv), given input as a $(t+1)-tuple$ $(g, g^{\alpha}, \cdots, g^{\alpha^t}) \in G^{t+1}$, the probability $Prob[Adv(g, g^{\alpha}, \cdots, g^{\alpha^t}) = (c, g^{\frac{1}{\alpha+c}})]$ is negligible for any value of $c \in Z_p^*/-\alpha$, where G is a cyclic group of order p and g is the generator of G.

Definition 4. ***t-Bilinear Strong Diffie-Hellman (t-BSDH) Assumption****[5]*

Let $\alpha \overset{R}{\leftarrow} Z_p^*$. For any probabilistic polynomial time adversary(Adv), given input as a $(t+1) - tuple$ $(g, g^{\alpha}, \cdots, g^{\alpha^t}) \in G^{t+1}$, the probability $Prob[Adv(g, g^{\alpha}, \cdots, g^{\alpha^t}) = (c, e(g,g)^{\frac{1}{\alpha+c}})]$ is negligible for any value of $c \in Z_p^*/-\alpha$,where G is a multiplicative cyclic group of order p and g is the generator of G.

3 Technique Preliminaries

3.1 Bilinear Map

For a Bilinear Map[3]: $e : G \times G \rightarrow G_T$, where G and G_T are multiplicative cyclic groups of the same prime order p, the following properties hold:

- Bilinear: for any $a, b \overset{R}{\leftarrow} Z_p^*$ and $g_1, g_2 \in G$, $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$.
- Non-Degenerate: for any $g \in G$, $e(g, g) \neq 1$.
- Computable: a Bilinear Map e can always be computed efficiently with a computable algorithm.

3.2 Merkle Hash Tree

Merkle hash tree was first proposed in Ref.[9] to prove that a set of elements has not been modified. In a Merkle hash tree, each leaf node contains the hash value of the corresponding data and each non-leaf node contains the hash value of the concatenation of its children's values. Specially, for two leaf nodes $leaf_1$ and $leaf_2$, whose values are $hash(data_1)$ and $hash(data_2)$, the value of their father node is $hash(hash(data_1)||hash(data_2))$. For verification purpose, the hash value of the root of a Merkle hash tree is published. To check the integrity of data associated with a leaf node, a verifier first generates the hash value of the data. Then, by combining the generated hash value and the siblings of nodes on the path to the root, the verifier can calculate the value of the root. If the calculated root hash value is equal to the published value, the checking data are valid; otherwise, the data has been modified. For more details, please refer to Ref.[9].

3.3 Constant Size Polynomial Commitment

A secure polynomial commitment scheme allows a committer to commit a polynomial with a short string. Based on the algebraic property that polynomials $f(x) \in Z[x]$: $f(x) - f(r)$ can be perfectly divided by $(x - r)$, where $r \overset{R}{\leftarrow} Z_p^*$, Kate et.al.[7] proposed a polynomial commitment scheme with a constant size communication cost. In their construction, to verify the correctness of a polynomial evaluation $f(r)$, the committer can aggregate all the proof information into a single element. Specifically, the construction of the constant size polynomial commitment scheme [7] can be summarized as follows.

- **Setup:** Given a security parameter λ and a fixed number s, a trust authority outputs the public key and private key as:

  $$PK = (G, G_T, g, g^{\alpha}, \cdots, g^{\alpha^{s-1}}),\ SK = \alpha \overset{R}{\leftarrow} Z_p^*$$

  where G and G_T are two multiplicative cyclic groups with the same prime order p, g is the generator of G and $e : G \times G \rightarrow G_T$.
- **Commit:** Given a polynomial $f_{\boldsymbol{m}}(x) \in Z_p[x]$, where $\boldsymbol{m} = (m_0, m_1, \cdots, m_{s-1}) \overset{R}{\leftarrow} Z_p^*$ is the coefficient vector, a committer generates the commitment $C = g^{f_{\boldsymbol{m}}(\alpha)} \in G$ and publishes C.
- **CreateWitness:** Given a random index $r \overset{R}{\leftarrow} Z_p^*$, the committer computes $f_{\boldsymbol{w}}(x) \equiv \frac{f_{\boldsymbol{m}}(x) - f_{\boldsymbol{m}}(r)}{(x-r)}$ using polynomial long division, and denote the coefficients vector of the resulting quotient polynomial as $\boldsymbol{w} = (w_0, w_1, \cdots, w_{s-1})$. Based on the public key PK, the witness ψ can be computed as $\psi = g^{f_{\boldsymbol{w}}(\alpha)}$.
- **VerifyEval:** Given the witness ψ, a verifier checks whether or not $f_{\boldsymbol{m}}(r)$ is the evaluation at index r of the polynomial committed by C as:

  $$e(C, g) \overset{?}{=} e(\psi, g^{\alpha}/g^{r}) \cdot e(g, g)^{f_{\boldsymbol{m}}(r)}$$

For the detailed analysis on security and correctness of this polynomial commitment scheme, please refer to Ref.[7].

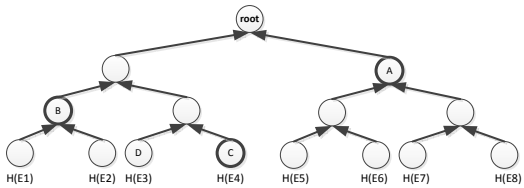

Fig. 1. Merkle Hash Tree: the auxiliary information AU for element E_3 is: the node value of Nodes A,B and C.

4 Our Construction

In this section, we first design two building blocks for our efficient verifiable SQL query scheme on outsourced database: Authenticated outsourced ordered data set (AORDS) and Polynomial based authentication tag (PAT). Then, we describe how to construct different type of verifiable SQL query based on these two blocks.

4.1 Authenticated Outsourced Ordered Data Set

Authenticated Outsourced ordered data set (AORDS) is constructed based on the Merkle hash tree[9]. Let E be an ordered set of elements and $Sign()$ be a signature scheme[1], we describe our construction of AORDS as below.

- **KeyGen**$(1^\lambda) \rightarrow (PK, SK)$: Given a selected security parameter λ, the randomized $KeyGen$ algorithm produces the public-private key pairs (PK, SK) of $Sign()$.
- **SetUp**$(E, SK) \rightarrow (AU, State_R)$: Given SK and an ordered data set $E = \{E_1, E_2, \cdots, E_n\}$, the $SetUp$ algorithm generates a Merkle hash tree. In the generated tree, each node stores the hash value of one element in E and the value of each internal node is the hash value of the concatenation of its children's values. The state information of the root is $State_R = Sign(Root)$ and the auxiliary information AU for each leaf node $leaf_i$ is node values on paths from $leaf_i$ to the root and the values of these nodes' siblings' as shown in Fig.1.
- **Update**$(SK, E) \rightarrow (E', AU', State'_R)$: $Modification$: To modify an element E_i to E'_i, the owner sends E'_i to the server. The server replaces E_i with E'_i and recomputes the root value and auxiliary information as $Root', AU'$ based on E'_i. The owner computes the new root value $Root''$ and compares it with $Root'$. The owner accepts the update on the server and publishes new root state information $Sign(Root')$ if $Root' = Root''$. $Insertion$: Given an element E_i, which need to be inserted between E_k and E_{k+1}, the cloud sets node for E_k as the farther node of two new leaf nodes $leaf', leaf''$, where $leaf' = H(E_k)$, $leaf'' = H(E_i)$ and $father = H(H(E_k)||H(E_i))$. Then the server recomputes $Root', AU'$ and the owner verifies them same as the step $Modification$. $Deletion$: To delete an element E_i, the cloud replaces the father node of $leaf_i$ with its sibling node. Then the cloud and owner check the update with $Root', AU', Root''$ same as $Modification$.

