Oracle und XML

Rene Schneider
Hochschule der Medien
Nobelstr. 10, 70569 Stuttgart
e-mail: rs034@hdm-stuttgart.de

Abstract

Die Bedeutung von XML-basierten Datenformaten im Enterprise-Bereich ist im Laufe des letzten Jahrzehnts stark gestiegen. Oracle bietet mit Oracle XML DB eine in Oracle-Datenbanken integrierte Unterstützung für sowohl strukturierte als auch unstrukturierte XML-Daten. Durch XPath-Ausdrücke ist dabei der wahlfreie Zugriff in gespeicherte XML-Dokumente möglich, was in Verbindung mit gewöhnlichem SQL vielfältige Zugriffsmöglichkeiten auf XML-Daten bietet. Dieses Paper erklärt grundlegende Techniken im Umgang mit den XML-Funktionalitäten in Oracle-Datenbanken und wirft einen Blick auf die Performance-Implikationen, die beim praktischen Einsatz der Technologie zu beachten sind.

Inhaltsverzeichnis

1 Einleitung und Motivation **2**

2 Oracle-XML-Grundlagen **2**

2.1 Der Datentyp XMLTYPE . . . 2
2.2 Speichern und Abfragen von XML-Daten . . . 3
2.2.1 Der einfache Weg: XML-Text direkt speichern . . . 3
2.2.2 Erzeugen von XML-Dokumenten direkt in der Datenbank . . . 3
2.3 XPath-Ausdrücke in Abfragen . . . 5
2.4 XQuery-Abfragen auf XML-Daten . . . 5
2.5 Speicherungsformen . . . 6
2.5.1 Textbasiert . . . 6
2.5.2 Objektrelational . . . 6
2.5.3 Binär . . . 7
2.6 Indizierung von XML-Daten . . . 8

3 Performance-Messungen **8**
3.1 Insert/Update-Performance 9
3.2 Select-Performance 10
3.2.1 Select-Performance auf ganze Dokumente 10
3.2.2 Select-Performance mit XPath-Abfrage 10
3.3 Bedeutung des Query Rewrite 11
3.4 Speicherbedarf 13
4 Fazit und Bewertung der Erkenntnisse **13**

1 Einleitung und Motivation

XML und XML-basierte Datenformate haben im letzten Jahrzehnt eine beispiellose Verbreitung erfahren. Die einfache Lesbarkeit von XML-Dokumenten, die Verfügbarkeit performanter Parser, Validatoren und Softwarebibliotheken, aber auch die Nähe von XML zu HTML, dem Seitenbeschreibungsstandard im World Wide Web, haben dazu beigetragen, dass XML heute die Basis für viele Austausch- und Speicherformate darstellt, insbesondere beim Datenaustausch in heterogenen Systemen.

Durch die Omnipräsenz von XML-basierten Formaten ist der Wunsch nur allzu verständlich, XML-Daten in relationalen Datenbanksystemen speichern und verarbeiten zu können. Während eine Speicherung in jedem beliebigen Datenbanksystem mit Unterstützung zur Speicherung größerer Mengen von Textdaten kein Problem ist - letztlich sind XML-Daten auch „nur“ Textdaten - ist eine Verarbeitung der gespeicherten XML-Dokumente in Queries oder Stored Procedures nur möglich, wenn das Datenbanksystem XML grundsätzlich enquoteversteht und im Optimalfall eine Schemadefinition für ein konkretes XML-basiertes Format zur Verfügung hat.

Genau für diesen Zweck gibt es im Oracle-Datenbankserver den speziellen Spaltentyp „XMLTYPE“. In Spalten dieses Typs gespeicherte XML-Daten kann die Oracle-Datenbank auf vielfältige Weise verarbeiten. Jedoch gibt es eine Reihe von Besonderheiten bei der XML-Nutzung in Oracle-Datenbanken zu beachten, die sich bedeutend auf die Performance auswirken und deren Verständnis zur erfolgreichen Nutzung der XML-Features in einem Softwareprojekt essenziell ist. Mit diesen Besonderheiten befasst sich das vorliegende Paper.

2 Oracle-XML-Grundlagen

Zunächst soll die Art und Weise, wie Oracle-Datenbanken XML-Daten in XMLTYPE-Spalten speichern, erläutert werden. Die Kenntnis dieser Grundlagen ist Voraussetzung für das Verständnis der Performance-Aspekte, die in Kapitel 3 behandelt werden.

2.1 Der Datentyp XMLTYPE

Der Oracle-Spaltentyp XMLTYPE ist der zentrale Spaltentyp beim Umgang mit XML-Daten. Zur Vorbereitung einer Tabelle auf die Speicherung von XML-Dokumenten genügt es im einfachsten Fall, eine Spalte

der Tabelle auf den Datentyp XMLTYPE zu setzen. Oracle erlaubt fortan nur noch die Speicherung valider XML-Dokumente in der betreffenden Spalte und stellt bei Queries bzw. in Stored Procedures die speziellen XML-Kommandos und Abfragefunktionen wie z.B. XPath-Abfragen zur Verfügung. Natürlich wäre es auch möglich, XML-Dokumente in CLOB- oder BLOB-Spalten als einfache Large Objects zu speichern - da Oracle in diesen Fällen aber die Daten als „Black Box“ behandelt und keinerlei Kenntnis über ihre XML-Natur hat, stehen dabei die speziellen XML-Funktionalitäten nicht zur Verfügung. Dasselbe gilt für die ebenfalls denkbare Möglichkeit, (große) XML-Dateien als BFILE - also als extern im Dateisystem abgelegtes File, das Oracle nur über eine Referenz darauf bekannt gemacht wird - zu speichern.

XMLTYPE ist jedoch nicht gleich XMLTYPE - es gibt 3 unterschiedliche interne Speicherformen für XMLTYPE-Spalten, wobei Oracle dem Nutzer die freie Wahl lässt. Die Speicherungsformen unterscheiden sich sowohl in Hinsicht auf Performance als auch unterstützte Features erheblich - eine detaillierte Erklärung aller Formate ist in Kapitel 2.5 zu finden.

