Towards Efficient Verifiable SQL Query for Outsourced Dynamic Databases in Cloud

Abstract. With the raising trend of outsourcing databases to the cloud server, it is important to efficiently and securely assure that the clients' queries on the databases are executed correctly. To address this issue, server schemes have been proposed based on various cryptographic tools. However, these existing schemes have limitations in either communication cost or computational cost for verification. Meanwhile, only four types of SQL functional queries are supported in these schemes. It still remains as an open problem to design a verifiable SQL query scheme that provides affordable storage overhead, communication cost, computational cost and more SQL functional queries.
In this paper, we investigate this open problem and propose an efficient verifiable SQL query scheme for outsourced dynamic databases. Different from the previous state-of-the-art schemes, we reduce the complexity of storage overhead from $O(mn)$ to $O(n)$ and move most computation tasks from client side to cloud server side. Compared with the recently proposed scheme that also achieves $O(n)$ storage overhead, we not only cut the communication complexity for verification from $O(n)$ to $O(log^n)$, but also release the client from $O(n)$ exponentiation operations to $O(1)$. In addition, our proposed scheme improves the previous ones by allowing more aggregate queries including variance query, weighted exponentiation sum query of any degrees, etc. Thorough analysis shows the efficiency and scalability of our proposed scheme. The security of our scheme is proved based on Strong Diffie-Hellman Assumption, Bilinear Strong Diffie-Hellman Assumption and Computational Diffie-Hellman Assumption.

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

1 Introduction

The unprecedented advantages of cloud computing (e.t., on-demand self-service, transference of risk and resource elasticity, etc[19]) make it become a raising prevalent choice for database outsourcing. To take advantages of cloud computing, millions of organizations and individuals have moved their databases as well as services to the cloud infrastructures including Amazon EC2, Microsoft Azure and Google Cloud. Utilizing cloud platforms as data centers is increasingly becoming a clear trend in today's Internet.

Although cloud computing introduces the above advantages more appealing than ever, it also causes challenging security concerns due to the cloud servers'

shortages. For databases that outsourced to cloud for query, there are three major security concerns: *Integrity*, *Completeness* and *Freshness*. Specifically, for any query request Qry with range $[a, b]$ on the database of version Ver, we need to assure that the query result is correct(i.e., the data executed for query and the result returned have not been modified), complete(i.e., the result cannot belong to some other ranges $[a', b']$, where $a' > a$ and $b' < b$) and up-to-date(i.e., the query must be executed on the database of latest version Ver instead of any other version Ver'). To ensure the users' confidence of their query results' integrity, completeness as well as freshness, a series of schemes have been proposed[14, 16, 9, 12, 13, 5, 17, 11, 15, 20]. These existing schemes can be mainly divided into two groups: the tree-based scheme[14, 9, 5, 11, 15] and the signature-based scheme[16, 12, 13, 17]. The best tree-based scheme[9] enables the users to simultaneously check the integrity, completeness and freshness for the query result. However, this scheme requires storage overhead linear to the tuple and attribute numbers in the databases, and only supports selection query, projection query and join queries with pre-defined keyword attributes. Moreover, the communication complexity of Ref.[9] is linear to the number of tuples and attributes associated with the query result. Compared with the tree-based schemes, the best signature-based scheme[17] can achieve the same functionality with lower communication complexity. While, this scheme introduces exponentiation operations linear to the tuple number associated with query result on user side and makes no contributions to the storage overhead as well as SQL functional query types. To overcome the limitations in the previous schemes, Zheng et.al[20] recently proposed a novel query integrity check approach for outsourced dynamic databases. By utilizing Merkle hash tree and Homomorphic Linear Tag (HLT), their proposed approach makes the storage overhead on cloud server side independent to the number of attributes in each tuple and supports flexible join queries as well as aggregate queries (only weighted sum queries). Nevertheless, each query verification in Ref.[20] introduces communication and computational complexities on user side linear to the number of tuples associated with the query result, which limit its scalability in database size and query ranges. To our best knowledge, there is no existing scheme proposed for verifiable SQL query on outsourced dynamic databases that achieves affordable communication cost, computational cost, storage overhead and more types of SQL functional query at the same time. The problem of designing such a verifiable SQL query scheme is still largely open.

In this paper, we investigate this open problem and 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[10], our scheme can efficiently and simultaneously check the integrity, completeness and freshness of SQL query in cloud. Moreover, besides SQL query types achieved in the previous schemes, our scheme supports additional aggregate queries including variance query, weighted exponentiation sum query of any degrees, etc. 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 authentication tags $\{\sigma_i\}$ for each tuple $\{r_i\}, 1 \leq i \leq n$ in the database. Then,

