Group-by Query Verification by Untrusted Clients on Outsourced Data Streams

Suman Nath, Ramarathnam Venkatesan

Microsoft Research
Redmond, WA, USA
{sumann, venkie}@microsoft.com

***Abstract*— Outsourcing data streams and desired computations to a third party such as the cloud is practical for many companies due to overwhelming flow of information and excessively high resource requirements of their data stream applications. However, data outsourcing and remote computations intrinsically raise issues of trust, making it crucial to verify results returned by third parties. In this context, we propose a novel solution to verify outsourced `Group-By, Sum` (or `histogram`) queries that are common in many business applications. We consider a setting where a data owner employs an untrusted remote server to run continuous `Group-By, Sum` queries on a data stream it forwards to the server. Untrusted clients then query the server for the computed results. More importantly, a client can efficiently verify the correctness of the results by using a small and easy-to-compute signature provided by the data owner. Our work complements previous works on authenticating remote computation of selection and aggregation queries. Moreover, unlike prior work on remote Group-by queries, we support untrusted clients (who can collude with other clients or with the server) and provide stronger cryptographic guarantees. Experimental results on real and synthetic data show that our solution is practical and efficient.**

I. Introduction

Data Stream Management Systems (DSMS) have become increasingly important in many real world applications that need to process massive amounts of streaming data. Many companies rely on DSMSs to query important statistics about their day-to-day operations. However, acquiring and managing a DSMS system capable of providing fast and reliable querying service on high-throughput streaming data is expensive. The cost further exacerbates when the target queries are resource incentive and hence fault tolerance techniques such as replication become very expensive [1]. Therefore, not surprisingly, outsourcing data stream and the desired computations to a third party server becomes a practical alternative to many companies. Such outsourcing of data and computation has received a lot of attention in recent years, partly due to the increasing availability of cloud computing. Outsourcing makes a DSMS service, especially computation of expensive functions on high volume streams, more affordable for parties with limited resources. Since cloud providers have the advantage of expertise consolidation, they can potentially offer the service with much lower cost and also with better scalability, performance, and availability guarantees.

As an example, consider an online marketplace where sellers sell their items and buyers browse, buy, and leave feedbacks on various items sold by sellers. To provide sellers hints on what items buyers prefer to buy, the marketplace collects users' ratings on various products and let sellers query such ratings. For example, the marketplace can provide an average rating for various groups defined by $<$product id, user demographic$>$ pair. The information can be important to the sellers to identify popular products within various user demographics. However, memory footprint of such a query increases linearly with the number of groups, which in this case can be excessively high due to a large number of product id and user demographic combinations. In one real click log from Microsoft Bing search engine, we see $\approx 10^{12}$ different groups, requiring more than 3TB main memory footprint,[1] even without any replication. The infrastructure required for this can be too expensive for many companies. Similar scenarios are common in sensor networks or in IP-networks, where a base station or a network operator may want to maintain statistics for a large number of groups defined by $<$sensor event, time$>$ or $<$IP address, port$>$ combinations. In these examples, the online marketplace, a base station, or a network operator can offload the task of maintaining statistics for various groups to a third party such as a Microsoft Windows Azure cloud service, which can use techniques such as consolidation and partitioning to make such computation affordable.

In this paper we model the above examples with the following setting. A data owner (e.g., the marketplace) with limited resources, such as memory and bandwidth, outsources (i.e., forwards) its data stream to one of more remote, untrusted servers (e.g., the cloud). The servers can be potentially compromised, malicious, running buggy software, etc. The owner can register continuous queries on the servers and allow clients (e.g., the sellers) to query the server to receive results upon requests (Figure 1). Clients are in general untrusted; they can be compromised, malicious, and colluding with the server or competing with one another.

Outsourcing data and computation to third parties in the

[1]Assuming that an uncompressed list of 4-byte counters, one for each group, is maintained in memory in order to support fast update on arrival of every streaming tuple.

above setting raises issues of trust. There are several reasons why the data owner may not trust the server, and thus would like to make sure that the clients are receiving the correct result from the server. First, the server may run buggy software or have a slow network, resulting in incorrect results. Second, the server may have an incentive to return incorrect answers. Such an incentive may be a financial one, if the real computation requires a lot of work, whereas computing incorrect answers requires less work and is unlikely to be detected by the client. Moreover, assuming that a server charges the owner according to computation and network resources consumed, a server can have the incentive to deceive the owner and client with fake results for increased profit. Third, in some cases, the applications outsourced to the server may be so critical that the data owner wishes to rule out accidental errors during the computation. Finally, a client may have competing interest with other clients and it can collude with the server to provide wrong answers to its competing clients.

To address the above issues, it is desirable for the data owner and a client to be able to verify the correctness of the results returned by the server. The verification process should be significantly cheaper than actually running the query at the owner and then transmitting the results to the client. We aim at designing a small *signature* that the resource-limited owner can maintain on streaming data and can send it to a client for verification of results. The signature is query dependent and in this paper, we consider the "`Group-By, Sum`" queries (and other related queries such as Count and Average on each group), on any type of grouping imposed on input data. Such queries are very common on data streams. In our previous marketplace example, we need to compute the average (or total) ratings for various groups determined by product id and buyer demographics—a large value of a group implies popularity of the corresponding product within the corresponding buyer demographic. The information can later be used for product recommendation or targeted advertisements.

Previous solutions on authenticating outsourced databases [2], [3], [4], [5] consider simple selection and aggregation queries in a database and do not apply to Group-by queries in a streaming setting. Several recent works consider outsourced data stream [6], [7], but do not consider Group-by queries. PIRS [8] is the most closely related to our work—it proposes a small synopsis, based on a data owner's secret, for Group-by query verification on outsourced data streams. However, PIRS and other outsourcing algorithms for data streams [9], [10], [11], [12], [13] *do not support untrusted clients* that are natural in many practical scenarios. In our marketplace scenario, a client may have incentives to collude with the server to provide wrong answers to competing clients, and hence the data owner cannot necessarily trust the clients with its secret (as a client may reveal the secret to the untrusted server). Existing algorithms assume the owner and the client to be the same entity. If they are separate entities, the owner in existing schemes share the *plain text* of its secret with clients for verification of query answers and therefore the clients must be trusted not to share the secret with the untrusted server. Moreover, PIRS is an algebraic solution. In Section IV, we show that even when the clients are trusted by the owner, such an algebraic scheme might be broken silently by the server to guess the owner's secret (albeit with very small probability), and once the secret is correctly guessed by the server, it can safely cheat on every subsequent queries without the knowledge of the owner.

In this paper we address the above limitations with a novel cryptographic solution called SHAC (*S*treaming *H*istogram *A*uthentication *C*ode). A SHAC consists of a *secret* and a *signature*. In our protocol, the owner initializes a SHAC with its random secret and incrementally updates the signature on arrival of every streaming tuple. The signature is updated based on the value of the tuple and the secret of the SHAC. The signature is computed in such a way that results obtained from the server can be verified for correctness by using the signature and *encrypted* versions of the secrets in SHAC. Under the hardness assumption of the Discrete Log Problem [14], the server cannot provide an incorrect answer that passes the SHAC verification. Moreover, since the owner provides the clients only encrypted versions of its SHAC secret, malicious clients cannot exploit the SHAC, even by colluding with other clients or the server, to compromise the soundness of the protocol. The most fundamental difference between SHAC and previous related work is that we allow a client to be separate from, and potentially untrsuted to, the data owner. To the best of our knowledge, we are the first to consider this model in a streaming setting.

We also consider the *group subset query* where the server continuously evaluates a `Group-By, Sum` query over a large collection of groups, but various clients query and verify only subsets of the groups. This is natural in scenarios where the number of groups is large and different clients are interested in different groups. In our previous marketplace example, one seller may be interested in statistics related to electronics products only, while another seller may be interested in fashion products only. Requiring a client to verify all the groups together, instead of only the groups it is interested in, may incur large communication and computation overhead. However, we show that it is impossible for a limited-memory owner to support such queries where clients can choose arbitrary subsets of groups during query time. On the positive side, we show that our protocol can be used to efficiently support such queries if the subsets of groups various clients are interested in are known a priori. In such a case, our protocol can maintain multiple SHACs, which can be combined later to verify various queries. The extension exploits an interesting composability property of SHAC. We also show how this property of SHAC can be used to support various other scenarios such as distributed data collection, queries over a sliding window, etc.