– **QueryVerify**$(PK, State_R) \rightarrow (VRst)$: Given a range query $Qry(a, b)$ request, the cloud server gives the query result Rst and corresponding AU to the client.
Case 1: $Rst = \{E_c, E_{c+1}, \cdots, E_{c+k-1}\}$ is not empty, the cloud server sends AU of nodes E_{c-1} and E_{c+k} to the client. The client recomputes the root value $Root'$ based on the AU and Rst. After running signature verification with PK, the client accepts Rst if $Root = Root'$.
Case 2: Rst is empty. In this case, there must have a E_c that $E_c < a, b < E_{c+1}$. The cloud server sends AU of E_c and E_{c+1} to the client. Same to Case 1, the client produces $Root'$. $VRst = accpet$ if $Root = Root'$; otherwise $VRst = reject$.

4.2 Polynomial Based Authentication Tag

Construction Description In this section, we propose a polynomial based authentication tag (PAT), which can be used to verify the integrity of the query result. We consider a database DTB consisting of n tuples $\{r_1, r_2, \cdots, r_n\}$, each of which has s attributes $\{a_0, a_1, \cdots, a_{s-1}\}$. For an attribute in a tuple, we denote it as $r_i.a_j$. Let $e : G \times G \rightarrow G_T$ and H be the one-way hash function, where G is a multiplicative cyclic group of prime order p and u, g be two random generators of G. We define $f_{\boldsymbol{c}}(x)$ as a polynomial with coefficient vector $\boldsymbol{c} = (c_0, c_1, \cdots, c_{s-1})$ and describe our PAT construction as follows.

– **KeyGen**$(1^\lambda) \rightarrow (PK, SK)$: The database owner chooses a random prime p (λ bits security) and generates a signing keypair (spk, ssk) using BLS signature[1]. The owner then chooses two random numbers $\alpha, \epsilon \xleftarrow{R} Z_p^*$ and computes $v \leftarrow g^\epsilon$, $\kappa \leftarrow g^{\alpha\epsilon}$ as well as $\{g^{\alpha^j}\}_{j=0}^{s-1}$. The public and private keys are

$$PK = \{p, v, \kappa, spk, u, \{g^{\alpha^j}\}_{j=0}^{s-1}\}, SK = \{\epsilon, ssk, \alpha\}$$

– **Setup**$(PK, SK) \rightarrow (\sigma, \tau)$: The database owner randomly choose a random table name (index) $name \in Z_p^*$. Let τ_0 be "$name||n$"; the table tag τ is τ_0 together with a signature on τ_0 signed by ssk: $\tau \leftarrow \tau_0||Sign(\tau_0)$. For each tuple $r_i, 1 \leq i \leq n$, its authentication tag is computed as:

$$\sigma_i = (u^{H(name||i)} \cdot \prod_{j=0}^{s-1} g^{(r_i.a_j)\alpha^j})^\epsilon = (u^{H(name||i)} \cdot g^{f_{\boldsymbol{\beta_i}}(\alpha)})^\epsilon \qquad (1)$$

where $\boldsymbol{\beta_i} = \{\beta_{i,0}, \beta_{i,1}, \cdots, \beta_{i,s-1}\}$ and $\beta_{i,j} = r_i.a_j$.

– **QueryVerify Phase1**$(PK, \tau) \rightarrow Chall$:
The client verifies the signature on τ: if the signature is not valid, it rejects and halts; otherwise, the client parses τ to recover $name, n$. W.o.l.g, to check the integrity of any k query results form $\{r_i.a_j, 1 \leq i \leq n, 0 \leq j \leq s-1\}$, the client randomly chooses k numbers $v_i \xleftarrow{R} Z_p^*$ for these tuples and gets a $k-$elements set $K = \{(i, v_i)\}$. The client chooses another random number $q \xleftarrow{R} Z_p^*$ and challenges the server with the challenge message $Chall = \{q, K\}$.

– **Prove**$(PK, Chall, \tau) \rightarrow (\psi, y, \sigma)$: On receiving $Chall$, the server first computes $\sigma = \prod_{(i,v_i)\in K} \sigma_i^{v_i}$ and $y = f_{\boldsymbol{A}}(q)$, where $\boldsymbol{A} = (\sum_{(i,v_i)\in K} v_i * (r_i.a_0), \cdots, \sum_{(i,v_i)\in K} v_i * (r_i.a_{s-1}))$. Since polynomials $f(x) \in Z[x]$ have the algebraic property that $(x-q)$ perfectly divides the polynomial $f(x) - f(q), q \xleftarrow{R} Z_p^*$. The server divides the polynomial $f_{\boldsymbol{A}}(x) - f_{\boldsymbol{A}}(q)$ with $(x-q)$ and denotes the coefficients vector of the resulting quotient polynomial as $\boldsymbol{w} = (w_0, w_1, \cdots, w_{s-1})$, i.e., $f_{\boldsymbol{w}}(x) \equiv \frac{f_{\boldsymbol{A}}(x) - f_{\boldsymbol{A}}(q)}{x-q}$. Then, the server produces $\psi = \prod_{j=0}^{s-1} (g^{\alpha^j})^{w_j} = g^{f_{\boldsymbol{w}}(\alpha)}$ and responds to the client with $Prf = \{\psi, y, \sigma\}$.
– **QueryVerify Phase2**$(PK, Prf) \rightarrow VRst$:
On receiving the proof response Prf, the client first computes $\eta_i = u^{H(name||i)v_i}$, $(i, v_i) \in K$ and $\eta = \prod_{(i,v_i)\in K} \eta_i$. Then, the client parses Prf as $\{\psi, y, \sigma\}$ and checks

$$e(\eta, v) \cdot e(\psi, \kappa \cdot v^{-q}) \stackrel{?}{=} e(\sigma, g) \cdot e(g^{-y}, v) \tag{2}$$

The $QueryVerify$ algorithm outputs $VRst = accept$ if Eq.2 holds; otherwise, $VRst = reject$.

