Efficient Tree Pattern Queries On Encrypted XML Documents

Jianneng Cao
Purdue University
caojn@purdue.edu

Fang-Yu Rao
Purdue University
raof@purdue.edu

Mehmet Kuzu
University of Texas at Dallas
mehmet.kuzu@utdallas.edu

Elisa Bertino
Purdue University
bertino@purdue.edu

Murat Kantarcioglu
University of Texas at Dallas
muratk@utdallas.edu

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. In addition, they adopt a *filtering-and-refining* framework, which requires the users to prune false positives from the query results. 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 [10] 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. In the past decade, the subject of query processing over encrypted relational data has been heavily investigated [11, 12, 22, 19]. Past research, however, has not addressed well how to search over encrypted XML documents.

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 be encrypted. This makes the existing outsourcing techniques [11, 12, 22] developed for relational data inadequate for XML documents, since they do not support searching on encrypted structural information.

Past research [23, 6] has investigated query processing over encrypted XML documents. However, such approaches leak structural information [23] or fail to support searches that have constraints on node contents [6] (see Section 8 for details). In addition, they adopt a *filtering-and-refining* framework, in which the cloud service provider returns a superset of the search result. The user thus has to remove false candidates from the returned result with an additional post-processing step. To cope with such issues, in this paper we consider tree pattern query (TPQ) [7, 18] over encrypted XML documents. TPQ is a core operation of XQuery [3], which is now the de facto standard of XML query processing language. Specifically, a TPQ is a tree, which consists of labeled nodes and predicates that specify the constraints on the nodes. To support efficient evaluation of TPQs, we build a vector (i.e., an index) for each document as follows.

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. We then encrypt the vectors by ASPE [24], which ensures privacy and at the same time supports distance comparison (i.e., KNN search) on encrypted vectors. Furthermore, to improve the search efficiency and thus to

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy

scale well for large datasets, we adopt a k-d tree to partition the vectors into non-overlapping subsets, such that non-matchable XML documents for a TPQ are pruned as early as possible.

Compared with existing approaches [23, 6], our solution protects both the structure and content of XML documents, while supporting query processing on them. Our extensive experimental results also show that our solution is efficient. In addition, we do not adopt a *filtering-and-refining* framework as [23, 6]. Our solution returns exact query results without false positive or false negative.

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, and analyze its privacy in Section 6. Section 7 reports our experimental results. Finally, we review related work in Section 8 and conclude our work in Section 9.

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.

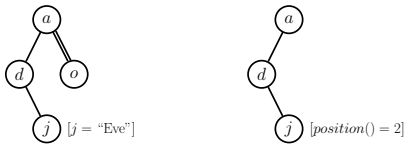
In our setting there are three roles: data user Alice, data owner Bob, and cloud server Charlie. We assume that Bob has a set of XML documents that contain sensitive information, and that he wants to outsource them to the cloud server Charlie. To assure confidentiality, data is stored in an encrypted 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) [7], 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: The 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: $Path_1$ = /`child` :: `a`/`child` :: `d`/`child` :: `j`[`j` = "Eve"] and $Path_2$ = /`child` :: `a`/`descendant` :: `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.

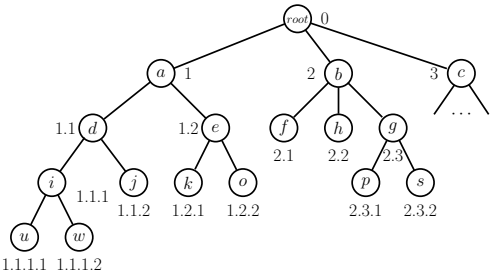


Figure 2: An example of domain hierarchy

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 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

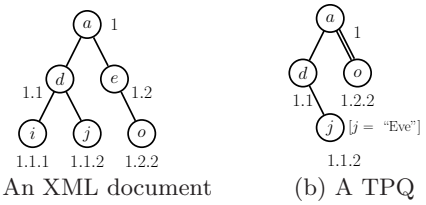
domain hierarchy, or more precisely, an *embedding* (see the next definition) of the domain hierarchy.

