

# Efficient XSLT Processing in Relational Database System

Salman Said

Seminar Datenbanken und Information Retrieval  
30.03.2007

Martin-Luther-Universität-Halle-Wittenberg  
Institut für Informatik  
Betreuer: Dipl.Inf. Goldberg, Dr. rer. nat. Hinneburg

# Überblick

- Ein Paar Begriffe
- Einführung
- Motivation
- Umschreiben von XSLT in XQuery
- XSLT Partielle Auswertung
- Experimentelle Abschätzung
- Referenzen

# Begriffe

- **XSL** (Extensible Stylesheet Language) ist eine Familie von Sprachen zur Erzeugung von Layouts (*stylesheets*) für XML-Dokumente. Dazu gehören:
  - 1) das XML-basierte eigentliche XSL (zur Unterscheidung genannt **XSL-FO**) (XSL Formatting Objects) für die Beschreibung eines Dokuments als Baum mit Formatierungsanweisungen und Stilangaben,
  - 2) das XML-basierte **XSLT** (XSL Transformationen) für die Transformation eines beliebigen XML-Dokuments in einen anderen Baum
  - 3) und indirekt auch **XPath** für die Adressierung von Baumbestandteilen.

# Begriffe

- **XSL-FO** (Extensible Stylesheet Language – Formatting Objects) ist eine XML-Anwendung, die beschreibt, wie Text, Bilder, Linien und andere grafische Elemente auf einer Seite angeordnet werden. Mit Hilfe von XSL-FO ist es möglich, qualitativ hochwertige Druckerzeugnisse entweder auf Papier oder auf dem Bildschirm zu erzeugen
- **XPath** (XML Path Language) ist eine vom W3C-Konsortium entwickelte Anfragesprache, um Teile eines XML-Dokumentes zu adressieren, das dabei als Baum betrachtet wird. XPath dient als Grundlage einer Reihe weiterer Standards wie XSLT, XPointer und XQuery.

# Begriffe

- **XQuery** steht für **XML Query Language** und bezeichnet eine vom W3C spezifizierte Abfragesprache für XML-Datenbanken.

XQuery benutzt eine an XSLT, SQL und C angelehnte Syntax und verwendet XPath sowie XML Schema für sein Datenmodell und seine Funktionsbibliothek.

XQuery ist dazu gedacht, aus großen XML-Datensammlungen einzelne Teile herauszusuchen. Im Gegensatz dazu dient XSLT dazu, komplette XML-Dokumente zu transformieren.

# Einführung

- XML-Typ ist ein Standardtyp in RDBMS geworden.
- Users können eine Tabelle oder eine Spalte vom XML-Typ erstellen, um XML-Dokumente zu speichern.
- XML-Typ-Werte können von rationalen Daten durch SQL/XML standardisierte Funktionen wie `XMLElement()` und `XMLAgg()` erzeugt werden, so dass XML-Typ-Views über RD erstellt werden können.
- XML-Typ kann durch in SQL/XML eingebettete standardisierte Anfragefunktionen des XQuery/XPath wie `XMLQuery()`, `XMLExist()` und von Oracle erweiterte Funktionen wie `extract()`, `existsNode()` und `extractValue()` aufgerufen werden.

# Einführung

- Effiziente Methoden zur Ausführung von XQuery und XPath in RDBMS durch *rewrite*- und *indexing*-Methoden wurden gut studiert und finden schon industrielle Anwendung.
- Seit Oracle 9i, XMLDB unterstützt XMLTransform()-Funktion, welche den Nutzern ermöglicht, XSLT-Transformationen auf XML-Typ-Werte anzuwenden.
- Zur Zeit wird das XSLT, was in XMLTransform() ist, funktional ausgewertet. Der XSLT-Prozessor führt Transformationen auf das Eingabedokument von XML-Typ aus, ohne manche Informationen auszunutzen, z.B. wie das Eingabedokument in der DB gespeichert, erstellt oder ob es mit Indexen versehen ist. Auch die strukturellen Informationen, d.h. das Schema des Eingabedokument, werden nicht ausgenutzt.

