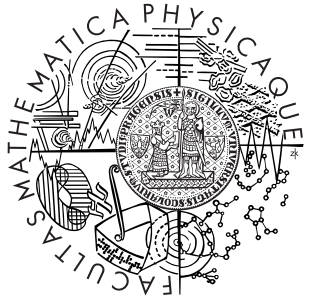
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Bc. Stanislav Fajt

Mining XML Integrity Constraints

Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Martin Nečaský, Ph.D.
Studijní program: Informatika, ISS

2010

I would like to thank to my supervisor Mgr. Martin Nečaský, Ph.D., for his helpful advices, corrections and suggested real data sets for testing. ...

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

V Praze dne 10.12.2010 Bc. Stanislav Fajt

Title: Mining XML Integrity Constraints
Author: Stanislav Fajt
Department: Department of Software Engineering
Supervisor: Mgr. Martin Nečaský, Ph.D.
Supervisor's e-mail address: martin.necasky@mff.cuni.cz

Abstract: The most important integrity constraints in XML are primary keys and foreign keys. In general, keys are essential in understanding both the structure and properties of data. They provide an instrument by which values from a given set of attributes uniquely identify tuples in a database. As a result, keys are important to main database operations. Since XML becomes lingua franca for data exchange on the web, it is widely accepted as a model of real world data. Because XML documents in general can appear in any semi-structured form, structural constraints (including keys) are often imposed on the data that are to be modified or processed These constraints are formally defined in a schema.Unfortunately, in spite of the obvious advantages, the presence of a schema is not mandatory and many XML documents are not joined with any. Consequently, no integrity constratins are specified in those documents, neither. This thesis is mainly focused on the inference of primary and foreign keys from XML documents.

Keywords: XML, XML Schema, schema inference, integrity constraints

Název práce: Mining XML Integrity Constraints
Autor: Stanislav Fajt
Katedra: Katedra Softwarového Inženýrství
Vedoucí diplomové práce: Mgr. Martin Nečaský, Ph.D.
E-mail vedoucího: martin.necasky@mff.cuni.cz

Abstrakt : Nejduležitějšími integritními omezeními jsou v XML primární a cizí klíče. Obecně vzato jsou klíče základním kamenem k pochopení struktury a vlastností dat. Nabízejí nástroj, s jehož pomocí lze jednoznačně identifikovat jednotlivé řádky tabulky pomocí hodnot z dané množiny atributů. Z toho plyne, že klíče jsou důležité pro provádění základních databázových operací. Od té doby, kdy se XML stalo jedním z nejpoužívanějších jazyků pro výměnu informací na internetu, je všeobecně přijímáno jako model pro reprezentaci skutečných dat. Protože XML dokumenty mohou v podstatě mít jakoukoli semistrukturovanou formu, jsou mnohdy během procesu zpracování či modifikace dat vyžadována strukturální omezení (například klíče). Tato omezení jsou definovaná ve schématu. I přes zjevné přínosy není bohužel přítomnost schématu povinná a k mnoha XML dokumentům není žádné schéma připojeno. Následkem toho nejsou pro tyto dokumenty specifikována ani žádná integritní omezení. Tato diplomová práce je zaměřená zejména na odvození primárních a cizích klíčů z XML dokumentů.

Klíčová slova: XML, XML Schema, odvození schématu, integritní omezení

Contents

Chapter 1

Introduction

1.1 Introduction

Keys are essential in understanding both the structure and properties of data. They provide an instrument by which values from a given set of attributes uniquely identify tuples in a database. As a result, keys are important to main database operations. For example, they enable us to guarantee within update that the process will affect only one tuple. More philosophically, if a tuple is considered as representing some real-world entity, the key can be the connection between the tuple and entity.

The conception of keys is closely interconnected with many settings, such as XML repositories, document collections or object databases. Identification of keys plays crucial role in many areas of modern data management, including data modeling, indexing, data integration, and query optimization. The knowledge of keys can be utilized to:

1. obtain better selectivity estimates in cost-based query optimization
2. automate the data-integration process
3. accelerate query processing using new access paths
4. improve the efficiency of data access through methods of physical design such as creation of indexes or data partitioning
5. provide new understanding of application data

Since the eXtensible Markup Language (XML)[2] becomes lingua franca for data exchange on the web, it is widely accepted as a model of real world data. Because XML documents in general can appear in any semi-structured form, structural constraints (including keys) are often imposed on data that are to be modified or processed. These constraints are formally specified in a schema. Two schema languages for XML have been proposed and these

are Document Type Definition (DTD)[2] and the XML Schema Definition (XSD)[4, 5].

Unfortunately, in spite of the obvious advantages, the presence of the schema is not mandatory and many XML documents are not joined with any. For instance, a recent study[8] have demonstrated that approximately half of the XML documents on the Internet are not accompanied by a schema. Thus, no integrity constraints are specified in those documents, neither. It should be noted that even when a schema is available, inference of keys could be useful. One such situation is schema cleaning: sometimes a schema is too general with respect to the document which is supposed to be described. In that case, it can be advantageous to derive some additional keys from solely XML document.

Based on the lack of schemas, many proposals of XML schema inference algorithms have appeared [19 - 27]. Nevertheless, none of these approaches of XML schema extraction are dealing with integrity constraints.

According to our best knowledge, the only studies considering XML key discovery are [39, 45]. The former one derives only primary keys from XML data in quite inefficient way. The experiments are there demonstrated only on small database containing only 8 instances of keys. The latter case represents a query-driven approach when both primary keys and foreign keys are extracted from XQuery[6] queries evaluated by the system. The queries can be obtained, for instance, from an XML database log, but in many cases only XML documents are at the disposal.

1.2 Main Contributions of the Thesis

In this thesis, the following contribution are presented.

- Based on the extensive research, all methods which are concerned with elicitation of primary or foreign keys are described in detail and their pros and cons are discussed. This analysis covers also approaches which are being used in the theory of relative databases. Furthermore, the most important algorithms from inference of inclusion dependencies are mentioned there, too (inclusion dependency is closely associated with foreign keys).

- The automated, efficient, data-driven algorithm for key inference is proposed. Its features are inference of primary/ foreign keys, absolute/ relative keys and even composite primary keys from a given XML document.

- The presented method is experimentally evaluated on synthetic as well as real data sets.

1.3 Organization of the Thesis

The remainder of the thesis is organized as follows. Second Chapter contains basic definitions and theory which are necessary for the contiguous chapters. In the Chapter 3 recent proposals of key discovery are presented. The next Chapter is the most important, it holds the proposed algorithm. Consequently, experimental results are shown in Chapter 5. Related work is described in Chapter 6. Finally, Chapter 7 consists of a conclusion and several suggestions for the future work.

Chapter 2

Preliminaries

2.1 XML Fundamentals

XML [2] standardized by the W3C consortium [1] is a simple text-based format for representing semistructured information: documents, books, data, transactions, configuration and much more. XML's design is derived from an older standard format called SGML (Standard Generalized Markup Language) [9] in order to achieve more simplicity, generality, and usability over the Internet. At present, XML serves as probably the most widely-used meta-format for sharing structured information via both locally and across networks. It becomes widely accepted model for real-world data representation.

2.1.1 XML Syntax

Markup and Content

By definition, an XML document consists of a string of characters and there may appear almost every legal Unicode character. These characters can be divided into two categories - *markup* and *content*. All strings considered as *markup* either start with the character `"<"` and end with `">"` or start with the character `"&"` and end with a `";"`. *Content* are those strings of characters which are not *markup*.

Tag

The *tag* is represented as a *markup* construct which starts with `"<"` and ends with `">"`. There are three types of *tags* - *start-tags*, *end-tags*, and *empty-element tags* which is not pairwise.

EXAMPLE 1 (Tags): *Examples of tags in Figure 2.1 can be, for instance,* `<Article>` *as a start-tag,* `</Article>` *as an end-tag, and*

```
<xml version ="1.0" encoding =" ISO -8859 -1" standalone =" no "?>
<Articles>
    <Article>
        <Title>OOP </Title>
        <Author posi = "13"> Tom </Author>
        <Release year = 2003 />
    </Article>
    <Article>
        <Title>UML </Title>
        <Author posi = "10"> Jeremy </Author>
        <Release year = 2005 />
    </Article>
</Articles>
```

Figure 2.1: Example of XML document

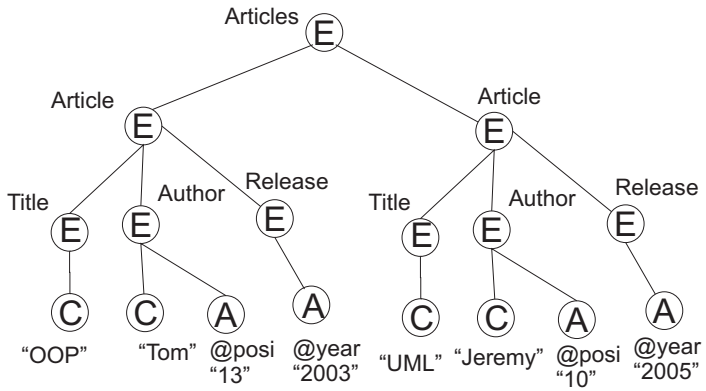


Figure 2.2: Example of XML tree

`<Release year = "2005"/>` *as an empty-element tag* `Release` *with an attribute* `year = "2005"`.

Attribute

An *attribute* is a notation for a *markup* consisting of a name/value pair that occurs within a *start-tag* or an *empty-element tag*.

Example 2 (Attribute): *For instance,* `posi = "10"` *is the attribute in the start-tag* `<Author>` *in the Figure 2.1.*

Element

An *element* is a part of an XML document which is either delimited by a *start-tag* and the corresponding *end-tag* or contains only an *empty-*

element tag. The characters appearing between the start- and end-tags are the *element's content* and may hold *markup*. Each *element* includes a type, is identified by a name and may have a *content* (except elements containing only *empty-element tag*) and a set of attributes.

Definition 1 (Child Element)**.** *A child element of the given element* e *is an element that occurs in the content of* e*. For the child element,* e *is a parent element or his ancestor.*

Definition 2 (Root Element)**.** *A root element or a document element is an element which is not included in a content of any other element in the document. An XML document may include only one root element.*

In other words, if an element's content holds other element, this one is so-called the *child element* and the beginning element is so-called a *parent element* or an *ancestor*. Moreover, an element which does not belong to any ancestor is named *root element*.

EXAMPLE 3 (Elements): *Examples of elements in Figure 2.1 are following,* `<Author>...</Author>` *as the element,* `Tom` *as his content. Moreover,* `<Author>...</Author>` *is the child element of* `<Article>...</Article>` *and* `<Articles>...</Articles>` *is the root element.*

Tree

The capability of elements to refer to other elements causes the inclusion in an XML document. Consequently, the document can be depicted as so-called *XML tree* where elements are represented as nodes and nesting of elments is graphed as edges between child nodes and their parents.

EXAMPLE 4 (XML Tree): *Figure 2.2 presents the* XML tree *which conforms the XML document showed in Figure 2.1. Attribute names are preceded with "@"for reasons of lucidity.*

Well-Formed XML Document

Definition 3 (Well-Formed XML Document)**.** *A document is an XML document if it is* well-formed *as it is defined in the XML specification.*

A *well-formed XML document* must satisfy at least the following criteria:

- Only one root element can exist in the document.
- Non-empty elements must be delimited by a start-tag and the corresponding end-tag. Empty elements may be denoted by an empty-element tag.
- Values of all attributes must be demarcated with apostrophes - a single apostrophe or a double apostrophe. However, pairs must be single/single or double/double. The opposite pair of apostrophes may occur in the value of the attribute.

```
<!ATTLIST Person posi ID #REQUIRED>
<!ATTLIST Author ref IDREF #REQUIRED>
```

Figure 2.3: Example of Primary and Foreign Keys in DTD

- Each (non-root) element must be included in the parent element as a whole element. In other words, child elements can not overlap.

Furthermore, an XML document is *valid* if it conforms to its associated *XML schema*.

2.2 XML Keys

2.2.1 Keys in DTD

DTD provides a mechanism to express primary keys as well as foreign keys. Nevertheless, it utilizes quite simple notation `ID`/`IDREF` defined by the XML specification itself [2]. A value of an `ID` attribute of the corresponding element (for example, named `person`) represents a primary key, which can be only *global*. The value must be unique among values of all other attributes in the document and all the `person` elements in the document must contain the defined `ID` attribute. Whereas `IDREF` attribute carries a value which must refer to a value of some `ID` attribute in the document.

Example 5 (Primary and Foreign Key in DTD): *In Figure 2.3 an example of a primary and a foreign keys in DTD is presented. Each* `Person` *is identified by the attribute* `posi` *and the element* `Author` *references* `Person` *via the attribute* `ref`*. Both cases occur in the scope of the whole XML document. Furthermore, each element* `Author` *must hold attribute* `ref` *and each element* `Person` *must contain attribute* `posi` *in the scope of the whole document.*

2.2.2 Keys in XSD

Analogous to DTD, XSD can express primary and foreign keys, too. Here, the key specification depends on XPath [6] and in addition, the specification allows to utilize different *context* from just the entire XML document (*context* or *context path* are explained in detail in the section 2.2.4). As a result, the notation of keys in XSD is notably enhanced in comparison with DTD.

Primary Key

The foundation of a key is created by a construct `key` in the declaration of *context elements*. The notation `selector` determines *target elements* of

the key and `field` establishes a *key element*. More *key elements* may be entered, each described by a separate `field`. Let `c` be a *context element*. The key declares that every *target element* in `c` contain a unique value in the *key element*. On the contrary to DTD, two *target elements* in different *context elements* having the same value of the *key element* can exist in terms of one key declaration.

```
<element name="Volume">
    <key name="AuthorPK">
        <selector path="./Authors/Author"/>
        <field path="@posi"/>
    </key>
    <keyref name="AuthorFK" refer="AuthorPK">
        <selector path="./Articles/Article/Author"/>
        <field path="@ref"/>
    </keyref>
</element>
```

Figure 2.4: Example of Primary and Foreign Key in XSD in reference to Figure 2.7

Example 6 (Primary Key in XSD): *A sample primary key in XSD is shown in Figure 2.4 (the* `key` *construct), it results from Figure 2.7. The key is stated in the declaration of the element* `Volume` *(the root element). Consequently, the element* `Volume` *is denoted as the* context element *of the key. Concretely, it is specified that each* `Authors.Author` *element in the scope of a given element* `Volume` *must hold a unique value in its* `posi` *attribute.*

Foreign Key

The declaration of a foreign key in XSD is similar to the key construct. It is created in the declaration of some *context element* and consists of *target elements* determined by `selector` and one or more *foreign key elements* described by the accordant number of `field`. Moreover, it contains an attribute `refer` which must have a value of the specified *key element*. 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. The foreign key declares that the *target element* of the foreign key references the *target element* of the key through the pair foreign key/key element in the *context* of `c`, which means the value equality between the *foreign key element* of each *target element* in `c` and *key element* of the *target element* of some referred key in `c`.

Example 7 (Foreign Key in XSD): *In the Figure 2.4 is possible to see an example of a foreign key (the* `keyref` *construct) in reference to Figure*

2.7. Like in the case of the primary key, it is again declared in the context of the element `Volume`*. The depicted construct determines that each* `Articles.Article.Author` *references* `Authors.Author` *via the attribute* `ref` *in the context of the element* `Volume`*.*

Unique

In the event of requirement of only the unique values of some element, the `unique` construct is offered by XSD. Syntax is the same as in the case of `key` notation. `unique` can be applied instead of a `key` when the user is not sure whether each *target element* will contain the *key element*. Nevertheless, the `unique` construct is not considered in this thesis.

2.2.3 Background for the XML Key Formalization

Since XSD has more expressive power than DTD, the formalism of keys will correspond to XSD. Many efforts to formalize XML keys has appeared e.g. [10-15]. In this thesis, the popular definition of keys by Bunemann et al.[11] is going to be adopted.

An XML document can be described as an *XML tree* (node-labeled tree) where the labels can be categorized into the following cases: `E` - *element tags*, `A` - *attribute names* and the singleton {S} representing text (PCDATA).

Definition 4 (XML Tree)**.** *An* XML tree *is formally a six-tuple* $T = (r, V, lab, ele, att, val)$*, where* r *stands for a unique root node,* V *is the set of nodes of* T, lab *is the function mapping each node* $v \in V$ *to labels from* `E` *(*v *is an element node) or from* `A` *(*v *is an attribute node) or to* S *(*v *is a text node). The components* ele *and* att *are partial mappings on* V*. Let* $v \in V$, $ele(v)$ *stands for a sequence of elements and* $att(v)$ *denotes a set of attribute-nodes. Then* val *maps attribute and text node to a string.*

Example 8 (XML Tree): *Figure 2.2 presents the* XML tree *which conforms the XML document showed in Figure 2.1. Text (PCDATA) is represented as* `C` *there.*

Definition 5 (Node Equivalence)**.** *In an XML tree, two nodes* n_1, n_2 *are* value equal*, noted as* $n_1 =_v n_2$*, iff (a):* $lab(n_1) = lab(n_2)$*, and (b): if* n_1, n_2 *are attribute or text nodes then* $val(n_1) = val(n_2)$ *or (c): if* n_1, n_2 *are element nodes, then* $\forall a_1 \in att(n_1)$*, it is possible to discover* $a_2 \in att(n_2)$ *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* $\forall j \in [1, \ldots, k], v_j =_v v_j'$.

To identify nodes in an XML tree, Buneman et al. utilize *path language*, called *PL* [11]. The syntax of an expression p in *PL* is following:

$$p ::= \varepsilon \mid l \mid p.p \mid \_ \mid *$$

Where ε stands for an empty path, l represents a label in $\mathtt{E} \cup \mathtt{A} \cup \{\mathrm{S}\}$, `"."` denotes concatenation, `"*"` is every (possibly empty) finite sequence of node labels, and `"_"` represents any node label.

Next, some helpful notations are presented:

- A path expression p is called a *valid* path, if for any $l \in p$ and $l \in \mathtt{A}$ or $l = \{\mathrm{S}\}$, l is the last symbol in p.
- The notation $n[P]$ corresponds to the set of nodes in T which is situated on the path expression P and starts from the node n. The set of nodes represented by $[P]$ starts in the root node of the XML tree.
- And let A, B be set of nodes. Then $A \cap_v B$ denotes the set of nodes containing the same values from both sets.
- Let P_i, P_j be two path expressions. Then $P_i \sqsubseteq P_j$ corresponds to the set of labels occuring in both path expressions.

2.2.4 Formalization of Primary Keys in XML

Definition 6 (Primary Key (PK))**.** *A* primary key *(or* key*)* ψ *in an XML document is an expression* $(\mathrm{Q}, (\mathrm{Q}', \mathrm{S}))$ *where* Q *is called* context path*,* Q' *is called* target path*, and* $\mathrm{S} = \{\mathrm{P}_1, \ldots, \mathrm{P}_\mathrm{k}\}$*, such that* $\forall \mathrm{P_i},\ \mathrm{Q}.\mathrm{Q}'.\mathrm{P_i}$ *is a* valid *path expression. The paths* $\mathrm{P}_1, \ldots, \mathrm{P}_\mathrm{k}$ *are called* key paths *of* ψ*. If* $\mathrm{Q} = \varepsilon$*,* ψ *is an* absolute key*, otherwise* ψ *is so-called* relative key*. A* composite key *(or* k-key*) is a key with more than one* key path*. The* key degree *entails the number of* key paths*.*

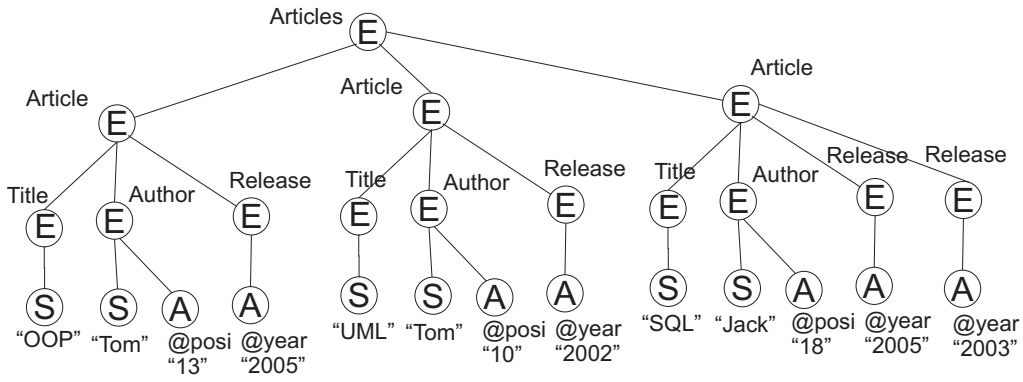


Figure 2.5: Example of Primary Keys in XML

Example 9 (Primary Key): *To illustrate, in Figure 2.5* $(\varepsilon, (\text{Article.Author}, \{@\text{posi}\}))$ *is a key expression and it is an* absolute key*, too.*

Each object rooted in the branches Article.Author *is uniquely identified by the value of* {key}*. On the other hand,* (Article, (Release, {@year})) *is another key expression and because* Article *is not equal to* ε*, it is a* relative key*. Inside each element* Article, Release *is identified by* @year*, therefore there can occur more elements* Article *with the same value of* @year *in* Release *.*

Definition 7 (Primary Key Satisfaction)**.** *Let* $\psi = (\mathrm{Q}, (\mathrm{Q}', \{\mathrm{P}_1, \ldots, \mathrm{P}_\mathrm{k}\}))$ *be a key expression. An XML tree T* satisfies ψ ($\mathrm{T} \models \psi$)*, iff for any* $\mathrm{n} \in [\mathrm{Q}]$*, given any two nodes* $\{\mathrm{n}_1, \mathrm{n}_2\} \subseteq \mathrm{n}[\mathrm{Q}']$*, either* $\mathrm{n}_1 = \mathrm{n}_2$ *or there exists at least one path* $\mathrm{p} \in \mathrm{P}_\mathrm{i}$*, and nodes* $\mathrm{x} \neq_\mathrm{v} \mathrm{y}$*, such that* $\mathrm{x} \in \mathrm{n}_1[\mathrm{p}]$ *and* $\mathrm{y} \in \mathrm{n}_2[\mathrm{p}]$*. Satisfaction can be expressed as a clause:*

$$\forall n \in [Q], \forall n_1, n_2 \in n[Q'] : \quad (\bigvee_{1 \leq i \leq k} n_1[P_i] \cap_v n_2[P_i] = \epsilon) \vee n_1 = n_2$$

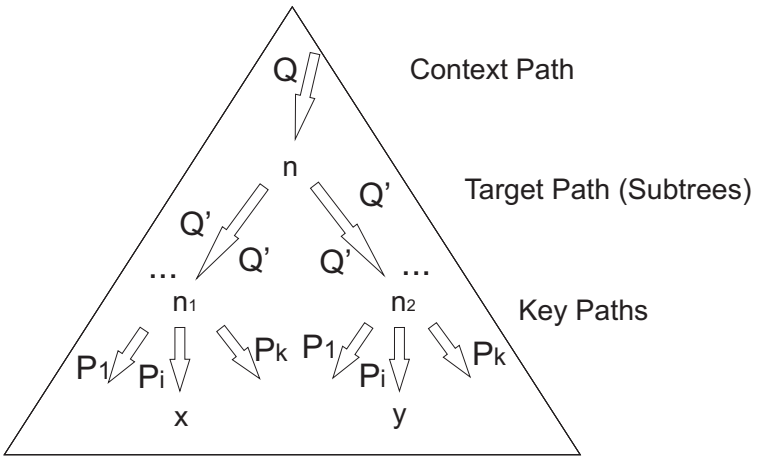


Figure 2.6: Visualisation of Key Satisfaction in XML

Example 10 (Primary Key Satisfaction): *Figure 2.6 illustrates satisfaction for a key* $(\mathrm{Q}, (\mathrm{Q}', \{\mathrm{P}_1, \ldots, \mathrm{P}_\mathrm{k}\}))$*.*

2.2.5 Formalization of Foreign Keys in XML

Since [11] does not propose a formalism for foreign keys, the new one will be provided in this thesis as follows.

