Query Access Assurance in Outsourced Databases

Wangchao Le, Feifei Li *Member, IEEE*

Abstract—Query execution assurance is an important concept in defeating lazy servers in the database as a service model. We show that extending query execution assurance to outsourced databases with multiple data owners is highly inefficient. To cope with lazy servers in the distributed setting, we propose *query access assurance* (QAA) that focuses on IO-bound queries. The goal in QAA is to enable clients to verify that the server has honestly accessed all records that are necessary to compute the correct query answer, thus eliminating the incentives for the server to be lazy if the query cost is dominated by the IO cost in accessing these records. We formalize this concept for distributed databases, and present two efficient schemes that achieve QAA with high success probabilities. The first scheme is simple to implement and deploy, but may incur excessive server to client communication cost and verification cost at the client side, when the query selectivity or the database size increases. The second scheme is more involved, but successfully addresses the limitation of the first scheme. Our design employs a few number theory techniques. Extensive experiments demonstrate the efficiency, effectiveness and usefulness of our schemes.

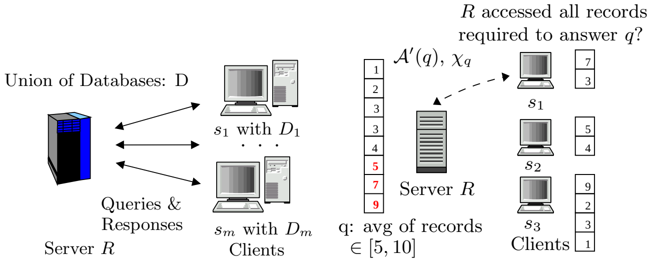
Index Terms—Database as a service, database security, quality of services, query assurance, service enforcement and assurance.

1 Introduction

Data are increasingly collected in a distributed fashion. In such systems, data are generated by multiple clients and forwarded to a central server for management and query processing. We denote these systems as the *outsourced databases with multiple data owners*. Since data are distributed in nature, we can also view such systems as the *outsourced distributed databases*, and in short, the *distributed databases*, as shown in Figure 1(a). The remote access of the database inevitably raises the issue of trust, especially when the server is provided as a service [14] by a third party that clients may not fully trust, thus brings the needs of auditing the query processing efforts performed by the server.

There are two types of dishonest servers. A *lazy server* [32] returns incorrect responses for saving his computation resources. The incentive is to provide services to more clients or lower his operation cost. A *malicious server* is willing to pay considerable amounts of efforts (much more than honestly executing the query if necessary) to manipulate the clients so that they will accept wrong query results.

Not surprisingly, to guard against a malicious server is much harder and more costly than defeating a lazy server. In many practical applications, clients only need to worry about a lazy server instead of a malicious one. The choice of selecting a particular server's service has indicated a reasonable degree of trust between data owners and the server, thus the chance that the selected server has any malicious intent should be low. However, the server still has plenty of incentive to be lazy to lower his operation cost or save his computation resources [32].



(a) System model.

(b) A query example.

Fig. 1. Query access assurance in distributed databases.

An important result in defeating a lazy server is the *query execution assurance* [32]. Formally, denote the amount of work (computation and IO costs) that a server has to do in honestly executing a batch of queries Q as $\mathcal{W}_Q$ and the correct query answers for Q as $\mathcal{A}_Q$, and let the actual amount of work the server has paid for Q be $\mathcal{W}'_Q$ and the corresponding query results as $\mathcal{A}'_Q$. Query execution assurance is to ensure the followings. If a client accepts $\mathcal{A}'_Q$, then $\mathcal{W}'_Q/\mathcal{W}_Q \geq \theta$ holds with a high probability, where θ is some threshold value that is close to 1.

Sion [32] has designed an elegant scheme (based on the *ringers concept*) that achieves the query execution assurance in a single data owner model. It also needs the assumption that queries are submitted in batches. The core idea is to let the client execute a subset of queries (from the batch to be submitted) locally and obtain their answers first. Challenge-tokens, using some one-way, collision resistant hash function, are then constructed based on these answers. These tokens are submitted together with the batch of queries and the server is required to return, along with the query answers to all queries, the query ids that produced these challenge-tokens. We denote this scheme as the *ringers scheme*.

- *Authors are with the Computer Science Department, Florida State University, Tallahassee, FL 32306. E-mail:* {le,lifeifei}@cs.fsu.edu. *Contact Author: Feifei Li.*

Motivation. We can certainly apply the ringers scheme in distributed databases to provide the query execution assurance. Unfortunately, there are some major limitations. Firstly, it requires a client to submit his queries in a batch that contains at least tens of queries [32]. This means a client must delay and buffer his queries into a group of sufficient size, which introduces latency that is not desired and/or might not be feasible in time-sensitive scenarios.

Secondly, to produce the challenge-tokens, the client needs to obtain all data records that must be involved in answering any query in the query batch. There are two ways of addressing this issue. The first approach is to assume that some query authentication techniques (e.g., [16], [18], [19], [23], [25]) have been employed when publishing databases to the server, and the queries in the same batch only touch records from one common segment of records in the database [32]. Given these assumptions, the client can execute one range selection query to obtain and authenticate the required data records for all queries in the same batch. The client then randomly selects a few queries from the query batch and these queries are executed on these records to produce challenge tokens. This assumption could be very restrictive and incurs significant overheads for both the client and the server, especially in distributed fashions where ensuring the query authentication with multiple data owners requires more expensive techniques and special assumptions [22], [26]. In the worst case, the entire database has to be transferred and authenticated to the client to construct some challenge tokens.

When there is only a single data owner and the client is the data owner himself, the second solution is to ask the client to retain a copy of the database, thus removing the needs to obtain and authenticate the raw data records first before executing queries locally. In distributed databases, each data owner's database is relatively small, hence it is reasonable to ask a client to retain a copy of his database locally. However, since data are distributed in multiple locations, collecting all data records for answering queries *exactly* at one place still incurs expensive communication cost.

Distributed data and the need of auditing a single query make the task of realizing query execution assurance in distributed databases *efficiently* very difficult. Given these challenges, our observation is that a large number of queries are IO-bound, i.e., the dominant cost when executing these queries is contributed by IOs. Thus, a reasonable compromise in these cases is to verify that the server has *honestly accessed all data records that are required to answer a query*. Essentially, we seek efficient solutions that can verify the data access pattern of the server for a single query. We denote this problem as the *query access assurance* problem. This problem is also important when clients want to ensure that the server has honestly retrieved all relevant records for general search queries in distributed databases.

Problem Formulation. Formally, assume that m data owners $\{s_1, \dots, s_m\}$ forward their databases to the server R. We focus on the case where the clients in the system are data owners themselves. The general case in which a client other than a data owner may request a query is addressed in Section 4. The databases from all clients conform to the same relational schema. For a client s_i, his database D_i is a collection of records $\{r_{i,1}, \dots, r_{i,n_i}\}$. Without loss of generality, we assume that clients have distinct records, i.e., for $\forall i, j \in [1, m]$, $i \neq j$, $D_i \cap D_j = \emptyset$. This is easily achievable by imposing a primary key field id_r. The server R maintains the union of all databases, denoted as D, and answers queries from clients, as shown in Figure 1(a). The number of records in D is $|D| = N$ (hence $\sum_{i=1}^{m} n_i = N$). In the sequel, unless otherwise specified, $|X|$ for a set X denotes the number of elements in X.

A query q's *selection predicate* defines a set of records in every D_i (and D) that satisfy its query condition. Consider the following example in SQL: "*select sum(A_3) from D where* $10 \leq A_1 \leq 100$ *group by A_2*". The selection predicate of this query is $A_1 \in [10, 100]$ and it defines the set of records that must be involved in order to produce the final query result. For a database D_i and a query q, we denote this set of records as $q_{i,t}$. We can apply the same definition to the database D and define the set q_t as well. Obviously, $q_t = \cup_{i=1}^{m} q_{i,t}$. We also define $\rho_q = |q_t|/|D|$ as the *query selectivity* of q.

The client expects a query answer $\mathcal{A}'_q$ for q and an evidence χ_q that R has honestly accessed all records in q_t. Let the actual set of records from q_t that R has accessed to produce $\mathcal{A}'_q$ be q_a, i.e., $q_a \subseteq q_t$. Formally,

Definition 1. *Query Access Assurance* [Qᴀᴀ]*: For a query q issued by a client s_i ($i \in [1, m]$), the server R should return an answer $\mathcal{A}'_q$ with an evidence χ_q. The client s_i performs a verification procedure $\mathcal{V}$ with χ_q that outputs either 0 or 1. If $q_t = q_a$, then $\mathcal{V}(\chi_q) = 1$ always. If $q_t \neq q_a$, $\mathcal{V}(\chi_q) = 0$ with a high probability δ (negatively correlated with $|q_a|/|q_t|$).*

An example is shown in Figure 1(b), where s_1 needs to ensure that R has accessed all records in $q_t = \{5, 7, 9\}$. In general, the server may only need to access a small subset of attributes for every record in q_t to answer a query q. To simplify the discussion, we assume that the goal is to ensure that the complete content of the record has been accessed for every record in q_t and address the above issue in Section 4.

A lazy server will try to reduce $|q_a|$ as much as possible. Meanwhile, Qᴀᴀ may introduce overhead for an honest server (to generate the evidence χ_q). Thus, it is important to keep the overhead for the server in producing χ_q small, relatively to the query cost. In addition, the following costs are also critical: the communication cost among clients, and between the client and the server, the computation cost for clients.

Our Contribution. We present efficient solutions to ensure query access assurance that guards against a lazy server in distributed databases for any single query initiated by any client. Specifically,

- We first present a basic scheme (QAA-BSC) in Section 2 with a simple construction and achieves a high success probability. It has a small communication cost among clients and almost no query overhead for the server. However, the server to client communication cost and the client's verification cost become expensive when either ρ_q or N increases.
- To overcome the above limitation, we design an advanced scheme (QAA-ADV) in Section 3. It utilizes *group generators* to produce one-way, collision-resistant, online permutations of indices for a set of records. It then uses a novel, efficient algorithm to map unique ids to unique primes on the fly. The mapping is different for different queries. QAA-ADV enjoys the same high success probability as QAA-BSC. It introduces very small communication overheads and some small computation costs.
- We show that both schemes are update-friendly and address some other issues in Section 4.
- We illustrate the efficiency and effectiveness of our schemes by extensive experiments in Section 5.