the database, tags $\{\sigma_i\}$ and auxiliary information Au of 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 returns the result and the corresponding auxiliary information to it. The client then runs the first part of $QueryVerify$ algorithm to check the completeness and freshness of the result. After that, the client generates a challenge message for integrity check and sends it to the cloud. Based on the received message, the cloud server produces the proof information that it actually executes the query correctly and sends it to the client. On receiving the proof information, the client runs the second part of $QueryVerify$ algorithm to check the correctness of the query result. In our proposed scheme, we tailor a constant size polynomial commitment technique[8] and allow the cloud server to aggregate all the proof information into three elements to reduce communication cost. In addition, our authentication tag also enables the client to move most computation tasks to the cloud server. Any client can perform the verification process without the help of database owner, who is only responsible for the database update after data outsourcing. Thorough analysis shows that our proposed scheme is efficient and scalable. We prove that our proposed scheme is secure under based on Computational Diffie-Hellman Assumption(CDH), Strong Diffie-Hellman(SDH) Assumption and 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, which reduces storage overhead, communication cost and computational cost simultaneously compared with the existing schemes.
- Our proposed scheme achieves aggregate queries that cannot be supported in the previous schemes, including variance query, weighted exponentiation sum query of any degrees, etc.
- We formally prove the security of our proposed scheme and valid the its advantages through thorough 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: We review and discuss the related works in Section 2. Section 3 describes the models and assumptions of our scheme. In Section 4, we introduce the technique preliminaries of this work, which is followed by the solution space in Section5. In Section 6, we analyze our proposed scheme in terms of security and performance. We conclude our paper in Section 7.

2 Related Work

There are mainly two groups of approaches for verifiable query on outsourced databases: the tree-based approaches[14, 9, 5, 11, 15] and the signature-based approaches[16, 12, 13, 17].

In the tree-based approaches, a Merkle hash tree[10] or its variants[14, 9, 5, 11, 15] are always used to assure the integrity of the data associated with the leaves for query. Among the existing tree-based approaches, the best one is proposed by Li et.al.[9], which introduces an embedded Merkle B^+-Tree structure to reduce I/O operations and first considers the dynamic databases. In Ref.[9], the integrity, completeness and freshness of the query result can be achieved at the same time with simple hash operations. However, the communication complexity of Ref.[9] is linear to the number of tuples and attributes associated with the query results, which limits its performance for queries on large ranges. In addition, Ref.[9] requires $O(mn)$ storage overhead on cloud server side and only supports selection query, projection query as well as joint query with pre-defined keyword attributes.

The signature-based approaches[16, 12, 13, 17] always utilize signature aggregation technique[2] and its variants to aggregate the proof information of query result. Compared with the tree-based approaches, this kind of approaches greatly reduce the communication complexity for query verification. But these signature-based approaches require more computational cost to handle query verification, especially for some powerful queries(e.g., projection query, joint query). In the best signature-based approach[17], a chaining signing technique is introduced to facilitate queries and the proof information of query result are aggregated into one single signature. However, this approach needs the client to perform expensive exponential operations linear to the number of tuples and attributes associated with the query result, which limits its application for large range queries. No improvement is brought for both storage overhead and SQL functional query types.

To improve the previous approaches, Zheng et.al[20] proposed a SQL query integrity check scheme for outsourced dynamic database based on Merkle hash tree and HLT. Compared with the previous schemes[9, 17], the storage overhead on cloud side in Ref.[20] is reduced from $O(mn)$ to $O(n)$. Moreover, their proposed scheme supports flexible joint query and weighted sum query. However, Ref.[20] requires the transmission of authentication tags that linear to the number of tuples associated with the query results, which can be worse in their weighted sum query(i.e, instead of returning the tag for the single sum query result, tag for every tuple involved in the sum computation are needed). What is more, the client in Ref.[20] has to perform all integrity verification operations, which are linear to the number of tuples associated with the result and make it become impractical to be applied on large database or large query ranges.

3 Model and Assumption

3.1 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 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 by running $QueryVerify$ algorithm. W.l.o.g, we define the one-round version of our system model as below.

- **KeyGen:** Given a security parameter λ, the randomized $KeyGen$ algorithm produces the public key and private key for the system as (PK, SK)
- **Setup:** Given a database DTB, the public key PK and private key SK, the *Setup* algorithm outputs the authentication tag σ, auxiliary information AU and root state information $State_R$ of Merkle hash tree, in which σ and AU will be outsourced to the cloud server and $State_R$ will be published.
- **Update:** Given the public key PK and private key SK, the $Update$ algorithm generates the updated authentication tag σ', auxiliary information AU' and root state information $State'_R$ and verifies whether or not the outsourced database is updated correctly.
- **Prove:** Given the public key PK, a query request and a challenge message $Chall$, the *Prove* algorithm generates the proof information Prf.
- **QueryVerify:** Given the public key PK, the query result and the corresponding auxiliary information, root state information, the proof information Prf, the *QueryVerify* algorithm checks the integrity, completeness and freshness of the query result and outputs result as either *accept* or $reject$.

3.2 Security Model

We consider the cloud server as untrusted and potentially malicious, which is consistent with the previous schemes[9, 17, 20]. 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 makes the $QueryVerify$ algorithm output *accept*, it must execute the query correctly on the right query range and up-to-date database. For the correctness, we require that the $QueryVerify$ algorithm outputs *accept* for 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 $State_R$ 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 and asks it to generate the corresponding updated authentication tag σ', auxiliary information AU' and root state information $State'_R$. TA runs $Update(D', SK, PK) \rightarrow (\sigma', AU', State'_R)$ and returns the output to Adv. TA publishes $State'_R$.
- With regard to the updated database 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 and Prf without executing the query correctly and make C output $VRst$ as *accept*.

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

3.3 Assumption

Definition 2. ***Computational Diffie-Hellman (CDH) Assumption****[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****[6]*

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.

4 Technique Preliminaries

4.1 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, it has the following properties:

– Bilinear: for any $a, b \overset{R}{\leftarrow} Z_p^*$ and $g_1, g_2 \in G$, there exists $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$.
– 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.

