Privacy-Aware Verification of Aggregate Queries on Outsourced Databases with Applications to Historic Data Integrity

Stuart Haber[1], William G. Horne[1], Tomas Sander[1], and Danfeng Yao[2]

[1] Hewlett-Packard Labs
5 Vaughn Drive, Suite 301
Princeton, NJ 08540
{stuart.haber, william.horne, tomas.sander}@hp.com
[2] Department of Computer Science
Rutgers University
Piscataway, NJ 08854-8019
danfeng@cs.rutgers.edu

Abstract. It is often desirable to be able to guarantee the integrity of historical data, ensuring that any subsequent modifications to the data can be detected. It would be especially convenient to extend such proofs of integrity to certain computations performed later using the historic data. We raise this question in the context of outsourced databases, where a data owner delegates the ability to answer users' queries to a service provider, and distrustful users may desire to verify the integrity of responses to their queries on the data. We present a solution for integrity verification of aggregate database queries, such as SUM and MAX, with efficient proofs of correctness and completeness of responses to the queries. What makes the problem challenging is that individual data entries may be sensitive, and should not be revealed to the user. Our protocols are secure, under reasonable cryptographic assumptions.

Keywords: Database query, privacy, outsource, algorithm.

1 Introduction

For many applications, it is desirable to have *historical data integrity*, in which the integrity of some data is established at a specific point in time, and any subsequent modifications to that data can be detected. Of particular interest is the ability to establish the historical integrity of transactions and event logs, which are routinely collected in IT systems for a variety of applications such as intrusion detection, forensics, fraud detection, network monitoring and quality control. Recently, audit logs and IT auditing have become increasingly important as a means of assuring compliance with financial and legal regulations, such as the Sarbanes-Oxley Act (SOX) in the US and similar regulations worldwide.

Consider the following example. A corporation logs financial transactions into a general ledger. At periodic time intervals, a third-party audit is performed to verify that the corporation is following legally acceptable accounting practices. Although there are checks and balances in place, there is always a threat of fraud

if an adversary is able to get access to the system and modify entries in the ledger. These threats are traditionally addressed using carefully managed access control systems and techniques such as segregation of duties. Cryptographic techniques have also been proposed as a solution to this problem [3–5, 28].

Besides the integrity of the data, we still must be concerned about the parties to whom the data can be disclosed. For many applications, it is undesirable to disclose specific data elements, for example due to privacy concerns. However, it may be acceptable to disclose aggregate statistics about the data. This is a common problem that occurs in many areas, including censuses, medical research, and educational testing. For example, the static aggregates of confidential medical records of a group of patients might be accessible by the public; however, the medical record of individual patients should be kept private. Several approaches to this problem have been proposed, including certain methods for perturbing individual data elements (e.g. [2]), but none of them attempt to simultaneously guarantee the integrity of the underlying data, nor do they enable an end user to formally verify the correctness and completeness of the aggregate statistics.

1.1 Our contribution

We formalize the model and definitions for the properties of integrity for privacy-preserving aggregate queries on outsourced databases. We give a general model for querying outsourced data in a three-player setting, in which a data owner delegates to a third-party service provider the task of answering queries from users.

We give protocols for privacy-preserving verification of aggregate queries including SUM, MAX, MIN, COUNT, AVERAGE, and MEDIAN. The protocols allow a user to verify both correctness and completeness of aggregate results while the individual data values contributing to the results are kept secret from the user. The user interacts with the service provider to obtain aggregate results, and can verify whether or not the service provider returns the correct and complete results.

Our solutions for SUM-related aggregate queries are based on a homomorphic commitment scheme, making use of its linearity property. Our solutions for MIN and MAX queries are based on a zero-knowledge proof of knowledge of a greater-than relation of two values. We also use Merkle hash trees for efficient authentication of commitment values.

Our algorithms are efficient. Let n be the number of elements in the data set, and m be the number of elements to be aggregated in response to a specific query. The space complexity for the data owner and the service provider are $O(n)$ (per setup), and $O(m + \log n)$ for the user (per query). The time complexity is $O(n \log n)$ for the data owner (per setup), and $O(m + \log n)$ for both the service provider and the user (per query).

2 Preliminaries

In this section, we define our trust model and provide background on the cryptographic building blocks we use to construct our solution.

2.1 Trust model

We use a three-party model in which a *data owner*, who is the originator or creator of a database, delegates to a semi-trusted *service provider* the ability to answer queries from a *user*. The data owner gives the service provider a copy of the database, along with auxiliary information that enables the verification of query results. The user submits queries to the service provider, and verifies the correctness and completeness of the results returned by the service provider.

In practice, the data owner and the service provider may be the same entity, but this abstraction allows us to clearly separate the tasks that must be performed and where trust must be placed in the system to achieve our end goals.

- *Between data owner and service provider.* The data owner provides the data to a semi-trusted third-party to be able to assert that the data has not be corrupted by insider fraud within the data owner's organization. The data owner must trust the service provider not to disclose the data directly to a user, but rather only to answer well-formed queries from a user.
- *Between service provider and user.* The service provider is not necessarily trusted to answer queries correctly since it may be compromised by outside attacks or insider fraud. Therefore, the user should be able to verify that responses from the service provider are correct and complete.
- *Between data owner and user.* The user must trust the data owner in the sense that the user trusts any messages signed with respect to the data owner's public key. This model is similar to the trust assumptions of the existing literature on outsourced databases [18, 21].

2.2 Cryptographic tools

We describe the building blocks that are used to construct our verification protocols. All of the algorithms discussed in this paper can be stated in terms of any sort of proofs of integrity that begin by hashing their inputs with a one-way hash function, including both digital signatures and time-stamp certificates. Since precise definitions of the security of time-stamping schemes are not yet clear in the cryptographic literature (see [16, 7, 8]), we state all our security results only in terms of digital signatures.

In order to protect the privacy of individual data items while providing verifiable responses to aggregate queries, we make use of the cryptographic commitment scheme due to Pedersen [26], which we review here. We assume that the attribute values to be aggregated are numeric values that can be represented as positive integers. Let $\mathbb{G}$ be any group of large prime order q in which the computation of the discrete log is believed to be hard (and where q is large enough so that our data values can be taken as integers modulo q). Let g and $h \in \mathbb{G}$ be group elements of order q for which $\log_g(h)$ is unknown. A *commitment value* for the data value x is computed by choosing a random value r mod q and computing the group element $C_r(x) = g^x h^r$. A commitment value c can be "opened" or de-committed as a correct commitment to x by revealing r to a verifier (who

checks the equation $c = g^x h^r$). This commitment scheme is computationally binding and unconditionally hiding.

The Pedersen scheme enjoys a convenient homomorphic property: Given two commitments $c_i = C_{r_i}(x_i)$ $(i = 1, 2)$, it is easy to compute a commitment to the sum of the unknown values x_1 and x_2 simply by computing the group element $c_1 c_2 = C_{r_1+r_2}(x_1 + x_2)$.

We will also require a cryptographic hash function H with domain $[0, q-1]$.

A *zero-knowledge proof of knowledge* allows a *prover* to demonstrate to a *verifier* the knowledge of secret values or their relations (such as $\geq$) without revealing them. For example, a proof of knowledge of a Pedersen committed integer x demonstrates the knowledge of a value r such that $C_r(x) = g^x h^r$. We use *non-interactive* proofs of knowledge, where the proof is contained in a single message.

