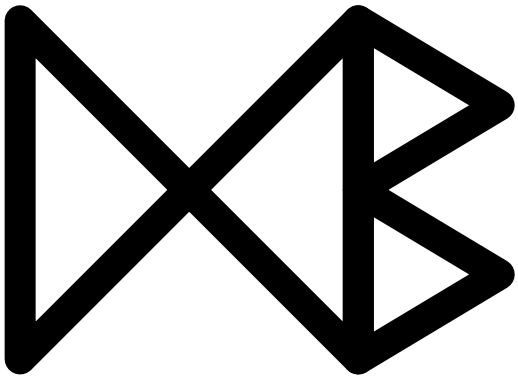


RELAXML: Bidirectional Transfer between Relational and XML Data

Steffen Ulsø Knudsen, Torben Bach Pedersen, Christian Thomsen and Kristian Torp

May 11, 2005

TR-11

A DB Technical Report

Title	RELAXML: Bidirectional Transfer between Relational and XML Data
Author(s)	Steffen Ulsø Knudsen, Torben Bach Pedersen, Christian Thomsen and Kristian Torp
Publication History	Extended version of: Steffen Ulsø Knudsen, Torben Bach Pedersen, Christian Thomsen and Kristian Torp: "RELAXML: Bidirectional Transfer between Relational and XML Data" to appear in *Proceedings of the Ninth International Database Applications & Engineering Symposium*, Montreal, Canada, July 2005, 12 pages.

For additional information, see the DB TECH REPORTS homepage: ⟨www.cs.aau.dk/DBTR⟩.

The DB TECH REPORTS icon is made from two letters in an early version of the Rune alphabet, which was used by the Vikings, among others. Runes have angular shapes and lack horizontal lines because the primary storage medium was wood, although they may also be found on jewelry, tools, and weapons. Runes were perceived as having magic, hidden powers. The first letter in the logo is "Dagaz," the rune for day or daylight and the phonetic equivalent of "d." Its meanings include happiness, activity, and satisfaction. The second letter is "Berkano," which is associated with the birch tree. Its divinatory meanings include health, new beginnings, growth, plenty, and clearance. It is associated with Idun, goddess of Spring, and with fertility. It is the phonetic equivalent of "b."

Abstract

In modern enterprises, almost all data is stored in relational databases. Additionally, most enterprises increasingly collaborate with other enterprises in long-running read-write workflows, primarily through XML-based data exchange technologies such as web services. However, bidirectional XML data exchange is cumbersome and must often be hand-coded, at considerable expense. This paper remedies the situation by proposing RELAXML, an automatic and effective approach to bidirectional XML-based exchange of relational data. RELAXML supports re-use through multiple inheritance, and handles both export of relational data to XML documents and (re-)import of XML documents with a large degree of flexibility in terms of the SQL statements and XML document structures supported. Import and export are formally defined so as to avoid semantic problems, and algorithms to implement both are given. A performance study shows that the approach has a reasonable overhead compared to hand-coded programs.

1 Introduction

Most enterprises store almost all data in relational databases. Additionally, most enterprises increasingly collaborate with other enterprises in long-running read-write workflows. This primarily takes place through XML-based data exchange technologies such as web services, which ensures openness and flexibility.

As an example, consider a database for a fictitious grocery supplier. The database has the relations Products(PID, PName), Customers(CID, CName), Orders(OID, CID), and OrderLines(OID, PID, Qty, Date) where Orders.CID references Customers.CID and OID and PID in OrderLines references OID of Orders and PID of Products, respectively. The data is as shown below.

PID	PName
1	Cola
2	Candy
3	Bread

Products

CID	CName
1	Mini Market
2	Smith's
3	Kiosk24

Customers

OID	CID
1	1
2	3
3	1

Orders

OID	PID	Qty	Date
1	1	200	04/03/05
1	3	50	03/01/05
2	2	100	04/05/05
3	2	75	05/01/05

OrderLines

Using a web-service call, a customer, e.g., Mini Market, requests an XML document with information on all their orders and the ordered products, see Figure 1 (for now, please ignore the concept and structure attributes in the root element). To save space, we use attributes in the shown XML, but in RELAXML the user can choose freely between elements and attributes. This document can easily be created by RELAXML. After receiving the document, the customer updates it to change the quantity of the bread ordered and the delivery date for the candy, and sends it back to the supplier using another web-service call. The database can then be automatically updated by RELAXML to reflect the changes made to the XML document. Using traditional approaches, significant hand-coding would be necessary.

This paper presents RELAXML, a flexible approach to bidirectional data transfer between relational databases and XML documents. Figure 2 shows the procedure when RELAXML exports relational data to an XML document. An export is specified using a *concept* (a view-like construct), and a *structure definition*, which specify the data to export and the structure of the exported XML document, respectively. From the concept, SQL that extracts the data, is generated, resulting in a *derived table* that can be changed by user-specified *transformations*. The resulting data is exported to an XML document with an XML Schema

```
<Orders concept="B.rxc" structure="B.rxs">
   <Customer CID="1">Mini Market</Customer>
   <Order OID="1">
      <OrderLines>
         <Product PID="1" Qty="200" Date="04/03/05">Cola</Product>
         <Product PID="3" Qty="50" Date="03/01/05">Bread</Product>
      </OrderLines>
   </Order>
   <Order OID="3">
      <OrderLines>
         <Product PID="2" Qty="75" Date="05/01/05">Candy</Product>
      </OrderLines>
   </Order>
</Orders>
```

Figure 1: Example of an XML document.

specified by the structure definition. Using both concepts and structure definitions separates data from structure, i.e., a single concept can be associated with multiple structure definitions. The import procedure is basically the reverse of the procedure shown in Figure 2 and allows for insert, update, and delete of data from the database.

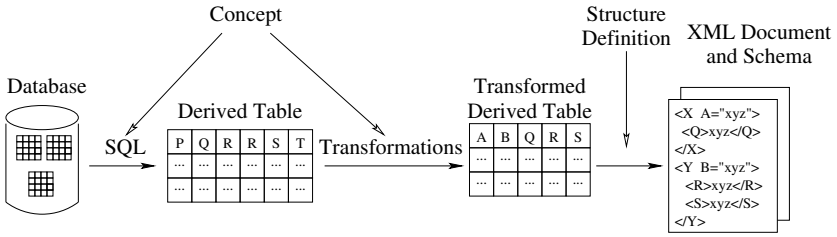


Figure 2: The export procedure.

The SQL statement used for an export can include inner and outer joins plus filters. The structure of the XML documents is very flexible and supports, e.g., *grouping* (or *nesting*) of XML elements, data as XML elements or attributes, and additional container XML elements. Export and import are formally defined, including definitions of concepts, structure definitions, and transformations. In addition, it is specified how to determine at export time if an XML document may be imported into the database again and how an XML document must be self-contained if the data is to be imported into an empty database, so that integrity constraints are not violated. Algorithms for export and import are given. Performance studies of the DBMS independent prototype show that the algorithms are efficient, have a reasonable overhead compared to hand-coded programs, and can handle large documents ($>$ 200 MB) with a small main memory usage.

The mapping of XML data to new (specialized) relational schemas has been widely studied [2, 18]. The mapping of the result of an SQL query to an XML document (termed an *export*) has also been widely studied [7, 11, 17, 18, 19], and recently SQL/XML [13] has been proposed as a standard for this mapping. However, unlike RELAXML, none of this work supports the *import* of XML documents into an *existing* database. Only few papers [1, 3, 6] have studied how to do a bidirectional (both export and import) mapping between *existing* databases and XML documents. Again, note that SQL/XML only maps from databases to XML documents. Further, some of the bidirectional approaches have limited capabilities, i.e., can only map

an XML document to a single table [3]. A number of so-called *XML-enabled databases* with extensions for transferring data between XML documents and themselves exist [5, 8]. However, the solutions in these products are vendor specific and do not provide full support for transferring data into existing databases with given schemas.

There exist many *middleware products* (such as RELAXML) for transferring data between databases and XML documents [3], including products that can either export, import, or both. Examples are JDBC2-XML [12], DataDesk [14] and XML-DBMS [4]. Of these, XML-DBMS is the most interesting since it can perform both import and export. It uses a mapping language to provide flexible mappings between XML elements and database columns and mappings can be automatically generated from a DTD or database schema. However, compared to RELAXML, XML-DBMS is not as scalable as it uses DOM instead of SAX, does not support inheritance or transformations, and gives no guarantee for import at export time. In [1], bidirectional transfer of data is also considered. The main differences are that [1] creates new views in the underlying database and updates through these views. Each query (tree) may need multiple new views. In contrast, we update the underlying database tables directly and do not need to modify the database schema at all. Additionally, we consider θ-joins (instead of only inner joins), we provide a performance study of an open-source prototype, and we support multiple inheritance. Compared to existing work on updating relational databases through views [9, 10], the RELAXML approach differs as 1) the SQL update statements are not known, but instead deduced from the XML document by RELAXML and 2) the needed execution order of the update statements (due to integrity constraints), is deduced from the underlying database schema by RELAXML.

The remainder of the paper is structured as follows. Sections 2 and 3 provide definitions of basic constructs, and export and import, respectively. Sections 4 and 5 present the design of export and import, respectively. Experimental results are presented in Section 6. Finally, Section 7 concludes the paper and points to directions of future research.

2 Basic definitions

We now formally define the used constructs. When transferring relational data to an XML document, the user may want to *transform* the data in some way, e.g., by converting a price to another currency. This transformation multiplies the price by c when exporting to XML, and divides the price by c when importing from the XML.

In the following, we consider *rows* as relational tuples, i.e., a row has a number of unique attribute names (also denoted columns) and for each attribute name, an attribute value exists. For a row r and an attribute name a, $r[a]$ denotes the attribute value for a in r. Further, $\mathcal{N}(r)$ denotes the set of attribute names in r. The set of all rows is denoted $\mathcal{R}$. With this, we can define transformations formally.

Definition 2.1 (Transformation) *A* transformation t *is a function* $t : \mathcal{R} \rightarrow \mathcal{R}$ *that fulfills* $\mathcal{N}\big(t(r)\big) = \mathcal{N}\big(t(s)\big)$ *for all* $r, s \in dom(t)$ *where* $dom(t)$ *is the domain of* t.

