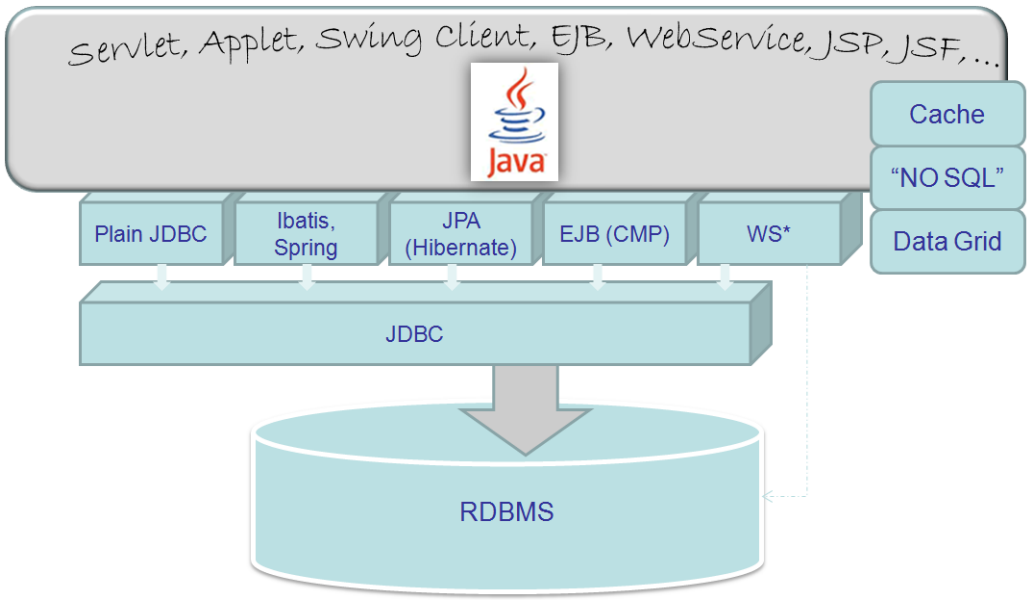
Java ontwikkelaars: zet de Database aan het werk

Tijdens de recente NLJUG JFall conferentie was een van de best bezochte sessies de presentatie 'Java Developers, make the database work for you'. Deze presentatie besprak de toegevoegde waarde die een database kan hebben in een Java applicatie bovenop de pure kaartenbak-functie. De grote interesse voor het verhaal was opmerkelijk, gezien de toch wat moeizame relatie die van oudsher bestaat tussen de werelden van Java en de database. Overigens, de eerlijk gebiedt aan te geven dat pikant genoeg op datzelfde tijdstip een nog beter bezochte sessie de NOSQL hype als onderwerp had.

Dit artikel is het eerste in een serie van twee, die laat zien welke meerwaarde een database kan hebben binnen Java applicaties. Daarbij kijken we ondermeer 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 kijken we naar de historische en actuele verhouding van Java en J(2)EE en de database, data integriteit en constraints en naar stored procedures, database triggers en een paar opmerkelijke SQL functies.

Java en Database

Vrijwel iedere Java (Web) applicatie maakt gebruik van een database. Vaak is dat een relationele database die naast functionaliteit voor query en data opslag nog aanzienlijk meer in zijn mars heeft - en waarvoor niet zelden ook dure licenties zijn betaald. Van oudsher zijn Java ontwikkelaars wat terughoudend in het toepassen van de kracht van de database. Daarvoor zijn verschillende redenen, waaronder aarzelingen over de architectuur, emoties over (vermeende) database (vendor) afhankelijkheid en gebrek aan kennis over wat een database kan betekenen voor de Java applicatie naast de 'kaartenbak' functie. In deze artikelenreeks zullen we zien dat een database een veel grotere rol in Java applicaties kan spelen dan veelal wordt aangenomen.



Het is van belang om een goede balans te vinden in de toepassing van database functies en het uitwerken van oplossingen in Java. Ook moet de aansluiting van de Java tier op de database aan voorwaarden voldoen, zoals encapsulatie van database functionaliteit, zoveel mogelijk ontkoppelde interfaces en toepassing van standaarden. We bespreken hoe specifieke database functionaliteit via JDBC en via JPA (Hibernate en EclipseLink) ontsloten kan worden - op een manier die de applicatie minimaal koppelt aan een bepaald database-product.

