Mining XML Functional Dependencies through Formal Concept Analysis

Viorica Varga

May 6, 2010

Outline

Outline

Outline

Outline

Outline

Outline

XML Design

- XML data design: choose an appropriate XML schema, which usually come in the form of DTD (Document Type Definition) or XML Scheme.
- Functional dependencies (FDs) are a key factor in XML design.
- The objective of normalization is to eliminate redundancies from an XML document, eliminate or reduce potential update anomalies.
- Arenas, M., Libkin, L.: A normal form for XML documents. TODS 29(1), 195-232 (2004)
- Yu, C., Jagadish, H. V.: XML schema refinement through redundancy detection and normalization. VLDB J. 17(2): 203-223 (2008)

Schema definition

Definition

(Schema) *A schema is defined as a set* $S = (E, T, r)$*, where:*

- E *is a finite set of element labels;*
- T *is a finite set of element types, and each* $e \in E$ *is associated with a* $\tau \in T$*, written as* $(e : \tau)$*,* τ *has the next form:* $\tau ::= \textbf{str} \mid \textbf{int} \mid \textbf{float} \mid \textbf{SetOf}\ \tau \mid \textbf{Rcd}\ [e_1 : \tau_1, \ldots, e_n : \tau_n]$*;*
- $r \in E$ *is the label of the root element, whose associated element type can not be* **SetOf** τ*.*

- Types **str**, **int** and **float** are the system defined *simple types* and **Rcd** indicate *complex scheme elements*.
- Keyword **SetOf** is used to indicate *set schema elements*
- Attributes and elements are treated in the same way, with a reserved "@" symbol before attributes.

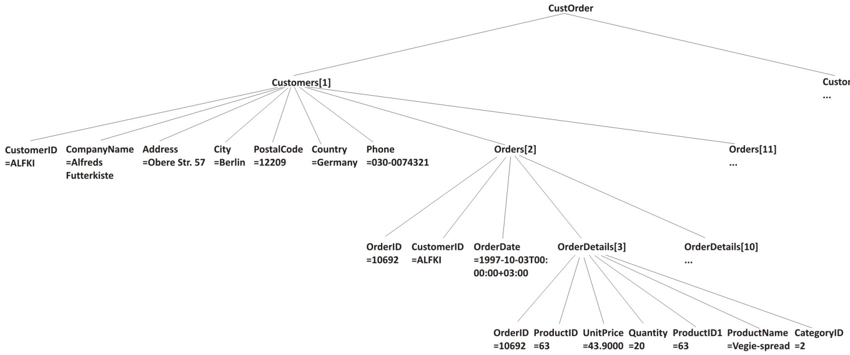


Figure: CustOrder XML tree

Customer’s Orders Example Scheme

```
CustOrder:Rcd
        Customers:SetOf Rcd
            CustomerID: str
            CompanyName: str
            Address: str
            City: str
            PostalCode:str
            Country: str
            Phone: str
            Orders: SetOf Rcd
                OrderID: int
                CustomerID: str
                OrderDate: str
                OrderDetails: SetOf Rcd
                    OrderID: int
                    ProductID: int
                    UnitPrice: float
                    Quantity: float
                    ProductName: str
                    CategoryID: int
```

- A schema element e_k can be identified through a path expression, $path(e_k) = /e_1/e_2/.../e_k$, where $e_1 = r$, and e_i is associated with type $\tau_i ::= \textbf{Rcd}\ [..., e_{i+1} : \tau_{i+1}, ...]$ for all $i \in [1, k-1]$.
- A path is *repeatable*, if e_k is a set element. We adopt XPath steps "." (self) and ".." (parent)

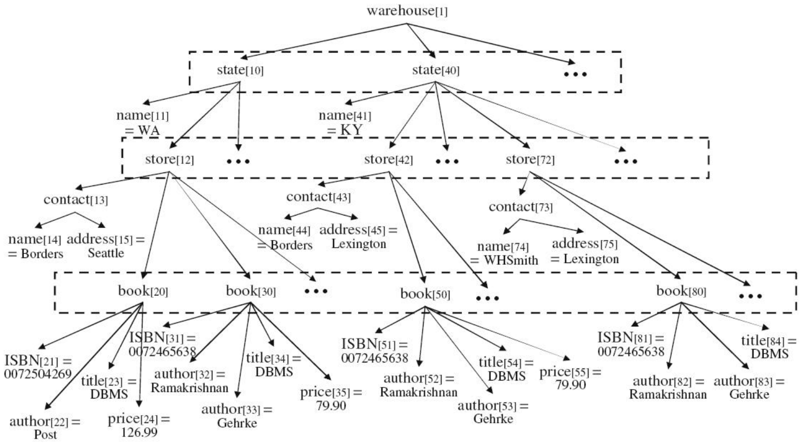
Definition (Data tree) An XML database is defined to be a rooted labeled tree $T = \langle N, \mathcal{P}, \mathcal{V}, n_r \rangle$, where:

- N is a set of labeled data nodes, each $n \in N$ has a label e and a node key that uniquely identifies it in T;
- $n_r \in N$ is the root node;
- $\mathcal{P}$ is a set of parent-child edges, there is exactly one $p = (n', n)$ in $\mathcal{P}$ for each $n \in N$ (except n_r), where $n' \in N, n \neq n', n'$ is called the parent node, n is called the child node;
- $\mathcal{V}$ is a set of value assignments, there is exactly one $v = (n, s)$ in $\mathcal{V}$ for each leaf node $n \in N$, where s is a value of simple type.

Descendant, repeatable element definition

- We assign a node key, referred to as @key, to each data node in the data tree in a pre-order traversal.
- A data element n_k is a descendant of another data element n_1 if there exists a series of data elements n_i, such that $(n_i, n_{i+1}) \in \mathcal{P}$ for all $i \in [1, k-1]$.
- Data element n_k can be addressed using a path expression, $path(n_k) = /e_1/\dots/e_k$, where e_i is the label of n_i for each $i \in [1, k]$, $n_1 = n_r$, and $(n_i, n_{i+1}) \in \mathcal{P}$ for all $i \in [1, k-1]$.
- A data element n_k is called **repeatable** if e_k corresponds to a *set element* in the schema.
- Element n_k is called a *direct descendant* of element n_a, if n_k is a descendant of n_a, $path(n_k) = \dots/e_a/e_1/\dots/e_{k-1}/e_k$, and e_i is not a set element for any $i \in [1, k-1]$.

