Data Min Knowl Disc (2009) 18:472–516
DOI 10.1007/s10618-009-0126-5

Incremental sequence-based frequent query pattern mining from XML queries

Guoliang Li · Jianhua Feng · Jianyong Wang · Lizhu Zhou

Received: 16 May 2008 / Accepted: 20 January 2009 / Published online: 12 February 2009

Abstract Existing algorithms of mining frequent XML query patterns (XQPs) employ a candidate generate-and-test strategy. They involve expensive candidate enumeration and costly tree-containment checking. Further, most of existing methods compute the frequencies of candidate query patterns from scratch periodically by checking the entire transaction database, which consists of XQPs transferred from user query logs. However, it is not straightforward to maintain such discovered frequent patterns in real XML databases as there may be frequent updates that may not only invalidate some existing frequent query patterns but also generate some new frequent query patterns. Therefore, a drawback of existing methods is that they are rather inefficient for the evolution of transaction databases. To address above-mentioned problems, this paper proposes an efficient algorithm ESPRIT to mine frequent XQPs without costly tree-containment checking. ESPRIT transforms XML queries into sequences using a one-to-one mapping technique and mines the frequent sequences to generate frequent XQPs. We propose two efficient incremental algorithms, ESPRIT-i and ESPRIT-i^+, to incrementally mine frequent XQPs. We devise several novel optimization techniques of query rewriting, cache lookup, and cache replacement to improve

Responsible editor: M. J. Zaki.

G. Li (✉) · J. Feng · J. Wang · L. Zhou
Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China
e-mail: liguoliang@tsinghua.edu.cn

J. Feng
e-mail: fengjh@tsinghua.edu.cn

J. Wang
e-mail: jianyong@tsinghua.edu.cn

L. Zhou
e-mail: dcszlz@tsinghua.edu.cn

the answerability and the hit rate of caching. We have implemented our algorithms and conducted a set of experimental studies on various datasets. The experimental results demonstrate that our algorithms achieve high efficiency and scalability and outperform state-of-the-art methods significantly.

Keywords XML query patterns · Frequent query patterns · XML frequent pattern mining · Incremental mining · Sequential pattern mining

1 Introduction

XML has become a standard for information representation and exchange over the Internet. Many researchers have studied the problem of XML indexing (Milo et al. 1999; Kaushik et al. 2002; Qun et al. 2003), XML document clustering (Aggarwal et al. 2007), frequent XQP discovering (Chen et al. 2004; Li et al. 2006b; Yang et al. 2003a,b), and XML query caching and answering (Yang et al. 2003a; Balmin et al. 2004; Mandhani and Suciu 2005; Li et al. 2006a; Feng et al. 2007).

Discovering frequent XML query patterns (XQPs) 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 thus discovering frequent XQPs has attracted great attention recently as the answers of these queries can be stored and cached so as to improve query performance. The advantage of caching is that when a user refines a query by adding or removing one or more query terms, many of the answers that have already been cached can be delivered to the user right away. This avoids the expensive evaluation of repeated or similar queries. For a heavily loaded backend server, these savings can be significant. This kind of middle-tier caching has become popular for web applications using relational databases (Luo et al. 2002). Further, the caching module can also be maintained on a different database system, on a remote host. Thus, unlike the page-based buffer cache, it can be employed in a distributed setting too. The caching system can also be employed in a setting like distributed XQuery (Re et al. 2004) where sub-queries of a query might refer to remote XML data sources, connected over a WAN. Here, a sub-query that hits in the local cache will not have to be sent over the network, and the savings can be significant.

It is a big challenge to select suitable queries for effectively caching. Existing methods (Yang et al. 2003a,b, 2004) mine the frequent XML queries for caching. FastXMiner (Yang et al. 2003a) and 2PXMiner (Yang et al. 2004) are the state-of-the-art algorithms on this topic. Given the user log database composed of a set of XML queries, they model the queries as trees with special XML query constructs like descendant edges ('//') or wildcards ('*'), and extend frequent structure mining techniques to extract frequent subtrees following XML query semantics. They typically follow a candidate generate-and-test strategy, which includes two steps: (1) enumerate the candidates by extending an edge from query patterns that have been mined or joining two frequent query patterns; and (2) test the candidates through tree-containment checking. The first step is expensive to join XQPs if there are large numbers

of frequent query patterns. The second step is rather expensive as it has been proven to be NP-complete (Yang et al. 2003a). 2PXMiner proposes the transaction summary structure to reduce the number of costly tree-containment tests.

Another drawback of existing methods is that they cannot handle the evolution of log databases. They assume the user logs are static and always compute the frequencies of candidate query patterns from scratch in order to get the up-to-date frequent query patterns. However, the log database is frequently updated at a relatively high rate, and existing methods are rather inefficient to mine the dynamic log database using a from-scratch way. Moreover, it is not straightforward to maintain such discovered frequent $XQPs$ in the real XML-DBMS. Because the updates may not only invalidate existing frequent $XQPs$ but also generate some new frequent $XQPs$. Accordingly, it is important to study the efficient algorithms for incremental updates of frequent $XQPs$ as the query collection changes. Chen et al. (2004) proposed an algorithm increQPMiner, which incrementally mines the frequent XML queries based on the same candidate generate-and-test strategy as FastMiner (Yang et al. 2003a).

Based on above observations, in this paper, we propose an Efficient Sequence-based mining of frequent Pattern algoRIThm, namely ESPRIT. ESPRIT transforms XML queries into sequences with a one-to-one mapping, and extends the well-known algorithm PrefixSpan (Pei et al. 2001) to mine frequent sequences with constraint so as to generate frequent XQPs. To facilitate the mining of evolving databases, we propose two efficient incremental algorithms ESPRIT-i and ESPRIT-i^+ to incrementally mine frequent XML query patterns by employing novel indices and making full use of the frequent sequences that have already been mined. To summarize, we make the following contributions:

- We transform XQPs into sequences using a one-to-one mapping from an XQP to an Inverted Labeled Prüfer Sequence (ILPS). We mine frequent XQPs by mining frequent sequences.
- We devise two novel algorithms ESPRIT-i and ESPRIT-i^+ to incrementally mine frequent XQPs from continuously updated databases. We propose two efficient indices to facilitate the incremental mining and incorporate the novel indices into our incremental algorithms.
- We examine several optimization techniques of query rewriting, cache lookup, and cache replacement to improve the answerability and the hit rate of caching.
- We have implemented our proposed algorithms and conducted an extensive set of experimental studies. The experimental results show that our algorithms achieve much better performance and scalability, and outperform the state-of-the-art approaches significantly.

The rest of this paper is organized as follows. We formalize the frequent XQP mining problem in Sect. 2. Section 3 presents our sequence based algorithm ESPRIT to mine frequent XQPs on static databases. In Sect. 4, we present two efficient algorithms ESPRIT-i and ESPRIT-i^+ to incrementally mine frequent XQPs. We present optimization techniques to improve cache performance in Sect. 5. A thorough experimental study is demonstrated in Sect. 6. Section 7 reviews related works. We conclude the paper in Sect. 8.

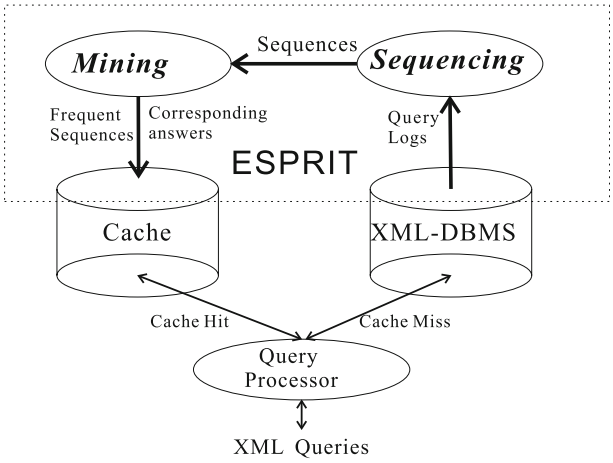


Fig. 1 The architecture of ESPRIT

2 Sequence-based frequent XML query pattern mining

We transform XML queries into sequences with a one-to-one mapping, and extend algorithm PrefixSpan (Pei et al. 2001) to mine frequent sequences with constraint in order to generate frequent XQPs. Figure 1 illustrates the overall architecture of our strategy to mine frequent XQPs from scratch. Moreover, we can also mine the evolving databases by making use of the mined results of the original databases. Note that the difference between them is that the incremental mining utilizes the mined results for efficiently discovering up-to-date frequent query patterns.

2.1 Notations

This section briefly introduces some notations for ease of presentation. XML queries are mainly expressed with XPath or XQuery, which conform to the regular path expressions. An XML query is usually modeled as a rooted, unordered and labeled tree.[1] In an XML query tree, each vertex denotes a node of the XML query, and each edge denotes the relationship between two nodes connected by the edge. In addition to element label/tag names, a query pattern tree may also contain wildcards '*' and relative paths '//'. The wildcard '*' indicates ANY label/tag in the DTD, while the relative path '//' indicates zero or many labels (i.e., the descendant-or-self relationship). Now, we formally model XML query patterns as a tree structure as follows.

Definition 1 (XQP). An XQP is modeled as a tree $XQP = <\mathcal{V}, \mathcal{E}>$, where $\mathcal{V}$ is the vertex set and $\mathcal{E}$ 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 {'//', '*'} $\cup$tagSet, where

[1] Note that, we can also model XML queries as ordered trees. This is a special case of modeling XML queries to unordered trees and our method can also apply to this case.

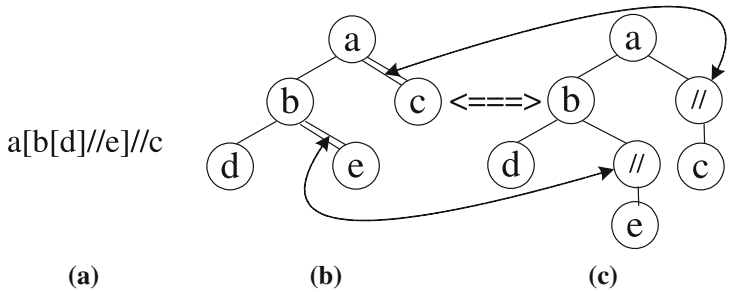


Fig. 2 XML query pattern. **a** An XPath, **b** a general XML query, **c** XQP

tagSet is the set of all element and attribute names in the schema. We define a partial order $\prec$. For any label $t \in$ tagSet, $t \prec$'*'$\prec$'//' denotes that t can be matched by '*', which in turn can be matched by '//'. '$*$' can match a single label while '//' can match zero or many labels.

Note that, for simplicity we assume that all queries are issued to an XML document, and thus all XQPs are targeted at the same XML tree. This assumption is valid and employed in the literature (Yang et al. 2003a). Moreover, our method can be adapted to the case when multiple XML documents are used. To differentiate '/' and '//' and make an XQP conform to a general tree, we use XQP to represent a query pattern by taking '/' and '//' as nodes in XQP. For example, in Fig. 2, given an XPath (a), we model it as a tree (b). We take the edge "//" as a node and get the XQP (c). XQPs in (a) and (b) are equivalent. XQPs are represented in the form of (b) in this paper.

Given an XQP xqp, we denote root(xqp) as the root of xqp. Given a node q in xqp, subtree(q) denotes the subtree rooted at q and children(q) returns the set of children of q. Based on these notations, we review two concepts of *rooted subtree* and *extended subtree inclusion* (Yang et al. 2003a).

Definition 2 *Rooted SubTree* (RST) (Yang et al. 2003a). Given an $XQP = <\mathcal{V}, \mathcal{E}>$, $RST = <\mathcal{V}', \mathcal{E}'>$ is a rooted subtree, if it satisfies, (1) root (RST) = root (XQP); and (2) $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{E}' \subseteq \mathcal{E}$.

To discover frequent query patterns, one important issue is testing the occurrence of a pattern in the database. We adopt the concept of *extended subtree inclusion*, a sound approach to test the containment of different query pattern trees (Feng et al. 2006; Li et al. 2006b; Yang et al. 2003a, 2004).

Definition 3 *Extended Subtree Inclusion* (Yang et al. 2003a). Let subtree(p) and subtree(q) be two subtrees from the same tree with root nodes p and q respectively. subtree(p) is included/contained in subtree(q) (subtree(q) is also said to include/contain subtree(p)), denoted by subtree(p) $\subseteq$ subtree(q), if $p \prec q$, and subtree(p) and subtree(q) satisfy,

1. both p and q are leaf nodes; or
2. p is a leaf node and $q =$ //, and $\exists q' \in$ children(q), subtree(p) $\subseteq$ subtree(q'); or
3. both p and q are non-leaf nodes, and one of the following conditions holds:

(i) $\forall p' \in \mathsf{children}(p), \exists q' \in \mathsf{children}(q), \mathsf{subtree}(p') \subseteq \mathsf{subtree}(q')$; or
(ii) $q = //$ and $\forall p' \in \mathsf{children}(p), \mathsf{subtree}(p') \subseteq \mathsf{subtree}(q)$; or
(iii) $q = //$ and $\exists q' \in \mathsf{children}(q), \mathsf{subtree}(p) \subseteq \mathsf{subtree}(q')$.

Based on Definition 3, we can recursively check whether an RST is included by an XQP. Accordingly, we can account the occurrence of an RST in a given database of XQPs transformed from user query logs. For example, Fig. 3 gives some XQPs and RSTs. The XQPs in (d), (e), and (f) are the rooted subtrees of XQPs in (a), (b), 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. Figure 4 shows the inclusion.

A transaction database $\mathcal{D}$ is a collection of XQPs, $\mathcal{D} = \{XQP_1, XQP_2, \ldots, XQP_n\}$, which is transformed from a set of XML queries $\{q_1, q_2, \ldots, q_n\}$, issued against a given XML data source. To compute the occurrences of an XQP in $\mathcal{D}$, we use the concept of *absolute support* and *relative support*. The absolute support of a rooted subtree RST refers to the total occurrences of XQPs in $\mathcal{D}$ which contain RST, denoted as $ASupp^{\mathcal{D}}(RST)$. The relative support is the percentage of XQPs that contain RST in $\mathcal{D}$, denoted as $RSupp^{\mathcal{D}}(RST) = ASupp^{\mathcal{D}}(RST)/|\mathcal{D}|$, where $|\mathcal{D}|$ is

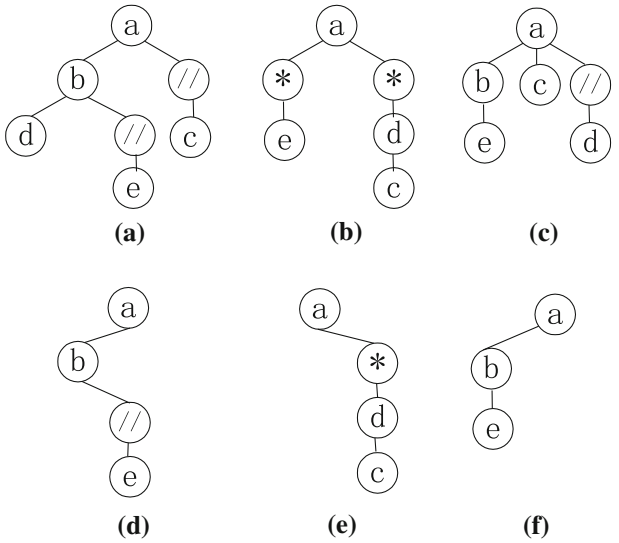


Fig. 3 XQPs and RSTs. **a** XQP_a, **b** XQP_b, **c** XQP_c, **d** RST_a, **e** RST_b, **f** RST_c

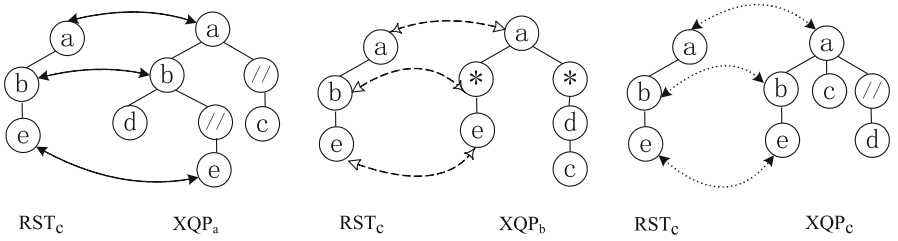


Fig. 4 XQP inclusion

the size of $\mathcal{D}$, i.e., the number of $XQPs$ in $\mathcal{D}$. For example, consider the $XQPs$ in Fig. 3, we have $ASupp^{\mathcal{D}}(RST_c) = 3$ and $ASupp^{\mathcal{D}}(RST_b) = 2$ as RST_c is included in XQP_a and XQP_b (also in XQP_c); RST_b is included in XQP_a (also in XQP_b) but not in XQP_c. As there are three $XQPs$, $RSupp^{\mathcal{D}}(RST_b) = \frac{2}{3}$.

Now, we formalize the problem of *frequent XQP mining* as follows.

Frequent XQP Mining Given an XQP database $\mathcal{D}$ and a minimum relative support min_sup in the range of (0, 1], *frequent XQP mining* finds a complete set, denoted as $\mathcal{F}(\mathcal{D})$, of all frequent $RSTs$ in $\mathcal{D}$ such that for each RST in $\mathcal{F}(\mathcal{D})$, $RSupp(RST) \geq min\_sup$ holds. We adopt the relative support as our measure of frequency in this paper.

2.2 Incremental frequent XQP mining

Traditionally, the transaction database is dynamically changing as new queries will be added into the database. Thus, the updates of new queries may not only invalidate some existing frequent query patterns but also generate some new frequent query patterns. However, general mining algorithms always directly mine the updated database from scratch, and thus involve too many scans on the original database $\mathcal{D}$ and the incremental database d without utilizing the existing mined frequent query patterns that have been already gotten ($\mathcal{F}(\mathcal{D})$). To make full use of $\mathcal{F}(\mathcal{D})$, in this paper, we study the problem of incremental frequent XQP mining. To formalize this problem, the concepts of *incremental database* and *updated database* (Chen et al. 2004) are used in this paper.