The set of attribute names added by a transformation t is denoted $\alpha(t)$, and the set of names deleted by a transformation t is denoted $\delta(t)$. Formally, $\alpha(t) = \mathcal{N}(t(r)) \setminus \mathcal{N}(r)$ and $\delta(t) = \mathcal{N}(r) \setminus \mathcal{N}(t(r))$ for all $r \in dom(t)$. Note that for efficiency reasons, transformations are pipe-lined in the RELAXML implementation.

We now define *join tuples*, which are used for defining concepts formally. Intuitively, a join tuple defines a relation derived by joining existing relations like an SQL query, i.e., the relations to join, the join operator(s), and the join predicate(s) should be specified. For example, the join tuple for the example in Section 1 says that Orders and OrderLines are inner joined on the OIDs, the resulting relation is inner joined with Customers on the CIDs, and finally, this result is inner joined with Products on the PIDs.

Let θ be a theta join, and LOJ/ROJ/FOJ be a left/right/full outer join. $\Omega = I \cup O$ where $I = \{\theta\}$ and $O = \{LOJ, ROJ, FOJ\}$ is the set of RELAXML join operations (the operators in O are neither commutative nor associative).

Definition 2.2 (Join tuple) *A* join tuple *is a three-tuple of the form* $((r_1, \ldots, r_m), (\omega_1, \ldots, \omega_{m-1}), (p_1, \ldots, p_{m-1}))$ *for* $m \geq 1$ *and where*

1) r_i *is a relation or another join tuple for* $1 \leq i \leq m$

2) $\omega_i \in \Omega$ *for* $1 \leq i \leq m-1$

3) p_i *is a predicate for* $1 \leq i \leq m-1$.

Further, we require that if $\omega_i \in O$ *then* $\omega_j \in I$ *for* $j < i$.

For an $\omega \in \Omega$ and a predicate p, $A\ \omega^p\ B$ denotes the join (of type ω) where the predicate p must be fulfilled. For a given join tuple, it is then possible to compute a relation by means of the $eval$ function where

$$eval(r) = \begin{cases} eval(r_1)\ \omega_1^{p_1}\ eval(r_2)\ \omega_2^{p_2}\ \cdots\ \omega_{m-1}^{p_{m-1}}\ eval(r_m) & \text{if } r = ((r_1, \ldots, r_m), \\ & (\omega_1, \ldots, \omega_{m-1}), \\ & (p_1, \ldots, p_{m-1})) \\ r & \text{if } r \text{ is a relation.} \end{cases}$$

To avoid ambiguity, only one join operator from O can be used in a join tuple since they are neither commutative nor associative. If more are needed, several join tuples are used (similar to requiring parentheses in an expression).

A *concept* is used for defining which data to transfer, and thus includes a join tuple, along with a list of columns used in a projection of the relation resulting from the join tuple, a predicate to restrict the considered row set, and a list of transformations to apply. Further, as concepts support inheritance, a concept also lists its ancestors (if any). An example of concept inheritance appears in Example 2.4.

Definition 2.3 (Concept) *A* concept *is a 6-tuple* (n, A, J, C, f, T) *where* n *is the concept's caption,* A *is a possibly empty sequence of unique parent concepts to inherit from,* J *is a join tuple,* C *is a set of included columns from the base relations of* J*,* f *is a row filter predicate, and* T *is a possibly empty sequence of transformations to be applied.*

For a concept with join tuple J and ancestors $a_1, \ldots, a_n$, we require that the relations $D(a_1), \ldots, D(a_n)$ (defined below) are included by J.

The relation valued function D computes the base (not yet transformed) data for a concept. For a concept $k = (n, (a_1, \ldots, a_m), J, C, f, T)$, D is defined as follows, where $\nu(c)$ denotes the name of the table from which a column c originates and $cols(x)$ gives all columns in a relation x.

$$D(k) = \bigcirc_{c \in C} \rho_{[\langle k \rangle \# \nu(c) \$ c / c]}(\pi_{C \cup \{\tilde{c} \mid \tilde{c} \in cols(D(a_i)),\ i=1,\ldots,n\}}(\sigma_f(eval(J)))) \tag{1}$$

First, $eval$ computes the relation that holds the data from the base relations, followed by performing a selection and then a projection of all columns included by k or any of its ancestors. Finally, a renaming schema of the columns included by k is used by means of the rename operator where # and $ represent separator characters. This 3-part naming schema (concept name, table name, column name) is necessary in order have a one-to-one mapping from the columns of $D(k)$ to the columns of the database. With the renaming schema, both table and concept names are part of the column names of $D(k)$, which is necessary in order to separate the scopes of different concepts.

As shown above, $D(k)$ denotes a relation with the data of the concept k before transformations are applied. For a concept k with parent list $(a_1, \ldots, a_u)$ and transformation list $T = (t_1, \ldots, t_p)$, the resulting data is given by the relation valued function R defined as follows.

$$R(k) = \bigcup_{d \in D(k)} \big(\gamma(k)\big)\big(d\big), \tag{2}$$

where

$$\gamma(k) = \Big(\bigcirc_{n \in ((\cup_{t \in T} \alpha(t)) \setminus (\cup_{t \in T} \delta(t)))} \rho_{[\langle k \rangle \# n/n]}\big(t_p \circ \cdots \circ t_1\big)\Big) \circ \gamma(a_u) \circ \cdots \circ \gamma(a_1).$$

When a concept inherits from parent concepts, parent transformations are evaluated before child transformations. When all the transformations have been evaluated, all the attribute names they have added are prefixed with an encoding of the concept, so it is possible to distinguish between identically named attributes added by transformations from different concepts. With the definition in (2), a problem may emerge if a concept is inherited from twice, namely that, when transformed, an attribute included by a common ancestor could have an unexpected value, set by a transformation included by another concept. To avoid problems, we require for a concept's parent list L that $\psi(L)$ does not contain duplicates, where ψ is recursively defined as $\psi(()) = ()$ and $\psi(l_1 :: \cdots :: l_n) = l_1 :: \cdots :: l_n :: \psi\big(p(l_1)\big) :: \cdots :: \psi\big(p(l_n)\big)$ where $p(x)$ is concept x's list of parents.

Example 2.4 *Consider again the data in Section 1. We now define a concept A which extracts information on which customers have placed orders, and another concept, B, which inherits from A and adds details on the ordered products. B restricts the data to the customer with CID = 1. Thus, B extracts the data shown in Figure 1. We use C for Customers, O for Orders, OL for OrderLines, and P for Products.*

$$\begin{aligned} A =& (CustomersWithOrders, (), \\ & ((C, O), (\theta), (C.CID = O.CID)), \\ & \{C.CID, C.CName, O.OID\}, (\textit{true}), ()) \end{aligned}$$

$$\begin{aligned} B =& (Orders, (A), \\ & ((P, OL, D(A)), (\theta, \theta), ((OL.PID = P.PID), \\ & (OL.OID = A\#O\$OID))), \{P.PID, \\ & P.PName, P.Qty, P.Date\}, A\#C\$CID = 1, ()) \end{aligned}$$

Concept A has the caption CustomersWithOrders and does not inherit from other concepts. The join tuple of A states that C and O must be joined by a θ-join on the CIDs. The columns C.CID, C.CName, and O.OID are included by A. Each row from the join tuple should be included by A (each row fulfills the condition "true"). A does not use any transformations. Concept B has the caption Orders and inherits from A. The join tuple specifies how to join the relations P and OL to the relation found by A, $D(A)$. B adds three columns to those considered by A and adds a row filter such that only rows regarding a specific customer are considered.

A *structure definition* is used to define the structure (i.e., the schema) of the XML containing the data. The structure is described by means of a tree where a node represents an XML element or attribute. The structure definition for the example in Figure 1 is shown in Figure 3. A structure definition has two kinds of elements: elements that hold data but not elements, and elements that only hold other elements. A node in the structure definition can be a node that we *group by*, i.e., in the XML, elements represented by that node

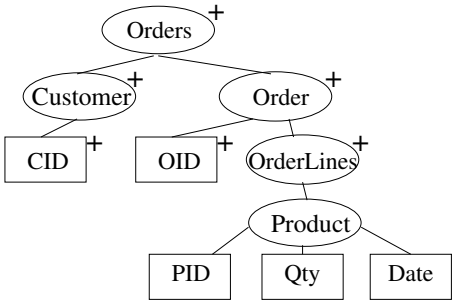


Figure 3: Structure definition example.

are coalesced into one if they have the same data values. The resulting element then holds the children of all the coalesced elements, e.g., informations on a customer and each distinct order only appear once in the XML in Figure 1. This is achieved by using group by nodes (marked with a +) in the structure definition in Figure 3. The names shown are the names used in the XML, not the relational attribute names. Below is the formal definition of structure definitions. Here, an *ordered tree with vertex set* V means that an injective order function $o : V \to \mathbb{N} \cup \{0\}$ exists.

Definition 2.5 (Structure definition) *A* structure definition $S = (V_d, V_s, E)$ *is an ordered rooted tree where* $V_s \cap V_d = \emptyset$ *and* $V = V_s \cup V_d$ *is the set of vertices and* E *is the set of edges. Members of* V_s *and* V_d *are denoted as* structural *and* data *nodes, respectively. A vertex* $v \in V$ *is a tuple* (c, t, g) *where* c *is a name,* $t \in \{element, attribute\}$ *is the type and* $g \in \{true, false\}$ *shows if the XML data is grouped by the vertex. The root* $\rho = (c, element, true) \in V_s$ *and for every* $v = (c, t, g) \in V_s$ *it holds that* $t = element$*. For* $v = (c, t, g) \in V_d$ *it holds that if* $t = attribute$ *then* v *has no children whereas if* $t = element$ *then for each child* (d, u, h) *of* v *we have* $u = attribute$*.*

We say that a structure definition $S = (V_d, V_s, E)$ *complies* with a concept k iff for each column of $R(k)$ there exists exactly one node in V_d with identical name and the name of the root of S equals the caption of the concept k. For a concept k, a vertex $v \in V_d$ represents a column of $R(k)$ and gives rise to elements that hold *data*, while a vertex in V_s does not represent a column and gives rise to *structural* elements holding other elements. We let the function κ be a mapping between the names of the vertices and XML tag names. Thus, the XML elements represented by v in the structure definition will be named $\kappa(v)$.