# Einführung

- Wenn wir diese Informationen haben, können wir ähnliche XQuery/XPath-*rewrite*- und Ausführungsmethode zum effizienten XSLT und ohne funktionale Auswertung der XMLTransform() verwenden.
- In diesem Vortrag wird eine Methode namens **XSLT rewrite** vorgestellt.
- Man schreibt das XSLT stylesheet in eine sehr effiziente XQuery-Anfrage durch **partially evaluating** XSLT anhand von den strukturellen Informationen eines Eingabe-XML-Dokuments um.
- Dann wird die XQuery/XPath **native rewrite** Methode zur effizienten Ausführung der XQuery/XPath-Anfrage diskutiert.

# Motivation

- Betrachten wir die beiden Tabellen in relationalen DBMS

<b>deptno</b>	<b>Dname</b>	<b>loc</b>
<i>10</i>	<i>ACCOUNTING</i>	<i>NEW YORK</i>
<i>40</i>	<i>OPERATIONS</i>	<i>BOSTON</i>

**Table 1 - Table "dept" content**

<b>empno</b>	<b>ename</b>	<b>Job</b>	<b>Sal</b>	<b>deptno</b>
<i>7782</i>	<i>CLARK</i>	<i>MANAGER</i>	<i>2450</i>	<i>10</i>
<i>7934</i>	<i>MILLER</i>	<i>CLERK</i>	<i>1300</i>	<i>10</i>
<i>7954</i>	<i>SMITH</i>	<i>VP</i>	<i>4900</i>	<i>40</i>

**Table 2 - Table "emp" content**

- Um XML aus den relationalen Tabellen *dept* und *emp* zu erzeugen, definiert man in Oracle eine View *dept-emp*:

# Motivation

```
CREATE VIEW dept_emp
AS
SELECT
  XMLElement("dept",
  XMLElement("dname", dname),
    XMLElement("loc", loc),
  XMLElement("employees",
    (SELECT XMLAgg(XMLElement("emp",
      XMLElement("empno", empno),
        XMLElement("ename", ename),
        XMLElement("sal", sal)))
    FROM emp
    WHERE emp.deptno = dept.deptno))) as dept_content
FROM dept
```

# Motivation

```
<dept>
<dtype>ACCOUNTING</dtype>
<loc>NEW YORK</loc>
<employees>
<emp>
<empno>7782</empno>
<ename>CLARK</ename>
<sal>2450</sal>
</emp>
<emp>
<empno>7934</empno>
<ename>MILLER</ename>
<sal>1300</sal>
</emp>
</employees>
</dept>
```

```
<dept>
<dtype>OPERATIONS</dtype>
<loc>BOSTON</loc>
<employees>
<emp>
<empno>7954</empno>
<ename>SMITH</ename>
<sal>4900</sal>
</emp>
</employees>
</dept>
```

Die dept-emp-View erzeugt diese zwei „rows“.

# Motivation

- Um eine HTML-Datei zu bekommen, die die „highly paid employees“ (Angestellte, deren monatliches Gehalt mehr als 2000 ist) in jedem Departement anzeigt, verwendet man die Oracle SQL/XML erweiterte Funktion XMLTransform(), die ein „stylesheet“ auf die XML-Datei anwendet und als Ergebnis die XSLT-Transformation ausgibt.
- Eine direkt funktionale Auswertung der Anfrage materialisiert zuerst den XML-Inhalt der dept\_xml durch Erstellen einer XMLType-Instanz von den relationalen Daten und dann werden die XSLT Transformationen auf ihn angewendet.
- Diese funktionale Auswertung ist nicht ganz optimal, weil große XML-Eingabedaten müssen zuerst materialisiert werden bevor die eigentliche XSLT Transformation ausgeführt werden kann.