We will be using a specific (i.e., not generic) zero-knowledge proof of knowledge for comparing pairs of data values with Pedersen commitments. The protocol was proposed by Durfee and Franklin [14] and is described in the appendix of our full version [17]. Suppose C_1 and C_2 are commitments of x_1 under random value r_1 and x_2 under random value r_2, respectively. Following the notation of [9], we will write

$$POK(x_1, r_1, x_2, r_2 | C_1 = g^{x_1} h^{r_1}, C_2 = g^{x_2} h^{r_2}, x_1 - x_2 \geq 0)$$

to denote a proof of knowledge that $x_1 - x_2 \geq 0$ holds.

3 Overview of data structures

In order to illustrate the sorts of queries that we handle and how we are going to approach the problem, in this section we give a simple example.

The data owner (and the service provider) use an expanded table T for storing and maintaining data entries. The table not only stores the plaintext data entries, but also stores their sorting indices and commitments of values (discussed in details in Section 2.2). Intuitively, a commitment of a value uniquely binds to the value, but does not reveal the value.

For example, consider a regular database table that has l attributes, such as age, salary, and number of dependents. An expanded table T contains $3l$ columns: $o_1, \ldots, o_l, C(o_1), \ldots, C(o_l), \pi(o_1), \ldots, \pi(o_l)$, where o_i is the plaintext value of attribute i, $C(o_i)$ is the commitment of o_i, and $\pi(o_i)$ is the ordering index of o_i. See Table 1. In more detail:

- Plaintext attribute values $o_1, \ldots, o_l$ are stored in case they are insensitive and can be directly revealed.
- Commitments values $C(o_1), \ldots, C(o_l)$ are used for proving the correctness and completeness of aggregate query results.
- Rankings $\pi(o_1), \ldots, \pi(o_l)$ are used for proving completeness, and are obtained by the data owner sorting the data according to each attribute.

In our protocols, the data owner constructs a *Merkle hash tree* whose leaves consist of entries of the entire table, including plaintext data, their commitments,

Age	*Salary*	*Num*	*C(age)*	*C(sal)*	*C(Num)*	π *(age)*	π *(sal)*	π *(num)*
25	\$65K	0	C(25)	C(65)	C(0)	1	2	1
30	\$50K	2	C(30)	C(50)	C(2)	2	1	3
35	\$70K	1	C(35)	C(70)	C(1)	3	3	2
40	\$80K	3	C(40)	C(80)	C(3)	4	4	4

Table 1. An example of the expanded table maintained by the data owner and the service provider. *Num* represents the number of dependents. $C(i)$ is the commitment of value i. The plaintext data is in columns *Age*, *Salary*, and *Num*, and their rankings are in columns π*(age)*, π*(sal)*, and π*(num)*.

and their rankings. Each row of the table corresponds to a subtree whose leaves are cells of the row. An internal node contains the hash value of its child nodes. The root hash of the tree represents the digest of the entire table, and is signed by the data owner. The purpose of using Merkle hash tree is to efficiently link data elements so that integrity proofs enjoy logarithmic complexities. As it will become clearer soon, using Merkle hash tree a verifier does not need all the data elements (or their commitments) in order to verify a specific element in the set. Only a logarithmic number of elements (or their commitments) is needed.

4 Verification protocols for aggregate queries

We present our verification protocols for sum, max/min, count, and average queries. Our protocols can be generalized to answer combined aggregate queries, aggregate query with selection clause, generalized sum queries such as linear combination, generalized max queries such as median and kth-element. In this section, to clarify our explanations, we use a simple table with one attribute and no plaintext data. These building blocks can be easily expanded to include the general form of data structure as in §3.

For each query-type, we present four operations: **Commit**, **Query**, **Respond**, and **Verify**. We present protocols for correctness verification first, and then give our solution for verification of completeness. The following protocols are run by the data owner, service provider, and the user to answer aggregate queries and verify the correctness of results. In the protocols described in §4.1 and §4.2, we assume that the query is over the complete set of unsorted data $(a_1, \ldots, a_n)$. To handle tables with multiple attributes, we generalize our protocol in §5.1.

General Setup: The data owner chooses a public/private key-pair (PK, SK) for a secure digital signature scheme. The data owner chooses a group $\mathbb{G}$ and elements g and h in $\mathbb{G}$ that specify an instance of the Pedersen commitment scheme; let C denote this specification. The public parameters are given by $param = (PK, C, H, S, POK)$, where H is a hash function, S is the signature scheme, and POK is a zero-knowledge proof of knowledge for the greater-than comparison.

Denote the set A of data by $(a_1, \ldots, a_n)$. Denote the Pedersen commitment of data a_i with random value r_i by C_i (see §2.2). The data owner has an unsorted

set A of data $(a_1, \ldots, a_n)$. The data (in plaintext) is given to the service provider along with auxiliary information including commitments and a signature on the digest of commitments. The service provider answers aggregate queries on behalf of the data owner without revealing the data $(a_1, \ldots, a_n)$. Yet, the user is able to verify the result.

4.1 SUM queries

For sum query, the user obtains the sum s of set A from the service provider, and verifies the correctness of s with respect to the commitments $(C_1, \ldots, C_n)$ of the data along with data owner's signature on the root hash of commitments. The details are as follows.

Commit: The data owner with public/private key pair (PK, SK) and data $(a_1, \ldots, a_n)$ commits and signs the data as follows. Choose n random values $(r_1, \ldots, r_n)$. Compute the Pedersen commitment C_i of a_i with r_i as $C_i = g^{a_i} h^{r_i}$. Construct a Merkle hash tree with commitments $C_1, \ldots, C_n$ as leaf nodes of the tree, and denote the root hash of the tree by h_r. Sign the root hash h_r with the private key SK of the data owner, which gives a signature Sig. Send the following information to the service provider in a secure channel: $\{(a_1, \ldots, a_n), (r_1, \ldots, r_n), (C_1, \ldots, C_n), Sig)\}$. The random value r_is are for the service provider to open commitments of the sum (see operation **Respond**).

Query: User queries for the sum of the data set A.

Respond: The service provider obtains from the data owner the following information: $\{(a_1, \ldots, a_n), (r_1, \ldots, r_n), (C_1, \ldots, C_n), Sig)\}$. It prepares the sum and its proofs as follows. Compute the sum of data $s = \sum_{i=1}^{n} a_i$. Compute the sum of random values $r' = \sum_{i=1}^{n} r_i$. Send the following information to the user: $\{s, r', (C_1, \ldots, C_n), Sig\}$.

Verify: The user receives $\{s, r', (C_1, \ldots, C_n), Sig\}$ from the service provider. The user verifies the correctness of sum s as follows. The user confirms that $g^s h^{r'} = \prod_{i=1}^{n} C_i$, and constructs a Merkle hash tree with $(C_1, \ldots, C_n)$ as leaf nodes. Next, the user computes the root hash h_r, and verifies that Sig is a correct signature for h_r with respect to the public key of the data owner, PK. We assume that the user obtains PK through a regular public key certificate process. Sum s is accepted if all verifications are successful, and rejected otherwise.

The security and efficiency of the protocol are described in §6. Our above protocol can be generalized to a query for any linear combination of sum without revealing the data values themselves. For example, a user can query for the sum of $3a_1 + 5a_2 + 12a_3 + \ldots$, which can be easily computed by the service provider who has a_i values. Our protocol can be easily modified to allow verification of the sum in a privacy-preserving fashion.

The verification protocols of count and average queries can be built based on the sum query. We consider aggregation over the entire set A. For verification of count result n, the user simply counts the number of commitments $(C_1, \ldots, C_n)$, and confirms that it is n. The user constructs the Merkle hash tree of commitments C_is and verifies the signature Sig of the root hash with the public key of the data owner as in sum protocol. The protocol for average query can be built by combining the sum and count verifications, and is not repeated here.