Historie en Toekomst

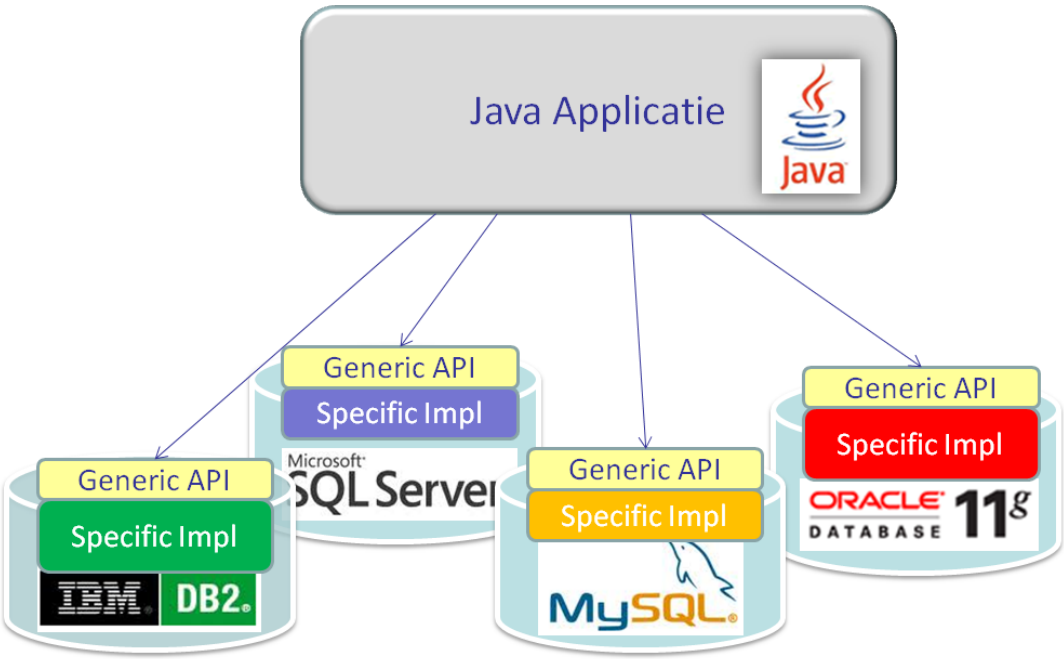
Al sinds Java 1.1 - 1997 - is JDBC de API om vanuit Java applicaties een relationele database te benaderen. Voor vrijwel alle relationele databases zijn sinds 1997 JDBC Drivers ontwikkeld die op database specifieke wijze de JDBC API implementeren. Het gebruik van databases vanuit Java applicaties via JDBC heeft vanaf dat moment een enorme vlucht genomen.

Het programmeren tegen de database bleek overigens ook al snel voor veel ontwikkelaars een uitdaging: JDBC is een tamelijk technische API, kennis van SQL en database specifieke concepten was en is niet altijd even diep geworteld bij Java programmeurs en de werelden van Objecten en Relationele Records pasten ook niet per sé een op een. Al in 1998 verscheen TopLink, het eerste Object-Relational Mapping framework dat de taak van de vertaling van de relationele database wereld naar de wereld van Java objecten en terug op zich nam. Dat was ook de eerste stap in het terugdringen van SQL uit Java applicaties en het vertrouwen op generieke en gegenereerde SQL statements. Met de komst van tientallen andere ORM frameworks (zoals Hibernate vanaf 2001) en de opkomst van Enterprise Java Beans werd die ontwikkeling verder versterkt. TopLinkQL, HQL, EJB QL en JP-QL zijn voorbeelden van object georiënteerde, database onafhankelijke query-talen die door Java ontwikkelaars binnen de

applicatie konden worden ingezet en die door de respectievelijke frameworks werden vertaald in meestal generieke SQL statements die door de onderliggende database werden verwerkt.

Eind jaren negentig was een veelgehoorde kreet 'we moeten database onafhankelijk zijn'. Dat betekende concreet dat een applicatie niet afhankelijk mocht zijn van een specifieke database en haar functionaliteit, maar dat van de database niet meer gevraagd en gebruikt mocht worden dan in iedere andere database - commercieel of open source - ook beschikbaar was. De generieke, gegeneerde SQL statements van de ORM frameworks en EJB paste daar goed bij. Overigens konden de discussies over die aanpak hoog oplopen, tussen Java programmeurs en zeker J2EE architecten aan de ene kant van het strijdperk en DBAs en Database ontwikkelaars aan de andere, niet zelden meer gedreven door passie, emotie, onbegrip en dogmatisch geloof dan door rationele overwegingen.

Zo'n beetje vanaf de publicatie van het fameuze boek van Rod Johnson: "J2EE Development without EJB" kwam er langzamerhand een veel pragmatischer inzicht. Allereerst: kijk eens naar wat een enterprise database allemaal kan (en naar wat die gekost heeft) - je moet wel heel zeker zijn van je zaak om dat te negeren en kostte wat kost database onafhankelijk te blijven. Organisaties stappen zelden over van database - het is niet bepaald agile (die term werd naar in 2002 nog niet bij gebruikt) om nu een verlies aan productiviteit en performance te nemen voor een vorm van flexibiliteit die vrijwel zeker nooit nodig zal gaan. Daarnaast: hoeveel applicaties moeten voor meer dan één database geschikt zijn? Dat zijn er eigenlijk maar heel weinig. En bovendien: je kunt best een applicatie maken die portable is tussen verschillende databases maar die wel van elke database de specifieke eigenschappen kan benutten.