Definition 4 *Incremental Database and Updated Database* (Chen et al. 2004). Suppose that a set of new XQPs, d, is to be added into the transaction database $\mathcal{D}$. The database $\mathcal{D}$ is referred to as the original database, the database d as the incremental database, and the database $\mathcal{D}^u = \mathcal{D} + d$ as the updated database.

Now, we introduce the problem of *incremental frequent XQP mining* as follows.

Incremental Frequent XQP Mining Given an original database $\mathcal{D}$ with its frequent $RSTs$, $\mathcal{F}(\mathcal{D})$, a minimum relative support min_sup in the range of (0, 1], and the incremental database d. *Incremental frequent XQP mining* is finds the complete set of all frequent $RSTs$ in $\mathcal{D} + d$.

Note that our algorithms can be extended to handle the case of deletions. For deletions, frequent sequences may cause to be infrequent after deletions. We only need scan the deleted portion and update the corresponding frequencies.

2.3 Frequent subsequence mining

2.3.1 Sequencing

We transform $XQPs$ to sequences with a one-to-one mapping and mine frequent sequences to generate frequent $XQPs$. Many sequencing methods have been proposed for XML indexing, such as depth-first and Prüfer (Rao and Moon 2004; Wang et al. 2003). Prüfer sequence (Rao and Moon 2004; Kwon et al. 2005) and ViST (Wang et al. 2003) 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. One can generate a labeled tree's Prüfer sequence by iteratively removing nodes from the tree until only two nodes remain. Consider a labeled tree with nodes $\{1, 2, \ldots, n\}$. At step i, one can remove the leaf with the smallest label and set the ith element of the Prüfer sequence to be the label of this leaf's parent. The sequence of a labeled tree is clearly unique and has length $n - 2$. In fact, one can construct a Prüfer sequence of length $n - 1$ by continuing the removal of nodes until only one node is left. Any numbering scheme can be used to label an XML document tree as long as it associates each node with a unique number. This guarantees a one-to-one mapping between the tree and the sequence.

Rao and Moon (2004) used the post-order to uniquely number tree nodes. The sequence consisting of post-order numbers is called Numbered Prüfer Sequence (NPS) (Rao and Moon 2004). When each number in an NPS is replaced by its corresponding tag, a new sequence that consists of XML tags can be constructed, and this sequence is called Labeled Prüfer Sequence. On the basis of Labeled Prüfer Sequence (LPS), Extended Labeled Prüfer Sequence (ELPS) and Extended Numbered Prüfer Sequence (ENPS) can be constructed by extending leaf nodes of the document tree with dummy child nodes as in Rao and Moon (2004). Clearly the leaf node labels of the original tree are kept in ELPS. However, NPS, LPS, ELPS and ENPS are not suitable to mining frequent XQPs as the first node in the sequences is not the root node. We introduce Inverted Labeled Prüfer Sequence (ILPS) and Inverted Numbered Prüfer Sequence (INPS), which invert the ELPS and ENPS respectively. ILPS (INPS) contains the same information provided by ELPS (ENPS).

Note that ILPS captures the parent–child and the sibling–order relationships, which can facilitate the mining of frequent XQPs. It is easy to figure out that *post-order* preserves the sibling–order relationship and ILPS keeps the parent–child and the ancestor–descendant relationships (Lemma 1). Note that some properties in Lemma 1 are supplementary to those in Rao and Moon (2004) and build upon some of the results already provided in Rao and Moon (2004).

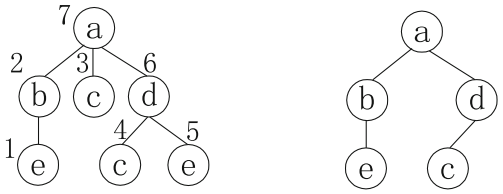
Lemma 1 *Suppose* $(e_1, e_2, \ldots, e_m)$ *and* $(n_1, n_2, ..., n_m)$ *are respectively ILPS and INPS of an XQP.* $\forall i, j, 1 \leq i < j \leq m$, *we have,*

1. *If* $n_i > n_{i+1}$, e_i *is the parent of* e_{i+1}*; and*
2. *If* $n_i < n_j$, e_j *is an ancestor of* e_i*; and*
3. *If* $n_i > n_j$ *and* $\nexists t, i < t < j, n_i > n_t > n_j$, e_i *is the parent of* e_j.

Proof We first prove (1). As $n_i > n_{i+1}$, e_{i+1} must be removed before e_i according to the removal-based sequencing method. As e_{i+1} and e_i are neighbors, e_i is the parent of e_{i+1} according to the post-order encoding scheme.

We then prove (2). As $n_i < n_j$, e_i must be removed before e_j. According to the sequencing method, all the nodes, which have larger post-order than n_i, must be removed after e_i, thus those nodes must be before e_i in INPS based on the construction method of INPS. As $i < j$ and $n_i < n_j$, e_j must be an ancestor of e_i.

We finally prove (3). As $n_i > n_j$, e_i must be removed after e_j. e_i may be an ancestor of e_j or a following sibling of e_j. As $\nexists t, i < t < j, n_i > n_t > n_j$, e_i must be the



(a) **(b)**

(c)

	LPS	ELPS	ILPS	NPS	ENPS	INPS
XQP	baadda	ebacacdeda	adedcacabe	277667	1273746567	7656473721
RST	bada	ebacda	adcabe	2767	127467	764721

Fig. 5 An example XQP and RST. **a** XQP, **b** RST, **c** sequences

parent of e_j, as there is no node between e_i and e_j which has larger post-order than n_j. □

Example 1 In Fig. 5, we construct LPS and NPS of the corresponding XQP and RST according to the removal-based sequencing method. ELPS can be constructed by inserting leaf nodes, i.e., e, c, c, e, into the corresponding positions of LPS. We note that the leaf node must be preceding and neighboring its parent. We can get ILPS of Fig. 5a, `adedcacabe` by inverting its ELPS `ebacacdeda`. ILPS of the RST in Fig. 5b is `adcabe`, and its INPS is `764721`.

Consider ILPS and INPS of the RST in Fig. 5b, `adcabe` and `764721`. As $n_2(6) > n_3(4)$, $e_2 = d$ is a parent of $e_3 = c$ according to Lemma 1(1); as $n_3(4) < n_4(7)$, $e_3 = c$ is a descendent of $e_4 = a$ according to Lemma 1(2); as $n_1 > n_4$, and $n_2 \not> n_4$ and $n_3 \not> n_4$, $e_1 = a$ is the parent of e_4=c according to Lemma 1(3).

2.3.2 *Standardization of XQPs*

Note that the sequences of the equivalent XQPs may be different. The reason is that ILPS keeps the sibling order, but general XML queries do not require it. For example, the four XML queries in Fig. 6 are equivalent, and their absolute support should be four. But their ILPSs are not the same, and the absolute support of each query is one.

To address this issue, we introduce a concept of *standard form* and transform the XML queries into their standard forms. Given two XQPs, $\mathtt{xqp}_1$ and $\mathtt{xqp}_2$, if the root label of $\mathtt{xqp}_1$ is smaller than that of $\mathtt{xqp}_2$ in lexicographical order, we say $\mathtt{xqp}_1$ is smaller than $\mathtt{xqp}_2$; Otherwise, we sort their children in lexicographical order and compare the subtrees rooted at their children one by one. If we find a subtree of $\mathtt{xqp}_1$ is smaller than that of $\mathtt{xqp}_2$, we say $\mathtt{xqp}_1$ is smaller than $\mathtt{xqp}_2$. For example, we have $a[b][c]/d$ is smaller than b/d, $a[b][c]/d$ is smaller than $a[b]/d$, and $a[b/c]/d$ is smaller than $a[b/d]/d$.

Definition 5 *Standard Form* Given an XQP `xqp`, `xqp` is a standard form if for any node in `xqp`, the subtrees rooted at its children are sorted.

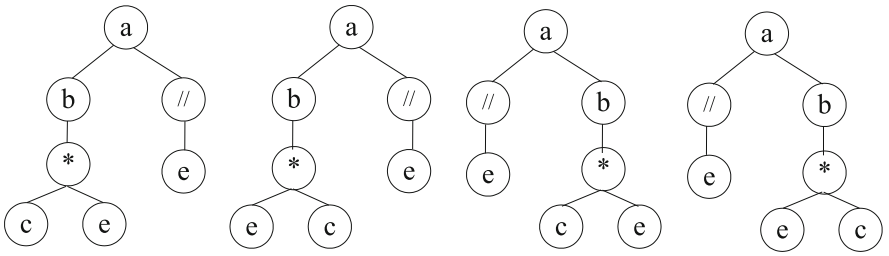


Fig. 6 Four equivalence queries

Given an XQP, we can get its standard form as follows. For the node which only has leaf children, we sort its children by their labels in lexicographical order. For the node which has non-leaf children, we sort its children by comparing the subtrees rooted at the children. For example, the four queries in Fig. 6 can be normalized to their standard form, XQP_i. Accordingly, all the equivalent queries can be transformed into a unique standard form and we use the sequence of the standard form to represent the equivalent queries.

Given an xqp, its sequencing cost is $O(|\texttt{xqp}|)$ as shown in Rao and Moon (2004), where $|\texttt{xqp}|$ is the number of nodes in xqp. Here, we give the standardizing cost in Lemma 2.

Lemma 2 *Given an XQP xqp, the cost of standardizing xqp is $O(|\texttt{xqp}|^2)$.*

Proof We standardize an XQP from leaf nodes to the root. For the node which only has leaf children, we only need compare their children and sort the children in lexicographical order. For the node which has non-leaf children, we need compare the subtree rooted at such children. Note that the subtree must have been standardized and the nodes with the same parent must be sorted. Thus, we only need traverse the subtree in pre-order and compare corresponding nodes. In the worst case, we at most compare all the node pairs in xqp, and thus the complexity of standardizing xqp is $O(|\texttt{xqp}|^2)$. □

Moreover, there is a one-to-one mapping between equivalent queries and a certain $ILPS$ as formalized in Lemma 3.

Lemma 3 *There is a one-to-one mapping between equivalent XQPs and ILPSs.*

Proof It is easy to figure out that any XQP can be uniquely represented by an $ILPS$ according to the sequencing method. On the other hand, according to the standardization of XQPs, all the equivalent XQPs have a same $ILPS$. Hence, there is a one-to-one mapping between equivalent XQPs and $ILPS$s. □

2.3.3 Valid subsequence

Let $\mathcal{S} = e_1, e_2, \ldots, e_n$ denote a sequence with length n. Given two sequences $\mathcal{S}_a = a_1, a_2, \ldots, a_n$ and $\mathcal{S}_b = b_1, b_2, \ldots, b_m (n \leq m)$, if there exists $1 \leq i_1 < i_2 < \cdots < i_n \leq m, a_1 = b_{i_1}, a_2 = b_{i_2}, \ldots$, and $a_n = b_{i_n}$, $\mathcal{S}_b$ is called a supersequence of $\mathcal{S}_a$.$\mathcal{S}_a$ is said to be a subsequence of $\mathcal{S}_b$. Some subsequences may be invalid as they may not represent an RST of an XQP and even cannot follow a tree structure constraint. Denote $RST^L(RST^N)$ as the ILPS (INPS) of an RST. For example, in Fig. 5, $RST^L = adcabe$ is a subsequence of $XQP^L = adedcacabe$. It can represent a valid RST of this XQP. However, some sequences (e.g., $aeace$) cannot represent a valid RST. To address this issue, we introduce the concept of *valid subsequence*.

Definition 6 *Valid Subsequence*. Given an XQP and its ILPS $\mathcal{S}, \mathcal{S}_a$ is a valid subsequence of $\mathcal{S}$, iff $\mathcal{S}_a$ is a subsequence of $\mathcal{S}$ and the subtree that $\mathcal{S}_a$ represents is an RST of the XQP.

We can test whether a subsequence is a valid subsequence by checking ancestor–descendant and sibling–order relationships. Note that any subsequence must follow the sibling–order relationship. Thus, we only need check whether the nodes in the subsequence are connected by checking their INPSs in Lemma 4.

Lemma 4 *Consider an INPS* $\mathcal{S} = S_1, S_2, \ldots, S_n$.$\mathcal{S}_a = s_1, s_2, \ldots, s_m$ *is a subsequence of* $\mathcal{S}(s_1 = S_{i_1}, s_2 = S_{i_2}, \ldots,$ *and* $s_m = S_{i_m})$. $\mathcal{S}_a$ *is a valid subsequence of* $\mathcal{S}$, *if it satisfies,*

1. $s_1 = max(S_1, S_2, \ldots, S_n)$[2]*; and*
2. $\forall k, 1 \leq k < m, s_k > s_{k+1}$ *and* $\nexists t, i_k < t < i_{k+1}$ *and* $S_{i_k} > S_t > S_{i_{k+1}}$.

Proof Suppose the XQP w.r.t. $\mathcal{S}$ is xqp. As $s_1 = max(S_1, S_2, \ldots, S_n)$, s_1 is the root of xqp. We only need to prove that the subtree w.r.t. $\mathcal{S}_a$ is a connected subtree of the XQP. We prove it by mathematical induction. Suppose $s_2, s_3, \ldots, s_k$ connect to s_1, we prove that s_{k+1} also connects to s_1, that is, s_{k+1} is a descendant of s_1. If $s_k > s_{k+1}$, S_{i_k} is the parent of $S_{i_{k+1}}$. As $\nexists t$, such that $i_k < t < i_{k+1}$ and $S_{i_k} > S_t > S_{i_{k+1}}$, s_{k+1} is a child of s_k according to Lemma 1. Thus, s_{k+1} connects to s_k. As s_k connects s_1, s_{k+1} also connects s_1. If $s_{k+1} > s_k$, s_{k+1} must be an ancestor of s_k according to Lemma 1. As s_k connects to s_1, its ancestors must also connect to s_1, that is, s_{k+1} connects to s_1. Thus, $\mathcal{S}_a$=$s_1, s_2, \ldots, s_m$ represents a rooted subtree, and $\mathcal{S}_a$ is a valid subsequence. □

We give a running example to show how to check whether a subsequence is valid. For example, in Fig. 5, $adcabe(764721)$ is a valid subsequence of $adedcacabe$ as it can represent a connected subtree. However, $aeace(75731)$ is not a valid subsequence of $adedcacabe$, because there is an item $d(6)$ between $a(7)$ and $e(5)$ (break the connection).

To compute the occurrence of a valid subsequence in a sequence database, we introduce the concept of *subsequence inclusion*, which corresponds to *extended subtree inclusion* in Definition 3.

[2] For ease of presentation, in this paper, given a sequence $\mathcal{S}_a = s_1, s_2, \ldots, s_m$, $\mathcal{S}_a$ can refer to ILPS or INPS if there is no ambiguous.

Definition 7 *Subsequence Inclusion*. Given two sequences s and S. Their ILPSs are $s_1, s_2, \ldots, s_p$ and $S_1, S_2, \ldots, S_m$, and their INPSs are $n_1, n_2, \ldots, n_p$ and $N_1, N_2, \ldots, N_m$ respectively. s is included/contained in S, if (1) there exists $1 \leq i_1 \leq i_2 \leq \cdots i_p \leq m$, s.t. $s_1 \prec S_{i_1}, s_2 \prec S_{i_2}, \ldots, s_p \prec S_{i_p}$; (2) '//' matches zero or many labels; (3) '//' cannot match any leaf label; (4) "*" can match any label; (5) if $n_k < n_{k+1}$, s_{k+1} must be a leaf node. If there is no label in S that does not appear in s, s is said to be properly included in S.

To better understand our concepts, we give a running example in EXAMPLE 2.

Example 2 In Fig. 2, $RST_c^L = abe$, $XQP_a^L = a//cab//ebd$, $XQP_b^L = a*dca*e$. RST_c^L is included in XQP_a^L and XQP_b^L, where (i_1, i_2, i_3) are (4,5,7), (5,6,7) respectively. Especially, in XQP_b^L, b is matched by *. abe is not properly included in XQP_a^L, because there is no item in abe to match d in XQP_a^L. abe is properly included in $RST_a^L(ab//e)$. $RST_b^L = a*dc$, is included in XQP_a^L, where (i_1, i_2, i_3, i_4) is (1,2,2,3). Especially, '*' and d are both matched by '//'.

Consider the test whether $a/b/f$ is included in $a/b//e$. If it is unknown that there exists a path $a/b/f//e$, 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 $a/b/f//e$. To address this issue, it is needed to take into account the DTD and perform expansions of the XQPs. Interested readers are referred to Yang et al. (2003a) for the details.

2.3.4 Frequent valid sequence mining

We transform a user log database with a set of XQPs to a sequence database with a set of corresponding sequences. A sequence database SDB contains a set of tuples in the form of $(Sid, \mathcal{S})$, where Sid is the identifier of a sequence $\mathcal{S}$. The absolute support of a valid subsequence vs is the number of tuples that contain vs in SDB, denoted by $ASupp(\texttt{vs})$. The relative support is the percentage of tuples that contain vs in SDB, denoted by $RSupp(\texttt{vs})$, where $RSupp(\texttt{vs}) = ASupp(\texttt{vs})/|\text{SDB}|$.

Now we introduce the problems of *frequent valid subsequence mining* and *incremental frequent valid subsequence mining*.

Frequent Valid Subsequence Mining. Given a sequence database $\mathcal{D}$ and a minimum relative support min_sup in the range of (0, 1], *frequent valid subsequence mining* finds a complete set of all frequent valid subsequences $vseq$s in $\mathcal{D}$.

Incremental Frequent Valid Subsequence Mining. Given an original sequence database $\mathcal{D}$ with its frequent valid subsequence set $\mathcal{F}(\mathcal{D})$, a minimum relative support min_sup in the range of (0, 1], and an incremental database d for $\mathcal{D}$. *Incremental frequent valid sequence mining* finds the complete set of all frequent valid subsequences $vseq$s in $\mathcal{D} + d$.