4.2 MAX/MIN queries

For a sorted list, the max/min query can be easily solved as follows. The data owner sorts and signs the root hash of the Merkle tree. The service provider returns the max or min element, and proves that the element is the last or the first element of the sorted list. The user trusts the data owner for sorting the list, and therefore the verification of correctness is equivalent to verifying the position of the result in the list.

We focus on how to verify the correctness of max/min query for unsorted data. We present our correctness verification protocol for max query, which can be easily modified to answer min query. The proofs generated by the service provider for max query are more complex than for sum query. We use the zero-knowledge proof of knowledge for greater-than comparison described in Section 2.2 for the proof of comparison result.

Commit: Same as in sum protocol.

Query: The user queries for the maximum element of set A.

Respond: The service provider computes the maximum element a_j of data set A which contains $(a_1, \ldots, a_n)$. For each data value $a_i \in A$ and $i \neq j$, prepare the zero-knowledge proof of knowledge p_i for $a_j \geq a_i$:

$$p_i = POK(a_i, r_i, a_j, r_j | C_i = g^{a_i} h^{r_i}, C_j = g^{a_j} h^{r_j}, a_j - a_i \geq 0)$$

The service provider gives the following information to the user: $\{a_j, r_j, (C_1, \ldots, C_n), (p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_n), Sig\}$. **Note that** all the data in A except the max is not revealed to the user. **Also note** that values of a_i and a_j are not revealed, as the proofs are for the algebric equations. That is, the verifier only learns about the comparison result, not the values themselves.

Verify: The user obtains the following information from the service provider: $\{a_j, r_j, (C_1, \ldots, C_n), (p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_n), Sig\}$, where p_i is the zero-knowledge proof of knowledge for $a_j \geq a_i$. The user does: Open commitment C_j with a_j and r_j. Construct a Merkle hash tree with $(C_1, \ldots, C_n)$ as leaf nodes. Compute the root hash h_r and verify signature Sig of h_r with the public key PK of the data owner. We assume that the user obtains PK through a regular public key certificate process. The response is accepted if all verifications are successful, and rejected otherwise.

A similar technique can be used to answer a query for the median or the kth element of a set of data items (by changing the comparison proofs appropriately).

4.3 Nested aggregate queries

The above protocols assume that the queries are over the complete set of data. We generalize our solutions to handle tables with multiple attributes in §5.1.

Our correctness protocols can be composed and generalized to verify more complex aggregate queries, namely, nested aggregate queries. Nested aggregate queries are an important and expressive type of query in database systems. For example, a query asks for the max of the counts of numbers of cancer patients

per year. The yearly cancer patient numbers are first counted, and then the maximum is found. Or, for example, a query asks for the max of the sums of revenues per quarters. The quarterly revenues are first summed up, and then the maximum is computed. Our previously presented protocols can be composed with an arbitrary depth to support nested aggregate queries. The integrity verification hides not only individual data entries but also intermediate values in nested aggregate queries.

To give a concrete example, we present the verification protocol for max-sum query. There are m data sets: $A_1, \ldots, A_m$. The user wants the maximum sum of individual sets.

Commit: The data owner commits and signs elements in each set of $A_1, \ldots, A_m$, similar to the sum protocol. The data owner computes commitments of each data value in all sets, and signs the root hash of Merkle tree built over the commitments. Let $\mathcal{C}$ represent all the commitments, $C_{i,k}$ be the commitment of k-th element in set A_i, and Sig be the signature.

Query: The user queries for the maximum number of the sums of individual set $A_1, \ldots, A_m$.

Respond: For each set A_i ($i \in [1, m]$), the service provider computes the sum s_i, and compute the commitment D_i for s_i by multiplying commitments of A_i's data in $\mathcal{C}$: $D_i = \prod_{k=1}^{|A_i|} C_{i,k}$. Note that the service provider also has the random values to open D_is. Compute the maximum number of all sums, which is denoted by s_j ($j \in [1, m]$). For each sum s_i, prepare the zero-knowledge proof of knowledge p_i for $s_j \geq s_i$:

$$p_i = POK(s_i, t_i, s_j, t_j | D_i = g^{s_i} h^{t_i}, D_j = g^{s_j} h^{t_j}, s_j - s_i \geq 0)$$

The above proofs are for the maximum computation. The service provider also needs to show proofs for summation computation using commitments $\mathcal{C}$, and the authenticity of commitments $\mathcal{C}$ is proved with signature Sig. The service provider gives the following information to the user: $\{s_j, \mathcal{C}, (p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_m), Sig\}$. Note that the intermediate sums are not revealed except the max.

Verify: The user obtains the following information from the service provider: $\{s_j, \mathcal{C}, (p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_m), Sig\}$, where p_i is the zero-knowledge proof of knowledge for $s_j \geq s_i$. The user constructs a Merkle hash tree with $\mathcal{C}$ as the leaf nodes. The user then computes the root hash and verifies signature Sig with the public key PK of the data owner. We assume that the user obtains PK through a regular public key certificate process. She then computes the commitment D_i for the intermediate sum, for all $i \in [1, m]$ as $D_i = \prod_{k=1}^{|A_i|} C_{i,k}$. Finally, the user verifies p_i for all $i \in [1, m]$ and $i \neq j$ using commitments D_i. Max s_j is accepted if all verifications are successful, rejected otherwise.

5 Verification of completeness

The definition of the completeness of aggregate queries is directly based on the completeness of selection queries. The basic building block is the existing proof-

of-knowledge protocol for proving greater-than relation of two values. One requirement of our solution is that the attributes used for selection need to be sorted by the data owner. For a relational database table, indices can be built for arbitrary attributes. For a table that has multiple attributes, the attribute used for selection can be different from the attribute for aggregation, for example, average blood pressure for patients older than 55. In this example, we require the table to be sorted under attribute age, but not under attribute blood pressure.

Our description of completeness verification proof requires a generalization of **Commit** operation in the previous section to support multiple attributes.

5.1 Support of multiple attributes

To support flexible aggregate with selection queries, we generalize our **Commit**, **Respond**, and **Verify** operations in the previous section to handle tables with multiple attributes. The main addition to **Commit** operation is that for a data entry with multiple attributes, each attribute value is committed and the hash value of concatenated commitments is used to build a Merkle hash tree. The commitments are also required for verification in **Verify** operation.

Let $(T_1, \ldots, T_l)$ be the attributes of a database table, and l is the number of attributes. Denote the value of attribute T_i by t_i (for $i \in [1, l]$). We assume that all the attributes are sensitive and cannot be revealed to users. If certain attributes are insensitive (and the problem becomes simpler), then the attribute values rather than their commitments are computed in the hash value.

In **Commit**, for each database table entry, the data owner commits to attribute value t_i for all $i \in [1, l]$, by computing $C_i = g^{t_i} h^{r_i}$, where r_i is chosen at random. The data owner computes the hash value of concatenated commitments: $h = H(C_1, \ldots, C_l)$, and constructs Merkle hash tree with all the hash values as leaf nodes. As before, commitments, database tables, random values, and the signature are given to the service provider.

Suppose a user submits an aggregate query for attribute T_1. In response, the service provider prepares correctness proofs for attribute T_1 as in previous protocols. The service provider also gives commitments $C_1, \ldots, C_l$ of all attributes $T_1, \ldots, T_l$ for each entry and proofs to the user. The user reconstructs the hash root and verifies the correctness of query result.

5.2 Completeness verification protocol

We present the completeness verification protocol for aggregate queries with selection. Consider a table with l attributes $T_1, \ldots, T_l$. Our presentation of the protocol uses a range query $[y_1, y_2]$ for an attribute T_j, and aggregation is over attribute T_i. We augment the operations to support proof of completeness.