4.2 Merkle Hash Tree

Merkle hash tree is first proposed in Ref.[10] 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 always 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 that leads the checking node to root, the verifier can calculate the value of root. If the calculated root hash value is equal to the published value, the checking data is valid; otherwise, the data has been modified. For more details, please refer to Ref.[10].

4.3 Constant Size Polynomial Commitment

Secure polynomial commitment scheme is proposed to allow a committer to commit a polynomial with a short string. Based on the algebraic property of 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.[8] proposed a constant size polynomial commitment scheme. In their construction, to verify the correctness of a polynomial evaluation $f(r)$, where r is a random index on $f(x)$, the committer of $f(x)$ can aggregate all the proof information into a single element. Specially, the construction of polynomial commitment scheme with constant communication size in Ref.[8] 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 $\psi = g^{f_{\boldsymbol{w}}(\alpha)}$.

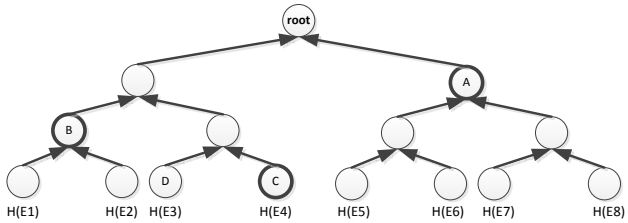

Fig. 1. Merkle Hash Tree: the auxiliary information AU for element E_3 is: the node value of Nodes A,B and C.

– **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:

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

For the detail security and correctness of this polynomial commitment scheme, please refer to Ref.[8].

5 Solution Space

In this section, we first introduce the 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.

5.1 Authenticated Outsourced Ordered Data Set

Authenticated Outsourced ordered data set (AORDS) is constructed based on the Merkle hash tree[10]. Let E be an ordered set of elements and $Sign()$ as 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 concatenation of its children's values. The state information of 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 root and the values of these nodes's siblings' as shown in Fig.1.
– **Update**$(SK, E) \rightarrow (E', AU', State'_R)$: Here we describe the update for data modification in E. Our construction can also support insertion and deletion update, which are the same as introduced in Ref.[9, 20]. Due to the space limitation, we leave details of insertion and deletion operations for reference

[9, 20]. Given a some modified elements E_i', the cloud server replaces the E_i with E_i' and also updates the corresponding $AU, State_R$ to $AU', State_R'$ based on E_i'. $State_R'$ is sent to the data owner. The owner computes the new root value $State_R$" based on E_i' and checks $State_R'' \stackrel{?}{=} State_R'$. If so, the owner publishes $State_R'$ as the new root state information; otherwise, reject the update.

– **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 then recomputes the root value $Root'$ based on the AU and Rst. After decrypting the published signature of root $Sign(Root)$ with PK, the client checks $Root \stackrel{?}{=} Root'$. $VRst = accpet$ if $Root = Root'$; otherwise $VRst = reject$.
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 $Root$ and $Root'$. $VRst = accpet$ if $Root = Root'$; otherwise $VRst = reject$.

5.2 Polynomial Based Authentication Tag

Construction Description In this section, we propose a polynomial based authentication tag (PAT), which can be used to verity the integrity of query result. We consider a database DTB consists of n tuples $\{r_1, r_2, \cdots, r_n\}$, each of which has s attributes $\{a_0, a_1, \cdots, a_{s-1}\}$. 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 g be a generator 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**$(1^\lambda) \rightarrow (PK, SK)$: Choose a random prime $p(\lambda$ bits security) and generate a random signing keypair $((spk, ssk) \stackrel{R}{\leftarrow} Sign())$ using BLS signature[1]. Choose two random numbers $\alpha, \epsilon \stackrel{R}{\leftarrow} Z_p^*$ and compute $v \leftarrow g^\epsilon$, $\kappa \leftarrow g^{\alpha\epsilon}$ as well as $\{g^{\alpha^j}\}_{j=0}^{s-1}$. The public and private keys are

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

– **Setup**$(PK, SK) \rightarrow (\sigma, \tau)$: Choose a random table name $name$ from some sufficiently large domain(e.g., Z_p^*). Let τ_0 be "$name||n$"; the table tag τ is τ_0 together with a signature on τ_0 under ssk: $\tau \leftarrow \tau_0||Sign(\tau_0)$. For each tuple $r_i, 1 \leq i \leq n$, an authentication tag is computed as:

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

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

– **QueryVerify**$(PK, \tau) \rightarrow Chall$:
Stage 1: Verify the signature on τ: if the signature is not valid, reject and halt; otherwise, parse 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 $v_i \overset{R}{\leftarrow} Z_p^*$ for these tuples and gets $k-$elements set $K = (i, v_i)$. Choose a random number $q \overset{R}{\leftarrow} Z_p^*$. Produce the challenge message as

$$Chall = \{q, K\}$$

Challenge the server with $Chall$.
– **Prove**$(PK, Chall, \tau) \rightarrow (\psi, y, \sigma)$: Compute

$$\sigma = \prod_{(i,v_i) \in K} \sigma_i^{v_i} \tag{2}$$

We denote vector

$$\boldsymbol{A} = (\sum_{(i,v_i) \in K} v_i * r_i.a_0, \cdots, \sum_{(i,v_i) \in K} v_i * r_i.a_{s-1})$$