Correctness of PAT For the cloud server that correctly executes the query request on the right data and generates proof information $Prf = \{\psi, y, \sigma\}$, we analyze the correctness of our proposed PAT as follows.

$$\begin{aligned}
& e(\eta, v) \cdot e(\psi, \kappa \cdot v^{-q}) \\
&= e(u, g)^{\epsilon(\sum_{(i,v_i)\in K} H(name||i)v_i)} \cdot e(g^{f_{\boldsymbol{w}}(\alpha)}, g^{\epsilon(\alpha - q)}) \\
&= e(u, g)^{\epsilon(\sum_{(i,v_i)\in K} H(name||i)v_i)} \cdot e(g, g)^{\frac{f_{\boldsymbol{A}}(\alpha) - f_{\boldsymbol{A}}(q)}{\alpha - q} \cdot \epsilon(\alpha - q)} \\
&= e(u, g)^{\epsilon(\sum_{(i,v_i)\in K} H(name||i)v_i)} \cdot e(g, g)^{\epsilon(f_{\boldsymbol{A}}(\alpha) - f_{\boldsymbol{A}}(q))} \\
&= e(u^{\epsilon(\sum_{(i,v_i)\in K} H(name||i)v_i)}, g) \cdot e(g^{\epsilon f_{\boldsymbol{A}}(\alpha)}, g) \cdot e(g, g)^{-\epsilon f_{\boldsymbol{A}}(q)} \\
&= e(u^{\epsilon(\sum_{(i,v_i)\in K} H(name||i)v_i)} \cdot g^{\epsilon f_{\boldsymbol{A}}(\alpha)}, g) \cdot e(g^{-y}, v) \\
&= e(\sigma, g) \cdot e(g^{-y}, v)
\end{aligned} \tag{3}$$

Based on Eq.3, it is easy to verify that our construction of PAT is correct if the cloud server honestly produces the Prf.

Properties of PAT We first show that our PAT supports homomorphic addition. Specifically, considering any $kattributes, 1 \leq k \leq n$, in the same position of different tuples (e.g., $\{r_c.a_j, r_{c+1}.a_j, \cdots, r_{c+k-1}.a_j\}$) and their corresponding tags, we can calculate the tag for the sum of these k attributes as:

$$\sigma = \prod_{i=c}^{c+k-1} \sigma_i = (u^{\sum_{i=c}^{c+k-1} H(name||i)} \cdot \prod_{i=c}^{c+k-1} \prod_{j=0}^{s-1} g^{(r_i.a_j)\alpha^j})^{\epsilon} \tag{4}$$

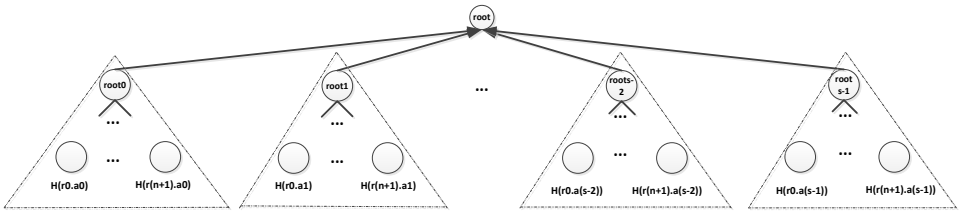

Fig. 2. Hash Tree for TB.

$$= (u^{\sum_{i=c}^{c+k-1} H(name||i)} \cdot \prod_{j=0}^{s-1} g^{\sum_{i=c}^{c+k-1}(r_i.a_j)\alpha^j})^{\epsilon}$$

$$= (u^{\sum_{i=c}^{c+k-1} H(name||i)} \cdot g^{f_{\mathfrak{S}}(\alpha)})^{\epsilon}$$

where $\mathfrak{S} = \{\sum_{i=c}^{c+k-1} \beta_{i,0}, \sum_{i=c}^{c+k-1} \beta_{i,1}, \cdots, \sum_{i=c}^{c+k-1} \beta_{i,s-1}\}$.

4.3 Construction of Efficient Verifiable SQL Query Scheme for Outsourced Dynamic Database

Considering a table TB consists of n tuples $\{r_1, r_2, \cdots, r_n\}$, each of which has s attributes $\{a_0, a_1, \cdots, a_{s-1}\}$. $r_i.a_j$ denotes attribute j in tuple i. For simplicity, TB is ordered by attribute a_0 (it can also be ordered by any other attributes). We set L and U as the lower and upper bounds of the search key attribute a_0. Let $e : G \times G \to G_T$ and H be the one-way hash function, where G is a multiplicative cyclic group of prime order p and g, u be two random generators of G. Based on our two building blocks $AORDS$ and PAT, we describe our efficient verifiable SQL query scheme as below.

KeyGen$(1^\lambda) \to (PK, SK)$: Given a security parameter λ, the database owner runs $AORDS.KeyGen \to (AORDS.PK, AORDS.SK)$ and $PAT.KeyGen \to (PAT.PK, PAT.SK)$. Get the public key and private key as:

$$PK = \{p, v, \kappa, spk, u, \{g^{\alpha^j}\}_{j=0}^{s-1}\}, SK = \{\epsilon, ssk, \alpha\}$$

where $v \leftarrow g^\epsilon$, $\kappa \leftarrow g^{\alpha\epsilon}$ and $\alpha, \epsilon \xleftarrow{R} Z_p^*$.

SetUp$(PK, SK, TB) \to (\sigma, \tau, State_R, AU)$: Generate two additional tuples r_0 and r_{n+1} for the table, where $r_0.a_0 = L$ and $r_{n+1}.a_0 = U$. For each attribute $a_j, 0 \le j \le s-1$, we build a Merkle hash tree for it with $root_j$, and these Merkle hash trees will be combined as a hash tree $Tree_T$ (with $root$) for TB as shown in Fig.2. The state information and auxiliary information of $Tree_T$ are denoted as $State_R$ and AU. Run $PAT.SetUp$ to generate authentication tags σ_i for each tuple $r_i, 0 \le i \le n+1$. Outsource TB, AU and σ_i to the cloud server. Make $State_R$ as public information.

Update$(PK, SK, TB) \to (TB', \sigma', State'_R, AU')$: *Modification:* Suppose the database owner modifies the tuple r_i to r'_i. The owner first generates the authentication tag σ'_i for r'_i and sends it to the cloud server. Then, the owner runs $AORDS.Update$ and updates the r_i, AU and $State_R$. *Insertion:* Suppose

the database owner inserts the tuple r_i between r_c and r_{c+1}. The owner first generates the authentication tag σ_i for r_i and outsources it to the cloud server together with r_i. Then, the owner runs $AORDS.Update$ to adds r_i and updates the AU and $State_R$. *Deletion:* Suppose the database owner deletes the tuple r_i. The owner runs $AORDS.Update$ and updates AU and $State_R$.

Since the $Prove$ and $QueryVerify$ algorithms for different types of SQL queries have some difference, we describe them according to query types.

Selection Query: Suppose a selection query Qry = "select * from TB where $b \leq a_0 \leq d$". The client first runs $AORDS.QueryVerify$ with the range query $Qry(b, d)$ to check the freshness and completeness. If the output is $reject$, the client aborts. Otherwise, if the query result Rst is empty, the client accepts the result as $null$. If Rst consists of k tuples $\{r_t, r_{t+1}, \cdots, r_{t+k-1}\}$, the client runs $PAT.QueryVerify.Phase1$ to generate the challenge message $Chall = \{q, K\}$ and sends it the cloud server. The cloud server then produces the proof information $Prf = \{\psi, y, \sigma\}$ by running $PAT.Prove$. On receiving Prf, the client runs $PAT.QueryVerify.Phase2$ to verify the integrity of these k tuples. If the output $VRst$ is *accept*, accept Rst as the query result; otherwise, reject.

