SQL und XML

- Publizieren von XML
  - XML-Sicht über relationalen Daten,
  - XML-Anfragen über der XML-Sicht.
- Speicherung von XML
  - Wahl eines relationalen Schemas,
  - Übersetzen von XML-Anfragen nach SQL.

(A) SQL/XML (SQL:2003 Standard)

- Funktionen zum Publizieren, XML Datentyp, Abbildungsregeln.

Beispiel:

```
<students>
 <student id="s1">
  <name>aaa</name>
  <city>Wien</city>
  <projects>
   <project id="p1">
    <name>SSD1</name>
   </project>
  </projects>
 </student>
 <student id="s2">
  <name>bbb</name>
  <city>Wien</city>
  <projects>
   <project id="p1">
    <name>SSD1</name>
   </project>
   <project id="p2">
    <name>SSD2</name>
   </project>
  </projects>
 </student>
</students>
```

```
SELECT
 xmlelement(name student,
  xmlattribute(s.StudId as id),
  xmlforest (s.Name as name, s.City as city),
  xmlelement (name projects,
   (SELECT xmlagg(xmlelement(name project,
             xmlattributes(p.ProjectId as id),
             xmlforest (p.Name as name)))
    FROM Projects p
    WHERE p.StudId = s.StudId)))
          as "student-projects"
FROM Students s
```

Funktionen zum Publizieren:

- xmlelement(): Creates an XML element, allowing the name to be specified.
- xmlattributes(): Creates XML attributes from columns, using the name of each column as the name of the corresponding attribute.
- xmlroot(): Creates the root node of an XML document.
- xmlcomment(): Creates an XML comment.
- xmlpi(): Creates an XML processing instruction.
- xmlparse(): Parses a string as XML and returns the resulting XML structure.
- xmlforest(): Creates XML elements from columns, using the name of each column as the name of the corresponding element.
- xmlconcat(): Combines a list of individual XML values to create a single value containing an XML forest.
- xmlagg(): Combines a collection of rows, each containing a single XML value, to create a single value containing an XML forest.

Bemerkung: Funktion xmlgen zum Erzeugen eines Wertes vom Typ XML mittels XQuery ist zwar in den Drafts, jedoch nicht im verabschiedeten Standard beschrieben - also aufgeschoben.

Abbildungsregeln:

- SQL character sets to Unicode,
- SQL identifiers to XML names,
- SQL data type to XML Schema data type,
- values of SQL data types to values of XML Schema data types,
- SQL table to an XML document and XML Schema document,
- SQL schema to an XML document and XML Schema document,
- SQL catalog to an XML document and XML Schema document,
- Unicode to SQL character sets,
- XML names to SQL identifiers.

SQL Tabelle zu XML element oder forest:

StudId	StudName	City
s1	aaa	Wien
s2	bbb	Wien

```
<students>
 <row>
  <StudId>s1</StudId>
  <StudName>aaa</StudName>
  <City>Wien</City>
 </row>
 <row>
  <StudId>s2</StudId>
  <StudName>bbb</StudName>
  <City>Wien</City>
 </row>
</students>
```

```
<students>
  <StudId>s1</StudId>
  <StudName>aaa</StudName>
  <City>Wien</City>
</students>
<students>
  <StudId>s2</StudId>
  <StudName>bbb</StudName>
  <City>Wien</City>
</students>
```

SQL data type to XML Schema datatype:

```
<xsd:simpleType name="INTEGER">
 <xsd:restriction base="xsd:int"/>
</xsd:simpleType>
```

```
<xsd:simpleType name="CHAR_50">
 <xsd:restriction base="xsd:string">
  <xsd:length value="50"/>
 </xsd:restriction>
</xsd:simpleType>
```

Nullwerte?

```
<students>
 <row>
  <StudId>s1</StudId>
  <StudName>aaa</StudName>
  <City xsi:nil="true"/>
 </row>
</students>
```

```
<students>
  <StudId>s1</StudId>
  <StudName>aaa</StudName>
</students>
```

Die Abbildung der Typen ist nicht trivial (Zitat SQL:2003 Standard):