In order to represent a meaningful XML structure, a structure definition must be *valid*. For a vertex v, let $De(v)$ denote the set of descendants of v and $Ch(v)$ the set of children of v.

Definition 2.6 (Valid structure definition) *A structure definition* $S = (V_d, V_s, E)$ *with root* ρ *and order* o *is* valid *iff*

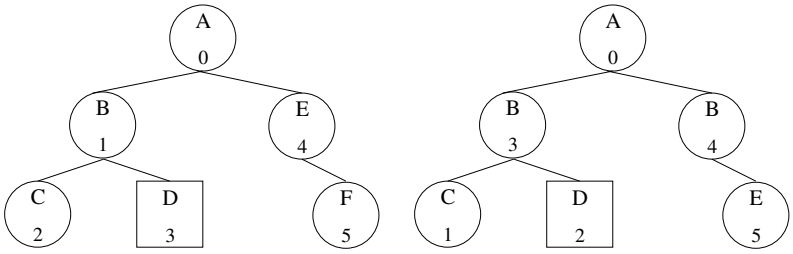
S1) $o(\rho) = 0$

S2) For all $v \in (V_d \cup V_s)$ *we have for all* $c \in De(v)$ *that* $o(c) > o(v)$

S3) For all $a, b \in (V_d \cup V_s)$*,* $b \notin De(a)$*, we have for all* $c_a \in De(a)$ *that* $o(a) < o(b) \Rightarrow o(c_a) < o(b)$

S4) For all $v \in (V_d \cup V_s)$ *there do not exist* $c, d \in Ch(v)$ *such that* $c \neq d$ *and* $\kappa(c) = \kappa(d)$

S5) For all $(c, t, g) \in Ch(\rho)$ *we have* $t = element$*.*



(a) Valid (b) Not valid

Figure 4: Structure definitions.

Requirements S1, S2 and S3 intuitively correspond to saying that the order numbers are assigned in a depth-first fashion (this is automatically done by the RELAXML implementation and is thus of no concern for the user). Requirements S4 and S5 say that siblings should be distinguishable by having non-identical names and that the root should have only element children. Figure 4(a) shows an example of a valid structure definition. A node of type $element$ is represented as a circle and a node of type $attribute$ is represented as a square. A letter represents the name and a number the order. The structure definition shown in Figure 4(b), is not valid since the A element has two children with the name B, and the B with order 3 has children with lower order than itself.

For a vertices v, f, p, we say that f is a *following relative* to v if f has higher order than v, and p is a *preceeding relative* to v if p has lower order than v. It is not possible to group by an arbitrary node in the tree, so we define a *valid grouping* below. Note that any valid structure definition that does not group by any nodes (the root node is trivially grouped by), is automatically a valid grouping.

Definition 2.7 (Valid grouping) *A* valid grouping *is a valid structure definition $S = (V_d, V_s, E)$ where for $v = (n, t, g) \in (V_d \cup V_s)$ where $g = true$ the following holds.*

G1) For all preceding relatives (a, b, c) of v, $c = true$.

G2) A following relative (a, b, c) of v exists with $c = false$.

G3) If a following relative that is not a descendant of v exists, then for all descendants (a, b, c) of v, it holds that $c = true$.

G4) For all children (a, b, c) of v where $b = attribute$, it also holds that $c = true$.

Requirement G1 says that when we group by a node, we have to group by its ancestors as well. Otherwise there would be no elements of the same type to coalesce in the XML. Further, the requirement ensures efficiency at import time. Without it, we risk that to regenerate a single row, many rows have to be read partly, e.g., if we in Figure 4(a) only grouped by E, we could have to read many B elements before the first E element, leading to a significant memory usage. Requirement G2 ensures that for each row exported, at least one element is written to the XML, ensuring that each exported row can be recreated at import time such that a grouping is not lossy. To understand requirement G3, consider Figure 4(a). If we group by B, we should also group by C and D. Then, an entire element, including children, represented by B can be written when the data in one row has been seen. Without G3, this would not hold, and the writing of the element represented by E would have to be postponed. Requirement G4 ensures that a specific element's attributes only appear once in that element.

Consider again Figure 4(a). Now assume that we group by E. Then to have a valid grouping we must also group by A, B, C, and D, but not by F.

3 Export and import

3.1 Export

We now define the function XML that computes XML containing the data from R. The function XML uses two auxiliary functions: $Element$, which adds an element tag, and $Content$, which adds the content of an element. These two functions depend on the structure definition used (given by the subscript). In the following, we consider the concept c with caption n and the valid grouping $\lambda = (V_d, V_s, E)$ that has the root ρ, complies with c and has order o. A string and a white space added to the XML is written in **another font** and as an underscore, respectively.

$$XML(c, \lambda) = \texttt{<}n\texttt{\_concept="}\langle c\rangle\texttt{"\_structure="}\langle\lambda\rangle\texttt{">}Content_\lambda(\rho, R(c))\texttt{</}n\texttt{>} \tag{3}$$

The function XML adds the root element of the XML which is named after the caption of the concept c. Further, informations about the concept and structure definition are always added. The content (i.e., children) of the root element is added by $Content$. In the following, for a vertex $v = (x, y, z)$ in the structure definition, we let $v^1 = x$. Further, we let $Att(v)$ denote the ordered (possibly empty) list of attribute children of v. Then for $v = (N, t, g)$ with $Att(v) = (a_1, \ldots, a_n)$ and $Ch(v) \setminus Att(v) = \{e_1, \ldots, e_m\}$, we define $\bar{v}$ as

$$\bar{v} = \begin{cases} (v^1, a_1^1, \ldots, a_n^1) & \text{if } v \in V_d \\ (a_1^1, \ldots, a_n^1, \bar{e}_1, \ldots, \bar{e}_m) & \text{if } v \in V_s,\ g = true \text{ and } v \text{ has a following relative } f \notin De(v) \\ (a_1^1, \ldots, a_n^1) & \text{otherwise.} \end{cases} \tag{4}$$

$\bar{v}$ is used in the following to find lists of columns that should be used in projections when data to be put in the XML should be found. The function $Element_\lambda$ is defined as

$$Element_\lambda(v, P) = \bigcirc_{\forall r \in \pi_{\bar{v}}(P)} \left(\texttt{<}\kappa(v^1)\texttt{\_}\kappa(a_1^1)\texttt{="}r[a_1]\texttt{"\_}\ldots\texttt{\_}\kappa(a_n^1)\texttt{="}r[a_n]\texttt{">}Content_\lambda(v, \sigma_{\bar{v}=r}(P))\texttt{</}\kappa(v^1)\texttt{>}\right) \tag{5}$$

for a relation P and a vertex v with $\bar{v} \neq ()$ and with attribute children $\{a_1, \ldots, a_n\}$ where a_i has lower order than a_j for $i < j$. If $\bar{v} = ()$, $Element_\lambda(v, P) = \texttt{<}\kappa(v^1)\texttt{">}Content_\lambda(v, P)\texttt{</}\kappa(v^1)\texttt{">}$.

In (6), where $Ch(v) = \{e_1, \ldots, e_m\}$, $o(e_i) < o(e_j)$ for $i < j$, $(x, y, z) \in \{e_1, \ldots, e_h\}$ implies that $z = true$ and $(x, y, z) \in \{e_{h+1}, \ldots, e_m\}$ implies that $z = false$, we define the function $Content_\lambda$ for a structure node we group by.

$$\begin{aligned} Content_\lambda(v, P) = \bigcirc_{\forall w_1 : w_1 \in \pi_{\bar{e}_1}(P)} \Bigg(& Element_\lambda\big(e_1, \sigma_{\bar{e}_1 = w_1}(P)\big) \\ & \bigcirc_{\forall w_2 : (w_1 :: w_2) \in \pi_{\bar{e}_1, \bar{e}_2}(P)} \Big(Element_\lambda\big(e_2, \sigma_{\bar{e}_1 = w_1, \bar{e}_2 = w_2}(P)\big) \\ & \quad \ldots \\ & \quad \bigcirc_{\forall w_h : (w_1 :: \cdots :: w_h) \in \pi_{\bar{e}_1, \ldots, \bar{e}_h}(P)} \Big(Element_\lambda\big(e_h, \sigma_{\bar{e}_1 = w_1, \ldots, \bar{e}_h = w_h}(P)\big) \\ & \qquad \bigcirc_{\forall r \in \sigma_{\bar{e}_1 = w_1, \ldots, \bar{e}_h = w_h}(P)} \big(Element_\lambda(e_{h+1}, \{r\}) \cdots Element_\lambda(e_m, \{r\})\big) \Big) \cdots \Big) \Big) \end{aligned} \quad \text{if } v \in V_s \text{ and } g = true \tag{6}$$

Equation (6) shows that when we group by the children $e_1, \ldots, e_h$, for each distinct value of the attributes in P that are represented by e_1 and its children, we create an XML element inside which data or other elements are added recursively by means of $Element_\lambda$ which itself uses $Content_\lambda$. After each of these elements for e_1, other elements are added for those attributes that are represented by e_2 and its children. Here we have to ensure that the values for e_1 match such that we correctly group by e_1. After the elements for e_2, elements for e_3 follow and so on until elements for all group-by nodes have been added. Then elements for non-group-by nodes are added. For these nodes exactly one tuple is used for each application of $Element_\lambda$.

When using $Content_\lambda$ on non-group-by nodes, it is only given one tuple at a time. The definition of $Content_\lambda$ is

$$Content_\lambda(v, \{r\}) = Element_\lambda(e_1, \{r\}) \cdots Element_\lambda(e_m, \{r\}) \quad \text{if } v \in V_s \text{ and } g = false, \qquad (7)$$

where $Ch(v) \setminus Att(v) = \{e_1, \ldots, e_m\}$ and for $i < j : o(e_i) < o(e_j)$. That is, when not grouping by $v \in V_s$, we simply add one element for each element child of v.

