Discovering XML Keys and Foreign Keys in Queries

Martin Nečaský
Department of Software Engineering, Charles University in Prague, Czech Republic
martin.necasky@mff.cuni.cz

Irena Mlýnková
Department of Software Engineering, Charles University in Prague, Czech Republic
irena.mlynkova@mff.cuni.cz

ABSTRACT

The XML has undoubtedly become a standard for data representation and manipulation. But most of XML documents are still created without the respective description of their structure, i.e. an XML schema. In this paper, we further enhance current methods for automatic inferring of an XML schema with discovering keys and foreign keys. We do not consider sample XML data for discovery but a set of queries in XQuery and we show how constructs utilized in the queries can be used for the discovery.

Categories and Subject Descriptors

I.7.1 [**Document and Text Processing**]: Document and Text Editing – languages, document management

General Terms

Measurement, Algorithms

Keywords

DTD, XML Schema, schema inference, integrity constraints.

1. INTRODUCTION

Today, the XML [6] is currently a de-facto standard for data representation. To enable users to specify their own structure of XML documents, so-called *XML schema*, the W3C[1] has proposed two languages – DTD [6] and XML Schema [15, 4]. The former is directly part of the XML specification and due to its simplicity it is one of the most popular formats for schema specification. The other was proposed later, in reaction to the lack of constructs of DTD.

However, statistical analyses of real-world XML data show that a significant portion of XML documents (in particular, 52% [11] of randomly crawled or 7.4% [12] of semi-automatically collected[2]) still have no schema at all. What is more, XML Schema definitions (XSDs) are used even less (only for 0.09% [11] of randomly crawled or 38% [12] of semi-automatically collected XML documents) and even if they are used, they often (in 85% of cases [3]) define so-called *local tree grammars* [14], i.e. languages that can be defined using DTD as well. It is also common that even there is an XML schema, the XML documents are not valid against it. It is because XML documents evolve in time independently of the XML schema and the respective changes are not propagated back to the XML schema.

In reaction to this situation a new research area of automatic inference of an XML schema has opened. The key aim is to create an XML schema for a given sample set of XML documents that is neither too general, nor too restrictive. It means that the set of document instances of the inferred schema is not too broad in comparison with the provided set of sample data but it is not equivalent to the sample set.

Currently there are several proposals of respective algorithms such as [1, 9, 13, 16]. However, they focus only on inference of concise content models and they do not cover more advanced aspects of XML schema inference. Therefore, there is still a space for further improvements. In this paper we will focus on discovering keys and foreign keys that can be captured using XML Schema `unique`, `key`, `keyref` constructs.

As recent research shows, there are various situations when knowing XML keys and foreign keys helps. Keys and foreign keys are utilized, for example, to ensure data integrity in XML databases and during XML data exchange. Keys and foreign keys are also used for optimization of XML data storage, i.e. XML data normalization [2, 17], and XML query evaluation.

If keys and foreign keys are not specified explicitly in the XML schema, we can try to discover them. However, there is only a few of proposals dealing with XML key discovery [10, 17] and no proposal dealing with XML foreign key discovery at all to our best knowledge. The existing methods propose to discover keys in a sample set of XML documents. To make these methods as precise as possible, it is necessary to provide a large sample set. Hence, the methods can be inefficient because it can be necessary to process a lot of data. However, it is of course not possible to discover all keys precisely. The precision always depends on the provided sample data set. The less the sample data set is representative, the less the methods are precise. It would be therefore useful to discover keys not only in sample data sets but also in other available sources of information to increase the precision of the methods.

[1] `http://www.w3.org/`
[2] Data collected with the interference of a human operator.

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

Contribution. In this paper, we show that it is possible to discover keys and foreign keys in queries evaluated by the system. Such possibility has not been addressed yet to our best knowledge. The queries can be obtained for example from an XML database log. Similarly to the approach introduced in [17], our method is however limited by its precision. It is because our method is based on intuition of how query constructs are usually used which can be misleading in some cases. Our approach is more effective on large data because the amount of queries in such cases is significantly lower. To reach more precise results, a combination of our method with [17] could bring interesting results.

The paper is structured as follows: Section 2 overviews constructs for key/foreign key specification. Section 3 presents how query constructs can be utilized to discover XML key/foreign keys. Section 4 provides a scoring function for discovered keys and foreign keys. Finally, Section 5 provides conclusions and outlines possible future work.

2. KEYS AND FOREIGN KEYS

Keys and foreign keys are considered in DTD as well as XML Schema. DTD provides data types `ID` and `IDREF` that can be assigned to attributes. A value of an `ID` attribute must be unique among values of all `ID` attributes in the XML document. A value of an `IDREF` attribute must refer to a value of some `ID` attribute in the XML document.