Warehouse Example



Warehouse Example Scheme

```
warehouse:Rcd
        state:SetOf Rcd
           name: str
           store: SetOf Rcd
             contact: Rcd
              name: str
              address: str
              book: SetOf Rcd
                  ISBN: str
                  author: SetOf str
                  title: str
                  price: str
```

Element-value equality

Definition(Element-value equality) Two data elements n_1 of $T_1 = \langle N_1, \mathcal{P}_1, \mathcal{V}_1, n_{r1} \rangle$ and n_2 of $T_2 = \langle N_2, \mathcal{P}_2, \mathcal{V}_2, n_{r2} \rangle$ are *element-value equal* (written as $n_1 =_{ev} n_2$) if and only if:

- n_1 and n_2 both exist and have the same label;
- There exists a set M, such that for every pair $(n_1', n_2') \in M$, $n_1' =_{ev} n_2'$, where n_1', n_2' are children elements of n_1, n_2, respectively. Every child element of n_1 or n_2 appears in exactly one pair in M.
- $(n_1, s) \in \mathcal{V}_1$ if and only if $(n_2, s) \in \mathcal{V}_2$,where s is a simple value.

Example Data elements node 30 and 50 are *element value equal* if and only if the subtrees rooted at those two elements are identical when the order among sibling elements is ignored.

Path-value equality

Definition(Path-value equality) Two data element paths p_1 on $T_1 = \langle N_1, \mathcal{P}_1, \mathcal{V}_1, n_{r1} \rangle$ and p_2 on $T_2 = \langle N_2, \mathcal{P}_2, \mathcal{V}_2, n_{r2} \rangle$ are *path-value equal* (written as $T_1.p_1 =_{pv} T_2.p_2$) if and only if there is a set M' of matching pairs where

- For each pair $m' = (n_1, n_2)$ in M', $n_1 \in N_1$, $n_2 \in N_2$, $path(n_1) = p_1$, $path(n_2) = p_2$, and $n_1 =_{ev} n_2$;
- All data elements with path p_1 in T_1 and path p_2 in T_2 participate in M', and each such data element participates in only one such pair.

Value equality between two paths is complicated by the fact that a single path can match multiple data elements in the data tree. This definition consider two paths value equal if each node which is pointed to by one path must have a corresponding node that is pointed to by the other path, where the two nodes are element value equal.

Generalized tree tuple

Definition A generalized tree tuple of data tree $T = \langle N, \mathcal{P}, \mathcal{V}, n_r \rangle$, with regard to a particular data element n_p (called pivot node), is a tree $t^T_{n_p} = \langle N^t, \mathcal{P}^t, \mathcal{V}^t, n_r \rangle$, where:

- $N^t \subseteq N$ is the set of nodes, $n_p \in N^t$;
- $\mathcal{P}^t \subseteq \mathcal{P}$ is the set of parent-child edges;
- $\mathcal{V}^t \subseteq \mathcal{V}$ is the set of value assignments;
- n_r is the same root node in both $t^T_{n_p}$ and T;
- $n \in N^t$ if and only if:
  - n is a descendant or ancestor of n_p in T, or
  - n is a non-repeatable direct descendant of an ancestor of n_p in T ;
- $(n_1, n_2) \in \mathcal{P}^t$ if and only if $n_1 \in N^t$, $n_2 \in N^t, (n_1, n_2) \in \mathcal{P}$;
- $(n, s) \in \mathcal{V}^t$ if and only if $n \in N^t$, $(n, s) \in \mathcal{V}$.

Tuple class

- A generalized tree tuple is a data tree projected from the original data tree.
- It has an extra parameter called a pivot node. In contrast with tree tuple defined in Arenas and Libkin's article, which separate sibling nodes with the same path at all hierarchy levels, the generalized tree tuple separate sibling nodes with the same path above the pivot node
- Based on the pivot node, generalized tree tuples can be categorized into tuple classes:

Definition(Tuple class) A tuple class C_p^T of the data tree T is the set of all generalized tree tuples t_n^T, where $path(n) = p$. Path p is called the *pivot path*.

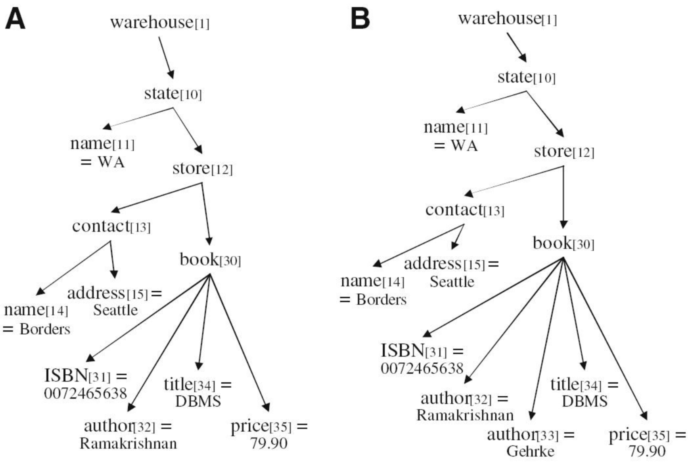


Figure: Example tree tuple

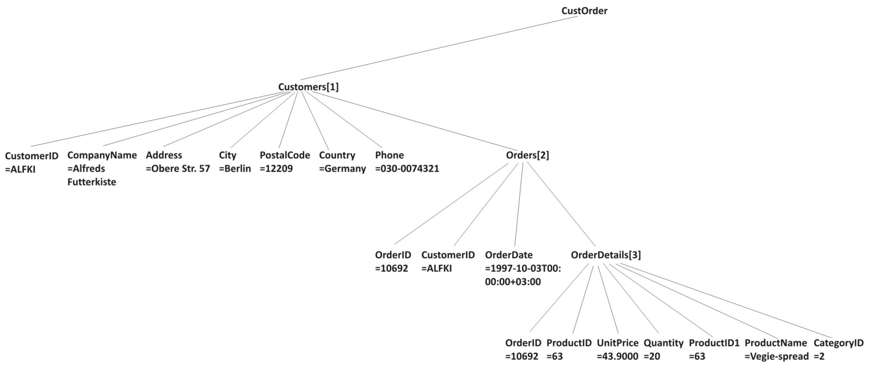


Figure: Example tree tuple

XML Functional Dependency

Definition(XML FD) An XML FD is a triple $\langle C_p, LHS, RHS \rangle$, written as $LHS \rightarrow RHS$ w.r.t. C_p, where C_p denotes a tuple class, LHS is a set of paths (P_{li} , $i = [1, n]$) relative to p, and RHS is a single path (P_r) relative to p.
An XML FD holds on a data tree T (or T satisfies an XML FD) if and only if for any two generalized tree tuples $t_1, t_2 \in C_p$
- $\exists i \in [1, n]$, $t_1.P_{li} = \perp$ or $t_2.P_{li} = \perp$, or
- If $\forall i \in [1, n]$, $t_1.P_{li} =_{pv} t_2.P_{li}$, then
$t_1.P_r \neq \perp, t_2.P_r \neq \perp, t_1.P_r =_{pv} t_2.P_r$.
A null value, $\perp$, results from a path that matches no node in the tuple, and $=_{pv}$ is the path-value equality defined previous.

