Incremental Mining of Frequent Query Patterns from XML Queries for Caching

Guoliang Li, Jianhua Feng, Jianyong Wang, Yong Zhang, Lizhu Zhou
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, P. R. China
{liguoliang,fengjh,jianyong,dcszlz}@tsinghua.edu.cn; zhangy@tsinghua.org.cn

Abstract

Existing studies for mining frequent XML query patterns mainly introduce a straightforward candidate generate-and-test strategy and compute frequencies of candidate query patterns from scratch periodically by checking the entire transaction database, which consists of XML query patterns transformed from user queries. However, it is nontrivial to maintain such discovered frequent patterns in real XML databases because there may incur frequent updates that may not only invalidate some existing frequent query patterns but also generate some new frequent ones. Accordingly, existing proposals are inefficient for the evolution of the transaction database.

To address these problems, this paper presents an efficient algorithm IPS-FXQPMiner for mining frequent XML query patterns without candidate maintenance and costly tree-containment checking. We transform XML queries into sequences through a one-to-one mapping and then mine the frequent sequences to generate frequent XML query patterns. More importantly, based on IPS-FXQPMiner, an efficient incremental algorithm, Incre-FXQPMiner is proposed to incrementally mine frequent XML query patterns, which can minimize the I/O and computation requirements for handling incremental updates. Our experimental study on various real-life datasets demonstrates the efficiency and scalability of our algorithms over previous known alternatives.

1. Introduction

XML has become a standard for information representation and exchange over the Internet. As many researches have been undertaken on XML indexing [MS99, KSB+02, CLO03], caching [YLH03] and answering [BOB+04, MS05], discovering frequent XML query patterns turns out to be a significant and effective premise of query optimization for its capability of "focus" capturing. The rapid growth of XML repositories has provided the impetus to design and develop systems that can store and query XML data efficiently, and discovering frequent XML query patterns (FXQPs) has recently attracted large amount of attention because the answers of these queries can be stored and cached to improve query performance.

There have been many studies on efficient discovery of frequent patterns for XML queries. Traditional frequent pattern mining approaches typically follow a straightforward candidate generate-and-test strategy, which includes two phases of frequent pattern generation and containment testing. The recent tree-structured data mining research mainly moves towards efficient frequent pattern enumeration and fast containment testing algorithms.

FastMiner proposed in [YLH03] is the current state-of-the-art algorithm about this problem. Given the user log database composed of a set of XML queries, it models them as unordered trees with special XML query constructs like descendant edges or wildcards, and extends frequent structure mining techniques to extract frequent subtrees based on the semantics of the queries. However, a drawback of existing studies is that they can't handle the evolution of the log database, because they have to compute the frequencies of candidate query patterns from scratch in order to get the most up-to-date frequent query patterns. Therefore, as the user queries dynamically join the original database, existing methods are not suitable for the condition that the log database is updated at a relatively high rate since it is very expensive to rerun the discovering program on the set of all transactions by unnecessary I/O cost and computation. From this point, it is important to study efficient algorithms for incremental update of FXQPs as the query collection changes. Chen et al. [CYW04] propose an algorithm about incremental mining of frequent XML queries based on FastMiner [YLH03]. However, it is also based on the straightforward candidate generate-and-test strategy and will induce inefficiency.

Motivated by the need of this observation, this paper proposes, *IPS-FXQPMiner*, an efficient algorithm to mine frequent XML query patterns with neither candidate maintaining nor costly tree-containment checking. We transform XML queries to sequences via a one-to-one mapping and mine the frequent sequences to generate frequent XML query patterns. Subsequently, an efficient incremental algorithm is proposed to incrementally mine frequent XML query patterns.

Our contributions:

- We introduce the notion of Inverted Prüfer Sequence (IPS), and subsequently, an efficient algorithm, *IPS-FXQPMiner*, based on a novel application of the sequence mining approach, is proposed to mine frequent XML query patterns.
- We present an efficient index for incrementally mining frequent XML query patterns, accordingly, an incremental algorithm, *Incre-FXQPMiner*, is demonstrated for incrementally mining these frequent patterns from continuously updated database.
- We conducted an extensive performance study using both real and synthetic datasets of various characteristics. The results show that our algorithms outperform existing proposals in terms of efficiency, scalability as well as answerability

The paper is organized as follows. We formalize the frequent XML query pattern mining problem in section 2. Section 3 presents an effective sequencing algorithm for mining frequent XML query patterns on the static database. In section 4, we present an algorithm to incrementally mine frequent XML query patterns. A thorough experimental study is demonstrated in section 5. We review the related work in section 6 and make a conclusion in section 7.

2. Preliminaries

XML queries are mainly expressed with XPath or XQuery, which conform to the regular path expressions. An XML query can be modeled as a labeled tree, and each vertex denotes a node of the query and each edge denotes the relationship of two nodes. In addition to tag names, a query pattern tree may also contain wildcards * and //. The wildcard * indicates ANY label in DTD, while // indicates zero or more labels.

Definition 1 XML Query Pattern (XQP). *An XML Query Pattern can be defined as a tree XQP=<V, E>, where V is the vertex set, E is the edge set. Each edge $e=(v_1,v_2)$ indicates node v_1 is the parent of node v_2. Each vertex v's label is one of the tag values in {"//", "*"}∪tagSet, where tagSet is the set of all element and attribute names in a schema. We define a partial order ≤′, which is reflexive, and for any label t in tagSet, t ≤′ * ≤′ //, that is, t can match *, which in turn can match //.*

Definition 2 **Rooted SubTree (RST).** *Given an XQP=<V, E>, a rooted subtree RST=<V′, E′> is defined as follows:*

(1) Root(RST)=Root(XQP) and

(2) V′⊆V, E′⊆E

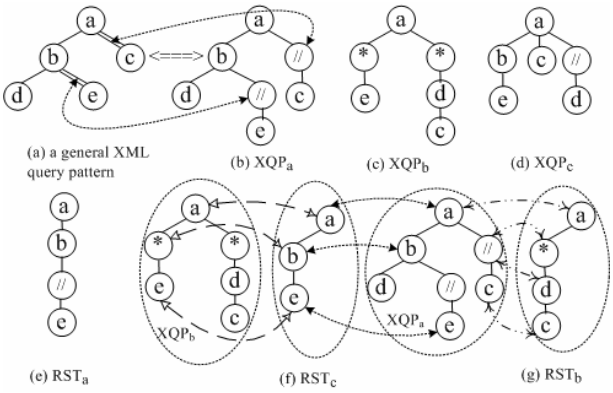

Fig. 1. XML query patterns and their RSTs

To discover frequent query patterns, one important issue is how to count the occurrence of a tree pattern in the database. In this paper, we use the concept of extended subtree inclusion, a sound approach to testing containment of query pattern trees.

Definition 3 **Extended Subtree Inclusion** [YLH03]**.** *Let subtree(p) and subtree(q) be two subtrees with root nodes p and q respectively. Let children(v) denote the set composed of child nodes of v. We can recursively determine if subtree(p) is included/contained in subtree(q) (subtree(q) is also said to include/contain subtree(p)), denoted by subtree(p)⊆subtree(q), as follows:*

p≤′q and

(1) both p and q are leaf nodes; or

(2) p is a leaf node and q=//, then ∃q′∈children(q) such that subtree(p)⊆subtree(q′); or

(3) both p and q are non-leaf nodes, and one of the following holds: i) ∀p′∈children(p), ∃q′∈children(q), s.t., subtree(p′)⊆subtree(q′); or ii) q=// and ∀p′∈children(p), s.t. subtree(p′)⊆subtree(q); or iii) q=// and ∃q′∈children(q), s.t. subtree(p) ⊆subtree(q′).

To make an XML query conform to a general XML query pattern tree, the ancestor-descendant relationship in the XML query is denoted as a vertex "//" in this paper. For example, the two XQPs, in Fig. 1 (a) and (b), are equivalent. All XQPs are represented as (b) in this paper. The patterns (e), (f) and (g) are rooted subtrees of (b), (d) and (c) respectively. RST$_c$ is included in XQP$_a$ and XQP$_b$ (also in XQP$_c$), and RST$_b$ is included in XQP$_a$ but not in XQP$_c$. Thus, *ASupp*(RST$_c$)=3, and *ASupp*(RST$_b$)=2. Fig. 1 shows the inclusion of XQPs.

