Authenticating Multi-Dimensional Query Results in Data Publishing

Weiwei Cheng[1], HweeHwa Pang[2], and Kian-Lee Tan[1]

[1]Department of Computer Science, National University of Singapore
[2]School of Information Systems, Singapore Management University

Abstract. In data publishing, the owner delegates the role of satisfying user queries to a third-party publisher. As the publisher may be untrusted or susceptible to attacks, it could produce incorrect query results. This paper introduces a mechanism for users to verify that their query answers on a *multi-dimensional dataset* are correct, in the sense of being complete (i.e., no qualifying data points are omitted) and authentic (i.e., all the result values originated from the owner). Our approach is to add authentication information into a spatial data structure, by constructing certified chains on the points within each partition, as well as on all the partitions in the data space. Given a query, we generate proof that every data point within those intervals of the certified chains that overlap the query window either is returned as a result value, or fails to meet some query condition. We study two instantiations of the approach: Verifiable KD-tree (VKDtree) that is based on space partitioning, and Verifiable R-tree (VRtree) that is based on data partitioning. The schemes are evaluated on window queries, and results show that VRtree is highly precise, meaning that few data points outside of a query result are disclosed in the course of proving its correctness.

1 Introduction

In data publishing, a data owner delegates the role of satisfying user queries to a third-party publisher [6, 10]. This model is applicable to a wide range of computing platforms, including database caching [8], content delivery network [23], edge computing [9], P2P databases [7], etc.

The data publishing model offers a number of advantages over conventional client-server architecture where the owner also undertakes the processing of user queries. By pushing application logic and data processing from the owner's data center out to multiple publisher servers situated near user clusters, network latency can be reduced. Adding publisher servers is also likely to be a cheaper way to achieve scalability than fortifying the owner's data center and provisioning more network bandwidth for every user. Finally, the data publishing model removes the single point of failure in the owner's data center, hence reducing the database's susceptibility to denial of service attacks and improving service availability.

However, since the publishers are outside of the administrative domain of the data owner, and in fact may reside on poorly secured platforms, the query

results that they generate cannot be accepted at face value, especially where they are used as basis for critical decisions. Instead, there must be provisions for the user to check the "correctness" of a query answer.
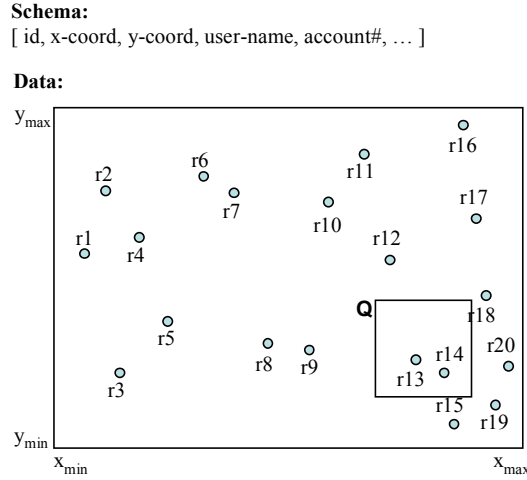
Schema:
[id, x-coord, y-coord, user-name, account#, ...]

Data:

Fig. 1. Running Example

Consider a dataset containing 20 data points in two-dimensional space as shown in Figure 1. The figure also includes a window query **Q**, for which $\{r13, r14\}$ is the correct result. A rogue publisher may return a wrong result $\{r13, r14, r100\}$, which includes a spurious point $r100$, or $\{r13^*, r14\}$ in which some attribute values of $r13$ have been tampered with. To detect such incorrect values, the user should be able to verify the *authenticity* of query result. A different threat is that the publisher may omit some result points, for example by returning only $\{r13\}$ for query **Q**. This threat relates to the *completeness* of query result.

Most of the existing works provide for checking the authenticity [11, 16] and completeness [6, 15] of query results on *one-dimensional* datasets. The exception is Devanbu's scheme [6] which handles multiple key attributes by essentially concatenating them in some preferred order $key_1|key_2|..|key_d$. However, the scheme is expected to be very inefficient for symmetric queries, such as window and nearest neighbor queries, that are typical in multi-dimensional context.