XML Functional Dependency Example

Example

(XML FD) In our running example whenever two products agree on $\mathrm{ProductID}$ values, they have the same $\mathrm{ProductName}$. This can be formulated as follows:

$./\mathrm{ProductID} \rightarrow ./\mathrm{ProductName}$ w.r.t. $C_{OrderDetails}$

Another example is:

$./\mathrm{ProductID} \rightarrow ./\mathrm{CategoryID}$ w.r.t $C_{OrderDetails}$

Example

(XML FD) In warehause tree:

$./\mathrm{ISBN} \rightarrow ./\mathrm{title}$ w.r.t C_{book}

$../\mathrm{contact}/\mathrm{name},./\mathrm{ISBN} \rightarrow ./\mathrm{price}$ w.r.t C_{book}

$./\mathrm{ISBN} \rightarrow ./\mathrm{author}$ w.r.t C_{book}

$./\mathrm{author},./\mathrm{title} \rightarrow ./\mathrm{ISBN}$ w.r.t C_{book}

Trivial XML FD

Definition: (Trivial XML FD) An XML FD $\langle C_p, LHS, RHS \rangle$ is trivial if:

1. $RHS \in LHS$, or
2. For any generalized tree tuple in C_p, there is at least one path in LHS that matches no data element.

The 2. point can arrise, because of the existence of **Choice** elements.

Example If *Contact* is a **Choice** element instead of **Rcd**, i.e. it can have either name or address as its childs, but not both, then the XML FD:

$\langle C_{store}, ./contact/name, ./contact/address, ./@key \rangle$

is trivial, because no C_{store} tuple will have both LHS node.

XML key

Definition (XML key) An XML Key of a data tree T is a pair $\langle C_p, LHS \rangle$, where T satisfies the XML FD $\langle C_p, LHS, ./@key \rangle$.

Example

We have the XML FD: $\langle C_{Orders}, ./OrderID, ./@key \rangle$, which implies that $\langle C_{Orders}, ./OrderID \rangle$ is an XML key.

Example

$\langle C_{State}, ./name \rangle$
$\langle C_{Store}, ./contact/name, ./contact/address \rangle$ are XML keys.

Structurally redundant XML FDs

Theorem

- Let $FD = \langle C_p, LHS, RHS \rangle$,
- **if** none of the paths in LHS and RHS specifies a data element that is descendent of the pivot node in the tuple,
- **then** FD **holds** on a data tree T
- **if and only** if $FD' = \langle C_{p'}, LHS', RHS' \rangle$ **holds** on T, where
- $C_{p'}$ is the lowest-repeatable-ancestor tuple class of C_p
- paths in LHS' and RHS' are equivalent to paths in LHS and RHS (i.e. they correspond to the same absolute paths).

Example

$../\mathrm{ISBN} \rightarrow ../\mathrm{title}$ w.r.t C_{author}
is structurally redundant with
$./\mathrm{ISBN} \rightarrow ./\mathrm{title}$ w.r.t C_{book}

Interesting XML FD

Tuple classes with repeatable pivot paths are called *essential tuple classes*.

Definition(Interesting XML FD) An XML FD $\langle C_p, LHS, RHS\rangle$ is *interesting* if it satisfies the following conditions:

- RHS $\notin$ LHS;
- C_p is an essential tuple class;
- RHS matches to descendent(s) of the pivot node.

An interesting XML FD is a non-trivial XML FD with an essential tuple class and is not structurally redundant to any other XML FD.

XML data redundancy

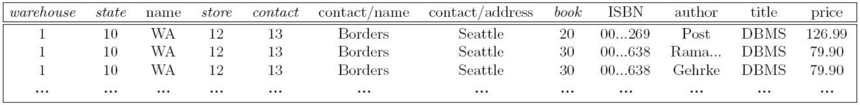
Definition(XML data redundancy) A data tree T contains a redundancy if and only if T satisfies an interesting XML FD $\langle C_p, LHS, RHS \rangle$, but does not satisfy the XML Key $\langle C_p, LHS \rangle$.
Intuitively:

- if $\langle C_p, LHS \rangle$ is not a key for T, then there exists two distinct tuples in C_p that share the same LHS.
- T satisfies $\langle C_p, LHS, RHS \rangle$, so RHS of these two tuples must be value equal
- so: data is redundantly stored

GTT-XNF

Definition(GTT-XNF) An XML schema S is in GTT-XNF given the set of all satisfied interesting XML FDs if and only if for each such XML FD ($\langle C_p, LHS, RHS \rangle$), $\langle C_p, LHS \rangle$ is an XML key.
Intuitively: GTT-XNF disallows any satisfied interesting XML FD that indicates data redundancies.
Rule 1 (Reflexivity) $LHS \rightarrow P_1$ w.r.t. C_p is satisfied if $P_1 \subseteq LHS$.
Rule 2 (Augmentation) $LHS \rightarrow P_1$ w.r.t. $C_p \Rightarrow \{LHS, P_2\} \rightarrow P_1$ w.r.t. C_p.
Rule 3 (Transitivity) $LHS \rightarrow P_1$ w.r.t. $C_p \wedge \ldots \wedge LHS \rightarrow P_n$ w.r.t. $C_p \wedge \{P_1, \ldots, P_n\} \rightarrow P$ w.r.t. $C_p \Rightarrow LHS \rightarrow P$ w.r.t. C_p.

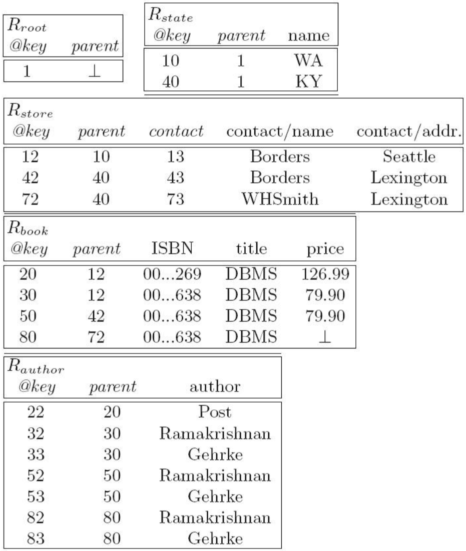
XML Data Flat Representation



warehouse	*state*	name	*store*	*contact*	contact/name	contact/address	*book*	ISBN	author	title	price
1	10	WA	12	13	Borders	Seattle	20	00...269	Post	DBMS	126.99
1	10	WA	12	13	Borders	Seattle	30	00...638	Rama...	DBMS	79.90
1	10	WA	12	13	Borders	Seattle	30	00...638	Gehrke	DBMS	79.90
...	...	...	...	...	...	...	...	...	...	...	...