We have evaluated our protocol with two real datasets as well as with synthetic datasets. Our experiments show that our protocol has a very small overhead. More specifically, on an off-the-shelf laptop, the owner can update a SHAC signature within a few tens of microseconds, while a client can verify a result within a few seconds. The overheads are reasonable and comparable to the non-cryptographic solution PIRS that does not support untrusted clients.

In summary, we make the following contributions.

1) We propose SHAC, a small and efficiently-computable signature to verify the results of streaming `Group-By, Sum` queries executed by a remote untrusted server. Unlike previous work, we support untrusted clients and provide stronger cryptographic guarantee.
2) We show the impossibility of supporting group subset queries without knowing the subsets of groups a priori. We show how our protocol can be efficiently used for such queries when the subsets are known a priori.
3) We propose a several extensions of SHAC to be used in a variety of scenarios including distributed data collection, queries with sliding window, etc.
4) We evaluate SHAC with real and synthetic data sets and show that SHAC is practical and efficient.

The rest of the paper is organized as follows. We review related prior work in Section II. We formulate the problem in Section III and point out limitations of several naïve solutions and existing solutions in Section IV. We present our solution in Section V. We present various extensions of our solution in Sections VI and VII. We present experimental results in Section VIII and conclude in Section IX.

II. Related Work

Motivated by real-world applications involving the delegation of computation to a third party, such as the cloud, there has recently been a lot of interest in proving that the third party is operating correctly. Database community has studied the outsourced database model, where data owners can create, store, and update their databases on third party servers, and users can query the outsourced database. Several existing mechanisms can ensure that the query answer provided by the third party is correct (i.e., data has not been modified by the third party) and complete (i.e., all data within a given range are returned) [15], [16], [17], [18]. The scheme by Li et al. [15] also supports the outsourced database to be dynamically updated by the data owner, and ensures data freshness (i.e., the latest version of data is returned). However, all these works mostly deal with a database snapshot and (range) selection queries on ordered data, join queries, and various set operations. In contrast, we consider a streaming scenario where the verifier is capable of accessing the data only via a single, streaming pass. Moreover, we consider a `Group-By, Sum` query that is not supported by these works.

Several recent works have considered the streaming scenario. For example, PIRS [8], like ours, consider the `Group-By, Sum` query in a streaming setting. Since PIRS is the closest to our work, we will discuss it in more details later in the paper. Other similar examples include work on verifying queries on a data stream with sliding windows using Merkle trees [6] and verifying continuous queries over

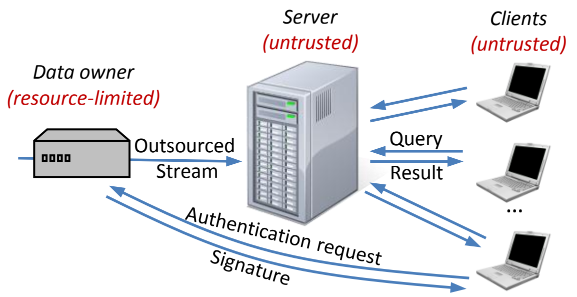

Fig. 1. Outsourced Stream Aggregation

streaming data[19]. The most fundamental difference between these works and our work is that we allow a client to be separate from, and potentially untrsuted to, the data owner. To the best of our knowledge, we are the first to consider this model in a streaming setting.

The cryptography community has also investigated delegation protocols that ensure verifiable results [9], [11], although many of these works are of theoretical interest only. For example, the memory delegation protocols in [11] rely on existence of efficient fully homomorphic encryption algorithm and a polylog PIR algorithm. The notion of a non-interactive streaming verifier, who must read first the input and then the proof under space constraints, was formalized in [10] and extended in [13]. Goldwasser et al. [20] give a powerful interactive protocol that achieves a polynomial time prover and super-efficient verifier for a large class of problems. Even though Goldwasser et al. do not explicitly present their protocols in a streaming setting, it has been subsequently noted that for a large class of computations, the verifier can operate in a streaming fashion. More recently, Cormode et al. [12] introduce the notion of streaming interactive proofs, extending the model of [10] by allowing multiple rounds of interaction between prover and verifier. In contrast to our work, all these works assume that the delegator (i.e., the data owner) and the verifier (i.e., the client) are the same entity, or mutually trusted. Moreover, our solution is non-interactive, and hence more efficient in practice than interactive protocols.

III. Problem Formulation

A. System Model

We consider the system model shown in Figure 1. There are three parties involved. The *Data Owner* owns the data in the sense that it collects relevant data and intends to provide certain service on top of the data. The *Server* is the third party that computes certain functions on the data forwarded by the Owner and answers queries on the Owner's behalf. Finally, multiple *Cliets* query the Server for answers computed on the data stream. In the online marketplace example described in Section I, the marketplace is the Owner, the cloud is the Server, and various sellers are the Clients. In some existing works [9], [10], [11], [12], [13], the Data Owner and the Clients are referred to as the *delegator* and the *verifiers* respectively.

Without loss of generality, we assume that time is measured in discrete ticks, incremented when a new tuple arrives. The Owner and the Server maintain their own clocks (counters) that simply count the number of tuples arrived so far. Let $\mathcal{X}$ denote the entire (potentially infinite) stream and $\mathcal{X}^\tau$ denote the portion of the stream so far at time τ (i.e., τ-length prefix of $\mathcal{X}$). Without explicitly mentioned, we assume that the communication between the Data Owner and the Server is lossless (e.g., with a TCP connection); although our solution can be used with a lossy communication channel as well (Section VII-C).

B. Query Model

We consider a `Group-by, Sum` query that partitions the streaming tuples into a set of groups, computing the sum of values for each group. Denote the τ-th tuple of $\mathcal{X}$ as $t^\tau = (a, b^\tau)$, an increment value of amount b^τ to the a'th group. Without loss of generality, assume that the tuples in $\mathcal{X}$ are partitioned into n groups $\{0, \ldots, n-1\}$. Thus, the query answer can be expressed as dynamic vector of integers $\mathbf{r}^\tau = [r_0^\tau, \ldots, r_{n-1}^\tau] \in \mathbb{N}^n$, containing one aggregate value per group. Initially $\mathbf{r}^0$ is the zero vector. A new tuple $t^\tau = (a, b^\tau)$ increments the corresponding group a in $\mathbf{r}^\tau$ as $r_a^\tau = r_a^{\tau-1} + b^\tau$. When Count queries are concerned, $b^\tau = 1$ for all τ. We also assume that the L_1 norm of the result $\mathbf{r}^\tau$ is bounded by some large m; i.e., for any τ, $\|\mathbf{r}^\tau\|_1 = \sum_{i=0}^{n-1} |r_i^\tau| \leq m$. Our query model is the same as the model considered in previous outsourced stream computation work [8].

In terms of SQL query language, we are interested in queries with the following structure:

```
SELECT G_1, ..., G_m, SUM(A_1), ..., SUM(A_n)
FROM Stream
WHERE ...
GROUP BY G_1, ... G_m
```

The query is analogous to maintaining a histogram, where one bucket is maintained for each group satisfying the `WHERE` predicate. An example query in online marketplace applications looks like the following:

```
SELECT product_id, demographic_id,
               SUM(purchase_volume)
FROM purchase_trace
GROUP BY product_id, demographic_id
```

In the rest of the paper, we will concentrate on the Sum aggregate. Count and a few other aggregates (e.g., average, standard deviation, etc.) can be easily supported as well.

C. Attack Model

We assume that none of the Server and Clients is trusted by the Data Owner and they can potentially be malicious and thus Byzantine. A malicious Server can provide incorrect results to Clients. A malicious Client can collude with the Server and other Clients, and can help the Server to learn Data Owner's secret. Note that our attack model is more general than previous related works [9], [10], [11], [13], [8], where Clients are not allowed to collude with the Server.