Without loss of generality, we assume that data entries on the Merkle hash tree are sorted under attribute T_j. That is, the left most entry on the tree has the smallest T_j value, and so on. Our protocol can be modified to allow arbitrary orderings without revealing unnecessary ranking information, which is discussed at the end of this section.

Commit: The data owner sorts data entries from small to large, based on one or more attributes that are used for selection, computes commitments as described in §5.1, constructs Merkle hash tree, and signs the root hash. Let the signature be Sig.

Query: Without loss of generality, let the user's query be an aggregation of attribute T_i ($i \in [1, l]$) over the selection over attribute T_j ($j \in [1, l]$) whose values lie between $[y_1, y_2]$.

Respond: To construct the proof for completeness, the service provider selects the entries that lie in the selection range, which are denoted by A. Then the service provider computes the required aggregate (such as sum, max, etc) of attribute T_i. The completeness proof shows two goals: (1) the unaggregated entries are out of the selection range, and (2) the aggregated entries are within the range. The techniques for (1) and (2) are essentially the same and use zero-knowledge proofs of knowledge. To show (1), we distinguish the following three cases.

- If the selected entries has two immediate neighboring entries, then the service provider constructs a zero-knowledge proof of knowledge that the two entries are beyond the selection range $[y_1, y_2]$. Denote the zero-knowledge proofs of knowledge as p_{lft} and p_{rgt}. The ZK proof p_{lft} shows that the entry immediately to the left [3] of the set of selected entries A has a T_j attribute value v_{lft} smaller than y_1, i.e., $v_{lft} < y_1$. The ZK proof p_{rgt} shows that the entry immediately to the right of the set of selected entries A has a T_j attribute value v_{rgt} larger than y_2, i.e., $v_{rgt} > y_2$.
  Let r_{lft} and r_{rgt} be the random values used by the data owner to compute the commitments of v_{lft} and v_{rgt} in **Commit**, respectively. The proofs p_{lft} and p_{rgt} are expressed below.

$$p_{lft} = POK(v_{lft}, r_{lft}, |C_{lft} = g^{v_{lft}} h^{r_{lft}}, y_1 - v_{lft} > 0)$$
$$p_{rgt} = POK(v_{rgt}, r_{rgt}, |C_{rgt} = g^{v_{rgt}} h^{r_{rgt}}, v_{rgt} - y_2 > 0)$$

  The service provider gives the following information to the user: commitments of selected entries denoted by C_A, commitments of neighbors C_{lft} and C_{rgt}, proofs p_{lft} and p_{rgt}, data owner's signature Sig, and companion hashes. Recall companion hashes are hash values at the roots of disjoint subtrees of the Merkle hash tree all of whose leaves correspond to commitments of unselected data entries (i.e., not in set A).
- If the selected entries has one immediate neighboring entry, then the service provider constructs a zero-knowledge proof of knowledge that the entry is beyond the selection range $[y_1, y_2]$ as above, i.e., either $v_{lft} < y_1$ or $v_{rgt} > y_2$.
- If no element is out of the selection range, i.e., all entries are selected, the Merkle tree construction implicitly proves the completeness. Hence, the service provider returns all the commitments and signature Sig.

[3] Sorting in **Commit** is from small to large.

For (2) (i.e., the aggregated entries are within the selection range), the service provider shows that the smallest aggregated entry is greater than the lower-bound y_1, and similarly, the largest selected entry is less than the upper-bound y_2, if both cases apply. In the selection range is unbounded at one end, then the service provider proves that (i) the lower (or upper) bound is satisfied using zero-knowledge proof of knowledge, and (ii) all the rest of the database entries are (selected and) aggregated, which is implicitly proved in the Merkle tree construction [4]. The details of the proofs are omitted here.

Verify: The user verifies the completeness of results by computing the root hash of Merkle hash tree with commitments C_A, C_{lft}, C_{rgt}, and companion hashes, verifying the signature Sig on the root hash with data owner's public key, and verifying the zero-knowledge proofs of knowledge. The query is accepted if all verifications are successful, rejected otherwise.

The above protocol uses a range as the selection clause. For just $\geq$ or $\leq$ predicates, a simplified version of our protocol suffices, as the proof of only one neighbor is needed. For an equality predicate, the service provider prepares a completeness ZK proof showing that immediate neighbors of selected entries are either larger or smaller than the predicate. The solution presented supports the selection of one attribute. For more complex selections of multiple attributes, for example, age ≥ 30 and height $\geq 6'$, a multi-dimensional range tree [24] has to be constructed by the data owner.

5.3 Generalizing the Completeness Proof

Above we assumed that entries on the hash tree are sorted under attribute T_j, which is also used for the selection. In order to support general completeness proof, the data owner also sorts the data under each attribute, and the indices or rankings are stored as part of the table, as shown, for example, in Table 1 in §3. To further hide the rankings, the rankings can also be committed and then folded into the hash tree, as described above. We demonstrate this using an example as follows.

Consider Table 1, instead of using ranking 3 for entry \$70K for attribute $\pi(sal)$ in the Merkle hash tree, the data owner computes a randomized commitment of 3, denoted by $C_\pi = g^3 h^r$ for a random r. Similarly, for entry \$65K, let $C'_\pi = g^2 h^{r'}$ be a commitment of ranking 2, for a random r'.

Suppose that the selection criteria of a user's query is for salary greater-than \$67K. This selects \$70K entry, but not \$65K entry. To prove that the selection is complete, the service provider shows that **(1)** \$65K $<$ \$67K, **(2)** ranking 3 is higher than ranking 2 by 1, and **(3)** \$65K has ranking 2 and \$70K has ranking 3. Requirement **(3)** is proved implicitly because in Merkle hash tree commitments of \$65K and ranking 2 are grouped and hashed together and similar for \$70K and ranking 3. Requirement **(1)** can be proved with zero-knowledge proofs of knowledge without revealing \$65K. Finally, requirement **(2)** can be proved by showing that C_π / C'_π is a commitment of 1: $C_\pi / C'_\pi = g^{3-2} h^{r-r'} = g h^{r-r'}$. The

[4] One can tell if an entry is missing, because its commitment is needed to obtain the root hash.

	Commit	Respond	Verify	Update	Storage
Data owner	$O(n \log n)$	–	–	$O(k \log n)$	$O(n)$
Service provider	–	$O(m + \log n)$	–	$O(k \log n)$	$O(n)$
User	–	–	$O(m + \log n)$	–	$O(m + \log n)$

Table 2. Time and space complexities of the protocol.

user is given $r - r'$ to open the commitment of 1. Due to space limit, we omit the formal description of this generalized protocol in this version.

6 Security and efficiency

We have analyzed the adversarial model and proved the security of our protocols. Our approach is to give a formal game-based security definition in the random oracle model. Given a set *param* of system parameters chosen by the challenger, an adversary is able to adaptively choose a set of *commit* and *query* requests to the challenger. The adversary's goal is have non-negligible probability of success either in making a guess that breaks the secrecy-preserving properties of our protocol, or in computing a new incorrect query-response pair that passes the **Verify** algorithm. We give a concrete-security proof, reducing the existence of a successful polynomially bounded adversary for our scheme to the existence of an adversary that successfully breaks one or more of the signature scheme, the Pedersen commitment scheme, or the one-way hash function. Our security definitions and proofs are given in the appendix of our full version [17].