2.2 Speichern und Abfragen von XML-Daten

Das Speichern von XML-Dokumenten in XMLTYPE-Spalten stellt sich - unabhängig von der gewählten internen Speicherform - immer gleich dar und unterscheidet sich im einfachsten Fall nicht signifikant von der Speicherung derselben XML-Dokumente in einer CLOB- oder BLOB-Spalte.

Zunächst muss eine Tabelle mit mindestens einer Spalte des Datentyps XMLTYPE erzeugt werden; für Demonstrationszwecke enthält die hier gezeigte Tabelle keine weiteren Spalten, in der Praxis ist es natürlich möglich, beliebig viele weitere Spalten einzusetzen:

```
1 CREATE TABLE xmltest (
2   dokument XMLTYPE
3 )
```

2.2.1 Der einfache Weg: XML-Text direkt speichern

Diese neue Tabelle kann nun auf verschiedenen Wegen mit Daten befüllt werden. Einerseits ist es möglich, XML-Dokumente als normale Textstrings in SQL-Queries zu verwenden:

```
1 INSERT INTO xmltest VALUES ( XMLTYPE(
2 '<products>
3   <product name="Gartentisch" price="2200" count="200"/>
4   <category name="Stühle">
5     <product name="Holzstuhl" price="3000" count="50"/>
6     <product name="Plastikstuhl" price="1000" count="20"/>
7   </category>
8 </products>'
9 ))
```

2.2.2 Erzeugen von XML-Dokumenten direkt in der Datenbank

Oracle unterstützt aber auch eine Reihe von XML-Funktionen, mit deren Hilfe XML-Datenstrukturen direkt in der Datenbank erschaffen werden können:

```
1 INSERT INTO xmltest VALUES (
2   XMLElement(
3     "products",
4     XMLElement(
```

```
5       "product",
6       XMLAttributes(
7         'Gartenfackel' as "name",
8         '1400' as "price",
9         '120' as "count"
10      )
11    )
12  )
13 )
```

In der Datenbank führt obiges INSERT-Kommando zu folgendem XML-Dokument:

```
1 <products>
2   <product name="Gartenfackel" price="1400" count="120"></product>
3 </products>
```

Dieser auf den ersten Blick umständlichere Weg, XML-Daten abzuspeichern, vereinfacht die Erzeugung von XML-Dokumenten aus bestehenden relationalen Tabellen oder innerhalb von PL/SQL-Scripts enorm, was die SQL/XML-Kommandos[1] zu einem wichtigen Kopplungselement zwischen der klassischen relationalen Datenbankwelt und der Welt der XML-Dokumente macht. Sie erzeugen direkt XMLTYPE-Daten, die sowohl in XMLTYPE-Spalten gespeichert als auch als normale XML-Textdaten ausgegeben werden können.

Ein kleines Beispiel: Wir nehmen an, es existiert eine Datenbanktabelle mit Produkten, die in ein XML-Dokument wie das oben gezeigte überführt werden und als XMLTYPE gespeichert werden soll.

```
1 CREATE TABLE produkte (
2   "name" VARCHAR2(100),
3   "price" NUMBER(10,0),
4   "count" NUMBER(4,0)
5 )
```

Nach dem Einfügen von ein paar Beispieldatensätzen...

```
1 INSERT INTO produkte VALUES ('Holzkohlegrill', 200, 10);
2 INSERT INTO produkte VALUES ('Schwenkgrill', 2200, 20);
```

...können die Daten aus der Tabelle „produkte“ mit einem einzigen INSERT-Statement in XML-Dokumente mit der aus 2.2.1 bekannten Struktur überführt und in die Tabelle „xmltest“ eingefügt werden:

```
1 INSERT INTO xmltest SELECT
2   XMLElement(
3     "products",
4     XMLElement(
5       "product",
6       XMLAttributes(
7         "name" as "name",
8         "price" as "price",
9         "count" as "count"
10      )
11    )
12  )
13 FROM produkte;
```

Hier finden die Kommandos zur Erstellung von XML-Dokumenten sinnvoll Anwendung, um die bestehenden Daten aus den Spalten „name“, „price“ und „count“ der „produkte“-Tabelle in Attribute des „product“-Elements zu wandeln und die Struktur des XML-Dokuments zu erstellen.

[1] Oracle-Dokumentation aller Kommandos: http://download-uk.oracle.com/docs/cd/B19306_01/appdev.102/b14259/xdb13gen.htm#sthref1484

Dieselbe Verfahrensweise kann natürlich auch genutzt werden, um per SELECT-Abfrage XML-Datensätze aus Daten in einer relationalen Tabelle zu generieren. Ebenso können die XML-SQL-Kommandos in PL/SQL-Scripts zur komfortablen Erzeugung von validen XML-Dokumenten genutzt werden. Einen weiteren interessanten Anwendungsfall stellen Views dar: eine View kann z.B. genutzt werden, um relational gespeicherte Daten direkt bei Abfragen auf die View in XML-Form zu konvertieren.

2.3 XPath-Ausdrücke in Abfragen

Bei der Abfrage der gespeicherten Daten können nicht nur ganze Dokumente ausgelesen werden. Für den gezielten Zugriff auf Teile der XML-Datensätze unterstützt Oracle die im XML-Bereich übliche Pfadbeschreibungssprache XPath, mit der Komponenten eines XML-Dokuments exakt selektiert werden können.

Ein einfaches Beispiel:

```
1 SELECT extractvalue(xmltest.dokument, '//products/product/@name') "names" FROM xmltest
```

Dieses Kommando liest nur die Namen aller gespeicherten Produkte aus der Datenbank und gibt sie in tabellarischer Form zurück. „extractvalue“ weist Oracle dabei an, die wirklichen Werte aus den per XPath selektierten Subelementen herauszuziehen und diese zurückzugeben. Alternativ können auch via „extract“ XML-Elemente (inklusive eventuell vorhandener Unterelemente) aus Dokumenten extrahiert werden; der Rückgabetyp der Daten ist dann XMLTYPE.