Note that frequent XQP mining problem can be transformed to frequent valid subsequence mining problem, and incremental frequent XQP mining problem can be transformed to incremental frequent valid subsequence mining problem. In this paper, we mine frequent sequences for generating frequent XQPs.

3 ESPRIT: an efficient sequence based frequent XML query pattern mining algorithm

This section proposes ESPRIT, an Efficient Sequence based frequent XML query PatteRns mining algorIThm. ESPRIT transforms XQPs to sequences and mines frequent sequences to generate frequent XQPs by extending algorithm PrefixSpan (Pei et al. 2001).

3.1 Preprocessing

Given a set of user's XML queries, $\mathcal{Q} = \{q_1, q_2, \ldots, q_m\}$, extracted from user query logs. In the preprocessing phase, ESPRIT first transforms the input XML queries into their standard forms as described in Section 2.3.2, and then constructs the sequence database $\mathcal{D}$ by transforming queries into their sequence representations ($ILPS$ and $INPS$). Accordingly, we can get the sequence database $\mathcal{D} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n\}$. We note that $n \leq m$ as there may be some equivalent queries in $\mathcal{Q}$.

For example, in Fig. 7a, b respectively depict the original database $\mathcal{D}$ and the incremental database d. We can transform the XQPs in Fig. 7 into their sequences as illustrated in Table 1. Table 1 shows the corresponding sequence databases. We take these sequences as our running example throughout this paper.

3.2 Valid subsequence extension

Existing sequence mining algorithms construct a sequence tree to mine frequent sequences. Assume there is a lexicographical ordering $\preceq$ among the set of items (i.e.,

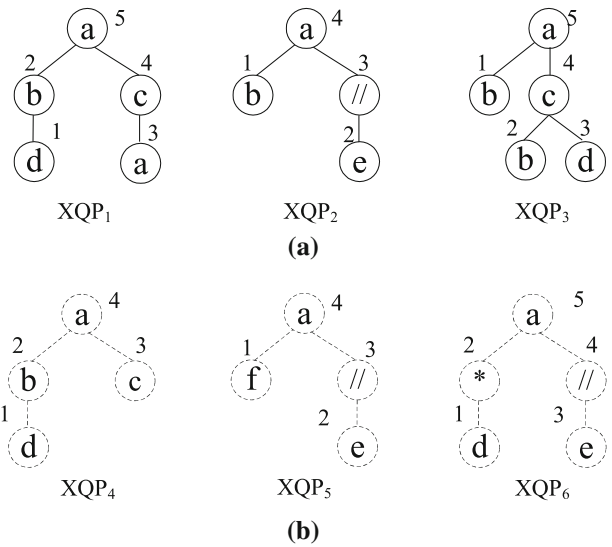


Fig. 7 A running example **a** the original database $\mathcal{D}$ and **b** the incremental database d

Table 1 $XQPs$ and the corresponding ILPS and INPS of the $XQPs$ in Fig. 7

SDB	XQP	Sid	ILPS	INPS
$\mathcal{D}$	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

the labels in tagSet and '*', '//'), $\mathcal{I}$, in the input sequence database (e.g., in our running example, one possible items ordering can be $a \preceq b \preceq c \preceq d \preceq e \preceq f \preceq * \preceq //$). Conceptually, the complete search space of sequence mining forms a sequence tree with the tree nodes of $XQPs$. The process of constructing the sequence tree is as follows. The root node of the tree is the root of the give XML document. (If there are multiple documents, the root is a dummy node.) 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 node N are generated and arranged according to the chosen lexicographical ordering. Recursively, we can construct the sequence tree as shown in Fig. 8. In Fig. 8, each node contains a frequent sequence and keeps its corresponding absolute support. As an assumption, min_sup is $\frac{1}{2}$ in all the examples of this paper.

Apparently, not all the frequent sequences in Fig. 8 correspond to valid tree structures. For instance, "ad" is not a valid frequent sequence though with support of 2, because "ad" is not a valid subsequence according to Lemma 4. Following we will briefly introduce the valid frequent sequence enumeration method, and discuss how to push the parent–child relationship constraint into the sequence enumeration to make sure each mined frequent sequence is a valid sequence.

3.2.1 Frequent sequence enumeration

Given a sequence database, most of previous frequent pattern mining algorithms (Pei et al. 2001) have elaborated that the depth-first searching is more efficient in mining long patterns than the breadth-first searching. Thus, we also traverse the $XQPs$ in depth-first order. Pei et al. (2001) introduced 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

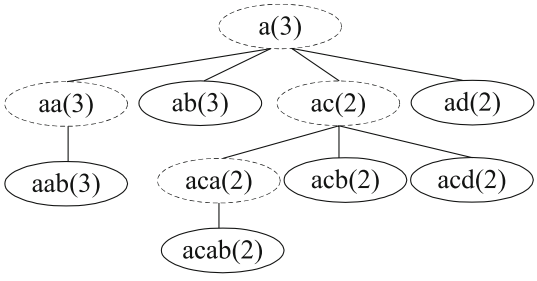


Fig. 8 The lexicographic frequent sequence tree in $\mathcal{D}$

is always treated as a prefix sequence. By adding one item in $\mathcal{I}$, a prefix sequence can grow to be a longer sequence as its child node. With respect to the corresponding prefix sequence, some items are not locally frequent. As our goal is to mine the frequent sequences, we only need to extend a prefix sequence using the set of its locally frequent items based on the downward closure property (Agrawal and Srikant 1994). To the best of our knowledge, to identify the local frequent items w.r.t a certain prefix, a well-known method is to build the projected database for the prefix and scans the projected database to count the local items. We employ the concept of the *projected sequence* of a given prefix sequence.

Definition 8 *Projected Sequence* (Pei et al. 2001). Given an input sequence $\mathcal{S}$, which contains a prefix $\mathcal{S}_i = e_1, e_2, \ldots, e_i$, the remaining part of $\mathcal{S}$ after we remove the first instance of the prefix $\mathcal{S}_i$ in $\mathcal{S}$ is called the projected sequence w.r.t. prefix $e_1, e_2, \ldots, e_i$ in $\mathcal{S}$.

Definition 9 *Projected Database* (Pei et al. 2001). Given an input sequence database SDB, the complete set of projected sequences in SDB w.r.t. a prefix sequence $e_1, e_2, \ldots, e_i$ is called the projected database w.r.t. prefix $e_1, e_2, \ldots, e_i$ in SDB.

For example, consider the sequence database in Table 1, the projected sequence of prefix sequence ab w.r.t. sequence $\mathtt{xqp}_1(acaabd)$ is d. The projected database of prefix sequence ab w.r.t. $\mathcal{D}$ is $\{d, \phi, ab\}$.

Now, we introduce the idea of pseudo-projection. Instead of physically constructing the projected database, we only need to keep a set of pointers, one for each projected sequence, pointing at the starting position in the corresponding projected sequence. By following the set of pointers, it is easy to locate the set of projected sequences. And by scanning forward each projected sequence w.r.t. a prefix S_p and count the items, which is the so-called forward-extension step (Wang and Han 2004), we will find the locally frequent items w.r.t. prefix S_p, which can be used to extend prefix S_p in order to get longer frequent prefix sequences. For example, consider $\mathcal{D}$ in Table 1, suppose $S_p = ac$, the set of its local frequent items is $\{a, b, d\}$ as shown in Fig. 8. However, how to extend local frequent valid items remains an issue for discovering frequent valid sequences. To address this issue, we introduce the technique of valid subsequence extension in Sect. 3.2.2.

3.2.2 Valid subsequence extension

There may be many subsequences of a certain sequence, but not all of them are meaningful from the user/application point of view. As some of subsequences cannot represent subtrees of an XQP. Although we can check whether a subsequence is a valid subsequence according to Lemma 4, it needs to traverse the whole sequence. To effectively mine the frequent valid sequences, we always add a valid item to extend a sequence. An item is valid if (1) its parent appears in the prefix sequence; or (2) its parent is '//' and its grandparent appears in the prefix sequence; or (3) it has already appears in the prefix sequence. Definition 10 gives a formal description.

Definition 10 *Valid Local Item*. Given a prefix sequence $\mathcal{S}$ and its projected sequence PS, where $\mathcal{S}^L = e_1, e_2, \ldots, e_i$ and $\mathcal{S}^N = n_1, n_2, \ldots, n_i$; $PS^L = pe_1, pe_2, \ldots, pe_j$

and $PS^N = ne_1, ne_2, \ldots, ne_j$, a local item e w.r.t. this prefix sequence is called a valid local item, if e satisfies:

1. $\exists m, 1 \leq m \leq j, e \prec pe_m, ne_{m-1} = n_i$ and $ne_m < n_i (pe_0 = e_i)$; or
2. $\exists m, 2 \leq m \leq j, e \prec pe_m, pe_{m-1} = \text{'//'}, ne_{m-2} = n_i$ and $ne_m < n_i (pe_0 = e_i)$; or
3. $\exists m, 1 \leq m \leq j, e \prec pe_m, ne_m > n_i$.

The local items that satisfy (1)/(2) are the children of e_i. In (2) '//' matches zero label. The local items that satisfy (3) are e_i's ancestors, which can be proved as Lemma 4.

Based on this notation, we introduce the technique of *valid local item enumeration*. We employ the depth-first method to enumerate the local item as follows. The root node is initialized as the first frequent sequence s_1. Then we check whether each local item e of the given prefix sequence s_1 is a valid local item; if so, we count the number of the sequences that contain the valid local item e, i.e., $ASupp(s_2 = s_1 \bullet e)$. If $RSupp(s_2) \geq min\_sup$, s_2 is a frequent sequence. We iteratively enumerate the valid local items until there is no valid local item. When enumerating a valid local item, item '//' in a projected sequence can match any zero or more labels or '*', and '*' can match any one label in tagSet. We note that the sequence extension framework in our approach follows left most extension as formalized in Lemma 5, which complies with the right most extension strategy adopted in Yang et al. (2003a) and Zaki (2002). It removes redundancies in frequent valid sequence mining. We will experimentally prove that this strategy is efficient to extend a valid local item in Sect. 6.

Lemma 5 *Given a valid local item e w.r.t. a prefix sequence $e_1, e_2, \ldots, e_i$, whose parent is $e_m (1 \leq m \leq i)$, the tree node corresponding to e_m must be on the left most path of the subtree corresponding to prefix $e_1, e_2, \ldots, e_i$.*

Proof We first prove that e_i must be on the left most path of the subtree corresponding to prefix $e_1, e_2, \ldots, e_i$, and we prove it by contradiction.

Suppose e_i is not on the left most path, there must exist an integer j such that $j < i$ and e_j is on the left most path. According to the construction of INPS, e_j must be removed before e_i, and in the ILPS e_j must be after e_i, which conflicts with $j < i$, thus e_i must be on the left most path.

According to Definition 10, e must be a child or an ancestor of e_i, as a result, e_m, which is e_i or an ancestor of e_i, must be on the left most path of the subtree corresponding to prefix $e_1, e_2, \ldots, e_i$. □

Example 3 Consider the three sequences of $\mathcal{D}$ in Fig. 7, $\mathtt{xqp}_a^L = a//cab//ebd$ and $\mathtt{xqp}_b^L = a * dc$. Suppose the current prefix sequence is a. As '//' can match '*', '*' is a valid item w.r.t. a for $\mathtt{xqp}_a^L$; as '//' can match zero or more labels, its projected sequence can be $cab//edb$ or $//cab//ebd$. Thus, a is extended to $a*$. Then the next item d is checked. It is not a valid item w.r.t. $cab//edb$, because the item d in this projected sequence does not satisfy the semantics of the valid local item as described in Definition 10. However, it is a valid item w.r.t. $//cab//ebd$, because item d can be matched by '//'. Accordingly, $a * dc$ is included in $a//cab//ebd$.

Algorithm 1: ESPRIT Algorithm

Input: $\mathcal{D}$: the sequence database composed of transformed sequences;
min_sup: a given minimum support.
Output: $\mathcal{F}(\mathcal{D})$: the updated set of frequent valid subsequences in $\mathcal{D}$.

```
1 begin
2   F(D)=φ;  /*the set of frequent valid subsequences*/
3   F(D) = getFrequentValidSequence(φ,D,min_sup, F(D));
4 end
```

Function `getFrequentValidSequence` Function

Input: PS: a prefix sequence;
$\mathcal{D}_{PS}$: a projected sequence database w.r.t. PS;
min_sup: a given minimum support;
$\mathcal{F}(\mathcal{D})$: the current set of frequent valid subsequences in $\mathcal{D}$.
Output: $\mathcal{F}(\mathcal{D})$: the updated set of frequent valid subsequences in $\mathcal{D}$.

```
1  begin
2    if PS is non-empty then
3      F(D)←PS;
4    VLF_PS = getFrequentValidLocalItems(PS, D_PS, min_sup);   /*get the frequent
     valid local items w.r.t. PS in D_PS according to DEFINITION 10.*/
5    for each frequent valid local item e_i∈VLF_PS do
6      PS_i = PS•e_i; /*concatenate e_i to PS*/
7      D_PS_i = getPseudoProjectedDatabase(PS_i, D_PS);  /*get the pseudo projected
       database w.r.t. sequence PS_i*/
8      F(D)←PS_i;
9      F(D)←getFrequentValidSequence(PS_i, D_PS_i, min_sup, F(D));
10   return F(D);
11 end
```

Fig. 9 ESPRIT algorithm

3.3 ESPRIT algorithm

By integrating standardization, sequencialization, valid sequence extension and frequent sequence enumeration, we derive our algorithm, ESPRIT, as shown in Fig. 9, which avoids costly tree containment testing and prunes the unrelated search space efficiently under the local item's validity checking. ESPRIT enumerates the complete set of frequent sequences by extending frequent valid local items, which is similar to the pseudo-projection-based PrefixSpan algorithm (Pei et al. 2001). ESPRIT takes the sequence database of sequences transformed from user query logs as described in Sect. 3.1 as input and outputs the frequent valid sequences.

ESPRIT first initializes the frequent sequence set as ϕ (line 2) and then calls its subroutine getValidFrequentSequence to identify frequent valid subsequences (line 3), which is a recursive function to get frequent valid sequences by extending valid items.

getValidFrequentSequence is used to mine the frequent valid subsequences recursively. Given a certain prefix PS, if it is non-empty, getFrequentValidSequence adds it into $\mathcal{F}(\mathcal{D})$ (line 7), scans the projected database $\mathcal{D}_{PS}$ w.r.t. PS and identifies the frequent valid local items through getFrequentValidLocalItems based on Definition 10 (line 8). For each frequent valid local item e_i, which can be chosen in lexicographical order, getFrequentValidSequence grows PS to get a new

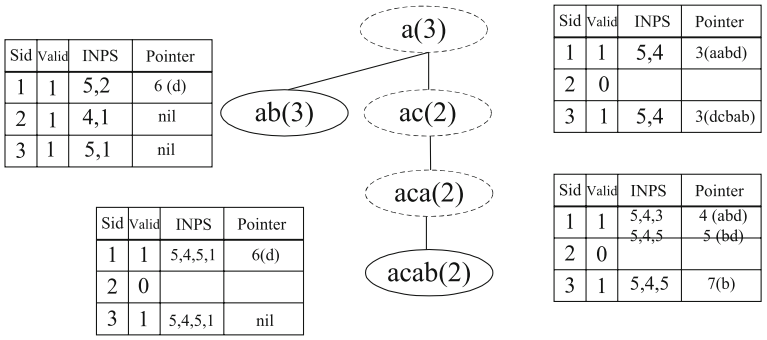


Fig. 10 Valid sequence extension in $\mathcal{D}$

prefix PS_i by extending a valid local item e_i (line 10), scans $\mathcal{D}_{PS}$ once again to build the pseudo-projection database $\mathcal{D}_{PS_i}$ for each new prefix PS_i (line 11), adds PS_i into the result set (line 12), and calls itself to recursively get the frequent valid sequences (line 13). 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. To better understand our algorithm, we walk through our algorithm with a running example.

Example 4 Figure 10 shows how to enumerate the valid local items. Consider the three sequences in $\mathcal{D}$ as shown in Table 1. Suppose that the current frequent valid prefix sequence is a, the corresponding projected sequences of the three sequences 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 the original sequence is preserved. It is obvious that c is a valid local item for $caabd(as 5(a) > 4(c))$ and $cdcbab$ (as $5(a) > 4(c)$). However, c is not a valid item for $//eab$ as there is no $ac//e$ in the DTD. Hence, $RSupp(ac) = 2/3 > 1/2$, and thus it is a frequent valid sequence. Then, we check whether item a is a valid item for $aabd, dcbab$ w.r.t. ac, and there are two items w.r.t. a which are both valid items for $aabd$. There are two projected sequences abd and bd for $aabd$, however only one of them is frequent in this running example. Note that we can prune many branches using our valid local item based method. For example, we can prune nodes aa and ad from Fig. 8, as there are no valid sequences for the two nodes. Compared Fig. 8 with Fig. 10, our method prunes 5 nodes.

3.4 ESPRIT versus FastXMiner

Traditional frequent subtree mining approaches such as FastXMiner (Yang et al. 2003a) and 2PXMiner (Yang et al. 2004) generate a large number of candidate subtree structures and have to perform a lot of costly subtree containment testing. Instead, to reduce the costly subtree containment testing, we exploit the parent–child checking scheme to identify the frequent sequences, which checks the parent–child relationship in sequences by both forward and backward strategies. It is obvious that the

parent–child checking is much cheaper than the containment testing of tree structured data. That is the key why we exploit the frequent sequence mining to resolve the problem of frequent XQP mining.

