Discovering Approximate Keys in XML Data

Gösta Grahne
Concordia University
grahne@cs.concordia.ca

Jianfei Zhu
Concordia University
j_zhu@cs.concordia.ca

ABSTRACT

Keys are very important in many aspects of data management, such as guiding query formulation, query optimization, indexing, etc. We consider the situation where an XML document does not come with key definitions, and we are interested in using data mining techniques to obtain a representation of the keys holding in a document. In order to have a compact representation of the set of keys holding in a document, we define a partial order on the set of all key expressions. This order is based on an analysis of the properties of absolute and relative keys for XML. Given the existence of the partial order, only a reduced set of key expressions need to be discovered.

Due to the semistructured nature of XML documents, it turns out to be useful to consider keys that hold in "almost" the whole document, that is, they are violated only in a small part of the document. To this end, the *support* and *confidence* of a key expression are also defined, and the concept of *approximate key expression* is introduced. We give an efficient algorithm to mine a reduced set of approximate keys from an XML document.

1. INTRODUCTION

Keys are important in many aspects of data management, such as query optimization, indexing, update anomaly prevention, and information preservation in data integration [2, 4, 15]. XML databases store data with partial structure. The data is integrated from various types of data sources. Since XML is becoming the universal format for (semi-)structured documents and data on the web, it is now widely accepted as a model of real world data. There are several proposals for integrity constraints for XML data [6, 5, 13, 7, 8, 9, 10].

In this paper we adopt the key definition of Buneman et al. [6]. Compared with other proposals for keys for XML, their definition can be reasoned about efficiently, and the scope of keys is considered. In the paper, Buneman et al also give a sound and complete axiomatization for logical implication of keys.

As an example of the keys in [6], consider the XML document in Figure 1. The document records information about articles in a journal. As in real life, we can expect that the *volume* and *number* identify an *issue* (*volume* and *number* are called *absolute key* of *issue*). We also can expect that the *position* of an author in an article identifies the author in a specified issue. This is an example of a *relative key*.

```
<issue>
  <volume>11</volume>
  <number>1</number>
  <articles>
    <article>
      <title>Data Mining</title>
      <authors>
        <author posi="00">Jim</author>
        <author posi="01">Tom</author>
        <author posi="02">Peter</author>
      </authors>
    </article>
    <article> ... </article>
    ...
  </articles>
</issue>
```

Figure 1: An example of XML document

Though the definition and the implication rules of the keys are given in [6], some questions still need to be considered. (1) : Given an XML dataset, since the data is integrated from various types of data sources, sometimes there are no clear keys in the data. (2) : According to the axiomatization in [6], sometimes the number of keys could be enormous. Then the question is, how to store the keys effectively? Most importantly (3) : Given an XML dataset, how to efficiently find the keys holding in it?

In this paper, we make the following contributions:

1. Based on an analysis of the inference rules in [6], we define a partial order on key expressions. We will see that only a minimal cover of the set of key expressions need to be mined and stored. Compared with the large set of key expressions that can be inferred by inference rules and will hold in the data, our way is more effective.

2. To measure the interestingness of a key expression, we define the *support* and *confidence* of a key expression. Thus, for some key expression, though it is not satisfied

CIKM'02, November 4–9, 2002, McLean, Virginia, USA.

in a small part of the XML document, it is satisfied in most of the data, and we consider it as a valid key in the data. As an example, in a document like the one in Figure 1, in every issue, there are many articles. More than 95% of the titles of articles can identify an issue. It's reasonable to think that the title of an article is a key.

3. An efficient algorithm for mining a reduced set of approximate keys in an XML document is given. Without the availability of a DTD, the "prefix tree" of the XML data is parsed out, and at the same time, some candidate key expressions will be produced. From these key expressions, an apriori-like algorithm will generate all other key expressions in the small set.

1.1 Related work

Many algorithms have been presented for the discovery of functional dependencies [17]. Approximate functional dependencies are considered in [16, 12]. In [16], Kivinen and Mannila define measures for the error of a dependency and derive bounds for discovering dependencies with errors. Huhtala et al. [12] present an algorithm called TANE, which is based on partitioning the set of rows with respect to their attribute values. None of the aforementioned papers deal explicitly with finding any sort of compact representation, or minimal cover of the keys holding in the data. Evidently, the question of a minimal cover is intimately related to the question of a sound and complete axiomatization of approximate keys. The work by Calders and Paredaens [18] is taking an important step in this direction.

Our definition of the approximate keys is based on the support and confidence of a key expression, which are similar to the definition of support and confidence of an association rule in [3]. Our algorithm for mining approximate keys is inspired by the apriori-like algorithm presented in [14] which discovers the prefix structure of semistructured data.

1.2 Paper organization

The rest of the paper is organized as follows. Section 2 reviews the definitions of key expressions in [6]. In Section 3, we introduce a partial order on the key expressions and discuss the reduced set of key expressions. Detailed algorithms and the experimental results are given in Section 4. In Section 5, we give the conclusions of this paper.

2. KEYS

In this paper, we adopt the definition of keys by Buneman et al [6], which we recall in the following.

An XML data document can be expressed as a node-labeled tree, where the labels are divided into three sets: $\mathbf{E}$ the set of *element tags*, $\mathbf{A}$ the a set of *attribute names*, and the singleton {S}, where S represents text (PCDATA).

Definition 1. An *XML tree* is formally a six-tuple $T = (r, V, lab, ele, att, val)$, where r is a unique and distinguished root node; V is the set of nodes of T; the function *lab* maps each node v in V to label in $\mathbf{E}$ (v is an element node) or in $\mathbf{A}$ (v is an attribute node) or to S (v is a text node); The components *ele* and *att* are partial mapping on V. For $v \in V$, $ele(v)$ is a *sequence* of elements in V and $att(v)$ is a set of attribute-nodes in V; Finally, *val* maps each maps each attribute and text node to a string.