Projection Query: Suppose a projection query Qry ="select $a_0, \cdots, a_k$ from TB", where $1 \leq k \leq s-1$. The cloud sends $Rst = \{r_i.a_0, \cdots, r_i.a_k\}, 1 \leq i \leq n$ to the client. The client first runs $AORDS.QueryVerify$ with the range query $Qry(L, U)$ to check the freshness and completeness. If the output is $reject$, the client aborts. Otherwise the client runs $PAT.QueryVerify.Phase1$ to generate the challenge message $Chall = \{q, K\}$ and sends it the cloud server. The cloud server then produces the proof information $Prf = \{\psi, y, \sigma\}$ by running $PAT.Prove$. On receiving Prf, the client runs $PAT.QueryVerify.\ Phase2$ to verify the integrity of Rst. If the output $VRst = accept$, accept Rst as the query result; otherwise, reject.

Join Query: Suppose there are two tables $\{TB_1, TB_2\}$ processed same as TB and a projection query Qry ="select R_1^*, R_2^* from TB_1, TB_2, where $R_1.a_d = R_2.a_t$". The cloud sends $Rst = \{R_1^*, R_2^*\}$ to the client. The client first runs $Projection\ Query$ algorithm for Qry ="select a_d from TB_1" and Qry ="select a_t from TB_2". If either query outputs $reject$, the client aborts, otherwise, the client gets $r1_i.a_d, r2_i.a_t, 1 \leq i \leq n$. The client then identifies the tuples that fulfills $r1_i.a_d = r2_j.a_t$ and gets two sets of index I_1, I_2, where $i \in I_1, j \in I_2$. Then client checks whether or not the number of elements in I_1 and I_2 are equal to the number of tuples in R_1^* and R_2^* respectively. If not, the client aborts; otherwise, the client runs $PAT.QueryVerify.Phase1$ to generate the challenge messages with $Chall_1 = \{q_1, K_1\}, Chall_2 = \{q_2, K_2\}$ and sends them the cloud server for TB_1 and TB_2 respectively. The cloud server then produces the proof information $Prf_1 = \{\psi_1, y_1, \sigma_1\}, Prf_2 = \{\psi_2, y_2, \sigma_2\}$ by running $PAT.Prove$. On receiving Prf_1 and Prf_2, the client runs $PAT.QueryVerify.Phase2$ to verify the integrity of these tuples. If the output $VRst$ is *accept*, accept Rst as the query result; otherwise, reject.

Weighted SUM Query: Suppose a weighted SUM query Qry ="select SUM($c_i * a_t$) from TB where $b \leq a_0 \leq d$". The cloud sends $Rst = \sum_{i=1}^{k} c_i * (r_i.a_t)$

to the client, where k is the number of tuples satisfying the query condition and c_i is the weight values. The client runs $AORDS.QueryVerify$ with range query $Qry(b,d)$ to check the freshness and completeness. If the output is reject, the client aborts. Otherwise, if the output is empty, the client accepts the result as null. If the output has k elements, the client runs $PAT.QueryVerify.Phase1$ to generate the challenge message $Chall = \{q, K\}$ and sends it the cloud server, in which the k random values v_i in set K is replaced with the k weight values c_i for sum computation. The cloud server then produces the proof information $Prf = \{\psi, y, \sigma\}$ by running $PAT.Prove$. Note that in both proof information ψ and the aggregated tag σ, the sum value $\sum_{i=1}^{k} c_i * (r_i.a_t)$ is embedded (i.e.,in ψ, it has term $g^{\frac{A_t\alpha^t - A_t q^t}{\alpha - q}}$, where $A_t = \sum_{i=1}^{k} c_i * (r_i.a_t)$; in σ, it has term $(g^{A_t\alpha^t})^{\epsilon}$. On receiving Prf, the client runs $PAT.QueryVerify.Phase2$ to verify the integrity of the sum value. If the output $VRst$ is *accept*, accept Rst as the query result; otherwise, reject.

Polynomial Query: Suppose a polynomial query Qry = "select $\sum_{i=1}^{k} c_i * (r_i.a_t)^x$ from TB where $b \leq a_0 \leq d$". The cloud sends $Rst = \sum_{i=1}^{k} c_i * (r_i.a_t)^x$ to the client, where k is the number of tuples satisfying the query condition. The client performs same as $Weighted\ SUM\ Query$ algorithm to generate $Chall = \{q, K\}$ and sends it the cloud server. On receiving the challenge message, the cloud first produces the tags for $(r_i.a_t)^x$ as $\sigma_{exp.(x,it)} = \sigma_{it}{}^{(r_i.a_t)^x} = (u^{H(name||i)\cdot(r_i.a_j)^x} \cdot g^{f_{\boldsymbol{B_i}}(\alpha)})^{\epsilon}$, where $\boldsymbol{B_i} = \{B_{i,0}, B_{i,1}, \cdots, B_{i,s-1}\}$ and $B_{i,j} = (r_i.a_t)^x * \beta_{i,j}$. Note that if all the tags of exponentiation values $exp.(x, it)$ are generated by the cloud, it can avoid computing the right polynomial to save computation cost. Specifically, instead of using the values $(r_i.a_t)^x$ and tags $exp.(x, it)$, the server can use the values $r_i.a.j$ and the corresponding tags σ_i to compute $\sum_{i=1}^{k} c_i * (r_i.a_t)$ and pass the verification. To avoid such dishonest behavior, we split proof generation on cloud into two parts as follows. The cloud server runs $PAT.Prove$ to generate the first part of proof information and sends it to the client as $Prf_1 = \{\psi, y\}$. The client chooses u random elements in K as set U and sends it to the cloud (we discuss the selection of U and the detection probability of cloud's cheating in Section4.4). The cloud returns the attributes $r_i.a_t$ as well as their tags σ_i to the client, where $i \in U$. The client runs $PAT.QueryVerify.Phase2$ to verify the integrity of tuples $r_i, i \in U$. If the output is $reject$, aborts; otherwise, the client generates $\sigma_{exp.(x,it)}, i \in U$ and aggregates them as $\sigma' = \prod_{i \in U} \sigma_{exp.(x,it)}{}^{c_i}, \eta' = \prod_{i \in U} u^{H(name||i)c_i(r_i.a_t)^x}$. The cloud generates the second part of proof information Prf_2 as $\{\sigma'' = \prod \sigma_{exp.(x,it)}{}^{c_i}, \eta'' = \prod u^{H(name||i)c_i(r_i.a_t)^x},\ i \in K, i \notin U\}$ and sends it to the client. The client then computes $\sigma = \sigma' * \sigma'', \eta = \eta' * \eta''$ and runs $PAT.QueryVerify.Phase2$ with $\{\psi, y, \sigma\}$ to verify the integrity of the Rst. If the output $VRst = accept$, accept Rst as the query result; otherwise, reject.

