Efficient Tree Pattern Queries On Encrypted XML Documents

Fang-Yu Rao[1], **Jianneng Cao**[2], **Mehmet Kuzu**[3], **Elisa Bertino**[1], **Murat Kantarcioglu**[3]

[1]Department of Computer Science/CERIAS, Purdue University, West Lafayette, IN 47906 USA.

[2]Institute for Infocomm Research, 1 Fusionopolis Way, Singapore 138632.

[3]Department of Computer Science, University of Texas at Dallas, Richardson, TX 75080 USA.

Abstract. Outsourcing XML documents is a challenging task, because it encrypts the documents, while still requiring efficient query processing. Past approaches on this topic either leak structural information or fail to support searching that has constraints on XML node content. To address these problems, we present a solution for efficient evaluation of tree pattern queries (TPQs) on encrypted XML documents. We create a domain hierarchy, such that each XML document can be embedded in it. By assigning each node in the hierarchy a position, we create for each document a vector, which encodes both the structural and textual information about the document. Similarly, a vector is created also for a TPQ. Then, the matching between a TPQ and a document is reduced to calculating the distance between their vectors. For the sake of privacy, such vectors are encrypted before being outsourced. To improve the matching efficiency, we use a k-d tree to partition the vectors into non-overlapping subsets, such that non-matchable documents are pruned as early as possible. The extensive evaluation shows that our solution is efficient and scalable to large dataset.

1 Introduction

Since Kodak signed a $1 billion contract with IBM, DEC, and Businessland [12] in 1989 to outsource its information system, data outsourcing has gained a widespread interest. Outsourcing is beneficial to the data owners. It helps to save the cost of building and maintaining a private database system, and thus allows data owners to focus on their core competencies. Recently, due to the advances in networking and computing technology, the cloud has emerged as a technology that can provide reliable and flexible data access service at a relatively low price. Therefore, data owners are even more likely to outsource their data. However, despite all the appealing features, moving data to a cloud server may endanger individual privacy, since data may contain sensitive information (e.g., medical records). To address such concerns, the data is usually encrypted prior to its outsourcing, which makes efficient query processing very challenging.

An XML document organizes data in a hierarchy and describes semantic relationships among data elements by user-defined tags. An XML document thus contains both structural and textual information. In order to achieve strong privacy, it is important that both types of information are encrypted. In the last decade, query processing on encrypted data has been heavily investigated. However, most of the works are proposed for relational data

[15, 16, 29, 33]; they do not have a support for searching on encrypted structures. Recently, query processing over encrypted XML documents [7, 34] has been investigated. However, these approaches either leak structural information [34] or fail to support searches that have constraints on node contents [7] (see Section 7 for details).

In this paper we consider tree pattern query (TPQ) [8, 28] over encrypted XML documents. A TPQ is a tree, which consists of labeled nodes and predicates that specify the constraints on the nodes. TPQ is a core operation of XQuery [3], which is now the de facto standard of XML query processing language. We propose a novel solution to evaluate TPQ on encrypted XML document. The key contribution is that we transform the tree pattern matching to a problem of vector comparison. We assume the existence of a domain hierarchy, which is composed of the document type definitions (DTDs) of all the XML documents in the dataset. Each document is an embedding in the hierarchy. By assigning each node in the hierarchy a position, we create for each document a vector, which encodes both the structural and textual information of the document. Similarly, we create a vector also for a TPQ. Thus, the matching of a TPQ with an XML document is reduced to calculating the distance between their vector representations. The generated vectors are encrypted via ASPE [35], which ensures privacy and at the same time supports distance evaluation (i.e., KNN search) on encrypted vectors.

Furthermore, we build a privacy-preserving indexing to improve the efficiency of our scheme. We apply a k-d tree to hierarchically partition the XML documents, so that irrelevant XML documents to a TPQ are pruned as early as possible. The k-d tree hierarchy may contain sensitive information about the data. Therefore, we adopt privacy metrics to guide the construction of the k-d tree, so that the disclosure of sensitive information is effectively controlled. Compared with existing approaches [7, 34], our solution protects both the structure and content of XML documents, while supporting query processing on them. We have also carried out extensive experiments, whose results confirm that our solution is efficient and scalable to large datasets.

The rest of our paper is organized as follows. The next section first formulates the problem. Then, we present our approach in Section 3, and improve its efficiency in Section 4. We discuss our approach in Section 5. Section 6 reports our experimental results. Finally, we review related work in Section 7 and conclude our work in Section 8.

2 Problem Formulation

We model an XML document as a labeled rooted tree. Each node in the tree has a name. A node is either an element or an attribute. A leaf element and an attribute may have content, which is either a string or a numerical value.

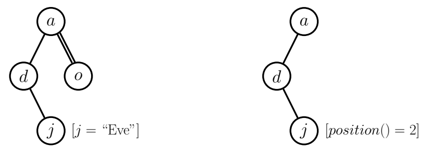
In our setting there are three roles: data user Alice, data owner Bob, and cloud server Charlie. We assume that Bob has a set of XML documents that contain sensitive information, and that he wants to outsource them to the cloud server Charlie. To assure confidentiality, data is stored in an encrypted form at Charlie. Authorized data user Alice should be able to selectively retrieve XML documents from Charlie through TPQs (see Definition 2 below). For the sake of query privacy, the TPQs are also encrypted. The server should be able to evaluate the encrypted TPQs, and return those encrypted documents that satisfy the constraints specified in the TPQs. Once receiving the query results, Alice decrypts them and obtains the desired documents in plaintext. Furthermore, we assume that Charlie is semi-honest, i.e., he strictly follows the query processing protocol as it is defined but he may try to infer private information from the query evaluation.

Tree pattern query. XQuery [3] is the currently de facto standard of XML query processing language. Due to its complexity, fully supporting it is beyond the scope of our work. Instead, we consider tree pattern query (TPQ) [8], one of its core operators. TPQ has a wide range of applications–besides XML query processing, it can also be applied to web data management and selective data dissemination. TPQ is compatible with the semantics of XPath [2]. Its formal definition is as follows.

Definition 1 (Tree pattern)**.** A tree pattern is a tree, such that 1) each of its nodes has a name, and 2) each of its edges is either a single edge representing parent-child (PC) relationship or a double edge representing ancestor-descendant (AD) relationship.

Definition 2 (Tree pattern query)**.** A tree pattern query is a pair (Q_t, Q_c), where Q_t is a tree pattern and Q_c is a boolean combination of predicates defined on the nodes of Q_t.

We support two kinds of predicates. The first is *content predicate* in the form of $[x\ op\ \alpha]$, where x represents the content of a node , $op \in \{>, \geq, <, \leq, =\}$, and α is a value. It selects nodes whose content values satisfy the predicate. As in XPath, we denote the content of a node by its node name. The second is *position predicate* in the form of $[position() = m]$. It selects the m-th child node of the current context node.



(a) Content Predicate (b) Position Predicate

Figure 1: Two Types of Predicates

Figure 1(a) shows an example of a TPQ, in which a and d (d and j) are connected by a PC relationship, a and o are connected by an AD relationship, and the content of j is required to be equal to "Eve". Since TPQ follows the semantics of XPath, the example TPQ is also equal to the combination of the following two XPath queries: $Path_1 = /\texttt{child} :: \texttt{a}/\texttt{child} :: \texttt{d}/\texttt{child} :: \texttt{j}[\texttt{j} = \text{“}Eve\text{”}]$ and $Path_2 = /\texttt{child} :: \texttt{a}/\texttt{descendant} :: \texttt{o}$. Figure 1(b) shows an example of a TPQ with a position predicate in which the 2nd child of d with node name j should be chosen.

3 The Solution

In this section, we present our approach of evaluating TPQs over encrypted XML documents. In Section 3.1, we will first present a strategy to encode XML documents (and TPQs) into vectors. Then, in Section 3.2, we adapt an encryption scheme to our specific requirements. Based on the first two sections, Section 3.3 gives the details of our approach.

3.1 Encoding XML Documents into Vectors

We create vectors for XML documents. Since such vectors encode both the structural and textual information about the documents, they can be regarded as indices for efficient query processing. To generate the vectors, we assume that all the XML documents in the database are constructed according to their Document Type Definitions (DTDs). We construct

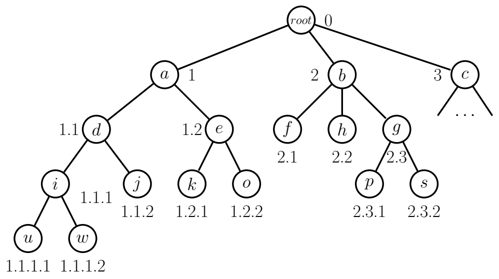


Figure 2: An Example of Domain Hierarchy

a domain hierarchy by appending a root node over all the DTDs. Consider the example in Figure 2, in which the subtrees rooted at nodes a, b, and c represent three DTDs. An XML document can be seen as a subset in the domain hierarchy, or more precisely, an *embedding* (see the next definition) of the domain hierarchy.

Definition 3 (Tree embedding [18])**.** Let T and T' be trees with sets of nodes V and V', respectively. An injective function $f : V \to V'$ is an embedding of T into T', if for all the nodes $x, y \in V$:

- $\mathsf{label}(f(x)) = \mathsf{label}(x)$, where $\mathsf{label}(f(x))$ and $\mathsf{label}(x)$ are the labels of $f(x)$ and x in trees T' and T, respectively,
- $f(x)$ is a descendant of $f(y)$ in T' if and only if x is a descendant of y in T.

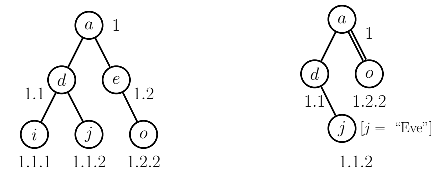
According to a DTD, some elements (e.g., the author of a book) are allowed to appear multiple times in an XML document. To ensure that each XML document is an embedding in the domain hierarchy, we duplicate certain nodes in the domain hierarchy. As an example, consider Figure 2. Suppose that node j represents the author name of a book, and that M is the maximum number of authors for a book in the dataset[1]. Then, we duplicate j M times under node d in the domain hierarchy (see Section 5 for a more detailed discussion). On the other hand, we note that the recursion of elements in a DTD is also supported in our solution after the depth of the recursion is determined beforehand.

Once the domain hierarchy is ready, we utilize Dewey labeling[2] scheme [36] to label the domain hierarchy. Let p and c be two nodes with Dewey labels $a_1.a_2 \dots a_m$ and $b_1.b_2 \dots b_n$, respectively. If p is the parent of c, then $a_i = b_i$ $(i = 1, 2, \dots, m)$ and $n = m + 1$. The last component b_n for node c denotes the local order of c among its siblings. Consider the example in Figure 2, in which each subtree under the root node is labeled by Dewey labeling. Based on such labeling on the domain hierarchy, an XML document can be labeled accordingly. Figure 3(a) shows an example XML document together with its Dewey labels. A TPQ can be considered as a tree, and thus can also be labeled. The example in Figure 3(b) is one possible labeling of the TPQ in Figure 1(a). A TPQ may have multiple embeddings in the domain hierarchy, thus it may have more than one encoding (refer to Section 5 for more details).