Definition 8 (Foreign Key (FK))**.** *A* foreign key ϕ *in an XML document is an expression* $(\mathrm{Q}, (\mathrm{Q}'_\mathrm{f}, \mathrm{S}_\mathrm{f})) \Rightarrow (\mathrm{Q}, (\mathrm{Q}'_\mathrm{k}, \mathrm{S}_\mathrm{p}))$ *where* $(\mathrm{Q}, (\mathrm{Q}'_\mathrm{k}, \mathrm{S}_\mathrm{p}))$ *is a primary key* ψ *with* $\mathrm{S}_\mathrm{p} = \{\mathrm{P}_1, \ldots, \mathrm{P}_\mathrm{k}\}$*,* Q'_f *is called* target path*, and* $\mathrm{S}_\mathrm{f} = \{\mathrm{F}_1, \ldots, \mathrm{F}_\mathrm{k}\}$*, such that* $\forall \mathrm{F}_\mathrm{i}$, $\mathrm{Q}.\mathrm{Q}'_\mathrm{f}.\mathrm{F}_\mathrm{i}$ *is a* valid *path expression. The paths* $\mathrm{F}_1, \ldots, \mathrm{F}_\mathrm{k}$ *are called* foreign key paths *of* ϕ*. Consequently, if* ψ *is an*

absolute key, *than* ϕ *must also be an* absolute foreign key *and vice versa. The left part of the foreign key expression is called the* dependent part, *while the* referenced part *is positioned on the right side. A* composite foreign key *is a foreign key with more than one* foreign key path.

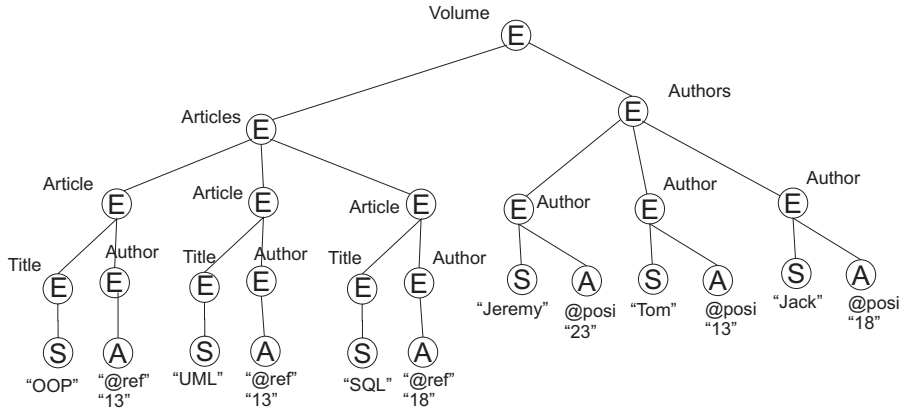


Figure 2.7: Example of Foreign Keys in XML

Example 11 (Foreign Key): *For instance, in Figure 2.7* $(\varepsilon, (\text{Articles.Article.Author}, \{@\text{ref}\})) \Rightarrow (\varepsilon, (\text{Authors.Author}, \{@\text{posi}\}))$ *is a foreign key expression and because* $(\varepsilon, (\text{Authors.Author}, \{@\text{posi}\}))$ *is an absolute key, a respective foreign key construct is an* absolute foreign key, *too. Each object rooted in the branches* Articles.Article.Author *is referenced to some branch* Authors.Author *through the foreign key/ primary key pair* @ref/@posi. *In other words, for each value in* /Articles.Article.Author.@ref *must occur an equal value in* /Authors.Author.@posi.

Example 12 (Composite Foreign Key): *A composite foreign key noted in XSD is shown in Figure 2.8 -* $(\varepsilon, (\text{Articles.Article.Author}, \{@\text{ref1}, @\text{ref2}\})) \Rightarrow (\varepsilon, (\text{Authors.Author}, \{@\text{posi1}, @\text{posi2}\}))$. *In the context of the root element* Volume, *for the values* @ref1, @ref2 *of each node targeted by* Articles.Article.Author *must exist a node targeted by* Authors.Author *which includes the same values of respective attributes* @posi1, @posi2. *Verification of values is proceeded in the specified order (it means* @ref1 = @posi1, @ref2 = @posi2. *To illustrate, a node with values* @posi1 = 5, @posi2 = 8 *satisfies* @ref1 = 5, @ref2 = 8 *but it does not satisfy* @ref1 = 8, @ref2 = 5.

Definition 9 (Foreign Key Satisfaction)**.** *Let* $\phi = (\text{Q}, (\text{Q}'_\text{f}, \{\text{F}_1, \ldots, \text{F}_\text{k}\})) \Rightarrow (\text{Q}, (\text{Q}'_\text{p}, \{\text{P}_1, \ldots, \text{P}_\text{k}\})$ *be a foreign key expression. An XML tree* T satisfies ϕ ($\text{T} \models \phi$), *iff for any* $\text{n} \in [\text{Q}]$, *for any* $\text{s}_\text{f} \in \text{n}[\text{Q}'_\text{f}]$, *given any sequence of nodes* $\{\text{f}_1, \ldots, \text{f}_\text{k}\}$, $\text{f}_1 \in \text{s}_\text{f}[\text{F}_1], \ldots, \text{f}_\text{k} \in \text{s}_\text{f}[\text{F}_\text{k}]$,

```
<Articles>                          <Authors>
...                                     <Author posi1="2" posi2="15">
    <Author ref1="5" ref2="8"/>             Jeremy Hill
...                                     </Author>
    <Author ref1="2" ref2="15"/>        <Author posi1="5" posi2="8">
...                                         Tom Bush
    <Author ref1="2" ref2="15"/>        </Author>
...                                   ...
</Articles>                         </Authors>

<element name="Volume">
    <keyref name="AuthorCompFK" refer="AuthorCompPK">
        <selector path="./Articles/Article/Author"/>
        <field path="./@ref1"/>
        <field path="./@ref2"/>
    </keyref>
</element>
```

Figure 2.8: Example Composite Foreign Key (extension of Figure 2.7)

$\exists s_p \in n[Q'_p]$, *where exists a sequence of nodes* $\{p_1, \ldots, p_k\}$, $p_1 \in s_p[P_1], \ldots, p_k \in s_p[P_k]$ *where* $f_1 = p_1, \ldots, f_k = p_k$.

2.3 Key Inference Rules and Redundance

2.3.1 Logical Implication on Keys

The purpose of this section is to formalize redundancy of keys in order to be able to recognize needless expressions. It also considers features of key expressions and offers the characterization of interesting cases to search for.

Definition 10 (Logical Implication 1)**.** *Let* τ *and* ω *be the key expressions. Then* τ logically implies ω *(*$\tau \models \omega$*), if every XML tree that satisfies* τ *also satisfies* ω*.*

Definition 11 (Logical Implication 2)**.** *A set* K *of key expressions* logically implies *a key expression* ω *(*$K \models \omega$*) if every XML tree that satisfies all key expressions in* K *also satisfies* ω*.*

2.3.2 Closure of Key Expressions and Key Redundance

Definition 12 (Closure)**.** *Let* K *be a set of key expressions.* $K^+ = \{\tau : K \models \tau\}$ *is called the* closure *of K. In other words,* K^+ *is the set of all key expressions implied by* K*.*

Definition 13 (Equivalence of Key Sets)**.** *Two sets* K *and* L *of key expressions are* equivalent *if* $\mathrm{K}^+ = \mathrm{L}^+$

Definition 14 (Key Redundance)**.** *Let* K *be a set of key expressions. A key expression* $\tau \in \mathrm{K}$ *is* redundant *when* $\mathrm{K}\backslash\{\tau\} \models \tau$*. In other words, if* $\{\mathrm{K}\backslash\{\tau\}\}^+ = \mathrm{K}^+$

2.3.3 Inference Rules for Key Implication

In this section, complete axiomatization presented by Buneman et al [12] is going to be assumed. The axiomatization consists of inference rules shown in the following Table 2.1. Next, interpretation of these rules within the scope of the key inference process will be discussed.

$\dfrac{(Q,(Q^{'},S)), P \in PL}{(Q,(Q^{'},S \cup \{P\}))}$	superkey
$\dfrac{(Q,(Q^{'}.Q^{''},\{P\}))}{(Q,(Q^{'},\{Q^{''}.P\}))}$	subnodes
$\dfrac{(Q,(Q^{'},S \cup \{P_i,P_j\})), P_i \sqsubseteq P_j}{(Q,(Q^{'},S \cup \{P_j\}))}$	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
$\dfrac{(Q_1,(Q_2,\{Q^{'}.P_1,\ldots,Q^{'}.P_k\}))\ (Q_1.Q_2,(Q^{'},\{P_1,\ldots,P_k\}))}{(Q_1,(Q_2.Q^{'},\{P_1,\ldots,P_k\}))}$	interaction
$\dfrac{Q \in PL, S\ is\ a\ set\ of\ PL\ expressions}{(Q,(\epsilon,S))}$	epsilon

Table 2.1: Inference Rules for Key Implication

Let K be a set of key expressions.

- Let $S^{'}$ be a superset of S. The *superkey rule* means that if $(Q,(Q^{'},S)) \in K$, then an expression $(Q,(Q^{'},S^{'}))$ is redundant in K.
- The *subnodes rule* entails that if $(Q,(Q^{'}.Q^{''},\{P\})) \in K$, then an ex-

pression $(Q, (Q', \{Q''.P\}))$ is redundant in K. In other words, the target path should be as long as possible.

- The *containment-reduce rule* indicates also that if $P_i \subseteq P_j$, then $(Q, (Q', S \cup \{P_i\}))$ is implied by $(Q, (Q', S \cup \{P_j\}))$. In other words, the key paths should be as general as possible.

- In the same manner as in the previous item, the *context-path-containment rule* entails that context path should be as general as possible and the *target-path-containment rule* induce that target path should be as general as possible.

- The *context-target rule* is similar to *subnodes rule* and it indicates that context path should be as short as possible.

- The *prefix-epsilon rule* concludes that when $\epsilon \in K$, the other key paths should be as short as possible.

- The *interaction rule* allows to move a prefix Q' which occurs in all key paths to the target path Q_2. The second part in the precondition prevents the existence of more than one Q' node under Q_2 that corresponds in their key paths.

- The *epsilon rule* indicates that if $Q.S$ is a valid path, then $(Q, (\epsilon, S))$ is a key. However, this type of keys is not very interesting thus, only key expressions $(Q, (Q', S))$ where $Q' \neq \epsilon$ will be discovered.

2.4 Inclusion Dependency

Inclusion dependencies (INDs) are strong preconditions for foreign key constraints (in relational databases as well as in XML data). The following description is sufficient for our purposes. The exact definition of INDs for relational databases is shown in Section 3.6.2. And the definition for inclusion dependency in XML is mentioned, for example, in [12, 13], however, the foreign key inference process proposed in this thesis utilize the definition of INDs for relational databases.

- A unary inclusion dependency (IND) $A \subseteq B$ represents an inclusion of a set of values of the *dependent* attribute A in a set of values of the *referenced* attribute B. The pair of attributes A, B is called IND *candidate*.

- An IND is *satisfied* if IND requirements are met and *unsatisfied* otherwise.

- An IND is *trivial* if $A \subseteq A$

- An attribute is *covered by an IND* if it is involved in stated IND as dependent or as referenced attribute.
- Inclusion dependencies containing on the right-hand side or on the left-hand side a set with more than one attribute are so-called *Composite INDs* (for instance, $AB \subseteq CD$), the number of attributes on both sides must be equal.
- The *level* of a *composite* IND entails the number of dependent (or referenced) attributes.

EXAMPLE 13 (Inclusion Dependency): *Let* A, B, C, D, E *be the set of values as follows:*

- A = *(3, 2, 5, 8)*
- B = *(5, 4, 9, 2)*
- C = *(3, 8, 5)*
- D = *(5, 2, 9)*
- E = *(2)*

Consequently, these INDs can be obtained - $C \subseteq A,\ D \subseteq B,\ E \subseteq D, E \subseteq A,$ $E \subseteq B,\ CD \subseteq AB$

Chapter 3

Analysis of Recent Approaches

3.1 Overview

This chapter includes the descriptions of recent approaches of searching for primary and foreign keys with focus on the data-driven principle. The most interesting and remarkable proposals are presented in detail and their pros and cons are evaluated.

3.1.1 Query-Driven Approaches

Since the XQuery queries evaluated by the system are not always available for the user, this diploma thesis is primarily aimed at data-driven methods. Nevertheless, understanding the logic of the principles how application data are utilized can be very useful in providing additional auxiliary information within the inference process.

Firstly, several older proposals from the scope of reverse engineering have appeared and are mentioned in Section 7.4.1. Secondly, a more precise research considering XML queries is presented a bit more in detail in Section 7.6.2.

3.1.2 Inclusion Dependencies

This chapter contains the two most important algorithms for locating INDs. They are the only algorithms which are able to detect all unary INDs during only one run through all the nodes (if a preprocessing phase is not counted) and they include the computation of composite inclusion dependencies, too. But, the inference of composite INDs is extremely difficult and time-consuming, therefore it is not considered in this thesis. As a result,

the methods of the composite INDs computation are described a bit less in detail (mostly without pseudo-code or detailed examples).

3.1.3 Functional Dependencies

The role of functional dependency (FD) in the integrity constraints discovery is described in Section 7.3. In short, in the relational data model a key is a special case of FD and in XML it is bit more complicated, because it depends on particular definitions. Despite, in Section 7.3.1 are described a few major proposals about detection of functional dependencies in XML data. However, as far as only primary key inference is concerned there, the proposed methods do not contain any special ideas and only check value uniqueness in constructed relations. Thus, FD discovery techniques are not going to be considered in this chapter.

3.2 Gordian

3.2.1 Overview

First Name	Last Name	Pages	Title
Tom	Black	15	OOP
Jeremy	Hill	15	UML
Tom	Black	8	SQL

Table 3.1: Example Dataset (derived from Article)

Pages	Title	**Count**
15	OOP	**1**
15	UML	**1**
8	SQL	**1**

First Name	Last Name	**Count**
Tom	Black	**2**
Jeremy	Hill	**1**

Figure 3.1: Examples of Projections

The Algorithm Gordian has been described in [41]. It is a powerful tool for extracting composite primary keys from datasets. The idea behind is an observation that a projection of entities corresponds to a key if each counted aggregation for a projection is equal to 1. Thus, this method searches for all possible projections of a dataset while computing aggregations on the projected part of the set of entities (so-called *slice-by-slice computation*).

Example 14 (Projections): *The Figure 3.1 shows some examples of projections made from the dataset of entities in Table 3.1. Since the right projection counts all aggregations equal to 1, columns* `Pages`, `Title` *can be*

considered as the composite key. However, columns `First Name`, `Last Name` *from the projection on the left can not be rated as a composite key.*

Slice, Segment and Subsuming

Terms *slice and segment* appear very often in the description of Gordian. The *slice* of the set of entities is defined as a subset of entities obtained through a selection operation on the dataset. Then a *segment* of the projections corresponds to the *slice* selection. In other words, if data are in a relational table, *slices* select rows (for instance, *slice* `First Name = Tom`) while *segments* select columns of some slice. Now, consider the slice $\mathcal{S}$ (`First Name = Tom`) depicted in Figure 3.2 and the *slice* $\mathcal{T}$ which corresponds to `Last Name = Black`. It is possible to note that the value `"Black"` appears only with the value `"Tom"`. Thus, the slice $\mathcal{S}$ *subsumes* the slice $\mathcal{T}$ and non-keys located in $\mathcal{T}$ would be redundant to non-keys from $\mathcal{S}$. This observation is the core of the pruning techniques and allows to skip the significant part of the search space.

First Name	Last Name	Pages	Title	**Count**
Tom	Black	5	OOP	**1**
Tom	Black	8	SQL	**1**

First Name	Last Name	**Count**
Tom	Black	**2**

Figure 3.2: Some Segments of the Slice First Name Equals Tom

Example 15 (Segments): *Some segments of the slice* `First Name = Tom` *are depicted in Figure 3.2. In the second example remains only one row, because value* `"Tom"` *appears only with value* `"Black"`.

Prefix Tree

First, the algorithm creates a compact representation of entities called *prefix tree* (PT). It facilitates powerful merging a pruning steps within the elicitation process. PT has a structure a bit like XML tree, but each *level of PT* corresponds to one attribute. The main elements are called *nodes* which are composed of *cells*. The *levels of PT* are represented by so-called *attribute numbers*, that are in Figure 3.3 displayed in brackets. Each leaf-node cell retains a *counter* which says the number of discovered occurrences of values in a cell on the current level of PT. In other words, It holds the counted aggregation for this value of the corresponding attribute. Whenever any *counter* exceeds 1, the corresponding attribute is set as the non-key attribute.

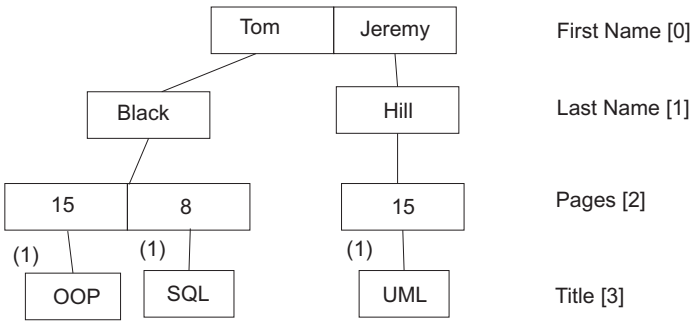


Figure 3.3: Example of Prefix-Tree in Reference to Table 3.1

EXAMPLE 16 (Prefix Tree): *Figure 3.3 presents PT built from data in Table 3.1*

Depth-First Search

When PT is created, the Gordian algorithm performs a depth-first (DF) traversal. Each DF path represents a slice in the entities and its computation of aggregations is provided by recursively merging the children. In short, a merge operation corresponds to the computation of one segment of the respected slice. Whereas all possible segments for all slices are traversed and checked, efficient pruning is utilized in order to visit as few nodes as possible.

Final Phase

Gordian is designed to avoid extracting redundant non-keys. They are discovered with the aid of pruning methods or during the course of insertion into the `nonKeySet` structure. Finally, all the non-redundant non-key candidates are obtained and non-redundant set of keys is extracted.

3.2.2 Finding Non-Keys

The function (Algorithm 1) provides the double DF-traversal of PT with merging of nodes during the backtrack in order to discover non-keys. It takes the root of PT and the corresponding attribute number (the level of PT) as input and fills the `nonKeySet` container with elicited non-keys. `curNonKey` is constructed from the set of attributes and examined if it should be stored in `nonKeySet` or not (lines 4,8). The first DF-recursion is performed on the lines 14-16 (the exploration of slices). After merging of visited nodes (line 23) (merging is properly explained in the following section 3.2.3), second recursion is performed (the exploration of segments). However, huge amount of nodes are not visited twice due to pruning (lines 11, 15, 19, 20), which is explained thoroughly in the section 3.2.4. Stop of recursion is managed on the lines 2-9 where the belonging to `nonKeySet` is checked, too.

Non-redundance of non-keys in `nonKeySet` is tested within the process of a non-key insertion into the container. Firstly, a new non-key is not added if it is covered by some recently discovered non-key. Secondly, all recently discovered non-keys in the container `nonKeySet` which are covered by the new non-key are discarded. An example of finding non-keys is demonstrated in the section 3.2.5.