In this paper, we propose a mechanism for users to verify that their query results on a *multi-dimensional dataset* are authentic and complete. Our approach is to build authentication information into a spatial data structure, by constructing certified chains on the points within each partition, as well as on all the partitions in the data space. We introduce two schemes based on this approach. The first, the Verifiable KD-tree (VKDtree), is based on the space partitioning k-d tree. The second, the Verifiable R-tree (VRtree), employs data partitioning

and is based on the R-tree. The schemes are evaluated on window queries, and results show that VRtree is highly precise, meaning that few data points outside of a query result are disclosed in the course of proving its correctness. Moreover, both schemes are computationally secure, and incur low processing and update overheads. To the best of our knowledge, the authentication mechanism introduced in this paper is the first that enables a user to verify the *completeness* of a *multi-dimensional* query result generated by an untrusted server.

The remainder of this paper is organized as follows. Section 2 provides the background on data publishing model and the associated threats, and describes some cryptographic primitives. Our authentication schemes are introduced in Sections 3 and 4, while Section 5 presents results from a performance study. Finally, Section 6 concludes the paper.

2 Background

In this section, we first present the target system deployment model and the associated security threats. Next, we define some cryptographic primitives that are used in our solution.

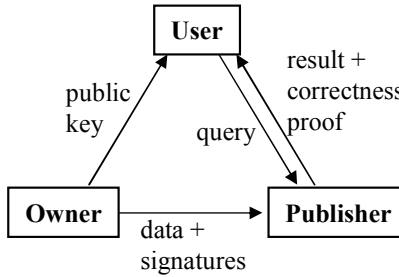### 2.1 System and Threat Models

Fig. 2. Data Publishing Model

Figure 2 depicts the data publishing model, which supports three distinct roles:

– The data owner maintains a master database, and distributes it with one or more associated signatures that prove the authenticity of the database. Any data that has a matching signature is accepted by the user to be trustworthy.
– The publisher hosts the database, and executes queries on behalf of the owner. There could be several publisher servers that are situated at the edge of the network, near the user applications. The publisher is not required to be trusted, so the query results that it generates must be accompanied by

some "correctness proof", derived from the database and signatures issued by the owner.

– The user issues queries to the publisher explicitly, or else gets redirected to the publisher, e.g. by the owner or a directory service. To verify the signatures in the query results, the user obtains the public key of the owner through an authenticated channel, such as a public key certificate issued by a certificate authority.

Our primary concern addressed in this paper is the threat that a dishonest publisher may return incorrect query results to the users, whether intentionally or under the influence of an adversary. An adversary who is cognizant of the data organization in the publisher server may make logical alterations to the data, thus inducing incorrect query results. Even if the data organization is hidden, for example through data encryption or steganographic schemes (e.g., [17]), the adversary may still sabotage the database by overwriting physical pages within the storage volume. In addition, a compromised publisher server could be made to return incomplete query results by withholding data intentionally. Therefore mechanisms for users to verify the completeness as well as authenticity of their query results are essential for the data publishing model. While there are several other security considerations in the data publishing model (such as privacy, user authentication and access control), these have been studied extensively (e.g. [1], [17], [12], [22]), and are orthogonal to our work here.

2.2 Cryptographic Primitives

Our proposed solution and many of the related work are based on the following cryptographic primitives:

One-way hash function: A one-way hash function, denoted as $h(.)$, is a hash function that works in one direction: it is easy to compute a fixed-length digest $h(m)$ from a variable-length pre-image m; however, it is hard to find a pre-image that hashes to a given hash value. Examples include MD5 [18] and SHA [3]. We will use the terms hash, hash value and digest interchangeably.

Digital signature: A digital signature algorithm is a cryptographic tool for authenticating the integrity and origin of a signed message. In the algorithm, the signer uses a private key to generate digital signatures on messages, while a corresponding public key is used by anyone to verify the signatures. RSA [19] and DSA [2] are two commonly-used signature algorithms.

Signature aggregation: As introduced in [5], this is a multi-signer scheme that aggregates signatures generated by distinct signers on different messages into one signature. Signing a message m involves computing the message hash $h(m)$ and then the signature on the hash value. To aggregate t signatures, one simply multiplies the individual signatures, so the aggregated signature has the same size as each individual signature. Verification of an aggregated signature involves computing the product of all message hashes and then matching with the aggregated signature.

Signature chain: In [15], a signature chain scheme is proposed that enables clients to verify the completeness of answers of range queries. A very nice property of the scheme is that only result values are returned, thus ensuring that there is no violation of access control. The scheme is based on two concepts: (a) The signature of a record is derived from its own digest as well as its left and right neighbors'. In this way, an attempt to drop any value from the answer of a range query will be detected since it would no longer be possible to derive the correct signature for the record that depends on the dropped value. (b) For the boundaries of the answer, a collaborative scheme that involves both the publisher and the client is proposed – the publisher performs partial computation based on but not revealing the two records bounding the answer and the query range, while the client completes the computation based on the two end points of the query range.