D. Security Goals

Our goal is to enable a Client to verify whether a Group-by result reported by the Server is correct. Specifically, a Client should accept a reported result if and only if it equals to the output of a correct execution of the Group-by query over all the items in the datastream observed by the Owner. More formally, we address the following problem: Given a continuous `Group-by, Sum` query, a data stream $\mathcal{X}^\tau$ with the correct results $\mathbf{r}^\tau$ at time τ, design a small and incrementally maintainable signature $\mathcal{T}^\tau$ such that for any τ, given a Group-by result $\mathbf{w}^\tau$ and a signature $\mathcal{T}^\tau$, we raise an alarm if and only if $\mathbf{w}^\tau \neq \mathbf{r}^\tau$.

IV. Possible Solutions

We start with a number of possible solutions and point out their limitations in supporting our goal.

A. A Naïve Solution

Here the Data Owner maintains $\mathbf{r}^\tau$ (along with the Server). On an authentication request from a Client at time τ, it computes a hash of target groups in $\mathbf{r}^\tau$ and sends the hash value to the Client. The Client can then compare the hash value with the hash value computed on the result given by the Server. Although this incurs a small network overhead for the Data Owner, it has two drawbacks. First, the Owner has a very large memory overhead for maintaining $\mathbf{r}$. Second, the Owner needs to recompute the hash value for a new query if aggregate values in even a single group changes.

B. Random Sampling

The Owner can reduce the memory overhead by maintaining only a small fraction $r < 1$ of randomly sampled groups, instead of all the n groups. For verification, the Server then sends a hash of the correct values of these sampled groups to the Client, who then compares it with the hash of the values returned by the Server. Then, if the Server cheats on γ of the n groups, the Client will be able to detect it with probability roughly r^γ, which is very small for practical values of r and γ. In other words, to ensure a reasonable accuracy, the value of r should be large, which means a large memory and communication overhead of the Owner.

Moreover, this solution does not work with untrusted Clients (who may collude with the Server). Note that the Owner must tell a Client which groups it is maintaining, so that the Client knows which groups, or their hash values, to compare. However, an untrusted Client can collude with the Server and leak the identity of those groups to the Server. After that the malicious Server can silently cheat on the groups not in the set maintained by the Owner.

C. PIRS

PIRS [8], like our work, considers verification of `Group-By, Sum` (or histogram) queries. However, unlike our cryptographic approach, PIRS uses algebraic and probabilistic techniques. In the basic version of PIRS, the Data Owner chooses a secret random number α in $\mathbb{Z}_p$ and over a given $\mathbf{r}$, incrementally maintains the synopsis

$$\mathcal{T}(\mathbf{r}) = (\alpha - 1)^{r_0} \cdot (\alpha - 2)^{r_2} \cdots (\alpha - n)^{r_{n-1}}$$

Given any $\mathbf{w}$ returned by the Server, the Client computes the following:

$$\mathcal{T}(\mathbf{w}) = (\alpha - 1)^{w_0} \cdot (\alpha - 2)^{w_2} \cdots (\alpha - n)^{w_{n-1}}$$

To verify the correctness of a result $\mathbf{w}$, the Client receives $\mathcal{T}(\mathbf{r})$ from the Owner and accepts $\mathbf{w}$ as correct if $\mathcal{T}(\mathbf{r}) = \mathcal{T}(\mathbf{w})$.

In PIRS, the Owner's secret α must be known by the Client for verification, and hence it requires the Owner to trust the Client. An untrusted Client can share the the value of α to the Server, who can then easily construct incorrect answers without being detected. Therefore, *the PIRS mechanism does not work in our model where clients can be untrusted and can potentially collude with the Server*. The limitation applies to other existing streaming delegation protocols as well [9], [10], [11], [12], [13].

► **An Attack**. Even with trusted Clients, PIRS may be compromised through the following *collision attack*. In PIRS, there is a small probability that the Server can find a $\mathbf{w} \neq \mathbf{r}$ such that $\mathcal{T}(\mathbf{w}) = \mathcal{T}(\mathbf{r})$. The Server can detect such a collision by providing $\mathbf{w}$ but not getting any alarm. Once such a collision is found, it can find the secret value α by solving the polynomial $\prod_i (\alpha - i)^{w_i - v_i} = 1$. This can be done efficiently for small values of $\sum_i |w_i - v_i|$ (See Corollary 14.16 in [21]). Once the Server can find the value of α, it can continue cheating indefinitely without the knowledge of the Owner. The important point is that even though the collision probability is small for a single answer, it is not unlikely to happen at least once over the lifetime of the stream. And, once such a collision happens, the Server can continue cheating *for the rest of the system lifetime without the Owner knowing about it.*

These limitations of PIRS motivate us for a cryptographic solution. Solutions based on sound cryptographic principles are more secure against strong adversarial attacks. We would like to design a small cryptographic signature that is a function of values in different groups and can be incrementally updated with arithmetic operations such as increment or decrement within each group. However, existing cryptographic signature of MAC techniques support updates for edit operations such as insertion and deletion of individual blocks of bits [22] and are not applicable for arithmetic updates required for our target `Group-By, Sum` query. We address this limitation in the next section by designing a novel cryptographic signature.

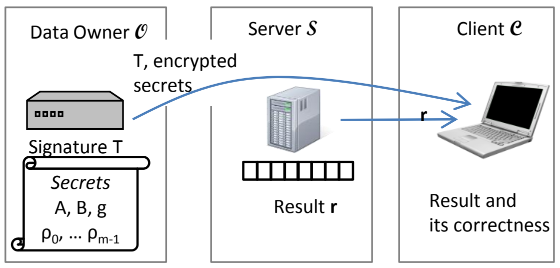

Fig. 2. The SHAC Protocol

V. Our Solution

Suppose each tuple $(a, b) \in \mathcal{X}$ belongs to the group $a \in [0, n)$. Let w_i^τ denote the number of tuples in $\mathcal{X}^\tau$ with the group i. Assuming that the size of the stream $\mathcal{X}$ is bounded by m, let p be a large prime number $p > \max(m, n)$. All our computations are in the field $\mathbb{Z}_p$, that is, all additions, subtractions, and multiplication are done modulo p. Let $\rho_0, \rho_1, \ldots, \rho_{m-1}$ be random numbers within the range $[0, p-1]$. Suppose a has the binary representation $a_0 a_1 \ldots a_{k-1}$, where $k = \lceil \log_2 n \rceil$, and β is the function $\beta(a) = \sum_{i=0}^{k-1} a_i \rho_i$. Finally, α is the function $\alpha(a) = g^{A\beta(a)+B}$, where $g \in (\mathbb{Z}_p)^*$.

A. The SHAC Protocol

A sketch of our solution is shown in Figure 2. It requires different parties to run the following protocol.

- *Protocol at the Owner $\mathcal{O}$:* $\mathcal{O}$ maintains a SHAC. A SHAC consists of two components: a secret and a signature. The secret component is initialized in the beginning once, while the signature component is incrementally undated as new tuples are seen. In the beginning of the protocol, $\mathcal{O}$ generates secret random numbers A, B, $\rho_0, \rho_1, \ldots, \rho_{k-1}$, all smaller than p, and $g \in (\mathbb{Z}_p)^*$. These secret values constitute the secret component of $\mathcal{O}$'s SHAC. The signature component of the SHAC is denoted by $\mathcal{T}^\tau$ and is updated by following the protocol below.
    1) Initialize: $\mathcal{T}^0 \leftarrow 1$
    2) On arrival of the τ'th tuple (a, b), set $\mathcal{T}^\tau \leftarrow T^{\tau-1} \times \alpha(a)^b$

  Thus, for any τ, $\mathcal{T}^\tau = \prod_{i=0}^{n-1} \alpha(i)^{w_i^\tau}$.
  Note that the value of b in the tuple (a, b) can be negative as well, which implies decrementing the accumulated value of group a by $|b|$. Since $\mathbb{Z}_p$ does not support division, we first need to compute $\alpha(a)^{-1}$, the multiplicative inverse of $\alpha(a)$ in $\mathbb{Z}_p$ (e.g., in $O(\log p)$ time by using Euclid's gcd algorithm [23]). Then we can compute $\alpha(a)^{-1.|b|}$.

- *Protocol at the Server $\mathcal{S}$:* $\mathcal{S}$ maintains a vector $\mathbf{r}^\tau$ of length n such that r_i^τ denotes the sum of values of all tuples with group i in $\mathcal{X}^\tau$. More formally,
    1) Initialize: $r_l^0 \leftarrow 0, 0 \leq l < n$

2) On arrival of the τ'th tuple (a, b) with group a, set $r_a^\tau \leftarrow r_a^{\tau-1} + b$. Also, set $r_i^\tau \leftarrow r_i^{\tau-1}$ for all groups $i \neq a$.