Algorithm 1 Finding Non-Keys

Input: *root*: node of PT, $attrNo$: attribute number
Output: inferred non-keys stored in $nonKeySet$

```
1:  curNonKey += attrNo
2:  if root is leaf then
3:    if ∃ cell in root with counter > 1 then
4:      nonKeySet ← curNonKey
5:    end if
6:    curNonKey −= attrNo
7:    if (root contains more than 1 cell) OR (counter of the only cell > 1)
      then
8:      nonKeySet ← curNonKey
9:    end if
10: else
11:   for each cell in root do
12:     if cell.child has not been previously traversed then
13:       findNonKey(cell.child, attrNo + 1)
14:     end if
15:   end for
16:   curNonKey −= attrNo
17:   if root contains more than 1 cell then
18:     if curNonKey is futile then
19:       return
20:     end if
21:     mergedTree ← merge all the children of root
22:     findNonKeys(mergedTree, attrNo + 1)
23:     delete mergedTree
24:   end if
25: end if
```

3.2.3 Merging

Merging is a typical recursive algorithm, depicted in the Algorithm 2 (adopted from [41]), which takes a group of nodes to merge as input and builds a modified tree where same values are grouped together. Firstly, the stop of recursion is proceeded when the last node remains (line 2). Secondly,

if there are only leaf-node cells required to merge, the method creates a new cell for each distinct value v in the nodes to merge and sets *counters* at sum of *counters* over all input cells containing value of v (line 9). Thirdly, the algorithm calls recursively the merging function at the child nodes of the cells that share the value v (line 12). An example of merging is described later in the section 3.2.5.

Algorithm 2 Prefix Tree Merging

Input: $toMerge$: group of nodes to merge
Output: $result$: modified tree with merged nodes on the current level

```
 1: if toMerge contains the last one node then
 2:   result ← toMerge
 3: else
 4:   result ← create new node
 5:   for each distinct value v_i in nodes in toMerge do
 6:     newCell ← create new cell in previously created node
 7:     newCell.value ← v_i
 8:     if all nodes in toMerge are leaves then
 9:       newCell.counter ← number of cells in toMerge with value = v_i
10:     else
11:       intoRec ← the set of children from the cells from toMerge having
          values = v_i
12:       newCell.child ← Merge(intoRec)
13:     end if
14:   end for
15: end if
16: return result
```

3.2.4 Pruning

The pruning quickens finding non-keys by orders of magnitude without an influence on preciseness.

Singleton Pruning

- line 15: After merging the algorithm may want to check again previously traversed subtrees. Consequently, any *shared* prefix subtree is not traversed at this point. In other words, if there existed a non-key at this moment, it would be previously discovered, because the new *slice* should be *subsumed* by some earlier located non-key.

- line 20: The merging operation would return the situation described in the first point (line 11). Thus, no new non-redundant nonkeys could be discovered.

Futility Pruning

(line 21) The futility pruning is based on earlier inferred nonkeys stored in the `nonKeySet` container. A new prefix subtree is checked, whether there occur non-keys in the container of discovered non-keys that cover all of the possible non-keys that could be found. The coverage test is performed efficiently utilizing bitmaps.

3.2.5 Example of Searching for Non-Keys

This section provides a detailed example of finding non-keys in the prefix-tree illustrated in Figure 3.3. Although it contains only three entities with four attributes each, it is sufficient to show the basic concepts discussed so far. Almost every point includes the current `curNonKey` displayed in brackets in the beginning and proceeded lines of code in parenthesis at the end.

1. The algorithm starts with the root node and runs recursively until it reaches the leaf node `OOP`

2. `<First Name, Last Name, Pages, Title>`
   The current slice is (`Tom, Black, 15, OOP`). Because `counter` of the only cell equals 1, no non-key is discovered. (2-5)

3. `<First Name, Last Name, Pages, Title>` Now, the algorithm projects out the current `attrNo = Title` in order to try if it is a non-key. (6)

4. `<First Name, Last Name, Pages>` If the current leaf node contained more than one cell, the current `curNonKey` would have to retain a non-key because it would be the result of previously shared nodes (nodes having same values within the function `merge`). But no non-key is found for now. (7-9)

5. `<First Name, Last Name, Pages, Title>` Gordian backtracks to the node `(15,8)` and performs the same operation (points 2-4) with the next child and it finishes with the same result.

6. `<First Name, Last Name, Pages>` The Algorithm is again back in the node `(15,8)` and have no more children for traversal, therefore the current `attrNo = Pages` is projected out (16).

7. `<First Name, Last Name>` Now if the current root node consist of 2 or more cells, the previous nodes (ancestors) must be parts of a non-key. Thus, the merging operation is performed in order to determine if that non-key can have more items than only ancestors. Because the node `(15,8)` is composed of two cells, the merging operation starts in order to state if `attrNo = Title` can be the part of the non-key, too. (17-22)

8. The resulting merged tree is composed only of the node `(OOP,SQL)`.

9. `<First Name, Last Name, Title>`
   The current slice is (`Tom, Black, 15, (OOP,SQL)`). Because `counter` of the only cell equals 1, no non-key is discovered. (2-5)

10. `<First Name, Last Name, Title>` Steps 2, 3 are performed with the current `curNonKey` and on the ground of containing 2 cells in the merged leaf node `(OOP,SQL)`, the first non-key `<First Name, Last Name>` is discovered. (7-9)

11. `<First Name, Last Name>` Next, the algorithm backtracks back to the node `(Black)` and projects out (`attrNo = <Last Name>`). (16)

12. `<First Name>` The current node `(Black)` contains only one cell, therefore no merging is proceeded and the algorithm backtracks to the node (`Tom, Jeremy`).

13. `<First Name>` Consequently, DF traversal continues through the right part of PT (the cell (`Jeremy`)) until the end - the leaf node (`UML`).

14. `<First Name, Last Name, Pages, Title>`
    The current slice is (`Jeremy, Hill, 15, UML`). Steps 2-4 are executed and no new non-key is inferred.

15. `<First Name, Last Name, Pages>` Now, steps 11, 12 are repeatedly processed (all the nodes on the way back contain only one cell) and Gordian backtracks back to the node (`Tom, Jeremy`).

16. `<First Name>` Steps 6, 7 are proceeded. (the current `attrNo = First Name` is projected out).

17. <empty> The constructed merged tree is visualized in Figure 3.4.

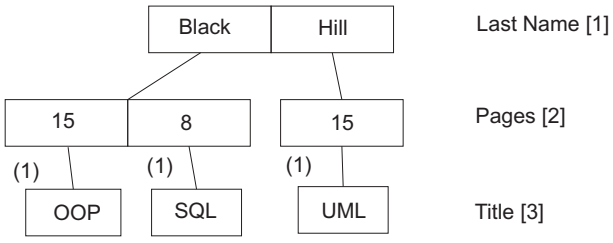


Figure 3.4: Merged Prefix-Tree

18. `<Last Name>` Figure 3.4 demonstrates, that the merged tree is composed only of the previously traversed branches, because no cells with

same values have not been found there. As a result, no part of a non-key have not been discovered and it has no reason to traverse children of the node `(Black, Hill)`. (12)

19. `<Last Name>` (`attrNo = <Last Name>`) is projected out and children of the node `(Black, Hill)` are merged. (16-21)

20. `<empty>` The constructed merged tree is visualized in Figure 3.5.

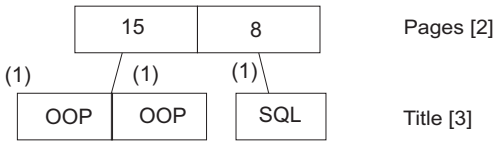


Figure 3.5: Merged Prefix-Tree 2

21. `<Pages>` On the contrary to the case in the step 18, the merging operation builds new consolidated node `(15)`, which means that current (`attrNo = <Pages>`) is the part of a non-key. Consequently, the DF-traversal of its descendants continues to test if this non-key could contain (`<Title>`), too.

22. `<Pages, Title>` The Algorithm proceeds steps 9,10.

23. `<Pages>` step 19

24. The resulting merged tree is composed of the node `(OOP, SQL, UML)`.

25. `<Title>` All counters are equal to 1, therefore no new non-key is discovered. Gordian leaves recursion. In the end, the `nonKeySet` container holds `(<First Name, Last Name>, <Pages>)`.

3.2.6 Keys from Non-Keys

The final step of the Gordian algorithm is the primary key inference from the set of discovered non-keys. The basic idea is that the set of keys corresponds to the cartesian product of the complement sets of the non-keys. In short, in the Algorithm 3 (adopted from [41]) the complement set is computed for each non-key, then the cartesian product is taken with the previously seen complement sets, and in the end any redundant keys are pruned. The following pseudo-code with the example does not require more detailed description.

Example 17 (Keys from Non-Keys): *In the foregoing section have been computed these non-keys* `(<First Name, Last Name>, <Pages>)`. *As a result, the complement sets are* `(<Pages, Title>, <First Name, Last Name,`

Title>). Then computed keys are following (<First Name, Pages>, <Last Name, Pages>, <Pages,Title>, <First Name, Title>, <Last Name, Title>, <Title>). After elimination of redundancies - (<First Name, Pages>, <Last Name, Pages>, <Title>).

Algorithm 3 Computing Keys

Input: $nonKeySet$: set of found non-keys
Output: $keySet$: set of discovered keys

```
1: for each nonKey in nonKeySet do
2:   complement ← complement of nonKey
3:   if keySet is empty then
4:     keySet ← complement
5:   else
6:     tempSet ← 0
7:     for each cKey in complement do
8:       for each key in keySet do
9:         tempSet ← (key ∪ cKey)
10:      end for
11:    end for
12:    eliminate redundant keys in tempSet
13:    keySet ← tempSet
14:  end if
15: end for
16: return keySet
```

3.2.7 Conclusion

In conclusion, the Algorithm Gordian is a novel method for very fast detecting composite primary keys in a dataset while avoiding the exponential processing and memory requirements. Experiments which are demonstrated in [41], show that Gordian can discover all composite keys in the time that other approaches required to extract single-attribute keys. In XML application it can be directly utilized to discover, for instance, composite absolute primary keys. In addition, the DF-traversal and the usage of the prefix tree representation make it easily possible to detach the computation process, which proceeds some bigger set of entities, in order to obtain keys also from parts of the original set. Consequently, it should be possible with some modifications of the algorithm to infer relative primary keys in XML, too. Unfortunately, it is very difficult to modify (or set) the algorithm to locate only 2 or 3-attribute composite keys, because all the non-keys must be found first.

3.3 Primary Keys from XML Data

3.3.1 Overview

An apriori-like algorithm for discovering of primary keys in XML data through association rules has been proposed in [39]. The method is capable even to extract keys which contain wildcard ? in path expressions. The authors have defined support and confidence of a key expression and only the minimal cover of the set of primary key expressions is considered during the inference process. At first, the algorithm finds absolute 1-keys and relative 1-keys. Later, from these key expressions the apriori-like algorithm try to generate all other k-keys afterwards.

3.3.2 Preliminaries

Support and Confidence

Support denotes the rate of occurrence of the given potential key expressions. Besides, uniqueness of the values of the given potential key expressions is indicated as *confidence*.

Definition 15 (Support and Confidence)**.** *Let* $\phi = (\mathrm{Q}, (\mathrm{Q}', \{\mathrm{P}_1, \ldots, \mathrm{P}_\mathrm{k}\}))$ *be a k-key expression,* T *an XML tree,* $\mathrm{n} \in [\mathrm{Q}]$ *and* $\mathrm{n}[\mathrm{Q}'] = \{\mathrm{n}_1, \ldots, \mathrm{n}_\mathrm{m}\}$. *Then* $\mathrm{branches}(\mathrm{n_j}, \mathrm{P_i})$ *entails the amount of* $\mathrm{P_i}$*-branches in the subtree rooted at* $\mathrm{n_j}$. *Next,* $\mathrm{values}(\mathrm{n_j}, \mathrm{P_i})$ *denotes the number of distinct values of the subtree rooted at* $\mathrm{n.n_j.P_i}$. *The* support *of* ϕ *in the subtree rooted at* n *is*

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

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

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

If $\mathrm{supp}(\mathrm{n}, \phi) = 0$, then $\mathrm{conf}(\mathrm{n}, \phi) = 1$.

Definition 16 (Support and Confidence in XML Tree)**.** *The support of* ϕ *in the whole tree* T *is*

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

And the confidence of ϕ *in* T *is*

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

The values of each branch $n.n_j.P_i$ are comprehended as a bag due to above definitions of *support* and *confidence*. And the *support* of the branches $n.n_j.\{P_1, \ldots, P_k\}$ corresponds to the amount of elements in the Cartesian product of all bags.

Example of Support and Confidence

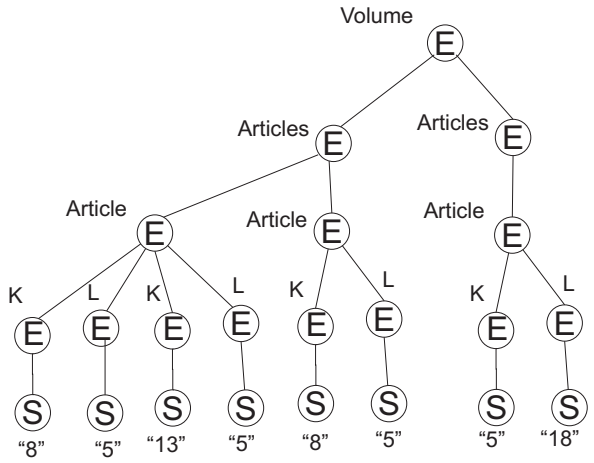


Figure 3.6: Example of XML Tree Demonstrating Support and Confidence

The Figure 3.6 is an example of XML tree for illustrating support and confidence of key expression `(Articles, (Article, {K, L}))`. The graph includes 3 branches for `Articles.Article`.

- The support for `(Articles, (Article, {K, L}))` in the first branch is 4, because there exist 2 branches for `Articles.Article.K` and 2 branches for `Articles.Article.L`. Then the support for the second and third branch amounts 1. Consequently, the support for the whole XML tree is 6.
- As far as confidence is concerned, in the first subtree occurs 2 distinct values of `{(8, 5), (13, 5)}`, so the confidence amounts 2/5. Second subtree has confidence 100%. In accordance to the definition, the confidence for `(Articles, (Article, {K, L}))` amounts $2/5 = 40\%$.

Threshold and Fake Keys

- There exists two types of thresholds:
  - The first one is called *min-support* and determines the matter of interest of key candidates.
  - The second one is titled *min-confidence* and corresponds to preciseness of the key candidates.

- A key is *accurate* if its confidence is bigger than a given *min-confidence.*
- The key is so-called *approximate key* if it passes both thresholds.
- A *valid path* expression p in PL is the path expression where $\forall$ label $l \in p$, if $l \in A$ or $l = \{S\}$, then l is at the end of p, and the support is greater than min-support.

Definition 17 (Fake Key)**.** *A k-key expression* $\phi = (\mathrm{Q_1}, \mathrm{Q_1^{'}}, \mathrm{S})$ *is called a* fake key *if it exceeds both thresholds and another expression* $\omega = (\mathrm{Q_2}, \mathrm{Q_2^{'}}, \mathrm{S})$ *exists such that* $\mathrm{Q_2} \subseteq \mathrm{Q_1}$ *and* $\mathrm{Q_1^{'}} = \mathrm{Q_2^{'}}$*, or* $\mathrm{Q_2^{'}} \subseteq \mathrm{Q_1^{'}}$ *and* $\mathrm{Q_1} = \mathrm{Q_2}$*, and* ω *does not pass any of the thresholds.*

Example 18 (Fake Key): *To show an example of a fake key, consider the following absolute key expressions:*

1. $(\epsilon, (\text{volume.article.author}, \{\epsilon\}))$ *supp: 10, conf: 50%*
2. $(\epsilon, (\text{volume.} * \text{.author}, \{\epsilon\}))$ *supp: 30, conf: 100%*

Assume the min-support of 10 and min-confidence is 100%. In accordance to the above statement, second key is an approximate key, while first one is not. Since `volume.article.author` *in the expression 1 is contained in* `volume.*.author` *from the expression 2, according to the target-path-containment rule(from the table 2.1), if the expression 2 is a key* $\Rightarrow$ *the expression 1 must be a key, too. Consequently, the expression 2 is declared as a fake key.*

Partial Order

The *partial order* shows which keys should not be hold in a minimum cover. Thus, it allows to an algorithm to mine a minimal cover for the set of keys of a given XML tree. The example is adopted from [39].

Definition 18 (Partial Order)**.** *Given key expressions* $\tau = (\mathrm{Q_1}, (\mathrm{Q_1^{'}}, \mathrm{S_1}))$ *and* $\omega = (\mathrm{Q_2}, (\mathrm{Q_2^{'}}, \mathrm{S_2}))$ *where* $\mathrm{S_1} = \{\mathrm{P_1}, \ldots, \mathrm{P_k}\}$, $\mathrm{S_2} = \{\mathrm{P_1^{'}}, \ldots, \mathrm{P_m^{'}}\}$*, then* τ precedes ω *(in the sense of* partial order*), denoted* $\tau \prec \omega$*, if at least one of the following conditions is satisfied:*

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

Example 19 (Partial Order): *The illustration is related to Figure 2.7.* (volume,(articles.article,{title,author})) $\prec$ (volume.articles,(article,{title,author}))

3.3.3 Prefix Tree

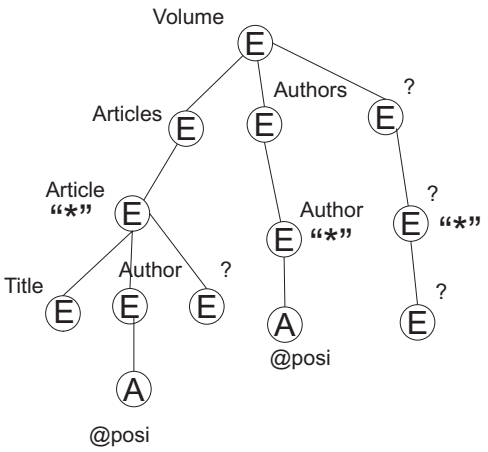


Figure 3.7: Example of Prefix-Tree of the XML Tree Depicted in Figure 2.7

A *prefix tree* is de facto an XML tree with merged nodes, so that each path from the root to a leaf occurs in the tree just once (not including data values). In addition, there is considered wildcard `'?'`. When `'?'` represents only one label, that branch is eliminated.

Example 20 (Wildcard '?' in Prefix Tree): *To illustrate, the prefix tree depicted in 3.7 could contain the path* `Volume.Authors.?`. *Because here* `'?'` *corresponds only to the label* `Author`, *it is excluded.*

In the prefix tree each leaf corresponds to a path expression from the root to the leaf, but labels which are situated more than once under their parents are signed with `'*'`. These marks will be useful for generating candidates of relative keys.

Example 21 (Prefix Tree): *The example of a prefix tree is demonstrated in Figure 3.7. It is based on the XML tree from Figure 2.7.*

3.3.4 Generating 1-Key Expressions

Absolute Keys

Firstly, candidate absolute keys are inferred among all the paths in the prefix tree. According to the Section 2.3.3, in the key expression $(Q, (Q', S))$ context-path Q should be as short as possible, target-path Q' should be as long as possible, and number of key paths in S should be as little as possible. Thus, key expressions consisting of $Q = \epsilon$ and $S = \epsilon$ are generated at first.

Example 22 (Candidate Absolute Keys): *The table 3.2 shows candidate*

	Key expressions	sup	conf
1	$(\epsilon, (Articles.Article.Title, \{\epsilon\}))$	3	100%
2	$(\epsilon, (Articles.Article.Author.posi, \{\epsilon\}))$	3	66, 7%
3	$(\epsilon, (Articles.Article.?, \{\epsilon\}))$	3	100%
4	$(\epsilon, (Authors.Author, \{\epsilon\}))$	3	100%
5	$(\epsilon, (Authors.Author.posi, \{\epsilon\}))$	3	100%
6	$(\epsilon, (?.?.?, \{\epsilon\}))$	6	100%

Table 3.2: Absolute 1-key expressions from the prefix tree depicted in Figure 3.7

absolute keys obtained from the prefix tree depicted in Figure 3.7.

Relative Keys

The relative keys can only occur within the context identified by the nodes which exist multiple times under their parents. The relative key candidates incurred from the expression $\phi = (\epsilon, (l_1, \ldots, l_n, \{\epsilon\}))$ are considered if the following conditions are met:

- The branch which determines the relative key is marked with `'*'`.
- ϕ has enough support, but does not hold enough confidence.
- ϕ is not absolute key. This pruning rule originates in the subnodes inference rule according to the Section 2.3.3

If the relative key candidate does not contain wildcard `'?'` in any path in the scope of relative key's context, then it can be directly concluded that its confidence is 100%. And therefore, it is assigned as an approximate relative 1-key.

EXAMPLE 23 (Candidate Relative Keys): *Considering support = 3 and confidence = 90% relative key candidate* (Articles.Article, (Author.posi, $\{\epsilon\}$)) *can be generated from the absolute key candidate contained in row 2 of the Table 3.2.*

Post-processing

Each generated key must pass the following post-processing steps:

1. For each new k-key ω, exclude all k-keys τ where $\omega \prec \tau$
2. remove fake keys

3.3.5 Mining Composite Keys

The oncoming method is typical apriori-like principle where k-candidates are constructed from unsatisfied (k-1)-candidates. Concretely, current candidates for composite keys are generated from 1-key expressions which are satisfied in the XML data with min-support, but does not involve enough confidence.