Variance Query: For any k numbers $c_i, 1 \leq i \leq k$, their variance is calculated as $Vari = \frac{\sum_{i=1}^{k} (c_i - c_m)^2}{k}$ and c_m is the mean value of the c_i. Suppose

a variance query Qry ="select Vari(a_t) from TB where $b \leq a_0 \leq d$. The cloud sends $Rst = Vari(a_t)$ to the client. Assume there are k tuples satisfying the query condition, the client first runs $Weighted\ SUM\ Query$ algorithm to get the verified mean value of k tuples, denoted as a_m. Since the tag for $-a_m$ can be generated as $\sigma_{-a_m} = \sigma_{a_m}^{-1}$, the clients can run $Polynomial\ Query$ algorithm to verify Rst. If the output $VRst$ is *accept*, accept Rst; otherwise, reject.

Other Linear Queries: Like variance query, our proposed scheme can also flexibly supports other linear queries based on weighted sum query and polynomial query (e.g., $\sum_{i=0}^{k} c_i * ((r_i.a_t) - (r_{(i+1)}.a_t))^x$).

4.4 Discussion

In this section, we discuss about how to choose the set U in polynomial query to faithfully detect the cloud's cheating and how to move computation tasks to the cloud side. Suppose there are k tuples that satisfying the query condition, when generating $Prf_1 = \{\psi, y\}$, the cloud server can guess the u tuples will be selected by the client with probability $1/\binom{k}{u}$ (e.g., $k = 100$, the client can set $u = 2$ to get 99.9899% confidence that the cloud server cannot guess the set U). In this scenario, if the cloud does not compute ψ rightly according to the correct exponentiation values and tags, it has only $1/\binom{k}{u}$ probability to pass the verification algorithm. Therefore, the client can choose the set U based on the size of set K. When K's size is really small (e.g.,$k = 5$), the client can locally generate the exponentiation tags and aggregate them with few computational cost. Similarly, most calculation tasks of η in the $PAT.QueryVerify.Phase2$ of other queries can also be outsourced to the cloud server. On receiving the proof information Prf, the client can randomly compute m η_i locally and aggregate them as η'. Then, the client lets the cloud calculate the rest η_i and aggregate them η''. The client finally gets $\eta = \eta' \cdot \eta''$.

5 Analysis Of Our Proposed Scheme

5.1 Security Analysis

This section sketches the security of our proposed scheme.

Theorem 1. *The design our AORDS is secure assuming the hash function is collision-resistance and the signature is secure.*

Proof. The construction of AORDS is purely based on the Merkle hash tree, which have been proved to be secure if the collision-resistance hash function and the signature scheme are secure[9]. Therefore, if the our AORDS can be broken by an existed probabilistic polynomial-time adversary, we can construct algorithm B that breaks the either collision-resistance hash function or signature scheme.

Theorem 2. *If a probabilistic polynomial time adversary A can forge $g^{f_c(x)}$, we can construct a polynomial time algorithm B that outputs the solution to the $t - SDH$ problem using A.*

Proof. Suppose there exists a probabilistic polynomial time adversity A that can forge $f_{\boldsymbol{c_1}}(\alpha)$ such that $g^{f_{\boldsymbol{c_1}}(\alpha)} = g^{f_{\boldsymbol{c}}(\alpha)}$, where $\boldsymbol{c}$ and $\boldsymbol{c_1}$ is the coefficient vector, he/she obtains $g^{f_{\boldsymbol{c_2}}(\alpha)} = g^{f_{\boldsymbol{c}}(\alpha)}/g^{f_{\boldsymbol{c_1}}(\alpha)} = g^{f_{\boldsymbol{c}}(\alpha)-f_{\boldsymbol{c_1}}(\alpha)} \in Z_p[x]$. Since $f_{\boldsymbol{c_1}}(\alpha) = f_{\boldsymbol{c}}(\alpha)$ and $f_{\boldsymbol{c_2}}(\alpha) = 0$, i.e., α is a root of polynomial $f_{\boldsymbol{c_2}}(x)$, where $f_{\boldsymbol{c_2}}(x) = f_{\boldsymbol{c}}(x) - f_{\boldsymbol{c_1}}(x)$. By factoring $f_{\boldsymbol{c_2}}(x)$[17], B can easily find $SK = \alpha$ and solve the instance of the t-SDH problem given by the system parameters.

Theorem 3. *If the CDH problem is hard, the BLS signature scheme is existentially unforgeable, the t-SDH assumption and the t-BSDH assumption hold. The proof information $Prf = (y, \psi, \sigma)$ in PAT is unforgeable.*

Proof. Suppose a probabilistic polynomial time adversity A can generate $Prf' = (y', \psi', \sigma')$ to forge Prf after receiving a challenge message from the client, $(y', \psi', \sigma') \neq (y, \psi, \sigma)$. As both Prf' and Prf can be accepted by the $QueryVerify$ algorithm, we can get the following two equations:

$$e(\eta, g) \cdot e(\psi, \kappa \cdot v^{-r}) = e(\sigma, g) \cdot e(g^{-y}, v) \tag{5}$$

$$e(\eta, g) \cdot e(\psi', \kappa \cdot v^{-r}) = e(\sigma', g) \cdot e(g^{-y'}, v) \tag{6}$$

Dividing Eq.5 with Eq.6, we obtain:

$$\left(\frac{e(\psi, g)}{e(\psi', g)}\right)^{\epsilon(\alpha - r)} = \frac{e(\sigma, g)}{e(\sigma', g)} \cdot e(g, g)^{\epsilon(y' - y)} \tag{7}$$

Now we do a case analysis on whether $\sigma = \sigma'$.

Case 1: $\sigma \neq \sigma'$. As $\left(\frac{e(\psi,g)}{e(\psi',g)}\right)^{\epsilon(\alpha-r)}$, $e(g,g)^{\epsilon(y'-y)}$ and $e(\sigma, g)$ are known to the adversary, we rewrite Eq.7 as

$$\begin{aligned} e(\sigma', g) &= e(\sigma, g) \cdot \Upsilon \\ e(\sigma', g) &= e(u^{\epsilon(\sum_{(i,v_i)\in Q} t_i)}, g) \cdot e(g^{\epsilon f_{\boldsymbol{A}}(\alpha)}, g) \cdot \Upsilon \end{aligned} \tag{8}$$

where we denote $\Upsilon = e(g, g)^{\epsilon(y'-y)} / \left(\frac{e(\psi,g)}{e(\psi',g)}\right)$ as a known value to the adversary and $t_i = H(name||i)v_i$.

Recall that in this proof, the CDH Problem is hard. If the any probabilistic polynomial time adversity A can find σ' with non-negligible probability and make the Eq.8 hold, we can construct an algorithm B that uses A to solve the instance of CDH Problem. Specifically, given $\sigma' \neq \sigma$ found by A, which makes Eq.8 hold, B can easily extract $g^{\epsilon f_A(\alpha)}$. With the given information, B can get $g^{\epsilon}, g^{f_A(\alpha)}$, and thus solve CDH problem for them with solution $g^{\epsilon f_A(\alpha)}$. Therefore, no probabilistic polynomial time adversity can find a valid forged response $(y, \psi, \sigma) \neq (y', \psi', \sigma')$ and $\sigma \neq \sigma'$ with non-negligible probability.