In general, each SQL predefined type or domain SQLT is mapped to the XML Schema built-in data type XMLT that is closest analog to SQLT. Since the value space of XMLT is frequently richer than the set of values that can be represented by SQLT, facets are used to restrict XMLT in order to capture the restrictions on SQLT as much as possible. In addition, many of the distinctions in the SQL type system (for example CHARACTER VARYING vs. CHARACTER LARGE OBJECT) have no corresponding distinction in the XML Schema type system. In order to represent the distinctions, XML Schema annotations are defined. The content of the annotations is defined by this standard; however, whether such annotations are actually generated is implementation-defined. Elements from the XML namespace identified by the XML namespace prefix ”sqlxml” are used to populate these annotations.

```
<xsd:simpleType name="CHAR_50">
 <xsd:annotation>
  <xsd:appinfo>
   <sqlxml:sqltype kind="PREDEFINED" name="CHAR" length="50"
                   characterSetName="LATIN1"
                   collation="DEUTSCH"/>
  </xsd:appinfo>
 </xsd:annotation>
 <xsd:restriction base="xsd:string">
  <xsd:length value="50"/>
 </xsd:restriction>
</xsd:simpleType>
```

(B) SilkRoute

M.Fernandez et al., *SilkRoute: A Framework for Publishing Relational Data in XML*, ACM ToDS, Vol. 27, No.4, 2002.

- Die relationalen Schemata werden als *kanonische* XML-Sicht präsentiert.
- Eine *public* XML-Sicht wird mittels einer (nicht-rekursiven) *public* Query in XQuery spezifiziert.
- *Anwendungsanfragen* werden in XQuery über der public XML-Sicht definiert.
- Die Anwendungsanfragen werden nach SQL übersetzt und relational ausgewertet.
  - Die XQuery-Anfragen werden in einen *View-Forest* übersetzt, damit XML-Ausgabe und XML-Berechnung getrennt sind.
  - Jede Zerlegung eines View-Forest in zusammenhängende Teile entspricht einem Ausführungsplan.
  - Die Auswahl eines Plans findet in Abhängigkeit von Kosten-Parametern des relationalen Datenbanksystems statt.
  - Die Auswertung wird in SQL vorgenommen.
- Die Ergebnisse werden nach XML transformiert.

Beispiel: Supplier-Reseller-Szenario

Relationsschemata eines Textil-Fabrikanten:

```
CREATE TABLE Clothing (
 pid CHAR(10) PRIMARY KEY,
 item VARCHAR(30),
 category VARSCHAR(20),
 description VARCHAR(200),
 price REAL,
 cost REAL )

CREATE TABLE Discount (
 pid CHAR(10) PRIMARY KEY,
 item VARCHAR(30),
 discount REAL )

CREATE TABLE Problems (
 pid CHAR(10),
 code CHAR(10),
 comments VARCHAR(200) )
```

Kanonische XML-Sichten (XQuery Typen):

```
elem CanonicalView {
 elem Clothing,
 elem Discount,
 elem Problems
}
elem Clothing {
 elem Tuple {
  elem pid {integer},
  elem item {string},
  elem category {string},
  elem description {string},
  elem price {float},
  elem cost {float}
 }*
}
elem Discount {
 elem Tuple {
  elem pid {integer},
  elem item {string},
  elem discount {float}
 }*
}

...
```

Beispiel: Public Query ”element fusion”

```
<view> {
 let $items := for $c in $CanonicalView/Clothing/Tuple
               return <product id={ data($c/pid) }>
                      { $c/price } </product>,
     $discounted-items := $d in $CanonicalView/Discount/Tuple
               return <product id={ data($d/pid) }>
                      { $d/discount }</product>,
     $allitems := $items union $discounted-items,
     $prodids := distinct-value($allitems/product/@id
 for pid in $prodids
 return
  <fused-product id="{ $pid }">
   { for $p in $allitems/product
     where $pid = data($p/@id)
     return $p/*
   }
  </fused-product>
} </view>
```