We analyze the complexities of FastXMiner and ESPRIT. FastXMiner initially enumerates all the frequent 1-edge RSTs by scanning the database. In the subsequent passes, it generates the frequent $(k+1)$-edge RSTs from the frequent k-edge RSTs in two phases as follows (Yang et al. 2003a). In the first phase, it generates the candidate set C_{k+1} by using the previously found frequent set F_k by (1) extending an edge or (2) joining two frequent k-edge RSTs to generate $(k+1)$-edge RSTs. Any unqualified candidate RST is pruned. The frequency for each candidate RST is counted, and those RSTs that do not satisfy the minimal support criteria are pruned. The candidate set C_{k+1} contains all RSTs to be matched with the XQPs in the database. In the second phase, it refines C_{k+1} to get F_{k+1} by checking if RST_{k+1} in C_{k+1} is valid through tree-containment testing. This test is based on the extended tree inclusion, which has been proven to be NP-complete (Yang et al. 2003a). 2PXMiner proposes the transaction summary structure to reduce the number of costly tree inclusion tests beyond FastXMiner. However, it still joins frequent XQPs and checks tree-containment.

ESPRIT firstly extends a valid local item from the root, which is the same as enumerating 1-edge RSTs in FastXMiner. Note that we can use the DTD of an XML document to guide the enumeration (Yang et al. 2003a). Thus, the complexity is the number of elements in the DTD. Then, ESPRIT always extends valid local items based on the mined frequent sequences, which is the same as extending an edge to generate $(k+1)$-edge RSTs in FastXMiner. In addition, FastXMiner needs to join different RSTs. The complexity is $O(k*|T_{Rst}|^2)$, where T_{Rst} is the set of frequent RSTs. It is very expensive if there are large numbers of frequent XQPs.[3] Finally, ESPRIT checks whether the valid local item is frequent. We only need check whether the local item is valid in the sequences in the projected database based on Definition 10, and count the number. The complexity of checking whether a local item is valid in an xqp is $O(|\mathrm{xqp}|)$. Note that we need not test whether the sequence is valid subsequence based on Lemma 4 and Definition 7, which is similar to tree-containment testing and thus is very expensive. While FastXMiner needs to check whether the frequent RST in C_{k+1} is valid by tree-containment checking, which is NP-complete (Yang et al. 2003a). Accordingly, ESPRIT is more efficient than FastXMiner and 2PXMiner. We will experimentally prove that our method is more efficient and it is linear with the size of query patterns in Sect. 6.

4 Incremental mining of frequent XML query patterns

Generally, the transaction database is frequently updated as some queries are added to the database. Such updates may invalidate some existing frequent query patterns and generate new frequent query patterns. Although we can mine the new frequent query patterns on the updated database using ESPRIT from scratch, it is only efficient

[3] Even if they only join the RSTs with the same number of edges, in the worst case the complexity is still $O(k*|T_{Rst}|^2)$.

for the static transaction database but inefficient for the evolving transaction database as it cannot make use of the frequent sequences which have already been mined. To address this issue, we propose effective index structures and algorithms in this section.

4.1 F-index and Q/F-index

To facilitate the incremental mining of frequent valid subsequences, we introduce two novel indices in this section.

Suppose $\mathcal{F}(\mathcal{D})$, $\mathcal{F}(d)$, and $\mathcal{F}(\mathcal{D}^u)$ are the sets of frequent sequences of $\mathcal{D}$, d, and $\mathcal{D}^u$ respectively. The original database $\mathcal{D}$ has been already mined and $\mathcal{F}(\mathcal{D})$ has been gotten according to our algorithm ESPRIT. To get the frequent valid subsequences of the updated database, we first mine the incremental database d using our algorithm ESPRIT and get $\mathcal{F}(d)$, and then generate the set of up-to-date frequent sequences $\mathcal{F}(\mathcal{D}^u)$ by checking $\mathcal{F}(\mathcal{D})$ and $\mathcal{F}(d)$. We can classify the valid sequences (abbreviated as vseqs) in $\mathcal{F}(\mathcal{D})$ and $\mathcal{F}(d)$ into four categories:

1. $\mathcal{F}(\mathcal{D}) \cap \mathcal{F}(d)$ all of the vseqs that are frequent in both $\mathcal{D}$ and d.
2. $\mathcal{F}(\mathcal{D}) - \mathcal{F}(d)$ all of the vseqs that are frequent in $\mathcal{D}$ but infrequent in d.
3. $\mathcal{F}(d) - \mathcal{F}(\mathcal{D})$ all of the vseqs that are frequent in d but infrequent in $\mathcal{D}$.
4. *Others* all of the vseqs that are infrequent in both $\mathcal{D}$ and d.

Lemma 6 *vseqs in the first category must be frequent in the updated database $\mathcal{D}^u$. vseqs in the fourth category cannot be frequent in the updated database $\mathcal{D}^u$.*

Proof It is obvious and we omit the proof. □

Consider the sequences in the first categories or in the fourth categories, we can check whether they are frequent or not in $\mathcal{D}^u$ efficiently as formalized in Lemma 6. However, it is not straightforward to check whether the sequences in the second categories or the third categories are frequent. We need to check whether each sequence in $\mathcal{F}(\mathcal{D}) - \mathcal{F}(d)$ is still frequent in $\mathcal{D}^u$, and whether each sequence in $\mathcal{F}(d)$-$\mathcal{F}(\mathcal{D})$ is a new frequent sequence of $\mathcal{D}^u$.

For each sequence vseq in $\mathcal{F}(\mathcal{D})$-$\mathcal{F}(d)$, we first scan d to count the number of sequences that contain vseq ($ASupp^d(\text{vseq})$) using our algorithm ESPRIT, and then check whether $\frac{ASupp^{\mathcal{D}}(\text{vseq}) + ASupp^d(\text{vseq})}{|\mathcal{D}^u|} \geq min\_sup$ holds; if so, this sequence must be frequent; otherwise, this sequence is infrequent. Similarly, we can check whether each sequence in $\mathcal{F}(d)$-$\mathcal{F}(\mathcal{D})$ is frequent.

Although this approach is much more efficient than mining the updated database from scratch, it may involve some unnecessary scans on $\mathcal{D}$ and d. For example, consider the sequences in Table 1, abd is frequent in d and infrequent in $\mathcal{D}$. Although only $\text{xqp}_1^L(acaabd)$ contains abd, we have to scan all the sequences in $\mathcal{D}$ to count $ASupp^{\mathcal{D}}(abd)$. To address this issue and scan $\mathcal{D}$ and d as few as possible, we construct two novel indices for the original database and the incremental database to efficiently generate the up-to-date frequent valid subsequences of $\mathcal{D}^u$. For ease of presentation, we begin by introducing some notations.

Definition 11 *Direct Prefix Sequence*. Given a sequence $\mathcal{S} = e_1, e_2, \ldots, e_i, \mathcal{S}' = e_1, e_2, \ldots, e_{i-1}$ is called the direct prefix sequence of $\mathcal{S}$.

Definition 12 *Path Prefix Sequence*. Given a sequence $\mathcal{S} = e_1, e_2, \ldots, e_n, \mathcal{S}' = e_{i_1}, e_{i_2}, \ldots, e_{i_m}$ is a path prefix sequence of $\mathcal{S}$, where $1 \leq i_1 < i_2 < \cdots < i_m \leq n$, if $\mathcal{S}$ and $\mathcal{S}'$ satisfy,

1. $e_{i_1} = max(e_1, e_2, \ldots, e_n)$;
2. $\forall k, 1 \leq k < m, e_{i_k} > e_{i_{k+1}}$, and $\nexists j, i_k < j < i_{k+1}, e_j > e_{i_{k+1}}$.

If $\mathcal{S} \neq \mathcal{S}'$, $\mathcal{S}'$ is called a proper path prefix sequence of S.

A path prefix sequence $e_{i_1}, e_{i_2}, \ldots, e_{i_m}$ represents a path from e_{i_1} to e_{i_m} w.r.t its corresponding XQP, that is, e_{i_1} is the root of the XQP, and $\forall k, 1 \leq k < m, e_{i_k}$ is the parent of $e_{i_{k+1}}$. For example, consider the sequence $\mathtt{xqp}_1 = acaabd$ in Table 1, its direct prefix sequence is $acaab$ and its path prefix sequence is abd. abd is a proper path prefix sequence of $acaabd$.

Definition 13 *Quasi Frequent Sequence*. A sequence $\mathcal{S}$ is a frequent sequence if $RSupp(\mathcal{S}) \geq min\_sup$; A sequence $\mathcal{S}$ is a quasi frequent sequence if its direct prefix sequence is frequent. The quasi frequent sequence that is not frequent is called a quasi-frequent/frequent sequence (abbreviated as Q/F sequence).

For example, consider $\mathtt{xqp}_1^L = acaabd$. $acaab$ is the direct prefix sequence of $\mathtt{xqp}_1^L$. abd is the proper path prefix sequence of $\mathtt{xqp}_1^L$. $acab$ is a quasi-frequent sequence w.r.t $\mathcal{D}$ and it is also a frequent sequence. $acabd$ is a Q/F sequence w.r.t $\mathcal{D}$.

We note that if a sequence S is not a quasi-frequent sequence, it cannot be a frequent sequence, which complies with the apriori property. Accordingly, we can employ the apriori property to skip the non-frequent sequences. That is, if a sequence is not a quasi-frequent sequence, we need not scan the projected database to count its support as we can assure that it is not a frequent sequence.

To facilitate the identifying of frequent valid subsequences, we construct $F\text{-}index$ and $Q/F\text{-}index$ for the original database and the incremental database, which can improve the performance of mining frequent valid subsequences. $F\text{-}index$ maintains each frequent sequence and $Q/F\text{-}index$ maintains each Q/F sequence of the database. Sequences in F-index or $Q/F\text{-}index$ (called $F\&Q/F\text{-}index$) are sorted in lexicographical order and each sequence maintains its absolute support and an $IDList$, which records a set of tuples $(Sid, Pointer)$, where Sid is the identifier of its supersequences and $Pointer$ is used to record its corresponding projected sequence in the database. Given a certain frequent sequence or Q/F sequence, its supersequences and projected sequence can be gotten according to the $IDList$ efficiently.

The salient feature of $F\&Q/F\text{-}index$ is that, for any sequence in $F\&Q/F\text{-}index$, we have to scan the database to check whether it is frequent during mining the frequent valid sequences, and thus we can preserve them to facilitate the mining of up-to-date frequent sequences. During mining the frequent valid subsequences, once ESPRIT finds that a valid sequence vseq is frequent or quasi-frequent, ESPRIT records the $IDList$ and $ASupp$ of this vseq and inserts them into the corresponding $F\&Q/F\text{-}index$, which is similar to record a table in dynamic programming. To better understand $F\&Q/F\text{-}index$, we give a running example.

Table 2 F&Q/F-index

(a) $F\text{-}index$ of $\mathcal{D}$

F-Sequence	ab	ac	aca^1	$acab$
$ASupp$	3	2	2	2
$IDLists$	1, 6|2, nil|3, nil	1, 3|3, 3	1, 5|3, 7	1, 6|3, nil

(b) $Q/F\text{-}index$ of $\mathcal{D}$

Q/F-Sequence	abd	aca^2	$acabd$	acb	acd	$a//$
$ASupp$	1	1	1	1	1	1
$IDLists$	1,nil	1,4	1,nil	3,6	3,4	2,3

(c) $F\text{-}index$ of d

F-Sequence	ab	abd	ac	af	$a//$	$a//e$	$a//ea$	$a//eaf$
$ASupp$	2	2	2	2	2	2	2	2
$IDLists$	4,5|,6,6	4,nil|,6,nil	4,3|6,6	5,nil|6,6	5,3|6,3	5,4|6,4	5,5|6,5	5,nil|6,6

(d) $Q/F\text{-}index$ of d

Q/F-Sequence	aca	afd	$a*$	$a//eafd$
$ASupp$	1	1	1	1
$IDLists$	4,4	6,nil	6,6	6,nil

Example 5 Table 2a–d illustrate the $F\&Q/F\text{-}index$ of the original database $\mathcal{D}$ and the incremental database d respectively. In database d, abd is a valid subsequence of $\mathtt{xqp}_4 = acabd$, and abd is also a valid subsequence of $\mathtt{xqp}_6 = a//ea*d$, as $*$ can match any label. Thus, $ASupp(abd) = 2$ and abd is a frequent sequence in the incremental database d. $ASupp$ of $a//e$ in the $F\text{-}index$ of the incremental database d, denoted as $F^d\text{-}index$, is 2, which denotes that there are two sequences that contain $a//e$ in the incremental database d. The IDList of $a//e$ is (5,4) and (6,4), which means that its supersequences are $\mathtt{xqp}_5$ and $\mathtt{xqp}_6$, and the two corresponding projected sequence are, the subsequence of $\mathtt{xqp}_5$ obtained from the 4-th item to the last item and the subsequence of $\mathtt{xqp}_6$ obtained from the 4-th item to the last item.

4.2 Incremental sequential pattern mining for frequent XQP mining

In this section, we propose an efficient incremental algorithm, ESPRIT-i. The purpose of incremental mining is to discover the set of all the frequent vseqs of the updated database with minimal re-computation. Without loss of generality, suppose the original valid frequent sequences have been already gotten and $F\&Q/F\text{-}index$ has been built.

We first mine the incremental database d using ESPRIT, and then obtain the frequent vseqs of the updated database $\mathcal{D}^u$ through merging the original mined results of $\mathcal{D}$, i.e., $\mathcal{F}(\mathcal{D})$, and the mined results of d, i.e., $\mathcal{F}(d)$. Since the sequences in $\mathcal{F}(\mathcal{D})$ or $\mathcal{F}(d)$ are sorted in lexicographical order, $\mathcal{F}(\mathcal{D}) \cap \mathcal{F}(d)$ can be gotten through merge-joining $\mathcal{F}(\mathcal{D})$ and $\mathcal{F}(d)$, the complexity of which is $O(|\mathcal{F}(\mathcal{D})| + |\mathcal{F}(d)|)$. In addition, we need to check whether vseqs in $\mathcal{F}(\mathcal{D}) - \mathcal{F}(d) = \mathcal{F}(\mathcal{D}) - \mathcal{F}(\mathcal{D}) \cap \mathcal{F}(d)$ and

$\mathcal{F}(d) - \mathcal{F}(\mathcal{D}) = \mathcal{F}(d) - \mathcal{F}(\mathcal{D}) \cap \mathcal{F}(d)$ are frequent in $\mathcal{D}^u$. To mine the frequent sequence efficiently, we sort vseqs in $\mathcal{F}(\mathcal{D}) - \mathcal{F}(d)$ and $\mathcal{F}(d) - \mathcal{F}(\mathcal{D})$ by |vseq| in ascending order. Without loss of generality, we only introduce how to determine whether the sequences in $\mathcal{F}(\mathcal{D}) - \mathcal{F}(d)$ are frequent. The same method can be applied to process the sequences in $\mathcal{F}(d) - \mathcal{F}(\mathcal{D})$.

$\forall$vseq $\in (\mathcal{F}(\mathcal{D}) - \mathcal{F}(d))$, vseq cannot be frequent in the incremental databse d. Suppose DPvseq and PPvseq are the direct prefix sequence and the proper path prefix sequence of vseq. As |DPvseq| < |vseq| and |PPvseq| < |vseq|, whether DPvseq or PPvseq are frequent in $\mathcal{D}^u$ has been processed as we mine the sequences in ascending order by the number of items in the sequence. If one of PPvseq (if any) and DPvseq is infrequent, it is obvious that vseq is infrequent based on the a priori property. Otherwise, we need to scan the $Q/F\text{-}index$ of the incremental database d, i.e., $Q/F^d\text{-}index$, or the $F\text{-}index$ of the updated database $\mathcal{D}^u$, i.e., $F^{\mathcal{D}^u}\text{-}index$, to check whether vseq is frequent as follows.

1. If vseq is in $Q/F^d\text{-}index$, we can get $ASupp^{\mathcal{D}^u}$(vseq) and $\text{vseq}^{\mathcal{D}^u}.IDList$ according to $Q/F^d\text{-}index$. We have,

$$ASupp^{\mathcal{D}^u}(\text{vseq}) = ASupp^{\mathcal{F}(\mathcal{D})}(\text{vseq}) + ASupp^{Q/F^d\text{-}index}(\text{vseq}) \quad (1)$$

$$\text{vseq}^{\mathcal{D}^u}.IDList = \text{vseq}^{\mathcal{F}(\mathcal{D})}.IDList \cup \text{vseq}^{Q/F^d\text{-}index}.IDList \quad (2)$$

   As the sequences in $Q/F{-}index$ are sorted in lexicographical order, it is easy to get $ASupp^{Q/F^d\text{-}index}$(vseq) and $\text{vseq}^{Q/F^d\text{-}index}.IDList$. Moreover, $ASupp^{\mathcal{F}(\mathcal{D})}$(vseq) a nd $\text{vseq}^{\mathcal{F}(\mathcal{D})}.IDList$ can be easily gotten according to the $F\text{-}index$ of $\mathcal{D}$.
2. Otherwise, vseq is not in $Q/F^d\text{-}index$. Suppose DPvseq is the direct prefix sequence of vseq, where vseq = DPvseq $\bullet\, e$. DPvseq must be frequent in $\mathcal{D}^u$ and thus it is in $F^{\mathcal{D}^u}\text{-}index$. We scan each projected sequence $PS_$DPvseq of DPvseq based on $\text{DPvseq}^{F^{\mathcal{D}^u}}\text{-}index.IDList$ obtained from $F^{\mathcal{D}^u}\text{-}index$, check whether the item e is a valid item of $PS_$DPvseq w.r.t. DPvseq, and count the number of DPvseq that contain the valid item e, i.e., $ASupp^d$(DPvseq). Finally, we record $\text{DPveq}^d.IDList$. Accordingly, we can get $ASupp^{\mathcal{D}^u}$(vseq) and $\text{vseq}^{\mathcal{D}^u}.IDList$ as follows.

$$ASupp^{\mathcal{D}^u}(\text{vseq}) = ASupp^{d}(\text{vseq}) + ASupp^{\mathcal{F}(\mathcal{D})}(\text{vseq}) \quad (3)$$

$$\text{vseq}^{\mathcal{D}^u}.IDList = \text{vseq}^{d}.IDList \cup \text{vseq}^{\mathcal{F}(\mathcal{D})}.IDList \quad (4)$$

Based on above observations, we devise an efficient algorithm ESPRIT-i to incrementally mine the frequent valid subsequences (Fig. 11). ESPRIT-i first mines frequent valid subsequences of the incremental database d by calling our algorithm ESPRIT (line 2), gets $\mathcal{F}(\mathcal{D}) \cap \mathcal{F}(d)$ (line 4) and adds the merged results to $\mathcal{F}(\mathcal{D}^u)$ (line 5). Finally, ESPRIT-i calls its subroutine `getFreqVSeqs` to get the frequent valid subsequences in $\mathcal{F}(\mathcal{D}) - \mathcal{F}(d)$ and $\mathcal{F}(d) - \mathcal{F}(\mathcal{D})$ (lines6–7).