2.1 Problem Description

We in this section introduce some concepts and then

formalize the frequent XQP mining problem and the incremental frequent XQP mining problem.

Definition 4 Transaction database. *A transaction database D is a collection of XQPs, D= {XQP$_1$,...,XQP$_n$}, which is transformed from a set of XML queries issued against a given XML data source.*

Definition 5 Absolute and Relative Support. *Absolute support of a rooted subtree RST refers to the number of XQPs that contain it in D, denoted as ASupp^D(RST). Relative support is the percentage of XQPs that contain RST in D, denoted as RSupp^D (RST)=ASupp^D(RST)/|D|, where |D| is the size of D.*

Definition 6 Incremental and Updated Database. *Suppose a set of new XML query patterns, d, is to be added to the transaction database D. The database D is referred to as the original database, the database d as the incremental database, and the database D^μ =D+d as the updated database.*

Frequent XQP Mining Problem: Given a database $D=\{XQP_1,XQP_2,...,XQP_n\}$ and a minimum relative support *min_sup* in the range of (0,1], find a set, denoted as $\Gamma(D)$, composed of all the frequent RSTs in D such that for each RST in $\Gamma(D)$, *RSupp*(RST)≥ *min_sup* holds. We use the relative support as our measure of frequency in this paper.

Incremental Frequent XQP Mining Problem: Given an original database D with its frequent RSTs $\Gamma(D)$, a minimum relative support *min_sup* in the range of (0,1], and the incremental database d, then the Incremental Frequent XQP mining is a process to find the set, denoted as $\Gamma(D^\mu)$, composed of all the frequent RSTs in D^μ such that for each RST in $\Gamma(D^\mu)$, *RSupp*(RST) ≥*min_sup* holds.

Existing mining algorithms always directly mine updated database D^μ from scratch, thus involve many unnecessary scans on D and d without help of $\Gamma(D)$. In next sections, by taking account of $\Gamma(D)$, we will present a novel algorithm that incrementally mine the frequent XML query patterns.

2.2 Sequencialization

In our approach, we use a sequence to represent an XQP and transform frequent XQP mining problem to frequent sequence mining problem, which leads to a dramatic improvement over existing mining approaches.

Tree Sequencialization. Our approach starts with a valid and effective sequencing method for XQP. Ad hoc sequencing methods such as depth-first and Prüfer have been used for XML indexing [KSB+02, CLO03]. Prüfer sequence [RM04, KRM+05] and ViST [WPF+03] are succinct tree encoding methods. Prüfer (1918) proposed a method that constructed a one-to-one correspondence between a labeled tree and a sequence by removing nodes from the tree one at a time [P18]. The algorithm to construct a sequence from tree T_n with n nodes labeled from 1 to n works as follows.

From T_n, delete a leaf with the smallest label to form a smaller tree T_{n-1}. Let a_1 denote the label of the node that is the parent of the deleted node. Repeat this process on T_{n-1} to determine a_2 (the parent of the next node to be deleted), and continue until only two nodes joined by an edge are left. The sequence (a_1, a_2,...,a_{n-2}) is called the Prüfer sequence of tree T_n. From the sequence (a_1,a_2,..., a_{n-2}), the original tree T_n can be reconstructed. The length of the Prüfer sequence of tree T_n is $n-2$. In fact, we can construct a Prüfer sequence of length n−1 for T_n by continuing the deletion of nodes till only one node is left. Any numbering scheme can be used in above process to label an XML document tree as long as it associates each node in the tree with a unique number between one and the total number of nodes. This guarantees a one-to-one mapping between the tree and the sequence. Without loss of generality, post-order is used to uniquely number tree nodes. It helps a Prüfer sequence be constructed for an XQP by using the node removal method.

This sequence consists entirely of post-order numbers and is called NPS (Numbered Prüfer Sequence) [RM04]. If each number in a NPS is replaced by its corresponding tag, a new sequence that consists of XML tags can be constructed, which is called LPS (Labeled Prüfer Sequence).

On the basis of LPS, ELPS (Extended Labeled Prüfer Sequence) and ENPS (Extended Numbered Prüfer Sequence) [KRM+05] can be constructed by extending leaf nodes of the document tree with dummy child nodes [RM04]. Clearly the leaf node labels of the original tree are kept in ELPS. However, NPS, LPS, ELPS and ENPS are not suitable to Frequent XQP Mining Problem. Therefore, IPS (Inverted labeled Prüfer Sequence) and INPS (Inverted Numbered Prüfer Sequence) are introduced, which invert ELPS and ENPS respectively. Observe that, IPS preserves the parent-child, ancestor-descendant and sibling order relationships as shown in Property 1.

Property 1. *Suppose IPS=(e_1,e_2,...,e_m), INPS=(n_1, n_2,...,n_m) are sequences of an XQP.* $\forall i,j$, $1\le i<j\le m$. *If $n_i>n_{i+1}$, then e_i is the parent of e_{i+1}; if $n_i<n_j$, e_j is an ancestor of e_i; if $n_i>n_j$ and* $\sim\exists t$, $i<t<j$, $n_i>n_t>n_j$, *then e_i is the parent of e_j.*

Example: *In Fig.* 2, *ELPS is constructed by inserting leaf nodes e,c,c,e into corresponding position of LPS,*

and the leaf node must be preceding and neighboring its parent. We can get IPS of the XQP in Fig. 2(a), adedcacabe, by inverting its ELPS, ebacacdeda. IPS of the RST in 2(b) is adcabe, and its INPS is 764721; As $n_2(6)>n_3(4)$, e_2=d is a parent of e_3=c; as $n_3(4)<n_4(7)$, e_3=c is a descendent of e_4=a.

(a) XQP and its IPS (b) A RST of XQP and its IPS

Fig. 2. A sample tree structure

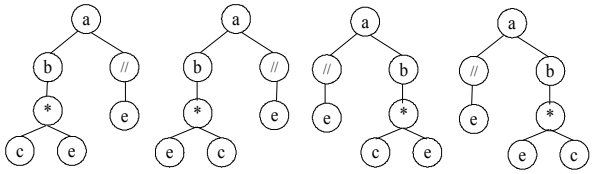

Fig. 3. Four equivalent queries

However, IPSs of some equivalent queries may be different. For example, the four XML queries in Fig. 3 are equivalent, but their IPSs are not the same. Since these four queries are equivalent, the absolute support of them should be four. However, the IPSs of these queries are different, and the absolute support of them is one for the four queries respectively. To address this issue, the query is normalized into a unique form in a following way: for any node that has more than one child, its children are sorted by their labels in lexicographical order. Accordingly, we can transform the queries to their unique forms, and all the equivalent queries are normalized to a same unique form. Hence, there is a one-to-one mapping between equivalent queries and a certain IPS. For example, the queries in Fig. 3 can be normalized to their unique form, XQP$_i$.

Consider S_a and S_b are two sequences, $S_a=(a_1, a_2,...,a_n)$, $S_b=(b_1,b_2,...,b_m)$ ($n\le m$), if there exists $1\le i_1<i_2<...<i_n\le m$, such that $a_1=b_{i1}$, ..., $a_n=b_{in}$, then S_b is called a super-sequence of S_a, and S_a is a sub-sequence of S_b. However, not all of the sub-sequences of IPS are valid and some subsequences do not represent a RST of an XQP and even can not represent a tree structure. For example, in Fig. 2, RST^{IPS}=*adcabe* (RST^{IPS} denotes the IPS of RST in this paper) is a subsequence of XQP^{IPS}=*adedcacabe*, which can represent a valid RST of this XQP, but some sequences e.g. *aeace* cannot represent a valid RST. Hence, we introduce the notion of valid subsequence and Lemma 1 is proposed to distinguish which subsequences are valid ones.

