Enterprise

Dit is het tweede in een serie van twee artikelen over de meerwaarde van een database binnen Java-applicaties. Daarbij kijken we naar mogelijkheden om applicaties productiever te ontwikkelen, beter te laten performen, veiliger en meer integer te maken en zelfs functioneel aan rijkheid te laten winnen. In deze aflevering gaan we dieper in de op database specifieke karakteristieken die kunnen worden benut binnen de database onafhankelijke API die in het vorige deel werd beschreven.

De database maakt Java-applicatie beter

Benutten van specifieke karakteristieken

Lucas Jellema
is Senior Consultant voor Java en SOA bij AMIS.

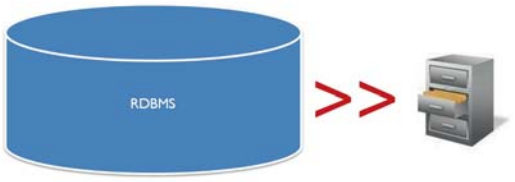
We kijken dan bijvoorbeeld naar features die SQL ons biedt in de Oracle RDBMS – zoals Analytische Functies, Pivot en Flashback Queries. Daarnaast kijken we naar het gebruik van cursors, Database Change Notifications, HTTP als alternatief voor JDBC, asynchrone operaties en autonome transacties.

Geavanceerde SQL-functionaliteit

In het eerste deel hebben we besproken hoe Views kunnen worden gebruikt om databasespecifieke SQL-faciliteiten aan applicaties beschikbaar te stellen via eenvoudige select * from view statements. De view kan er in alle databases hetzelfde uitzien qua interfaces (lees: kolomdefinities) maar heel verschillend worden geïmplementeerd. Nog meer afscherming wordt bereikt als de SQL-queries in Stored Procedures zijn opgenomen. Deze kunnen bijvoorbeeld een cursor als output parameter opleveren of geëmbed zijn in Views.
Specifieke SQL-functionaliteit kan diverse soorten zoekoperaties en bewerkingen vereenvoudigen, vaak met een substantiële ontlasting van de applicatiecode tot gevolg.

Analytische Functies

Rapportages, overzichten, analyses en vergelijkingen, aggregaties en grafieken in Java-applicaties kunnen aanzienlijk worden versneld en vereenvoudigd door de mogelijkheden van SQL optimaal te benutten. De analytische functies in Oracle SQL zijn een goed voorbeeld. Deze functies maken het mogelijk om bij het opvragen van records data uit



andere records te benaderen, vergelijkbaar met formules in een spreadsheet. De volgende query toont voor iedere medewerker het salaris van de persoon die net na hem in dienst kwam:

```
select ename
,      sal
,      lead(sal) over (order by hiredate)
salaris_van_nakomer
from   emp
```

Andere analytische functies kunnen worden gebruikt om een bewegend gemiddelde uit te rekenen – bijvoorbeeld de gemiddelde omzet over een periode van dertig dagen gedurende het jaar – of de running sum van het aantal nieuwe abonnees door het jaar heen.

Bijvoorbeeld de gemiddelde omzet door het jaar heen, steeds over een periode van een maand gemeten op het moment van iedere order:

```
select order_date
,      avg(order_total) over (order by order_date
       RANGE NUMTODSINTERVAL(15,'day')
       PRECEDING and NUMTODSINTERVAL(15,'day')
       FOLLOWING) as gemiddelde_omzet from  orders
```

Analytische functies kunnen ook worden gebruikt

voor percentielberekeningen, ordening van gegevens – op welke plek staat iedere medewerker qua salaris binnen zijn afdeling, functiegroep en jaarclub) – en samenvoeging van string-waarden tot CSV lijstjes. Van dat laatste als voorbeeld deze query die voor iedere medewerker een alfabetisch comma-separated-values lijstje van zijn directe collega's binnen de afdeling oplevert:

```
select ename
,      deptno
,      listagg( ename, ',')
       within group (order by ename)
       over (partition by deptno)
       from emp
```