Beispiel: Public Query ”product information outerwear” Q_P

```
<supplier>
 <company>Acme Clothing</company>
 { for $c in $CanonicalView/Clothing/Tuple
   where data($c/category) = "outerwear"
   return
    <product>
     <name>{data($c/item)}</name>
     <category>...
     <description>...
     <retail>{ data($c/price) }</retail>
     { for $d in $CanonicalView/Discount/Tuple
       where $d/pid = $c/pid
       return
        <sale>{ data($c/price) * data($d/discount) }</sale>
     }
     { for $p in $CanonicalView/Problems/Tuple
       where $p/pid = $c/pid
       return
        <report code="{ data($p/code) }">
         {$p/comments }
        </report>
     }
     </product>
 }
</supplier>
```

Beispiel: Application Query ”alles zum halben Preis” Q_A

```
for $s in $PublicView/supplier
 return
 <supplier> {
  <name>{ data($PublicView/supplier/company) }</name>
  <discounted>
  { $p in $s/product
    where data($p/sale) < 0.5 * data($p/retail)
    return <product>{ data($p/name) }</product>
  }
  </discounted>
 } </supplier>
```

Beispiel: XQuery-Darstellung der ausgeführten Query: $Q_P \& Q_A$

```
<supplier>
 <name>Acme Clothing</name>
 <discounted>
 { for $c in $CanonicalView/Clothing/Tuple,
       $d in $CanonicalView/Discount/Tuple
   where data($c/category) = "outerwear",
         data($c/pid) = data($d/pid),
         data($c/price) = data($d/discount) < 0.5 * data($c/price)
   return
    <product>{ data($c/item) }</product>
 </discounted>
</supplier>
```

SQL Anfrage

```
SELECT c.pid as pid, c.item as item
FROM   Clothing c, Discount d
WHERE  c.category = "outerwear",
       c.pid = d.pid,
       c.price * d.discount < 0.5 * c.price
ORDER BY c.pid
```

XML Template

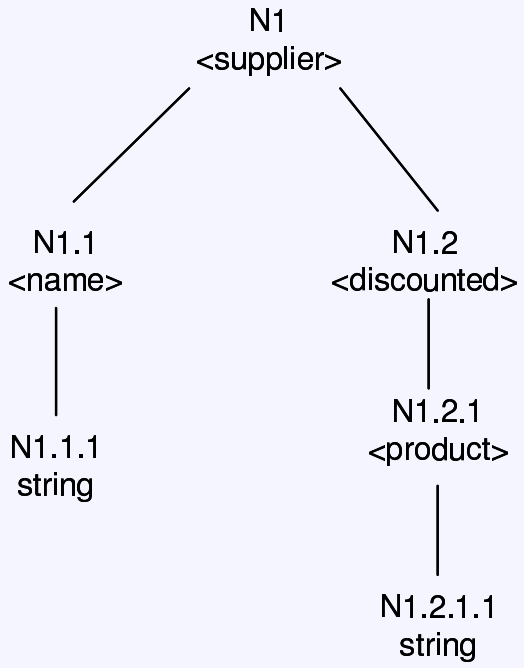
```
<supplier>
 <name>Acme Clothing</name>
 <discounted>
  <product> {$item} </product>
 </discounted>
</supplier>
```

Grundlage der Transformationen: View-Forest

- Abbildung, die einer relationalen Datenbank-Instanz, beschrieben in Form einer kanonischen Sicht, ein XML-Dokument zuordnet.
- Die Struktur des XML-Dokumentes ist getrennt von der Werteberechnung.
- Berechnung der Werte mittels SQL.
- Sowohl die Public Sicht als auch die Anwendungssichten werden intern als View Forest dargestellt.

Ein Forest ist ein Graph, in dem zwischen je zwei Knoten höchstens ein Weg existiert.