Now we define $Content_\lambda$ for nodes in V_d. But from (5) we have that whenever $Content_\lambda$ is given a node $v \in V_d$, the given data has exactly one value for the attribute that v represents. Thus, all that $Content_\lambda$ should do is to add this value: $Content_\lambda(v, P) = Content_\lambda\big(v, \pi_v(P)\big)$ if $|P| > 1$ and $v \in V_d$ and $Content_\lambda\big(v, \{r\}\big) = r[v]$ if $v \in V_d$.

For an example, consider again the data in Section 1 and the structure definition in Figure 3 where the order of nodes is increasing from top to bottom, left to right.

3.2 Import

In the following, we refer to different states of the database. The value of the function D from (1) depends on the state of the database and we therefore refer to the value of $D(c)$ in the specific state s as $D_s(c)$. Now consider an XML document

$$\begin{aligned} X \;=\; & \texttt{<}n\texttt{\_concept="}\langle c\rangle\texttt{"\_structure="}\langle s\rangle\texttt{">} \\ & \cdots\texttt{</}n\texttt{>}, \end{aligned} \qquad (8)$$

created by means of the concept c. By $D^{XML}(X)$ we denote a table with column names as $D(c)$ that holds exactly the values resulting when the inverse transformations from c have been applied to the data in X. It is a requirement for importing Xthat the transformations of c are invertible. This is, in the general case, undecidable and, thus, it is left to the user to ensure this. In the following, we do not consider the possible impacts of triggers and assume that foreign keys can only reference primary keys.

We now give definitions of inserting and updating from the XML. The definitions give the states of the database before and after the modifications, not the individual operations performed on the database. When inserting, the data from the XML file should be inserted into tables in the database, e.g., it should be possible to insert the data in the XML in Figure 1 into a database with a schema similar to that described in Section 1.

Definition 3.1 (Inserting from XML) *For a given database,* inserting *from the XML document X in* (8) *is to bring the database that holds the relations used by c from a valid state a to a valid state b where $D_b(c) = D_a(c) \cup D^{XML}(X)$ such that the only difference between a and b is that tuples may have been added to relations used by c.*

The data in D^{XML} or some of it can be in the database before the insertion but only in such a way that no updates are necessary, i.e., data is only inserted. We now define *updating* from the XML. If an exported XML document is changed and the changes should be propagated to the database, updating is used. For

example, the quantity Cola in line 5 in Figure 1 can be changed to 300. In that case, updating results in the database with the value 300 for Qty in the corresponding row (where OID = 1 and PID = 1) in the OrderLines table shown in Section 1.

Definition 3.2 (Updating from XML) *Consider the XML document X in* (8) *and assume that k is the set of renamed primary keys in the relations used by the concept c.*

For a given database that holds the relations used by c and tuples such that $\pi_k(D^{XML}(X)) \subseteq \pi_k(D_a(c))$, updating *from the XML document X is then, by only updating tuples in base relations used by c, to bring the database from a valid state a to a valid state b where for any tuple t*

$$t \in D^{XML}(X) \Rightarrow t \in D_b(c),$$
$$\big(t \in D_a(c) \wedge \pi_k(\{t\}) \nsubseteq \pi_k\big(D^{XML}(X)\big)\big) \Rightarrow t \in D_b(c)$$
$$t \notin D^{XML}(X) \wedge t \notin D_a(c) \Rightarrow t \notin D_b(c).$$

Informally, the first requirement says that a tuple read from the XML will be in the database after the updating. The second says that a tuple which is in the database before the updating, but not in the XML, is left untouched in the database. The third says that new tuples, that are neither in the database or XML, are not introduced in the database. It is also possible to combine inserting and updating, such that tuples are updated if possible and otherwise inserted. This is called *merging*.

Definition 3.3 (Merging from XML) *Consider the XML document X in* (8) *and assume that k is the set of renamed primary keys in the relations used by the concept c.*

For a given database that holds the relations used by c, merging *from the XML document X is then, by only adding tuples to or updating tuples in base relations used by c, to bring the database from a valid state a to a valid state b where for any tuple t*

$$t \in D^{XML}(X) \Rightarrow t \in D_b(c),$$
$$\big(t \in D_a(c) \wedge \pi_k(\{t\}) \nsubseteq \pi_k\big(D^{XML}(X)\big)\big) \Rightarrow t \in D_b(c)$$
$$t \notin D^{XML}(X) \wedge t \notin D_a(c) \Rightarrow t \notin D_b(c).$$

Notice that the requirement $\pi_k(D^{XML}(X)) \subseteq \pi_k(D_a(c))$ from Definition 3.2 is not present in Definition 3.3. In Definition 3.3 it is implied by $t \in D^{XML}(X) \Rightarrow t \in D_b(c)$ that a tuple in the database in state a, for which a tuple t with matching values for the primary keys exists in $D^{XML}(X)$, is replaced in the state b by t.

Further, *deletion* via XML is supported under some circumstances. To delete, we use a *delete document* which has the same structure as XML documents generated by RELAXML. As many as possible of the tuples in the database with data present in the delete document will be deleted. The reason that everything is not always removed, is that foreign key constraints may inhibit this.

Since delete documents must have the same structure as the XML documents being exported/imported by RELAXML, D^{XML} can be computed for identification of the data to delete from the base relations.

Definition 3.4 (Deleting via XML) *For a given database* deleting *base data by means of the XML document X in* (8)*, is to bring the database that holds the relations used by the concept c from a valid state a to a valid state b. This should be done by deleting the tuples contributing to $D^{XML}(c)$ from the base relations used by c but without violating the integrity constraints of the database.*

It should hold that $t \in D^{XML}(c) \Rightarrow t \notin D_b(c)$ unless some value in t is referenced by a foreign key not included by c and in a relation that has not been declared to set the foreign keys to a null or default value or delete referencing tuples if t is deleted.

The deletion of tuples from relations used by c may lead to updates or deletion of tuples of other relations in the database according to the integrity constraints defined on the database. Apart from this, only tuples in relations used by c will be deleted.

4 Design of export

We now focus on the design and implementation of RELAXML. When exporting, an SQL statement for retrieval of the data is created based on the concept. Figure 5 shows the RELAXML flow when exporting. A JDBC [16] `ResultSet` is decorated with an iterator and a number of transformations. If the XML should be grouped by one or more elements, a database sort is required, since we do not want to hold all data in main memory when writing. Finally, the data rows are handed to an XML writer.

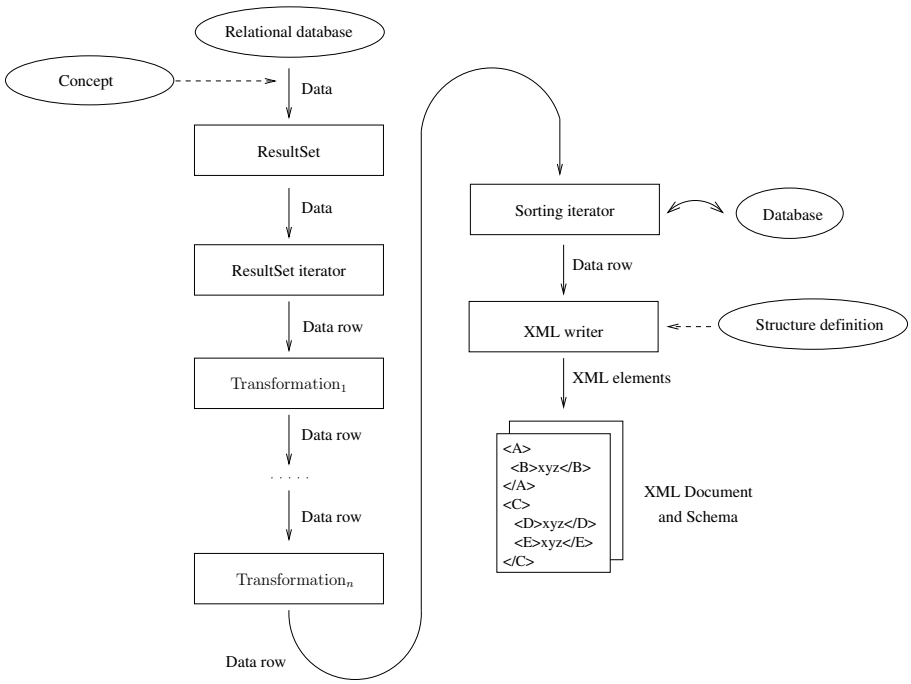


Figure 5: The flow of data in an export.

4.1 SQL statements

The SQL statement to extract data from the database is generated from the concept of the export. SQL statements for parent concepts appear as nested SQL statements in the FROM clause. Note that due to inheritance the actual columns and row filter of the concept consist of the columns and row filters of parent concepts together with included columns and row filter defined in the concept itself.

Example 4.1 *Canonically, the SQL for the retrieval of the data of concepts A and B from Example 2.4 is as follows. Note how the three-part naming schema is imposed and how the SQL code of parent concepts appears as nested sub-queries. Modern DBMSs will, when optimizing, flatten this expression out to a regular four-way join.*

X	Y	Z
1	A	null
2	B	1
3	C	1

Figure 6: Data where dead links can arise.

```
-- Concept A --
SELECT C.CID AS A#C$CID, C.CName AS A#C$CName,
O.OID AS A#O$OID FROM C JOIN O ON (C.CID = O.CID)

-- Concept B --
SELECT A#C$CID, A#C$CName, A#O$OID, P.PID AS B#P$PID,
P.PName AS B#P$PName, OL.Qty AS B#OL$Qty,
OL.Date AS B#OL$Date FROM
OL JOIN P ON (OL.PID = P.PID) JOIN
(SELECT C.CID AS A#C$CID, C.CName AS A#C$CName,
O.OID AS A#O$OID FROM C JOIN O ON (C.CID = O.CID)) RXTMP0
ON (OL.OID = A#O$OID) WHERE (A#C$CID = 1)
```

The code generation shown above generalizes to situations with multiple inheritance. In the implementation, the generated SQL does not contain the long names with #'s and $'s. Instead COL0, COL1, ... are used to avoid problems with DBMSs that do not support special characters and long names. RELAXML automatically handles this mapping.

