Exploit Sequencing to Accelerate Hot XML Query Pattern Mining*

Jianhua Feng Qian Qian Jianyong Wang Lizhu Zhou
Department of Computer Science and Technology
Tsinghua University, Beijing 100084, China
{fengjh, jianyong, dcszlz}@tsinghua.edu.cn qqpeter99@mails.tsinghua.edu.cn

ABSTRACT
Speeding up query evaluation in large XML repositories becomes a challenging and all-important problem with vast XML-related applications arising. Upon discovery of hot XML query patterns, indexing and caching can be effectively adopted for query performance enhancement. Previous algorithms for finding hot query patterns basically introduced a straightforward generate-and-test strategy. In this paper, we present, SOLARIA, an efficient algorithm for mining hot XML query patterns without candidate maintenance and costly tree-containment checking. Efficient algorithm of sequence mining is involved in discovering frequent tree-structured patterns, which aims at replacing expensive containment testing with cheap parent-child checking in sequences. SOLARIA deeply prunes unrelated search space for frequent pattern enumeration by parent-child relationship constraint. With the motivation of indexing and caching in XML query optimization, we also propose the derived algorithm SOLARIA* for mining hot “closed” XML query patterns which provide compact and complete structure information. By a thorough experimental study on various real-life data, we demonstrate the efficiency and scalability of SOLARIA over the previous known alternative. SOLARIA is also linearly scalable in terms of XML queries' size.

1. Introduction
With the rapid growth for the last decade, XML data has become one of the most important collections of knowledge that the human being ever had. The discovery of XML query patterns gains its focus with huge demands of efficient query evaluation and information retrieval arising. As much research has been undertaken on XML indexing [12, 6, 14] and caching [25], discovering frequent XML query patterns turns out to be a significant and effective premise of query optimization for its capability of “focus” capturing.

As an example, Figure 1 shows two sample XML queries occurring in an XML database of information about professors in a university. Query(1) finds all names of professors who published a paper on “SAC” with the year of “2004”, while Query(2) finds names of professors who teach class “Data Mining” and published a paper on “SAC”.

Consider the two query patterns in Figure 1, the sub-structure “Prof/Paper/Conference” is a frequent pattern, which hints that extra indexing or caching established on this “focus” will definitely bring benefits for query evaluation. For this purpose, hot query pattern mining gains the momentum for its crucial role about query optimization.

Figure 1: Sample XML queries

Traditional frequent pattern mining approaches typically follow straightforward generate-and-test strategy, which includes two passes of frequent pattern generation and containment testing [22, 7, 5, 17]. The recent tree-structured data mining research mainly moves towards efficient frequent pattern enumeration and fast containment testing algorithms. Motivated by the need of reducing costly containment testing, we propose an efficient algorithm, SOLARIA, which involves a novel application of frequent sequence mining and can be used to accelerate hot XML queries. SOLARIA mines the set of frequent sequences that preserve the parent-child and sibling ordering relationships among the XML elements/attributes, which avoids expensive tree containment testing elegantly.

Our contributions: The contributions of this paper can be summarized as follows.

1. SOLARIA stands for Sequence based Hot XML query Pattern Mining Algorithm

* The work was supported by National Natural Science Foundation of China under Grant No.60573094 & No.60573061 and Tsinghua Basic Research Foundation under Grant No. JCrq2005022

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC’06, April, 23-27, 2006, Dijon, France.
Copyright 2006 ACM 1-59593-108-2/06/0004/55.00
1) First, we demonstrate a novel application of the sequence mining approach to the filed of hot XML query pattern mining.

2) Second, we devise an efficient algorithm, SOLARIA, by pushing the parent-child and sibling ordering constraint into the traditional sequential pattern mining. As it gets rid of the candidate maintenance and costly sub-tree containment testing, it greatly enhances the mining performance in hot XML query pattern discovery.

3) To reduce the redundancy among frequent substructures, SOLARIA is extended to mine closed frequent XML query patterns, which provides solid support for self-tuning and intelligent XML query optimization research.

4) A thorough performance study is conducted to evaluate SOLARIA’s efficiency and scalability in comparison with two state of the art algorithms, XQPMiner[26] and FastXMiner[25].

