XML en Oracle11g RDBMS

Nieuwe features op een rij

Begin september vorig jaar werd Cumquat Information Technology gevraagd om te participeren in het bètatraject voor Oracle11g RDBMS. Inmiddels is de eerste productieversie beschikbaar en in de tussenliggende periode heeft Cumquat een groot aantal nieuwe onderdelen kunnen bekijken en testen. In dit artikel zal Jan Vissers ingaan op een tweetal belangrijke nieuwe features die geboden worden op het gebied van XML in de Oracle database.

Oracle11g RDBMS bouwt met de verbeteringen op het gebied van XML ondersteuning voort op de succesvolle XMLDB component van de database. Zoals dat in de afgelopen versies van de Oracle database meestal het geval was, richten de nieuwe uitbreidingen zich vooral op het vlak van beheer(s)baarheid, performance en schaalbaarheid – en voor XML geldt feitelijk hetzelfde. Uiteraard worden met 11g ook weer nieuwe *applicatieontwikkeling* faciliteiten toegevoegd, maar deze vormen toch duidelijk de minderheid in vergelijking met de 'goodies' die de Oracle DBA krijgt. Is dat erg? Nee, denk ik. Je kunt je zelfs afvragen of met Oracle11g RDBMS het product als zodanig voor de applicatieontwikkelaar niet 'af' is. Hoeveel nieuwe dingen gaat Oracle er nog aan toevoegen – en gaan we die ook nog daadwerkelijk gebruiken of nodig hebben? Maar ik dwaal af. In dit artikel bespreek ik twee van de, naar mijn mening, meest in het oogspringende uitbreidingen/verbeteringen op het gebied van XML ondersteuning in de Oracle database. Ten eerste zal ik *Binary XML* bespreken. Dit is een nieuwe, efficiënte manier om XML op te slaan in de database. Daarna licht ik *XMLIndex* toe. Deze nieuwe *logische* index kan de performance sterk verbeteren van queries op XML opgeslagen in de database.

Binary XML: 'One size fits all'

In de vorige versie van Oracle RDBMS hadden voor de opslag van XML in de database keuze uit twee varianten. *Unstructured storage* – opslag van de XML data "zonder fratsen" in een CLOB en *Structured storage* – opslag van de XML data waarbij de structuur, na XML Schema definitie registratie bij de database, direct werd omgezet naar een verzameling van objecten/tabellen. Het belangrijkste verschil tussen beide komt feitelijk tot uitdrukking in de toepasbaarheid van ieder afzonderlijk opslagmodel; Ongestructureerde opslag van XML vooral in geval van *document-centric* en gestructureerde opslag bij *data-centric* toepassingen. Praktisch gezien wordt toch vaak in beide gevallen gegrepen naar ongestructureerde opslag, omdat het 'zo lekker gemakkelijk' is. De introductie van Binary XML gaat hier, denk ik, verandering in brengen. Gedurende het bètatraject van Oracle11g RDBMS heb ik hier mee kunnen experimenteren en mijn ervaringen zijn – op een paar bugjes na – over het algemeen goed te noemen. Zo goed, dat ik me afvraag of we op termijn niet al onze XML in de database op basis van Binary XML opslag (moeten) doen.

> ***Binary XML lijkt met zijn voordelen de geprefereerde opslagstructuur van XML te kunnen worden***

Wat is Binary XML?

De Oracle documentatie omschrijft Binary XML storage als:

"Binary XML storage – XMLType data is stored in a post-parse, binary format specifically designed for XML data. Binary XML is compact, post-parse, XML Schema-aware XML. This is also referred to as post-parse persistence."

Er zullen ongetwijfeld mensen zijn die deze zin in een keer kunnen lezen en ook snappen, maar ik heb daar wat moeite mee. Wat ik belangrijke dingen vind in dit geheel is dat binaire opslag van XML *compact* dus efficiënt is en dat het XML *Schema-aware*

is zonder daarbij alleen maar XML te kunnen opslaan wat gekoppeld is aan één bepaalde XML Schema definitie. Daarmee verenigt binaire opslag van XML feitelijk de potentiële voordelen van gestructureerde en ongestructureerde opslag.
Alvorens een aantal voorbeelden van het gebruik van binary XML opslag te tonen, is het goed om de belangrijkste voordelen van deze manier van opslag ten opzichte van gestructureerde en ongestructureerde opslag te benoemen.

