TRANSAKCIJE I ORACLE - BAZA, FORMS, ADF

Zlatko Sirotić
Istra informatički inženjering d.o.o., Pula
e-mail: zlatko.sirotic@iii.hr

SAŽETAK

Često se transakcije u Oracle bazi, te u alatima Forms i ADF, koriste na "standardan", tj. uobičajeni način. Uglavnom je tako i najbolje. No ima trenutaka kada je bolje posegnuti za nekim rješenjem koje nije "u glavnom toku". U radu će biti prikazani standardni i neki nestandardni načini upravljanja transakcijama u Oracle bazi, Forms-ima i ADF-u.

Transactions in Oracle Database, Forms and ADF, are often used in a "standard", common way. In general, that is the best way to use them. But, there are cases when it is better to find a solution that is not "in the book". Throughout the paper I will be demonstrating some standard and non-standard ways of handling transactions in the Oracle database, Forms and ADF.

UVOD

Ispravno rukovanje transakcijama u bazama podataka vrlo je bitno za poslovne aplikacije.

No, treba napomenuti da razumijevanje transakcija postaje sve važnije, jer su se one "izvukle" iz okvira (samo) baza podataka i "ušle" u programske jezike (bilo direktno, bilo indirektno, kroz odgovarajuće library-e), pa i u hardver, u vidu softverskih, hardverskih i hibridnih transakcijskih memorija (STM, HTM i hibridni TM) - vidjeti npr. [5] i [6]. Npr. nova generacija Intelovih procesora (Haswell) ima hardversku transakcijsku memoriju.

Transakcije su tako postale povezane sa (vjerojatno) najvećim današnjim problemom softverskog inženjerstva (iako se o njemu u javnosti puno manje priča nego npr. o Cloud Computingu, Big Data, SOA arhitekturi i dr.) – kako pisati pouzdane konkurentne i paralelne programe.

U radu su prikazane različite teme vezane za upravljanje transakcijama, kroz tri poglavlja.

U prvom poglavlju prikazane su teme vezane za transakcije i Oracle bazu.

Prikazane su "standardne" teme: Osnove arhitekture Oracle DBMS-a (vezano za transakcije); Transakcije općenito; Transakcije i zaključavanje; Nove mogućnosti u bazi 12c (vezano za transakcije).

Nešto teže teme su: Distribuirane transakcije; Rješavanje mutiranja okidača baze.

Prikazane su i specijalne teme (naša rješenja): Simulacija COMMIT okidača pomoću odgođenih deklarativnih integritetnih ograničenja; Simulacija INSERT WAIT naredbe; Simulacija ROLLBACK TO SAVEPOINT naredbe u okidaču baze; Kako generirati dokumente bez rupa u brojevima.

U drugom poglavlju prikazane su teme vezane za transakcije i Oracle Forms.

Prikazane su "standardne" teme: Forms - razvoj, varijante, arhitektura; Neka svojstva Forms modula i Forms bloka (vezano za transakcije); Svojstva tipičnih vrsta tekstualnih polja (text item); Transakcijski i validacijski status Forms objekata; Master-detalj relacije.

Nešto teže teme su: COMMIT i POST Forms procesi; Različiti načini poziva Forms modula.

Prikazane su i specijalne teme (naša rješenja): Template i library za POST-iranje kod relacije master–detalj; Forms i odgođena deklarativna integritetna ograničenja na bazi.

U trećem poglavlju prikazane su teme vezane za transakcije i Oracle ADF (ADF BC i Task Flow).

Prikazane su "standardne" teme: ADF - razvoj i arhitektura; Entity Object; View Object; Application Module; Kako pomiriti HTTP stateless protokol i statefull zahtjeve.

Nešto teže teme su: Application Module Pooling; Task Flow i transakcije.

Budući da je tema o transakcijama povezana i s drugim temama, ponekad će ukratko biti prikazane i neke stvari koje nisu direktno vezane uz transakcije.

1. TRANSAKCIJE I ORACLE BAZA

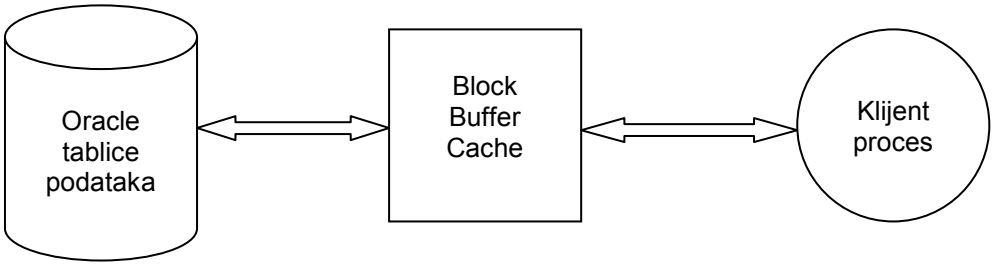
1.1. Osnove arhitekture Oracle DBMS-a (vezano za transakcije)

Oracle sustav čine dva glavna dijela:

- baza podataka (BP), koju čine različite vrste datoteka; najvažnije su datoteke podataka, no postoji i desetak drugih vrsta datoteka, od kojih su posebno važne Redo Log datoteke, koje se dijele na Online i Archive; Online Redo Log datoteke služe (između ostalog) za oporavak baze u slučaju pada sustava (npr. programske greške ili nestanaka struje), dok Archive Redo Log datoteke služe za uspostavu prijašnjeg stanja, zajedno sa backup datotekama (na trakama ili diskovima) u slučaju kvara medija (u pravilu – diska);

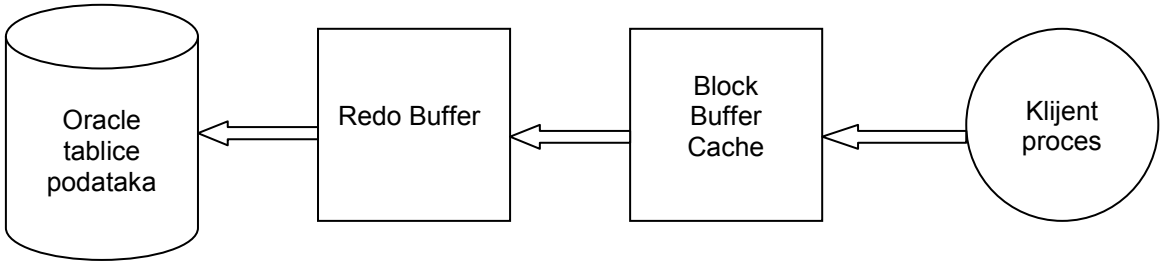
- instanca (jedna ili više) baze podataka, koju čine memorijske strukture i procesi u memoriji; od memorijskih struktura, naročito su zanimljivi Block Buffer Cache i Redo Bufer.

Na koji način radi čitanje (SELECT) i izmjena podataka (INSERT / UPDATE / DELETE) u Oracle sustavu? Klijentski procesi (klijentski proces može biti proces na drugom računalu, proces na istom računalu na kojem se nalazi i Oracle DBMS, ali može biti i neki proces unutar Oracle DBMS-a) nikad ne dobivaju podatke direktno sa diska. Svi podaci koji se čitaju sa diska smještaju se u Block Buffer Cache. Također, kada klijentski proces mijenja podatke, ne mijenja ih direktno na disku, već se promjene prvo spremaju u Block Buffer Cache, kako to prikazuje slika 1.1.



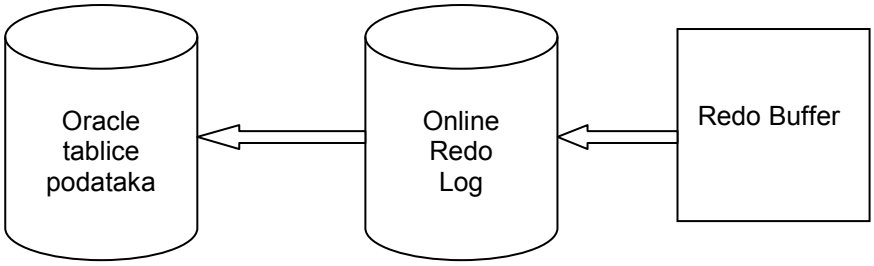
Slika 1.1. Klijentski procesi uvijek čitaju / mijenjaju podatke preko Block Buffer Cache-a

Zapravo, mijenjanje podataka ne ide tako da se podaci direktno iz Block Buffer Cache-a upisuju na disk, već se upis radi preko Redo Buffer-a, kako prikazuje slika 1.2.



Slika 1.2. Izmjena podataka radi se kroz Redo Buffer

Osim toga, mijenjanje podataka ne ide tako da se podaci odmah upisuju u tablice podataka, već se iz Redo Buffer-a prvo upisuju u Online Redo Log datoteku, kako prikazuje slika 1.3.

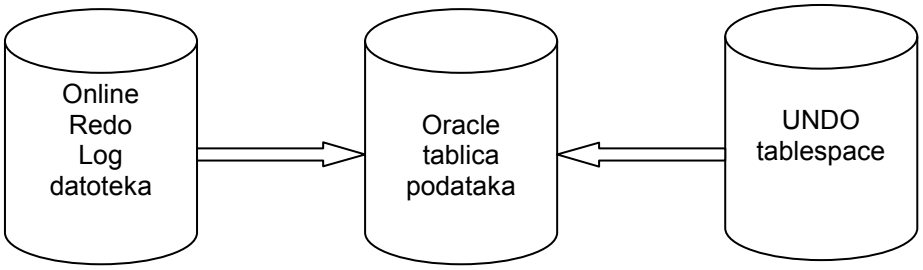


Slika 1.3. Podaci se na disk prvo zapisuju u Online Redo Log

Online Redo Log čine minimalno dvije datoteke (a preporučljivo je da postoje barem tri) koje se koriste u krug – kada se prva napuni, Oracle počinje pisati u drugu, a kada se druga napuni Oracle se vraća na prvu. Kada su podaci zapisani u Online Redo Log može doći do pada sustava (to nije kvar medija) prije nego Oracle te podatke upiše u prave tablice podataka. No nakon što se Oracle sustav ponovno podigne, pročitat će Online Redo Log i upisati podatke u prave tablice. Naravno, pitanje je da li ti podaci zaista trebaju biti trajno zapisani u tablicu podataka – to ovisi o tome da li su oni COMMIT-irani.

Moglo bi se pomisliti da Oracle sve podatke koji nisu COMMIT-irani drži u memoriji, u Block Buffer Cache-u, a da se tek kod COMMIT-a svi podaci prepisuju na disk. No to je nemoguće, jer Block Buffer Cache, koliko god bio velik, ne može uvijek biti dovoljno velik da svi mijenjani podaci stanu u njega. Stoga se podaci iz Block Buffer Cache-a često upisuju u tablice podataka (posredstvom Redo Buffer-a i Online Redo Log-a) prije nego je transakcija COMMIT-irana. Stoga se nakon pada sustava, i nakon što Oracle pročita Online Redo Log i upiše podatke (koji još nisu upisani) u prave tablice, tj. nakon faze koju bismo mogli nazvati REDO fazom, zbiva UNDO faza, u kojoj se oni podaci koji nisu COMMIT-irani brišu iz Oracle tablica podataka (slika 1.4.). Takav postupak, da se prvo radi REDO, a onda UNDO faza, naziva se ARIES (Algorithms for Recovery and Isolation Exploiting Semantics) [3].

Može se postaviti pitanje – otkuda Oracle uzima stare podatke, koji su mu potrebni da napravi UNDO fazu? Moglo bi se pomisliti da se ti podaci nalaze u Online Redo Log datoteci, tj. da ona čuva sliku redaka kakvi su bili prije promjene (pa bi se ta slika koristila za UNDO) i sliku redaka nakon promjene (pa bi se ta slika koristila za REDO). U stvarnosti Online Redo Log, kako mu i samo ime kaže, sadrži samo sliku za REDO. Podaci potrebni za UNDO nalaze se u jednom posebnom prostoru na disku, koji se naziva UNDO tablespace (naravno, postoje i tablespaceovi za standardne tablice podataka). Oracle zapisuje staro stanje redaka u UNDO tablespace uvijek kada radi izmjenu podataka (INSERT / UPDATE / DELETE).



Slika 1.4. Nakon pada sustava, u Oracle tablice podataka
prvo se dodaju nedostajući podaci iz Online Redo Log-a,
a onda se za ne-COMMIT-irane transakcije vraćaju podaci iz UNDO tablespacea

UNDO tablespace, osim što služi za eliminiranje (iz tablica podataka) promjena koje nisu COMMIT-irane, kod oporavka sustava nakon pada, služi i za omogućavanje višekorisničkog rada. Naime, jedan od važnijih zahtjeva na RDBMS sustav je da jedna sesija (baze podataka) ne vidi promjene koje je napravila druga sesija, sve dok ta druga sesija ne napravi COMMIT. No budući da se često ne-COMMIT-irani podaci moraju spremati na disk, pitanje je otkuda prva sesija može čitati stare podatke (prije promjene). Oni nisu u tablici podataka (tamo su promijenjeni podaci), nisu niti u Online Redo Log datoteci – oni se nalaze u UNDO tablespaceu. Oracle ih iz UNDO tablespacea čita u Block Buffer Cache, jer klijentski proces sve podatke (pa i stare) čita iz Block Buffer Cache-a, a ne direktno sa diska.

Postojanje UNDO tablespacea (ili neke slične strukture u nekom drugom RDBMS sustavu) omogućava da mijenjanje podataka ne utječe na čitanje podataka, tj. mijenjanje podataka ne sprečava da se ti podaci istovremeno i čitaju (zapravo, čitaju se stara stanja tih podataka). U RDBMS sustavima koji nemaju nešto slično kao što je UNDO tablespace, mijenjanje podataka utječe na čitanje, tj. sesija koja mijenja podatke postavlja ključ nad tim mijenjanim redovima, taj ključ onda sprečava druge sesije čak i da čitaju podatke, čime dolazi do znatnog usporavanja korisničkog rada.

Osim UNDO tablespacea i Online Redo Log datoteka, postoje i Archive Redo Log datoteke. One, za razliku od UNDO tablespacea i Online Redo Log datoteka nisu obavezne, tj. Oracle sustav može raditi i bez njih. No Archive Redo Log datoteke su, uz backup datoteke, praktični nužne za oporavak u slučaju kvara medija (diska). Archive Redo Log datoteke dobivaju se kopiranjem napunjenih Online Redo Log datoteka. Naravno, Archive Redo Log datoteke se ne koriste u krug, tako da za njihovo spremanje treba rezervirati dovoljno mjesta na nekom mediju. To može biti traka, ali danas je najpogodnije (zbog relativno niskih cijena diskova) da to budu diskovi - naravno, ne oni isti na koje se spremaju tablice podataka (jer ako se pokvare ti diskovi, izgubili smo oboje).

Napominjemo da je prikaz arhitekture baze podataka vrlo općenit, bez puno detalja. Preporučamo čitanje izvrsne knjige Toma Kytea [7].

1.2. Transakcije općenito

Kod baza podataka, važni su pojmovi konekcija, sesija i transakcija baze podataka. Moramo napomenuti da se ti pojmovi ponekad brkaju, vjerojatno zbog toga što se na aplikacijskoj strani (npr. kod aplikacijskih servera) često koriste isti pojmovi, ali ne znače uvijek isto što i kod DBMS-a. Npr. u web programiranju, korisnička (user) sesija (neki kažu i aplikacijska sesija) nije isto što i sesija baze. Na bazi, jednu sesiju čini jedna ili više slijednih transakcija, a transakcija uvijek pripada jednoj sesiji (izuzetak su distribuirane transakcije, gdje jednu transakciju realizira više sesije, koje pripadaju različitim bazama podataka). Konekcija na bazu je (pojednostavljeno rečeno) veza između klijentskog programa (to može biti i program na aplikacijskom serveru) i baze, a kroz jednu konekciju može u određenom trenutku ići nula, jedna ili više sesija (baze). Najčešće kroz jednu konekciju ide jedna sesija (baze), pa se često pojmovi konekcija (na bazu) i sesija (baze) poistovjećuju. No npr. Oracle Forms koristi mogućnost da više sesija (baze) ide kroz samo jednu konekciju.

Možemo reći da su ovo neke najvažnije osobine Oracle transakcije (u ne-distribuiranom slučaju):

1. Sesija (baze) ne vidi promjene koje je napravila druga sesija, dok druga sesija ne napravi COMMIT (ili ROLLBACK). Međutim, sesije nisu nezavisne, jer zaključavanje redaka u jednoj sesiji utječe na drugu sesiju koja pokušava ažurirati redak koji je zaključala prva sesija.

2. Kada sesija izvršava DML naredbu (INSERT / UPDATE / DELETE), automatski se zaključa redak. Redak se može otključati tek na kraju transakcije (COMMIT ili ROLLBACK), ili pomoću ROLLBACK TO SAVEPOINT. Zaključavanje će biti detaljnije prikazano u sljedećem potpoglavlju.

3. Transakcija može ili u cijelosti uspjeti (COMMIT), ili se u cijelosti poništiti (ROLLBACK). Naravno, ne znači da sve DML radnje unutar transakcije moraju uspjeti, jer one DML radnje koje su uzrokovale grešku, ali je greška obrađena, neće uspjeti.

4. DML naredba može ili u cijelosti uspjeti ili se njen efekt u cijelosti poništava. Pritom se mogu desiti tri tipa grešaka:
- narušeno ograničenje na tip stupca, npr. ako u NUMBER (2) pokušamo upisati broj 1000;
- narušeno deklarativno integritetno ograničenje (PK, UK, FK, CK, NOT NULL); zapravo provjera deklarativnih ograničenja može se i odgoditi (najkasnije do COMMIT) - tada naredba može uspjeti i ako deklarativna ograničenja nisu zadovoljena;
- narušena proceduralna ograničenja (okidači baze); to je najkompliciraniji slučaj, jer npr. jedna UPDATE naredba može okinuti različite UPDATE okidače, koji mogu pozivati procedure koje dalje rade DML, čime se okidaju drugi okidači.

5. Ako se desila greška na bazi koja nije obrađena, a početak je DML naredba, svi efekti se poništavaju (prethodna točka). Ako se desila greška na bazi koja nije obrađena, a početak je poziv procedure (ili funkcije) sa strane klijenta (npr. Forms ili Java), ili poziv sa strane druge baze (udaljena procedura), svi efekti procedure se poništavaju.

Često se kaže (npr. [2]) da transakcija treba zadovoljavati ACID svojstva (ACID property):
- A označava atomarnost (Atomicity);
- C označava konzistentnost (Consistency);
- I označava izoliranost (Isolation);
- D označava trajnost (Durability).

Napomenimo da je konzistentnost transakcije usko povezana sa atomarnošću, ali je za konzistentnost odgovoran programer, a ne baza. Za transakciju se često kaže da je ona jedinica integriteta (a unit of integrity). Zanimljivo je naglasiti da Date (npr. u [3]) tvrdi kako bi zapravo i svaka naredba (statement) trebala biti jedinica integriteta (što današnji DBMS-i ne omogućavaju).

Izoliranost se često zove i serijabilnost (serializability). Misli se na to da bi transakcije trebale uvijek ostaviti efekt kao da se izvršavaju serijski (jedna za drugom), iako se izvršavaju konkurentno (paralelno ili kvazi-paralelno). Da bi se postigla serijabilnost, DBMS sustavi primjenjuju dvofazno zaključavanje (two-phase locking), koje ne treba miješati sa dvofaznim commit protokolom (koji se koristi kod commit-iranja distribuirane transakcije). U fazi širenja (growing phase), transakcija zaključava retke, a poslije ih samo otključava, u fazi stezanja (shrinking phase).

Skoro svi DBMS sustavi koriste specifičnu varijantu dvofaznog zaključavanja, striktno dvofazno zaključavanje (strict two-phase locking), kod kojeg se faza stezanja radi neposredno prije kraja transakcije, prije commit-iranja (ili rollback-iranja). Ta varijanta eliminira problem kaskadnog abortiranja (cascading abort).

1.3. Transakcije i zaključavanje

Kako je rečeno u prethodnom potpoglavlju, kada sesija izvršava DML naredbu (INSERT / UPDATE / DELETE), automatski se zaključa redak. Redak se može otključati tek na kraju transakcije (COMMIT ili ROLLBACK), ili pomoću ROLLBACK TO SAVEPOINT. No treba naglasiti da sesiji koja pokuša pristupiti zaključanim redovima prije nego druga sesija napravi ROLLBACK TO SAVEPOINT, ti redovi i daje ostaju zaključani (do COMMIT ili ROLLBACK).

Iako se zaključavanje često radi implicitno, pomoću DML naredbi (INSERT, UPDATE, DELETE), ponekad je potrebno koristiti eksplicitno zaključavanje pomoću SELECT ... FOR UPDATE (rijetko se koristi LOCK TABLE) kako bi se sačuvao integritet podataka.

Sesija koja čeka na zaključane retke u pravilu čeka neograničeno, tj. dok joj druga sesija ne otključa retke. Postoje dva izuzetka:

1. Sesija može pokušati zaključati retke sa
SELECT ... FOR UPDATE;

i navesti da ne želi čekati
SELECT ... FOR UPDATE NOWAIT;

ili želi čekati određeni broj sekundi
SELECT ... FOR UPDATE WAIT timeout;

Ako je druga sesija zaključala redak, onda se prvoj sesiji javlja greška
ORA-00054: resource busy and acquire with NOWAIT specified;

2. Ako sesija čeka na otključavanje redaka sa udaljene baze, onda je čekanje definirano parametrom DISTRIBUTED_LOCK_TIMEOUT, koji ima standardnu vrijednost 60 sekundi. Ako druga sesija drži zaključan redak, nakon isteka tog vremena prvoj sesiji javit će se greška
ORA-02049: timeout: distributed transaction waiting for lock;
Ovu smo činjenicu koristili za trik (koji smo prikazali na HrOUG 2003, a kratko ćemo ga prikazati i u potpoglavlju 1.7.) – kako izbjeći neograničeno čekanje otključavanja retka kod INSERT naredbe, gdje prethodno korištenje naredbe SELECT .. FOR UPDATE NOWAIT nema efekta (jer uneseni redak za druge sesije nije još vidljiv).

Osim zaključavanja redaka (jednog ili više), ponekad želimo zaključati cijelu tablicu. Zanimljivo je da zaključavanje tablice možemo izvesti u djeljivom ili ekskluzivnom načinu (modu).

Više sesija može zaključati tablicu u djeljivom načinu:
LOCK TABLE tablica IN SHARE MODE NOWAIT;

Samo jedna sesija može zaključati tablicu u ekskluzivnom načinu (ako ju neka druga sesija nije već zaključala u djeljivom ili ekskluzivnom načinu):
LOCK TABLE tablica IN EXCLUSIVE MODE NOWAIT;

Zaključavanje tablice u djeljivom i ekskluzivnom načinu može se iskoristiti npr. za omogućavanje da više sesija mijenja neku tablicu dokumenata (npr. tablicu narudžbi), a da samo jedna sesija može raditi obradu narudžbi. Pritom se može zaključavati izvorna tablica dokumenata (npr. narudžbi), ili neka pomoćna tablica, koja može imati malo redaka (može i jedan, pa i nijedan).

Kod zaključavanja (implicitnog ili eksplicitnog) može se desiti deadlock - ako dvije transakcije pokušaju zaključati dva retka, ali u suprotnom redoslijedu. Npr. u sljedećem primjeru transakcija T1 uspješno zaključa račun broj 1, transakcija T2 uspješno zaključa račun 2, a obje žele zaključati i suprotni račun, pa će se desiti deadlock:

T1: SELECT iznos ... FROM racuni WHERE broj = 1 FOR UPDATE; -- uspješno
T2: SELECT iznos ... FROM racuni WHERE broj = 2 FOR UPDATE; -- uspješno
T1: SELECT iznos ... FROM racuni WHERE broj = 2 FOR UPDATE; -- čeka
T2: SELECT iznos ... FROM racuni WHERE broj = 1 FOR UPDATE; -- čeka

Srećom, Oracle baza otkrit će da je došlo do deadlocka, te će jedna od dvije transakcije dobiti grešku ORA-00060: deadlock detected while waiting for resource.

Nakon što ta transakcija (koja je dobila grešku) napravi ROLLBACK, druga transakcija će nastaviti sa radom.

1.4. Distribuirane transakcije

Sustav distribuiranih baza podataka (ili jednostavnije - distribuirana baza podataka) je sustav od dvije ili više baza podataka koje bi aplikacijama (korisničkim programima) trebale izgledati kao jedna jedinstvena baza podataka. Date je u svojoj knjizi [3] postavio "temeljni princip za distribuirane baze podataka", a to je: "Za korisnika, distribuirani sustav (baza podataka) treba izgledati potpuno isto kao ne-distribuirani sustav". Na temelju tog principa, Date u knjizi daje 12 zahtjeva koje distribuirana baza mora zadovoljiti. Činjenica je da je 100%-tno zadovoljenje svih tih zahtjeva gotovo nemoguće, ali ti zahtjevi služe kao ideal prema kojemu bi trebale težiti konkretne implementacije distribuiranih baza podataka.

Ne ulazeći detaljnije u definiranje svakog od ovih zahtjeva, može se reći da ih Oracle baza podataka zadovoljava u velikoj mjeri. U zadovoljavanju tih zahtjeva veliku ulogu igra nepostojanje "pravog" globalnog rječnika podataka u Oracle bazi, jer svaka Oracle baza ima svoj lokalni rječnik podataka. S druge strane, lokalna baza ipak mora sadržavati neke informacije o bazama sa kojima komunicira. Oracle baza (ali i druge baze) to radi tako da lokalni rječnik sadrži podatke o udaljenim objektima baze.

Za čuvanje informacija o udaljenim objektima, Oracle baza podataka koristi tzv. "database link". Može se reći da je database link objekt baze podataka koji definira jednosmjernu vezu baze podataka na drugu bazu podataka. Postoji više vrsta database linkova. Jedna je podjela na globalne (koji pripadaju bazi) i privatne database linkove (koji pripadaju određenoj shemi baze), a druga podjela dijeli database link-ove prema načinu na koji se korisnik prijavljuje na udaljenu bazu (connected user, fixed user ili current user link). Slijedi primjer definiranja privatnog database linka koji je po drugoj podjeli "fixed user link" (pretpostavimo da smo database link kreirali kao objekt sheme "shemaX" unutar baze "bazaA"):

```
CREATE DATABASE LINK neki_link
  CONNECT TO shemaY IDENTIFIED BY zaporka USING "bazaB";
```

Ponekad nužno moramo raditi sa distribuiranom bazom podataka i distribuiranim transakcijama. Distribuirana transakcija koristi tzv. dvofazni commit protokol, koji ima faze Prepare i Commit (u Oracle realizaciji on se, zapravo, sastoji od tri faze – treća faza je Forget).

Ovako možemo ukratko prikazati zbivanja kod uspješne distribuirane transakcije, koja završava sa COMMIT i kod koje nije bilo nikakvih problema:

1. Klijentska aplikacija šalje DML naredbe (ili/i pozive udaljenih procedura) po distribuiranoj bazi i na kraju završava sa COMMIT.
2. Ovdje počinje prva faza. Globalni koordinator (baza na koju je direktno vezana klijentska aplikacija) određuje koja će baza biti tzv. commit point site (u tome pomažu i ostale baze, lokalni koordinatori).
3. Globalni koordinator svim bazama, osim commit point site bazi, šalje naredbu da se pripreme.
4. Sve baze javljaju potvrdan odgovor (Prepared).
5. Ovdje počinje druga faza. Globalni koordinator javlja commit point site bazi da izvrši lokalni COMMIT.
6. Commit point site baza izvršava lokalni COMMIT i obavještava globalnog koordinatora.
7. Globalni i lokalni koordinatori javljaju svim ostalim bazama da naprave lokalni COMMIT.
8. Sve baze rade lokalni COMMIT i obavještavaju koordinatore.
9. Ovdje počinje treća faza. Globalni koordinator obavještava commit point site bazu da zaboravi distribuiranu transakciju.
10. Commit point site baza briše svoje podatke o distribuiranoj transakciji i obavještava globalnog koordinatora.
11. Globalni koordinator briše svoje podatke o distribuiranoj transakciji.

Nažalost, kod dvofaznog commit protokola postoji period u kojem je transakcija osjetljiva na pad (nekog) servera baze ili veze između baza. Greške koje se mogu desiti jesu:

- palo je računalo na kojem radi Oracle baza;
- prekinula se veza između dvije ili više baza koje sudjeluju u distribuiranoj transakciji;
- desio se neki softverski problem.

U tom slučaju (tj. ako se desi greška u osjetljivoj fazi) transakcija postaje in-doubt. U fazi pripreme, baze stavljaju distribuirani lokot na sve modificirane tablice. Taj lokot sprečava čak i čitanje podataka! Ako to traje kratko, nije problem. No ako transakcija postane in-doubt, može se desiti da duže vrijeme drži zaključane podatke (čak i za čitanje). Najbolje je ostaviti da baza sama razriješi in-doubt transakciju. No ako je zadovoljen jedan od ova dva uvjeta, trebali bismo ručno razriješiti in-doubt transakciju:

1. in-doubt transakcija je zaključala kritične podatke ili undo segmente;
2. pad računala, prekid veze, ili softverski problem ne mogu se riješiti u kratkom vremenu.

1.5. Rješavanje mutiranja okidača baze

Pretpostavimo da želimo u Oracle bazi realizirati sljedeće poslovno pravilo (koje nije baš realno, ali služi za prikaz problema):

"Niti jedan radnik ne smije imati veću plaću od radnika koji se zove KING".