Pivot

De pivot table (ook wel draaitabel) ken je misschien in het kader van Excel. De kreet Pivot slaat op het omdraaien in een dataset van rijen en kolommen, bijvoorbeeld om de presentatie van de data te verbeteren of juist om berekeningen mogelijk te maken op onhandig gestructureerde maar voor mensen plezierig gepresenteerde data. In SQL brengt de volgende query de data uit de tabelstructuur naar een vorm die voor de applicatie eenvoudig te presenteren is. Bijvoorbeeld om uit de EMP-tabel met medewerkers (kolommen DEPTNO, JOB, SAL) een net matrix-overzicht te presenteren van het aantal medewerkers in een bepaalde job, per afdeling en ook hun salaris totaal, volstaat de volgende pivot-query:

```
select *
from (select job
      ,      deptno
      ,      sal
      from emp)
      pivot ( count(*) count_staff
      ,      sum(sal) salsum
             for deptno in (10, 20, 30))
```

Deze query resulteert in twee kolommen in de matrix voor elk van de afdelingen 10, 20 en 30 – een kolom voor count_staff en eentje voor salsum. Daarnaast is er de 'gewone' kolom job in iedere rij. De cellen van de matrix bevatten voor de job de waarden van count_staff en salsum:

Een ander voorbeeld – maar dan met unpivot: een tijdschrijfapplicatie toont rijen per week en per urencode met een kolom voor iedere dag in de week. Voor automatische verwerking zou een tabelstructuur met vier kolommen – medewerker, urencode, datum en aantal uren – het meest eenvoudig zijn om optellingen en andere bewerkingen te doen. Stel dat de tabel juist de makkelijk-te-presenteren structuur heeft (met kolommen voor iedere dag van de week, het weeknummer, de projectcode en het employeenummer), dan kunnen met behulp van de unpivot de rijen worden gesplitst in meerdere rijen, voor iedere dag-kolom een andere rij:

```
JOB       10_COUNT_STAFF 10_SALSUM 20_COUNT_STAFF 20_SALSUM 30_COUNT_STAFF 30_SALSUM
--------- -------------- --------- -------------- --------- -------------- ---------
CLERK     1              1300      2              1900      1              950
SALESMAN  0                        0                        4              5600
PRESIDENT 1              5000      0                        0
MANAGER   1              2450      1              2975      1              2850
ANALYST   0                        2              6000      0
```

```
select *
from employee_week_records unpivot (hours for day
in (mon,tue,wed,thu,fri,sat,sun))
```

De tabel met 10 kolommen levert in deze query als gevolg van de unpivot rijen met vijf kolommen: empno, project_code, hours, weekno, day. Iedere record in de tabel resulteert in zeven rijen in het resultaat van de query, een voor iedere dag in de week. NB: de kolommen in de tabel heten mon, tue, wed enzovoorts.

Flashback Query

Een geavanceerde optie in queries in de Oracle database is de flashback optie. Kort gezegd komt deze neer op het terugkijken in de tijd. In plaats van de data uit de database te halen zoals die er nu in staat (de gebruikelijke manier van query-en), kun je het tijdstip aangeven waarnaar je wilt terugkijken. Bijvoorbeeld om de data te zien in de tabel EMP van vorige week kun je deze query uitvoeren:

```
select *
from emp as of timestamp (systimestamp - interval '7'
day)
```

Oracle doet de flashback query met hetzelfde 'versioning' mechanisme dat wordt gebruikt voor read-consistency: als de ene sessie records heeft gewijzigd in een tabel maar nog niet heeft gecommit, zal een query in een andere sessie de records van voor de wijziging tonen. Dat interne multi-versie mechanisme is uitgebreid en via flashback beschikbaar voor applicatieontwikkelaars. Toepassingen van flashback zijn ondermeer het vinden van verschillen tussen twee momenten in de tijd en eventueel het herstellen van fouten. Het volgende statement draait bijvoorbeeld alle salariswijzigingen van de afgelopen 24 uur terug:

```
update emp emp_now
set sal = (select sal from emp_was as of timestamp
                (systimestamp - interval '24' hour)
                where emp_was.empno = emp_now.empno)
```

In Oracle 11g is het flashback data archive geïntroduceerd. Daarmee kan een DBA gericht flashback data archiveren voor tabellen gedurende instelbare periodes – bijvoorbeeld overeenkomstig wettelijke bewaartermijnen of historische rapportage-eisen. Een andere toepassing van flashback query is trend-watching. Een variant op de AS OF syntax maakt het mogelijk om van databaserecords al hun vroege-

Met de flashback optie wordt terugkijken in de database mogelijk.