Compute

$$y = f_{\boldsymbol{A}}(q) \tag{3}$$

Since polynomials $f(x) \in Z[x]$ have the algebraic property that $(x-q)$ perfectly divides the polynomial $f(x) - f(q), q \overset{R}{\leftarrow} Z_p^*$. Now, divide the polynomial $f_{\boldsymbol{A}}(x) - f_{\boldsymbol{A}}(q)$ with $(x-q)$ and denote the coefficients vector of the resulting quotient polynomial as $\boldsymbol{w} = (w_0, w_1, \cdots, w_{s-1})$, that is, $f_{\boldsymbol{w}}(x) \equiv \frac{f_{\boldsymbol{A}}(x) - f_{\boldsymbol{A}}(q)}{x-q}$. Produce

$$\psi = \prod_{j=0}^{s-1} (g^{\alpha^j})^{w_j} = g^{f_{\boldsymbol{w}}(\alpha)} \tag{4}$$

Response $Prf = \{\psi, y, \sigma\}$.
– **QueryVerify**$(PK, Prf) \rightarrow VRst$:
Stage 2: On receiving the proof response Prf, compute

$$\eta_i = v^{-H(name||i)v_i}, (i, v_i) \in K \tag{5}$$

$$\eta = \prod_{(i,v_i) \in K} \eta_i \tag{6}$$

where $v = g^{\epsilon}$ in PK. Parse Prf as $\{\psi, y, \sigma\}$ and check

$$e(\psi, \kappa \cdot v^{-q}) \overset{?}{=} e(\sigma \cdot \eta, g) \cdot e(g^{-y}, v) \tag{7}$$

where $\kappa = g^{\epsilon\alpha}$ in PK. The $QueryVerify$ algorithm outputs $VRst = accept$ if Eq.7 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. Consider the left part and right part of Eq.7, we have: Left Part:

$$\begin{aligned} & e(\psi, \kappa \cdot v^{-q}) \\ & = e(g^{f_{\boldsymbol{w}}(\alpha)}, g^{\epsilon(\alpha - q)}) \\ & = e(g, g)^{\frac{f_{\boldsymbol{A}}(\alpha) - f_{\boldsymbol{A}}(q)}{\alpha - q} \cdot \epsilon(\alpha - q)} \\ & = e(g, g)^{\epsilon(f_{\boldsymbol{A}}(\alpha) - f_{\boldsymbol{A}}(q))} \end{aligned} \tag{8}$$

Right Part:

$$\begin{aligned} & e(\sigma \cdot \eta, g) \cdot e(g^{-y}, v) \\ & = e(g^{\epsilon(\sum_{(i,v_i)\in Q} H(name||i)v_i + f_{\boldsymbol{A}}(\alpha)) + \epsilon \sum_{(i,v_i)\in Q} -H(name||i)v_i} \\ & , g) \cdot e(g^{-f_{\boldsymbol{A}}(q)}, g^{\epsilon}) \\ & = e(g, g)^{\epsilon f_{\boldsymbol{A}}(\alpha)} \cdot e(g, g)^{-\epsilon f_{\boldsymbol{A}}(q)} \\ & = e(g, g)^{\epsilon(f_{\boldsymbol{A}}(\alpha) - f_{\boldsymbol{A}}(q))} \end{aligned} \tag{9}$$

Based on Eq.8 and Eq.9, 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 $k, 1 \leq k \leq n$ attributes in same position of different tuples(i.e.,$\{r_i.a_j, r_{i+c}.a_j, \cdots, r_{i+d}.a_j\}$) and their corresponding tags, we can calculate the tag for the sum of these k attributes as:

$$\begin{aligned} \sigma = \prod_{i=1}^{k} \sigma_i & = (g^{\sum_{i=0}^{k} H(name||i)} \cdot \prod_{i=0}^{k} \prod_{j=0}^{s-1} g^{r_i.a_j \alpha^j})^{\epsilon} \\ & = (g^{\sum_{i=0}^{k} H(name||i)} \cdot \prod_{j=0}^{s-1} g^{\sum_{i=0}^{k} r_i.a_j \alpha^j})^{\epsilon} \\ & = (g^{\sum_{i=0}^{k} H(name||i)} \cdot g^{f_{\mathfrak{S}}(\alpha)})^{\epsilon} \end{aligned} \tag{10}$$

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

Now, we describe how to construct the tag for any attribute's exponential value. For the exponential value $(r_i.a_j)^x$, $1 \leq i \leq n, 1 \leq j \leq s, x \in Z_p^*$, we can calculate its tag as:

$$exp_x.\sigma_{ij} = \sigma_{ij}^{(r_i.a_j)^x} \cdot v^{(-(r_i.a_j)^x + 1)H(name||i)} \tag{11}$$

$$= g^{H(name||i)\epsilon} \cdot (g^{f_{\boldsymbol{\beta_i}}(\alpha)\epsilon})^{(r_i.a_j)^x} \tag{12}$$

$$= (g^{H(name||i)} \cdot g^{f_{\boldsymbol{B_i}}(\alpha)})^{\epsilon}$$

where $v = g^{\epsilon}$ in PK, $B_{i,j} = (r_i.a_k)^x * \beta_{i,j}$ and $\boldsymbol{B_i} = \{B_{i,0}, B_{i,1}, \cdots, B_{i,s-1}\}$. It is easy to verify that the exponential tag still has additional homomorphic property similar to Eq.10. Meanwhile, the exponential tag fullfills the $Proof$ and $QueryVerify$ algorithms in our PAT construction. Due to the space limitation, we do not provide detail discussion here.