Pretpostavimo da smo na neki način (to je drugo poslovno pravilo) osigurali da u tablici RADNIK postoji radnik imena KING i to samo jedan. Kreirajmo jednostavnu tablicu radnik za ovu potrebu:

```
CREATE TABLE radnik (
   sifra NUMBER (10),
   ime VARCHAR2 (30) NOT NULL,
   placa NUMBER (10,2) NOT NULL,
   CONSTRAINT sifra_pk PRIMARY KEY (sifra),
   CONSTRAINT ime_uk UNIQUE (ime)
)
/
```

Dakle, definirali smo PK ograničenje nad šifrom i UK nad imenom (prezimena u ovom slučaju nema), te neprazan (NOT NULL) stupac PLACA. Sada unesemo redak za radnika KING (kako je rečeno, pretpostavljamo da smo na neki način osigurali da će on uvijek postojati).

```
INSERT INTO radnik VALUES (1, 'KING', 100000);
```

Sada pokušavamo zadovoljiti gornje poslovno pravilo pomoću okidača baze (jer deklarativno ne možemo). Možemo birati da li ćemo to napraviti u BEFORE ili AFTER okidaču za INSERT i UPDATE (kod naredbe DELETE ne moramo provjeravati to pravilo). Uzimimo da smo odabrali AFTER okidač, koji se dešava nakon fizičkog unosa / izmjene retka u bazi, pa su već provjerena deklarativna integritetna ograničenja (ako nisu odgođena), pa je provjereno i da je plaća NOT NULL. Pokažimo kako bi mogao izgledati pokušaj realizacije okidača za INSERT naredbu (za UPDATE bi bilo slično):

```
CREATE OR REPLACE TRIGGER air_radnik
   AFTER INSERT ON radnik
   FOR EACH ROW
DECLARE
   placa_za_king NUMBER (10, 2);
BEGIN
   SELECT placa INTO placa_za_king
     FROM radnik
    WHERE ime = 'KING';

   IF :NEW.placa > placa_za_king THEN
      RAISE_APPLICATION_ERROR
      (-20000, 'Nitko ne smije imati veću plaću nego KING');
   END IF;
END;
/
```

Sada pokušavamo unijeti radnika koji nema veću plaću od KING, pa bi unos trebao proći, ali dešava se sljedeće:

```
INSERT INTO radnik VALUES (2, 'Ana', 8000)
            *
ERROR at line 1:
ORA-04091: table I3RAZVOJ.RADNIK is mutating,
    trigger/function may not see it
ORA-06512: at "I3RAZVOJ.AIR_RADNIK", line 4
ORA-04088: error during execution of trigger 'I3RAZVOJ.AIR_RADNIK'
```

Kako se vidi, Oracle baza kaže: table I3RAZVOJ.RADNIK is mutating trigger/function may not see it. Riječ je o tome da se za vrijeme izvršenja retčanih (row) okidača, tablica nad kojom je okidač definiran, nalazi u stanju promjene – mutiranja. U Oracle bazi row okidači ne smiju čitati, niti mijenjati tablicu koja mutira. Ovo djeluje kao ograničenje Oracle sustava (neki sustavi to nemaju), ali je ovo ograničenje vrlo važno kod sprečavanja vrlo čudnih grešaka koji bi se inače mogli desiti.

Moramo napomenuti da se ovo mutiranje ne bi javilo u slučaju da imamo BEFORE ROW (a ne AFTER ROW) okidač i da unosimo (ili mijenjamo) samo jedan redak. No u općenitom slučaju možemo jednom naredbom unijeti ili mijenjati više redaka, tako da se općenito ne smijemo osloniti na to da se mutiranje ne javlja u BEFORE ROW okidaču ako ažuriramo samo jedan redak.

Sada ćemo riješiti problem mutiranja na sljedeći način – umjesto u ROW okidač, provjeru ćemo staviti u STATEMENT okidač, i to u AFTER STATEMENT, jer se tamo mutiranje više ne javlja (kada dođe do tamo, svi redovi su već ažurirani). No postoji jedan problem – kako da AFTER STATEMENT okidač zna koji redak (ili koji redovi, jer može ih biti više) je ažuriran. Za to će nam poslužiti tzv. PL/SQL (memorijska) tablica, koju ćemo inicijalno prazniti u BEFORE STATEMENT okidaču, puniti svaki put u ROW (npr. AFTER ROW) okidaču i čitati, tj. primijeniti poslovno pravilo, u AFTER STATEMENT okidaču. Umjesto da programski kod pišemo direktno u okidaču, sada ćemo ga pisati u posebnom paketu, tako da ćemo taj paket moći pozvati sa više mjesta (npr. iz AFTER INSERT i AFTER UPDATE okidača).

Prvo ćemo napraviti samo tzv. specifikaciju paketa (bez tijela paketa, tj. bez realizacije) i okidače:

```
CREATE OR REPLACE PACKAGE pravilo_king IS
   PROCEDURE brisi_plsql_tablicu;
   PROCEDURE zapamti_u_plsql_tablicu (p_placa NUMBER);
   PROCEDURE provjeri_pravilo;
END;
/
-- Poziva proceduru koja prazniti PL/SQL tablicu
CREATE OR REPLACE TRIGGER bis_radnik
   BEFORE INSERT ON radnik
BEGIN
   pravilo_king.brisi_plsql_tablicu;
END;
/
-- Poziva proceduru koja u PL/SQL tablicu pamti
-- plaću radnika koji se trenutačno unosi
CREATE OR REPLACE TRIGGER air_radnik
   AFTER INSERT on radnik
   FOR EACH ROW
BEGIN
   pravilo_king.zapamti_u_plsql_tablicu (:NEW.placa);
END;
/
-- Poziva proceduru koja provjerava poslovno pravilo
CREATE OR REPLACE TRIGGER ais_radnik
   AFTER INSERT on radnik
BEGIN
   pravilo_king.provjeri_pravilo;
END;
/
```

Tijelo paketa izgledat će ovako:

```
CREATE OR REPLACE PACKAGE BODY pravilo_king AS
   -- Prvo se definira tip, čiji su elementi tipa NUMBER,
   -- za pamćenje plaća radnika koji se unose ili mijenjaju
   TYPE tip_plsql_tablica IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
   -- Definira se varijabla prethodnog tipa,
   -- tj. niz koji će pamtiti plaće radnika
   plsql_tablica tip_plsql_tablica;

   -- Brojač koji će pamtiti redni broj tekućeg elementa niza
   broj_retka BINARY_INTEGER;

   PROCEDURE brisi_plsql_tablicu IS
   BEGIN
      -- Ne briše se tablica, već se brojač postavlja na nulu
      broj_retka := 0;
   END;
```

```
   PROCEDURE zapamti_u_plsql_tablicu (p_placa NUMBER) IS
   BEGIN
      -- Broj redaka povećava se za 1 i na odgovarajuće
      -- mjesto u nizu stavlja se plaća tekućeg radnika
      broj_retka := broj_retka + 1;
      plsql_tablica (broj_retka) := p_placa;
   END;

   PROCEDURE provjeri_pravilo IS
      placa_za_king NUMBER (10, 2);
   BEGIN
      -- Čita se PL/SQL tablica, te se provjerava pravilo
      -- za svaku zapamćenu plaću
      FOR i IN 1..broj_retka LOOP
         -- Pretpostavljamo da postoji (točno jedan) KING
         -- inače bi sljedeći SELECT javio grešku
         SELECT placa INTO placa_za_king
           FROM radnik
          WHERE ime = 'KING';

         IF plsql_tablica (broj_retka) > placa_za_king THEN
            RAISE_APPLICATION_ERROR (-20000,
               'Nitko ne smije imati veću plaću nego KING');
         END IF;
      END LOOP;
   END;
END;
/
```

Ako sada pokušamo unijeti radnika koji nema veću plaću od KING, unos će proći, više se ne dešava mutiranje:

```
INSERT INTO radnik VALUES (2, 'Ana', 8000);
```

Ako pokušamo unijeti radnika koji ima veću plaću od KING, javit će grešku, tj. pravilo neće biti narušeno:

```
INSERT INTO radnik VALUES (3, 'Pero', 110000)
            *
ERROR at line 1:
ORA-20000: Nitko ne smije imati veću plaću nego KING
ORA-06512: at "I3RAZVOJ.PRAVILO_KING", line 32
ORA-06512: at "I3RAZVOJ.AIS_RADNIK", line 2
ORA-04088: error during execution of trigger 'I3RAZVOJ.AIS_RADNIK'
```

Primijetimo da smo mogli napisati još bolji kod, u kojem bi poruka javila ne samo da plaća ne može biti veća od one koju ima radnik KING, već i šifru (i/ili ime) radnika kod kojeg smo pokušali narušiti to pravilo (u ovom slučaju smo unosili samo jednog radnika, ali mogli smo ih više). U tom slučaju bismo u PL/SQL tablicu trebali pamtiti ne samo plaću, već i šifru ili/i ime, ili pamtiti samo šifru, pa čitati tablicu radnika za ostale podatke.

Ovaj paket mogao bi se koristiti i za UPDATE okidače, koji bi bili vrlo slični ovima za INSERT. No naglasimo da možemo imati i okidače koji istovremeno rade i za INSERT i za UPDATE, npr.:

```
CREATE OR REPLACE TRIGGER bis_radnik
   BEFORE INSERT OR UPDATE -- !!!
   ON radnik
BEGIN
   pravilo_king.brisi_plsql_tablicu;
END;
/
```

1.6. Simulacija COMMIT okidača pomoću odgođenih deklarativnih integritetnih ograničenja

Postoje poslovna pravila koja prije verzije 8.0 Oracle baze nije bilo moguće u potpunosti podržati samo na strani baze, već je trebala "suradnja" klijent strane i baze. Vjerojatno najjednostavniji primjer takvog pravila je: "Instanca master-a mora imati barem jednu pripadajuću instancu detalja" (npr. "DEPT mora imati barem jedan EMP").

Takva pravila mogu se nazvati "COMMIT pravila", jer se njihovo (ne)zadovoljavanje može provjeriti tek prije COMMIT-a transakcije, a ne u trenutku kada se radi neka DML naredba. Naravno, poznato je da na Oracle bazi ne postoji (možda neće ni ubuduće) nekakav BEFORE COMMIT okidač (koji bi poslužio za rješavanje "COMMIT pravila"). Zato se “COMMIT pravila” mora pokušati riješiti kroz vlastito rješenje, i to na (barem) dva različita načina. Prvi način traži “suradnju” klijenta sa bazom i jedini je mogući na Oracle bazi prije verzije 8.0. Drugi način (indirektno) je omogućen Oracle verzijom 8.0 (ili većom).

Na Oracle bazi 7 problem se moglo rješavati npr. na taj način da se master tablici doda stupac npr. BR_VALID (BR kao Business Rule) i da se pomoću okidača baze osigura da redak koji ima BR_VALID = 'T' (True) zadovoljava "COMMIT pravilo". Npr. tablici DEPT doda se stupac BR_VALID i napuni ga se za početak ispravnom vrijednošću:

```
ALTER TABLE dept
   ADD (br_valid CHAR (1) DEFAULT 'F' NOT NULL,
        CONSTRAINT dept_br_valid_ck
        CHECK (br_valid IN ('T', 'F')))
/
UPDATE dept
   SET br_valid = 'T'
 WHERE EXISTS (SELECT 1
                 FROM emp
                WHERE emp.deptno = dept.deptno);
```

Zatim se dodaju okidači nad DEPT i EMP tablicama:

```
CREATE OR REPLACE TRIGGER bir_dept
   BEFORE INSERT ON dept FOR EACH ROW
BEGIN
   :NEW.br_valid := 'F';
END;
/
CREATE OR REPLACE TRIGGER bur_dept
   BEFORE UPDATE ON dept
   FOR EACH ROW
BEGIN
   IF :OLD.deptno <> :NEW.deptno THEN
      RAISE_APPLICATION_ERROR
         (-20001, 'Ne može se mijenjati deptno u DEPT!');
   END IF;

   IF :OLD.br_valid = 'F' AND :NEW.br_valid = 'T' THEN
      DECLARE
         nebitno_l NUMBER (1, 0);
         CURSOR c_emp IS
            SELECT 1 FROM emp WHERE deptno = :OLD.deptno;
      BEGIN
         OPEN c_emp;
         FETCH c_emp INTO nebitno_l;
         IF c_emp%NOTFOUND THEN
            CLOSE c_emp;
            RAISE_APPLICATION_ERROR
            (-20002,'DEPT mora imati barem jedan EMP!');
         END IF;
         CLOSE c_emp; -- u praksi bismo CLOSE radili i u EXCEPTION-u
     END;
   END IF;
END;
```

```
CREATE OR REPLACE TRIGGER aur_emp
   AFTER UPDATE ON emp
   FOR EACH ROW
BEGIN
   IF NVL (:OLD.deptno, 0) <>  NVL (:NEW.deptno, 0) THEN
      UPDATE dept
         SET br_valid = 'F'
       WHERE deptno = :OLD.deptno
         AND br_valid = 'T';
   END IF;
END;
/
CREATE OR REPLACE TRIGGER adr_emp
   AFTER DELETE ON emp
   FOR EACH ROW
BEGIN
   UPDATE dept
      SET br_valid = 'F'
    WHERE deptno = :OLD.deptno
      AND br_valid = 'T';
END;
/
```

Ako se radi npr. sa Oracle Forms razvojnim alatom, tada se u Forms okidaču Post-Forms-Commit (okida se neposredno prije COMMIT naredbe na bazi) može napisati:

```
UPDATE dept
   SET br_valid = 'T'
 WHERE deptno := :dept.deptno
   AND br_valid = 'F';
```

Ako Forms program (ili neki drugi klijentski program) nije dao tu naredbu prije COMMIT-a transakcije, na bazi su mogli ostati redovi tablice DEPT sa BR_VALID = 'F'. Dakle, u ovoj varijanti klijent mora "surađivati" sa bazom.

Metoda koja omogućava rješavanje "COMMIT pravila" u potpunosti na razini baze zasniva se na mogućnosti odgađanja provjere deklarativnih integritetnih ograničenja (NOT NULL, PK, UK, FK, CK) sve do COMMIT-a. Ta je mogućnost prvi put uvedena u bazi 8.0. Napomena: izvršavanje okidača baze ne može se odgoditi.

Npr. tablici DEPT doda se stupac NUM_EMPS čija će vrijednost uvijek biti jednaka broju pripadajućih EMP redaka i doda se odgođeni (deferred) CK koji koristi vrijednost iz tog stupca:

```
ALTER TABLE dept ADD num_emps NUMBER DEFAULT 0 NOT NULL
/
UPDATE dept
   SET num_emps = (SELECT COUNT (*)
                     FROM emp
                    WHERE emp.deptno = dept.deptno);

DELETE dept WHERE num_emps = 0; -- zbog sljedećeg CK

ALTER TABLE dept
   ADD CONSTRAINT dept_num_emps_ck
      CHECK (num_emps > 0) INITIALLY DEFERRED
/
```

Okidači koji osiguravaju rješavanje "COMMIT pravila" na strani baze su relativno jednostavni. No potrebna je pakirana varijabla koja se setira i resetira u EMP okidačima i čija se vrijednost čita u BUR_DEPT (naravno, moglo se varijablu staviti u specifikaciju paketa i mijenjati / čitati ju direktno, pa ne bi trebalo tijelo paketa, ali ovako je "čišće"):

```
CREATE OR REPLACE PACKAGE pack IS
   PROCEDURE set_flag;
   PROCEDURE reset_flag;
   FUNCTION dml_from_emp RETURN BOOLEAN;
END;
/
CREATE OR REPLACE PACKAGE BODY pack IS
   dml_from_emp_m BOOLEAN := FALSE;

   PROCEDURE set_flag IS
      BEGIN dml_from_emp_m := TRUE; END;
   PROCEDURE reset_flag IS
      BEGIN dml_from_emp_m := FALSE; END;
   FUNCTION dml_from_emp RETURN BOOLEAN IS
      BEGIN RETURN dml_from_emp_m; END;
END;
/
```

Okidač BIR_DEPT (before insert row - okida se jedanput za svaki uneseni redak) postavlja vrijednost NUM_EMPS na 0:

```
CREATE OR REPLACE TRIGGER bir_dept
   BEFORE INSERT ON dept
   FOR EACH ROW
BEGIN
   :NEW.num_emps := 0;
END;
/
```

Okidač BUR_DEPT (before update row - okida se jedanput za svaki mijenjani redak) zabranjuje promjenu DEPTNO i, ako PACK.DML_FROM_EMP ne označava da se promjena radi kontrolirano, vraća mijenjanu vrijednost NUM_EMPS na prethodnu:

```
CREATE OR REPLACE TRIGGER bur_dept
   BEFORE UPDATE ON dept
   FOR EACH ROW
BEGIN
   IF :OLD.deptno <> :NEW.deptno THEN
      RAISE_APPLICATION_ERROR
         (-20001, 'Ne može se mijenjati deptno u DEPT!');
   END IF;
   -- Samo EMP okidači smiju mijenjati "num_emps" stupac
   IF NOT pack.dml_from_emp THEN
      :NEW.num_emps := :OLD.num_emps;
   END IF;
END;
/
```

Okidač AIR_EMP (after insert row - okida se jedanput za svaki uneseni redak, nakon unosa retka) povećava vrijednost NUM_EMPS u retku tablice DEPT za jedan:

```
CREATE OR REPLACE TRIGGER air_emp
   AFTER INSERT ON emp
   FOR EACH ROW
BEGIN
   pack.set_flag;
   UPDATE dept
      SET num_emps = num_emps + 1
    WHERE deptno = :NEW.deptno;
   pack.reset_flag;
END;
/
```

Okidač AUR_EMP (after update row - okida se jedanput za svaki mijenjani redak, nakon izmjene retka), ako je došlo do promjene DEPTNO, smanjuje (za jedan) vrijednost NUM_EMPS u tablici DEPT u retku za koji je EMP prije bio vezan, a povećava (za jedan) vrijednost u retku za koji je EMP sada vezan:

```
CREATE OR REPLACE TRIGGER aur_emp
   AFTER UPDATE ON emp
   FOR EACH ROW
BEGIN
   IF NVL (:OLD.deptno, 0) <>  NVL (:NEW.deptno, 0) THEN
      pack.set_flag;
      UPDATE dept
         SET num_emps = num_emps - 1
       WHERE deptno = :OLD.deptno;
      UPDATE dept
         SET num_emps = num_emps + 1
       WHERE deptno = :NEW.deptno;
      pack.reset_flag;
   END IF;
END;
/
```

Okidač ADR_EMP (after delete row - okida se jedanput za svaki brisani redak, nakon brisanja retka) smanjuje vrijednost NUM_EMPS u retku tablice DEPT za jedan:

```
CREATE OR REPLACE TRIGGER adr_emp
   AFTER DELETE ON emp FOR EACH ROW
BEGIN
   pack.set_flag;
   UPDATE dept
      SET num_emps = num_emps - 1
    WHERE deptno = :OLD.deptno;
   pack.reset_flag;
END;
/
```

Ako se unese novi DEPT bez pripadajućih EMP ili se brišu svi EMP određenog DEPT, ili se presele svi EMP određenog DEPT, nakon COMMIT dobije se greška:

ORA-02091: transaction rolled back
ORA-02290: check constraint (SCOTT.DEPT_NUM_EMPS_CK) violated

Može se primijetiti da su oba rješenja tražila "pomoćne" stupce u DEPT tablici (BR_VALID u prvom rješenju, NUM_EMPS u drugom). Pritom u drugom rješenju, za razliku od prvog, treba (najčešće) po jedan "pomoćni" stupac za svako "COMMIT pravilo". Ako bi se DEPT tablici htjelo dodati još jedno "COMMIT pravilo", npr.:

"SUM (sal) FROM emp WHERE deptno = p_deptno must be <= p_max_dept_sal"

tada bi se kod prvog rješenja moglo koristiti isti stupac BR_VALID za oba pravila, a kod drugog rješenja morao bi se uvesti novi stupac, npr. DEPT_SAL.

Mora se naglasiti da se u stvarnom radu PL/SQL programski kod ne bi pisao direktno u okidačima baze, već u paketima, niti bi se naredba RAISE_APPLICATION_ERROR koristila direktno. Ovdje je tako pisano zbog jednostavnijeg praćenja suštine koda.

Može se primijetiti da, na prvi pogled, postoji puno jednostavnije rješenje pomoću odgođene provjere deklarativnih integritetnih ograničenja. Naime, umjesto da se uvodi "pomoćno" polje NUM_EMPS i CK koji ga provjerava, moglo bi se odgoditi validaciju FK sa EMP na DEPT i prvo unijeti EMP retke, a onda DEPT redak, te u BIR_DEPT okidaču provjeravati da li ima pripadajućih EMP redaka. Međutim, takvo "rješenje" je nepotpuno, tj. nije rješenje, jer bi se moglo npr. naknadnim brisanjem EMP narušiti pravilo.

Napomenimo da Date (npr. u [3]) nije pristalica odgađanja deklarativnih integritetnih rješenja, jer tvrdi kako bi svaka naredba, a ne tek transakcija, trebala biti jedinica integriteta.

Ovo je programsko rješenje bilo prikazano na web stranicama online Oracle magazina (stranica više nije dostupna), kao PL/SQL tip u 3.mjesecu 2002., te ukratko (bez detalja) na HROUG-u 2002.

1.7. Simulacija INSERT WAIT naredbe

Poznato je da SELECT naredba ima FOR UPDATE klauzulu, kojom se zaključavaju redovi koji se čitaju. FOR UPDATE klauzula ima opcionalnu riječ NOWAIT, kojom označavamo da ne želimo čekati da druga sesija otključa redak. Od baze 9i, postoji i opcija WAIT integer.

INSERT, UPDATE i DELETE naredbe nemaju (a možda nikad neće ni imati) NOWAIT ili WAIT opciju. To nije veći problem kod UPDATE i DELETE naredbe, zato jer prije njih možemo pozvati SELECT ... FOR UPDATE NOWAIT (ili WAIT n od Oracle 9i):

```
-- 1.sesija: mijenja redak i automatski ga zaključava
BEGIN
   UPDATE dept SET dname = dname  WHERE deptno = 10;
END;

-- 2.sesija: SELECT nalazi da je redak zaključan i (zbog NOWAIT) ne čeka
DECLARE
   l_rowid ROWID;
BEGIN
   SELECT ROWID INTO l_rowid
     FROM dept
    WHERE deptno = 10 FOR UPDATE NOWAIT;
   UPDATE dept SET dname = dname WHERE ROWID = l_rowid;
EXCEPTION
   WHEN OTHERS THEN
      -- ORA-00054: resource busy and acquire with NOWAIT specified
      IF SQLCODE = -54 THEN
         DBMS_OUTPUT.PUT_LINE
            ('DEPT je zaključan - ne možete mijenjati!');
      ELSE
         RAISE;
      END IF;
END;
```

Međutim, pozivanje SELECT ... FOR UPDATE NOWAIT prije INSERT naredbe nema efekta, zato što SELECT naredba u drugoj sesiji ne vidi redak koji je prva sesija unijela (a nije još COMMIT-irala):

```
-- 1.sesija: unosi redak i automatski ga zaključava
BEGIN
   INSERT INTO dept (deptno, dname) VALUES (99, 'DEPT 99');
END;

-- 2.sesija: SELECT ne nalazi ništa, INSERT čeka nedefinirano vrijeme
DECLARE
   l_dummy NUMBER;
BEGIN
   SELECT 1 INTO l_dummy
     FROM dept
    WHERE deptno = 99 FOR UPDATE NOWAIT;
   DBMS_OUTPUT.PUT_LINE ('DEPT postoji!');
   ROLLBACK; -- otključava redak
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      INSERT INTO dept (deptno, dname) VALUES (99, 'DEPT 99');
   WHEN OTHERS THEN
      IF SQLCODE = -54 THEN
         DBMS_OUTPUT.PUT_LINE ('DEPT je zaključan - ne možete unijeti!');
      ELSE
         RAISE;
      END IF;
END;
```

No Oracle baza ima parametar baze (koji se može mijenjati u parametarskoj datoteci INIT.ORA) DISTRIBUTED_LOCK_TIMEOUT. Njime se može specificirati koliko vremena (sekundi) distribuirana transakcija čeka na zaključani redak (default je 60 sekundi).

Mi stvarno nemamo distribuiranu transakciju, ali ćemo napraviti kvazi-distribuiranu transakciju, kako bismo mogli iskoristiti navedeni parametar. Kvazi-distribuiranu transakciju dobit ćemo tako da u DML naredbi koristimo database link koji ima alias (service name) na lokalnu (a ne na udaljenu) bazu:

```
CREATE DATABASE LINK local_db_link
   CONNECT TO scott IDENTIFIED BY tiger
   USING 'local_alias' -- alias na lokalnu bazu
/

-- 1.sesija: unosi redak i automatski ga zaključava
BEGIN
   INSERT INTO dept (deptno, dname) VALUES (99, 'DEPT 99');
END;

-- 2.sesija: unosi redak koji je već unijela prethodna sesija,
-- a budući da INSERT radi preko database link-a, nakon isteka vremena
-- određenog sa DISTRIBUTED_LOCK_TIMEOUT, dešava se EXCEPTION (ORA-02049)
BEGIN
   INSERT INTO dept@local_db_link -- KVAZI-UDALJENA NAREDBA
      (deptno, dname) VALUES (99, 'DEPT 99');
EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN
      DBMS_OUTPUT.PUT_LINE ('DEPT postoji!');
   WHEN OTHERS THEN
      -- ORA-02049: timeout: distributed transaction waiting for lock
      IF SQLCODE = -2049 THEN
         DBMS_OUTPUT.PUT_LINE ('DEPT je zaključan - ne možete unijeti!');
      ELSE
         RAISE;
      END IF;
END;
```

Ovo je programsko rješenje bilo prikazano na web stranicama firme Quest u 1.mjesecu 2003. (stranica više nije dostupna), te na HROUG-u 2003.

1.8. Simulacija ROLLBACK TO SAVEPOINT naredbe u okidaču baze

Ponekad želimo da transakcija uspije, bez obzira što neki njen dio nije uspio. U takvim slučajevima obično koristimo naredbe SAVEPOINT / ROLLBACK TO SAVEPOINT.

Međutim, naredbe SAVEPOINT / ROLLBACK TO SAVEPOINT ne možemo koristiti u okidaču baze, jer se javljaju greške:

ORA-04092: cannot SET SAVEPOINT in a trigger

ORA-04092: cannot ROLLBACK in a trigger

Moguće je simulirati SAVEPOINT / ROLLBACK TO SAVEPOINT naredbe u okidaču baze. Pokažimo to na jednom jednostavnom, izmišljenom primjeru.

Pretpostavimo da želimo imati transakciju koja se sastoji od 3. dijela:

1. unos jednog DEPT
2. unos dva EMP koji imaju job = MANAGER (i koji pripadaju prethodno unesenom DEPT)
3. unos dva EMP koji imaju job = PROGRAMER (i koji pripadaju prethodno unesenom DEPT)

Pretpostavimo dalje da želimo da transakcija uspije i ako 3.dio ne uspije, ali samo tako da se poništi ono što je 3.dio napravio (tj. unos samo jednog EMP). Dakle, transakcija je ispravna samo onda kada na kraju transakcije imamo:

a) 1 redak DEPT, 2 retka EMP = MANAGER, 2 retka EMP = PROGRAMER

ili

b) 1 redak DEPT, 2 retka EMP = MANAGER

Napravimo prvo paket koji ne radi dobro:

```
CREATE OR REPLACE PACKAGE example_pkg IS
   PROCEDURE insert_emps_for_dept (p_deptno dept.deptno%TYPE);
END;
/
CREATE OR REPLACE PACKAGE BODY example_pkg IS
   PROCEDURE insert_managers (p_deptno dept.deptno%TYPE) IS
   BEGIN
      INSERT INTO emp (empno, ename, job, mgr, sal, deptno)
         VALUES (1, 'EMP 1', 'MANAGER', NULL, 5000, p_deptno);

      INSERT INTO emp (empno, ename, job, mgr, sal, deptno)
         VALUES (2, 'EMP 2', 'MANAGER', 1,  4000, p_deptno);
   END;

   PROCEDURE insert_programmers (p_deptno dept.deptno%TYPE) IS
   BEGIN
      INSERT INTO emp (empno, ename, job, mgr, sal, deptno)
         VALUES (3, 'EMP 3', 'PROGRAMER', 1, 1000, p_deptno);
      RAISE_APPLICATION_ERROR
        (-20001, 'SIMULATED ERROR IN MIDLE OF 3.PART OF A TRANSACTION');
      INSERT INTO emp (empno, ename, job, mgr, sal, deptno)
         VALUES (4, 'EMP 4', 'PROGRAMER', 1, 1000, p_deptno);
   END;

   PROCEDURE insert_emps_for_dept (p_deptno dept.deptno%TYPE) IS
   BEGIN
      -- 2. part of a transaction
      insert_managers (p_deptno);
      -- 3. part of a transaction
      BEGIN
         insert_programmers (p_deptno);
      EXCEPTION
         WHEN OTHERS THEN NULL;
      END;
   END;
END;
/
```

Pozovimo sada proceduru iz neimenovanog PL/SQL bloka (bez okidača baze), sa:

```
BEGIN
   INSERT INTO dept (deptno, dname) VALUES (1, 'DEPT 1');
   example_pkg.insert_emps_for_dept (1);
END;
```

Pogledajmo šta smo dobili, sa SELECT upitom:

```
SELECT emp.empno, emp.ename, dept.deptno, dept.dname
  FROM emp, dept
 WHERE empno BETWEEN 1 AND 4
   AND emp.deptno = dept.deptno
 ORDER BY empno;

     EMPNO ENAME               DEPTNO DNAME
---------- --------------- ---------- --------------
         1 EMP 1                    1 DEPT 1
         2 EMP 2                    1 DEPT 1
         3 EMP 3                    1 DEPT 1
```

Naravno, transakcija nije dobra, jer je ostao upisan EMP 3.

Napravimo ROLLBACK i mijenjajmo proceduru insert_emps_for_dept tako da dodamo naredbe SAVEPOINT / ROLLBACK TO SAVEPOINT:

```
PROCEDURE insert_emps_for_dept (p_deptno dept.deptno%TYPE) IS
BEGIN
   -- 2. part of a transaction
   insert_managers (p_deptno);
   -- 3. part of a transaction
   BEGIN
      SAVEPOINT before_insert_programmers;
      insert_programmers (p_deptno);
   EXCEPTION
      WHEN OTHERS THEN ROLLBACK TO before_insert_programmers;
   END;
END;
```

Ako sada napravimo prethodni postupak, dobit ćemo ispravno:

```
     EMPNO ENAME                DEPTNO DNAME
---------- --------------- ---------- --------------
         1 EMP 1                    1 DEPT 1
         2 EMP 2                    1 DEPT 1
```

Napravimo opet ROLLBACK i kreirajmo okidač:

```
CREATE OR REPLACE TRIGGER air_dept
   AFTER INSERT ON dept
FOR EACH ROW
BEGIN
   example_pkg.insert_emps_for_dept (:NEW.deptno);
END;
/
```

te pokušajmo izvesti naredbu:

```
INSERT INTO dept (deptno, dname) VALUES (1, 'DEPT 1');
```

Naravno, dešava se greška:
ERROR at line 1:
ORA-04092: cannot ROLLBACK in a trigger
ORA-06512: at "SCOTT.EXAMPLE_PKG", line 33
ORA-04092: cannot SET SAVEPOINT in a trigger
ORA-06512: at "SCOTT.AIR_DEPT", line 2
ORA-04088: error during execution of trigger 'SCOTT.AIR_DEPT'

Napravimo opet ROLLBACK. Sada ćemo (konačno) primijeniti trik. On se temelji na činjenici da ako pozivamo udaljenu proceduru (preko database linka) i ako se u njoj desi neobrađena greška, njeni se efekti u cijelosti poništavaju (za razliku od lokalne procedure). Nama ne treba udaljena procedura, ali napravit ćemo kvazi-udaljenu proceduru, koristeći lokalni database link:

```
CREATE DATABASE LINK local_db_link
   CONNECT TO scott IDENTIFIED BY tiger using 'local_alias'
/
```

Mijenjajmo opet proceduru insert_emps_for_dept, tako da poziva proceduru insert_programmers preko database linka. Međutim, zbog toga moramo mijenjati i specifikaciju paketa:

```
CREATE OR REPLACE PACKAGE example_pkg IS
   PROCEDURE insert_emps_for_dept (p_deptno dept.deptno%TYPE);
   -- procedura mora biti u specifikaciji (zbog database linka)
   PROCEDURE insert_programmers (p_deptno dept.deptno%TYPE);
END;
/
```

```
CREATE OR REPLACE PACKAGE BODY example_pkg IS
   ...
   PROCEDURE insert_emps_for_dept (p_deptno dept.deptno%TYPE) IS
   BEGIN
      -- 2. part of a transaction
      insert_managers (p_deptno);
      -- 3. part of a transaction
      BEGIN
         example_pkg.insert_programmers@local_db_link (p_deptno);
      EXCEPTION
         WHEN OTHERS THEN NULL;
      END;
   END;
END;
/
```

Ako sada ponovno izvedemo naredbu:

```
INSERT INTO dept (deptno, dname) VALUES (1, 'DEPT 1');
```

dobit ćemo sa SELECT upitom sljedeće:

```
     EMPNO ENAME                DEPTNO DNAME
---------- --------------- ---------- --------------
         1 EMP 1                    1 DEPT 1
         2 EMP 2                    1 DEPT 1
```

Dakle, dobili smo dobar rezultat, kao i sa SAVEPOINT / ROLLBACK TO SAVEPOINT.