2.4 XQuery-Abfragen auf XML-Daten

Seit Oracle 10g umfassen die XML-Erweiterungen der Oracle-Datenbank einen vollständigen XQuery-Prozessor. Dieser bietet - neben dem Datenzugriff via SQL und den „extract“-Funktionen mit XPath-Ausdrücken - mit der Unterstützung des XQuery-Standards einen weiteren Weg, um Abfragen in XML-Dokumente vorzunehmen, Ergebnisse zu filtern und zu sortieren. Der Vorteil von XQuery ist, dass diese Abfragesprache speziell für XML-Abfragen optimiert ist, während SQL als Abfragesprache der relationalen Datenbankwelt klar auf relationale Daten optimiert wurde und Abfragen in XML-Dokumente hinein beispielsweise nur durch Erweiterungen wie die „extract“-Funktionen erlaubt.

Eine komplette Beschreibung der XQuery-Sprache geht weit über den Rahmen dieses Dokuments hinaus, daher sei an dieser Stelle auf die offizielle XQuery-Spezifikation der W3C verwiesen[2]. Ein kurzes Beispiel soll jedoch zeigen, wie ein typischer XQuery-Ausdruck aussieht, der aus einem Dokument wie der in den bisherigen Beispielen eingeführten Produkte-Sammlung die Namen einzelner Produkte mit bestimmten Eigenschaften (hier: einem Preis größer 2000) herausfiltert:

```
1 for $i in /products
2 let $j = 2000
3 where $i/@price > $j
4 order by $i/@price descending
5 return $i/@name
```

Im Grunde ähneln sich XQuery und SQL stark - es gibt dieselben Grundelemente, die in den meisten Queries in der soeben gezeigten Form miteinander verknüpft werden:

for...in entspricht dem SELECT ... FROM in SQL und bestimmt, durch welchen Datenbestand iteriert werden soll.

[2] http://www.w3.org/TR/xquery/

let entspricht dem SQL SET und erlaubt das Definieren von Variablen und Zuweisen von Werten.

where ist das Äquivalent der SQL-WHERE-Clause und erlaubt das Setzen von Filterkriterien.

order by entspricht ebenfalls direkt dem ORDER BY in SQL und bestimmt die Reihenfolge der Ergebnisdatenmenge.

return entspricht dem SQL RETURN-Statement und erlaubt das Formatieren der Rückgabedaten (in XML oder auf andere Weise).

XQuery ergänzt in Oracle-Datenbanken die Abfragemöglichkeiten durch reines SQL speziell im Hinblick auf XML-Daten und erlaubt gemeinsam mit SQL die Konstruktion so ziemlich jeder noch so komplexen Anfrage.

2.5 Speicherungsformen

Während die Speicherung von XML-Daten in einer Oracle-Datenbank durch die XMLTYPE-Spalte sehr einfach und intuitiv möglich ist, lauern im Detail doch einige Tücken, über die man insbesondere dann Bescheid wissen sollte, wenn man größere Datenmengen in XML-Form ablegen will. Denn XMLTYPE ist nicht gleich XMLTYPE - hinter dieser Tabellenspalte stecken drei unterschiedliche Speicherformen, unter denen der Nutzer beim Anlegen einer Tabelle die freie Wahl hat.

Wichtig ist: Die Nutzung einer Tabelle mit XMLTYPE-Spalte funktioniert immer auf die gleiche Weise, egal welche Speicherungsform zum Einsatz kommt! Die Wahl der Speicherungsform hat jedoch unter Umständen starken Einfluß auf die Performance von Abfrage- und Manipulations-Kommandos, weshalb im Folgenden die drei möglichen Speicherungformen vorgestellt und kritisch gegenübergestellt werden.

2.5.1 Textbasiert

Die textbasierte Speicherung ist die einfachste Speicherungsform und gleichzeitig der Standard, der zur Anwendung kommt, wenn nicht explizit eine Speicherungsform bei Erstellung einer Tabelle gewählt wird. XML-Dokumente werden in diesem Fall direkt in Textform in der Datenbank abgelegt; eine XMLTYPE-Spalte mit textbasierter Speicherung entspricht also weitestgehend einer CLOB-Spalte.

Es liegt auf der Hand, dass ein Zugriff „in“ die textbasiert gespeicherten XML-Dokumente hinein, etwa mit XPath-Queries, ein Parsen jedes einzelnen XML-Dokuments erfordert, was dementsprechend langsam ist und schlecht mit wachsender Zahl an Dokumenten skaliert. Aktionen, die gesamte Dokumente betreffen und nicht von deren Inhalt abhängig sind, wie etwa Abfragen gesamter Dokumente, das Einfügen neuer Dokumente in eine Tabelle sowie das Löschen von Dokumenten hingegen erfolgen in hoher Geschwindigkeit, da in diesem Fall lediglich ein einzelnes CLOB-Objekt zu modifizieren ist.

2.5.2 Objektrelational

Bei der objektrelationalen Speicherungsform führt Oracle eine Umwandlung jedes XML-Dokuments in eine relationale Form durch. Einzelne Attribute landen dabei z.B. in Spalten mit zu den enthaltenen Daten passenden Datentypen; für jede XMLTYPE-Spalte wird also ein Satz von relationalen Tabellen erzeugt, in welchen die Daten abgelegt werden. Das hat bedeutende Folgen.

Zur erfolgreichen objektrelationalen Speicherung ist es zunächst erforderlich, ein XML-Schema bei der Erstellung einer XMLTYPE-Spalte anzugeben, damit die Datenbank die Datentypen der Attribute und die Struktur

der XML-Elemente untereinander verstehen und in eine passende Tabellenstruktur umwandeln kann. Alle einzufügenden XML-Dokumente müssen sich zwingend an dieses Schema halten! In der Praxis führt man die Schemadefinition getrennt von der Erzeugung der Datenbanktabelle für die XML-Daten durch; zur „Anmeldung“ eines Schemas bei Oracle gibt es die Funktion „registerschema“, die wie in Listing 1 genutzt wird, um ein passendes Schema für die hier verwendeten Beispieldaten anzulegen.