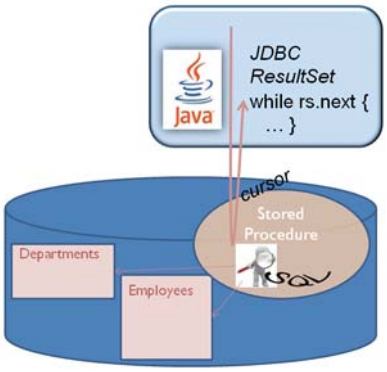
re versies over een bepaalde periode op te vragen:

```
select ename
,      sal
,      versions_operation INS_UPD_DEL
,      versions_starttime
,      versions_endtime
,      versions_xid transaction_id
from emp versions between
timestamp systimestamp - 365 and systimestamp
```

Op basis van deze query is het heel eenvoudig om een grafiek te tonen met de salarisontwikkeling van iedere medewerker – en het gemiddelde van alle medewerkers – door de tijd.

Cursor voor ontkoppeling

Databases kunnen vanuit stored procedures parameters publiceren van een cursor-type. Via een cursor krijgt een applicatie – Java, PL/SQL of .Net – toegang tot de resultaatset van een query, zonder iets over de query zelf te weten. Welke SQL-mechanismen zijn gebruikt, welke tabellen zijn benaderd en welke rechten daarvoor nodig waren – dat alles



is afgeschermd voor de applicatie.
In Java kan een cursor parameter eenvoudig worden gemapped naar een JDBC ResultSet. Bij het benaderen van de JDBC ResultSet maakt het niet uit of deze via een 'prepared SQL Query statement' aan zijn data is gekomen of door een cursor parameter aan een data set is geholpen. Bijvoorbeeld:

```
String query = "begin ? := hrm_api.get_dept_data(?);
end;";
CallableStatement stmt = conn.prepareCall(query);
// register the type of the out param -
// an Oracle specific type
    stmt.registerOutParameter(1, OracleTypes.CURSOR);
    stmt.setInt(2, deptno);
// execute and retrieve the result set
    stmt.execute();

ResultSet rs = (ResultSet)stmt.getObject(1);
// een geneste cursor
    while (rs.next()) {
    System.out.println(rs.getString(2) + "\t" + rs.
    getInt(1)
+"\t" + rs.getString(3));
    ResultSet staff = (ResultSet)rs.getObject(4);
    System.out.println("** Staff");
    while (staff.next()) {

    System.out.println("  "+staff.getString(2) + "\t"
+staff.getInt(1)

+ "\t" +staff.getInt(3) + "\t" +staff.getString(4));
```

In Java kan een cursor parameter simpel worden gemapped naar een JDBC ResultSet.

```
ResultSet subordinates = (ResultSet)staff.getObject(5);
// een dieper geneste cursor
```

De geneste dataset van departments, medewerkers en hun ondergeschikten wordt via een cursor aan de Java-applicatie beschikbaar gesteld. De stored procedure get_dept_data die wordt aangeroepen zou in PL/SQL als volgt gedefinieerd kunnen zijn:

```
  function get_dept_data
  ( p_deptno in number
  ) return   SYS_REFCURSOR
  is
    c_dept sys_refcursor;
  begin
    open c_dept for
    select deptno , dname, loc
    ,      cursor( select empno, ename, sal, job
    ,      cursor ( select empno,ename,job from emp sub
    where sub.mgr = e.empno) subordinates from emp
    e where e.deptno = dept.deptno) staff
      from   dept
      where  dept.deptno = p_deptno;
    return c_dept;
  end get_dept_data;
end hrm_api;
```

Cursoren zijn beschikbaar in vrijwel iedere database, net als stored procedures. Java-applicaties kunnen dus prima tegen een generieke cursor gebaseerde API worden ontwikkeld die vervolgens in iedere database specifiek wordt geïmplementeerd. Op deze manier kan de Java volledig SQL-vrij blijven, in elk geval voor read only toegang tot data!