We survey other related works in Section 6 and conclude the paper in Section 7. The frequently used notations are summarized in Figure 2. In this work, the $\log x$ operation by default refers to $\log_2 x$.

2 THE QAA-BSC SCHEME

The basic intuition in our design is to let a client randomly sample a *small* subset of records q_c from q_t for a query q by contacting a few other clients (and himself). The server is required to provide a proof that he has touched these sampled ones in answering q. An honest server has to touch all records from q_t, hence these sampled records will be touched anyway. A lazy server has no knowledge on how these samples were selected, thus, he has to retrieve all records from q_t to be absolutely sure that he will not be caught.

The simplest realization of this idea is to require the client to collect records into q_c, and the server to return all records from q_t to the client, along with the query response. However, such an approach is too expensive communication-wise. On the other hand, simply sending the ids or some pre-computed hash values for these records, to save the communication cost, has serious security loopholes. In this case, the server may utilize an index structure to retrieve the ids or pre-computed hash values (in main memory) for all records in q_t, but without reading the actual contents of any records. Our design addresses these challenges.

2.1 The Scheme

Pre-processing of auxiliary information. Let the nth prime as p_n, e.g., $p_1 = 2$. Before sending his database to the server, each client augments each record in his database with a set of τ unique prime numbers for some small integer τ (say $\tau = 4$). One way of achieving this is as follows. Client s_1 sets $p_1 = 2$ and attaches $\{p_{(j-1)\times\tau+1}, \ldots, p_{j\times\tau}\}$ to his jth record $r_{1,j}$ for $j = 1, \ldots, n_1$. Note that efficient polynomial algorithm exists for the primality test [1]. For an integer n, the state of the art deterministic algorithm has a complexity of $O((\log n)^4)$; and the probabilistic test (e.g., the Miller-Rabin primality test) is even faster in practice [15]. Hence, given the prime p_n, the next prime p_{n+1} could be found efficiently by repeatedly applying the primality test to consecutive odd numbers.

After processing r_{1,n_1}, s_1 sends n_1 and $p_{n_1\times\tau}$ to s_2 who follows the same procedure, but starts with the prime $p_{n_1\times\tau+1}$. Let $N_i = \sum_{\kappa=1}^{i} n_\kappa$, the ith client s_i attaches primes $\{p_{(N_{i-1}+(j-1))\times\tau+1}, \ldots, p_{(N_{i-1}+j)\times\tau}\}$ to his jth record $r_{i,j}$. The augmented databases are sent to the server. Note that this step is very efficient, for example, it only takes a few seconds to generate the first million primes in our testbed computer, a standard PC. This process is also only a one-time cost. In the sequel, a record r includes its augmented primes.

Symbol	Description
m	The number of data owners (clients).
s_i, D_i, n_i	The ith client, its database and the number of records it owns.
$r_{i,j}$	The jth record of D_i.
R, D	The server and its unioned database for all D_i's.
N, $\lvert D\rvert$	The number of records in D.
ρ_q	Query selectivity.
$q_{i,t}$	The set of records involved to answer q in D_i.
q_t	The set of records involved to answer q in D.
q_a	Actual set of records accessed by R to answer q in D.
χ_q	Evidence that R has honestly accessed all records in q_t.
p_n	The nth prime.
τ	The number of unique primes attached to each record in QAA-BSC.
C_q	Client's challenge for q.
ω_t, ω_f	The number of records in q_c and $\bar{q}_c$.
β_q	A random bit sequence.
q_c, $\bar{q}_c$	Two small subsets of q_t.
P_q, $P_{\bar{q}}$	Two sets of primes (from q_c and $\bar{q}_c$ respectively) that define C_q.
$\mathbb{P}$, ℓ	The set $\mathbb{P}$ contains ℓ unique primes.
C_q^t, C_q^f	Two products (based on P_q and $P_{\bar{q}}$) that compose C_q.
η	The number of bits needed to represent id_{max}.
γ	A random number to increase the bits for primes in QAA-ADV.
λ, F_λ	The first prime larger than $2^{\eta+\gamma}$ and its prime field (without 0).
g, G_{F_λ}	A generator and a set of generators for F_λ.

Fig. 2. Summary of frequently used notations.

Construct the challenge for a query q. Without loss of generality, suppose that client s_1 wants to send a query q to the server R. He will construct a challenge C_q for q. First, s_1 generates two small random integers ω_t and ω_f (say less than 30). He also generates a random bit-sequence β_q. The length of β_q is a random, small integer value (say in the range of $[10, 100]$). Client s_1 sends both q and β_q to the server for the query execution.

Next, s_1 repeats the followings. He contacts a randomly selected client s_α, and asks s_α to randomly select a few records from $q_{\alpha,t}$, randomly distribute them into two sets and return them to s_1. Note that s_α only needs to check his records against the selection predicate of q, but not to execute q itself. He does not need to find all records in $q_{\alpha,t}$, but randomly picks a few records that have been identified to satisfy q's selection predicate.

Client s_1 repeats this step with other randomly selected clients (s_1 could be chosen as well), until a total of ω_t and ω_f records for the two sets respectively have been collected in $q_{i,t}$'s from the sampled clients.

We denote the two sets of records sampled from $q_{i,t}$'s as q_c and $\bar{q}_c$. Clearly, $q_c \subset q_t$ and $\bar{q}_c \subset q_t$. Also, by our construction, $|q_c| = \omega_t$ and $|\bar{q}_c| = \omega_f$. As we will see shortly, ω_t and ω_f are very small values (less than 30). They are usually much smaller than the number of records in q_t.

For any record r, let $h(r) = H(r \oplus \beta_q) \bmod \tau + 1$, where $\oplus$ is the concatenation and H is a one-way, collision resistant hash function, such as SHA-1. Let $p(r, j)$ represent the jth attached prime of the record r for $j \in [1, \tau]$, client s_1 prepares the challenge C_q as follows, where ϑ_r is randomly chosen per record:

$$C_q : \begin{cases} C_q^t = \prod_{r \in q_c} p(r, h(r)) \\ C_q^f = \prod_{r \in \bar{q}_c} p(r, \vartheta_r) \text{ where } \vartheta_r \in [1, \tau] \text{ and } \vartheta_r \neq h(r). \end{cases}$$

In the sequel, we refer to the two sets of primes that were sampled to construct C_q^t and C_q^f as P_q and $P_{\bar{q}}$ respectively. Basically, s_1 selects the $h(r)$th prime from the τ augmented primes for each record r in q_c into P_q, and a random non-$h(r)$th prime for each record in $\bar{q}_c$ into $P_{\bar{q}}$. C_q^t and C_q^f are simply the two products of these selected primes. Since C_q^t and C_q^f are constructed at the client-side and retained locally by the client s_1, hence, their constructions are independent to the submission of the query q to the server (once β_q has been generated) and the server-side execution of q, thus can be done in parallel without delaying the normal query process.

Note that C_q only depends on the corresponding primes for records in q_c and $q_{\bar{c}}$, but not the records themselves; given the value of β_q, sampled clients can select which prime to use for a selected record locally. Hence, to reduce the communication cost, s_1 can forward β_q to sampled clients. With a pre-determined hash function H, each contacted client could pick the right prime for each sampled record and send back only the *two products* of these primes, plus a single bit per product to flag which set (q_c or $\bar{q}_c$) a product's associated records belong to.

Server's execution of the query q**.** While executing the query q in the database D, for a record $r \in q_t$ that the server R has touched, R finds r's augmented prime $p(r, h(r))$. It is possible for the server to compute $h(r)$, since β_q has been submitted with q, assuming that R and all clients have agreed on a pre-selected hash function H. The query overhead for the server is the calculation of the hash function for every record in q_t. The set that contains all primes selected by the server in answering q is clearly $\{p(r, h(r)) \text{ for } \forall r \in q_a\}$. The evidence χ_q is simply this set. R then returns to s_1 both $\mathcal{A}'_q$ and χ_q.

Client's verification. The verification is to try to divide C_q^t and C_q^f respectively by every prime in χ_q. If R is honest, clearly, $q_c \subseteq q_t = q_a$, and $P_q \bigcap \chi_q = P_q$, $P_{\bar{q}} \bigcap \chi_q = \emptyset$; s_1 should expect that, at the end of iterating through all primes in χ_q, C_q^t equals 1 and C_q^f equals the initial value of C_q^f. Essentially, the verification function $\mathcal{V}(x_q)$ does the followings,

$$\text{for } \forall p \in \chi_q \begin{cases} C_q^t = C_q^t / p, & \text{if } \gcd(C_q^t, p) = p; \\ \text{stops and outputs } 0, & \text{if } \gcd(C_q^f, p) \neq 1. \end{cases}$$

$\mathcal{V}(x_q)$ outputs 1 if and only if $C_q^t = 1$ when all primes in χ_q have been checked and it has not terminated and outputted 0. Clearly, there is no false positive in this scheme, i.e., if R is honest, then $\mathcal{V}(x_q) = 1$. A lazy server can be caught with a high probability.

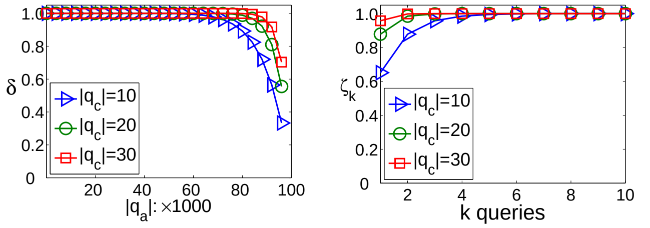
2.2 Theoretical Guarantee and Cost Analysis

We first explain some designs in Qaa-Bsc, specifically, the introduction of τ unique primes per record, the concatenation of β_q in computing $h(r)$ for a record r, and the inclusion of C_q^f in the challenge C_q.