Definition 7 Valid subsequence. *Given an XQP and its sequence S, S_a is a valid subsequence of S iff S_a is a subsequence of S and the subtree that S_a represents is a RST of XQP.*

Lemma 1. *Consider S is a sequence of an XQP, $S^{INPS}=(S_1,...,S_n)$, and $S_a=(s_1,...,s_m)$ is a subsequence of $S(s_1=S_{i_1},..., s_m=S_{i_m})$. S_a is a valid subsequence of S, if $s_1=max(S_1,...,S_n)$ and $\forall s_k>s_{k+1}(1\le k<m)$, $\sim\exists t$, $i_k<t<i_{k+1}$, $S_{i_k}>S_t>S_{i_{k+1}}$.*

Example: *In Fig. 2, adcabe(764721) is a valid subsequence of adedcacabe since it satisfies Lemma* 1. *aeace(75731) is not a valid subsequence of adedcacabe, because there is an item d(6) between a(7) and e(5), which violates the constraint of Lemma* 1.

As tree inclusion in Definition 3, we need to introduce subsequence inclusion, and accordingly present how to count the occurrence of a valid sequence in databases.

Definition 8 Subsequence inclusion. Given two sequences s and S of two different XQPs, $s^{IPS}=(s_1,s_2,...,s_p)$, $s^{INPS}=(n_1,n_2,...,n_p)$ and $S^{IPS}=(S_1,S_2,...,S_m)$, $S^{INPS}=(N_1, N_2, ...,N_m)$. s is included/contained in S, if there exists $1\le i_1\le...\le i_p\le m$, such that $s_1\le' S_{i_1},...,s_p\le' S_{i_p}$, and $\forall k,j$, $1\le k<j\le p$, satisfy:

i) if $i_k=i_{k+1}$ then S_{i_k}=//; and

ii) if $i_k\ne i_{k+1}$ and $n_k>n_{k+1}$, then $N_{i_k}>N_{i_{k+1}}$ holds and (1) S_{i_k+1}=//, $i_k=i_{k+1}$-2 or (2)$i_k=i_{k+1}$-1 hold; and

iii) if $n_k<n_{k+1}$, $s_k\ne$// then $S_{i_k}\ne$//(if $s_p\ne$//, $S_{i_p}\ne$//); and

iv) if $n_k=n_j$ then $N_{i_k}=N_{i_j}$.

s is properly included in S if $\forall v$, $1\le v\le m$, $S_v\ne$// then $\exists i_k=v$ ($1\le k\le p$). $s_k\le' S_{i_k}$ means s_k can match S_{i_k} according to the partial order. i) means that // can be matched by more than one labels. ii) and iv) ensure that s and S have the same tree structure, and // can match zero label in ii). iii) means that // can't be matched by any leaf label, such as, *a*dc* can not be included in *a*//, but included in *a*//*c*.

A sequence database SDB contains a set of tuples in the form of (*Sid*, *S*), where *Sid* is the identifier of *S*. The absolute support of a valid subsequence *Vs* is the number of tuples that contain *Vs* in SDB, denoted by *ASupp*(*Vs*). The relative support is the percentage of tuples that contain *Vs* in SDB, denoted by *RSupp*(*Vs*), where *RSupp*(*Vs*)= *ASupp*(*Vs*)/|SDB|.

Example: *In Fig. 1, XQP_a^{IPS}=a//cab//ebd, RST_c^{IPS}=abe, XQP_b^{IPS}=a*dca*e. RST_c^{IPS} is included in XQP_a^{IPS} and XQP_b^{IPS}, where (i_1, i_2, i_3) are (4,5,7), (5,6,7) respect-*

*tively, and b matches * in XQP_b^{IPS}. abe is not properly included in XQP_a^{IPS}, because there is no item in abe to match d in XQP_a^{IPS}, however, abe is properly included in RST_a^{IPS}(ab//e). RST_b^{IPS} =a*dc, is included in XQP_a, where (i_1, i_2, i_3, i_4) is (1,2,2,3)(* and d both match //).*

Consider the test whether a path a/b/f is included in a/b//e. If it is not known that there exists a path a/b/f//e, then it cannot be concluded that the first path is included in the second path. This is because it is possible for a DTD declaration to include a/b/d/e but not a/b/f//e. In order to handle these situations, it needs to take into account the DTD and perform some expansions of the XQPs. Interested readers are referred to [YLH03] for the details.

Frequent Valid Sequence Mining Problem: Given a sequence database $D=\{S_1,S_2, ...,S_n\}$ and a minimum relative support *min_sup* in the range of (0, 1], find a set, denoted as $\Gamma(D)$, composed of all the frequent valid subsequences *Vseqs* in *D* such that for each *Vseq*, $RSupp^D(Vseq) \geq min\_sup$ holds.

Incremental Frequent Valid Sequence Mining Problem: Given an original sequence database *D* with its frequent valid subsequence set $\Gamma(D)$, a minimum relative support *min_sup* in the range of (0, 1], and an incremental database *d* for *D*, then Incremental Frequent Valid Sequence Mining is a process to find the set, denoted as $\Gamma(D^\mu)$, composed of all the frequent valid subsequences, *Vseqs* in D^μ such that for each *Vseq* in $\Gamma(D^\mu)$, $RSupp^{D\mu}(Vseq) \geq min\_sup$ holds.

From the earlier observations, we can transform Frequent XQP Mining Problem to Frequent Valid Sequence Mining Problem, and transform Incremental Frequent XQP Mining Problem to Incremental Frequent Valid Sequence Mining Problem. Accordingly, we devise two efficient mining algorithms for the two sequence-based problems in next sections.

3. IPS-FXQPMiner Algorithm

3.1 Preprocessing

Given a set of user's XML queries $S=\{q_1, q_2, ..., q_n\}$, in the preprocessing phase, the first step in a sequence based XML query pattern mining algorithm is to normalize the input queries into their unique forms, and then transform each query into a sequence. The complexity of normalization and building IPSs are $O(|XQP|^2)$ and $O(|XQP|)$ respectively, where |XQP| is the number of nodes in XQP. In this way, we get the sequence database $D=\{S_1,S_2, ..., S_n\}$. Fig. 4 illustrates the original database *D* and incremental database *d* in our running example of this paper, while Table 1 shows the corresponding sequence database.

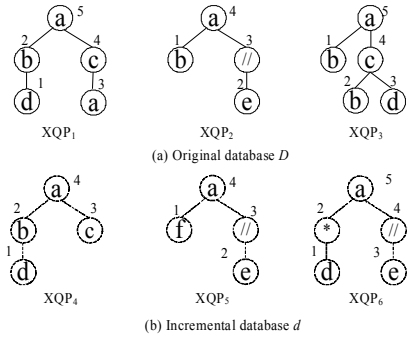

Fig. 4. Original database *D* and incremental database *d*

Table 1. XQPs and corresponding IPS and INPS in Fig.4

Sequence database	XQP	Sid	S (Sequence)	
			IPS	INPS
D	XQP_1	1	acaabd	543521
	XQP_2	2	a//eab	43241
	XQP_3	3	acdcbab	5434251
d	XQP_4	4	acabd	43421
	XQP_5	5	a//eaf	43241
	XQP_6	6	a//ea*d	543521

3.2 Valid Sequence Extension

Traditional frequent subtree mining approaches, such as FastXMiner, typically follow the straightforward candidate generate-and-test strategy. The algorithms with this strategy usually generate a large number of candidate subtrees and need to perform a lot of costly subtree containment testing. To avoid the generate-and-test paradigm and reduce the costly subtree containment testing, we exploit the BI-Directional parent-child checking scheme to find the frequent sequences based on the property of IPS. Obviously parent-child checking is much cheaper than containment testing of tree structure data. That's the key why we exploit frequent sequence mining to resolve frequent XQP mining problem. Assisted with the parent-child information embedded in IPS, it will be proved in following that the parent-child relationship checking is efficient and linear with the size of query patterns.

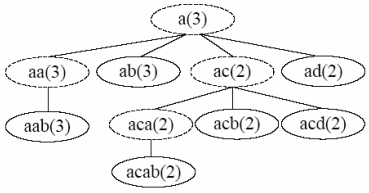

Fig. 5. The lexicographic frequent sequence tree of *D*

Assume there is a lexicographical ordering ≤ among the set of items (labels in *tagSet* and *, //), I, in the input sequence database (e.g., a≤b≤c≤d≤e≤f≤*≤//), conceptually the complete search space of sequence

mining forms a sequence tree. The process of constructing the sequence tree is as follows. The root node of the tree is the root of the XQPs, recursively a node N at level L in the tree is extended by adding one item to get a child node at the next level L+1 and the children of a node N are generated and arranged according to the chosen lexicographical ordering. In Fig. 5, each node contains a frequent sequence and its corresponding absolute support. As an assumption, *min_sup* is 1/2 in all the examples of this paper. Apparently, not all the frequent sequences in Fig. 5 correspond to valid tree structures. For example, "*ad*" is not a valid sequence though with support of 2, since "*ad*" is not a valid subsequence according to Lemma 1.