Mining absolute 2-key expressions

First of all, absolute 2-key candidates will be generated. The process is managed by two main rules:

1. A 2-key candidate can not be created from two 1-key expressions where one precedes the other by $\prec$.
2. From two 1-key expressions $(Q, (l_1.\cdots.l_k.l_{k+1}.\cdots.l_m, \{\epsilon\}))$ and $(Q, (l_1.\cdots.l_k.l'_{k+1}.\cdots.l'_p, \{\epsilon\}))$ the following 2-key expression is constructed $(Q, (l_1.\cdots.l_j, \{l_{j+1}.\cdots.l_m, l'_{j+1}.\cdots.l'_p, \}))$, where $j = 1, \ldots, k$.

Example 24 (2-key candidates): *For instance, from the two supported 1-key expressions* $(\epsilon, (\text{Articles.Article.Author.FirstName}, \{\epsilon\}))$ *and* $(\epsilon, (\text{Articles.Article.Author.LastName}, \{\epsilon\}))$ *the following 2-key candidates are generated:*

- $(\epsilon, (\text{Articles.Article.Author}, \{\text{FirstName}, \text{LastName}\}))$
- $(\epsilon, (\text{Articles.Article}, \{\text{Author.FirstName}, \text{Author.LastName}\}))$
- $(\epsilon, (\text{Articles}, \{\text{Article.Author.FirstName}, \text{Article.Author.LastName}\}))$

Mining k-keys

The basic principle is same as in the previous section. The oncoming theorem for elicitation of absolute or relative k-keys from two (k-1)-key expressions with $\text{k} > 2$ is adopted from [39], where the proof is published, too.

Theorem 1. *Let* $\text{k} > 2$. *A k-key expression* $(\text{Q}, (\text{Q}', \{\text{P}_1, \ldots, \text{P}_{\text{k}-2}, \text{P}_{\text{k}-1}, \text{P}_\text{k}\}))$ *can be supported and passes min-confidence only if the* (k-1)-key *expressions* $(\text{Q}, (\text{Q}', \{\text{P}_1, \ldots, \text{P}_{\text{k}-2}, \text{P}_\text{k}\}))$ *and* $(\text{Q}, (\text{Q}', \{\text{P}_1, \ldots, \text{P}_{\text{k}-2}, \text{P}_{\text{k}-1}\}))$ *are candidate* (k-1)-key *expressions, and all m-key expressions* $(\text{Q}, (\text{Q}', \text{S}))$, *where* $\text{m} < \text{k}$ *and* $\text{S} \subseteq \{\text{P}_1, \ldots, \text{P}_{\text{k}-2}, \text{P}_{\text{k}-1}, \text{P}_\text{k}\}$ *are supported.*

As a result, two basic rules have arisen:

- When $k > 2$, any two (k-1)-key expressions allow to construct at most one k-key candidate.

- K-key expression can not be created if one of the (k-1)-key expressions is already satisfied in the XML data (according to the superkey rule from Table 2.1).

3.3.6 Conclusion

In conclusion, the authors demonstrated very precise method for finding primary keys in XML (and composite primary keys, too).And on the ground of preciseness it includes some good ideas and observations. However, this algorithm seems to be quite ineffective. The authors have also presented experiments only on small database where have been in total only 8 keys discovered. Next, mining absolute keys like $(\epsilon, (Articles.Article.Author.posi\{\epsilon\}))$ seems to be a bit unrealistic. Inference rules must be kept, but users mostly specify some key path. Next unrealistic example is for instance utilizing '?' at the end of the target path or in other redundant places.

3.4 Combined Approach

3.4.1 Overview

[32] proposes the VQT algorithm (Value cardinality, Query pattern, Translation into XML). A relational schema to XML Schema translation method that analyzes the cardinalities between implicit data values and the equi-join characteristic in user queries and infers implicit referential integrities. The whole process consists of several different simple approaches, which are proceeded in parallel. At the end, it creates an XML Schema as the result. Since this diploma thesis concerns only about integrity constraints, the construction of XML Schema is going not to be described.

3.4.2 Value Cardinality Module

First, the Value cardinality module (VCM) acquires metadata of the relational schema (R-schema) and provides the analysis of cardinalities between the data values. Second, referential integrity relation information is detected by utilizing inclusion dependency property between the fields of primary and foreign keys. The whole process is composed of four steps: preprocessing, wordnet-based column extraction, candidate extraction, and refinement.

Preprocessing

VCM obtains metadata information of the relational database. It contains i.a. primary keys, property of the columns (data type and nullable information) and foreign key constraints. Then VCM contains a so-called

data structure Excepted column list (ECL) which for each primary key column holds a list of columns that can not be referenced from the primary key column. ECL is filled by this way:

- other primary key columns
- columns with different data types
- columns which are connected via foreign key constraint from metadata, there is no need to discover this referential relationship again

Wordnet-Based Column extraction

In this extraction step, the semantics between the columns are examined. Authors have implemented WordNet - widely used ontology for the semantic analysis and comparison between words. Concretely, functions SearchwordNet() and CompareSynset() have been utilized.

- First, list of words with similar or same meaning is acquired from SearchwordNet() for each column.
- Second, lists from the previous step are compared by CompareSynset() function. Pairs of columns that satisfy the comparison are stored in the data structure for referential integrity candidates (RIC).

Candidate Extraction and Refinement

In the first part, the RIC is filled according to the test of the value similarity. This test is performed on the pairs of columns whose selection is based on the ECL. Then inclusion dependency of candidates in RIC is checked. Everything is computed in the straightforward way - value by value in many for-loops.

3.4.3 Query Pattern Module

The Query pattern module (QPM) obtains user queries, analyzes them and educes referential integrity relation information by utilizing the equi-join property. In accordance to this property, columns which are related by equi-join in a query share a close interrelationship with each other. The whole process is composed of three steps: resource generation, candidate extraction,and refinement.

- The user queries are obtained in the resource generation phase and clauses containing "where"are stored in the data structure so-called Where clause stack (WCS).
- Next, RIC is filled only with candidates that in WCS reference other column.

- This phase is like in the VCM - candidates in RIC are tested for inclusion dependency property.

3.4.4 Conclusion

This method is interesting because of using several methods together - ontology analysis, data value analysis as well as query analysis. Consequently, the inference process can be more precise. Unfortunately the authors rely on the obtaining of metadata information of a relational database, therefore there is no primary key discovery. Although all phases are very simple and straightforward, the authors presented very good results of their experiments. According to these results, VQT is in [32] demonstrated as more efficient and precise method than other well-known translation methods. However, the potential of combined approach could be a lot better exploited and implemented (for instance, to merge some modules).

3.5 IND - SPIDER

3.5.1 Overview

The algorithm SPIDER (Single Pass Inclusion DEpendency Recognition) has been been proposed in [51, 53]. It detects all unary INDs during only one race through all the nodes. The process consists of two steps - sets of values are sorted during the first one and then all the candidates are analyzed in parallel. The core of the method is utilizing the data structure called min-heap which synchronizes the processing of all values of all attributes. Furthermore, some pruning strategies are presented in [53] and the algorithm is expanded for the elicitation of composite INDs, too.

3.5.2 Parallel test for all IND Candidates

Overall Description

SPIDER uses sorted attribute lists and obtains values with one cursor per attribute. Because each particular attribute must be tested if it is the referenced or dependent attribute simultaneously, the challenge is to decide when the cursor for each attribute should be shifted. In general, the main idea is to process sorted attributes blockwise in parallel - value by value. In addition, each attribute maintains a list of IND candidates, which is decreasing during the process of checking attribute values. Consequently, the algorithm is based on a data structure which sorts all attribute values and stores equal values grouped together.

Example 25 (Placing of Values in the Data Structure of SPIDER): *The Table 3.3 demonstrates the way how values are stored in the data structure*

A	B	C
1		1
2	2	2
3		
4	4	4
		5

Table 3.3: SPIDER Data Structure

of SPIDER. The illustration is adopted from [53]. Attributes hold following values:

- A = {3, 2, 4, 1}
- B = {2, 4}
- C = {2, 5, 4, 1}

Representation of IND candidates

The authors utilize the following observation: IND candidates can be gathered into disjoint sets according to their dependent attribute. In other words, IND candidates covering a given dependent attribute determine a set. As a result, each attribute X preserves a list `X.refs` of attributes which holds IND candidates satisfied so far. To illustrate, $X \subseteq Y$ entails that $Y \in$ `X.refs`.

3.5.3 SPIDER Algorithm

The algorithm 4 demonstrates in pseudo-code the process broached in the foregoing section. It takes on the input attributes with their sorted values and their beginning `refs` lists and the output provides the set of satisfied INDs. SPIDER iterates row by row over the data structure proceeding the following steps:

1. (line 3) Obtain the set `curAtts` of all attributes holding given row's value, which means attributes with currently minimal equal value.
2. (line 5) Update lists `refs` of all attributes in `curAtts` by intersecting `refs` and `curAtts`. In other words, all attributes not containing the current minimal value (for instance, $B \notin$ `curAtts`) are excluded from `refs` of attributes which carry the current minimal value.
3. (line 6) Process the next value, attribute is re-inserted into the min-heap.

Algorithm 4 Spider

Input: attributes with their sorted values and their respective `refs` lists
Output: satisfied INDs

```
1: construction of heap using attributes
2: while heap is non-empty do
3:    curAtts ← attributes with current min. value
4:    for each A in curAtts do
5:       A.refs ← A.refs ∩ curAtts
6:       if A contains next value then
7:          A.shiftCursor()
8:          heap.add(A)
9:       else
10:         for each B ∈ A.refs − {A} do
11:            result ← result ∪ {A ⊆ B}
12:         end for
13:      end if
14:   end for
15: end while
16: return result
```

	cols	A.refs	B.refs	C.refs
Initialization:		B,C	A,C	A,B
Step 1:	A,C	C	A,C	A
Step 2:	A,B,C	C	A,C	A
Step 3:	A	∅	A,C	A
Step 4:	A,B,C	∅	A,C	A
Step 5:	C	∅	A,C	∅

Table 3.4: Example of SPIDER Run

3.5.4 Example of Parallel Run

Table 3.4 (adopted from [53]) visualizes in detail the process of obtaining INDs from data presented in Table 3.3. The column `atts` refers to attributes containing the value which is currently processing. When the first step is explained, the rest will have to be clear.

The first iteration step works with value `'1'`. Therefore, `curAtts = {A,C}` and the corresponding `refs` are updated as follows:

$$\text{A.refs} = \text{A.refs} \cap \text{curAtts} = \{\text{B}, \text{C}\} \cap \{\text{A}, \text{C}\} = \{\text{C}\}$$
$$\text{C.refs} = \text{C.refs} \cap \text{curAtts} = \{\text{A}, \text{B}\} \cap \{\text{A}, \text{C}\} = \{\text{A}\}$$

Consequently, IND candidates $A \subseteq B$ and $C \subseteq B$ are unsatisfied.

After processing all values, the algorithm provides those satisfied INDs $B \subseteq A$ and $B \subseteq C$.

3.5.5 Pruning Strategies

Simple Strategies

- If an attribute A carries more distinct values than attribute B, then the IND candidate $A \subseteq B$ can be excluded.
- An IND candidate can be also excluded if the maximum dependent value is greater than the maximum referenced value.
- In the same way, an IND candidate can be also excluded if the minimum dependent value is lower than the minimum referenced value.

All three tests are inexpensive, because the computations can be evaluated within the process of sorting the attributes. The authors have presented that these 3 strategies together reduce the beginning number of IND candidates by 80-90%.

Bloom Filter

The simple filters described above utilize only very little information about data. Bloom filters hash all values of an attribute into a bit-array. In order to prune IND candidates, bit-arrays of two attributes are compared trying to find bits that are 1 in the dependent array but 0 in the referenced array. If one such a bit is located, the candidate is not satisfied. Candidates which passed this test still require to be checked in the parallel run of SPIDER. The test between bit-arrays can be implemented efficiently by a bitwise $\texttt{dep} \wedge \neg\texttt{ref}$ operation. If the result of this operation is 1, the IND candidate can be excluded. The challenge is to set the optimal length of a bit-array, to find balance between the efficient pruning and the time required for bit-array testing. The authors have proposed in the face of many experiments that the length of 2^{17} bit (which requires 20MB memory for 1000 attributes) is a very good choice.

3.5.6 Composite INDs

GenNext Algorithm

The authors have been inspired by the GenNext algorithm presented in [52] (the summary is described in Section 3.6.4) to infer the IND candidates level-wise. The elicitation of INDs of level $l > 1$ is separated into two steps:

1. determination of all IND candidates which are chosen with respect to the satisfied INDs of level $l - 1$

2. test those IND candidates

This method is an adapted AprioriGen algorithm utilizing an order on attributes. It creates IND candidates of level l by sorting all satisfied INDs of level $l - 1$.

SPIDER

The SPIDER algorithm still test all IND candidates of a given level in parallel. The main modification is that the min-heap structure contains attribute tuples with their values instead of single attribute values.

Pruning Strategies

In the process of discovering composite INDs, two very weak restrictions are utilized. As far as later foreign key inference is concerned they should not exclude interesting INDs. However, the number of IND candidates is significantly reduced.

- The referenced attribute must consist of more than one distinct value
- At least 1% of all distinct values of the referenced attribute must be covered by values of the dependent attribute.

3.5.7 Conclusion

To conclude, SPIDER is the powerful method for discovering INDs where all data values are read only once. As far as unary INDs are concerned utilizing fastest possible SQL approach has number of comparisons $\boldsymbol{O(n^2 t \log t)}$ where n is a number of attributes and t entails maximum number of values in attributes. In this computation is used one join query per IND candidate assuming sort merge join on both attributes. In contrast to the SQL approach, SPIDER requires only $\boldsymbol{O(nt \log t)}$ comparisons, assuming $t > n$.

Results of experiments presented by the authors are impressive. Even very large databases are analyzed within hours. Utility for inferring foreign keys in XML is unquestionable, because within the process of locating relative foreign keys all data values could be read just once, only bigger number of `refs` lists would be held, and therefore number of intersections would be expanded. Moreover, when primary keys were first discovered, IND candidates in SPIDER could be significantly pruned.

3.6 IND - DBA Companion

3.6.1 Overview

In the paper [53] authors have described a different approach implemented in a DBA Companion project, which can detect all satisfied INDs in a given relational database. In addition to SPIDER, this is the only method which can compute all unary INDs during only one race through all the values. The idea is to create binary relations which connect every value of the database with attributes having this value. This new data organization so-called extraction context considers various data types and can be perceived as a transaction database in which attributes are items and values are transactions. In addition, unary INDs correspond to exact association rules and they can be extracted in one pass. Furthermore, the authors use a levelwise method based on Apriori algorithm to compute n-ary INDs.

3.6.2 Definitions

Basic relational database concept

Letters from the beginning of the alphabet means single attributes whereas letters from the end denotes attribute sets.

Let R be a finite set of *attributes* A.

- $\forall A \in R$, a *domain* ($Dom(A)$) represents the set of all possible values of A
- A *tuple* u over R is a total mapping $R \to \bigcup_{A \in R} Dom(A)$ where $u(A) \in Dom(A), \forall A \in R$
- A *relation* indicates a set of tuples over R, then R corresponds to its *relational schema*
- $|X|$ entails the cardinality of set X
- $u[X]$ denotes the restriction of u to X where $X \subseteq R$ is an attribute set and u is a tuple
- $\pi_x(r)$ means the *projection* of a relation r to X and is defined as $\pi_x(r) = \{u[X] | u \in r\}$
- A *database schema* $\mathbf{R}$ is a finite set relation schemas R_i
- A *relational database instance* $\mathbf{d}$ (or only **database**) over $\mathbf{R}$ represents a set of relations r_i over each R_i of $\mathbf{R}$

- An *attribute sequence* $X = ABC$ entails an ordered set of distinct values
- $X[i]$ refers to the i^{th} element of the sequence X
- Attributes A, B are *compatible* if $Dom(A) = Dom(B)$
- Distinct attribute sequences X, Y are *compatible* if $|X| = |Y| = m$ and for $j = [1, m], Dom(A) = Dom(B)$

Inclusion Dependency

Definition 19 (Inclusion Dependency)**.**

An inclusion dependency *over a database schema* $\boldsymbol{R}$ *is a statement of the form* $R_i[X] \subseteq R_j[Y]$ *where* $R_i, R_j \in \boldsymbol{R}$*,* $X \subseteq R_i$*,* $Y \subseteq R_j$*,* X, Y *are compatible sequences.*

- A *trivial* IND indicates that it is of the form $R[X] \subseteq R[X]$
- An IND $A \subseteq B$ is *of size* i if $|X| = i$.
- An *unary inclusion dependency* means an IND of size 1

Definition 20 (Inclusion Dependency Satisfaction)**.**

An IND $A \subseteq B$ *is* satisfied *in a database* $\boldsymbol{d}$ *over a database schema* $\boldsymbol{R}$*, denoted by* $\boldsymbol{d} \models R_i[X] \subseteq R_j[Y]$*, iff* $\forall u \in r_i, \exists v \in r_j$ *such that* $u[X] = v[Y]$ *where* $r_i, r_j \in \boldsymbol{d}$ *are relations over* $R_i, R_j \in \boldsymbol{R}$

3.6.3 Unary IND Discovery

The goal of the algorithm is to elicit all satisfied unary INDs in given a database. The core of the process is to set a binary relation between attributes and corresponding values.

A	B	C	D	E
3	X	4	Z	4
5	Y	5	U	3
3	X	7	X	4
3	Z	3	Z	4

Table 3.5: Example of Dataset

Data Preprocessing

Definition 21 (Extraction Context)**.** *Given a database* $\boldsymbol{d}$ *over a database schema* $\boldsymbol{R}$*, and a data type* t *of* $\boldsymbol{d}$*, an* extraction context *indicates the triple* $\mathbb{D}_t(\boldsymbol{d}) = (\mathbb{V}, \mathbb{U}, \mathbb{B})$ *as follows:*

- $\mathbb{U} = \{$ R.A | A *is of type* $\mathrm{t}, \mathrm{A} \in \mathrm{R}, \mathrm{R} \in \boldsymbol{R}\}$. $\mathbb{U}$ *is the set of attributes whose type is* t
- $\mathbb{V} = \{$ $\mathrm{v} \in \pi_{\mathrm{A}}(\mathrm{r})$ | $\mathrm{R.A} \in \mathbb{U}, \mathrm{r} \in \boldsymbol{d}$, r *defined over* R$\}$. $\mathbb{V}$ *is the set of values taken by attributes in their relations.*
- $\mathbb{B} \subseteq \mathbb{U} \times \mathbb{V}$ *is a binary relation:* $(\mathrm{v}, \mathrm{R.A}) \in \mathbb{B} \iff \mathrm{v} \in \pi_{\mathrm{A}}(\mathrm{r})$ *where* $\mathrm{r} \in \boldsymbol{d}$ *and* r *defined over* R.

Example 26: *The database depicted in Table 3.5 consists of 2 types :* int *and* string*. As far as the type* int *is concerned,* $\mathbb{U} = \{\mathrm{A}, \mathrm{C}, \mathrm{E}\}$ *and* $\mathbb{V} = \{3, 4, 5, 7\}$*. As a result,* (3, A), (3, C), (3, E), (4, C), (4, E), (5, A),$\ldots \in \mathbb{B}$*. The table 3.8 demonstrates the extraction context associated with both data types.*

$\mathbb{V}$	$\mathbb{U}$
3	A, C, E
4	C, E
5	A, C
7	C

$\mathbb{V}$	$\mathbb{U}$
U	D
X	B, D
Y	B
Z	B, D

Figure 3.8: Extraction Context of the Dataset in Table 3.5. Type *int* is shown on the right side and type *string* on the left.

Algorithm

The new data organization allows a new insight on the unary INDs. The satisfied IND $A \subseteq B$ entails that for each value v such that (v, A) belonging to the extraction context exists also (v, B), which is a part of the the extraction context, too. The following theorem is adopted from [52], its proof is obvious.

Theorem 2. *Given a database* $\boldsymbol{d}$*, a data type* t *and an extraction context* $\mathbb{D}_{\mathrm{t}}(\boldsymbol{d}) = (\mathbb{V}, \mathbb{U}, \mathbb{B})$*,*

$$\boldsymbol{d} \models \mathrm{A} \subseteq \mathrm{B} \iff \mathrm{B} \in \bigcap_{v \in \mathbb{V} | (v,A) \in \mathbb{B}} \{\mathrm{C} \in \mathbb{U} | (\mathrm{v}, \mathrm{C}) \in \mathbb{B}\}$$

where $\mathrm{A}, \mathrm{B} \in \mathbb{U}$*.*

The whole process is possible to imagine as a transaction database where attributes mean items and values refer to transactions. Then association rules whose confidence is 100% and whose left and right-hand sides are composed of only one attribute corresponds to unary INDs. The oncoming pseudo-code (Algorithm 6) is adopted from [52].

Algorithm 5 Unary IND discovery

Input: the extraction context $\mathbb{V}, \mathbb{U}, \mathbb{B}$
Output: the set of unary INDs satisfied by a given database

```
1: for each A ∈ 𝕌 do
2:    A.refs ← 𝕌
3: end for
4: for each v ∈ 𝕍 do
5:    for each A such that (v, A) ∈ 𝔹 do
6:       A.refs ← A.refs ∩ {B|(v, B) ∈ 𝔹}
7:    end for
8: end for
9: for each A ∈ 𝕌 do
10:   for each B ∈ A.refs − {A} do
11:      result ← result ∪ {A ⊆ B}
12:   end for
13: end for
14: return result
```

Each attribute A preserves a list A.`refs` of attributes which holds IND candidates satisfied so far. To illustrate, $A \subseteq B$ entails that $B \in$ A.`refs`. This algorithm is linear in relation to the size of the binary relation of the extraction context, because the size of A.`refs` is smaller than the size of $\mathbb{U}$ by orders of magnitude. As a result, the intersection can be treated as a constant.

First, the initialization process is performed on the lines 1-3 and all `refs` are filled. Second, the main computation is proceeded on the lines 4-8. Each value of attribute A is checked by intersection whether it is included in other attributes. Third, the rest of the method provides the fill of the output variable with the inferred INDs.