Naravno, mogli smo raditi i na (barem) dva druga načina:

a) Jedan način je da proceduru insert_programmers radimo kao AUTONOMOUS_TRANSACTION. Mana je da se (istina, vrlo rijetko) može desiti da autonomna transakcija uspije, a glavna ne, pa bismo imali 2 retka EMP = PROGRAMER bez 1 retka DEPT i 2 retka EMP = MANAGER (naravno, tako nešto ne bi uspjelo ako postoji FK sa EMPO na DEPT)

b) Drugi način je da se u proceduri insert_programmers doda EXCEPTION u kojem se pokušava poništiti djelomični rezultat:

```
   PROCEDURE insert_programmers (p_deptno dept.deptno%TYPE) IS
   BEGIN
      INSERT INTO emp (empno, ename, job, mgr, sal, deptno)
         VALUES (3, 'EMP 3', 'PROGRAMER', 1, 1000, p_deptno);

      RAISE_APPLICATION_ERROR
         (-20001, 'SIMULATED ERROR IN MIDLE OF 3.PART OF A TRANSACTION');

      INSERT INTO emp (empno, ename, job, mgr, sal, deptno)
         VALUES (4, 'EMP 4', 'PROGRAMER', 1, 1000, p_deptno);
   EXCEPTION
      WHEN OTHERS THEN
         DELETE emp WHERE empno = 3;
   END;
```

Međutim, moguće je (istina, vrlo rijetko) da i to poništavanje ne uspije. Osim toga, ovakvo poništavanje moglo bi u realnoj situaciji biti vrlo kompliciranо.

Ovo je programsko rješenje bilo prikazano na web stranicama firme Quest u 6.mjesecu 2002. (stranica više nije dostupna), te ukratko (bez detalja) na HROUG-u 2002.

1.9. Kako generirati dokumente bez rupa u brojevima

Što napraviti ako zakonski propisi određuju da brojevi (npr.) računa budu bez rupa, tj. da ne nedostaje niti jedan broj. Nažalost, Oracle sekvence to ne osiguravaju. No možemo napraviti naš vlastiti generator brojeva, tj. tablicu i pripadajuću funkciju:

```
CREATE TABLE sequence_generator (
   table_name VARCHAR2 (30) PRIMARY KEY,
   id         NUMBER (32) NOT NULL)
/

CREATE OR REPLACE FUNCTION generated_id (p_table VARCHAR2) RETURN NUMBER
IS
   l_generated_id NUMBER;
   l_table_name   VARCHAR2 (30) := UPPER (p_table);
BEGIN
   BEGIN
      SELECT id INTO l_generated_id
        FROM sequence_generator
       WHERE table_name = l_table_name
         FOR UPDATE; -- zaključava tekući redak

      l_generated_id := l_generated_id + 1;

      UPDATE sequence_generator
         SET id = l_generated_id
       WHERE table_name = l_table_name;
   EXCEPTION
       WHEN NO_DATA_FOUND THEN
          l_generated_id := 1;
          INSERT INTO sequence_generator (table_name, id)
             VALUES (l_table_name, l_generated_id);
   END;
   RETURN l_generated_id;
END;
/
```

Funkciju možemo koristiti npr. u PRE-INSERT Forms okidaču, ali bolje je koristiti okidač baze:

```
CREATE OR REPLACE TRIGGER bir_dept
   BEFORE INSERT ON dept
   FOR EACH ROW
BEGIN
   :NEW.deptno := generated_id ('DEPT');
END;
/
```

Koristit ćemo i POST-INSERT Forms okidač (alternativa je da se svojstvo bloka DML Returning Value postavi na Yes, kako je navedeno u potpoglavlju 2.2.) za osvježavanje polja na ekranu, u skladu sa bazom (polje :dept.deptno mora imati svojstva Query Only = Yes i Insert / Update Allowed = No):

```
SELECT deptno INTO :dept.deptno -- server-derived column
  FROM dept
 WHERE ROWID = :dept.ROWID;
```

Ova tehnika osigurava da se novi broj iskoristi samo ako transakcija uspješno završi i time se osiguravaju brojevi bez rupa.

No ako na klijent strani (npr. Forms ili ADF) radimo POST proces (tj. DML naredbe, bez COMMIT), onda nastaje veliki problem – druge sesije moraju čekati da ova sesija završi i otključa redak tablice za generiranje brojeva. Postoji rješenje i za takvu situaciju. Ideja je da se koriste dva generatora sekvenci – jedan generator za privremeni broj (taj generator može biti Oracle sekvenca, jer nam rupe ovdje nisu važne) i drugi generator za konačni broj dokumenta.

Privremeni broj punit ćemo u BIR (Before Insert Row) okidaču baze, a krajnji broj u BUR (Before Update Row) okidaču, na kraju transakcije. No kako će BUR okidač znati da je riječ o kraju transakcije? Moramo u tablicu dokumenata dodati stupac imena npr. "record_status" i pomoću okidača baze osigurati se da redak koji ima status "V" (Valid) ima brojeve bez rupa. Pretpostavimo da "record_status" može imati tri vrijednosti: N(ew), V(alid) i C(hanged), te da su validne tranzicije: N => V, V => C, C => V, N => N, V => V, C => C:

```
ALTER TABLE dept ADD (
   record_status CHAR (1) DEFAULT 'N' NOT NULL,
   CONSTRAINT dept_record_status_ck
      CHECK (record_status IN ('N', 'V', 'C')))
/
CREATE OR REPLACE TRIGGER bir_dept
   BEFORE INSERT ON dept FOR EACH ROW
BEGIN
   :NEW.record_status := 'N';
   SELECT test_seq.NEXTVAL INTO :NEW.deptno FROM DUAL; -- privremeni id
END;
/
CREATE OR REPLACE TRIGGER bur_dept
   BEFORE UPDATE ON dept FOR EACH ROW
BEGIN
   IF :NEW.record_status IS NULL THEN
      RAISE_APPLICATION_ERROR
         (-20001, 'Record status ne može biti NULL!');
   END IF;
   IF NOT (
      :OLD.record_status = 'N' AND :NEW.record_status = 'V' OR
      :OLD.record_status = 'V' AND :NEW.record_status = 'C' OR
      :OLD.record_status = 'C' AND :NEW.record_status = 'V' OR
      :OLD.record_status = :NEW.record_status)
   THEN
      RAISE_APPLICATION_ERROR (-20002, 'Pogrešna tranzicija!');
   END IF;
   IF :OLD.record_status = 'N' AND :NEW.record_status = 'V' THEN
      :NEW.deptno := generated_id ('DEPT'); -- konačni id
   END IF;
END;
/
```

Na Forms strani, u Post-Forms-Commit okidaču (koji se okida neposredno prije nego što Forms pošalje naredbu COMMIT na bazu), napišimo sljedeći kod:

```
BEGIN
   IF :SYSTEM.FORM_STATUS = 'QUERY' THEN RETURN; END IF;

   UPDATE dept
      SET record_status = 'V'
    WHERE deptno = :dept.deptno
      AND record_status = 'N';

   SELECT deptno INTO :dept.deptno -- server-derived column
     FROM dept
    WHERE ROWID = :dept.ROWID;
END;
```

Ako Forms (ili neki drugi alat) ne bi napravio navedeni UPDATE (prije COMMIT naredbe), na bazi bi mogli ostati dokumenti koji bi imali vrijednost "N" u statusu i privremeni broj (dokumenta). Napominjemo da u praksi ne bismo pisali PL/SQL kod direktno u okidačima, nego u (pakiranim) procedurama / funkcijama, te ne bismo direktno koristili RAISE_APPLICATION_ERROR.

Ovo je programsko rješenje bilo prikazano na web stranicama firme Quest u 11.mjesecu 2002. (stranica više nije dostupna), te na HROUG-u 2003.

1.10. Nove mogućnosti u bazi 12c (vezano za transakcije)

Vezano za transakcije, najvažnije nove mogućnosti u bazi 12c jesu Transaction Guard i Application Continuity (koja se temelji na Transaction Guard mogućnosti). Treba naglasiti da obje mogućnosti ima samo Enterprise edicija baze, dok ih Standard edicija nema. Osim toga, Application Continuity traži i Active Data Guard opciju ili Real Application Clusters opciju EE baze.

Transaction Guard rješava jedan veliki mogući problem u transakcijama. Do sada, kada je klijent baze (to može biti i aplikacijski server, pa čak i pohranjena PL/SQL procedura) poslao bazi COMMIT naredbu, i ako je baš tada došlo do pada veze, klijent je dobio informaciju da je veza pala, ali ne i informaciju da li je COMMIT uspješno napravljen, ili nije. Klijent nije mogao niti naknadno dobiti tu informaciju (osim, u nekim slučajevima, pomoću relativno složenih aplikacijskih rješenja). Zbog toga se moglo desiti da klijent (softver ili korisnik) pokrene dva puta istu transakciju, jer ne zna da je prethodna uspješno završila, ili ne pokrene niti jedanput. Transaction Guard zasniva se na tome što se u trenutku COMMIT-a pamti tzv. logical transaction identifier (LTXID), koji se kasnije može koristiti za biranje odgovarajućeg postupka u aplikaciji, kako bi se osiguralo da se neka transakcija neće izvršiti dva puta, a eventualno hoće jedanput. Sljedeći pseudokod iz [13] pokazuje tipično korištenje Transactional Guard mogućnosti:

```
Receive a fast application notification (FAN) down event
   (or recoverable error)
FAN aborts the dead session
IF recoverable error (new OCI_ATTRIBUTE for OCI, isRecoverable for JDBC)
   Get last LTXID from dead session using getLTXID or from your callback
   Obtain a new session
   Call GET_LTXID_OUTCOME with last LTXID
         to obtain COMMITTED and USER_CALL_COMPLETED status
IF COMMITTED and USER_CALL_COMPLETED
   Then return result
ELSEIF COMMITTED and NOT USER_CALL_COMPLETED
   Then return result with a warning
         (that details such as out binds or row count were not returned)
ELSEIF NOT COMMITTED
   Cleanup and resubmit request, or return uncommitted result to the client
```

Application Continuity temelji se na Transaction Guard mogućnosti, i omogućava da se automatsko nastavljanje nakon popravljivih (recoverable) grešaka napravi bez pisanja složenog programskog koda.

Osim navedenih (vrlo značajnih) mogućnosti, u bazi 12c novost je da se tzv. Basic Flashback Data Archive mogućnost sada može koristiti i u Standard ediciji baze. Do sada su se u SE ediciji mogle koristiti samo Flashback Query i Flashback Version Query mogućnosti (Flashback Transaction Query, Flashback Transaction Backout, Flashback Table, Flashback Drop, Flashback Database, i dalje se mogu koristiti samo u EE ediciji). Flashback Data Archive, zapravo, nije tehnologija koja je korisna sama po sebi, već nadograđuje ostale Flashback tehnologije. Naime, sve one (osim Flashback Database) rade na logičkoj razini i koriste UNDO tablespace. Budući da se UNDO tablespace koristi za normalan rad baze, on lako postaje vrlo velik. Ako želimo pratiti dugotrajnu povijest određene tablice, ili nekog skupa tablica, Flashback Data Archive omogućava da se za to odredi posebni tablespace (ili dio nekog tablespace-a). Za svaku tablicu treba unaprijed reći da se ona sprema u Flashback Data Archive.

Na kraju, napomenimo da je Oracle 12c donio i jednu (relativno) negativnu novost. Od baze 11.1 postoji PL/SQL paket DBMS_XA. Taj paket predstavlja PL/SQL sučelje za realizaciju modela X/Open Distributed Transaction Processing Model. Inače, XA standard je X/Open specifikacija za procesiranje distribuiranih transakcija (distributed transaction processing) između heterogenih izvora podataka, npr. između Oracle i DB2 baza podataka, a objavljen je 1991. Zapravo, važnije od DBMS_XA paketa (koji je potreban ako se želi programirati isključivo kroz PL/SQL) je takav način rada Oracle baze od verzije 10.2 (zapravo, sa patchevima i od verzije 9.2.0.6.), kod kojeg se u distribuiranim transakcijama nikada ne dešava zaključavanje redaka za čitatelje (tj. za SELECT naredbe), što je inače mana standardne Oracle distribuirane transakcije (kako je kratko navedeno u potpoglavlju 1.4.). No 19. poglavlje u [13] počinje napomenom da korištenje XA treba izbjegavati, ako je moguće: "The use of XA should be avoided, if possible. In some cases (for example, if Oracle and non-Oracle resources must be used in the same transaction) it may seem unavoidable, but even in some of these cases it may be possible to avoid using XA. The use of XA can cause performance issues and lead to in-doubt transactions. It also might not be able to take advantage of certain Oracle Database 12c Release 1 (12.1.0.1) features that enhance the ability of applications to continue after recoverable outages."

2. TRANSAKCIJE I ORACLE FORMS

2.1. Forms - razvoj, varijante, arhitektura

Oracle Forms je Rapid Application Development (RAD) alat, koji je Oracle korporacija napravila početkom 80-ih, nedugo nakon nastanka Oracle baze verzije 2 (verzija 1 nije nikada postojala). Njegovo prvo ime bilo je Interactive Application Facility (IAF), i radio je kao znakovno orijentirana (character mode) aplikacija na serveru. Nakon toga, otprilike kod pojave verzije 4 Oracle baze, ime mu promijenjeno u FastForms, a u vrijeme baza 5 i 6 dobio je ime SQL Forms (Forms verzije 2 i 3). Nakon toga (nešto prije pojave baze 7) ime mu je postalo (i ostalo) Oracle Forms, te je postao klijent-server GUI alat.

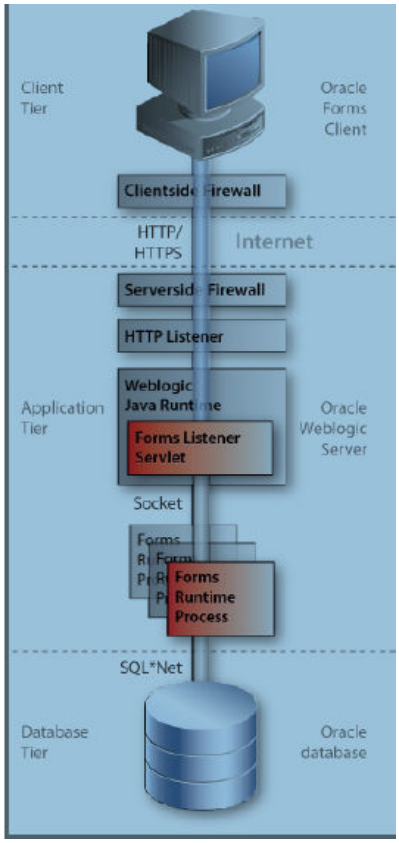
Klijent-server varijanta pratila je baze 6, 7 i 8, a Forms verzije bile su 4, 4.5, 5, 6 i 6i. U Forms verziji 6 pojavila se paralelno i web varijanta - Web Forms. Forms verzije 7 i 8 nikad nisu postojale, jer je broj verzije "skočio" na 9, kako bi pratio verzije Oracle baze. Nakon verzije 6i, klijent-server varijanta više ne postoji, tj. verzije 9i, 10g, 10gR2, 11g i 11gR2 (u 2014. se očekuje pojava verzije 12c) rade isključivo kao web Forms aplikacije. Oracle Forms je izvorno pisan u jeziku C, a i sadašnji kod je uglavnom C kod, osim Java apleta (koji se izvršava u pregledniku), te određenih pomoćnih dijelova na aplikacijskom serveru.

Iako Oracle preferira jezik Java i Oracle Application Development Framework (ADF) za svoje Oracle Fusion aplikacije, materijal "Oracle's development tools statement of direction" jasno kaže da će Oracle i dalje nastaviti podržavati Oracle Forms (za razliku od npr. CASE alata Oracle Designer, koji se prestao razvijati nakon verzije 10g) i razvijati ga u sljedećim područjima:

- dodavati nove mogućnosti, koje su važne za web, na što elegantniji način;
- omogućiti da Forms i Reports aplikacije što više koriste mogućnosti servisa aplikacijskog servera i da što lakše rade sa Java EE aplikacijama.

Npr., već Oracle Forms 6i omogućava prilagodljivi UI kroz Pluggable Java Components (PJCs). Forms 11g uveo je mogućnost obostrane interakcije Forms apleta sa ostalim aplikacijama na strani Internet preglednika, pomoću Javascript koda. Također, Forms 11g omogućava obradu eksternih događaja, kao što su asinkroni događaji, korištenjem redova (database queue) baze 10g R2 ili veće, pomoću novog Forms okidača When-Event-Raised. U Forms 11g važna je i podrška za Unicode stupce.

Oracle Fusion Middleware Forms Services sastoji se od tri komponente: Forms Client, Forms Listener Servlet, Forms Runtime. Slika 2.1. prikazuje osnovnu arhitekturu Oracle Fusion Middleware Forms Services.



Forms klijent (Forms Client) je 100%-tni Java aplet, koji se dinamički učitava sa Oracle Fusion Middleware aplikacijskog servera. On ostvaruje korisničko sučelje, tj. upravlja interakcijom sa korisnikom i "crta ekrane". Da bi Java aplet mogao raditi, korisnikovo računalo mora imati instaliran JVM.

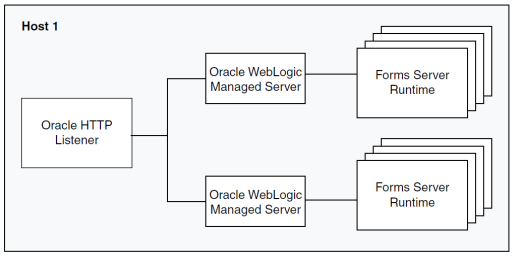
Forms Listener Servlet upravlja kreiranjem i stopiranjem Forms Runtime procesa za svakog korisnika, te mrežnom komunikacijom između Forms klijenta i pripadajućeg Forms Runtime procesa.

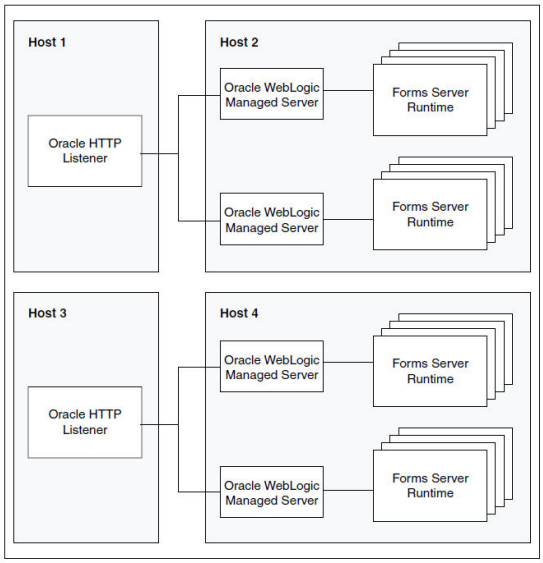
Forms Runtime proces upravlja vezom sa bazom podataka, na isti način kako to radi klijent-server Forms varijanta.

Slika 2.1. Arhitektura Oracle Fusion Middleware Forms Services; Izvor [14]

Moguće su različite kombinacije korištenja Oracle Forms Runtime procesa, Oracle HTTP Listenera i Oracle WebLogic servera, na različitim hostovima (fizičkim ili virtualnim serverima).

Npr., lijeva strana slike 2.2. prikazuje slučaj gdje jedan host sadrži jedan HTTP Listener i dva WebLogic servera (svaki WebLogic server podržava više Forms Runtime procesa). Desna strana slike 2.2. prikazuje slučaj gdje dva hosta sadrže po jedan HTTP Listener, a druga dva hosta sadrže po dva WebLogic servera. Moguće su i drugačije varijante.

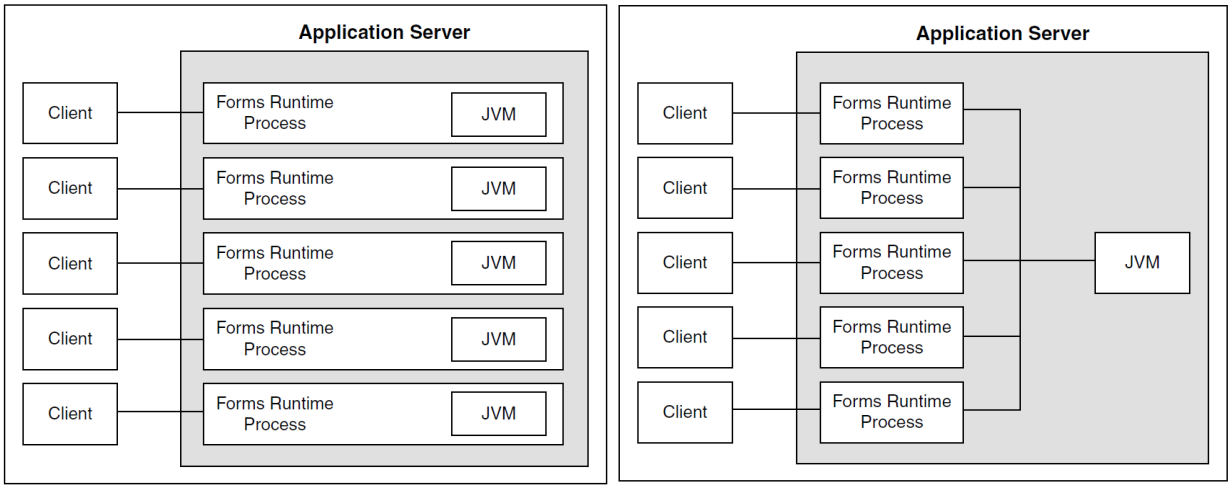




Slika 2.2. Više Oracle WebLogic servera na istom hostu na kojem je Oracle HTTP Listener (lijevo); Više Oracle WebLogic servera i više Oracle HTTP Listenera na različitim hostovima (desno); Izvor [15]

Kada Forms Runtime proces koristi Javu na strani aplikacijskog servera, moguće su dvije glavne varijante korištenja Java Virtual Machine (JVM), te različite međuvarijante:

- Svaki Forms Runtime proces koristi svoj vlastiti JVM, kako prikazuje lijeva strana slike 2.3. Ako svaki Forms Runtime proces koristi JVM samo mali dio vremena, onda se u ovoj varijanti nepotrebno troše memorijski resursi aplikacijskog servera;
- Svi Forms Runtime procesi koriste isti JVM, kako prikazuje desna strana slike 2.3. Koristi se tzv. JVM pooling, koji je odvojeni proces, i koji sadrži JVM kontroler. Sa JVM poolingom, JVM se izvodi izvan Forms Runtime procesa. Kad Forms Runtime proces zatreba izvršavanje Java koda, šalje poruku JVM-u koji je sadržan u JVM kontroleru, pa JVM kreira novu Java dretvu (thread) za njega. Forms Runtime proces cijelo vrijeme koristi tu istu Java dretvu. To rezultira znatnim smanjenjem utroška memorije aplikacijskog servera.
- Različite međuvarijante, npr. jedna skupina Forms Runtime procesa koristi jedan zajednički JVM, druga skupina koristi drugi zajednički JVM, a neki Forms Runtime procesi koriste svoj vlastiti JVM.



Slika 2.3. Forms Runtime procesi bez korištenja JVM poolinga-a (lijevo) i Forms Runtime procesi sa korištenjem JVM poolinga-a (desno); Izvor [15]

2.2. Neka svojstva Forms modula i Forms bloka (vezano za transakcije)

Forms blokovi čine najvažnije dijelove Forms modula. Forms blok, kao i Forms modul, može imati Forms okidače (triggers). Forms blok sadrži polja (items; ona isto mogu imati vlastite okidače) i retke. U ovom potpoglavlju ukratko će se prikazati svojstva Forms modula i Forms bloka, koja su vezana za transakcije.

Svojstva Forms modula

Validation Unit

Specificira kada će se izvršiti validacija. Moguće vrijednosti su polje, redak, blok i Forms modul (item, record, block, forms). Podrazumijevana (default) vrijednost za većinu platformi je polje, što znači da će se validacija pokrenuti kod izlaska iz polja.

Maximum Query Time

Omogućava da se upit (query) prekine kad mu proteklo vrijeme premaši zadanu vrijednost. Ovo svojstvo ima i Forms blok. Korisno je samo kad je svojstvo Query All Records property postavljeno na Yes.

Maximum Records Fetched

Omogućava da se upit (query) prekine kad broj prenesenih (fetched) redaka premaši zadanu vrijednost. Ovo svojstvo ima i Forms blok. Korisno je samo kad je svojstvo Query All Records property postavljeno na Yes.

Isolation Mode

Moguće vrijednosti su Read Committed (podrazumijevana vrijednost) i Serializable.

Vrijednost Serializable određuje da će transakcije u sesiji (baze) biti serijabilne, tj. da će transakcija od početka do kraja vidjeti uvijek iste podatke drugih transakcija (u drugoj sesiji) čak i ako su te (druge) transakcije napravile COMMIT. Serijabilni mod nije dobar ako postoji veća šansa da dvije transakcije različitih sesija modificiraju iste retke, jer se tada često javlja greška ORA-08177: Cannot serialize access. Ako se postavi vrijednost Serializable, poželjno je svojstvo bloka Locking Mode postaviti na Delayed.

Svojstva Forms bloka

Query Array Size, DML Array Size

Query Array Size specificira maksimalni broj redaka koje Forms odjednom dohvaća (fetch) sa baze. Podrazumijevana vrijednost je broj redaka koji se prikazuju na bloku (svojstvo Number of Records Displayed). Mala vrijednost (npr. 1) opterećuje komunikaciju između klijenta i baze.

Slično tome, DML Array Size specificira veličinu niza (array) kojim se odjedanput radi unos, izmjena ili brisanje više redaka na bazi.

Number of Records Buffered

Specificira minimalni broj redaka (za blok) koji će se spremiti u memoriju. Forms smješta retke iznad tog broja u privremenu datoteku na disku, što može značajno usporiti rad (to ovisi i o operacijskom sustavu). S druge strane, veći memorijski buffer traži veće memorijske resurse. Podrazumijevana vrijednost je NULL, a ona zapravo označava da će broj redaka u memoriji biti jednak onome definiranom u svojstvu Number of Records Displayed property, uvećanom za 3.

Query All Records, Precompute Summaries

Query All Records specificira da li se svi redovi koji odgovaraju upitu nad blokom trebaju učitati odjednom, kod izvršavanja upita. Podrazumijevana vrijednost je No. Vrijednost Yes korisno je postaviti kad imamo sumarna polja, za koja želimo da odmah prikažu ispravnu vrijednost (a ne tek kada učitamo sve retke).

Umjesto toga, može se raditi i tako da se svojstvo Precompute Summaries postavi na Yes, čime se određuje da se vrijednost svakog sumarnog polja na bloku izračunava prije nego što se izvrši standardni upit na bloku.

Database Block

Specificira da je blok temeljen na nekom izvoru podataka nad kojim se (izvorom) mogu raditi upiti i/ili DML naredbe. Ako je svojstvo označeno sa No, to znači da je riječ o kontrolnom bloku.

Query Allowed, Insert Allowed, Update Allowed, Delete Allowed

Ova četiri svojstva (koja imaju i polja) specificiraju da li se nad blokom može postaviti upit, odnosno odgovarajuća DML naredba.

Query Data Source Type, DML Data Target Type

Query Data Source Type specificira tip izvora podataka za upite: Table, Procedure, Transactional Trigger, FROM clause query. Kod opcije Table, umjesto tablice može se "podmetnuti" i view (baze). Čak je i preporučljivo (u većini slučajeva) umjesto tablice koristiti view, jer se time može izbjeći potreba za POST-QUERY Forms okidačima, koji uglavnom služe za naknadno dohvaćanje redaka ne-osnovnih tablica (lookup tablica). Naime, nakon što se upit nad osnovnom tablicom već izvršio, za svaki dohvaćeni redak osnovne tablice, POST-QUERY okidač treba izvršiti poseban upit za lookup tablicu (ili više njih, ako ima više lookup tablica). Također, korištenjem viewa može se izbjeći potreba za PRE-QUERY okidačima, koji (između ostalog) služe za sortiranje po poljima lookup tablica (ta polja predstavljaju nebazna polja).

DML Data Target Type specificira tip podatkovnog odredišta za DML naredbe: Table, Procedure, Transactional Trigger. Umjesto tablice, kao Table se isto može podmetnuti view (baze). On može biti view koji se može ažurirati (updateable view), ili može biti view koji ima odgovarajuće INSTEAD OF okidače baze (koji zamjenjuju standardne DML naredbe na bazi).

Locking Mode

Specificira kada će Forms pokušati zaključati retke (na bazi) koje korisnik mijenja u Forms bloku. Postoje tri vrijednosti: Immediate, Delayed i Automatic. Automatic je podrazumijevana vrijednost, ali nad Oracle bazom znači isto što i Immediate.

Vrijednost Immediate znači da će Forms zaključati redak (naredbom SELECT ... FOR UPDATE) odmah nakon što je korisnik počeo nešto mijenjati na bilo kojem polju retka. U tom slučaju nije moguće da neki drugi korisnik "pregazi" promjenu koju je napravio prvi korisnik.

Vrijednost Delayed označava da će se mijenjani redak zaključati tek neposredno prije procesa postiranja (tj. kad Forms pošalje bazi DML naredbe). Ako je u ovom slučaju neki drugi korisnik već mijenjao redak, Forms upozorava korisnika da je redak već mijenjan i ne dozvoljava COMMIT - korisnik mora dati upit za uvid u novo stanje.