Figure 2 shows an XML tree corresponding to an XML document, such as the one in Figure 1.

Definition 2. In a XML tree, two nodes n_1 and n_2 are *value equal*, written as $n_1 =_v n_2$, if and only if $(a) : lab(n_1) = lab(n_2)$, and (b) : if n_1 and n_2 are attribute or text nodes then $val(n_1) = val(n_2)$, or (c) : If n_1, and n_2 are element nodes, then for each a_1 in $att(n_1)$, we can find a_2 in $att(n_2)$ such that $a_1 =_v a_2$, and vice versa; and if $ele(n_1) = [v_1, \ldots, v_k]$, then $ele(n_2) = [v'_1, \ldots, v'_k]$, and for all $j \in [1, k]$, $v_j =_v v'_j$.

To define keys, Buneman et al [6] use *path languages* for identifying nodes in an XML tree. In this paper we will only use the more general one of their path languages, called PL. The syntax of expressions ρ in PL is:

$$\rho ::= \epsilon \mid \ell \mid \rho.\rho \mid *$$

where ϵ represents the empty path, ℓ is a label in $\mathbf{E} \cup \mathbf{A} \cup \{S\}$, the symbol "." is concatenation, and $*$ represents any (possibly empty) finite sequence of node labels.

A *valid* path in PL is the path expression such that for any label $\ell \in \rho$, if $\ell \in \mathbf{A}$ or $\ell = \{\mathrm{S}\}$, then ℓ is the last symbol in ρ. By counting the number of labels in ρ, we get the *length* of the path ρ (the empty path has length 0). The expression $n[\![P]\!]$ denotes the set of nodes in T that can be reached by following the path expression P from node n. The expression $[\![P]\!]$ is the abbreviation of $r[\![P]\!]$, where r is the root node of the XML tree. For two path expression P and Q in PL, P is said to be *contained* in Q, written as $P \subseteq Q$, if for any node n in any XML tree T, $n[\![P]\!] \subseteq n[\![Q]\!]$. For example, in Figure 2, $issue.articles.article.author \subseteq issue.*.author$.

Definition 3. A *key constraint* ϕ in an XML document is an expression $(Q, (Q', S))$ where Q is called the *context path*, Q' is called the *target path*, and $S = \{P_1, \ldots, P_k\}$, such that for each P_i, $Q.Q'.P_i$ is a valid path expression in PL. The paths $P_1, \ldots, P_k$ are called the *key paths* of ϕ. If $Q = \epsilon$, ϕ is called an *absolute key*, otherwise ϕ is called a *relative key*.

For example, in Figure 2, *(ϵ,(issue, {volume, number}))* is a key expression, it is also an absolute key. *(issue, (articles.article, {author}))* is another key expression, since here Q is not ϵ, it is a relative key.

Definition 4. Let $\phi = (Q, (Q', \{P_1, \ldots, P_k\}))$ be a key expression. An XML tree T *satisfies* ϕ, written $T \models \phi$, iff for any $n \in [\![Q]\!]$, given any two nodes $\{n_1, n_2\} \subseteq n[\![Q']\!]$, either $n_1 = n_2$ or there exists at least one path $\rho \in P_i$, and nodes $x \neq_v y$, such that $x \in n_1[\![\rho]\!]$ and $y \in n_2[\![\rho]\!]$. If we express satisfaction as a clause, we have

$$\forall n \in [\![Q]\!], \forall n_1 n_2 \in n[\![Q']\!] : \left(\bigvee_{1 \leq i \leq k} n_1[\![P_i]\!] \cap n_2[\![P_i]\!] = \emptyset \right) \vee n_1 = n_2$$

Figure 3 illustrates a key $(Q, (Q', \{P_1, \ldots, P_k\}))$. The key means that in a subtree rooted at each node n in $[\![Q]\!]$, if two nodes in $n[\![Q']\!]$ are distinct, then the two sets of nodes reached on some P_i must be disjoint.

As an example, in the XML tree T in Figure 2, T satisfies the absolute key *(ϵ,(issue, {volume, number}))*. The key means that each object rooted at an *issue*-branches is