# Motivation

## **SELECT**

```
XMLTransform(dept_emp.dept_content,  
'<?xml version="1.0"?><xsl:stylesheet  
version="1.0"  
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">  
<xsl:template match="dept">  
<H1>HIGHLY PAID DEPT EMPLOYEES</H1>  
<xsl:apply-templates/>  
</xsl:template>  
<xsl:template match="dname">  
<H2>Department name: <xsl:value-of  
select="."/></H2>  
</xsl:template>  
<xsl:template match="loc">  
<H2>Department location: <xsl:value-of  
select="."/></H2>  
</xsl:template>
```

```
<xsl:template match="employees">  
<H2>Employees Table</H2>  
<table border="2">  
<td><b>EmpNo</b></td>  
<td><b>Name</b></td>  
<td><b>Weekly Salary</b></td>  
<xsl:apply-templates select="emp[sal >  
2000]"/>  
</table>  
</xsl:template>  
<xsl:template match = "emp">  
<tr>  
<td><xsl:value-of select="empno"/></td>  
<td><xsl:value-of select="ename"/></td>  
<td><xsl:value-of select="sal"/></td>  
</tr>  
...  
</xsl:stylesheet>')  
FROM dept_emp;
```

# Motivation

```
<H1>HIGHLY PAID DEPT  
EMPLOYEES</H1>
```

```
<H2>Department name:  
ACCOUNTING</H2>
```

```
<H2>Department location: NEW  
YORK</H2>
```

```
<H2>Employees Table</H2>
```

```
<table border="2">
```

```
<td><b>EmpNo</b></td>
```

```
<td><b>Name</b></td>
```

```
<td><b>Weekly Salary</b></td>
```

```
<tr>
```

```
<td>7782</td>
```

```
<td>CLARK</td>
```

```
<td>2450</td>
```

```
</tr>
```

```
</table>
```

```
<H1>HIGHLY PAID DEPT  
EMPLOYEES</H1>
```

```
<H2>Department name:  
OPERATIONS</H2>
```

```
<H2>Department location: BOSTON</H2>
```

```
<H2>Employees Table</H2>
```

```
<table border="2">
```

```
<td><b>EmpNo</b></td>
```

```
<td><b>Name</b></td>
```

```
<td><b>Weekly Salary</b></td>
```

```
<tr>
```

```
<td>7954</td>
```

```
<td>SMITH</td>
```

```
<td>4900</td>
```

```
</tr>
```

```
</table>
```

# Motivation

- Ein optimaler Auswertungsplan soll die Bedingung *emp[sal > 2000]* ausnutzen (diese Eigenschaft liegt schon auf der untersten Ebene der Tabelle *emp!*) und deshalb einen Index verwenden, um die unnötigen Tupel zu filtern, die zum Endergebnis nichts beitragen.
- Außerdem können die „*XSLT stylesheet template bodies*“ alle zusammen zu einer Anfrage eingeführt werden, die die HTML-Datei aus der rationaler Spalte konstruiert.
- Um das zu erreichen, verwenden wir die ***XSLT rewrite*** Methode und wir schreiben das originale *user stylesheet* mit der XSLT Transformation in eine SQL/XML-Anfrage.

# Motivation

```
SELECT XMLConcat(
    XMLElement( "H1", 'HIGHLY PAID DEPT EMPLOYEES'),
    XMLElement( "H2", 'Department name: '
// "SYS_ALIAS_4"."DNAME"),
    XMLElement( "H2", 'Department location: '
// "SYS_ALIAS_4"."LOC"),
    XMLElement( "H2", 'Employees Table'),
    XMLElement( "table", XMLAttributes('2' AS "border"),
    XMLElement( "td",
        XMLElement( "b", 'EmpNo')),
        XMLElement( "td", XMLElement( "b", 'Name')),
        XMLElement( "td", XMLElement( "b", 'Weekly Salary')),
    (SELECT XMLAGG(
        XMLElement( "tr",
            XMLElement( "td", "EMP"."EMPNO"),
            XMLElement( "td", "EMP"."ENAME"),
            XMLElement( "td", "EMP"."SAL")))
    FROM EMP
    WHERE SAL > 2000
        AND DEPTNO=DEPT.DEPTNO)))
FROM DEPT
```