XML Schema notably enhances DTD keys and foreign keys as depicted in Figure 1. A key is declared by a separate construct `key` in the declaration of *context elements*. The construct `selector` describes *target elements* of the key and `field` describes a *key element*. A key can have more key elements as well as attributes, each described by a separate `field`. We consider only one key element in this paper. Let `c` be a context element. The key specifies that each target element in `c` has a unique value of the key element. There can be two target elements in different context elements that have the same value of the key element.

EXAMPLE 1. *An example key is depicted in Figure 1. The key is declared in the declaration of* `project` *elements. Therefore,* `project` *elements are context elements of the key. The key specifies that in the context of a given* `project` *element, each* `member` *child element has a unique value of its* `mid` *child element. In other words, each* `member` *is identified by its* `mid` *in the scope of its* `project` *parent.*

XML Schema also offers the `unique` construct that can be used instead of `key`. If `key` is applied, each target element must have its key element. If `unique` is applied, the key element is optional. However, we do not consider `unique` further in this paper.

The `keyref` construct is similar to `key` and declares a foreign key. It references the corresponding key by the `refer` attribute. It has *target elements* described by `selector` and *foreign key element* described by `field`. Let `c` be a context element. The foreign key specifies that the foreign key element of each target element in `c` must be equal to the key element value of a target element of the referred key in `c`. In other words, it specifies that the target element of the foreign key references a target element of the key via the pair foreign key/key element in the context of `c`.

EXAMPLE 2. *An example foreign key is depicted in Figure 1. It is declared in the same context as the previous key and specifies that in the context of a given* `project` *element, each* `document` *child element references a* `member` *element via* `mid`.

```
<element name="project">
 <key name="MemberKey">
  <selector path="member"/>
  <field path="mid"/>
 </key>
 <keyref name="MemberKeyRef"
         refer="MemberKey">
  <selector path="document"/>
  <field path="mid"/>
 </keyref>
</element>
```

Figure 1: XML Schema key and foreign key

There have been several efforts to formalize XML keys such as [2, 7, 8]. In this paper, we utilize the formalism proposed in [7] that formalizes XML Schema keys. A key in this formalism is a construct

$$(C, P, \{L\})$$

where C, P and L are XPath paths without predicates that use only child and descendant axes. C is called *context path*, P *target path* and L *key path*. C can be omitted, i.e. we can write $(P, \{L\})$. This is equivalent to $(/, P, \{L\})$. If C is omitted we call the key *global key*. Otherwise, it is called *relative key*. The authors define keys with more key paths but it is not important for this work.

The key specifies the following condition. Let `c` be an element targeted by C and `p` and `p'` be two elements targeted by P from `c`. If the value targeted by L from `p` equals to the value targeted by L from `p'`, then `p` and `p'` are the same elements. In other words, no two different elements targeted by P from `c` can have the same value of L.

This formalism corresponds to XML Schema keys. C specifies context elements, P target elements (`selector`), and L key elements (`field`).

EXAMPLE 3. *A key* $(//\texttt{project}, \texttt{member}, \{\texttt{mid}\})$ *formalizes the XML Schema key depicted in Figure 1.*

Because [7] does not provide a formalism for foreign keys, we provide our own as follows. A foreign key is a construct

$$(C, (P_1, \{L_1\}) \Rightarrow (P_2, \{L_2\}))$$

where $(C, P_2, \{L_2\})$ is a key and P_1 and L_1 are XPath paths without predicates that use only child and descendant axes. C can be omitted as in the case of keys.

Let `c` be an element targeted by C and $\texttt{p}_1$ be a element targeted by P_1 from `c`. The foreign key specifies that there is an element $\texttt{p}_2$ targeted by P_2 from `c` such that the value targeted by L_1 from $\texttt{p}_1$ equals to the value targeted by L_2 from $\texttt{p}_2$. In other words, each element targeted by P_1 from `c` refers to an element targeted by P_2 from `c` via the pair L_1 and L_2.

A foreign key in this formalism corresponds to XML Schema foreign keys. $(C, P_2, \{L_2\})$ is the referenced key (`refer`), C specifies context elements, P_1 target elements (`selector`), and L_1 foreign key elements (`field`).

EXAMPLE 4. *A foreign key* $(//\texttt{project}, (\texttt{document}, \{\texttt{mid}\}) \Rightarrow (\texttt{member}, \{\texttt{mid}\}))$ *formalizes the XML Schema foreign key depicted in Figure 1.*

3. KEY AND FOREIGN KEY DISCOVERY

In this section, we show how we can utilize queries evaluated over XML documents to discover keys and foreign keys

```
01 for $e1 in P1              for $e1 in P1
02 return                     return
03   for $e2 in                 let $e2 :=
       P2[L2 = $e1/L1]            P2[L2 = $e1/L1]
04   return CR                  return CR
```

Figure 2: `for` and `let` join pattern

satisfied by the XML documents. Concretely, we concentrate on element/element joins in queries. As a query language, we consider XQuery [5]. We suppose that we have a log of queries evaluated by the system in some sufficiently long period of time.