Kaže se da je Immediate način rada "jedini pravi način rada za web aplikacije", i takav je način rada podrazumijevani (default) kod web aplikacija, npr. kod ADF JSF aplikacija.

Update Changed Columns Only

Označava da se naredbom UPDATE na bazu šalju samo mijenjani stupci (zapravo, polja), a ne svi stupci u Forms bloku. Podrazumijevana vrijednost je No, što znači da se standardno na bazu šalju svi stupci. Ako je svojstvo DML Array Size postavljeno na vrijednost veću od 1, onda se uvijek podrazumijeva No, tj. ignorira se ako je Update Changed Columns Only postavljeno na Yes.

Kada je Update Changed Columns Only postavljeno na No, Oracle baza može koristiti istu UPDATE naredbu za svaki redak, bez potrebe za ponovnim parsiranjem UPDATE naredbe. Stoga je preporučljivo postaviti svojstvo na Yes samo u dva slučaja. Jedan je slučaj kada ne želimo na bazu slati velika (LONG) polja, ako ta polja korisnik nije mijenjao. Drugi je slučaj ako ne želimo da se na bazi izvršavaju okidači koji su vezani za određena polja, ako ta polja stvarno nisu mijenjana.

DML Returning Value

Poznato je da još od baze 8, DML naredbe sadržavaju tzv. DML Returning clause, kojom baza može vratiti vrijednosti (nekih) polja iz tekućeg retka, a uglavnom je riječ o onim poljima čije je vrijednosti baza sama mijenjala, najčešće pomoću okidača baze (npr. polje ID, audit polja i dr.).

Podrazumijevana vrijednost svojstva DML Returning Value je No. Ako se postavi na Yes, Forms automatski ažurira vrijednosti (mijenjane na bazi) tih polja kod sebe. Na taj način nije potrebno da programer sinkronizira ("osvježava") ta Forms polja pomoću POST-INSERT / POST-UPDATE Forms okidača. Ako se DML Returning Value ostavi na No, a programer ne napravi sinkronizaciju pomoću POST-INSERT / POST-UPDATE Forms okidača, onda će korisnik imati dva problema, Ne samo da neće vidjeti nove vrijednosti tih polja dok ponovo ne pokrene upit (re-query), nego će, ako ne izvrši ponovni upit, dobiti čudnu grešku ako ponovno mijenja redak koji je malo prije sam mijenjao (i COMMIT-irao na bazu): FRM-40654: Record has been updated by another user. Re-query to see change.

No, da bi postavka DML Returning Value = Yes imala efekta (tj. da bi Forms koristio Returning clause), Forms blok mora sadržavati ROWID, tj. svojstvo (bloka) Key Mode mora biti postavljeno na Automatic ili Unique. Također, Forms ne koristi Returning clause kod procesiranja LONG polja. U takvim slučajevima treba i dalje koristiti sinkronizaciju pomoću POST-INSERT / POST-UPDATE Forms okidača, kao što se radilo prije pojave baze 8, odnosno Formsa 6 (Forms 5 nije imao svojstvo DML Returning Value). Osim toga, nelogično je da se poljima generiranim na bazi mora postaviti vrijednost Query Only = No, da bi DML Returning Value = Yes radilo.

2.3. Svojstva tipičnih vrsta tekstualnih polja (text item)

U ovom potpoglavlju prikazuju se svojstva različitih vrsta tekstualnih polja (text item) iz prakse. Ne prikazuju se samo svojstva vezana za transakcije, nego i svojstva vezana npr. za prikaz na ekranu, kako bi se lakše uvidjela razlika između tih vrsta polja.

Pretpostavimo da je blok temeljen na tablici (što se često radi), a ne na viewu (iako bi to bilo poželjno u dosta slučajeva). Dakle, imamo polja osnovne tablice, koja su bazna polja, i polja ostalih tablica (lookup tablica), koja su nebazna polja (napomena: nebazna su i polja koja ne pripadaju nijednoj tablici). Vrlo često, Forms blok sadrži ova polja:

- ID osnovne tablice: generira se na bazi; uglavnom je PK;
- šifra osnovne tablice: uglavnom je UK;
- naziv osnovne tablice: može, i ne mora biti UK;
- FK polja osnovne tablice, tj. polja koja su veza na ID-ove tablica-roditelja;
- audit polja osnovne tablice: generiraju se na bazi;
- druga polja osnovne tablice;
- polja lookup tablica: nebazna, najčešće su to (barem) šifra i naziv.

Tablica 2.1. prikazuje neka svojstva tipičnih vrsta tekstualnih polja. Svojstva su data u redovima, a tipične vrste tekstualnih polja u stupcima.

Vrsta polja => Svojstvo	ID i FK polja osnovne tablice	Šifra, naziv i ostala polja osnovne tablice	Audit polja osnovne tablice	Polja lookup tablica (npr. šifra i naziv)
Enabled	nije važno, ID nije vidljiv	Yes	Yes	Yes
Keyboard navigable	nije važno, ID nije vidljiv	Yes	No	šifra = Yes; ostala polja = No
Database item	Yes	Yes	Yes	No
Query only (ako je Yes, polje nije uključeno u DML)	vlastiti ID = Yes; FK polja = No	No	Yes	nije važno, nije bazno polje
Query allowed	No	Yes	Yes	Yes
Insert/update allowed	No	Yes	No	šifra = Yes; ostala polja = No
Visible	No	Yes	Yes	Yes

Tablica 2.1. Svojstva tipičnih vrsta tekstualnih polja

Tablica 2.2. prikazuje slične podatke kao prethodna tablica, samo što ih prikazuje u obliku koji je uobičajen kod tablica odlučivanja: tipične vrste polja, koje odgovaraju unesenim vrijednostima u određenom stupcu, date su u zadnjem retku (na mjestu gdje se u tablici odlučivanja prikazuju akcije).

Database item	Y	Y	Y	Y	N	N
Query Only	N	N	Y	Y	-	-
Visible	Y	N	Y	N	Y	N
Operator može raditi Insert/Update	DA	NE (not visible)	NE (query only)	NE (not visible + query only)	DA	NE (not visible)
Programski se može raditi Insert/Update	DA	DA	NE (query only)	NE (query only)	DA	DA
Tipična vrsta polja (sa takvim vrijednostima svojstava)	obično polje iz bazne tablice	surogatni ključ (ID) bazne tablice, punjen na klijentu	audit polje bazne tablice ili polje iz nebazne join tablice	surogatni ključ (ID) bazne tablice, punjen na bazi	polje iz lookup tablice; ne unosi se, osim šifra (deskriptor)	pomoćno polje

Tablica 2.2. Svojstva tipičnih vrsta tekstualnih polja, prikazana na još jedan način

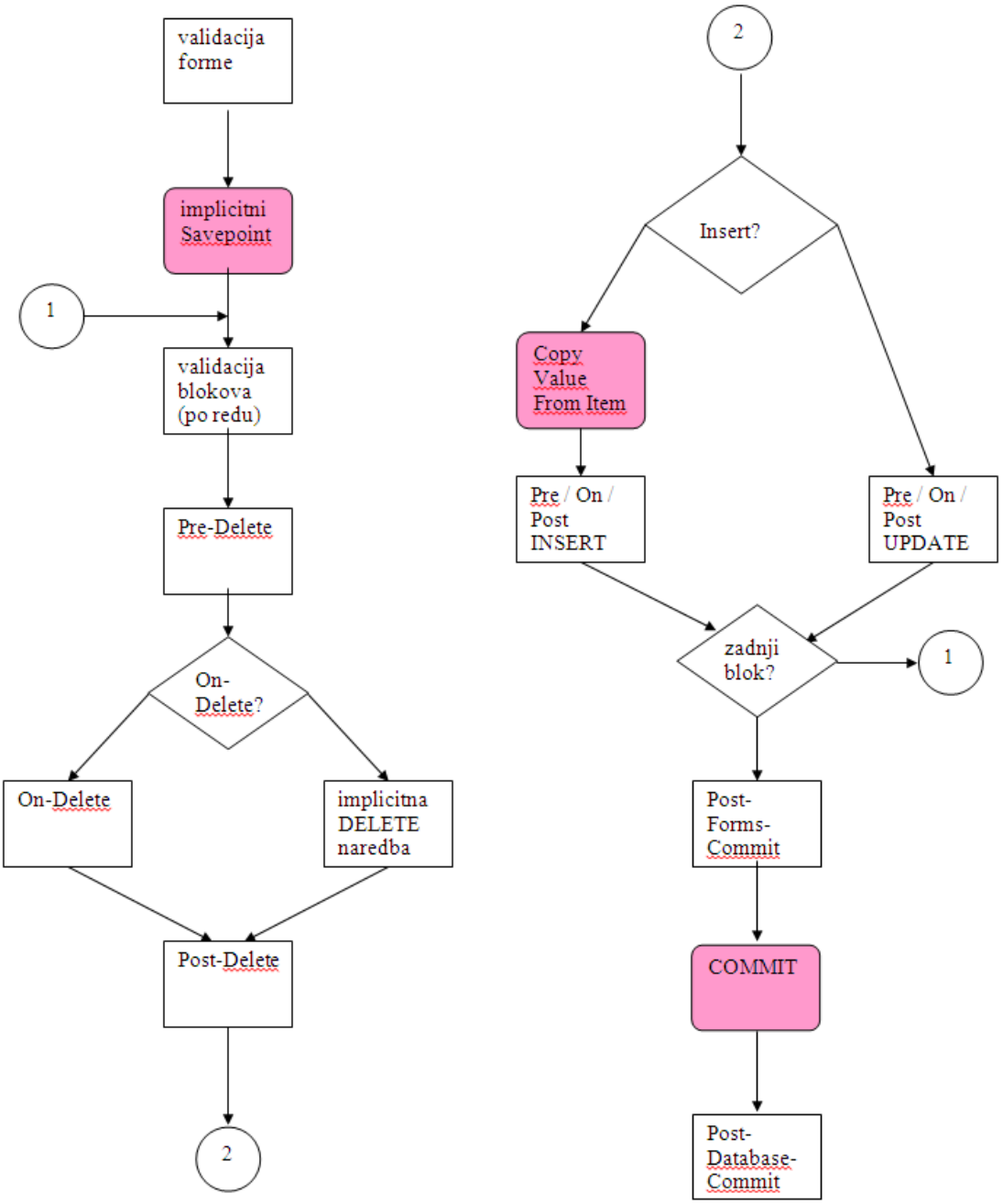
2.4. COMMIT i POST Forms procesi

Među najvažnijim, ali i najsloženijim Forms procesima, jesu COMMIT i POST procesi. Zapravo, COMMIT proces može se jednostavno (i točno) prikazati kao:

COMMIT proces = POST proces + COMMIT naredba na bazi + Forms okidač Post-Database-Commit

pa je dovoljno opisati POST proces.

Suština je u tome da se ažuriranje redaka (unos / izmjena / brisanje) prvo radi samo unutar Forms bloka. Redovi u Forms bloku dobivaju (kako je prikazano u sljedećem potpoglavlju) odgovarajući transakcijski status. Kad korisnik odabere gumb Save ili funkcijsku tipku F10 (ili neki drugi način koji je postavio programer), Forms šalje odgovarajuće DML naredbe na bazu, te daje COMMIT na bazu (ako je riječ o COMMIT procesu). Slika 2.4. prikazuje detalje Forms COMMIT procesiranja (napomena: COMMIT procesiranje je puno detaljnije dijagramski opisano u Forms helpu verzije 6i; Forms help u verzijama 10g / 11g nema tih dijagrama).



Slika 2.4. Forms COMMIT procesiranje

Kako se vidi iz dijagrama, nakon validacije na razini Forms modula, Forms šalje bazi naredbu SAVEPOINT. U slučaju da nešto krene krivo, Forms implicitno (ili programer eksplicitno) kasnije može izvršiti ROLLBACK TO SAVEPOINT, sve do trenutka kad se bazi pošalje COMMIT. Zapravo, proces se može prekinuti i kod COMMIT-a, i to ne samo ako se koristi ON-COMMIT okidač (koji na dijagramu nije prikazan), već i ako se koriste odgođena integritetna ograničenja (vidjeti potpoglavlje 1.6.). Nakon implicitnog SAVEPOINT-a, radi se validacija na razini bloka (validacija je, sama po sebi, složeni proces).

Nakon validacije bloka, dolaze na red DML naredbe. Prvo se radi DELETE. Prije slanja DELETE naredbe izvršava se (ako postoji) Pre-Delete okidač. Nakon njega (ako postoji) izvršava se On-Delete okidač (zamjena za DELETE naredbu; uglavnom se koristi sa ne-SQL bazama) ili DELETE naredba (ako On-Delete okidač ne postoji). Nakon toga izvršava se Post-Delete naredba. Nakon obrade brisanih redaka (iz bloka), Forms obrađuje retke koji su uneseni ili mijenjani. Slično kao kod DELETE, mogu se koristiti Pre-Insert / Pre-Update, On-Insert / On-Update i Post-Insert / Post-Update okidači.

Važno je napomenuti da se prije Pre-Insert okidača automatski izvršava Copy Value From Item potproces. To je važno npr. za slučaj kad imamo master – detalj blokove (vidjeti potpoglavlje 2.6.), a master ima PK ili UK stupac (npr. imena ID), koji se puni na bazi, a za njega je vezan vanjski ključ (FK) tablice detalja. Zbivanja onda idu ovako: na bazu se uvijek prvo šalje redak master bloka, pri čemu baza generira vrijednost za PK. Ta se vrijednost pomoću Post-Insert okidača na master bloku (ili na temelju postavljenog svojstva DML Returning Value = Yes) puni u master blok. Kada (poslije) dođe vrijeme za INSERT detalja, potproces Copy Value From Item puni PK iz master bloka u odgovarajuće polje (vanjskog ključa) bloka detalja. Da nije toga, detalji se ne bi povezali sa pravim master retkom.

Nakon obrade jednog bloka, obrađuje se sljedeći. Kad završi obrada zadnjeg bloka, izvršava se (ako postoji) Post-Forms-Commit okidač. Njegovo ime djeluje prilično zbunjujuće, jer se može pomisliti da je prije njega već izvršen COMMIT na bazi (možda bi bolje ime za njega bilo npr. Pre-Database-Commit). U potpoglavlju 1.6. prikazana je uloga Post-Forms-Commit okidača kod provjere odgođenih integritetnih ograničenja na bazi.

Time završava POST procesiranje. Ako je riječ o COMMIT procesiranju, nakon toga se na bazu šalje COMMIT naredba, a poslije toga izvršava se (ako postoji) okidač Post-Database-Commit.

Za pokretanje COMMIT procesiranja postoji standardna Forms ikona (u meniju) Save i standardna funkcijska tipka F10 (naravno, može se odrediti da to bude druga tipka). Za pokretanje POST procesiranja ne postoji standardna Forms ikona i funkcijska tipka. Programer može sam kreirati npr. gumb u koji stavlja poziv naredbe POST (ali, gumb se uglavnom ne radi), ili POST naredbu ugrađuje u neki drugi dio programskog koda.

Ako se želi nadopuniti COMMIT procesiranje, onda se kreira Key-Commit okidač i u njega se upiše odgovarajući kod, koji sadrži i naredbu COMMIT_FORM (kojom se zapravo pokreće COMMIT procesiranje).

Treba naglasiti da Forms ima malo nekonzistentan način obrade grešaka.

Greške u "pravim" PL/SQL naredbama rješavaju se na isti način kao i greške u PL/SQL kodu na bazi. Usput, treba napomenuti da Forms ima svoj vlastiti PL/SQL interpreter (a SQL naredbe šalje bazi na obradu), koji je već odavno zaostao u odnosu na PL/SQL interpreter baze. Forms PL/SQL interpreter otprilike odgovara PL/SQL interpreteru baze 8.0.

Osim "pravog" PL/SQL koda, Forms sadrži i vlastite naredbe - Forms built-in naredbe. Te se naredbe kod obrade grešaka ne ponašaju na isti način kao "pravi" PL/SQL, već se ponašaju slično kao funkcije u jeziku C, koji nema obradu grešaka (exception handling; C++ ima obradu grešaka). Kod poziva Forms built-in naredbe, u slučaju greške Forms neće skočiti na EXCEPTION dio bloka (kao što će napraviti npr. kod greške u SQL naredbi). Programer je dužan (slično kao u jeziku C) nakon svakog poziva Forms built-in naredbe pomoću funkcije FORM_SUCESS provjeriti da li je Forms naredba uspjela, npr.:

```
ENTER; -- primjer Forms built-in naredbe
IF FORM_SUCESS THEN ...
```

No, da stvar bude još kompleksnija, postoje i tako složene Forms built-in naredbe kod kojih provjera pomoću FORM_SUCESS ne radi. Upravo su COMMIT_FORM i POST takve naredbe. Njihov (ne)uspjeh provjerava se tako da se vidi je li Forms modul u transakcijskom statusu QUERY (vidjeti statuse Forms objekata u sljedećem potpoglavlju). Ako nije, znači da se desila greška, jer još ima mijenjanih redaka:

```
COMMIT_FORM;
IF :SYSTEM.FORM_STATUS <> 'QUERY' THEN
  MESSAGE ('An error prevented your changes from being committed.'); ...
```

Spomenimo još da postoji i okidač On-Lock koji (ako postoji) zamjenjuje standardno zaključavanje koje se pokreće kada Forms šalje bazi naredbu SELECT FOR ... UPDATE LOCK, ili kada se izvršava Forms built-in naredba LOCK_RECORD.

2.5. Transakcijski i validacijski status Forms objekata

Neki objekti Forms modula imaju transakcijski (ili COMMIT) status, koji pokazuju Forms runtimeu koje DML operacije treba napraviti za vrijeme COMMIT_FORM naredbe (ili POST naredbe). Objekti koji imaju transakcijski status jesu redak, blok i sam Forms modul. Tablica 2.1. prikazuje navedene objekte i moguće vrijednosti transakcijskog statusa.

Objekt => Transakcijski status	RECORD	BLOCK	FORMS MODUL
NEW	Sva polja u retku su prazna. Samo jedan redak u bloku može imati status NEW.	Postoji samo jedan redak i ima status NEW.	Svi blokovi imaju status NEW.
INSERT	Novi redak. Izvršava se DML naredba INSERT.	NE POSTOJI	NE POSTOJI
QUERY	Nepromijenjeni postojeći redak.	Barem jedan redak ima status QUERY, a jedan redak može imati status NEW.	Barem jedan blok ima status QUERY, a ostali blokovi imaju status NEW.
CHANGED	Promijenjeni postojeći redak. Izvršava se DML naredba UPDATE.	Barem jedan redak ima status INSERT, CHANGED, ili je izbrisan (na formi).	Barem jedan blok ima status CHANGED.

Tablica 2.3. Transakcijski status retka, bloka i Forms modula

Transakcijski status može se čitati na dva načina:

- pomoću sistemskih varijabli, koje daju informaciju o TEKUĆEM retku / bloku / formi: :SYSTEM.RECORD_STATUS / :SYSTEM.BLOCK_STATUS / :SYSTEM.FORM_STATUS

- pomoću GET_RECORD_PROPERTY (… STATUS) / GET_BLOCK_PROPERTY (… STATUS) naredbi, koje daju informaciju o bilo kojem retku / bloku, a ne samo o tekućem.(napomena: :SYSTEM.RECORD_STATUS i GET_RECORD_PROPERTY (… STATUS) ne daju uvijek istu vrijednost, jer :SYSTEM.RECORD_STATUS ponekad može biti NULL).

Transakcijski status retka može se mijenjati pomoću naredbe SET_RECORD_PROPERTY (… STATUS …). Time se (indirektno) mijenja i transakcijski status bloka / forme.

Postoji i validacijski status, koji Forms runtimeu pokazuje da li treba napraviti validaciju nad tim objektom. Validacijski status imaju polje i (opet) redak, kako prikazuje tablica 2.4.

Objekt => Validacijski status	ITEM	RECORD
NEW	Prazno polje u praznom redu.	Sva polja u retku imaju status NEW.
CHANGED	Polje je mijenjano - potrebna je validacija.	Barem jedno polje u retku ima status CHANGED.
VALID	Polje nije mijenjano.	Sva polja u retku imaju status VALID.

Tablica 2.4. Validacijski status polja i retka

Između ostalog, validacijski status pokazuje i to da li se treba okinuti WHEN-VALIDATE-ITEM / WHEN-VALIDATE-RECORD okidači.

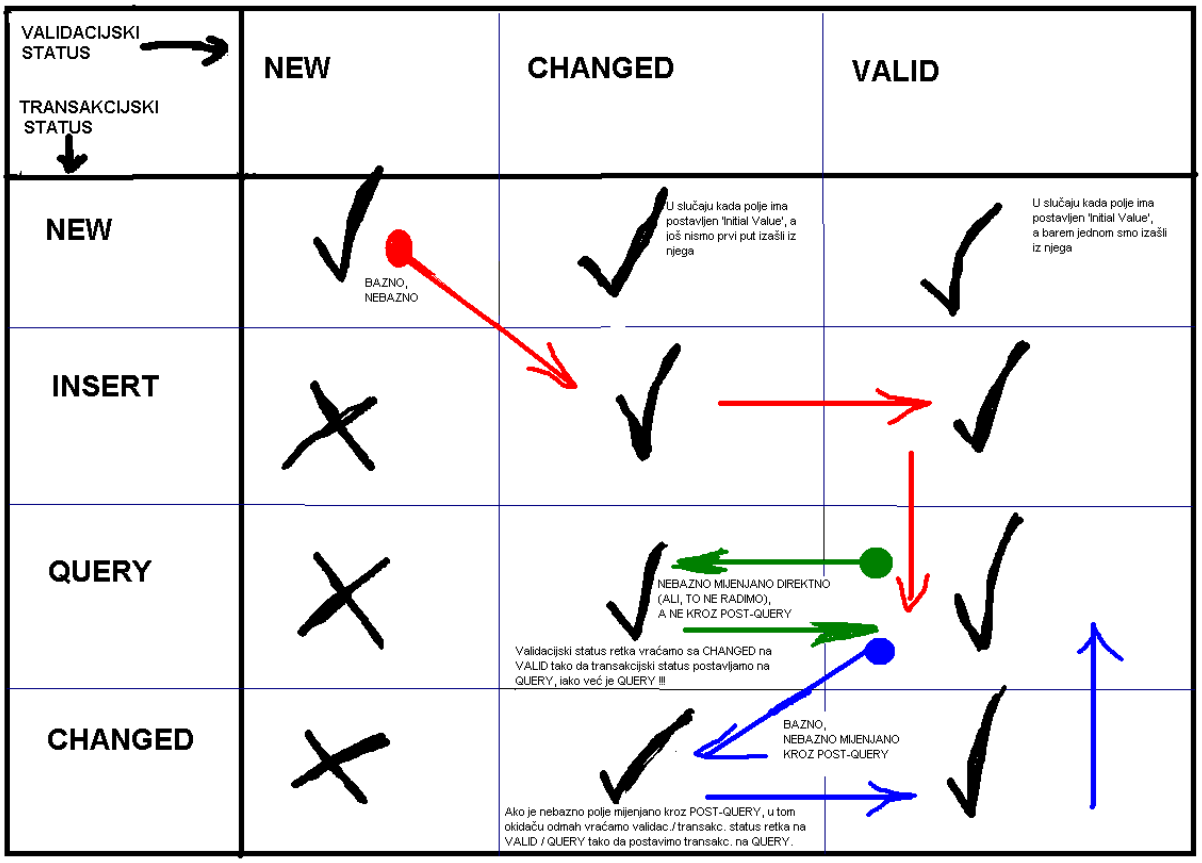
Za razliku od transakcijskog statusa, ne postoje sistemske varijable pomoću kojih se može čitati validacijski status. Međutim, postoje naredbe GET_ITEM_PROPERTY (…ITEM_IS_VALID) / SET_ITEM_PROPERTY (…ITEM_IS_VALID…) pomoću kojih se može čitati / mijenjati validacijski status polja. Status retka može se čitati / mijenjati samo indirektno - tako da se čitaju / mijenjaju statusi polja u retku.

Dakle, (samo) redak ima i transakcijski i validacijski status. Tablica 2..5 prikazuje moguće kombinacije transakcijskog i validacijskog statusa retka.

Validacijski status => Transakcijski status	NEW	CHANGED	VALID
NEW	Prazan redak.	U slučaju kada polje ima postavljen 'Initial Value', a još nismo prvi put izašli iz njega, validacijski status polja je FALSE	U slučaju kada polje ima postavljen 'Initial Value', a izašli smo barem jednom iz njega, validacijski status polja je TRUE
INSERT	NE POSTOJI	Novi redak, još nije provjeren.	Novi redak, provjeren.
QUERY	NE POSTOJI	Ako mijenjamo nebazno polje direktno (ali, to ne radimo)!	Nepromijenjeni postojeći redak.
CHANGED	NE POSTOJI	Promijenjeni postojeći redak, još nije provjeren. Ovo se dešava i ako mijenjamo nebazno polje kroz okidač POST_QUERY! U tom okidaču odmah vraćamo transakcijski status na QUERY, čime se i validacijski status postavlja na VALID.	Promijenjeni postojeći redak, provjeren.

Tablica 2.5. Kombinacije transakcijskog i validacijskog statusa retka

Slika 2.5. prikazuje tranzicije između kombinacija transakcijskog i validacijskog statusa retka.



Slika 2.5. Tranzicije između kombinacija transakcijskog i validacijskog statusa retka

2.6. Master-detalj relacije

Za povezivanje dva bloka, mastera i detalja, koristi se tzv. Forms relacija (relation), koja kao Forms objekt pripada master bloku. Forms relacija ima sljedeća svojstva.

Detail Block
Ime bloka detalja.

Join Condition
Veza između mastera i detalja (npr. stavka.zag_id = zaglavlje.id), koja je zapravo dio WHERE klauzule u SQL naredbi koju Forms automatski generira.

Delete Record Behavior
Specificira da li se može brisati redak master bloka, ako postoje njegovi redovi u bloku detalja. Vrijednosti su Non-Isolated (podrazumijevana vrijednost), Isolated i Cascading.

Non-Isolated ne dozvoljava (na Forms strani) brisanje master retka koji ima retke-detalje.

Isolated dozvoljava (na Forms strani) brisanje master retka, bez brisanja redaka-detalja. No, ako na bazi postoji FK integritetno ograničenje (a uobičajeno postoji), onda će baza spriječiti brisanje retka, osim ako je na bazi deklarirano kaskadno brisanje detalja.

Cascading briše (na Forms strani) master redak i pripadajuće retke-detalje.

Prevent Masterless Operations
Specificira da li korisnik može postaviti upit nad blokom detalja, ili ažurirati blok detalja, ako ne postoji odgovarajući master redak. Kad je postavljeno na Yes (podrazumijevana vrijednost je No), ne može se postaviti upit ili ažurirati detalje ako nema master retka, i javljaju se poruke:

FRM-41105: Cannot create records without a parent record;

FRM-41106: Cannot query records without a parent record.

Coordination = Deferred + Automatic Query
Ova dva svojstva zajedno određuju kako i kada će se izvršiti faza punjenja (population phase) redaka u bloku detalja. Oba svojstva imaju moguće vrijednosti Yes ili No.
Podrazumijevana vrijednost je: Deferred = No + Automatic Query = No.
Teoretski su moguće 4 kombinacije, ali stvarno postoje samo tri:

- Deferred = No (svojstvo Automatic Query tada nije važno): upit nad blokom detalja izvršava se čim odaberemo drugi master redak, bez ulaska u blok detalja;

- Deferred = Yes + Automatic Query = Yes: upit nad blokom detalja izvršava se čim uđemo u blok detalja, bez potrebe za eksplicitnim upitom;

- Deferred = Yes + Automatic Query = No: upit nad blokom detalja izvršava se kad uđemo u blok detalja i eksplicitno damo upit (npr. F8).

Na temelju navedenih svojstava Forms relacije, Forms Builder automatski generira sljedeće objekte u Forms modulu:

- okidač na razini Forms modula ON-CLEAR-DETAILS; kako mu ime kaže, on briše iz bloka detalje prethodnog master retka, prije nego ih napuni detaljima novog master retka;

- okidače na razini (master) bloka ON-POPULATE-DETAILS i ON-CHECK-DELETE-MASTER;

- PL/SQL procedure: Check_Package_Failure, Clear_All_Master_Details, Query_Master_Details.

Kako je već rečeno u potpoglavlju 2.4., važno je postaviti svojstvo polja vanjskog ključa u bloku detalja Copy Value From Item, tako da uzima vrijednost iz odgovarajućeg (PK ili UK) polja master retka.

Oracle Designer omogućava još neke mogućnosti u odnosu na standardno Forms rješenje. Npr. u Designeru se može postaviti da se upit nad blokom detalja izvršava automatski čim mijenjamo master, ali samo ako je taj blok vidljiv, da se ne bi nepotrebno trošilo vrijeme servera i klijenta. Ako blok detalja nije vidljiv, upit se automatski izvršava kad postane vidljiv (nije nužno da uđemo u blok detalja).

Designerovo rješenje temelji se na sljedećim objektima:

- okidači na razini Forms modula ON-CLEAR-DETAILS i ON-POPULATE-DETAILS; kod Designera je ON-POPULATE-DETAILS okidač na razini Forms modula, a ne bloka, a i njegov programski kod je različit u odnosu na Forms varijantu;

- procedure iz Designer PL/SQL librarya.

Ako kroz Forms Builder naknadno mijenjamo formu koja je napravljena kroz Designer, i dodamo novu relaciju, vrlo je preporučljivo da nakon toga rješenje svedemo na Designerovu varijantu, kako ne bismo dobili neočekivano ponašanje Forms modula.

Konkretno, treba napraviti sljedeće:

- brisati okidače na razini (master) bloka ON-POPULATE-DETAILS i ON-CHECK-DELETE-MASTER;

- brisati procedure Check_Package_Failure, Clear_All_Master_Details i Query_Master_Details;

- mijenjati okidač na razini Forms modula ON-CLEAR-DETAILS, tj. u njemu brisati programski kod koji je Forms dodao na kraju sljedećeg programskog koda (dio za brisanje označen je sa italic):

```
/* CGBS$ON_CLEAR_DETAILS */
/* clear all detail blocks for the given master block */
BEGIN
  IF (FORM_FAILURE
      AND :SYSTEM.COORDINATION_OPERATION IN ('MOUSE', 'DUPLICATE_RECORD'))
  THEN
    RAISE FORM_TRIGGER_FAILURE;
  END IF;
  IF (:SYSTEM.MASTER_BLOCK = :SYSTEM.TRIGGER_BLOCK) THEN
    CGBS$.CLEAR_MASTER_DETAIL
      (:SYSTEM.MASTER_BLOCK, :SYSTEM.COORDINATION_OPERATION );
  END IF;
END;
--
-- Begin default relation program section
--
BEGIN
  Clear_All_Master_Details;
END;
--
-- End default relation program section
--
```

Na kraju, treba naglasiti jednu veliku manu Forms relacija.