- Snelheid in de verwerking van afzonderlijke XML data is groot – doordat op basis van *post-parse* het laden van de DOM (Document Object Model) sneller kan worden uitgevoerd.
- Opslagcapaciteit noodzakelijk voor XML is met binaire opslag minder groot – doordat veel van de "ballast" van het oorspronkelijke XML wordt weggelaten en/of verkleind.
- Flexibiliteit in de opslag van XML data is groot – doordat niet alleen data opgeslagen kunnen worden die voldoen aan een bepaald XML Schema definitie, maar ook *Schemaless* XML.
- Flexibiliteit in de opslag van XML data wat voldoet aan XML Schema is groot – doordat in een en dezelfde tabel data kan worden opgeslagen van verschillende XML Schema definities.
- Snelheid van het verwerken van updates op XML data is

```
<xsd:schema                              elementFormDefault="unqualified"
attributeFormDefault="unqualified"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xdb="http://xmlns.oracle.com/xdb">
   <xsd:element name="state" type="stateType"/>
   <xsd:complexType name="stateType">
       <xsd:sequence>
           <xsd:element name="stateImage" type="imageType"
minOccurs="0"/>
           <xsd:element name="abbreviation" minOccurs="0">
               <xsd:simpleType>
                   <xsd:restriction base="xsd:string">
                       <xsd:length value="2"/>
                   </xsd:restriction>
               </xsd:simpleType>
           </xsd:element>
           <xsd:element name="name"    type="xsd:string" minOccurs="0"/>
           <xsd:element name="capital" type="xsd:string" minOccurs="0"/>
           <xsd:element name="nickname"
                                       type="xsd:string" minOccurs="0"/>
           <xsd:element name="population"
                               type="xsd:positiveInteger" minOccurs="0"/>
           <xsd:element name="land-area"
                                     type="xsd:decimal" minOccurs="0"/>
           <xsd:element name="water-area"
                                     type="xsd:decimal" minOccurs="0"/>
           <xsd:element name="entered-union"
                                      type="xsd:string" minOccurs="0"/>
           <xsd:element name="description"
                                  type="descriptionType" minOccurs="0"/>
           <xsd:element name="largest-cities"
                                type="largestCitiesType" minOccurs="0"/>
       </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="descriptionType">
       <xsd:sequence>
           <xsd:element ref="p" minOccurs="0" maxOccurs="unbounded"/>
       </xsd:sequence>
   </xsd:complexType>
   <xsd:element name="p" type="pType"/>
   <xsd:complexType name="pType" mixed="true">
       <xsd:choice minOccurs="0" maxOccurs="unbounded">
           <xsd:element ref="geographic-name"/>
           <xsd:element ref="historic-event"/>
           <xsd:element ref="person"/>
           <xsd:element ref="natural-resource"/>
           <xsd:element ref="product"/>
       </xsd:choice>
   </xsd:complexType>
   <xsd:complexType name="geographicNameType">
       <xsd:simpleContent>
           <xsd:extension base="xsd:string">
               <xsd:attribute name="id"
                              type="xsd:positiveInteger" use="required"/>
           </xsd:extension>
       </xsd:simpleContent>
   </xsd:complexType>
   <xsd:complexType name="historicEventType" mixed="true">
       <xsd:sequence>
           <xsd:choice minOccurs="0" maxOccurs="unbounded">
               <xsd:element ref="geographic-name"/>
               <xsd:element ref="person"/>
               <xsd:element ref="natural-resource"/>
               <xsd:element ref="product"/>
           </xsd:choice>
           <xsd:element ref="date" minOccurs="0"/>
           <xsd:choice minOccurs="0" maxOccurs="unbounded">
               <xsd:element ref="geographic-name"/>
               <xsd:element ref="person"/>
               <xsd:element ref="natural-resource"/>
               <xsd:element ref="product"/>
           </xsd:choice>
       </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="largestCitiesType">
       <xsd:sequence>
           <xsd:element name="city" maxOccurs="10">
               <xsd:complexType>
                   <xsd:attribute name="name" type="xsd:string"
use="required"/>
                   <xsd:attribute name="population"
                       type="xsd:positiveInteger" use="required"/>
               </xsd:complexType>
           </xsd:element>
       </xsd:sequence>
   </xsd:complexType>
   <xsd:element name="geographic-name" type="geographicNameType"/>
   <xsd:element name="historic-event" type="historicEventType"/>
   <xsd:element name="person" type="xsd:string"/>
   <xsd:element name="natural-resource" type="xsd:string"/>
   <xsd:element name="product" type="xsd:string"/>
   <xsd:element name="date" type="xsd:string"/>
   <xsd:complexType name="imageType">
       <xsd:attribute name="id" type="xsd:positiveInteger"
use="required"/>
   </xsd:complexType>
</xsd:schema>
```