Assume a query Q from the log that joins a sequence of elements S_1 targeted by a path P_1 with a sequence of elements S_2 targeted by a path P_2 on a condition $L_1 = L_2$. It means that Q joins an element e_1 from S_1 with an element e_2 from S_2 if e_1/L_1 equals to e_2/L_2. An example of such query is depicted in Figure 3. The query joins a sequence of elements targeted by a path `//employee` with a list of elements targeted by a path `//member` on a condition `eid = mid` at line `05`, i.e. an `//employee` element e is joined with a `//member` element m if `eid` of e equals to `mid` of m.

For the purposes of this paper, let us suppose that each join is done via a key/foreign key pair. On the base of this assumption, we infer from Q that L_1 is a key for elements in S_1 or L_2 is a key for elements in S_2 and the other is a foreign key referencing the key. From our example query in Figure 3, we can therefore infer (`//employee`, $\{eid\}$) or (`//member`, $\{mid\}$). We can also infer the respective foreign key, i.e. (`//member`, $\{mid\}$) $\Rightarrow$ (`//employee`, $\{eid\}$) or (`//employee`, $\{eid\}$) $\Rightarrow$ (`//member`, $\{mid\}$), respectively.

The problem is how to decide which of L_1 and L_2 is the key and which is the foreign key. In this section, we show that the structure of Q can help us with this decision. Let us first make the following observations:

(O1) Assume that an element e_1 from S_1 can be joined with more elements from S_2. It means that there can be more different elements in S_2 having their value of L_2 equal to e_1/L_1. In other words, there can be more different elements in S_2 having the same value of L_2. Therefore, we infer that L_2 can not be a key of the elements in S_2. Moreover, because we suppose that one of L_1 and L_2 is a key and L_2 can not be a key, we infer that L_1 is a key of the elements in S_1. Consequently, we infer that L_2 is a foreign key referring L_1.

(O2) Vice versa, assume that e_1 can be joined with maximally one element from S_2. It means that there is maximally one element in S_2 having its value of L_2 equal to e_1/L_1. In that case we suppose that each element in S_2 has a unique value of L_2. Therefore, we infer that L_2 is a key of the elements in S_2 and consequently that L_1 is a foreign key referring L_2. We can not infer whether L_1 is a key or not.

Hence, we need to decide whether an element e_1 from S_1 can be joined with more (O1) or only one (O2) element from S_2. If we do not have any explicit information, there are two ways of how to decide:

```
01 for $e in //employee
02 return
03  <employee>
04   {$e/name}
05   {for $m in //member[mid=$e/eid]
06   return
07    <member>{$m/../code,$m/position}</member>}
08   {let $d := //document[mid=$e/eid]
09   return
10    <doccnt>{count($d)}</doccnt>
11    <docavgpages>{avg($d/pages)}</docavgpages>}
12  </employee>
```

Figure 3: Query with repeating join pattern

1. If we have a sample set of XML documents, we can decide by analyzing it. It means to count the numbers of elements from S_2 joined with each element from S_1.

2. We can decide by analyzing the query itself. It means to analyze the constructs used in the query and decide on the base of their usual usage.

In this paper, we concentrate on the second possibility. We do not try to cover all query constructs but only two special cases. We concentrate on queries that contain parts characterized by so called *join patterns* depicted in Figure 2. P_1, P_2, L_1 and L_2 in the patterns are XPath paths without predicates. Both patterns differ only at line `03` where the former applies `for` while the other applies `let`. The former pattern is called `for` *join pattern* and the other `let` *join pattern*. Line `01` is called *declaration clause*. Line `03` is called *join clause*. The path P_1 is called *source path*, P_2 *target path*, and the condition L_2 `= $e1/`L_1 *join condition*. C_R is called *return clause*. All paths P_1^R, ..., P_k^R in C_R starting with `$e2` are called *target return paths*.

We search a given set of queries for occurrences of the patterns. Pattern occurrences are localized as follows. We search for `let` and `for` clauses that have the form of a join clause (line `03`). Each such clause determines an occurrence of a `let` or `for` pattern. For this clause, we find the corresponding return clause and the `for` clause that declares the variable `$e1`. These clauses form the occurrence.