Http gebaseerde interactie

Java-applicaties zullen de database veelal benaderen via JDBC drivers die via het netwerk connecties leggen met de database. Via de JDBC connectie kunnen applicaties SQL-statements uitvoeren en Stored Procedures benaderen – direct of via persistence frameworks. Er is voor ondermeer de Oracle RBDMS aan alternatieve benaderingswijze beschikbaar, die geen JDBC en speciale netwerktoegang behoeft. Stored Procedures in de Oracle-database kunnen benaderbaar worden gemaakt via HTTP. Een normaal http get- of post-request kan door de applicatie worden verstuurd en door de database worden ontvangen en omgezet in een aanroep van een Stored Procedure. Het antwoord van de procedure wordt als het http-response bericht door de database teruggestuurd naar de aanroeper.
Bijvoorbeeld deze http aanroep:
http://thedataserver:2100/hrmapi/rest/department/10 wordt verwerkt door een aanroep van de volgende procedure:

```
procedure process_department(p_deptno in number) is
begin
  for l_rec in (select * from dept where deptno
= p_deptno) loop
    htp.p(l_rec.deptno || ';' || l_rec.dname || ';' ||
l_rec.loc || '');
  end loop;
```

```
end;
```

Alles wat door de procedure naar het buffer package HTP wordt geschreven, wordt door de HTTP infrastructuur van de Embedded PL/SQL Gateway van de Oracle database in de HTTP response geschreven. In dit geval levert dat een vrij sober HTTP response-bericht op:

Via deze weg kunnen ook andere mimetypes – zoals binary file types voor plaatjes en PDF of Word documenten worden gedownload – rechtstreeks vanuit de database naar de http client. De HTTP requests die naar de database worden gestuurd kunnen ook multipart post requests zijn met geuploade documenten.

Om de HTTP infrastructuur te kunnen gebruiken moet door de DBA via het standaard package dbms_epg de toegang tot databaschema's via HTTP worden aangezet.
Oracle stelt een licentievrij PaaS-platform beschikbaar onder de naam Application Express, kortweg APEX. Met dit platform kunnen browser based applicaties worden ontwikkeld via een browser. APEX applicaties – en het APEX ontwikkelplatform zelf – draaien volledig binnen de Oracle database, via packages die met dbms_epg via http beschikbaar worden gesteld.

De HTTP interactie met de Oracle RDBMS is niet alleen van applicatie naar database, maar kan ook andersom zijn: de applicatie kan via HTTP ook de applicatie aanspreken. Vanuit Stored Procedures – en zelfs binnen SQL Queries – kan via het standaard package utl_http of met de built in functie httpuritype een HTTP request verstuurd worden.

Zo kan vanuit de database ondermeer een RESTful of SOAP WebService worden aangeroepen en ook een RSS feed worden ingelezen.
Een heel simpele HTTP GET aanroep vanuit PL/SQL ziet er als volgt uit:

```
select utl_http.request('www.nu.nl')
from   dual
```

Het package utl_http bevat allerlei procedures om een volwaardig GET of POST request samen te stellen. PUT en DELETE worden niet ondersteund. De volgende query vraagt een RSS feed op via HTTPURITYPE van een externe URL en gebruikt vervolgens de XMLTABLE operator die een XML document met een XQuery statement kan omzetten in een relationele data set – waartegen een normaal select-statement kan worden uitgevoerd:

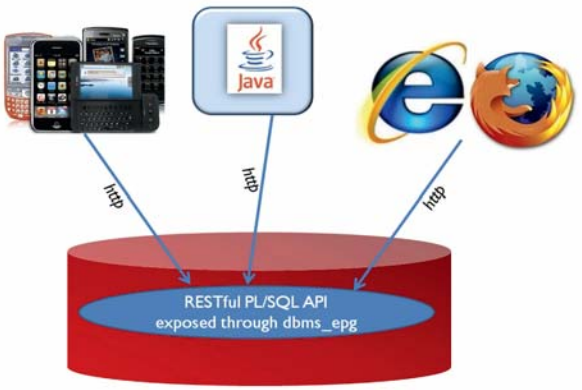
```
select title
,      link
,      to_date(substr(publication_date,6,20),
'dd mon yyyy hh24:mi:ss') timest
from   xmltable('for $i in //item

return <Article>{$i/title}{$i/pubDate}{$i/link}</
Article>'

passing httpuritype('http://technology.amis.nl/
blog/?feed=rss2').getXML()
                    COLUMNS

title               varchar2(100) path 'title'

, link                varchar2(100) path 'link'
                    , publication_date  varchar2(100)
path 'pubDate'
                  )
```