3   Signature Chain in Multi-Dimensional Space

The goal of our work is to devise a solution for checking the correctness of query answers on multi-dimensional datasets. The design objectives include:

– Completeness: The user can verify that all the data points that satisfy a window query are included in the answer.
– Authenticity: The user can check that all the values in a query answer originated from the data owner. They have not been tampered with, nor have spurious data points been introduced.
– Precision: Proving the correctness of a query answer entails minimal disclosure of data points that lie beyond the query window. We define precision as the ratio of the number of data points within the query window, to the number of data points returned to the user.
– Security: It is computationally infeasible for the publisher to cheat by generating a valid proof for an incorrect query answer.
– Efficiency: The procedure for the publisher to generate the proof for a query answer has polynomial complexity. Likewise the procedure for the user to check the proof has polynomial complexity.

Without loss of generality, we assume that the data in the multi-dimensional space are split into partitions – this can be done using a spatial data structure. To ensure that the answer for a window query is complete, two issues must be addressed. First, we need to prove that the answer covers all the partitions that overlap the query window. We refer to these partitions as candidate partitions. Second, we need to prove that all qualifying values within each candidate partition are returned. The first issue is dependent on the partitioning strategy adopted, and is deferred to Section 4. In the rest of this section, we shall focus on the second issue.

Assuming we have proven that the query answer covers all the candidate partitions, we now need to ensure that all the qualifying values in those partitions have not been dropped. Consider a candidate partition P for the window query

$Q = [(q_{l1}, q_{l2}, \ldots, q_{ld}), (q_{u1}, q_{u2}, \ldots, q_{ud})]$. There are three possible cases: (a) Q contains P. Since the window query bounds the partition, we need to ensure that *all* the points in P are returned. (b) P contains Q. The query window is within the space covered by the partition. A naive solution is to return all the points in P. A better solution, which we advocate, is to return only those points that are necessary for users to check for completeness. In both cases, our concern is to ensure the secrecy of points that are outside Q. (c) P overlaps Q. This case can be handled by splitting P into two parts: the part of P that contains Q, and the part of P that does not overlap Q. The former is handled in case (b), while nothing needs to be done for the latter. Thus, we shall focus on cases (a) and (b), and not discuss case (c) any further.

Our solution extends the signature chain concept in [15] to multi-dimensional space. This is done by ordering the points within the partition, and then constructing the signature chain. In this paper, we adopt a simple scheme of ordering the points based on increasing $(x_1, x_2, \ldots, x_d)$ value. In 2-d space, (x_1, y_1) is ordered before (x_2, y_2) if $x_1 < x_2$, or $x_1 = x_2$ and $y_1 < y_2$. Based on this ordering, we need to return all the points whose first dimension is within the range $[q_{l1}, q_{u1}]$, as well as the bounding points. Of course, some of these points may fall beyond the query window along the second dimension. *For such points that should not be part of the answer, we return only their digests rather than the actual values*, in order to protect their secrecy and achieve high precision.

We choose this simple ordering scheme over more sophisticated space filling curves [20] because: (a) A partition (corresponding to a 4K or 8K block/page) typically consists of a small number of points (100-200). Moreover, the actual number of points within a partition would be smaller than the maximum capacity (since the page is typically not full). As such, it may not be worthwhile to employ a complicated scheme. (b) None of the existing space filling curves perform well in all cases. Thus, they really offer no significant advantage over the simple scheme (especially given the small number of points).

For the example in figure 1, assuming that the entire space corresponds to one partition, the points would be ordered from r_1 to r_{20}. For case (a) where the query bounds the partition, r_1 to r_{20} would be returned; for case (b) where the query (i.e., the box that bounds r_{13} and r_{14}) is within the partition, we return the values of r_{13} and r_{14} and the digest of the various dimensions for r_{11}, r_{12}, r_{15}, r_{16} and r_{17}. We now present the details of our solution that extends the signature chain scheme to multi-dimensional setting.