EXAMPLE 5. *For example, the query depicted in Figure 3 contains an occurrence of the `for` join pattern. It is localized on the base of the `for` clause at line `05` which is the join clause. Line `01` is the declaration clause and line `07` is the return clause. The return clause contains the following target return paths: $`m/../code` and $`m/position`. The source path is `//employee` and target path is `//member`.*

The query also contains an occurrence of the `let` join pattern. Line `01` is the declaration clause, line `08` is the join clause, and lines `10` and `11` form the return clause. It contains two target return paths: $`d` and $`d/pages`. The source path is `//employee` and target path is `//document`.

Each pattern occurrence is marked as *repeating* or *non–repeating*. An occurrence marked as repeating means that each element targeted by P_1 can be joined with more than one elements targeted by P_2. In contrary, a pattern occurrence marked as non–repeating means that each element targeted by P_1 can be joined with zero or one element targeted by P_2. In other words, the former means that the

observation (O1) can be applied while the other means that (O2) can be applied to infer keys and foreign keys.

In the following text we show how to decide whether a pattern occurrence should be marked as repeating or non–repeating. The decision is made on the base of constructs applied in the occurrence and is intuitive – it is based on common usage of constructs and as such it can not be precise. We therefore also assign a weight to each occurrence determining how sure we are about the inferred statement.

Assume a pattern occurrence π. The decision is made on the base of the following rules (R1) – (R5). Only one rule can be applied. We start with (R1). If (Ri) can not be applied, we try (Ri+1). First, we introduce the rules (R1) – (R3). If any of them can be applied, π is marked as *repeating* with the respective weight listed below.

(R1) π is an occurrence of the `for` pattern (weight: 1).

(R2) Aggregation function `avg`, `min`, `max` or `sum` is applied on a return path P_i^R in π (weight: 1).

(R3) Aggregation function `count` is applied on a return path P_i^R in π (weight: 0.75).

EXAMPLE 6. *To motivate (R1) – (R3) assume the query depicted in Figure 3. As we showed in Example 5, there is the* `for` *and* `let` *pattern occurrence. Let denote them π_1 and π_2, respectively.*

Assume first π_1. Because it is a `for` *pattern occurrence, (R1) is applied and π_1 is marked as repeating with weight* 1*. This corresponds to a common intuition. The* `//member` *elements joined to a given* `//employee` *element are iterated with the* `for` *clause at line* `05`*. Because it is very rare to apply* `for` *to iterate sequences with only one item, we suppose that there is more such* `//member` *elements.*

The other pattern occurrence is π_2. It is a `let` *pattern occurrence and (R1) can not therefore be applied. Because of the aggregation function* `avg` *applied on the return path* `$d/pages`*, (R2) is applied and π_2 is marked as repeating with weight* 1*. The motivation for (R2) is following. The applied aggregation function suggests that* `$d` *contains more elements than one because it is very rare to apply aggregation functions (excluding* `count`*) on sequences with only one item. In other words, there is more* `//document` *elements joined with an* `employee` *element at line* `08`*.*

Assume further that line `11` *is not present in the query. In that case only the aggregation function* `count` *is applied on* `$d` *at line* `10`*. Usually, we apply* `count` *on sequences with more than one item. However, there is not so small group of queries that use* `count` *for testing an existence of a node, i.e. testing if a sequence with not more than one item is empty or not. In that case we would mark π_2 as repeating but only with weight* 0.75*. This is a motivation for (R3).*

If (R1) – (R3) can not be applied, π is marked as *non–repeating*. The weight is assigned according to ($R4$) and ($R5$) listed bellow where k denotes the number of target return paths in π.

(R4) $k > 1$ (weight: 1)

(R5) $k <= 1$ (weight 0.5)

EXAMPLE 7. *(R4) and (R5) are motivated by the query depicted in Figure 4. There is an occurrence π_3 of the* `let` *pattern with the join clause at line* `05` *and declaration clause*

```
01 for $p in //project
02 return
03  <project>{$p/title,
04   for $m in $p/member
05   let $e := //employee[eid = $m/mid]
06   return
07    <member>
08     {$e/name,$e/salary,$m/position}
11    </member>
12  }</employee>
```

Figure 4: Query with non–repeating join pattern

at Line `04`*. The return clause at lines* `07` *–* `11` *contains the return paths* `$e/name` *and* `$e/salary`*. We can not apply (R1) – (R3) and π_3 is hence marked as non–repeating. Because there are two return paths, (R4) is applied and π_3 has assigned weight* 1*. This corresponds to a common intuition. If the result of a join path is neither iterated by a* `for` *clause nor an aggregation function is applied on the result, we can assume that* `$e` *contains one item. This is further supported by the structure of the return clause.*