Figure: One relation for the whole XML data

XML Data Hierarhical Representation



R_{root}

@key	*parent*
1	$\perp$

R_{state}

@key	*parent*	name
10	1	WA
40	1	KY

R_{store}

@key	*parent*	*contact*	contact/name	contact/addr.
12	10	13	Borders	Seattle
42	40	43	Borders	Lexington
72	40	73	WHSmith	Lexington

R_{book}

@key	*parent*	ISBN	title	price
20	12	00...269	DBMS	126.99
30	12	00...638	DBMS	79.90
50	42	00...638	DBMS	79.90
80	72	00...638	DBMS	$\perp$

R_{author}

@key	*parent*	author
22	20	Post
32	30	Ramakrishnan
33	30	Gehrke
52	50	Ramakrishnan
53	50	Gehrke
82	80	Ramakrishnan
83	80	Gehrke

Intra-relation FDs/Keys, Inter-relation FDs/Keys

Example

(XML FD) In warehause tree:
$./\mathrm{ISBN} \rightarrow ./\mathrm{title}$ w.r.t C_{book} intra-relation FD
$../\mathrm{contact}/\mathrm{name},./\mathrm{ISBN} \rightarrow ./\mathrm{price}$ w.r.t C_{book} inter-relation FD
$./\mathrm{ISBN} \rightarrow ./\mathrm{author}$ w.r.t C_{book} inter-relation FD
$./\mathrm{author},./\mathrm{title} \rightarrow ./\mathrm{ISBN}$ w.r.t C_{book} inter-relation FD

Yu, C., Jagadish, H. V.: XML schema refinement through redundancy detection and normalization. VLDB J. 17(2): 203-223 (2008) presents algorithm based on **partitioning** to detect intra-relation FDs, another very complicated for inter-relation FDs.

Eliminating redundancy-indicating FDs

- if $\langle C_p, LHS\rangle$ is not a key for T
- T satisfies $\langle C_p, LHS, RHS\rangle$, so RHS is redundantly stored
- to eliminate such FD, the schema element corresponding to RHS is moved into a new schema location, such that those data elements are no longer redundantly stored.
- Let Σ be the set of redundancy-indicating FDs.

Example

$./\mathrm{ISBN} \rightarrow ./\mathrm{title}$ w.r.t C_{book}

$\{\ ../../\mathrm{name},\ ../\mathrm{contact}/\mathrm{name}, ./\mathrm{ISBN}\ \} \rightarrow ./\mathrm{price}$ w.r.t C_{book}

Assumption:

$\{\ ../\mathrm{name},\ ./\mathrm{contact}/\mathrm{name}\}$ is a key for C_{store}.

Local/global XML FD

Definition (Local/global XML FD) An XML FD $\langle C_p, LHS, RHS \rangle$ is local if there exists $LHS' \subset LHS$ such that $\langle C_{p'}, LHS' \rangle$ is an XML key, where $C_{p'}$ is an ancestor tuple class of C_p (i.e. p' is a prefix of p). Otherwise, the FD is *global*.

Example

$./\mathrm{ISBN} \rightarrow ./\mathrm{title}$ w.r.t C_{book} is global, because no subset of its LHS is a key for any tuple class above C_{book}

means: 2 books, regardless whether they are under the same store or state, if they have the same ISBN, then they will have the same title

Example

$\{ ../../\mathrm{name}, ../\mathrm{contact}/\mathrm{name}, ./\mathrm{ISBN} \} \rightarrow ./\mathrm{price}$ w.r.t C_{book} is local, because $\{ ../../\mathrm{name}, ../\mathrm{contact}/\mathrm{name} \}$ is a key for C_{store}.

means: $\mathrm{state\ name}$ and $\mathrm{store\ name}$ uniquely identifies each store, any 2 books, if they have the same ISBN, they will have the same price, as long as they are under the same store.

Eliminate global FD

Procedure 1

- Let $F = \{P_1, \ldots, P_n\} \rightarrow P_r$ w.r.t. C_p be a redundancy indicated global FD on Schema S_{root};
- $\{e_i | i \in [1, n]\}$ and $\{\tau_i | i \in [1, n]\}$ be the sets of schema element labels and types, respectively, associated with each P_i;
- e_r and τ_r be the schema element label and type,respectively, associated with P_r;
- τ_{parent} be the schema element type of the parent element of P_r;
- τ_{root}=Rcd$[e'_1 : \tau'_1, \ldots, e'_m : \tau'_m]$ be the element type of the *root* element.

Eliminating redundancy:

- Create a new schema element with label e_{new} and type τ_{new}=SetOfRcd$[e_1 : \tau_1, \ldots, e_n : \tau_n, e_r : \tau_r\ ]$
- Set τ_{root}=Rcd$[e'_1 : \tau'_1, \ldots, e'_m : \tau'_m, e_{new} : \tau_{new}]$
- Remove $(e_r : \tau_r)$ from τ_{parent}

Eliminate global FD Example

$./\mathrm{ISBN} \rightarrow ./\mathrm{title}$ w.r.t C_{book}

Scheme after eliminating global FD:

```
warehouse:Rcd
        state:SetOf Rcd
           name: str
           store: SetOf Rcd
             contact: Rcd
              name: str
              address: str
              book: SetOf Rcd
                  ISBN: str
                  author: SetOf str
                  price: str
        new—book:SetofRcd
            ISBN: str
            title:str
```

Adjusting FD

- remove F from Σ.
- the semantics of F is captured by: $\{P_1, \ldots, P_n\} \rightarrow P_r$ w.r.t. C_{new}, it is not redundancy, does not need to be added to Σ
- remove all FDs from Σ that are affected by the move of P_r.

Example

$\{$ ./author, ./title$\} \rightarrow$ ./ISBN w.r.t C_{book}
is removed, because it is not valid.
It is safe to do, because ISBN is no longer redundant.

Eliminate local FD

Procedure 2

- Let $F = \{P_1, \dots, P_{k-1}, P_k, \dots, P_n\} \rightarrow P_r$ w.r.t. C_p be a redundancy indicated local FD on Schema S_{root};
- $\{P_1, \dots, P_{k-1}\}$ is the key for C'_p;
- C'_p is is an ancestor tuple class of C_p and there is no other subset L of $\{P_i | i \in [1, n]\}$ such that L is a key for C''_p
- C''_p is an ancestor of C_p and a descendant of C'_p (i.e. C'_p is the lowest tuple class that can be identified)
- $\{e_i | i \in [k, n]\}$ and $\{\tau_i | i \in [k, n]\}$ be the sets of schema element labels and types, respectively, associated with each P_i;
- e_r and τ_r be the schema element label and type,respectively, associated with P_r;
- τ_{parent} be the schema element type of the parent element of P_r;
- $\tau_{p'}$=Rcd$[e'_1 : \tau'_1, \dots, e'_m : \tau'_m]$ be the element type of the schema element corresponding to the pivot path of $C_{p'}$.