DEFINITION 3 (TREE EMBEDDING [13]). *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).

Once the domain hierarchy is ready, we utilize Dewey labeling[2] scheme [25] 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 labeling (refer to Section 5 for more details).



(a) An XML document (b) A TPQ

Figure 3: The Dewey labels of a document/TPQ

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 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.

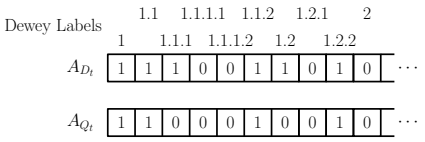


Figure 4: Bit vectors for a document/TPQ

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 structure matches 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:

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

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 $h(Adam)$, i.e., $A_{D_c}[6] = h(\text{"}Adam\text{"})$.*

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 [24] 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 an 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:

[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.

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

•**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 [24].

Fact 1.

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

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

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

which implies 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:

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

Wong et al. [24] 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 [24] 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:

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

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:

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

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 the 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 Construction, 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 into 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 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.

[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.

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 $Q_{1\lambda}$, 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:

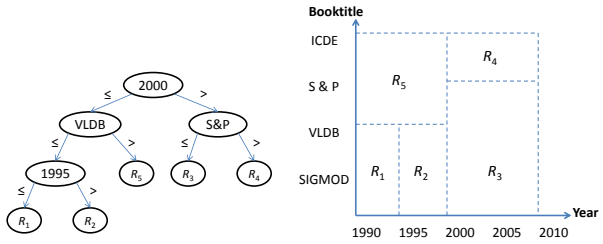
$$\mathsf{Comp}(\mathsf{E}_1(Q_{1\lambda}), \mathsf{E}_1(Q_{2\lambda}), \mathsf{E}_2(A_D)) = \begin{cases} 0, & \text{if } op = \text{'='} \\ -1, & \text{if } op = \text{'<'} \\ 1, & \text{if } op = \text{'>'} \end{cases}$$

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, and thus to scale well for large data sources, in this section we propose two optimization techniques.

The first technique is simple and effective. It partitions the XML documents by their DTDs, such that in each resulting partition all the documents have the same DTD. Then, it treats each partition independently, as if each partition 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, it first finds the partitions, such that the TPQ can be embedded in their domain hierarchies (Definition 3). Then, the TPQ is evaluated only with the XML documents in these partitions. Clearly, this improves the query efficiency. Furthermore, the above optimization technique also saves storage space for secure indices, since these indices are built according to the domain hierarchies, which are smaller than before.

EXAMPLE 2. *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* subtree(a)*, i.e., the subtree rooted at node a in Figure 2, the second DTD corresponds to* subtree(b)*, and the third corresponds to* subtree(c)*. Then, by the above optimization technique, the outsourced database is divided into three partitions, and* subtree(a)*,* subtree(b)*, and* 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* subtree(a)*.*



(a) k-d tree (b) The space partition

Figure 5: An example of k-d tree partitioning

The second optimization technique is a further improvement on top 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, we build 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 [5]. 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 3. *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.*

When a k-d tree is available, a TPQ needs to be evaluated only in 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 3, if Alice is interested in *SIGMOD* papers from 1992 to 1994, then only the documents in partition R_1 should be evaluated. As a result, the performance is improved.

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.

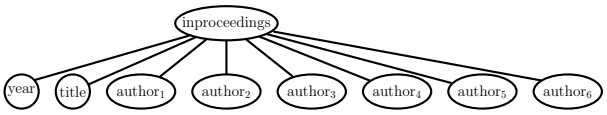


Figure 6: An example domain

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.

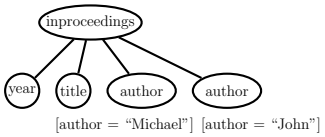
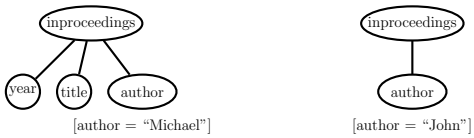


Figure 7: An example query



(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. THE 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.