- *Verification protocol at a Client $\mathcal{C}$.* $\mathcal{C}$ receives the result from $\mathcal{S}$ and verifies its correctness by receiving from $\mathcal{O}$ its SHAC signature in plain text and SHAC secret in encrypted form. More specifically:
  1) $\mathcal{C}$ retrieves the result vector $\mathbf{r}^\tau$ from $\mathcal{S}$.[2]
  2) $\mathcal{C}$ then retrieves $\mathcal{T}^\tau, g^{A\rho_0}, \ldots, g^{A\rho_{m-1}}, g^B$ from $\mathcal{O}$.
  3) Finally, $\mathcal{C}$ accepts $\mathbf{r}^\tau$ as correct only if $\mathcal{T}^\tau = \prod_{i=0}^{n-1}((\prod_{j|i_j=1} g^{A\rho_j})g^B)^{r_i^\tau}$.

Note that the signature $\mathcal{T}^\tau$ is essentially a MAC computed incrementally over the stream $\mathcal{X}^\tau$. As we will show next, the MAC has strong security properties that make it hard for the Server to cheat. Also note that the secrets of the Owner are shared with a Client only in encrypted form (i.e., raised to the powers of g), which ensures the security of our protocol even when clients are malicious or colluding with the Server. We will elaborate on this security property in the next section.

B. Synchronization

As mentioned before, the Owner and the Server maintain own clocks in terms of the number of tuples in the stream; i.e., both increment their own clocks on arrival of a new tuple. Due to communication lag between the Owner and the Server, their clocks can differ. Now, since the signature $\mathcal{T}$ at the Owner and the result $\mathbf{r}$ at the Server evolve as new tuples arrive, a client needs to run the verification protocol with $\mathcal{T}^\tau$ and $\mathbf{r}^\tau$ for a same τ. Such synchronization can be achieved easily with the following schemes.

► **Query-ahead**. When a Client wants to verify the query results at a certain time τ, it will send its requests to the Owner and the Server shortly before time τ. The Owner will send its SHAC signature and the Server will send the result when their own clocks reach τ.

This scheme requires the Client to plan ahead and make requests some time before it needs the result. If the Client cannot afford this, it can use the next scheme.

► **Buffering**. For this scheme, we require that:

1) The Client contacts the Owner and the Server within a small time window of length k (i.e., within which at most k tuples arrive). And,
2) The Owner keeps a buffer of the last k tuples.

Then, the Client first requests the Server for $\mathbf{r}^\tau$ for the Server's current clock time τ and then it contacts the Owner to get a $\mathcal{T}^\tau$. Even if the Owner's current time is $\mathbf{r}^{\tau'}, \tau' - \tau \leq k$, it can produce $\mathbf{r}^\tau$ from its buffer of k previous tuples. This works because the Owner gets to see a tuple before the Server sees it. Alternative to Step (2) above, the Owner can maintain last k versions of its signature $\mathcal{T}$, in which case the Client can first get a $\mathbf{r}^\tau$ from the Server and then ask the Owner to provide the appropriate signature at time τ.

[2]As an optimization, the Server can send only the nonzero groups.

C. Security Analysis

We now show that, following our protocol, a Client will always be able to detect whenever the Server cheats with an incorrect result, even when the Server colludes with a number of other Clients. The security analysis of our protocol uses the hardness assumption of the Discrete Log problem.

► **Discrete Log Problem**. If p is prime and $g, h \in \mathbb{Z}_p^*$, we write $log_g(h) = x$ if $x \in \mathbb{Z}$ satisfies $g^x = h$. The problem of finding such an integer x for a given $g, h \in \mathbb{Z}_p^*$ (with $g \neq 1$) is called the Discrete Log Problem. The problem is conjectured to be hard: there is no known polynomial time algorithm for the Discrete Log problem [14]. Many cryptographic algorithms, including elliptic curve cryptography, are based on the hardness assumption of the Discrete Log problem. (By using standard Pholig-Hellman method, one can always assume that the underlying group is of prime order [24].)

We also use the following lemma (Lemma 2.4.1 in [25]):

LEMMA 5.1. *[25] If $k_1, k_2, \ldots, k_s$ are nonzero integers such that, for $i \neq j, k_i \neq \pm k_j$ (mod p); and $\mathbf{w} \in \mathbb{Z}_p^s \backslash \{\mathbf{0}\}$, then*[3]

$$\Pr_{\sigma \in S_s}\left[\sum_{i=1}^{s/2} k_{\sigma^{-1}(i)} w_i = \sum_{j=s/2+1}^{s} k_{\sigma^{-1}(j)} w_j = 0 (mod\ p)\right] \leq 1 - \frac{2}{s^2}.$$

Finally, we use the following results, proved based on the hardness assumption of the Discrete Log Problem.

LEMMA 5.2. *There is no efficient algorithm $\mathcal{D}$ that, given the inputs $(g^{c_1}, g^{c_2}, \ldots, g^{c_k})$, can compute $(\alpha_1, \alpha_2, \ldots, \alpha_k) \neq (0, 0, \ldots, 0)$ such that $\prod_i g^{c_i \alpha_i} = 1$.*

PROOF: The proof is by reduction from the Discrete Log problem. Suppose such a $\mathcal{D}$ exists. We now show that, $\mathcal{D}$ can be used to efficiently solve the Discrete Log Problem, which contradicts with its hardness assumption.

Suppose we want to solve the Discrete Log Problem $h = g^x$ (i.e., given h and g, we want to compute x). Then, we generate random numbers $\gamma_1, \gamma_2, \ldots, \gamma_k$ and give a random permutation of $G = (h^{\gamma_1}, \ldots, h^{\gamma_{k/2}}, g^{\gamma_{k/2+1}}, \ldots, g^{\gamma_k})$ as inputs to $\mathcal{D}$. Then, the output of $\mathcal{D}$ is $(\alpha_1, \alpha_2, \ldots, \alpha_k) \neq (0, 0, \ldots, 0)$ such that

$$x \sum_{i=1}^{k/2} \alpha_i \gamma_i + \sum_{i=k/2+1}^{m} \alpha_i \gamma_i = 0 \tag{1}$$

According to Lemma 5.1,

$$\sum_{i=1}^{k/2} \alpha_i \gamma_i \neq 0$$

for large k with a high probability. Thus, Equation 1 gives the target discrete log answer x. This completes the proof. ■

LEMMA 5.3. *There is no efficient algorithm $\mathcal{F}$ that, given $g^B, g^{A\rho_0}, g^{A\rho_1}, \ldots, g^{A\rho_{k-1}}$ for any values of $\rho_0, \rho_1, \ldots, \rho_{k-1}$, can compute A/B.*

[3]S_s denotes the permutation group over $\{1, 2, \ldots, s\}$.

PROOF: If such an $\mathcal{F}$ exists, one can solve the Discrete Log Problem $\alpha = \beta^x$ as follows. He generates k arbitrary numbers $\gamma_0, \gamma_1, \ldots, \gamma_{k-1}$ and gives $\beta, \beta^{x\gamma_0}, \beta^{x\gamma_1}, \ldots, \beta^{x\gamma_{k-1}}$ as inputs to $\mathcal{F}$ to produce x and, thus to solve the Discrete Log Problem. ■

We now prove the soundness of our protocol under the worst-case assumption that $\mathcal{S}$ colludes with some malicious Clients $\mathcal{C}$ and exploits all the information $\mathcal{O}$ shares with the $\mathcal{C}$ for verification.