For the current `$e`*, the query returns* `$e/name` *followed by* `$e/salary`*. If* `$e` *contained more elements, line* `08` *would return a list of* `name` *elements followed by a list of* `salary` *elements and it would not be possible to associate a given* `name` *element with the corresponding* `salary` *element. In that case it would be much better to use a* `for` *clause instead of* `let` *at line* `05`*. We suppose that* `let` *is applied only in a minority of such queries. This is a motivation for (R4).*

Assume that line `08` *returns only* `$e/name`*. In that case the application of* `let` *at line* `05` *can be meaningful even if* `$e` *contains more elements (the query returns a list of* `name` *elements). However, we suppose that it is still more frequent in these cases to apply* `for` *than* `let`*. If* `let` *is applied, we suppose that* `$e` *contains only one element. However, the assigned weight is lower (*0.5*). This is a motivation for (R5).*

For each pattern occurrence marked as repeating or non–repeating, we infer statements about keys and foreign keys on the base of the observations (O1) and (O2), respectively. Let w be the weight assigned to a pattern occurrence π. If π is marked as repeating, (O1) is applied and the following statements with weight w are inferred:

- $(P_2^{\downarrow}, \{L_2\})$ is not satisfied
- $(P_1^{\downarrow}, \{L_1\})$ is satisfied
- $(P_2^{\downarrow}, \{L_2\}) \Rightarrow (P_1^{\downarrow}, \{L_1\})$ is satisfied

where for a path P, $P^{\downarrow}$ denotes:

- P, if P does not start with a variable
- $P^{\$e\downarrow}/P'$ (or $P^{\$e\downarrow}//P'$), if P is `$e`$/P'$ (or `$e`$//P'$, respectively) where $P^{\$e}$ is a path used for the declaration of the variable `$e` in the query

If π is marked as non–repeating, (O2) is applied and the following statements with weight w are inferred:

- $(P_2^{\downarrow}, \{L_2\})$ is satisfied
- $(P_1^{\downarrow}, \{L_1\}) \Rightarrow (P_2^{\downarrow}, \{L_2\})$ is satisfied

EXAMPLE 8. *π_1 in Example 6 is marked as repeating with weight* 1*. We infer the following statements with weight* 1*:*

- (*//member*, {*mid*}) *is not satisfied*
- (*//employee*, {*eid*}) *is satisfied*
- (*//member*, {*mid*}) ⇒ (*//employee*, {*eid*}) *is satisfied*

π_2 *is also marked as repeating with weight* 1 *and we infer the following statements with weight* 1*:*

- (*//document*, {*mid*}) *is not satisfied*
- (*//employee*, {*eid*}) *is satisfied*
- (*//document*, {*mid*}) ⇒ (*//employee*, {*eid*}) *is satisfied*

EXAMPLE 9. π_3 *from Example 7 is marked as non–repeating with weight* 1*. We infer the following statements:*

- (*//employee*, {*eid*}) *is satisfied*
- (*//project/member*, {*mid*}) ⇒ (*//employee*, {*eid*}) *is satisfied*

If we had a schema of the data (user specified or inferred by some inference method), we could further determine that the path `//project/member` *targets the same elements as* `//member`*. Therefore, the inferred foreign key is the same as the first one inferred in the previous Example 8.*

The keys and foreign keys inferred from the previous examples were global. There can also be queries from which we can infer only relative keys and foreign keys.

```
01 for $p in //project
02 return <project>{$p/code,
03  for $m in $p/member
04  let $d := $p/document[mid=$m/mid]
05  return <member>{$m/mid}
            <d-cnt>{count($d)}</d-cnt></member>
06 </project>
```

Figure 5: Relative key inference

EXAMPLE 10. *Assume the query depicted in Figure 5. There is an occurrence* π_4 *of the* `let` *pattern with a join clause at line 04. It is marked as non–repeating with weight* 0.5 *according to the rule (R5). The source path* $`$p/member`$ *as well as target path* $`$p/document`$ *start with the variable* $`$p`$*. In other words, the join is done in the context of the ancestor element* `//project` *currently assigned to* $`$p`$*.*

As the example demonstrates, if the source and target path start at the same variable `$x`, we can infer the corresponding keys and foreign keys but relatively to the context specified by the path which declares `$x`.

This can be further generalized. Suppose that the source path P_1 starts with a variable `$x` and target path P_2 starts with a variable `$y`. If they are the same variable, we have the common context for P_1 and P_2. If not, we find the path P_1' that declares `$x` and path P_2' that declares `$y`. If any of them is an absolute path (i.e. it does not start with a variable), the P_1 and P_2 do not have a common context. Otherwise, the P_1' starts with a variable `$x'` and P_2' with a variable `$y'` and we continue recursively.