Dit doe je door een generieke API te definiëren waartegen de applicatie aanpraat. De API wordt uitgedrukt in Views en Stored Procedures. Voor iedere database wordt een specifieke implementatie gemaakt van die API, waarin database ontwikkelaars volledig los kunnen gaan om alle toeters en bellen van hun database te benutten om een zo krachtig mogelijke en zo goed mogelijk performende API

implementatie op te leveren. Exact dezelfde applicatie wordt vervolgens voor iedere database met de specifieke implementatie voor die database uitgeleverd.

De voorzieningen in het Spring Framework (Spring JDBC), in Hibernate en Toplink/EclipseLink en het meest recent ook in JPA bieden steeds meer gelegenheid om binnen een standaard aanpak - die gebruikt wordt over alle databases heen - toch gebruik te maken van database specifieke functionaliteit. Veel belangrijker nog is dat de geesten er rijp voor zijn - de zwart/wit tegenstelling van eind jaren '90 en daarna en de religieuze debatten zijn gelukkig goeddeels achtergelaten.

De combinatie van deze veel pragmatischer houding ten opzichte van de database en de faciliteiten binnen frameworks maakt dat er nu weer veel meer ruimte is om de kracht van de database daadwerkelijk te benutten vanuit Java applicaties. Nu wordt de vraag: waar ligt die kracht precies, en hoe benut je hem. En dat is precies de vraag waar dit artikel verder op ingaat.

Oracle RDBMS als voorbeeld

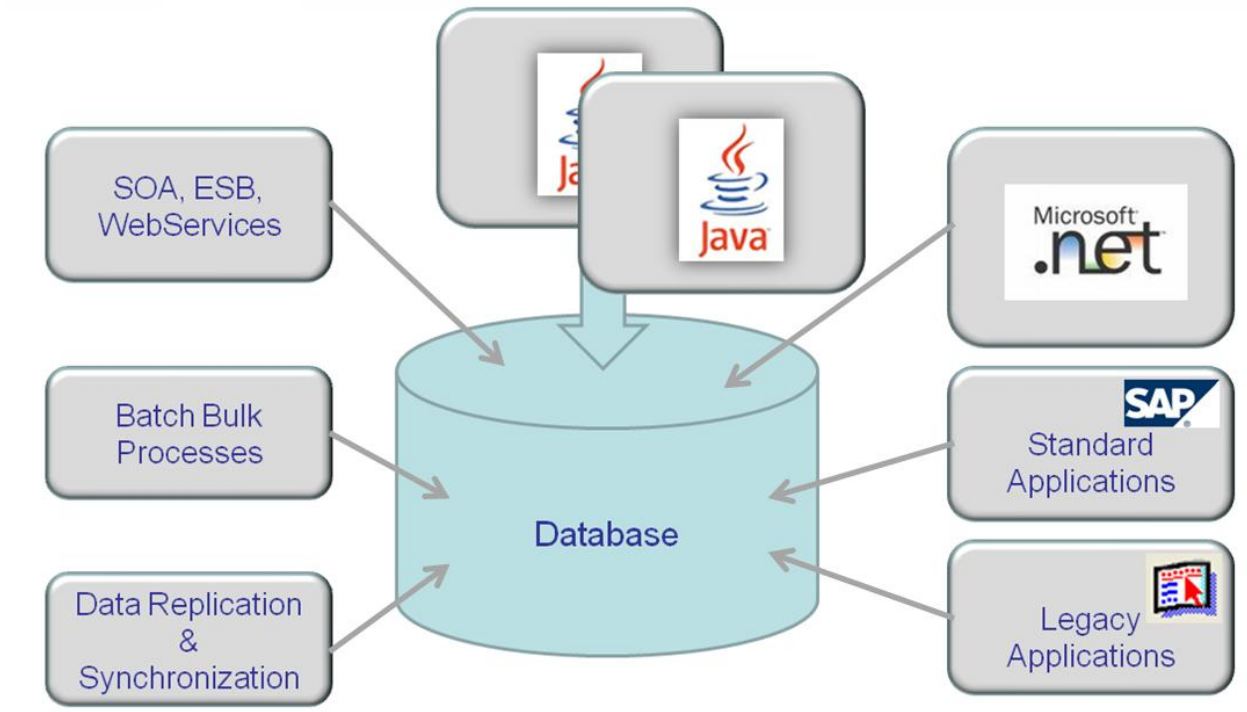
Als je in een artikel als dit gaat spreken over specifieke functionaliteit van databases en je wilt ook code laten zien, is het handig om één specifieke database als voorbeeld te kiezen. De meeste onderwerpen die we bespreken gelden in meer of mindere mate met alle veel gebruikte databases, dus de keuze voor het concrete voorbeeld maakt niet zo heel veel uit. Aangezien de Oracle RDBMS een van de meest toegepaste is, deze de meeste database functionaliteit die ik wil beschrijven ook bevat (en steeds meer ook nog volgens de ANSI/SQL standaarden) en bovendien dit de database is die ik zelf het beste ken, heb ik de concrete code voorbeelden op basis van Oracle RDBMS ontwikkeld. Echter, in vrijwel iedere database zijn op de meeste punten vergelijkbare voorbeelden te creëren.