4.2 Dead links

When exporting a part of the database, we may risk that the data is not self-contained. If an element represents a foreign key it may reference data not included in the XML document. We refer to such a situation as the referencing element having a *dead link*. Figure 6 shows an example where dead links can arise. In the example, Z is a foreign key referencing X. The data in the figure has no dead links but if the tuple with $X = 1$ is removed, the data set contains two dead links since $X = 1$ is referenced by the other tuples.

A dead link does not limit the possibility of updates during import assuming that the element referenced in the dead link still exists in the database. Insertion into a new database is limited by a dead link because of integrity constraints.

In order to *detect dead links* we use Algorithm 1. Here, we iterate through each table used in the derived table. We find the foreign keys and the corresponding referenced keys. In line 5 we find the dead links of the derived table.

When *resolving dead links*, the goal is to expand the selection criteria such that the missing tuples are added. This may be done by adding OR clauses. Note that the SQL statement consists of possibly many nested SELECT statements in the FROM clause and that because of the scope rules, specialized concepts may include a WHERE clause on the columns of ancestor concepts. For this reason, an expansion of the condition must in some cases be added several places in the SQL. This means, that instead of the SQL statement described in Section 4.1, we move the WHERE clauses of the nested queries to the outermost query where they are AND'ed together. The dead link resolution algorithm shown in Algorithm 2 recursively invokes Algorithm 1 to find dead links, manipulating the WHERE clause such that the referenced tuples are included. When a fix point is reached, all the dead links are resolved.

Algorithm 1 Detect dead links

1: **for** each table T part of the derived table DT **do**
2: find the sequence $A = (a_1, \ldots, a_n)$ of foreign keys in T also included in DT
3: find the corresponding sequence $B = ((b_{1,1}, \ldots, b_{1,m_1}), \ldots, (b_{n,1}, \ldots, b_{n,m_n}))$ of candidate keys that are referenced by the foreign keys in A where B is also in DT
4: **for** each $a_i \in A$ **do**
5: $M \leftarrow$ SELECT DISTINCT a_i FROM DT WHERE NOT EXISTS (SELECT B FROM DT WHERE $a_i = b_{i,1}$ OR ... OR $a_i = b_{i,m_i}$)
6: $result[T][a_i] \leftarrow M$
7: **end for**
8: **end for**
9: return $result$

Algorithm 2 Resolve dead links

1: determine the derived table DT which may have dead links
2: $deadlinks =$ find dead links in DT by means of Algorithm 1
3: **for** each $deadlinks[t]$ **do** // Consider tables contributing to DT
4: **for** each $deadlinks[t][a]$ **do** // Consider columns
5: **for** each value v in $deadlinks[t][a]$ **do** // Consider rows (i.e., cells)
6: expand DT's SQL expression with "OR $a = v$"
7: **end for**
8: **end for**
9: **end for**
10: **if** DT's SQL has been expanded **then**
11: Invoke recursive call and find new DT to resolve dead links in
12: **else**
13: return DT which is the original derived table expanded with rows referenced from (previous) dead links
14: **end if**

4.3 XML writing

A desirable characteristic is that we do not want to rely on having all data stored in memory at one time. Thus, the algorithm for writing the XML works such that whenever it gets a new data row, it writes out some of the data to the XML. If grouping is not used, all the data represented in a data row is written to the XML when a data row is received. If grouping is used, some of the data might already be present in the current context in the XML and should not be repeated. To ensure this, the write algorithm compares the new row to write out and the previous row that was written. When grouping is used, it is a precondition that the data rows are sorted by the columns corresponding to the nodes that we group by. This is ensured by a DBMS-based sorting iterator. When grouping by more than one node, the sort order is determined by the order of the structure definition. The procedure for writing the XML is outlined in Algorithm 3.

To support type checking and validation on the XML document structure, RELAXML can generate an XML Schema based on the concept and structure definition. The user chooses at export time if a Schema should be generated or if he wants to use an existing Schema.

In order to generate the XML Schema for an export, we need information on the available columns, their types and the structure of the XML document. A `Concept` object reveals the columns and their SQL types (the types are from `java.sql.Types`) when the `getDataRowTemplate()` method is invoked, and the structure of the XML document is given in the structure definition. For each column in the data

Algorithm 3 Write the XML

- Write the root element including information about concept and structure definition.
- For each data row do:
  - Find a node we do not group by or a mismatching node (considering this and the previous row). The node should have the lowest order possible. If no rows have been seen before, we let this be the node with the lowest order apart from the root. Denote this node x.
  - If we at this point have any unmatched opening tags for x and/or nodes with higher order than x, print closing tags for them.
  - Print opening tags for ancestor nodes of x that are not already open.
  - For x and each of its siblings of type element and container and with higher order do:
    - Print a **<** followed by the tag name for the node
    - Print each tag name for the node's attribute children followed by **="**, the data for the attribute node and a **"**.
    - Print a **>**.
    - If the node is an element, print its data. Else if the node is a container, perform the inner most steps recursively for all its element and container children.
    - If the node is an element or a container that we do not group by or that has a sibling with higher order, print a closing tag for the node.
- Print closing tags for any unmatched opening tags (this at least includes the root tag).

row template, a data type is generated in the XML Schema. The generated type is a `simpleType` which is restricted to the XML Schema type that the columns SQL type is mapped to. It is, however, necessary to take special considerations if the column can hold the value null, i.e., if the column is *nullable*. When exporting, RELAXML will write the null value as a string chosen by the user. But if, for example, a column of type integer is nullable, then the type generated in the XML Schema should allow both integers and the string used to represent the null value. Therefore, the generated type should be a `union` between integers and strings restricted to one string (the one chosen by the user).

The `StructureDefinition` holds a tree of structure nodes representing the tree structure of the XML document. The Schema is generated by traversing this tree. Three types of nodes exist: container nodes, element nodes and attribute nodes. The container nodes have no associated data type since their only content is elements. Elements and attributes on the other hand have associated data types since they have text-only content. These associated data types are those generated as described above.

When container nodes are treated, the Schema construct `sequence` is used. For a container that we do not group by, all its children (which by definition also are not grouped by) are declared inside one `sequence`. This ensures that in the XML instances of the considered element type each has exactly one instance of each of its children element types.

For a container that we do group by there are more considerations to take. If we consider a node x which we group by and which has at least one descendant which we do not group by, then, for each child we group by, we start a new nested `sequence` with `maxOccurs='unbounded'`. These `sequences` are not ended until all children of x have been dealt with. All children of x that we do not group by are declared inside one `sequence` which has the attribute `maxOccurs='unbounded'`. For a structure definition as the one shown in Figure 7 where we assume that we group by A, B and C, these rules ensure that in the

XML an instance of B is always followed by one instance of C which is followed by one or more instances of D. It is, however, possible for an instance of C to follow an instance of D as long as the C instance is followed by at least one other instance of D.

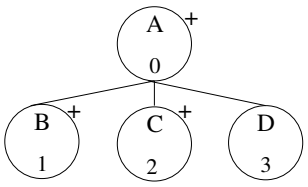


Figure 7: Example of a structure definition.

If we consider a container x where we group by x and all its descendants, then all elements types for children of x are declared inside one single `sequence`.

5 Design of import

The flow of the import operation is the reverse of the flow in Figure 5, except that no sorting iterator is needed. Thus, the XML data is converted to data rows as the XML document is read. These data rows are sent through the inverse transformations and finally an *importer* takes appropriate action based on the data rows. We now discuss insertion, update, and deletion via XML documents.

The user may specify a commit interval such that the importer commits for every n data rows. If $n = \infty$ we may take advantage of deferrable constraints and may do a complete roll-back in case of integrity constraints. If $n \neq \infty$ we cannot defer the deferrable constraints and cannot do a complete roll-back.

We extend the description of concepts given in Section 2 by allowing a column to be marked "not updateable". If this is the case, the data in the database for that column will not be modified by RELAXML.

5.1 Requirements for importing

For a concept to be *insertable*, *updateable*, or *deleteable*, it must fulfill the following requirements.

The **common requirements for insert, update, and delete** are:

c1) all transformations have an inverse;

c2) all columns used in joins occur in the derived table.

Requirement c1) is obvious. Requirement c2) is needed to support θ-joins. If we do not have values for all join columns, we cannot insert/update rows in the underlying tables. If only equijoins were supported, values for half the join columns could be derived.

The **requirements for insert** are:

i1) all non-nullable columns without default values from included tables are in the export;

i2) if a foreign key column is included, then the referenced column is also included;

i3) the exported data contains no dead links;

i4) if all deferrable and nullable foreign keys are ignored, there are no cycles in the part of the database schema used in the export.

Requirement i1) corresponds to Date's rule for insert on a view with projection [9]. Requirements i2) and i3) ensure that inserts do not cause foreign key constraint violations due to foreign keys pointing to non-existing rows. Requirement i4) ensures that rows are inserted in an order in the underlying tables that avoids immediate foreign key constraint violations.

The **requirements for update** are:

u1) each included table has a primary key which is fully included in the export;

u2) primary key values are not updated.

Requirement u1) is a restriction on Date's rule for updating a view with projection [9]. Requirement u2) ensures that primary keys can be used to identify the tuples to update. To ensure that primary keys are not updated, a checksum transformation may be used to include a primary key checksum in the XML file. The **requirements for delete** are the same as for update.

If a concept A uses inheritance, all A's ancestors must be insertable or updateable for A to be insertable or updateable, as we want to ensure that the requirements described above are fulfilled for each row in the export. Otherwise, we would risk that for a concept c, one parent p_1 included some, but not all, columns from a table t required for c to be importable, while another parent p_2 included the remaining columns from t required for c to be importable. But if p_1 only includes the rows where the predicate b is fulfilled whereas p_2 includes those rows where b is not fulfilled, we cannot combine the resulting row parts to insertable rows.

In summary, concepts are much more flexible than modification through SQL views [9], e.g., multiple tables may be updated and consistency is guaranteed. Compared to Date's general specification of modification through views [9] we have stricter requirements on projection for insert and update and do not consider SQL statements with union, intersect, and difference. Concepts involving only joins of tables are insertable and updateable in the same way as views in Date's general specification. Compared to Date we support inheritance and guarantee that updates are consistent as discussed next.