If each record has only one associated prime, the server can find all primes for records in q_t as χ_q, without retrieving the records themselves, since the mapping between a record and its prime is deterministic. A prime is much smaller in size than a record, the server may store all primes in a hash table and significantly reduces his IO cost. Using multiple primes per record and making the choice on the prime selection random per query in the challenge construction solve this issue. We further enforce that this choice has to be made with the combined knowledge of a record r (so that r has to be accessed for the server to make the right choice for records in q_c) and a random bit-sequence β_q per query. The dependence on β_q means that a fresh, random bit-sequence needs to be used for a new query, to ensure that the selections on primes are random across different queries, so that the server cannot store the choice from one query and use it for other queries without reading the contents of records any more. The inclusion of C_q^f in the challenge C_q is also necessary, since otherwise the server may simply return all augmented primes for records in q_t for any query q. This will guarantee that they factor any C_q^t to 1 eventually.

Next, we analyze the success probability δ of Qaa-Bsc for catching a lazy server R, in terms of the number of records that R has retrieved from the set q_t. Recall that in Definition 1, the actual set of records from q_t that R has accessed in answering q is referred to as q_a.

Firstly, for any record r in q_a, the server is able to find its $h(r)$th prime to include in χ_q. The server may randomly select one prime per record for the remaining records in $q_t - q_a$. The probability for the server to select all "correct" (the $h(r)$th) primes for these records is $(\frac{1}{\tau})^{|q_t - q_a|}$, which is negligible unless $|q_a|$ is very close to $|q_t|$. For a lazy server, this is clearly not the case. Hence, to simplify our analysis and without affecting it to a notable degree, we ignore the chance that the server may select correct primes for records that he has not accessed. The complete analysis, with the account for the slight chance that the server may find correct primes for records in $q_t - q_a$ by randomly selecting one prime per such a record, appears in the full version of the paper.



(a) A single query. (b) k queries, $|q_a| = 0.9|q_t|$.

Fig. 3. The success probability, $|q_t| = 10^5$.

Given this observation, clearly, when $|q_a| < |q_c|$, $\delta = 1$, i.e., a lazy server that has touched less than $|q_c|$ number of records from q_t (no matter which they are) will always be caught. When $|q_a| \geq |q_c|$, whether the client can catch such a lazy server depends on the event if $q_c \subseteq q_a$, i.e., $\delta = 1 - \Pr[q_c \subseteq q_a]$. In our scheme, records in q_c are selected uniformly at random from q_t, there are $\binom{|q_t|}{|q_c|}$ number of possibilities for the formation of q_c. On the other hand, since R has no knowledge on records in q_c, any choice he has made in choosing a record to be included in q_a could be viewed as being made independently at random. Hence, given one fixed q_c, R has at most $\binom{|q_a|}{|q_c|}$ number of chances to include all records from this q_c in q_a. Hence:

$$\Pr[q_c \subseteq q_a] = \binom{|q_a|}{|q_c|} \Big/ \binom{|q_t|}{|q_c|} = \prod_{i=1}^{|q_c|} \frac{|q_a| - |q_c| + i}{|q_t| - |q_c| + i}. \quad (1)$$

Theorem 1. *For a single query q, the probability δ that a client catches a lazy server in* QAA-BSC *is,*

$$\delta = \begin{cases} 1, & \text{if } |q_a| < |q_c| \\ 1 - \prod_{i=1}^{|q_c|} \frac{|q_a| - |q_c| + i}{|q_t| - |q_c| + i} \geq 1 - (\frac{|q_a|}{|q_t|})^{|q_c|}, & \text{otherwise.} \end{cases} \quad (2)$$

Since the challenges are independent for different queries, the probability that a lazy server avoids being caught drops exponentially after multiple queries. As a result, we have:

Corollary 1. *The success probability ζ_k for clients to catch a lazy server after k queries $\{q_1, \dots, q_k\}$ in* QAA-BSC *is: $\zeta_k = 1 - \prod_{i=1}^{k}(1 - \delta_i)$, where δ_i is given by applying Theorem 1 to the query q_i.*

To help understand these results, say $|q_t| = 10^5$, we plot the values of δ with $|q_c| = 10$, 20 and 30 respectively when we change the values of $|q_a|$ from 0 to $|q_t|$ in Figure 3(a). It shows that even if the client has only sampled 10 records in q_t (only 0.01% of $|q_t|$), the server has to retrieve 70% records in q_t in order to have a slight chance (1% probability) of escaping from being caught lazy. If the client has increased his sample size to 20 records, the server has to retrieve about 80% records to have a slight chance (1% probability) to escape. Extending this experiment to k queries, for simplicity and without loss of generality, suppose that we have k queries with the same sizes for q_c, q_a and q_t. Let $|q_t| = 10^5$ and $|q_a| = 0.9|q_t|$, Figure 3(b) indicates that the escaping probability for a lazy server drops quickly to 0 for very small values of k (3 or 4), even if the client has just sampled $|q_c| = 10$ records from q_t (0.01% of $|q_t|$) to build his query challenge and the server has accessed 90% of records in q_t. An *even nicer property* is that $|q_c|$ *does not need to increase* to maintain the same high success probability as $|q_t|$ increases, as long as $|q_a|/|q_t|$ keeps the same ratio (based on equation 2). These results suggest that, even with a tiny set q_c, the server has to access a significantly large portion of records in q_t to avoid being caught in the QAA-BSC scheme for a single query. If a client has asked a few queries, it is almost impossible for a lazy server to escape.

Cost Analysis. The client-side communication cost is $8(|q_c| + |\bar{q}_c|)$ bytes in the worst case, assuming that each prime takes 8 bytes (64-bit prime). In practice, this cost is smaller as the sizes of the two products of these primes from each contacted client typically require less number of bits. The computation cost for all clients at the client-side is dominated by the computation of the hash function $h(r)$ for every record in q_c or $\bar{q}_c$, which is in turn determined by the cost of the one-way, collision resistant hash function H, denoted by $\mathcal{C}_H$. The client-side computation cost is then $\mathcal{C}_H(|q_c| + |\bar{q}_c|)$. $\mathcal{C}_H$ is very small in practice, for example, SHA-1 takes a few μs for inputs that are up to 500 bytes in a standard PC.

The client-server communication cost consists of two parts. For the client-to-server part, the only overhead is the transmission of the random bit sequence β_q, which is very small in tens of bits. For the server-to-client part, the communication overhead is contributed by the number of primes in χ_q, denoted as $|\chi_q|$. For an honest server, $|\chi_q| = |q_t| = \rho_q N$. Hence, the communication overhead is $8\rho_q N$ bytes. The server-side query overhead is contributed by the calculation of the hash function $h(r)$ for every record r in q_t, which is very small comparing to the normal query execution cost, and is calculated as $\mathcal{C}_H|q_t|$.

In the verification step, for a prime $p \in \chi_q$, the client can check if p divides C_q^t (C_q^f) efficiently by finding the $\gcd(p, C_q^t)$ ($\gcd(p, C_q^f)$). If $\gcd(p, C_q^t) = p$, then p divides C_q^t, otherwise it does not. Computing the gcd can be done efficiently in logarithmic time [4] w.r.t. C_q^t (C_q^f). Since $|\chi_q| = |q_t| = \rho_q N$, the client-side's verification cost is $O(\rho_q N \log(\max(C_q^t, C_q^f)))$. Finally, there is a storage overhead of $8\tau N$ bytes in the QAA-BSC scheme, to store τ distinct primes per record.

3 THE QAA-ADV SCHEME

A major limitation in the QAA-BSC scheme is that the server-to-client communication cost becomes expensive when $|q_t|$ or N increases. It also implies a linearly more expensive verification cost for the client. We introduce an advanced scheme, QAA-ADV, to address these issues. The basic intuition is to, instead of keeping the challenge C_q locally, ask the client to forward C_q to the server. This requires that the valid evidence χ_q must be built based on both the challenge C_q and the honest execution of q.

3.1 The Scheme

In this scheme, we do not require any auxiliary information to be stored in the databases. Instead, we assume that a distinct, one-way and collision-resistant function $f_q : r, \beta_q \rightarrow i \in [1, \ell]$ exists per query q, where r is a record in the database D and β_q is a random bit sequence. We also assume the existence of a collision-resistant mapping function $\mathcal{G} : i \rightarrow p \in \mathbb{P}$, where $i \in [1, \ell]$ and the set $\mathbb{P}$ contains ℓ unique primes $\{\nu_1, \ldots, \nu_\ell\}$, where $\ell \in \mathbb{Z}^+$ and $\ell \geq N$. The exact elements in $\mathbb{P}$ as well as the value of ℓ and the constructions of f_q and $\mathcal{G}$ will be discussed in Section 3.2. Note that $\mathbb{P}$ is a *conceptual* set without being materialized.

Construct the challenge for a query q. Without loss of generality, assume that client s_1 has a query q to be submitted to the server. Following the exactly same fashion as the construction in the QAA-BSC scheme, s_1 generates $\beta_q, \omega_t, \omega_f$ and collects the two sets of records q_c and $\bar{q}_c$ by randomly contacting a few other clients. Similar to the QAA-BSC scheme, the records in q_c and $\bar{q}_c$ are not required to be sent back to s_1 by different clients that have been sampled. Rather, they compute one prime per record, $\mathcal{G}(f_q(r, \beta_q))$, in these two sets and send back only the two products of these primes. To achieve this, s_1 also generates function f_q and $\mathcal{G}$ as described above. The mappings produced by f_q's must also be *different for different queries*.

For a client s that has been contacted by s_1, assume that s is contributing a record r to either q_c or $\bar{q}_c$ to s_1, s generates the prime $\mathcal{G}(f_q(r, \beta_q)) \in \mathbb{P}$. He computes the two products (one for q_c and one for $\bar{q}_c$) of all these primes. He sends back these two products and a single bit per product to indicate whether a product corresponds to records in q_c or $\bar{q}_c$. We denote the set of primes that are produced by records from q_c as P_q and the set of primes that are produced by records from $\bar{q}_c$ as $P_{\bar{q}}$. Let $C_q^t = \prod_{\nu \in P_q} \nu$, and $C_q^f = \prod_{\nu \in P_{\bar{q}}} \nu$, The query challenge C_q for q is then constructed as: $C_q = C_q^t \cdot C_q^f$. Finally, s_1 submits the query q, the random bit sequence β_q, the challenge C_q, and the function f_q to the server R; s_1 also keeps the product C_q^f locally for verification purpose.