Nije moguće na standardan način u jednoj transakciji (baze) ostvariti relaciju "master–detalj–detalj od detalja", ili u jednoj transakciji ažurirati više zaglavlja i njihove detalje. U jednoj transakciji (na standardan način) moguće je ažurirati samo jedan master redak i njegove detalje, ili ažurirati više master redaka, bez ažuriranja detalja. Razlog je taj što Forms blok detalja može sadržavati retke-detalje samo jednog master retka. Čim odaberemo drugi master redak, Forms (kako je već rečeno) briše detalje prethodnog master retka (Clear_All_Master_Details) i puni blok detalja redovima drugog master retka (Query_Master_Details).

Za razliku od Formsa, ADF to može, kako će biti prikazano u 3. poglavlju.

No i Forms to može, ali na nestandardan način. Jedan od tih načina (vjerojatno "najčišći") prikazujemo u sljedećem potpoglavlju, a suština mu je u korištenju POST naredbe.

2.7. Template i library za POST-iranje kod relacije master–detalj

Rješavanje prethodno navedenog problema, pomoću POST naredbe, nije jednostavno, ali suština je jednostavna: kod odabira novog master retka, prethodni (mijenjani) master redak i svi njegovi redovi-detalji POST-iraju se na bazu (dakle, izvršavaju se DML naredbe, ali bez COMMIT-a).

Mi smo to rješenje izveli kao Designer template i pripadajući PL/SQL library.

Master redak ima oznaku valjanosti, koja se ažurira na klijentu i ima četiri vrijednosti: * (samo na klijentu, predstavlja NULL), N(ovi), V(aljan), P(romijenjen).

Suština rada template-a je sljedeća:

- oznaka valjanosti (master bloka) postavlja se u formi na default * i ne smije biti server derived;
- u WHEN-NEW-RECORD-INSTANCE provjerava se status forme; ako je mijenjana, radi se POST;
- u WHEN-VALIDATE-RECORD mijenja se oznaka valjanosti na formi (ali samo * u N ili V u P);
- u ON-CLEAR-DETAILS provjerava se oznaka valjanosti (provjera se ne radi ako je riječ o brisanju mastera) i, ako je N(ovi) ili P(romijenjen), ne dozvoljava prijelaz na novi master;
- gumb "Spremi" postavlja oznaku valjanosti na V(aljan) (ako već nije bila) i radi COMMIT_FORM;
- gumb "Poništi" radi CLEAR_FORM.

PL/SQL library sadrži dva paketa, a njihove osnovne procedure / funkcije (za potrebe POST-iranja) su sljedeće:

Paket APL_TRG (pozivi iz okidača)

FUNCTION **fs_key_clrfrm** RETURN BOOLEAN;
Implementira funkcionalnost Poništi.
iii_fnc.ponisti_promjene;
Zaustavlja procesiranje Designer-ovog koda sa RETURN FALSE;

PROCEDURE **fe_key_commit**;
Implementira funkcionalnost Spremi.
iii_fnc.spremi_promjene;

PROCEDURE **fe_key_delrec**;
iii_fnc.komitiraj_brisanje_mastera;

FUNCTION **bs_key_delrec** RETURN BOOLEAN;
iii_fnc.provjeri_dozvoljenost_brisanja;
Traži potvrdu brisanja reda. Dalje, ako tekući blok nije master, radi
iii_fnc.postavi_status_mastera;
Brisanje retka obavit će, nakon potrebnih provjera, Designerov kod.

PROCEDURE **fs_key_entqry**;
Ne može se raditi upit ako nisu spremljene promjene.
iii_fnc.provjeri_status_mastera;

PROCEDURE **fs_key_exeqry**;
Ne može se raditi upit ako nisu spremljene promjene.
iii_fnc.provjeri_status_mastera;

PROCEDURE **fs_key_cquery**;
Ne može se raditi upit ako nisu spremljene promjene.
iii_fnc.provjeri_status_mastera;

PROCEDURE **fs_key_exit**;
Ne može se izaći ako nisu spremljene promjene.
iii_fnc.provjeri_status_mastera;

PROCEDURE **fs_on_clear_details**;
iii_fnc.provjeri_status_mastera;

FUNCTION **fs_when_validate_record** RETURN BOOLEAN;
iii_fnc.postavi_status_mastera;

FUNCTION **fs_when_new_block_instance** RETURN BOOLEAN;
Ako je iii_fnc.zabrani_detalje_bez_zaglavlja = FALSE, zaustavi daljni Designerov kod.
Inače, nastavi.

FUNCTION **fs_when_new_record_instance** RETURN BOOLEAN;
iii_fnc.postiraj;

Paket III_FNC (funkcionalnosti)

FUNCTION **blok_je_prometni** (blok_p IN VARCHAR2) RETURN BOOLEAN;
Pokazuje da li je blok zasnovan nad tablicom čiji naziv počinje slovom 'T', što znači da se radi o transakcijskim podacima. Koristi se kako bi se zabranilo brisanje transakcijskih redaka, koji imaju redne brojeve, i stvaranje "rupa" u rednim brojevima.

FUNCTION **oznaka_valjanosti** RETURN VARCHAR2
Vraća vrijednost polja za oznaku valjanosti.

PROCEDURE **spremi_promjene**;
Ako se dalo 'Spremi' dok je kurzor bio na novom/mijenjanom retku,
potreban je POST da se izvrši INSERT / UPDATE retka.
POST / COMMIT_FORM je uspio ako je nakon njega status forme QUERY.
Pamti se (u lokalnu varijablu) stara vrijednost oznake valjanosti (da bi se vratila nakon eventualnog neuspjeha), pa se stavlja oznaka da je master valjan, za provjeru COMMIT pravila. Ako nije uspjela provjera COMMIT pravila, vraća se stara oznaka.

PROCEDURE **ponisti_promjene**;
Nakon provjere, promjene se poništavaju.

PROCEDURE **postiraj**;
Vraća varijablu ozn_brisan_master_m na 'N' onda kad se NIJE izvršila procedura postavi_status_mastera nakon brisanja master retka.
Radi se samo ako je bilo promjena na formi.

PROCEDURE **postavi_status_mastera**;
Ne radi ako se pokreće iz spremi_promjene sa oznakom valjanosti 'V'.
Ne radi ako se pokreće iza provjeri_status_master nakon brisanja mastera.

PROCEDURE **provjeri_status_mastera**;
Ova validacija potrebna je ako se mijenja samo master.
Ako je pokrenuto brisanje mastera, ne provjerava se oznaka valjanosti.

PROCEDURE **komitiraj_brisanje_mastera**;
Postira promjene mastera.
Ako nakon toga status forme nije QUERY, znači da postiranje nije uspjelo, javlja grešku i poništava promjene.
Inače, radi COMMIT_FORM.

PROCEDURE **provjeri_dozvoljenost_brisanja**;
Redak se ne može obrisati ako je tekuća pozicija na zaglavlju i njegova bazna tablica je prometna i memorijska oznaka statusa je V(aljan) ili P(romijenjen).

FUNCTION **zabrani_detalje_bez_zaglavlja** RETURN BOOLEAN;
Osigurava zabranu prelaska na novi blok, ukoliko u bloku zaglavlja nije dovršen unos (klijent nije generirao ID na PRE-INSERT)
Ako je ostao na master bloku, onda ne treba provjera.

2.8. Različiti načini poziva Forms modula

Osim vrlo jednostavnih (ili vrlo specijalnih) aplikacija, većina aplikacija sastoji se od više Forms modula. Naravno, postavlja se pitanje na koji način povezati te module, npr. kako pozivati jedan modul iz drugog.

Najčešće realiziramo poziv drugog modula iz prvog modula tako da korisnik odabere modul iz menija koji je pridružen svakom modulu. Pritom se modul može pozvati na tri načina, tj. pomoću tri Forms naredbe:

- OPEN_FORM;
- CALL_FORM;
- NEW_FORM.

NEW_FORM nismo nikada koristili, jer (najčešće) nije primjeren za Windows način rada, zato što zatvara prvi modul kod pokretanja drugog modula. Prednost korištenja NEW_FORM načina rada je u tome da se smanjuju memorijski zahtjevi na strani Forms klijenta (u klijent-server Forms varijanti) ili Forms aplikacijskog servera. Kod poziva novog modula sa NEW-FORM (a isto vrijedi i za CALL_FORM) mogu se odabrati tri vrijednosti ROLLBACK_MODE parametra, kojim se određuje kako će Forms runtime postupiti u slučaju ROLLBACK-a koji se desio u pozvanoj formi:

- TO_SAVEPOINT (default): izvršit će se ROLLBACK TO SAVEPOINT svih ne-COMMIT-iranih promjena (uključujući i POST-irane promjene) do SAVEPOINT točke, koja se inače automatski postavlja kod ulaska u pozvani modul; to znači da će se poništiti sve promjene koje je napravio pozvani modul, ali se neće poništiti promjene koje je napravio pozivajući modul (i oni prije njega, ako ih ima);

- NO_ROLLBACK: neće se izvršiti ROLLBACK, što znači da će ostati sačuvane sve promjene (a ostat će i zaključani redovi), pa i one koje je napravio pozvani modul;

- FULL_ROLLBACK: poništit će se sve promjene; ne može se navesti FULL_ROLLBACK kod poziva iz modula koja radi u tzv. post-only modu, koji je opisan dolje, kod CALL_FORM naredbe.

OPEN_FORM i CALL_FORM vidljivo se razlikuju po tome što CALL_FORM ne dozvoljava "šetanje" između pozivajućeg i pozvanog modula. Možemo raditi samo u pozvanom modulu, jer je pozvani modul modalan – moramo izaći iz njega da bismo se mogli vratiti u pozivajući modul. Ako jedna forma pozove drugu sa OPEN_FORM, slobodno se možemo "šetati" između te dvije forme (osim ako smo napravili modalan prozor, ali to obično ne radimo).

No, važnija razlika je u tome što OPEN_FORM dozvoljava da se pozvani modul pokrene u posebnoj sesiji baze (u odnosu na pozivajući modul), dok kod CALL_FORM naredbe (kao i NEW_FORM) pozvani i pozivajući modul rade uvijek u istoj sesiji baze.

Kada pozivamo module iz menija, uvijek koristimo OPEN_FORM u novoj sesiji. Istina, to povećava broj sesija na bazi, ali to nam se čini sasvim normalnim. Ako npr. korisnik otvori dva Forms modula, od kojih jedan služi za ažuriranje npr. dobavljača, a drugi služi za ažuriranje npr. artikala, izgleda nam logično da su to dvije nezavisne sesije (kao da dva korisnika rade na dva računala). Treba naglasiti da, iako se za svaki modul otvara nova sesija, sve te sesije (kao i sve sesije Forms procesa) koriste istu konekciju na bazu, i (što je možd ai važnije) sve te sesije na bazi poslužuje isti background proces baze. Znači, osim što se povećava broj sesija na bazi, takav način rada ne opterećuje bazu značajnije.

CALL_FORM koristimo uvijek za poziv LOV Forms modula, a i u nekim drugim specijalnim slučajevima kada je proces jako kompliciran, pa je zato riješen kroz više Forms modula.

Važno je napomenuti da kod CALL_FORM možemo imati samo jedan tzv. "Call form stack", tj. ne možemo imati dva (ili više) CALL_FORM "lanaca" (vidjeti Forms help).

Također, kod CALL_FORM (i NEW _FORM) postoji dodatno ograničenje u radu s transakcijom. Ako pozivajući modul nije napravio POST (ili COMMIT_FORM, ali ovo drugo nema puno ako su oba modula dio jedne transakcije), pozvani modul je u tzv. post-only modu, tj. pozvani modul ne može napraviti COMMIT ili (kompletni) ROLLBACK. Pozvana forma može napraviti samo ROLLBACK TO SAVEPOINT (SAVEPOINT se automatski radi kod poziva), tj poništiti samo ono što je on napravio. Pritom se, zbog ROLLBACK TO SAVEPOINT, može desiti sljedeći problem (koji je vezan za uobičajeno ponašanje baze, vidjeti početak potpoglavlja 1.3.): ako je pozvana forma zaključala neke retke, a onda napravila ROLLBACK TO SAVEPOINT, oni su i dalje ostali zaključani za neku treću formu koja radi u posebnoj sesiji baze i koja je te retke već pokušavala zaključati.

2.9. Forms i odgođena deklarativna integritetna ograničenja na bazi

Kao što je prikazano u potpoglavlju 1.6., Oracle baza ima još od verzije 8.0 mogućnost odgađanja provjere deklarativnih integritetnih ograničenja (najkasnije do COMMIT-a).

Nažalost, ako koristimo Oracle Forms, greške vezane za odgođena deklarativna ograničenja ne obrađuju se dobro u Forms runtimeu. Još jedan problem (puno veći) javlja se ako u Formsima koristimo POST built-in.

Npr. dodajmo DEPT tablici odgođeni primarni ključ:

```
ALTER TABLE dept
   ADD CONSTRAINT dept_pk PRIMARY KEY (deptno) INITIALLY DEFERRED
   USING INDEX
/
```

Ako unesemo npr. dva retka sa istom vrijednošću u stupcu DEPTNO, te damo COMMIT naredbu kroz SQL+, dobijemo očekivane poruke o grešci:

ORA-02091: transaction rolled back

ORA-00001: unique constraint (SCOTT.DEPT_PK) violated

Sada kreirajmo najjednostavniji Forms modul koristeći Forms Data Block Wizard i unesimo iste podatke kroz Forms. Nakon Save (tj. COMMIT_FORM) ne dobijemo odgovarajuću poruku o grešci, čak niti ako koristimo ON-ERROR Forms okidač ili SQLERRM. Naprotiv, Forms prvo javlja poruku kao da je sve u redu:

FRM-40400: transaction finished, 2 Record saved

a nakon toga javlja poruku:

FRM-99999: Error 408 occurred.

Naravno, podaci ipak nisu mogli biti uneseni u bazu, jer deklarativno ograničenje na bazi radi, ali Forms očito javlja pogrešnu (prvu) poruku. Po svemu se čini da je to Forms bug, koji možda nikada neće biti riješen (postoji od Forms 6, pa do danas).

No čak i da ova pogrešna poruka bude riješena u nekoj budućoj Forms verziji (npr. Forms 12c), postoji i veći problem, ako koristimo POST built-in. Kako je navedeno u potpoglavlju 2.4., POST built-in upisuje podatke iz Forms modula u bazu, ali ne izvršava COMMIT (naredbu baze). Nakon POST-iranja redaka, Forms "zaboravlja" koji su redovi uneseni / mijenjani / brisani. Problem je sa odgođenim deklarativnim ograničenjima u tome da nakon COMMIT naredbe, ako ograničenja nisu zadovoljena, baza automatski radi ROLLBACK cijele transakcije.

Rješenje za oba problema je relativno jedostavno – trebamo sami provjeriti da li je odgođeno deklarativno ograničenje zadovoljeno prije nego što se desi COMMIT naredba na bazi, koristeći naredbu:

```
SET CONSTRAINT dept_pk IMMEDIATE
/
```

Navedenu naredbu možemo staviti u jedan “generički” paket, koji koristi dinamički SQL:

```
CREATE OR REPLACE PACKAGE deferred_constraints AS
   PROCEDURE set_immediate (p_constraint VARCHAR2);
   PROCEDURE set_deferred  (p_constraint VARCHAR2);
END deferred_constraints;
/
CREATE OR REPLACE PACKAGE BODY deferred_constraints AS
   PROCEDURE set_immediate (p_constraint VARCHAR2) IS
   BEGIN
      EXECUTE IMMEDIATE 'SET CONSTRAINT ' || p_constraint || ' IMMEDIATE';
   END;

   PROCEDURE set_deferred (p_constraint VARCHAR2) IS
   BEGIN
      EXECUTE IMMEDIATE 'SET CONSTRAINT ' || p_constraint || ' DEFERRED';
   END;
END deferred_constraints;
/
```

Naravno, da bi drugi korisnici (a ne samo vlasnik sheme SCOTT) mogli koristiti paket, treba (npr. kao korisnik SYSTEM) napraviti globalni sinonim:

```
CREATE PUBLIC SYNONYM deferred_constraints FOR scott.deferred_constraints
/
```

Zatim korisnik SCOTT daje pravo određenom korisniku da koristi taj paket:

```
GRANT EXECUTE ON deferred_constraints TO your_user
/
```

Nakon toga dodamo u Forms modul ON-COMMIT okidač (koji se okida umjesto standardnog COMMIT_FORM built-in-a), sa sljedećim programskim kodom:

```
BEGIN
   deferred_constraints.set_immediate ('dept_pk');
   deferred_constraints.set_deferred  ('dept_pk');
   COMMIT_FORM;
EXCEPTION
   WHEN OTHERS THEN
      MESSAGE
         ('COMMIT ERROR: ' || DBMS_ERROR_CODE || ' ' || DBMS_ERROR_TEXT);
      PAUSE;
      deferred_constraints.set_deferred ('dept_pk');
      RAISE FORM_TRIGGER_FAILURE;
END;
```

Naravno, umjesto naredbi MESSAGE i PAUSE u praksi ćemo koristiti neki drugi način prikaza poruke o grešci (najvjerojatnije će to biti ALERT), koji je više "user friendly".

Primijetimo da smo naizgled mogli postići isti rezultat i bez paketa na bazi, pišući sav programski kod u ON-COMMIT Forms okidaču, npr:

```
FORMS_DDL ('SET CONSTRAINT dept_pk IMMEDIATE');

IF NOT FORM_SUCCESS THEN
   MESSAGE
      ('COMMIT ERROR: ' || DBMS_ERROR_CODE || ' ' || DBMS_ERROR_TEXT);
   PAUSE;
   FORMS_DDL ('SET CONSTRAINT dept_pk DEFERRED');
   RAISE FORM_TRIGGER_FAILURE;
ELSE
   FORMS_DDL ('SET CONSTRAINT dept_pk DEFERRED');
   COMMIT_FORM;
END IF;
```

Međutim, taj način radi dobro samo ako Forms modul pokrenemo kao korisnik SCOTT, tj. kao korisnik koji je vlasnik tablice DEPT. Naime, ako ne-vlasnik tablice pokušava izvršiti SET naredbu na odgođenom deklarativnom ograničenju, dobije poruku:

ORA-02448 Constraint does not exist

Ovo je programsko rješenje bilo prikazano na web stranicama firme Quest u 4.mjesecu 2003. (stranica više nije dostupna), te na HROUG-u 2002.

3. TRANSAKCIJE I ORACLE ADF

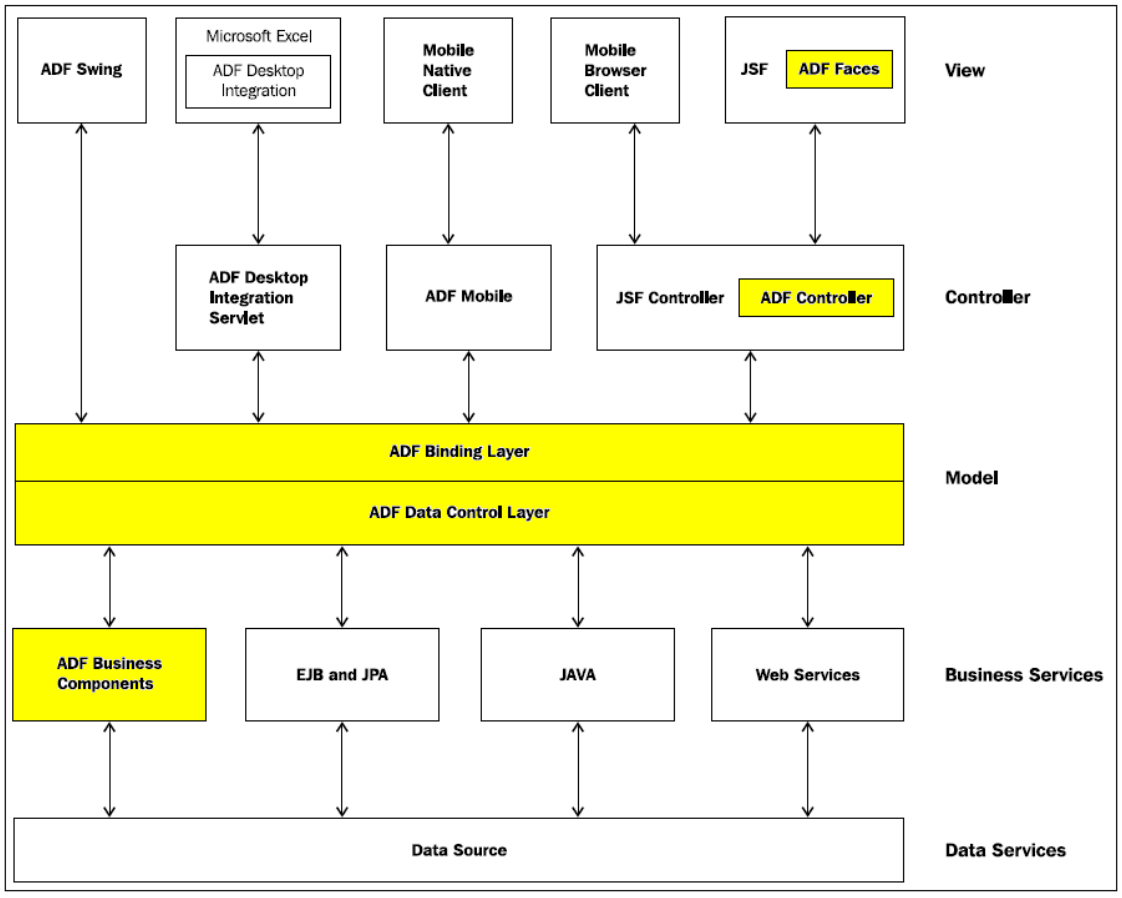
3.1. ADF - razvoj i arhitektura

Kako kaže Steve Munch, jedan od kreatora Oracle ADF-a (Application Development Framework), u predgovoru knjige [10] svojih kolega iz Oraclea, već u ljeto 1996. godine, 6 mjeseci nakon što je Sun izdao verziju Jave 1.0, u Oracleu su odlučili raditi deklarativni, RAD (Rapid Application Development) alat temeljen na jeziku Java. Prvo izdanje frameworka, koji se tada nije zvao ADF već JBO (Java Business Objects), uslijedilo je 1999. godine. Ubrzo mu je ime promijenjeno u BC4J (Business Components for Java). BC4J je pokrivao onaj dio koji danas pokriva ADF BC (Business Components), evolutivni nasljednik BC4J-a.

Uz BC4J, Oracle je počeo razvijati i odgovarajući IDE (Interactive Development Environment) imena JDeveloper, licencirajući 1998. tadašnji Borlandov alat JBuilder. Vrlo brzo, 2001., Oracle je temeljito preradio JDeveloper, pri čemu ga je u potpunosti "prepisao" u Java kod (to je bila verzija 9i; verzije od 4 do 8 su preskočene, kako bi alat pratio verziju tadašnje baze 9i). Ubrzo se, u verziji 10g (koja je zapravo bila 9.0.5), prvi put pojavio termin ADF. 2005. godine, u novoj verziji 10g (10.1.3), Oracle JDeveloper je postao besplatan softver (free software).

Za razliku od "nižeg" dijela, tj. ADF BC-a (bivšeg BC4J-a), "viši" dio, koji se odnosi na Controller i View dio MVC (Model View Controller) arhitekture, razvijao se manje evolutivno, sa velikim skokovima, što je pratilo uobičajena zbivanja u cijeloj softverskoj industriji vezanoj za Java web aplikacije. Poznato je da su se dinamičke stranice u JEE arhitekturi (Java Enterprise Edition; prije se označavalo kao J2EE) prvo radile u servlet tehnologiji. Na temelju nje nastala je JSP (Java Server Pages) tehnologija. Oracle je ubrzo uvidio da te tehnologije nisu dovoljno produktivne, jer ne omogućavaju odgovarajuće module ili komponente, pa je razvio svoju specijalnu tehnologiju UIX (User Interface XML), temeljenu na komponentama. UIX Components su služile za definiranje i za renderiranje (rendering, prikazivanje) stranica, a UIX Controller za upravljanje događajima i interakciju između stranica u aplikaciji. Na neki način, UIX su bili preteča standardne JSF (Java Server Faces) tehnologije, čija je (usavršena) varijanta i ADF JSF.

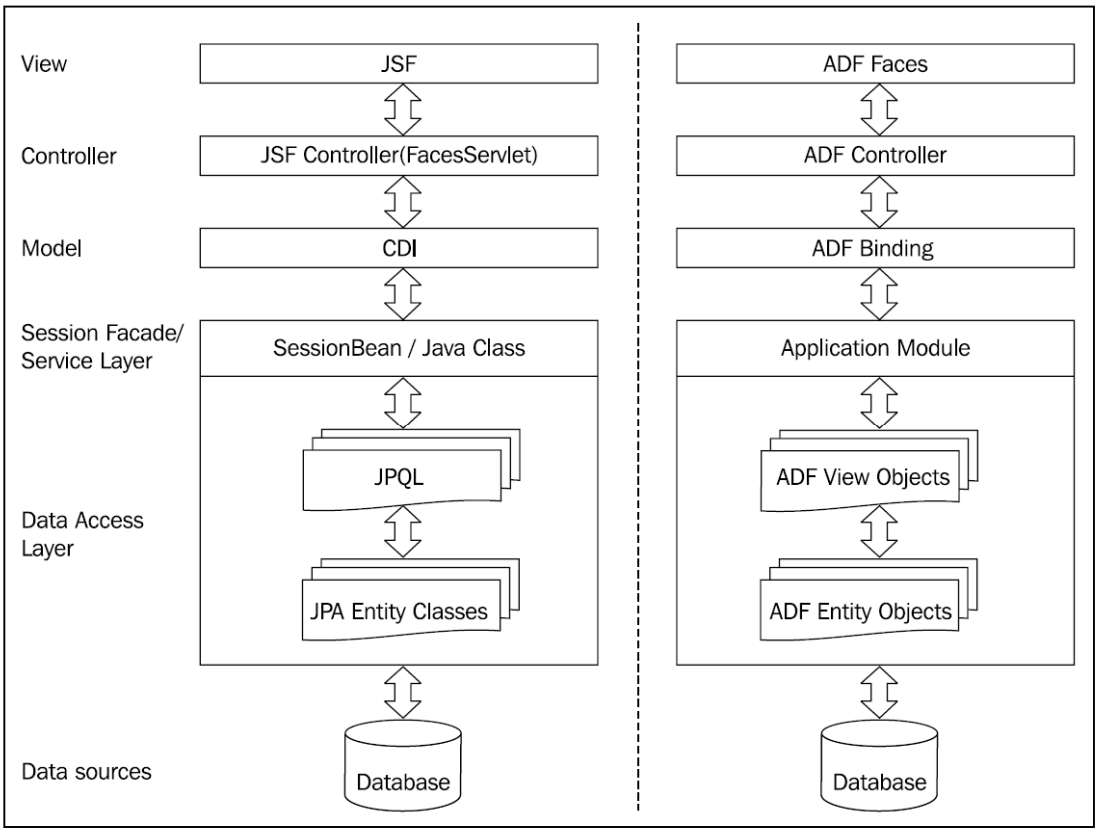
Slika 3.1. prikazuje različite sastavne dijelove ADF frameworka, podijeljene u četiri sloja (sloj izvora podataka, npr. baze podataka, ne spada u ADF).



Slika 3.1. Sastavni dijelovi ADF frameworka; Izvor [11]

Kako se vidi na slici 3.1., uz tri MVC sloja postoji i Business Services sloj, čiji je jedan "predstavnik" ADF BC (zapravo se i Model sloj na slici dijeli na podslojeve ADF Data Control i ADF Binding). Iako se ADF framework može slagati od različitih "kockica" na pojedinom sloju, uobičajeni "izbor" (onaj koji je zapravo i Oracleov izbor za izradu Oracle Fusion aplikacija) označen je žutom bojom, a čine ga (od dolje prema gore) ADF BC, ADF Model, ADF Controller, ADF JSF.

Koje su sličnosti između standardne JEE arhitekture i Oracle ADF arhitekture (koja se u suštini temelji na JEE arhitekturi, ali ju nadograđuje, i nije standardna arhitektura) prikazano je na slici 3.2.



Slika 3.2. Usporedba JEE i ADF arhitekture; Izvor [11]

ADF Business Service sloj čine tri vrste ADF komponenti (ili objekata; no pojam "objekt" je tako preopterećen da je vrlo nezgodno što ga Oracle koristi i za te komponente): Entity Object (ADF BC EO, ili samo EO), View Object (ADF BC VO, ili VO), Application Module (ADF BC AM, ili AM). EO su slični JEE JPA (Java Persistence API) Entity klasama, ali EO imaju neke prednosti, npr. podršku za keširanje podataka (na aplikacijskom serveru), upravljanje transakcijama, deklarativnu validaciju i dr. VO su slični JEE JPQL-u (Java Persistence Query Language), ali bolje podržavaju vizualno i deklarativno programiranje, deklarativno upravljanje stanjima (declarative state management) i dr. AM je transakcijska komponenta koja je konceptualno slična Session Facade sloju izgrađenom na temelju Session Bean-ova iz EJB (Enterprise JavaBeans) arhitekture (ili specifikacije), ali razlike su među njima fundamentalne.

Data Binding sloj u ADF-u, poznat i kao ADF Model, je jedinstven i nema odgovarajuću kopiju u JEE svijetu. ADF Model odvaja UI od implementacije poslovnih servisa i pruža generička rješenja za kolekcije podataka koje vraćaju poslovni servisi. Konceptualno slične mogućnosti ima Context and Dependency Injection (CDI) sloj iz Java EE, ali na drugačiji način.

ADF Controller je donekle sličan standardnom JEE JSF Controlleru (čiji su glavni zadaci obrada zahtjeva prema stranicama, i navigacija među stranicama), ali omogućava modularniji razvoj aplikacija, jer razbija monolitnu web aplikaciju u višestruko iskoristive dijelove, tzv. ADF Task Flows. Svaki Task Flow može imati svoje vlastite transakcijske atribute, upravljanje resursima, definicije za managed beanse, rješenja za navigaciju.

ADF Faces su vrlo slični standardnim JSF, i izgrađeni su povrh njih, ali imaju brojne dodatne mogućnosti, kao što su grafovi i dijagrami, komponente za dijaloge, deklarativne komponente, data streaming, mogućnost ugradnje Task Flows, AJAX-enabled UI komponente i dr.