Case 2: $\sigma = \sigma'$. In this case, we can rewrite Eq.7 as:

$$\left(\frac{e(\psi, g)}{e(\psi', g)}\right)^{\epsilon(\alpha - q)} = e(g, g)^{\epsilon(y' - y)} \tag{9}$$

Now We do a case analysis on whether $y = y'$.

Case 2.1: $y = y'$. As $(y, \psi, \sigma) \neq (y', \psi', \sigma')$, $\sigma = \sigma'$ and $y = y'$, we can infer that $\psi \neq \psi'$. In this case, since $y = y'$, we rewrite the Eq.9 as:

$$\left(\frac{e(\psi, g)}{e(\psi', g)}\right)^{\epsilon(\alpha - q)} = 1 \tag{10}$$

As $\psi \neq \psi'$, i.e., $\frac{e(\psi,g)}{e(\psi',g)} \neq 1$, and $\epsilon \neq 0$, we can obtain $\alpha = q$ from Eq.10. In PAT, q is known to the A(i.e., A can find $SK = \alpha$). As we proved in Theorem 2, if A can find $SK = \alpha$, we can can construction an algorithm B to solve the instance of the t-SDH problem. Thus, A cannot find a valid forged $(y, \psi, \sigma) \neq (y', \psi', \sigma')$ and $y = y'$ with non-negligible probability.

Case 2.2: $y \neq y'$. From Eq.9 and $y \neq y'$, we can imply that $\alpha \neq q$. In this case, we show how to construct an algorithm B, using the A, that can break the t-BSDH Assumption with a valid solution $(-q, \left(\frac{e(\psi,v)}{e(\psi',v)}\right)^{\frac{1}{y'-y}})$.

We denote ψ as g^{θ} and ψ' as $g^{\theta'}$, and rewrite Eq.9 as :

$$\begin{aligned} &\left(\frac{e(\psi, v)}{e(\psi', v)}\right)^{(\alpha - q)} = \frac{e(g, v)^{-y}}{e(g, v)^{-y'}} \\ &\theta(\alpha - q) + y = \theta'(\alpha - q) + y' \\ &\frac{(\theta - \theta')}{y' - y} = \frac{1}{\alpha - q} \end{aligned} \tag{11}$$

Therefore, algorithm B can compute

$$\left(\frac{e(\psi, v)}{e(\psi', v)}\right)^{\frac{1}{y'-y}} = e(g, v)^{\frac{(\theta - \theta')}{y'-y}} = e(g, g)^{\frac{1}{\alpha - q}} \tag{12}$$

and returns $(-q, e(g, g)^{\frac{1}{\alpha - q}})$ as a solution for t-BSDH instance. It is easy to see that the success probability of solving the instance is the same as the success probability of the adversity, and the time required is a small constant larger than the time required by the adversary.

Therefore, the security of our PAT construction is proved.

Theorem 4. *If an existed probabilistic polynomial time adversity A can convince the querier with an invalid query result for Selection Query, Projection Query, Join Query, Weighted SUM Query, Polynomial Query or Variance Query in our proposed scheme, we can construct an algorithm B using A to break either AORDS or PAT.*

Proof. With regard to Selection Query, Projection Query, Join Query and Weighted SUM Query, the querier directly verifies the completeness and freshness of the query result using $AORDS$ and checks its integrity using PAT. Therefore, if A can convince the querier with an invalid result with non-negligible probability, it can break either $AORDS$ or PAT, which have been proved to be secure above.

For Weighted Exponentiation SUM Query and Variance Query, the difference between them and the other query types is the tag generation outsourcing. As described in Section 4.4, the querier can outsource some tag generation and aggregation to A and easily achieve more than 99.99% confidence security. If the client processes all the tag generation and aggregation locally, the Weighted Exponentiation SUM Query and Variance Query become purely based on $AORDS$ or PAT. Therefore, if A can convince the querier with an invalid result with non-negligible probability, it can break either $AORDS$ or PAT, which have been proved to be secure above.

5.2 Performance Evaluation

In this section, we numerically evaluate the performance of our proposed scheme in terms of computational complexity, communication complexity and and storage overhead. For simplicity, we denote the complexity of one multiplication operation and one exponentiation operation on Group G as MUL and EXP[1] respectively. Notably, in our evaluation on computational complexity, both the cheap $Hash$ operation and expensive operations such as as EXP, MUL and $Pairing$ are presented for completeness. However, in practice $Hash$ operations can be very efficiently performed by contemporary devices (its execution time is usually several magnitudes less than that for EXP, MUL and $Pairing$ operations operations). Therefore, when evaluating the computational complexity we mainly focus on comparing the number of EXP, MUL and $Pairing$ operations. Due to the space limit, this section sketches the performance evaluation.

Database Pre-processing: Before outsourcing the database to the cloud, the owner needs to generate the authentication tags σ, auxiliary information AU and root state information $State_R$ for the database. With regard to the tag generation, the owner performs $O(sn)MUL + O(sn)EXP$ operations, where n is the number of tuples in the database and s is the attribute number in each tuple. For the computation of AU and $State_R$, the owner needs $O(n)Hash$ and $O(1)Sig$ respectively, where Sig is the signature operation. As all the generated tags and AU need to be outsourced to the cloud server, the communication complexity is $O(n)|G| + O(n)|Hash| + O(1)|Sig|$, where $|G|, |Hash|$ and $|Sig|$ are the size of a group element, hash value, and signature respectively. The cloud side storage complexity is $O(n)|G| + O(sn)|Hash| + O(1)|Sig|$.

Update: To modify or insert a tuple in the outsourced database, our proposed scheme requires $O(s)MUL + O(s)EXP$ operations on the owner side to generate the new tag and $O(log\ n)Hash$ and one Sig operation to update the auxiliary information and root state information. For deleting a tuple,

[1] When the operation is on the elliptic curve, EXP means scalar multiplication operation and MUL means one point addition operation.