Server's execution of the query q. The goal of our challenge is to require the server to factor out C_q^t from C_q, i.e., server needs to produce: $\chi_q = C_q / C_q^t = C_q^f$. Next, we show that this is efficiently achievable for an honest server. Besides the normal execution of the query q, the server maps each record $r \in q_t$ to a prime in $\mathbb{P}$ using the function f_q submitted by the client and the function $\mathcal{G}$ that is available to both the clients and the server. The server R initializes $\chi_q = C_q$, and for every record $r \in q_t$ he does the following, let $\nu_r = \mathcal{G}(f_q(r, \beta_q))$ and $\chi_q = \chi_q / \nu_r$, if $\gcd(\chi_q, \nu_r) = \nu_r$. At the end of the query processing, R returns the query response $\mathcal{A}'_q$ and χ_q. Clearly, for an honest server that has touched all records in q_t, $\chi_q = C_q / C_q^t = C_q^f$ as required. A lazy server that has only touched a subset of records from q_t (or not at all) will not be able to produce the correct evidence $\chi_q = C_q^f$ with a high probability (details in Section 3.3), due to the one-way and collision resistant properties of f_q, the collision-free property of $\mathcal{G}$ and the fact that f_q changes for different queries, which ensure that the server has to read the actual content of a record in order to find its mapping prime in the set $\mathbb{P}$ for a query q.

Client's verification. The verification in the QAA-ADV scheme is extremely simple. An honest server should return a χ_q that equals C_q^f, which only takes $O(1)$ time to check (as s_1 has kept C_q^f). Hence, the verification function $\mathcal{V}(\chi_q) = 1$ if $\chi_q = C_q^f$; and $\mathcal{V}(\chi_q) = 0$ otherwise.

3.2 Constructions of f_q and $\mathcal{G}$

The Function f_q: The function f_q has to be a one-way, collision resistant mapping from records in D to an integer field $[1, \ell]$, for some $\ell \geq N$. A fresh mapping should be used per query, otherwise the server may remember this mapping and obtain the results without accessing records for new queries. This function must have a small description size (since it is sent from the client to the server for every query) and be efficient to compute. More importantly, the size of the output field, determined by the value ℓ, should not be too large. This is because that ℓ determines how many primes $\mathbb{P}$ contains, which in turn decides how large they are and has a significant impact on the efficiency of the scheme.

Consider a simplest implementation of f_q as follows. For each query q, we produce a random permutation for records in D (independently from other queries). For any record r, we simply let $f_q(r, \beta_q)$ output the index (position) of the record r in this permutation. However, this function has an $O(N)$ description size, as it has to remember the random permutation of all records per query.

In fact, f_q should be a one-way, *perfect hash function* and the above implementation is a *minimal perfect hash* where the output consists of consecutive integers. It has been shown that the lower bound on the space usage of the minimal perfect hashing is $O(N)$ for an input set of N elements. This is too expensive to use for our purpose. In our setting, a perfect hash function with a small space usage and the one-way property is sufficient.

This suggests that we can use a general one-way, collision resistant hash function H, and apply H to the concatenation of the record r and the random bit sequence β_q per query q as f_q, i.e., $f_q(r, \beta_q) = H(r \oplus \beta_q)$. However, a serious problem with this approach is that the output domain size is too large to be efficient. A too-big value of ℓ indicates a larger set $\mathbb{P}$ that requires more and larger primes (since the domain of function $\mathcal{G}$ is bound by ℓ and the average gap between prime numbers near n is $O(\ln n)$ on expectation [28]), which inherently introduces more computation and communication overhead to the scheme. General one-way, collision resistant hash functions do not limit the size of their inputs, they often require a large output domain size to be collision free. For example, SHA-1 takes variable-length inputs, and maps them to 20-byte (160 bits) outputs. This means that simply taking $f_q(r, \beta_q) = H(r \oplus \beta_q)$ will be too expensive for our purpose (as ℓ will be 2^{160} if SHA-1 is used).

Since the number of records in the database D is limited and each record has a limited length, we can design better alternatives. Specifically, we assume that each record r in distributed databases has a unique id id_r. This can be established by clients in a pre-processing step for static databases. For dynamic updates, there are many efficient ways to maintain a collision free set of id_r's over distributed data sets and this issue will be discussed in section 4. We also denote the maximum id value for all records as $id_{\max}$, and do not require id_r's for records in D to be strictly consecutive.

Assume that $id_{\max}$ has η bits (expected to be $O(\log N)$), and γ is a small integer (say less than 20) fixed in the scheme. Define the function $e_q : r, \beta_q \rightarrow \{0,1\}^{\eta+\gamma}$ as: $e_q(r, \beta) = id_r \cdot 2^\gamma + c$, where, *at the client-side*, c is determined by:

$$c = H(r \oplus \beta_q) \bmod 2^\gamma, \qquad \text{if } r \in q_c, \quad (3)$$

$$c = \text{a random } \vartheta \in [0, 2^\gamma) \text{ and } \vartheta \neq H(r \oplus \beta_q) \bmod 2^\gamma, \qquad \text{if } r \in \bar{q}_c. \quad (4)$$

Note that for a record $r \in q_t$, the server cannot distinguish if it belongs to q_c or $\bar{q}_c$ or neither of them. Hence, *at the server-side*, c is always determined by equation 3. Basically, e_q shifts the id of r to the left for γ bits, then adds a value of $c \in [0, 2^\gamma)$ to it, where c is $H(r \oplus \beta_q) \bmod 2^\gamma$ if $r \in q_c$, or a random value different from $H(r \oplus \beta_q) \bmod 2^\gamma$ if $r \in \bar{q}_c$. Since records have unique ids and $c < 2^\gamma$, e_q is clearly collision free. The output domain of $e_q(r, \beta_q)$ is bounded by $2^{\eta+\gamma}$.

Let the first prime that is larger than $2^{\eta+\gamma}$ be λ, which can be found efficiently using the Miller-Rabin primality test, as discussed at the beginning of Section 2.1. The prime gap is $\ln \lambda$ on expectation near λ [28], and the Miller-Rabin test has a complexity of $O((\log \lambda)^3)$ for a value λ. Hence, we can find λ in $O((\log \lambda)^4)$ on expectation. We denote λ's *prime field (without* 0*)* as $F_\lambda = \{1, 2, \ldots, \lambda - 2, \lambda - 1\}$.

In number theory, F_λ is a *cyclic group* that can be generated by its primitive root in *some* order [4]. A primitive root for a cyclic group is also referred as a *generator* for the cyclic group. If g is a *generator* for F_λ, then $F_{\lambda,g} = \{g^1 \bmod \lambda, \ldots, g^{\lambda-1} \bmod \lambda\}$ is a permutation of F_λ.

A cyclic group has at least one generator, and usually many in practice [28]. The best deterministic algorithm can find the generators for F_λ in $O(k^2 \log \lambda)$ [4], where k is the number of the prime factors of $\lambda - 1$. In practice, the occurrences of the generators are often dense enough such that a simple probabilistic search procedure can locate a generator much more efficiently [29].

Note that for different queries, the function e_q will be different due to the randomness introduced by β_q in equation 3. However, the output domains of these e_q's are the same, i.e., $[0, 2^{\eta+\gamma})$ (assuming that the number of bits required for $id_{\max}$ does not change, which is reasonable as long as N does not change dramatically, or we can simply use an η value that represents a few more bits than what is required to store the initial $id_{\max}$). This suggests that F_λ is the same for different queries.

Hence, in the QAA-ADV scheme, we only need to find the generators for F_λ once in a pre-processing step, once a γ value has been picked. Suppose these generators are $G_{F_\lambda} = \{g_1, \ldots, g_\kappa\}$ for some value κ. Finally, the function $f_q : r, \beta_q \rightarrow [1, \ell]$ is simply defined as:

$$f_q(r, \beta_q) = g_x^{e_q(r,\beta_q)} \bmod \lambda, \text{ for some } x \in [1, \kappa], \; g_x \in G_{F_\lambda}. \quad (5)$$

For each query q, the client selects a random generator g_x from G_{F_λ} to construct f_q (the bit sequence β_q is also randomly generated per query q). The function f_q in equation 5 is clearly collision resistant, by the property of the generator g_x and the collision free property of e_q. The server may still be able to reverse the function $f_q(r, \beta_q)$ if he is willing to pay considerable computation cost, however, that defeats the purpose for him being lazy. Hence, $f_q(r, \beta_q)$ is also one-way for *a lazy server* by the fact that it is very expensive to solve the discrete logarithm problem for a large prime field [9]. The introduction of γ bits (say 15 bits) appending to the record id in the function e_q is to ensure a large enough prime λ to have sufficient number of generators for its prime field, and to make it expensive to solve the discrete logarithm problem for a lazy server (in order to break the one-way property of the function f_q).

Lastly, the construction of the function f_q in equation 5 also explains that $\ell = \lambda - 1$ in the QAA-ADV scheme, which is expected to be $(\eta + \gamma)$ bits, i.e, $\ell = O(2^{\eta+\gamma})$.

The Function $\mathcal{G}$: The simplest function $\mathcal{G} : i \in [1, \ell] \rightarrow p \in \mathbb{P}$ is to pre-compute ℓ unique primes and maintain the set $\mathbb{P}$ in both the client and the server side. However, this approach is very expensive, since $\ell = 2^{\eta+\gamma}$ ($\ell > N$), indicating that a large number of primes has to be kept in $\mathbb{P}$ *for every client*.

Hence, the best approach is to generate only necessary primes corresponding to records in q_c and $\bar{q}_c$ (required to construct the challenge C_q for a query q) on the fly efficiently, i.e., the set $\mathbb{P}$ is never materialized. This requires a function $\mathcal{G}$ that produces collision free primes on the fly based on different index values outputted by the function f_q. $\mathcal{G}$ must also be deterministic to ensure that given the same index value produced by f_q, both the client and the server will generate the same prime.

If one can efficiently find p_n (recall that p_n is the nth prime) using a polynomial formula, then this task is trivial. However, how to do this is unclear to this date [4]. Existing methods for finding p_n efficiently given n as a parameter typically use the sieve algorithm (or the sieve of Eratosthenes), which is similar to an enumerating approach that starts with p_1 and finds consecutive primes repeatedly till p_n using a primality test. The sieve algorithm also has a nearly linear, high space consumption cost. This approach is obviously too expensive, especially for the client, since it has to be used for every record in q_c or $\bar{q}_c$ for each query.