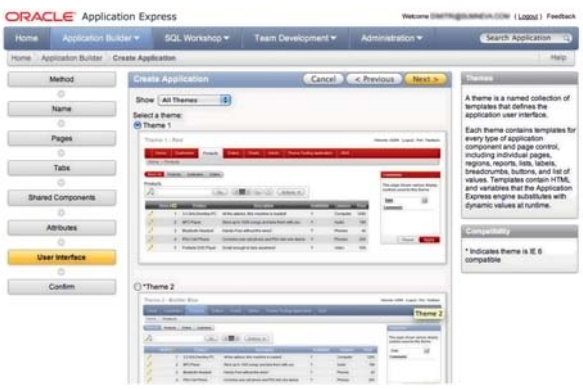
Of het een goed idee is dat de database rechtstreeks het internet opgaat zou je sterk kunnen en moeten afvragen. Het ligt wellicht meer voor de hand om de database in zijn HTTP verkeer te beperken tot de lokale middle tier.
Een potentieel waardevolle toepassing van HTTP verkeer vanuit de database ligt op het vlak van notificatie: de database kan bijvoorbeeld door een Servlet aan te roepen doorgeven aan een web applicatie dat data in een middle tier cache ververst moet worden. Daarnaast kan de middle tier als proxy optreden voor de database, die daarmee indirect wel degelijk aan ondermeer service interactie en internetbenadering zou kunnen doen.

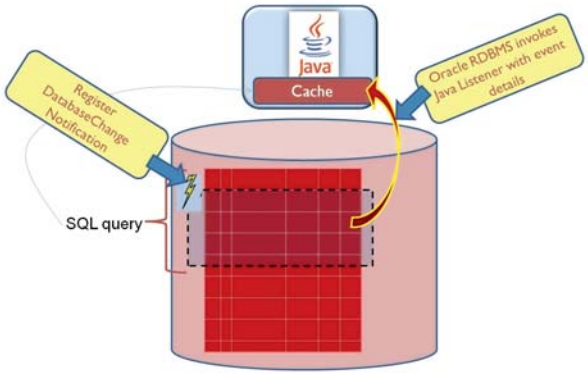
Het is de vraag of het goed is dat een database rechtstreeks het internet opgaat.

Database Change Notifications

Data wordt in veel webapplicaties vastgehouden in caches op de middle tier. Vanaf het moment van opvragen van data uit de database en het instantiëren van POJO's is de data in zekere zin al gecachet. Inzet van sessie level of application level (L2) cache of zelfs een datagrid (mogelijk over applicaties heen) maakt dat nog veel meer data buiten de database wordt gecachet. Veel van de wijzigingen in die data worden via de applicatie – en dus ook via de cache – doorgevoerd. Echter, sommige wijzigingen komen direct in de database binnen. Dan zou het handig zijn als de data-







base aan de applicatie laat weten dat er datawijzigingen zijn die wellicht interessant zijn voor de middle tier, bijvoorbeeld om caches bij te werken of gebruikers via server push een signaal te geven. We hebben net gezien hoe de database via HTTP de applicatie zou kunnen benaderen. Daarnaast biedt de Oracle RDBMS een krachtig mechanisme op dit vlak.

Met Database Query Result Change Notication implementeert de database het aloude Hollywood principe: don't call us, we will call you. (You will have to leave your number). De applicatie kan bij de database interesse aangeven in bepaalde categorieën datawijzigingen en een listener object registreren dat in geval van een relevante wijziging automatisch door de database wordt aangeroepen.
Een registratie van een change notification listener omvat een tabel of een specifieke SQL query en instrueert de database om de listener aan te roepen wanneer de inhoud van de tabel of het resultaat van de query door een gecommitte transactie is gewijzigd.

Als een database transactie – waar deze ook vandaan komt – een record wijzigt in de set die onder de registratie valt, dan wordt de callback naar de listener uitgevoerd – via de Oracle JDBC Driver. In de aanroep van de listener wordt een object doorgegeven dat ondermeer de ROWIDs bevat van alle betrokken records en een indicatie van het soort operatie -creatie, verwijdering of wijziging.
De registratie van de change notification listener wordt gecreëerd met deze code:

```
Properties props = new Properties();
props.setProperty(OracleConnection.DCN_QUERY_CHANGE_
NOTIFICATION,"true");
props.put(OracleConnection.DCN_NOTIFY_ROWIDS,"true");
dcr = conn.registerDatabaseChangeNotification (props);
     DCNListener listener = new DCNListener();
     dcr.addListener(listener);
     Statement stmt = conn.createStatement();
// Associate the statement with the registration.
     ((OracleStatement)stmt).setDatabaseChange
Registration(dcr);
     ResultSet rs = stmt.executeQuery
("select sal from emp where job='CLERK'");
```