5.3 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 $\{a_0, a_1, \cdots, a_{s-1}\}$. 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 be a generator 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:

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

  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 $r_{n+1}.a_0 = U$. Run $AORDS.SetUp$ to produce the root state information $State_R$ and auxiliary information 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, $State_R$ on the server side to r'_i, AU', $State'_R$ with verification. The owner publish $State'_R$.
  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 outsource it to the cloud server together with r_i. Then, the owner runs $AORDS.Update$ to adds r_i and updates the corresponding AU, $State_R$ on the server side to AU', $State'_R$ with verification. The owner publish $State'_R$.
  Deletion: Suppose the database owner delete the tuple r_i. The owner runs $AORDS.Update$ and updates AU, $State_R$ on the server side to AU', $State'_R$ with verification. The owner publish $State'_R$.

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 $PAT.QueryVerify.Stage1$ 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 $PAT.QueryVerify.Stage2$ to verify the integrity of these k tuples. If the output $VRst$ is $accept$, accept Rst as the query result; otherwise, reject Rst.
- **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 $PAT.QueryVerify.Stage1$ 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 $PAT.QueryVerify.Stage2$ to verify the integrity of Rst. If the output $VRst = accept$, accept Rst as the query result; otherwise, reject Rst.
- **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, a_0 from TB_1" and Qry ="select a_t, a_0 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 $PAT.QueryVerify.Stage1$ 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 $PAT.QueryVerify.Stage2$ to verify the integrity of these tuples. If the output $VRst$ is $accept$, accept Rst as the query result; otherwise, reject Rst.
- **Aggregate Query**:
  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 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

$PAT.QueryVerify.Stage1$ to generate the challenge message $Chall = \{q, K\}$ and sends it the cloud server, in which the k random elements 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 $PAT.QueryVerify.Stage2$ to verify the integrity of the sum value. If the output $VRst$ is *accept*, accept Rst as the query result; otherwise, reject Rst.

Weighted Exponentiation SUM Query: Suppose a weighted SUM query $Qry =$ "select SUM$(c_i * a_t)^x$ 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 query condition. The client performs same as in $Weighted\ SUM\ Query$ 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:

$$\begin{aligned} exp_x.\sigma_{it} &= {\sigma_{it}}^{(r_i.a_t)^x} \cdot v^{(-(r_i.a_t)^x+1)\cdot\sum_{i=1}^{k} H(name||i)} \\ &= (g^{H(name||i)} \cdot g^{f_{\boldsymbol{B_i}}(\alpha)})^{\epsilon} \end{aligned} \tag{13}$$

where $B_{i,j} = (r_i.a_k)^x * \beta_{i,j}, 0 \leq j \leq s-1$ and $\boldsymbol{B_i} = \{B_{i,0}, B_{i,1}, \cdots, B_{i,s-1}\}$. Then, 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 in Section5.4). The cloud returns the tuples r_i as well as their tags σ_i to the client, where $i \in U$. The client runs $PAT.QueryVerify.Stage2$ to verify the integrity of tuples $r_i, i \in U$. If the output is $reject$, aborts; otherwise, the client generates the exponentiation tags $exp_x.\sigma_{it}$ and aggregates them as $\sigma' = \prod_{i \in U} exp_x.\sigma_{it}{}^{c_i}$. The cloud generates the second part of proof information Prf_2 as $\{\sigma'' = \prod exp_x.\sigma_{it}{}^{c_i}, i \in K, i \notin U\}$ and sends it to the client. The client then computes $\sigma = \sigma' * \sigma''$ and runs $PAT.QueryVerify.Stage2$ with $\{\psi, y, \sigma\}$ to verify the integrity of the sum value. If the output $VRst = accept$, accept Rst as the query result; otherwise, reject Rst.

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 similar to the exponentiation tag:

$$\sigma_{-a_m} = \sigma_{a_m}^{-1} \cdot v^{(1+1)\sum_{i=1}^{k} H(name||i)} \tag{14}$$

the clients can run $Weighted\ Exponentiation\ SUM\ Query$ algorithm to verify Rst. If the output $VRst$ is *accept*, accept Rst as the query result; otherwise, reject Rst.

Note that, our proposed can also supports other aggregate queries based on weighted sum query and weighted exponentiation sum query like variance query does. Due to the space limitation, we does not provide details here.

5.4 discussion

In this section, we discuss about how to choose the set U in weighted exponentiation sum query 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 $\frac{1}{C_k^u}$(e.g., $k = 100$, the client can set $u = 2$ to get 99.9899% confidence that the cloud server cannot guess the set U). If the cloud does not compute ψ rightly according to the right exponentiation tags, it has only $1 - \frac{1}{C_k^u}$ 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, our proposed scheme also allows the client to outsource the most computation tasks for calculating η in the $PAT.QueryVerify.Stage2$ to the cloud server, where $\eta = \prod_{(i,v_i)\in K} \eta_i$. Specifically, after 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 $\eta = \eta' \cdot \eta''$.

6 Analysis Of Our Proposed Scheme

6.1 Security Analysis

In this section, we first prove the security for the two building blocks for our proposed efficient verifiable SQL query scheme. Then, we give the security analysis of each query type in our scheme.

Security of AORDS

Theorem 1. *The design our AORDS is secure assuming the collision-resistance of the hash function is computationally infeasible 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[10]. 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.