Next we analyze the complexities of operations in our verification protocols. We consider the verification of both correctness and completeness. The analysis is independent of the specific type of aggregate query. Let n be the size of all data, m be the size of data selected for query, and k be the number of data elements updated in an **Update** operation. Our verification algorithms have cost linear in the number of data elements selected by a query. This is to be expected, given our approach, since in essence our procedures verify each of these data elements' contribution to the correct response. A summary is given in Table 2.

7 Related work

Several cryptographic techniques have been proposed to protect the integrity of data even if the adversary is an insider, including *forward integrity* [4, 5, 28] and *time-stamping* [3]. However, these techniques do not directly address the problem of establishing the integrity of aggregate statistics computed over that data.

A substantial amount of research work has been done on how to verify outsourced data and computation [13, 19, 22, 23], including the verification of both correctness and completeness of relational database queries. The existing literature on database query verification has focused on non-aggregate queries such as SELECT, PROJECT, JOIN, SET UNION and INTERSECTION. Merkle hash trees have been used extensively for authentication of data elements [20]. Aggre-

gate signatures are another approach for data authentication, where each data tuple is signed by the data owner [23]. Most recently, the privacy issue in verifying non-aggregate queries was first addressed by [25], which gave an elegant solution using hashing for proving the completeness of selection queries without revealing neighboring entries. We provide an alternative solution for the privacy issue in completeness proof by utilizing zero-knowledge proofs of knowledge and a commitment scheme.

The aggregate query verification problem has been studied in database-as-a-service (DAS) model [19, 22]. This model is an instantiation of the computing model involving trusted clients, who store their data at an untrusted server that are administrated by the service provider. The challenge is to make it impossible for the system provider to correctly interpret the data. The data is owned by clients. The clients only have limited computational power and storage, and they rely on the server for the mass computational power and storage. The server exposes mechanisms for the clients to create and manage the client databases at the server. Data originates from the client. The recent paper by Hacigümüs, Iyer, and Mehrotra [19] addresses the execution of aggregate queries over encrypted data using a homomorphic encryption scheme. Mykletun and Tsudik [22] proposed an alternative approach where the data owner pre-computes and encrypts the aggregate results and stores them in the service provider. This approach avoids the use of homomorphic encryption, which was found to have a security flaw when used for DAS [22]. The correctness and completeness definitions do not apply to these models as the user is also the data owner in DAS. Our model is different from DAS, and is suitable for a more general security setting, as the data does not have to be originated from the client. We compare major features of our work with existing solutions in Table 3.

Searchable symmetric-key encryption schemes for private-key storage outsourcing have been previously studied (e.g., [1, 29]). Most recently, improved security definitions and constructions are proposed by Curtmola, Garay, Kamara, and Ostrovsky [11]. Public-key systems have also been used to construct searchable encryption schemes [1, 31], including a practical searchable and encrypted audit log system [31]. In general, symmetric key encryption is more efficient than public-key encryption. In the meantime, the symmetric key encryption typically requires live key updates, which incur communication costs. Our authentication protocol differs from the above work in that it focuses on the validation of query results, and supports data aggregate besides search (i.e., equality and comparison-based selection).

Damiani *et al* are the first to address the access control issue in encrypted outsourced data [12]. They proposed to use selective data encryption and tree-based key management to delegate to the service provider the access control enforcement. Their goal is to reduce the workload of data access management by the data owner in a dynamic scenario. Their security model is different from ours in that their clients have different access privileges to data and are allowed to view selective plaintext data.

	Ours	NT [23]	DGMS [13]	PJRT [25]	HIM [19]/MT [22]
Aggregate Q.	Yes	No	No	No	Yes
Correctness	Yes	N/A	N/A	N/A	N/A
Completeness	Yes	Yes	Yes	Yes	N/A
Authenticity	Yes	Yes	Yes	Yes	Yes
Privacy	Yes	No	No	Yes	Yes
Data Structure	Tree-based	Signature chain	Tree-based	Tree-based	N/A

Table 3. Comparisons of functionalities of our verification protocols with some of the existing approaches developed for outsourced systems.

In data mining literature [2, 27, 30], an important approach to protect data privacy is to modify database tables such that an individual entry enjoys certain degree of anonymity. Our solutions differ from existing efforts in that we support authenticated ad hoc data analysis without releasing the microdata to the public. Because the aggregate is computed over exact data instead of generalized data, there is no loss of data accuracy in the aggregate results.

8 Future work

One interesting future direction is to develop efficient data update algorithms. All of our discussion has been directed towards queries on a fixed database. A dynamically changing database can be handled with periodic time-stamped snapshots of the database, where each signature (of the root of the Merkle hash tree) is accompanied by a third-party time-stamp certificate. In this scenario, queries would include a time, and the **Verify** algorithm is changed to require verification of the time-stamp certificate with respect to the appropriate snapshot time. Note that changes to any data item require not only changes along the path from the corresponding Merkle-tree leaf to the root of the tree, but also sorting the elements and updating indices associated with the attribute. To avoid or to reduce the sorting overhead (e.g., sorting a subset of elements instead of all of them), it is conceivable that advanced data structures such as multi-dimensional range-trees may be used. More investigations on these topics need to be carried out.

Another challenging direction is to develop unlinkable and verifiable data aggregation. The linkage problem occurs when a prover (database holder) answers several *different* queries from the verifier and returns the *same* set of commitments. Then, there is a possible leakage of information. For example, if one query asks how many people live in Springfield and another query asks how many are over forty years old, then by viewing the returned commitments the verifier could determine how many people over forty there are in Springfield, i.e. an information leakage occurred. One way to prevent this is to change commitments over time; for example, a prover could randomize commitments. This procedure should not require interaction from the prover with the proof preparer. In the meantime, a user should still be able to verify the randomized commitments are generated from authentic data.

References

1. M. Abdalla, M. Bellare, D. Catalano, E. Kiltz, T. Kohno, T. Lange, J. M. Lee, G. Neven, P. Paillier, and H. Shi. Searchable encryption revisited: Consistency properties, relation to anonymous IBE, and extensions. In *In CRYPTO 2005*, volume 3621 of *LNCS*, pages 205–222. Springer, 2005.
2. Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 2000.
3. D. Bayer, S. Haber, and W.S. Stornetta. Improving the efficiency and reliability of digital time-stamping. In R.M. Capocelli, A. De Santis, and U. Vaccaro, editors, *Sequences II: Methods in Communication, Security, and Computer Science*, pages 329–334. Springer-Verlag, 1993. (Proceedings of the *Sequences* Workshop, Positano, Italy, 1991.).
4. M. Bellare and B. Yee. Forward integrity for secure audit logs. Technical report, University of California, San Diego, November 1997.
5. M. Bellare and B. Yee. Forward security in private-key cryptography. In *CT-RSA*, volume 2612 of *LNCS*, pages 1–18. Springer-Verlag, 2003.
6. F. Boudot. Efficient proofs that a committed number lies in an interval. In *Advances in Cryptology - EuroCrypt '00*, volume 1807 of *Lecture Notes in Computer Science*, pages 431 – 444. Springer-Verlag, 2000.
7. A. Buldas and M. Saarepera. On provably secure time-stamping schemes. In *Advances in Cryptology — ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 500–514, October 2004.
8. A. Buldas and M. Saarepera. Do broken hash functions affect the security of time-stamping schemes? In *Internatioanl Conference on Applied Cryptography and Network Security (ACNS '06)*, volume 3989 of *Lecture Notes in Computer Science*, pages 50 – 65, 2006.
9. Jan Camenisch and Markus Michels. Proving in zero-knowledge that a number is the product of two safe primes. In *Advances in Cryptology - EUROCRYPT '99*, volume 1592 of *LNCS*, pages 107–122. Springer Verlag, 1999.
10. R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Advances in Cryptology - CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 174 – 187. Springer-Verlag, 1994.
11. Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 13st ACM Conference on Computer and Communications Security (CCS)*, 2006.
12. Ernesto Damiani, Sabrina De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Selective data encryption in outsourced dynamic environments. *Electr. Notes Theor. Comput. Sci.*, 168:127–142, 2007.
13. P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic third-party data publication. *Journal of Computer Security*, 11(3), 2003.
14. Glenn Durfee and Matt Franklin. Distribution chain security. In *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS)*, pages 63–70, New York, NY, USA, 2000. ACM Press.
15. S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptively chosen message attacks. *SIAM Journal on Computing*, 7(2):281–308, 1988.