5.2 Avoiding inconsistency

Since the XML document may hold redundant data originating from the same cell in the database, it is a risk that the user makes an *inconsistent update*, e.g., if the same column from a table is selected twice. When the user is editing the XML, he is indirectly making updates to the transformed derived table. But since the derived table can contain redundant data, in the general case it is only in 1NF.

To detect inconsistent updates, we capture which values in the database are read from the XML, as further updates on these would be inconsistent. Thus, for all updated or accepted values (those that were identical in the database and the XML) we capture the table, row and column using a temporary *Touched* table (in the database or main memory). The Touched table has three columns; TableName, PrimaryKeyValue (the composite primary key), and ColumnName. When an update takes place, we check whether the value has been updated before. If so, an exception is raised. If not, the update can take place and information about it is added to the Touched table.

5.3 Inferring a plan for the import

5.3.1 Insert and update

We now consider how to do the actual work when inserting or updating from XML. Later we consider how to delete by means of an XML document. In order to reason on importability of the data of a concept, we build a *database model*, used for inferring database properties, and decide whether there is enough information to import the data and to infer an insertion order. A specific order may be required because of integrity constraints on the database. The database model holds information on the included tables and

columns and their types. Furthermore, the model holds information on the primary keys of the included tables and links (foreign key constraints) between the tables of the concept. We have three types of links in the database model. *Hard links* represent foreign key constraints which are neither deferrable nor nullable; *semi-hard links* represent foreign key constraints which are not deferrable but nullable; *soft links* represent deferrable foreign key constraints.

A concept is viewed as an undirected *concept graph*, where nodes represent tables and edges represent the joins of the concept. Each edge is either an equijoin edge which follows the constraints of the database (represented as a solid line) or a non-equijoin edge or an equijoin edge which does not follow the constraints of the database (both represented as a dotted line). Figure 8 gives examples.

The *execution plan* determines the insertion order. Based on a concept and its database model, it is possible to build an execution plan to be used when importing.

The join types used in the concept, the columns joined and the structure of the database schema influence how to handle an insert or update. The data of a concept may be extracted from the database in many ways, some of which do not reflect the database constraints. For example, a concept may join on two columns not related by a database foreign key and may neglect another foreign key. Thus, data for a single data row may not always be consistent with the foreign key constraints, i.e., these are not fulfilled for the row.

For the import, we construct an insertion order which is a list of table lists. A table list shows tables which may be handled in the same run (parsing) through the XML document, as the data rows are consistent with the database constraints. Thus, the length of the insertion order list is the required number of runs through the XML document.

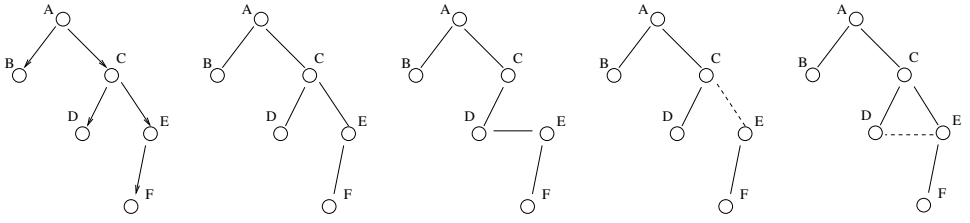


Figure 8: (a) Database model (b)-(e) Concept graphs.

The database model in Figure 8(a) shows that table A has foreign keys to tables B and C, table C has foreign keys to tables D and E, and table E has a foreign key to table F. Figures 8(b)-(e) show the concept graphs for four different concepts using the database modeled in Figure 8(a).

The concept graph in Figure 8(b) shows that the data of each data row is guaranteed to be consistent with the database constraints, as the joins used in the export reflect these constraints and because each join is an equijoin. This is also the case for the Mini Market example in Figure 1. Figure 8(b) gives the insertion order $((F, B, D, E, C, A))$. The data from F is inserted before the data from E because the database model shows that the foreign key in E references F. In Figure 8(c), only equijoins are present, but the foreign key constraint from table C to E is not represented in the concept. Compared to the database model there is also an extra equijoin between the tables D and E. The missing equijoin between tables C and E means that in general we cannot insert the data rows at one time but must break the insertion into multiple phases. A possible insertion order is therefore $((B, F, D, E), (C, A))$. In Figure 8(d), all the constraints of the database model are fulfilled, except that there is a non-equijoin between tables C and E. This leads to the same situation as in Figure 8(c). In Figure 8(e), we get the insertion order $((B, F, D), (E, C, A))$, since D has an equijoin to E. We cannot continue with E in the first run since the D-E join might include a tuple of E, which does not fulfill the foreign key constraint between E and F.

So far, the database models have had no cycles. If cycles are present, we may break a cycle if it has at least one soft link or semi-hard link. A soft link may be deferred and a semi-hard link may be set to null first and updated to the correct value as the final step in the import. We refer to columns having pending updates as *postponed columns*.

Now, the execution plan holds an insertion order (the tables of the concept in a list of table lists) and a list of postponed columns. In the following, let an *independent table* be a table which is guaranteed to fulfill the constraints, i.e., does not have any outgoing links in the current database model. Algorithm 4 takes as input a concept c. In line 1, we build the database model and in line 2 we initialize the set of postponed columns to the empty set. Lines 3-4 remove all soft-links from the database model, i.e., edges representing deferrable constraints. In lines 5-6, we remove all semi-hard links from the database model, i.e., the deferrable and nullable constraints. The columns involved are added to the set of postponed columns. In lines 7-8, we check that there are no cycles in the database model. In this highly unlikely situation we are not able to continue, because there is a cycle of hard links. In line 9, we build the concept graph and in line 10 we initialize the insertion order list to the empty list. The while loop in lines 11-20 builds the insertion order list that consists of table lists. In line 12, the table list that can be inserted in one pass is initialized to the empty list. Lines 13-15 add to the table list, all tables that are joined by equijoins. These tables are removed from the database model. Lines 16-19 do the same for all independent tables. In line 20 the table list is prefixed to the insertion order list. Finally, in line 21 we reverse the insertion list and line 22 returns this list along with the set of postponed columns.

The importer uses the insertion order and handles one data row at a time. The insertion order shows how the importer should progress in the current run through the XML file.

5.3.2 Delete

As described in Definition 3.4, we delete a tuple from the database if there is a match on all values in the corresponding data in the XML document.

We describe an algorithm that handles deletion in database schemas which may be represented as directed acyclic graphs (DAGs) and schemas that hold cycles with cascade actions on all constraints in the cycle (termed *cascade cycles*). In addition, we consider modifications to the delete operation such that a larger set of database schemas can be handled.

When deleting, tuples that are referencing one or more of the tuples to be deleted may block the deletion. Even though a foreign key constraint is fulfilled in the data row, the derived table is denormalized and we cannot delete a tuple that is referenced by another tuple before we reach the last occurrence in the derived table. We do not know which data row is the last with regards to the specific constraint. For this reason, we delete rows from lists of tables which are independent with regards to delete and foreign key constraints.

It is possible to specify delete actions on foreign key constraints, such that a deletion causes a side effect. Delete actions can be defined on foreign key constraints and resolve constraint violations in case referenced tuples are deleted. Possible delete actions are *set null* (the foreign keys are set to null), *set default* (the foreign keys are set to a default value) and *cascade* (the referencing tuples are deleted).

The deletion order is very important. Consider a database schema where table A references table B. A tuple from B may only be deleted when no tuples in A reference the tuple in B. For efficiency reasons we do not want to query the database for referencing tuples for all tuples to delete. Instead we run through the XML twice. First deleting the data from A and then the data from B. Because of the definition of delete we may get a situation where tuples in A are updated as a side effect to deletion in B such that we cannot delete them. This is the case if a set null or set default action is defined in the database such that deletion of a tuple in B has a side effect on tuples in A. If the action is cascading delete, the side effect does the job and one run suffices.

Algorithm 4 Building an execution plan

1: $dbm \leftarrow$ a database model for the concept c
2: $ppCols \leftarrow \emptyset$
3: **if** $commitInterval = \infty$ **then**
4: remove soft links from dbm
5: **end if**
6: **if** cycles are present in dbm **then**
7: break the cycles by postponing a number of semi-hard foreign key columns, add them to $ppCols$
8: **end if**
9: **if** cycles are still present in dbm **then**
10: Error - not importable (cycle of hard links exists)
11: **end if**
12: $conceptGraph \leftarrow$ a concept graph of the concept
13: $iOrder \leftarrow ()$
14: **while** dbm has more nodes **do**
15: $tableList \leftarrow ()$
16: **while** dbm has an independent node n referenced by m where n and m are joined using an equijoin in $conceptGraph$ and n is not joined with other tables **do**
17: $tableList \leftarrow n :: tableList$
18: $dbm \leftarrow dbm$ without n
19: **end while**
20: $indep \leftarrow$ independent nodes in dbm
21: **for** each node $node$ in $indep$ **do**
22: $tableList \leftarrow node :: tableList$
23: $dbm \leftarrow dbm$ without $node$
24: **end for**
25: $iOrder \leftarrow reverse(tableList) :: iOrder$
26: **end while**
27: $iOrder \leftarrow reverse(iOrder)$
28: return $(iOrder, ppCols)$

We use the database model for inferring a deletion order. The deletion order is a list of lists of tables. The inner lists show deletion safe tables to be handled in the same run.

As when inserting, it is possible to specify a commit interval. If the commit interval is set to ∞ we may defer deferrable constraints. In this way, we may break some of the cycles in the database model.

In the following, we assume that the database schema can be represented as a DAG. When inferring a deletion order, actions have an impact on the deletion order.

In Figure 9(a), no actions are defined. We can use the order $((A), (B, C), (D, E, F, G))$. In Figure 9(b), if the action is a cascading delete action, we may delete A and in the same run delete C since the action solves constraint violations. An order is therefore $((A, C), (B, F, G), (D, E))$. If the action is a set null action we cannot proceed to C in the same run since deletion in C may update tuples in A. This can have an impact on equality of the tuples in A with regards to delete.