All the Dewey labels in the domain hierarchy, denoted as U, can be regarded as a universal set. Correspondingly, the set of Dewey labels in an XML document (or a TPQ) can then be

[1]For dynamic data settings, we may increase M to $M' = M + x$ so that M' can be a safe upperbound for the number of authors in a book.

[2]Other labeling schemes like containment scheme and pre/post labeling scheme are also applicable. We choose Dewey labeling because it intuitively encodes AD/PC relationship.



(a) An XML Document (b) A TPQ
Figure 3: The Dewey Labels of a Document/TPQ

regarded as a subset of U. If we further order[3] the labels in U, then we can transform the structure of a document (or TPQ) into a binary vector. To be formal, let X be the set of Dewey labels for a document (or TPQ) such that $X \subseteq U$. Let $pos(x)$ be the order of a label $x \in U$. We will construct a bit vector A for X in the following way: $A[pos(x)] = 1$ if $x \in X$ and $A[pos(x)] = 0$ otherwise. Consider the example in Figure 4. A_{D_t} and A_{Q_t} are the bit vectors for the structural information of the XML document and the TPQ in Figure 3, respectively.

Dewey Labels	1	1.1	1.1.1	1.1.1.1	1.1.1.2	1.1.2	1.2	1.2.1	1.2.2	2	
A_{D_t}	1	1	1	0	0	1	1	0	1	0	$\cdots$
A_{Q_t}	1	1	0	0	0	1	0	0	1	0	$\cdots$

Figure 4: Bit Vectors for a Document/TPQ

The bit vector representation facilitates structural comparison. Given the bit vectors of a document and a TPQ denoted as A_{D_t} and A_{Q_t}, respectively, we can easily check whether their structures match or not. Particularly, if the inner product between A_{Q_t} and A_{D_t} is equal to the number of 1's in A_{Q_t}, then it can be concluded that the document matches the query. Furthermore, we can also encode the node values of XML documents into vectors to support content-based matching. Given an XML document D, we create its textual vector A_{D_c} as follows. For any node $x \in U$, if $x \in D$ and x has a value in D, then $A_{D_c}[pos(x)]$ is:

$$A_{D_c}[pos(x)] = \begin{cases} h(content(x, D)) & \text{if } x \text{ is a string} \\ content(x, D) & \text{if } x \text{ is a number,} \end{cases} \tag{1}$$

where h is a cryptographic hash function from $\{0,1\}^*$ to $\{0,1\}^\ell$ and $content(x, D)$ represents the value of node x in D. On the other hand, if $x \notin D$, or $x \in D$ but it does not have any value, then a special integer ω, which is different from any value defined in Equation 1 is assigned to $A_{D_c}[pos(x)]$. As a simple solution, we can shift all numerical values, so that all of them are larger than or equal to 1, and set $\omega = 0$. Note that the encoding of predicates in a TPQ is different from the above (see query construction in Section 3.3 for details).

Example 1. Consider the XML document D in Figure 3(a). Suppose that node j represents forename, and its value in D is equal to "Adam". Furthermore, suppose that the order of j in the label domain is 6. Then, the sixth dimension of the textual vector for D is set to $h(\text{"}Adam\text{"})$, i.e., $A_{D_c}[6] = h(\text{"}Adam\text{"})$.

[3]For example, we can order the labels by the pre-order traversal of the domain hierarchy.

3.2 Asymmetric Scalar-product Preserving Encryption

The encryption strategy we adopt here is based on asymmetric scalar-product preserving encryption (ASPE) scheme, which was proposed in [35] for efficient secure nearest neighbor search on the cloud. In particular, suppose that P_1 and P_2 are two data points, and Q is a query point in Euclidean space. If P_1, P_2 and Q are encrypted by ASPE, then a third party will not learn the values of the points and the query. But it can still determine whether P_1 is closer to Q than P_2. The building blocks of ASPE are briefly summarized as follows:

•**Key**. Two $(n+1)\times(n+1)$ invertible matrices M_1 and M_2, and a binary string S of length $n+1$.

•**Data encryption function** E_1. Let P be an n-dimensional data point. Extend P to $\overline{P} = (P^T, -0.5\|P\|^2)^T$. Create (P_a, P_b), such that: 1) if $S[i]=1$, set $P_a[i]=\rho_i$ and $P_b[i]=\overline{P}[i]-\rho_i$, where ρ_i is a random number, and 2) if $S[i]=0$, set $P_a[i]=P_b[i]=\overline{P}[i]$. The encryption of P is $\mathsf{E}_1(P)=[(M_1^T P_a)^T, (M_2^T P_b)^T]^T$.

•**Query encryption function** E_2. Let Q be a query point. Extend Q to $\widehat{Q}=r(Q^T,1)^T$, where r is a positive random number. Create (Q_a, Q_b), such that: 1) if $S[i]=1$, set $Q_a[i]=Q_b[i]=\widehat{Q}[i]$, and 2) if $S[i]=0$, set $Q_a[i]=\sigma_i$ and $Q_b[i]=\widehat{Q}[i]-\sigma_i$, where σ_i is a random number. The encryption of Q is $\mathsf{E}_2(Q)=[(M_1^{-1}Q_a)^T, (M_2^{-1}Q_b)^T]^T$.

•**Comparison function** Comp. Let $\mathsf{E}_1(P_1)$, $\mathsf{E}_1(P_2)$, and $\mathsf{E}_2(Q)$ be the encryptions of two points P_1 and P_2, and a query Q, respectively. To check whether P_1 is nearer to Q than P_2, the function checks if $(\mathsf{E}_1(P_1)-\mathsf{E}_1(P_2))\odot \mathsf{E}_2(Q)>0$, where $\odot$ is the inner product.

In the following we briefly discuss the correctness of the protocol. The formal proof can be found in [35].

Fact 1.

$$P_a \odot Q_a + P_b \odot Q_b = \overline{P} \odot \widehat{Q}$$

Fact 2.

$$(\overline{P_1}-\overline{P_2})\odot \widehat{Q} = 0.5r(\mathsf{d}^2(P_2,Q)-\mathsf{d}^2(P_1,Q)),$$

where function d measures the Euclidean distance between two points.

Based on the two facts, we have

$$\begin{aligned}
&(\mathsf{E}_1(P_1)-\mathsf{E}_1(P_2))\odot \mathsf{E}_2(Q)\\
&=([P_{1a}^T, P_{1b}^T]^T - [P_{2a}^T, P_{2b}^T]^T)\odot [Q_a^T, Q_b^T]^T\\
&=(\overline{P_1}-\overline{P_2})\odot \hat{Q}\\
&=0.5r(\mathsf{d}^2(P_2,Q)-\mathsf{d}^2(P_1,Q)).
\end{aligned} \tag{2}$$

The first equation comes from directly expanding the first expression according to the definitions of E_1 and E_2. Note that for two points A and B of the same number of dimensions, $A\odot B = A^T B$. The second and the third equation hold true according to *Fact 1* and *Fact 2*, respectively. It can be seen that if $(\mathsf{E}_1(P_1)-\mathsf{E}_1(P_2))\odot \mathsf{E}_2(Q)>0$, then P_1 is nearer to Q than P_2. For a clear presentation, in the following we denote the comparison function by the following equation:

$$\begin{aligned}
&\mathsf{Comp}(\mathsf{E}_1(P_1),\mathsf{E}_1(P_2),\mathsf{E}_2(Q))\\
&=\begin{cases} 0, & \text{if } \mathsf{d}(P_1,Q)=\mathsf{d}(P_2,Q)\\ -1, & \text{if } \mathsf{d}(P_1,Q)<\mathsf{d}(P_2,Q)\\ 1, & \text{if } \mathsf{d}(P_1,Q)>\mathsf{d}(P_2,Q)\end{cases}
\end{aligned}$$

Below we also give a concrete example of how a third party could determine for a data user whether or not a query point Q is closer to P_1 than P_2.

Example 2. Assume that $P_1^T = (1, 3)$ and $P_2^T = (1, 11)$ be two points in the database, and that a data user would like to know which one of them is closer to the point $Q^T = (3, 4)$. Let (M_1, M_2, S) be the key that is pre-shared between the data owner and the data user such that

$$M_1 = \begin{bmatrix} 0 & 1 & 5 \\ -1 & -1 & 0 \\ -4 & -5 & -4 \end{bmatrix},$$

$$M_2 = \begin{bmatrix} 2 & -4 & -13 \\ 1 & -2 & -6 \\ -1 & 1 & 4 \end{bmatrix},$$

and $S^T = (1, 0, 1)$. From the above we can see

$$M_1^{-1} = \begin{bmatrix} 4 & -21 & 5 \\ -4 & 20 & -5 \\ 1 & -4 & 1 \end{bmatrix},$$

and

$$M_2^{-1} = \begin{bmatrix} -2 & 3 & -2 \\ 2 & -5 & -1 \\ -1 & 2 & 0 \end{bmatrix}.$$

To encrypt P_1, the data owner chooses three random numbers: $\rho_{11} = 9$, $\rho_{12} = 3$, and $\rho_{13} = 8$. Then the data owner extends P_1 to $\overline{P_1}$ such that $\overline{P_1}^T = (1, 3, -5)$. With those three random numbers selected, the data owner creates $P_{1a}^T = (9, 3, 8)$ and $P_{1b}^T = (-8, 3, -13)$. Thus,

$$\begin{aligned} \mathsf{E}_1(P_1) &= [(M_1^T P_{1a})^T, (M_2^T P_{1b})^T]^T \\ &= [(-35, -34, 13), (0, 13, 34)]^T \end{aligned}$$

Similarly, to encrypt P_2 with three random numbers: $\rho_{21} = 7$, $\rho_{22} = 4$, and $\rho_{23} = 2$, after extending P_2 to $\overline{P_2}$ such that $\overline{P_2}^T = (1, 11, -61)$, the data owner creates $P_{2a}^T = (7, 11, 2)$ and $P_{2b}^T = (-6, 11, -63)$. Finally, $\mathsf{E}_1(P_2) = [(-19, -14, 27), (62, -61, -240)]$.

Now with a random number $r = 7$, the data user first extends Q to $\hat{Q}$ such that $\hat{Q}^T = (21, 28, 7)$. After that $Q_a^T = (21, 10, 7)$ and $Q_b^T = (21, 18, 7)$ are created. Therefore,

$$\begin{aligned} \mathsf{E}_2(Q) &= [(M_1^{-1} Q_a)^T, (M_2^{-1} Q_b)^T]^T \\ &= [(-91, 81, -12), (-2, -55, 15)]^T. \end{aligned}$$

The data user sends $\mathsf{E}_2(Q)$ to the third party.