groot – doordat deze operaties plaatsvinden op basis van *in-place, piecewise updates*. Met andere woorden niet de complete data hoeft opnieuw geschreven te worden.
- Snelheid van het uitvoeren van XPath queries op de XML data is groot – doordat deze operaties op basis van *streaming* geschieden, waardoor er geen DOM voor de data geconstrueerd hoeft te worden – een relatief dure operatie zeker bij grotere stukken XML.
- Constraints kunnen worden aangemaakt op de opgeslagen XML data – hierbij kunnen (dus) regels worden vastgelegd waaraan de inhoudelijke XML data dient te voldoen.
- Snelheid kan nog verder worden vergroot door de ondersteuning voor *XMLIndex,* naast function-based en Oracle Text indexes.

Hoe gebruik ik Binary XML?

In deze paragraaf wordt uitgelegd hoe je ervoor zorgt dat XML op basis van XML Binary opslag wordt gedaan. Hierboven werd beschreven dat deze XML zowel *Schemaless* – dus geen gekoppelde definitie – als op basis van een XML Schema definitie kan zijn. In het voorbeeld gaan we uit van XML die voldoet aan de XML Schema definitie op pagina 27.

Hierna laten we zien hoe we een tabel aanmaken waarbij de XML opgeslagen wordt op basis van binary XML storage. Om deze binary XML storage bovendien gebruik te kunnen laten maken van de hierboven gepresenteerde XML Schema definitie dient deze definitie te worden geregistreerd.

```
--
-- aanmaken van Oracle directory waarin het XML Schema staat en gelezen
kan worden
--
create or replace directory xmldir as 'D:\data\proj\Oracle11gRDBMS';

--
-- registreren van het XML Schema bij de Oracle database
--
begin
  dbms_xmlschema.registerSchema(
     schemaurl => 'http://www.cumquat.nl/xsd/state'
    ,schemadoc => bfilename('USSTATESDIR','US-States/xsd/state_xdb.
xsd')
    ,csid      => nls_charset_id('AL32UTF8')
    ,options   => DBMS_XMLSCHEMA.REGISTER_BINARYXML
    ,gentypes  => false
  );
end;
/
```

In onderstaand voorbeeld wordt gekozen om deze documenten in een kolom op te slaan en als extra een *virtuele* kolom te definiëren die een bepaald gegeven uit het XML document weergeeft. Een dergelijke kolom is een *synthesized* property. Dit is een gegeven dat feitelijk niet wordt opgeslagen maar afleidbaar kan worden vastgesteld.

```
--
-- definieren van een opslagstructuur voor US State XML documenten
--
create table cxd_xmldata_states
( id          number(10,0)        not null
, xmldata     xmltype             not null
, v_state     varchar2(2)         generated always as
(extractvalue(xmldata,'/state/abbreviation'))
, constraint cxd_xml_states_pk  primary key (id)
) xmltype column xmldata store as binary xml
  xmlschema "http://www.cumquat.nl/xsd/state" element "state"
/
```