3.2.1 Frequent Sequence Enumeration

For a given sequence database, many previous frequent pattern mining algorithms have elaborated that depth-first searching is more efficient in mining long patterns than breadth-first searching. Thus, in our approach we traverse XQPs in depth-first order. Pei et al [PHM+01] introduce an efficient pseudo-projection method for enumerating frequent sequences. In this paper, a similar pseudo-projection method is adopted in order to reduce space complexity. A certain node in the sequence tree is always treated as a prefix sequence. By adding one item in I, a prefix sequence can grow to be a longer sequence as its child node. According to downward closure property [AS94], it only needs to extend a prefix sequence using the set of its locally frequent items. To our best knowledge, to establish the frequent items w.r.t a prefix, a well-known method is used that builds the projected database for the prefix and scans it to count the items.

Definition 9 Projected sequence of a prefix sequence. *Given an input sequence $S=e_1e_2...e_n$ which contains a prefix $S_i=e_1e_2...e_i$, the remaining part of S after we remove the first instance of the prefix S_i in S is called the projected sequence w.r.t. prefix $e_1e_2...e_i$ in S.*

Definition 10 Projected database of a prefix sequence. *Given an input sequence database SDB, the complete set of projected sequences in SDB w.r.t. a prefix sequence $e_1e_2...e_i$ is called the projected database w.r.t. prefix $e_1e_2...e_i$ in SDB.*

For example, the projected sequence of prefix sequence *ac* in sequence *acabd* is *abd*. The projected database of prefix sequence *ab* in *D* is (*d*,∅, *ab*).

3.2.2 Valid Sequence Extension

There are many subsequences of a certain sequence, but not all of them are meaningful from the application point of view, that is, some of them can represent a subtree of XQP, but some others may not. We can check whether a subsequence is a valid subsequence via Lemma 1. However, to mine the frequent valid sequences, we need to address the issue that how to add a valid local item to extend a prefix sequence.

Definition 11 Valid local item. *Given a prefix sequence S and its projected sequence PS, where $S^{IPS}=e_1e_2...e_i$, $S^{INPS}=n_1n_2...n_i$, $PS^{IPS}=(pe_1,pe_2,...,pe_j)$, $PS^{INPS}=(ne_1,ne_2,...,ne_j)$, then a local item e w.r.t. S is called a valid local item of S, if e satisfies:*
i) $\exists m$, $1\le m\le j$, $e\le' pe_m, ne_{m-1}=n_i$ and $ne_m<n_i$ ($pe_0=e_i$); or
ii) $\exists m$, $2\le m\le j$, $e\le' pe_m, pe_{m-1}=//, ne_{m-2}=n_i$ and $ne_m<n_i$; or
iii) $\exists m$, $1\le m\le j$, $e\le' pe_m, ne_m>n_i$.

The local items that satisfy i) or ii) are children of e_i, and // will be matched by zero label in ii); while the local items that satisfy iii) are ancestors of e_i.

We employ the depth-first method to enumerate the item. We first initialize the root node as the first frequent sequence s_1, and then enumerate the valid local item *e* of the given prefix sequence s_1 and count the number of the sequences that contain the valid local item *e*, i.e., *ASupp* ($<s_1,e>$). If $RSupp(<s_1,e>)\ge min\_sup$, $<s_1,e>$ is a frequent sequence. We iteratively enumerate the valid local items until there is no valid local item. Especially, when enumerating a valid local item, // in projected sequence can be matched by any zero or more labels or *, while * can be matched by any label.

Example: *In Fig. 1, $XQP_a^{IPS}=a//cab//ebd$, $RST_b^{IPS}=a*dc$. Suppose the current sequence is a, as * can match //, so * is a valid item w.r.t. a for XQP_a^{IPS}; as // can be matched by zero or more labels, its projected sequence can be cab//edb, and also can be //cab//ebd. In this way, a is extended to a*, then the next item d is checked, but it is not a valid item w.r.t. cab//edb, because the item d in this projected sequence does not satisfy Definition 11. However, it is a valid item w.r.t. //cab//ebd since d can match //. In the same way, a*dc is included in a//cab//ebd.*

Lemma 2. *Given a valid local item e w.r.t. a prefix sequence $e_1e_2...e_i$, whose parent is e_m ($1\le m\le i$), the tree node corresponding to e_m must be on the left most path of the subtree corresponding to prefix $e_1e_2...e_i$.*

The sequence extension framework in our approach is left most extension, which complies with the right most extension strategy adopted in [Z02,YLH03], and it removes redundancy in frequent sequence mining.

3.3 IPS-FXQPMiner Algorithm

By integrating normalization, sequencialization, valid sequence extension and frequent sequence enumeration, we derive our algorithm, *IPS-FXQPMiner* as illustrated in Fig. 6, which avoids costly tree containment testing and prunes the unrelated search space efficiently under the local item's validity checking. *IPS-FXQPMiner* enumerates the complete set of frequent sequences, which is similar to the pseudo-projection-based PrefixSpan algorithm. It normalizes XML query patterns into their unique forms (line 2), and converts the input XML query patterns into a set of sequences through the sequencing method described in section 2.2(line 3). To mine the frequent valid sequence, it recursively calls its subroutine *Freq_Valid_Seq*(line 4): for a certain prefix PS, if it is non-empty, output it (line 6), scan projected database PS_SDB once to find the locally valid frequent items (line 7) via Definition 11, each frequent item e_i can be chosen in lexicographical order to grow PS to get a new prefix PS^i (line 10), scan PS_SDB once again to build pseudo-projection database for each new prefix PS^i (line 11). Furthermore, one can easily figure out that the order of the frequent sequence enumeration is consistent with the depth-first traversal of the frequent sequence tree.

IPS-FXQPMiner (*D*, *min_sup*, *Γ*(*D*))
Input: *D*: the database composed of user's XML queries
min_sup: a minimum support
Output: *Γ*(*D*): a set of frequent valid sequences in *D*
1: *Γ*(*D*)=∅;
2: normalization (*D*);
3: build-Sequence (*D*, *D*_SDB); //*D*_SDB:the sequence database for *D*
4: Freq_Valid_Seq (*D*_SDB, ∅, *min_sup*, *Γ*(*D*));
5: return *Γ*(*D*);
Freq_Valid_Seq (PS_SDB, PS, *min_sup*, *Γ*(*D*))
Input: PS_SDB: a projected sequence database w.r.t. PS
PS: a prefix sequence; *min_sup*: a minimum support
Output: *Γ*(*D*): a set of frequent valid sequences in *D*
6: if PS is non-empty then *Γ*(*D*)←PS;
7: VLF_PS=Valid_local_frequent_items(PS_SDB, PS, *min_sup*);
8: if VLF_PS is empty then return;
9: for each valid locally frequent item e_i *do*
10: PS^i = <PS, e_i >;
11: PS_SDB^i= pseudo projected database (PS^i, PS_SDB);
12: Freq_Valid_Seq(PS_SDB^i, PS^i, *min_sup*, *Γ*(*D*));
Valid_local_frequent_items(PS_SDB, PS, *min_sup*);
return frequent valid items w.r.t. PS in PS_SDB via Definition 11.