The third party now computes $\mathsf{E}_1(P_1) - \mathsf{E}_1(P_2) = [(-16, -20, -14), (-62, 74, 274)]^T$, and then $(\mathsf{E}_1(P_1) - \mathsf{E}_1(P_2)) \odot \mathsf{E}_2(Q) = 168$. Because the inner product is greater than 0, the third party is able to tell that P_1 is closer to Q than P_2.

Wong et al. [35] show that the security of ASPE is roughly equal to a symmetric encryption scheme with n-bit key. To ensure sufficient security, it sets $n \geq 80$. If the data point has less than 80 dimensions, some extra dimensions would be added (refer to [35] for details).

In the context of our work, we utilize the features of ASPE to support queries on particular dimensions of an n-dimensional point P. In particular, suppose that $P[\lambda]$ is a dimension of P, and α is a numerical value that is compared against the content of $P[\lambda]$. We initially generate the following two n-dimensional vectors:

$$Q_{1\lambda} = (\gamma_1, \gamma_2, \cdots, \gamma_{\lambda-1}, \alpha - s, \gamma_{\lambda+1}, \cdots, \gamma_n)$$
$$Q_{2\lambda} = (\gamma_1, \gamma_2, \cdots, \gamma_{\lambda-1}, \alpha + s, r_{\lambda+1}, \cdots, \gamma_n),$$

where s and γ_i ($i = 1, 2, \ldots, \lambda-1, \lambda+1, \ldots, n$) are randomly chosen positive numbers.

We then encrypt $Q_{1\lambda}$, $Q_{2\lambda}$, and P by E_1 and E_2, respectively. That is, we use data encryption function E_1 to encrypt $Q_{1\lambda}$ and $Q_{2\lambda}$, and use query encryption function E_2 to encrypt P. Such a construction enables comparison of query content α with $P[\lambda]$ through ASPE function Comp as follows:

$$\begin{cases} P[\lambda] = \alpha & \text{if } \mathsf{Comp}(\mathsf{E}_1(Q_{1\lambda}), \mathsf{E}_1(Q_{2\lambda}), \mathsf{E}_2(P)) = 0 \\ P[\lambda] < \alpha & \text{if } \mathsf{Comp}(\mathsf{E}_1(Q_{1\lambda}), \mathsf{E}_1(Q_{2\lambda}), \mathsf{E}_2(P)) = -1 \\ P[\lambda] > \alpha & \text{if } \mathsf{Comp}(\mathsf{E}_1(Q_{1\lambda}), \mathsf{E}_1(Q_{2\lambda}), \mathsf{E}_2(P)) = 1 \end{cases} \quad (3)$$

3.3 Private TPQ on Encrypted XML Documents

Given a TPQ, its tree pattern and predicates can be evaluated separately. In particular, let A_{D_t} and A_{D_c} be the structural and textual vectors of an XML document D. Suppose that $Q = (Q_t, Q_c)$ is a TPQ, and A_{Q_t} is the bit vector that encodes its structure. Then, the inner product between A_{D_t} and A_{Q_t} can determine whether D matches Q with respect to structure. Let $[x, op, \alpha]$ be a predicate in Q_c. Then, the evaluation that takes A_{D_c} and $[x, op, \alpha]$ as input (i.e., Equation 3) determines whether D satisfies the predicate. However, in such an approach, there is a possibility that a document only matches the tree pattern of a TPQ, but does not match the predicates of the TPQ. The cloud server would notice this after the query evaluation. Such information leakage due to the separate treatment for structure and content may be undesirable in certain scenarios.

To address the above information leakage, we develop a strategy to combine the structural and textual encodings as a whole. The strategy is composed of four steps: 1) Key generation, 2) Index Building, 3) Query Construction and 4) Query Evaluation.

Key Generation. Let U be the universal set containing all the Dewey labels in the domain hierarchy. Data owner Bob generates two $(|U|+1) \times (|U|+1)$ invertible matrices M_1 and M_2 in which the entries are rational. Bob also creates a $(|U|+1)$-bit vector S. The two matrices and S are the secret keys, which are shared with data user Alice and will be used in ASPE encryption.

Index Building. For each XML document D, Bob first creates two vectors A_{D_t} and A_{D_c}, which encode the structure and the node contents of D, respectively. Then, he combines them to form a single vector A_D. In particular, suppose that the bit length[4] of $A_{D_c}[i]$ is at most ℓ, where $i = 1, 2, \ldots, |U|$. Bob sets $A_D[i] = A_{D_t}[i] \times 2^\ell + A_{D_c}[i]$, where $i = 1, 2, \ldots, |U|$. In such a way, A_D encapsulates both the structural and textual information of document D. Finally, A_D is encrypted using ASPE function E_2 and $\mathsf{E}_2(A_D)$ is transferred to Charlie as the index of D.

Query Construction. Let $Q = (Q_t, Q_c)$ be a TPQ, and x be a node in it. Alice first embeds

[4]We assume that the output length ℓ of hash function h is longer than the bit length of any numerical value in the database.

Q_t in the domain hierarchy. According to the embedding, suppose that the order of x in the label domain U is $pos(x) = \lambda$. Then, Alice creates a sub-query for x in one of the following two ways, based on whether Q_c contains content predicate on x.

Case 1. There exists content predicate $[x, op, \alpha] \in Q_c$ on x, where $op \in \{>, \geq, <, \leq, =\}$. In such a case, Alice computes $\bar{\alpha} = 2^{\ell} + \alpha$, and generates the following two vectors:

$$Q_{1\lambda} = (\gamma_1, \gamma_2, \cdots, \gamma_{\lambda-1}, \bar{\alpha} - s, \gamma_{\lambda+1}, \cdots, \gamma_{|U|}), \text{ and}$$
$$Q_{2\lambda} = (\gamma_1, \gamma_2, \cdots, \gamma_{\lambda-1}, \bar{\alpha} + s, \gamma_{\lambda+1}, \cdots, \gamma_{|U|}),$$

where s and γ_i ($i = 1, 2, \ldots, \lambda-1, \lambda+1, \ldots, |U|$) are positive random numbers. Both vectors are encrypted by encryption function E_1. Finally, Alice creates a triple ($\mathsf{E}_1(Q_{1\lambda})$, $\mathsf{E}_1(Q_{2\lambda})$, *op*). It can be easily verified that document D satisfies the predicate $[x, op, \alpha]$, if and only if its index makes "$A_D[\lambda]$ *op* $\bar{\alpha}$" hold.

Case 2. There is no content predicate on node x. In this case, Alice calculates $\bar{\alpha} = 2^{\ell}$. Then, she also generates $Q_{1\lambda}$ and $Q_{2\lambda}$, and encrypts them as in case 1. According to the index construction, if a document D contains node x, no matter whether D has content at x, the λ-th dimension in its index A_D must be greater than or equal to 2^{ℓ}, i.e., $A_D[\lambda] \geq 2^{\ell}$. Therefore, to check whether document D contains node x, Alice finally creates the triple ($\mathsf{E}_1(Q_{1\lambda})$, $\mathsf{E}_1(Q_{2\lambda})$, $\geq$). Furthermore, we can see that *case 2* is actually a special case of *case 1*, in which the content predicate is $[x, \geq, 0]$.

A TPQ may contain multiple nodes. On each node there might be one or more sub-queries (e.g., two content predicates on a node connected by '$\wedge$' or '$\vee$'). Thus, Alice needs to generate multiple such triples, one for each sub-query. After that, she connects the triples in conjunctive normal form (CNF), and sends them to Charlie.

Query Evaluation. Given an encrypted TPQ, characterized by a set of triples ($\mathsf{E}_1(Q_{1\lambda})$, $\mathsf{E}_1(Q_{2\lambda})$, *op*), for each triple, Charlie evaluates it and returns all the XML documents D such that the encrypted query evaluates to true. A triple ($\mathsf{E}_1(Q_{1\lambda})$, $\mathsf{E}_1(Q_{2\lambda})$, *op*) evaluates to true with respect to a document D if the following holds:

$$\mathsf{Comp}(\mathsf{E}_1(Q_{1\lambda}), \mathsf{E}_1(Q_{2\lambda}), \mathsf{E}_2(A_D)) = \begin{cases} 0, & \text{if } op = \text{'='} \\ -1, & \text{if } op = \text{'<'} \\ 1, & \text{if } op = \text{'>'} \end{cases}$$

3.4 Complexity Analysis

The above protocol consists of an offline step and an online step. The offline step includes *index building*, which is done by the data owner Bob only once. For each XML document, the time complexity for its secure index construction is $\Theta(|U|^2)$. Therefore, given an XML database with N documents, the time complexity for the offline step is $\Theta(N \times |U|^2)$. The online step is composed of *query construction* and *evaluation*. It runs for each submitted query. Given a TPQ Q, the time cost for constructing a sub-query for a node in Q is $\Theta(|U|^2)$. Hence, the time complexity for creating an encrypted query is $\Theta(|Q| \times |U|^2)$, where $|Q|$ is the number of nodes involved in Q. Every sub-query of a TPQ needs to be evaluated with the index of each XML document in the database. Thus, the evaluation can be done in $\Theta(|Q| \times |U| \times N)$ time. In general $N >> |U|$, so the online step has a time complexity of $\Theta(|Q| \times |U| \times N)$.

Now let us consider the space complexity. Each XML document is encrypted and stored at Charlie. If a standard encryption algorithm such as AES is adopted, the storage cost for the encrypted XML documents is the same for all the possible solutions. Therefore, we focus on the additional space cost due to the secure indexing. Given an XML document, it is easy to see that the index on the structure (content) requires space in $\Theta(|U|)$. If the total number of XML documents is N, then the space complexity for indexing is $\Theta(N \times |U|)$.

3.5 Privacy Analysis

According to Section 3.3, a TPQ is decomposed into a set of sub-queries, each for one node in the TPQ. More precisely, for a node x with content predicate $[x, op, \alpha]$, where $op \in \{>, \geq, <, \leq =\}$ and α is a value, a sub-query is constructed for this predicate (case 1 in *Query construction*). However, when node x does not have the content predicate, a sub-query is also constructed as if there were a content predicate $[x, \geq, 0]$ (case 2 in *Query construction*). Therefore, each sub-query can be generalized to the form of $[x, op, \alpha]$. A TPQ is a combination of the sub-queries for such predicates. Thus, in the following, for the simplicity of the analysis, we discuss these predicates, instead of the TPQ directly.