Assume that the database schema contains cascade cycles. We may delete data from such a cycle if all incoming links also have cascading delete actions. In such a situation we may still perform the delete operation.

[15] provides an algorithm for inferring a deletion order in schemas without cycles or where the only cycles present are cascade cycles. As argued earlier, other delete actions invalidates the delete operation

because of side effects that influence if a row should be deleted. Consider cycle in Figure 9(c) where a schema with a cycle with a set null action is shown. We can break such a cycle if we delete from A and then proceed on the remaining graph (B, C, D).

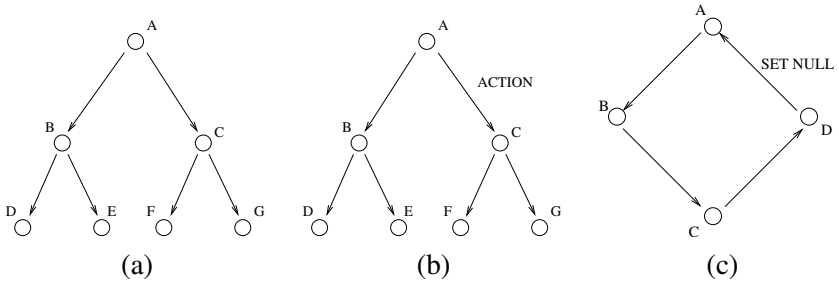


Figure 9: Schemas (a) DAG, no action (b) DAG, action (c) Cycle, action.

However, the side effect on the tuples of D and the definition of delete may cause that we cannot delete tuples in D. If we change the delete operation to only consider equality of the primary keys, the cycle in Figure 9(c) may be broken. The alternative solution can handle schemas with non-overlapping cycles (cascade cycles or at least one set null or set default action) but updates to the database are not retained which may be surprising to the user.

6 Performance study

An implementation with approximately 15,000 lines of Java is done. The implementation is open source and is available from www.cs.aau.dk/∼chr/relaxml/ and www.relaxml.com. Performance tests have been carried out on a 2.6 GHz Pentium 4 with 1GB RAM, running SuSE Linux 9.1, PostgreSQL 8.0, and Java 1.4.2 SE. Every measurement is performed 5 times. The highest and lowest values are discarded and an average is computed using the middle three (all results can be found in Appendix A). The used test data and the test suites can be downloaded from the same places as the implementation.

The data is placed in a table with five integer columns and one varchar column: (ID, ParentID, GroupID, DLLevel, Random, Fixed). The values of the rows are as described in Table 1.

Table 1: Description of data in performance study.

Name	**Range**	**Description**
ID	$0, \ldots, r-1$	Primary key for the table.
ParentID	$0, \ldots, r-1$, null	Foreign key to ID. For each row, the value is ID $-$ 1 if ID mod $5 \neq 0$ Otherwise null.
GroupID	$0, \ldots, \lceil \frac{r}{5} \rceil - 1$	For each row, the value is $\lceil (\text{ID}+1)/5 \rceil - 1$
DLLevel	$0, \ldots, 4$	For each row, the value is ID mod 5
Random	$0, \ldots, 9$	Each row holds a random value
Fixed	c	For each row, the value is c

For $r = 10$ this could result in the data in Table 2.

Table 2: Example data.

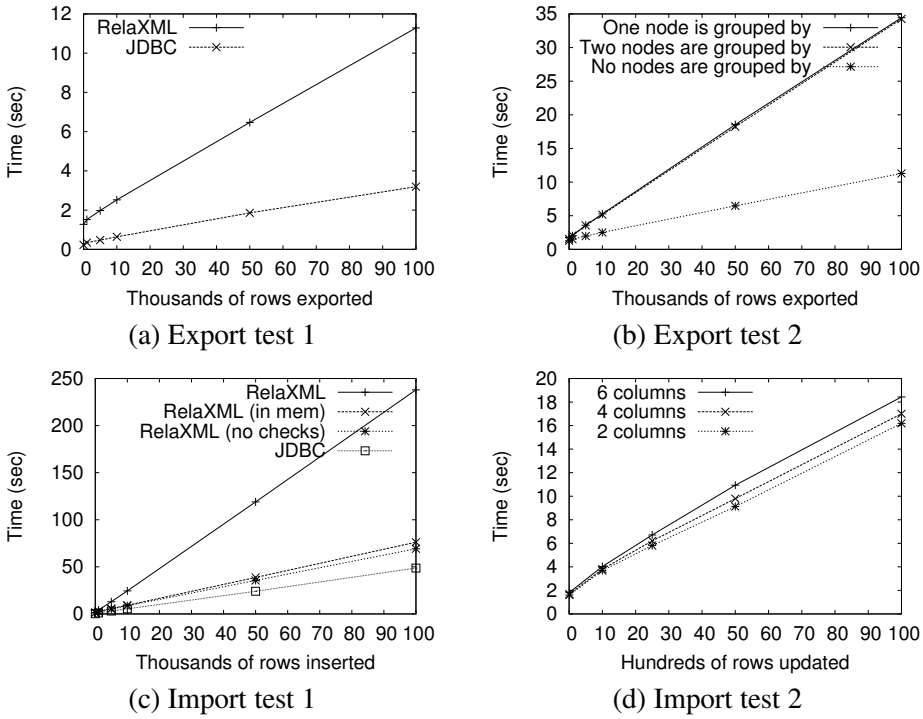
ID	ParentID	GroupID	DLLevel	Random	Fixed
0	null	0	0	3	c
1	0	0	1	8	c
2	1	0	2	2	c
3	2	0	3	7	c
4	3	0	4	1	c
5	null	1	0	4	c
6	5	1	1	0	c
7	6	1	2	5	c
8	7	1	3	9	c
9	8	1	4	6	c

Export test 1 - Scalability in the number of rows This test exports all six columns. The data is exported to an XML structure as the following (but without unecessary spaces) where no grouping is used.

```
<Data>
   <GroupID value="[GroupID]">
      <Fixed value="[Fixed]">
         <ID>[ID]</ID>
         <ParentID>[ParentID]</ParentID>
         <DLLevel>[DLLevel]</DLLevel>
         <Random>[Random]</Random>
      </Fixed>
   </GroupID>
   <GroupID ...>
      ...
   </GroupID>
   ...
</Data>
```

Figure 10(a) compares the running time of RELAXML with that of a specialized JDBC application that executes the SQL query corresponding to the used RELAXML concept. Both write the result set to an XML file, the structure of which has been hard-coded into the JDBC application. The results show that both RELAXML and the JDBC application scale linearly in the number of rows to export. From the slopes, it is seen that RELAXML handles on average 10.4 rows each millisecond (ms) whereas the JDBC application handles 37.5 rows each ms, i.e., the RELAXML overhead is 260%. This is a reasonable overhead given the flexibility and labor-savings of using RELAXML, especially taking into account that the XML documents used in web services are usually not very large.

Export test 2 - Scalability when grouping Here, the same data as in Export test 1 is exported, but now grouping is used. The data is grouped by one (GroupID) and two (GroupID and Fixed) nodes. The running time for no grouping, is the same for RELAXML in Export test 1. The results, in Figure 10(b), show that RELAXML also scales linearly in the number of rows when grouping. The performance suffers when grouping is used, as one row takes approximately 3.3 times longer to export. This is as expected, since the use of grouping requires all the rows to be inserted into a temporary table in the database before they are sorted and then retrieved by the XML writer. The performance is the same when we are grouping by one and two nodes even though there is more sorting to do when grouping by two nodes. However, more



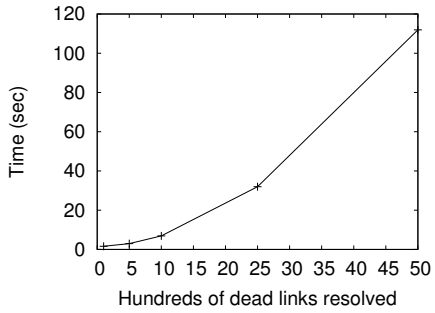
(a) Export test 1 (b) Export test 2

(c) Import test 1 (d) Import test 2

Figure 10: Performance tests.

data (30%) has to be written when we group by one node, as more tags are written since fewer elements are coalesced.

Export test 3 - Scalability in the number of dead links This test selects the rows where DLLevel = 4. Here, each selected row leads to four dead links which are resolved by RELAXML. The results are shown below.



The running time of RELAXML does not scale linearly in the number of dead links resolved. This is expected since each time Algorithm 1 is invoked there will be more rows to search for dead links (leading to an approximately quadratic complexity). Further, the query gets more complicated to process as more OR clauses are added. Note that typical data sets will not contain so many dead links.

Import test 1 - Scalability in the number rows to insert We now compare the time used by RELAXML for inserting with the time used for parsing the XML file with a SAX parser and inserting the data through JDBC prepared statements, checking that this will not lead to a primary key violation. Further, we consider the time used by RELAXML when the inconsistency checks are done in main memory or disabled.

The data to insert originates from Export test 1. The table is emptied before the test is executed. The times used for inserting different numbers of rows are shown in Figure 10(c). The results show that both RELAXML and the JDBC application scale linearly. The average time to import a row using RELAXML is 2.38 ms. When checks for inconsistencies are performed entirely in main memory, RELAXML handles a row in 0.75 ms. If RELAXML does not check for inconsistencies, it handles a row in 0.67 ms, compared to 0.49 ms for using JDBC directly, i.e., the overhead from using RELAXML is only 37%.

Import test 2 - Scalability in the number rows to update We now focus on the scalability in the number of updates. We consider the impacts of updates to one column in rows from the table, varying the number of updated rows. When the XML document is processed, all the included rows have been updated. Only the column Fixed is updated, but the test has been performed with 2, 4, and 6 columns in the used concept. The results, in Figure 10(d), show that the running times are growing linearly in the number of rows after the data sets reach a certain size. The checks for inconsistencies are performed in main memory. More time is used when more columns are included, since more data has to be read from the XML document and more comparisons have to be performed. When 10,000 rows with 6 columns are included, it takes 1.8 ms to read a row and update it in the database. When 4 and 2 columns are included, it takes 1.7 ms and 1.6 ms, respectively.