Construction: Let $L = (L_1, L_2, \ldots, L_d)$ and $U = (U_1, U_2, \ldots, U_d)$ be two points that bound the entire data space, where $L_r \leq U_r$ for all r. L and U are known to all users. Consider a partition P bounded by two points $p_0 = (x_{01}, x_{02}, \ldots, x_{0d})$ and $p_{k+1} = (x_{(k+1),1}, x_{(k+1),2}, \ldots, x_{(k+1),d})$ where $x_{0r} \leq x_{(k+1),r}$ for all r. Suppose P contains k data points $p_1 = (x_{11}, x_{12}, \ldots, x_{1d}), \ldots p_k = (x_{k1}, x_{k2}, \ldots, x_{kd})$. Without loss of generality, we assume that p_i is ordered before p_j for $1 \leq i < j \leq k$. Clearly, p_0 is ordered before p_1 and p_{k+1} is ordered after p_k.

Our multi-dimensional signature chain constructs for each point within P an associated signature (based on [15]):

$$sig(p_i) = s(h(g(p_{i-1})|g(p_i)|g(p_{i+1}))) \tag{1}$$

where s is a signature function using the owner's private key, h is a one-way hash function, and $|$ denotes concatenation. $g(p_i)$ is a function to produce a digest for point p_i:

$$g(p_i) = \sum_{r=1}^{d} h^{U_r - x_{ir} - 1}(x_{ir})|h^{x_{ir} - L_r - 1}(x_{ir}) \tag{2}$$

where $h^j(x_{ir}) = h^{j-1}(h(x_{ir}))$ and $h^0(x_{ir})$ applies a one-way hash function on x.[1]

Moreover, for the two delimiters,

$$sig(p_0) = s(h(h(L_1|\ldots|L_d)|g(p_0)|g(p_1))) \tag{3}$$

$$sig(p_{k+1}) = s(h(g(p_k)|g(p_{k+1})|h(U_1|\ldots|U_d))) \tag{4}$$

In addition, each partition P has an associated signature:

$$sig(P) = s(h(g(p_0)|g(p_{k+1})|h(k))) \tag{5}$$

Query Processing: Assuming that a partition P is returned. We have to prove that all the data points within P that fall within the query window Q are returned.

Case (a): Q contains P. The verification process for this case is straightforward. The publisher server returns p_0 to p_{k+1}, and k, together with the respective signatures $sig(p_0)$ to $sig(p_{k+1})$ and $sig(P)$. (To reduce traffic overhead, we could send just one combined signature instead of the individual signatures, using the signature aggregation technique in [5].) The user first verifies that

$$s^{-1}(sig(P)) = h(g(p_0)|g(p_{k+1})|h(k))$$

Then, for each p_i, $1 \leq i \leq k$, the user verifies that p_i is indeed in P (by checking that P bounds p_i). Finally, for each p_i, $1 \leq i \leq k$, the user computes its digest and checks whether

$$s^{-1}(sig(p_i)) = h(g(p_{i-1})|g(p_i)|g(p_{i+1}))$$

If all the above checks are successful, the answer contains all the data points in P.

Case (b): P contains Q. Let $p_i = (x_{i1}, x_{i2}, \ldots, x_{id})$. The data points in P can be separated into: (a) $p_\alpha, p_{\alpha+1}, \ldots, p_{\beta-1}, p_\beta$ such that $x_{i1} \in [q_{l1}, q_{u1}]$ for

---

[1] To achieve tighter security, $h^0(x_{ir})$ can be redefined as $h^0(x_{ir}|rand(p_i))$ where $rand(p_i)$ is a random number associated with p_i; in which case we will need to supply the corresponding $rand(p_i)$ with each returned record. For ease of presentation, we shall adopt the simpler definition of $h^0(x_{ir})$.

$\alpha \leq i \leq \beta$. These points can be further categorized into answer points ($\mathcal{A}$) and false positives ($\mathcal{F}$). For each answer point $p_i \in \mathcal{A}$, $\forall r \ x_{ir} \in [q_{lr}, q_{ur}]$, whereas for each false positive $p_i \in \mathcal{F}$, $\exists r \ x_{ir} \notin [q_{lr}, q_{ur}]$. (b) $p_1, \ldots, p_{\alpha-1}, p_{\beta+1}, \ldots, p_k$, which are clearly not answer points.