Integriteit en Data Constraints

Een essentieel element in iedere database is de bewaking van de data integriteit ofwel de regels of constraints waaraan de data ten alle tijde moet voldoen om geldig of betrouwbaar te zijn. Voorbeelden van constraints zijn ondermeer de 'primary key' ofwel de unieke identificatie van ieder database record, de 'unique key' - een combinatie van velden waarvan dezelfde verzameling waarden slechts een keer mag voorkomen (en die dus ook identificerend is) en de 'foreign key' die beschrijft hoe een record een verwijzing bevat naar - de primaire of andere unieke key van - een ander gegeven in de datatabase.

Er lijkt weinig discussie over de vraag dat deze basis-constraints door de database zouden moeten worden afgedwongen. Niet alleen is de database geoptimaliseerd voor de implementatie van deze constraints - met ondermeer indexen en snelle algoritmes - ook moeten we bij het nadenken over de integriteit ons realiseren dat heel vaak onze (Java) applicatie niet de enige gebruiker is van de data in de database. Ook andere systemen en applicaties kunnen de data benaderen en potentieel wijzigen. De database is in al die gevallen de laatste verdedigingslinie, dus moet de integriteit op dat niveau worden afgedwongen. Daar komt overigens bij dat een efficiënte en echt veilige implementatie van deze regels

op de middle tier eigenlijk niet mogelijk is: er zal altijd sprake zijn van extra netwerk verkeer, data vasthouden op de middle tier die daar niet echt nodig is en ingewikkelde cross-sessie en cross-tier locking mechanisme. Het nabouwen van essentiële database functies lijkt weinig zinvol.



Als we de discussie uitbreiden naar andere, minder kern-achtige data -regels dan is de vanzelfsprekendheid over de database als bewaker niet altijd even groot. Voor zogenaamde tuple-rules die regels stellen binnen een record - zoals *begin datum moet voor eind datum liggen* of *de prijs moet afgerond zijn op 5 cent* - of entity en inter-entity rules die over records een de constraints definiëren - bijvoorbeeld *producten in order-regels binnen een order met spoed-status mogen niet breekbaar zijn* of *een reisgezelschap per vliegtuig moet minimaal twee volwassenen bevatten* - kennen databases beperkte declaratieve ondersteuning. Zogenaamde Check Constraints kunnen de declaratieve controle voor de meeste tuple-rules doen.

```
ALTER TABLE PROJECTS ADD CHECK CONSTRAINT (start_date < end_date)
```

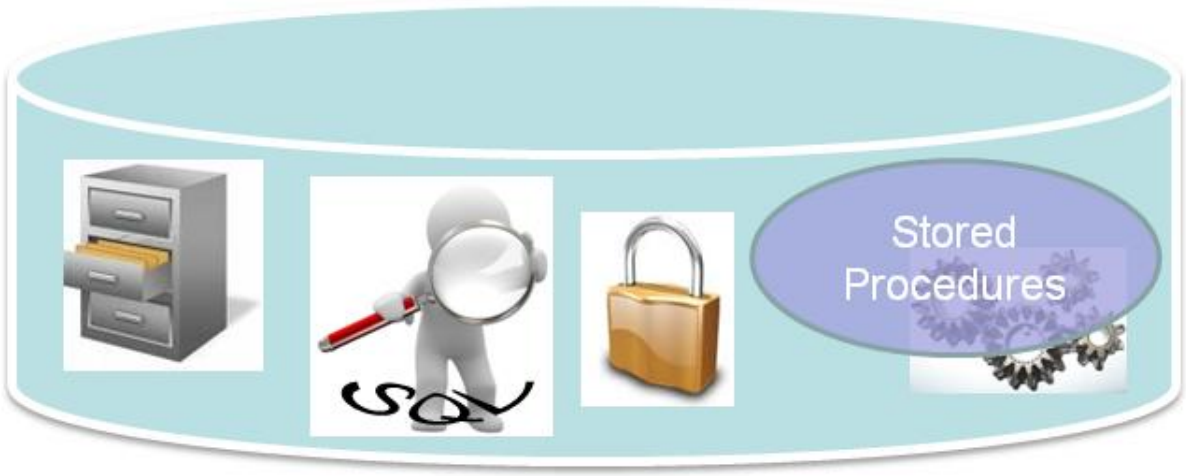
Het afdwingen van entity en inter-entity regels vereist ook in de database programma code - meestal in de vorm van database triggers en stored procedures (zie volgende twee secties).