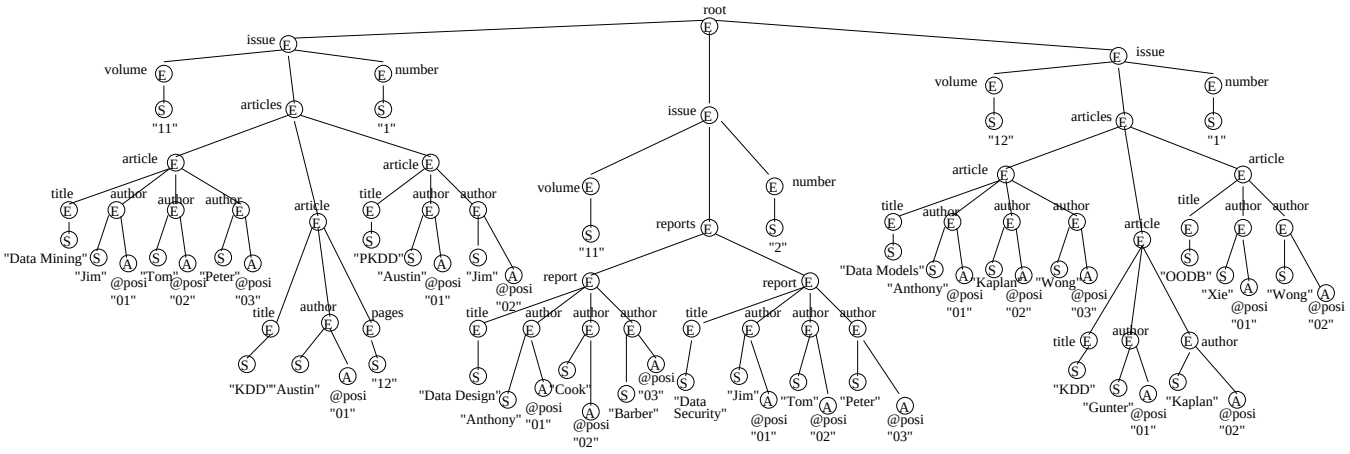

Figure 2: An XML tree constructed from XML data

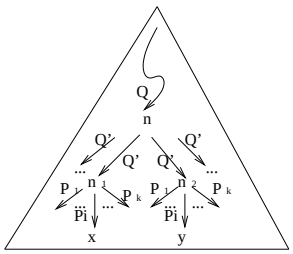

Figure 3: Illustration of a key $(Q, (Q', \{P_1, \ldots, P_k\}))$

uniquely identified by the value of *(volume,number)*. The XML-tree T also satisfies the relative key *(issue, (articles. article, {author}))*. This relative key means that inside an *issue*-object, an *article* is identified by an *author*.

3. MINIMAL COVER FOR A SET OF KEYS

Logical implication on keys is defined in the usual manner, i.e. let ϕ and ψ be key expressions. Then ϕ *logically implies* ψ, written $\phi \models \psi$, if every XML tree that satisfies ϕ also satisfies ψ. A set K of key expressions logically implies a key expression ψ, written $K \models \psi$, if every XML tree that satisfies all key expressions in K also satisfies ψ.

Let K be a set of key expressions. We denote by K^+ the set of all key expressions implied by K, that is the set $\{\psi : K \models \psi\}$. K^+ is also called the *closure* of K. Two sets K and L of key expressions are *equivalent* if $K^+ = L^+$. Obviously it makes sense to choose some sort of minimum cover to represent a set K of key expressions. In other words, we are looking for a set L, such that $L^+ = K^+$, and L that is "minimal", in the sense that it doesn't contain any key expression inferable from others.

To gain insight into the form of minimality we are looking for, we'll examine the sound and complete axiomatization given by Buneman et al [6]. The axiomatization consists of the inference rules in Table 1.

For a given finite set of key expressions K we can compute the closure K^+ starting from K, applying appropriate rules in Table 1 until no new keys can be derived.

It is proved in [6] that given a finite set K of keys and a key expression ϕ, we can determine whether $K \models \phi$ in quadratic time. We shall now call a key expression $\phi \in K$ *redundant* if $K \setminus \{\phi\} \models \phi$, in other words, if $(K \setminus \{\phi\})^+ = K^+$.

One approach to minimizing a set K of key expressions is to examine each element and test whether it is redundant or not. Redundant key expressions are removed. Notice that in this approach, the set K is given beforehand. In our scenario, only the XML tree T is given, we don't know what the keys are. The task is to mine T and determine a minimal set K, such that $T \models \phi$ if and only if $\phi \in K^+$. To see how to obtain a minimal set of key expressions, we first analyze the inference rules in Table 1.

- The *superkey rule* entails that if $(Q, (Q', S)) \in K$ then any expression $(Q, (Q', S'))$ where S' is a superset of S is redundant in K.

$\dfrac{(Q, (Q', S)), P \in PL}{(Q, (Q', S \cup \{P\}))}$	superkey
$\dfrac{(Q, (Q'.Q", \{P\}))}{(Q, (Q', \{Q".P\}))}$	subnodes
$\dfrac{(Q, (Q', S \cup \{P_i, P_j\})), P_i \subseteq P_j}{(Q, (Q', S \cup \{P_i\}))}$	containment-reduce
$\dfrac{(Q, (Q', S)), Q_1 \subseteq Q}{(Q_1, (Q', S))}$	context-path-containment
$\dfrac{(Q, (Q', S)), Q_2 \subseteq Q'}{(Q, (Q_2, S))}$	target-path-containment
$\dfrac{(Q, (Q".Q', S))}{(Q.Q", (Q', S))}$	context-target
$\dfrac{(Q, (Q', S \cup \{\epsilon, P\})), P' \in PL}{(Q, (Q', S \cup \{\epsilon, P.P'\}))}$	prefix-epsilon
$\dfrac{(Q_1, (Q_2, \{Q'.P_1, \ldots, Q'.P_k\})) \quad (Q_1.Q_2, (Q', \{P_1, \ldots, P_k\}))}{(Q_1, (Q_2.Q', \{P_1, \ldots, P_k\}))}$	interaction
$\dfrac{Q \in PL, S \text{ is a set of } PL \text{ expressions}}{(Q, (\epsilon, S))}$	epsilon

Table 1: Inference rules for key implication

- The *subnodes rule* indicates that if $(Q, (Q'.Q", \{P\})) \in K$, then K should not contain $(Q, (Q', \{Q".P\}))$. In other words, in a key expression in K the target path should be as long as possible. A similar rule, the *context-target rule*, indicates that any key expression in K should have a context path that is as short as possible.

- The *containment-reduce rule* seems not to be useful for minimizing a set of key expressions because of the superkey rule. However, it implies another rule: if $P_i \subseteq P_j$, then $(Q, (Q', S \cup \{P_i\}))$ is implied by $(Q, (Q', S \cup \{P_j\}))$. According to this rule, the key paths of S in $(Q, (Q', S))$ should be as general as possible. Similarly, from the *context-path-containment* and *target-path-containment* rules, we conclude that in an expression $(Q, (Q', S))$, the path expressions Q and Q' should be as general as possible.

- The *prefix-epsilon rule* says that when ϵ is one of the key paths, the other key paths should be as short as possible.