Fig. 6. IPS-FXQPMiner algorithm

Example: *Fig. 7 lists how to enumerate the valid local items. For the three sequences in D, suppose current frequent valid prefix sequence is a, and the projected sequences of them are caabd, //eab and cdcbab respectively. To reduce the storage space, only a pointer is recorded for each projected sequence instead of the whole sequence. In this paper, the starting position of the projected sequence in original sequence is recorded. It is obvious that c is a valid item for caabd(5(a)>4(c)) and cdcbab(5(a)>4(c)), but c is not a valid item for //eab (there is no ac//e in DTD). In this way, RSupp(ac)=2/3>1/2, and it is a frequent valid sequence. Then, we want to check whether item a is a valid item for aabd, dcbab w.r.t. ac, and there are two a which are both valid items for aabd, thus there are two projected sequences abd and bd for aabd, however only one of them is frequent in this running example.*

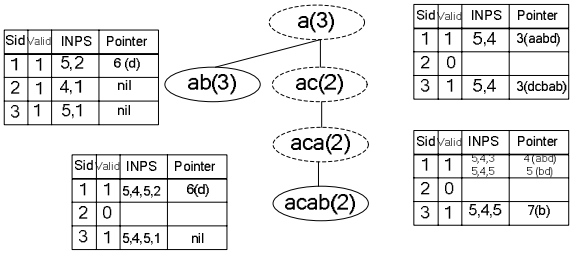

Fig. 7. Valid sequence extension of *D*

4. Incremental Mining Algorithm

Although we can mine the frequent query patterns on the updated database via *IPS-FXQPMiner* from scratch, it is not efficient for the evolving database. Therefore, we in this section present how to incrementally mine frequent XML query patterns.

4.1 F&Q/F-index

Suppose *Γ(D)*, *Γ(d)*, $\Gamma(D^u)$ are the sets composed of frequent sequences of *D*, *d*, D^u respectively, and original database *D* is already mined and *Γ*(*D*) has been gotten through *IPS-FXQPMiner*. We first mine the incremental database *d* using *IPS-FXQPMiner* and get *Γ*(*d*), and then generate the up-to-date frequent sequence set, i.e., $\Gamma(D^u)$ through *Γ(D)* and *Γ(d)*.

Valid sequences (*Vseqs*) of *D* and *d* are classified into four categories:

1) $\Gamma(D) \cap \Gamma(d)$, that is, all of the *Vseqs* that are frequent in both *D* and *d*.

2) $\Gamma(D) - \Gamma(d)$, that is, all of the *Vseqs* that are frequent in *D* but infrequent in *d*.

3) $\Gamma(d) - \Gamma(D)$, that is, all of the *Vseqs* that are frequent in *d* but infrequent in *D*.

4) Other sequences, that is, all of the *Vseqs* that are infrequent in both *D* and *d*.

Lemma 3. *Vseqs in the first category must be frequent in the updated database D^u. Vseqs in the fourth category can not be frequent in D^u.*

We can determine whether the sequences in the first category or fourth category are frequent or not in D^u according to Lemma 3. However, it is not easy to check whether the sequences in the second category or third category are frequent. Accordingly, we need to check whether each sequence in $\Gamma(D)$-$\Gamma(d)$ is still frequent in D^u, and each sequence in $\Gamma(d)$-$\Gamma(D)$ is a new frequent sequence of D^u.

For each sequence *Vseq* in $\Gamma(D)$-$\Gamma(d)$, we scan d to count the number of sequences ($ASupp^d(Vseq)$), which contain *Vseq*, and then check whether $(ASupp^{\Gamma(D)}(Vseq) + ASupp^d(Vseq))/|D^u| \geq min\_sup$ holds; In the same way, we can check whether each sequence in $\Gamma(d)$-$\Gamma(D)$ is frequent. Although this approach is more efficient than mining the updated database from scratch, it may involve some unnecessary scans of D and d. For example, *abd* is frequent in d, but infrequent in D. Although only $XQP_1^{IPS}(acaabd)$ contains it, it has to scan all the sequences in D. To scan D and d as few as possible, we introduce some concepts and accordingly present the novel index, which is constructed for the original and incremental database.

Definition 12 Direct-Prefix Sequence. *Given a sequence $S=(e_1e_2...e_i)$, $S'=(e_1e_2...e_{i-1})$ is the direct-prefix sequence of S.*

Definition 13 Path-Prefix Sequence. *Given a sequence S, $S^{INPS}=(e_1...e_n)$. $S'^{INPS}=(e'_1=e_{i_1},...,e'_j=e_{i_j}=e_n)$ is path-prefix sequence of S ($1\leq i_1<,...,<i_j\leq n$), if $e'_1=max(e_1,...,e_n)$ and $\forall k$, $1\leq k<j$, $e'_k>e'_{k+1}$ and $\sim\exists m$, $i_k<m<i_{k+1}$, $e_m>e'_{k+1}$. If $S'\neq S$, S' is called a proper path-prefix sequence of S.*

A path-prefix sequence $e_1...e_j$ represents the path from node e_1 to node e_j w.r.t its corresponding XQP, that is, e_1 is the root of the XQP, and $\forall k$, $1\leq k<j$, e_k is the parent of e_{k+1}.

Definition 14 Quasi-Frequent Sequence. *Sequence S is a frequent sequence if $RSupp(S)\geq min\_sup$; Sequence S is a quasi-frequent sequence if its direct-prefix sequence and its proper path-prefix sequence(if any) are both frequent. The quasi-frequent sequence but not frequent is called a Q/F sequence.*

According to the apriori property, if a sequence is not a quasi-frequent sequence, it can not be frequent. Accordingly, if a specified sequence is a quasi-frequent sequence, we need not scan the projected database to count its absolute support.

In our approach, we construct the *F&Q/F-index* for the transaction database. *F-index* preserves each frequent sequence, while *Q/F-index* preserves each *Q/F* sequence [1]. Sequences in *F&Q/F-index* are sorted in lexicographical order and each sequence maintains its absolute support and *IDList*, where *IDList* records a set of tuples (*Sid*, *Pointer*), *Sid* is the identifier of its super-sequences and *Pointer* is used to record its corresponding projected sequence in D. Given a frequent or *Q/F* sequence, its super-sequences and projected sequences can be gotten easily through its *IDList*.

The reason why *F&Q/F-index* is constructed is that: for any sequence in *F&Q/F-index*, the database has to be scanned to check whether it is frequent during mining frequent sequences, thus its absolute support and *IDList* are recorded by the way, which is similar to record a table in dynamic programming. *F&Q/F-index* can be constructed during mining the frequent query patterns. Once *IPS-FXQPMiner* detects a valid sequence is frequent or quasi-frequent, we record the *IDList* and *ASupp* for this sequence and inserts it into *F&Q/F-index* in lexicographical order.

Example: *Table 2 shows the F&Q/F-index of D and d respectively. In database d, abd is a valid subsequence of Seq_4 (XQP_4^{IPS}) and a*d is a valid subsequence of Seq_6, since * can be matched by any label, ASupp(abd)=2 and abd is a frequent sequence in d. ASupp of a//e in F-index^d is 2, and this means that there are two sequences which contain a//e in d. Its IDList is (5,4) and (6,4) means its super-sequences are Seq_5 and Seq_6, and the two corresponding projected sequences are: the subsequence of Seq_5 obtained from the 4^{th} item to the last item and the subsequence of Seq_6 obtained from the 4^{th} item to the last item.*

Table 2. *F&Q/F-index*

(a) *F-index* of D

F-Seq	*ab*	*ac*	*aca*[1]	*acab*
ASupp	3	2	2	2
IDList	1,6|2,nil|3,nil	1,3|3,3	1,5|3,7	1,6|3,nil

(b) *Q/F-index* of D

Q/F-Seq	*abd*	*aca*[2]	*acabd*	*acb*	*acd*	*a//*
ASupp	1	1	1	1	1	1
IDList	1,nil	1,4	1,nil	3,6	3,4	2,3

(c) *F-index* of d

F-Seq	*ab*	*abd*	*ac*	*af*
ASupp	2	2	2	2
IDList	4,5|,6,6	4,nil|,6,nil	4,3|6,6	5,nil|6,6
F-Seq	*a//*	*a//e*	*a//ea*	*a//eaf*
ASupp	2	2	2	2
IDList	5,3|6,3	5,4|6,4	5,5|6,5	5,nil|6,6

(d) *Q/F-index* of d

Q/F-Seq	*aca*	*afd*	*a**	*a//eafd*
ASupp	1	1	1	1
IDList	4,4	6,nil	6,6	6,nil

[1] *For Q/F sequences, we could build Q/F-index for a part of them if the index size is beyond memory limited, and we can build indices for those sequences whose relative support is larger than δ, where $0\leq\delta\leq min\_sup$. If $\delta=min\_sup$,there is no Q/F-index; if $\delta=0$,all the Q/F sequences are reserved.*

4.2 Incre-FXQPMiner

In this section, we present an algorithm, *Incre-FXQPMiner*, to incrementally mine frequent query patterns with the help of *F&Q/F-index*. Without loss of generality, suppose the original database has been mined and *F&Q/F-index* has been constructed.