Listing 1: Anmeldung des XML-Schemas bei der Datenbank

```
1  dbms_xmlschema.registerschema(
2    schemaurl => 'http://www.firehead.de/xmltest/schema.xsd',
3    schemadoc => '<?xml version="1.0" encoding="UTF-8"?>
4                  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
5                   <xs:element name="products">
6                    <xs:complexType>
7                     <xs:sequence>
8                      <xs:element name="product">
9                        <xs:complexType>
10                         <xs:attribute name="name" type="xs:string"/>
11                         <xs:attribute name="price" type="xs:integer"/>
12                         <xs:attribute name="count" type="xs:integer"/>
13                       </xs:complexType>
14                     </xs:element>
15                    </xs:sequence>
16                   </xs:complexType>
17                  </xs:element>
18                 </xs:schema>'
19 );
```

Das Schema wird wie gewohnt durch eine URL identifiziert, die grundsätzlich frei wählbar ist und über welche das Schema sowohl den später einzufügenden XML-Dokumenten als auch bei der Erstellung einer XMLTYPE-Spalte mit objektrelationaler Speicherungsform referenziert werden kann. Ein Schema muss jeweils nur einmal bekannt gemacht werden, nicht vor jedem Einfügen eines das Schema referenzierenden XML-Dokuments.

Der Lohn für die Mühe der Schema-Nutzung und -Bekanntmachung ist eine drastisch bessere Leistung bei XPath-Abfragen „in“ die Dokumente hinein. Oracle kann bei der objektrelationalen Speicherung von XML-Dokumenten all seine Performance-Optimierungen beim Umgang mit relationalen Daten ausspielen, weil die XML-Dokumente für die Datenbank nichts anderes als normale Tabellen mit Indizes, Schlüsseln, definierten Datentypen etc. sind.

Fairerweise muss man anmerken, dass sich der Aufwand für das Einfügen eines weiteren Dokuments oder das Löschen eines Dokuments gegenüber der Speicherung in einem einzelnen CLOB-Feld wie bei der textbasierten Speicherung erhöht, die Performance bei solchen Aktionen also etwas sinkt: das erforderliche Parsen des ganzen Dokuments benötigt zusätzliche Rechenpower.

Ein weiterer Nachteil der objektrelationalen Speicherung ist der Speicher-Overhead, der beim Umgang vor allem mit großen Mengen an XML-Daten entsteht. Da die XML-Daten in der Datenbank nicht in XML- sondern in relationaler Form gespeichert sind, müssen Dokumente vor dem Auslesen zunächst zur XML-Form serialisiert und beim Einlesen in relationale Form geparsed werden. Speziell bei großen XML-Dokumenten ist der Overhead erheblich, da der eingesetzte Parser nicht streambasiert arbeitet und jedes Dokument zunächst zu einem vollständig im Speicher gehaltenen DOM-Baum einliest. In der späteren relationalen Form ist der Speicherbedarf der Daten auf der Festplatte dann jedoch geringer als im textbasierten Format.

2.5.3 Binär

Die binäre Speicherungsform ist in Oracle 11g neu hinzugekommen und stellt einen Zwitter zwischen textbasierter und objektrelationaler Speicherung dar. Dabei werden XML-Dokumente in ein proprietäres Binärformat konvertiert, welches speziell zur Speicherung von XML-Daten entwickelt wurde.

Aufgrund der proprietären Natur des Formats sind leider wenig Details über seinen internen Aufbau bekannt. Oracle zufolge[1] wird bei der Konvertierung ein einfaches Parsing des Dokuments durchgeführt, bei dem Tags in binäre Tokens konvertiert und Texte sowie numerische Daten wenn möglich in ihre jeweiligen nativen Speicherungsformen umgeformt werden. Auch soll das Binärformat durchgängig in allen Komponenten einer Oracle-Datenbank verwendet werden, so dass eine Konvertierung von Daten in das ursprüngliche textbasierte XML-Format erst dann erforderlich ist, wenn die Daten an eine Anwendung weitergeleitet werden sollen. Dies reduziert den Speicher- und CPU-Overhead.

Ebenfalls ressourcenschonend wirkt sich der Streaming Parser aus, der bei der Konvertierung und Validierung von Daten ins binäre Speicherungsformat zum Einsatz kommt. Vor allem der Nachteil des hohen Speicherbedarfs der objektrelationalen Speicherungsform beim Umgang mit großen XML-Dokumenten wird dadurch abgemildert.

Um das binäre Speicherungsformat nutzen zu können ist keine Angabe eines XML-Schemas erforderlich. Auch müssen die Dokumente in einer Spalte mit dem binären Speicherungsformat nicht alle derselben Struktur folgen, wie das etwa bei der objektrelationalen Speicherungsform durch den Zwang zur Angabe eines Schemas der Fall ist. Das binäre Format ist dennoch laut Oracle „Schema-aware“, d.h. wenn ein Schema angegeben wird weiß das Format dieses zur Optimierung des Umgangs mit passenden XML-Dokumenten zu nutzen.

2.6 Indizierung von XML-Daten

Je nach genutzter Speicherungsform bieten sich unterschiedliche Möglichkeiten, wie XML-Daten für die Beschleunigung von Abfragen indiziert werden können. Im Falle von strukturierten XML-Daten - d.h. Daten, die einer festen, durch ein XML-Schema vorgegebenen Struktur folgen und daher in objektrelationaler Form gespeichert werden können - ist es auf einfachste Weise möglich, Indizes über bestimmte Elemente bzw. Attribute der XML-Dokumente zu erstellen. Listing 2 etwa zeigt den Befehl zum Anlegen eines Index über das Attribut „price“ sämtlicher Produkte.