- The *interaction rule* allows us to shift a prefix Q', that is common to all key paths, to the target path Q_2, provided Q' is an absolute key under Q_1Q_2, with key-paths $P_1, \ldots, P_k$.

- The *epsilon rule* shows that if $Q.P$ is a valid path, then $(Q, (\epsilon, P))$ is a key. It's not very interesting to mine these types of keys, and therefore, in the minimal cover only key expressions $(Q, (Q', S))$, where $Q' \neq \epsilon$ will be kept.

Based on the above analysis, we define a partial order on key expressions as follows:

Definition 5. Given two key expressions $\phi = (Q_1, (Q_1', S_1))$ and $\psi = (Q_2, (Q_2', S_2))$ where $S_1 = \{P_1, \ldots, P_k\}$, $S_2 = \{P_1', \ldots, P_m'\}$, we say that ϕ precedes ψ, denoted $\phi \prec \psi$, if at least one of the following conditions is satisfied:

1. $k < m$, $Q_1 = Q_2$, $Q_1' = Q_2'$ and $S_1 \subseteq S_2$.
2. $k = m$, $Q_2 \subseteq Q_1$, $Q_2' \subseteq Q_1'$, for any $P_i \in S_1$, there exists a $P_j' \in S_2$ such that $P_j' \subseteq P_i$ and for any $P_j' \in S_2$, there exists $P_i \in S_1$ such that $P_j' \subseteq P_i$.
3. $k = m$, Q_1 is a prefix of Q_2, there exists a P, such that $Q_1.Q_1' = Q_2.Q_2'.P$ and $\{P.P_1, \ldots, P.P_k\} = \{P_1', \ldots, P_k'\}$
4. $k = m$, $Q_1 = Q_2$, $Q_1' = Q_2'$, there exists S and P, where $S = \{\epsilon, P_1", \ldots, P_{k-2}"\}$, S_1=$S \cup \{P\}$, and S_2=$S \cup \{P.P'\}$.

Lemma 1. The relation $\prec$ is a partial order on the set of all key expressions.

For example, relating to Figure 2, (*issue*, (*articles.article*, {*title*, *author*})) $\prec$ (*issue.articles*, (*article*, {*title*, *author*})) and (*issue*, (*.article*, {*, *author.posi*})) $\prec$ (*issue*, (*articles.article*, {*title*, *author.posi*}))

Obviously, if $\phi \prec \psi$, then ψ will not be kept in minimum cover. Based on the partial order $\prec$, an algorithm can be given for mining a minimal cover for the set of keys holding in a given XML tree T. However, remember that XML data is *semi-structured*, and that in typical applications the data is integrated from all types of data sources. Since every key in the data must be 100% satisfied, a key could contain many key paths. At the extreme case, the whole XML data tree is the only key. Such a key is not interesting. We therefore deem a key expression interesting if it is satisfied in a subset very close to T, and violated only in a very small part of T. As an example, in a document like the one in Figure 1, in every issue, there are many articles, and in 95% of the cases the titles of articles can identify an issue. It is therefore reasonable to say that the title of an article is a key for an issue. Another problem is that in some XML data, many expressions appear only a few times or even only once or twice. These expressions, though they are satisfied to a 100%, are not very interesting. For instance, in Figure 2, the path *issue.articles.article.pages* occurs in the tree only once, and though (*issue.articles.article.pages*, $\{\epsilon\}$)) is an absolute key according to the definition, it is not very interesting. To measure the interestingness and accuracy of a key expression with respect to a tree T, we define the *support* and *confidence* of the expression. In the definition, an expression $(Q, (Q', S))$ is called a *k-key expression* if there are k key paths in S.

Definition 6. Consider a k-key expression $\phi = (Q, (Q', \{P_1, P_2, \ldots, P_k\})$ and an XML tree T. Let $n \in [\![Q]\!]$, and $n[\![Q']\!] = \{n_1, n_2, \ldots, n_m\}$. Then we denote by $branches(n_j, P_i)$ the number of P_i-branches in the subtree rooted at n_j. We denote by $values(n_j, P_i)$ the number of distinct values of the subtree rooted at $n.n_j.P_i$. The *support* of ϕ in the subtree rooted at n is

$$support(n, \phi) = \sum_{j=1}^{m} \prod_{i=1}^{k} branches(n_j, P_i),$$

and the *confidence* of ϕ in the subtree rooted at n is

$$conf(n, \phi) = \frac{\sum_{j=1}^{m} \prod_{i=1}^{k} values(n_j, P_i)}{support(n, \phi)}$$

If $support(n, \phi) = 0$, we set $conf(n, \phi) = 1$.

The support of ϕ in the whole tree T is

$$support(T, \phi) = \sum_{m \in n[\![Q]\!]} support(m, \phi)$$

The confidence of ϕ in T is

$$conf(T, \phi) = min\{conf(m, \phi) : m \in n[\![Q]\!]\}$$

The above definitions of *support* and *confidence* considers the values of each branch $n.n_j.P_i$ as a bag, and the support of the branches $n.n_j.\{P_1, P_2, \ldots, P_k\}$ is the number of elements in the Cartesian product of all bags. There could be other different definitions for *support* and *confidence*, for example, the support of $n.n_j.\{P_1, P_2, \ldots, P_k\}$ could be the minimum number of elements of all bags. However, since XML essentially is an object-oriented model, where two distinct objects can have the same value, the support of $n.n_j.\{P_1, P_2, \ldots, P_k\}$ should be number of all possible combinations of values of $n.n_j.P_i$. Therefore bag product seems to be the most natural operation for defining support and confidence.