Suppose that $[x, op, \alpha]$ is a predicate, and that the position assigned to x is $pos(x) = \lambda$. To evaluate the query represented by the predicate, we generate the following two vectors (see Section 3.3):

$$Q_{1\lambda} = (\gamma_1, \gamma_2, \cdots, \gamma_{\lambda-1}, \alpha - s, \gamma_{\lambda+1}, \cdots, \gamma_{|U|}), \text{ and}$$
$$Q_{2\lambda} = (\gamma_1, \gamma_2, \cdots, \gamma_{\lambda-1}, \alpha + s, \gamma_{\lambda+1}, \cdots, \gamma_{|U|}),$$

where s and γ_i ($i = 1, 2, \ldots, \lambda - 1, \lambda + 1, \ldots, |U|$) are positive random numbers. Let A_D be a vector (i.e., the index) generated for an XML document. By checking whether A_D is equally close to $Q_{1\lambda}$ and $Q_{2\lambda}$, the server can decide whether $A_D[\lambda] = \alpha$. Such a comparison allows the server to learn which outsourced documents satisfy the formula $A_D[\lambda] = \alpha$, although the server knows neither $A_D[\lambda]$ nor α, which have been encrypted by ASPE. We remark that such kind of information disclosure is the so-called access pattern [10], which is difficult to prevent and almost all efficient private keyword search schemes leak this information.

Our protocol in Section 3.3 supports range queries. Given a predicate $[x, op, \alpha]$, its related vectors $Q_{1\lambda}$ and $Q_{2\lambda}$, and the vector A_D of an XML document, the server knows whether $A_D[\lambda] < \alpha$ by checking whether A_D is closer to $Q_{1\lambda}$ than $Q_{2\lambda}$. Therefore, after the evaluation of the sub-query corresponding to predicate $[x, op, \alpha]$, the server is able to partition all the outsourced XML document into three disjoint subsets: 1) documents with x value equal to α, 2) documents with x value less than α, and 3) documents with x value greater than α. However, in any of the latter two subsets, the server cannot order the documents in either ascending or descending order of their x values.

Lemma 1. Suppose that $[x, op, \alpha]$ is a predicate, and the order of x is $pos(x) = \lambda$. Let A_{D_1} and A_{D_2} be two data points, such that $A_{D_1}[\lambda] < \alpha$ and $A_{D_2}[\lambda] < \alpha$. The server cannot determine whether or not $A_{D_1}[\lambda] > A_{D_2}[\lambda]$.

Proof. Let $Q_{1\lambda}$ and $Q_{2\lambda}$ be the points generated according to the predicate. Notice that $A_{D_1}[\lambda] > A_{D_2}[\lambda]$ if only if $\mathsf{d}^2(Q_{2\lambda}, A_{D_2}) - \mathsf{d}^2(Q_{1\lambda}, A_{D_2}) > \mathsf{d}^2(Q_{2\lambda}, A_{D_1}) - \mathsf{d}^2(Q_{1\lambda}, A_{D_1})$. But since $\mathsf{d}^2(Q_{2\lambda}, A_{D_2}) - \mathsf{d}^2(Q_{1\lambda}, A_{D_2})$ and $\mathsf{d}^2(Q_{2\lambda}, A_{D_1}) - \mathsf{d}^2(Q_{1\lambda}, A_{D_1})$ are protected by different random values, i.e., $(\overline{Q_{1\lambda}} - \overline{Q_{2\lambda}})^T \widehat{A_{D_1}} = 0.5 r_1(\mathsf{d}^2(Q_{2\lambda}, A_{D_1}) - \mathsf{d}^2(Q_{1\lambda}, A_{D_1}))$ and $(\overline{Q_{1\lambda}} - \overline{Q_{2\lambda}})^T \widehat{A_{D_2}} = 0.5 r_2(\mathsf{d}^2(Q_{2\lambda}, A_{D_2}) - \mathsf{d}^2(Q_{1\lambda}, A_{D_2}))$, the cloud server cannot determine whether or not $A_{D_1}[\lambda]$ is greater than $A_{D_2}[\lambda]$. □

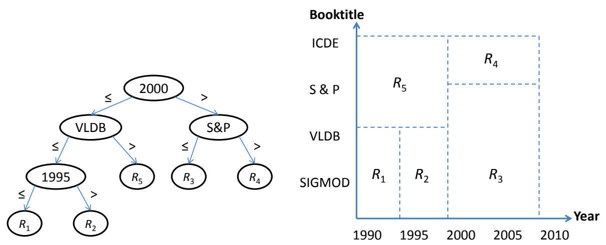
Note that in the scenario where only the function of equality matching is needed, we can actually prevent the server from learning whether or not $A_D[\lambda] < \alpha$ by randomizing the order of $\mathsf{E}_1(Q_{1\lambda})$ and $\mathsf{E}_1(Q_{2\lambda})$ before sending them to the server. In such a way, the equality comparison is still possible by checking whether A_D is at the same distance from $Q_{1\lambda}$ and $Q_{2\lambda}$. However, since the server cannot distinguish $\mathsf{E}_1(Q_{1\lambda})$ from $\mathsf{E}_1(Q_{2\lambda})$, it cannot tell whether $A_D[\lambda] < \alpha$.

4 The Efficiency Improvement

In the last section, we present a solution, which requires a TPQ to be evaluated with each XML document in the outsourced database. To improve its computational efficiency so that it can be well scaled for large data sources, in this section we propose an optimization technique, which consists of two phases: 1) partition XML documents by DTD, and 2) build a k-d tree on each partition.

In the first phase, the optimization technique partitions the XML documents by their DTDs, such that in each resultant partition all the documents have the same DTD. It treats each resulting partition independently, as if each one were an independent outsourced database. In particular, for each partition it builds independently a domain hierarchy, based on which the secure indices for all the documents in the partition are constructed. Given a TPQ, then all the documents in irrelevant partitions can be pruned. Clearly, this improves the query efficiency. Furthermore, it also saves storage space for secure indices, since these indices are built according to the domain hierarchies, which are smaller than the complete domain hierarchy.

Example 3. Suppose that there are three DTDs in an outsourced database, and Figure 2 is the domain hierarchy built on these three DTDs. Suppose that the first DTD corresponds to $\mathsf{subtree}(a)$, i.e., the subtree rooted at node a in Figure 2, the second DTD corresponds to $\mathsf{subtree}(b)$, and the third corresponds to $\mathsf{subtree}(c)$. Then, by the above optimization technique, the outsourced database is divided into three partitions, and $\mathsf{subtree}(a)$, $\mathsf{subtree}(b)$, and $\mathsf{subtree}(c)$ become their domain hierarchies, each exclusively for one partition. Now consider the TPQ in Figure 3(b). Before the optimization, it needs to be evaluated with the documents in the whole database. With the optimization, it only needs to be evaluated with the documents in the first partition, since it can only be embedded in $\mathsf{subtree}(a)$.



(a) A k-d Tree (b) The Space Partition
Figure 5: An Example of k-d Tree Partitioning

The second phase is a further efficiency improvement on the basis of the first one. After partitioning the XML documents according to DTDs, the number of documents in a partition might still be large. To further improve the performance, the data owner builds a k-d tree for each partition, such that documents with similar node values are clustered in the same leaf in a k-d tree.

A k-d tree (short for k-dimensional tree) is a binary tree for indexing k-dimensional data points. It recursively partitions the data space into two subspaces by a $(k-1)$-dimensional hyperplane. Each node n in the tree is associated with one dimension ax and a splitting point sp on the axis of ax. In general, sp is the median of all the ax values in node n. All the data points in the left subtree of node n have an ax value less than or equal to sp, and all the data points in the right subtree have an ax value greater than sp. In this way, the splitting point sp actually sets a $(k-1)$-dimensional hyperplane perpendicular to the axis of ax. The hyperplane splits the space covered by node n into two (i.e., one covered by the left subtree and the other by the right subtree). Usually, the dimension to be associated with a node is decided in a round-robin way. That is, a node at level dp is associated with the i-th dimension, where $i = dp \bmod (k) + 1$.

To build a k-d tree for a partition of XML documents that are created according to the same DTD, we select k nodes from the DTD to act as the k dimensions. The node values can be numerical or categorical. If the values in a dimension are numerical, we can use them directly. For categorical values, we need to map them into integers. There are various methods for such mapping. In the following we give some possible methods. If the categorical values in a dimension are organized semantically in a hierarchy, we can assign a distinct integer to each value by the pre-order traversal of the hierarchy as in [6]. If such a hierarchy is unavailable, we may sort the categorical values in the ascending order of the number of XML documents containing them, and then assign sequential integers to them. In addition, we may also consider generalizing categorical values before assigning integers (e.g., names, like John, Jane, and Jack, starting by letter 'J' can be generalized to 'J*').

Example 4. Consider the set of XML documents in the *Inproceedings* dataset that record the publications of authors. In an XML document, there is a '`year`' node indicating when the work was published, and there is also a '`booktitle`' node indicating where the work was published. We can take '`year`' and '`booktitle`' as the first and the second dimension, respectively, and build a k-d tree on them. Thus, here $k = 2$. As shown in Figure 5(a), the data is first partitioned on '`year`', then on '`booktitle`', and then on '`year`' (i.e., in a round-robin way). Figure 5(b) shows the leaf nodes in the k-d tree, which forms a partitioning of XML documents of the *Inproceedings* dataset.

The k-d tree is shared between the data owner and the user, but hidden from the cloud server. Given a TPQ, the user can refer the server to a subset of leaf nodes (in the k-d tree), which might contain matching documents. The evaluation in all the remaining leaf nodes is pruned. In Example 4, if Alice is interested in *SIGMOD* papers from 1992 to 1994, then only the documents in partition R_1 should be evaluated.

Although the k-d tree can improve the query efficiency as shown above, it may potentially leak sensitive information to the third party. Therefore, we adopt privacy metrics to guide its construction. So far the proposed privacy metrics can be divided into two categories: *syntax-based* and *mechanism-based*. The former defines the format that the sanitized dataset should comply with. Its representatives include k-anonymity [31], ℓ-diversity [23], t-closeness [22], and β-likeness [5]. The latter, instead of imposing any constraint on the dataset to be published, specifies requirements on the mechanism, which releases the data. Its popular representative is the differential privacy [13]. In the following, we adopt J-S divergence, an instantiation of t-closeness, to guide the anonymity of the k-d tree in Section 4.1, while in Section 4.2, we use differential privacy.

4.1 k-d Tree Guided by J-S Divergence

The use of a k-d tree does not change the query results. Thus, the server does not gain extra information from the query results alone. However, the k-d tree allows the server to prune leaf nodes, which definitely do not contain XML documents satisfying the query constraints. Such pruning potentially makes it possible for the server with certain background knowledge to learn more about a query. Consider once again Example 4. Suppose that the year frequencies for '1992' and '2002' are close to 10%. Given a query on year 2002, then around 10% XML documents appear in the query result. The cloud server with approximate background knowledge about the year distribution can guess that the query is for either year 1992 or 2002. Still, it cannot determine which one of 1992 and 2002 is correct. However, with the k-d tree the server is directly referred to partition R_3 for the query processing. If the server also has the background knowledge of the partition (e.g., the extent of R_3 and the data distribution within it), then it can infer the query is for year 2002.