Listing 2: Erstellung eines Index für relational gespeicherte Daten

```
1 create index idx_or_price on xmltest_or (extractvalue(dokument, '//products/product/@price'
     ))
```

Im Hintergrund erstellt Oracle einen normalen Index über die entsprechende relationale Tabelle, in der die strukturierten XML-Daten gespeichert sind, und nutzt diesen Index bei Abfragen automatisch - genau, wie dies auch bei der Arbeit mit normalen, relationalen Daten geschieht.

Für die Indizierung von XML-Daten in textbasierter und vor allem binärer Speicherungsform bietet Oracle ab 11g eine neue Funktionalität namens XMLIndex. XMLIndex bietet einen generischen Index, der spezifisch auf die Indizierung von unstrukturierten XML-Dokumenten zugeschnitten wurde. Zu indizierende Werte bzw. Elemente werden durch XPath-Ausdrücke identifiziert - ähnlich also wie im in Listing 2 gezeigten Beispiel.

3 Performance-Messungen

Die im Folgenden vorgestellten Performance-Messdaten entstammen einem beispielhaften Messversuch, bei dem per PL/SQL-Funktion eine größere Zahl kurzer, mit Zufallsdaten gefüllter XML-Dokumente wie das in Listing 3 gezeigte erstellt wurden. Diese Dokumente wurden in drei Tabellen mit XMLTYPE-Spalten (je eine für jede Speicherungsform) eingefügt, wobei jede Tabelle eine weitere, aufsteigend befüllte numerische Spalte als Primary Key erhielt. Jede Messung ist dreimal in Folge durchgeführt worden, die Ergebnisse stellen gemittelte Werte dar.

Listing 3: Ein Beispiel-Dokument

```
<products>
  <product name="Gartentisch" price="2200" count="200"/>
  <category name="Stühle">
    <product name="Holzstuhl" price="3000" count="50"/>
    <product name="Plastikstuhl" price="1000" count="20"/>
  </category>
</products>
```

Das zum Test genutzte System hat die folgenden Eckdaten:

- Oracle 11g-Installation auf Windows 7 RC1
- Core 2 Duo, 2.2 GHz
- 4 GByte RAM
- Intel X25-M 80 GByte SSD

Bei der Testausführung ist darauf geachtet worden, dass nebenläufige Tätigkeiten bzw. Programmaktivitäten so weit wie möglich unterbunden waren. Die Queries wurden mit dem Oracle SQL Developer ausgeführt und getimed.

Wie bei allen Benchmarks sind auch die Ergebnisse dieses Tests mit Vorsicht zu genießen: sie treffen auf eine Anwendungssituation wie sie im Test simuliert wurde zu (zahlreiche kleine XML-Dokumente mit einfacher Struktur), sind aber nicht notwendigerweise auf andere Situationen übertragbar! Auch werden zunächst nur die nackten Zahlen präsentiert, Interpretationen und Schlussfolgerungen daraus folgen im Anschluß daran und in Kapitel 4

3.1 Insert/Update-Performance

Im ersten Schritt wurde die Insert-Performance beim Einfügen neuer Dokumente in die drei Tabellen gemessen. Zum Einsatz kam dabei ein PL/SQL-Script, welches die XML-Dokumente mit immer gleicher Struktur mit zufälligen Daten füllte (siehe Listing 4).

Listing 4: Der XML-Dokumentengenerator

```
create or replace procedure generate_random_data_xmlstring(
  p_count in number,
  p_table in varchar2
) is
  v_xmltext varchar2(32767);
begin
  for i in 1..p_count loop
    v_xmltext := '<products>';
    v_xmltext := v_xmltext ||'<product name="Produkt #'||to_char(round(dbms_random.value
         *1000000000))||'" price="'||to_char(round(dbms_random.value*10000))||'" count="'||
         to_char(round(dbms_random.value*2000))||'"/>';
    v_xmltext := v_xmltext ||'</products>';
    execute immediate
       'INSERT INTO '||p_table||' VALUES (:1, xmltype(:2).createschemabasedxml(''http://www
            .firehead.de/xmltest/schema.xsd''))'
    using i, v_xmltext;
  end loop;
end;
```

Um sicherzustellen, dass der Generierungsprozess selbst die Messergebnisse nicht nennenswert verfälscht, wurde das Script einmal ohne Insert der Daten in die Tabelle ausgeführt. Dieser Lauf dauerte auf dem Testsystem nur 500ms, womit garantiert ist, dass die Messergebnisse wirklich die für den Insert benötigte Zeit wiedergeben.

Ausgeführt wurden 100.000 Inserts auf jede der drei Tabellen; die Ergebnisse sind in Abbildung 1 zu sehen.