3.2. Entity Object

Kako je rečeno u prethodnom potpoglavlju, ADF BC Entity Object (ADF BC EO, ili samo EO) su jedna od tri najvažnije komponente ADF Business Service sloja. Termin Entity Object nije baš najsretniji, npr. kada se koristi unutar šireg termina Entity Object Instance (a znamo da su u objektnoj paradigmi pojmovi objekt i instanca (klase) sinonimi). EO zapravo predstavlja na ADF strani (određene) objekte iz baze. Jedan EO predstavlja jednu tablicu, jedan pogled (view) baze (i termin view je jako preopterećen, kao i termin object), jedan sinonim ili jedan materijalizirani pogled (materialized view). U nastavku ćemo (zbog kraćeg izražavanja) uvijek govoriti kao da EO predstavlja (samo) tablicu.

EO se zapravo sastoji od četiri potkomponente, a to su:

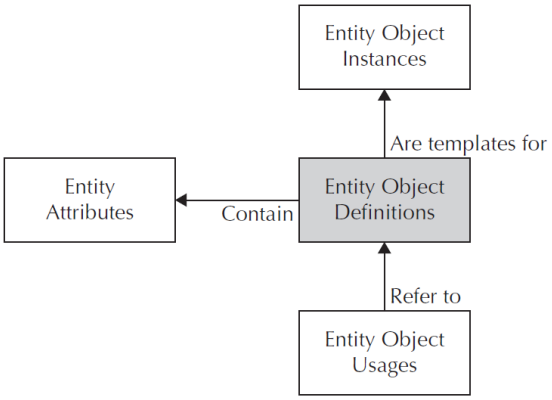
- EO XML definicija (EO definition XML metadata file): XML datoteka koja sadrži metapodatke za EO;

- EO klasa za definiciju (EO definition): to je klasa koja predstavlja definiciju za EO tokom izvođenja; podrazumijevana (default) klasa je oracle.jbo.server.EntityDefImpl, a može se napraviti podklasa te klase, što se rjeđe radi;

- EO klasa: ta klasa predstavlja instance danog entiteta, pa se često naziva i EO Instance (više termina istog značenja, te ponekad loše odabrani termini, sigurno ne olakšavaju učenje ADF-a); kako literatura često navodi: "pojednostavljeno rečeno, instanca te klase predstavlja redak u tablici baze" (no to nije baš najpreciznije, jer se redovi iz baze na kraju drže u entity cacheu); podrazumijevana (default) klasa je oracle.jbo.server.EntityImpl, a može se napraviti podklasa te klase, što se obično i radi;

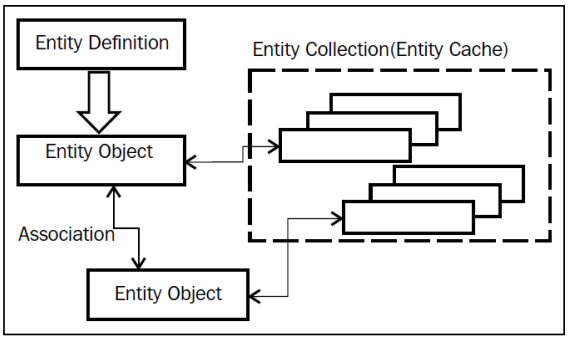
- Entity collection (ili Entity cache) klasa: ta klasa predstavlja cache za instance (objekte) danog entiteta; podrazumijevana (default) klasa je oracle.jbo.server.EntityCache, a može se napraviti podklasa te klase, što se vrlo rijetko radi.

EO definicija (pa onda i EO instanca) sadrži atribute, koji odgovaraju atributima tablice na bazi. Korištenje EO (u drugim komponentama, u VO) naziva se EO Usage. Slika 3.3. prikazuje odnos između EO definicije, EO instance, EO atributa i EO Usagea.



Slika 3.3. EO definicija je predložak za EO instancu; Izvor [8]

Dvije EO definicije (koje predstavljaju npr. dvije tablice na bazi) mogu biti povezane asocijacijom, koja je najčešće nastala iz FK veze među tablicama na bazi (ali to ne mora biti). Budući da EO nastaju na temelju EO definicija, tada su i odgovarajući EO međusobno povezani. Slika 3.4. prikazuje asocijaciju između dva EO, te entity cache.



Slika 3.4. Veza EO definicije i EO, asocijacija između dva EO, te entity cache; Izvor [11]

Kada definiramo EO atribute, za rukovanje transakcijama važna su nam sljedeća svojstva atributa.

Persistent / Transient

Označava da li je atribut perzistentan, tj. izvor / odredište mu je baza podataka (napomena: ADF BC može raditi i s drugim izvorima podataka, ali u tekstu se ograničavamo na bazu podataka) ili tranzijentan, tj. ne sprema se u bazu podataka. Vrijednosti tranzijentnih atributa derivirane su na neki način (preko Groovy izraza ili Java koda) iz drugih atributa. Tranzijentni atributi ponekad mogu poslužiti kao privremena spremišta za podatke.

Refresh on Insert, Refresh on Update

Može se odabrati da se vrijednost podataka osvježi s baze, nakon što se redak pošalje na bazu, tj. nakon što se na bazi izvrši INSERT ili UPDATE. Ovo svojstvo je slično svojstvu Forms bloka DML Returning Value (vidjeti potpoglavlje 2.4.), tj. koristi se to što (još od baze 8) DML naredbe sadržavaju tzv. DML Returning clause, kojom baza može vratiti vrijednosti (nekih) polja iz tekućeg retka, a uglavnom je riječ o onim poljima čije je vrijednosti baza sama mijenjala, najčešće pomoću okidača baze.

Change Indicator

Za razliku od Formsa, kod kojih je podrazumijevano (default) pesimistično zaključavanje redaka (redak se zaključava čim ga korisnik počne mijenjati, dakle prije nego što se radi UPDATE), ADF podrazumijevano zaključavanje je optimističko (redak se zaključava tek neposredno prije UPDATE). Kod optimističkog zaključavanja, ADF standardno treba usporediti sve atribute, da bi vidio da li je netko drugi u međuvremenu mijenjao redak. Kako bi se ubrzala usporedba, može se označiti da su samo neki atributi indikatori promjene (npr. audit polje koje sadrži datum i vrijeme izmjene).

Type: DBSequence

Kao tip atributa može se odabrati DBSequence, čime se automatski postavi i Refresh on Insert. Tako se postavlja kada baza generira vrijednost atributa za PK / UK (npr. ID) pomoću okidača baze. ADF privremeno puni PK / UK atributu jedinstvene negativne brojeve, koji se nakon INSERT-a na bazu zamjenjuju vrijednostima koje je generirao okidač baze. Kod master-detalj veze jako je važan redoslijed slanja DML-a na bazu. Trebao bi se prvo slati roditelj, a onda djeca. Taj redoslijed osigurava, bez dodatnog programiranja, kompozitna asocijacija (opisana u nastavku ovog potpoglavlja).

Kod svojstva Change Indicator spomenuta su dva načina zaključavanja u ADF-u, pesimističko i optimističko (kakve ima i Forms). Međutim, ADF ima četiri načina provjere da li netko drugi mijenja redak, od kojih je jedno zapravo takvo da se nikakva provjera ne radi. Ostaju tri korisna načina, tj. uz pesimističko i optimističko zaključavanje, postoji i provjera koja ne traži zaključavanje. Izbor se radi kroz jbo.locking.mode konfiguracijsko svojstvo.

To svojstvo može se postaviti na dva načina:

- za određeni aplikacijski modul: kroz Edit, u Properties tab, mijenja se jbo.locking.mode;
- za cijelu aplikaciju: u adf-config.xml postavlja se odgovarajući Locking Mode.

Moguće vrijednosti za jbo.locking.mode su sljedeće:

- None: ništa se ne radi;

- Pessimistic: pesimističko zaključavanje, kako je prije objašnjeno; ne preporuča se za web aplikacije, jer čim netko pokuša promijeniti bilo koji podatak retka, redak ostaje zaključan do kraja transakcije;

- Optimistic: optimističko zaključavanje (default), kako je prije objašnjeno;

- Optupdate: slično kao optimističko zaključavanje, ali bez zaključavanja (samo provjerava da li su stare vrijednosti polja iz retka jednake onima koji su sada na bazi), pa ne pruža istu sigurnost kao optimističko zaključavanje; i kod ovog načina, kao i kod optimističkog, usporedba se može ubrzati pomoću svojstva Change Indicator.

Asocijacije između entiteta mogu imati različite kardinalnosti, između ostalog i 1 : N. No, postoji posebna vrsta 1 : N asocijacija, tzv. kompozitna asocijacija ili kompozicija (nekad se koristio i naziv jaka agregacija, jer se do (uključujući) UML-a 2 smatralo da postoji i slaba agregacija; slaba agregacija se u UML-u 2 prikazivala pomoću neispunjenog dijamanta; kompozicija se prikazuje pomoću ispunjenog (crnog) dijamanta). Kod kompozitne asocijacije, instanca jedne klase može postojati samo kao dio instance druge klase (npr. stavka ne može postojati bez dokumenta).

Kada se u ADF-u označi da je asocijacija između dva EO kompozitna, time se rješava i pravilan redoslijed slanja redaka na bazu (prvo roditelj, pa djeca), što je važno (i) kada se za PK / UK atribut roditelja (npr. ID) postavi tip DBSequence. Kompozitnoj asocijaciji se mogu postaviti sljedeća svojstva:

- Use Database Key Constraints: nema utjecaja na ADF, omogućava da se na bazi generira FK na bazi ADF definicije;

- Implement Cascade Delete: brisanjem roditeljskog entiteta, automatski se brišu (na ADF strani) entiteti-djeca; ako nije označeno, ADF ne dozvoljava brisanje roditelja koji ima djecu;

- Optimize for Database Cascade Delete: ako je selektirano, kod brisanja roditelja, ADF neće slati na bazu DELETE naredbe za brisanje djece, jer se tada pretpostavlja da na bazi postoji ON DELETE CASCADE FK;

- Cascade Update Key Attributes: automatski ažurira FK polja djece kod promjene PK / UK roditelja; međutim, nije preporučljivo da aplikacija mijenja PK / UK;

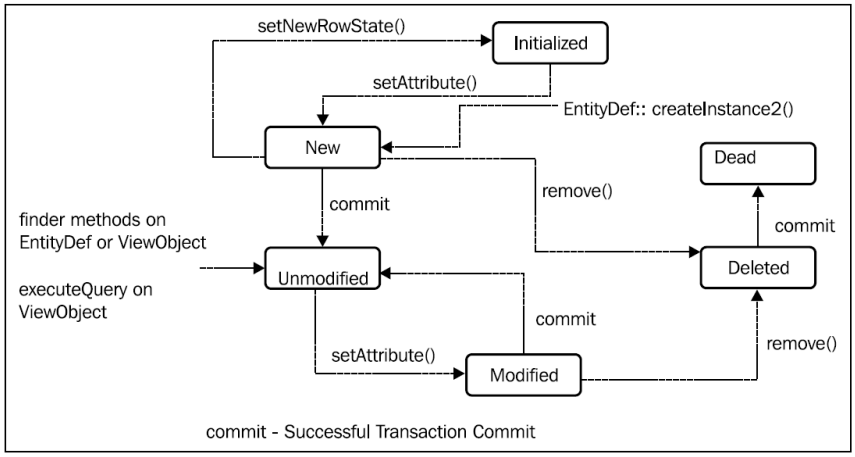
- Update Top-level History Columns: kada se unosi / mijenja / briše dijete, automatski se ažuriraju audit polja roditelja;

- Lock Level: određuje se da li će se zaključati redak roditelja, ako se zaključa bilo koje njegovo dijete; moguće varijante su:
  - None: roditelj se neće zaključati;
  - Lock Container: zaključat će se prvi (neposredni) roditelj;
  - Lock Top-level Container: zaključat će se vršni roditelj, ili onaj u hijerarhiji koji ima isključeno Lock Top-level Container svojstvo.

Kod asocijacije, važno je spomenuti tzv. association accessor. To je operacija (ponekad implementirana kao Java metoda), kojom EO s jedne strane veze, preko tzv. accessor atributa, može pristupati redovima EO koji se nalazi na drugoj strani veze. JDeveloper standardno generira association accessor za svaku stranu veze. Postoji svojstvo kojim se može optimizirati ponašanje za association accessor. U EO roditelja, unutar General grupe svojstava i unutar Tuning sekcije, može se uključiti svojstvo Retain association accessor rowset. Tada će ADF izvršiti upit za dohvat rowseta djece samo prvi put, tj. samo kod prvog pristupanja accessor atributu, a kasnije će biti korišten isti rowset. No u tom slučaju klijent mora pozvati reset() kod svakog korištenja, kako bi resetirao stanje iteratora, kako prikazuje sljedeći kod iz [11]:

```
RowIterator rowIter= DeptEOImpl.getEmpEO();
rowIter.reset();
while(rowIter.hasNext()){
   Row row=rowIter.next();
   //Row represent Emp entity instance
}
```

Slično kao što postoje Forms validacijski i transakcijski atributi (vidjeti potpoglavlje 2.6.), postoje slični atributi i za ADF EO retke, što prikazuje slika 3.5.



Slika 3.5. Različiti statusi EO retka kod transakcije; Izvor [11]

3.3. View Object

ADF BC View Object (ADF BC VO, ili samo VO) ima tri potkomponente koje su slične odgovarajućim potkomponentama za EO (prve tri navedene), te četvrtu koja je specifična za VO:

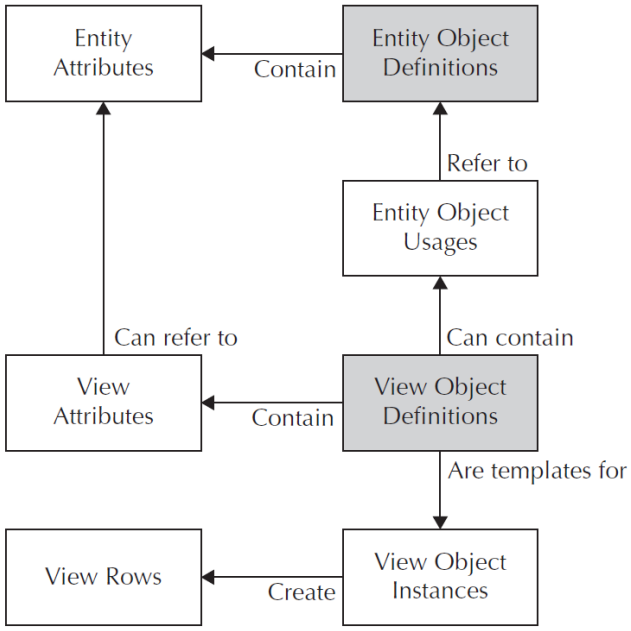
- VO XML definicija (VO definition XML metadata file): XML datoteka koja sadrži metapodatke za VO;

- VO klasa za definiciju (VO definition): to je klasa koja predstavlja definiciju za VO tokom izvođenja; podrazumijevana (default) klasa je oracle.jbo.server.ViewDefImpl, a može se napraviti podklasa te klase, što se rjeđe radi;

- VO klasa: ta klasa predstavlja instance danog viewa, pa se često naziva i VO Instance; podrazumijevana (default) klasa je oracle.jbo.server.ViewImpl; može se napraviti podklasa te klase, što se obično i radi, a tada se može označiti pomoću Include bind variable accesors da se generiraju odgovarajuće seter / geter metode, pomoću kojih se može pisati programski kod u kojem se pogrešna imena atributa otkrivaju kod kompajliranja (u suprotnom se imena atributa pišu u navodnicima, pa se greške otkrivaju tek kod izvođenja);

- VO Row klasa: ta klasa predstavlja retke dobivene na temelju rezultata upita; može se napraviti podklasa te klase, što se obično i radi, a tada se može označiti pomoću Include accesors da se generiraju odgovarajuće seter / geter metode za VO Row atribute.

VO definicija (pa onda i VO instanca) sadrži atribute. Budući da VO može biti temeljen na EO (jednom ili više), ali može biti temeljen i na SQL upitu, VO atributi mogu odgovarati atributima EO, ili stupcima iz SQL upita. Slika 3.6. prikazuje odnos između VO definicije, VO instance, VO atributa i već prije prikazanih EO komponenti.

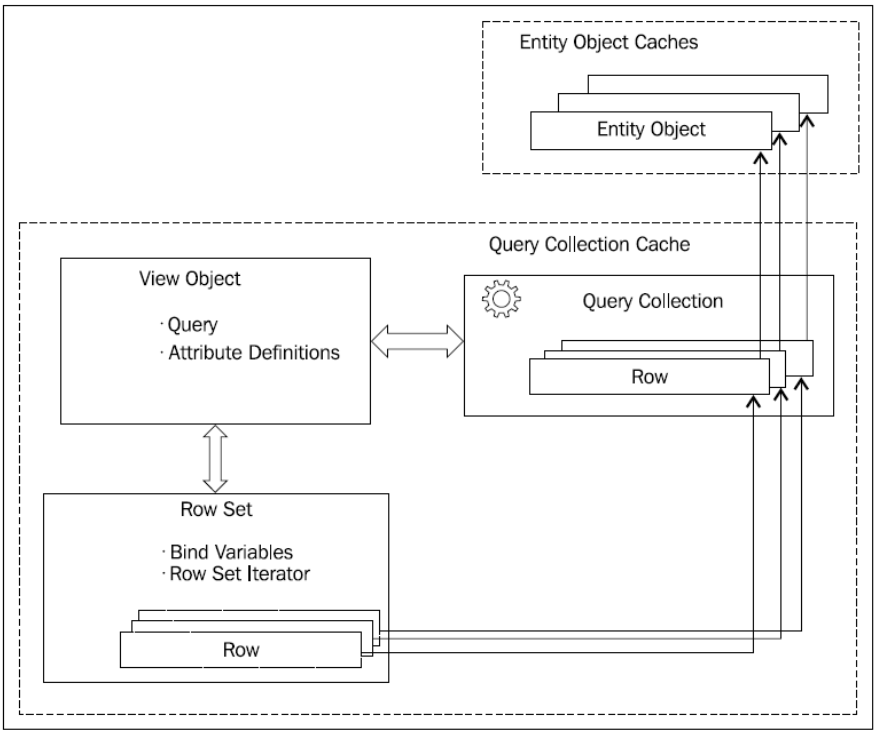


Slika 3.6. VO definicija je predložak za VO (VO instancu), sadrži VO atribute, temelji se na EO (ne uvijek, može i na SQL upitu); Izvor [8]

Na slici 3.6. vidi se i komponenta View Rows. View Rows se uvijek nalaze unutar skupa redaka, koji se naziva View Row set (ili samo Row set). Row set ima i barem jedan row set iterator, koji služi za iteriranje kroz row set. Jedan row set može imati više row set iteratora, a jedan VO može imati više row setova. Row set sadrži i pripadajuće Bind varijable.

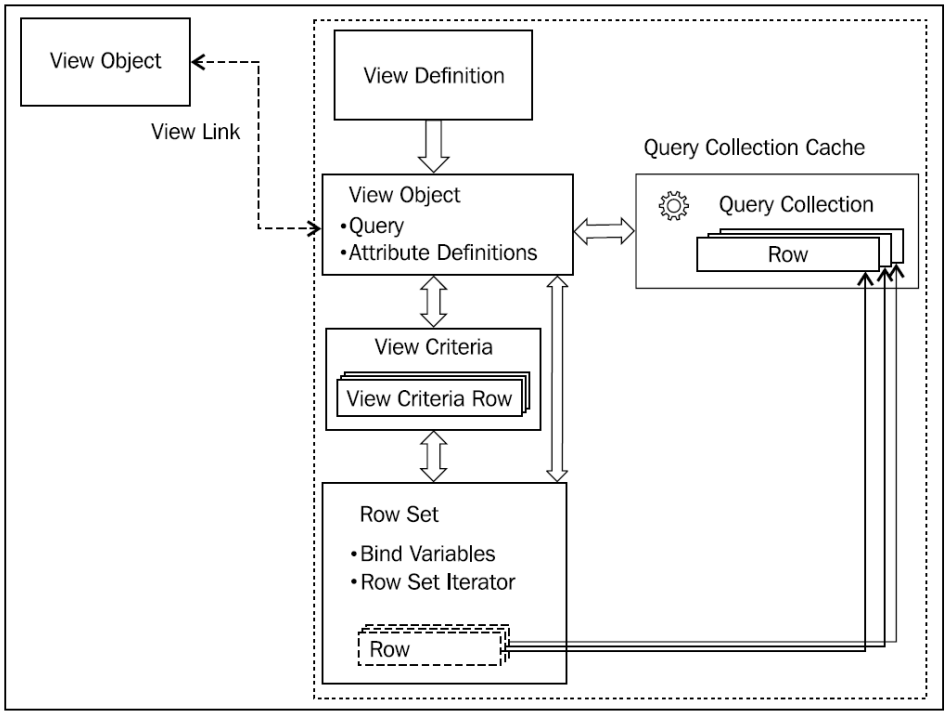
Treba naglasiti da row set ne sadrži potpune retke, već samo "pokazivače" na tzv. query collection (koji se ponekad naziva i View cache, slično kao entity cache; ponekad se naziva i View Object Row cache – zaista previše naziva). No, niti query collection ne sadrži uvijek retke. Naime, ako je VO temeljen na SQL upitu, onda query collection zaista sadrži potpune retke. Ako je VO temeljen na EO (jednom ili više), onda query collection sadrži samo "pokazivače" na entity cache.

Sljedeća slika 3.7. prikazuje VO (koji sadrži definiciju upita i definiciju atributa), koji ima Row set (na slici je prikazan samo jedan row set, ali VO ih može imati više). Row set ima Bind varijable i Row set iteratore (na slici je prikazan samo jedan row set iterator, ali row set ih može imati više). Kako slika prikazuje, row set sadrži retke, ali to su samo pokazivači, koji pokazuju na query collection. Budući da se pretpostavlja da je prikazani VO temeljen na EO (na slici je temeljen samo na jednom EO), query collection dalje sadrži pokazivače na entity cache.



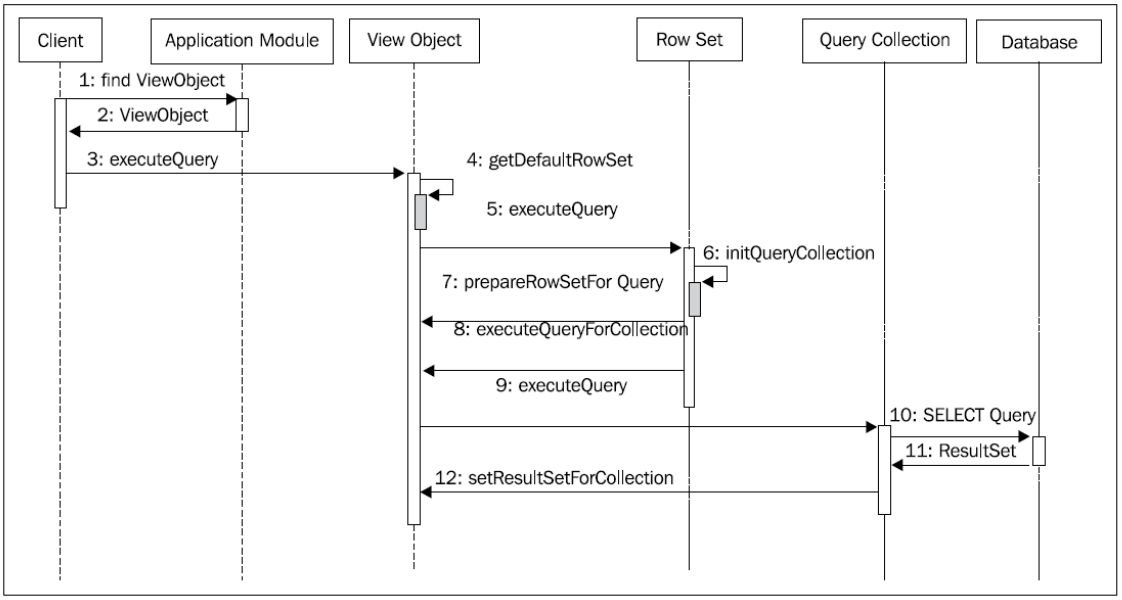
Slika 3.7. Odnos između VO, row seta, query collection i entity cachea; Izvor [11]

No to nije sve o VO komponentama. VO sadrži i tzv. View criteria, koji predstavljaju "filter" za upit definiran u VO, tj. služe za dinamičko nadograđivanje WHERE klauzule VO upita. Jedan VO se može povezati s drugim VO, pomoću tzv. View linka (slika 3.8.). View link može nastati na temelju asocijacije između EO na kojima su VO temeljeni, ali ne mora.



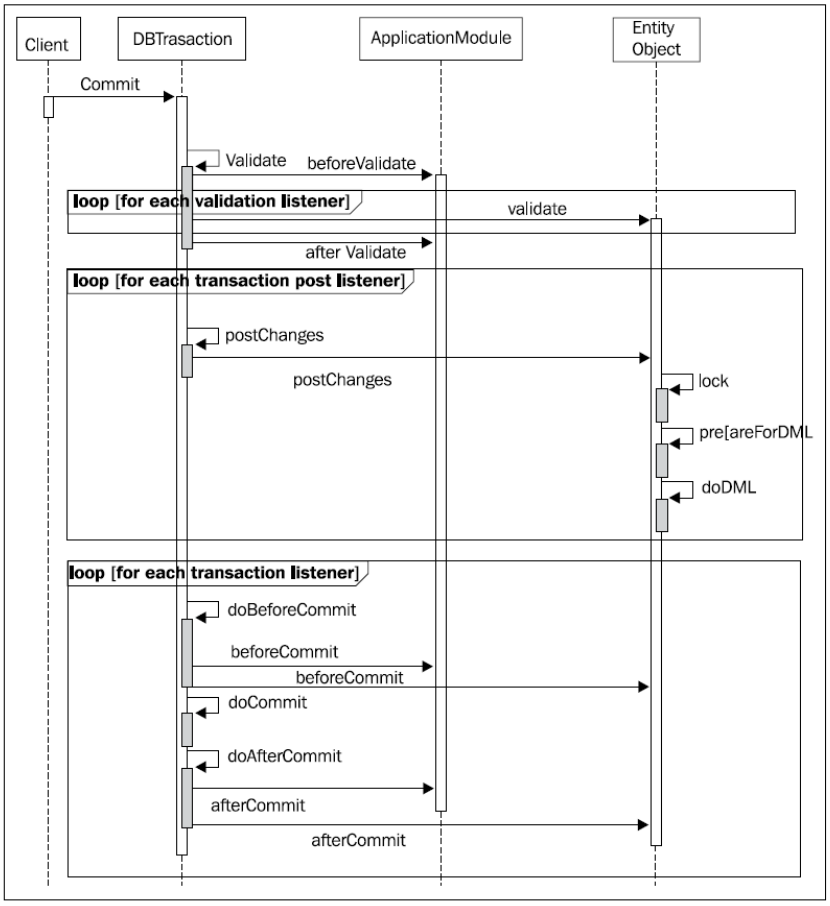
Slika 3.8. Još jedan prikaz VO komponenti – novost su View criteria i View link; Izvor [11]

Slika 3.9. prikazuje kako različite VO (i ne-VO) komponente rade upit nad bazom. Na slici se nalazi i jedna komponenta koja će biti objašnjena u sljedećem potpoglavlju – aplikacijski modul (Application Module). Treba napomenuti da slika nije potpuno detaljna za slučaj kada je VO temeljen na EO (jednom ili više). Naime, već je rečeno da u tom slučaju (kada je VO temeljen na EO), query collection ne sadrži potpune retke, već samo pokazivače na entity cache (koji na slici 3.8. nije prikazan). Dakle, u tom slučaju slici bi trebalo dodati i entity cache, te pokazati kako se on puni redovima, a query collection se puni samo s pokazivačima na entity cache.



Slika 3.9. Uloge različitih komponenti kod izvođenja upita nad bazom; Izvor [11]

VO služi za izvođenje upita nad bazom, ali ne služi (direktno) za ažuriranje baze. Ažuriranje baze radi se uvijek kroz EO (na kojem je VO temeljen), kako prikazuje slika 3.10.



Slika 3.10. Ažuriranje baze radi se kroz EO; Izvor [11]

Kako je prije rečeno, VO se može temeljiti na SQL upitu, ili na jednom ili više EO (zapravo se VO može temeljiti i na statičkim podacima, ali taj slučaj je puno rjeđi u praksi). Oracle priručnik [16] navodi (u 42.2.2) kako je poželjno (skoro) svaki VO, pa i onaj koji služi samo za čitanje (read only), temeljiti na EO - kaže se da dodatni posao punjenja entity cachea ne narušava performanse za više od 5% u odnosu na varijantu kada read only VO nije temeljen na EO (nego na SQL upitu). Prednost toga da se VO temelji na EO dolazi do izražaja naročito u slučaju kada se VO sastoji od više EO. Tada ADF može pokušati kreirati optimiziraniji upit nego što bi bio onaj ručno pisan. Dalje, tada se kod punjenja query collectiona pune i oba (ili više) entity cachea. Ako neki drugi VO (unutar istog aplikacijskog modula, kako će biti objašnjeno u sljedećem potpoglavlju) koristi jedan ili više tih istih EO, onda se neće morati uvijek izvršavati upit na bazu, jer će ADF naći podatke za drugi VO u entity cacheu koji je napunio prvi VO.

Kada se VO sastoji od više EO, a VO služi za ažuriranje (indirektno, jer se stvarno ažuriranje uvijek radi kroz EO), onda se obično samo jedan EO u VO označi kao Updateable, a ostali su vrste Reference. To ne mora uvijek biti tako - jedan VO može imati i više EO označenih kao Updateable, ali najčešće je samo jedan EO označen sa Updateable.

Budući da više VO može biti temeljeno na istom EO (bilo da je to primarni ili sekundarni EO), pitanje je što se dešava kada se kroz jedan VO (indirektno) mijenjaju podaci u EO (primarnom, najčešće), i ti se podaci još nisu poslali na bazu, tj. pitanje je koje podatke tada vidi drugi VO koji koristi isti EO (obično kao sekundarni EO). Ako su to VO u različitim aplikacijskim sesijama, onda drugi VO neće vidjeti mijenjane podatke, osim ako se koriste shared (na razini aplikacije, ne sesije) aplikacijski moduli, što će biti prikazano u sljedećem potpoglavlju. Ako su to VO u istoj aplikacijskoj sesiji, onda se željeno ponašanje može birati pomoću VO svojstva za View link consistency. Npr., može se izabrati da drugi VO vidi promjene koje je napravio prvi VO. Napomena: Oracle priručnik [16] u 42.1.2 kaže da je naziv View link consistency "povijesan" (mi bismo rekli zbunjujući) – njegova je primjena nekada bila ograničena samo na specijalne slučajeve, a sada vrijedi za svaki VO, neovisno da li se taj VO nalazi u View link vezi, ili ne.