`getFreqVSeqs` identifies the frequent valid sequences in $\mathcal{F}(\mathcal{D}) - \mathcal{F}(d)$ or $\mathcal{F}(d) - \mathcal{F}(\mathcal{D})$. For each sequence vseq in $\mathcal{F} = \mathcal{F}(\mathcal{D}) - \mathcal{F}(d)$ (or $\mathcal{F}(d) - \mathcal{F}(\mathcal{D})$),

Algorithm 2: ESPRIT i-Algorithm

Input: $\mathcal{D}$: the original database;
$\mathcal{F}(\mathcal{D})$: the frequent vseq set of $\mathcal{D}$;
$F\&Q/F^{\mathcal{D}}$-$index$: the $F\&Q/F$-$index$ of $\mathcal{D}$;
d: the incremental database;
min_sup: a given minimum support.
Output: $\mathcal{F}(\mathcal{D}^u)$: the complete set of frequent valid subsequences in $\mathcal{D}^u$.

```
1 begin
2   F(d) = ESPRIT(d, min_sup);
3   F(D^u) = φ;
4   F(D) ∩ F(d) = MergeJoin(F(D), F(d));
5   F(D^u) ∪= F(D) ∩ F(d);
6   F(D^u) ∪= getFreqVSeqs(F(D)-F(d), d, min_sup, F&Q/F^D-index, F&Q/F^d-index);
7   F(D^u) ∪= getFreqVSeqs(F(d)-F(D), D, min_sup, F&Q/F^d-index, F&Q/F^D-index);
8 end
```

Function `getFreqVSeqs` Function

Input: $\mathcal{F}$: a set of frequent valid sequences;
db: a database, $\mathcal{D}$ or d;
min_sup: a given minimum support;
$F\&Q/F^{\mathcal{D}} - index$: the $F\&Q/F - index$ of $\mathcal{D}$;
$F\&Q/F^{d} - index$: the $F\&Q/F - index$ of d.
Output: $\mathcal{F}(\mathcal{D}^u)$: a set of frequent valid subsequences in $\mathcal{D}^u$.

```
1  begin
2    F(D^u) ← φ;
3    for each vseq ∈ F do
4      DPvseq ← the direct prefix sequence of vseq and vseq = DPvseq • e_i;
5      PPvseq ← the path prefix sequence of vseq;
6      if DPvseq and PPvseq are both in F^{D^u}-index then
7        if vseq is in Q/F^db-index then
8          ASupp^{D^u}(vseq) = ASupp^{Q/F^db - index}(vseq) + ASupp^F(vseq);
9        else
10         ASupp^db(vseq) ← 0;
11         for each PS_DPvseq in DPvseq^db.IDList do
12           if e_i is a valid item of PS_DPvseq w.r.t. DPvseq then
13             ASupp^db(vseq)++;
14         ASupp^{D^u}(vseq) = ASupp^db(vseq) + ASupp^F(vseq);
15       if vseq is frequent then
16         F(D^u) ← vseq;
17     if vseq is frequent or quasi-frequent in D^u then
18       buildF&Q/F-Index(vseq);
19   return F(D^u);
20 end
```

Procedure buildF&Q/F-Index Procedure

Input: vseq: a valid sequence.

```
1 begin
2   Insert vseq into the F&Q/F^{D^u}-Index;
3 end
```

Fig. 11 ESPRIT-i algorithm

`getFreqVSeqs` checks whether the path prefix sequence (DPvseq) and the direct prefix sequence (PPvseq) of vseq are in $\mathcal{F}^{\mathcal{D}^u}$-$index$ (line 14). If so, vseq may be frequent in $\mathcal{D}^u$; otherwise, it cannot be. Then, `getFreqVSeqs` checks whether vseq is in Q/F^{db}-$index$. If so, it computes the absolute support of vseq according to the corresponding Q/F-$index$ (line 16); otherwise, scans the $F\&Q/F$-$index$ of $\mathcal{D}^u$, which has been already constructed, to count the absolute support (line 21). If vseq is frequent, `getFreqVSeqs` adds it to the result (line 24). Finally, if vseq is frequent or quasi-frequent, `getFreqVSeqs` inserts vseq into the $F\&Q/F-index$ of $\mathcal{D}^u$ by calling buildF&Q/F-Index (line 26). To better understand our algorithm, we walk through the algorithm with a running example.

Example 6 Consider the $F\&Q/F$-$index$ of $\mathcal{D}$ and d in Table 2. As $\mathcal{F}(\mathcal{D}) = \{ab; ac; aca; acab\}$ and $\mathcal{F}(d) = \{ab; abd; ac; af; a//; a//e; a//ea; a//eaf\}$, we have $\mathcal{F}(\mathcal{D}) \cap \mathcal{F}(d) = \{ab; ac\}$, $\mathcal{F}(\mathcal{D}) - \mathcal{F}(d) = \{aca; acab\}$ and $\mathcal{F}(d) - \mathcal{F}(\mathcal{D}) = \{abd; af; a//; a//e; a//ea; a//eaf\}$. According to Lemma 6, all the sequences in $\mathcal{F}(\mathcal{D}) \cap \mathcal{F}(d)$ must be frequent in $\mathcal{D}^u$. We only need to check whether the sequences in $\mathcal{F}(\mathcal{D}) - \mathcal{F}(d)$ and $\mathcal{F}(d) - \mathcal{F}(\mathcal{D})$ are frequent. Consider vseq=abd in $\mathcal{F}(d) - \mathcal{F}(\mathcal{D})$, we have DPvseq=$ab$, which is the direct prefix sequence of vseq. DPvseq is frequent in $\mathcal{D}^u$ and abd is quasi-frequent in $\mathcal{D}^u$. As abd is in Q/F-$index$ of $\mathcal{D}$, we have

$$\begin{aligned} ASupp^{\mathcal{D}^u}(\mathsf{vseq}) &= ASupp^{d}(\mathsf{vseq}) + ASupp^{Q/F^{\mathcal{D}}\text{-}index}(\mathsf{vseq}) = 3; \\ \mathsf{vseq}^{\mathcal{D}^u}.IDList &= \mathsf{vseq}^{d}.IDList \cup \mathsf{vseq}^{Q/F\text{-}index^{\mathcal{D}}.IDList} \\ &= \{(1, \text{nil}), (4, \text{nil}), (6, \text{nil})\}. \end{aligned}$$

Thus, abd is frequent in $\mathcal{D}^u$. Similarly, $aca, acab, a//, a//e, a//ea$ are also frequent in $\mathcal{D}^u$, and $\mathcal{F}(\mathcal{D}^u) = (\mathcal{F}(\mathcal{D}) \cap \mathcal{F}(d)) \cup \{aca; acab\} \cup \{abd; a//; a//e; a//ea\} = \{ab; abd; ac; aca; acab; a//; a//e; a//ea\}$. The F-$index$ of $\mathcal{D}^u$ is illustrated in Table 3.

4.3 Interactively mining frequent sequences

ESPRIT-i can avoid unnecessary scans of the original database and the incremental database. However, it mines the incremental database independently and does not interact with the original database fully. It may involve additional scans on the incremental database. For example, consider the sequences in Table 1, $a//eaf$ is frequent

Table 3 F-$index$ of $\mathcal{D}^u$

F-Sequence	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-Sequence	$acab$	$a//$	$a//e$	$a//ea$
$ASupp$	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

in the incremental database d, and the incremental database d has to be scanned to check whether $a//eafd$ is frequent when mining database d. However, $a//eaf$ is not a frequent sequence of the updated database and thus $a//eafd$ cannot be frequent in $\mathcal{D}^u$ even if we do not scan the incremental database according to the a priori property.

To address this issue, ESPRIT-i^+ is proposed to mine the incremental database by interacting with the original database, which can improve ESPRIT-i effectively as it avoids scanning the unnecessary items in the original databases. The idea behind ESPRIT-i^+ is that, during mining the incremental database d, we can make full use of the mining results of the original database to discover frequent valid subsequences of the updated database.

ESPRIT-i^+ enumerates the set of frequent valid subsequences by extending valid local items, which is similar to the pseudo-projection-based ESPRIT. When enumerating a valid local item e w.r.t. a frequent valid prefix sequence DPvseq, we first compute the absolute support of vseq=DPvseq $\bullet\, e$ in the incremental database $d(ASupp^d(\mathsf{vseq}))$ by counting the number of projected sequences w.r.t. DPvseq that have a valid local item e, and then compute $ASupp^{\mathcal{D}^u}(\mathsf{vseq})$ by considering the following two cases.

1. $RSupp^d(\mathsf{vseq}) \leq min\_sup$.
   In this case, the necessary condition for vseq to be frequent in $\mathcal{D}^u$ is that vseq is frequent in $\mathcal{D}$. That is, if vseq is not in $F\text{-}index$ of $\mathcal{D}$, it cannot be frequent in $\mathcal{D}^u$. Thus, we compute $ASupp^{\mathcal{D}^u}(\mathsf{vseq})$ according to the $F\text{-}index$ of $\mathcal{D}$,

$$ASupp^{\mathcal{D}^u}(\mathsf{vseq}) = ASupp^d(\mathsf{vseq}) + ASupp^{F^{\mathcal{D}}\text{-}index}(\mathsf{vseq}). \quad (5)$$

$$\mathsf{vseq}^{\mathcal{D}^u}.IDList = \mathsf{vseq}^d.IDList \cup \mathsf{vseq}^{F^{\mathcal{D}}\text{-}index}.IDList. \quad (6)$$

   $ASupp^d(\mathsf{vseq})$ and $\mathsf{vseq}^d.IDList$ can be gotten through extending valid local items.
2. $RSupp^d(\mathsf{vseq}) > min\_sup$.
   (i) vseq is frequent in $\mathcal{D}$, that is, it is in $F\text{-}index$ of $\mathcal{D}$. vseq must be frequent in $\mathcal{D}^u$ of this case. Thus we compute $ASupp^{\mathcal{D}^u}(\mathsf{vseq})$ based on the $F\text{-}index$ of $\mathcal{D}$,

$$ASupp^{\mathcal{D}^u}(\mathsf{vseq}) = ASupp^d(\mathsf{vseq}) + ASupp^{F^{\mathcal{D}}\text{-}index}(\mathsf{vseq}). \quad (7)$$

$$\mathsf{vseq}^{\mathcal{D}^u}.IDList = \mathsf{vseq}^d.IDList \cup \mathsf{vseq}^{F^{\mathcal{D}}\text{-}index}.IDList. \quad (8)$$

   (ii) vseq is quasi-frequent but infrequent in $\mathcal{D}$, that is, it is in $Q/F\text{-}index$ of $\mathcal{D}$. We need to compute $ASupp^{\mathcal{D}^u}(\mathsf{vseq})$ to determine whether vseq is frequent in $\mathcal{D}^u$ according to $Q/F\text{-}index$ of $\mathcal{D}$,

$$ASupp^{\mathcal{D}^u}(\mathsf{vseq}) = ASupp^d(\mathsf{vseq}) + ASupp^{Q/F^{\mathcal{D}}\text{-}index}(\mathsf{vseq}). \quad (9)$$

$$\mathsf{vseq}^{\mathcal{D}^u}.IDList = \mathsf{vseq}^d.IDList \cup \mathsf{vseq}^{Q/F^{\mathcal{D}}\text{-}index}.IDList. \quad (10)$$

   (iii) vseq is not quasi-frequent in $\mathcal{D}$, that is, it is not in $F\&Q/F\text{-}index$ of $\mathcal{D}$. As we have inserted DPvseq into the $F\&Q/F\text{-}index$ of $\mathcal{D}^u$, we scan the pro-

jected sequences w.r.t. DPvseq in $\mathcal{D}^u$ to count $ASupp^{\mathcal{D}}(\text{vseq})$ and record $\text{vseq}^{\mathcal{D}}.IDList$ according to $\text{DPvseq}^{\mathcal{D}}.IDList$ based on $F\text{-}index$ of $\mathcal{D}^u$. Thus, we compute $ASupp^{\mathcal{D}^u}(\text{vseq})$ according to the the entry of DPvseq in the $F\text{-}index$ of $\mathcal{D}^u$,

$$ASupp^{\mathcal{D}^u}(\text{vseq}) = ASupp^{d}(\text{vseq}) + ASupp^{\mathcal{D}}(vseq); \quad (11)$$

$$\text{vseq}^{\mathcal{D}^u}.IDList = \text{vseq}^{d}.IDList \cup \text{vseq}^{\mathcal{D}}.IDList. \quad (12)$$

Based on $ASupp^{\mathcal{D}^u}(\text{vseq})$, we can check whether vseq is frequent in $\mathcal{D}^u$. Accordingly, we devise a novel algorithm ESPRIT-i^+ to incrementally mine the frequent valid sequences of $\mathcal{D}^u$ by interacting with the original database as illustrated in Fig. 12. ESPRIT-i^+ first calls its subroutine getFreqVseqsInteract to identify the frequent valid sequences (line 3), and then validates whether the frequent sequences in $\mathcal{D}$ that do not appear in d are still frequent in $\mathcal{D}^u$ (line 6). Finally, ESPRIT-i^+ builds the $F\&Q/F\text{-}index$ (line 7).

getFreqVseqsInteract iteratively identifies the frequent valid sequences by calling itself. For each prefix frequent valid sequence, PS, if PS is non-empty, getFreqVseqsInteract adds into the result set $\mathcal{F}(\mathcal{D}^u)$ (line 12). Then, getFreqVseqsInteract computes the set of valid local items w.r.t. PS (line 13). For each such item, e_i, getFreqVseqsInteract calls algorithm getUpdatedFrequentValidLocalItem (Fig. 13) to check whether the valid item e_i is frequent w.r.t. PS. If e_i is a frequent valid item, getFreqVseqsInteract generates a new frequent valid sequence PS_i (line 16), gets the pseudo projected sequence w.r.t. PS_i (line 17), adds PS_i in to the result set $\mathcal{F}(\mathcal{D}^u)$ (line 18), inserts PS_i into the $F\&Q/F\text{-}index$ of $\mathcal{D}^u$ (line 19), and calls itself to find frequent valid sequences (line 20).

getUpdatedFrequentValidLocalItem is used to check whether a given valid item is frequent in the updated database $\mathcal{D}^u$ as shown in Fig. 13. getUpdatedFrequentValidLocalItem first extends DPvseq by adding item e_i to generate a new sequence vseq and scans the incremental database d to count the number of occurrences that vseq appears in d (line 3). If the relative support of vseq is no larger than min_sup, getUpdatedFrequentValidLocalItem only needs to scan the $F\text{-}index$ of $\mathcal{D}$ and then computes the absolute support of vseq by adding $ASupp^{F^{\mathcal{D}}\text{-}index}(\text{vseq})$ and $ASupp^{d}(\text{vseq})$ (line 6); otherwise, getUpdatedFrequentValidLocalItem checks whether vseq is frequent in three aspects (lines 8–17). If vseq is in $F\text{-}index$ of $\mathcal{D}$, getUpdatedFrequentValidLocalItem computes the absolute of vseq by summing up $ASupp^{F^{\mathcal{D}}\text{-}index}(\text{vseq})$ and $ASupp^{d}(\text{vseq})$ (line 9); if vseq is in $Q/F\text{-}index$ of $\mathcal{D}$, getUpdatedFrequentValidLocalItem computes the absolute of vseq by summing up $ASupp^{Q/F^{\mathcal{D}}\text{-}index}(\text{vseq})$ and $ASupp^{d}(\text{vseq})$ (line 11); otherwise scans the sequences in $\text{DPvseq}^{\mathcal{D}}.IDList$ according to $\mathcal{F}\text{-}index$ of $\mathcal{D}^u$ (line 16) and computes the absolute support of vseq (line 17). Thus, based on the absolute support of vseq, if vseq is frequent, getUpdatedFrequentValidLocalItem returns true (line 19); otherwise returns false (line 21).