# Motivation

- Man sieht hier, dass die umgeschriebene Anfrage hauptsächlich aus SQL/XML-Funktionen besteht, wie z.B. XMLConcat(), XMLElement(), XMLAgg(). Sie enthält keine XSLT- oder XPath-Operationen. Die umgeschriebene Anfrage ist eine relationale Anfrage an die relationale Tabelle, dadurch kann der standard-relationaler Optimizer einen Index auf der Spalte *sal* der Tabelle *emp* verwenden und somit läuft die Anfrage schneller

# Motivation

```
SELECT XMLQuery( 'declare variable
$var000 := .; (: builtin template :)
(let $var002 := $var000/dept
return (: <xsl:template match="dept"> :)
(<H1>HIGHLY PAID DEPT
EMPLOYEES</H1>,
( let $var003 := $var002/dname
return (: <xsl:template match="dname"> :)
<H2>{fn:concat("Department name: ",
fn:string($var003))}</H2>,
let $var003 := $var002/loc
return (: <xsl:template match="loc"> :)
<H2>{fn:concat("Department location:
",fn:string($var003))}</H2>,
let $var003 := $var002/employees
return (: <xsl:template match="employees">
:))
```

```
( <H2>Employees Table</H2>,
<table border="2">
{ <td><b>EmpNo</b></td>,
<td><b>Name</b></td>,
<td><b>Weekly Salary</b></td>,
( for $var005 in ($var003/emp[sal > 2000])
return (: <xsl:template match="emp"> :)
<tr> <td>{fn:string($var005/empno)}</td>
<td>{fn:string($var005/ename)}</td>
<td>{fn:string($var005/sal)}</td>
</tr> ) }
</table> ) ) )
)' PASSING dept_emp.dept_content
RETURNING CONTENT)
FROM dept_emp
```

# Motivation

- Die letzte Anfrage ist sehr effizient, weil sie keine unnötige Daten aufruft oder berechnet, die zum Endergebnis nichts beitragen, und weil sie einen B-Tree-Index zum Berechnen der Bedingung  $sal > 2000$  verwendet.

# Rewriting XSLT to XQuery

- Der Hauptschlüssel ist also die Umschreibung des *stylesheets* in ein äquivalentes XQuery. XSLT und XQuery teilen dasselbe XPath und viele Funktionen und Operatoren als Grundbaustein.
- Beide haben ähnliche „XML *node constructs*“, „*iterations with sort*“, „*conditional testing*“ und „*variable access*“. So kann man zwischen diesen „*constructs*“ direkt übersetzen.
- Der Hauptunterschied ist aber, dass XSLT „*templates*“ Ergebnisse vom dynamischen „*pattern matching*“ sind, während XQuery-Funktionen explizit aufgerufen werden.
- Generell wird ein XSLT stylesheet von mehreren *templates* gebildet.
- Jede *template* kann in eine von dem User definierte XQuery-Funktion übersetzt werden und jeder XSL-Befehl in dem *template body* kann in einen entsprechenden XQuery-Ausdruck umgewandelt werden.