We first mine d through *IPS-FXQPMiner*, and then obtain the frequent *Vseq*s of the updated database D^{μ} through merging original mining results of D, i.e., $\Gamma(D)$ and the new mining results of d, i.e., $\Gamma(d)$. In addition, to efficiently mine the frequent sequences, *Vseq*s in $\Gamma(D)$-$\Gamma(d)$ and $\Gamma(d)$-$\Gamma(D)$ are sorted by $|Vseq|$ in ascending order, respectively.

Since all the sequences in $\Gamma(D)$ or $\Gamma(d)$ are sorted in lexicographical order, $\Gamma(D)\cap\Gamma(d)$ can be gotten through the merge-join of $\Gamma(D)$ and $\Gamma(d)$, which only costs $O(|\Gamma(D)|+|\Gamma(d)|)$. We need to check whether the *Vseqs* in $\Gamma(D)$-$\Gamma(d)$=$\Gamma(D)$-$\Gamma(D)\cap\Gamma(d)$ and $\Gamma(d)$-$\Gamma(D)$= $\Gamma(d)$-$\Gamma(D)\cap\Gamma(d)$ are frequent or not in D^{μ}. Without loss of generality, we only introduce how to check the former.

$\forall Vseq\in \Gamma(D)$-$\Gamma(d)$, *Vseq* cannot be frequent in d. Let *DVseq*, *PPVseq* are direct-prefix and proper path-prefix sequences of *Vseq* respectively. As $|DVseq|<|Vseq|$, $|PPVseq|<|Vseq|$, so whether *DVseq* and *PPVseq* are frequent in D^{μ} has been processed. Therefore, if one of *PPVseq* (if any) and *DVseq* is infrequent, then it is obvious that *Vseq* is infrequent. Otherwise, it needs to scan *Q/F-index* of d or *F-index* of D^{μ} to check whether *Vseq* is frequent:

1) if *Vseq* is in *Q/F-index* of d, $ASupp^{D+d}(Vseq)$ and $Vseq^{D+d}.IDList$ can be gotten according to *Q/F-index* of d, and thus,
$$ASupp^{D+d}(Vseq)=ASupp^{\Gamma(D)}(Vseq)+ASupp^{d.Q/F\text{-}index}(Vseq);$$
$$Vseq^{D+d}.IDList=Vseq^{\Gamma(D)}.IDList\cup Vseq^{d.Q/F\text{-}index}.IDList.$$
2) else, as *DVseq* is frequent, *Vseq* must be in *F-index* of D^{μ}, thus we scan each projected sequence, *PS_DVseq* of *DVseq* according to $DVseq^{d}.IDList$ in *F-index* of D^{μ}, then check whether item e is a valid item of *PS_DVseq* w.r.t. *DVseq*(*Vseq*=<*DVseq*,*e*>) and count the number of *PS_DVseqs* in d, i.e., $ASupp^{d}(Vseq)$, where each *PS_DVseq* contains the valid item e, finally, record $Vseq^{d}.IDList$, and thus,
$$ASupp^{D+d}(Vseq)=ASupp^{d}(Vseq)+ASupp^{\Gamma(D)}(Vseq);$$
$$Vseq^{D+d}.IDList=Vseq^{d}.IDList\cup Vseq^{\Gamma(D)}.IDList.$$

Incre-FXQPMiner(Fig. **8**) first mines the frequent sequences of d (line 2), and then gets $\Gamma(D)\cap\Gamma(d)$ (line 3). For each sequence *Vseq* in $\Gamma(D)$-$\Gamma(d)$ or $\Gamma(d)$-$\Gamma(D)$, it checks whether *Vseq* is frequent in D^{μ} by calling its subroutine *IncreMiner* (lines 5, 6): if *Vseq* is in *Q/F-index* of *db* (*db*=*D* or *db*=*d*), it checks whether *Vseq* is frequent according to *Q/F-index* of *db* (lines 10, 11); otherwise, checks whether *Vseq* is frequent according to its direct-prefix sequence's *F-index* in D^{μ} (lines 12-15), and lastly constructs *F&Q/F-index* (lines 16-20).

Incre-FXQPMiner (***D*,*F&Q/F-index*^{D},*d*, *min_sup***)
Input: D: the original database; *F&Q/F-index*^{D}: *F&Q/F-index* of D
d: the incremental database; *min_sup*: a minimum support.
Output: $\Gamma(D^{\mu})$: a set of frequent valid sequences in D^{μ}
1: $\Gamma(D^{\mu})=\varnothing$;
2: *IPS-FXQPMiner*(d, *min_sup*, $\Gamma(d)$);
3: $\Gamma(D)\cap\Gamma(d)$=*merge_join*($\Gamma(D)$,$\Gamma(d)$);
4: $\Gamma(D^{\mu})=\Gamma(D^{\mu})\cup(\Gamma(D)\cap\Gamma(d))$;
5: $\Gamma(D^{\mu})\cup$=*IncreMiner* ($\Gamma(D)$-$\Gamma(d)$,d,*min_sup*,*F&Q/F-index*^{D},*F&Q/F-index*^{d});
6: $\Gamma(D^{\mu})\cup$=*IncreMiner*($\Gamma(d)$-$\Gamma(D)$,D,*min_sup*,*F&Q/F-index*^{D},*F&Q/F-index*^{d});
7: return $\Gamma(D^{\mu})$;
IncreMiner(Γ, *db*, *min_sup*, *F&Q/F-index*^{D}, *F&Q/F-index*^{d})
Input: Γ: a set of frequent valid sequences; *db*: a database (D or d);
min_sup: a minimum support;
Output: $\Gamma(D^{\mu})$: a set of frequent valid sequences in D^{μ}
8: for each $VSeq\subseteq\Gamma$ do // $VSeq$=<$DVSeq$,e_i>
9: if *DVSeq* and *PPVSeq* are both in *F-index* of D^{μ} then
10: if *VSeq* is in *Q/F-index* of *db* then
11: $ASupp^{D+d}(VSeq)=ASupp^{db.Q/F\text{-}index}(VSeq)+ASupp^{\Gamma}(VSeq)$;
12: else for each *PS_DVSeq* in $DVSeq^{db}.IDList$ do
13: if e_i is a valid item of *PS_DVSeq* *w.r.t.* *DVSeq* then
14: $ASupp^{db}(VSeq)$++;
15: $ASupp^{D+d}(VSeq)=ASupp^{db}(VSeq)+ASupp^{\Gamma}(VSeq)$;
16: if *VSeq* is frequent in D^{μ} then
17: *build-F&Q/F-index*(*F-index*, *VSeq*, D^{μ}); $\Gamma(D^{\mu})\leftarrow VSeq$;
18: else if *VSeq* is quasi-frequent in D^{μ} then
19: *build-F&Q/F-index*(*Q/F-index*, *VSeq*, D^{μ});
20: return $\Gamma(D^{\mu})$;
***build-F&Q/F-index*(*F-indexorQ/F-index*, *VSeq*, D^{μ})**
21: $VSeq^{D+d}.IDList= VSeq^{D}.IDList\cup VSeq^{d}.IDList$
22: build *F-index or Q/F-index* for *VSeq* in D^{μ};
***merge_join*($\Gamma(D)$,$\Gamma(d)$)**
return the sequences that are both in $\Gamma(D)$, $\Gamma(d)$ via merge-join algorithm

Fig. 8. Incre-FXQPMiner algorithm

Table 3. *F-index* of D^{μ}

F-Seq	***ab***	***abd***	***ac***	***aca***
ASupp	5	3	4	3
IDList	1,6|2,nil|3,nil|4,5|6,6	1,nil|4,nil|6,nil	1,3|3,3|4,3|6,6	1,5|3,7|4,4
F-Seq	***acab***	***a//***	***a//e***	***a//ea***
ASupp	3	3	3	3
IDList	1,6|3,nil|4,5	2,3|5,3|6,3	2,4|5,4|6,4	2,5|5,5|6,5