The Border algorithm (Aumann et al. 1999) is similar to our ESPRIT-i. ESPRIT-i extends a *valid* item to generate valid subsequence while the Border algorithm extends an item to generate frequent subsequences. Moreover, ESPRIT-i need consider the

Algorithm 3: ESPRIT i^+ Algorithm

Input: $\mathcal{D}$: the original database;
$\mathcal{F}(\mathcal{D})$: the frequent vseq set of $\mathcal{D}$;
$F\&Q/F^{\mathcal{D}}$-$index$: the $F\&Q/F$-$index$ of $\mathcal{D}$;
d: the incremental database;
min_sup: a given minimum support.
Output: $\mathcal{F}(\mathcal{D}^u)$: the complete set of frequent valid sequences in $\mathcal{D}^u$

```
1 begin
2   F(D^u) = φ;
3   F(D^u) ∪= getFreqVseqsInteract(φ, d, min_sup, D, F(D), F&Q/F^D-index);
4   for each vseq in F(D) do
5     if vseq is not in F(D^u) AND ASupp^{D^u}(vseq) ≥ min_sup then
6       F(D^u) ← vseq;
7       buildF&Q/F-Index(vseq);
8   return F(D^u);
9 end
```

Function `getFreqVseqsInteract` Function

Input: PS: a prefix sequence;
d_{PS}: the projected database w.r.t. prefix PS of database d;
min_sup: a given minimum support;
$\mathcal{D}$: the original database;
$\mathcal{F}(\mathcal{D})$: the set of frequent sequence w.r.t. $\mathcal{D}$;
$F\&Q/F^{\mathcal{D}}$-$index$: the $F\&Q/F$-$index$ of $\mathcal{D}$.
Output: $\mathcal{F}(\mathcal{D}^u)$: a set of frequent valid subsequences in $\mathcal{D}^u$

```
1  begin
2    if PS is non-empty then
3      F(D^u) ← PS;
4    VLF_PS = getFrequentValidLocalItems(PS, d_PS, min_sup);   /*get the frequent
     valid local items w.r.t. PS in d_PS according to DEFINITION 10.*/
5    for each frequent valid local item e_i ∈ VLF_PS do
6      if getUpdatedFrequentValidLocalItem(PS, e_i, min_sup, D, F&Q/F^D-index) then
7        PS_i = PS•e_i;  /*concatenate e_i to PS*/
8        d_{PS_i} = getPseudoProjectedDatabase(PS_i, d_PS);
9        F(D^u) ← PS_i;
10       buildF&Q/F-Index(PS_i);
11       F(D^u) ∪= getFreqVseqsInteract(PS_i, d_{PS_i}, min_sup, D, F(D), F&Q/F^D-index);
12   return F(D^u);
13 end
```

Fig. 12 ESPRIT-i^+ algorithm

case of '//' and '*'. ESPRIT-i^+ efficiently mines the incremental database by interacting with the original database, which can improve ESPRIT-i effectively as it avoids scanning the unnecessary items in the original databases.

5 Cache lookup and replacement

This section proposes several novel techniques of query rewriting, cache lookup and cache replacement to enhance the answerability of caching.

Algorithm 4: getUpdatedFrequentValidLocalItem Algorithm

Input: DPvseq: a direct prefix sequence;
e_i: a local item;
min_sup: a given minimum support;
$\mathcal{D}$: the original database;
$F\&Q/F^{\mathcal{D}}$-$index$: the $F\&Q/F$-$index$ of $\mathcal{D}$;
Output: if e_i is a valid frequent item return **true**; otherwise return **false**.

```
1  begin
2    vseq=DPvseq•e_i;
3    ASupp^d(vseq)=countASupp(vseq,d);//get the absolute support of vseq in d.
4    if RSupp^d(vseq)≤min_sup then
5      if vseq is in F-index of D then
6        ASupp^{D^u}(vseq) = ASupp^{F^D-index}(vseq) + ASupp^d(vseq);
7    else
8      if vseq is in F-index of D then
9        ASupp^{D^u}(vseq) = ASupp^{F^D-index}(vseq) + ASupp^d(vseq);
10     else if vseq is in Q/F-index of D then
11       ASupp^{D^u}(vseq) = ASupp^{Q/F^D-index}(vseq) + ASupp^d(vseq);
12     else
13       ASupp^D(vseq)←0;
14       for each PS_vseq in DPvseq^D.IDList do
15         if e_i is a local frequent valid item w.r.t. DPvseq then
16           ASupp^D(vseq)++;
17       ASupp^{D^u}(vseq) = ASupp^D(vseq) + ASupp^d(vseq);
18   if vseq is frequent in D^u(RSupp^{D^u}(vseq)≥min_sup) then
19     return true;
20   else
21     return false;
22 end
```

Fig. 13 getUpdatedFrequentValidLocalItem algorithm

5.1 Query rewriting

Frequent XQPs capture the frequent queries issued in the past and they form the ideal candidates for caching. However, for a certain frequent XQP, it is a challenge to select which nodes and their answers for caching. Note that, the more nodes selected to cache, the higher hit rate is, but the more storage space is needed to cache them. Selecting more nodes of a certain frequent query or more frequent queries to cache is an alternative with limited memory.

In addition, selecting which node for caching will influence the performance of the underlying XML-DBMS. For example, in Fig. 14, consider that $CXQP^{i}$ and $CXQP^{ii}$ are cached XQPs , where the dashed nodes of $CXQP^{i}$ and $CXQP^{ii}$ (and their corresponding answers) are selected to cache. Although a new query $NXQP$ is issued with a returned node b, which is contained in $CXQP^{i}$ and $CXQP^{ii}$(if we do not consider the returned node), it cannot be answered by their answers (Mandhani and Suciu 2005). As the cached results do not contain the results of b. However, we can integrate $CXQP^{i}$ and $CXQP^{ii}$ into $CXQP^{iii}$ and cache $CXQP^{iii}$ and its

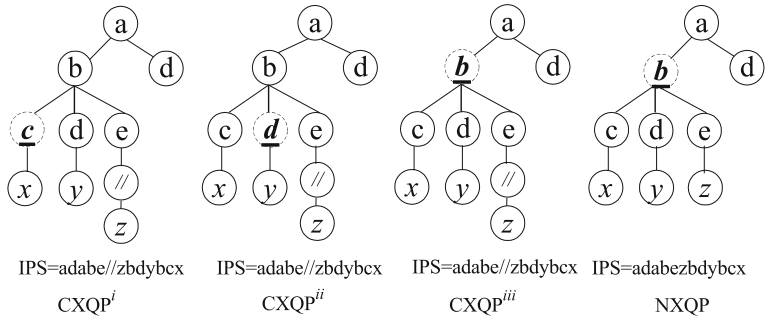


Fig. 14 Three cached XQPs: $CXQP^i$, $CXQP^{ii}$, $CXQP^{iii}$ and a new XQP, $NXQP$

corresponding answers. Accordingly, only $CXQP^{iii}$ and the answers of the returned node b w.r.t. $CXQP^{iii}$ are cached. Note that, we can utilize $CXQP^{iii}$ to answer $CXQP^i$ and $CXQP^{ii}$ as follows. We first retrieve the elements w.r.t. c or d, and then check whether the elements w.r.t. c or d are children of the cached elements for b. Finally, we return the elements which are children of the cached elements for b. This strategy not only saves the storage space, but also can improve the answerability so as to answer many more queries, e.g., $CXQP^i$, $CXQP^{ii}$ and $NXQP$. Accordingly, we integrate the equivalent queries to enhance the answerability of caching in this paper. However, it is not easy to integrate multiple random XQPs, which is beyond the scope of this paper. Interested readers are referred to our prior works (Li et al. 2006a) for more details.

5.2 Cache lookup

It is important to efficiently find a suitable XQP in the cache to answer a new query. A cached query C_v and a query C_q having the same $ILPS$ is not a necessary condition for C_v to answer C_q, but only a sufficient condition. This paper gives a weak sufficient condition for C_v to answer C_q, which can improve the cache hit rate without involving a lot of computations. We begin by introducing two concepts.

Definition 14 *Query-Axis*. Given an XQP, its query axis is the path from the root node to the returned node. Nodes on this path are called "*axis nodes*", while the others are called "*predicate nodes*".

Definition 15 $Preds(\mathtt{xqp}, k)$. Given an XQP $\mathtt{xqp}$, $Preds(\mathtt{xqp}, k)$ denotes the k-th axis node with its predicates. $\mathtt{xqp}_k$ is the path from the k-th axis node to the last axis node of $\mathtt{xqp}$ including the corresponding predicates. Query-Axis sequence is the corresponding sequence of Query-Axis.

We can obtain Query-Axis sequence of an XQP through its XQP^L, which is the path prefix sequence (as described in Definition 12) of the subsequence of XQP^L that is obtained from the first item to the first position where the returned node of the XQP appears in the sequence. We define $Preds(\mathtt{xqp}, k)$ sequences and $\mathtt{xqp}_k$ sequences, which can also be obtained from $\mathtt{xqp}^L$. In our approach, given a cached

query, we cache the axis nodes and the answers of the axis nodes which satisfy the query. This strategy can improve the answerability of caching, but does not involve too much storage space for preserving the answers. To better understand our method, we give a running example.

Example 7 Consider the XQPs in Fig. 14, the query axis of $CXQP^{i}$ is $a/b/c$. The query axis nodes of $CXQP^{ii}$ are a, b, and d. We cache the answers of the axis nodes. The query-axis sequence of $CXQP^{i}$ is abd. $Preds(CXQP^{ii}, 1) = a[d].Preds(CXQP^{ii}, 2) = b[c/x][e//z].Preds(CXQP^{ii}, 3) = d[y].CXQP_2^{ii} = b[c/x][e//z]/d[y]$.

Based on the concepts of Query-Axis and $Preds(\mathtt{xqp}, k)$, we introduce a weak sufficient condition of C_v to answer C_q.

Lemma 7 *A cached query V can answer a new query Q, denoted as $V \rightarrow Q$, if*

1. *Query-Axis sequence of V is a prefix of that of Q; and*
2. *$\forall k, 1 \leq k \leq m$, $Preds(Q, k)$ sequence is properly included in that of $Preds(V, k)$, where m is the total number of the axis nodes in V.*

Proof As the Query-Axis sequence of V is a prefix of that of Q and each $Preds(Q, k)$ is properly included in $Preds(V, k)$, V must be contained in Q. In addition, all the results of axis-nods of V are cached, therefore we can answer Q using V. □

Given a new query Q, to determine whether a cached view V can answer Q, we first check whether (1) of Lemma 7 is true, the complexity of which is the number of nodes in the query axis of V. If (1) is true, we check whether (2) is true. The check of whether $Preds(Q, i)$ is properly included in $Preds(V, i)$ is much easier than the check of whether Q^L is properly included in V^L. If both (1) and (2) are true, we can reconstruct V to answer Q as follows.

As the Query-Axis sequence of V is a prefix of that of Q, any axis node in V must be in Q. Suppose the number of axis nodes in V is m. For each axis node of V, denoted as A_{V_i}, if the predicates of A_{V_i} in V are different from those of A_{Q_i} in Q, we get the answers of A_{V_i} through querying A_{Q_i} on the cached results of A_{V_i}; otherwise, the answers of A_{Q_i} are exactly the cached answers of A_{V_i}. Then, we generate the answers of the $m\text{-}th$ axis node of Q by retrieving the answers of A_{V_m}, which satisfy the path of Q from the $1\text{-}th$ axis node to the $m\text{-}th$ axis node. Finally, we generate the final answers of Q through querying Q_m based on the answers of the $m\text{-}th$ axis node. Interested readers are referred to our previous works (Li et al. 2006a) for the details about how to reconstruct V to answer Q.

5.3 Cache replacement

As the frequent query patterns are more likely to be issued subsequently, we cache the recently discovered frequent query patterns. When cache replacement is needed, we first replace the infrequent query patterns and their corresponding answers. If the

space for admitting the new query result is still not sufficient, the cached results corresponding to some frequent query patterns will be replaced according to replacement policies.

Inspired from LFU and LRU, in this paper, we integrate LFU and LRU into $LFRU$ and propose a novel cache replacement based on $LFRU$. We always replace the least frequently and recently used query. In our approach, the cached queries are classified into two categories according to the visited time. One category is the recent 20% visited queries and the other category is the other 80% queries. We assign the two parts with two importance ratios, α and β.

Suppose the queries in the database is $\{q_1, q_2, \ldots, q_n\}$, we record the visited frequency f_i, the recent visited time t_i, the execution cost c_i and the occupied size s_i for each query q_i. We always first replace query q_i if $(\gamma_i * f_i * c_i)/s_i$ is minimal among all such queries, where

$$\gamma_i = \begin{cases} \frac{\alpha}{\beta} \text{ if } \quad q_i \quad \text{is in the category of 20\% recent queries} \\ 1 \quad \text{otherwise} \end{cases}$$

We note that recently issued queries are more important, therefore $\frac{\alpha}{\beta}$ should be larger than 1. We set it to 4 in the experiments as the algorithm achieves the highest performance at this point. When the number of new queries increases to 10% of that of original queries, we use incremental algorithms ESPRIT-i/ESPRIT-i^+ to incrementally mine the updated database and cache the up-to-date frequent queries while the XML-DBMS is not busy. We will experimentally demonstrate the effectiveness of our proposed techniques in Sect. 6.

6 Experimental study

This section evaluates the performance of our proposed algorithms and demonstrates the efficiency and scalability of our approaches in mining frequent XQPs.

To the best of our knowledge, FastXMiner (Yang et al. 2003a) and 2PXMiner (Yang et al. 2004) are the most efficient algorithm for frequent XQP discovery. increQPMiner (Chen et al. 2004) is the state-of-the-art algorithm to incrementally mine frequent XQPs. We compared ESPRIT with state-of-the-art algorithm FastXMiner (Yang et al. 2003a) and 2PXMiner (Yang et al. 2004) on static databases, and compared our incremental algorithms ESPRIT-i/ESPRIT-i^+ with the best algorithm increQPMiner (Chen et al. 2004) on evolving databases. We used different datasets by varying different values of min_sup and the numbers of selected queries.

The datasets we used are real-world data DBLP,[4] SigmodRecord,[5] TreeBank,[6] and synthetic data XMark.[7] All the queries generated from the datasets followed the default Zipfian distribution. According to the DTDs of these three datasets, some '//' and '*'

[4] http://dblp.uni-trier.de/xml/.

[5] http://www.sigmod.org/record/xml/.

[6] http://www.cs.washington.edu/research/xmldatasets/.

[7] http://www.xml-benchmark.org/.

Table 4 The characteristics of XQPs we used in the experiments

Datasets	Average # of nodes	Max depth	Max fan-out
XMark	8.4	11	11
DBLP	7.6	8	12
SigmodRecord	5.4	5	4
TreeBank	9.2	16	8

nodes are added to construct the $XQPs$. Each XQP has 0.3 probability containing '//' and 0.8 probability containing '*'. Different characteristics of $XQPs$ are shown in Table 4. There were 100–1000K queries. The number of transactions was about 100 and the size of each transaction was about 32 MB. In contrast, the average number of nodes, maximum depth, and fan-out of $XQPs$ reflect the complexity of the queries. All the experiments were carried out on a computer with Pentium 2.4 GHz and 2GB RAM running Windows XP. All the algorithms were implemented in C++.

6.1 Comparison on static databases

6.1.1 Evaluation of minimum support on static databases

In this section, we compared ESPRIT with FastXMiner and 2PXMiner by varying the values of min_sup on static databases. In the comparison, we chose 100K $XQPs$ for min_sup in the range of 0.2% and 2.5%, and 500K $XQPs$ for min_sup in the range of 2% and 12% in every dataset as our experimental inputs. Figures 15, 16, 17, and 18 show the experimental results of ESPRIT, FastXMiner and 2PXMiner on static databases by varying different values of min_sup on different datasets. Even if on the dataset with nested structures (TreeBank), our method still achieves higher efficiency.

We observe that ESPRIT outperforms FastXMiner and 2PXMiner on each dataset significantly, and 2PXMiner is better than FastXMiner. ESPRIT is about 5–15 times faster than FastXMiner, and is about 3–8 times faster than 2PXMiner. This is because, FastXMiner and 2PXMiner always need to match an increasing number of frequent XQP candidates and involves costly tree-containment testing, while ESPRIT avoids

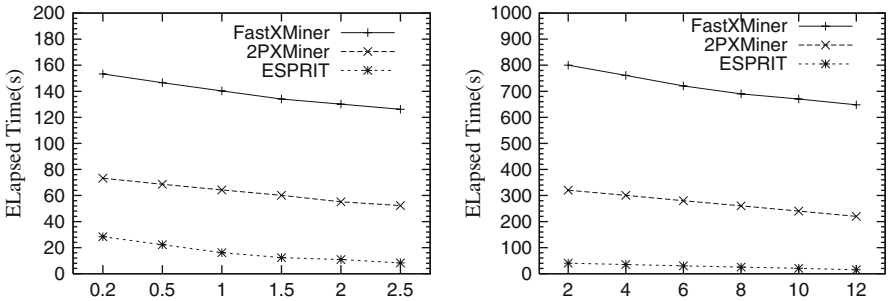


Fig. 15 Evaluation on static databases by varying min_sups (SigmodRecord dataset)

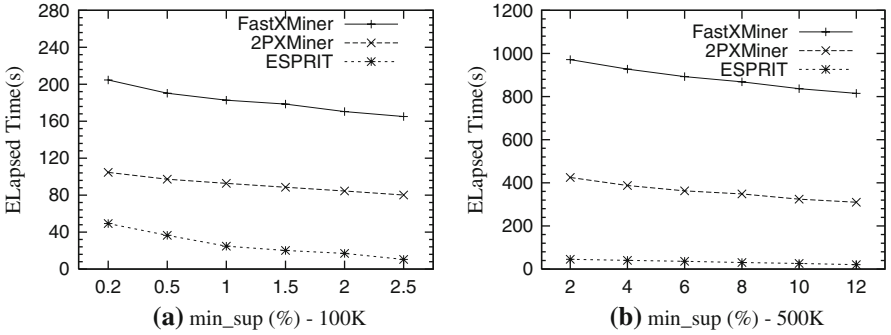


Fig. 16 Evaluation on static databases by varying *min_sup*s (DBLP dataset)

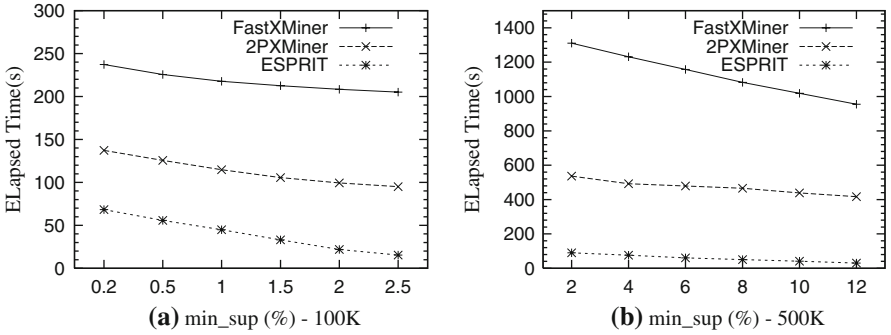


Fig. 17 Evaluation on static databases by varying *min_sup*s (XMark dataset)

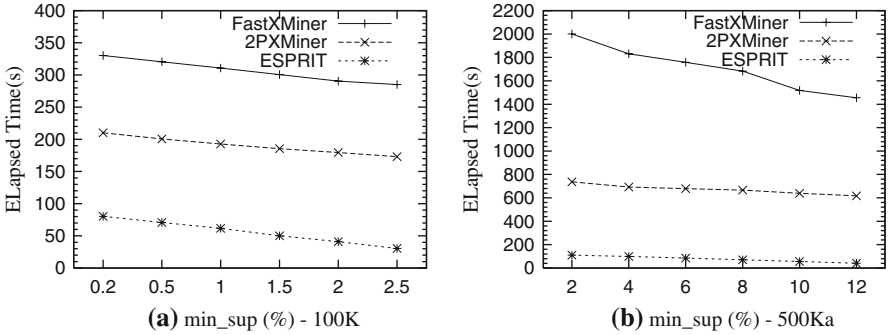


Fig. 18 Evaluation on static databases by varying *min_sup*s (TreeBank dataset)

redundant sequences testing by dynamic enumeration and pruning after the parent–child constraint is applied based on our proposed sequence enumeration technique.