Figure 4 is an example for illustrating the support and confidence of key expression $(a, (b, \{c, d\}))$. There are three

branches for $a.b$. In the first branch, since there are 2 branches for $a.b.c$ and 2 branches for $a.b.d$, the support for $(a,(b,\{c,d\}))$ in the first branch is 4. The support for $(a,(b,\{c,d\}))$ in the second and the third branch are all 1. Thus the support for $(a,(b,\{c,d\}))$ in the whole tree is 6. In the first subtree of the root, the distinct values of $(a,(b,\{c,d\}))$ are $\{(0,2),(1,2)\}$, so the confidence is $2/5 = 40\%$. In the second subtree, the confidence is 100%. According to the definition, the confidence for $(a,(b,\{c,d\}))$ in the tree is 40%.

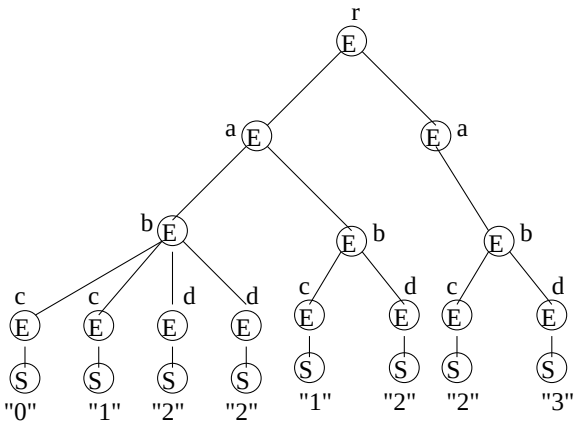

Figure 4: An example XML tree for illustrating support and confidence

A key is interesting if and only if its support is greater than a given threshold s and the key is *accurate* if its confidence is greater than a given threshold c. The threshold s is called *min-support*, and c is called *min-confidence*. If a key expression exceeds the thresholds, it is called an *approximate key*. Now the *valid* path expression needs to be redefined. A *valid* path in PL is the path expression such that for any label $\ell \in \rho$, if $\ell \in \mathbf{A}$ or $\ell = \{\mathrm{S}\}$, then ℓ is the last symbol in ρ, and the support of PL is greater than s.

Basically, support and confidence are two constraints for the key expressions to be approximate keys of XML data. The inference rules in Table 1 still hold, except for superkey, context-path containment and target-path containment. As an example of why these rules fail, we could have the following key expressions, listed with their supports and confidence:

1. $(\epsilon,(a.b.d,\{\epsilon\}))$ support: 5, confidence: 20%
2. $(\epsilon,(a.c.d,\{\epsilon\}))$ support: 20, confidence: 100%
3. $(\epsilon,(a.*.d,\{\epsilon\}))$ support: 25, confidence: 80%

Assume the min-support is 5 and min-confidence is 80%. According to the definition, key expression 1 is not an approximate key, while key expression 2 and 3 both are approximate keys. Notice that $a.b.d$ in expression 1 is contained in $a.*.d$, and according to the target-path-containment rule, if expression 3 is a key, expression 1 should also be a key. Thus, we call expression 3 a *fake* approximate key because now expression 3 can not derive expression 1, and therefore expression 3 should not be retained.

Formally, a k-key expression (Q_1, Q_1', S) is called a *fake key* if its support is greater than min-support and its confidence is greater than min-confidence, and there exists another expression (Q_2, Q_2', S) such that $Q_2 \subseteq Q_1$ and $Q_1' = Q_2'$, or $Q_2' \subseteq Q_1'$ and $Q_1 = Q_2$, and this expression has either support less than min-support or confidence less than min-confidence.

4. ALGORITHM

In [3], Agrawal et al. give the classical algorithm, *Apriori*, to mine all frequent item sets. The pruning rule used in *Apriori* is that if a (k-1)-item set is infrequent, then all its supersets are also infrequent. To mine a set of approximate key expressions, we still use the superkey rule as a pruning rule, even if it is not necessarily sound for approximate keys. Our superkey pruning rule says that if a k-key is found, this key will not be extended to a candidate (k+1)-key expression. Such a (k+1)-key expression could or could not hold, but we do not consider them interesting. Furthermore, another pruning rule is that if the support of a k-key expression is not greater than min-support, it will not be extended to a candidate (k+1)-key expression.

To simplify our algorithm, we use "?" to represent "*". The symbol "?" stands for any one node label, while "*" represents any (possibly empty) finite sequence of node labels. Thus, for any valid path expression in XML tree, if that path expression contains "*", we can replace "*" with 0 or more occurrences of "?". As an example, in Figure 2, path expression **.title* can be expressed as *?.?.?.title*.

4.1 Phase I: preparing seeds

4.1.1 Constructing a Prefix Tree

Assume that the DTD of the XML data does not exist, or is not available. By looking at the XML data document as a description of its XML tree in pre-order, we parse the XML tree, and generate its *prefix tree*. In the prefix tree, any two paths from the root to a leaf, but not including the data values, are different. Figure 5 shows the algorithm for constructing the prefix tree. In the algorithm, a *token* is a tag or the string between two tags.

A mapping table M is used to store data values of path expressions. For each distinct data value, we assign it a short identifier, thus saving space if the value occurs in many paths.

In the prefix tree every leaf represents a path expression with labels from root to the leaf. If a label appears more than once under its parent in the data, it will be marked with "$\star$". The marks will be used for generating relative key expressions. Finally, in the leaf, the support and the distinct values of the path are recorded, for later use in generating candidate 1-key expressions.

In the algorithm, the wild-card "?" is considered. For any path expression that contains "?", if "?" only represents one label in the tree then this "?" does not have to be considered. Thus in the end of the algorithm, some branches of the prefix tree are cut off. As an example, in Figure 6, there could be a path *root.issue.?.article.title*, since here the ? only represents "*articles*", it is cut off from the tree.

Figure 6 is a prefix tree for the corresponding XML data in Figure 2. Note that the support and distinct values for each leaf are not shown in the figure due to lack of space.