# Rewriting XSLT to XQuery

- Der herausfordernde Aspekt hier ist, wie man `<apply-templates/>`-Befehle übersetzt, die implizit das „*template pattern matching*“ verlangen.
- Idee: übersetze XSLT `<apply-templates>` Befehle in eine Kombination von Konditionalen XQuery-Ausdrücken, wo die Ausdruck-Bedingungen das „*template pattern matching*“ wörtlich modellieren und die „Ausdruck-bodies“ Funktionsaufrufe enthalten, die die entsprechende von der XSLT „*template*“ übersetzte XQuery-Funktion aufruft.
- Im Wesentlichen wandelte diese Methode das „*pattern matching*“ und die „*template selection*“ in explizite konditionale XQuery-Ausdrücke um, die von dem XQuery- Prozessor ausgeführt werden.

# Rewriting XSLT to XQuery

- Diese **direkt Übersetzungsmethode** führt meistens zu einer uneffizienten Auswertung, weil der XSLT-Prozessor intern „scharfe“ (eng. aggressive) Optimierung zum Auffinden der richtigen *template* (z.B. durch interne Hashtabelle) anwendet, wobei die übersetzte Anfrage eine große Zahl von konditionalen Ausdrücken verwendet, um sequenziell zu testen, welche *template* zu instanziiieren ist.
- Daher müssen wir „aggressive“ Optimierungen anwenden, um effiziente Anfrage zu bekommen.
- Wenn die strukturellen Informationen der Eingabe von XML-Typ fehlen, wäre das Verwenden dieser direkten Übersetzungsmethode richtig.
- Wir können jedoch diese strukturellen Informationen im Rahmen von RDBMS ableiten und sie in der Umschreibung von XSLT in XQuery verwenden.

# Rewriting XSLT to XQuery

- Die ***partielle Auswertungsmethode*** zum Erzeugen spezieller XQuery-Anfrage von XSLT *stylesheet* kann sehr effiziente XSLT-Transformation erzeugen.
- Woher können wir solche Informationen bekommen?
- Von den *meta-data*-Informationen einer DB! Z.B. von:
  1. XML-Schema oder DTD,
  2. relationaler Schema,
  3. der Struktur des XQuery/XPath, falls der XML-Typ davon stammte,
  4. falls der XML-Typ ein XSMIL war, dann schreibe ins XQuery um → 3 oder
  5. falls der XML-Typ von View-Spalte ist, dann verfolge diese Spalte

# Rewriting XSLT to XQuery

- Wo verwenden wir diese Informationen?
  1. Template instantiation inlining,
  2. *children template instantiation with leveraging children model group and cardinality information,*
  3. *removing unnecessary backward XPath testing* und
  4. *removing unused templates.* Wir erstellen keine entsprechende XQuery-*templates* für *templates*, die nicht instanziiert werden.

# Rewriting XSLT to XQuery

- **Template instantiation inline**
- Die aktivierten templates werden entweder explizit durch `<call-template>` oder implizit durch `<apply-template>` aufgerufen.

```
<xsl:template match="dept">
  ...
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="employees">
  ...
  <xsl:apply-templates select="emp[sal > 2000]"/>
  ...
</xsl:template>
```

- Wir ersetzen den ersten `<apply-template>` mit den template bodies, die durch `"dept"`, `"loc"` und `"employees"` aktiviert sind.

# Rewriting XSLT to XQuery

- Auf den zweiten <apply-template>, der die aktivierte template für das Element „employees“ ist, wenden wir rekursiv den gleichen Algorithmus aber mit dem Element „emp“ an.
- ***Children template instantiation with leveraging children model group and cardinality information,***
- Das XML-Schema bestimmt die Kindermodellgruppen. Sie können eine von den folgenden sein: „*sequence*“, „*choice*“ oder „*all*“.
- Wenn wir wissen, dass es drei Kinderelemente „*dname*“, „*loc*“, und „*employees*“ unter „*dept*“, aber wir wissen nicht in welcher Reihenfolge sie auftauchen (also Gruppe „*all*“), dann schreiben wir das XSLT ins XQuery wie in Figur 12 um (nächste Folie).