6.1.2 Scalability on static datasets

We evaluated the scalability of our algorithms by varying the number of XQPs on the three datasets and fixing *min_sup* at 1%.

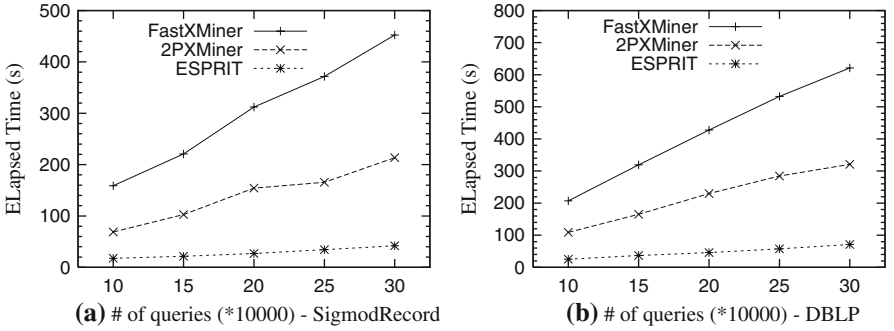


Fig. 19 Evaluation on static databases by varying numbers of queries ($min\_sup = 1\%$)

Figure 19 shows the performance results obtained on DBLP and SigmodRecord by varying numbers of XQPs. ESPRIT has better scalability compared with FastXMiner and 2PXMiner. On SigmodRecord (Fig. 19a), when the number of XQPs is two million, ESPRIT costs only about 25s while FastXMiner costs 320s and 2PXMiner costs 150s. This is so because, (1) ESPRIT need not join frequent XQPs; and (2) ESPRIT avoids the expensive tree-containment testing.

6.2 Comparison on synthetic databases

We evaluate different algorithms on synthetic datasets. We generate different synthetic datasets as follows. In the DTD of the dataset, each node has k children with distinct labels (except the root node and leaf nodes). The DTD has l levels. Figure 20 illustrates the DTD structure. We vary k to change the fan-out of the datasets, and vary l to change the depth of the datasets. We select 100K queries from different datasets and randomly add '//' and '*' nodes in the queries. Figure 21 shows the experimental results by varying k and fixing $l = 6$. Figure 22 illustrates the experimental results by varying l and fixing $k = 4$.

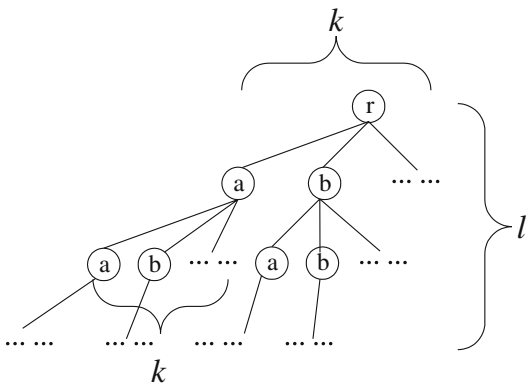


Fig. 20 DTD of synthetic datasets

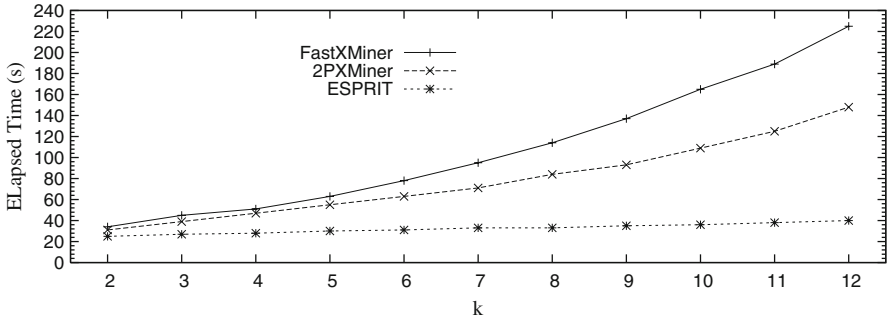


Fig. 21 Evaluation on different datasets by varying k($l = 6$, $min\_sup = 1\%$ and $|\mathcal{D}| = 100$K)

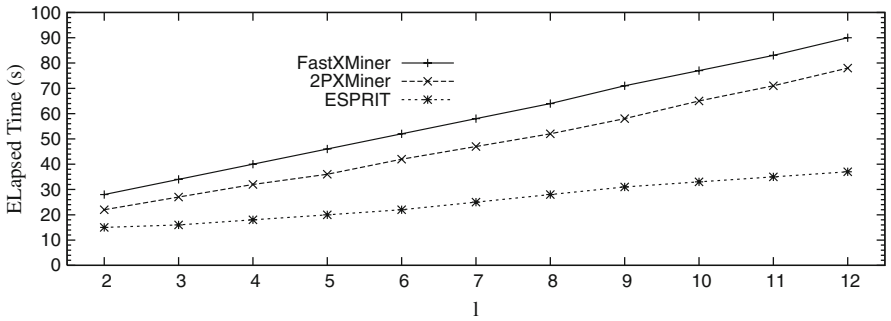


Fig. 22 Evaluation on different datasets by varying l($k = 4$, $min\_sup = 1\%$ and $|\mathcal{D}| = 100$K)

We observe that with the increase of k, the elapsed time of FastXMiner and 2PXMiner increases sharply. While ESPRIT increases a little. This is so because, the two previous algorithms need join many frequent XQPs. The larger k, the more numbers of XQPs need to be joined. It is very expensive to join frequent XQPs if there are large numbers of frequent XQPs. On the other hand, the elapsed time of the three methods is nearly linear with l. This is so because FastXMiner and 2PXMiner need not join large numbers of frequent XQPs as the fan-out of the datasets is not large.

6.3 Comparison on evolving databases

In this section, we compared ESPRIT-i and ESPRIT-i^+ with increQPMiner, ESPRIT and 2PXMiner on evolving transaction databases.[8]

6.3.1 Evaluation of minimum support on evolving databases

We selected 200K queries as the original database and added 100K queries as the incremental database. The three incremental algorithms, ESPRIT-i, ESPRIT-i^+ and

[8] As 2PXMiner always outperforms FastXMiner, we only compare with 2PXMiner in the remainder of this paper.

increQPMiner, incrementally mine the frequent $XQPs$, while the other two algorithms, ESPRIT and 2PXMiner, mine the frequent $XQPs$ on $\mathcal{D}^u$ from scratch.

Figures 23, 24, and 25 show the experimental results of the five algorithms on different datasets. We can see our two incremental algorithms outperform the other algorithms significantly. This further demonstrates that the incremental mining is more efficient than mining from scratch. When min_sup is 2%, ESPRIT-i is about 15–30 times faster than 2PXMiner, 5–10 times faster than ESPRIT and 3–6 times faster than increQPMiner. As the incremental algorithms are always better than the static algorithms, we only compare the incremental algorithms in the remainder part.

To better understand our algorithms, we compared ESPRIT-i^+ with ESPRIT-i on DBLP dataset and we set min_sup as 1%. We varied the ratio between $|d|$ and $|\mathcal{D}|$. The obtained experimental results are shown in Fig. 26. Note that, ESPRIT-i^+ achieves better performance and outperforms ESPRIT-i, as ESPRIT-i^+ makes full use of the mined results of $\mathcal{D}$ and interacts with the original database $\mathcal{D}$ fully to mine

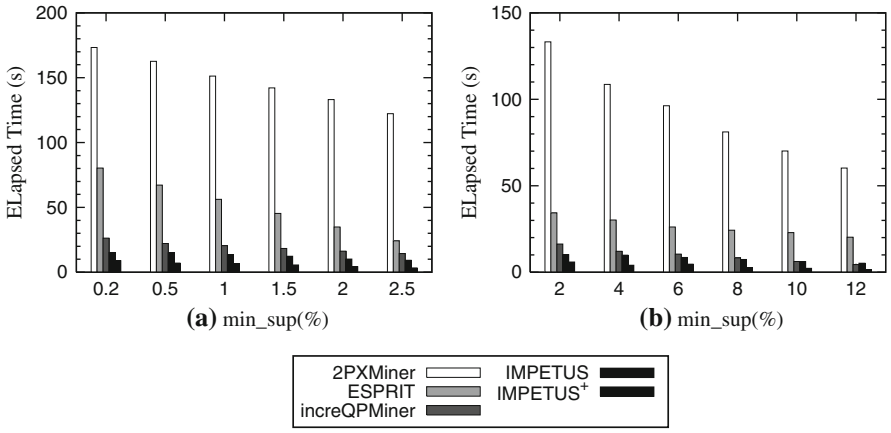


Fig. 23 Evaluation of incremental mining on SigmodRecord dataset ($|\mathcal{D}| = 200$K, $|d| = 100$K)

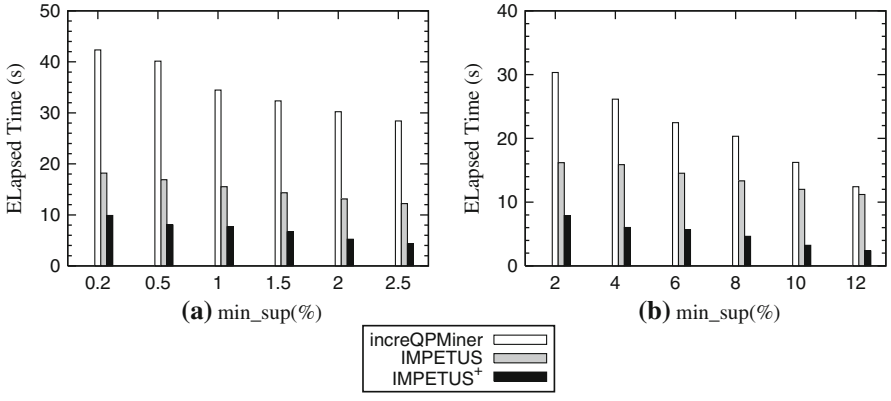


Fig. 24 Evaluation of incremental mining on DBLP dataset ($|\mathcal{D}| = 200$K, $|d| = 100$K)

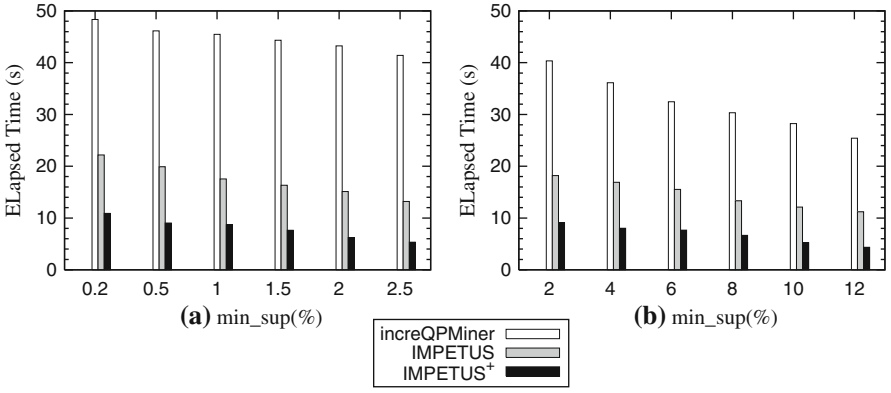


Fig. 25 Evaluation of incremental mining on XMark dataset ($|\mathcal{D}| = 200$K, $|d| = 100$K)

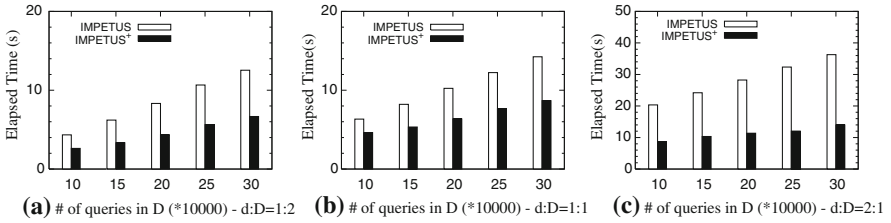


Fig. 26 Evaluation of incremental mining on DBLP by varying $|d|/|\mathcal{D}|(min\_sup = 1\%)$

the up-to-date frequent queries. With the increase of $|d|/|\mathcal{D}|$, ESPRIT-i^+ achieves much better performance than ESPRIT-i as the latter needs to scan the incremental database independently and does not interact with the original database.

6.3.2 Scalability on evolving database

This section evaluates the scalability of our algorithms by varying the number of $XQPs$ on different datasets. We selected 200K queries as the original database, fixed the min_sup at 1%, and varied the numbers of queries in the incremental database d. Figure 27 illustrates the experimental results obtained.

We can see ESPRIT-i and ESPRIT-i^+ outperform increQPMiner significantly. This is because the former two algorithms take full advantage of mined results of $\mathcal{D}$ while increQPMiner makes use of a part of the mined results. More importantly, our methods use efficient indices, which can facilitate the incremental mining.

6.4 Incremental mining

To further evaluate the performance of the incremental algorithms, we introduce another good metric, $Speedup$. Speedup of algorithm A over algorithm B, is T_B/T_A, where T_A and T_B are the elapsed time of A and B respectively.

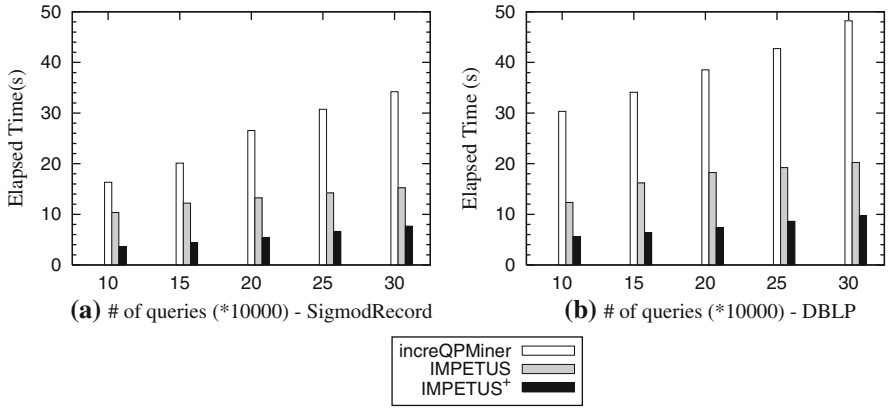


Fig. 27 Evaluation of incremental mining by varying number of queries ($min\_sup = 1\%$)

We selected 100K $XQPs$ on DBLP dataset as the original database $\mathcal{D}$. We added an incremental database d into $\mathcal{D}$. For each d, we used the algorithms to mine the up-to-date frequent query patterns on $\mathcal{D}^u$ until $|\mathcal{D}^u| = 2 * |\mathcal{D}|$ and compared the total elapsed time of different algorithms.

In Fig. 28a, we increased $\mathcal{D}^u$ by adding d with $|d| = 10$K at each time and compared the total elapsed time. We can observe that, with the increase of $\mathcal{D}^u$, the speedups of ESPRIT-i over 2PXMiner, ESPRIT and increQPMiner also go up. It reflects that incremental mining methods are very efficient for the evolving databases. In Fig. 28b, we fixed $|d|/|\mathcal{D}| = 5\%$ and added d into $\mathcal{D}^u$ until $|\mathcal{D}^u| = 2 * |\mathcal{D}|$ with different values of min_sup. Figure 28b gives the speedups of the selected algorithms by varying different values of min_sup. The experimental results show that the incremental mining methods are more efficient than the static ones on whatever the values of min_sup. Especially, when $min\_sup = 2\%$, the speedups of ESPRIT-i over 2PXMiner, ESPRIT, increQPMiner are 206, 148, 44 respectively.