Formally, assume a pattern occurrence π with weight w. If the source and target path in π have a common context determined by a path C, we infer keys and foreign keys as follows. If π is marked as repeating, the following statements with weight w are inferred:

- $(C^{\downarrow}, P_2^{\downarrow C}, \{L_2\})$ is not satisfied
- $(C^{\downarrow}, P_1^{\downarrow C}, \{L_1\})$ is satisfied
- $(C^{\downarrow}, (P_2^{\downarrow C}, \{L_2\}) \Rightarrow (P_1^{\downarrow C}, \{L_1\}))$ is satisfied

where for paths C and P, $P^{\downarrow C}$ denotes:

- P, if P does not start with a variable
- P' (or $.//P'$), if P is `$e/`P' (or `$e//`P', respectively) and `$e` is declared by C
- $P^{\$e\downarrow C}$`/`$P'$ (or $P^{\$e\downarrow C}$`//`$P'$), if P is `$e/`P' (or `$e//`P', respectively) and `$e` is not declared by C

If π is marked as non–repeating, the following statements with weight w are inferred:

- $(C^{\downarrow}, P_2^{\downarrow C}, \{L_2\})$ is satisfied
- $(C^{\downarrow}, (P_1^{\downarrow C}, \{L_1\}) \Rightarrow (P_2^{\downarrow C}, \{L_2\}))$ is satisfied

EXAMPLE 11. π_4 *from Example 10 is marked as repeating with weight* 0.5*. Because its source path* $`$p/member`$ *and target path* $`$p/document`$ *start with the same variable* $`$p`$*, they have a common context determined by the path* `//project` *that declares the variable* $`$p`$*. Therefore, we infer the following statements with weight* 0.5*:*

- (*//project*, *document*, {*mid*}) *is not satisfied*
- (*//project*, *member*, {*mid*}) *is satisfied*
- (*//project*, (*document*, {*mid*}) ⇒ (*member*, {*mid*})) *is satisfied*

4. COUNTER EXAMPLES

The rules (R1) – (R5) are based on common intuition which can be sometimes misleading. We show some examples of queries which lead to bad results in this section. On the other hand, we observe in real scenarios that such queries are not so frequent. The scoring function presented in the next section utilizes this observation to minimize the influence of the queries that do not fit common intuition.

```
01 for $e in //employee
02 return <employee>{$e/name,
03  for $m in //member[mid=$e/boss/eid]
04  return
05   <member>{$m/../code,$m/position}</member>
06 }</employee>
```

Figure 6: Counter example – join is not done via a key/foreign key pair

EXAMPLE 12. *The query depicted in Figure 6 joins each employee with a list of project memberships of his/her boss (if there is any). The join is performed at line 03 which is the join clause of a* `for` *pattern occurrence. Because of (R1), the occurrence is marked as repeating with weight* 1*. Hence, we infer the following statements with weight* 1*:*

- (*//member*, {*mid*}) *is not satisfied*
- (*//employee*, {*boss/eid*}) *is satisfied*
- (*//member*, {*mid*}) ⇒ (*//employee*, {*boss/eid*}) *is satisfied*

```
01 for $e in //employee
02 return <employee>{$e/name,
03  let $m := //member[mid=$e/eid]
04  return {$m/../code}
05 }</employee>
```

Figure 7: Counter example to non–repeating join pattern

Because there can be two different employees with the same boss, `boss/eid` *can not be a key and the statements are therefore wrong. The query does not correspond to our assumption that joins are done via key/foreign key pairs – both* `mid` *and* `boss/eid` *are not keys. However, we observe that in majority of cases, joins are done via key/foreign pairs.*

EXAMPLE 13. *The query depicted in Figure 7 does not correspond to our assumptions as well. It returns for each employee a list of codes of the projects the employee participates in. It contains an occurrence of the* `let` *pattern with the join clause at line* `03`*. On the base of (R5), the pattern occurrence is marked as non–repeating with weight* 0.5*. We infer the following statements with weight* 0.5*:*

- (`//member`, {`mid`}) *is satisfied*
- (`//employee`, {`boss/eid`}) ⇒ (`//member`, {`mid`}) *is satisfied*

The statements are wrong because an employee can be a member of more projects and there can therefore be more different `//member` *elements with the same value of* `mid`*.*

5. SCORING FUNCTION

As we demonstrated in the previous section, there can be queries that do not correspond to our intuition summarized by the rules (R1) – (R5) and by the assumption that joins are done via key/foreign key pairs. On the other hand, we observe that majority of queries correspond to this intuition. Moreover, statements inferred from a misleading query, like the one depicted in Figure 6, can be negated by statements inferred from another query as the following example shows.

```
01 for $e in //employee
02 return <employee>{$e/name,
03  for $i in //employee[boss/eid = $e/eid]
04  return <inferior>{$i/name}</inferior>
05 }</employee>
```