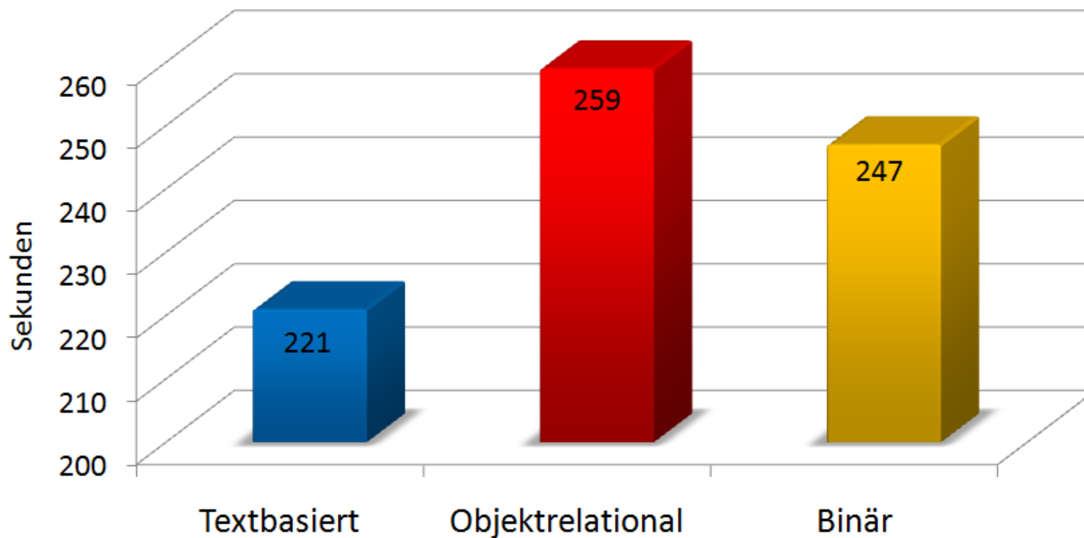

Abbildung 1: 100.000 Inserts, Zeit gemessen in Sekunden

3.2 Select-Performance

3.2.1 Select-Performance auf ganze Dokumente

Bei dieser Messung wurden 1 Million einzelne Dokumente aus den jeweiligen Tabellen abgerufen - es fand dabei keine Abfrage „in die Dokumente hinein“ statt, die Auswahl der Dokumente erfolgte über den numerischen Primary Key. Für die Durchführung des Tests wurde ebenfalls ein PL/SQL-Script genutzt, die Ergebnisse sind in Abbildung 2 zu sehen.

3.2.2 Select-Performance mit XPath-Abfrage

Zur Ermittlung der Performance bei Anfragen, die ein „Hineingreifen“ in das Dokument erforderlich machen, wurde eine einzige Anfrage mit einer XPath-Expression in der Where-Clause gestellt und die dafür benötigte Zeit gemessen. Die genutzte Anfrage ist in Listing 5 einsehbar, die Ergebnisse zeigt Abbildung 3.

Listing 5: Die Select-Anfrage mit XPath-Komponente

```
1 SELECT idx FROM xmltest_txt WHERE extractvalue(xmltest_txt.dokument, '//products/product/
      @price') = 1
```

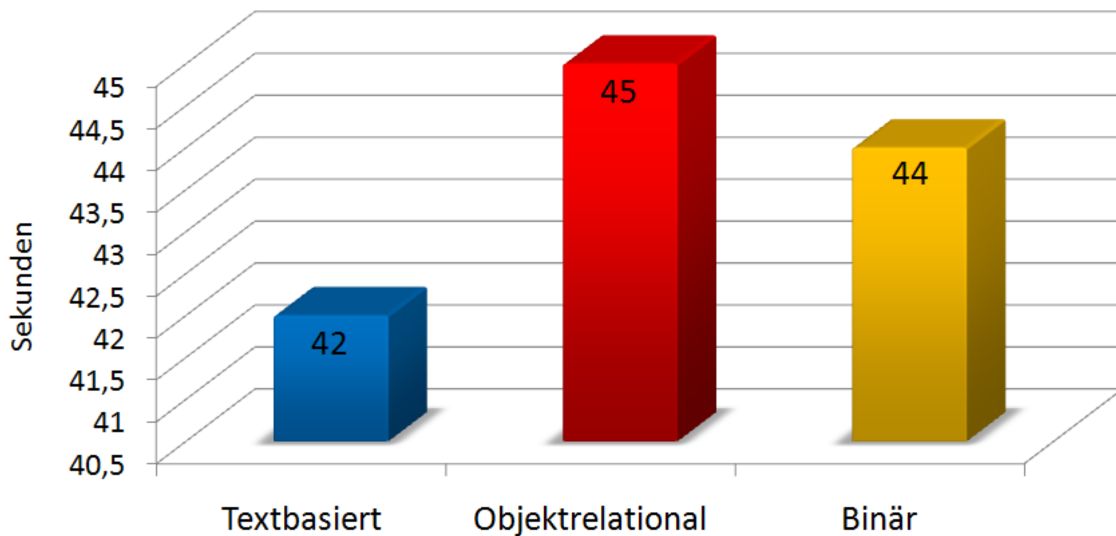

Abbildung 2: 1.000.000 Selects ganzer Dokumente, Zeit gemessen in Sekunden

3.3 Bedeutung des Query Rewrite

Betrachtet man die ermittelten Zahlen, fällt besonders der große Unterschied bei Abfragen mit XPath-Expressions auf: in diesem Anwendungsfall ist die objektrelationale Speicherungsform fast um Faktor 10 schneller als die textbasierte und immerhin um Faktor 1,7 schneller als die binäre Speicherungsform.

Der Grund liegt im sogenannten Query-Rewrite-Prozess, der beim Zugriff auf objektrelational gespeicherte XML-Daten durchgeführt wird. Dabei formt die Oracle-Datenbank eine XPath-Anfrage wie die aus Listing 5 in eine reine SQL-Abfrage um, stets passend zu den relationalen Tabellenstrukturen, welche die Datenbank für die objektrelationale Speicherung der Daten im Hintergrund angelegt hat. Aus einer Anfrage, die bei einer rein textbasierten Speicherung ein vollständiges Parsen jedes Dokuments erfordert, wird so eine rein relationale Anfrage auf normale Tabellenstrukturen, auf welche die Datenbank ihr gesamtes Arsenal an Optimierungsstrategien für relationale Daten anwenden kann. Entsprechend liegt die Performance einer durch Query Rewrite erfolgreich umgeformten Anfrage auch auf relationalem Niveau.

Wichtig ist, zu wissen, dass nicht alle XPath-Anweisungen auch erfolgreich vom Query Rewrite umgeschrieben werden können (eine Liste der unterstützten Expressions kann in der XML-DB-Dokumentation[2] nachgelesen werden). Verwendet man eine nicht unterstützte XPath-Anweisung, so wird die Anfrage zwar korrekt ausgeführt, aber es findet kein Query Rewrite statt: auch im Falle der objektrelationalen Speicherungsform muss dann geparsed werden, die Performancesteigerung bleibt also aus.