In summary, we find that the overhead of RELAXML is very reasonable considering the flexibility, simplicity, and labor-savings of RELAXML compared to hard-coded applications. Further, to the best of our knowledge this is the first paper to present a performance study of a general framework for bidirectional transfer of data between relations and XML documents.

7 Conclusion

Motivated by the increasing exchange of relational data through XML based technologies such as web services, this paper investigated automatic and effective bidirectional transfer between relational and XML data.

As the foundation, we proposed the notion of *concepts*, which are view-like mechanisms, for specifying the subset of data to export from a database to XML documents. Concepts supports multiple inheritance and are therefore flexible to use. In addition, this allows specializations to be specified in an incremental fashion. The separation of concept from *structure definition* allows multiple XML representations of the same data. Further, the user-defined transformations allow changes to data that can be difficult to implement in SQL, e.g., ensure that parts of an export XML document are not altered. The specification of import and export ensured that data set are self-contained so that data sets can be imported into an empty database without violating integrity constraints. Performance studies showed a reasonable overhead when exporting and importing compared to the equivalent hand-coded programs. This overhead is easily offset by the offered flexibility, simplicity, and labor-savings of RELAXML compared to hand-coded programs, e.g., in web-service applications.

There are a number of interesting directions of future research. Currently only inheritance between concepts is allowed. It would be interesting to allow aggregation such that the data from one concept can be included as a single element in another concept. It could also be investigated how to extend the approach to support XML documents with more freely defined structures, e.g, mixed content and irregular nesting structures. Another topic is the dead-link detection algorithm that possibly can be tuned by using SQL IN or BETWEEN statements instead of ORing as it is done now. Finally, it would be interesting to investigate whether the overhead compared to hand-coded applications can be avoided altogether by using the concept specification to auto-generate concept-specific Java code which is then just-in-time compiled before the execution.

Acknowledgements

We would like to thank the Oticon Fonden for supporting Steffen Ulsø Knudsen and Christian Thomsen financially. We would also like to thank Logimatic A/S and Lyngsoe Systems A/S for comments and example data. This work was in part supported by the European Internet Accessibility Observatory (EIAO) project, funded by the European Commission under Contract no. 004526.

References

[1] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. *From XML view updates to relational view updates: old solutions to a new problem*. VLDB, pp. 276–287, 2004.

[2] P. Bohannon, J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Simeon. *LegoDB: Customizing Relational Storage for XML Documents*. VLDB, pp. 1091–1094, 2002.

[3] R. Bourret. *XML Database Products: Middleware*. www.rpbourret.com/xml/ProdsMiddleware.htm, as of February 21, 2005.

[4] R. Bourret. *XML-DBMS*. www.rpbourret.com/xmldbms/index.htm, as of February 21, 2005.

[5] R. Bourret. *XML Database Products*. www.rpbourret.com/xml/XMLDatabaseProds.htm, as of February 21, 2005.

[6] R. Bourret., C. Bornövd, A. Buchman. *A Generic Load/Extract Utility for Data Transfer between XML Documents and Relational Databases*. Workshop on Advanced Issues of E-Commerce and Webbased Information Systems, pp. 134–143, 2000.

[7] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. *XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents*. VLDB, pp. 646–648, 2000.

[8] A. B. Chaudhri, A. Rashid, and R. Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems*. ISBN 0201844524. Addison-Wesley, 2003.

[9] C.J. Date. *An Introduction to Database Systems*. 7. edition, ISBN 0-201-68419-5, Addison-Wesley, 2000.

[10] U. Dayal and P. A. Bernstein. *On the Correct Translation of Update Operations on Relational Views*. ACM TODS 8(2), pp 381–418, 1982.

[11] M. .F. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. Tan. *SilkRoute: A Framework for Publishing Relational Data in XML*. ACM TODS 27(4), pp. 438–493, 2002.

[12] Intelligent Systems Research. *JDBC2XML: Merging JDBC Data into XML Documents*. www.intsysr.com/jdbc2xml.htm, as of February 21, 2005.

[13] International Organization for Standardization/International Electrotechnical Commission *XML-Related Specifications (SQL/XML)*. INCITS/ISO/IEC 9075-14:2003, 2003.

[14] Netbryx Technologies. *DataDesk v. 1.0*. www.netbryx.com/DataDesk.aspx, as of February 21, 2005.

[15] S. U. Knudsen and C. Thomsen. RELAXML *A Tool for Transferring Data Between Relational Databases and XML Files*. Master thesis, Aalborg University, Denmark.

[16] G. Reese. *Database Programming with JDBC and Java*. ISBN 1565926161, O'Reilly, 2000.

[17] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. *Querying XML Views of Relational Data*. VLDB, pp. 261–270, 2001.

[18] J. Shanmugasundaram, H. Gang, K. Tufte, D. Zhang, D.J. DeWitt, and J.F. Naughton. *Relational Databases for Querying XML Documents: Limitations and Opportunities*. VLDB, pp. 302–314, 1999.

[19] J. Shanmugasundaram, E. Shekita, K. Tufte, R. Barr, M. Carey, B.R.B. Lindsay, and H. Pirahesh. *Efficiently Publishing Relational Databases as XML Documents*. VLDB, pp. 65–76, 2000.

A Results from performance study

All shown results are in milliseconds.

Table 3: Export test 1 – RELAXML

# of rows	**Average**	Run 1	Run 2	Run 3	Run 4	Run 5
1	1275	1276	1272	1276	1270	1279
1000	1518	1522	1512	1523	1518	1515
5000	1979	1981	1975	1980	1977	1979
10000	2523	2519	2545	2527	2519	2524
50000	6470	6470	6473	6473	6463	6466
100000	11287	11306	11295	11288	11277	11262

Table 4: Export test 1 – JDBC

# of rows	**Average**	Run 1	Run 2	Run 3	Run 4	Run 5
1	229	228	229	230	229	240
1000	348	350	352	347	347	346
5000	483	481	483	481	484	484
10000	658	659	658	660	658	657
50000	1963	1962	1968	1965	1962	1957
100000	3417	3419	3429	3415	3415	3417

Table 5: Export test 2 – Group by 1 node

# of rows	**Average**	Run 1	Run 2	Run 3	Run 4	Run 5
1	1787	1953	1883	1765	1713	1413
1000	2019	2013	2023	2015	2018	2042
5000	3575	3613	3633	3532	3538	3575
10000	5270	5261	5217	5278	5271	5374
50000	18544	18670	18612	18756	18189	18349
100000	34434	34378	34577	35347	34346	34230

Table 6: Export test 2 – Group by 2 nodes

# of rows	**Average**	Run 1	Run 2	Run 3	Run 4	Run 5
1	1541	1567	1554	1503	1380	1587
1000	2080	2125	2026	2071	2044	2580
5000	3555	3554	3559	3552	3565	3551
10000	5163	5100	5129	5155	5300	5205
50000	18233	18170	18320	18516	18210	18059
100000	34253	34397	34069	33905	34292	34423

Table 7: Export test 3

# of d.l.s	**Average**	Run 1	Run 2	Run 3	Run 4	Run 5
100	1610	1605	1608	1610	1619	1612
500	2941	2938	2939	2923	2947	3143
1000	6907	6849	6874	6800	7004	6998
2500	31983	32039	31963	31967	32020	31824
5000	111921	111852	111774	111948	111964	111986

Table 8: Import test 1 – RELAXML

# of rows	**Average**	Run 1	Run 2	Run 3	Run 4	Run 5
1	1290	1275	1291	1290	1290	1299
1000	4056	3983	4082	4048	4072	4047
5000	12993	12858	12940	13057	13119	12981
10000	24451	24342	24294	24179	25305	24717
50000	118959	118813	120841	119726	118339	116743
100000	237974	236953	238734	235851	238236	240419

Table 9: Import test 1 – RELAXML, Touched table in memory

# of rows	**Average**	Run 1	Run 2	Run 3	Run 4	Run 5
1	1377	1363	1373	1376	1383	1381
1000	2547	2543	2549	2548	2543	2549
5000	5620	5604	5536	5636	5621	5986
10000	9214	9228	9185	9266	9229	9166
50000	38693	38855	39038	38470	38753	37994
100000	76180	76961	74500	76826	76531	75184

Table 10: Import test 1 – RELAXML, No checks for inconsistencies

# of rows	**Average**	Run 1	Run 2	Run 3	Run 4	Run 5
1	1412	1408	1417	1412	1410	1413
1000	2524	2509	2517	2535	2526	2528
5000	5336	5222	5291	5438	5363	5355
10000	8683	8646	8681	8721	9119	8572
50000	35581	35399	35889	35173	35962	35456
100000	69252	69004	69167	70377	69585	68427

Table 11: Import test 1 – JDBC

# of rows	**Average**	Run 1	Run 2	Run 3	Run 4	Run 5
1	180	180	180	185	179	180
1000	1100	1120	1099	1100	1099	1100
5000	3051	3046	3011	3038	3070	3570
10000	5409	5422	5444	5383	5423	5382
50000	24093	24324	23839	23619	24115	25327
100000	48719	48467	48805	48754	48597	49379

Table 12: Import test 2 – 2 columns

# of rows	**Average**	Run 1	Run 2	Run 3	Run 4	Run 5
10	1608	1480	1887	1554	1497	1773
1000	3683	3681	3689	3698	3679	3665
2500	5808	5777	5790	5774	5857	6018
5000	9120	8912	9386	8995	8979	9562
10000	16190	15868	16222	15926	16422	16647

Table 13: Import test 2 – 4 columns

# of rows	**Average**	Run 1	Run 2	Run 3	Run 4	Run 5
10	1685	1690	1674	1673	1690	1690
1000	3803	3881	3806	3788	3790	3813
2500	6206	6167	6268	6468	6183	6009
5000	9799	9547	9922	9764	10991	9712
10000	17005	17110	17751	17018	16888	16884

Table 14: Import test 2 – 6 columns

# of rows	**Average**	Run 1	Run 2	Run 3	Run 4	Run 5
10	1685	1690	1674	1673	1690	1690
1000	3803	3881	3806	3788	3790	3813
2500	6206	6167	6268	6468	6183	6009
5000	9799	9547	9922	9764	10991	9712
10000	17005	17110	17751	17018	16888	16884