Example

Table 3.6 visualizes in detail the process of obtaining INDs of the data type *int* from the extraction context presented in Table 3.8. The column `value` contains the currently processed value from $\mathbb{V}$ and the column `atts` refers to the attributes holding the value from the foregoing column. When the second step is explained, the rest will have to be clear.

	value	atts	A.refs	C.refs	E.refs
Initialization:			A,C,E	A,C,E	A,C,E
Step 1:	3	A,C,E	A,C,E	A,C,E	A,C,E
Step 2:	4	C,E	A,C,E	C,E	C,E
Step 3:	5	A,C	A,C	C	C,E
Step 4:	7	C	A,C	C	C,E

Table 3.6: Example of the run

The second iteration step works with value of `'4'`. Therefore, attributes $\{C, E\}$ are considered and the corresponding `refs` are updated as follows:

$$\text{C.refs} = \text{C.refs} \cap \{\text{C}, \text{E}\} = \{\text{A}, \text{C}, \text{E}\} \cap \{\text{C}, \text{E}\} = \{\text{C}, \text{E}\}$$
$$\text{E.refs} = \text{C.refs} \cap \{\text{C}, \text{E}\} = \{\text{A}, \text{C}, \text{E}\} \cap \{\text{C}, \text{E}\} = \{\text{C}, \text{E}\}$$

Consequently, IND candidates $C \subseteq A$ and $E \subseteq A$ are unsatisfied.

After processing all values, the algorithm provides those satisfied INDs $A \subseteq C$ and $E \subseteq C$.

3.6.4 Composite IND Discovery

The oncoming method is an apriori-like approach, but first the search space must be reduced in order to be able to utilize the method efficiently.

Preprocessing

Theorem 3 (Projection and Permutation of INDs)**.** *if* $\text{R}[\text{A}_1, \ldots, \text{A}_\text{n}] \subseteq \text{S}[\text{B}_1, \ldots, \text{B}_\text{n}]$ *then* $\text{R}[\text{A}_{\delta 1}, \ldots, \text{A}_{\delta \text{m}}] \subseteq \text{S}[\text{B}_{\delta 1}, \ldots, \text{B}_{\delta \text{m}}]$ *for each sequence* $\delta 1, \ldots, \delta \text{m}$ *of distinct integers* $\{1, \ldots, n\}$.

Thanks to this rule all the permutations to create a left-hand side or a right-hand side of INDs do not have to be considered.

Definition 22 (Relation between Candidate INDs)**.** *Let* I_1 *:* $\text{R}_\text{i}[\text{X}] \subseteq \text{R}_\text{j}[\text{Y}]$ *and* I_2 *:* $\text{R}'_\text{i}[\text{X}'] \subseteq \text{R}'_\text{j}[\text{Y}']$ *be two candidate INDs. Then* $\text{I}_2 \lhd \text{I}_1$ *is defined iff:*

- $\text{R}_\text{i} = \text{R}'_\text{i}$ *and* $\text{R}_\text{j} = \text{R}'_\text{j}$ *and*
- $\text{X}' = \text{A}_1 \ldots \text{A}_\text{k}$, $\text{Y}' = \text{B}_1 \ldots \text{B}_\text{k}$ *and there exists a set of indexes* $\text{i}_1 < \ldots < \text{i}_\text{h} \in \{1, \ldots, \text{k}\}$ *with* $\text{h} \leq \text{k}$ *such that* $\text{X} = \text{A}_{\text{i}_1} \ldots \text{A}_{\text{i}_\text{h}}$, $\text{Y} = \text{B}_{\text{i}_1} \ldots \text{B}_{\text{i}_\text{h}}$

Theorem 4. *Let* I_1, I_2 *be two candidate INDs such that* $\text{I}_1 \lhd \text{I}_2$*. If* $\boldsymbol{d} \not\models \text{I}_1$*, then* $\boldsymbol{d} \not\models \text{I}_2$.

In other words, this property entails that the relation $\lhd$ is *anti-monotony* with respect to the satisfiability of INDs. Generally, only satisfied INDs will be utilized to produce candidate INDs for the subsequent level. As a result, the search space is significantly pruned.

Algorithm 6 Composite IND discovery

Input: I_1 - the set unary INDs satisfied by a given database **d**
Output: the set of composite INDs satisfied by a given database **d**

```
1: candidates ← GenNext(I_1)
2: while candidates not empty do
3:    for each I in candidates do
4:       if d ⊨ I then
5:          toGen ← toGen ∪ I
6:       end if
7:       result ← result ∪ I
8:    end for
9:    candidates ← GenNext(toGen)
10:   empty toGen
11: end while
12: return result
```

Algorithm

The algorithm (demonstrated in Algorithm 6) is quite simple. In the first phase, the candidate INDs of size 2 are computed by the GenNext function on basis of satisfied INDs of size 1. Next, these candidates are checked against the database and the satisfied ones are utilized in the generation of IND candidates of level 3. This process is repeated until GenNext produces new candidates for evaluation.

The GenNext function extends the Apriori principle of the frequent itemsets discovery. The function is composed of two steps - generation phase and pruning phase, both parts results from the relation $\lhd$ and its *anti-monotony* property.

3.6.5 Conclusion

To conclude, the described algorithm is the powerful method for discovering INDs where all data values are read only once. It is possible to observe some similarities with SPIDER, however, the main data structure tha,t holds attributes with values and performs the computation process, is totally different. In contrast to SPIDER, the method from DBA Companion considers data types. But on the other side, it involves apparently the more time-consuming preprocessing phase which assigns to each value in a database the list of attributes holding this value. This process is very costly, because all values of all attributes must be combined into one data structure. Authors of SPIDER have presented in [53] that their approach outperforms the method from DBA Companion by orders of magnitude. Also some pruning of IND candidates within the preprocessing phase would

helpful. For instance, the pruning techniques of SPIDER can be very effective.

3.7 Summary of Approaches

This section summarizes in Table 3.7 the main characteristics of algorithms mentioned in the foregoing sections. Moreover, in Table 3.8 are concluded their important advantages and disadvantages. RDB means relational database.

Name	**Output**	**Requirements**
Gordian	RDB Composite PKs	data in tuples
XML Primary Keys	XML PKs	XML document
Combined Approach	RDB FKs	metadata to obtain PKs, queries, data types
Spider	Composite INDs	sorted column lists
DBA Companion	Composite INDs	binary relation between values and attributes

Table 3.7: Main Characteristics of the approaches

Name	Description
Gordian	very fast detection of all composite primary keys, simple to expand for the purpose of relative primary keys discovery, the process can not be quickened in order to detect only composite 2-keys or 3-keys
XML Primary Keys	very precise, but ineffective, inference keys can involve '?' wildcard in their specification, a minimal cover for the set of XML keys of a given XML tree
Combined Approach	according to the authors, the most precise and effective RDB to XML translational method, mix of several approaches (data-driven, query-driven, ontologies), each of them is very simple and straightforward utilized
Spider	computation of all unary INDs within only one run through all the values, efficient pruning techniques of IND candidates
DBA Companion	computation of all unary INDs within only one run through all the values, considers data types, in comparison with SPIDER very time-consuming preprocessing

Table 3.8: Brief summary of the approaches

Chapter 4

New Proposal - KeyMiner

4.1 Overview

The core of KeyMiner are algorithms Gordian and Spider, both are expanded in order to detect relative primary keys and relative foreign keys in XML data. The main procedure is following:

1. Discover primary keys.
2. Determine foreign key candidates in such a way that the referenced part of the candidate FK expression must be one of the discovered primary key. In addition, there must be a high chance for the inclucion dependency.
3. Elicitation of foreign keys by evaluating the inclusion dependencies of the foreign key candidates from the previous step.

4.1.1 Main Features

- Inference of all n-ary primary keys that conform the characterization presented in Section 4.1.2 - absolute as well relative. Generally, the most of keys have in practice key degrees at most 2. Consequently, KeyMiner detects also only unary and 2-key expressions, nevertheless mining higher key degrees would not be more time consuming.
- Elicitation of all unary foreign keys that meet point 5 in Section 4.1.2, by satisfying the inclusion dependency property with a primary key - absolute as well as relative. With the knowledge of primary keys, mining foreign keys with key degree equal 2 should not cost much more time. But similarly to the previous case, inference of these FKs is not very interesting.
- The main advantage of KeyMiner is probably the very effective acquisition of relative primary keys (RPK) and relative foreign keys (RFK).

The nature of the algorithm provides the following feature: In the scope of detection of absolute n-ary primary keys/ absolute foreign keys (which is extremely fast itself), the inference of RPK/RFK does not need much more time. In principle, the values are not processed again.

- Detection of PKs/ FKs containing only XPath expressions without any wildcards as, for instance, `'?'`, or `'*'`.
- As a result of the previous point and the features presented in Section 4.1.2, the algorithm does not generate redundant key expressions. It can find more candidates to a key path within the process of evaluating the current context path/target path pair. If KeyMiner do not distinguish the right key path, it will inform the user about that.
- The user defines *support* - see Definition 33 in Section 4.2.

4.1.2 Features Based on the Inference Rules

The following features are based on the inference rules mentioned in Section 2.3.3. These characteristics relates only to the primary key candidates (except the point 5 which relates to FK candidates, too) - according to the definition, each consist of a context path, a target path and one or more key paths (number of key path entails key degree). The foreign key candidates hold expressions in accordance to the discovered primary keys.

1. The key degree is as small as possible. (*superkey rule*)
2. The target path is as long as possible. (*subnodes rule*)
3. With respect to the previous point, the key path should be as short as possible. However, in practice key path mostly consists of at least one element or an attribute. As a result, the detected key expressions in KeyMiner always include key paths with one element or an attribute.
4. The context path should be as short as possible. (*context-target rule*) Therefore, firstly candidates for absolute keys are evaluated. Then if no one is satisfied, the computation process continues in such a way that in each iteration step the candidates holding currently the possible shortest context path is examined until the key is confirmed.
5. The target path and the context path are as general as possible. (*context-path-containment rule, target-path-containment rule*) Thus, KeyMiner does not consider XPath expressions in keys like, for example,
   `/volume/articles/article[2]`, or
   `/volume/articles/article[@posi="13 and autor="Tom"]`. In other

words, the discovered relative primary keys/ relative foreign keys must be satisfied in each subtree of an XML tree determined by the context path. In case of foreign keys, inclusion dependency must be confirmed separately in each subtree of an XML tree.

4.2 Preliminaries

As it was noticed in the point 5 of Section 4.1.2, all mentioned XPath expressions are simple without any wildcards or any other special characters. Moreover, all mentioned path expressions do not contain `'?'` or `'*'`, too.

Definition 23 (Path of element)**.** *Let* e *be an element of an XML tree* T*. Then* path *of element* e*, denoted as* $[P_e]$*, represents the path expression* P *which starts in the root of* T *and ends in* e*.*

4.2.1 Main Database Terms

The following definitions of the *relation*, the *tuple* and the *column* are analogous to the main database terms of the table, the table row and the table column.

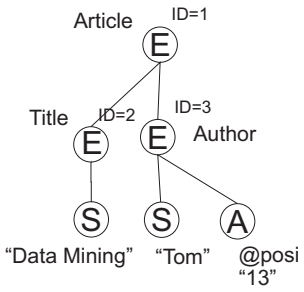


Figure 4.1: Small XML Tree

Definition 24 (Tuple)**.** *A* tuple *is an element* e *of an XML tree and it is formally denoted as a two-tuple* $e = (id, V)$ *where* id *stands for an unique element identifier in the respected XML tree and* V *means a set of pairs:* $\{attr_1 = value_1, \ldots, attr_n = value_n\}$*. Then* $attr_i$ *represents either an attribute of the element* e *and* $value_i$ *represents the value of that attribute, or* $attr_i$ *refers to a child element* c *which includes a text, and* $value_i$ *refers to the text of* c*.*

Example 27 (Tuple): *Figure 4.1 contains these* tuples*:*

- *(*`id = 1`*,* $V = \{$`Title = "Data Mining", Author = "Tom"` $\}$

- *(*`id` $= 2$, $V = \epsilon$*)*
- *(*`id` $= 3$, $V = \{$`posi = "13"`$\}$*)*

Definition 25 (Column)**.** *A* column c *is a set of values which correspond to an XPath expression* X*.*

- *If* X *aims a set of attributes, then* c *contains the values ot these attributes.*
- *Or if* X *locates a set of elements with a content then* c *holds as values the contents of these elements.*
- *Otherwise,* c *is empty.*

Example 28 (Column): *Figure 4.4 contains, for instance, the following* columns*:*

- *XPath expression =* `/Issue/Volume`
  values $= \{$`"11", "12"`$\}$
- *XPath expression =* `/Issue/Articles/Article/Author`
  values $= \{$`"Tom", "Jeremy", "Jacob"` $\}$
- *XPath expression =* `/Issue/Reports/Report/Author`
  values $= \{$`"Jane"` $\}$
- *XPath expression =* `/Issue/Articles/Article/Author/posi`
  values $= \{$`"13", "10", "19"` $\}$

Definition 26 (Relation)**.** *A* relation *entails to a set of* tuples *which corresponds to an XPath expression aiming elements. Consequently, first parts of the pairs* $attr_i = value_i$ *from* V *of the tuples correspond to* columns*. As a result, each* column *is part of a particular* relation*.*

Example 29 (Relation): *The following tables shows 2 examples of relations which are represented as XPath expressions ('I' is a shortcut for* `Issue`*) and obtained from the XML tree depicted in Figure 4.4.*

Volume	Number
11	1
11	2
12	1

Figure 4.2: /Issue

Title	Author	Pages
Data Mining	Tom	10
UML	Jeremy	16
OOP	Jacob	16

Figure 4.3: /I/Articles/Article

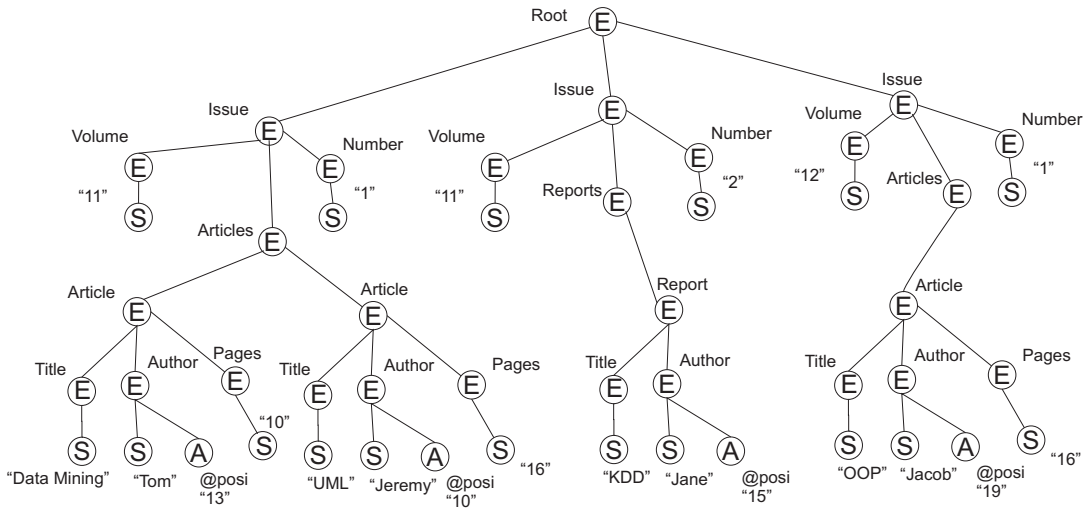


Figure 4.4: Big XML Tree

4.2.2 Terms Concerning Inference of Relative Keys and Support

For the reasons of the inference of relative keys, some support to manipulation with subtrees and positioning of elements into the respected subtrees is required. Definitions 30 and 31 stands for columns as well as for relations.

Definition 27 (trav)**.** *Let* x, y *be elements specified by the same XPath expression and* $[P_x], [P_y]$ *are paths of the elements* x, y*. Then* trav(x, y) *denotes to the number* $n = (\#\text{elements in } P_x) - (\#\text{elements in } P_x \cap P_y)$*.*

Example 30: *For the XML tree visualized in Figure 4.4 and the elements specified by the XPath expression* `/Issue/Articles/Article/Title` *stands:*

- *trav(* `Title = "Data Mining"`, `Title = "UML"` *) =* ***1***
- *trav(* `Title = "Data Mining"`, `Title = "OOP"` *) =* ***3***

Definition 28 (Level of XML Tree)**.** *The* levels *of an XML tree T are numbers which specify disjoint sets of elements of T. When an element* e *is located on the* level n *of* T*, the path expression specifying* e *contains* n *elements (not including the root element).* Level *0 holds only the root element* r *of T and* level 1 *consist only of the child elements of* r*.*

Definition 29 (Level of Relation and Column)**.** *Let* $S = \{e_1, \ldots, e_n\}$ *represent the set of elements of the relation* r *of the XML tree T and* $P = \{[P_{e_1}], \ldots, [P_{e_n}]\}$ *be the set of paths of the elements in* S*. Let A be an XML tree which is constructed from the nodes situating in the path expressions of* P*. Then* levels *of* r *correspond to the* levels *of A. Furthermore,*

Levels *of a column* c *of* r *correspond to the* levels *of* r*. The set of elements specified by a* level k *of* r *is denoted as* r[k]*, the same with columns.*

Definition 30 (Branching)**.** *Let* r *be a relation.* Branching *of level* k *of* r *is denoted as* branch(r[k]) *and returns* true *or* false*. The positive return value appears when an element* `e` *in located on* r[k] *and paths of some elements of* r *are being divided in* `e`*.*

Definition 31 (Forked Nodes)**.** *Let* r *means a relation and* k *its level. Then* forked nodes *(denoted as* frnodes(r[k])*) is a mapping from* r[k] *into to the set* S *of elements. If* branch(r[k]) = true*, then* frnodes(r[k]) = r[k]*, otherwise* frnodes(r[k]) = r[l] *where* l *is the first level such that* l > k; branch(r[l]) = true*.*

Definition 32 (Values of Subtree)**.** *Let* f *be a forked node of a column* c*. Then* values of a subtree *specified by* f *of* c *(denoted as* c[f]*) entails a subset of values of* c *whose elements (or attributes)* $e_1, \ldots, e_n$ *are positioned on the path expressions* $[P_{e_1}], \ldots, [P_{e_n}]$ *including* f*.*

In summary, having a column c and notation *c[x]*. Then when x is a number, *c[x]* represents the set of elements specified by the level x of c. Or else if x is an element identifier *c[x]* corresponds to the values of the subtree specified by the element x.

Definition 33 (Support)**.** Support *is a number declared by the user. Let* support *be equal to* n*. Each set of values must consist of at least* n *values, in order to be able to contain a primary key or a foreign key. The number of tuples (values) in a relation, which is checked against* support*, depends on the length of the context path (the level of the relation from which the key expression is derived). The set* S *containing* k *values* is supported *if* k ≥ n*.*

Definition 34 (Support of Level)**.** *Let* r *means a relation,* k *its level and* s *support declared by the user. The notion* support(r[k]) *specifies if the level* k *of* r is supported*. And that is untruth when* ∃n ∈ r[k]*; the number of values in* r[n] *is lower than* s*.*

Example 31 (Support): *Let* support *= 3. Key expressions are derived from the relation* r *specified as* `/Issue/Articles/Article/Author` *and basically, then context path of these key expressions corresponds to a level of the relation. According to the XML tree demonstrated in Figure 4.4, the first three key expressions can not be keys, but the last one is supported. For instance, the third one has number of values = 1 because the whole right branch of the XML tree contains only one value.*

1. *(* `Issue.Articles.Article`*, (*`Author`*,* {`posi`}*))* *#values =* ***1***
2. *(* `Issue.Articles`*, (*`Article.Author`*,* {`posi`}*))* *#values =* ***1***

3. (`Issue`*, (*`Articles.Article.Author`*,* {`posi`}*))* *#values =* ***1***

4. (ϵ*, (*`Issue.Articles.Article.Author`*,* {`posi`}*))* *#values =* ***3***

In consequence, r[0] *is not supported while* r[1] *is not.*

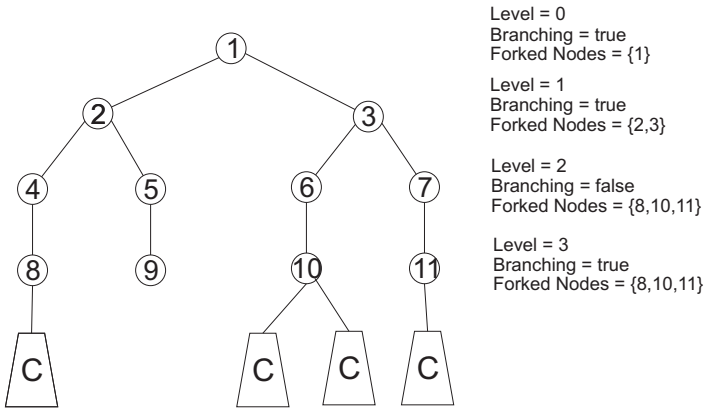


Figure 4.5: Simplified XML Tree

Example 32 (Level of Column, Branching, Forked Nodes): *Figure 4.5 visualizes the XML tree which contains in some subtrees values of a column* C *and demonstrates the corresponding levels, branchings and forked nodes of the column* C*.*

4.2.3 Non-Keys

The basis of the primary key discovery in KeyMiner is the detection of non-keys. Therefore, they must be properly formalized.