The above inference is possible, because the distribution of year values in partition R_3 is *different* from that in the whole dataset. J-S divergence is a well-known method to measure the similarity of two probability distributions. In the following we apply it to control the data distribution of leaf nodes, and thus to limit the disclosure of sensitive information.

Given a node x in the XML documents, let $\{v_1, v_2, \ldots, v_m\}$ be its value domain. Suppose that the global distribution of x in the whole dataset is $G = (G[1], G[2], \ldots, G[m])$, where $G[i]$ is the probability of v_i in the whole dataset and $i = 1, 2, \ldots, m$. Furthermore, suppose that the local distribution of x in a leaf node of the k-d tree is $L = (L[1], L[2], \ldots, L[m])$, where $L[i]$ is the probability of v_i in the leaf node and $i = 1, 2, \ldots, m$. The J-S divergence to measure the difference between G and L is:

$$\mathsf{JS}(G, L) = \frac{1}{2}\mathsf{KL}(G, M) + \frac{1}{2}\mathsf{KL}(L, M), \tag{4}$$

where $M = \frac{1}{2}(G + L)$, $\mathsf{KL}(G, M) = \sum_i \ln\left(\frac{G[i]}{M[i]}\right) \cdot G[i]$ is the K-L divergence between G and M, and $\mathsf{KL}(L, M) = \sum_i \ln\left(\frac{L[i]}{M[i]}\right) \cdot L[i]$ is K-L divergence between L and M. Intuitively, if J-S divergence is smaller, then G and L are more similar and referring the server to one partition leaks less information. Based on the J-S divergence, we give the following definition, which decides whether a node in the k-d tree can be further split.

Definition 4 (Splitting eligibility)**.** Let $x_1, x_2, \ldots, x_k$ be k nodes, which constitutes the k dimensions of a k-d tree. Suppose that G_j is the global distribution of x_j values in the whole dataset, and that t_j is a threshold, where $j = 1, 2, \ldots, k$. Then, a node in the k-d tree can be split into two children C^1 and C^2 along the j-th dimension, only if $\mathsf{JS}(G_j, L_j^1) \le t_j$ and $\mathsf{JS}(G_j, L_j^2) \le t_j$, where L_j^1 and L_j^2 are the local distributions of x_j in C^1 and C^2, respectively.

4.2 k-d Tree Guided by Differential Privacy

Differential privacy requires that the data of any particular person should not have obvious impact on the query output. Intuitively, it ensures the privacy of individuals, since whether a person's data is in the dataset or not, the inferred knowledge by an attacker is the essentially the same. In the following we first introduce some background knowledge about differential privacy, before applying it to the k-d tree.

Definition 5. $\mathcal{D}_1$ and $\mathcal{D}_2$ are two neighboring datasets, if $\mathcal{D}_1 = \mathcal{D}_2 \cup \{t\}$ or $\mathcal{D}_2 = \mathcal{D}_1 \cup \{t\}$, where t is a tuple and $\mathcal{D}_2 \cup \{t\}$ (or $\mathcal{D}_1 \cup \{t\}$) represents the resulting dataset after adding tuple t to $\mathcal{D}_2$ (or $\mathcal{D}_1$). We use $\mathcal{D}_1 \simeq \mathcal{D}_2$ to denote that $\mathcal{D}_1$ and $\mathcal{D}_2$ are two neighboring datasets.

Given a function $f : D \to \mathbb{R}$, its L_1 *sensitivity* is defined to be

$$\sigma(f) = \max_{(\mathcal{D}_1,\mathcal{D}_2):\mathcal{D}_1 \simeq \mathcal{D}_2} |f(\mathcal{D}_1) - f(\mathcal{D}_2)|.$$

In the case that f is a count query, $\sigma(f) = 1$ since for any pair of neighboring datasets $\mathcal{D}_1$ and $\mathcal{D}_2$, the values of $f(\mathcal{D}_1)$ and $f(\mathcal{D}_2)$ only differ by 1.

Definition 6. Let $\mathcal{D}_1$, $\mathcal{D}_2$ be any two neighboring datasets, $\mathcal{A}$ a randomized algorithm on the datasets, and S be any subset of the output domain of $\mathcal{A}$. Algorithm $\mathcal{A}$ is said to be ε-differentially private if

$$\Pr[\mathcal{A}(\mathcal{D}_1) \in S] \leq e^{\varepsilon} \Pr[\mathcal{A}(\mathcal{D}_2) \in S].$$

The most commonly used technique for designing algorithms that satisfy differential privacy was proposed by Dwork et al. in [13], which is also called the *Laplace mechanism*. Let $f(\mathcal{D})$ denote a function on a dataset $\mathcal{D}$. An ε-differentially private mechanism for releasing $f(\mathcal{D})$ is to publish $\mathcal{L}(\mathcal{D}) = f(\mathcal{D}) + X$, where X is a random variable drawn from the Laplace distribution $Lap(\sigma(f)/\varepsilon)$.

A second mechanism to achieve differential privacy is the *exponential mechanism* by McSherry and Talwar [27]. To apply this mechanism, a score function $q : \mathcal{D} \times \mathcal{R} \to \mathbb{R}$ is defined, where $\mathcal{D}$ is the input dataset and $\mathcal{R}$ is the output domain of the mechanism. Given a value $r \in \mathcal{R}$, the score $q(D, r)$ shows the closeness of r to the ideal output—the higher the score is, the closer r is to the ideal output. Given a dataset D, an exponential mechanism M satisfying ε-differential privacy outputs r with a probability of

$$\Pr[M(D) = r] \propto \exp\left(\frac{\varepsilon}{2\sigma(q)} q(D, r)\right),$$

where

$$\sigma(q) = \max_{(\mathcal{D}_1,\mathcal{D}_2):\mathcal{D}_1 \simeq \mathcal{D}_2} \{\max_{\forall r \in \mathcal{R}} |q(\mathcal{D}_1, r) - q(\mathcal{D}_2, r)|\}.$$

Composability. Differential privacy has the property of *composability*. Given a set of mechanisms, which satisfy differential privacy with respect to $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$, respectively, they satisfy as a whole ε-differential privacy where $\varepsilon = \sum_{i=1}^{n} \varepsilon_i$. Parameter ε is usually called the total *privacy budget*. Given a task consisting of multiple steps, to satisfy ε-differential privacy, a portion of privacy budget is assigned to each step such that the summation of all the portions is upper-bounded by ε.

To build a k-d tree satisfying differential privacy, we adopt the technique proposed by Cormode et al. in [9]. Two types of queries are involved when building the k-d tree. a) *Count query*, which is to release the sizes of the nodes in the tree. During the building process, to ensure each node size used satisfies differential privacy, Laplace mechanism is adopted and appropriate noise is added to the true node size. In particular, if a privacy budget of ε_i is assigned to a node, then the Laplace noise drawn from $Lap(\frac{1}{\varepsilon_i})$ is added to the true node size. b) *Median query*, which is to identify the splitting points of the nodes in the tree. The k-d tree structure may potentially disclose private information. To ensure that tree structure satisfies differential privacy, exponential mechanism is applied to select the splitting points. Specifically, let ε_i be the privacy budget allocated to split a given node on dimension ax. Suppose that r is in the range of ax values in the node and x_m is the median of all ax values in the node. Then the exponential mechanism returns the splitting point r with the probability $\Pr[r] \propto \exp\left(-\frac{\varepsilon_i}{2}|rank(r) - rank(x_m)|\right)$, where $rank(r)$ and $rank(x_m)$ are the ranks of r and x_m in the node, respectively.

Next, we briefly explain how they build such a k-d tree in a differentially private way [9]. At the very beginning, given the total privacy budget ε, the algorithm constructing the k-d tree would allocate a fixed percentage mb of ε for the method that outputs the private medians, and the rest of the private budget will be used for the private count method. The algorithm starts by trying to split the root node containing all the tuples in the input dataset. More precisely, the algorithm first makes a call to the private count method to get a noisy count of the tuples in the root node, which is then appended to the tail of a FIFO queue. Depending on the noisy count and the current height of the node removed from the head of the queue, the algorithm would either 1) stop the splitting process of that node if the noisy count is less than or equal to *splittingThreshold* or the current height of that node has reached *maxHeight*, or 2) split that node into subnodes according to the private median method. For each subnode created, a subsequent call to the noisy count method would be made and the subnode is then added to the tail of the queue. This process continues until the FIFO queue is empty, i.e., all the leaf nodes of the k-d tree are identified.

Since we adopt the *hybrid tree* proposed in [9], there is one additional parameter *switch-Level* that could be adjusted, which is used to indicate when the algorithm should switch from the data-dependent decompositions to the deta-independent decompositions. To be specific, after level *swithLevel*, instead of calling the method to choose a private median, the algorithm will divide a node into regions of equal size via the midpoint induced by the tuples within the node along the current axis.

As for the strategies for allocating privacy budget along each root-to-leaf path, authors in [9] proposed to use the *geometric budgeting* strategy: from a parent node to a child node, the privacy budget increases geometrically by a factor of $\sqrt[3]{2}$. The interested reader is referred to [9] for their thorough discussions regarding the strategies and the method they deploy to boost the range query accuracy.

One thing to note is that the counts for the leaf nodes in a k-d tree described above are noisy ones. If the added noise is positive, then the data owner has to create some *fake* XML documents residing in the bounding box of that leaf node and then encrypt them by the secret key shared with the data user. Data owner also needs to mark all the *fake* XML documents so that when they are decrypted later by the data user, they can be pruned. On the other hand, if the added noise is negative, the data owner has to randomly select documents within the leaf node and suppress them. We can see that the use of such a k-d tree would result in false positive documents returned to the data user and false negative documents suppressed by the data owner.

5 Multiple Embeddings of a TPQ

Given a domain hierarchy with Dewey labels, it is possible that a TPQ has multiple encodings. This happens mainly because more than one node in the domain hierarchy may have the same name, although they have different Dewey labels. Consider the domain hierarchy in Figure 6. Suppose that we have a TPQ with a single node `author`. In this case, the TPQ can be mapped to any of the 6 `author` nodes in the domain hierarchy. As a consequence, we have altogether 6 embeddings. In general, the number of embeddings of this kind of query is not large, i.e., it is upper-bounded by $|U|$, the size of label domain.