Eine Prüfung auffällig langsamer Queries auf korrekte Funktion des Query Rewrite ist daher empfehlenswert[3]. Diese Prüfung kann mittels des Oracle-Kommandos „EXPLAIN PLAN“ durchgeführt werden. Eine entsprechende Anweisung plus die interessanten Teile der Ausgabe für die im Benchmark genutzte Select-Anfrage sähe z.B. folgendermaßen aus:

```
1 EXPLAIN PLAN FOR
2 SELECT idx FROM xmltest_txt WHERE extractvalue(xmltest_txt.dokument, '//products/product/
      @price') = 1;
3
4 SELECT * FROM table(dbms_xplan.display());
5
6 Predicate Information (identified by operation id):
```

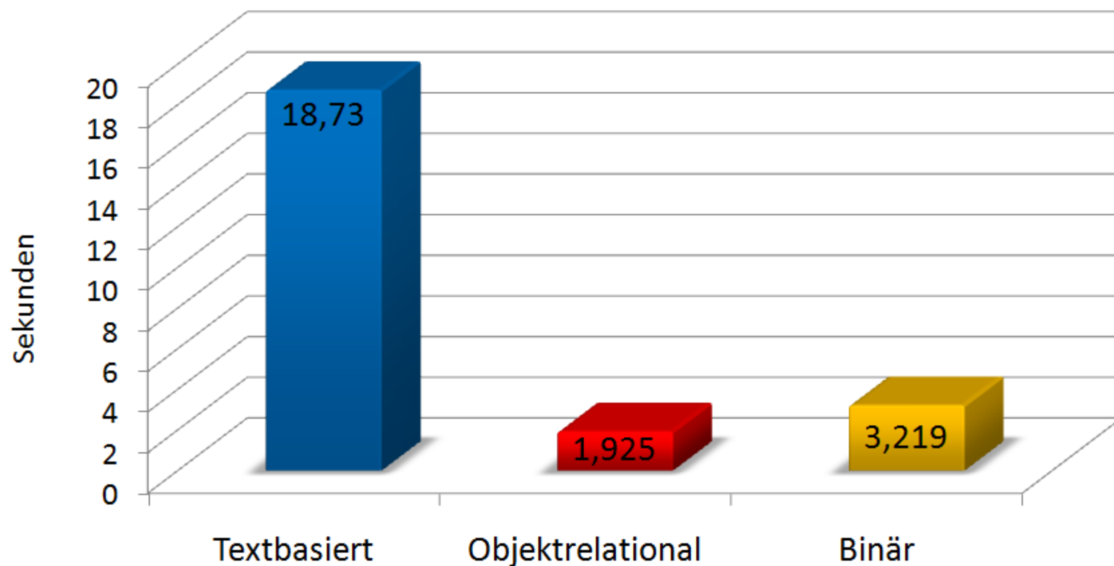

Abbildung 3: Select-Anfrage mit XPath-Komponente

```
7    1 - filter(TO_NUMBER(EXTRACTVALUE(SYS_MAKEXML("XMLTEST_TXT"."SYS\_NC00003\$"),'//
         products/product/@price'))=1)
```

Vor allem interessant ist die „Predicate Information“, wo in diesem Fall ein Filter gesetzt wird, der den XPath-Ausdruck unverändert enthält - ein sicheres Zeichen dafür, dass kein Query Rewrite stattgefunden hat. Das ist auch kein Wunder, denn die Anfrage geht an die Tabelle mit textbasierter XML-Speicherung (Tabellenname „xmltest_txt“). Bei einer Anfrage an die objektrelational gespeicherten Daten sieht das Ergebnis anders aus:

```
1 EXPLAIN PLAN FOR
2 SELECT idx FROM xmltest_or WHERE extractvalue(xmltest_or.dokument, '//products/product/
      @price') = 1;
3
4 SELECT * FROM table(dbms_xplan.display());
5
6 Predicate Information (identified by operation id):
7    1 - filter("XMLTEST\_OR"."SYS_NC00010\$"=1)
```

Auch diesmal wird ein Filter gesetzt, aber der XPath-Ausdruck ist verschwunden. Stattdessen selektiert der Filter einen automatisch erzeugten Spaltennamen „SYS_NC00010“ auf normale, relationale Art und Weise. Exakt diese Umwandlung der ursprünglichen Anfrage ist der Lohn des Query Rewrite und der Grund für die deutlich bessere Performance.

Noch ein weiteres Beispiel, diesmal mit einer etwas anderen XPath-Anfrage:

```
1 EXPLAIN PLAN FOR
2 SELECT idx FROM xmltest_or WHERE existsnode(xmltest_or.dokument, '//products/product/@name[
      contains(text(), "Produkt 1")]') = 1;
3
4 SELECT * FROM table(dbms_xplan.display());
5
6 Predicate Information (identified by operation id):
7    1 - filter(EXISTSNODE(SYS\_MAKEXML('D5AC9070DF98459C857A2AA752ACB0AF',4730,"XMLTEST\_OR"
         ."SYS_NC00003\$","XMLTEST\_OR"."SYS_NC00006\$"),'/child::products/child::product/
         attribute::name[contains(text(), "Produkt 1")]',' ')=1)
```

Der XPath-Ausdruck nutzt hier die nicht vom Query Rewrite unterstützte Funktion „contains“. Die Folge davon ist, dass der gesetzte Filter den XPath-Ausdruck weiterhin enthält, die Anfrage also nicht relational abgearbeitet werden kann, sondern ein Parsen jedes einzelnen Dokuments erforderlich wird. Anfragen dieser Art müssen wenn möglich umgeschrieben werden, so dass sie nur normale SQL-Kommandos sowie unterstützte XPath-Ausdrücke enthalten, um die hohe relationale Abfrageperformance zu erreichen.