The rest of this paper is organized as follows. Section 2 gives related work about frequent XML pattern mining. We formalize the frequent XML query pattern mining problem in Section 3. Section 4 presents SOLARIA, an effective and elegant sequencing algorithm and discusses the extension of SOLARIA to mine closed XML query patterns. In Section 5, we present a thorough experimental study to evaluate SOLARIA. We will conclude in Section 6.

2. Related Work

Mining frequent substructure 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. [8] proposes an efficient algorithm to mine frequent substructures in protein and chemical compounds. In graph database, algorithm FSG in [11] is treated as a fast miner for discovering connected sub-graphs by extending the notion of level-by-level expansion of [1]. Motivated by discovering user navigation patterns in web surfing, [28] proposes sub tree mining algorithm in forest, which faces more complex data situation. FREQT [2] and TreeFinder [20] aim at finding frequent subtrees in a collection of semi-structured documents, but still cannot solve the problem of XML query pattern mining due to the existence of "*" and "//". To our best knowledge, FastXMiner [25] is the most efficient mining algorithm for XML frequent query pattern discovery, as only valid candidate XSQPs are enumerated by FastRSTGen for costly containment testing. It still follows the traditional idea of generate-and-test paradigm for tree-structured data mining. Global query pattern tree needs to be generated for XSQ enumeration, as well as expensive candidate generation and containment testing.

On the other hand, for sequence mining, [13, 19, 10] are mainly focusing on general and constraint-based sequence mining problems. Various researches have been done on frequent episode mining [24], cyclic association rule mining [15], temporal relation mining [4], partial periodic pattern mining [9], and long sequential pattern mining in noisy environment [27]. 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[24] and BIDE[21] 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.

3. Preliminaries

XML Query Pattern (XQP). The XML queries are mainly expressed with XPath [7] or XQuery [5] nowadays, whose basic features are regular path expressions. XML queries also conform to a labeled tree structure. According to the definition in [25], XML Query Pattern can be defined as a tree XQP=<V, E>, where V is the vertex set, E is the edge set. Each edge e=(v1, v2) indicates node v1 is the parent of node v2. Each vertex v’s label is one of the tag values in {"*/", "+*"} ∪ tagSet, where tagSet is the set of all element and attribute names in the schema. Figure 2 shows an XML data schema in DTD and a query to retrieve the names of professors who have two projects, in which one is focusing on “Tree Mining”, the other is about the topic of “Intelligent Miner on RDB”.

XML Sub Query Pattern (XSQP). Our mining task aims at finding all the XML Sub Query Patterns (XSQP) for a given set of XQP=<V, E>, where XSQP=<V’, E’> is defined as follows:

(1) Root(XSQP)=Root(XQP)
(2) V’ ⊆ V, E’ ⊆ E

Tree Sequentialization. Our approach starts with a valid and effective sequencing method for XQP. Ad hoc sequencing methods such as depth-first and Prüfercode have been used for XML indexing [12, 6]. The Prüfercode [17, 18] is a succinct tree encoding method. In which, an n-node tree is labeled arbitrarily from 1 to n. We encode the tree by deleting the leaf node which has the smallest label and appending the label of its parent to the sequence. So the Prüfer sequence for tree Figure 3 is (2, 6, 3, 6, 6). While if we use depth-first traversal method, we can get the sequence as (P, R, T, D, T, N).

On the other hand, for sequence mining, [13, 19, 10] are mainly focusing on general and constraint-based sequence mining problems. Various researches have been done on frequent episode mining [24], cyclic association rule mining [15], temporal relation mining [4], partial periodic pattern mining [9], and long sequential pattern mining in noisy environment [27]. 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[24] and BIDE[21] 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.

3. Preliminaries

XML Query Pattern (XQP). The XML queries are mainly expressed with XPath [7] or XQuery [5] nowadays, whose basic features are regular path expressions. XML queries also conform to a labeled tree structure. According to the definition in [25], XML Query Pattern can be defined as a tree XQP=<V, E>, where V is the vertex set, E is the edge set. Each edge e=(v1, v2) indicates node v1 is the parent of node v2. Each vertex v’s label is one of the tag values in {"*/", "+*"} ∪ tagSet, where tagSet is the set of all element and attribute names in the schema. Figure 2 shows an XML data schema in DTD and a query to retrieve the names of professors who have two projects, in which one is focusing on “Tree Mining”, the other is about the topic of “Intelligent Miner on RDB”.