4.1.2 Generating absolute 1-keys

Among all the paths in the prefix tree, candidate 1-key expressions will be generated. Remember that in a key expression $(Q,(Q',S))$, Q should be as short as possible, Q'

INPUT: An XML data document
OUTPUT: The prefix tree Υ for the document
METHOD:
Create an empty tree, initialize the mapping table M
$currentPath == \{\epsilon\}$
for each token t read from the document
  if t is a start tag
    if t is a new tag of $currentPath$
      for each path P in Υ that is more general
      than $currentPath$
        follow P from root to the last node n
          insert t as the child of n
          if ? is not the child of n
            insert ? as the child of n
    In the data, if t appears more than once under $currentPath$
      mark t with "$\star$"
    $currentPath == currentPath.t$
  else if t is an end tag
    remove the last tag from $currentPath$
  else if t is a data value and t doesn't appear in M
    assign t an identifier VID
    for each path P that is more general that $currentPath$
      add the support of the $currentPath$ to the support of P
      if t is a new value
        insert VID to the leaf,
        add 1 to the number of distinct values
In the prefix tree, check the key expressions with "?". For each ?, if this ? only can be replaced by one particular label to make it valid, cut the branches corresponding to the expression with "?" in the tree.

Figure 5: Parsing XML data

should be as long as possible, and the number of elements in S should be as small as possible, so for each path, we generate a candidate 1-key expressions with both Q and S equal to $\{\epsilon\}$. Table 2 shows the support and confidence of absolute 1-key expressions corresponding to the branches of the prefix tree for the XML data in Figure 2. In the table, i in key expressions represents *issue*.

Suppose that the min-support is 3, and the min-confidence is 80%. In Table 2, many 1-key expressions are supported, while their confidences are less than min-confidence. The key expressions in rows 8 and 9 are infrequent though their confidences are 100%. This leaves us with the approximate key expressions in rows 4, 10, 12 and 13.

4.1.3 Generating candidate relative 1-key expressions

The relative keys can only exist in the branches marked with "$\star$". For relative keys, we use a pruning rule based on the "subnodes" inference rule. The pruning rule says that if an expression $(Q_1, (Q_1', \{P_1\}))$ is an absolute key, relative keys $(Q_2, (Q_2', \{P_2\}))$ such that $Q_1.Q_1'.P_1 = Q_2.Q_2'.P_2$ will not be generated. For instance, if $(\epsilon, (issue.articles.article.title, \{\epsilon\}))$ is an absolute key, though label *article* is marked with "$\star$", relative key $(issue.articles.article, (title, \{\epsilon\}))$ will not be generated.

Next we will look for candidate relative 1-keys. If $(\epsilon, (l_1 \ldots l_n, \{\epsilon\}))$ is an absolute 1-key we don't generate any relative 1-key candidates from it. Otherwise, if the expression is supported, but do not have enough confidence to be an absolute key, and if there is one or more labels l_{i_1}, ..., l_{i_k} marked with "$\star$", starting from l_{i_1}, we generate

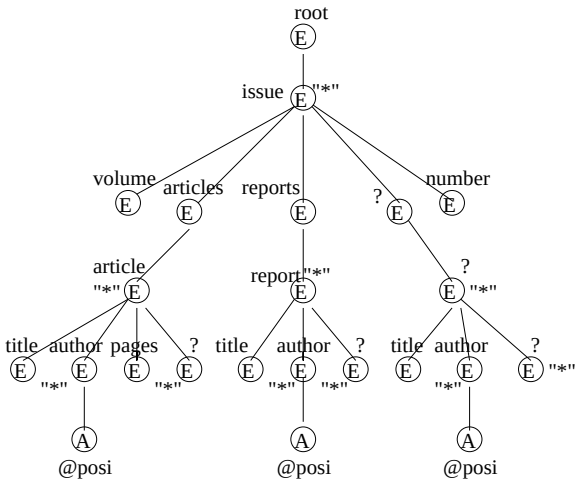

Figure 6: prefix tree of the XML data

	Key expressions	sup	conf
1	$(\epsilon, (i.volume, \{\epsilon\}))$	3	66.7%
2	$(\epsilon, (i.number, \{\epsilon\}))$	3	66.7%
3	$(\epsilon, (i.?, \{\epsilon\}))$	6	66.7%
4	$(\epsilon, (i.articles.article.title, \{\epsilon\}))$	6	83.3%
5	$(\epsilon, (i.articles.article.author, \{\epsilon\}))$	13	69.2%
6	$(\epsilon, (i.articles.article.author.posi, \{\epsilon\}))$	13	23%
7	$(\epsilon, (i.articles.article.?, \{\epsilon\}))$	20	75%
8	$(\epsilon, (i.articles.article.pages, \{\epsilon\}))$	1	100%
9	$(\epsilon, (i.reports.report.title, \{\epsilon\}))$	2	100%
10	$(\epsilon, (i.reports.report.author, \{\epsilon\}))$	6	83.3%
11	$(\epsilon, (i.reports.report.author.posi, \{\epsilon\}))$	6	50%
12	$(\epsilon, (i.reports.report.?, \{\epsilon\}))$	8	87.5%
13	$(\epsilon, (i.?.?.title), \{\epsilon\}))$	8	100%
14	$(\epsilon, (i.?.?.author), \{\epsilon\}))$	19	57.9%
15	$(\epsilon, (i.?.?.author.posi), \{\epsilon\}))$	19	15.8%

Table 2: Absolute 1-key expressions from the prefix tree

candidate relative 1-keys $(l_1 \ldots l_{i_j}, (l_{i_j+1} \ldots l_n, \{\epsilon\}))$, where $j = 1, \ldots, k$. In these key expressions, if in label $l_{i_{k+1}}, \ldots, l_n$ there is no "?", we can conclude directly that $(l_1 \ldots l_{i_k}, (l_{i_{k+1}} \ldots l_n, \{\epsilon\}))$ has confidence 100% and is an approximate relative 1-key. Otherwise, we designate the expression as a candidate relative 1-key. For instance, the expression in the fifth row generates the candidate relative 1-keys $(issue, (articles.article.author, \{\epsilon\}))$, and $(issue.articles.article, (author, \{\epsilon\}))$. Furthermore, from the expression in row 1, we generate the candidate relative 1-key expression $(issue, (volume, \{\epsilon\}))$. Since the path *volume* does not contain the symbol "?" we conclude that it is an approximate relative 1-key.