Eliminate redundancy

- Create a new schema element with label e_{new} and type τ_{new}=SetOfRcd[$e_k : \tau_k, \ldots, e_n : \tau_n, e_r : \tau_r$]
- Set $\tau_{p'}$=Rcd[$e'_1 : \tau'_1, \ldots, e'_m : \tau'_m, e_{new} : \tau_{new}$]
- Remove $(e_r : \tau_r)$ from τ_{parent}

Explanation

- to eliminate a local FD like $\{$ ../../name, ../contact/name,./ISBN $\} \rightarrow$ ./price w.r.t C_{book}
- create a new schema element containing the subset of its LHS (ISBN) that are not part of the key for ancestor tuple class (C_{store}) and RHS element (price)
- put this new element under the schema element corresponding to the pivot path of the ancestor tuple class (/warehouse/state/store).
- RHS element is removed from its original position

Eliminate redundancy cont.

- by creating the new schema element under the non-root ancestor, fewer elements needs to be copied under the new scheme
- after the the modification of the scheme remove any FD that is affected by the move of P_r

Scheme after eliminating local FD:

```
warehouse:Rcd
       state:SetOf Rcd
          name: str
          store: SetOf Rcd
            contact: Rcd
             name: str
             address: str
             book: SetOf Rcd
                 ISBN: str
                 author: SetOf str
                 title: str
             new—book: SetOfRcd
                 ISBN: str
                 price: str
```

Special case for Procedure 2

- if the entire LHS of the FD is a key for some ancestor tuple class

Example In DBLP scheme *year* of an *article* is determined by the identity (@key) of the *issue* containing the *article*

- instead of creating a new scheme element containing a single element *year*
- we move *year* after *issue*

Normalization algorithm

```
Algorithm SchemaNormalization:
Input: Schema $S$, a set $\Sigma$ of redundancy-indicating FDs,
       a set $\Upsilon$ of XML Keys (to determine local vs. global FD).
 1. Group FDs in $\Sigma$ based on tuple class $C_p$ and $LHS$,
    order them according to decreasing depth of $C_p$ (lowest first)
    and increasing number of paths in $LHS$ second (fewest first);
 2. while $\Sigma$ is not empty:
 3.   let $\mathcal{F}$ be the first set of FDs in $\Sigma$ with the same LHS and $C_p$;
 4.   let $F$ be the first FD in $\mathcal{F}$;
 5.   if $F$ is local: Modify $S$ by applying Procedure 2;
 6.   else Modify $S$ by applying Procedure 1; // $F$ is global
 7.   foreach additional $F' \in \mathcal{F}$:
 8.     Modify $S$ in the same way by applying procedures 1 or 2,
          but using the new schema element already created in
          dealing with $F'$
 9.   remove all FDs in $\mathcal{F}$ from $\Sigma$;
10.   foreach $F \in \Sigma$:
11.     if $F$ is no longer valid: remove $F$ from $\Sigma$;
12.     if $F$ is now structurally-redundant:
13.       convert $F$ into its equivalent $F'$ that is not structurally
            redundant and add $F'$ to $\Sigma$ (see Theorem 2);
Output: Schema $S$, the modified redundancy-free schema
```

Normalization algorithm explanation

- FDs are grouped according to their LHS
  Example
  $./\mathrm{ISBN} \rightarrow ./\mathrm{title}$ w.r.t C_{book}
  $./\mathrm{ISBN} \rightarrow ./\mathrm{author}$ w.r.t C_{book}
- if they are dealt separately two new scheme element are created
- FDs are processed according to the number of paths in their LHS to reduce the storage cost.
  Example
  $\{ ./\mathrm{title}, .\mathrm{author}\} \rightarrow ./\mathrm{ISBN}$ w.r.t C_{book}
- If this FD is processed first, then the elements title and author will remain under *book*, not ISBN

Normalization algorithm explanation cont.

- FDs are processed according to the hierarchy depth of their tuple class in a bottom-up fashion (lowest first)
- this is because during the process of FDs for a lower hierarchy tuple class, redundancy-indicating FDs for a higher hierarchy tuple class may be created
- algorithm terminates because each application of Procedure 1 and 2 either removes one redundancy-indicating FD or converts one redundancy-indicating FD into another one with a tuple class at a higher hierarchy

GTT-XNF Scheme of Warehouse xml data

- Eliminating first:
  $./\mathrm{ISBN} \rightarrow ./\mathrm{title}$ w.r.t C_{book}
  $./\mathrm{ISBN} \rightarrow ./\mathrm{author}$ w.r.t C_{book}
- this FD is no longer redundancy indicating
  $\{ ../../\mathrm{name}, ../\mathrm{contact}/\mathrm{name}, ./\mathrm{ISBN} \} \rightarrow ./\mathrm{price}$ w.r.t C_{book}

```
warehouse:Rcd
        state:SetOf Rcd
           name: str
           store: SetOf Rcd
             contact: Rcd
                name: str
                address: str
             book: SetOf Rcd
                  ISBN: str
                  price: str
        new—book: SetOfRcd
            ISBN: str
            title: str
            author: SetOf str
```

Introduction to FCA

- From a philosophical point of view a concept is a unit of thoughts consisting of two parts:
  - the extension, which are objects;
  - the intension consisting of all attributes valid for the objects of the context;
- Formal Concept Analysis (FCA) introduced by Wille gives a mathematical formalization of the concept notion.
- A detailed mathematic foundation of FCA can be found in:
  - Ganter, B., Wille, R.: Formal Concept Analysis. Mathematical Foundations. Springer, Berlin-Heidelberg-New York. (1999)
- Formal Concept Analysis is applied in many different realms like psychology, sociology, computer science, biology, medicine and linguistics.
- FCA is a useful tool to explore the conceptual knowledge contained in a database by analyzing the formal conceptual structure of the data.

Introduction to FCA

- FCA studies how objects can be hierarchically grouped together according to their common attributes. In FCA the data is represented by a cross table, called formal context.
- A *formal context* is a triple (G, M, I).
- G is a finite set of objects
- M is finite set of attributes
- The relation $I \subseteq G \times M$ is a binary relation between objects and attributes.
- Each couple $(g, m) \in I$ denotes the fact that the object $g \in G$ is related to the item $m \in M$.