Example: *F&Q/F-index of D and d are constructed as shown in Table 2.* $\Gamma(D)$=*{ab;ac;aca;acab}*, $\Gamma(d)$=*{ab;abd;ac;af;a//;a//e;a//ea;a//eaf}*, $\Gamma(D)\cap\Gamma(d)$=*{ab;ac}*, $\Gamma(D)$-$\Gamma(d)$=*{aca;acab}*, $\Gamma(d)$-$\Gamma(D)$=*{abd;af;a//;a//e; a//ea;a//eaf}*. *The F-index of* D^{μ} *is illustrated in Table3. All the sequences in* $\Gamma(D)\cap\Gamma(d)$ *must be frequent in* D^{μ}. *Suppose VSeq=**abd** in* $\Gamma(d)$-$\Gamma(D)$ *and DVSeq=ab is the direct-prefix sequence of VSeq. As DVSeq is frequent in* D^{μ}, *so **abd** is quasi-frequent in* D^{μ}. *As **abd** is in Q/F-index of D,* $ASupp^{D+d}(VSeq)=ASupp^{D.Q/F\text{-}index}(VSeq)+$

$ASupp^{\Gamma(d)}(VSeq)=3$; $VSeq^{D+d}.IDList=VSeq^{\Gamma(d)}.IDList \cup VSeq^{D.Q/F\text{-}index}.IDList=\{(1,nil),(4,nil),(6,nil)\}$, thus ***abd*** *is frequent in* D^{μ}. *In the same way, aca,acab,a//,a//e, a//ea are also frequent in* D^{μ}. $\Gamma(D^{\mu})=(\Gamma(D)\cap\Gamma(d))\cup\{aca;acab\}\cup\{abd;a//;a//e;a//ea\}=\{ab;abd;ac;aca;acab;a//;a//e;a//ea\}$.

5. Performance Evaluation

This section evaluates the performance of our algorithms and demonstrates the efficiency and scalability of our approach. FastXMiner is the most efficient algorithm for frequent XML query pattern discovery in candidate generate-and-test manner. increQPMiner [CYW04] is the best algorithm for incrementally mining frequent XML query patterns, which is evolved from FastXMiner. We compare *IPS-FXQPMiner* with FastXMiner on the static database, and compare our incremental algorithm with increQPMiner on the evolving database for different datasets varying *min_sup* values and the number of queries. The datasets we used are DBLP, XMark and SigmodRecord. According to the DTDs of these three datasets, some "//" and "*" nodes are added to construct the XQPs as the input of our experiments. Different characteristics of XQPs are shown in Table 4. In contrast, the average number of nodes, maximum depth and fan-out of XQPs reflect the complexity of the dataset. All the datasets follow the default Zipfian distribution. All the experiments are carried out on a computer with Pentium III 1.14 GHz and 1G RAM by implementing in C++.

Table 4. Characteristics of datasets used

Datasets	Average # of nodes	Max depth	Max fan-out
XMark	8.4	11	11
DBLP	7.6	8	12
SigmodRecord	5.4	5	4

5.1 Evaluation on the Static Database

Firstly, we compare *IPS-FXQPMiner* with FastXMiner by varying *min_sup*. In this comparison, we chose 200,000 XQPs in each dataset as our experimental data. Fig. 9 shows the comparison results between the two algorithms on the static database. We can see *IPS-FXQPMiner* outperforms FastXMiner on each dataset. As well, the time needed for FastXMiner at support 0.2% is always a bit more than that at 2%. This is because with the decreasing of *min_sup*, the "straightforward generate-and-test" style mining algorithms need to match an increasing number of frequent candidates, while *IPS-FXQPMiner* avoids redundant sequences testing by dynamic enumeration and pruning after parent-child constraint is applied.

Secondly, we evaluate the scalability of our algorithm by varying the number of XQPs on three datasets and fixing *min_sup* at 1%. Fig. 10 shows the performance results on XMark, DBLP and SigmodRecord respectively. We can observe that *IPS-FXQPMiner* has better scalability than FastXMiner. Especially, on SigmodRecord, when the number of XQPs is 200,000, *IPS-FXQPMiner* costs only 25s while FastXMiner costs 45s. This further demonstrates the effectiveness of our sequence enumeration method

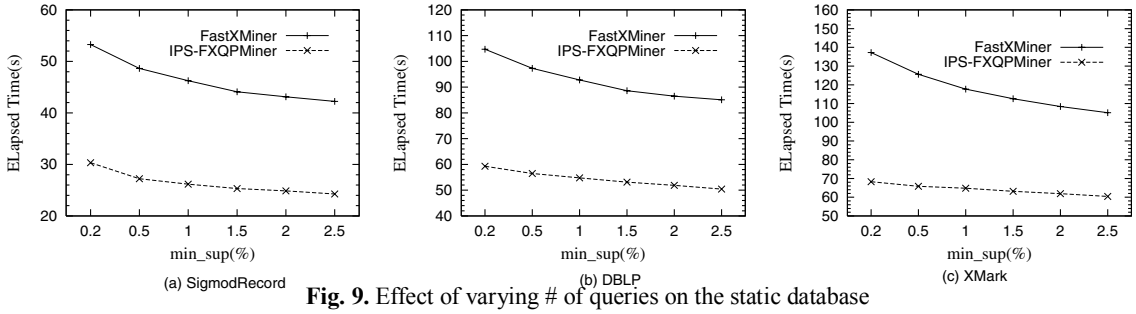

Fig. 9. Effect of varying # of queries on the static database

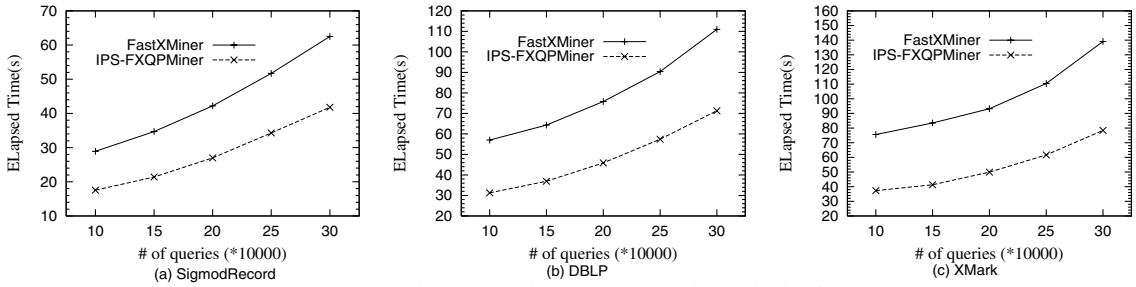

Fig. 10. Effect of varying *min_sup* on the static database

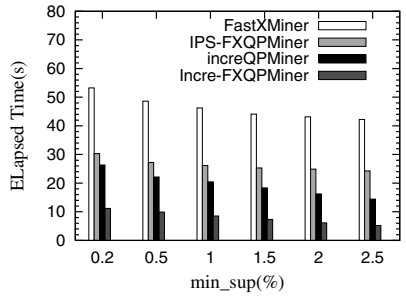

Fig. 11. Effect of varying *min_sup* on the evolving database

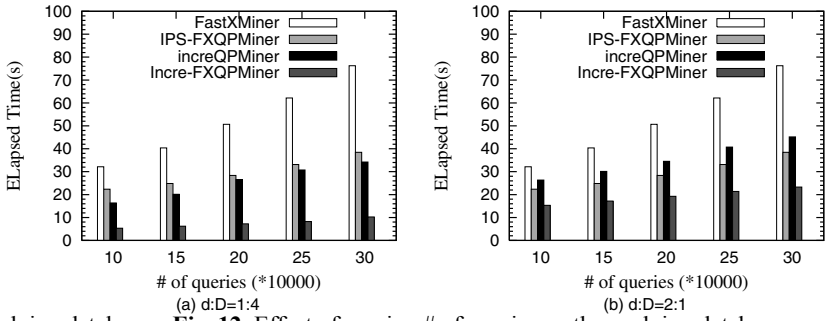

Fig. 12. Effect of varying # of queries on the evolving database

5.2 Evaluation on the Evolving Database

We compare *Incre-FXQPMiner* with increQPMiner, *IPS-FXQPMiner* and FastXMiner on the evolving transaction database in this section.