XML Sub Query Pattern (XSQP). Our mining task aims at finding all the XML Sub Query Patterns (XSQP) for a given set of XQP=<V, E>, where XSQP=<V’, E’> is defined as follows:

(1) Root(XSQP)=Root(XQP)
(2) V’ ⊆ V, E’ ⊆ E

Tree Sequentialization. Our approach starts with a valid and effective sequencing method for XQP. Ad hoc sequencing methods such as depth-first and Prüfercode have been used for XML indexing [12, 6]. The Prüfercode [17, 18] is a succinct tree encoding method. In which, an n-node tree is labeled arbitrarily from 1 to n. We encode the tree by deleting the leaf node which has the smallest label and appending the label of its parent to the sequence. So the Prüfer sequence for tree Figure 3 is (2, 6, 3, 6, 6). While if we use depth-first traversal method, we can get the sequence as (P, R, T, D, T, N).

On the other hand, for sequence mining, [13, 19, 10] are mainly focusing on general and constraint-based sequence mining problems. Various researches have been done on frequent episode mining [24], cyclic association rule mining [15], temporal relation mining [4], partial periodic pattern mining [9], and long sequential pattern mining in noisy environment [27]. 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[24] and BIDE[21] 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.

3. Preliminaries

XML Query Pattern (XQP). The XML queries are mainly expressed with XPath [7] or XQuery [5] nowadays, whose basic features are regular path expressions. XML queries also conform to a labeled tree structure. According to the definition in [25], XML Query Pattern can be defined as a tree XQP=<V, E>, where V is the vertex set, E is the edge set. Each edge e=(v1, v2) indicates node v1 is the parent of node v2. Each vertex v’s label is one of the tag values in {"*/", "+*"} ∪ tagSet, where tagSet is the set of all element and attribute names in the schema. Figure 2 shows an XML data schema in DTD and a query to retrieve the names of professors who have two projects, in which one is focusing on “Tree Mining”, the other is about the topic of “Intelligent Miner on RDB”.

XML Sub Query Pattern (XSQP). Our mining task aims at finding all the XML Sub Query Patterns (XSQP) for a given set of XQP=<V, E>, where XSQP=<V’, E’> is defined as follows:

(1) Root(XSQP)=Root(XQP)
(2) V’ ⊆ V, E’ ⊆ E

Tree Sequentialization. Our approach starts with a valid and effective sequencing method for XQP. Ad hoc sequencing methods such as depth-first and Prüfercode have been used for XML indexing [12, 6]. The Prüfercode [17, 18] is a succinct tree encoding method. In which, an n-node tree is labeled arbitrarily from 1 to n. We encode the tree by deleting the leaf node which has the smallest label and appending the label of its parent to the sequence. So the Prüfer sequence for tree Figure 3 is (2, 6, 3, 6, 6). While if we use depth-first traversal method, we can get the sequence as (P, R, T, D, T, N).
denoted as \( S_a \subseteq S_b \). A sequence database \( \text{SDB} \) contains a set of tuples in the form of \((\text{Sid}, S)\). The absolute support of a sequence \( S_a \) is the number of tuples in \( \text{SDB} \) that contain \( S_a \), denoted by \( \text{ASup}(S_a) \). The relative support is the percentage of tuples in \( \text{SDB} \) that contain \( S_a \), denoted by \( \text{RSup}(S_a) = \frac{\text{ASup}(S_a)}{|\text{SDB}|} \). If there exists no proper super-sequence of \( S_a \) with the same support as \( S_a \) in \( \text{SDB} \), i.e., not \( \exists S_b \) such that \( S_a \subseteq S_b \) and \( \text{ASup}(S_a) = \text{ASup}(S_b) \) (or \( \text{RSup}(S_a) = \text{RSup}(S_b) \)), we call \( S_a \) a closed sequence.

**Frequent XQP Mining Problem.** Given a set of XQPs \( \{XQP_1, XQP_2, \ldots, XQP_n\} \) and a minimum relative support \( \text{min}_\text{sup} \) ranging from \((0, 1]\), find the set \( S \) of XSQPs such that for each XSQP in \( S \) \( \text{RSup}(XSQP) \geq \text{min}_\text{sup} \) holds. We use the relative support as our measure of frequency throughout the paper. XQPs in our work strictly comply with the schemas of XML documents, which can be considered as ordered, labeled and rooted trees.