(i) For each point $p_i \in \mathcal{A}$, the server returns p_i and $sig(p_i)$.
(ii) For each point $p_i \in \mathcal{F} \cup \{p_{\alpha-1}, p_{\beta+1}\}$, the server returns several pieces of information: (i) if $x_{ir} \in [q_{lr}, q_{ur}]$, $h^{U_r - x_{ir} - 1}(x_{ir}) | h^{x_{ir} - L_r - 1}(x_{ir})$ is returned; (ii) if $x_{ir} < q_{lr}$, $h^{q_{ur} - x_{ir} - 1}(x_{ir})$ and $h^{x_{ir} - L_r - 1}(x_{ir})$ are returned; (iii) if $x_{ir} > q_{ur}$, $h^{U_r - x_{ir} - 1}(x_{ir})$ and $h^{x_{ir} - q_{lr} - 1}(x_{ir})$ are returned.
(iii) The server also returns p_0, p_{k+1}, k, $sig(p_0)$, $sig(p_{k+1})$ and $sig(P)$.

With information from step (ii), the user can compute $g(p_i)$ without knowing the actual value of p_i:

- If $x_{ir} < q_{lr}$, the user applies h on ($h^{q_{ur} - x_{ir} - 1}(x_{ir})$) ($U_r - q_{ur}$) times to get ($h^{U_r - x_{ir} - 1}(x_{ir})$).
- If $x_{ir} > q_{ur}$, the user applies h on ($h^{x_{ir} - q_{lr} - 1}(x_{ir})$) ($q_{lr} - L_r$) times to get ($h^{x_{ir} - L_r - 1}(x_{ir})$).
- The user computes $g(p_i)$ using Equation (2).

The above procedure is secure against cheating by the publisher provided $h^i(p)$ for $i < 0$ is either undefined or computationally infeasible to derive. We use an iterative hash function for $h^i(p)$, because there is no known algebraic function that satisfies the requirement. To ensure that $h^{-1}(p) \neq p$, a hash function is chosen that outputs a different digest length from the length of p.

Similar to case (a), the user verifies the completeness of the query answer as follows:

- Verify that the bounding box is correct using information from step (iii), and determine whether $s^{-1}(sig(P)) = h(g(p_0) | g(p_{k+1}) | h(k))$.
- Verify that each point p in $\mathcal{A}$ is in P by checking that p is bounded by P.
- Verify that each point $p_i \in \mathcal{A}$ is authentic using information in step (ii) and the derived information to check $s^{-1}(sig(p_i)) = h(g(p_{i-1}) | g(p_i) | g(p_{i+1}))$.

Again, any attempt by the publisher server to cheat would lead to an unsuccessful match in at least one of the above cases.

Finally, we emphasize that extra data points that are returned for proving completeness are in the form of digests. Thus only the existence of the data points are revealed, but not their actual content. [2]

---

[2] If a non-answer $p_i \in \mathcal{F}$ has the same coordinate as an answer point $p_j \in \mathcal{A}$ along some dimension, both points will have the same digest for that dimension and p_i's coordinate will be revealed. This can be overcome by simply adopting $h^0(x_{ir} | rand(p_i))$ as explained in footnote 1.

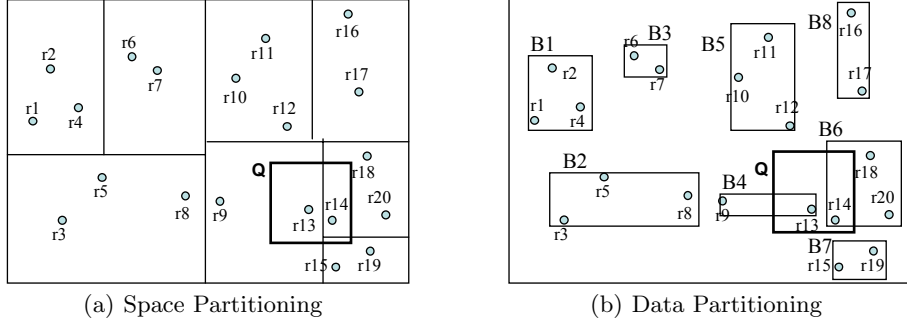(a) Space Partitioning   (b) Data Partitioning

Fig. 3. Partitioning Strategies

4 Verifying the Data Partitions

Having shown how to prove that all qualifying data points in a candidate partition (that overlaps the query window) are returned correctly, we now look at the first issue of verifying that the query answer covers all the candidate partitions.

A naive solution is to treat the entire data space as a single large partition, so that the mechanism described in Section 3 alone suffices. However, we expect this solution to have poor precision.

To achieve high precision, we adopt partition-based strategies so that only those partitions that contain some qualifying data points need to be considered for a query. In this way, any potential information leakage is limited to only those partitions that contribute to the query answer, rather than across the entire data space. We present our solution based on two partitioning techniques (see Figure 3): space partitioning and data partitioning.