Security of PAT We prove that our proposed polynomial based authentication tag is secure and unforgeable as below:

Theorem 2. *If t-SDH Assumption holds and an existed probabilistic polynomial time adversary A can forge $g^{f_c(x)}$, we can construct an algorithm B that outputs the solution to t-SDH problem based on A efficiently.*

Proof. Suppose A can forge $f^1_{\boldsymbol{c}}(\alpha)$ that achieves $g^{f^1_c(\alpha)} = g^{f_c(\alpha)}$, where $\boldsymbol{c}$ is the coefficient vector, he can obtain $g^{f^2_c(\alpha)} = g^{f^1_c(\alpha)}/g^{f_c(\alpha)} = g^{f^1_c(\alpha)-f_c(\alpha)}$. Since $f^1_{\boldsymbol{c}}(\alpha) = f_{\boldsymbol{c}}(\alpha)$ and $f^2_{\boldsymbol{c}}(\alpha) = 0$, α is a root of polynomial $f^2_{\boldsymbol{c}}(x)$. By factoring $f^2_{\boldsymbol{c}}(x)$[18], B can find $SK = \alpha$ and solve the instance of the t-SDH problem given by the system parameters.

Theorem 3. *If CDH problem is hard, BLS signature scheme is existentially unforgeable, t-SDH Assumption and 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:

$$e(\psi, \kappa \cdot v^{-q}) = e(\sigma \cdot \eta, g) \cdot e(g^{-y}, v) \tag{15}$$

$$e(\psi', \kappa \cdot v^{-q}) = e(\sigma' \cdot \eta, g) \cdot e(g^{-y'}, v) \tag{16}$$

Dividing Eq.15 with Eq.16, we obtain:

$$\frac{e(\psi, g)^{\epsilon(\alpha-q)}}{e(\psi', g)^{\epsilon(\alpha-q)}} = \frac{e(g,g)^{\epsilon E - \sum_{(i,v_i)\in Q} H(name||i)v_i - y}}{e(g,g)^{\epsilon E' - \sum_{(i,v_i)\in Q} H(name||i)v_i - y'}}$$

$$\left(\frac{e(\psi, g)}{e(\psi', g)}\right)^{\epsilon(\alpha-q)} = e(g,g)^{\epsilon(E-E')+y'-y} \tag{17}$$

where we denote σ as $g^{E\epsilon}$ and σ' as $g^{E'\epsilon}$ for simplicity.

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

Case 1: $\sigma \neq \sigma'$. As $g^{E\epsilon} = \sigma$ and $g^{E'\epsilon} = \sigma'$, we can infer $E \neq E'$. Since $e(g,g)^{y'-y}$, $e(g,g)^{E\epsilon} = e(g, \sigma)$ and $\left(\frac{e(\psi,g)}{e(\psi',g)}\right)^{\epsilon(\alpha-q)}$ are knowledge to a, we rewrite Eq.17 as

$$\Upsilon = e(g,g)^{\epsilon E'} \tag{18}$$

where we denote $\Upsilon = e(g,g)^{E\epsilon+y'-y} / \left(\frac{e(\psi,g)}{e(\psi',g)}\right)^{\epsilon(\alpha-q)}$ as knowledge to the adversary.

Suppose any A can find $E' \neq E$ and makes Eq.18 hold with non-negligible probability, we can construct an algorithm B that computes $\Upsilon = e(g,g)^{\epsilon E'}$ as solution for CDH problem of $e(g,g)^{\epsilon}$ and $e(g,g)^{E'}$. Therefore, a valid forged response $(y, \psi, \sigma) \neq (y', \psi', \sigma')$ and $\sigma \neq \sigma'$ cannot be found by A with non-negligible probability.

Case 2: $\sigma = \sigma'$. In this case, we can rewrite Eq.17 as:

$$\left(\frac{e(\psi, g)}{e(\psi', g)}\right)^{\epsilon(\alpha - q)} = e(g,g)^{y'-y} \tag{19}$$

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.19 as:

$$\left(\frac{e(\psi, g)}{e(\psi', g)}\right)^{\epsilon(\alpha - q)} = 1 \tag{20}$$

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.20. 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.19 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 $(-q, \left(\frac{e(\psi,g)}{e(\psi',g)}\right)^{\frac{1}{y'-y}})$.

We denote ψ as g^{θ} and ψ' as $g^{\theta'}$, and rewrite Eq.19 as :

$$\begin{aligned} \left(\frac{e(\psi, g)}{e(\psi', g)}\right)^{\epsilon(\alpha - q)} &= \frac{e(g,g)^{-y}}{e(g,g)^{-y'}} \\ \theta\epsilon(\alpha - q) + y &= \theta'\epsilon(\alpha - q) + y' \\ \frac{\epsilon(\theta - \theta')}{y' - y} &= \frac{1}{\alpha - q} \end{aligned} \tag{21}$$

Therefore, algorithm B can compute

$$\left(\frac{e(\psi, v)}{e(\psi', v)}\right)^{\frac{1}{y'-y}} = e(g,g)^{\frac{\epsilon(\theta-\theta')}{y'-y}} = e(g,g)^{\frac{1}{\alpha-q}} \tag{22}$$

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.

Security of Selection Query, Projection Query, Join Query, Weighted SUM Query, Weighted Exponentiation SUM Query and Variance Query

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, Weighted Exponentiation SUM 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 5.4, 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.

6.2 Performance Evaluation

In this section, we numerically evaluate the performance of our proposed scheme and compare it with the existing schemes[9, 17, 20] 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.