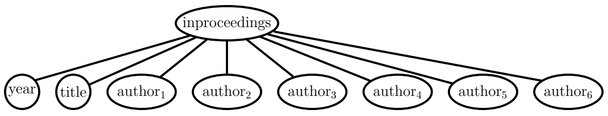


Figure 6: An Example Domain

In a more complicated case, a TPQ has multiple nodes, some of which can be mapped to multiple nodes with the same names in the domain hierarchy. If this is the case, some mechanism needs to be developed to avoid the generation of exponentially many embeddings for the TPQ. Consider once again the domain hierarchy in Figure 6. Suppose that Figure 7 is a TPQ. Since the `inproceedings` element has 6 child elements of `author`, this query has $\binom{6}{2} = 15$ different embeddings. One way to deal with this problem is for the data user to give the specific positions (i.e., by position predicate $[position() = m]$), to which such query nodes should be mapped. For instance, once the user requires that the two authors in the TPQ should be mapped to the first two authors in the domain hierarchy, then there is only one embedding for the query. Alternatively, the user can also decompose the original query into multiple sub-TPQs. For instance, the user can decompose the query in Figure 7 into the two sub-TPQs in Figure 8. The sub-TPQ in Figure 8(a) has 6 embeddings, corresponding to 6 content predicates: $\pi_i = [\texttt{author}_i, =, \text{“Michael”}]$ for i from 1 to 6. Similarly, the sub-TPQ in Figure 8 (b) also has 6 embeddings, corresponding to 6 content predicates, i.e., $\tau_j = [\texttt{author}_j, =, \text{“John”}]$ for j from 1 to 6. The third party now should return those documents such that $(\bigvee_i \pi_i) \bigwedge (\bigvee_j \tau_j)$ evaluates to true. Note that right now we only have 12 embeddings in total after the query decomposition.

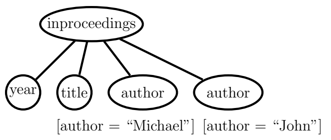
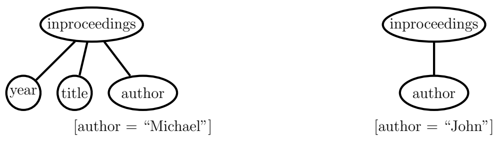


Figure 7: An Example Query



(a) Split Query Tree 1 (b) Split Query Tree 2

Figure 8: An Example of Query Decomposition

Just like a TPQ, an XML document may also have multiple embeddings in the domain hierarchy. However, we only need to consider one possible embedding for each XML document. This suffices for the matching between a TPQ and an XML document, because we have already enumerated all the possible embeddings of the TPQ. In our work, for each XML document, we only use its left-most embedding.

6 Experimental Report

In this section we give a thorough experimental evaluation of the proposed scheme. We use 3 XML datasets (Table 1). The first dataset is *Inproceedings*, which is a subset of DBLP dataset [24]. Each subtree under the element node `Inproceedings` in the DBLP dataset represents a publication, which appears in a major computer science journal or conference. We take each such subtree as an XML document. Altogether this set has 205,404 documents. *Stock Quotes* is a randomly generated Nasdaq stock quotes [25] containing information about 5,000 stocks. The third dataset is *University Courses* [26], which records the course data from the websites of Reed College. It contains 703 documents. In the following experiments, unless otherwise specified, we will use *Inproceedings* dataset as the default dataset. The prototype of our solution is implemented in Java, and the experiments were carried out on an Intel Core i7-2600 3.40GHz CPU machine with 8G bytes memory running Linux 3.4.13.

Table 1: The Datasets

Dataset	Num. of Documents	Subtree size
Inproceedings	205,404	91
Stock Quotes	5,000	19
University Courses	703	15

6.1 Index Building

We build the secure indices for XML documents through the following three steps: 1) building a domain hierarchy, 2) generating for each XML document a vector (i.e., an index), which encodes structural and textual information about the document, and 3) using ASPE to encrypt the vectors.

Each dataset in the experiment has a DTD. We build the domain hierarchy by adding a root node over the three DTDs of the three datasets (Section 3.1). Thus, each DTD becomes a subtree under the root node of the domain hierarchy. A publication in *Inproceedings* dataset generally contains multiple authors, and the number of authors varies from one publication to another. We scanned the whole dataset, and found that most of the publications have at most 10 authors. Thus, in the subtree representing the DTD of *Inproceedings*, we duplicate the `author` node 10 times. After this expansion, this subtree contains 91 nodes. The sizes of another two subtrees, which represent the DTDs of another two datasets (i.e., *Stock Quotes* and *University Courses*), are 19 and 15, respectively (Table 1). As a result, the domain hierarchy contains $91 + 19 + 15 = 125$ nodes (excluding the root). We label the domain hierarchy by Dewey labeling, and order the Dewey labels.

Once the domain hierarchy is ready, we start to encode each XML document into a vector. We first embed each XML document in the domain hierarchy. In particular, we use the algorithm proposed in [1]. An XML document is decomposed into edges. For each of such edges, the algorithm finds its potential matching edges in the domain hierarchy. All these potential matching edges are joined together. A join result is a valid embedding, if the PC/AD relationships among its nodes are consistent with those in the XML document. Among all the possible embeddings for an XML document, we select the left-most one (Section 5). Then, according to the Dewey labels and their ordering in the domain hierarchy, a 125-dimensional vector is built for the XML document. Table 2 shows the encoding time for each dataset.

Table 2: Time for Building Secure Indices (ms)

	Inproceedings	Stock Quotes	University Courses
Encoding	277,680	8,849	2,005
Encryption	198,306	5,591	825

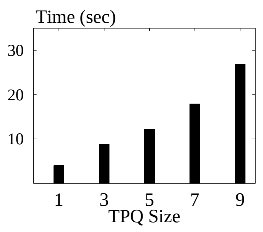
We use ASPE [35] (Section 3.2) to encrypt the vectors (i.e., the indices) of the XML documents. Table 2 gives the elapsed time of the encryption for each dataset. If we consider Table 1 and Table 2 together, we can see that the encryption time grows linearly as a function of the dataset size. This is consistent with the analysis (in Section 3.3), which says the time complexity of encrypting a vector is $\Theta(|U|^2)$. Here, $|U| = 125$.

6.2 Query Evaluation without k-d Tree

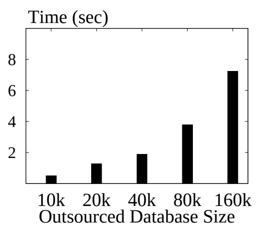
In the basic scheme, the secure indices for all the three datasets are put together. Given a TPQ, it is evaluated against each encrypted index.

We first analyze the effect of TPQ size on the query efficiency. We generate 5 TPQs containing $1, 3, 5, 7$, and 9 nodes, respectively. Vectors for these 5 TPQs are then created according to the steps of *query construction* in Section 3.3. Figure 9(a) shows the result, in which the elapsed query processing time grows linearly as a function of the TPQ size. This happens, because a sub-query has to be created for each node in a TPQ.

Next, we fix the TPQ size to 3, and examine the impact of the outsourced database size on the query efficiency. The complete outsourced database in our experiments consists of three datasets—*Inproceedings*, *Stock Quotes*, and *University Courses*. We generate 5 databases by randomly sampling 10k to 160k documents from the complete database. As expected, the elapsed time for the query processing increases as a function of database size (Figure 9 (b)).



(a) Varying TPQ Size (3 Datasets Combined)



(b) Varying Database Size

Figure 9: Varying TPQ Size and Database Size

6.3 Query Evaluation with k-d Tree Built by J-S Divergence

To improve the query efficiency (Section 4), we can partition the XML documents by DTDs. However, the resultant partition for *Inproceedings* in our experiments is still quite large (i.e., 205,404 documents), and thus the query processing on it is still time consuming. To further boost the efficiency, we employ a k-d tree, which splits the *Inproceedings* into smaller partitions. Node `year` and node `booktitle` exist in the XML documents in the *Inproceedings*. The value of `year` ranges from 1959 to 2002, and there are 1,744 distinct values for `booktitle`. We take these two nodes as the dimensions of a k-d tree, where $k = 2$. We

Table 3: Change of $t_{booktitle}$ when $t_{year} = 0.6$

$t_{booktitle}$	# of Leaves
0.500	17
0.525	24
0.550	33
0.575	46
0.600	71
0.625	103
0.650	179
0.675	456
0.700	2,406

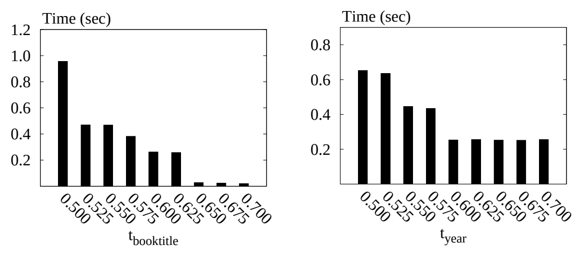
Table 4: Change of t_{year} when $t_{booktitle} = 0.6$

t_{year}	# of Leaves
0.500	16
0.525	30
0.550	48
0.575	61
0.600	71
0.625	74
0.650	74
0.675	74
0.700	74

use the *splitting eligibility* condition specified in Definition 4 to determine whether a node in the k-d tree can be further split. For the dimension associated with `year`, we set threshold t_{year}, which requires that the local distribution of `year` values in a k-d tree node should not be different (by J-S divergence) from that in the whole *Inproceedings* dataset by more than t_{year}. In a similar fashion, we set the threshold $t_{booktitle}$ for dimension `booktitle`.

With the k-d tree constructed, each point query with constraints on `year` and `booktitle` will be evaluated only with the XML documents in a leaf of k-d tree. This improves the efficiency. However, the distributions of `year` and `booktitle` could vary from one tree node to another. Thus, to satisfy the splitting eligibility condition, the leaf size in a k-d tree could be different from one leaf to another. To better measure the evaluation time, we randomly generate 100 queries of 5 predicates, two of which are constraints on the dimension of `year` and `booktitle` and the constraints on those two dimensions are uniformly generated over the domains of `year` and `booktitle`.

To see the improvement effect, we first fix the J-S divergence threshold for `year` t_{year} to 0.6 and vary the threshold $t_{booktitle}$ for `booktitle`. Table 3 lists the total number of leaves for each configuration of $t_{booktitle}$. It can be seen that the total number of leaves increases, when the J-S divergence increases. This is as expected, since larger divergence threshold is more tolerant of the difference between the global and local distributions of `booktitle` and more leaves could be generated. As the number of leaves increases, the leaf size on average decreases instead. Thus, the query efficiency, which is linear to the leaf size, is improved. Figure 10(a) supports this. In addition, it also shows that the efficiency improvement is more obvious when $t_{booktitle} > 0.625$, since beyond that threshold value the number of leaves increases in a steep fashion.



(a) (b)

Figure 10: Varying J-S Divergence

Next, we fix $t_{booktitle}$ to 0.6 and vary t_{year}. The number of resulting leaves in the k-d tree for each configuration of t_{year} is given in Table 4. The number of leaves generated stops increasing when t_{year} reaches 0.625. This results from the strict constraints we impose on $t_{booktitle}$, which prevents the k-d tree nodes from being further split. The evaluation time given in Figure 10(b) also reflects this fact; it stops decreasing after t_{year} reaches 0.625. We also note that varying J-S threshold would also affect the efficiency of building a k-d tree. In general, higher J-S thresholds would lead to higher time needed to build a k-d tree. According to our experiments, it takes at most 2,676 ms to build a k-d tree.