THEOREM 1. *If $\mathcal{S}$ gives an incorrect output $\mathbf{r}' \neq \mathbf{r}$, computed by exploiting all information shared between $\mathcal{O}$ and $\mathcal{C}$, $\mathcal{C}$ will be able to detect the incorrectness.*

PROOF: Under our protocol, $\mathcal{C}$ concludes $\mathbf{r} = \mathbf{r}'$ only if $T = \prod_{i=0}^{n-1}((\prod_{k|i_k=1} g^{A\rho_k})g^B)^{r'_i}$, i.e., if

$$\sum_{i=0}^{n-1}(A\beta(i) + B)r_i = \sum_{i=0}^{n-1}(A\beta(i) + B)r'_i$$

Denoting $r'_i = r_i + \delta_i \pmod{p}$, the above equation is true only if

$$A\sum_{i=0}^{n-1}\beta(i)\delta_i = -B\sum_{i=0}^{n-1}\delta_i$$

Since $\mathbf{r} \neq \mathbf{r}'$, $(\delta_0, \delta_2, \ldots, \delta_{n-1}) \neq (0, 0, \ldots, 0)$. According to Lemma 5.3, $\mathcal{S}$ or $\mathcal{C}$ cannot compute A/B or B/A. Therefore, the only way $\mathcal{S}$ can cheat $\mathcal{C}$ with $\mathbf{r}'$ in place of $\mathbf{r}$ is if both the following equations are true:

$$\begin{aligned}\sum_{i=0}^{n-1}\delta_i &= 0\\ \sum_{i=0}^{n-1}\beta(i)\delta_i &= 0\end{aligned}$$

However, according to Lemma 5.2, given $(g^{\beta(0)}, g^{\beta(1)}, \ldots, g^{\beta(n-1)})$, $\mathcal{S}$ cannot efficiently generate $\mathbf{r}'$ with deltas $(\delta_0, \delta_1, \ldots, \delta_{n-1}) \neq (0, 0, \ldots, 0)$ such that $\prod_i g^{\beta(i)\delta_i} = 1$, or $\sum_i \beta(i)\delta_i = 0$. This completes the proof. ■

D. Complexity Analysis

For analysis purpose, we assume that p is the smallest prime number that is larger that $\max(m, n)$. According to Bertrand-Chebyshev theorem [26], $p \leq 2\max(m, n)$. The space complexity of SHAC at the Data Owner is $O(\log n)$, due to the logarithmic number of ρ values it needs to maintain. On arrival of each tuple, the Owner needs to compute the function $\beta()$, which can be done in $O(\log n)$ time and the function $\alpha()$, which can be done in time $O(\log m + \log n)$ (by exponentiation with repeated squaring). In addition, for a Sum query with a new tuple (a, b), the Owner needs to compute $\alpha(a)^b$, which needs $O(\log b)$ time.

For verification or a single query result, a Client needs to receive $\mathcal{T}$ and encrypted secrets, which incurs $O(\log n)$ communication cost for the first time verification and $O(1)$ cost for subsequent verifications (since encrypted secrets of a SHAC can be reused). To verify the result, for every nonzero group a, the Client needs to first multiply g^B and all $g^{A\rho_i}, 1 \leq i \leq \log n$, in $O(\log n)$ time, and then compute its power of w_i in $O(\log w_i)$ time. Denoting the number of nonzero groups as $|\mathbf{r}|$, the total time required for a single verification is $O(|\mathbf{r}|(\log n + \sum_i \log w_i)) = O(|\mathbf{r}| \log \frac{mn}{|\mathbf{r}|})$.

THEOREM 2. *SHAC requires $O(\log n)$ space at the Data Owner, spends $O(\log mn)$ (respectively, $O(\log mnb)$) time to process a tuple for count (respectively, sum with value b) queries, transfers $O(\log n)$ bits from the Owner to a Client, and $O(|\mathbf{r}| \log \frac{mn}{|\mathbf{r}|})$ time to verify a result $\mathbf{r}$.*

VI. QUERIES ON DIFFERENT SUBSETS OF GROUPS

We have so far considered a single continuous Group-by query and verifying all the groups together. In many applications, a Client may be interested in only a subset of the groups the Server is monitoring. In our online marketplace example, the marketplace may run a continuous Group-by query on all possible product types. Then, some sellers may be interested in only the groups involving electronics products, while some others may be interested in groups involving fashion products only. One naïve approach to this would be to require all Clients to retrieve and verify all the groups and then to ignore the groups they are not interested in. However, when the total number of groups is large, retrieving and verifying all the groups may impose prohibitively large communication and computation overhead. In this section we consider more efficient solutions that enable a Client to retrieve and verify only the subset of groups it is interested in.

We consider two variants of such queries. In *dynamic subset queries*, Clients can choose arbitrary subsets of groups, without telling the Owner a priori which groups they will be querying on. In *static subset queries*, Clients a priori decide which subsets of groups they will be querying in future so that the Owner can tailor its signatures accordingly. Although dynamic subset queries are ideal, we show that supporting such queries is impossible for a limited memory Owner. On the other hand, static subset queries can be supported efficiently.

A. Hardness of Dynamic Subset Queries

We now show that supporting dynamic subset queries with a small limited memory at the Data Owner is hard. The argument is information theoretic and it depends on how the protocols at the Data Owner and the Client work and our conclusion holds for any general protocols. For concreteness, we first assume that the Owner and the Client use protocols similar to ours; we will relax this assumption later. The Discrete Log problem is considered hard on multiplicative groups inside finite fields, and our protocols use linear combinations that appear in exponents of a generator g. For simplicity, we now consider the group written additively so that we can focus on linear combinations.

Let $\mu \ll n$ be the size of the Owner's memory. Given a query, we need to verify the result vector $\mathbf{r}$ of length n with non-negative integer entries (with zero values in the groups not appearing in the query). In a protocol like ours, the Owner also uses a secret vector θ of length with n non-zero entries from the finite field F_p, where θ_i is the secret used for group i. In our protocol, θ_i is random and given by the function $\alpha(i)$.[4] We also assume that the Owner is deterministic and works in two modes: traffic monitoring mode, and the verification mode. In the *traffic monitoring mode* it inspects a new tuple and updates its signature based on the group and the value of the tuple. In the *verification mode* it accepts an input $S \subset \{0, .., n-1\}$. If it can produce a verification signature restricted to the groups in S, it returns the signature to the Client. However, the Owner may not have necessary information in its limited memory to produce such a signature, in which case it outputs SORRY, denoting that it is unable to verify the query. In a protocol like ours, the Owner computes the inner product of $\mathbf{r}$ and θ restricted to S, namely $F_v(S) = \sum_{i \in S} r_i \theta_i \bmod p$. Note that in the verification mode, the Owner does not change its state. When it is clear we write $F(S)$ instead of $F_v(S)$. We have assumed $\mu \ll n$ and so we expect the Owner not to be able to store information about every subset S. We now show that under mild assumptions even a random S can make the Owner to fail to verify the results of groups in S, making it to output SORRY.

We consider a *query adversary*, who can generate arbitrary queries to the Owner and tries to make the Owner to fail to verify the results of his queries.[5] We assume that the traffic through the Owner is generated by a stochastic process (or a deterministic process, with a mild assumption we mention next). In view of $\mu \ll n$, we now make the following mild assumption:

ASSUMPTION 1 (ENTROPY ASSUMPTION). *The entropy of the vector* $\mathbf{r}$ *exceeds* 2μ. *This means the Owner can not store the entire vector* $\mathbf{r}$ *in its memory using any encoding techniques.*

However, the Owner may store some limited information in its memory about the traffic it sees. For example, it may store information about a subset S' and this set S' may change with time.

LEMMA 6.1. *Let the traffic be generated by a stochastic source so that the entropy assumption holds. The query adversary can find a subset* S *of groups such that the Owner can not verify the query on this set correctly.*

PROOF: Let $i \geq 1$ be fixed. After the arrival of i-th tuple let $\Sigma = \{S_1, S_2, ..., S_R\}$ be the queries that the Owner can verify correctly. We do not make any assumption on what the Owner has stored but we assume that Σ is well-defined. For an $S \in \Sigma$ let I_S be the binary vector that is its characteristic function, namely $I_S(j) = 1$, if and only if $j \in S$. Let $\Sigma' \subset \Sigma$ be a largest subset such that $\{I_S | S \in \Sigma'\}$ are linearly independent as vectors over F_p. We consider two cases.