16. S. Haber and W.S. Stornetta. Secure names for bit-strings. In *Proceedings of the 4th ACM Conference on Computer and Communication Security*, pages 28–35. ACM Press, April 1997.
17. Stuart Haber, William G. Horne, Tomas Sander, and Danfeng Yao. Privacy-aware verification of aggregate queries on outsourced databases with applications to historic data integrity, February 2008. `http://www.cs.rutgers.edu/~danfeng/papers/agg-ver-full.pdf`.
18. H. Hacigümüs, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service provider model. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 216 – 227. ACM Press, June 2002.
19. H. Hacigümüs, B. Iyer, and S. Mehrotra. Efficient execution of aggregation queries over encrypted databases. In *Proceedings of International Conference on Database Systems for Advanced Applications (DASFAA)*, 2004.
20. R. Merkle. Protocols for public key cryptosystems. In *Proceedings of the 1980 Symposium on Security and Privacy*, pages 122–133. IEEE Computer Society Press, 1980.
21. E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *Proceedings of Symposium on Network and Distributed Systems Security (NDSS)*, February 2004.
22. E. Mykletun and G. Tsudik. Aggregation queries in the database-as-a-service model. In *IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec)*, July 2006.
23. M. Narasimha and G. Tsudik. Authentication of outsourced databases using signature aggregation and chaining. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, April 2006.
24. Glen Nuckolls, Charles U. Martel, and Stuart G. Stubblebine. Certifying data from multiple sources. In *Data and Applications Security XVII: Status and Prospects, IFIP TC-11 WG 11.3 Seventeenth Annual Working Conference on Data and Application Security*, pages 47–60, 2003.
25. HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 407–418, 2005.
26. T. P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract). In *Advances in Cryptology - EuroCrypt '91*, volume 547 of *Lecture Notes in Computer Science*, pages 522 – 526. Springer-Verlag, 1991.
27. Periangela Samarati. Protecting respondent's privacy in microdata release. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):1010 – 1027, 2001.
28. B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the 7th USENIX Security Symposium*, pages 53–62. USENIX Press, 1998.
29. D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *In Proceedings of 2000 IEEE Symposium on Security and Privacy*, pages 44 – 55, May 2000.
30. Latanya Sweeney. k-Anonymity, a model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5):557 – 570, 2002.
31. Brent R. Waters, Dirk Balfanz, Glenn Durfee, and Diana K. Smetters. Building an encrypted and searchable audit log. In *Proceedings of Symposium on Network and Distributed Systems Security (NDSS '04)*, 2004.

A Security definitions and proof

In this section we give our game-based definition of security, and then prove that our protocols satisfy this definition. For simplicity, the game definition is for a single attribute table. It can be generalized to multiple attribute queries and is omitted.

Setup: The challenger takes a security parameter k, and generates public parameters *param*, which is given to the adversary. The challenger keeps the private key SK to itself.

Phase 1: The adversary issues queries $q_1, \ldots, q_m$, where q_i is one of the followings:

1. Commit query (A): The challenger computes commitments of data elements in set A. The commitments and random values used are given to the adversary.
2. Sign query (h_r): The challenger signs the root hash h_r with its private key.
3. Aggregate query (A, Q): The challenger runs the corresponding **Respond** algorithm to answer query Q of A. The resulting answer *ans*, correctness and completeness proofs *pf*, and the signature Sig of A are sent to the adversary.

These queries may be asked adaptively. Also, the queried set at each query may be distinct. Once the adversary decides that **Phase 1** is over, she chooses a challenge for attacking privacy. (No need of choosing challenge for attacking correctness and completeness.)

Privacy challenge: The adversary outputs two distinct equal size sets A_0 and A_1 and an aggregate query Q^* to be challenged, such that query Q^* on set A_0 and A_1 gives the same result – the adversary cannot tell them apart by just seeing the query result. The challenger picks a random bit $b \in \{0, 1\}$, computes the query results and proofs on set A_b by running **Respond** algorithm, which outputs (ans^*, pf^*, Sig^*). It sends (ans^*, pf^*, Sig^*) as a challenge to the adversary. The adversary needs to guess whether A_0 or A_1 is used to produce the aggregate result ans^*.

Correctness challenge: The adversary outputs a data set $\tilde{A} = (\tilde{a}_1, \ldots, \tilde{a}_n)$, commitments $\{\tilde{C}\} = (\tilde{C}_1, \ldots, \tilde{C}_n)$ of data elements, and random values $\tilde{r}_1, \ldots, \tilde{r}_n$ used in computing the commitments. The challenger opens the commitments by re-computing them with $\tilde{a}_i$ and $\tilde{r}_i$. If all the commitments are verified successfully, the challenger constructs the Merkle hash tree and signs the root hash. The signature $\tilde{Sig}$ is given to the adversary.

Phase 2: The adversary issues more queries. The challenger responds as in **Phase 1**.

Guess: The adversary outputs one or more of three guesses for attacking correctness, completeness, and privacy, respectively.

Privacy guess: The adversary outputs a guess $b' \in \{0, 1\}$. The adversary wins the game if $b = b'$. We define its advantage in attacking the scheme to be $|\Pr[b = b'] - \frac{1}{2}|$.

Correctness guess: The adversary outputs $(\tilde{Q}, \tilde{A}, \tilde{ans}, \tilde{pf}, \tilde{Sig^*})$, such that $\tilde{ans}$ is *not* the correct result of query $\tilde{Q}$ over data set $\tilde{A}$, however $\tilde{pf}$ is an ac-

ceptable proof of correctness of result $\tilde{ans}$, and $\tilde{Sig}^*$ is an acceptable signature. We allow $\tilde{Sig}^*$ to be different from what is given in **Correctness challenge**. However, the additional constraint is that $\tilde{Sig}^*$ is a signature of a message (root hash) that has not been signed in **Phase 1** or **Phase 2**. In other words, the adversary can demonstrate that a wrong answer can pass the correctness verification. Note that the adversary needs to output the individual data values of $\tilde{A}$ in her attack.

Completeness guess: The adversary outputs $(\tilde{Q}, \tilde{A}, \tilde{ans}, \tilde{pf}, \tilde{Sig})$, such that $\tilde{ans}$ is *not* the complete result of query $\tilde{Q}$ over the data set $\tilde{A}$, however $\tilde{pf}$ is an acceptable proof of completeness and $\tilde{Sig}$ is an acceptable signature of commitments of the data. In other words, the adversary can demonstrate that an incomplete answer can pass the completeness verification.

Theorem 1. *Our verification protocols of aggregate query results on outsourced databases preserve the correctness, completeness, and privacy properties, assuming the existence of a collision-free hash function, a secure commitment scheme with hiding and binding properties, and a signature scheme secure against existential forgery.*

Proof: For simplicity, our proof is for a single attribute table. The proof generalizes to multiple attribute queries naturally, which is omitted. Let Adv_a be the adversary that has advantage against our correctness or completeness verification protocol. Let us construct an adversary Adv_b that uses Adv_a to gain advantage against collision-free hash function, secure commitment scheme, secure signature scheme, or zero-knowledge proof of knowledge for the greater-than comparison. The adversary Adv_b acts as the challenger for Adv_a and uses Adv_a's outputs as her own outputs. Adv_b answers Adv_a's queries as follows.