In addition to the point queries, we also consider the effect of k-d tree on the efficiency improvement for range queries. We generate 7 range queries of 5 predicates. Out of those 5 predicates, 4 of them are constraints on the dimensions of `year` and `booktitle`, i.e., the lowerbound and upperbound of `year` and `booktitle`, respectively. The last one is a constraint on the dimension of `title`. The minimum bounding box for each range query and the number of XML documents that need to be compared are given in Table 5. Figure 11 reports the elapsed time, where selectivity denotes the proportion of documents involved in the evaluation of a given query. As the selectivity increases, the number of XML documents needed to be evaluated for a query increases. Thus, the time cost also increases.

Table 5: Range Information for Different Range Queries ($t_{booktitle} = 0.625$ and $t_{year} = 0.625$)

year	booktitle	# documents
[1968,1989]	[1,1388]	7,879
[1968,1992]	[1,1388]	13,765
[1968,1992]	[1,1631]	27,453
[1968,1995]	[0,1631]	46,399
[1968,1996]	[0,1696]	71,006
[1959,1996]	[0,1743]	105,888
[1959,2002]	[0,1743]	205,404

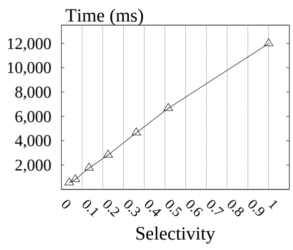


Figure 11: Varying Query Ranges

6.4 Query Evaluation with k-d Tree Built by Differential Privacy

We now study the effectiveness of our method optimized via a k-d tree, which is built under the constraints of differential privacy. Since noises are added, the computed query results by the cloud server are no longer precise. To measure their accuracy, we adopt two metrics:

$$\text{Recall} = \frac{tp}{tp + fn},$$

and

$$\text{Precision} = \frac{tp}{tp + fp},$$

where tp is *true positive* (i.e., the number of relevant documents in the query result), fn is *false negative* (i.e., the number of relevant documents not in the query result), and fp is *false positive* (i.e., the number of irrelevant documents in the query result). Basically, Recall is to measure the fraction of documents, which are relevant to the query and successfully retrieved, and Precision is the fraction of the retrieved documents that are relevant to the query. The approach [9] we apply to generate the differentially-private k-d tree has a few parameters as listed in Table 6. By default we set $\varepsilon = 1.0$, $splittingThreshold = 2^{10}$, $maxHeight = 8$, $switchLevel = 5$, and $mb = 0.3$. In the following we tune these parameters to study their effect on Precision and Recall.

Table 6: The Parameters in [9]

Parameter	Function	Value
ε	The total privacy budget	0.2 0.4 0.6 0.8 **1.0** 1.2 1.4 1.6 1.8 2.0
$splittingThreshold$	The minimum size of a splittable node	2^6 2^7 2^8 2^9 $\mathbf{2^{10}}$ 2^{11} 2^{12} 2^{13} 2^{14}
$maxHeight$	The maximum height of the k-d tree	1 2 3 4 5 6 7 **8** 9 10
$switchLevel$	The level from data-dependant splitting to data independent	1 2 3 4 **5** 6 7 8
mb	The percentage of privacy budget for splitting	0.1 0.2 **0.3** 0.4 0.5 0.6 0.7 0.8 0.9

We first vary ε. Figure 12 shows that both the recall and precision rates increase as a function of ε. This is as expected. As ε is higher, the added noise to each leaf node in the k-d tree is lower. Thus, the number of suppressed tuples due to negative noise is lower and the recall rate is higher. At the same time, the number of fake tuples due to positive noise is also lower and the precision rate is higher. The experimental result also shows a trade-off between privacy and performance—relaxed privacy with higher ε value gives better performance of our proposed approach. However, as pointed out in [14], choosing an appropriate ε value is by no means an easy task for the laymen. Recent expositions by Lee et al. [20, 21] attempt to give some guidelines for selecting a proper ε such that the posterior belief in a particular individual's presence in a dataset given the query result is upper-bounded by a user defined threshold. On the other hand, the work in [17] derives a lower bound on how much noise should be added to ensure the privacy of sensitive data. We refer the interested readers to the papers above since the setting of ε is not in the scope of our work.

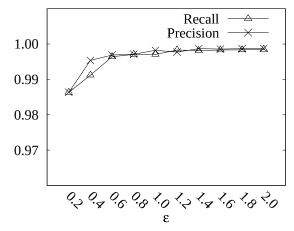
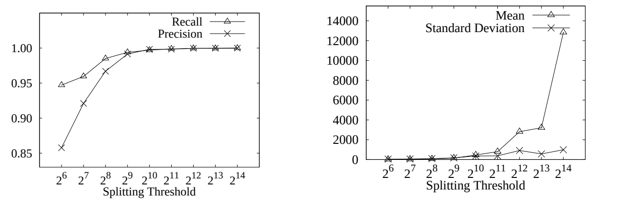


Figure 12: Varying Privacy Budget ε

Next, we vary *splittingThreshold*. If the noisy count of a node in the k-d tree is smaller than this threshold, further splitting of the node stops. Hence, $splittingThreshold$ controls the sizes of leaves. When its value increases, the leaf nodes become bigger, and therefore, the ratio of the added Laplace noise to leaf size is smaller, resulting in higher recall and precision rates (Figure 13(a)). An interesting point in the figure is at $splittingThreshold = 2^{10}$. When the threshold is beyond 2^{10}, the recall and precision rates are greater than 0.99. But below that, the rates are much lower. We take a closer look at the experimental outputs.

When $splittingThreshold < 2^{10}$, a big portion of leaf nodes are small compared to the magnitude of Laplace noises added to them. For instance, when $splittingThreshold = 2^6$, the average leaf node size is 22.586 while the standard deviation of the leaf node size is 46.078, indicating that the size varies a lot and the number of small leaves is big [5] For those nodes with small sizes, it is possible that most of the tuples or even all of them are suppressed due to negative noise or that a considerable amount of fake tuples are added due to positive noise. In either case, the accuracy of the query output is low. To address this issue, we vary $splittingThreshold$ and compare the mean leaf size and its standard deviation. It turns out that after $splittingThreshold$ reaches 2^{10}, the former is always greater than the latter (Figure 13(b)), indicating most of leaf nodes are 'big' enough. For example, when $splittingThreshold = 2^{10}$, the average absolute value of noise added to leaf nodes is 2.400, and 68% the leaf nodes have a size larger than 2.400.



(a) Recall and Precision (b) Noise Magnitude / Node Size

Figure 13: Varying $splittingThreshold$

We now examine $maxHeight$. Given a node, if the length of the path from the k-d tree root to it reaches $maxHeight$, further splitting of the node stops and the node becomes a leaf. From Figure 14, we can see that in general lower $maxHeight$ results in high recall and precision rates. This is reasonable, because lower $maxHeight$ results in large-sized leaf nodes in the end. Hence, the inserted noises have less effect on the query output. An interesting thing to note in Figure 14 is that the recall and precision rates are V-shaped and they reach minimum when $maxHeight$ is 5. The main reason behind is as follows. A k-d tree node at level 5 usually has a size smaller than 2^{10}. Thus, the default value (i.e., 2^{10}) of $splittingThreshold$ usually stops the node from further splitting. Consequently, when $maxHeight \geq 5$, the average root-to-leaf length in the k-d tree is around 4.57. That is, once $maxHeight \geq 5$, the tree structure becomes stable. However, as $maxHeight$ increases, the privacy budget allocated to the leaf nodes by the geometric budgeting strategy is higher. Therefore, smaller noise is added to a leaf node, and the query output is more accurate. The following is a specific example. When $maxHeight$ is set to 5, the budget allocated to level 5 is 0.192. But if $maxHeight$ is 6, the budget for level 5 is 0.142 and 0.180 for level 6. The constraint by $splittingThreshold$ usually does not allow a node to further split after it reaches level 5. Thus, in the case of $maxHeight = 6$, the privacy budget allocated to the leaf could be $0.142 + 0.180 = 0.322$, which is higher than 0.192 for the case of $maxHeight = 5$.

We also vary $switchLevel$ to study its effect on query output. $switchLevel$ controls the level, beyond which the node splitting switches from data-dependent to data-independent. The experimental results show that recall and precision rates are high (i.e., greater than 0.99) for all the cases (Figure 15). After $swithLevel$ is 4, the rates do not increase obviously. This suggests that a *switchLevel* of 4 fits with the given dataset.

[5]In this case, the average absolute value of noise added to leaf nodes is 5.592, 69% of the leaf nodes with a size smaller than 5.592

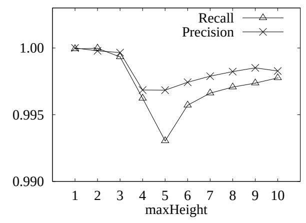


Figure 14: Varying $maxHeight$

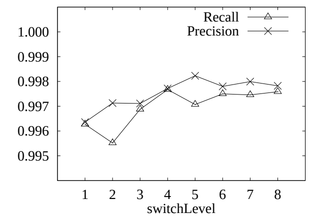


Figure 15: Varying $switchLevel$

Finally, we investigate the percentage mb of privacy budget allocated for private median selection. Figure 16 shows that higher mb results in lower recall and precision rates. This is as expected since higher mb implies a lower privacy budget for counting when creating a leaf node, which in turn results in larger magnitude of the noise generated. According to Figure 16, it is advised that at least half of the privacy budget should be allocated for private counting in order to have high recall and precision rates.

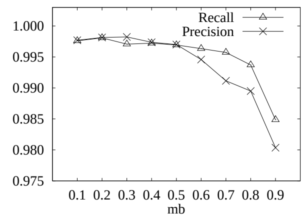


Figure 16: Varying mb

6.5 Time Overhead

In this subsection, we evaluate the efficiency of our proposed approaches. We employ two benchmarks, both of which process the queries on the plaintext XML documents. We compare our approaches with the benchmarks in terms of elapsed time. We first carry out the experiments, in which there is no query optimization, i.e., the k-d tree is not used. Figure 17 gives the results. As expected, the benchmark approach (i.e., Base) is faster than ours (i.e., Embedding).

Next, we study the efficiency improvement when a k-d tree built according to J-S divergence is used. Three schemes are involved in the experiments: a) *Embedding-JC-tree*, our scheme with a k-d tree built via J-S divergence, b) *Base*, the benchmark on plaintext XML

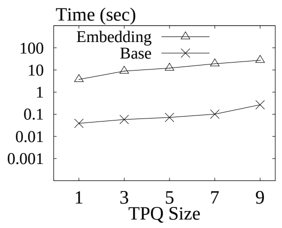
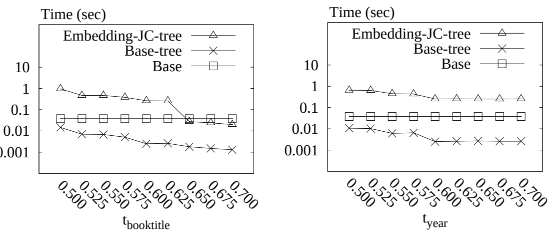


Figure 17: Varying TPQ Size

documents without the support of a k-d tree, and c) *Base-tree*, the benchmark on plaintext XML documents with the support of a k-d tree. We consider queries consisting of 5 predicates, of which two are constraints on `booktitle` and `year`. In Figure 18(a), we fix t_{year} to 0.6 and vary $t_{booktitle}$, while in Figure 18(b), we fix $t_{booktitle}$ to 0.6 and vary t_{year}. Obviously, the k-d tree optimizes the query efficiency. When $t_{booktitle} \geq 0.650$, the elapsed time of *Embedding-JC-tree* is around 38 ms, even lower than that of *Base*, which shows that our approach is efficient.