# Rewriting XSLT to XQuery

```
declare variable $var000 := .;  
(  
  let $var002 := $var000/dept  
  return  
    for $var003 in $var002/node()  
    return  
      (  
        if ($var003 instance of element(dname)) then  
          inlined xquery from template for "dname",  
        else if ($var003 instance of element(loc)) then  
          inlined xquery from template for "loc" ,  
        else if ($var003 instance of element(employee)) then  
          inlined xquery from template for "employee"  
      )  
)
```

# Rewriting XSLT to XQuery

- Bei „*choice*“ also nur ein Kindelement, dann entfernen wir „*for each*“

```
declare variable $var000 := .;  
(  
  let $var002 := $var000/dept  
  return  
    if ($var002/dname) then  
      inlined xquery from template for "dname"  
    else if ($var002/loc) then  
      inlined xquery from template for "loc"  
    else if ($var002/employee) then  
      inlined xquery from template for "employee"  
)
```

- Bei „*sequence*“ also wenn das „*dept*“-Element die drei Kindelemente hat und wir kennen die Reihenfolge, dann entfernen wir den ganzen Falltest (nächste Folie).

# Rewriting XSLT to XQuery

```
declare variable $var000 := .;  
(  
let $var002 := $var000/dept  
return  
(  
let $var003 := $var002/dname  
inlined xquery from template for "dname",  
let $var003 := $var002/loc then  
inlined xquery from template for "loc"  
let $var003 := $var002/employee then  
inlined xquery from template for "employee"  
)  
)
```

# Rewriting XSLT to XQuery

- *Removing unnecessary backward XPath testing*
- Z.B. bei dem XPath

*<xsl:template match = „emp/empno“>.*

Wenn wir nicht wissen, dass das „empno“-Element nur ein „parent“-Element hat, dann müssen wir es so schreiben:

*(( $\$var$  instance of element(empno)) and fn:exists( $\$var$ /parent::emp) )*

# XSLT Partial Evaluation

- Eine Applikationsberechnung kann als eine Funktion  $F(X,Y)$  beschrieben werden, wo  $X$  sich seltener als  $Y$  verändert, und wo der größte Teil der  $F$ -Berechnungen nur von  $X$  abhängt.
- Wenn wir das XSLT stylesheet als die Funktion  $F$  betrachten und das Eingabe-XML-Dokument in einen „*structural information part (x)*“ und einen „*actual content data (Y)*“ zerlegen, können wir sehen, dass die partielle Auswertung eine perfekte Applikation im Rahmen von XSLT hat.
- Da das pattern matching im typischen XSLT stylesheet hauptsächlich in der Struktur des Eingabe-XML-Dokuments liegt und die eigentliche Prüfung des Dateninhaltes meistens in der XPath-Eigenschaft liegt, können wir das auf partielle Auswertung basierte XSLT auf die strukturelle Informationen des Eingabe-XML vereinfachen.
- Der Ausdruck, welcher von eigentlichen Inhaltsdaten abhängt, wie XPath-Eigenschaft, wird zum restlichen XQuery überlassen und kann effizient durch Indexprobe in der DB bearbeitet werden.

# XSLT Partial Evaluation

- Beim Anwenden von partieller Auswertung erstellen wir zuerst ein spezielles „**sample XML document**“, was alle strukturelle Informationen von dem Eingabe-XML-Typ aber nicht die eigentlichen Inhaltswerte erfasst.
- Dann führen wir das mit speziellen „*trace instructions*“ verbesserte XSLTVM aus, um die „**execution trace information**“ zu bekommen, wie z.B. die Liste der eigentlichen instanziierten *templates* für jede `<apply-template>` Befehl. Wir erstellen auch einen „**template call execution graph**“, der die exakte Folge von *template*-Aktivierungen modelliert.
- Weil wir den eigentlichen Wert des Textknotens nicht kennen, sollen wir annehmen, dass das Ergebnis des matching pattern mit der Eigenschaft z.B. *empno=3456* für ein *emp*-Elementknoten mit dem Kindknoten „*empno*“ immer erfüllt ist.