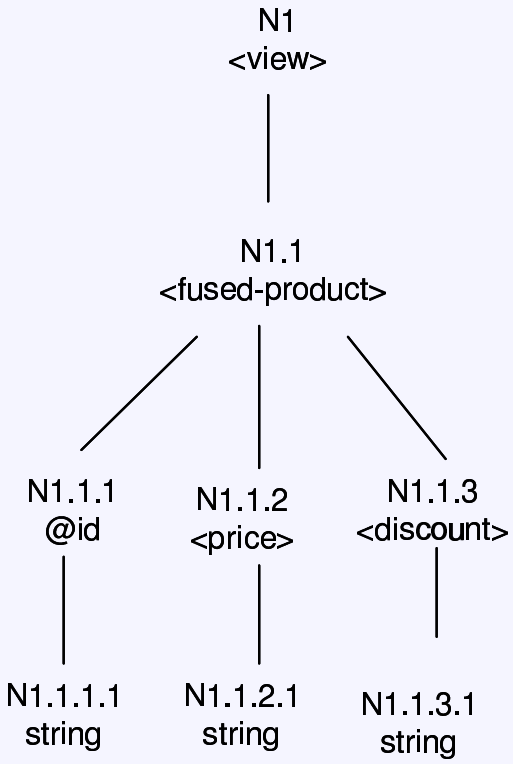
Sei S ein relationales Schema. Ein View-Forest V ist ein Forest, in dem jeder Knoten mit einem XML-Label und einem SQL-Fragment beschriftet ist.




```
N1.1.1 :- SELECT "ACME Clothing"

N1.2.1 :- FROM Clothing c, Discount d
          WHERE c.category = "outerwear" AND
          c.pid = d.pid AND
          c.price = d.discount < 0.5 * c.price

N1.2.1.1 :- SELECT c.item
```




```
N1.1 :- FROM (SELECT DISTINCT c.pid AS PID
              FROM Clothing c) UNION
             (SELECT DISTINCT d.pid AS PID
              FROM Discount d) pid

N1.1.1.1 :- SELECT pid.PID

N1.1.2 :- FROM (SELECT * FROM Clothing c) p
          WHERE p.pid = pid.PID

N1.1.2.1 :- SELECT p.price

N1.1.3 :- FROM (SELECT * FROM Discount d) d
          WHERE d.pid = pid.PID

N1.1.3.1 :- SELECT d.discount
```

- Innere Knoten: Das XML-Label ist ein Element- oder Attribut-Bezeichner. Das SQL-Fragment besteht aus einer FROM- und einer optionalen WHERE-Klausel
- Blatt-Knoten: Das XML-Label ist ein atomarer Typ und das SQL-Fragment besteht aus einer SELECT-Klausel, die gerade einen Wert des entsprechenden Typs definiert und einer optionalen FROM- und Wert-Klausel.
  - Für jede Relation innerhalb der FROM-Klausel wird eine Tupel-Variable festgelegt.
  - Jede Tupel-Variable eines Fragments wird in diesem, oder in dem Fragment eines Vorgängerknotens gebunden.
- Jedem Knoten n wird ein SQL-Ausdruck C_n wie folgt zugeordnet:
  - Die FROM-Klausel ergibt sich als Konkatenation der Relationen in den FROM-Klauseln von n und allen Vorgängern von n.
  - Die WHERE-Klausel ist eine Konjunktion der Bedingungen der WHERE-Klauseln von N und allen Vorgängern von n.
  - Für Blatt-Knoten ergibt sich die SELECT-Klausel aus der N zugeordneten Klausel; anderenfalls ist sie SELECT *.

Effiziente Evaluierung eines View-Forest

Strategien:

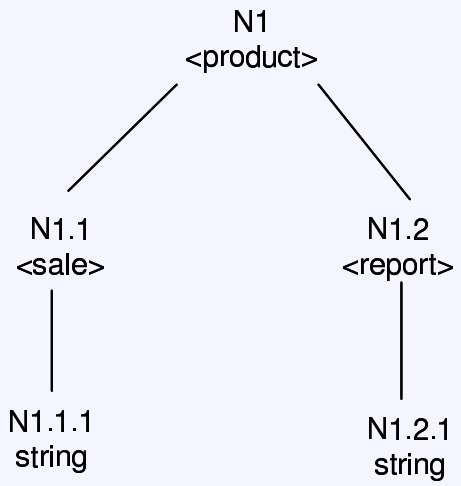
- *fully partitioned*: jedem Knoten des Forest wird ein SQL-Ausdruck zugeordnet; alle diese Ausdrücke werden von der relationalen Datenbank separat ausgewertet.
- *unified*: ein SQL-Ausdruck für den gesamten Forest.
- *view forest decomposition*: zusammenhängenden Teilen des Forest, den *Partitionen*, werden jeweils ein SQL-Ausdruck zugeordnet. Das Finden einer (möglichst) optimalen *Dekomposition* ist Aufgabe eines *Planers*; es existieren im Allgemeinen exponentiell viele unterschiedliche Pläne.