Privacy without k-d Tree. 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 [8], 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 subquery 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 $A_D[\lambda]$ 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 $A_D[\lambda] < \alpha$.

Privacy with k-d Tree. To improve the performance of query processing, we introduced the k-d tree in Section 4. 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, 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 to learn more about the predicate $[x, op, \alpha]$, if the server also knows the data distribution in the leaf nodes. Consider Figure 5. Suppose that the year frequencies for '1992' and '2002' are close to 10%. Given a predicate on year, suppose that around 10% XML documents appear in the query result. The server with the approximate background knowledge about the year distribution may guess that α is equal to either 1992 or 2002. But it cannot determine which one of 1992 and 2002 is correct. However, with the k-d tree, the server may directly be referred to partition R_3. 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 α is equal to 2002. Such an inference is possible because the distribution of year values in partition R_3 is *different* from that in the whole dataset. We introduce the strategy below to control the difference between the two distributions, and thus also to control the inference.

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 distribution 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 distribution of v_i in the leaf node and $i = 1, 2, \ldots, m$. We adopt J-S divergence to measure the difference between G and L.

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

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, ..., 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, only if $\mathsf{JS}(G_j, L_j^1) \leq t_j$ and $\mathsf{JS}(G_j, L_j^2) \leq t_j$, where L_j^1 and L_j^2 are the local distributions of x_j in C^1 and C^2, respectively, where $j = 1, 2, \ldots, k$.*

7. 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 [15]. 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 [16] containing information about 5,000 stocks. The third dataset is *University Courses* [17], which records the course data from the websites of Reed College. It contains 703 documents. 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

7.1 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 [24] (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, $|U| = 125$.

7.2 Query Evaluation without Optimization

In the basic scheme without any optimization, the secure indices for all the three datasets are put together. Given any TPQ, it should be 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. Queries for these 5 TPQs are then created according to the step 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 linearly as a function of database size (Figure 9 (b)).

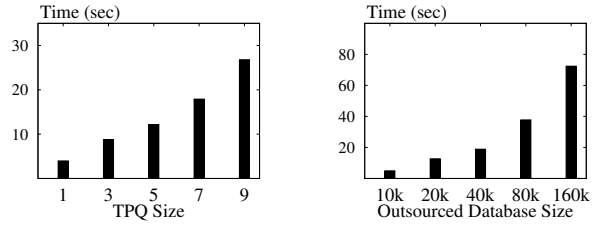


Figure 9: Varying TPQ size and database size

7.3 Query Evaluation with Optimization

In this section, we present the experimental results after optimization techniques are applied. As discussed in Section 4, one straightforward strategy to improve the efficiency is to partition XML documents by their DTDs, such that in each resultant partition all the XML documents share the same DTD. For each partition, a domain hierarchy is built solely on its DTD, and the secure indices of its XML documents are then built according to this domain hierarchy. In our experiments, we have three DTDs. Thus, the complete outsourced database is divided into three partitions. For each partition we generate 3 TPQs with the sizes of 1, 3, and 9, respectively. Figure 10 reports the result, in which the elapsed time increases as a function of query size. In addition, the time cost for smaller partitions (i.e., *Stock Quotes* and *University Courses*) is less than 1 second, which is much lower than that for the larger one (i.e., *Inproceedings*). This is as expected, since now a TPQ is evaluated only with the XML documents in one partition. Therefore, we can see that the above strategy is especially effective for TPQs on small partitions.

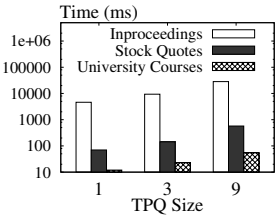


Figure 10: Query evaluation on Separate Datasets

Although the partitioning of documents by DTDs can improve the query efficiency to some extent, there might be still some partitions (e.g., *Inproceedings*) which are quite large and evaluating queries over them is time consuming. To further boost the efficiency, we adopt 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 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 con-

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

$t_{booktitle}$	# of Leaves
0.500	17
0.525	24
0.550	35
0.575	47
0.600	72
0.625	103
0.650	179
0.675	456
0.700	2,409

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

t_{year}	# of Leaves
0.500	16
0.525	30
0.550	49
0.575	62
0.600	72
0.625	75
0.650	75
0.675	75
0.700	75

straints 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, whose constraints 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 11 (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.