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. Therefore, the total

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

		Ref.[9]	Ref.[17]	Ref.[20]	Our Scheme
Data Pre-Processing	$Comp._C$	$O(sn)Hash$ $+O(1)Sig$	$O(sn)EXP$	$O(n)Hash + O(n)EXP$ $+O(1)Sig$	$O(sn)MUL + O(sn)EXP$ $+O(n)Hash + O(1)Sig$
	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. Oerhead		$O(sn)\|Hash\|$ $+O(1)\|Sig\|$	$O(sn)\|AggSig\|$	$O(n)\|Hash\| + O(n)\|Tag\|$ $+O(1)\|Sig\|$	$O(n)\|G\| + O(n)\|Hash\|$ $+O(1)\|Sig\|$
Update	$Comp._S$	$O(log^n)Hash$	N/A	$O(log^n)Hash$	$O(log^n)Hash$
	$Comp._O$	$O(log^n)Hash$	$O(s)EXP$	$O(log^n)Hash$ $+O(s)EXP$	$O(log^n)Hash$ $O(s)MUL + O(s)EXP$
	Comm.	$O(z)$	$O(z)$	$O(z)$	$O(z)$
Selection Query	$Comp._S$	N/A	$O(k)MUL$	N/A	$O(s+k)MUL + O(s+k)EXP$
	$Comp._C$	$O(sk)Hash$	$O(k)EXP$	$O(k)Hash + O(k)EXP$	$O(k)Hash + O(1)EXP+$ $O(1)MUL + O(1)Pairing$
	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	$O(mn)MUL$	$O(n)MUL$	$O(s+n)MUL + O(s+n)EXP$
	$Comp._C$	$O(mn)Hash$	$O(mn)EXP$	$O(n)Hash + O(n)EXP$	$O(n)Hash + O(1)EXP+$ $O(1)MUL + O(1)Pairing$
	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	$O(n)MUL$	$O(n)MUL$	$O(s+k)MUL + O(s+k)EXP$
	$Comp._C$	$O(nlog^n)Hash$	$O(n)EXP$	$O(n)Hash + O(n)EXP$	$O(n)Hash + O(1)EXP+$ $O(1)MUL + O(1)Pairing$
	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	$O(s+k)MUL + O(s+k)EXP$
	$Comp._C$	N/A	N/A	$O(k)Hash + O(k)EXP$	$O(k)Hash + O(1)EXP+$ $O(1)MUL + O(1)Pairing$
	Comm.	N/A	N/A	$O(log^n)\|Hash\| + O(k)\|Tag\|$	$O(log^n)\|Hash\| + O(1)\|G\|$
Weighted Exponentiation SUM Query	$Comp._S$	N/A	N/A	N/A	$O(s+kx)MUL + O(s+kx)EXP$
	$Comp._C$	N/A	N/A	N/A	$O(n)Hash + O(1)MUL$ $+O(1)EXP + O(1)Pairing$
	Comm.	N/A	N/A	N/A	$O(log^n)\|Hash\| + O(1)\|G\|$
Variance Query	$Comp._S$	N/A	N/A	N/A	$O(s+k)MUL + O(s+k)EXP$
	$Comp._C$	N/A	N/A	N/A	$O(n)Hash + O(1)MUL$ $+O(1)EXP + O(1)Pairing$
	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.[20]. $|AggSig|$ and $|Bitmap|$ is the size of aggregated signature and associated tuple information in Ref.[17]

computational complexity for the pre-processing is $O(sn)MUL + O(sn)EXP + O(n)Hash + O(1)Sig$. For communication cost, all the generated tags as well as Au need to be outsourced to the cloud server, and thus cause a communication complexity as $O(n)|G| + O(n)|Hash| + O(1)|Sig|$, where $|G|, |Hash|$ and $|Sig|$ are the size of a group element, hash value, and signature respectively. With regard to the storage overhead, our proposed scheme requires the cloud to store $O(n)|G| + O(n)|Hash| + O(1)|Sig|$.

Compared with the existing schemes[9, 17, 20] as shown in Table 1, although our proposed require more computational cost and communication cost during the pre-processing, they are one-time cost and will not influence the real-time query performance. For storage cost, Table 1 shows that our proposed scheme achieves comparable complexity to Ref.[20], which outperforms Ref.[9, 17] by removing the influence of number of attributes in each tuple.

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, $O(log^n)Hash$ and one Sig operations are required in our scheme. Since the communication data are the new root state information, new tuples as well as new tags, its complexity is $O(z)$, where z is the number of modified tuples. With regard to the cloud server, it takes $O(log^n)Hash$ and one Sig operations to update the its stored database.

Compared with the existing schemes[9, 17, 20], our scheme introduces more MUL operations to the owner side for the tag update as shown in Table 1. However, the EXP operations needed in our scheme and Ref.[17, 20] are comparable, which is about 10 times more expensive than MUL operation[7]. Thus, our scheme can achieve similar cost compared to Ref.[17, 20]. In addition, our tag construction makes great contributions to the real-time query performance in terms of computational and communication cost. For the computational cost on cloud server side and communication cost, our scheme achieves the comparable complexity to the existing schemes[9, 17, 20].