A tempting choice is to apply the Bertrand-Chebyshev's theorem [15], which says that given a number $n > 1$, there always exists at least one prime between n and $2n$. Hence, we can design the function $\mathcal{G}$ as follows. For an input i, it simply finds the first prime after 2^i using an efficient primality test method. This function is deterministic, efficient and collision free, since there must exist at least one prime between 2^i and 2^{i+1}. However, this approach suffers from generating too large primes (approximately 2^ℓ in the worst case and ℓ is about $2^{\eta+\gamma}$), which significantly degrades QAA-ADV's efficiency as it involves multiplication and division of these primes ($O(|q_t|)$ gcd operations for the server).

We use a similar intuition as above, but require that $\mathcal{G}$ generates primes as dense as possible so that the maximum prime from the ℓ possible outcomes does not become excessively large. To this end, we leverage on well-established, extensively-tested conjecture on primes from the number-theory field.

Since we are willing to work with well-established conjectures. The first natural choice is to use the Legendre's conjecture [15], which states that there exists at least one prime between n^2 and $(n+1)^2$ for all positive integer n. As a result, we can let $\mathcal{G}(n)$ return the first prime that is after n^2. This will ensure that $\mathcal{G}(n)$ generates unique primes and is collision free for all $n \in \mathbb{Z}^+$. However, this may still generate very large primes for $n \in [1, \ell]$, since ℓ is roughly $2^{\eta+\gamma}$ and $(\eta+\gamma)$ could be as big as 64 or even larger. We remedy this problem by leveraging on another well-established conjecture, the *Cramér's conjecture* [8]:

$$p_{n+1} - p_n = O\left((\ln p_n)^2\right) \text{ holds for all } n \in \mathbb{Z}^+. \tag{6}$$

The constant in Cramér's conjecture is upper bounded by 1.13 for all known prime gaps [13], [21]. We first prove the next technical Lemma.

Lemma 1. *Assume the Cramér's conjecture, then for all $n \in \mathbb{Z}^+$ and $n \geq 2$, $p_n - p_{n-1} < 4.52(\ln n)^2$.*

Proof: Firstly, by the well-known upper-bound for the nth prime, $p_n < n \ln n + n \ln \ln n$ for all $n \in \mathbb{Z}^+$ and $n \geq 6$ [4], [15]. Clearly, this indicates that $p_n < (n+1)^2$ for all $n \in \mathbb{Z}^+$ (we can easily verify that this holds for $n = 1, \ldots, 6$). On the other hand, since the constant in the big-O notation in equation 6 is upper bounded by 1.13 for the largest known maximal prime gap to this date [13], [21], we have: $p_n - p_{n-1} \leq 1.13(\ln p_{n-1})^2$, for $n \in \mathbb{Z}^+$ and $n \geq 2$. Combining it and $p_n < (n+1)^2$, we get: $p_n - p_{n-1} \leq 1.13(\ln p_{n-1})^2 < 1.13(\ln(n^2))^2 < 4.52(\ln n)^2$, for $n \in \mathbb{Z}^+$ and $n \geq 2$, which completes the proof. □

Theorem 2. *Assume the Cramér's conjecture, if we define $\mathcal{S}(n)$ as: $\mathcal{S}(n) = 18.08n(\ln n)^2$ for all $n \in \mathbb{Z}^+$, then there exists at least one prime between $\mathcal{S}(n)$ and $\mathcal{S}(n+1)$ for all $n \in \mathbb{Z}^+$. The function $\mathcal{G}(n)$ that returns the first prime that is larger than $\mathcal{S}(n)$ is collision free for all $n \in \mathbb{Z}^+$.*

Proof: We can easily verify the correctness of the theorem for $n \in [1, 812)$ empirically. As such, we only need to show that there exists at least one prime in $(\mathcal{S}(n), \mathcal{S}(n+1)]$ for all $n \in \mathbb{Z}^+$ and $n \geq 812$. We prove this by contradiction. Firstly, obviously for all n:

$$\begin{aligned} \mathcal{S}(n+1) - \mathcal{S}(n) &= 18.08[(n+1)(\ln(n+1))^2 - n(\ln n)^2] \\ &> 18.08(\ln n)^2. \end{aligned} \tag{7}$$

Assume that our claim does not hold, i.e. there exists at least one integer $j \in \mathbb{Z}^+$ and $j \geq 812$, such that there is no prime in $(\mathcal{S}(j), \mathcal{S}(j+1)]$. Let the first prime larger than $\mathcal{S}(j)$ be p_{z+1} for some value z, then the first prime smaller than or equals $\mathcal{S}(j)$ must be p_z. By our assumption, there is no prime in $(\mathcal{S}(j), \mathcal{S}(j+1)]$, hence $p_z \leq \mathcal{S}(j) < \mathcal{S}(j+1) < p_{z+1}$, which implies that (the last step is derived by inequality 7):

$$p_{z+1} - p_z > \mathcal{S}(j+1) - \mathcal{S}(j) > 18.08(\ln j)^2. \tag{8}$$

On the other hand, by Rosser's theorem [15], which says that $p_n > n \ln n$ for all $n \in \mathbb{Z}^+$, we have:

$$(z+1) < z \ln z < p_z \leq \mathcal{S}(j) = 18.08j(\ln j)^2 < j^2. \tag{9}$$

In the above, the first inequality holds for all $z \in \mathbb{Z}^+$ and $z \geq 4$. Since j is at least 812, z cannot be too small, so it trivially holds. The second inequality is based on the Rosser's theorem, and the last inequality is based on the fact that for all $j \geq 812$, $j > 18.08(\ln j)^2$.

Now, applying Lemma 1 on p_{z+1} and p_z, and then the fact that $(z+1) < j^2$ by inequality 9, we have:

$$\begin{aligned} p_{z+1} - p_z &< 4.52[\ln(z+1)]^2 \\ &< 4.52(\ln j^2)^2 = 18.08(\ln j)^2. \end{aligned} \tag{10}$$

We reach a contradiction (Inequality 10 contradicts with inequality 8). This completes the proof. □

Theorem 2 relies on the Cramér's conjecture that is widely-believed to be true and has been tested extensively in the literature. It holds for all known prime gaps [13], [21]. Nevertheless, we empirically verified Theorem 2's correctness. We have tested for n from 1 to more than 1 billion and it always holds. Our result in Theorem 2 is very tight in terms of generating primes that are not far apart from each other, since they cannot be too dense, given that the prime gap is $O(\ln n)$ on expectation near n [28]. This indicates that the best generating function $\mathcal{G}$ one can hope for is based on $n \ln n$, and we have managed to obtain one based on $18.08n(\ln n)^2$.

By Theorem 2, both the client and the server do not need to pre-generate and store the set $\mathbb{P}$ of ℓ unique primes. Instead, in the QAA-ADV scheme, $\mathbb{P}$ is simply a *conceptual set*. For any record r, in order to find its mapping prime $\nu_{f_q(r,\beta_q)}$ in the *conceptual set* $\mathbb{P}$ for a query q, the client (or the server) simply computes $\nu_{f_q(r,\beta_q)} = \mathcal{G}(f_q(r,\beta_q))$ on the fly.

3.3 Theoretical Guarantee and Cost Analysis

The inclusion of C_q^f is necessary, otherwise the server may simply return $\chi_q = 1$ for any query q. Recall that C_q^f is the product of primes corresponding to sampled records from $\bar{q}_c$. For a record in $\bar{q}_c$, it is first randomly mapped to an index value in the range $[id_r \cdot 2^\gamma, id_r \cdot 2^{\gamma+1})$ by the function e_q, which is further mapped to an index value in $[1,\ell]$ by the function f_q, and a prime in $\mathbb{P}$ by the function $\mathcal{G}$. Essentially, for any record in q_t, there are 2^γ number of possible primes in $\mathbb{P}$ that it corresponds to. One of them (determined by $H(r \oplus \beta_q) \bmod 2^\gamma$) is used for the construction of C_q^t if this record is sampled into q_c. To distinguish from the records in q_c (that contribute to C_q^t), a record in $\bar{q}_c$ randomly selects one of its other $2^\gamma - 1$ primes to contribute to the C_q^f.

With the presence of C_q^f, in order to terminate earlier at any point, the server has to decide that the remaining C_q no longer has the contribution from any prime factor in C_q^t. The only way that he can make sure this is to ensure that primes in both C_q^t and C_q^f have been exhausted by records he has accessed, since he can not distinguish C_q^t and C_q^f. This means that for every accessed record r, if r's mapping prime (generated by f_q and $\mathcal{G}$) does not factor C_q (i.e., $r \notin q_c$), the server needs to try to factor C_q with $2^\gamma - 1$ number of other possible mapping primes of r to decide whether r belongs to $\bar{q}_c$ (only if one of them factors C_q^f). The chance that he can terminate earlier without accessing a significant number of records in q_t is very small (since q_c and $\bar{q}_c$ are randomly selected from q_t by the client, this probability is essentially $\Pr[q_c \subseteq q_a \text{ and } \bar{q}_c \subseteq q_a]$). Furthermore, in order to realize this small chance, the server has to generate on expectation half of the $2^\gamma - 1$ possible mapping primes for every record he accessed and performs the $\gcd$ operation for each of them, which is clearly too expensive for a lazy server.