4.1 Space Partitioning

With space partitioning schemes, the partitions are disjoint but their union covers the entire data space. As such, all we need to do is to verify that the bounding boxes of the returned partitions are correct, and that the union of these partitions covers the query scope. The former has already been addressed in Section 3, while the latter is just a simple check on the partition boundaries.

To illustrate, Figure 3(a) shows the data space being partitioned through a k-d tree [4]. In the figure, the window of the query **Q** overlaps three partitions, so only data from these three partitions are returned in the answer.

Besides the k-d tree, other spatial indexing techniques like the grid file [13] and quadtree [21] can also be employed to help the publisher to locate the candidate partitions quickly. Our authentication mechanism entails no changes to the spatial data structures. (As we shall see shortly, this is not the case for data partitioning schemes.)

4.2 Data Partitioning

With data partitioning approach (e.g., R-tree), the union of all the partitions may not cover the entire data space. Thus, space that contains no data points may not be covered by any partition, as illustrated in Figure 3(b). The existence of empty space poses a challenge to verifying the completeness of query answers: How does the user know that portions of a query window that are not covered by any returned partitions indeed are empty spaces, without physically examining all the partitions? Referring to Figure 3(b), how can the user be sure that Q only intersects boxes B4 and B6 and not the other partitions?

Our solution is to extend the signature chain concept to the partitions. Specifically, we order the partitions by their starting boundaries along a selected dimension (as is done for point data), then chain the partitions so that the signature of a partition is dependent on the neighboring partitions to its left and right.

Let the bounding box of the ith partition be demarcated by $[l, u]$ where $l = (l_{i1}, l_{i2}, \ldots, l_{id})$, and $u = (u_{i1}, u_{i2}, \ldots, u_{id})$. Each partition P_i has an associated signature (based on signature chaining):

$$sig(P_i) = s(h(g(P_{i-1})|g(P_i)|g(P_{i+1}))) \tag{6}$$

where P_{i-1} and P_{i+1} are the left and right sibling partitions of P_i, and $g(P_i)$ is defined as follows:

$$g(P_i) = h(h(l_{i1}|\ldots|l_{id})|h(u_{i1}|\ldots|u_{id})|h(k_i)) \tag{7}$$

where k_i is the number of points within P_i.

In addition, we define two fictitious partitions as delimiters. This is similar to what we did in building the signature chain for data points in Section 3, so we shall not elaborate further.

During query processing, all the partition information along with their signatures are returned as part of the query answer. The user can be certain that no partition is omitted, otherwise some signatures will not match. For those partitions that overlap the query window, the user then proceeds to check their data points using the mechanism in Section 3. The remaining partitions that do not intersect the query window are dropped from further consideration.

To minimize the extra partitions that are disclosed to the user, and to reduce performance overheads, we apply a hierarchical data partitioning indexing structure like the R-tree on the data. The partitions within each internal node of the R-tree are chained as described above. Given a window query, the publisher server iteratively expands the child nodes corresponding to those candidate partitions in the current node, starting from the root down to the leaf nodes. All the partition information and signatures along the path of traversal are added to the query answer for user verification.

5   A Performance Study

In this section, we report results of an experimental study conducted to evaluate the effectiveness of our authentication mechanisms, which we have implemented

in Java. We study three schemes: Verifiable KDtree (VKDtree) scheme that is based on space partitioning using the k-d tree; Verifiable Rtree (VRtree) scheme that is based on data partitioning using the R-tree; and Z-ordering scheme which employs Z-ordering [14] on the entire data space (as a single partition). The performance metric is the precision of query answers. Again, a low precision reveals the existence of extra data points and incurs traffic overhead, but not the actual content of those data points.

Unless stated otherwise, the following default parameter settings are used: the number of dimensions is 4, the data distribution is Gaussian, the number of data points is $1,000,000$. The domain of each dimension is [1, 10M]. The node capacity is 50 (i.e., each node holds up to 50 data points). Queries are generated by picking a point randomly from the dataset, then marking out the query window with the chosen point as center. The length of the query window along each dimension is $l \times domain\_size$; by default, l is set to 0.1. For each experiment, we run 500 queries, and take the average precision.

5.1   Effect of Number of Dimensions

We first vary the number of dimensions from 2 to 5. The results are summarized in Figure 4(a). As expected, as the number of dimensions increases, all the schemes lose precision, because more non-answer points must be provided to verify the completeness of the query answers.