# XSLT Partial Evaluation

- „**Partial Evaluation**“-Schritt besteht aus zwei Phasen.
  - 1) Zuerst übersetzen wir das stylesheet in einen XSLTVM Bytecode zusammen mit der speziellen „**trace-instructions**“ für das Sammeln der Laufzeit-Informationen. Wir bilden eine „**template-table**“ für jede in dem stylesheet gelistete template. Sie beinhaltet wichtige template Informationen, einschließlich der template formalen Parameter (falls vorhanden).

Wir bilden auch eine „**trace-table**“, wo jeder Tabelleneintrag (table-entry) ein <apply-templates> Befehl abbildet. Der Eintrag hat eine „**trace-call-list**“ mit der laufzeitinstanziierten template und die eigentliche Parameter, zusammen mit den matching XML Knoten, die die template-Aktivierung auslösen.

# XSLT Partial Evaluation

2) Dann wird das XSLTVM aufgerufen, um das *sample XML document* zu übersetzen.

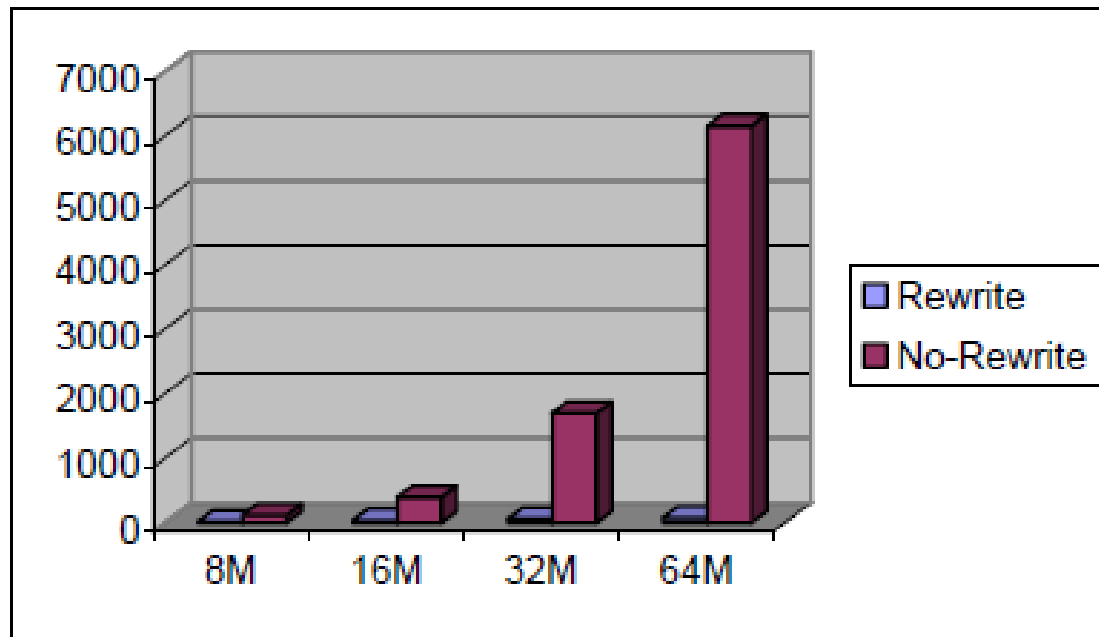
Die *trace-instructions* senden beim Ausführen die eigentlichen Informationen zum *Execution Graph Builder*, der den *template execution graph* erstellt.

Jede *template*-Instanz bildet einen neuen Zustand des Graphen (solange es keine Rekursion gibt). Dieser Zustand entspricht der der aktivierten *template* und der Übergangsweg zu den aktuellen Knoten.

**XQuery Erstellen:** Basiert auf den *template execution graph* und der *trace-call-list* und nutzt die Übersetzungsmethoden, die wir diskutiert haben.