Introduction to FCA

- For a set $A \subseteq G$ of objects we define

$$A' := \{m \in M \mid gIm \text{ for all } g \in A\}$$

  the set of all attributes common to the objects in A.
- Dually, for a set $B \subseteq M$ of attributes we define

$$B' := \{g \in G \mid gIm \text{ for all } m \in B\}$$

  the set of all objects which have all attributes in B.
- A *formal concept* of the context $\mathbb{K} := (G, M, I)$ is a pair (A, B) where $A \subseteq G$, $B \subseteq M$, $A' = B$, and $B' = A$.
- We call A the *extent* and B the *intent* of the concept (A, B).
- The set of all concepts of the context (G, M, I) is denoted by $\mathfrak{B}(G, M, I)$.

Example formal context

- The following cross table describes for some hotels the attributes they have.
- In this case the objects are: *Oasis, Royal, Amelia, California, Grand, Samira*;
- and the attributes are: *Internet, Sauna, Jacuzzi, ATM, Babysitting*.
- $(\{California, Grand\})' := \{Sauna, Jacuzzi\}$.

	Internet	Sauna	Jacuzzi	ATM	Babysitting
Oasis	X	X	X	X	
Royal	X	X	X		
Amelia	X				X
California		X	X		
Grand		X	X		
Samira				X	X

Table: Formal context of the Hotel facilities example

FCA tool to detect XML FDs

- we elaborate an FCA based tool that identify functional dependencies in XML documents.
- to achieve this, as a first step, we have to construct the Formal Context of functional dependencies for XML data.
- we have to identify the objects and attributes of this context in case of XML data.
- tuple-based XML FD notion proposed in the above section suggests a natural technique for XFD discovery
- XML data can be converted into a fully unnested relation, a single relational table, and apply existing FD discovery algorithms directly.
- given an XML document, which contains at the beginning the schema of the data, we create generalized tree tuples from it.

Construct Formal Context of XML FDs

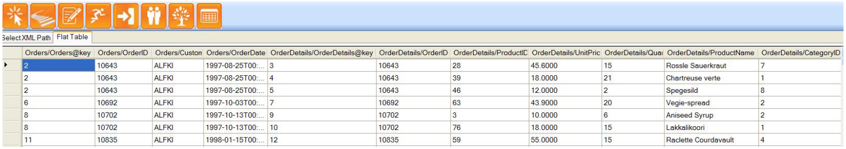
- each tree tuple in a tuple class has the same structure, so it has the same number of elements.
- we use the flat representation which converts the generalized tree tuples into a flat table
- each row in the table corresponds to a tree tuple in the XML tree
- in the flat table we insert non-leaf and leaf level elements (or attributes) from the tree
- for non-leaf level nodes the associated keys are used as values
- we include non-leaf level nodes with associated key values, to detect XML keys

Flat table for tuple class C_{Orders}

Example

Let us construct the flat table for tuple class C_{Orders}. There are two non-leaf nodes:

- Orders, appears as Orders@key
- OrderDetails, appears as OrderDetails@key.

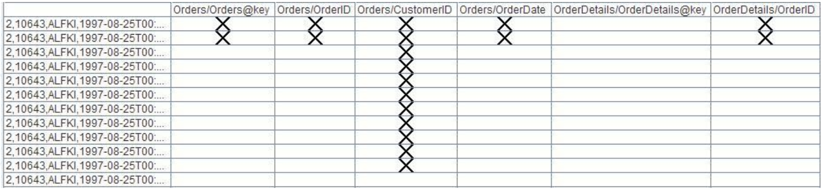


Orders/Orders@key	Orders/OrderID	Orders/Custon	Orders/OrderDate	OrderDetails/OrderDetails@key	OrderDetails/OrderID	OrderDetails/ProductID	OrderDetails/UnitPric	OrderDetails/Quai	OrderDetails/ProductName	OrderDetails/CategoryID
2	10643	ALFKI	1997-08-25T00:...	3	10643	28	45.6000	15	Rossle Sauerkraut	7
2	10643	ALFKI	1997-08-25T00:...	4	10643	39	18.0000	21	Chartreuse verte	1
2	10643	ALFKI	1997-08-25T00:...	5	10643	46	12.0000	2	Spegesild	8
6	10692	ALFKI	1997-10-03T00:...	7	10692	63	43.9000	20	Vegie-spread	2
8	10702	ALFKI	1997-10-13T00:...	9	10702	3	10.0000	6	Aniseed Syrup	2
8	10702	ALFKI	1997-10-13T00:...	10	10702	76	18.0000	15	Lakkalikoori	1
11	10835	ALFKI	1998-01-15T00:...	12	10835	59	55.0000	15	Raclette Courdavault	4

Formal Context for class C_{Orders}

- Context's Attributes: *PathEnd*/*ElementName*
  - for non-leaf level nodes: the name of the attribute is constructed as: <ElementName>+"@key" and its value will be the associated key value
  - for non-leaf level nodes: the element names of the leaves.
- Context's Objects: the objects are considered to be the tree tuple pairs, actually the tuple pairs of the flat table. The key values associated to non-leaf elements and leaf element's values are used in these tuple pairs.
- Context's Properties: the mapping between objects and attributes is defined by a binary relation, this incidence relation of the context shows which attributes of this tuple pairs have the same value.

Beginning of the Formal Context of functional dependencies for tuple class C_{Orders}

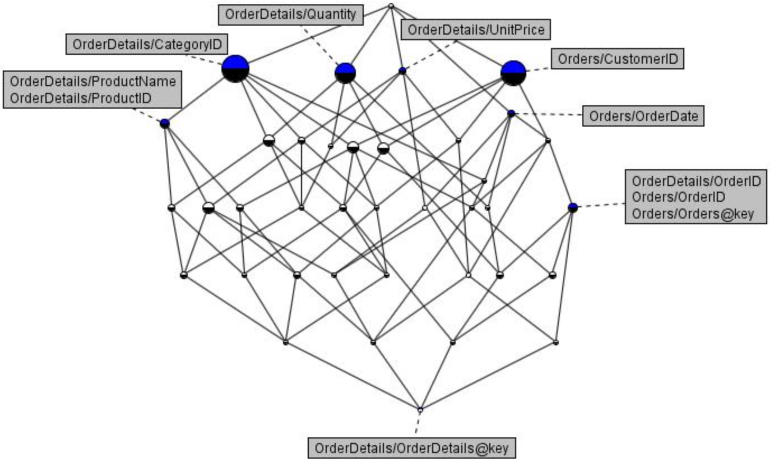


	Orders/Orders@key	Orders/OrderID	Orders/CustomerID	Orders/OrderDate	OrderDetails/OrderDetails@key	OrderDetails/OrderID
2,10643,ALFKI,1997-08-25T00:...	X	X	X	X		X
2,10643,ALFKI,1997-08-25T00:...	X	X	X	X		X
2,10643,ALFKI,1997-08-25T00:...			X			
2,10643,ALFKI,1997-08-25T00:...			X			
2,10643,ALFKI,1997-08-25T00:...			X			
2,10643,ALFKI,1997-08-25T00:...			X			
2,10643,ALFKI,1997-08-25T00:...			X			
2,10643,ALFKI,1997-08-25T00:...			X			
2,10643,ALFKI,1997-08-25T00:...			X			
2,10643,ALFKI,1997-08-25T00:...			X			
2,10643,ALFKI,1997-08-25T00:...			X			
2,10643,ALFKI,1997-08-25T00:...						