		Ref.[10]	Ref.[16]	Ref.[19]	Our Scheme
Data Pre-Processing	$Comp._C$	$O(sn)Hash$ $+O(1)Sig$	$O(sn)\mathbf{EXP}$	$O(n)Hash$ $+O(n)\mathbf{EXP}$ $+O(1)Sig$	$O(sn)MUL + O(sn)\mathbf{EXP}$ $+O(n)Hash$
	Comm.	$O(sn)\|Hash\|$ $+O(1)\|Sig\|$	$O(sn)\|AggSig\|$	$O(n)\|Hash\|$ $+O(n)\|Tag\|$ $+O(1)\|Sig\|$	$O(n)\|G\| + O(n)\|Hash\|$ $+O(1)\|Sig\|$
Stor.d Overhead		$O(sn)\|Hash\|$ $+O(1)\|Sig\|$	$O(sn)\|AggSig\|$	$O(n)\|Hash\|$ $+O(n)\|Tag\|$ $+O(1)\|Sig\|$	$O(n)\|G\| + O(sn)\|Hash\|$ $+O(1)\|Sig\|$
Update	$Comp._S$	$O(log\ n)Hash$	N/A	$O(log\ n)Hash$	$O(log\ n)Hash$
	$Comp._O$	$O(log\ n)Hash$	$O(s)\mathbf{EXP}$	$O(log\ n)Hash$ $+O(s)\mathbf{EXP}$	$O(log\ n)Hash$ $O(s)MUL + O(s)\mathbf{EXP}$
	Comm.	$O(z)$	$O(z)$	$O(z)$	$O(z)$
Selection Query	$Comp._S$	N/A	$O(k)MUL$	N/A	$O(s+k)MUL + O(s+k)\mathbf{EXP}$
	$Comp._C$	$O(sk)Hash$	$O(k)\mathbf{EXP}$	$O(k)Hash$ $+O(k)\mathbf{EXP}$	$O(k)Hash + O(1)\mathbf{EXP}+$ $O(1)MUL + O(1)Pairing$
	Comm.	$O(slog\ n)\|Hash\|$	$O(k)\|Bitmap\|$	$O(log\ n)\|Hash\|$ $+O(k)\|Tag\|$	$O(log\ n)\|Hash\| + O(1)\|G\|$
Projection Query	$Comp._S$	N/A	$O(mn)MUL$	$O(n)MUL$	$O(s+n)MUL + O(s+n)\mathbf{EXP}$
	$Comp._C$	$O(mn)Hash$	$O(mn)\mathbf{EXP}$	$O(n)Hash$ $+O(n)\mathbf{EXP}$	$O(n)Hash + O(1)\mathbf{EXP}+$ $O(1)MUL + O(1)Pairing$
	Comm.	$O(mlog\ n)\|Hash\|$	$O(n)\|Bitmap\|$	$O(log\ n)\|Hash\|$ $+O(n)\|Tag\|$	$O(log\ n)\|Hash\| + O(1)\|G\|$
Join Query	$Comp._S$	N/A	$O(n)MUL$	$O(n)MUL$	$O(s+k)MUL + O(s+k)\mathbf{EXP}$
	$Comp._C$	$O(nlog\ n)Hash$	$O(n)\mathbf{EXP}$	$O(n)Hash$ $+O(n)\mathbf{EXP}$	$O(n)Hash + O(1)\mathbf{EXP}+$ $O(1)MUL + O(1)Pairing$
	Comm.	$O(nlog\ n)\|Hash\|$ $+\|\hat{R}\|$	$O(n)Bitmap$ $+\|\hat{R}\|$	$O(log\ n)\|Hash\|+$ $O(n)\|Tag\| + \|\hat{R}\|$	$O(log\ n)\|Hash\| + O(1)\|G\|$ $+\|\hat{R}\|$
Weighted SUM Query	$Comp._S$	N/A	N/A	N/A	$O(s+k)MUL + O(s+k)\mathbf{EXP}$
	$Comp._C$	N/A	N/A	$O(k)Hash$ $+O(k)\mathbf{EXP}$	$O(k)Hash + O(1)\mathbf{EXP}+$ $O(1)MUL + O(1)Pairing$
	Comm.	N/A	N/A	$O(log\ n)\|Hash\|$ $+O(k)\|Tag\|$	$O(log\ n)\|Hash\| + O(1)\|G\|$
Polynomial Query	$Comp._S$	N/A	N/A	N/A	$O(s+kx)MUL + O(s+kx)\mathbf{EXP}$
	$Comp._C$	N/A	N/A	N/A	$O(n)Hash + O(1)MUL$ $+O(1)\mathbf{EXP}+O(1)Pairing$
	Comm.	N/A	N/A	N/A	$O(log\ n)\|Hash\| + O(1)\|G\|$
Variance Query	$Comp._S$	N/A	N/A	N/A	$O(s+k)MUL + O(s+k)\mathbf{EXP}$
	$Comp._C$	N/A	N/A	N/A	$O(n)Hash + O(1)MUL$ $+O(1)\mathbf{EXP}+O(1)Pairing$
	Comm.	N/A	N/A	N/A	$O(log\ n)\|Hash\| + O(1)\|G\|$

Table 1. Complexity Summary: In this table, n is the number of tuples in the database, s is the number of attributes in each tuple, Sig is the sign operation for signature function, z is the number of modified tuples, k is the number of tuples satisfying the query condition, m is the attributes chosen in projection, $|G|, |Hash|$ and $|Sig|$ are the size of a group element, hash value, and signature respectively. $|Tag|$ is the size of authentication tag in Ref.[19]. $|\hat{R}|$ denotes the attributes in the projection query result that does not match the join query condition. $|AggSig|$ and $|Bitmap|$ is the size of aggregated signature and associated tuple information in Ref.[16].

$O(log\ n)Hash$ and one Sig operations are required in our scheme. Since the data transmitted for update operation are the new root state information, the new tuples and the new tags, the communication complexity is $O(z)$, where z is the number of modified tuples. On the cloud server side, $O(log\ n)Hash$ and one Sig operations shall be executed to update the database.

Selection Query: For selection query, the client needs $O(k)Hash$ operations to check the result's completeness and freshness, where k is the number of tuples satisfying the query condition. Moreover, 5 $Pairing$ operations and no more than 8 EXP and 8 MUL operations are needed to ensure the integrity of result (e.g., for a query result consists of 100 tuples, only $2MUL$ and $2EXP$ operations are required as we discussed in Section 4.4). On the cloud side, it performs

$O(s+k)MUL$ and $O(s+k)EXP$ operations to generate the proof information. The communication complexity is $O(log\ n)|Hash| + O(1)|G|$ since our proposed scheme aggregates the proof information for integrity check into 3 elements.

Projection Query: For a projection query, our proposed scheme requires the client to perform $O(n)Hash$ operations to verify the result's completeness and freshness. Since the most integrity checking tasks are moved to the cloud server, the client performs $O(1)MUL$, $O(1)EXP$ and $O(1)Pairing$ operations. On the cloud server side, $O(s+k)MUL + O(s+k)MUL$ operations shall be performed to produce all the proof information. By aggregating the integrity checking proof information into 3 elements, our scheme keeps the communication complexity as $O(log\ n)|Hash| + O(1)|G|$.

Join Query: To verify a join query, the our scheme requires the client first to perform two projection query for freshness checking, each of which causes a computational complexity of $O(n)Hash+O(1)MUL+O(1)EXP+O(1)Pairing$ and a communication complexity of $O(log\ n)|Hash| + O(1)|G| + |\hat{R}|$, where $|\hat{R}|$ denotes attributes in the projection query result that does not match the join query condition. To check the integrity of the join query result, the client needs to perform additional $O(1)MUL+O(1)EXP + O(1)Pairing$ operations, and the communication complexity is $O(1)|G|$. On the cloud server side, the computational complexity is $O(s+k)MUL + O(s+k)EXP$.

Weighted SUM Query: In our proposed scheme, the only difference between weighted SUM query and selection query is the random numbers chosen in the challenge message generation stage: in weighted SUM query, the client replaces the random numbers with the weight values. As shown in Table 1, the computational complexity and communication complexity on the client side for weighted SUM query are $O(k)Hash+O(1)MUL+O(1)EXP+O(1)Pairing$ and $O(log\ n)|Hash|+O(1)|G|$ respectively, which are same as the selection query. On cloud server side, the computational complexity is $O(s+k)MUL+O(s+k)EXP$.