Definition 35 (Non-Key)**.** *A* non-key *is represented as two-tuple* N = {r, K} *where* r *denotes a relation with the set* C = $\{c_1, \ldots, c_n\}$ *of column identifiers and* K $\subseteq$ C*.*

Definition 36 (Non-Key Coverage)**.** *Let* r *be a relation and* a = (r, K) , b = (r, L) *stand for non-keys* a, b*. Then* a covers b *if* K $\supseteq$ L*.*

Definition 37 (Non-Key Redundancy)**.** *Let* b *be a non-key and* S *be a set of non-keys. Then* b is redundant *in* S *when* $\exists$a $\in$ S*;* a covers b*.*

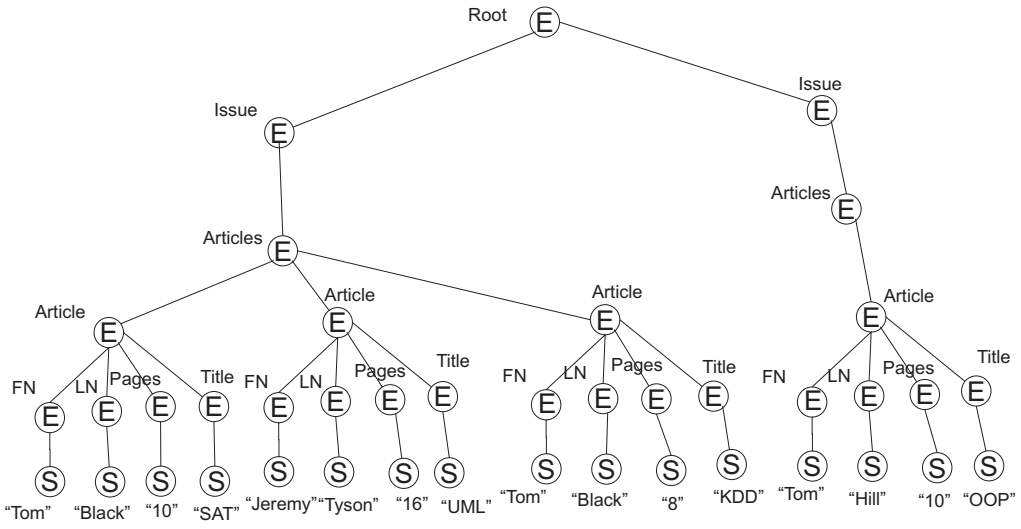


Figure 4.6: Example of XML Tree to Show Construction of P-Tree

4.3 Inference of Primary Keys

4.3.1 Overview

Firstly, XML data are traversed in order to obtain all possible relations to evaluate. These two types of columns are excluded for now and they will be considered only in the course of foreign key inference.

- columns which are not situated in each tuple of a relation
- columns holding in one tuple more than one value

Next, the computation process for each relation is demonstrated in Algorithm 7. In the first phase, the set of so-called prefix-trees is constructed. Then non-keys on all levels of relation are elicited. And finally, the primary keys are computed from received non-keys.

Algorithm 7 Main Run

Input: sequence of relations with corresponding *trav*
Output: set of discovered keys

1: **for** each *relation* **do**
2: create set of prefix-trees
3: find non-keys
4: compute keys from non-keys
5: **end for**

4.3.2 P-tree

A *prefix-tree* (PT) is a data structure, which facilitates powerful merging a pruning steps within the elicitation process. Generally, PT looks like a simplified XML tree, but each *level* of PT corresponds to one column of a relation (and this *level* is identified by `ColNumber`). The main entities are named *nodes* and they are composed of *cells* that are able to reference to other *nodes*. Every node as well as cell includes the variable `isModified` that signals whether current node (or cell) is a part of a non-key and must be processed. In addition, root nodes contain `trav` number (`travs`) which inform about the position of the contiguous root node in the original XML tree (the root node in PT and its position in the XML tree). `Travs` are quantified in accordance to tuples of relations. They are important because they record the tree structure of the respective relation.

EXAMPLE 33: *Figure 4.7 visualizes the prefix tree created from the XML tree depicted in Figure 4.6.* `Trav` *numbers are symbolized in parentheses,* `ColNumber` *in brackets and* `isModified` *by values* ***`true/false`****.*

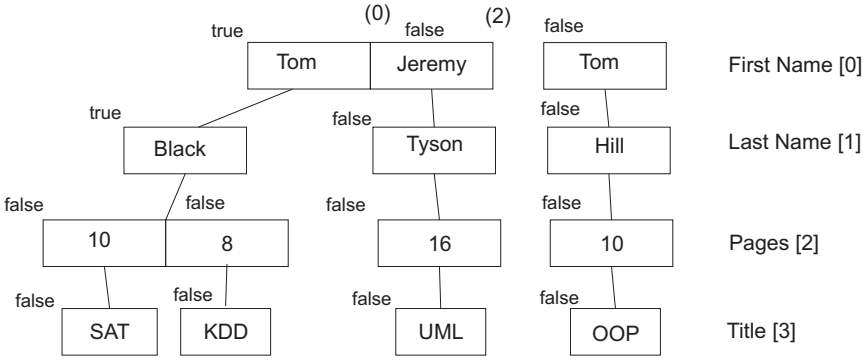


Figure 4.7: Example of P-Tree

Construction of P-tree

Algorithm 8 demonstrates the whole operation. The input is a sequence S of tuples (they must be in the same order as in the XML tree) with corresponding `travs`. S is divided into disjoint sets $D_1, \ldots, D_n$ - borders are `Travs` > 0 (lines 21, 22). Next, each tuple is added as a new tree (line 20). The only exception occurs when two root nodes with the same value have appeared in D_i (line 4). Then *merging* of these nodes is performed on the lowest level of XML tree - one tree is discarded and children of these root nodes are attached into one cell. If the children (next column) hold the same value as well, the merging process continues (see Figure 4.7) (lines 4-17). If all columns of 2 tuples are *merged* (it means 2 tuples with identical values

in all columns), then the whole computation process is finished (lines 9, 10), because this relation can not contain any primary key. Cells (and their superior nodes) which consist of values that have appeared more than once are marked as modified (their variable `isModified` equals `true`). Otherwise, they are signed as not modified.

Algorithm 8 PT Build

Input: *Relation*: sequence of tuples with corresponding *trav*
Output: *Result*: sequence of built prefix trees

```
 1: CurValues ← new empty set for values
 2: CurTrav ← −1
 3: for each Tuple in Relation do
 4:   if (CurTrav is 0) and (Tuple[0].Value ∈ CurValues) then
 5:     FoundNode ← root node of already built tree containing in the
        root Tuple[0].Value
 6:     for each Column in Tuple do
 7:       if FoundNode includes a Cell with the same value as
          Column.Value then
 8:         if FoundNode is a leaf node then
 9:           report that no key can be found
10:           return empty set
11:         end if
12:         mark FoundNode and Cell as modified
13:         FoundNode ← Cell.child
14:       else
15:         attach Column as a cell into FoundNode
16:       end if
17:     end for
18:     build branch from the rest of Columns in Tuple starting at already
        attached point and mark all its nodes and cells as not modified
19:   else
20:     Result ← newly created tree of Tuple and mark all its nodes and
        cells as not modified
21:     if CurTrav ≠ 0 then
22:       empty CurValues
23:     end if
24:     CurValues += Tuple[0].Value
25:   end if
26:   CurTrav ← Tuple.Trav
27: end for
28: return Result
```

4.3.3 Searching for Non-Keys

The current algorithm takes the sequence of constructed prefix trees from the previous function as the input and fills a non-key container `NonKeySet` in the function `TraverseTree`. It is presented in Algorithm 9. Each call of `TraverseTree` produce new set of non-keys in `NonKeySet`. The whole process proceeds in accordance to the structure of the original XML tree. It iterates from the bottommost level of the XML tree level by level and infers non-keys in each set of tuples of the respected subtree. The oncoming example demonstrates it very clearly. If the current relation does not include branching at the current level, then searching for non-keys is needless on this level, because there exist no subtrees to evaluate (lines 5,6). When a non-key covering all the columns of the current relation is detected on level k, then obviously each level $j <= k$ can not contain any primary key and level k is stored into the variable `FirstLevNoKeys` (lines 19-20). Moreover, there is no need to merge within the first iteration step, because merging is performed during creation of a prefix tree (line 10). The functions `Merge` and `TraverseTree` are explained in the next sections.

The function PrepareTrees(`CurTrav`, `Trees`) is very simple. It takes the sequence of trees `Trees` and joins them into disjoint supersets. If `Trav` < `CurTrav`, then corresponding root nodes are joined into a set.

EXAMPLE 34 (Non-Key Inference): *Figure 4.8 depicts an example of searching for non-keys in the original XML Tree.* $\mathrm{T_i}$ *denotes set of trees and* $\mathrm{N_i}$ *corresponds to non-keys discovered so far. Assume that all non-keys in this example do not cover any other* $\mathrm{N_i}$*. After computing non-keys* $\mathrm{N_1}, \ldots, \mathrm{N_5}$*, the next major steps of the algorithm would be following:*

- *Preparation and merge* $\mathrm{T_6}$ *(from* $\mathrm{T_1}, \mathrm{T_2}$*) and* $\mathrm{T_7}$ *(from* $\mathrm{T_3}, \mathrm{T_4}$*).*
- *Sending discovered non-keys* $(\mathrm{N_1}, \mathrm{N_2} \rightarrow \mathrm{T_6})$ *and* $(\mathrm{N_3}, \mathrm{N_4} \rightarrow \mathrm{T_7})$*.*
- *Elicitation of new non-keys* $\mathrm{N_6}$ *in* $\mathrm{T_6}$ *and* $\mathrm{N_7}$ *in* $\mathrm{T_7}$*.*
- *Preparation and merge* $\mathrm{T_8}$ *(from* $\mathrm{T_6}, \mathrm{T_7}$*).*
- *Sending discovered non-keys* $(\mathrm{N_1}, \mathrm{N_2}, \mathrm{N_3}, \mathrm{N_4}, \mathrm{N_6}, \mathrm{N_7} \rightarrow \mathrm{T_8})$*.*
- *Elicitation of new non-key* $\mathrm{N_8}$ *in* $\mathrm{T_8}$*.*
- *Preparation and merge* $\mathrm{T_9}$ *(from* $\mathrm{T_8}, \mathrm{T_5}$*).*
- *Sending discovered non-keys* $(\mathrm{N_1}, \ldots, \mathrm{N_8} \rightarrow \mathrm{T_9})$*.*
- *Elicitation of new non-key* $\mathrm{N_9}$ *in* $\mathrm{T_9}$*.*

Algorithm 9 Search for Non-Keys

Input: $Trees$: sequence of prefix trees

Output: inferred non-keys stored in the function $TraverseTree$

```
1:  FirstLevNoKeys ← −1
2:  CurTrav ← 1
3:  for each possible Level of current relation Relation commencing at the
    maximum (bottommost level of the XML tree) and ceasing with level=0
    (root) do
4:    CurTrav++
5:    if branch(Relation[Level]) is false then
6:      continue with the next item of the for-loop
7:    end if
8:    SeqOfSetsOfTrees ← PrepareTrees(CurTrav, Trees)
9:    for each Set in SeqOfSetsOfTrees do
10:     if CurTrav > 2 then
11:       empty Trees
12:       Merged ← Merge(Set)
13:       pass founded non-keys from previous level
14:       Trees += Merged (new sequence is being produced)
15:       TraverseTree(Merged, 0)
16:     else
17:       TraverseTree(Set, 0)
18:     end if
19:     if discovered two tuples identical in all values then
20:       FirstLevNoKeys ← Level
21:       return
22:     end if
23:   end for
24:   if all prefix trees are merged into one tree then
25:     pass founded non-keys to all levels which are closer to the root
26:     break
27:   end if
28: end for
```

4.3.4 Function Merge

The `Merge` function is utilized in the Algorithm 9 as well as in the function `TravrseTree`. In fact, it is almost the same as the one in Gordian, that has been already described in Section 3.2.3. The major difference is that the new `Merge` function does not use `counter` in leaf nodes. Nevertheless, it utilizes the variable `isModified` which is included in each cell and node. Consequently, when new `Merge` discovers identical values, cells holding these values are marked as modified (`isModified` = true) and nodes containing

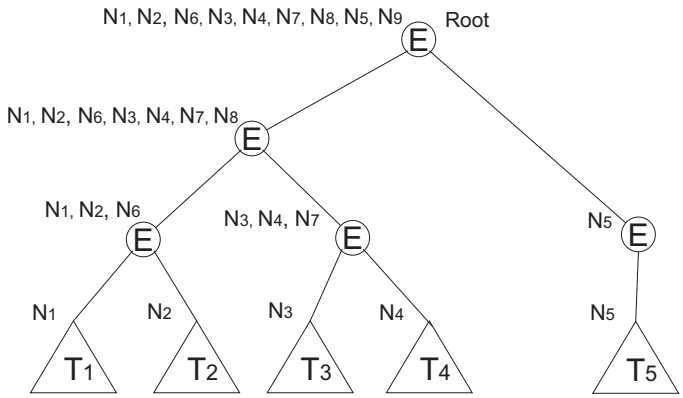


Figure 4.8: Process of Non-Key Inference on the Original XML Tree

that cells are signed as modified, too. The second difference is that `Merge` in KeyMiner finishes the function when a merged leaf cell is discovered (like in the function creating PT), which can be signifficant quickening in some cases.

4.3.5 Function TraverseTree

The function provides the double DF-traversal of PT with very effective pruning combined with merged functionality and stores discovered non-keys into `NonKeySet`. The main idea behind is the exploration of all slices and their segments - the theory is introduced in Section 3.2. First, this main principle is demonstrated on the simplified version - Algorithm 10. Inspection of all slices is provided on the line 5 and exploration of all segments on the line 8.

Algorithm 10 simplified TraverseTree

Input: *Root*: root node of PT

```
 1: if Root is leaf then
 2:    detect and store non-keys
 3: else
 4:    for  each Cell in Root do
 5:       TraverseTree(Cell.child)
 6:    end for
 7:    MergedTree ← merge children of Root
 8:    TraverseTree(MergedTree)
 9:    discard MergedTree
10: end if
```

The original version of the whole function is precisely described in Section

3.2.2. On the contrary to the merging function, `TraverseTree` contains a bit more modifications and improvements. Thus, Algorithm 11 is presented. Probably, the best upgrade is utilizing of the variable IsModified which provides more effective pruning. Then the checking of futility pruning is performed becomingly via the global variable `ToCheckFutility`. In the beginning of recursion, it is set to hold all columns of the current relation. And within the computation process it always holds the most general non-key for the given state of recursion. Then the verification if `CurNonKey` is futile returns true if exists already elicited non-key which covers `ToCheckFutility`. All operations with non-keys (and primary keys in Section 4.3.6) are implemented as bitwise operations on arrays of bits where each bit corresponds to one `ColNumber`. For instance, a, b entails non-keys, then a *covers* b when $\neg(a \vee \neg b)$ equals 0.

Example of the run is not necessary, because all basic principles are in detail demonstrated in the example in Section 3.2.5.

4.3.6 Compute Keys from Non-Keys

In principle, the procedure, presented in Algorithm 12, is evaluating primary keys with reference to the points described in Section 4.1.2. For example, in contrast with the algorithm computing non-keys, the current algorithm searches from upper levels of the original XML tree and when a primary key is detected and satisfied, the procedure is finished (point 4). Functions `FindKeysInSet` and `EvaluateKeys` are the essential parts there.

Function FindKeysInSet

This method is same as the one demonstrated in Section 3.2.6. But for the reasons of implementation of keys as arrays of bits, the elimination of redundant keys (line 12) is automatic.

Function EvaluateKeys

The present function supervises the fulfilment of points 1, 5 from Section 4.1.2

- If `EvaluateKeys` observes an unary PK for `NoOfSubtreesToCheck` times, then it stores all unary PKs occurring `NoOfSubtreesToCheck` times and returns `true`.
- Else if `EvaluateKeys` notices an PK with key degree = 2 for `NoOfSubtreesToCheck` times then it stores all PKs with key degree = 2 occurring `NoOfSubtreesToCheck` times and returns `true`.
- otherwise, it returns `false`

Algorithm 11 TraverseTree

Input: *Root*: root node of PT, $ColNumber$: attribute number
Output: inferred non-keys stored in $NonKeySet$

```
1: CurNonKey += ColNumber
2: if Root is leaf then
3:   if Root.IsModified is true then
4:     Root.IsModified ← false
5:     if CurNonKey holds ColNumber of all columns of the current
       relation then
6:       report that 2 tuples are identical in all values
7:       leave recursion
8:     end if
9:     NonKeySet ← CurNonKey
10:  end if
11:  CurNonKey −= ColNumber
12:  if Root contains more than 1 cell then
13:    NonKeySet ← CurNonKey
14:  end if
15: else
16:  if Root.IsModified is true then
17:    Root.IsModified ← false
18:    for each Cell in Root do
19:      if Cell.IsModified is true then
20:        Cell.IsModified ← false
21:        TraverseTree(Cell.child, ColNumber + 1)
22:      end if
23:    end for
24:  end if
25:  CurNonKey −= ColNumber
26:  if Root contains more than 1 cell then
27:    if any discovered non-key covers ToCheckFutility (futility control)
       then
28:      return
29:    end if
30:    ToMerge ← distinct children of all cells from Root
31:    ToCheckFutility −= ColNumber
32:    MergedTree ← Merge(ToMerge)
33:    TraverseTree(MergedTree, ColNumber + 1)
34:    ToCheckFutility += ColNumber
35:    discard MergedTree
36:  end if
37: end if
```

Algorithm 12 Compute Primary Keys from Non-Keys

Input: $NonKeySet$: set of discovered non-keys
$FirstLevNoKeys$: first level in the original XML tree that holds no primary keys for sure

Output: inferred primary keys stored in the function $EvaluateKeys$

```
1: create KeysOfTheCurLevel
2: for each level Level > FirstLevNoKeys of the current relation Relation do
3:   empty KeysOfTheCurLevel
4:   if (branch(Relation[Level]) is false) or (support(Relation[Level]) is false) then
5:     continue with the next item in the for-loop
6:   end if
7:   NoOfSubtreesToCheck ← count of elements in Relation[Level]
8:   for each SetOfNonKeys on the level Level in NonKeySet do
9:     if SetOfNonKeys is empty then
10:      NoOfSubtreesToCheck − −
11:    end if
12:    KeysOfTheCurLevel += FindKeysInSet(SetOfNonKeys)
13:  end for
14:  if EvaluateKeys(KeysOfTheCurLevel, NoOfSubtreesToCheck) is true then
15:    return
16:  end if
17: end for
```

4.4 Inference of Foreign Keys

The current part of KeyMiner consists of 3 steps:

1. Preprocessing
2. Evaluation of INDs
3. Postprocessing

First, KeyMiner determines foreign key candidates from columns, with the aid of just detected PKs, and prepares IND candidates to evaluate. Second, IND candidates are analyzed by the algorithm SPIDER. Lastly, satisfied IND candidates are interpreted according to the KeyMiner purposes.

4.4.1 IND Candidate

IND candidate is a structure which represents a subset of values of a column (its subtree).

Definition 38 (IND candidate)**.** *Let* c *be a column,* k *its level and node* n ∈ c[k] *, then* ind(c[n]) *denotes an* IND candidate *whose set of values is* c[n].

Each IND candidate includes these items:

- `Vals` - a set of sorted values
- `Refs` - a set of potential supersets (referenced attributes of potential INDs)
- `CursorIntoVals` - a cursor pointing at the value which is being currently processed during the evaluation of INDs, starts at minimum
- `Relatives` - a set of other FK candidates
- `Min` - minimum from `Vals`
- `Max` - maximum from `Vals`
- `Dist` - number of distinct values in `Vals`
- `Hash` - array of bits representing `Vals`

The items `dist` and `hash` are "for free", because they can be obtained within the process of sorting, which is necessary for the next phase. Because of using only bitwise manipulations, operations with the item `hash` are very efficient. As far as `Relatives` is concerned, let c be a column, k any of its levels and node $n \in c[k]$, then `ind(c[n]).Relatives = c[k]`. As a result, if $k = 0$ then `ind(c[n])`.Relatives includes only one member - itself.

4.4.2 Preprocessing

The aim of the present phase is to prepare IND candidates in order to be validated by SPIDER. Algorithm 13 is self-explanatory, only the reason of line 4 is depicted in Figure 4.9. The core of the algorithm is the procedure `CheckCandidate`, which is described in the oncoming section.

Example 35 (Branching on Different Levels): *The matter of line 4 is showed in Figure 4.9. It is important to cover with the discovered key* PK *as big radius as possible.* `KeyLevel` *of* PK *is the level of node* 3 *(the level where the node* 3 *is situated), however, it is necessary to "move" the context path of* PK *to the level of the node* 2 *(*`LevelToTry`*) in order to include the foreign key candidate of column* FKC*. The "movement" can not be to the level of node* 1*, because* PH *has branching there and therefore, the uniqueness of* PK *could be influenced.*

Algorithm 13 FK Candidate Preprocessing

Input: $KeySet$: set of discovered keys,
$Columns$: set of all columns from XML data
Output: $SetForSpider$: set of IND candidates to evaluate in the next phase

```
1: for each unary Key in KeySet do
2:   KeyCol ← column from which Key is derived
3:   KeyLevel ← the level on which Key has been discovered
4:   LevelToTry ← the closest level k to the root of XML tree having
     frnodes(KeyCol[k]) = frnodes(KeyCol[KeyLevel])
5:   ColumnsToTry ← the set of all columns whose XPath expression
     contains the context path of Key
6:   for each Column in ColumnsToTry do
7:     if support(Column[LevelToTry]) is true then
8:       CheckCandidate(KeyCol, Column, LevelToTry, SetForSpider)
9:     end if
10:  end for
11: end for
```

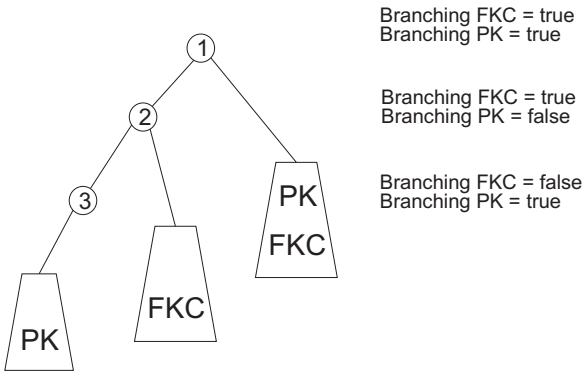


Figure 4.9: Branching on Different Levels

CheckCandidate

The procedure `CheckCandidate(KeyCol, Column, LevelToTry, SetForSpider)` checks whether the foreign key `KeyCol` → `Column` with the context path specified by the level LevelToTry can exist. If the program reaches line 4 and IND candidates derived from `KeyCol` and `Column` have not been already constructed, they are created within the execution on lines 4-10. The verification process is composed of 2 steps:

- (lines 1,2) First is verified whether `KeyCol` holds its unique values in each subtree where values of `Column` occur.
- (lines 4-6) Second, the process iterates over each subtree where values

of `Column` exist and proceeds four basic tests of inclusion dependency between the values of `KeyCol` and the values of `Column`. These tests are adopted from Section 3.5.5.