In veel gevallen gaat de discussie er overigens ten onrechte over of dit soort data specifieke business rules in de middle tier óf door de database moeten worden afgedwongen. Waar het gaat om de integriteit van data - die mogelijk van verschillende kanten en dus niet alleen via de Java applicatie wordt gemanipuleerd - is implementatie in de database noodzakelijk. Daarnaast is het met het oog op (snelheid van) user feedback en mogelijke ontlasting van de database vaak wenselijk om een subset van die regels ook in de client of de middle tier te controleren. Daarnaast is het belangrijk om in geval van

overtredingen van data regels zinvolle foutboodschappen uit de database over te dragen aan de Java applicatie.

Stored Procedures

De meeste database kunnen naast de basisvoorzieningen - dataopslag, SQL query en security- ook programma code uitvoeren. Deze wordt geschreven in een database specifieke procedurele taal (PL/SQL in Oracle, Transact-SQL in SQL Server, SQL/PSM en SQL/PL in DB2, SPL in Informix, pl/perl en pl/php in PostgreSQL and MySQL Stored Procedure Language. Bovendien ondersteunen external procedures die worden aangeroepen als een stored procedure maar buiten de database in bijvoorbeeld C, C# of Java zijn geïmplementeerd. Tenslotte ondersteunen diverse databases ook Java Stored Procedures.



Stored procedures worden gebruikt voor operaties die complexere, meer procedurele operaties uitvoeren op grote sets data dan in SQL statements mogelijk is - maar waarvoor de data niet op de middle tier aanwezig is (of hoeft te zijn). Typische toepassingen van stored procedures zijn data intensieve berekeningen, validaties en data bewerkingen zoals bulk kopieer- en update-acties. Overigens zullen we later leren dat SQL door de jaren heen zo'n krachtige taal is geworden dat veel van de vroegere redenen om stored procedures te gebruiken niet meer van toepassing zijn. Een algemene stelregel is dat als het in SQL kan, dan moet je het vooral niet in een Stored Procedure doen. Hoewel stored procedures dicht bij de data worden uitgevoerd - in min of meer dezelfde memory en process space - is er toch meer overhead dan in SQL om data te benaderen.

Stored procedures kennen input en output parameters, maar geen return waarde (als een methode die void aangeeft). Output parameters kunnen binnen de procedure gewijzigd worden. Een speciaal soort output parameter wordt gebruikt in geval van een stored function, een object dat wel een return waarde heeft.

JDBC kent het CallableStatement als vehikel om stored procedure rechtstreeks aan te roepen. ALs alternatief kunnen stored functions ook worden aangeroepen via SQL queries zoals deze:

```
select my_stored_function( :param1, :param2)
```

```
from dual
```

NB: dual in de Oracle RDBMS is een onechte tabel die wordt gebruikt om systeem parameters of de uitkomsten van stored functies op te vragen; het equivalent in DB2 heet sysibm.sysdummy1 en SQL Server kan een select uitvoeren zonder dat een tabelnaam wordt opgegeven. Overigens kan een stored function ook onderdeel uitmaken van queries tegen echte tabellen of views.

JPA - de Java Persistence API - kent nog geen formele ondersteuning voor Stored Procedures of Stored Functions. Dit staat gepland voor release 2.1 - onderdeel van JEE 7. Stored Functions kunnen aangeroepen worden via JPA door gebruik te maken van Native Queries die ook de waarde van een Stored Function opvragen. Door een SqlResultSetMapping te definiëren kan de native query ook JPA entities opleveren of bijwerken.

Een native query in JPA mag een ResultSet teruggeven maar ook *niets*. Een stored procedure die geen result set oplevert of andere output parameters definieert kan als volgt worden aangeroepen:

```
Query query = entityManager.createNativeQuery("BEGIN PREPARE_REPORT(P_DEPARTMENT_ID=>?); END;");

query.setParameter(1, departmentId);

query.executeUpdate();
```

JPA implementaties als Hibernate - via

```
getSession().doWork( new Work(){public void execute (final java.sql.Connection
connection){...}} )
```

en EclipseLink - via de NamedStoredProcedureQuery en StoredProcedureParameter annotaties:

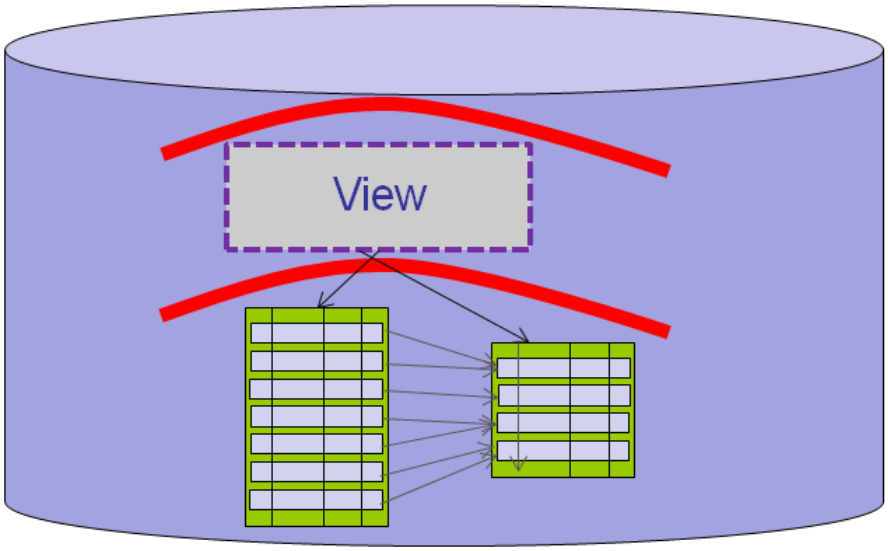
```
@Entity

@Table(name="ORDERS")

@NamedStoredProcedureQuery(

    name="RetrieveOrder",

    procedureName="Read_Order",

    parameters={

        @StoredProcedureParameter(queryParameter="ORD_ID")}

)

public class Order implements Serializable{
```

geven nog meer specifieke aanknopingspunten om vanuit een JPA gebaseerde applicatie toch direct met Stored Procedures en Functions te communiceren. NB: deze aanknopingspunten maken je applicatie afhankelijk van de specifieke JPA implementatie!

Tenslotte is er nog een truc - die een tikje vooruit loopt op de volgende sectie: als een JPA entity gemapped is naar een View in plaats van een tabel, en die View is voorzien van een Instead Of trigger - dan wordt voor een DML actie tegen de view - een persist, merge of delete op de entity - in werkelijkheid een Database Trigger aangeroepen die een Stored Procedure kan uitvoeren.

Views

Data in een database ligt vast in een tabel. Een view ziet er voor applicaties net zo uit als een tabel. Een view bevat echter geen data - maar is in feite niets meer dan een opgeslagen SQL statement waarvan het resultaat als een soort tabel kan worden gelezen.



Het belang van views voor onze discussie is potentieel heel groot: een view kan de meest ingewikkelde, database specifieke SQL constructies bevatten:

```
create or replace view special_orders

as

with yesterdays_order_totals

as ( select avg(price) , product

     from   orders as of timestamp (sysdate - 1)

     group

     by     rollup(product)

)

select case product

       when 'STOFZUIGER' then 'INTERIEURVERZORGER'

       else initcap(product)
```

```
        end "product label"
,       avg(price) over (partition by category)  avg_cat_price
,       price
,       ( select clg.price from catalog clg where clg.product = o.product) catalog_price
from    orders o
        left outer join
        yesterdays_order_totals yo
        on (o.product = yo.product)
```

en toch vanuit onze Java applicatie met een simpele select * from <view> worden aangesproken. Met andere woorden: zonder dat er (complexe, database afhankelijke) SQL in onze applicatie binnenkomt, kunnen we wel optimaal gebruik maken van wat onze database te bieden heeft aan query faciliteiten. Dit geldt in het bijzonder voor read-only views, maar we zullen in de volgende sectie lezen dat instead-of triggers ervoor kunnen worden dat ook voor updateable views dit verhaal opgaat.

Database Triggers - AOP op Data Manipulatie (DML)

Een database trigger is een soort stored procedure die wordt gekoppeld aan een tabel. In de definitie van trigger is vastgelegd voor welke data manipulaties - insert, update of delete - de code van de trigger moet worden uitgevoerd, en of dat voor of na de daadwerkelijke manipulatie van de tabel-rij moet gebeuren. Triggers zijn op deze manier een soort aspects die via pointcuts - before of after voor insert, update or delete - aan tabellen worden gekoppeld - onzichtbaar voor de aanroeper van de DML operatie.

Triggers worden gebruikt om verschillende redenen: zetten van default waarden na een insert , afleiden of berekenen van waarden na een insert of update (in de rij wordt gemanipuleerd of in een heel ander record), verzenden van een notificatie (event) aan geïnteresseerde derden en de uitvoering van complexe validaties.