### SOLARIA: an Efficient Sequence based Hot XML Query Pattern Mining Algorithm

#### 4.1 Unique Sequence Representation

Given a set of input XQPs \( \{XQP_1, XQP_2, \ldots, XQP_n\} \), the first step in a sequence based XML query pattern mining algorithm is to transform the tree representation of an XML query into a sequence representation. In SOLARIA, we adopt the depth-first traversal as our framework. Figure 4 depicts the input XQP set in our running example, while Table 1 shows the corresponding sequence database, \( \text{SDB} \), generated by depth-first traversing the tree nodes of each XQP. Following we use DFTS to denote a sequence generated with depth-first traversing method.

![Figure 4: An Example of XML query patterns](image)

**Table 1: An example SDB by depth-first traversal**

<table>
<thead>
<tr>
<th>Sequence Identifier</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PRTDT</td>
</tr>
<tr>
<td>2</td>
<td>RDTOM</td>
</tr>
<tr>
<td>3</td>
<td>RDITU</td>
</tr>
<tr>
<td>4</td>
<td>PLUTTM</td>
</tr>
</tbody>
</table>

**False Sequencing Problem.** It is evident that a DFTS preserves much structural information of the original XQP, such as the sibling ordering and ancestor-descendant ordering. That is, a node always appears before its descendants and its right siblings in the DFTS. However, DFTS representation does not preserve the complete structure information of an XQP. For example, if a node \( B \) appears before a node \( A \) in a certain DFTS, we cannot judge whether \( B \) is an ancestor or a left sibling of \( A \). Thus, a DFTS cannot uniquely represent an XQP, and two different XQPs may have the same DFTS. Figure 5(1) and 5(2) are apparently two different tree structures, whereas they have the same DFTS. A ramification from this problem is that the tree structure cannot be reconstructed from its DFTS sequence. For example, the DFTS w.r.t. the XQP shown in Figure 5(3) is “PRRTD”, we cannot determine which node among “P”, “R”, “R”, and “D” is the parent of “T”.

To solve the above false sequencing problem, we propose a unique sequence representation method under the depth-first traversing framework. The unique sequence representation of an XQP using our method is denoted by UDFTS. Given an XQP, let its DFTS be \((L_1, L_2, \ldots, L_n)\), its UDFTS is defined as \((L_1(P_1), L_2(P_2), \ldots, L_n(P_n))\), where \( P_i \) is a code used to uniquely identify the parent of \( L_i \) (i.e., \( P_i \) is a unique code of \( L_i \’s \) parent). In SOLARIA, the code of \( L_1 \) is defined as 1 and the code of \( L_i \) is determined by the pre-order coding scheme under the depth-first traversing framework. Thus, the code of \( L_i \’s \) parent (i.e., \( P_i \)) is defined as \(-1\). Figure 5 shows the UDFTSs of the corresponding XQPs.

![Figure 5: XQP sequencing in DFTS and UDFTS](image)

Because a UDFTS preserves both the parent-child relationships and sibling ordering information, it uniquely represents an XQP and can be used to reconstruct the tree structure of the XQP. This sequencing method also facilitates once-scan index establishment for efficient mining. Figure 6 shows the facilitation of parent-child checking in SOLARIA.

![Figure 6: Parent-child checking in SOLARIA](image)

### 4.2 Efficient Sequence Mining with Parent-Child Relationship Constraint

Using the above sequencing method, we can convert the set of XQPs into a database of UDFTS sequences, following we will discuss how to mine frequent XML query patterns from UDFTS database. As we know, traditional frequent subtree mining approaches such as FastXMiner [25] typically follow the straightforward generate-and-test strategy. The algorithms with this strategy usually generate a large number of candidate subtree structures and need to perform a lot of costly subtree containment testing. To avoid the generate-and-test paradigm and reduce the costly subtree containment testing, we exploit the BI-Directional parent-child checking scheme to find the frequent XSQPs, which checks the parent-child relationship in sequences by both forward and backward strategies. Assisted with the parent-child information embedded in UDFTS sequences, we have rich
evidences to ensure the process of parent-child checking is efficient and linear with the size of query patterns. Obviously, parent-child checking is much cheaper than containment testing of tree structure data. That’s the key why we exploit sequence mining in SOLARIA.