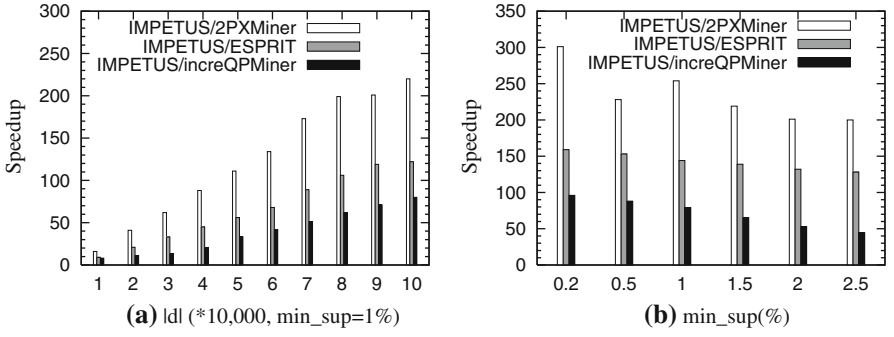


Fig. 28 Speedup over different algorithms on DBLP ($|\mathcal{D}| = 100$K, $|\mathcal{D}^u| = 200$K)

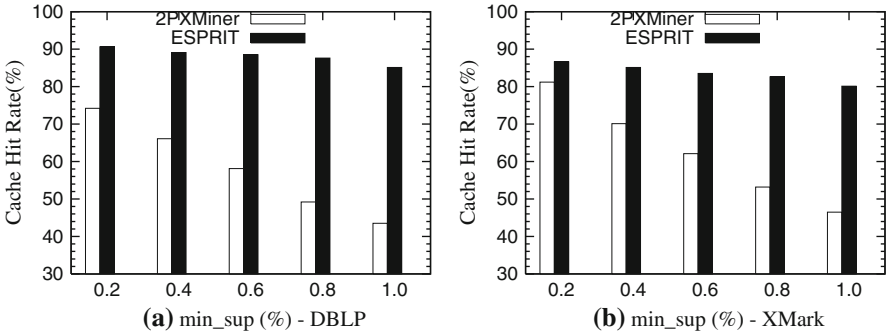


Fig. 29 Cache hit rate on static databases ($|\mathcal{D}| = 100$K, varying min_sup)

6.5 Effectiveness of caching

This section demonstrates how the frequent query patterns discovered can be used to improve the performance of caching and how the replacement techniques improve cache hit rate. Our optimization policies of query rewriting, cache lookup and cache replacement are adopted in our algorithms, ESPRIT, ESPRIT-i and ESPRIT-i^+.[9]

We first evaluated the cache hit rate on the static database. We selected 100K XQPs as the original database. We mined these 100K XQPs by varying the different values of min_sup and cached all the the mined queries and the corresponding answers. Then, we added 100K new queries as an incremental database d to join the original database. We evaluated the average cache hit rate of evaluating these 100K queries. The experiential results are illustrated in Fig. 29.

We can observe that ESPRIT outperforms the traditional method significantly as the optimization techniques we proposed can improve the answerability and the hit rate of caching. In contrast, on DBLP when min_sup is 0.2%, the cache hit rate of 2PXMiner is 74%; when min_sup is 1%, the cache hit rate of 2PXMiner falls to 42%. However, the cache hit rate of our algorithms is always more than 85%. Moreover, ESPRIT leads to 10-40% cache hit rate over 2PXMiner as illustrated in Fig. 29. This is so because we employed several more effective strategies of query rewriting, cache lookup and cache replacement.

We then evaluated the cache hit rate on evolving databases. We first selected 100K XQPs as the original database. Then the new queries to join the database were varied from 10K to 100K. Noted that, we incrementally mined the up-to-date XQPs by employing the three incremental algorithms when the number of XQPs are 110, 120, . . ., and 200K. Figure 30 shows the experimental results of cache hit rate.

We observe that ESPRIT, ESPRIT-i and ESPRIT-i^+ achieve more efficient answerability and higher hit rate than 2PXMiner and increQPMiner. Note that the hit rate of our algorithms is barely affected by the increase of min_sup; while the hit rate of other algorithms falls rapidly. The experimental results show that our policies of cache replacement, cache lookup and query rewriting indeed improve the query per-

[9] We set $\alpha/\beta = 4$ in all the experiments.

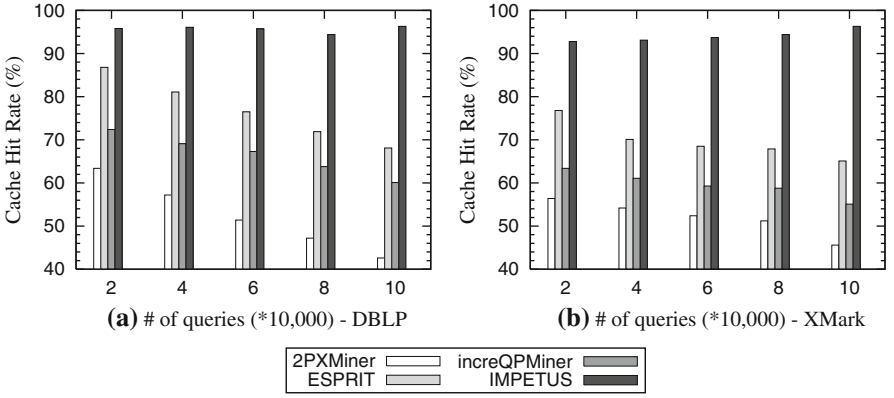


Fig. 30 Cache hit rate on evolving databases ($min\_sup = 1\%$, $|\mathcal{D}| = 100$K, varying $|d|$)

formance of XML-DBMS. Note that if cache hit, we use the cached results to answer the query. If cache miss, we need access disk to answer the query. Thus, they involve more I/O consumption as their cache hit rates are lower than those of our methods.

Finally, we investigated the average response time (the average time taken to answer a query) to answer new queries. We selected 100K queries as the original database. We then added 10K queries at each time. Noted that, we mined the up-to-date XQPs by employing the three incremental algorithms when the numbers of XQPs are 110, 120, . . ., and 200K. The experimental results are illustrated in Fig. 31. The average response time of ESPRIT-i is about fifth of that of 2PXMiner and third of that of increQPMiner. The reason is that, ESPRIT-i improves the cache hit rate significantly and thus improves the answerability of caching. More importantly, ESPRIT-i is much more efficient than processing queries directly without using the cached views.

Observed from the above experimental results, we see that our algorithms are much more efficient and have better scalability and answerability than the state-of-the-art algorithms, FastXMiner, 2PXMiner and increQPMiner. Moreover, ESPRIT-i and ESPRIT-i^+ are very efficient for the evolving transaction databases.

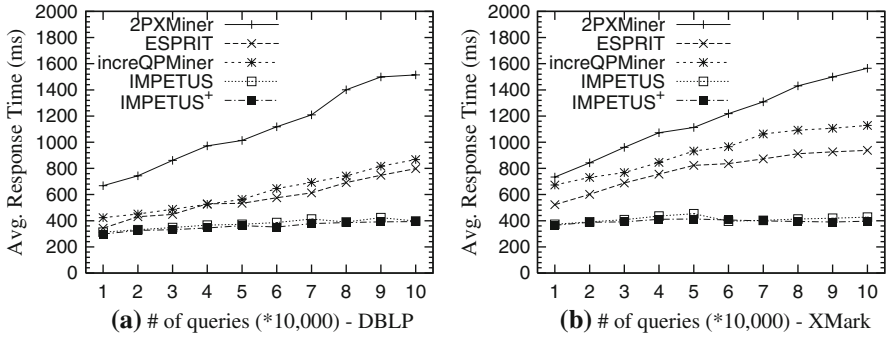


Fig. 31 Average response time on evolving databases ($min\_sup = 1\%$, $|\mathcal{D}| = 100$K, varying $|d|$)

7 Related work

As XML has become de facto standard for information representation and exchange over the Internet, many researchers have studied the problem of semantic caching for XML databases. Chen et al. (2002) first proposed to apply the ideas of semantic caching to XML query processing systems, in particular the XQuery engine. Semantic caching implies view-based query answering and cache management. Hristidis and Petropoulos (2002) presented a novel framework for semantic caching of XML databases. The cached XML data were organized using a modification of the incomplete tree, which has many desirable properties, such as incremental maintenances, containment decidability and remainder queries generation in PTIME. Xu (2005) introduced a novel framework for a new semantic caching system, which offers the representation system of cached XML data, the algorithms to decide whether a new query can be totally answered by cached XML data or not, and to incrementally maintain the cached XML data.

More recently, we devised an efficient approach to improve the answerability of semantic cache by decomposing XML queries into simple components and employing a technique of the divisibility of prime number products (Li et al. 2006a), which leads to a significant improvement over previous works in terms of the elapsed time of cache lookup and cache hit rate. We proposed a novel method of exploiting sequencing views in semantic cache to accelerate XPath query evaluation so as to improve the answerability of caching (Feng et al. 2007). We have studied the problem of clustering XML documents by employing the technique of transforming XML documents to sequences (Aggarwal et al. 2007).

As to frequent XQPs mining, to the best of our knowledge, XQPMiner (Yang et al. 2003b) is the first algorithm to mine frequent $XQPs$ with a global XQP schema guided enumeration mining algorithm for frequent XQP mining. It 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. FastXMiner (Yang et al. 2003a) is proposed to improve the performance beyond XQPMiner, as only valid candidate $XQPs$ are enumerated for costly tree-containment testing, as opposed to all the candidates of XQPMiner (Yang et al. 2003b). 2PXMiner (Yang et al. 2004) is proposed to reduce the number of costly tree inclusion tests to improve query performance beyond FastXMiner. However, these methods have to enumerate all of the candidates and involve the costly tree-containment checking, and thus lead to inefficiency. increQPMiner (Chen et al. 2004) studies the problem of incremental mining by using the mined results of the original databases. However, increQPMiner is not as efficient as our incremental algorithms (Li et al. 2006b), as increQPMiner does not take full advantages of the mined results of the original database. Our previous work SOLARIA (Feng et al. 2006) uses a sequence based method to mine frequent patters. However SOLARIA does not support '//', and thus it is not as powerful as other methods. SOLARIA also cannot incrementally mine the evolving databases.

In addition, 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 for caching, and so on. For tree and graph mining, frequent pattern discovering was first addressed in biological science. An adaptive path index for XML data (APEX) (Chung et al. 2002) was proposed to utilize frequently used paths to improve the query performance. Dehaspe et al. (1998) proposed an efficient algorithm to mine frequent substructures in protein and chemical compounds. In graph database, algorithm FSG proposed in Kuramochi and Karypis (2001) is considered as a fast miner for discovering connected sub-graphs by extending the notion of level-by-level expansion of Agrawal and Srikant (1994). Motivated by discovering user navigation patterns in web surfing. Zaki (2002, 2005) proposed subtree mining algorithm in forest, which faces more complex data situation. FREQT (Asai et al. 2002) and TreeFinder (Termier et al. 2002) aimed at finding frequent subtrees in a collection of semi-structured documents, but still cannot solve the problem of XQP mining due to the existence of '*' and '//' for XPath queries. Another closest related work is finding the frequent substructures from a collection of semi-structured Web documents (Wang and Liu 2000), and mining frequent sequential patterns (Ayres et al. 2002; Han et al. 2000; Masseglia et al. 1998; Srikant and Agrawal 1996) which mainly focus on general and constraint-based sequence mining problems. Frequent episode mining (Yan et al. 2003), cyclic association rule mining (Ozden et al. 1998), temporal relation mining (Bettini et al. 1998), partial periodic pattern mining (Han et al. 1999), and long sequential pattern mining in noisy environment (Yang et al. 2002) have been studied. 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 (Yan et al. 2003) and BIDE (Wang and Han 2004) are 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 and outperforms prior works. In future work, we want to study the problem of mining closed frequent query patterns.

8 Conclusion

In this paper, we have studied the problem of incrementally mining frequent query patterns from XML queries for caching to improve the performance of XML query processing. We presented an efficient algorithm ESPRIT to mine frequent XQPs. ESPRIT employs the idea of sequential pattern mining for replacing expensive tree-containment testing with cheap parent–child validity checking. To enhance the efficiency of mining the frequent patterns, we proposed two algorithms ESPRIT-i and ESPRIT-i^+ to incrementally mine the frequent valid sequences for the evolving transaction databases by incorporating two novel indices of $F\text{-}index$ and $Q/F\text{-}index$. ESPRIT-i^+ is much more efficient and outperforms other algorithms as ESPRIT-i^+ makes full use of the mined results of the original database $\mathcal{D}$ and interacts with $\mathcal{D}$ fully to discover the up-to-date frequent queries. We also examined several optimization techniques of query rewriting, cache lookup, and cache replacement to improve the answerability and the hit rate of caching. We have implemented our algorithms and conducted an extensive set of experimental studies. The thorough experimental

results show that our algorithms significantly outperform state-of-the-art algorithms in terms of efficiency, scalability as well as answerability.

Acknowledgements This work is partly supported by the National Natural Science Foundation of China under Grant No. 60873065, the National High Technology Development 863 Program of China under Grant No. 2007AA01Z152, and the National Grand Fundamental Research 973 Program of China under Grant No. 2006CB303103.

References

Agrawal R, Srikant R (1994) Fast algorithms for mining association rules in large databases. In: VLDB, pp 487–499

Aggarwal C, Ta N, Wang J, Feng J, Zaki MJ (2007) Xproj: a framework for projected structural clustering of xml documents. In: KDD

Asai T, Abe K, Kawasoe S, Arimura H, Sakamoto H, Arikawa S (2002) Efficient substructure discovery from large semi-structured data. In: SDM

Aumann Y, Feldman R, Liphstat O, Mannila H (1999) Borders: an efficient algorithm for association generation in dynamic databases. J Intell Inf Syst 12(1):61–73

Ayres J, Flannick J, Gehrke J, Yiu T (2002) Sequential pattern mining using a bitmap representation. In: KDD

Balmin A, Ozcan F, Beyer K, Cochrane R, Pirahesh H (2004) A framework for using materialized xpath views in xml query processing. In: VLDB, pp 60–71

Bettini C, Wang XS, Jajodia S (1998) Mining temporal relationships with multiple granularities in time sequences. IEEE Data Eng Bull 21(1):32–38

Chen L, Rundensteiner EA, Wang S (2002) Xcache: a semantic caching system for xml queries. In: SIGMOD

Chen Y, Yang L, Wang YG (2004) Incremental mining of frequent xml query patterns. In: ICDM, pp 343–346

Chung C-W, Min J-K, Shim K (2002) Apex: an adaptive path index for xml data. In: SIGMOD Conference, pp 121–132

Dehaspe L, Toivonen H, King R (1998) Finding frequent substructures in chemical compounds. In: KDD, pp 30–36

Feng J, Qian Q, Wang J, Zhou L (2006) Exploit sequencing to accelerate hot xml query pattern mining. In: ACM SAC

Feng J, Ta N, Li G (2007) Exploit sequencing views in semantic cache to accelerate xpath query evaluation. In: WWW

Han J, Dong G, Yin Y (1999) Efficient mining of partial periodic patterns in time series database. In: ICDE, pp 106–115

Han J, Pei J, Mortazavi-Asl B, Chen Q, Dayal U, Hsu M (2000) Freespan: frequent pattern-projected sequential pattern mining. In: KDD, pp 355–359

Hristidis V, Petropoulos M (2002) Semantic caching of xml databases. In: WebDB

Kaushik R, Shenoy P, Bohannon P, Gudes E (2002) Exploiting local similarity for indexing paths in graph-structured data. In: ICDE, pp 129–140

Kuramochi M, Karypis G (2001) Frequent subgraph discovery. In: ICDM, pp 313–320

Kwon J, Rao P, Moon B, Lee S (2005) Fist: scalable xml document filtering by sequencing twig patterns. In: VLDB, pp 217–228

Li G, Feng J, Ta N, Zhang Y, Zhou L (2006a) Scend: an efficient semantic cache to exploit xpath query/view answerability. In: WISE, pp 460–473

Li G, Feng J, Wang J, Zhang Y, Zhou L (2006b) Incremental mining of frequent query patterns from xml queries for caching. In: ICDM, pp 350–361

Luo Q, Krishnamurthy S, Mohan C, Pirahesh H, Woo H, Lindsay BG, Naughton JF (2002) Middle-tier database caching for e-business. In: SIGMOD, pp 600–611

Mandhani B, Suciu D (2005) Query caching and view selection for xml databases. In: VLDB

Masseglia F, Cathala F, Poncelet P (1998) The psp approach for mining sequential patterns. In: PKDD

Milo T, Suciu D (1999) Index structures for path expressions. In: ICDT, pp 277–295

Ozden B, Ramaswamy S, Silberschatz A (1998) Cyclic association rules. In: ICDE, pp 412–421

Pei J, Han J, Mortazavi-Asl B, Pinto H, Chen Q, Dayal U, Hsu M (2001) Prefixspan: Mining sequential patterns by prefix-projected growth. In: ICDE, pp 215–224
Prüfer H (1918) Neuer beweis eines satzes uber permutationen. Archiv fur Mathematik und Physik 27:142–144
Qun C, Lim A, Ong KW (2003) D(k)-index: an adaptive structural summary for graph-structured data. In: SIGMOD, pp 134–144
Rao PR, Moon B (2004) Prix: indexing and querying xml using prufer sequences. In: ICDE, pp 288–299
Re C, Brinkley J, Hinshaw K, Suciu D (2004) Distributed xquery. In: Information integration on the web (IIWeb)
Srikant R, Agrawal R (1996) Mining sequential patterns: generalizations and performance improvements. In: EDBT, pp 3–17
Termier A, Rousset M-C, Sebag M (2002) Treefinder: a first step towards xml data mining. In: ICDM, pp 450–457
Wang J, Han J (2004) Bide: efficient mining of frequent closed sequences. In: ICDE, pp 79–90
Wang K, Liu H (2000) Discovering structural association of semistructured data. IEEE TKDE 12(2):353–371
Wang H, Park S, Fan W, Yu PS (2003) Vist: a dynamic index method for querying xml data by tree structures. In: SIGMOD, pp 110–121
Xu W (2005) The framework of an xml semantic caching system. In: WebDB
Yan X, Han J, Afshar R (2003) Clospan: mining closed sequential patterns in large databases. In: SDM
Yang J, Wang W, Yu PS, Han J (2002) Mining long sequential patterns in a noisy environment. In: SIGMOD, pp 406–417
Yang LH, Lee M-L, Hsu W (2003a) Efficient mining of xml query patterns for caching. In: VLDB, pp 69–80
Yang LH, Lee M-L, Hsu W, Acharya S (2003b) Mining frequent query patterns from xml queries. In: DASFAA, pp 355–362
Yang LH, Lee ML, Hsu W, Guo X (2004) 2pxminer: an efficient two pass mining of frequent xml query patterns. In: KDD
Zaki MJ (2002) Efficiently mining frequent trees in a forest. In: SIGKDD, pp 71–80
Zaki MJ (2005) Efficiently mining frequent trees in a forest: algorithms and applications. IEEE TKDE 17(8):1021–1035