Setup: Adv_b's challenger chooses hash function H, commitment scheme C, signature scheme S, and the zero-knowledge proof of knowledge P for the greater-than comparison. Adv_b's challenger gives Adv_b a public key PK of the signature scheme S. Adv_b then gives the adversary Adv_a the resulting public parameters $param = (PK, H, C, S, P)$. Note that Adv_b does not know the private key SK of signature scheme S.

Phase 1: For query q_i, Adv_b answers Adv_a's queries as follows. The queries may be asked adaptively. Also, the queried document at each query may be distinct.

1. Commit query (A): Adv_b runs the first several operations in **Commit** algorithm on A, including computing commitments, building hash tree over commitments, and gathering auxiliary information $Info$. The commitments and $Info$ are given to the user.
2. Sign query (h_r): Adv_b does not know how to sign the root hash, because he does not have the private key. Therefore, Adv_b submits a signing query on the root hash to his challenger (of the signature scheme to break), and obtains a signature Sig (the game definition for signature scheme is not given, please see [15]). Signature Sig is given to the user.

3. Aggregate query (A, Q): Adv_b runs a commit query and a sign query on A to obtain the signature Sig, commitments, and auxiliary information $Info$. Then Adv_b runs the corresponding **Respond** algorithm to compute query Q on A. The resulting answer *ans*, correctness and completeness proofs *pf*, and the signature Sig of data set A are sent to Adv_a.

Once Adv_a decides that **Phase 1** is over, she chooses a challenge for attacking privacy.

Privacy challenge: Adv_a outputs two distinct equal-size (n) data sets A_0 and A_1 and an aggregate query Q^* to be challenged, such that query Q^* on set A_0 and A_1 gives the same result (Adv_a should not be able to tell them apart by just seeing the query result or the size.) Adv_b chooses a random $i \in [1, n]$, such that the i^*-th elements of A_0 and A_1 are distinct. Denote the two elements by $a^0_{i^*}$ and $a^1_{i^*}$, respectively.

Adv_b needs to use Adv_a's advantage against the confidentiality to break the hiding property of commitment scheme. Adv_b needs to embed his commitment challenge in the challenge of Adv_a. Values $a^0_{i^*}$ and $a^1_{i^*}$ are Adv_b's two messages of choice for breaking the hiding property of commitment C. Adv_b's challenger generates a challenge for Adv_b as follows. Adv_b's challenger picks a random bit $b \in \{0, 1\}$, and computes a commitment of $a^b_{i^*}$. Denote this challenge as C^b_*.

Although Adv_b does not know b, Adv_b needs to compute commitments of elements in A_b ($b \in \{0, 1\}$) such that the correctness verification can pass. Adv_b first computes the aggregate result ans^* of query Q^*, then Adv_b chooses a random guess $b'' \in \{0, 1\}$. Note that, $b'' = b$ with probability $1/2$. For each j-th element in $A_{b''}$ for all $j \neq i^*$ and $j \neq 1$, Adv_b computes a commitment C_j.

Adv_b distinguishes two cases.

(1) For sum-related query Q^*, Adv_b chooses random r and computes a commitment for the sum: $C_s = g^{ans^*} h^r$. (The sum is the same for A_0 and A_1 as defined.) Adv_b then computes the commitment C_1 for the first element as $C_1 = C_s / (C^b_* \times \prod_{j=2, j\neq i^*}^{n} C_j)$. Now, Adv_b has embedded his commitment challenge at the i^*-th position of A's challenge. *ans* is the aggregate result. Random value r and commitments C_j ($j \neq i^*, j \in [1, n]$) and C^b_* are correctness proof pf^* for summation. Adv_b also obtains a signature Sig^* as in sign query. (ans^*, pf^*, Sig^*) is given to Adv_a. Readers can verify that the correctness verification of (ans^*, pf^*, Sig^*) should be successful even though Adv_b does not know b.

(2) For max/min type of query Q^*, for the i^*-th position, Adv_b does not know $a^b_{i^*}$, therefore, he has to simulate the greater-than proof (i.e., transcript). The simulation can be done, because of the zero-knowledge property of greater-than protocol, which guarantees that the verifier sees during the protocol (i.e., the transcript) can be simulated by anyone without knowing the secret. Denote the simulated proof by p_{i^*}. One goal in simulating the proof is to prepare commitments $\alpha_0, \ldots, \alpha_{t-1}$, each corresponding to a random commitment of 0 or 1, such that $\prod_{i=0}^{t-1} \alpha_i^{2^i}$ is a commitment of $|a^b_{i^*} - ans^*|$, as in the zero-knowledge proof of knowledge for greater-than. W.l.o.g., we assume that $ans^* > a^b_{i^*}$ and the query is max. Denote the commitments of $a^b_{i^*}$ and query result ans^* by C^b_*

and C_{ans^*}, respectively. Note that Adv_b does not know $a^b_{i^*}$, thus does not know the bit presentation of $ans^* - a^b_{i^*}$. Adv_b first chooses random commitments for $\alpha_1, \ldots, \alpha_{t-1}$, and then computes $\alpha_0 = C_{ans^*}/(C^b_* \prod_{i=1}^{t-1} \alpha_i^{2^i})$.

In addition, Adv_b has to simulate zero knowledge proofs to show that values committed by $\alpha_0, \ldots, \alpha_{t-1}$ are either 0 or 1. This can be done without Adv_b knowing the real values committed, because of the zero-knowledge property of the OR proof (see the previous section for the definition of zero-knowledge property). Therefore, Adv_b can simulate proof p_{i^*}. We omit further details of proof simulation and refer readers to literature for details [6, 10, 14]. The correctness proof pf^* contains p_j ($j \in [1, n], j \neq i^*$) and p_{i^*}. Adv_b also obtains a signature Sig^* as in sign query. (ans, pf^*, Sig^*) is given to Adv_a.

In both the above cases, Adv_b can also generate completeness proof in pf^* (if it applies). Because completeness proof would be the same for A_0 or A_1, it cannot be used to gain advantage for privacy attack, hence is omitted here.

Correctness challenge: Adversary Adv_a outputs a data set $\tilde{A} = (\tilde{a}_1, \ldots, \tilde{a}_n)$, commitments $\{\tilde{C}\} = (\tilde{C}_1, \ldots, \tilde{C}_n)$ of data elements, and random values $\tilde{r}_1, \ldots, \tilde{r}_n$ used in computing the commitments. Adversary Adv_b opens the commitments by re-computing them with $\tilde{a}_i$ and $\tilde{r}_i$. If all the commitments are verified successfully, Adv_b constructs the Merkle hash tree. Then, Adv_b asks its challenger to sign the root hash. The resulting signature $\tilde{Sig}$ is given to Adv_a.

Phase 2: The adversary issues more commit queries, sign queries, and aggregate queries. The challenger responds as in **Phase 1**.

Guess: Adversary Adv_a outputs one or more of three guesses for attacking correctness, completeness, and privacy, respectively.

Privacy guess: Adversary Adv_a outputs a guess $b' \in \{0, 1\}$. Adv_b outputs b' as his guess for breaking the hiding property of the commitment scheme. If Adv_a has advantage ϵ_1 in breaking the confidentiality property, then Adv_b has advantage at least $\epsilon_1/2$ in breaking the commitment scheme. Recall that Adv_b does not know b and thus when computing commitments in the proofs for Adv_a, it guesses randomly whether to use elements from A_0 or A_1. For half of the time, Adv_a is given the right combination of committed values. Thus, Adv_b carries over advantage $\epsilon_1/2$ in breaking the hiding property of the commitment scheme.