- the analyzed XML document may have a large number of tree tuples.
- we filter the tuple pairs and we leave out those pairs in which there are no common attributes, by an operation called "clarifying the context", which does not alter the conceptual hierarchy.

Concept Lattice of functional dependencies' Formal Context for tuple class C_{Orders}

- we run the Concept Explorer (ConExp) engine to generate the concepts and create the concept lattice.



Processing the Output of FCA

- a concept lattice consists of the set of concepts of a formal context and the subconcept-superconcept relation between the concepts;
- every circle represents a formal concept;
- each concept is a tuple of a set of objects and a set of common attributes, but only the attributes are listed;
- an edge connects two concepts if one implies the other directly;
- each link connecting two concepts represents the transitive subconcept-superconcept relation between them;
- the top concept has all formal objects in its extension;
- the bottom concept has all formal attributes in its intension.

The relationship between FDs in databases and implications in FCA

a FD $X \rightarrow Y$ holds in a relation r over R iff the implication $X \rightarrow Y$ holds in the context (G, R, I) where $G = \{(t_1, t_2) | t_1, t_2 \in r, t_1 \neq t_2\}$ and $\forall A \in R$, $(t_1, t_2) IA \Leftrightarrow t_1[A] = t_2[A]$.

- objects of the context are couples of tuples and each object intent is the agree set of this couple
- the implications in this lattice corresponds to functional dependencies in XML.

Example

$$\langle C_{Orders}, ./OrderID, ./CustomerID \rangle$$
$$\langle C_{Orders}, ./Orders@key, ./CustomerID \rangle$$
$$\langle C_{Orders}, ./OrderDetail/OrderID, ./CustomerID \rangle$$

Reading the Concept Lattice

- in the lattice we list only the attributes, these are relevant for our analysis;
- let there be a concept, labeled by A, B and a second concept labeled by C, where A, B and C are FCA attributes;
- let concept labeled by A, B be the subconcept of concept labeled by C;
- tuple pairs of concept labeled by A, B have the same values for attributes A, B, but for attribute C too.
- tuple pairs of concept labeled by C do not have the same values for attribute A, nor for B, but have the same value for attribute C.
- tuple pairs of every subconcept of concept labeled by A, B have the same values for attributes A, B.
- the labeling of the lattice is simplified by putting each attribute only once, at the highest level.

Reading the Concept Lattice

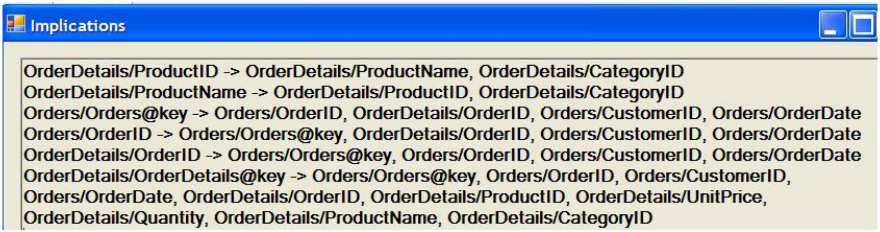
- we analyze attributes A and B:
    - if we have only $A \longrightarrow B$, then A would be a subconcept of B;
    - if only $B \longrightarrow A$ holds then B should be a subconcept of A;
    - we have $A \longrightarrow B$ and $B \longrightarrow A$, that's why they come side by side in the lattice.
    - So attributes from a concept imply each other.

Example

We have the next XML FDs:

$$\langle C_{Orders}, ./OrderID, ./OrderDetails/OrderID \rangle$$
$$\langle C_{Orders}, ./OrderID, ./Orders@key \rangle$$
$$\langle C_{Orders}, ./Orders@key, ./OrderID \rangle$$
$$\langle C_{Orders}, ./Orders@key, ./OrderDetails/OrderID \rangle$$
$$\langle C_{Orders}, ./OrderDetails/OrderID, ./Orders@key \rangle$$
$$\langle C_{Orders}, ./OrderDetails/OrderID, ./OrderID \rangle$$

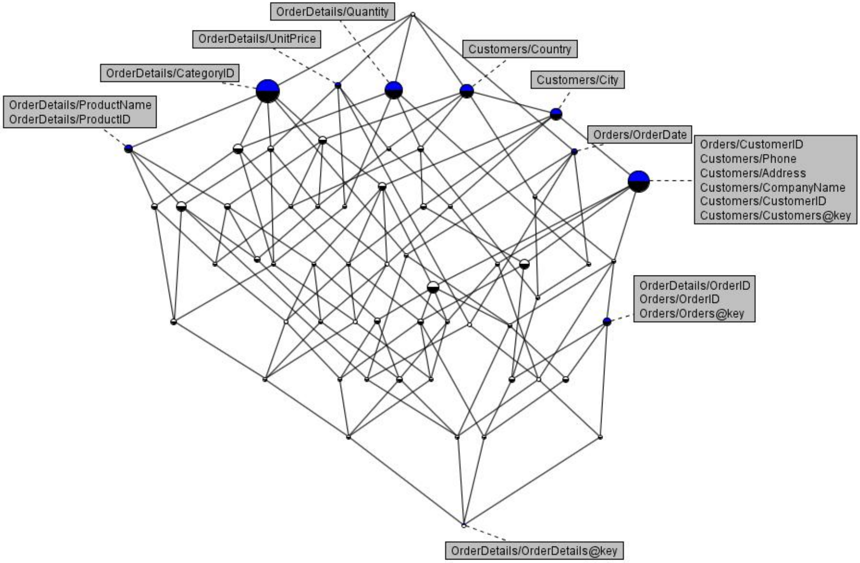
The functional dependencies found by software FCAMineXFD



```
OrderDetails/ProductID -> OrderDetails/ProductName, OrderDetails/CategoryID
OrderDetails/ProductName -> OrderDetails/ProductID, OrderDetails/CategoryID
Orders/Orders@key -> Orders/OrderID, OrderDetails/OrderID, Orders/CustomerID, Orders/OrderDate
Orders/OrderID -> Orders/Orders@key, OrderDetails/OrderID, Orders/CustomerID, Orders/OrderDate
OrderDetails/OrderID -> Orders/Orders@key, Orders/OrderID, Orders/CustomerID, Orders/OrderDate
OrderDetails/OrderDetails@key -> Orders/Orders@key, Orders/OrderID, Orders/CustomerID,
Orders/OrderDate, OrderDetails/OrderID, OrderDetails/ProductID, OrderDetails/UnitPrice,
OrderDetails/Quantity, OrderDetails/ProductName, OrderDetails/CategoryID
```