De hiervoor beschreven *virtuele* kolom **v_state** kan handig worden gebruikt om bijvoorbeeld een aanvullende constraint op de betreffende tabel te definiëren. Bijvoorbeeld:

```
--
-- definieren aanvullende constraint op cxd_xmldata_states
--
alter table cxd_xmldata_states
add constraint cxd_xmldata_state_uk1
unique(v_state)
/
```

Abusievelijk aanleveren van eenzelfde *US State* document kan ten gevolge van deze constraint dan ook niet meer.

```
.ORA-00001: Schending van UNIQUE-beperking (JVISSERS.CXD_XMLDATA_STATE_
UK1).
```

Niet alleen uniciteit van een XML document over de gehele set van documenten kan hierdoor worden gecontroleerd, ook het afdwingen van *foreign key* relaties tussen XML documenten is hiermee mogelijk. Dit is als het ware een bonus bovenop de functionaliteit die geboden wordt door het feit dat XML gerelateerd is aan een XML Schema definitie. In bovenstaand voorbeeld betekent dit dat **xmldata** alleen gegevens zal toestaan die voldoet aan de *US State* definitie. Als we bijvoorbeeld een ongeldige state document (*errorstate.xml*) willen toevoegen dan krijgen we een foutmelding, terwijl een "juist" document (uiteraard) geen problemen oplevert.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- "Fout" document: errorstate.xml -->
<state>
          <stateImage id="XYZ" />
</state>
```

```
--
-- Toevoeging van een "foute" staat
--
insert into cxd_xmldata_states
(id
,xmldata
) values
(51
,xmltype(bfilename('USSTATESDIR','US-States/data/errorstate.xml'),nls_
charset_id('AL32UTF8'))
);

FOUT in regel 6:
.ORA-31011: Ontleden van XML is mislukt.
ORA-19202: Fout in XML-verwerking (
LSX-00234: invalid decimal "XYZ"
).
```

In bovenstaand voorbeeld is een tabel aangemaakt met daarin een XMLType kolom, waarbij de gegevens te allen tijde moeten voldoen aan de XML Schema definitie. In het geval dat we het *errorstate.xml* document wel hadden willen toelaten tot de tabel dan hadden we deze op een iets andere wijze moeten definiëren.

```
--
-- definieren van een opslagstructuur voor US State XML documenten
--
create table cxd_xmldata_states
( id          number(10,0)        not null
, xmldata     xmltype             not null
, v_state     varchar2(2)         generated always as
(extractvalue(xmldata,'/state/abbreviation'))
, constraint cxd_xml_states_pk  primary key (id)
) xmltype column xmldata store as binary xml
  xmlschema "http://www.cumquat.nl/xsd/state" element "state" allow
nonschema
/
```

Opties	Resultaat
store as binary XML	Schemaless XML.
store as binary XML xmlschema …	Alleen XML documenten volgens de genoemde XML Schema definitie worden toegelaten.
store as binary XML xmlschema … allow nonschema	XML documenten die de XML Schema definitie refereren worden toegelaten en Schemaless XML.
store as binary XML allow anyschema	Alleen XML documenten volgens een of ander geregistreerde XML Schema definitie worden toegelaten.
store as binary XML allow anyschema allow nonschema	XML documenten die een XML Schema definitie refereren, moeten daaraan voldoen om toegelaten te worden en Schemaless XML worden toegelaten.

Tabel 1.

Door de toevoeging **allow nonschema** kunnen ook XML documenten die niet verwijzen naar enig andere geregistreerd XML Schema definitie – met andere woorden schemaless zijn – worden opgeslagen in de tabel. Naast allow nonschema toevoeging kan ook nog gebruik worden gemaakt van **allow anyschema.** Hierdoor ontstaan de beslissingsopties zoals weergegeven in tabel 1.

Is dat alles?

Zo ongeveer wel ja. Het verrassende van Binary XML is dat het transparant in gebruik is. Voor een applicatieontwikkelaar verandert er relatief weinig. We krijgen met Binary XML nu de gelegenheid om alles te doen wat we al konden/wilden doen met *gestructureerde* opslag – zonder dat we daarbij moeten inleveren op de flexibiliteit die we altijd al hadden met *ongestructureerde* opslag. Ter afsluiting; een van de belangrijke minpunten van *gestructureerde opslag* en Java/JDBC toegang is het feit dat je – in bepaalde situaties – gebruik moet maken van een JDBC thick driver. Met Binary XML is deze randvoorwaarde er niet en kan "gewoon" gebruik worden gemaakt van een thin driver.

XMLIndex

Er is niet alleen een nieuwe manier om XML in een database op te slaan. Ook aan het verbeteren van query performance op XML data is gedacht met de introductie van XMLIndex. Dit indexeringsmechanisme is een uitbreiding op B-tree, function-based en Oracle Text indexering. (Overigens is de CTXXPath index vanaf Oracle11g deprecated.) Om maar direct met de deur in huis te vallen, wordt hieronder een voorbeeld gegeven van het aanmaken van een dergelijke index:

```
create index state_xmldata_idx on cxd_xmldata_states(xmldata) indextype
is XDB.XMLIndex
parameters ('PATH TABLE states_path_table');
```

Feitelijk is het gedeelte dat begint met ***parameters*** ('… tot en met het eind optioneel. Dit deel is met opzet aan het voorbeeld toegevoegd ter ondersteuning van de uitleg van XMLIndex. In tegenstelling tot een "normale" B-tree index is een XMLIndex zeer algemeen van aard. Bij een B-tree index dienen te indexeren paden expliciet aangegeven te worden, terwijl bij een standaard aangemaakte XMLIndex alle mogelijke XPath expressies geïndexeerd worden.

Waaruit bestaat een XMLIndex

Een XMLIndex bestaat uit drie onderdelen.

- **PATH index** – hiermee worden de verschillende XML tags in een document geïndexeerd, waarmee feitelijk alle afzonderlijke document*fragmenten* gelokaliseerd kunnen worden.
- **ORDER index** – waarmee de hiërarchische ordening van

een document geïndexeerd wordt en daarmee dus onder andere parent-child relaties.
- **VALUE index** – waarmee de feitelijke *waarden* in een document, de "dingen" die tussen tags staan, geïndexeerd worden.

De index wordt geïmplementeerd met behulp van een zogenaamde PATH table aangevuld met een aantal daarbij behorende *secundaire* indexen. De PATH table bevat een enkele rij voor iedere geïndexeerde node, waarbij voor iedere node wordt bijgehouden:

- ROWID – van de tabel waarin het document is opgeslagen.
- LOCATOR – die toegang verschaft tot het feitelijke document fragment.
- ORDER KEY – die de hiërarchische positie van de node in het document aangeeft.

De PATH table is een speciaal soort tabel waarvan de definitie met een normale describe operatie te bekijken is. Indien je bij het aanmaken van de XMLIndex geen parameters hebt meegegeven, dan zal Oracle een systeemgegenereerde naam maken (PATH_TABLE_NAME kolom in USER_XML_INDEXES tabel). In ons geval weten we echter de naam van de echte tabel.

```
SQL> desc states_path_table
 Name                                      Null?    Type
 ----------------------------------------------------------------------
 RID                                               ROWID
 PATHID                                            RAW(8)
 ORDER_KEY                                         RAW(1000)
 LOCATOR                                           RAW(2000)
 VALUE                                             VARCHAR2(4000)
```

Da's mooi, dan zou je er dus ook uit moeten kunnen selecteren?

```
SQL> select * from states_path_table;
select * from states_path_table
              *
ERROR at line 1:
ORA-30967: operation directly on the Path Table is disallowed
```

Nee dus. Hoewel de PATH table volgens mij ook wel interessante informatie bevat om inzichtelijk te maken, heeft Oracle ervoor gekozen om dit concept als *black* box neer te zetten. Letterlijk wordt gezegd: "Ignore the Path Table; It Is Transparent"; schoorvoetend geef ik dan maar toe.
Zoals hierboven werd aangegeven wordt de PATH table aangevuld met secundaire indexen. Oracle zal deze indexen als onderdeel van het ***create index*** commando automatisch aanmaken. Informatie over deze indexen kan op de normale wijze worden opgevraagd, bijvoorbeeld zo:

```
SQL> select index_name, column_name, column_position from user_ind_
columns
  1  where  table_name  = 'STATES_PATH_TABLE'
  2  order  by index_name, column_name;

INDEX_NAME                                                  COLUMN_NAME
COLUMN_POSITION
------------------------------------------------------------------------
SYS70406_STATE_XMLD_ORDKEY_IX                                ORDER_KEY
2
SYS70406_STATE_XMLD_ORDKEY_IX                                    RID
1
SYS70406_STATE_XMLD_PATHID_IX                                 PATHID
1
SYS70406_STATE_XMLD_PATHID_IX                                    RID
2
SYS70406_STATE_XMLD_VALUE_IX                               SYS_NC00006$
1
```

Merk op dat hier gaat om de drie eerder genoemde onderdelen van de XMLIndex; *PATH, ORDER* en *VALUE* index. De VALUE index is hierbij voor wat betreft een SQL Query predicaat (WHERE clausule) de belangrijkste. Oracle zal hiervoor een indexwaarde op basis van iedere afzonderlijk voorkomend XML (text) element en ieder afzonderlijke attribuutwaarde aanmaken.

XMLQuery met XMLIndex

Met deze basisuitleg, wordt hieronder een voorbeeld van de uitwerking van de XMLIndex op een redelijk triviale query gepresenteerd. Om bijvoorbeeld de eerste paragraaf uit de beschrijving te laten zien van het XML document dat hoort bij de staat met als nickname: "Sunflower State, Jayhawk State" voeren we in:

```
SQL> select xmlquery('/state/description/p[1]' passing xmldata retur-
ning content)
  1  from   cxd_xmldata_states
  2  where  xmlexists('/state[nickname="Sunflower State, Jayhawk
State"]' passing xmldata)

XMLQUERY('/STATE/DESCRIPTION/P[1]'PASSINGXMLDATARETURNINGCONTENT)
<p>
  Spanish explorer Francisco de Coronado, in 1541, is
  considered the first European to have traveled this region.
  Sieur de la

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter( (SELECT "SYS_P3"."VALUE" FROM "JVISSERS"."STATES_PATH_
TABLE"
       "SYS_P0","JVISSERS"."STATES_PATH_TABLE" "SYS_P3" WHERE "SYS_
P3"."RID"=:B1 AND
```

```
"SYS_P3"."PATHID"=HEXTORAW('2491')  AND SYS_XMLI_LOC_ISNODE("SYS_
P3"."LOCATOR")=1 AND
        "SYS_P0"."RID"=:B2 AND "SYS_P0"."PATHID"=HEXTORAW('761C')  AND

SYS_XMLI_LOC_ISNODE("SYS_P0"."LOCATOR")=1 AND "SYS_P0"."ORDER_KEY"<"SYS_
P3"."ORDER_KEY" AND
        "SYS_P3"."ORDER_KEY"<SYS_ORDERKEY_MAXCHILD("SYS_P0"."ORDER_
KEY") AND

SYS_ORDERKEY_DEPTH("SYS_P0"."ORDER_KEY")+1=SYS_ORDERKEY_DEPTH("SYS_
P3"."ORDER_KEY") AND
         "SYS_P3"."RID"="SYS_P0"."RID")='Sunflower State, Jayhawk
State')
   5 - filter(SYS_XMLI_LOC_ISNODE("SYS_P3"."LOCATOR")=1)
   6 - access("SYS_P3"."PATHID"=HEXTORAW('2491')  AND "SYS_
P3"."RID"=:B1)
   7 - access("SYS_P0"."PATHID"=HEXTORAW('761C')  AND "SYS_
P0"."RID"=:B1)
       filter("SYS_P3"."RID"="SYS_P0"."RID")
   8 - filter(SYS_XMLI_LOC_ISNODE("SYS_P0"."LOCATOR")=1 AND "SYS_
P0"."ORDER_KEY"<"SYS_P3"."ORDER_KEY"
        AND "SYS_P3"."ORDER_KEY"<SYS_ORDERKEY_MAXCHILD("SYS_P0"."ORDER_
KEY") AND