(a) Varying $t_{booktitle}$ (b) Varying t_{year}

Figure 18: Varying $t_{booktitle}$ and t_{year}

We have also evaluated the efficiency of our approach when the k-d tree is built according to differential privacy. Figure 19 reports the elapsed time of our approach *Embedding-DP-tree* and that of the other two benchmarks introduced above. We can see that *Embedding-DP-tree* is as efficient as *Base*. One thing to notice is that in Figure 19, there is an increase in time from $\varepsilon = 0.2$ to $\varepsilon = 0.4$. We check the generated k-d trees in the experiments, and find that when $\varepsilon = 0.2$, the average leaf size is 348 while it is 404 for $\varepsilon = 0.4$. This says that the k-d tree in the former case has more small leaf nodes. Therefore, the former can better prune irrelevant XML documents and has a better time performance. Still, when $\varepsilon \geq 0.6$, the structure of the k-d tree becomes stable with the average leaf size ranging from 443 to 455, and the efficiency of *Embedding-DP-tree* also becomes stable.

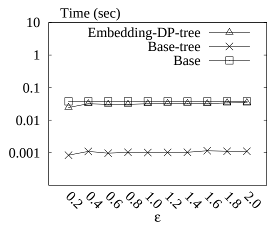


Figure 19: Varying ε

7 Related Work

Protecting data privacy in relational databases has been extensively investigated [4, 11, 15, 16, 30]. Hacigümüs et al. [15] divide each attribute domain into ranges, and group tuples into buckets by the ranges. The tuples in the buckets are encrypted, and buckets are assigned random IDs. Given a query, the buckets with tuple values potentially overlapping the query are returned to the user. The query results are refined by a post-processing step by the user to remove false positives. Damiani et al. [11] propose an indexing scheme on encrypted relational data based on direct encryption and hashing. They further analyze the tradeoff between the efficiency improved by the indexing and the extra information disclosed by it.

XML documents contain both structural and content information. Thus, secure query processing over encrypted XML documents is more challenging. Proposed solutions in this area includes [4, 34, 37]. Brinkman et al. [4] use a relational table to index the structural information of XML documents, and store the table at the third party. Such an approach thus compromises the privacy of the structure. Wang and Lakshmanan [34] selectively groups a subtree in the XML document into blocks according to user specified security constraints to hide the structural information. But it assigns each node (block) an interval, such that the interval of a child node is contained in that of its parent. Consequently, some structural information is still leaked. In addition, the approach in [34] assumes that certain nodes in the XML document are not sensitive and their contents do not need to be encrypted. However, sometimes it is difficult to decide which data is sensitive and which is not. An attacker may even be able to infer sensitive information from the partially revealed plaintext information [19]. Yang et al. [37] encode each root-to-leaf path in an XML document by a tuple. The generated tuples are then outsourced to a cloud server running a relational database system. Their solution supports secure XPath query. However, since each path is encoded independently, the server cannot process TPQ, which contains multiple correlated paths. Interested readers can refer to [32] for a more complete survey about the outsourcing of XML documents.

Query processing over encrypted graph-structured data is also related to our work. [7] is a scheme, which returns all the graphs that contain the query graph as a subgraph. It first mines a set of frequent subgraphs (from the whole graph dataset) as features. Then, given a query graph, it extracts all the features existing in the query. All the graphs in the dataset containing the features in the query are returned as possible candidates to the user. Finally, the user prunes all the false positives by a post-processing step. Such an approach can potentially be applied to search XML documents. However, it only supports structural matching, and lacks the flexibility of querying on node values. In addition, it may also incur high false positive rates: for a query not containing any feature, the whole database would be returned.

8 Conclusion and Future Work

In this paper, we propose an efficient approach to evaluate TPQs on encrypted XML documents. The key novelty is that we encode the XML documents and TPQs into vectors, such that the matching between TPQs and documents is reduced to calculating the distance between their corresponding vectors. We have also proposed optimized techniques to prune non-matched XML documents in the query processing and thus make our solution scalable well to large datasets. Furthermore, our approach supports both point and range queries.

The extensive experimental results show that it is efficient, and scales well to large datasets.

One possible direction of our future work is to deploy a proxy in the outsourcing framework. The proxy can serve as an agent between the user and cloud server, so that the user can be relieved of the burden of TPQ encoding and data decryption. Also, in our current implementation, we only support the static setting, in which all the DTDs are known to the data owner prior to the encoding of XML documents. We note that it is possible to extend our solution to tackle the case in which XML documents coming from other new DTDs are added. Therefore, supporting the extension of the domain hierarchy would be another possible future direction. Moreover, our solution only supports exact string matching via hash function. Another possible item on our agenda of future work is thus to include the functionality of fuzzy string matching. Finally, in this paper, we assume that the database is static when employing the k-d tree to boost the query efficiency. In practice, it may not always be true. Being able to deal with the continual updates of XML documents would be another meaningful yet challenging work, especially in the case of differential privacy, since the privacy budget is only limited.

9 Acknowledgment

The work reported in this paper has been funded by NSF under award CNS-1016722 and by the Air Force Office of Scientific Research under Award FA9550-08-1-0265.

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient xml query pattern matching. In *In ICDE*, pages 141–152, 2002.

[2] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. Xml path language (xpath) 2.0 (second edition). Technical report, W3C recommendation, December 2010.

[3] S. Boag, D. Chamberlin, M. F. Fernádez, D. Florescu, J. Robie, and J. Siméon. Xquery 1.0: An xml query language (second edition). Technical report, W3C Recommendation, December 2010.

[4] R. Brinkman, L. Feng, J. Doumen, P. H. Hartel, and W. Jonker. Efficient tree search in encrypted data. *Information Systems Security*, 13(3):14–21, 2004.

[5] J. Cao and P. Karras. Publishing microdata with a robust privacy guarantee. *Proc. VLDB Endow.*, 5(11):1388–1399, July 2012.

[6] J. Cao, P. Karras, P. Kalnis, and K.-L. Tan. Sabre: a sensitive attribute bucketization and redistribution framework for t-closeness. *VLDB J.*, 20(1):59–81, 2011.

[7] N. Cao, Z. Yang, C. Wang, K. Ren, and W. Lou. Privacy-preserving query over encrypted graph-structured data in cloud computing. In *ICDCS*, pages 393 –402, 2011.

[8] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of xquery. In *VLDB*, pages 237–248, 2003.

[9] G. Cormode, C. Procopiuc, D. Srivastava, E. Shen, and T. Yu. Differentially private spatial decompositions. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 20–31, Washington, DC, USA, 2012. IEEE Computer Society.

[10] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS*, pages 79–88, 2006.

[11] E. Damiani, S. D. C. di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational dbmss. In *CCS*, pages 93–102, 2003.

[12] J. Dibbern, T. Goles, R. Hirschheim, and B. Jayatilaka. Information systems outsourcing: a survey and analysis of the literature. *DATA BASE*, 35(4):6–102, 2004.

[13] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third conference on Theory of Cryptography*, TCC'06, pages 265–284, Berlin, Heidelberg, 2006. Springer-Verlag.

[14] K. E. e. Fida K. Dankar. Practicing differential privacy in health care: A review. *Transactions on Data Privacy*, 6(1):35–67, 2013.

[15] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *SIGMOD*, pages 216–227, 2002.

[16] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *VLDB*, pages 720–731, 2004.

[17] S. P. Kasiviswanathan, M. Rudelson, A. Smith, and J. Ullman. The price of privately releasing contingency tables and the spectra of random matrices with correlated rows. In *STOC*, pages 775–784, 2010.

[18] P. Kilpel*ä*inen. Tree matching problems with applications to structured text databases. Ph.D. dissertation Report A-1992-6, University of Helsinki, Finland, November 1992.

[19] A. Kundu and E. Bertino. Structural signatures for tree data structures. *PVLDB*, 1(1):138–150, 2008.

[20] J. Lee and C. Clifton. How much is enough? choosing for differential privacy. In *ISC*, pages 325–340, 2011.

[21] J. Lee and C. Clifton. Differential identifiability. In *KDD*, pages 1041–1049, 2012.

[22] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *ICDE*, pages 106–115, 2007.

[23] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *ICDE*, page 24, 2006.

[24] DBLP Computer Science Bibliography. http://www.cs.washington.edu/research/xmldatasets/.

[25] Stock Quotes. http://research.cs.wisc.edu/niagara/data/cq/.

[26] University Courses. http://www.cs.washington.edu/research/xmldatasets/.

[27] F. McSherry and K. Talwar. Mechanism design via differential privacy. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '07, pages 94–103, Washington, DC, USA, 2007. IEEE Computer Society.

[28] M. M. Moro, Z. Vagena, and V. J. Tsotras. Tree-pattern queries on a lightweight xml processor. In *VLDB*, pages 205–216, 2005.

[29] M. Nabeel, N. Shang, and E. Bertino. Efficient privacy preserving content based publish subscribe systems. In *SACMAT*, pages 133–144, 2012.

[30] D. E. Robling Denning. *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[31] L. Sweeney. Achieving k-anonymity privacy protection using generalization and suppression. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):571–588, Oct. 2002.

[32] O. Ünay and T. I. Gündem. A survey on querying encrypted xml documents for databases as a service. *SIGMOD Record*, 37(1):12–20, 2008.

[33] S. Wang, D. Agrawal, and A. El Abbadi. A comprehensive framework for secure query processing on relational data in the cloud. In *Secure Data Management*, pages 52–69, 2011.

[34] W. H. Wang and L. V. S. Lakshmanan. Efficient secure query evaluation over encrypted xml databases. In *VLDB*, pages 127–138, 2006.

[35] W. K. Wong, D. W.-L. Cheung, B. Kao, and N. Mamoulis. Secure knn computation on encrypted databases. In *SIGMOD*, pages 139–152, 2009.

[36] L. Xu, T. W. Ling, and H. Wu. Labeling dynamic xml documents: An order-centric approach. *IEEE Trans. Knowl. Data Eng.*, 24(1):100–113, 2012.

[37] Y. Yang, W. Ng, H. L. Lau, and J. Cheng. An efficient approach to support querying secure outsourced xml information. In *CAiSE*, pages 157–171, 2006.