Assume there is a lexicographical ordering \( \leq \) among the set of labels in the input sequence database, conceptually the complete search space of sequence mining forms a sequence tree with the root of XQPs. The process of constructing the sequence tree can be as follows. The root node of the tree is the root of XQPs, recursively we extend a node \( N \) at level \( L \) in the tree by adding one item (i.e., one XML element or attribute) to get a child node at the next level \( L+1 \) and the children of a node \( N \) are generated and arranged according to the chosen lexicographical ordering. In Figure 7, each node contains a frequent sequence and its corresponding absolute support, and the sequences in the dotted ellipses are non-closed ones. As an assumption, the absolute support, and the sequences in the dotted ellipses are non-closed ones. As an assumption, the absolute support is 2, and the chosen lexicographical order is \( I = \{P,R,T,D,M,U\} \).

Apparent, not all the frequent sequences in Figure 7 correspond to valid tree structures. For instance, “PM” is not a valid frequent sequence though with frequency of 2, because “P” is not the parent of “M”. This problem can be easily solved by parent-child checking scheme. Following we will briefly introduce the frequent sequence enumeration method, and discuss how to push the parent-child relationship constraint into the sequence enumeration in order to make sure that each mined frequent sequence corresponds to a valid subtree.

![Figure 7: The lexicographic frequent sequence tree in our running example](image)

### 4.2.1 Frequent Sequence Enumeration

For a given sequence database, many previous frequent pattern mining algorithms have elaborated that depth-first searching is more efficient in mining long patterns than breadth-first searching. Thus, in our approach we traverse our XQPs in depth-first order. [16] introduces an efficient pseudo-projection method for enumerating frequent sequences. In SOLARIA, a similar pseudo-projection method is adopted in order to reduce space complexity. A certain node in the sequence tree is always treated as a prefix sequence. By adding one item in I, we can find all the children of each prefix sequence. With respect to the corresponding prefix sequence, some items are not locally frequent. As our goal is mining frequent sequences, according to downward closure property [1], we only need to grow a prefix sequence using the set of its locally frequent items. To our best knowledge, in order to establish the locally frequent items w.r.t a certain prefix, we use a well-known method which is to build the projected database for the prefix and scan it to count the items.

### Definition 1: (Projected sequence of a prefix sequence)

Given an input sequence \( S \) which contains a prefix i-sequence \( e_1e_2...e_i \), (Here we ignore the parent code w.r.t. each item), the remaining part of \( S \) after we remove the first instance of the prefix i-sequence \( e_1e_2...e_i \) in \( S \) is called the projected sequence w.r.t. prefix \( e_1e_2...e_i \) in \( S \). For example, the projected sequence of prefix sequence \( PR \) in sequence \( PRT \) is \( T \).

### Definition 2: (Projected database of a prefix sequence)

Given an input sequence database \( SDB \), the complete set of projected sequences in \( SDB \) w.r.t. a prefix sequence \( e_1e_2...e_i \) is called the projected database w.r.t. prefix \( e_1e_2...e_i \) in \( SDB \). For example, the projected database of prefix sequence \( PR \) in our example is (\( TD,T^* \)T) in which symbol * stands for wildcard.

After giving the definition of the projected database for a certain prefix sequence, the idea of pseudo-projection can be described as follows. Instead of physically constructing the projected database, we only need to record 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 \( Sp \) and count the items (This is the so-called Forward-extension step), we will find the locally frequent items w.r.t. prefix \( Sp \), which can be used to grow prefix \( Sp \) in order to get longer frequent prefix sequences. For example, if \( Sp \rightarrow PR \), the set of its locally frequent items is \( \{T:2,D:2\} \).

### 4.2.2 Parent-Child Checking Scheme in SOLARIA

In figure 7, we can get the set of frequent sequences, \{PR, PRT, PRD, PT, PDT, PD, PDT, PDD, PM, PU\}. However, not all of these sequences are interesting from the application point of view, because the users usually expect each mined frequent sequence corresponds to a connected subtree in each XQP, in which case the sequence is called a valid sequence. To meet this requirement, the items in a valid sequence should satisfy the parent-child relationship according to the following Theorem.