Case 1: Σ' is unique. Let $\mathbf{M}$ be the matrix formed by writing the vectors $I_S, S \in \Sigma'$ as rows. If $|\Sigma'| = n$, then we can uniquely find $\mathbf{r}$ since $\mathbf{MDr} = \mathbf{h}$ where $\mathbf{D} = diag(\theta_1, ...\theta_n)$ and $\mathbf{h}$ is the column vector of respective answers from the Owner for inputs S in Σ'. But this contradicts the entropy assumption. If $|\Sigma'| < n$, the dimension of the space V spanned by Σ' is less than n. Then, the probability that the vector I_S for a random S is linearly dependent of the row vectors in $\mathbf{M}$ is the same as the probability that it falls in V, and the probability is $1/p \ll 1/2$ for a large prime p. Thus $S \notin \Sigma'$ and by the definition of Σ' the Owner has to output SORRY in this case. Since this holds for any $i \geq 1$, we have our lemma.

Case 2: Σ' is not unique. Then we find a collection of vectors $\Sigma' \subset \Sigma$ that span the vector space of largest dimension and take this space as V in the above analysis to have our lemma. This space will be unique, since given two distinct vector spaces V and V' the span of their unions will have a larger dimension. ■

In the above, we assumed that the Owner generates signature in a way our protocol does. Now we relax these assumptions to show that the hardness conclusion holds for arbitrary protocols.

► **Probabilistic Signature Function**. In the above we assumed that the Owner follows a deterministic protocol and does not change its verification state so that the set Σ is well defined. But our incompressibility arguments extend to any probabilistic protocol in a simple fashion. Let ω be a sequence whose entropy is denoted by $H(\omega)$. Now, a randomized algorithm uses a sequence of random unbiased coin flips (denoted by ρ) along with its input ω and one can view it as a deterministic algorithm once ρ is specified. Since ρ may help in compression of ω, the conditional entropy $H(\omega|\rho) \leq H(\omega)$. However, since ρ is independent of ω, these entropies are equal. Thus we have the standard fact that ω can not be compressed any better by a randomized algorithm than a deterministic one; for more details see [27].

► **General Signature Function**. In the above we used the function $F_r(S) = F(S) = \sum_{i \in S} r_i \theta_i$ where θ_i is fixed, and the linear structure allowed us to solve a resulting system of equations *efficiently*. A general deterministic function $F_r(S)$ need not be linear, and given many query values one may not get a system of equations that is easy to solve. For example, this would be the case if a protocol outputs a encrypted version of $\sum_{i \in S} r_i \theta_i$. But all $F_r(S)$ must satisfy the following *consistency condition* so that it can be useful for answering `Group-By, Sum` queries: If $\mathbf{r} \neq \mathbf{r}'$ then there must exist an S such that $F_r(S) \neq F_{r'}(S)$. The consistency condition puts a restriction on F_r: querying on all possible S and obtaining the values of $F(S)$ will define a unique $\mathbf{r}$. Such an F, and the associated recovery algorithm for $\mathbf{r}$ (using exhaustive search over a table rows indexed by r, columns indexed by S, with

[4]Our proof does not depend on the randomness property of θ_i, suggesting that θ_i can be deterministic as well.

[5]Another more powerful type of adversary is a *traffic adversary*, who, in addition to generating arbitrary queries to the Owner, can introduce arbitrary tuples or re-arrange the order of arrival of tuples in the data stream. Since a traffic adversary is strictly more powerful than a query adversary, our claims for a query adversary naturally follow to a traffic adversary.

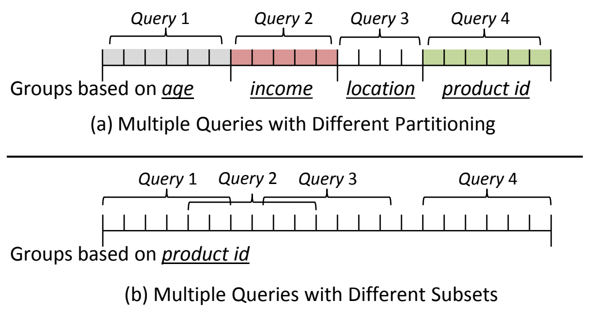
Query 1
Query 2
Query 3
Query 4
Groups based on age
income
location
product id
(a) Multiple Queries with Different Partitioning
Query 1
Query 2
Query 3
Query 4
Groups based on product id
(b) Multiple Queries with Different Subsets

Fig. 3. Multiple queries with (a) different partitioning and (b) different subsets of groups.

$r, S)$ entry being $F_r(S)$ will yield a compression and coding technique for the vector $\mathbf{r}$ using only a coding of size μ. This contradicts the entropy assumption which states the Owner has too small a memory in comparison to the entropy of $\mathbf{r}$.

B. Static Subset Queries

We now consider a relatively easier problem of answering multiple queries on subset of groups where the subsets are known a priori. Our solution uses an important property of SHAC.

PROPOSITION 1. *SHAC is decomposable, that is, for any* $\mathbf{r}_1$ *and* $\mathbf{r}_2$, $\mathcal{T}(\mathbf{r}_1 + \mathbf{r}_2) = \mathcal{T}(\mathbf{r}_1)\mathcal{T}(\mathbf{r}_2)$, *under the same secret component of SHAC.*

The above property allows us to maintain SHAC over small subsets of groups and to combine them to produce SHAC for larger subsets.

► **Overlapping Subsets of Groups**. In general, subsets of groups different Clients are interested in may overlap. For example, one Client may be interested in the sales counts of various electronic products, while another Client may be interested in that of various fashion products (some electronic products can be classified as fashion products as well). Figure 3(b) shows the scenario. Assume that various Clients are interested in k such queries, where the i'th query is interested in a subset s_i of groups and the subsets in various queries may overlap with each other.

One naïve solution would be to maintain one SHAC at the Owner for all groups and to require each Client to verify the complete result (including the groups it is not interested in). However, this would require the Server to send the complete result (with all the groups) to the Client, incurring high communication and computation overhead, especially when the Client is interested in only a small subset of all the groups. Another solution would be to maintain k SHAC synopses at the Owner, where the i'th SHAC is computed over only the tuples that belong to any of the groups in the ith query. Then a Client receives only the results of groups in s_i and verifies it with the i'th SHAC signature received from the Owner. However, this requires the Data Owner to update up to k SHAC signature, one for each Client, on arrival of every tuple. This can be prohibitively expensive for a large number of Clients and high-speed streams.

The $O(k)$ update cost can be avoided by partitioning and merging the query ranges as follows. For simplicity assume that k queries are interested in k different ranges of groups (e.g., Client i is interested in the range of groups $\sigma_i = l_i, l_i + 1, \ldots, r_i$, denoted as $[l_i, r_i]$, and so on). For example, assume that the partitioning scheme creates 10 groups and three Clients are interested in the groups with ranges $\sigma_1 = [1, 5]$, $\sigma_2 = [1, 6]$, and $\sigma_3 = [3, 8]$ respectively. We can partition these ranges into the smallest number of non-overlapping sub-ranges that can be combined to produce all the query ranges: $\zeta_1 = [1, 2]$, $\zeta_2 = [3, 5]$, $\zeta_3 = [6, 6]$, $\zeta_4 = [7, 8]$. Then, the Data Owner can maintain one SHAC signature for each of these sub ranges. Due to the decomposability property of SHAC synopses, these can be combined to produce SHAC signature for any of the original ranges. For example, the synopses for ζ_1 and ζ_2 can be multiplied to produce the signature for σ_1 to verify the first Client's answer.

It is easy to see that given k queries with different ranges, corresponding non-overlapping sub-ranges can be computed in $O(k \log k)$ time by sorting. Moreover, there can be at most $2k$ such non-overlapping sub-ranges. Therefore, the Owner needs to maintain at most $2k$ SHAC synopses—a small overhead given the small size of SHAC signatures. However, since sub-ranges are non-overlapping, each tuple belongs to only one sub-range and hence the Owner needs to update only one of the SHAC synopses, making the update cost $O(1)$.

► **Concurrent Queries on Various Partitioning Schemes**. In real applications, subset queries may involve groups on various dimensions. Consider a Client who is interested in two Group-by queries on a single attribute (e.g., number of mp3 players sold) but with different partitioning on the input tuples (e.g., one on various age groups, and the other on various income levels), and wishes to verify both of them together. Figure 3(a) shows such a scenario. Suppose a Client registers queries on k such orthogonal dimensions, where the ith query partitions the tuples into n_i groups for a total of $n = \sum_{i=1}^{k} n_i$ groups.