Next, if `Column` is successful in basic tests, respected sets of values are marked to be checked later value by value in thorough IND test. This means to tag respected sets of values in subtrees of `KeyCol` as possible supersets of the sets of values in subtrees of `Column`. This is done via the item `Refs` (lines 9,10). Last, IND candidates must be stored in order to be processed by SPIDER if they are not already placed there.

Algorithm 14 CheckCandidate

Input: $KeyCol$: the column from which key is derived,
$Column$: a column to verify if it can be IND candidate
$LevelToTry$: the level of XML tree on which key has been discovered
$SetForSpider$: set of IND candidates to evaluate in the next phase
Output: set of IND candidates stored in $SetForSpider$

```
1: if Column[LevelToTry] ⊃ KeyCol[LevelToTry] then
2:   return
3: end if
4: for each Node in Column[LevelToTry] do
5:   if ( (ind(Column[Node]).Min < ind(KeyCol[Node]).Min)
     or (ind(Column[Node]).Max > ind(KeyCol[Node]).Max)
     or (ind(Column[Node]).Dist > ind(KeyCol[Node]).Dist)
     or (ind(Column[Node]).Hash ⊃ ind(KeyCol[Node]).Hash)
     ) is true then
6:     return
7:   end if
8: end for
9: for each Node in Column[LevelToTry] do
10:  ind(Column[Node]).Refs += ind(KeyCol[Node])
11:  if ind(KeyCol[Node]) not in SetForSpider then
12:    SetForSpider += ind(KeyCol[Node])
13:  end if
14:  if ind(Column[Node]) not in SetForSpider then
15:    SetForSpider += ind(Column[Node])
16:  end if
17: end for
```

EXAMPLE 36: *The process of the procedure* `CheckCandidate` *is visualized in Figure 4.10.* P *entails* `KeyCol` *and* F *corresponds to* `Column`*. Values of* P *are in accordance to* `LevelToTry` *divided into the sets* $\mathrm{P}_1, \mathrm{P}_2, \mathrm{P}_3$*, the same with* $\mathrm{F} = \mathrm{F}_1 \cup \mathrm{F}_2$*. The following steps would be performed:*

- *test if* $\{1, 2, 3\} \supseteq \{1, 2\}$

- *basic tests of inclusion dependencies* $P_1 \supseteq F_1$ *and* $P_2 \supseteq F_2$
- *if tests are passed then add* $\text{ind}(P_1[1])$ *into* $\text{ind}(F_1[1])$.`Refs` *and add* $\text{ind}(P_2[2])$ *into* $\text{ind}(F_2[2])$.`Refs`
- *store IND candidates if it is necessary for the later complete evaluation of* $P_1 \supseteq F_1$ *and* $P_2 \supseteq F_2$

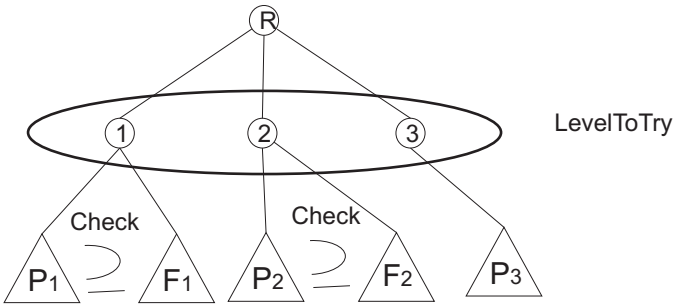


Figure 4.10: Checking of IND Candidates

4.4.3 SPIDER

SPIDER is presented in detail in Section 3.5 and the algorithm itself is demonstrated in Algorithm 4. There are two differences in the implementation in KeyMiner (Algorithm 15) - in the code situated on line 5 of the original algorithm and storing of satisfied INDs are changed.

- Update of `Refs` is expanded, because the information about non-existence of IND is necessary to sent to other "parts"of the relative foreign key candidate.
- Because IND candidate is satisfied only in case each member of its `Relatives` is satisfied, storing of satisfied IND candidates must be moved out from the main loop. Furthermore, only one member must be stored from each common set `Relatives`.

4.4.4 Postprocessing

In conclusion, the analysis of not resolved primary keys is performed. More primary keys of one relation with the same key degree can occur within the primary key inference process and KeyMiner tries to determine the right one.

- Firstly, recognition on the basis of being a referenced part in some IND.

Algorithm 15 modified SPIDER

Input: $SetForSpider$: set of IND candidates
Output: inferred foreign keys

```
1: construction of Heap using minimal values of members in SetForSpider
2: while Heap is non-empty do
3:   CurCands ← IND candidates with current minimal value
4:   for each Cand in CurCands do
5:     ToDel ← CurCands − Cand
6:     for each RelativeCand in Cand.Relatives do
7:       RelativeCand.Refs ← RelativeCand.Refs − ToDel
8:     end for
9:     if Cand contains next value then
10:      Cand.CursorIntoVals ++
11:      add value under Cand.CursorIntoVals into Heap
12:    end if
13:  end for
14: end while
15: for each Cand in SetForSpider do
16:   store each Superset in Cand.Refs as the new foreign key
      Cand → Superset
17:   delete all members of Cand.Relatives from SetForSpider
18: end for
```

- Secondly, a simple lexical analysis about the containment of some keywords is proceeded.

If KeyMiner do not distinguish the right key path, it informs the user about that.

Chapter 5

Experimental Results

In this chapter, experimental results of the application KeyMiner are presented. XML data sets have been taken as an input and durations of the particular phases of program execution have been gathered. Tests have been performed on the synthetic as well as real datasets.

5.1 Datasets and Configuration

5.1.1 Synthetic Data

Two synthetic data sets (named as `SYN1`, `SYN2`) have been utilized. They have been constructed in order to perform a thorough test containing many large sets of potential relative foreign keys.

5.1.2 Real Data

Data sets originate from the raw data catalog of the U.S. government (`http://www.data.gov/catalog/raw`). Concretely,

- Data set `FR` is from Federal Register.
- Two data sets `CFR1`, `CFR2` originating from the Code of Federal Regulations.

Unfortunately, provided datasets have not been accompanied by schemas, however, all discovered keys/ foreign keys have been properly verified.

5.1.3 Configuration

- The configuration of the computer on which tests have been proceeded is `Intel Core2 CPU T5600 1.83GHz, 1GB RAM`.

- Support has been set to 5 in order to find a big amount of instances of key expressions.

5.2 Results

	FR	CFR1	CFR2	SYN1	SYN2
file size	5,8 MB	11,3 MB	14,6 MB	14 MB	55 MB
# tuples	442	186	263	65	260
# columns	706	295	437	115	460
# PKs	49	21	28	15	60
# FKs	11	6	1	10	40
PK inference	0,12 s	0,13 s	0,15 s	6 s	6,85 s
FK preprocessing	9 s	2 s	4 s	33 s	126 s
FK inference	0.03 s	0,32 s	0,26 s	3,5 s	13,2 s

Table 5.1: Experimental Results

From the gathered information is obvious that as a result of acquiring values of subtrees and their subsequent sorting, FK preprocessing has the most processing requirements. Next itemize contains some additional observations.

- As it was mentioned before, the synthetic data sets have been generated to be difficult with reference to detection of relative PKs/relative FKs which is noticeable at most in the FK preprocessing phase.
- KeyMiner "likes"data that are very structured and do not consist of one or more very big "tables". 4 MB of `FR` is composed of only one big table and therefore the result of FK preprocessing lasts longer than in the events of other cases.

Chapter 6

Counter-Examples

This chapter contains some examples where can happen that the user obtains different solution than it he/she expected.

6.1 Similar Values

The problem is that inclusion dependency with a primary key not always means a foreign key. To illustrate, suppose each column holds distinct numbers 1,. . . ,100 as values, then a huge amount of unwanted foreign keys would be detected.

6.2 Main Features

KeyMiner do not discover keys which do not meet main features noted in Sections 4.1.1 and 4.1.2. For instance,

- keys with XPath expressions containing wildcards
- keys which are not general, for example, key with the XPath expression `/volume/articles/article[2]`
- keys having empty key paths or longer key paths than stated in Section 4.1.2
- when a context path of a wanted key is longer than stated in Section 4.1.2

Chapter 7

Related Work

7.1 Theory of Keys in XML

In recent years, several types of XML Integrity Constraints (XICs) have been studied. Key and foreign key specifications for XML have been proposed in the XML standard [2], XML Data [3], XML Schema [4, 5].

7.1.1 Definition by Buneman et. al.

Maybe the most popular theoretical proposal was published by Buneman et al. in [10, 11, 12]. Their work addressed the issue of hierarchical keys. The definition of keys was independent from any schema such as DTD or XSD, however, the authors defined keys for XML using path expression. They rather based keys and foreign keys on the representation of XML data as trees. Keys uniquely identify nodes in such a tree according to selected nodes. Their queries were defined by the *path language* (PL) which has expressive power enough to capture most practical cases, yet simple enough to be studied. The syntax of expression p in PL is following:

$$p \ ::= \ \varepsilon \mid l \mid p.p \mid \_*$$

where l stands for any type of a node. PL consists of all path expressions over the alphabet $\mathcal{L} \bigcup \{\_, *\}$ including the binary operation of concatenation and the empty path expression ε as identity. In [12] the authors have investigated a key constraint language introduced in previous works. Moreover, they have studied the associated satisfiability and implication problems in the absence of DTDs.

7.1.2 Definition by Fan et. al.

Next, in [13] a formalization for XML DTDs that specifies both the syntactic structure and integrity constraints has been proposed by Fan et al.

The authors defined three constraint languages L, L_u, L_{id} which support a reference mechanism as well as better semantics. L_u is a simple extension of the key/ foreign key mechanism, suitable for native XML documents. Languages L, L_{id} can be used to represent semantic constraints in case of data originating in object-oriented and relational model. The study of implication and finite implication problems was made for these three languages. Furthermore, the implication problems of more general forms of constrains, including functional, inclusion and inverse constraints were investigated. However, again in the absence of DTDs, which trivializes the consistency analysis. Later, Fan et al. made a first step towards understanding the interaction between DTDs and integrity constraints [14]. The decision problems of keys and foreign keys were studied in the presence of DTD.

7.1.3 Improvement of Mentioned Proposals and Surveys

Latterly, in [15] the authors tried to contribute by some advantages over other definitions proposed in the literature [11],[13]. Particularly, the authors have introduced the concept *P-tuple* for the production of semantically correct tuples during key satisfaction.

In conclusion, some helpful surveys have been published. Firstly, XML constraints have been described from many points of view in [16]. There have been reviewed constraint languages, static and run-time analysis, application of XML constraints etc. Secondly, as far as only XML keys is concerned [17] gives a brief overview of definitions of keys that have been proposed in the research literature with the main notion of Buneman et al.

7.2 XML Schema Inference

The recent approaches to the problem of automatic schema extraction can be distinguished according to two main criterions.

Firstly, it is the type of the result which can be either DTD or XSD. Nevertheless, not every method which produces XSDs has expressive power beyond DTD. Moreover, according to our best knowledge exists no method that would output XML schema where key/ foreign keys would be included (or schema which would be combined with some related integrity constraints).

Secondly, XML schema inference methods can be divided by the way it is constructed into *heuristic-inferring* (HI) and *grammar-inferring* (GI). A nice survey of many proposals using these kinds of methods has been presented in [18]. Both extraction processes mostly utilize the strategy called *merging state algorithm*. They construct grammars from elements in XML data. Next, these grammars are used to build a *prefix tree automaton* (PTA

tree). Then states of this structure are merged in order to obtain the optimal solution.

EXAMPLE 37 (PTA tree): *An example of grammar generation is depicted in Figure 7.1 (inspired by [18]) and the corresponding PTA tree for the element* ***author*** *is visualized in Figure 7.2.*

```
<author>                              author -> name title title
  <name>                              author -> name title
    <first> John </first>
    <surname> Black </surname>        name -> first surname
  </name>                             name -> surname first
  <title>Mgr.</title>
  <title>Ph.d.</title>                first -> PCDATA
</author>                             surname -> PCDATA
<author>                              title -> PCDATA
  <name>
    <surname> Hill </surname>
    <first> Jane </first>
  </name>
  <title>Mgr.</title>
</author>
...
```

Figure 7.1: The creation of grammars (on the right side) from XML data

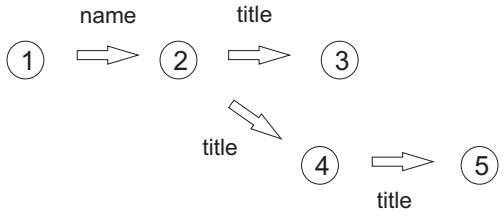


Figure 7.2: PTA tree of the element `author`, whose grammar is presented in the Figure 7.1

7.2.1 Heuristic-Inferring Methods

Heuristic methods build the solution manually. The predefined heuristic rules can be very simple - for instance, $A \rightarrow B\ B\ B\ B \quad \Rightarrow \quad A \rightarrow B+$ or they can be inspired even by artificial intelligence.

XTRACT

Probably the first representative of the merging state algorithm is XTRACT [19]. The authors generate a set of possible candidates and choose the optimal one. In addition, the so-called approach *minimum description length principle* (MDL principle) has been proposed. It represents the quality of a result utilizing two aspects - conciseness and preciseness, which are able to be set by an user. Consequently, this method aids to the user in choosing quality of an output DTD.

Later Approaches

Later proposals[20, 21, 22, 23] extend XTRACT in various aspects. Favorite ones are *ant colony optimization* (ACO) or *sk-strings*. ACO metaheuristic is inspired by processes in nature where ants exchange information by pheromones. In general, artificial ants search space using only a simple heuristic, nevertheless ants leave pheromones on the trail they follow. This information helps ants during further iterations to find better sub-optimal solutions (merging states of PTA tree). Crucial parts are to define evaluate function (sometimes MDL principle is utilized) and to define one step in an ant movement, where *sk-strings* can be useful as a merging criterion. In short, *sk-strings* consists of several algorithms. The basis of these algorithms is a variant of the *Nerode equivalence relation*. Under the *Nerode relation*, a pair of states are equivalent if they are indistinguishable in all paths leading from these states. For the reasons of effectiveness, the authors in sk-strings method check only the most probable paths of length of k or the most probable paths terminating in terminal states. In addition, in the last proposal [23] authors discussed and implemented the strategy that significantly considers user interaction. As far as the type of the output is concerned [21, 22, 23] already produce XSDs.

7.2.2 Grammar-Inferring Methods

On the other hand, grammar-inferring proposals produce a particular class of languages with specific characteristics. Consequently, a certain degree of quality of the created schema is guaranteed. Generally, the main idea of these methods is following, we consider an XML schema as a grammar and an XML document validation as a word generated by the grammar. Since grammars accepting XML documents are context-free, the investigation process is pruned to searching for a set of regular expressions. As a result, each element in XML document is expressed with a regular expression (RE). Although, many of studies related to this topic have been proposed, not a few of them remained only theoretical.

1-unambiguity

Probably the first experimentally evaluated approach is mentioned in [24]. They have devised a *restricted element content model* (specified below) and considered the important aspect of XML - *1-unambiguity.*

- (Restricted element content model)
  $E := (T_1 \dots T_k)^{<min,max>}$
  (Term)
  $T_i := (s_{i1}^{opt} \dots s_{ij}^{opt})^{<min,max>}$ //sequence of symbols
  or $(s_{i1}^{opt} | \dots | s_{ij}^{opt})^{<min,max>}$ //choice of symbols
  where $min = 0$ or 1, $max \geq 1$, and $opt = true$ or $false$

EXAMPLE 38 (Utilization of Element Content Model): *For example, a regular expression* (a b^* c?)* *can be represented as* $E = (T_1\ T_2\ T_3)^{<0,\infty>}$ *where* $T_1 = (a)$, $T_2 = (b)^{<0,\infty>}$, *and* $T_3 = (c^{opt})$.

The term *1-unambiguity* means determinism of content models in XML schema which is imposed by the W3C specification. The authors have described 9 heuristic rules for the DTD schema generation so that the language restriction is fulfilled in the result.

XStruct

Based on ideas of [24], XStruct[25] extracts a schema for XML data by applying some heuristics to derived regular expressions. The authors improve the previous approach to process multiple or very large XML documents. Moreover, this method infers XSDs with data types.

SOREs, CHAREs

Following the idea of the 1-unambiguity from [24], [26] introduces an algorithm for the inference of concise DTDs. The authors were inspired by analysis of real-world XML data and schemas and have proposed classes which cover a great deal of real-world examples. They have presented two such classes.

1. The *class of single occurrence REs* (SOREs) which consists of REs in which each element exists at most once.

   EXAMPLE 39 (SORE): *To illustrate,* $((d?(b + c))^*)a^+$ *is* SORE, *while* $a(a + b)^+$ *is not.*

2. The class of *chain regular expressions* (CHAREs) which consists of those *SOREs* composed by a sequence of factors $f_i, \dots, f_n$ where every factor is an expression of the form $(a_i + \dots + a_k), (a_i + \dots + a_k)^?, (a_i + \dots + a_k)^+$,or, $(a_i + \dots + a_k)^*$ where $k \geq 1$ and every a_i is an alphabet symbol.

EXAMPLE 40 (CHARE): *For example,* $a(b + c)^?(e + f)^*d$ *is a* CHARE*, while* $(ab + c)*$ *is not. The former one consists of 4 factors.*

The introduced algorithm is a typical merging state approach starting with *PTA tree* using rules which ensure required class of the output.

k-local SOXSDs

[27] extends of the foregoing work, the authors proposed a theoretically complete strategies for inferring XSDs from a corpus of XML documents - iXSD and iLocal. They again focussed on real-world schemas and created a new class based on SOREs called *k-local single occurrence XSDs* (k-local SOXSDs). While k-local XSD means that content models of the schema is dependent only on elements up to the k-th ancestor.

Name	Schema	Features
[24]	DTD	1-unambiguity
[25]	XSD	1-unambiguity, multiple or very large XML documents, data types
[26]	DTD	SOREs, CHAREs, based on real-world data
[27]	XSD	k-local SOXSDs, based on real-world data

Table 7.1: Main characteristics of Schema Inference methods

7.3 The Role of Inclusion/Functional Dependencies in the Inference of XML Integrity Constraints

In the relational data model a key is a special case of *functional dependency* (FD), which is described as following.

- A functional dependency denoted by $R_i : Y \to Z$ on $R_i(X_i)$
  is satisfied by r_i iff $\forall t, t' \in r_i;\ t[Y] = t'[Y] \Rightarrow t[Z] = t'[Z]$
  Where a relation $R_i(X_i)$ is defined with a relation name R_i and a set of attributes X_i, and t, t' are tables from a set of tables r_i and Y, Z are subsets of tables.

Then a question comes naturally, whether an XML key is a special instance of XML functional dependency (XFD). In contrast to relational databases, XML data can be flexible and hierarchical. For this reason, defining XFD is not a trivial task and many proposals have appeared depending on the way how data items (in XML data) and keys are defined, consequently. For instance, in [46] the authors have followed a path-based approach and have

created their XFD notion in accordance with the XML key notion presented in [12]. A nice survey of six major approaches of defining XFD has been presented in [47]. On the other hand, an inclusion dependency (IND) $A \subset B$ means that each value of the dependent attribute A is contained in the value set of the referenced attribute B (the definition for inclusion dependency in XML is mentioned for example in [12, 13]. Consequently, it is possible to assume that inclusion dependency (defined only as a subset as mentioned in the previous sentence) is a generalized case of a foreign key in XML data.

7.3.1 XFD Inference

Firstly, inference of XFDs will be considered. A huge amount of algorithms dealing with FD discovering exists and even a few algorithms concerning XFDs have appeared, too. In [48] have been defined approximately satisfied XFDs (according to tree similarity) by a respective part of XML document. Furthermore, there has been also addressed in definitions the problem of inferring approximate XFDs from data.

DisoverXFD

In the paper [49] authors have closely analyzed previous popular proposals of defining XFDs and have introduced a new GTT-XML (Generalized Tree Tuple-based XML) XFD notion and a new XML key notion, consequently. Founded on foregoing definitions, the authors have designed and implemented the DiscoverXFD system which employs a new XML data structure and some novel partition-based algorithms that can be utilized to the inference of XFDs and the detection of XML data redundancies. Furthermore, a normalization algorithm for XML schema has been described. In short, DiscoverXFD converts XML data into a set of hierarchically linked relational tables. Next, amended methods from FD inference are used. XFDs including absolute primary keys are discovered if the grouping over determinant attributes and the grouping over dependent attributes result in the same group.

Later, algorithm proposed in [50] utilizes the XFD definition from [49] and considerably enhances the previous method by changing the order of execution of elements and by a powerful pruning inspired from [41].