Als een trigger tijdens een insert of update wijzigingen aanbrengt in de data zoals die vanuit de Java applicatie aan de database zijn doorgegeven, is de data in de applicatie niet meer gesynchroniseerd met de database. Dat is een ongewenste situatie. JPA heeft daar zelf niet direct een oplossing voor - maar zowel Hibernate als EclipseLink ondersteunen annotaties die de JPA implementatie instrueren om direct na een persist of merge ook een refresh van de geannoteerde attributen uit te voeren:

```
@Entity
@Table(name = "EMP")
```

```
public class Employee
…
@ReturnInsert //EclipseLink

@Generated (value= GenerationTime.INSERT)
            // Hibernate

@Column(name="sal")

private Double salary
```

In dit geval veronderstellen we een Insert trigger op tabel EMP die de waarde van het salaris kan manipuleren. Door de annotaties wordt afgedwongen dat na afloop van de transactie de waarde van het attribuut salary in de entity weer synchroon is met de waarde in database. NB: veel databases ondersteunen een soort korte variant voor het afleiden van default waarden voor een kolom: in de definitie van de kolom wordt de default waarde gespecificeerd, als een constante of als een systeem variable (zoals huidige tijd, huidige user). De annotaties die hierboven zijn getoond zorgen ook voor het verversen van een attribuut dat op deze wijze een waarde krijgt opgelegd tijdens de JPA persist-actie.

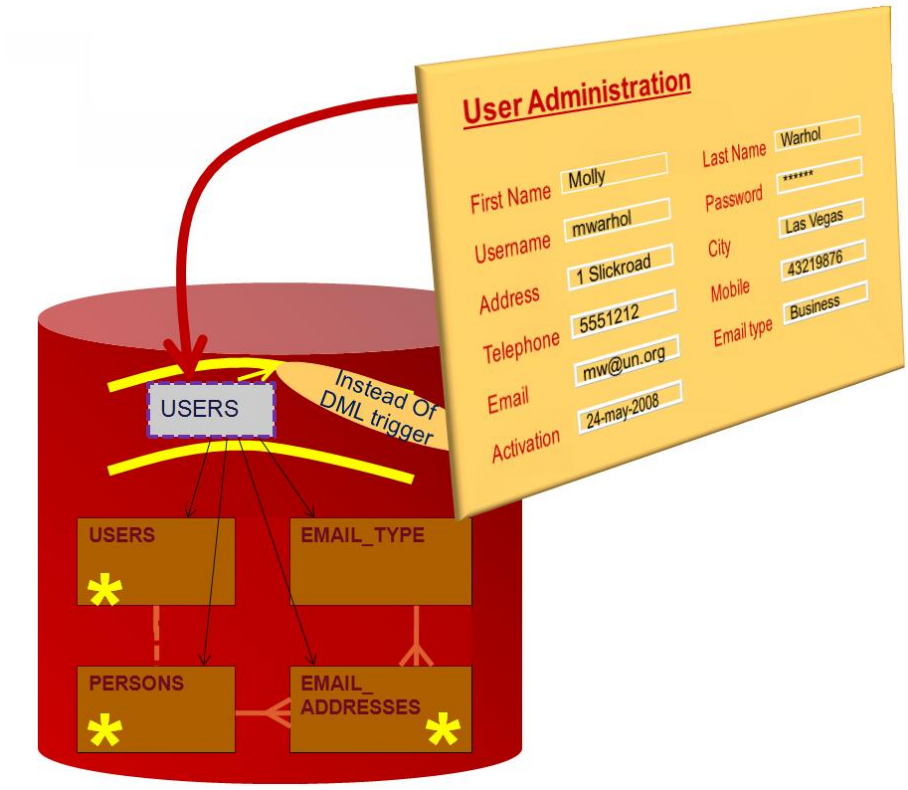
```
CREATE TABLE Persons
(
  ...

  , country varchar2(2) DEFAULT 'nl'
)
```

Met de combinatie van Table Triggers en Stored Procedures kan de implementatie van entity en inter entity (cross table) business rules worden bereikt. Bijvoorbeeld de eerder genoemde regel *een reisgezelschap per vliegtuig moet minimaal twee volwassenen bevatten* wordt geïmplementeerd met triggers die afgaan wanneer het reisgezelschap wordt gecreëerd of van vervoermiddel wijzigt en wanneer er leden van het gezelschap worden verwijderd of wanneer de geboortedatum van een van de leden wordt gewijzigd. Elk van die triggers zou dezelfde stored procedure kunnen aanroepen om de geldigheid van de data ten opzichte van de regel te herbevestigen. NB: de validatie zou niet na iedere data manipulatie moeten plaatsvinden, maar alleen als de transactie logisch compleet is en afgesloten gaat worden.

Sommige databases bieden naast triggers op data manipulatie ook triggers op andere events. Bijvoorbeeld in de Oracle database kan trigger worden gedefinieerd op het LOGON event - waarbij direct na het initialiseren van een sessie al context parameters kunnen worden gezet of caches geinitialiseerd - en op DDL events, zoals creatie, drop en wijziging van tabellen of andere database objecten.