A naïve solution for this approach would be to maintain k SHACs and to apply k instances of our protocol. This would result in $O(k)$ space, $O(k)$ update cost, and $O(k)$ verification cost. However, by treating all k queries as one unified query with n groups, we can use our protocol to verify the combined vector $\mathbf{r}$. More specifically, the modified protocol maintains one SHAC and on receiving one tuple, it updates the SHAC signature k times, once for each of the k groups the tuple belongs to. Thus, even though the update cost remains $O(k)$, the space complexity and the verification cost reduces down to $O(1)$, which is same as a single query verification.

Overlapping subsets and concurrent partitioning can be combined: a query can use various partitioning schemes and choose a subset of groups from each partitioning scheme.

VII. Extensions

The decomposability property of SHAC allows our protocol to support various other scenarios, as discussed below.

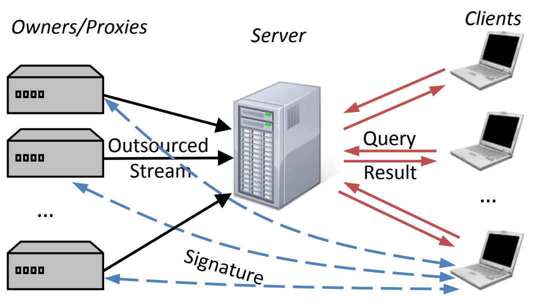

Fig. 4. Distributed Authentication

A. Distributed Data Collection

In some scenarios, it is natural for a Data Owner to employ multiple *proxies* each of which independently forwards tuples to the Server. In some other scenarios, multiple mutually trusted Data owners may push data to a single data repository in one Server. Queries from Clients are then made over all the tuples forwarded by all the proxies. For example, a sensor network may have multiple base stations for practical reasons. Each base station can forward its tuples to a central Server so that users can make queries on data collected by all base stations. The model is shown in Figure 4.

The decomposability property of SHAC can naturally allows verification of `Group-By, Sum` queries in the above model: All proxies or Data Owners use the same SHAC secret (this is why they all need to be mutually trusted) and independently maintains local SHAC signatures on whatever tuples they see and forward to the Server. To verify a query result, a Client collects SHAC signatures from all proxies or Owners, multiplies them to generate a global SHAC signature, and uses the global SHAC to verify the answer according to our original protocol in Section V-A.

B. Handling a Sliding Window

The decomposability property of SHAC also allows us to extend SHAC for periodically sliding windows using standard techniques [28]. Suppose a Client is interested in the statistics collected over a window of last k days, sliding the window by 1 day at a time. Then, we can build a SHAC for every 1-day period, and keep it in memory until it expires from the sliding window. The SHAC for the entire window of last k days is given by multiplying all the unexpired SHACs. SHAC for any subset of days within the last k days can be computed in a similar fashion in needed. Various window sizes consisting of between 1 to k periods can also be supported by decomposing the k periods into a number of dyadic intervals, as discussed in [8].

C. Tolerating Communication Losses

In [8], authors present how to use PIRS to verify Group-by results when the communication between the Owner and the Server is lossy and hence incorrect values of a small number γ of groups are acceptable. Authors present an exact and an approximate solution for raising alarms only when the number of errors exceeds a predefined threshold. This solution thus allows some room of error for the server (e.g., using semantic load shedding): as long as there are not too many errors (less than a threshold) in the final result, the Server is still considered trustworthy. The authors also present a polylogarithmic space solution to locate and correct the incorrect groups when the number of errors is tolerable.

The above solutions use PIRS as a black box. Since SHAC provides the same semantic interface as PIRS (but stronger security guarantee in the presence of untrusted clients), SHAC can also be used as a black box in these solutions. We omit the details here for brevity.

Data set	Update time	Verification time
Bing Click Log	$27\mu Sec$	$\approx 5Sec$
World Cup Dataset	$30\mu Sec$	$\approx 1Sec$

TABLE I
COMPUTATIONAL OVERHEAD AT THE OWNER (UPDATE TIME PER TUPLE) AND AT A CLIENT (VERIFICATION TIME PER RESULT) UNDER TWO REAL DATA SETS

VIII. EXPERIMENTS

We have implemented our protocol using GNU C++ and the NTL library[6], which provides big integers and modular arithmetic. We assume that group ids are 64-bit numbers and the total number of items is less than 2^{64}. We set p as the smallest prime above 2^{64}. The experiments are run on an off-the-shelf desktop PC with Intel Core2 Duo 2.5GHz CPU and 4GB RAM.

A. Real workloads

We first use two real data sets to evaluate the computation overhead at the Owner and the Client. The *Bing Click Log* is a stream of clicks from the Microsoft Bing search engine. The log contains around 10 million records, with each record containing attributes such as user IP address, search term, clicked url, etc. Here we are interested in a `Group-By, Count` query: counting the the frequencies of various (search-term, clicked URL) pairs. Such statistics are important to discover high click-through-rate URLs for various search terms. The second data set, *World Cup Dataset*, consists of Web server logs from the 1998 Soccer World Cup. Each record in the log contains several attributes such as a timestamp, a client ID, response size, etc. We used the log of days 50 and 51 that have about 100 million records and 370,000 unique users. Here we are interested in a `Group-By Sum` query: counting the total number of bytes sent to each user (i.e., sum of response sizes of all requests by a user).

On the Data Owner side, we are interested in how much time it takes to update the SHAC signature on arrival of a single tuple. On the Client side, we are interested in how much time it takes to verify the result it receives from the Server. Table I shows the overhead of our protocols in terms of these two metrics for our two real data sets. As shown, for both data sets,

[6]http://shoup.net/ntl/

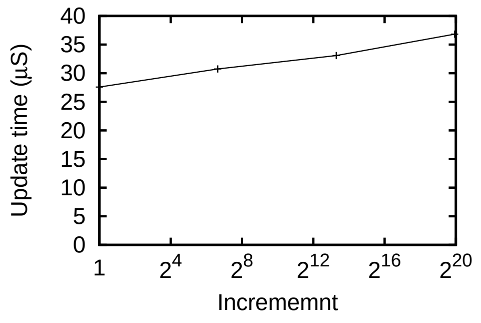

Fig. 5. Update cost per tuple at the Owner

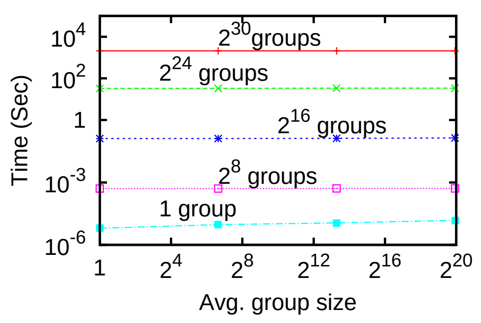

Fig. 6. Verification cost per query at a Client

the Owner takes around $30\mu Sec$ to update its signature for a single tuple, implying that it can handle more than $30,000$ tuples per second on a off-the-shelf desktop, several orders of magnitude more than the tuple arrival rates in both our real data sets. The update cost is slightly higher for the World Cup Dataset, showing the additional overhead of handling a Sum query over a Count query.

For comparison, we have also implemented PIRS. We found that our protocol is 5-10$\times$ slower that PIRS. This is due to the cryptographic operations we need in order to support untrusted clients and to provide stronger cryptographic security guarantee.

Table I also shows the time required to verify a result at a Client. For the Bing Click Log and the World Cup Dataset, it takes around 5 sec and 1 sec to verify a result respectively. The higher verification overhead for the Bing Click Log is due to a large number of groups in the data set. Such a verification time is reasonable in practice because (a) it is comparable to the time required to download a new result from the Server and the verification can be done in the background while the result is being downloaded, and (b) clients are not expected to make queries very frequently since the result may not change significantly within a small time window.[7]

B. Synthetic Datasets

The computation complexity of our protocol depends on three factors: (i) number of groups, and (ii) average increment value per tuple (for `Group-by, Count` queries, the value is 1), and (iii) average accumulated value per group. The cost of updating a signature at the Owner depends on (ii), while the cost of verifying an answer by a Client depends on (i) and (iii). We now use synthetic data sets to experimentally evaluate the overhead of our protocol under these various factors.