4.1.4 Post-processing

For this and each subsequent pass, a post-processing does the following:

1. For each new k-key ϕ, remove all k-keys ψ where $\phi \prec \psi$.

2. Remove fake keys.

For instance, in our example, the expression in row 12 precedes the expression in row 4, so the latter will be removed from the reduced set.

4.2 Phase II: Mining absolute and relative k-keys

From pass one we can get some absolute 1-key expressions that are satisfied in the XML data with min-support and min-confidence, some supported absolute 1-key expressions and some candidate relative 1-key expressions.

4.2.1 Pass two: generating candidates

In pass two, first of all, we generate candidate absolute 2-key expressions from pairs of supported absolute key expressions. There are two rules for generating candidate absolute 2-key expressions.

1. A candidate can not be generated by two 1-key expressions such that one precedes the other by $\prec$. For example, in Table 2, since the expression in row 1 precedes the expression in row 3, the pair (1,3) will not generate any candidate.

2. To generate candidates, from two key expressions $(Q, (l_1.\cdots.l_k l_{k+1}.\cdots.l_m, \{\epsilon\}))$ and $(Q, (l_1.\cdots.l_k l'_{k+1}.\cdots.l'_p, \{\epsilon\}))$ we generate the candidate 2-key expressions $(Q, (l_1.\cdots.l_j, \{l_{j+1}.\cdots.l_m, l'_{j+1}.\cdots.l'_p\}))$, where $j = 1, \ldots, k$.

For instance, from the two supported 1-key expressions $(\epsilon, (a.b.c.d, \{\epsilon\}))$ and $(\epsilon, (a.b.c.e, \{\epsilon\}))$, we generate the candidate 2-key expressions $(\epsilon, (a.b.c, \{d, e\}))$, $(\epsilon, (a.b, \{c.d, c.e\}))$ and $(\epsilon, (a, \{b.c.d, b.c.e\}))$.

Obviously, the support and confidence for a 2-key expression cannot be calculated directly from the support and confidence of the 1-key expressions that generated it. We are not going to use a complicated match algorithm to count the support of a 2-key expression, as is done in [14, 11]. We notice that the support of $(Q, (Q', \{P_1, P_2\}))$ is the sum of the support of $(Q', \{P_1, P_2\})$ in the subtrees rooted at the nodes in $[\![Q]\!]$. For this we scan the data and for each $n \in [\![Q]\!]$ we calculate $support(n, (Q', \{P_1, P_2\}))$. At the same time, we map the data value of every expression in the data to the corresponding identifier in the mapping table.

The confidence for a relative key expression is the least confidence in all subtrees rooted at some $n \in [\![Q]\!]$. When we have reached n, we calculate the confidence of the expression in current subtree, and update the confidence of the expression in whole tree if necessary.

4.2.2 Mining k-keys

From the relative 1-key expressions with enough support and low confidence, some candidate relative 2-key expressions can be generated. The rule is the same as the rule for generating candidate absolute 2-key expressions.

To generate candidate absolute or relative k-keys from two (k-1)-key expressions with $k > 2$, we use the following Theorem.

Theorem 1. Let $k > 2$. A k-key expression $(Q, (Q', \{P_1, \ldots, P_{k-2}, P_{k-1}, P_k\}))$ can be supported and have confidence only if the (k-1)-key expressions $(Q, (Q', \{P_1, \ldots, P_{k-2}, P_k\}))$ and $(Q, (Q', \{P_1, \ldots, P_{k-2}, P_{k-1}\}))$ are candidate (k-1)-key expressions, and all m-key expressions $(Q, (Q', S))$, where $m < k$ and $S \subseteq \{P_1, \ldots, P_{k-2}, P_{k-1}, P_k\}$ are supported.

Proof sketch: First, any m-key expressions $(Q, (Q', S))$, where $m < k$ and $S \subseteq \{P_1, \ldots, P_{k-2}, P_{k-1}, P_k\}$, if it is not supported, it's not valid. From an invalid key expression, we will not generate any candidates. Second, noted that in pass two, from two 2-key expressions, by rule 2, all possible candidates are generated. When $k > 2$, from $(Q, (Q', \{P_1, \ldots, P_{k-2}, P_k\}))$ and $(Q, (Q', \{P_1, \ldots, P_{k-2}, P_{k-1}\}))$, only $(Q, (Q', \{P_1, \ldots, P_{k-2}, P_{k-1}, P_k\}))$ need to be generated. All other k-key expressions, for instance, $(Q, (Q", \{PP_1, \ldots, PP_{k-2}, PP_{k-1}, PP_k\}))$ where $Q" \subset Q'$ and $Q = Q"P$, will not be generated. They can be generated by other two supported (k-1)-key expressions with context path Q and target path $Q"$.

Thus, in the algorithm, when $k > 2$ any two (k-1)-key expressions can generate at most one k-key expression. By the superkey rule, if a k-key expression can be generated from two (k-1)-key expressions while one of the (k-1)-key expressions is already satisfied in the XML data, this k-key expression candidate will not be generated.