View link consistency ponašanje može se mijenjati na razini aplikacijskog modula, kroz svojstvo jbo.viewlink.consistent (ili programski na razini VO). Moguće vrijednosti su:

- DEFAULT: znači da je za single EO usage, View link consistency uključen; za višestruke EO usages, View link consistency nije uključen samo za one sekundarne EO usages koji su označeni kao Updateable (tj. nisu označeni kao Reference);
- true: znači da je View link consistency uključen u svim slučajevima;
- false: znači da je View link consistency isključen u svim slučajevima.

Treba napomenuti da View link consistency ne radi (automatski se isključuje) kada se VO upitu dinamički (pomoću setWhereClause() metode) mijenja WHERE klauzula.

View link služi za "deklarativno" pristupanje podacima redaka-detalja iz master retka (ili obrnuto). U View Link Properties može se odabrati da li će view link biti jednosmjeran (unidirectional - default) ili dvosmjeran (bidirectional).

Za programsko pristupanje podacima koriste se View Link Accessor atribut i Java metode. Napomena: ovaj accessor nema nikakve veze sa već spomenutim accessor geter / seter metodama, koje se dobiju kada se odabere Include bind variable accesors / Include accesors kod generiranja vlastitih VO i VO Row klasa; također, ovaj accesor nije isto što i View Accessor, koji služi npr. kod korištenja LOV-ova i kod validacije. Ako se želi optimizirati pristup pomoću View Link Accessora, može se na VO, u Tuning sekciji stranice General, označiti Retain View Link Accessor Row Set. Za programski pristup, tj. preko View Link Accessor metoda, nije potrebno da postoji view link između VO (za razliku od "deklarativnog" pristupa).

Osim navedenih svojstava jbo.viewlink.consistent i Retain View Link Accessor Row Set, VO ima još svojstava koja služe za optimizaciju. U Tuning sekciji stranice General (za VO), mogu se postaviti i sljedeća svojstva, vezana za dohvaćanje redaka iz baze:

- All rows ili Only up to row number: označava da li nema ograničenja na broj redaka, ili je broj redaka ograničen na uneseni broj;
- At Most one Row: korisno je kada se samo želi vidjeti da li ima redaka u nekom upitu;
- No Rows: korisno je kada se redovi samo unose.

Za izbor All rows ili Only up to row number, moguće je dalje odabrati sljedeće:

- In Batches of (ili fetch size): označava koliko redaka odjednom ADF dohvaća sa baze (u Formsima se to svojstvo bloka zove Query Array Size); ako se pusti podrazumijevana (default) vrijednost, koja je 1, to za većinu slučajeva nije dobro;
- As needed ili All at once: označava da se redovi sa baze dohvaćaju kad je potrebno (npr. za prikaz na ekranu, ili programski dohvat), ili se svi redovi dohvaćaju odjednom (kao kada se u Formsima odabere Query All Records na bloku), npr. zbog izračuna nekog sumarnog polja.

Treba napomenuti još jedno važno VO svojstvo - Auto Refresh. Ono će ukratko biti objašnjeno u sljedećem potpoglavlju, kad bude riječi o dijeljenim (shared) aplikacijskim modulima.

Može se napraviti usporedba između Forms bloka i ADF EO i VO komponenti. Može se reći da je Forms blok monolitan, tj. sadrži funkcionalnosti koje su u ADF-u podijeljene između EO i VO komponenti (zapravo, Forms blok ima i neke funkcionalnosti, vezane za korisničko sučelje i navigaciju, koje u ADF arhitekturi pripadaju ADF View i ADF Controller slojevima). U odnosu na ADF EO i VO, Forms blok ima barem ove dvije (velike) mane:

- u master-detalj relaciji, Forms u detaljnom bloku ne može istovremeno sadržavati detalje za više mastera; za razliku od toga, jedan VO može imati više row setova, pa i query collectiona; u ADF-u je zato lako napraviti rješenje za situaciju "master-detalj-detalj od detalja" u istoj transakciji; zbog toga u ADF-u nije potrebno raditi POST-iranje (barem ne zbog master-detalj potreba), kao što je ono spomenuto kod Formsa u potpoglavlju 2.6.;
- Forms blok se ne može pretraživati ili sortirati bez upita na bazu (zapravo, moguće je da se programski rade trikovi, npr. pomoću Record group, ali takva rješenja su neskalabilna i nisu elegantna); za razliku od toga, VO se može pretraživati, i njegovi podaci sortirati, bez upita na bazu, što će se kratko prikazati u nastavku.

Kada se izvodi upit (query) nad VO, može se odrediti koji će se izvor podataka koristiti:

- Scan database tables: čita se baza, što je podrazumijevano (default) ponašanje;
  postavlja se programski sa vo.ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES;
- Scan view rows: čita se query collection (kolekcija prvo mora biti napunjena upitom na bazu);
  postavlja se programski sa vo.ViewObject.QUERY_MODE_SCAN_VIEW_ROWS;
- Scan entity cache: čita se entity cache (moguće je samo za VO temeljene nad EO);
  postavlja se programski sa vo.ViewObject.QUERY_MODE_SCAN_ENTITY_ROWS.

Slijedi primjer iz [11], u kojem se vidi i primjena kombinacije tih postavki:

```
// Defined In custom ApplicationModuleImpl
public void queryUsingMulitpleQueryModes(){
   //Find the view object
   ViewObject employeeVO = findViewObject("EmployeeDetails");

   //Combine both database mode and entity cache query mode
   employeeVO.setQueryMode(
      ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES |
      ViewObject.QUERY_MODE_SCAN_ENTITY_ROWS);

   //Execute query
   employeeVO.executeQuery();

   // Business logic to manipulate row goes here
}
```

Također, VO omogućavaju sortiranje redaka u memoriji, što je korisno npr. i za slučaj kada se želi sortirati po VO tranzijentnim podacima (koji niti ne postoje na bazi). Sortiranje redaka u memoriji može raditi sporo, ako je broj redaka jako velik – treba uzeti u obzir da je baza (tj. DBMS sustav) ipak puno pogodnija za sortiranje velikog broja redaka. Primjer sortiranja:

```
vo.setSortBy("ImeAtributa desc");
```

VO omogućavaju i filtriranje podataka u memoriji. Postoje dva (programska) načina:

- filtriranje (u memoriji) pomoću RowMatch - koristi se klasa oracle.jbo.RowMatch;
- filtriranje (u memoriji) nadjačavanjem metode rowQualifies() iz ViewObjectImpl klase.

I u View criteria potkomponenti moguće je definirati izvor podataka. Moguće postavke su:

- Database: čita se baza, što je podrazumijevano (default) ponašanje;
  programski se postavlja sa vc.setCriteriaMode(ViewCriteria.CRITERIA_MODE_QUERY);
- In-memory: čita se u memoriji;
  programski se postavlja sa vc.setCriteriaMode(ViewCriteria.CRITERIA_MODE_CACHE);
- Both: prvo se postavlja upit na bazu, a onda se dodaje upit u memoriji;
  programski se postavlja kombinacijom prethodne dvije metode:

```
vc.setCriteriaMode
   (ViewCriteria.CRITERIA_MODE_QUERY |
    ViewCriteria.CRITERIA_MODE_CACHE).
```

3.4. Application Module

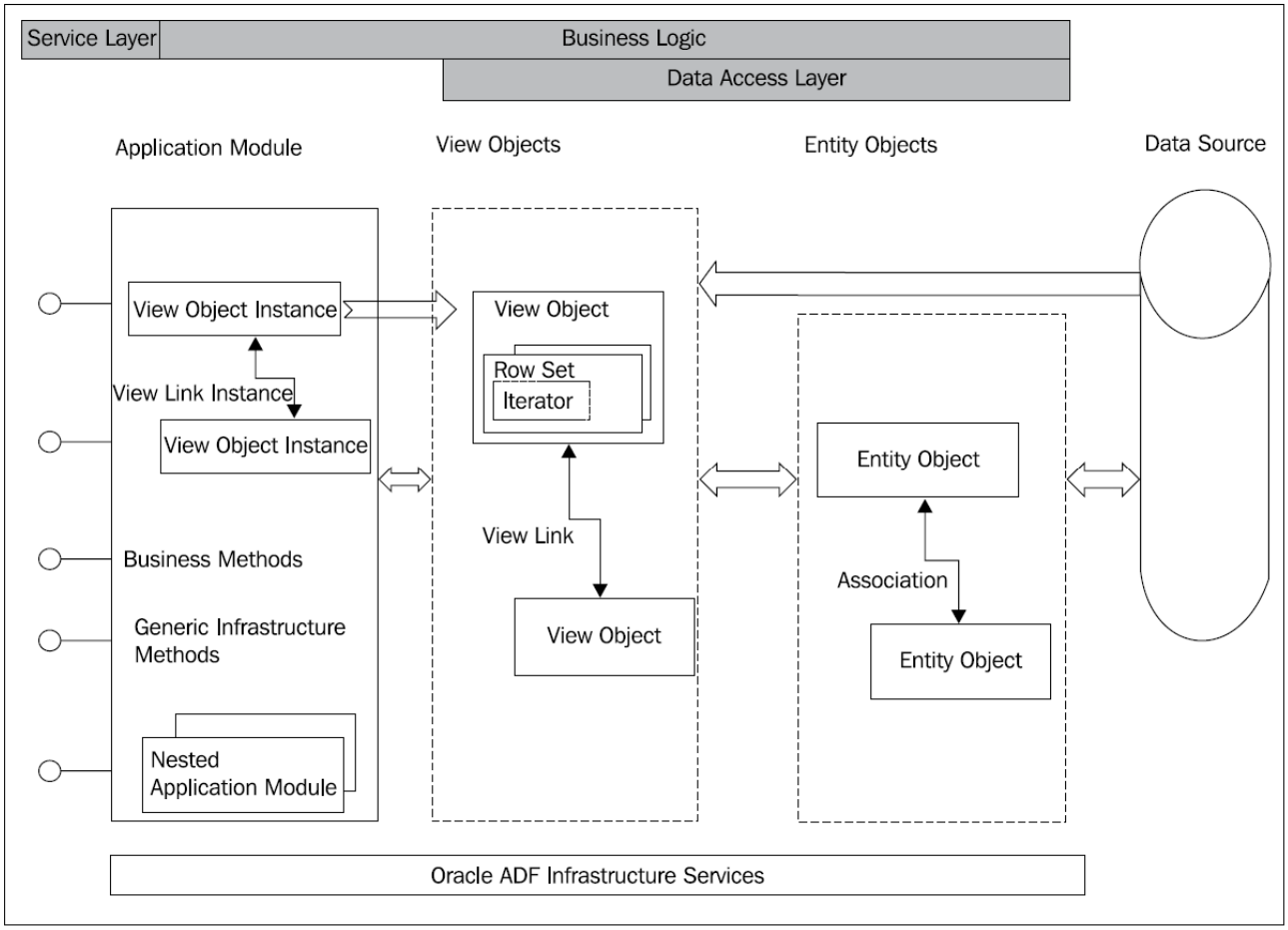
ADF BC Application Module (ADF BC AM, ili samo AM) ima tri potkomponente koje su formalno slične odgovarajućim potkomponentama za EO i VO:

- AM XML definicija (AM definition XML metadata file): XML datoteka koja sadrži metapodatke za AM;

- AM klasa za definiciju (AM definition): to je klasa koja predstavlja definiciju za AM tokom izvođenja; podrazumijevana (default) klasa je oracle.jbo.server.ApplicationModuleDefImpl, a može se napraviti podklasa te klase, što se rjeđe radi;

- AM klasa: ta klasa predstavlja instance danog AM, pa se često naziva i AM Instance; podrazumijevana (default) klasa je oracle.jbo.server.ApplicationModuleImpl; može se napraviti podklasa te klase, što se obično i radi.

Slika 3.11. prikazuje glavne sastavne dijelove AM, a to su VO instance (zajedno s View link instancama), te drugi (ugniježđeni) AM.



Slika 3.11. Glavni sastavni dijelovi AM, te prikaz svih najvažnijih BC komponenti i njihovih odnosa; Izvor [11]

Slika 3.11. daje i kompletan prikaz svih najvažnijih BC komponenti i njihovih odnosa. Vidi se da AM ne koristi EO na direktan način, već indirektno, preko VO.

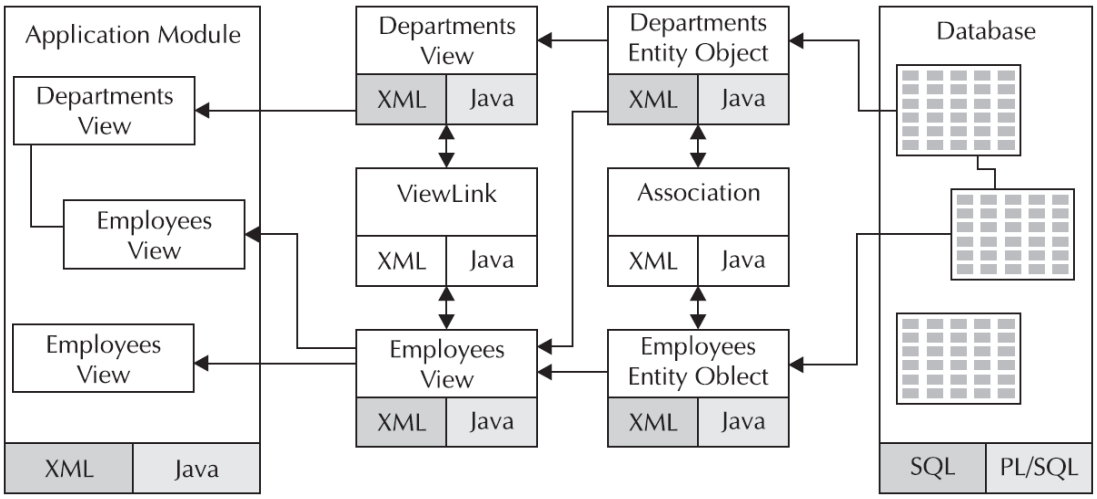
VO i EO čine sloj pristupa podacima (Data Access Layer).

Sve tri glavne komponente zajedno (AM, VO i EO) čine sloj poslovne logike (Business Logic).

Na "rubu" AM je servisni sloj, tj. ono što je izloženo gornjim slojevima, a izložene su VO instance, vlastite (uglavnom Java) metode, generičke metode koje pruža ADF, te ugniježđeni AM.

Sve zajedno čini ADF infrastrukturne servise.

Još jedan dobar prikaz najvažnijih BC komponenti i njihovih odnosa daje slika 3.12. Na toj se slici vidi da jedan AM (odnosno AM instanca) može sadržavati dvije (ili više) instance istog VO. Npr. AM instanca na slici ima dvije instance Employees VO, od kojih je jedna vezana View linkom na drugi VO (Departmens) u master-detalj vezu, a druga instanca je samostalna.



Slika 3.12. Još jedan prikaz svih najvažnijih BC komponenti i njihovih odnosa; Izvor [8]

Osim prethodno navedenih komponenti, postoje još neke komponente koje ADF koristi za izvođenje AM instance:

- Application Pool: odgovoran je za upravljanje pričuvom (pool) instanci AM; podrazumijevana (default) klasa je oracle.jbo.common.ampool.ApplicationPoolImpl; u bc4j.xcfg datoteci određenom AM može se postaviti svojstvo PoolClassName tako da pokazuje na neku drugu klasu;

- Connection Strategy: Application pool koristi ovu komponentu za kreiranje novih instanci AM i za konektiranje na izvor podataka (uglavnom bazu); podrazumijevana (default) klasa je oracle.jbo.common.ampool.DefaultConnectionStrategy; u bc4j.xcfg datoteci može se postaviti svojstvo jbo.ampool.connectionstrategyclass tako da pokazuje na neku drugu klasu;

- Session: session objekt sprema kontekst sesije za klijenta; session objekt se instancira za svaki vršni (root) AM, kada se taj AM aktivira; podrazumijevana (default) klasa je oracle.jbo.server.SessionImpl; u bc4j.xcfg datoteci može se postaviti svojstvo SessionClass tako da pokazuje na neku drugu klasu;

- Transaction Handler: upravlja transakcijom (baze), koja je vezana za korisničku sesiju; podrazumijevana (default) klasa je oracle.jbo.server.DefaultTxnHandlerImpl; može se postaviti drugačije, na malo kompleksniji način nego što je bilo kod prethodnih komponenti.

Navedene podrazumijevane (default) komponente se, zapravo, vrlo rijetko zamjenjuju s nekim drugim komponentama.

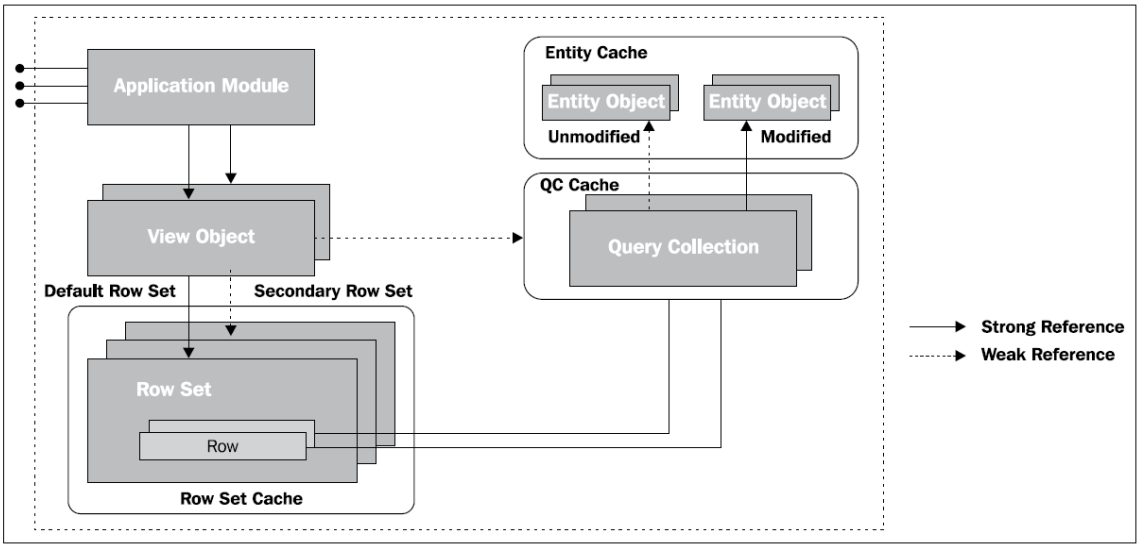
Često je potrebno dijeliti podatke između više korisničkih sesija (opet napominjemo da korisnička sesija nije isto što i sesija baze, jer se jedna korisnička sesija može realizirati kroz više sesija baze, što je standardno u web aplikacijama). Naročito to vrijedi za šifarnike i slične podatke, koji se rijetko mijenjaju, a potrebni su skoro svim korisnicima (poslovne aplikacije). Uglavnom nije dobro da su takvi podaci multiplicirani u memoriji aplikacijskog servera, već je poželjno da se u memoriji nalaze samo jednom i da sve aplikacijske sesije koriste iste podatke.

Za potrebe dijeljenja podataka, ADF ima tzv. dijeljene (ili zajedničke - shared) AM instance. AM instanca može biti dijeljena na razini aplikacije (application level shared application module), pa tada svi korisnici koriste istu AM instancu i vide iste podatke. AM može biti dijeljena i samo na razini korisničke sesije (session level shared application module), pa tada sve AM instance u istoj aplikacijskoj sesiji, koje se nalaze unutar vršnog (root) AM (unutar kojeg se nalazi i djeljiva AM instanca) vide njene podatke (AM koje se nalaze u istoj aplikacijskoj sesiji, ali unutar drugog vršnog AM, ne vide njene podatke).

AM se označava djeljivim tako da se izabere na odgovarajućem tabu, Application ili Session, unutar Project Properties, ADF Business Components, Application Module Instances.

VO instance koje se nalaze u AM dijeljenom na razini aplikacije, koriste više row iteratora za isti row set, kako bi svaki korisnik mogao nezavisno iterirati kroz podatke. Ako je VO upit parametriziran, onda se za različite korisnike kreiraju i njihovi row setovi, te različiti query collection, ali se uvijek koristi isti entity cache.

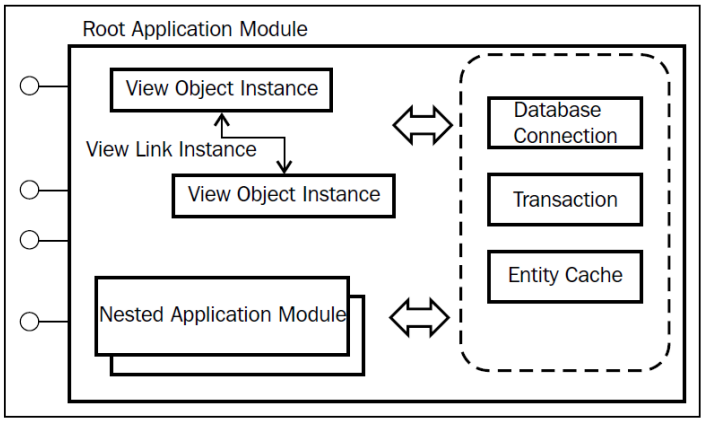
Query collection koja se nalazi u ne-dijeljenom (non-shared) AM je labavo referencirana (weakly referenced), dok je query collection koja se nalazi u dijeljenom AM čvrsto referencirana (strongly referenced). To je važno za JVM, jer garbage collector može ukloniti one objekte koji su labavo referencirani, ali ne i one koji su čvrsto referencirani. Slika 3.13. pokazuje labavo referencirane i čvrsto referencirane VO potkomponente kod ne-dijeljenog AM. Vidi se npr. da je samo primarni row set čvrsto referenciran, a sekundarni row setovi su labavo referencirani. U djeljivom AM su sve VO potkomponente čvrsto referencirane.



Slika 3.13. Labavo i čvrsto referencirane VO potkomponente kod ne-dijeljenog AM; Izvor [11]

Važno je napomenuti da svi ugniježđeni AM, koji se nalaze unutar istog vršnog (root) AM, dijele zajednički entity cache, konekciju na bazu i transakciju, kako prikazuje slika 3.14. AM dijeljeni na razini korisničke sesije, zapravo se kreiraju se kao ugniježđeni AM.

Osim eksplicitno, AM se mogu gnijezditi i implicitno. Naime, svi AM koji se nalaze unutar istog task flowa automatski se gnijezde. O task flowu bit će govora u potpoglavlju 3.7.



Slika 3.14. Ugniježđeni AM dijele zajednički entity cache, konekciju na bazu i transakciju; Izvor [11]

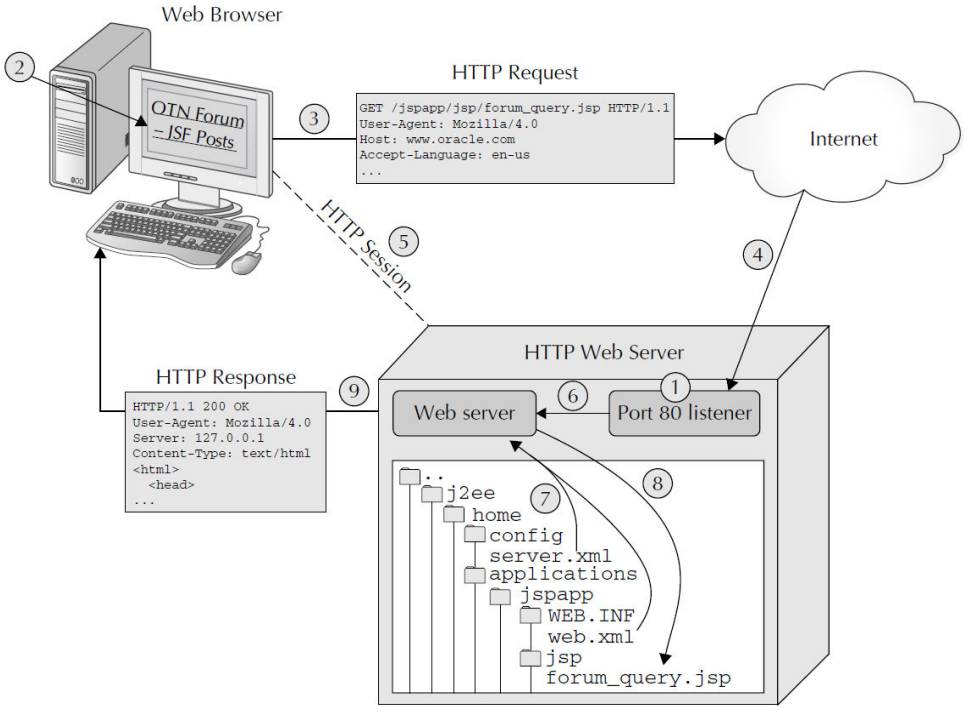
U prethodnom potpoglavlju (3.3.) napomenuto je da VO, uz svojstva koja su tamo prikazana, ima i svojstvo Auto Refresh, koje se može postaviti u Tuning sekciji stranice General. Iako to svojstvo nije usko vezano za dijeljene (shared) AM, ipak se najčešće koristi uz njih. To se svojstvo može koristiti na Oracle bazi 10.2 ili većoj, koje podržavaju tzv. Continuous Query Notification (CQN). CQN od baze 11g postoji u dva oblika [12], kao Object Change Notification (OCN) i Query Result Change Notification (QRCN). Suština je u tome da baza gura (push) informacije o promjeni redaka prema klijentu (naravno, klijent može biti i aplikacijski server), a klijent na temelju dobivenih informacija izvršava upit i osvježava svoj cache. ADF kroz JDBC API prije izvršenja VO upita šalje registraciju tog upita bazi. U ADF-u se najčešće tako osvježavaju VO koji se nalazi u AM dijeljenom (shared) na razini aplikacije.

3.5. Kako pomiriti HTTP stateless protokol i statefull zahtjeve

HTTP protokol radi iznad TCP / IP (ili UDP / IP) para protokola, a kao HTTPS iznad TLS (SSL) protokola (koji radi iznad TCP protokola). Poznato je da HTTP protokol izvorno nije bio mišljen kao temelj za rad poslovnih aplikacija (koje koriste transakcije), već za primanje zahtjeva (requests) i za slanje (response) statičkih HTML stranica.

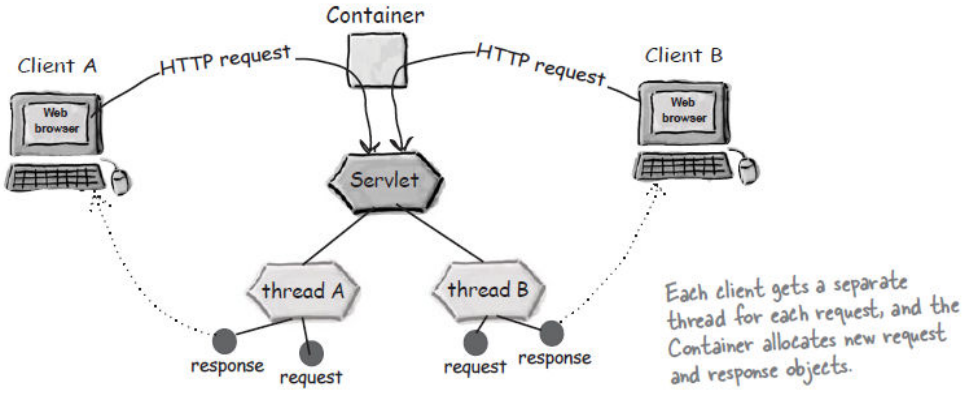
Vrlo brzo je bilo potrebno prikazivati dinamičke stranice, što se u Java tehnologiji prvo radilo (a i danas je to temelj rada) pomoću Java servleta. Pojednostavljeno rečeno, Java servleti su Java programi na aplikacijskom serveru, koji dinamički generiraju web stranice.

Slika 3.15. prikazuje korake u HTTP komunikaciji. Vidi se da (1) HTTP listener osluškuje zahtjeve (requests) na portu 80 (default). Korisnik u web pregledniku klikne na neki link (2), čime pokreće HTTP zahtjev (3), koji preko interneta dolazi do HTTP listenera (4). HTTP listener uspostavlja komunikaciju s klijentom (5) i predaje kontrolu web server programu (6). Web server obrađuje zahtjev (7). Ako je sadržaj dinamički, web server poziva odgovarajući program, npr. Java servlet, koji generira stranicu (8). Na kraju web server šalje stranicu (statičku ili dinamički generiranu) klijentu, a klijentov web preglednik prikazuje tu stranicu (9). Time se HTTP komunikacija (request-response par) završava, a TCP veza u pravilu ostaje i dalje aktivna (tzv. perzistentna TCP veza).



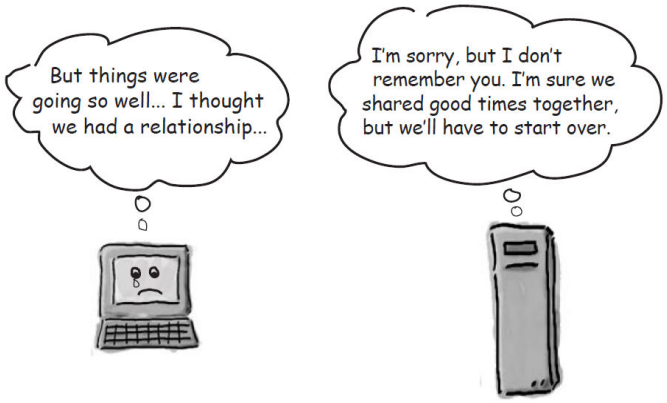
Slika 3.15. Koraci u web (HTTP) komunikaciji; Izvor [8]

Slika 3.16. prikazuje osnove rada Java servleta. Java servlet svaki zahtjev obrađuje u posebnoj Java dretvi (thread), koja referencira svoje vlastite request-response objekte. Na kraju svakog HTTP request-response para, dretva se zatvara (ili vraća u thread pool), a request-response objekte će počistiti garbage collector.