Figure: Functional dependencies in tuple class C_{Orders}

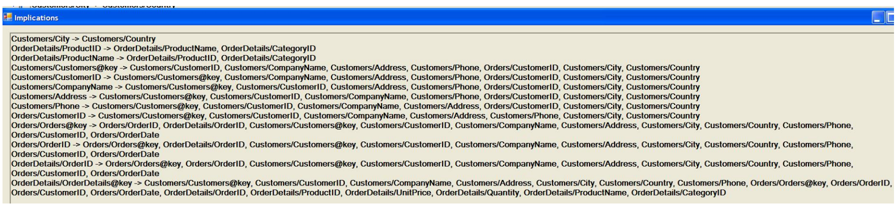
The concept lattice for the whole XML document



Data Analysis

- we can see the hierarchy of the analyzed data:
- the node labeled by *Customers*/*Country* is on a higher level, than node labeled by *Customers*/*City*;
- the Customer's node with every attribute is a subconcept of node labeled *Customers*/*City*;
- in our XML data, every customer has different name, address, phone number, so these attributes appear in one concept node and **imply each other**;
- the Orders node in XML is child of Customers, in the lattice, the node labeled with the key of Orders node, is subconcept of Customers node, so the hierarchy is visible;
- these are 1:n relationships, from Country to City, from City to Customers, from Customers to Orders.
- information about products is on the other side of the lattice; Products are in n:m relationship with Customers, linked by OrderDetail node in this case.

FDs for the whole XML document


Implications

```
Customers/City -> Customers/Country
OrderDetails/ProductID -> OrderDetails/ProductName, OrderDetails/CategoryID
OrderDetails/ProductName -> OrderDetails/ProductID, OrderDetails/CategoryID
Customers/Customers@key -> Customers/CustomerID, Customers/CompanyName, Customers/Address, Customers/Phone, Orders/CustomerID, Customers/City, Customers/Country
Customers/CustomerID -> Customers/Customers@key, Customers/CompanyName, Customers/Address, Customers/Phone, Orders/CustomerID, Customers/City, Customers/Country
Customers/CompanyName -> Customers/Customers@key, Customers/CustomerID, Customers/Address, Customers/Phone, Orders/CustomerID, Customers/City, Customers/Country
Customers/Address -> Customers/Customers@key, Customers/CustomerID, Customers/CompanyName, Customers/Phone, Orders/CustomerID, Customers/City, Customers/Country
Customers/Phone -> Customers/Customers@key, Customers/CustomerID, Customers/CompanyName, Customers/Address, Orders/CustomerID, Customers/City, Customers/Country
Orders/CustomerID -> Customers/Customers@key, Customers/CustomerID, Customers/CompanyName, Customers/Address, Customers/Phone, Customers/City, Customers/Country
Orders/Orders@key -> Orders/OrderID, OrderDetails/OrderID, Customers/Customers@key, Customers/CustomerID, Customers/CompanyName, Customers/Address, Customers/City, Customers/Country, Customers/Phone,
Orders/CustomerID, Orders/OrderDate
Orders/OrderID -> Orders/Orders@key, OrderDetails/OrderID, Customers/Customers@key, Customers/CustomerID, Customers/CompanyName, Customers/Address, Customers/City, Customers/Country, Customers/Phone,
Orders/CustomerID, Orders/OrderDate
OrderDetails/OrderID -> Orders/Orders@key, Orders/OrderID, Customers/Customers@key, Customers/CustomerID, Customers/CompanyName, Customers/Address, Customers/City, Customers/Country, Customers/Phone,
Orders/CustomerID, Orders/OrderDate
OrderDetails/OrderDetails@key -> Customers/Customers@key, Customers/CustomerID, Customers/CompanyName, Customers/Address, Customers/City, Customers/Country, Customers/Phone, Orders/Orders@key, Orders/OrderID,
Orders/CustomerID, Orders/OrderDate, OrderDetails/OrderID, OrderDetails/ProductID, OrderDetails/UnitPrice, OrderDetails/Quantity, OrderDetails/ProductName, OrderDetails/CategoryID
```

Finding XML keys

FDs with RHS as ./@key values can be used to detect the keys in XML.

In tuple class C_{Orders} we have XML FD:

- $\langle C_{Orders}, ./OrderID, ./@key \rangle$, which implies that
  - $\langle C_{Orders}, ./OrderID \rangle$ is an XML key.
- $\langle C_{Orders}, ./OrderDetails/OrderID, ./@key \rangle$, so
  - $\langle C_{Orders}, ./OrderDetails/OrderID \rangle$ is an XML key too.

In tuple class $C_{Customers}$ software found XML FD:

- $\langle C_{Customers}, ./CustomerID, ./@key \rangle$, which implies that
  - $\langle C_{Customers}, ./CustomerID \rangle$ is an XML key.
- other detected XML keys are:
  - $\langle C_{Customers}, ./Orders/CustomerID \rangle$;
  - $\langle C_{Customers}, ./CompanyName \rangle$;
  - $\langle C_{Customers}, ./Address \rangle$;
  - $\langle C_{Customers}, ./Phone \rangle$.

Detecting XML data redundancy

- having the set of functional dependencies for XML data in a tuple class, we can detect interesting functional dependencies.
- in essential tuple class C_{Orders} an interesting FD: $\langle C_{Orders}, ./OrderDetails/ProductID, ./OrderDetails/ProductName\rangle$
- but $\langle C_{Orders}, ./OrderDetails/ProductID\rangle$ is not an XML key
  - **So it is a data redundancy.**
- the same reason applies for XML FD $\langle C_{Orders}, ./OrderDetails/ProductName, ./OrderDetails/ProductID\rangle$.
- the other XML FD's have as LHS a key for tuple class C_{Orders}.

Conclusions

- This paper introduces an approach for mining functional dependencies in XML documents based on FCA.
- Based on the flat representation of XML, we constructed the concept lattice.
- We analyzed the resulted concepts, which allowed us to discover a number of interesting dependencies.
- Our framework offers an graphical visualization for dependency exploration.

Future Work

- given the set of dependencies discovered by our tool:
- propose a normalization algorithm for converting any XML schema into a correct one