► **The Owner**. Computational complexity of the owner depends on the average increment value per tuple. Figure 5 shows the cost of updating a SHAC signature for a single tuple, as a function of the increment value. As shown, the update can be done very fast ($< 50\mu S$), allowing the Owner to be capable to handle more than 20,000 updates per second. Note that updating a signature for a value v requires an exponentiation of power v, which can be done in $O(\log v)$ time. This is also shown in Figure 5—the update cost increases logarithmically with the increment size.

[7]With a large number of clients, the server may still need to answer queries frequently; but the verification cost is distributed among all the clients.

► **The Client**. Computational complexity of a Client depends on both the number of nonzero groups and the average size of a nonempty group. Figure 6(b) shows the verification time at a Client as functions of the these two factors. As expected, the cost increases linearly with the number of nonempty groups. Within each group, the verification requires exponentiation, and hence the cost increases logarithmically with the average group size. Overall, the cost is reasonably small—it is less than 1 second even for a result containing 2^{16} nonempty groups.

C. Subset Queries

We now use the Bing Click Log to evaluate our protocols to verify queries involving subsets of groups. We consider queries involving three orthogonal dimensions we can extract from the log: user's location (obtained from the geo-location of user's IP address), clicked business category (obtained by using a yellowpage directory), and query time. Within each dimension, we consider the same query as before: a `Group-by, Count` query on groups given by various (search-term, clicked URL) pairs. We vary the number of Clients. Each Client makes a query on a random subset of 300 consecutive groups. For comparison, we consider three different schemes:

- *Single-SHAC*: Here the Owner maintains a single SHAC over all the groups, and each Client verifies all the groups together.
- *Unoptimized-SHAC*: Here the Owner maintains multiple SHACs, one for each query. On arrival of a tuple, the Owner updates all of them. To verify a result, a Client retrieves the appropriate SHAC. This is essentially one of the naïve solutions mentioned in Section VI-B.
- *Optimized-SHAC*: Here the Owner uses the optimizations described in Section VI-B.

Table II shows the computational overheads of various schemes. As shown, the single-SHAC scheme that uses a single SHAC for all queries has a very high verification overhead (the communication overhead not shown here is also large). This is because a Client needs to verify all the groups, even though it is interested in a small subset of groups. The use of subset-aware multiple SHACs significantly reduces this

	single-SHAC	UnOpt-SHAC	Opt-SHAC
Update time (*At the Owner*)	79μSec	823 μSec (10 Clients) 8191 μSec (100) 79.5 mSec (1000)	81μSec
Verification time (*At a Client*)	14.6 Sec	29.6 μSec	12.1 μSec

TABLE II
COMPUTATIONAL OVERHEAD AT THE OWNER (UPDATE TIME PER TUPLE) AND AT A CLIENT (VERIFICATION TIME PER RESULT) FOR SUBSET QUERIES

verification cost, since a Client can verify only the groups it is interested in. However, a naïve way of using multiple SHACs, as is done in unoptimized SHAC, can introduce significant update cost. In the unoptimized SHAC scheme, the update cost per tuple increases almost linearly with the number of Clients in the system. For example, with only 1000 Clients, the Owner needs close to 80 milliseconds to update its signature for each tuple, limiting it to process only 12 tuples per second. Thus, the scheme can easily become impractical for a large number of Clients. Our optimizations (Optimized-SHAC) avoid this problem: in addition to reducing the verification time at a Client, it also reduces the update time at the Owner. Moreover, these costs remain independent of the number of Clients in the system, allowing Optimized-SHAC to scale to a large number of Clients.

IX. CONCLUSION

We have proposed SHAC, a small and efficient signature to verify outsourced `Group-By, Sum` queries. We considered a setting where a data owner employs an untrusted remote server to run a continuous `Group-By, Sum` query on a data stream it forwards to the server. Untrusted clients then query the Server for computed results. More importantly, a client can efficiently verify the correctness of the results by using a small and easy-to-compute signature provided by the owner. Our work complements previous works on authenticating remote computation of selection and aggregation queries and unlike prior work on remote Group-by queries, we support untrusted clients (who can collude with themselves or with the server) and provide stronger cryptographic guarantees. Experimental results on real and synthetic data show that our solution is practical and efficient.

REFERENCES

[1] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Fault-tolerance in the borealis distributed stream processing system," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005.

[2] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine, "Authentic data publication over the internet," *J. Comput. Secur.*, vol. 11, pp. 291–314, April 2003.

[3] 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*, 2006.

[4] 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*, 2005.

[5] H. Pang and K.-L. Tan, "Authenticating query results in edge computing," in *Proceedings of the 20th International Conference on Data Engineering (ICDE)*, 2004.

[6] F. Li, K. Yi, M. Hadjieleftheriou, and G. Kollios, "Proof-infused streams: enabling authentication of sliding window queries on streams," in *Proceedings of the 33rd international conference on Very large data bases (VLDB)*, 2007.

[7] S. Papadopoulos, Y. Yang, and D. Papadias, "Cads: continuous authentication on data streams," in *Proceedings of the 33rd international conference on Very large data bases (VLDB)*, 2007.

[8] K. Yi, F. Li, G. Cormode, M. Hadjieleftheriou, G. Kollios, and D. Srivastava, "Small synopses for group-by query verification on outsourced data streams," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 3, pp. 1–42, 2009.

[9] S. Benabbas, R. Gennaro, , and Y. Vahlis, "Verifiable delegation of computation over large datasets," in *Proceedings of the 31st International Cryptology Conference (CRYPTO)*, 2011.

[10] A. Chakrabarti, G. Cormode, and A. Mcgregor, "Annotations in data streams," in *Proceedings of the 36th International Colloquium on Automata, Languages and Programming: Part I (ICALP)*, ser. ICALP '09, 2009, pp. 222–234.

[11] K.-M. Chung, Y. Kalai, F.-H. Liu, and R. Raz, "Memory delegation," in *Proceedings of the 31st International Cryptology Conference (CRYPTO)*, 2011.

[12] G. Cormode, J. Thaler, and K. Yi, "Verifying computations with streaming interactive proofs," *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 17, no. 159, 2010.

[13] G. Cormode, M. Mitzenmacher, and J. Thaler, "Streaming graph computations with a helpful advisor," in *Proceedings of the 18th annual European conference on Algorithms: Part I (ESA)*, 2010, pp. 231–242.

[14] D. Stinson, *Cryptography: Theory and Practice,Second Edition*, 2nd ed. CRC/C&H, 2002.

[15] 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*, 2006.

[16] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and integrity in outsourced databases," *ACM Trans. Storage*, vol. 2, no. 2, pp. 107–138, 2006.

[17] G. Nuckolls, "Verified query results from hybrid authentication trees," in *Proceedings of Data and Applications Security (DBSec)*, 2005.

[18] H. Pang and K. Tan, "Verifying completeness of relational query answers from online servers," *ACM Transactions on Information and System Security (TISSEC)*, vol. 11, no. 2, 2007.

[19] S. Papadopoulos, Y. Yang, and D. Papadias, "Continuous authentication on relational streams," *VLDB Journal*, vol. 10, pp. 161–180, April 2010.

[20] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: interactive proofs for muggles," in *Proceedings of the 40th annual ACM symposium on Theory of computing (STOC)*, 2008, pp. 113–122.

[21] J. V. Z. Gathen and J. Gerhard, *Modern Computer Algebra*, 2nd ed. New York, NY, USA: Cambridge University Press, 2003.

[22] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental cryptography and application to virus protection," in *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing (STOC)*, 1995.

[23] D. E. Knuth, *The Art of Computer Programming, (Addison-Wesley Series in Computer Science and Information*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1978.

[24] S. Pohlig and M. Hellman, "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance," *IEEE Transactions on Information Theory*, vol. 24, pp. 106–110, 1978.

[25] J. A. Horwitz, "Applications of cayley graphs, bilinearity, and higher-order residues to cryptology," Ph.D. dissertation, Stanford University, September 2004.

[26] D. Bressoud and S. Wagon, *A Course in Computational Number Theory*. New Jersey, USA: John Wiley and Sons, 2000.

[27] A. K. Zvonkin and L. A. Levin, "The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms," *Russian Mathematical Surveys*, vol. 25, no. 6, pp. 83–124, 1970.

[28] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows: (extended abstract)," in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, 2002.