Selection Query To perform a verified selection query, the client in our scheme 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, 3 $Pairing$ operations, small number of EXP and 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 5.4). On the cloud side, it performs $O(s+k)MUL$ and $O(s+k)EXP$ operations to generate the proof information. For communication complexity, since our proposed scheme aggregates the proof information for integrity check into 3 elements, it causes $O(log^n)|Hash| + O(1)|G|$ complexity.

Compared with the existing schemes[9, 17, 20], Table 1 demonstrates that the computational complexity and communication complexity in our proposed scheme outperforms Ref.[9, 17, 20]. This is because our scheme allows the client to move most computation tasks of verification to the cloud and the proof information of integrity check is aggregated into 3 element. Differently, the tag construction in Ref.[20] requires all the integrity checking operations to be performed by the client itself, which introduces EXP operations linear to k as well as the communication of k tags. The same issue also occurs in Ref.[17], which requires expensive EXP operations to linear to k.

Projection Query To perform a projection query, our proposed scheme requires the client to perform $O(n)Hash$ operations for the verification of result's completeness and freshness. Since the most integrity checking tasks in our scheme 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, it performs $O(s+k)MUL + O(s+k)MUL$ operations to produce all the proof information. For communication complexity, by aggregating the integrity checking proof information into 3 elements, our scheme introduces $O(log^n)|Hash| + O(1)|G|$ complexity.

Compared with the existing schemes[9, 17, 20], our proposed scheme outperforms them in terms of both computational cost and communication cost as shown in Table 1. Specifically, the computational cost of Ref.[9, 17] on client side are linear to m(the number of attributes chosen for query), which is independent to the computational cost in our scheme. Unlike our proposed scheme that moves most computation tasks from client side to cloud side, Ref.[20] remains the operations on client side, which is linear to n(the number of tuples in the

database). For communication cost, Ref.[9] is linear to m and Ref.[20] is linear to n. While, our scheme makes communication cost independent to m and the proof information of integrity checking as constant size.

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

Compared with the existing schemes[17, 20], our proposed scheme has better performance in both computational cost and communication cost. Specifically, Ref.[17, 20] requires expensive EXP operation linear to n. Differently, by moving computation tasks to the cloud, our scheme only requires the client to perform cheap $Hash$ operation as well as small number of EXP, MUL and $Pairing$ operations. Additionally, we reduce the $O(n)$ communication complexity in Ref.[17, 20] to $O(log^n)$. With regard to Ref.[9], we reduce the $O(nlog^n)Hash$ computational complexity to $O(n)Hash + O(1)MUL + O(1)EXP + O(1)Pairing$(since, the value of n is always large in outsourced database, our scheme can achieve lower computation complexity). Moreover, instead of the $O(nlog^n)|Hash|$ communication complexity introduced in Ref.[9], our scheme just requires $O(log^n)|Hash|$.

Weighted SUM Query In our proposed scheme, the only difference between weighted SUM query and selection query is the random numbers chosen in challenge message generation. For weighted SUM query, the client replaces the random numbers with the weight values for calculation. Therefore, as shown in Table 1, the computational complexity and communication complexity on client 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. For cloud server side, its computational complexity is $O(s+k)MUL + O(s+k)EXP$.

Compared with Ref.[20], which introduces EXP operations linear to k, our proposed scheme moves it to the cloud server side to enhance the query performance. In addition, our scheme enables the aggregation of authentication tags into 3 elements, and thus outperforms Ref.[20] in communication complexity, which requires the transmission of all tags.

Weighted Exponentiation SUM Query and Variance Query To perform a weighted exponentiation SUM query, our proposed scheme needs the client side to perform u more MUL and EXP operations than weighted SUM query(e.g., $u = 2$ when $k = 100$ as we discussed in Section 5.4). Therefore, the computation complexity on client side is $O(n)Hash + O(1)MUL + O(1)EXP + O(1)Pairing$.

For communication complexity, our scheme causes $O(log^n)|Hash| + O(1)|G|$ on the client side. As our scheme requires the cloud server to generate the authentication tags for the exponentiation values, its computation complexity 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 weighted exponentiation SUM query, its computational complexity on client side and communication complexity are $O(n)Hash + O(1)MUL + O(1)EXP + O(1)Pairing$ and $O(log^n)|Hash| + O(1)|G|$ respectively. For the cloud server, since the value of x for variance query is 2, it has computational complexity is $O(s + k)MUL + O(s + k)EXP$

7 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 the previous solutions, our scheme achieves better computational, communication and storage performance, especially for the powerful queries(e.g., projection query, aggregate queries). In addition, our proposed scheme supports more powerful aggregate queries that are never achieved in previous works, including weighted exponentiation sum queries of any degrees, variance queries, etc. 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. M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Super-efficient verification of dynamic outsourced databases. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA conference on Topics in cryptology*, CT-RSA'08, pages 407–424, Berlin, Heidelberg, 2008. Springer-Verlag.

6. 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.
7. 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.
8. A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, pages 177–194, 2010.
9. 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.
10. 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.
11. 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.
12. E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *Trans. Storage*, 2(2):107–138, May 2006.
13. 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.
14. 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.
15. 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.
16. 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.
17. H. Pang, J. Zhang, and K. Mouratidis. Scalable verification for outsourced dynamic databases. *Proc. VLDB Endow.*, 2(1):802–813, Aug. 2009.
18. V. Shoup. *A computational introduction to number theory and algebra*. Cambridge University Press, New York, NY, USA, 2005.
19. G. Timothy and M. M. Peter. The nist definition of cloud computing. NIST SP - 800-145, September 2011.
20. 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.