Adversary Adv_a may also try to gain advantage from proof information other than commitments, for example, from zero-knowledge proofs of knowledge for a great-than relation. Reduction can be directly constructed from advantages in such attacks to breaking the zero-knowledge property of the greater-than proof protocol, and is omitted here.

Correctness guess: Adversary Adv_a outputs $(\tilde{Q}, \tilde{A}, \tilde{ans}, \tilde{pf}, \tilde{Sig^*})$, such that $\tilde{ans}$ is *not* the correct result of query $\tilde{Q}$ over data set $\tilde{A}$, however $\tilde{pf}$ is an acceptable proof of correctness, and $\tilde{Sig^*}$ is an acceptable signature of commitments of data. Adv_b's goal is to try to convert Adv_a's output into either breaking the binding property of commitment scheme or an existential signature forgery. We distinguish two cases.

- $\tilde{Sig^*} \neq \tilde{Sig}$: $\tilde{Sig^*}$ is not the same as given in **Correctness challenge**. As defined, the constraint is that $\tilde{Sig^*}$ is a signature of a message (root

hash) that has not been signed in **Phase 1** or **Phase 2**. This means that Adv_b obtains a signature that breaks the existential unforgeability property. Adv_b outputs $\tilde{Sig}^*$ and the corresponding message (which is the root hash of Merkle tree and can be easily obtained from the proof $\tilde{pf}$). If Adv_a has advantage ϵ_2 in this attack, then Adv_b has advantage ϵ_2 in breaking the existential unforgeability of the signature scheme.

- $\tilde{Sig}^* = \tilde{Sig}$: Denote the commitments in $\tilde{pf}$ by $\{\tilde{C}^*\}$. We distinguish the following three cases.
  (1) If the commitments $\{\tilde{C}^*\}$ are the same as the commitments $\{\tilde{C}\}$ (computed in **Correctness challenge**) and the query $\tilde{Q}$ is summation-based (e.g., sum, count, etc), then Adv_b can break the binding property of commitment scheme as follows. Adv_b computes the *correct* answer *ans* of set $\tilde{A}$ for query $\tilde{Q}$, and computes the commitment C_{ans} of *ans* based on the commitments $\tilde{C}_1, \ldots, \tilde{C}_n$: $C_{ans} = \prod_{i=1}^{n} \tilde{C}_i$. C_{ans} is also the commitment of the *incorrect* result $\tilde{ans}$, because $\prod_{i=1}^{n} \tilde{C}_i = \prod_{i=1}^{n} \tilde{C^*}_i$. Adv_b outputs C_{ans} as the commitments for both *ans* and $\tilde{ans}$ to its challenger. If Adv_a has advantage ϵ_3 in this attack, then Adv_b has advantage ϵ_3 in breaking the binding property of the commitment scheme. (Adv_b also knows how to open commitment C_{ans}.)
  (2) If the commitments $\{\tilde{C}^*\}$ are the same as the commitments $\{\tilde{C}\}$ (computed in **Correctness challenge**) and the query $\tilde{Q}$ is comparison-based (e.g., min, max), then Adv_a can cheat on the greater-than protocol in the correctness proofs of max/min query. This means that Adv_b can break the soundness of zero-knowledge proof of knowledge for the greater-than comparison. The analysis is similar to our completeness analysis (below), and is omitted here.
  (3) If the commitments $\{\tilde{C}^*\}$ are different from the commitments $\{\tilde{C}\}$, then Adv_a has find a hash collision. That is, Adv_a has find at least a different message pair (commitments) giving the same hash value (and thus same signature). If Adv_a has advantage ϵ_4 in finding such a message-signature pair, then Adv_b has advantage ϵ_4 in breaking the collision-free hash function.

Completeness guess: Adversary Adv_a outputs $(\tilde{Q}, \tilde{ans}, \tilde{A}, \tilde{pf}, \tilde{Sig})$, such that $\tilde{ans}$ is *not* the complete result of query $\tilde{Q}$ over a set of data, however $\tilde{pf}$ is an acceptable proof of completeness and $\tilde{Sig}$ is an acceptable signature of commitments of data. Let the selection range be $[x, y]$. This means that Adv_a cheats on the zero-knowledge greater-than proof in either one or both cases: (1) proving in zero-knowledge that $a_{lft} < x$, however $a_{lft} \geq x$; (2) proving in zero-knowledge that $a_{rgt} > y$, however $a_{rgt} \leq y$. If Adv_a achieves this, Adv_b can break the soundness of zero-knowledge proof of knowledge for the greater-than relation. Recall that soundness means that no one who does not know the secret can convince the verifier with non-negligible probability. In this proof protocol, it means that no one who does not know a secret satisfying the greater-than relation can convince the verifier with non-negligible probability. If Adv_a has advantage ϵ_5 in cheating the completeness proof, then Adv_b has advantage ϵ_5 in breaking the soundness of zero-knowledge proof of knowledge for the greater-than relation. □

B Durfee and Franklin's Zero-Knowledge Proofs of Knowledge Protocol

Our protocols need proofs that two committed integers, x_1 and x_2, satisfy an inequality such as $x_1 \geq x_2$ [6, 10, 14]. One approach is to show that $x_1 - x_2 \geq 0$. We review the greater-than proof by Durfee and Franklin that is based on the bit commitments of the difference $x_1 - x_2$, following their description [14]: $POK(x_1, r_1, x_2, r_2 | C_1 = g^{x_1} h^{r_1}, C_2 = g^{x_2} h^{r_2}, x_1 - x_2 \geq 0)$.

The prover can compute the commitment $C' = C_{r_1 - r_2}(x_1 - x_2)$, and the verifier can compute this as $C' = C_1/C_2$. Let $(\gamma_i)_{i=0}^{t-1}$ be the binary representation of $x_1 - x_2$, i.e., $x_1 - x_2 = \sum_{i=0}^{t-1} 2^i \gamma_i$, where t is the bit length of the difference. Choose random values $s_1, \ldots, s_{t-1}$ and set $s_0 = r_1 - r_2 - \sum_{i=1}^{t-1} 2^i s_i$. Let $\alpha_i = C_{\gamma_i}(s_i) = g^{\gamma_i} h^{s_i}$ for all $i \in [0, t-1]$. Suppose the verifier knows that the bound $x_1, x_2 \in [0, 2^t)$ holds. The prover provides the commitments $\alpha_0, \ldots, \alpha_{t-1}$ along with a proof that each γ_i is a bit (either 0 or 1). The verifier checks the proof that each bit committed by α_i is either 0 or 1 and confirms that equation $C_2 \prod \alpha_i^{2^i} = C_1$ holds. Suppose the verifier does not know the bound of x_1 or x_2. Then, a zero-knowledge proof of knowledge that $x_1 \in [0, 2^t)$ and $x_2 \in [0, 2^t)$ can be constructed in a similar fashion. In our protocol, we assume the bounds of x_1 and x_2 are known by the verifier (or the user), which is usually the case for most database entries such as zip code, salary, age, etc.

A special case of greater-than proof called interval proof. An interval proof proves that a committed integer satisfies an inequality such as $x \geq A$ or $y \leq B$, where A and B are constants: $POK(x_1, r_1 | C_1 = g^{x_1} h^{r_1}, x_1 - A \geq 0)$, $POK(x_2, r_2 | C_2 = g^{x_2} h^{r_2}, B - x_2 \geq 0)$.