Since that the server cannot factor C_q into a set of individual prime factors easily unless he is willing to pay some significant computation cost (by the hardness of the integer factorization problem [15]), R cannot efficiently find all prime factors of C_q while he is trying to be lazy. Even if he does, by the one-way property of the f_q function, he still cannot figure out which record maps to a particular prime factor. Hence, the only way that R can be sure with 100% probability to return $\chi_q = C_q^f$ is to honestly execute the query, touch every record r in q_t and check if r maps to a prime $\mathcal{G}(f_q(r,\beta_q))$ that can factor C_q. By a similar analysis as that in the QAA-BSC scheme, we can show that:

Theorem 3. QAA-ADV *has the same success probability for a client to catch a lazy server as that of the* QAA-BSC *scheme, in both the single query (as in Theorem 1) and the k-queries (as in Corollary 1) cases.*

Cost Analysis. The client-side communication cost is the same as that in the QAA-BSC scheme. The client-side computation cost is dominated by the computation of the function $f_q(r,\beta_q)$ and the function $\mathcal{G}(f_q(r,\beta_q))$ for sampled records in q_c and $\bar{q}_c$. The cost of $f_q(r,\beta_q)$ consists of the computation of the hash function H and the modular exponentiation of $g^x \bmod \lambda$ where $x \le \lambda - 1$ and $\lambda = O(2^{\eta+\gamma})$. Since the modular exponentiation ($g^x \bmod \lambda$) can be computed in $O(\log x)$ time [4], the cost of the function $f_q(r,\beta_q)$ is $O(\mathcal{C}_H + \log\lambda)$. For the function $\mathcal{G}(i)$ for $i \in [1, \lambda - 1]$, since the prime gap is $\ln\lambda$ on expectation near λ [28], and the Miller-Rabin test has a complexity of $O((\log\lambda)^3)$ for a value λ, its cost is simply $O((\log\lambda)^4)$. The client-side computation cost (the sum of all clients) is $O((|q_c| + |\bar{q}_c|)(\mathcal{C}_H + (\log\lambda)^4))$.

The client-server communication overhead includes the transmission of the challenge C_q, the random bit sequence β_q, the value γ, the generator g and the prime λ used in function f_q, which is dominated by the size of C_q, i.e., $8(|q_c| + |\bar{q}_c|)$ bytes in the worst case, but usually smaller in practice as the product of two 8 bytes numbers may need less than 16 bytes to store. The server-client communication overhead simply consists of χ_q, which is $8|\bar{q}_c|$ bytes in the worst case and usually smaller in practice for the similar reason.

The server-side's query overhead for an honest server is the test to be performed for every record in q_t to see if its mapping prime factorizes C_q or not. This involves the computation of the function $\mathcal{G}(f_q(r,\beta_q))$ and the call to a $\gcd$ procedure with C_q and another smaller prime number. Hence, the overall cost is $O(|q_t|(\mathcal{C}_H + (\log\lambda)^4 + \log C_q))$. The client-side's verification is done in constant time, by checking if $C_q^f = \chi_q$. Finally, there is no storage overhead.

4 OTHER ISSUES

Both schemes support the dynamic updates efficiently. First of all, changing the values of some attributes of an existing record or deleting a record does not require any action for both schemes. To support insertions, in the QAA-BSC scheme, one needs to assign τ distinct primes to a new record at a site s_i such that these τ primes are not currently assigned to any other records in distributed databases. We can achieve this goal by applying a standard technique for maintaining unique ids in distributed sites, and assigning τ distinct ids for one record. For each id of a record, we use the mapping function $\mathcal{G}$ in Section 3.2 to generate a distinct prime. For insertions in the QAA-ADV scheme, as long as a standard technique for maintaining unique ids for distributed sites has been used, it can support the dynamic updates seamlessly. One simple way (definitely not the only way) of maintaining a set of unique ids in distributed sites is to use a coordinator at the client-side. We initially assign unique ids to all records in distributed sites and store the max id $id_{\max}$ in the coordinator. When a record has been deleted, its id will be reported to the coordinator. A new record will simply recycle an id from the coordinator. When the coordinator runs out of available ids, it will return $id_{\max}+1$, and update $id_{\max}$ to $id_{\max}+1$. The coordinator could simply be one of the clients.

To support clients that are not from $\{s_1,\ldots,s_m\}$, we denote such clients as $\{s_{m+1},\ldots,s_{m+k}\}$ for some value k. The only change in our schemes is to request any client s to sample clients only from $\{s_1,\ldots,s_m\}$, when generating the sets of q_c and $\bar{q}_c$. When answering a query q only depends on a subset of attributes A for any record r from q_t, we can apply our schemes by replacing r with its projection on A, i.e., $\pi_A(r)$.

In the QAA-BSC scheme, it is possible to use τ random hash values that are related to $H(r\oplus\beta_q)$ for a record r when it is mapped to τ primes by $H(r\oplus\beta_q)$ in our scheme. This will require some changes to the scheme, but the basic intuition of our design stays the same. The main reason for using primes in our design is to reduce the communication cost and at the same time, keep the scheme simple and efficient. The communication cost is arguably the most critical cost metric in distributed systems and the typical outputs by one-way, collision-resistant hash functions are considerably big, for example, SHA-1 outputs a hash value of 20 bytes (has at least 3 times of communication cost as using the primes). More secure one-way, collision-resistant hash functions require even larger output sizes. On the other hand, since the challenge C_q has been forwarded to the server, a hash-value based approach is not feasible for the QAA-ADV scheme, as the server could terminate earlier much cheaply in this case (since H is cheap to compute).

When $|q_t|=0$, both schemes have to contact all clients to realize that $|q_c|=0$, which does not change the correctness of our schemes. But to avoid contacting all clients in this case and to reduce the number of clients to contact for constructing the challenge in general, clients may deploy some peer-to-peer range queries techniques [5], [30], [33]. A related issue is what happens when some clients are temporarily offline. Since both schemes rely on randomly sampling records (from q_t) to construct the query challenges, this does not pose a limitation unless all clients that possess some records in q_t become offline at the same time, which happens rarely in practice for typical queries and data distributions. To further alleviate this problem, one can produce duplicates of records and store them in some other clients.

In the basic setup of our schemes, data owners need to store their databases locally. However, since our schemes were built using random samples of records in the databases, this assumption can be removed in practice. Without compromising the soundness of the verification, the data owners can store random samples of records from their databases instead of maintaining the whole databases locally. The drawback of this approach is that this will increase the chance that q_t might become empty w.r.t. the sampled databases.

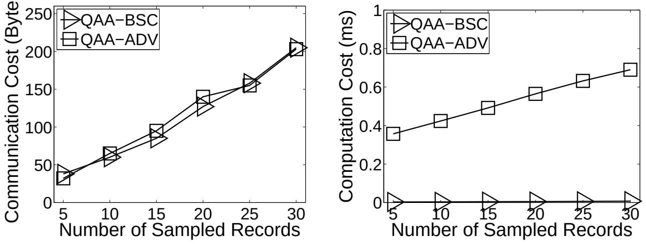
Lastly, we would like to point out that both schemes still allow the server to utilize the cached records in the main memory buffers to answer subsequent queries, if they have been accessed by some precedent queries and kept in the memory buffer at the server side. The key observation is that the server only needs to prove that he has honestly accessed the records in q_t, it does not matter if he accesses them from a memory buffer or the disk. In both schemes, this proof is done by constructing the respective evidence χ_q, which only depends on the content of the records, not where the records come from.

5 EXPERIMENT

We implemented both schemes in C++. The GMP library was used for big numbers (more than 64 bits). The NTL library was used for some number theory functionality. We tested both schemes over queries that have d-dimensional selection predicates (records may have other, additional attributes); d is also referred to as the query's dimensionality. The server utilizes a multi-dimensional indexing structure, specifically, the R-tree, to answer queries. All experiments were executed on a 64-bit Linux machine with a 2GHz Intel Xeon(R) CPU.

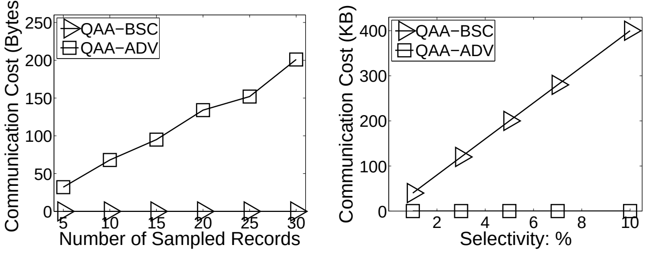
Data sets. Real data sets from the open street map project (http://www.openstreetmap.org/) were used for $d=2$. Specifically, we use the road network of California (*CA*), which contains 12 million points. To experiment with different sizes, we randomly sample the necessary number of points from *CA*. For larger d values, we generate synthetic values for $(d-2)$ attributes with the randomly clustered distributions, and append them to *CA*. We assigned a unique id for each point and treated each point as a record. In addition to the d attributes in the data set, each record is made as 200 bytes by pending some random bytes, to simulate the scenario where a record may have many other different attributes. The server uses the R-tree to index the d attributes for a data set. The default page size is 4 KB and the index utilization is 0.7. Finally, note that the costs of our schemes are determined by the size of the database and the query selectivity. Hence, using different datasets with various distributions and characteristics will result in similar results.

Default setup. Unless otherwise specified, we assume an honest server to reflect the overhead of our schemes. The default parameters are as follows. In both schemes, $|q_c|=20$, $|\bar{q}_c|=10$, and query selectivity $\rho_q=5\%$. In the QAA-ADV scheme, we also set the default value for γ as $\gamma=15$. In all experiments, 100 queries were generated independently at random and we report the average cost per query. The default query dimensionality is $d=3$ and the default data set size is $N=0.5\times10^6$ (half million). The default query type is the range selection query. In all communication costs, we do not include the part that is contributed by the normal query execution, i.e., the description of the query itself and the query results.



(a) Communication cost. (b) Computation cost.

Fig. 4. Client-side challenge construction cost.

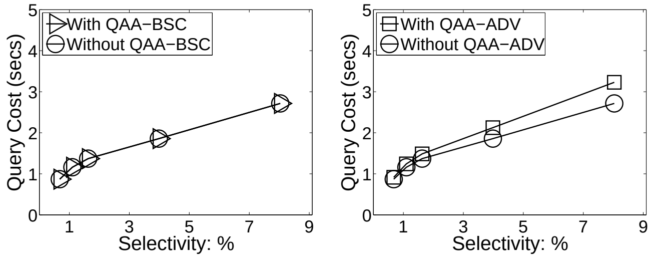


(a) Client to server. (b) Server to client.

Fig. 5. Client-Server communication costs.