Figure 8: Negation of example from Figure 6

EXAMPLE 14. *The query depicted in Figure 8 contains an occurrence of the* `for` *join pattern with a join clause at line* `03`*. It joins each employee with his/her inferior employees. On the base of (R1), the occurrence is marked as repeating with weight* 1 *and the following statements are inferred:*

- (`//employee`, {`boss/eid`}) *is not satisfied*
- (`//employee`, {`eid`}) *is satisfied*
- (`//employee`, {`boss/eid`}) ⇒ (`//employee`, {`eid`}) *is satisfied*

The first statement negates key (`//employee`, {`boss/eid`}) *inferred from the query depicted in Figure 6.*

These observations are utilized by the key scoring function explained in this section. We take a log with queries evaluated by the system in some sufficiently long time period (for example a month log). For each query Q in the log, we find occurrences of the `let` and `for` join pattern. For each such occurrence we infer the corresponding statements about keys and foreign keys as we described in the previous sections. If a new key K is going to be inferred, it has assigned an initial score 0. Each inferred positive statement about K increases the score of K by the weight of the statement. Respectively, each negative statement about K decreases the score by the respective weight. The resulting score therefore summarizes the weights of all inferred statements about K. A positive score means that K is probably satisfied while negative means that K is probably not satisfied. The higher the absolute value of the score is the higher the probability is.

Let $K_1, \ldots, K_n$ be the inferred keys. Let S_i be the score of K_i and N_i be the number of the inferred statements about K_i. We further enhance the precision of the scoring on the base of the following observation. Assume a key $K_i = (C, P, \{L\})$ and $K_j = (C', P, \{L\})$ where the path C targets ancestors of the elements targeted by C', i.e. the context specified by C covers the context specified by C'. In that case we also say that K_i *covers* K_j. For example, the context specified by `//company` covers the context specified by `//company/department`. It can be easily seen that if K_i is satisfied, K_j must be satisfied as well. On the other hand, if K_j is not satisfied, K_i can not be satisfied too. Therefore, if the score S_i of K_i is positive (i.e. K_i is satisfied), we increment S_j with S_i and N_j with N_i (i.e. K_j is satisfied as well). Conversely, if the score S_j of K_j is negative (i.e. K_j is not satisfied), we increment S_i with S_j and N_i with N_j (i.e. K_i can not be satisfied too).

Finally, the scores of the inferred keys are normalized to be comparable with each other. The normalized scores are from the range $\langle -1, 1\rangle$. The normalization takes into account not only the scores summarizing the weights of the statements about keys but also the number of the statements. The normalized score is computed as follows. Let S^{max} be the maximum from $|S_1|, \ldots, |S_n|$ and N^{max} be the maximum from $N_1, \ldots, N_n$. The normalized score S_i^{norm} of K_i is computed as follows:

$$S_i^{norm} = \frac{S_i}{S^{max}} * (1 - \frac{N^{max} - N_i}{\sum_{i=1}^{n} N_i})$$

The absolute value of the normalized score S_i^{norm} of the key K_i increases with the absolute value of S_i and also with the number of the inferred statements about K_i. This corresponds to common intuition – the more statements about the key is inferred the more we can be sure about whether the key is satisfied or not. Moreover, with each key we also get the list of all discovered foreign keys referencing the key. The foreign keys have the same score as the respective key with the same meaning of the absolute value of the score.

A user can also specify a threshold from the range $\langle 0, 1\rangle$. In that case we can provide him/her only with the keys having the absolute value of the normalized score above the threshold. The positive score means that the key is satisfied while the negative means that it is not satisfied. Therefore, we can return the list of keys that are satisfied and the list of keys that are not satisfied without the scores if we want provide the user with a simpler result.

Key	N_i	S_i	S_i^{norm}
$(//employee, \{eid\})$	8102	8102	1
$(//employee, \{boss/eid\})$	3486	−2970	−0.31
$(//member, \{mid\})$	2814	−1167	−0.12
$(//project, member, \{mid\})$	4082	3055	0.32
$(//document, \{mid\})$	5567	−4513	−0.51
$(//project, document, \{mid\})$	4082	−3055	−0.32

Table 1:

EXAMPLE 15. *Table 1 shows the normalized scores of the keys inferred from a query log containing the example queries presented in this paper. Each query occurs in the log repeatedly and each occurrence is analyzed independently. The second column shows the number of inferred statements about each key. The third column shows the non–normalized score of each key. The last column contains the normalized score. For the key* (`//document`, {`mid`})*, the number of statements about the key (column 2) and its non–normalized score (column 3) is increased by the respective values of the key* (`//project`, `document`, {`mid`}) *because the other is covered by the former global key (if the other can not be satisfied, the former can not be satisfied as well).*