In dit geval wordt de listener aangeroepen als het salaris van een CLERK in de EMP tabel wijzigt – of als er een CLERK bijkomt of afgaat.
De listener heeft de volgende method signatuur:

```
public void onDatabaseChangeNotification(DatabaseChange
Event databaseChangeEvent)
```

In het DatabaseChangeEvent object zitten ondermeer de ROWIDs van de gewijzigde database records, aan de hand waarvan snel de gewijzigde data uit de database kan worden opgehaald.

Autonome Transacties

Transacties zijn essentieel in relationele databases. Een transactie is een opvolgende, logisch samenhangende set van databasewijzigingen binnen een sessie. Transacties worden in hun geheel vastgelegd (commit) of ongedaan gemaakt (rollback). Pas na de commit zijn de wijzigingen die in de sessie zijn doorgevoerd zichtbaar voor alle andere databasesessies. Controle over de transactie ligt in de applicatie – bijvoorbeeld in JPA via de commit methode op het transaction object dat via de EntityManager kan worden opgevraagd.

Oracle RDBMS en IBM DB2 ondersteunen beide het concept 'autonomous transaction'. Een autonome transactie is een transactie binnen een transactie, een soort uitstapje uit de hoofdtransactie die de sessie momenteel uitvoert. Dit uitstapje kan ook wijzigingen doorvoeren – en committen – zonder de hoofdtransactie te beïnvloeden. Ook als de hoofdtransactie vervolgens wordt teruggerold, zijn de wijzigingen die de autonome transactie heeft aangebracht blijvend gecommit.
Een autonome transactie in Oracle RDBMS wordt gerealiseerd door middel van een Stored Procedure (of Trigger) met een speciale instructie:

```
create or replace
procedure log
( p_txt in varchar2) is PRAGMA AUTONOMOUS_TRANSACTION;
begin
  insert into log_tbl
  (log_time, log_text)
  values (SYSTIMESTAMP, p_txt);
  commit;
end;
```

Let op het commit statement aan het eind van de procedure: de autonome transactie moet worden gecommit (of teruggerold).
Autonome transacties worden bijvoorbeeld gebruikt om achtergrond jobs te starten, om berichten op een Advanced Queue te plaatsen en om logging te schrijven – die ook of misschien wel juist bewaard blijft als de transactie wordt teruggerold vanwege excepties.
Een PL/SQL functie met een pragma autonomous_transaction kan een SELECT statement uitvoeren en toch worden aangeroepen binnen een SQL query of vanuit een DML trigger. De functie wordt uitgevoerd in zijn eigen transactie en ziet dus niet de ongecommitte wijzigingen van de hoofdtransactie. Dat kan handig zijn om in de hoofdtransactie te achterhalen welke wijzigingen er in de transactie zijn doorgevoerd.

De beste manier om performance te verbeteren is door iets helemaal niet meer te doen.

Asynchrone Operaties

De beste manier om performance te verbeteren is niet door iets sneller te doen, maar door het helemaal niet meer te doen. Een manier waardoor het lijkt alsof niets meer gedaan hoeft te worden, is door het asynchroon te laten doen, in een andere thread. Java kent dat mechanisme uiteraard en ook de Oracle-database kan asynchroon taken uitvoeren. Een webapplicatie kan bijvoorbeeld een gebruiker op een knop laten drukken en binnen een seconde een terugmelding doen, ook al moet er een taak worden uitgevoerd die minstens dertig seconden duurt. Door de taak als 'job' te schedulen in de database, wordt deze op de achtergrond uitgevoerd.
De meest eenvoudige manier om een actie op deze manier asynchroon te laten uitvoeren is met een statement als dit:

```
declare
   l_job_id number;
   l_job varchar2(10000):= 'begin update emp_big_table
   set sal = complex_formula(sal); commit; end;';
begin
  dbms_job.submit( l_job_id, l_job);
  commit;
end;
```

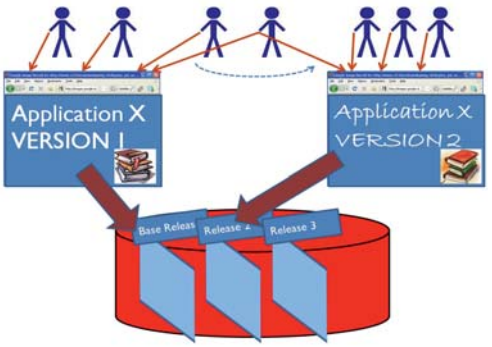
Het package dbms_scheduler kan worden gebruikt om complexe, CRON-of Quartz-achtige job scheduling in te richten.