We also observe that the VKDtree scheme performs well for two-dimensional space, but its precision drops dramatically at higher dimensions. This is because more partitions are returned as a result of their overlapping the query window. The result for Z-ordering is, surprisingly, similar to the VKDtree scheme. In fact, it even performs better than VKDtree in some cases. Investigation shows that this is because the coverage of the partitions returned under VKDtree may be larger than the region covered by the Z-ordering scheme. Finally, the VRtree scheme achieves precisions of at least 60%, is least affected by dimensionality, and appears to perform the best overall. This is because the data partitioning scheme is able to effectively limit the number of candidate partitions returned in the query answers.

5.2   Effect of Different Data Distributions

In the second experiment, we study the effect of different data distributions. Figure 4(b) shows the precisions of the various schemes under three different distributions: Exponential, Uniform and Gaussian. The precisions of all the schemes are better with the exponential dataset, because the data generated under the exponential distribution are clustered toward one corner (the origin) of the data space, whereas they are more spread out under the other two distributions.

The relative performance of the three schemes remain largely the same as before: with VRtree performing the best, while VKDtree and Z-ordering exhibit similar performance. We also note that VRtree is much more effective than VKDtree and Z-ordering under uniform data distribution.
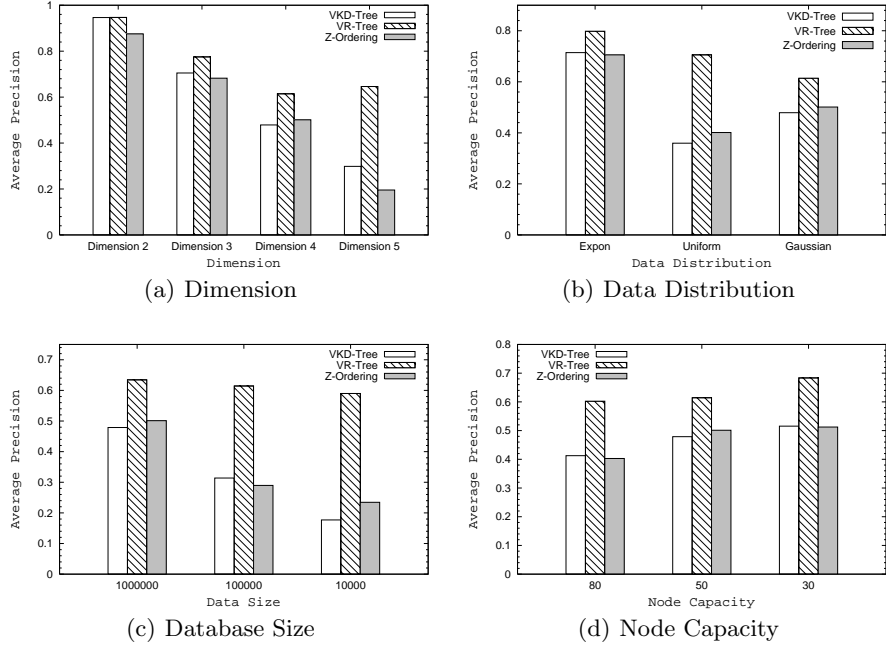
(a) Dimension   (b) Data Distribution

(c) Database Size   (d) Node Capacity

Fig. 4. Comparative Study

5.3 Effect of Dataset Sizes

With a fixed data space, the size of the dataset will have an effect on the performance of the schemes. In particular, for large datasets, the data space becomes more densely populated. For a fixed-size query, this means that the precision will, with high probability, be higher (compared to one with small dataset size). This intuition is confirmed in our study, as shown in Figure 4(c) which presents the results for dataset sizes of 1,000,000, 100,000, and 10,000. The relative performance of the various schemes remain largely the same as in the earlier experiments, though VRtree is less affected by the size of the datasets compared to VKDtree and Z-ordering.

5.4 Effect of Node Capacity

In this study, we examine the effect of node capacity, which determines the maximum number of points allowed per partition. Obviously, a larger node capacity means that it is more likely that more non-answer points are returned (compared to a smaller node capacity), thus yielding lower precisions. Figure 4(d) shows the results for node capacities of 30, 50 and 80. From the figure, we notice that the precision of all the schemes improve as the node capacity reduces from 80 to 50 and then to 30.