The calculation of the support and confidence of k-key expressions is done in the same way as in pass two.

The algorithm terminates when there are no more candidate absolute and relative keys generated.

4.3 Discovery from real datasets

To test if our algorithm gets the key expressions as expected, we applied the algorithm to the XML data at `http://www.acm.org/sigmod/record/xml/` [1]. The data contains 67 issues of SIGMOD Record and there are more than 1500 articles in these issues. The data in the file is similar to the data in Figure 1. We ran our program on a DELL PC with a CPU clock rate of 800MHz, 256 MB of main memory, on a Windows 2000 professional platform. The program is written in the C++ language. By selecting min-support as 67 and min-confidence as 90%, the running time was about 8 seconds, 5 absolute keys and 3 relative keys were found.

The absolute key expressions are:

$\phi_1 : (\epsilon, (issue, \{volume, number\}))$
$\phi_2 : (\epsilon, (issue.articles.article.title, \{\epsilon\}))$
$\phi_3 : (\epsilon, (issue, \{number, articles.article.initPage, articles.article.endPage\}))$
$\phi_4 : (\epsilon, (issue, \{volume, articles.article.?\}))$
$\phi_5 : (\epsilon, (issue.articles.article, \{?, authors.author\}))$

Key ϕ_1 tells us that in an issue, the volume and number form a key for the issue. Key ϕ_2 shows that a title is a key of an article, or a key of articles, or a key of an issue. The confidence for this key is 98%, which means that it's not a precise key. However, we have enough confidence to believe it is a key. Key ϕ_3 tells us that the issue number, the initpage and endpage of an article could be a key of issue. "?" appears in ϕ_4 and ϕ_5, in both expressions, "?" can be replaced with title, or initPage or endPage. Key ϕ_4 says that the volume of an issue and the title, or initPage or endPage of an article can identify an issue, and ϕ_5 says that the name of an author and the title, or initPage or endPage of an article can identify an article, articles or an issue.

The relative key expressions are:

$\psi_1 : (issue, (?, \{\epsilon\}))$
$\psi_2 : (issue, (articles.article.?, \{\epsilon\}))$
$\psi_3 : (issue.articles.article.authors.author, (position, \{\epsilon\}))$

All these relative keys have clear meaning, for instance, ψ_1 means that in an issue, either the volume or the number is unique while it's not unique in the whole data.

5. CONCLUSION

Based on the definition of absolute and relative keys, and their inference rules, we propose that for describing the keys that hold in an XML document only a minimal cover needs to be considered. Since XML data is semi-structured and usually integrated from heterogeneous sources, we cannot expect keys be satisfied at 100%. To overcome this we first define the support of a key expression. Considering that many keys are satisfied in the main part of the XML data, and not satisfied only in a small part of the data, we also define the confidence of a key expression. Only when the support of a key expression is greater than a given minimum support and the confidence is greater than a given minimum confidence, we consider it a key expression that is (approximately) satisfied in the XML data.

We give an algorithm for mining a reduced set of approximate keys with sufficient support and confidence. In the algorithm, the wild-card in key expressions is also considered. The effectiveness of the algorithm is then tested by real datasets. The results show that mining keys in XML documents is efficiently implementable and that the keys discovered make intuitive sense.

6. REFERENCES

[1] ACM SIGMOD Record: XML Version, http://www.acm.org/sigmod/record/xml/.

[2] S. Abiteboul, R. Hull and V. Vianu. *Foundations of databases*, Addison-Wesley, 1995.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Very Large Data Bases*, pages 487-499, Santiago, 1994.

[4] M. Arenas and L. Libkin. A normal form for XML documents, *Proceedings of the 21th Symposium on Principles of Database Systems (PODS'02)*, pages 85-96, 2002.

[5] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. *Extensive Markup Language (XML) 1.0.* World Wide Web Consortium (W3C), Feb. 1998. http://www.w3.org/TR/REC-xml.

[6] P. Buneman, S. Davidson, W. Fan, C. Hara, W. Tan. Reasoning about Keys for XML. In *8th International Workshop on Databases and Programming Languages (DBPL '01).*

[7] P. Buneman, W. Fan,J. Siméon, S. Weinstein. Constraints for semistructured data and XML. *SIGMOD Record*, 30(1):47-55, 2001.

[8] S. Davidson, Y. Chen and Y. Zheng. Technical report, Indexing Keys in Hierarchical Data, 2001.

[9] W. Fan, L. Libkin. On XML Integrity Constraints in the Presence of DTDs. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 114-125, Santa Barbara, California, May 2001.

[10] W. Fan, J. Siméon. Integrity Constraints for XML. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 23-34, Dallas, Texas, May 2000.

[11] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees, *Journal of the ACM*, 29(1):68-95, 1982.

[12] Y. Huhtala, J. Kivinen, P. Porkka and H. Toivonen. Efficient Discovery of Functional and Approximate Dependencies Using Partitions, *ICDE*, pages 392-401, 1998.

[13] A. Layman et al. XML-Data. W3C Note, Jan. 1998. http://www.w3.org/TR/1998/NOTE-XML-data.

[14] K. Wang, H. Liu. Discovering Typical Structures of Documents: A Road Map Approach. In *21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 146-154, 1998.

[15] P. Buneman, S. Khanna, K. Tajima, W. Tan, Archiving Scientific Data. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 1-12, 2002.

[16] J. Kivinen and H. Mannila Approximate dependency inference from relations. *Theoretical Computer Science*, 149:129-149, 1995.

[17] H. Mannila and K.-J. Räihä On the complexity of inferring functional dependencies. *Discrete Applied Mathematics*, 40:237-243, 1992.

[18] Calders T., Paredaens J. Axiomatization of frequent sets. In *Proceedings of the International Conference on Database Theory*, pages 204-218, London, 2001.