7.3.2 IND Inference

Secondly, as far as IND is concerned several proposals about IND elicitation have been published. Particularly, two methods [51], [53] in which very efficient approaches have been introduced appeared in the last 3 years.

Name	Features
[49]	GTT-XML notion, new notion for XML keys, DiscoverXFD system, the detection of XML data redundancies, normalization of XML schema
[50]	improves the previous method by changing the order of execution and pruning

Table 7.2: Main characteristics of mentioned XFD discovery methods

SPIDER

The algorithm SPIDER [51], [52] (Single Pass Inclusion DEpendency Recognition) detects all unary INDs during only one visit of all the nodes. It works in two steps - value sets are sorted during the first one and then all the candidates are examined in parallel. The core of the method is using the data structure min-heap which synchronizes the processing of all values of all attributes. Consequently, it is impossible to miss any IND or run into a deadlock. Moreover, the authors were inspired by method based on Apriori principle in [53] and have expanded SPIDER to generate also composite inclusion dependencies. Efficient pruning techniques within the preprocessing phase have been implemented, too.

DBA Companion

In the paper [53] authors have described a different algorithm implemented in a DBA Companion project which can detect all satisfied INDs in a given database. The idea is to create binary relations which connect every value of the database with attributes having this value. This new data organization, so-called *extraction context*, considers various data types and can be perceived as a transaction database in which attributes are items and values are transactions. In addition, unary INDs correspond to exact association rules and they can be extracted in one pass. In consequence, the authors use a levelwise method based on Apriori algorithm to compute n-ary INDs. In case of data inconsistencies in a database, approximate INDs are introduced and integrated into their algorithms.

7.4 Discovering Keys in Relational Databases

The problem of discovering keys or relations is studied in the scope of reverse engineering, whose aim is i.a. understanding of the data semantics. Most of the studies, that are related to the reverse engineering process, extract dependencies from a static analysis of the relational schema based on a consistent naming of key attributes (for instance, matching attribute

Name	Features
SPIDER	detection of all unary INDs within only one run through all the values, detection of composite INDs, efficient pruning techniques of IND candidates
DBA Companion	detection of all unary INDs within only one run through all the values, detection of composite INDs, considers data types, more time-consuming preprocessing

Table 7.3: Main characteristics of mentioned IND discovery methods

names, searching for synonyms etc.) or given by a database.

7.4.1 First Swallows

In 1994, 3 papers [28, 29, 30] appeared independently. These proposals try to elicit relations in a database from application data. The idea is simple: a process whose external data are processed in the application depends on the properties of these data. Consequently, understanding the logic of the principles how application data are used leads to the understanding many of the implicit relations and dependencies in the program. Especially, [28] is the detailed study of showing the way how specific types of SQL queries can aid to obtain a conceptual schema of a relational database. Moreover, [29] considers not only dataflow queries, but also SQL procedural patterns. The authors have implemented theirs methodology in Prolog. Thus, the knowledge base containing the pattern recognition rules can be easily modified.

Comprehensive Guide

In [31] the authors presented the full-range guide in data structures elicitation techniques. It contains an enumeration of many methods with short descriptions, which are demonstrated in detail on the example of discovering foreign keys. These techniques consist of many kinds of analysis such as physical structure analysis, dataflow analysis, usage pattern analysis, name analysis, domain knowledge, data analysis or program execution.

7.4.2 Logical Approach

Much later a different approach has been proposed [33]. The main idea is to obtain integrity constraints by iterative inspection of candidates. In short, the method generates sample database for each candidate to be an integrity constrain and then decides, whether this candidate should be specified or not. Authors have translated dependencies between properties of entities

into *propositional logic* and have utilized its techniques to offer decision support. In particular, they use a search version of *SAT-solvers* to semi-automatically generate sample databases. However, this technique is more like an auxiliary technique in the the process of selecting those constraints that capture important properties of the underlying application domain.

7.4.3 Many Approaches Together

In [32] has been proposed the VQT algorithm (Value cardinality, Query pattern, Translation into XML). A relational schema to XML Schema translation method that analyzes the cardinalities between implicit data values and the equi-join characteristic in user queries and infers the implicit referential integrities. Still the authors rely on the obtaining of the metadata information of the relational database. The whole process consists of several different approaches which are in fact very straightforward. But experiments in this paper demonstrate that the results are more exact than other translational methods. First of all, the metadata information of the relational database is acquired in order to define primary keys. Next, they apply an ontology for the semantic analysis and the comparison between columns. In parallel, very simple query pattern analysis is running and other candidates of referential relations are being located . Later, the 1:N cardinality between values in FK fields and PK fields is checked, however, only till some predefined number of verified values between columns.

7.4.4 Gordian

Gordian [41] has completely different and innovative approach for efficiently identifying all composite primary keys in a dataset. The main idea is behind a so-called *cube operator*, which is used in datawarehousing. The *cube operator* encapsulates all potential projections while computing aggregate functions on the projected entities. Firstly, Gordian creates a *prefix tree* (in short, branches in the tree represent rows in tables) and then performs a depth-first (DF) traversal with powerful merging and pruning methods. As a result, the algorithm requires maximally double recursive visit of all the nodes in order to extract all of the non-keys. In conclusion, the candidate primary keys are computed as complements of the discovered non-keys.

7.4.5 Quality Metrics

The following topic is connected to the key discovery process in relational database, too. Referential integrity quality metrics have been studied a lot in [40]. The authors have proposed a comprehensive set that evaluates referential completeness and consistency. Quality metrics are hierarchically defined at four granularities: database, relation, attribute and attribute value.

Name	Features
[28, 30]	detection of relations from SQL queries
[29]	detection of relations from SQL queries and SQL procedural patterns
[31]	enumeration of many methods for data structure elicitation
[33]	an auxiliary technique for detection of integrity constraints, propositional logic, SAT-solvers
[32]	detection of FKs, requires metadata, mix of several approaches (data-driven, query-driven, ontologies)
[41]	detection of composite PKs, DF traversal with powerful merging and pruning methods

Table 7.4: Key characteristics of mentioned methods considering integrity constraints discovery in relational databases

7.5 XML and Data Mining

Knowledge discovery techniques and classification with respect to XML is summarized in [34, 35]. The flexible format of XML makes it easy for defining arbitrary languages. One such example is the *Predictive Modeling Markup Language* (PMML) an industry standard for the representation of mined models as XML documents. In addition, the use of XML allows the description of complex structures, like trees or clusters, as well as the domain knowledge. In recent years most of the proposals considered *inductive databases* (IDB) or mining association rules. IDBs are general-purpose databases in which both the data and the patterns can be stored, retrieved or changed. Then knowledge discovery process can be controlled with a query language designed for a given data mining tasks.

7.5.1 Inductive Databases

Papers dealing with IDB are for instance [36, 37]. In the former one an XML-based data model XDM is presented. In XDM the pattern descriptions are stored together with data in the same XML document. This feature permits the reuse of patterns by the IDB. Moreover, it is possible to define new pattern models or even new mining operators. The latter case is probably the last step in defining XML query languages for mining purposes. Authors have proposed an approach capable of providing both a direct specification of mining techniques in XQuery-like language as well as the utilization of detached functions for the efficient implementation of crucial parts.

7.5.2 Association Rules

XMine

The XMine operator [38] describes the basic concept of mining association rules in XML documents which is based on XPath. XMine creates a representation of the XML mining task as a relational table and uses SQL-oriented algorithms to do the knowledge discovery.

Mining Primary Keys

Finally, [39] provides discovering of primary keys through association rules and an apriori-like algorithm. The authors have defined support and confidence of a key expression and only the minimal cover of the set of primary key expressions is considered during the inference process. At first, the algorithm finds absolute 1-keys and relative 1-keys. Later, from these key expressions the apriori-like algorithm try to produce all other k-keys afterwards. In addition, very interesting aspect is the capability to extract also primary keys containing wildcard `'?'` in path expressions.

Name	Features
[36]	XML-based data model
[37]	notable definition of XML query languages for mining purposes
[38]	XMine operator, SQL algorithms
[39]	association rules, apriori-like algorithm, very precise, basic wildcard `'?'`, a computation of a minimal cover

Table 7.5: Key characteristics of mentioned methods considering XML and data mining

7.6 Referential Integrity in XQuery

The query-driven approach is the significant aspect in the inference of keys. Thus, a proper analysis of constraints in XQuery [7] could be helpful. Reasoning about integrity constraints appears often in XQuery optimization techniques, especially in query reformulation, which is well studied by Deutsch et al. - for instance, [42, 43].

7.6.1 A Generalized Tree Pattern

On the other hand, in [44] has been introduced a structure called *generalized tree pattern* (GTP) which summarizes all relevant information in an XQuery expression into a pattern consisting of one or more trees. And with

the aid of GTP the authors have developed an evaluation for XML queries, involving join, quantifiers, grouping, aggregation, and nesting.

7.6.2 XQuery-driven key discovery

Last but not least, in the paper [45] has been proposed a direct approach of key/ foreign key inference based on analysis of XQuery queries. Query constructs were properly analyzed and several observations have been made, which considers aggregation functions like avg, min, count, too. In the contrary to reverse engineering techniques from the scope of relational databases, the authors have also discussed the distinction of keys and foreign keys in equi-join queries. Moreover, they have considered counter examples and therefore they have introduced a scoring function which statistically evaluates specified keys by a user given threshold, whether they are appropriate or not.

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

Figure 7.3: The Scoring Function

Where $K_1, \ldots, K_n$ are the discovered keys, S_i is the score of K_i and N_i is the number of discovered statements about K_i. Then S^{max} is the maximum from $|S_1|, \ldots, |S_n|$ and N^{max} is the maximum from $|N_1|, \ldots, |N_n|$. The normalized scores are from the range $< -1, 1 >$.

Name	**Features**
[44]	GTP, complex evaluation of XML queries
[45]	inference of PKs and FKs, proper analysis of XQuery queries, scoring function

Table 7.6: Key characteristics of mentioned methods considering referential integrity in XQuery

Chapter 8

Conclusion and Future Work

8.1 Conclusion

To conclude, the aim of this thesis was to study the problem of mining integrity constraints in XML data. First, a detail analysis of current proposals has been elaborated and their pros and cons have been discussed. The result was that there is no study offering an efficient solution of inference both - primary keys and foreign keys even in the scope of relational databases. Consequently, new algorithm named KeyMiner has been proposed in this thesis. The novel method provides very fast detection of n-ary PKs and unary FKs from XML data. The basis of KeyMiner is combination of the algorithms Gordian and Spider that are expanded and modified in order to be able to utilize them at XML data. Gordian is the only known method which can discover all composite keys while avoiding the exponential processing requirements. Moreover, KeyMiner's modification can detect relative PKs within the standard Gordian's process of inference of absolute PKs. In addition, the nature of SPIDER's algorithm allows KeyMiner to elicit relative FKs together with absolute FKs during only run through all the values. KeyMiner has been described in detail, its counter-examples have been discussed and it has been tested on synthetic as well as real data sets.

8.2 Future Work

As far as future work is concerned, there are two main tasks - wildcards and more precise detection of foreign keys. The former one is very difficult problem, nevertheless the only matter is to specify new relations according to the wildcards which will be added into the current set of relations to proceed. The latter one can be solved by a a query analysis (for instance, the proposal [45] can be very useful), a machine learning approach or more sophisticated

lexical analysis within the postprocessing phase. As far as machine learning is concerned the following features can be considered among others:

- How often one attribute appears in the referenced part of some key expression as well as in the dependent part of another key expression.
- The frequency of being a multi-referenced or multi-dependent attribute.
- The ratio of contained values
- The analysis of some features of values like the average length

Last but not least, propositional logic can be useful as a auxiliary technique like it has been proposed in [33].

Bibliography

[1] W3C. *World wide web consortium.* http://www.w3.org/, 1994-2008.

[2] 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/.

[3] A. Layman, E. Jung, E. Maler, H. S. Thompson, J. Paoli, J. Tigue, N. H. Mikula, and S. de Rose. *XML-Data.* W3C, 1998. http://www.w3.org/TR/1998/NOTE-XML-data-0105/.

[4] 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/.

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

[6] J. Clark and S. de Rose. *XML Path Language (XPath) Version 1.0.* W3C, 1999. http://www.w3.org/TR/xpath/.

[7] 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/.

[8] 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.

[9] W3C. *On SGML and HTML.* http://www.w3.org/TR/REC-html40/intro/sgmltut.html, 2009.

[10] P. Buneman, W. Fan, J. Siméon, and S. Weinstein. Constraints for Semistructured Data and XML. *SIGMOD Record*, 30(1):47-54, 2001.

[11] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. *Computer Networks,* 39(5):473-487, 2002.

[12] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about Keys for XML. *J. Inf. Syst.*, 28(8):1037-1063, 2003.

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

[14] W. Fan and L. Libkin. On XML Integrity Constraints in the Presence of DTDs. *J. ACM,* 49(3):368-406, 2002.

[15] Md. S. Shahriar and J. Liu. On Defining Keys for XML. *Proc. of the 8th Int. Conference on Computer and Information technology Workshops*, pages 86-91, Sydney, QLD, 2008. IEEE.

[16] Fan, W.: XML Constraints: Specification, Analysis, and Applications. In *DEXA'05: Proc. of the 16th Int. Workshop on Database and Expert Systems Applications*, pages 805-809, 2005. IEEE.

[17] S. Hartmann, H. Köhler, S. Link, T. Trinh, and J. Wang. On the notion of an XML Key. In *SDKB: Proc. of the 3th Int. Workshop on Semantics in Data and Knowledge Bases*, pages 103-112, Nantes, France, 2008. Springer.

[18] Mlýnkova, I.: An Analysis of Approaches to XML Schema Inference. *SITIS '08: Proc. of the 4th Int. Conf. on Signal-Image Technology and Internet–based Systems*, Bali, Indonesia, November/December 2008. IEEE.

[19] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: Learning Document Type Descriptors from XML Document Collections. *Data mining and knowledge discovery,* 7:23–56, 2003. Springer.

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

[21] O. Vošta, I. Mlýnková, and J. Pokorný. Even an Ant Can Create an XSD. In *DASFAA'08: Proc. of the 13th Int. Workshop on Database Systems for Advanced Applications*, LNCS, pages 35–50. 2008. Springer.

[22] M. Nečaský and I. Mlýnková. Towards Inference of More Realistic XSDs. *SAC '09: Proc. of the 2009 ACM symposium on Applied Computing*, Honolulu, Hawaii, pages: 639-646, 2009. ACM.

[23] J. Vyhnanovská and I. Mlýnková. Interactive Inference of XML Schemas. *RCIS '10: Proc. of the 4th Int. Conf. on Research Challenges in Information Science*, pages 191-202, Nice, France, 2010. IEEE

[24] J.-K. Min, J.-Y. Ahn, and C.-W. Chung. Efficient Extraction of Schemas for XML Documents. *Inf. Process. Lett.*, 85(1):7–12, 2003.

[25] J. Hegewald, F. Naumann, and M. Weis. XStruct: Efficient Schema Extraction from Multiple and Large XML Documents. In *ICDE'06: Proc. of the 22nd Int. Conf. on Data Engineering Workshops*, page 81, Atlanta, GA, USA, 2006. IEEE.

[26] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of Concise DTDs from XML Data. In *VLDB'06: Proc. of the 32nd Int. Conf. on Very Large Data Bases*, pages 115–126, Soul, Korea, 2006. VLDB Endowment.

[27] G. J. Bex, F. Neven, and S. Vansummeren. Inferring XML Schema Definitions from XML Data. In *VLDB'07: Proc. of the 33rd Int. Conf. on Very Large Data Bases*, pages 998–1009, Vienna, Austria, 2007. ACM.

[28] J-M. Petit, J. Kouloumdjian, J-F. Boulicaut, and F. Toumani. Using queries to improve Database Reverse Engineering. *ER '94: Proc. of the 13th Int. Conf. on the Entity-Relationship Approach*, Manchester, pages: 369-386, 1994. Springer.

[29] O. Signore, M. Loffredo, M. Gregori, and M. Cima. Reconstruction of ER Schema from Database Applications: a Cognitive Approach. *ER '94: Proc. of the 13th Int. Conf. on the Entity-Relationship Approach*, Manchester, pages: 387-402, 1994. Springer.

[30] M. Andersson. Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering. *ER '94: Proc. of the 13th Int. Conf. on the Entity-Relationship Approach*, Manchester, pages: 403-419, 1994. Springer.

[31] J. Hainaut, J. Henrard, D. Roland, V. Englebert, and J. Hick. Structure Elicitation in Database Reverse Engineering. *WCRE '96: Proc. of the 3rd Working Conference on Reverse Engineering*, Monterey, CA, pages: 131–140, 1996. IEEE.

[32] K. Jinhyung, J. Dongwon, and B. Doo-Kwon. A Translation Algorithm for Effective RDB-to-XML Schema Conversion Considering Referential Integrity Information *J. Information Science and Engineering* 25, pages: 137-166, 2009.

[33] S. Hartmann, S. Link, and T. Trinh. Constraint Acquisition for Entity-Relationship Models. *J. Data and Knowledge Engineering*, 68(10):1128–1155, 2009.

[34] M. N. Garofalakis, R. Rastogi, S. Seshari, and K. Shim. Data Mining and the Web: Past, Present and Future. *WIDM '99: Proc. of the 2nd Int. Workshop on Web Information and Data Management*. Kansas City, Missouri, USA, pages: 43-47, 1999. ACM

[35] R. Nayak, R. Witt, and A. Tonev. Data mining and XML documents. *IC'2002: International Conference on Internet Computing*, Las Vegas, Nevada, pages: 660-666, 2002.

[36] R. Meo, G. Psaila. An XML-Based Database for Knowledge Discovery. *EDBT 2006 Workshops: Proc. of the 10th Int. Conf. on Extending Database Technology*, Munich, Germany, LNCS, 4254:814-828, 2006. Springer.

[37] A. Romei, F. Turini. XML Data Mining. *Software: Practice and Experience*, 40(2):101-130, 2010.

[38] D. Braga, A. Campi, S. Ceri, M. Klemettinen, and P.L. Lanzi. A Tool for Extracting XML Association Rules from XML Documents. *ICTAI '02: Proc. of the 14th IEEE Int. Conf. on Tools with Artificial Intelligence*, Washington, DC, USA, pages: 57-64, 2002. IEEE.

[39] G. Grahne and J. Zhu. Discovering Approximate Keys in XML Data. In *CIKM' 02:Proc. of the 11th Int. Conf. on Information and Knowledge Management* , pages: 453–460, 2002. ACM.

[40] C. Ordonez and J. García-García. Referential Integrity Quality Metrics. *J. Decision Support Systems* 44(2):495-508, 2008.

[41] Y. Sismanis, P. Brown, P.J. Haas, and B. Reinwald. GORDIAN: Efficient and Scalable Discovery of Composite Keys. *VLDB' 06: Proc. of the 32nd Int. Conf. on Very Large Data Bases*, Seoul, Korea, pages: 691-702, 2006. ACM.

[42] A. Deutsch, L. Popa, and V. Tanen. Query Reformulation with Constraints. *SIGMOD Record*, 35(1):65-73, 2006.

[43] A. Deutsch and V. Tanen. XML Queries and Constraints, Containment and Reformulation. *J. Theoretical Computer Science - Database Theory*, 336(1):57–87, 2005.

[44] Z. Chen, H. V. Jagadish, L. V. S. Lakshaman, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. *VLDB '03: Proc. of the 29th Int. Conf. on Very Large Data Bases*, Berlin, Germany, pages: 237-248, 2003. ACM.

[45] Martin Nečaský and Irena Mlýnková. Discovering XML keys and foreign keys in queries. *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, Honolulu, Hawaii, pages: 632-638, 2009. ACM.

[46] M. W. Vincent, J. Liu, and Ch. Liu. Strong Functional Dependencies and Their Application to Normal Forms in XML. *J. ACM Transactions on Database Systems*, 29(3):445–462, 2004.

[47] Lv. Teng and Y. Ping. A Survey Study on XML Functional Dependencies. *ISPDE' 2007: Proc. of the first Int. Symposium on Privacy, Data, and E-Commerce*, pages: 143-145, 2007. IEEE.

[48] F. Fassetti and B. Fazzinga. Approximate Functional Dependencies for XML Data. *ADBIS' 2007: Proc. of the 11th East-European Conf. on Advances in Databases and Information Systems*, Varna, Bulgaria, pages: 86-95, 2007.

[49] Yu, C. and Jagadish, H. V.: XML Schema Refinement Through Redundancy Detection and Normalization. *The VLDB Journal: the Int. J. on Very Large Data Bases*, 17(2):203-223, 2007.

[50] S. Hang, A. Toshiyuki, and K. Hiroyuki. Fast Detection of Functional Dependencies in XML Data. *XSym' 2010: Proc. of the 7th Int. XML Database Symposium*, Singapore, LNCS, 6309:113-127, 2010. Springer.

[51] J. Bauckmann, U. Leser, F. Naumann, and V. Tietz. Efficiently Detecting Inclusion Dependencies. *ICDE' 2007: Proc. on the 23th Int. Conf. of Data Engineering*, Istanbul, pages: 1448-1450, 2007. IEEE.

[52] F. De Marchi, S. Lopes, and J.-M. Petit. Unary and n-ary inclusion dependency discovery in relational databases. *J. Intell. Inf. Syst.*,32(1):53-73, 2009.

[53] J. Bauckmann, U. Leser, F. Naumann. *Efficient and Exact Computation of Inclusion Dependencies for Data Integration.*

`http://www.hpi.uni-potsdam.de/fileadmin/hpi/source/Technische_Berichte/HPI_34_Efficient_and_exact_computation_of_inclusion_dependencies_for_data_integration.pdf`, 2010

List of Figures

List of Tables