As we can see, the largest number of statements was inferred about the first key. All these statements were positive and had the weight 1*. In that case the normalized score is* 1*. If all the statements were negative, the normalized score would be* −1*. In all other cases, the absolute value of the normalized score is lower than* 1*.*

Assume that the user specified a threshold 0.3*. The absolute value of the normalized score of* (`//member`, {`mid`}) *is lower. Therefore, we do not provide this key to the user.*

6. CONCLUSION

The aim of this paper was to enhance current XML schema inference methods with discovering XML keys and foreign keys. We introduced a novel approach based on analysis of XQuery queries. The output of our method is a list of scored keys and for each key a list foreign keys referencing the key. The score of a key can be negative or positive. A negative score means that the key is not specified by the XML documents while positive means that it is satisfied. The absolute value of the score means how sure we are about it. Because our method is based on intuition of how XQuery constructs are commonly applied in practice, it can be imprecise in certain cases. We believe that a combination of our method with some method discovering keys in sample data sets like [17] would increase the precision of both methods. Such a combination is a matter of our future research.

Currently, we are working on a deeper analysis of the introduced method. For the analysis, we are collecting real query logs and we are analyzing them. On the base of the results, we will further enhance the proposed rules for key and foreign key discovery. It will also be necessary to provide a normalization of XQuery queries to a unified shape since it is possible to express the same query with different XQuery expressions. Such normalization would enable to apply the defined rules more appropriately and accurately.

7. ACKNOWLEDGEMENT

This work was supported by the Ministry of Education of the Czech Republic (grant MSM0021620838).

8. REFERENCES

[1] H. Ahonen. *Generating Grammars for Structured Documents Using Grammatical Inference Methods*. Report A-1996-4, Dept. of Computer Science, University of Helsinki, 1996.

[2] M. Arenas and L. Libkin. A normal form for xml documents. *ACM Trans. Database Syst.*, 29:195–232, 2004.

[3] G. J. Bex, F. Neven, and J. V. den Bussche. DTDs versus XML Schema: a Practical Study. In *WebDB'04: Proc. of the 7th Int. Workshop on the Web and Databases*, pages 79–84, New York, NY, USA, 2004. ACM.

[4] P. V. Biron and A. Malhotra. *XML Schema Part 2: Datatypes (Second Edition)*. W3C, 2004. `http://www.w3.org/TR/xmlschema-2/`.

[5] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. *XQuery 1.0: An XML Query Language*. W3C, 2007. `http://www.w3.org/TR/2007/REC-xquery-20070123/`.

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. W3C, 2006. `http://www.w3.org/TR/2006/REC-xml-20060816/`.

[7] P. Buneman, S. B. Davidson, W. Fan, C. S. Hara, and W. C. Tan. Keys for xml. *Computer Networks*, 39(5):473–487, 2002.

[8] W. Fan and J. Siméon. Integrity constraints for xml. *J. Comput. Syst. Sci.*, 66(1):254–291, 2003.

[9] H. Fernau. Learning XML Grammars. In *MLDM'01: Proc. of the 2nd Int. Workshop on Machine Learning and Data Mining in Pattern Recognition*, pages 73–87, London, UK, 2001. Springer.

[10] G. Grahne and J. Zhu. Discovering approximate keys in xml data. In *CIKM*, pages 453–460. ACM, 2002.

[11] L. Mignet, D. Barbosa, and P. Veltri. The XML Web: a First Study. In *WWW'03: Proc. of the 12th Int. Conf. on World Wide Web, Volume 2*, pages 500–510, New York, NY, USA, 2003. ACM.

[12] I. Mlynkova, K. Toman, and J. Pokorny. Statistical Analysis of Real XML Data Collections. In *COMAD'06: Proc. of the 13th Int. Conf. on Management of Data*, pages 20–31, New Delhi, India, 2006. Tata McGraw-Hill.

[13] C.-H. Moh, E.-P. Lim, and W.-K. Ng. Re-engineering Structures from Web Documents. In *DL'00: Proc. of the 5th ACM Conf. on Digital Libraries*, pages 67–76, New York, NY, USA, 2000. ACM.

[14] M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages Using Formal Language Theory. *ACM Trans. Inter. Tech.*, 5(4):660–704, 2005.

[15] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures (Second Edition)*. W3C, 2004. `http://www.w3.org/TR/xmlschema-1/`.

[16] R. K. Wong and J. Sankey. *On Structural Inference for XML Data*. Technical Report UNSW-CSE-TR-0313, School of Computer Science, The University of New South Wales, 2003.

[17] C. Yu and H. V. Jagadish. Xml schema refinement through redundancy detection and normalization. *VLDB J.*, 17(2):203–223, 2008.