Een heel speciaal soort trigger is de Instead Of trigger, aanwezig in ondermeer SQL Server, DB2 en Oracle. Dit type trigger kan gedefinieerd worden op een view of tabel en dirigeert de database in het geval van een data manipulatie tegen de tabel of view naar een stored procedure. De database gaat geen poging doen om de DML operatie uit te voeren maar laat dat helemaal over aan de instead of trigger. Deze kan zelf een DML operatie uitvoeren - maar dat hoeft niet. En deze kan manipulaties uitvoeren in een of meerdere tabellen.

De instead of trigger is een van de meest krachtige instrumenten die database bieden om via POSQL (plain old SQL) ongemerkt te communiceren met Stored Procedures en daarmee een volledige ontkoppeling te bereiken van het onderliggende datamodel.



De illustratie toont een data model bestaand uit vier tabellen. Een Java Applicatie maakt via een view - USERS - gebruik van deze tabellen. Een JPA *persist* (creatie) vanuit de applicatie wordt door de Instead Of trigger vertaald in INSERT statements in vier verschillende tabellen. Dankzij de view met instead of trigger heeft de applicatie geen last van het legacy-datamodel.

SQL

De motor van de database is de SQL engine. Het opvragen van data is waarschijnlijk de meest frequente interactie van Java applicaties met de database, en SQL is uiteindelijk het mechanisme waarmee dat gebeurt. Veel van de SQL in Java-Database interactie wordt gegenereerd door frameworks, soms - maar lang niet altijd - op database specifieke wijze. Door SQL queries te verpakken in views of stored procedures kan vaak beter van de speciale voorzieningen in SQL gebruik worden gemaakt.

Uit performance oogpunt willen we bijvoorbeeld het aantal aanroepen vanuit de applicatie naar de database beperkt houden en de hoeveelheid data die wordt overgeheveld naar de Java applicatie beperken tot dat echt nodig is. Het volgende aggregatie statement toont een voorbeeld van de bijdrage die SQL daar kan leveren. Deze query geeft een aggregatie van alle verkopen - per maand, per kwartaal per jaar en over 'all time':

```
select  sum(o.totaal_bedrag) omzet
,       extract(month from o.verkoopdatum) maand
,       'Q'||to_char(o.verkoopdatum,'Q') kwartaal
,       extract(year from o.verkoopdatum) jaar
from    orders o
group
by      rollup(
          extract(year from o.verkoopdatum)
        , 'Q'||to_char(o.verkoopdatum,'Q')
        , extract(month from o.verkoopdatum)
        )
```

de *rollup* operator die in de group by wordt gebruikt instrueert de database om de aggregatie achtereenvolgens te bepalen over iedere unieke combinatie van jaar, kwartaal en maand, over jaar en kwartaal, per jaar en over alle jaren heen. In één round trip en met minimaal dataverkeer levert deze query totalen op diverse aggregatieniveaus - een heel efficiënte wijze om bijvoorbeeld data voor een rapportage ter vergaren.

Conclusie

Dit artikel doet een poging om te beargumenteren dat in veel gevallen een database een belangrijke bijdrage kan leveren binnen de architectuur van een Java applicatie. Het goed toepassen van de sterke kanten van een database kan leiden tot een applicatie die beter performt, eenvoudiger te ontwikkelen en rijker in functionaliteit is. Sterke kanten van de database - naast zijn persistente karakter - zijn ondermeer bewaking van data integriteit, geavanceerde zoekopdrachten, joinen van tabellen en benutten van indexen, aggregatie en sortering en complexe bulk data manipulatie.

Belangrijke uitgangspunten zijn dat er geen data naar de middle tier zou moeten gaan die daar niet strikt noodzakelijk is. Bijvoorbeeld het berekenen van aggregaties over data die zelf niet in de Java applicatie gebruikt wordt zou altijd door de database moeten worden gedaan. Ook moet de database zich bij zijn leest houden en geen grafische, applicatie of user interface gerichte zaken voor zijn rekening gaan houden. Een belangrijke overweging is ook dat de database in de meeste gevallen door meer partijen dan alleen de Java applicatie wordt benaderd. Bij de implementatie van data constraints maar ook bij andere implementatiekeuzes moet dat steeds meegewogen worden.

Dit artikel beschrijft constraints, stored procedures, database triggers, views en een snippertje SQL - features die in alle mainstream relationele databases beschikbaar zijn. Met deze mechanismes kan een generieke API gedefinieerd worden die op database specifieke wijze wordt geïmplementeerd.

In het volgende artikel gaan we dieper in de op die database specifieke karakteristieken. We kijken dan bijvoorbeeld naar exotische features die SQL ons biedt - zoals Analytische Functies, Pivot en Flashback Queries. Daarnaast kijken we ondermeer naar gebruik van cursors, Database Change Notifications, HTTP als alternatief voor JDBC, security, asynchrone operaties en autonome transacties.

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