Polynomial Query and Variance Query: To verify a polynomial query, our proposed scheme needs the client side to perform u more MUL and EXP operations than the weighted SUM query (e.g., $u = 2$ when $k = 100$ as we discussed in Section 4.4). Therefore, the computation complexity on the client side is $O(n)Hash + O(1)MUL+O(1)EXP + O(1)Pairing$. The communication complexity is $O(log\ n)|Hash|+O(1)|G|$. As our scheme requires the cloud server to generate the authentication tags for the exponentiation values, the computational complexity on the cloud side is $O(s+kx)MUL + O(s+kx)EXP$ as shown in Table 1, where x is the degree of exponentiation value. With regard to variance query, as it is purely based on weighted SUM query and polynomial query, its computational complexity on the client side and the communication complexity are $O(n)Hash + O(1)MUL+O(1)EXP + O(1)Pairing$ and $O(log\ n)|Hash| + O(1)|G|$ respectively. The cloud server it needs to perform $O(s+k)MUL + O(s+k)EXP$ operations.

Comparison: Now we compare our proposed scheme with existing schemes [10, 16, 19] and show the result in Table 1. For Pre-process, although our proposed requires more computational cost and communication cost, they are one-time

cost and will not influence the real-time query performance. For storage overhead, Table 1 shows that our proposed scheme achieves a comparable complexity than Ref.[10, 16]. Although Ref.[19] saves some hash values in the server storage overhead as compared with our scheme, it introduces more computational and communication cost on the client side. Since the cloud server is always much more powerful than the client, storing these hash values will not introduces any burden to the cloud.

For *Update*, our scheme introduces more MUL operations to the owner side for the tag update. However, the EXP operations needed in our scheme and Ref.[16, 19] are comparable, which is about 10 times more expensive than the MUL operation[6]. Thus, our scheme can achieve a similar cost as compared to Ref.[16, 19]. For communication cost, our scheme achieves the comparable complexity than existing schemes[10, 16, 19].

For *Selection Query*, *Projection Query* and *Join Query*, table 1 demonstrates that our proposed scheme outperforms Ref.[10, 16, 19] in terms of computational complexity and communication complexity. Specifically, Ref.[16, 19] requires a linear number of expensive EXP operations. Differently, by moving computation tasks to the cloud, our scheme only requires the client to perform cheap $Hash$ operations and a constant number of EXP, MUL and *Pairing* operations. Instead of transmitting all the tags associated with the query result in Ref.[19], our scheme aggregates the authentication tags into 3 elements. As compared with Ref.[10], our scheme not only removes the factors m,s in communication cost and computational cost for *Selection Query* and *Projection Query* respectively, it also reduces the $O(nlog\ n)$ communication complexity in *Join Query* to $O(log\ n)$.

For $Weighted\ SUM\ Query$, our proposed scheme only introduces a constant number of EXP, MUL and *Pairing* operations to the client, while moving other operations to the cloud server side. In comparison, Ref.[19] introduces $O(k)$ EXP operations to the client. In addition, our scheme enables the aggregation of authentication tags into 3 elements, and thus outperforms Ref.[19] in communication complexity, which requires the transmission of all the tags.

6 Conclusion

In this work, we present an efficient verifiable SQL query scheme in the setting of outsourced dynamic databases. Our proposed scheme not only allows the cloud server to perform most computational tasks for the query verification, but also aggregates the proof information to reduce communication cost. Compared with existing schemes, we reduce the expensive computational operations (e.g., exponentiation operation) on client side at least from a linear level to $O(1)$ and constrain the communication cost to $logn$, where n is the number of tuples in a table. In addition, our proposed scheme supports more queries than previous works, including polynomial queries, variance queries and other linear queries. Moreover, our proposed polynomial based authentication tag can also be used as an independent solution for other related application, such as database auditing,

encrypted key word search, etc. One interesting future work is to enable more powerful SQL queries in verifiable ways.

References

1. D. Boneh and X. Boyen. Short signatures without random oracles. pages 56–73. Springer-Verlag, 2004.
2. D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Proceedings of the 22nd international conference on Theory and applications of cryptographic techniques*, EUROCRYPT'03, pages 416–432, Berlin, Heidelberg, 2003. Springer-Verlag.
3. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '01, pages 514–532, London, UK, UK, 2001. Springer-Verlag.
4. W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, Sept. 1976.
5. V. Goyal. Reducing trust in the pkg in identity based cryptosystems. In *Proceedings of the 27th annual international cryptology conference on Advances in cryptology*, CRYPTO'07, pages 430–447, Berlin, Heidelberg, 2007. Springer-Verlag.
6. G. Grewal, R. Azarderakhsh, P. Longa, S. Hu, and D. Jao. Efficient implementation of bilinear pairings on arm processors. *IACR Cryptology ePrint Archive*, 2012:408, 2012.
7. A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, pages 177–194, 2010.
8. F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 121–132, New York, NY, USA, 2006. ACM.
9. R. C. Merkle. A certified digital signature. In *Proceedings on Advances in cryptology*, CRYPTO '89, pages 218–238, New York, NY, USA, 1989. Springer-Verlag New York, Inc.
10. K. Mouratidis, D. Sacharidis, and H. Pang. Partially materialized digest scheme: an efficient verification method for outsourced databases. *The VLDB Journal*, 18(1):363–381, Jan. 2009.
11. E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *Trans. Storage*, 2(2):107–138, May 2006.
12. M. Narasimha and G. Tsudik. Authentication of outsourced databases using signature aggregation and chaining. In *Proceedings of the 11th international conference on Database Systems for Advanced Applications*, DASFAA'06, pages 420–436, Berlin, Heidelberg, 2006. Springer-Verlag.
13. G. Nuckolls. Verified query results from hybrid authentication trees. In *Proceedings of the 19th annual IFIP WG 11.3 working conference on Data and Applications Security*, DBSec'05, pages 84–98, Berlin, Heidelberg, 2005. Springer-Verlag.
14. B. Palazzi, M. Pizzonia, and S. Pucacco. Query racing: Fast completeness certification of query results. In S. Foresti and S. Jajodia, editors, *Data and Applications Security and Privacy XXIV*, volume 6166 of *Lecture Notes in Computer Science*, pages 177–192. Springer Berlin Heidelberg, 2010.

15. H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 407–418, New York, NY, USA, 2005. ACM.
16. H. Pang, J. Zhang, and K. Mouratidis. Scalable verification for outsourced dynamic databases. *Proc. VLDB Endow.*, 2(1):802–813, Aug. 2009.
17. V. Shoup. *A computational introduction to number theory and algebra*. Cambridge University Press, New York, NY, USA, 2005.
18. G. Timothy and M. M. Peter. The nist definition of cloud computing. NIST SP - 800-145, September 2011.
19. Q. Zheng, S. Xu, and G. Ateniese. Efficient query integrity for outsourced dynamic databases. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, CCSW '12, pages 71–82, New York, NY, USA, 2012. ACM.