### Theorem 1: (Valid sequence)

Let \( S = (e_1, e_2, ..., e_n) \), where \( S \) is a subsequence of a certain UDFTS of an XQP. If \( \forall e_i \in \text{tag}(S) - e_1 \) \((1 < i \leq n)\), \( \exists e_j \) such that \( e_j \) is \( e_i \)'s parent \((1 \leq j < i)\), then \( S \) is a valid sequence w.r.t. the corresponding XQP.

Proof. We can prove it by contradiction. Suppose \( S \) is not a valid sequence, that is, it does not correspond to a connected tree structure, there must exist at least one item, \( e_i \) \((1 < i \leq n)\), for which we cannot find any \( j \) \((1 \leq j < i)\) such that \( e_j \) is \( e_i \)'s parent, which contradicts with the condition of the Theorem.

### Definition 3: (Valid local item)

Given a local item \( e \) w.r.t. a prefix sequence \( e_1e_2...e_i \), \( e \) is called a valid local item if \( \exists e_m \) such that \( e_m \) is \( e \)'s parent \((1 \leq m \leq i)\).

### Property 1: (Right most extension)

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

Proof. Because \( e \) is a local item w.r.t. prefix \( e_1e_2...e_i \), \( e \) appears after \( e_i \) and its code is larger than the code of \( e_i \); thus, the node corresponding to \( e \) must be a descendant of a node on the right most path of the subtree corresponding to prefix \( e_1e_2...e_i \), otherwise, according to the definition of a UDFTS sequence, the code of \( e \) must be smaller than the code of \( e_i \). This means the node
of e_m must be on the right most path or a descendant of the right most path of prefix e_1e_2…e_i. Because the code of e_m is not greater than the code of e_i, the node of e_m can only be on right most path.

**Theorem 2: (Valid local item checking)** Given any local item e w.r.t. a prefix sequence e_1e_2…e_i, if the code of e’s parent is not greater than the code of e_i, e must be a valid local item w.r.t. prefix e_1e_2…e_i.

Proof. As e is a local item of prefix e_1e_2…e_i, the node corresponding to e must be a descendant of a node on the right most path of the subtree corresponding to prefix e_1e_2…e_i. Because the code of e’s parent is not greater than the code of e_i, e’s parent must be on the right most path of prefix e_1e_2…e_i, thus, e must be a valid local item w.r.t. prefix e_1e_2…e_i.

Property 1 shows that the sequence extension framework in SOLARIA complies with the right most extension strategy adopted in [28, 25], which removes some redundancy in frequent subtree mining. According to Theorem 1, we only need to extend a prefix sequence with its locally valid frequent items in order to find the valid sequences, while Theorem 2 says that when checking if a local item e w.r.t. a prefix sequence e_1e_2…e_i, valid or not, we only need to compare the parent code of e and the code of e_i. If the parent code of e is not greater than the code of e_i, then e is a valid local item, otherwise e is invalid.

![Figure 8: An example of false locally frequent item](image)

For example, the two XQPs in Figure 8 can be converted to two sequences. Suppose “P(-1)D(1)” is the current prefix, where item P’s pre-order code is 1, item D’s is 2. As figure 8(2) shows, for local item T, its parent is node R with pre-order code 3, which is greater than D’s pre-order code, thus, T is an invalid item and cannot be used to grow prefix “P(-1)D(1)”.

Note that in order to uniquely represent a sub-tree structure, each frequent subsequence is also maintained in the UDFTS format. Because the right most extension to grow a prefix sequence preserves the depth-first traversing ordering, we can design the coding scheme for each item in the frequent sequence in such a way: the code of the i-th item in the sequence is i, and the parent code of the first item is -1. In this way, two frequent subsequences, (P(-1), R(1), Y(2)) and (P(-1), R(1), Y(1)), will map to two different XSQPs, as the ‘Y’ item has different parents in the two UDFTSs. In other words, even if two items have the same label, if they have different parents, they will be treated as two different items. For example, item ‘Y’ is a valid local item w.r.t. prefix (P(-1), R(1)) in both Figure 9(1) and Figure 9(2), but they lead to two new different prefixes as shown in Figure 9(1) and Figure 9(2), and their absolute support is 1 instead of 2, which are all infrequent, thus, ‘Y’ cannot be used to extend prefix (P(-1), R(1)).