5.5 Client Computation Cost

In this section, we evaluate the overhead of computation cost at the client side in authenticating the query results. For both VKDtree and VRtree, the client computation cost includes result entry verification cost (C_{RV}), boundary verification cost(C_{BV}) and signature verification cost (C_{SV}). Figure 5 shows the authentication overhead of VKD-tree and VR-tree conducted in our experiment, where the overhead is measured as

$$\frac{client\ computation\ cost\ -\ processing\ cost}{processing\ cost}$$

where the processing cost refers to the cost for verifying only answer tuples. It turns out that there is no significant differences between the two schemes - while VRtree incurs lower cost to verify the answers (lower false drops), it incurs additional cost to verify the chaining of partitions; whereas VKDtree does not need to deal with partition chaining but it returns more false drops and hence incur larger cost to verify the answers.
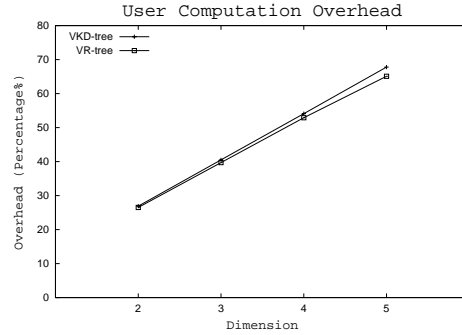

Fig. 5. Client Computation Cost

6   Conclusion

In this paper, we introduce a mechanism for users to verify that their query answers on a *multi-dimensional dataset* are correct. The mechanism follows a partition-based strategy, and comprises two steps: (a) verify that all partitions relevant to the query are returned, and (b) verify that all qualifying data points within each relevant partition are returned. The *signature chain* technique from [15] is used to chain up points and partitions so that any malicious omissions can be detected by the user. We study two schemes: Verifiable KD-tree (VKDtree) that is based on space partitioning, and Verifiable R-tree (VRtree) that is based on data partitioning. The schemes are evaluated on window queries, and results show that the VRtree is highly precise, meaning that few data points outside of a query answer are disclosed in the course of proving its correctness.

References

1. Encrypting File System (EFS) for Windows 2000. http://www.microsoft.com/windows2000/techinfo/howit works/security/encrypt.asp.
2. Proposed Federal Information Processing Standard for Digital Signature Standard (DSS). *Federal Register*, 56(169):42980–42982, 1991.
3. *Secure Hashing Algorithm*. National Institute of Science and Technology. FIPS 180-2, 2001.
4. J. Bentley. Multidimensional Binary Search Trees Used For Associative Searching. *Communications of the ACM*, 18(9):509–517, September 1975.
5. D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *Proceedings of Advances in Cryptology – EUROCRYPT'03, E. Biham, Ed., LNCS, Springer-Verlag*, 2003.
6. P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic Data Publication over the Internet. In *14th IFIP 11.3 Working Conference in Database Security*, pages 102–112, 2000.
7. R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of the 29th International Conference on Very Large Databases*, pages 321–332, 2003.
8. Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. Lindsay, and J. Naughton. Middle-Tier Database Caching for E-Business. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 600–611, 2002.
9. D. Margulius. Apps on the Edge. *InfoWorld*, 24(21), May 2002. http://www.infoworld.com/article/02/05/23/ 020527feedgetci_1.html.
10. G. Miklau and D. Suciu. Controlling Access to Published Data Using Cryptography. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 898–909, 2003.
11. E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and Integrity in Outsourced Databases. In *Proceedings of the Network and Distributed System Security Symposium*, February 2004.
12. B. Neuman and T. Tso. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications Magazine*, 32(9):33–38, 1994.
13. J. Nievergelt, H. Hinterberger, and K. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.
14. J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, pages 181–190, 1984.
15. H. Pang, A. Jain, K. Ramamritham, and K. Tan. Verifying Completeness of Relational Query Results in Data Publishing. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, 2005.

16. H. Pang and K. Tan. Authenticating Query Results in Edge Computing. In *IEEE International Conference on Data Engineering*, pages 560–571, March 2004.

17. H. Pang, K. Tan, and X. Zhou. StegFS: A Steganographic File System. In *Proceedings of the 19th International Conference on Data Engineering*, pages 657–668, Bangalore, India, March 2003.

18. R. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm*. Internet Activities Board, 1992.

19. R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

20. H. Sagan. *Space-Filling Curves*. Springer-Verlag, New York, 1994.

21. H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

22. R. Sandhu and P. Samarati. Access Control: Principles and Practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

23. S. Saroiu, K. Gummadi, R. Dunn, S. Gribble, and H. Levy. An Analysis of Internet Content Delivery Systems. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 315–327, 2002.