Parallelle Applicatie Versies

Sommige applicatieservers hebben een speciale voorziening om gedurende een overbruggingsperiode meerdere versies van een applicatie naast elkaar te draaien, zodat een zero-downtime upgrade mogelijk is. Alle nieuwe sessies gebruiken de nieuwe versie van de applicatie, terwijl huidige sessies doorgaan met de huidige versie. De Oracle Database biedt eenzelfde mechanisme, geïntroduceerd in de meest recente release (11gR2) onder de naam Edition Based Redefinition. Met EBR is het mogelijk om meerdere versies van database objecten naast elkaar te laten bestaan, zodat verschillende gebruikersgroepen met verschillende versies van applicatie en database kunnen werken. Een big bang upgrade is dan niet meer nodig. De applicatie moet bij het opbouwen van connecties met de database aangeven welke 'edition' – vergelijkbaar met een release – wordt vereist.

Conclusie

Het eerste deel van deze serie beschreef hoe je met Views en Stored Procedures een interface creëert tussen Java-applicatie en de database. Dit artikel beschrijft met name hoe je binnen die API een geëncapsuleerde implementatie kunt bouwen die optimaal gebruik kan maken van specifieke functies en optimalisaties van de database(s) die je gebruikt. We hebben ondermeer gezien hoe Analytische Functies, Pivot operaties en Flashback Queries veel voorbewerking kunnen doen die de applicatie van veel werk – en de ontwikkelaars van veel programmeren – vrijstelt. De autonome transacties en asynchrone jobs maken het mogelijk de database ook deels onafhankelijk van de applicatie aan het werk kan. Via de database query resultset change notification kan een Java listener door de database op de hoogte worden gebracht van belangrijke wijzigingen. Tenslotte is gedemonstreerd hoe Java applicatie en database ook HTTP als communicatieprotocol kunnen gebruiken; de database kan zowel HTTP requests verwerken en beantwoorden als zelf HTTP requests verzenden.



Databases kunnen uiteraard nog veel meer betekenen voor Java applicaties (en alle andere afnemers van de database). Bijvoorbeeld rondom security en auditing, tracing en monitoring en ook caching – van voorbewerkte gegevens en resultaten van berekeningen, met ondermeer de query en function result cache en met materialized views. Data integriteit is uiteraard een van de kernverantwoordelijkheden van de database – geïmplementeerd met constraints en DML triggers. Ook SQL kan nog meer dan hier beschreven. Denk bijvoorbeeld aan SQL/XML – waarmee de resultaten van SQL Queries als XML documenten kunnen worden opgeleverd en ook XML documenten doorzocht kunnen worden. En aan netwerk queries die ondermeer voor hiërarchische structuren kunnen worden gebruikt die vaak aan tree-componenten in ondermeer web applicaties ten grondslag liggen.

De voornaamste conclusie uit deze twee artikelen zou moeten zijn dat bij het ontwerpen van de architectuur voor Java applicaties en het beslissen over de wijze van te lijf gaan van technische uitdagingen ook de database een onderdeel van de toolset vormt. Waarbij je als een van de stelregels zou kunnen aanhouden: data die niet nodig is op de middle tier moet daar ook niet komen. Dat betekent bijvoorbeeld dat resultaten van aggregaties en sorteringen wel in de applicatie komen maar niet de onderliggende data. Het kan erg waardevol zijn om in het Java-projectteam ook iemand op te nemen met kennis van de huidige mogelijkheden van de database die door de applicatie wordt gebruikt. Met SQL en Stored Procedures kan veel gewonnen worden – maar dat is een vak apart. «

De database vormt wel degelijk een wezenlijk deel van de toolset.

Meer informatie

Meer informatie over de onderwerpen genoemd in dit artikel - en ook de broncode van de voorbeelden - vind je op: http://technology.amis.nl/blog/?p=9182.