SYS_ORDERKEY_DEPTH("SYS_P0"."ORDER_KEY")+1=SYS_ORDERKEY_DEPTH("SYS_
P3"."ORDER_KEY"))
```

Voorgaand executieplan is omwille van de layout aangepast. In het ontstane executieplan van de query kunnen we zien dat de XMLIndex gebruikt wordt, omdat de STATES_PATH_TABLE vermeld staat.

Aanvullende secundaire indexen

Naast dat Oracle bij het aanmaken van een XMLIndex zelf een aantal secundaire indexen aanmaakt, bestaat ook de mogelijkheid om expliciet nog aanvullende secundaire indexen op de VALUE kolom van de PATH table te definiëren. De standaard secundaire index is een tekstgeoriënteerde, maar zelf kun je ook indexen maken die numeriek van aard zijn. Dat gaat even anders, dan dat je dat jezelf wellicht zou voorstellen:

```
call dbms_xmlindex.createNumberIndex('JVISSERS','STATE_XMLDATA_IDX','A_
NUMERIC_INDEX');
```

Door een dergelijke index is bijvoorbeeld de volgende query te optimaliseren:

```
select xmlquery('/state/name' passing xmldata returning content) from
cxd_xmldata_states
where  xmlexists('/state/largest-cities/city[@population>1000000]' pas-
sing xmldata);
```

Op eenzelfde manier kan ook een secundaire index worden aangemaakt op een datumveld. Hiermee kan nog explicieter worden gestuurd op welke manier Oracle zijn queries uitvoert.

Voordelen van XMLIndex

XMLIndex kan gebruikt worden bij XMLQuery, XMLTable, XMLExists, XMLCast, extract, extractValue en existsNode en is daarmee zeer veelzijdig. Voor de volledigheid is het interessant om de belangrijkste voordelen van deze nieuwe toevoeging aan de XMLDB ondersteuning in Oracle op te sommen.

- XMLIndex gebruik beperkt zich niet alleen tot de WHERE clause – dit in tegenstelling tot andere indexeringstechnieken die gebruikt worden bij XML. Hierdoor kunnen ook de SELECT en FROM onderdelen profiteren van de eventueel te boeken snelheidswinst.
- XMLIndex is generiek in het gebruik – je hoeft op voorhand niet te weten welke XPath expressies je wilt gaan gebruiken om XML data te bevragen, ten einde de juiste indexen aan te kunnen maken. Aan de andere kant – als je al wel snel weet op welke manier je queries gaan verlopen, kun je deze kennis gebruiken om secundaire indexen aan te brengen op de XMLIndex waarin dit tot uiting komt.
- Indexering op basis van de XMLIndex beperkt zich niet tot XML data dat voldoet aan een bepaalde XML Schema definitie, maar kan even goed gebruikt worden bij schemaless XML. Ook is XMLIndex onafhankelijk van het uiteindelijke storage mechanisme, of dit nu structured, unstructured of binary is.
- Het aanmaken/bijhouden van een XMLIndex kan eventueel worden geparallelliseerd, door er meerdere database processen "aan te zetten". Indien XML binair is opgeslagen, kan het bijwerken bovendien op basis van piecewise updates worden gefaciliteerd door de database.

Samenvatting

In dit artikel zijn twee in het oog springende uitbreidingen op het gebied van XML in de Oracle database gepresenteerd. Binary XML lijkt met zijn voordelen de geprefereerde opslagstructuur te kunnen gaan worden van XML. Parallel hieraan vormt XMLIndex een volgende optimalisatie om ook de performance van het toegangspad tot XML (fragmenten) te verbeteren. Al met al dragen deze technieken bij aan de volgende (laatste?) stap van volwassenheid van Oracle's XML-ondersteuning in de database.

Jan Vissers is Technology Manager bij Cumquat Information Technology. Reacties op dit artikel kunnen gemaild worden naar jan@cumquat.nl.