Client-Side computation and communication costs. The first step in QAA is to prepare the challenge for a query q at the client side. This incurs both computation and communication costs for clients. We measure both costs in terms of the total costs for all clients, when one client is constructing a challenge C_q for a query q. Figure 4 shows the results for the average costs of one query. Clearly, both the communication and computation costs are determined by the number of sampled records for constructing the challenges in both schemes, i.e., $|q_c| + |\bar{q}_c|$. In both schemes, the communication cost is $8(|q_c|+|\bar{q}_c|)$ bytes in the worst case, since each contacted client only sends back the two products for the primes generated. This trends is clearly indicated in Figure 4(a). Both schemes have similar client-side communication costs when $|q_c| + |\bar{q}_c|$ is the same, and they are less than 240 bytes for $|q_c| + |\bar{q}_c| = 30$. In terms of the overall client-side computation cost, both schemes have costs that are linear to the number of sampled records as shown in Figure 4(b). The QAA-ADV scheme is more expensive since it needs to compute the function $f_q(r, \beta_q)$ and the function $\mathcal{G}(f_q(r, \beta_q))$ to generate a unique prime. However, even in this case, it only takes approximately 0.7 milliseconds (total computation costs of all clients, this cost is distributed among different clients) for 30 sampled records. The QAA-BSC scheme is very cheap in this respect and generates almost no computation overhead for clients to prepare the challenge. Finally, these costs are almost constants w.r.t. the dimensionality of the query and the query selectivity.

Client-Server communication costs. Figure 5(a) shows the communication cost from a client to the server when the *number of sampled records* ($|q_c| + |\bar{q}_c|$) in the construction of the challenge C_q changes. QAA-BSC has almost no communication cost from the client to the server, other than sending the random bit sequence β_q which is tiny in size; QAA-ADV needs to send the challenge C_q to the server and the size of C_q increases linearly w.r.t. the number of sampled records. Nevertheless, it is a very small cost (less than 240 bytes) since the necessary number of sampled records for our schemes is small. This cost remains as a constant for different query dimensionality and query selectivity.



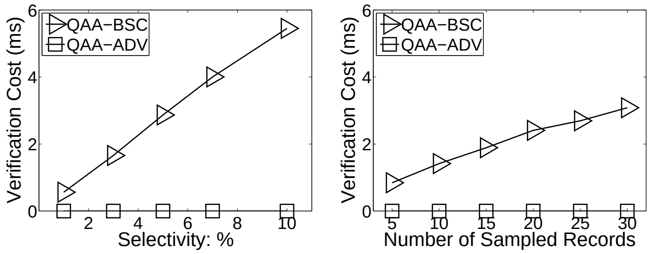
(a) QAA-BSC. (b) QAA-ADV.

Fig. 6. Server-side query cost vs. ρ_q.

Figure 5(b) shows the communication cost from the server to the client after the query has been executed when we vary the *query selectivity* ρ_q. It clearly shows that the communication cost in the QAA-BSC scheme grows linearly w.r.t. the increase of ρ_q, since it has to return all primes corresponding to the records in q_t. For $\rho_q = 10\%$, it reaches almost 400KB. On the other hand, the communication overhead incurred by QAA-ADV is very small (less than 100 bytes in this case) and does not change at all w.r.t. ρ_q because the size of the evidence χ_q solely depends on the number of primes in C_q^f that the client has adopted in C_q. For both schemes, this cost is a constant for queries of different dimensionality.

Server-Side execution cost. Next, we study the query execution overhead on the server side with the presence of QAA. We focus on the impact of query selectivity ρ_q in these experiments. The results were reported in Figure 6. We study two schemes separately, and report the query cost with or without QAA in each scheme. Clearly, for both schemes, the query costs increase linearly w.r.t. the query selectivity ρ_q in all cases. In Figure 6(a), it shows that QAA-BSC almost does not incur any overhead in the normal query execution, since the server only needs to compute a simple, collision resistant hash function H for every record in q_t and select the corresponding prime per such record. On the other hand, since QAA-ADV needs to compute functions $f_q(r, \beta_q)$ and $\mathcal{G}(f_q(r, \beta_q))$, that are much more expensive than the simple hash function H, for every record in q_t. We did observe some overhead to the normal query execution cost in Figure 6(b). However, this overhead is very small. For example, when $\rho_q = 8.5\%$, the normal query execution cost is about 2.9 seconds; and the overall query cost with the QAA-ADV scheme is only about 3.1 seconds.

Client-Side verification cost. The verification cost of the QAA-ADV scheme is to simply check if $\chi_q = C_q^f$, which is very small and almost a constant for all parameters. This cost in QAA-BSC is affected by both the query selectivity ρ_q and the number of sampled records $|q_c|+|\bar{q}_c|$, since the



(a) Varying ρ_q.

(b) Varying $|q_c| + |\bar{q}_c|$.

Fig. 7. Client-side verification cost.

client needs to iterate through all primes corresponding to records in q_a (sent back from the server, and it equals q_t for an honest server) and test if a prime divides C_q^t or C_q^f respectively. These results were plotted in Figure 7. The cost on checking χ_q in QAA-BSC builds up almost linearly as either the query selectivity (Figure 7(a)), or the number of sampled records increases (Figure 7(b)). Clearly, this cost is very small when ρ_q is small and increases to less than 6 milliseconds when $\rho_q = 10\%$ in the QAA-BSC scheme. This cost is negligible in the QAA-ADV scheme in all cases.

Impact of the query dimensionality d**.** Among all cost measures, the query dimensionality mainly affects the server's query cost. This is studied in Figure 8, by testing it without and with QAA using the two schemes respectively. Figures 8(a) and 8(c) report these results in wall clock time (seconds). Clearly, in both cases the server's query cost increases as d increases; and for similar reasons in experiments in Figure 6, QAA-BSC introduces almost no overhead and QAA-ADV presents small overhead. To demonstrate these trends, we plot the overheads by these two schemes in percentage to the normal query execution cost when we vary d in Figures 8(b) and 8(d). Clearly, the overhead for the server's query cost in the QAA-BSC scheme is about 1% when $d = 2$ and drops to near 0% when $d = 6$; it is 12% for QAA-ADV when $d = 2$ and drops to below 5% when $d = 6$. This indicated that the query execution cost increases as d increases, but the overheads in QAA for both schemes are not significantly affected by the value of d.

Scalability w.r.t. N**.** We test the scalability of the two QAA schemes w.r.t. the database size, by varying N from 0.1 million to 1 million. Among all cost measures, besides the server's query cost, only the server-to-client communication cost and the client's verification cost in the QAA-BSC scheme will be affected. These results were reported in Figure 9. Clearly, both the server-to-client communication cost and the client's verification cost for the QAA-BSC scheme increase linearly as the database contains more records, as shown in Figures 9(a) and 9(b), since these two costs are affected by the number records in q_t in the QAA-BSC scheme. In contrast, the corresponding costs in QAA-ADV are not affected by N. The query costs on the server side are expected to increase linearly as N increases, as shown in Figures 9(c) and 9(d). Both schemes exhibit similar trends as before: QAA-BSC has almost no impact to the query cost, and QAA-ADV has a small overhead to the query cost.

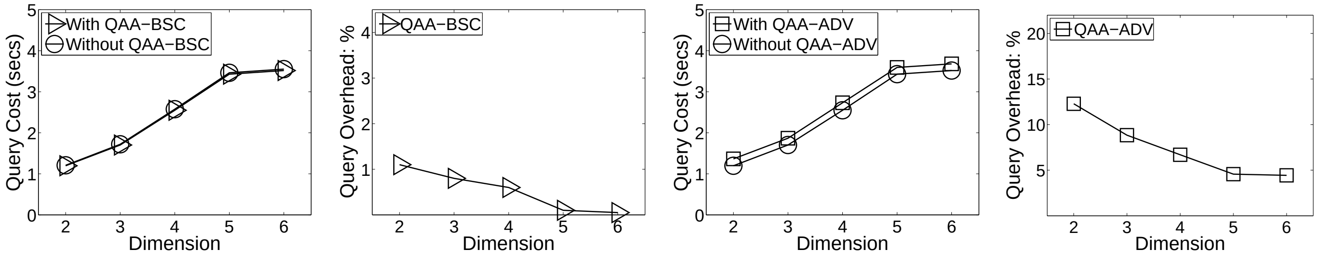
Summary of the results and comparison to other techniques. We also tested the effect of different γ values (used in the function f_q) to the computation cost in the QAA-ADV scheme, and observed that the impact is not significant for small changes in γ values, say γ values differ within 10. We have also simulated a lazy server by dropping some records and tested the effectiveness of our schemes in catching a lazy server. The success probability is almost identical to the theoretical results shown in Figure 3 for both schemes. With $|q_c|$ as small as 20, a lazy server has to access more than 90% of required records to have a reasonable chance of escaping the verification for a single query. It is almost impossible for him to escape after a few queries from the same client. For brevity, we omit these figures.

These findings clearly indicate that both of our schemes are efficient and effective for defeating a lazy server in distributed databases for IO-bound queries. However, if the query selectivity ρ_q is large or the number of records N in the database is huge, the QAA-BSC scheme may incur high costs for the server-to-client communication cost and the client's verification cost. In such scenarios, the QAA-ADV scheme should be applied, which does have a slightly higher overhead on the server's query execution cost than the QAA-BSC scheme, but such overhead is still very small relatively to the normal query execution cost itself.

There are two existing work that have addressed the query authentication problem with multiple owners, w.r.t. malicious (and lazy) servers [22], [26], that may be adapted to solve the QAA problem. The work from [26] assumes that the authentication is provided by a trusted third party, which is different from the model we used.

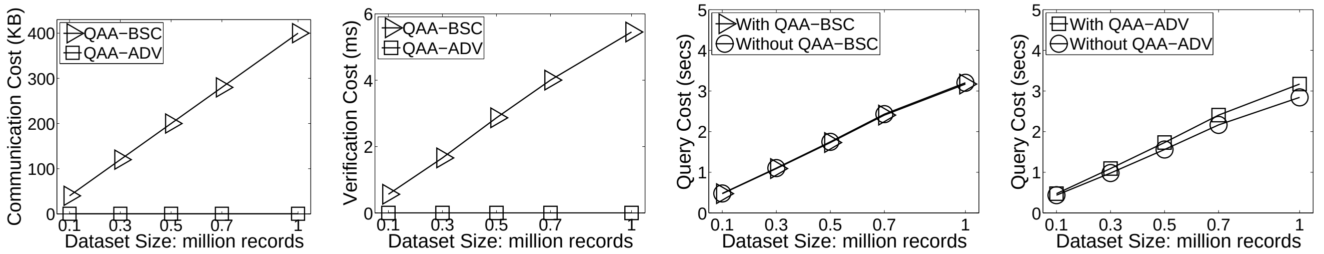
In [22], the authors presented theoretical results for certifying data from multiple data sources. This technique can be employed for query access assurance as well, by treating each query as a range selection query (using the current query's selection predicate) and authenticate the query results from this range selection query. However, in order to authenticate multi-dimensional data ($d > 2$) from multiple data sources, the technique in [22] relies on the *multi-dimensional range tree*, which is not a practical structure for large datasets. Furthermore, this technique also presents significant query and verification overheads when being used to ensure query access assurance.