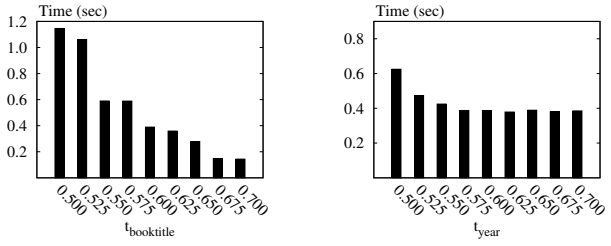


Figure 11: Varying J-S divergence

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 11 (b) also reflects this fact; it stops decreasing after t_{year} reaches 0.625.

We have also evaluated the time cost of building the k-d tree. The splitting of the k-d tree is guided by the J-S threshold. A higher J-S threshold relaxes the requirements on the data distributions in the leaf nodes. Thus, a bigger k-d tree can be built and the time cost is higher. Still, the building of a k-d tree is efficient. In all the cases in the experiments, it takes less than 2,676 ms.

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 on the dimensions of `year` and `booktitle`. 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 12 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	conference	# documents
[1968,1989]	[1,1388]	7,853
[1968,1992]	[1,1388]	13,790
[1968,1992]	[1,1631]	27,454
[1968,1992]	[0,1631]	46,399
[1968,1996]	[0,1696]	74,314
[1959,1996]	[0,1743]	105,888
[1959,2002]	[0,1743]	205,404

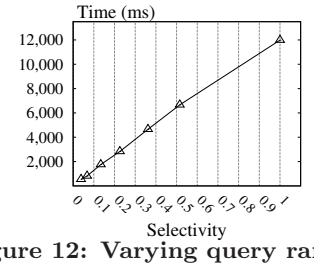


Figure 12: Varying query ranges

8. RELATED WORK

Protecting data privacy in relational databases has been extensively investigated [20, 11, 12, 9, 4]. Hacigümüs et al. [11] 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. [9] 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, 23, 26]. 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 [23] 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 [23] 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 [14]. Yang et al. [26] 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 [21] for a more complete survey about outsourcing of XML documents.

Query processing over encrypted graph-structured data is also related to our work. [6] 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 incurs high false positive rates: for a query not containing any feature, the whole database would be returned.

9. 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 information in XML documents and TPQs into vectors. In such a way, the matching between TPQs and documents is reduced to calculating the distance between their corresponding vectors. Our approach returns the exact query results to data users, so that they do not need to use a post-processing step to prune false candidates as in the existing approaches. Optimization techniques that prune non-matchable XML documents are also proposed to improve the query efficiency. Furthermore, our approach supports both point and range queries. The extensive experimental results show that it is efficient, and scales well to large dataset.

One possible future work is to deploy a proxy between the data user and the cloud server. In this way, the proxy can relieve the data user of the burden of query encoding/encryption. So far, our work only supports equality matching between string values. Thus, another possible future work on our research agenda is the inclusion of the functionality of fuzzy string matching.

10. 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. This work was also partially supported by The Air Force Office of Scientific Research MURI-Grant FA-9550-08-1-0265 and Grant FA-9550-08-1-0260, National Institutes of Health Grant 1R01LM009989, National Science Foundation (NSF) Grant Career-CNS-0845803, and NSF Grants CNS-0964350, CNS-1016343 and CNS-1111529.

11. 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. Fernández, 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, 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] P. Kilpeläinen. Tree matching problems with applications to structured text databases. Ph.D. dissertation Report A-1992-6, University of Helsinki, Finland, November 1992.

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

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

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

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

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

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

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

[21] 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.

[22] 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.

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

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

[25] 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.

[26] 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.