# Experimental Assessment

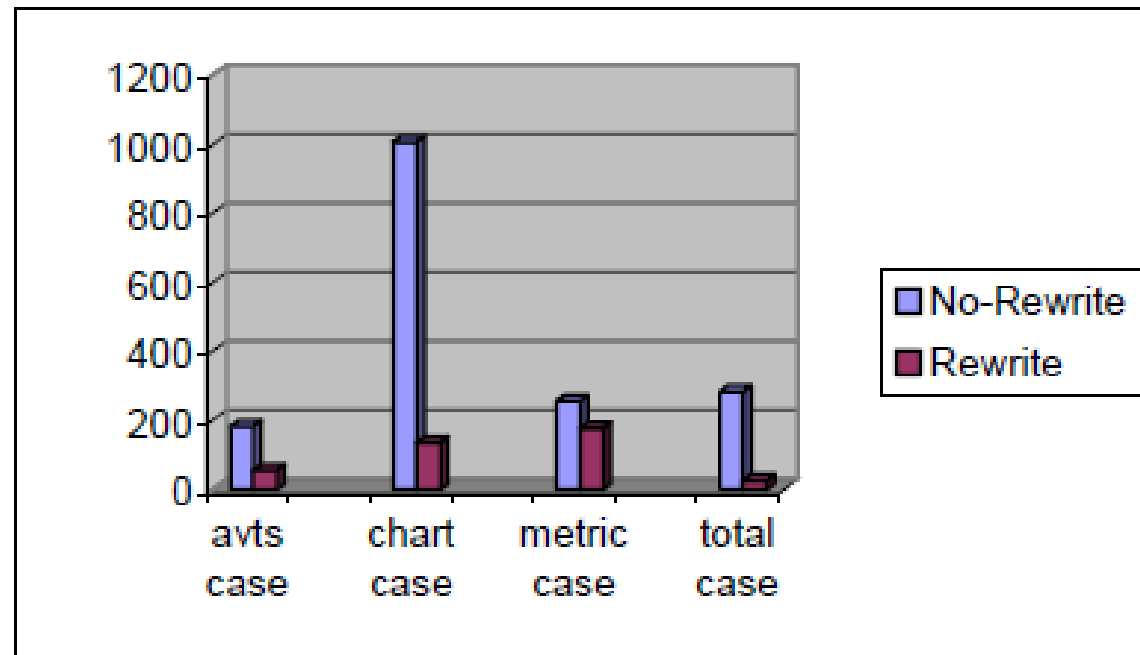
- Figur 2 zeigt die Leistungsstärke im Vergleich zwischen XSLT *rewrite* und XSLT nicht-*rewrite* für „bdonerow“-Testfall. „bbonerow“ XSLT verwendet „XPath value predicate“, um einen bedingten Knoten zu selektieren.



**Figure 2 - Performance comparison between XSLT rewrite versus XSLT no rewrite for few nodes selection**

# Experimental Assessment

- Figur 3 zeigt die Leistungsstärke im Vergleich zwischen XSLT rewrite und XSLT nicht rewrite für die „avts“, „chart“, etc Testfälle, wo es keine „XPath value predicate“ gibt, so dass kein Wertindex verwendet wird, um die Knoten zu filtern.



**Figure 3 - XSLT rewrite Vs XSLT no-rewrite Performance Comparison**

# Referenzen

- A Fokoue, K Rose, J. Simeon, L. Villard, "Compiling XSLT 2.0 into XQuery 1.0", Proceedings of the 14th international conference on Word Wide Web Publishing, May 2005.
- G. Moerkotte, "Incorporating XSL Processing Into Database Engines", VLDB 2002
- A Novoselsky, "The Oracle XSLT Virtual Machine", XTech 2005, Amsterdam, Netherlands.  
<http://www.idealliance.org/proceedings/xtech05/papers/04-02-01/>