Specifically, the technique in [22] requires a setup phase. In the setup phase, each client needs to certify that his records have been correctly stored on the server's database, incurring $O(N \log^d N)$ IOs on the server side and a total of $O(\mathcal{C}_H N \log^d N)$ verification cost for clients. There is also a communication cost of $O(|H| N \log^d N)$ bytes between the server and the clients. After the setup phase, for one query execution, the server has an IO overhead of $O(\log^d N + |q_t| \cdot |R|/B)$ IOs to prove QAA to



(a) QAA-BSC. (b) QAA-BSC percentage. (c) QAA-ADV. (d) QAA-ADV percentage.

Fig. 8. Server-side query cost vs. different query dimensionality d.



(a) Server to client communication cost. (b) Client's verification cost. (c) Server's query cost: QAA-BSC. (d) Server's query cost: QAA-ADV.

Fig. 9. Scalability by varying N.

the client (B is the page size and $|R|$ is the record size). It also incurs a communication cost of $O(|H| \cdot \log^d N + |q_t| \cdot |R|)$ bytes. On the client side, the verification cost is $O(\mathcal{C}_H \cdot (\log^d N - \log^d |q_t| + |q_t| \log^{d-1} |q_t|))$ to authenticate the query results (so that the client can ensure that all records in the query range, i.e., q_t, has been accessed). Finally, for any update in any client site, this technique requires the system to re-run the setup phase.

6 Related Work

The most related study is the query execution assurance for a single data owner with queries submitted in batches [32], which has been discussed in details in Section 1.

Another line of work concerns query authentication in the outsourced database (ODB) model [16], [18], [19], [23]–[26], [36], [38], where the goal is to defeat a malicious server. Any technique that can defeat a malicious server defeats a lazy server automatically. However, they require more efforts and generate more overheads than methods that are tailored to lazy servers, as shown in [32]. Nonetheless, they are related to our study and we next present a brief overview of these works.

Most existing query authentication techniques focused on the general selection and projection queries and they can be categorized into two groups, the aggregated signature based approaches [19], [20], [23], [25] and the merkle hash tree (MHT) embedded into various indexes approaches [3], [10], [16]–[18], [24], [26], [31], [39]. In addition, injecting random records by the data owner into the database has also been proposed [36]. This work uses a probabilistic approach for query authentication and hence it is more flexible and easier to realize in practice. However, unlike other work in query authentication, it does not guarantee absolute correctness.

Query authentication in multi-dimensional spaces have been addressed by extending the signature-based and the index-based approaches [7], [40]. Recent studies have also addressed other challenging problems in query authentication, such as join queries [25], [38], dealing with XML documents [6] and text data [24], handling dynamic updates [25], [37] efficiently. Improvement to existing techniques were also proposed, such as separating the authentication from the query execution when there is a trusted third party [26], or partially materializing the authenticated data structures to reduce the cost [18]. The index-based approach has also been extended to the PostgreSQL database [31].

Most studies for the query authentication problem assume one data owner, hence they are not applicable in distributed databases, except [22], [26]. Nuckolls et al. [22] proposed a protocol using the MHT, which requires each data owner to verify that every single record in his dataset has been correctly stored in the server's database, a very expensive step. Only range selection queries were supported and dynamic updates were not addressed. Papadopoulos et al. [26] also addressed the multiple data owners issue for the query authentication problem, but assuming that the authentication is provided by a trusted third party. Nonetheless, these techniques are designed for malicious servers, hence, they are overkill for the lazy server setting studied in this paper.

There has been some work on main memory authenticated structures for defeating malicious servers. However, these works [2], [11], [12], [27], [34], [35] focus on centralized, main memory structures and are not applicable to external memory, distributed databases.

7 Conclusion

This work examines the important problem of defeating a lazy server for distributed databases under the database as a service model [14]. We propose the query access assurance for IO-bound queries in the distributed and singe-query setting, and design two efficient and effective schemes, each with its own distinct advantages, that have achieved this goal with high success probabilities. A challenging future work is to design techniques that guarantee the more general concept in dealing with lazy servers in distributed databases, the query execution assurance, which could be applied to both IO-bound and CPU-bound queries.

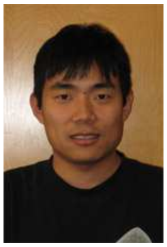
8 Acknowledgment

Wangchao Le and Feifei Li were partially supported by NSF Grant CNS-0831278.

References

[1] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Ann. of Math. (2)*, 160(2):781–793, 2004.
[2] A. Anagnostopoulos, M. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In *ISC*, 2001.
[3] M. J. Atallah, Y. Cho, and A. Kundu. Efficient data authentication in an environment of untrusted third-party distributors. In *ICDE*, 2008.
[4] E. Bach and J. Shallit. *Algorithmic Number Theory*. The MIT Press, Massachusetts, US, 1996.
[5] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in p2p systems. *CACM*, 46(2):43–48, 2003.
[6] E. Bertino, B. Carminati, E. Ferrari, B. Thuraisingham, and A. Gupta. Selective and authentic third-party distribution of XML documents. *TKDE*, 16(10):1263–1278, 2004.
[7] W. Cheng, H. Pang, and K. Tan. Authenticating multi-dimensional query results in data publishing. In *DBSec*, 2006.
[8] H. Cramér. On the order of magnitude of the difference between consecutive prime numbers. *Acta Arithmetica*, 2:23–46, 1936.
[9] R. Crandall and C. Pomerance. *Prime Numbers - A Computational Perspective*. Springer-Verlag, NY, US, 2000.
[10] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic third-party data publication. In *IFIP Workshop on Database Security (DBSec)*, pages 101–112, 2000.
[11] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Super-efficient verification of dynamic outsourced databases. In *CT-RSA*, 2008.
[12] M. T. Goodrich, R. Tamassia, N. Triandopoulos, and R. Cohen. Authenticated data structures for graph and geometric searching. In *CT-RSA*, 2003.
[13] A. Granville. Harald Cramér and the distribution of prime numbers. *Scandinavian Actuarial Journal*, 1:12–28, 1995.
[14] H. Hacigümüs, B. R. Iyer, and S. Mehrotra. Providing database as a service. In *ICDE*, 2002.
[15] G. J. O. Jameson. *The Prime Number Theorem*. London Mathematical Society Student Texts (No. 53). Cambridge University Press, Cambridge, UK, 2003.
[16] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, 2006.
[17] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.
[18] K. Mouratidis, D. Sacharidis, and H. Pang. Partially materialized digest scheme: an efficient verification method for outsourced databases. *The VLDB Journal*, 18(1):363–381, 2009.
[19] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *NDSS*, 2004.
[20] M. Narasimha and G. Tsudik. DSAC: Integrity of outsourced databases with signature aggregation and chaining. In *CIKM*, 2005.
[21] T. R. Nicely. New maximal prime gaps and first occurrences. *Math. Comput.*, 68(227):1311–1315, 1999.
[22] G. Nuckolls, C. Martel, and S. Stubblebine. Certifying data from multiple sources. In *ACM Electronic commerce*, 2003.
[23] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD*, 2005.
[24] H. Pang and K. Mouratidis. Authenticating the query results of text search engines. *PVLDB*, 1(1):126–137, 2008.
[25] H. Pang, J. Zhang, and K. Mouratidis. Scalable verification for outsourced dynamic databases. *PVLDB*, 2(1):802–813, 2009.
[26] S. Papadopoulos, D. Papadias, W. Cheng, and K.-L. Tan. Separating authentication from query execution in outsourced databases. In *ICDE*, 2009.
[27] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *CCS: Proceedings of the 15th ACM conference on Computer and communications security*, 2008.
[28] M. R. Schroeder. *Number Theory in Science and Communication (2nd Enlarged Edition)*. Information Sciences. Springer-Verlag, Berlin, German, 1990.
[29] V. Shoup. Searching for primitive roots in finite fields. *Mathematics of Computation*, 58(197):369–380, 1992.
[30] Y. Shu, B. C. Ooi, K.-L. Tan, and A. Zhou. Supporting multi-dimensional range queries in peer-to-peer systems. In *IEEE International Conference on Peer-to-Peer Computing*, 2005.
[31] S. Singh and S. Prabhakar. Ensuring correctness over untrusted private database. In *EDBT*, 2008.
[32] R. Sion. Query execution assurance for outsourced databases. In *VLDB*, 2005.
[33] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.
[34] R. Tamassia and N. Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. In *ICALP*, 2005.
[35] R. Tamassia and N. Triandopoulos. Efficient content authentication in peer-to-peer networks. In *ACNS*, 2007.
[36] M. Xie, H. Wang, J. Yin, and X. Meng. Integrity auditing of outsourced data. In *VLDB*, 2007.
[37] M. Xie, H. Wang, J. Yin, and X. Meng. Providing freshness guarantees for outsourced databases. In *EDBT*, 2008.
[38] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *SIGMOD*, 2009.
[39] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios. Spatial outsoucing for location-based services. In *ICDE*, 2008.
[40] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios. Authenticated indexing for outsourced spatial databases. *The VLDB Journal*, 18(3):631–648, 2009.

Wangchao Le received the B.E. degree from Guangdong University of Technology in 2004 and the M.E. degree from South China University of Technology in 2007, both in computer engineering. He has been a PhD student in the Computer Science Department, Florida State University, since 2007. His research interests include databases and data management, in particular, temporal data, RDF and security issues.



Feifei Li received the BS degree in computer engineering from Nanyang Technological University in 2002 and the PhD degree in computer science from Boston University in 2007. He has been an assistant professor in the Computer Science Department, Florida State University, since 2007. His research interests include data management, data structures, and databases, as well as security issues in data management.