### 4.3 The Algorithm

By integrating the XML query pattern sequentialization method, frequent sequence enumeration, and the parent-child checking scheme, we derive the SOLARIA algorithm as shown in Figure 10, which avoids costly tree containment testing and prunes the unrelated search space efficiently under the local item’s validity checking. SOLARIA uses the sequencing method described in

![Figure 9: Frequency counting in SOLARIA](image)

Section 4.1 to convert the input XML query pattern set into a set of UDFTS sequences (line 2), and the pushing the parent-child relationship constraint (lines 7-9) into the frequent sequence enumeration process (lines 10-13).

**SOLARIA (XS, min_sup, C-P-index, FVS)**

**Input:** an input XML query pattern set XS, a minimum support min_sup, C-P-index is child-parent index

**Output:** frequent and valid sequences for query patterns. FVS.

**Begin**

1: FVS=∅;
2: SDB=UDFTS (XS);
3: Valid_Frequent_Sequences(SDB, ∅, min_sup, C-P-index, FVS);
4: return FVS;

**End**

**Valid_Frequent_Sequences**

**Input:** PS_SDB: a projected sequence database, PS: Prefix sequence min_sup: minimum support, C-P-index: child-parent index

**Output:** FVS: Frequent Sequences Set

**Begin**

5: if PS is non-empty
6: FVS = FVS △ PS;
7: Valid_LocalFreq_PS = Valid_locally_frequent_items(PS_SDB, PS, min_sup);
8: if Valid_LocalFreq_PS is empty
9: Return;
10: for each valid locally frequent item i
11: PS’ = <PS, i>;
12: PS_SDB’= pseudo projected database (PS’, PS_SDB);
13: Valid_Frequent_Sequences(PS_SDB’, PS’, min_sup, c-p-index, FVS);
**End**

**Figure 10: Algorithm SOLARIA**

### 4.4 Further Discussion: SOLARIA*

In our running example, we finally find the complete set of frequent patterns: {PR, PRT, PRD, PD, PDT, PDD}. The goal of frequent pattern mining in XML database is to optimize query evaluation by indexing and caching based on the results we got from frequent query pattern mining.

Looking into the results in Figure 11, we find that query pattern (1)
is included in (2), (3), and query pattern (4) is included in (3), (5) and (6). In addition, sup(PR) = sup(PRT) = sup(PRD) = 2 and sup(PD) = sup(PDT) = 4 hold, we can conclude that PR and PD are not “closed” in their sequence database. But for our motivation, the result {PRT, PRD, PDT, PDD} with {PR, PD}

![Figure 11: Mining results in our running example](image)

omitted still supplies complete information with compact size. Enlightened by this point, we capture the idea of closure checking for closed query pattern generation. We add BI-Directional Extension closure checking of BIDE [21] in SOLARIA to generate the closed query patterns which have no super sequences having the same support, and the so derived algorithm is called SOLARIA*. The detailed techniques of BIDE, one of the most efficient closed sequence mining algorithms so far, is described in [21]. Our SOLARIA* for mining closed patterns is based on BIDE. Because of BIDE’s high performance, we still get a linearly scalable algorithm for closed pattern mining, which also can be proved in the experiments. Note according to our XML frequent pattern mining task, we have a root node constraint in SOLARIA/SOLARIA* to generate frequent/closed sequences with the same start item equivalent to the root node in XQPs.

5. Performance Evaluation

In this section, we evaluate the performance of SOLARIA and demonstrate the efficiency and scalability of our approach in frequent XML query pattern mining. First we compared SOLARIA with XQPMiner[26] and FastXMiner[25] on different datasets with various min_sup values. XQPMiner is a global QPT schema guided enumeration mining algorithm for XQP mining. In FastXMiner, only valid candidate XSQPs are enumerated in mining process. So far, FastXMiner is the most efficient algorithm for XML frequent query pattern discovery in generate-and-test manner. The datasets we used are DBLP, Shakespeare, SigmodRecord and XMark. According to the DTDs of these four datasets, we add some “/” and “*” nodes to construct the XQPs as the input of our experiments. Different characteristics on XQPs are shown in Table 2. All the experiments are carried out on a computer with AMD 2.0GHz with 1G RAM by implementing in C++.