Fig. 11 shows the performance results of the four algorithms by varying *min_sup* on SigmodRecord. The number of queries in the incremental database *d* is a quarter of that in the original database *D*. Two incremental algorithms incrementally mine the frequent XQPs, while the two incremental algorithms mine the frequent XQPs on D^{μ} directly. We can see our incremental algorithm outperforms the other ones. This demonstrates that incremental mining is more efficient than mining from scratch. When *min_sup* is 2%, *Incre-FXQPMiner* costs one seventh of FastXMiner, a quarter of *IPS-FXQPMiner* and half of increQPMiner.

The scalability of these algorithms is also evaluated by varying the number of XQPs on SigmodRecord and fixing *min_sup* at 1%. In Fig. 12(a), the number of queries in *d* is a quarter of that in *D*, while in Fig.12 (b), the number of queries in *d* is twice of that in *D*. We can see *Incre-FXQPMiner* outperforms increQPMiner, *IPS-FXQPMiner* and FastXMiner. This is because our incremental algorithm takes full advantage of mining results of *D*, while increQPMiner only makes use of a part of the mining results.

6. Related Work

Mining frequent substructures of trees, graphs and sequences has drawn much attention as an essential data mining task, with various applications including market and customer analysis, web log analysis, pattern discovery in protein sequences and XML frequent patterns, and so on.

For tree and graph mining, frequent pattern discovering was first addressed in biological science. Dehaspe et al [DTK98] proposed an efficient algorithm to mine frequent substructures in protein and chemical compounds. In graph database, algorithm FSG in [KK01] was considered as a fast miner for discovering connected sub-graphs by extending the notion of level-by-level expansion of [AS94]. Motivated by discovering user navigation patterns in web surfing, Zaki [Z02] proposed sub tree mining algorithm in forest, which faced more complex data situation. FREQT [AAK+02], TreeFinder [TRS02] aimed at finding frequent subtrees in a collection of semi-structured documents, but still cannot solve the problem of XML query pattern mining due to the existence of "*" and "//".

To our best knowledge, FastXMiner [YLH03] was the most efficient mining algorithm for XML frequent query pattern discovery, as only valid candidate XQPs are enumerated by FastRSTGen for costly containment testing. It still follows the traditional idea of generate-and-test paradigm for tree-structured data mining. Global query pattern tree needs to be generated for XQP enumeration, as well as expensive candidate generation and containment testing. Another closest related work is finding the frequent substructures from a collection of semi-structured Web documents [WL00]. On the other hand, for sequence mining, [SA96, MCP98, HPM+00, AJY+02] mainly focused on general and constraint-based sequence mining problems. Various researches have been done on frequent episode mining [YHA03], cyclic association rule mining [ORS98], temporal relation mining [BWJ98], partial periodic pattern mining [HDY99], and long sequential pattern mining in noisy environment [YYW+02]. But the voice of a frequent pattern mining algorithm should not mine all frequent patterns but only the closed ones come out with convincing arguments for its better efficiency and more compact results without valuable information loss. CloSpan [YHA03] and BIDE [WH04] were two well-known closed sequence mining algorithms, where CloSpan still follows the candidate maintenance-and-test paradigm and BIDE adopts BI-Directional Extension to avoid candidate maintenance.

7. Conclusion

This paper presents an efficient algorithm, *IPS-FXQPMiner* for mining frequent XML query patterns, which replaces expensive tree containment testing with

cheap parent-child validity checking. The novel techniques proposed in *IPS-FXQPMiner* include a unique sequence representation and an efficient frequent sequence enumeration method. More importantly, the proposed sequence-based method can speed up mining frequent XML query patterns through checking the parent-child relationship.

We introduce an effective index for incremental mining frequent query patterns, and accordingly, an efficient incremental mining algorithm is proposed, which can incrementally mine frequent patterns for the evolving transaction database effectively.

The thorough experimental results give us rich confidence to believe that our algorithms outperform existing algorithms in terms of efficiency, scalability as well as answerability.

Acknowledgement

This work is supported by the National Natural Science Foundation of China under Grant No. 60573094, Tsinghua Basic Research Foundation under Grant No. JCqn2005022, Basic Research Foundation of Tsinghua National Laboratory for Information Science and Technology (TNList), Zhejiang Natural Science Foun-dation under Grant No. Y105230, and 973 Program under Grant No.2006CB303103.

References

[AAK+02] T. Asai, K. Abe, S. Kawasoe et. al. Efficient Substructure Discovery from Large Semi-structured Data. In SDM, 2002.

[AJY+02] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick, Sequential PAttern Mining using a Bitmap Representation. In SIGKDD, 2002.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In VLDB, 1994.

[BOB+04]A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A Framework for Using Materialized XPath Views in XML Query Processing. In VLDB, pages 60-71, 2004.

[BWJ98] C. Bettini, X. Wang, and S. Jajodia. Mining Temporal Relationships with Multiple Granularities in Time Sequences. Data Engineering Bulletin 21(1), 1998.

[CLO03] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In SIGMOD, 2003.

[CYW04]Yi Chen, Lianghuai Yang, Yu Guo Wang. Incremental Mining of Frequent XML Query Patterns. In ICDM, 2004.

[DTK98] L. Dehaspe, H. Toivonen, R. D. King. Finding Frequent Substructures in Chemical Compounds. In KDD, 1998.

[HDY99] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In ICDE, 1999.

[HPM+00] J. Han, J. Pei, B. Mortazavi-Asl, et al. FreeSpan: Frequent pattern-projected sequential pattern mining. In SIGKDD, 2000.

[KK01] M. Kuramochi and G. Karypis. Frequent Subgraph Discovery. In ICDM, 2001.

[KRM+05] Joonho Kwon, Praveen Rao, Bongki Moon et al. FiST: Scalable XML Document Filtering by Sequencing Twig Patterns. In VLDB, 2005

[KSB+02] R. Kaushik, P. Shenoy, P. Bohannon et al. Exploiting Local Similarity for Efficient Indexing of Paths in Graph Structured Data. In ICDE, 2002.

[MCP98] F. Masseglia, F. Cathala, and P. Poncelet. The psp approach for mining sequential patterns. In PKDD, 1998.

[MS99] T. Milo and D. Suciu. Index structures for Path Expressions. In ICDT, 1999.

[MS05] B. Mandhani, D. Suciu. Query Caching and View Selection for XML Databases, In VLDB, 2005.

[ORS98] B. Ozden, S. Ramaswamy, and A. Silberschatz, Cyclic association rules. In ICDE, 1998.

[P18] H.Prufer. Neuer Beweis eines Satzes uber Permutationen. Archiv fur Mathematik und Physik, 27:142–144, 1918.

[PHM+01] J. Pei, J. Han, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In ICDE, 2001.

[RM04] Praveen R. Rao and Bongki Moon. PRIX: Indexing and Querying XML Using Prufer Sequences. In ICDE, pages 288–299, 2004

[SA96] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In EDBT, 1996.

[TRS02] A. Termier, M. C. Rousset, M. Sebag. TreeFinder: a First Step towards XML Data Mining. In ICDM, 2002.

[WH04] J. Wang and J. Han. BIDE: Efficient Mining of Frequent Closed Sequences. In ICDE, 2004.

[WL00]Ke Wang and Huiqing Liu. Discovering Structural Association of Semi-structured data. IEEE TKDE, 2000,12 (3) .

[WPF+03] H. Wang, S. Park, W. Fan and Philip S. Yu. ViST: A dynamic index method for querying XML data by tree structures. In SIGMOD, 2003.

[YHA03] X. Yan, J. Han, and R. Afshar. CloSpan: Mining Closed Sequential Patterns in Large Databases. In SDM, 2003.

[YLH03] L.H. Yang, M.L. Lee, W. Hsu. Efficient Mining of XML Query Patterns for Caching. In VLDB, 2003.

[YLH+03] L.H. Yang, M.L. Lee, W. Hsu, and S. Acharya. Mining Frequent Query Patterns from XML Queris. In DASFAA, 2003.

[YYW+02] J. Yang, P.S. Yu, W. Wang, and J. Han. Mining long sequential patterns in a noisy environment. In SIGMOD,2002.

[Z02] M. Zaki. Efficiently Mining Frequent Trees in a Forest. In SIGKDD, 2002.