Verfahren:

- Der SQL-Ausdruck einer Partition wird nach den folgenden Regeln gebildet:
  - Eine Kante zwischen Eltern- und Kind-Knoten bedingt einen *left-outer*-join zwischen dem Eltern-SQL-Ausdruck und dem Kind-SQL-Ausdruck.
  - SQL-Ausdrücke von Geschwister-Knoten werden mittels union zusammengeführt.
- Die Ergebnisse der einzelnen SQL-Ausdrucke werden durch den *XML Generator* zu dem gewünschten XML-Dokument zusammengeführt.

Beispiel:

```
for $c in $CanonicalView/Clothing/Tuple
 return
  <product>
  { for $d in $CanonicalView/Discount/Tuple
    where $d/pid = $c/pid
    return
     <sale>{ data($c/price) * data($d/discount) }</sale> }
  { for $p in $CanonicalView/Problems/Tuple
    where $p/pid = $c/pid
    return
     <report code="{ data($p/code) }"> {$p/comments } </report> }
 </product>
```

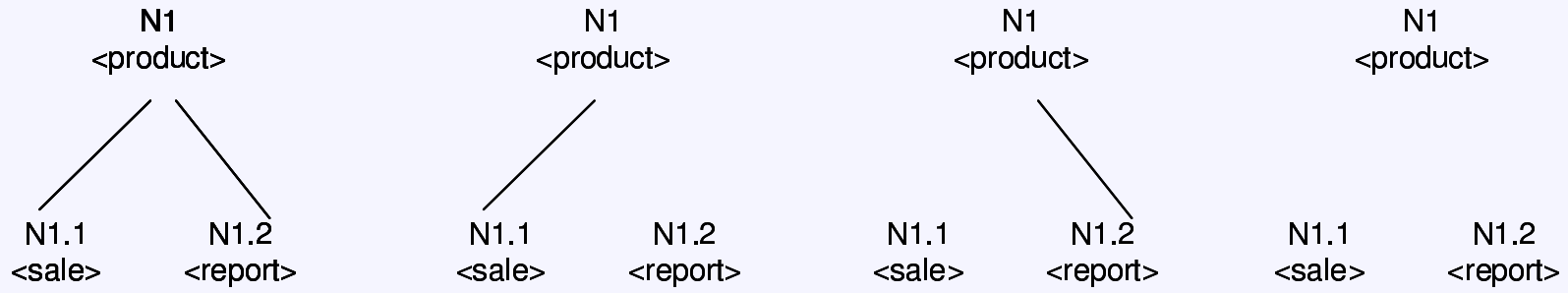



```
N1     :- FROM Clothing c
N1.1   :- FROM Discount d WHERE d.pid = c.pid
N1.1.1 :- SELECT d.discount * c.price
N1.2   :- FROM Problems p WHERE p.pid = c.pid
N1.2.1 :- SELECT p.comments
```

SQL-Anfrage für den gesamten Forest.

```
SELECT 1 AS L1, c.pid, L2, (c.price * Q.discount) as sale, Q.code, Q.comments
FROM Clothing c
LEFT OUTER JOIN
((SELECT 1 AS L2, d.pid AS pid, d.discount AS discount, null AS code, null AS comments
  FROM Discount d)
UNION
 (SELECT 2 AS L2, d.pid AS pid, null AS discount, p.code AS code, p.comments AS comments
  FROM Problems p)
) AS Q
ON c.pid = Q.pid
ORDER BY L1, c.pid, L2, Q.code
```

alle Partitionen




```
SELECT 1 AS L1, c.pid, (d.discount * c.price)
FROM Clothing c
LEFT OUTER JOIN
(SELECT d.pid, d.discount
 FROM Discount d)
ON c.pid = d.pid
ORDER BY c.pid
```

```
SELECT 2 AS L1, c.pid, p.code, p.comments
FROM Clothing c, Problems p
WHERE c.pid = p.pid
ORDER BY c.pid, p.code
```