3.4 Speicherbedarf

Die drei Speicherungsformen unterscheiden sich auch im Platzverbrauch der Daten auf der Festplatte bzw. dem Storage-System des Datenbankservers. In diesem Benchmark wurde der verbrauchte Speicherplatz der 100.000 zufällig erzeugten XML-Dokumente mit einer SQL-Abfrage wie in Listing 6 ermittelt, das Ergebnis ist in Abbildung 4 zu sehen.

Listing 6: Abfrage des Speicherplatzverbrauchs

```
select sum(BYTES) as TOTAL_BYTES from user_segments where SEGMENT_NAME = 'XMLTEST_BIN';
```

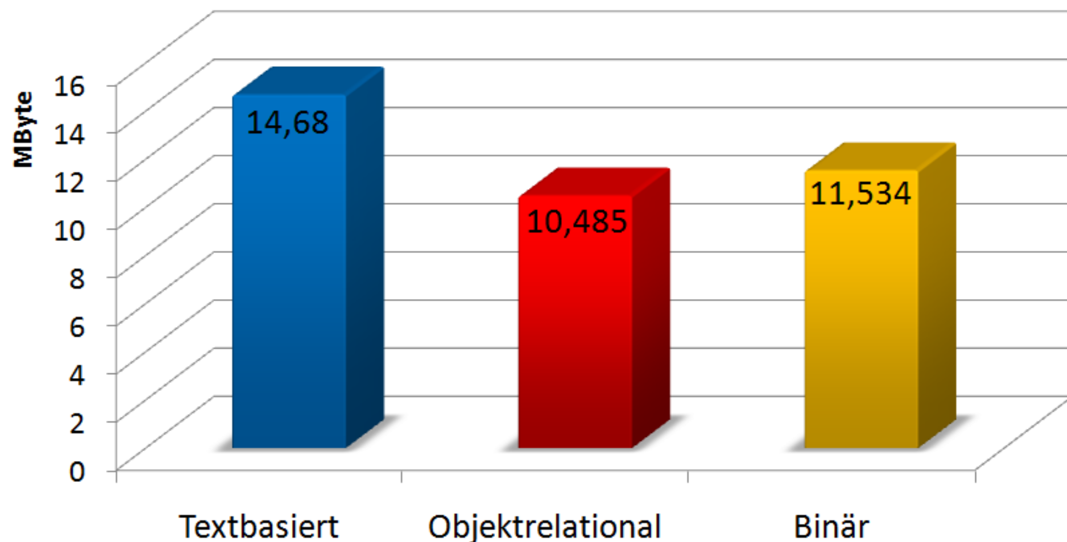

Abbildung 4: Platzverbrauch der Daten

4 Fazit und Bewertung der Erkenntnisse

Schon der im Rahmen dieses Papers durchgeführte kleine Benchmark zeigte teilweise fundamentale Unterschiede zwischen den drei XML-Speicherungsformen auf. Während die textbasierte Speicherungsform etwas schneller ist wenn es um Speichern oder Abfragen kompletter Dokumente geht, hat die objektrelationale Speicherungsform deutliche Vorteile bei Anfragen, die per XPath gezielt Elemente oder Werte aus den Dokumenten herausgreifen - sofern sichergestellt ist, dass der Query-Rewrite-Prozess solche Anfragen erfolgreich zu rein relationalen Anfragen konvertieren kann.

Interessant ist außerdem die binäre Speicherungsform: sie liegt in allen Tests stets zwischen der objektrelationalen und der textbasierten Speicherungsform. Insbesondere im Test mit den größten Unterschieden - der

XPath-Abfrage in die Dokumente hinein - zeigt die binäre Speicherungsform eine um Größenordnungen bessere Performance als die rein textbasierte Speicherung. Die binäre Speicherung kombiniert essentielle Vorteile der relationalen Speicherungsform - schnelle Abfragen in Dokumente, geringer Platzverbrauch der Daten - mit Vorteilen der textbasierten Speicherung. Das binäre Format prädestiniert sich so geradezu für den Einsatz in Situationen, in denen die objektrelationale Speicherung etwa aufgrund des Zwangs zur Angabe eines XML-Schemas pro Tabellenspalte oder aus anderen Gründen nicht genutzt werden kann.

Erwähnenswert ist an dieser Stelle auch, dass das binäre Format laut Oracle[1] einen neuen Streaming Parser verwendet, der nicht mehr den Aufbau eines das gesamte Dokument umfassenden DOM-Baumes im Speicher beim Parsen erfordert. Dies wirkt sich vor allem beim Umgang mit sehr großen Dokumenten aus und reduziert den Speicherbedarf des binären Formats gegenüber der objektrelationalen Speicherungsform in solchen Fällen deutlich, ohne dass man sich dazu auf die Nachteile der rein textbasierten Speicherungsform einlassen muss, die einen ähnlichen Vorteil bietet, so lange man kein Parsing der Dokumente etwa durch XPath-basierte Anfragen erzwingt.

Die wichtigste Erkenntnis ist jedoch: für jeden konkreten Einsatzfall muss bewusst die passende Speicherungsform gewählt werden. Hierzu sollten Tests mit realen Anwendungsdaten und realen Abfragesituationen durchgeführt und alle drei Varianten entsprechend evaluiert werden, bevor die Entscheidung fällt. Mit den drei Varianten ist das Oracle-Datenbanksystem in jedem Fall gut aufgestellt, um für die allermeisten Einsatzzwecke eine passende Lösung bereitstellen zu können.

Literatur

[1] Mark Drake,*New Features in Oracle XML DB for Oracle Database 11g Release 1*, Oracle Corporation, June 2007, http://www.oracle.com/technology/products/database/oracle11g/pdf/xml-db-11g-whitepaper.pdf

[2] Oracle® XML DB Developer's Guide, 11g Release 1 (11.1) Oracle Corporation, http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28369/xdb_rewrite.htm

[3] Carsten Czarski, XML in der Datenbank: Performance, SQL und EXPLAIN PLAN, http://sql-plsql-de.blogspot.com/2007/11/xml-in-der-datenbank-performance-sql.html