<table>
<thead>
<tr>
<th>Datasets</th>
<th># of XQP</th>
<th>Max.XQP.Len</th>
<th>Ave.Nodes.#</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>28563</td>
<td>8</td>
<td>6.2</td>
</tr>
<tr>
<td>Shakespeare</td>
<td>48620</td>
<td>9</td>
<td>7.4</td>
</tr>
<tr>
<td>SigmodRec.</td>
<td>12545</td>
<td>5</td>
<td>5.1</td>
</tr>
<tr>
<td>XMark</td>
<td>35162</td>
<td>11</td>
<td>8.3</td>
</tr>
</tbody>
</table>

Table 2: Characteristics of datasets used

We first compared the three algorithms by varying the minimum support. In this comparison, we chose 10,000 XQPs in each dataset as our experimental data. Figure 12 shows four sets of comparison results among SOLARIA, XQPMiner and FastXMiner. We can see SOLARIA outperforms XQPMiner and FastXMiner on each dataset.

![Figure 12: Comparison studies under various min_sup](image)
As well, the time needed for FastXMiner at support 0.2% is always a bit more than that at 3%. This is because with the decreasing of min_sup, the “straightforward generate-and-test” style mining algorithms always need to match an increasing number of frequent XQP candidates, while SOLARIA efficiently avoids redundant sequences testing by dynamic enumeration and pruning after parent-child constraint is applied.

We evaluated the scalability of SOLARIA by varying the number of XQPs in two datasets and fixing the min_sup at 2%. The testing data in experiment 2 is re-generated with max length of XQP from DBLP and SigmodRecord. Figure 13 shows the performance results for DBLP and SigmodRecord with varying number of XQPs. SOLARIA has better scalability compared with XQPMiner and FastXMiner. On SigmodRecord, when the number of XQPs is one thousand, SOLARIA needs only about 20s while FastXMiner needs 75s and XQPMiner 270s. This further demonstrates the effectiveness of our sequence enumeration method which avoids additional expensive containment testing.

In addition, we also did experiments to identify the performance of the SOLARIA* algorithm on DBLP testing data with changing numbers of XQPs. Figure 14 shows that both SOLARIA* and SOLARIA have linear scalability in terms of the base size. When we have 1000 XQPs, the runtime of SOLARIA accounts for about 50% of the whole SOLARIA* execution time, but the response time is still satisfied and acceptable. Especially, when we are facing the problem about XML indexing and caching, SOLARIA* is a good choice for discovering more compact result set of hot structures from XML queries.

From the above experimental results we can see that SOLARIA is more efficient and has better scalability than FastXMiner, and the derived algorithm SOLARIA* still has satisfying efficiency for the discovery of frequent closed query patterns, which is useful in automatic and intelligent XML indexing and caching tasks.

6. Conclusions and Future Work
In this paper we present SOLARIA, an efficient algorithm for frequent XML query pattern mining which involves the idea of sequence mining for replacing expensive tree containment testing with cheap parent-child validity checking. The novel techniques proposed in SOLARIA include a unique sequence representation for both an XML query and an XML query pattern, and an efficient frequent sequence enumeration method preserving the XML query pattern’s structure information, such as the parent-child relationship and the sibling ordering constraint. Our algorithm has been validated to be effective and practical for frequent XML query pattern mining. The thorough experimental results give us rich confidence to believe that SOLARIA outperforms the previous algorithms in terms of efficiency as well as scalability. We also derived an algorithm, SOLARIA*, from SOLARIA serving for closed query pattern mining. With the idea of SOLARIA*, we are moving on investigating the automatic and intelligent indexing and caching mechanisms which are based on convincing results outputted by SOLARIA*.

7. References


[9] J. Han, G. Dong, Y. Yin, Efficient mining of partial periodic patterns in time series database. Proc. of the 18th Int. Conf. on Data Engineering, Mar. 1999, Sydney, Australia.

[10] J. Han, J. Pei, B. Mortazavi-Asl, et al. FreeSpan: Frequent pattern-projected sequential pattern mining. Proc. of the 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, Aug. 2000, Boston, MA, USA.


[21] J. Wang, J. Han. BIDE: Efficient Mining of Frequent Closed Sequences. Proc. of the 20th Int. Conf. on Data Engineering, Mar. 2004, Boston, MA, USA.


[24] X. Yan, J. Han, R. Afshar. CloSpan: Mining Closed Sequential Patterns in Large Databases. Proc. of the 3rd SIAM Int. Conf. on Data Mining, May 2003, San Francisco, CA, USA.