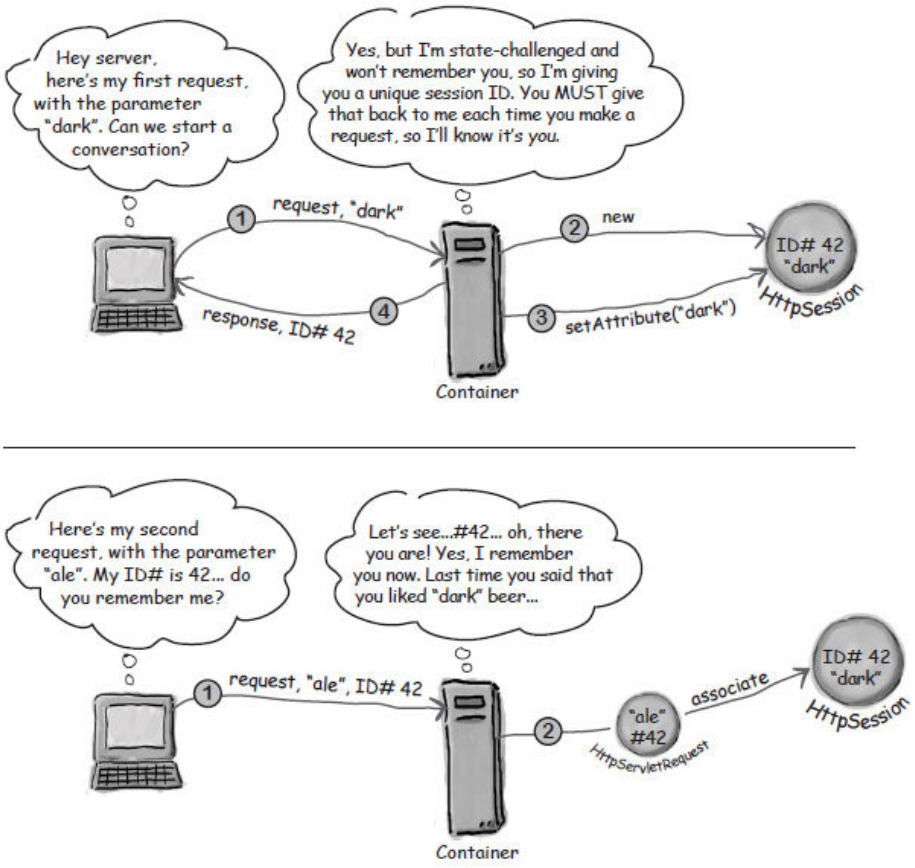
Slika 3.16. Osnove rada Java servleta; Izvor [1]

Dakle, HTTP protokol je stateless, tj. "ne čuva stanje". To je jako nezgodno npr. za poslovne transakcije. Kako duhovito prikazuje slika 3.17., nakon završetka HTTP request-response komunikacije, web server zaboravlja klijenta, tj. ne prepoznaje ga kod sljedećeg HTTP zahtjeva.



Slika 3.17. Nakon završetka HTTP komunikacije, web server zaboravlja klijenta; Izvor [1]

Naravno, znamo da se poslovne transakcije ipak obavljaju preko HTTP protokola, pa je jasno da je za navedeni problem nađeno (kakvo-takvo) rješenje. Riječ je o kolačićima (cookies), koji služe za različite namjene, ali i za realizaciju aplikacijskih sesija (jedna aplikacijska sesija sastoji se od jednog ili više HTTP request-response parova; kako je već rečeno, aplikacijska sesija nije isto što i sesija baze). Kako prikazuje slika 3.18., kod uspostavljanja HTTP komunikacije, aplikacijski server kreira novi ID sesije i šalje ga klijentu (web pregledniku). Kod svake sljedeće komunikacije, klijent šalje serveru taj ID sesije, na temelju čega ga server prepoznaje.



Slika 3.18. Realizacija aplikacijske sesije preko HTTP protokola; Izvor [1]

Prikazani način rada ne koristi se samo kod Java servleta, već i novijih tehnologija JSP (Java Server Pages) i JSF (Java Server Faces), koje se temelje na Java servlet tehnologiji (JSF se temelji i na JSP).

Treba napomenuti da je moguće realizirati aplikacijske sesije i onda kada klijent (iz nekog razloga) ne dozvoljava kolačiće. Tehnika se zove URL rewriting, jer svaki URL poziv tada dobiva oblik: URL+;jsessionid=1234567. Korisnik ju ne može spriječiti, ali treba ju eksplicitno programirati u odgovoru (response). Server prvo pokuša raditi s kolačićima, a ako ne uspije, prebacuje se na URL rewriting način.

3.6. Save Points, Application Module Pools i Connection Pools

U prethodnom potpoglavlju prikazane su osnove rada s kolačićima za sesije (session cookies), pomoću kojih HTTP stateless protokol radi sa aplikacijskim sesijama koje se protežu kroz dva ili više HTTP request-response para. No često poslovna aplikacija ima zahtjeve koji su složeniji od jednostavne identifikacije aplikacijske sesije, a to su zahtjevi za upravljanjem aplikacijskim stanjem (application state management).

ADF omogućava upravljanje aplikacijskim stanjem na dvije razine. Jedno je upravljanje na controller sloju, pomoću Save For Later mogućnosti u Task Flowu (TF), a drugo je upravljanje aplikacijskim modulima (AM) na model sloju.

Da bi se u aplikaciji koristila mogućnost Save for later, u aplikaciji treba postaviti odgovarajuće Save Points, tj. točke u kojima želimo da ADF zapamti stanje aplikacije i podataka, kako bi se kasnije moglo vratiti stanje koje je bilo u tim točkama, pomoću Save Point Restore. Važno je reći da ovdje nije riječ o pamćenju podataka na bazi (to nije SAVEPOINT na bazi), već o pamćenju stanja aplikacije i podataka u aplikaciji. Save for later može raditi (tj. snimiti podatke) eksplicitno, tj. na korisnikov zahtjev (ali to programer treba programirati) ili implicitno, npr. kada istekne timeout sesije, ili kada korisnik zatvori prozor web preglednika.

Važno je napomenuti da su spomenuti Save points (bilo eksplicitni, bilo implicitni) zapravo Controller Save Points. Uz njih, ADF ima i Model Save Points. Za razliku od Controller save point, koji pamti podatke i na razini controller i model sloja, Model save point pamti podatke samo na razini model sloja. Možemo reći da Model save point pamti stanje transakcije (ali, ne na bazi, nego u aplikaciji), a ne cjelokupne aplikacije. Kao i Controller save point, tako se i Model save point može pozvati implicitno (npr. automatski kod ulaza u novi TF, koji dijeli data controle s TF koji ga je pozvao) ili eksplicitno, programski. Model savepoint se može primijeniti sve dok commit ili rollback operacija ne kompletira transakciju.

Još važnija (i složenija) mogućnost od Save For Later, je upravljanje aplikacijskim modulima (AM) na model sloju. Za te potrebe koriste se Application Module Pools i Connection Pools (pričuve AM instanci, pričuve konekcija).

Application module pool (AM pool) je kolekcija AM instanci iste vrste (tj. iste AM definicije). AM pool omogućava da veći broj korisnika može (kvazi) istovremeno raditi na manjem broju AM instanci, time štedeći memoriju i procesne mogućnosti aplikacijskog servera (ili klijenta - AM pool, kao i cijeli ADF BC, može raditi i na klijentu, u klijent-server arhitekturi).

AM instanca u poolu može biti u jednom od tri stanja:

- bezuvjetno slobodna za korištenje (ili nereferencirana), bilo kom korisniku;
- slobodna za korištenje, ali referencirana na aplikacijsku sesiju koja ju je prethodno koristila i koja (aplikacijska sesija) još nije završila; u ovom slučaju AM instanca može se ipak predati drugom korisniku, ovisno o tzv. AM State Management Release Levelu, pri čemu će standardno doći do tzv. pasivacije (a kasnije aktivacije) AM instance, kako će kasnije biti prikazano;
- zauzeta – neki korisnik (odnosno njegova Java dretva na aplikacijskom serveru) trenutačno koristi tu AM instancu.

Osim AM poola, postoji i connection pool. Postoje dvije vrste connection poolova, koje se koriste u ovisnosti o tome da li se konekcije konfiguriraju kao JDBC URL konekcije, ili JNDI name for a data source konekcije. Ako se koriste JDBC URL konekcije, samo tada se koristi ADF connection pool (u nastavku se pretpostavlja da se koriste samo JDBC URL konekcije). Osnovno pravilo za ADF connection pool je: po jedan connection pool (dakle, skup konekcija, a ne jedna konekcija) se kreira za svaki par <JDBCURL, Username> na svakom JVM-u (aplikacija može raditi na više JVM instanci), pri čemu konekciju zahtijeva root AM instanca (ugniježđene AM instance koriste istu konekciju kao njihova root AM instanca).

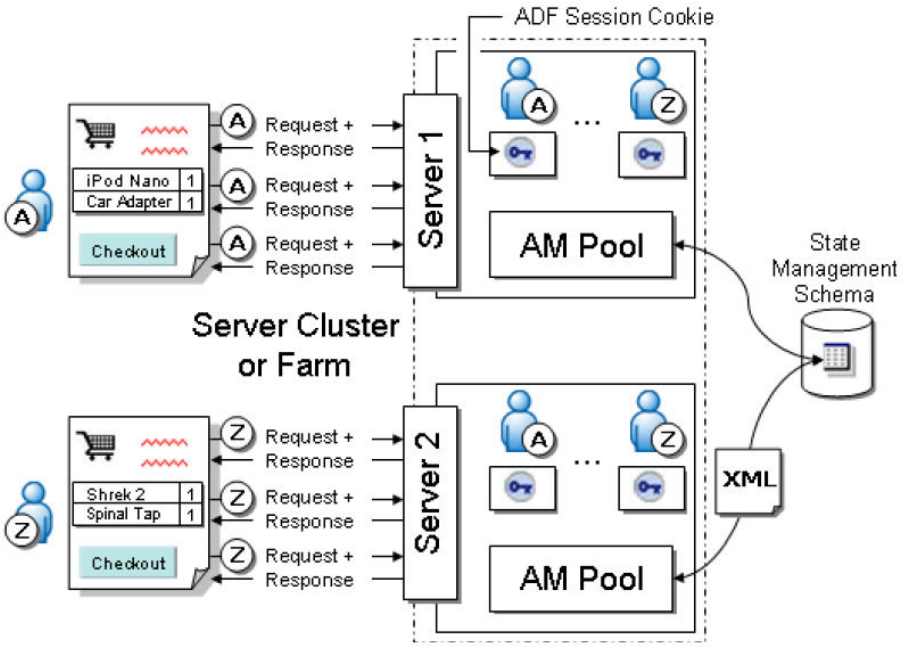
Kako je prethodno spomenuto, postoje AM State Management Release Leveli, koji se (općenito, ili za određenu AM instancu) mogu birati deklarativno ili programski. Postoje tri mogućnosti:

- Managed release level: on se podrazumijeva (default); ADF pokušava dodijeliti istu AM instancu određenoj aplikacijskoj sesiji, dok to može; kad ne može, doći će do pasivacije AM instance, čime se sprema aplikacijsko stanje, a kasnije do aktivacije (u nastavku se ti procesi detaljnije prikazuju);

- Unmanaged release level: nikakvo stanje se ne sprema između više HTTP request-response parova iste aplikacijske sesije; to je potpuno stateless ponašanje;

- Reserved release level: to je 1:1 veza između AM instance i aplikacijske sesije (preko data controle); to je potpuno statefull ponašanje, koje se ne preporučuje kod web aplikacija, osim kad zatreba.

Slika 3.19. prikazuje osnovnu arhitekturu za AM state management. Na slici su prikazane dvije aplikacijske sesije, koje se odvijaju (svaka) kroz tri HTTP request-response para. Postoje dvije instance aplikacijskog servera (zbog veće raspoloživosti), od kojih svaka ima svoje AM poolove. Aplikacijski serveri (tj. ADF runtime) spremaju (nakon svakog request-response para, ili samo kad dođe do potrebe za pasivacijom) stanje AM instance u XML, a taj XML sprema se u BLOB polje određene tablice u bazi (opcionalno se može spremati i u datoteku na serveru).



Slika 3.19. Osnovna arhitektura za AM state management; Izvor [16]

Prikažimo ukratko što se dešava kod pasivacije i aktivacije AM instance, a to se dešava onda kada se za AM koristi Managed release level (default):

- kod prvog HTTP request-response para nove aplikacijske sesije, aplikacijskoj sesiji dodjeljuje se neka nereferencirana (pretpostavimo da takva postoji) AM instanca iz AM poola;

- na kraju HTTP requesta, ta se AM instanca vraća u AM pool, ali sada kao referencirana (od strane data controla, koji je vezan za aplikacijsku sesiju); na taj način čuvaju se u memoriji npr. svi EO i VO cachevi, kursori baze i dr. za aplikacijsku sesiju, što je svakako brže nego da se oni pune svaki put kod novog HTTP request-response para iste aplikacijske sesije; treba napomenuti da se uobičajeno (default) čuva i veza između AM instance i konekcije (iz connection poola); dok god ne dođe do pasivacije, aplikacija se ponaša kao prava stetefull aplikacija;

- kod sljedećeg HTTP requesta, aplikacijskoj sesiji dodjeljuje se "njena" AM instanca, čime se uobičajeno (default) dodjeljuje i ista konekcija na bazu (što znači i ista sesija na bazi);

- kada neka druga aplikacijska sesija zatraži AM instancu iz AM poola, ali više nema slobodnih instanci, ADF runtime gleda da li ima referenciranih AM instanci koje trenutno nisu zauzete (rezervirane AM instance se nikada ne diraju); ako postoji referencirana i nezauzeta AM instanca, njeno stanje se prvo pasivira u XML oblik (XML se, kako je već rečeno, standardno sprema u BLOB polje tablice na bazi, ali može i u datoteku); AM instanca se onda "isprazni" i preda drugoj aplikacijskoj sesiji na korištenje;

- kod sljedećeg HTTP requesta od strane prve aplikacijske sesije, ADF runtime joj (kako je prethodno već prikazano) traži nereferenciranu AM instancu (ako takva postoji), ili radi postupak pasivacije nad AM instancom koja je referencirana (ali nezauzeta) od strane druge aplikacijske sesije; nakon toga izvodi se postupak aktivacije, tj. podaci koji su spremljeni u XML (kod pasivacije) sada služe za punjenje "prazne" AM instance.

Ukoliko se desi da neki zahtjev ne može biti zadovoljen (npr. nema nezauzetih instanci, ili su sve instance u rezerviranom modu), a AM pool je dostignuo maksimum, aplikacijska sesija se prekida.

Treba napomenuti da do pasivacije može doći još u dva slučaja. Jedan je implicitan, kada aplikacijskoj sesiji istekne vrijeme (timeout). Drugi je slučaj kada se parametar jbo.dofailover postavi na true (default je false), pa se pasivacija radi svaki put (za potrebe visoke raspoloživosti), tj. na kraju svakog HTTP requesta, a ne samo onda kada AM instancu treba dati drugoj aplikacijskoj sesiji.

Kod pasivacije se spremaju dvije vrste podataka - transakcijski i ne-transakcijski podaci. Transakcijski podaci su podaci o novim, mijenjanim i brisanim redovima iz EO cacheva koji pripadaju root AM instanci te aplikacijske sesije (za mijenjane retke čuvaju se i stare i nove vrijednosti).

Ne-transakcijski podaci su sljedeći, za svaki VO (bez obzira da li je VO kreiran statički ili dinamički):

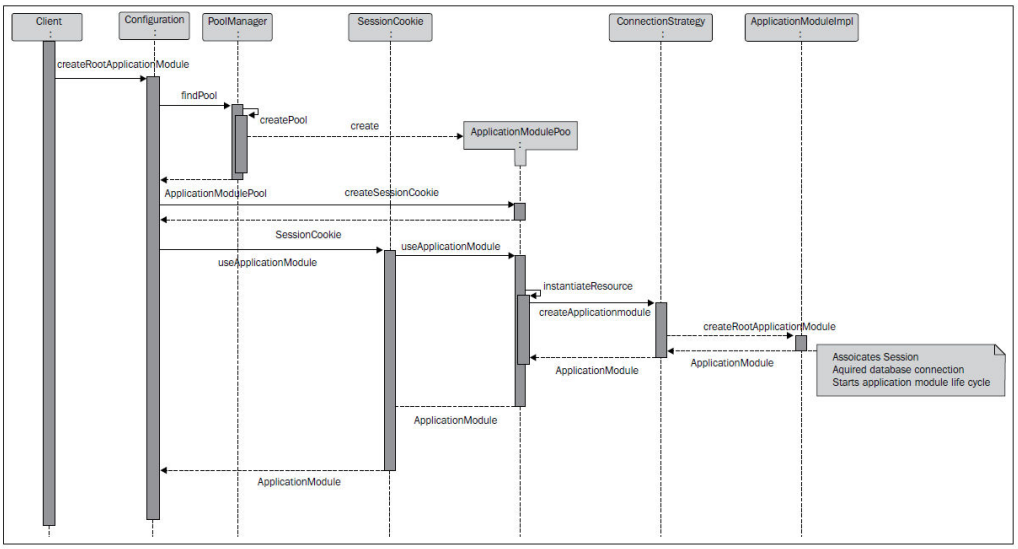
- indikator tekućeg retka, te novi redovi i njihove pozicije;
- ViewCriteria i odgovarajući parametri;
- flag koji pokazuje da li je row set bio izvršen;
- Range start i Range size, Acces mode, Fetch mode i Fetch size;
- bilo koji drugi VO podaci (napomena: i tranzijentni VO podaci se mogu snimiti, ako je tako odabrano na VO tranzijentnim atributima za vrijeme dizajna);
- SELECT, FROM , WHERE, ORDER BY klauzule koje su kreirane ili mijenjane dinamički.

Kako je već rečeno, kod pasivacije se stvara XML datoteka, koja se standardno (default) sprema u BLOB polje određene tablice na bazi. Parametrom jbo.passivationstore može se birati da li se želi spremati u bazu ili datoteku, a vrijednosti parametra su database ili file (to se može mijenjati i programski). Ako se XML sprema u bazu, tablica za spremanje zove se PS_TXN, a nalazi se u shemi (baze) koja je specificirana parametrom jbo.server.internal_connection. Kod pasivacije, koristi se sekvenca na bazi PS_TXN_SEQ za kreiranje ID-a retka te tablice. Prethodni redak (ako postoji) iste aplikacijske sesije se briše. Ako se desi neka greška, brisanje se može raditi i ručno, kroz odgovarajući PL/SQL paket BC4J_CLEANUP.

Ponekad je potrebno eksplicitno (programski) pasivirati neke podatke koje standardan ADF proces ne pasivira. Proces pasivacije tih korisnički-definiranih podataka opisan je u Oracle ADF priručnicima [16] i [17] (za ADF 11g i ADF 12c), ali i kao jedan "recept" u "kuharici" [4].

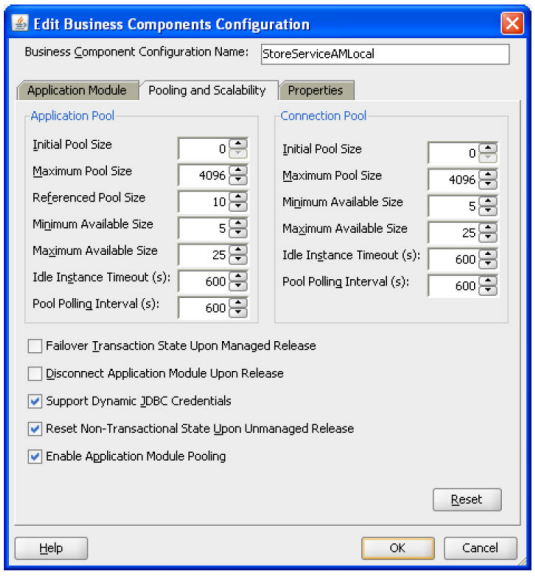
Kako se naglašava u [16] i [17], vrlo je važno testirati aplikaciju tako da se parametar jbo.ampool.doampooling (privremeno) postavi na false (default je true). Naime, kada imamo malo korisnika (npr. kod testiranja) može se desiti da nam se uopće neće desiti pasivacija, pa će nam bez problema proći situacije kada u ne-zadnjem HTTP request-response paru koristimo globalne varijable iz paketa na bazi, ili radimo postiranje na bazu, ili pozivamo procedure na bazi koje rade DML. Budući da nije došlo do pasivacije, a AM instanca standardno ima vezanu konekciju (pa onda i sesiju) na bazu, aplikacija nam se ponaša kao prava statefull aplikacija. Međutim, čim prvi put (najčešće je to u produkciji) dođe do pasivacije, vidjet ćemo da se u takvim slučajevima izgube vrijednosti pakirane varijable ili DML izmjene na bazi, zato jer nova AM instanca (nakon aktivacije), u pravilu dobije novu konekciju (pa onda i sesiju) na bazi. Naravno, pasivacija se dešava samo kod Managed release levela (default), a kod Reserved ne dolazi do pasivacije, niti do tih (mogućih) problema. Zato je poželjno u takvim slučajevima odabrati Reserved release level, ali samo za odabrane AM, a može se programski odrediti da AM bude Reserved samo određeno vrijeme, dok je to zaista potrebno.

Napomenimo da postoji literatura, npr. [11], u kojoj se mogu naći i detaljniji prikazi (npr. kao na slici 3.20.) od onih u Oracle priručnicima (koji imaju skoro 2000 stranica) [16] i [17] (za ADF 11g i ADF 12c).



Slika 3.20. Što se zbiva kada (novi) klijent zatraži AM instancu iz AM poola; Izvor [11]

Za Application Module Pools i Connection Pools postoje (brojni) parametri, koji se mogu podešavati i kroz tab Pooling and Scalability na dijaloškom prozoru Edit BC Configuration, kako prikazuje slika 3.21.



Slika 3.21. Application Module Pools i Connection Pools parametri; Izvor [16]

U nastavku se kratko prikazuju neki parametri (četiri od sedam) vezani za ponašanje AM poola:

- Failover Transaction State Upon Managed Release (jbo.dofailover):
  podrazumijevana vrijednost (default) je false; kada se postavi na true, tada se pasivacija dešava svaki put nakon završetka HTTP zahtjeva, tj. svaki put kada se AM instanca vraća u AM pool; default vrijednost ima prednost bržeg rada (do pasivacije, pa onda i aktivacije, dolazi puno rjeđe), a ne-default se koristi za povećanje raspoloživosti (high availability);

- Row-Level Locking Behavior Upon Release (jbo.locking.mode):
  podrazumijevana vrijednost je optimistic, a može biti i pesimistic i optupdate (kako je već bilo govora u potpoglavlju 3.2.);

- Disconnect Application Module Upon Release (jbo.doconnectionpooling):
  kako navodi priručnik [16], ime jbo.doconnectionpooling je zbunjujuće, a pravo značenje parametra je upravo Disconnect Application Module Upon Release; podrazumijevana vrijednost je false, što znači da će AM instanca, nakon vraćanja u AM pool, zadržati konekciju (pa time i sesiju) baze; default vrijednost donosi bolje performanse, a omogućava da se sačuvaju vrijednosti na bazi (globalne varijable u paketu, rezultati DML naredbi) koje je napravio prethodni HTTP request-response par (iste aplikacijske sesije), a koji (HTTP par) nije dao commit (naravno, kod pasivacije se ta veza gubi); kada se postavi jbo.doconnectionpooling = true, automatski se radi pasivacija nakon svakog vraćanja AM instance u AM pool;

- Enable Application Module Pooling (jbo.ampool.doampooling):
  kako je već prije rečeno, podrazumijevana vrijednost je true, a poželjno ju je postaviti na false kod testiranja aplikacije.

U nastavku se kratko prikazuju parametri vezani za veličinu AM poola:

- Initial Pool Size (jbo.ampool.initpoolsize):
  broj AM instanci koje se kreiraju kod inicijalizacije AM poola; podrazumijevana vrijednost je 0; priručnik [16] preporučuje da se stavi 10% više od očekivanog broja AM instanci potrebnih da zadovolje sve korisnike koji rade u isto vrijeme (to ne znači da mora biti 10% više od broja korisnika koji rade u isto vrijeme);

- Maximum Pool Size (jbo.ampool.maxpoolsize):
  maksimalni broj AM instanci koje će AM pool kreirati; podrazumijevana vrijednost je 4096;

- Referenced Pool Size (jbo.recyclethreshold):
  maksimalni broj AM instanci koje će ADF runtime pokušati sačuvati za aplikacijsku sesiju koja ih je prethodno koristila; trebao bi biti manji nego Maximum Pool Size; podrazumijevana vrijednost je 10.

U nastavku se kratko prikazuju parametri vezani za čišćenje (cleanup) AM poola:

- Pool Polling Interval (jbo.ampool.monitorsleepinterval):
  vrijeme (u milisekundama) između dva procesa čišćenja AM poola; podrazumijevana vrijednost je 600 000 (10 minuta);

- Maximum Available Size (jbo.ampool.maxavailablesize):
  idealni maksimalni broj AM instanci u AM poolu; nakon što se kod čišćenja eliminiraju sve AM instance koje su neaktivne duže vrijeme no što je definirano parametrom Idle Instance Timeout (parametar je naveden u nastavku), proces čišćenja će eliminirati i druge AM instance, dok ne dođe do tog broja; podrazumijevana vrijednost je 25;

- Minimum Available Size (jbo.ampool.minavailablesize):
  nakon što se kod čišćenja eliminiraju sve AM instance koje su neaktivne duže vrijeme no što je definirano parametrom Idle Instance Timeout, proces čišćenja neće eliminirati dodatne AM instance ispod tog broja; podrazumijevana vrijednost je 5;

- Idle Instance Timeout (jbo.ampool.maxinactiveage):
  vrijeme (u milisekundama) koliko AM instanca može biti neaktivna; kad se premaši to vrijeme, instanca je spremna za eliminiranje iz AM poola; podrazumijevana vrijednost je 600 000 (10 minuta);

- Maximum Instance Time to Live (jbo.ampool.timetolive):
  vrijeme (u milisekundama) nakon kojeg će se nekorištena AM instanca odrediti kao kandidat za eliminiranje iz AM poola, neovisno da li će se time broj AM instanci spustiti ispod Minimum Available Size; podrazumijevana vrijednost je 3 600 000 (1 sat).

U nastavku se kratko prikazuju parametri vezani za connection pool:

- Initial Pool Size (jbo.initpoolsize):
  broj JDBC konekcija koje se kreiraju kod inicijalizacije connection poola; podrazumijevana vrijednost je 0;

- Maximum Pool Size (jbo.maxpoolsize):
  maksimalni broj JDBC konekcija koje će connection pool kreirati; podrazumijevana vrijednost je 4096;

- Pool Polling Interval (jbo.poolmonitorsleepinterval):
  vrijeme (u milisekundama) između dva procesa čišćenja connection poola; podrazumijevana vrijednost je 600 000 (10 minuta);

- Maximum Available Size (jbo.poolmaxavailablesize)
  idealni maksimalni broj JDBC konekcija u connection poolu; nakon što se kod čišćenja eliminiraju sve JDBC konekcije koje su neaktivne duže vrijeme no što je definirano parametrom Idle Instance Timeout (parametar je naveden u nastavku), proces čišćenja će eliminirati i druge JDBC konekcije, dok ne dođe do tog broja; podrazumijevana vrijednost je 25;

- Minimum Available Size (jbo.poolminavailablesize)
  nakon što se kod čišćenja eliminiraju sve JDBC konekcije koje su neaktivne duže vrijeme no što je definirano parametrom Idle Instance Timeout, proces čišćenja neće eliminirati dodatne JDBC konekcije ispod tog broja; podrazumijevana vrijednost je 5;

- Idle Instance Timeout (jbo.poolmaxinactiveage):
  vrijeme (u milisekundama) nakon kojeg neaktivna JDBC konekcija postaje kandidat za eliminiranje iz connection poola; podrazumijevana vrijednost je 600 000 (10 minuta).

3.7. Task Flow i transakcije

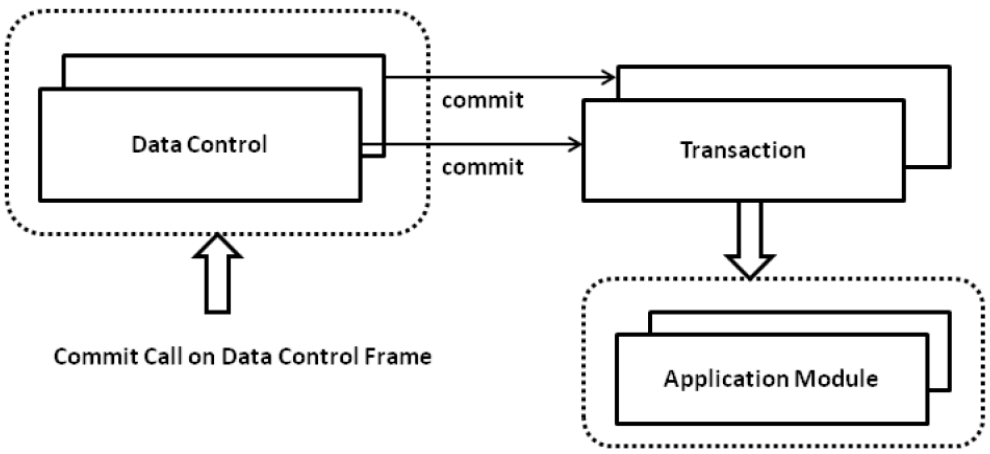
Jedan od najvažnijih dodataka u JDeveloperu / ADF-u 11g, u odnosu na JDeveloper / ADF 10g, bila je pojava Task Flows (ADF TF, ili samo TF), koji spadaju u ADF Controller sloj. TF omogućavaju izradu web aplikacija koje su modularnije nego prije, a UI dijelovi aplikacije mogu se višestruko koristiti puno lakše nego prije. Postoje dvije vrste TF: unbounded TF i bounded TF. TF se sastoje od niza aktivnosti, koje nisu samo aktivnosti prikaza stranice (View Activity), nego i sljedeće aktivnosti: Method, Control Flow Router (usmjerivač), Task Flow Call (poziv drugog TF), Parent Action, Task Flow Return (zadnje dvije aktivnosti ima samo bounded TF).

Unbounded TF može imati više ulaznih i više izlaznih točaka. Često se koristi za definiranje menija. Bounded TF ima samo jednu ulaznu točku, a može imati više izlaznih točaka. Bounded TF može kod ulaza primati parametre od pozivatelja, te vraćati povratne vrijednosti (return values) kod izlaska. Može se temeljiti na kompletnim JSF stranicama ili fragmentima stranice (page fragments). Sa stanovišta transakcija, najvažnije je što za bounded TF postoji razrađen sustav deklarativnog ili programskog upravljanja transakcijama (na Controller sloju).

Kada se TF koriste zajedno s ADF BC donjim slojem (kako se najčešće radi), tada BC sloj, konkretno root AM, posjeduje konekciju na bazu, transakciju, te podatke (kolekcije) i metode – tj. posjeduje (jednom riječju) servise. Te servise BC sloj predaje View i Controller slojevima preko apstrakcija zvanih Data Control. ADF Controller i TF (kao dio Controllera) mogu više data controla grupirati zajedno, omogućavajući time da se nad tom grupom data controla napravi jedan "aplikacijski" commit ili rollback (napomena: to nije SQL COMMIT / ROLLBACK naredba na bazi, već nešto slično kao COMMIT_FORM / CLEAR_FORM built-in naredba u Formsima; COMMIT / ROLLBACK naredba na bazi samo je mali dio tog procesiranja). Iako BC servisi i dalje posjeduju transakciju i izvršavaju svoj vlastiti djelomični commit ili rollback, ADF Controller definira granice ukupne transakcije (početak i kraj).

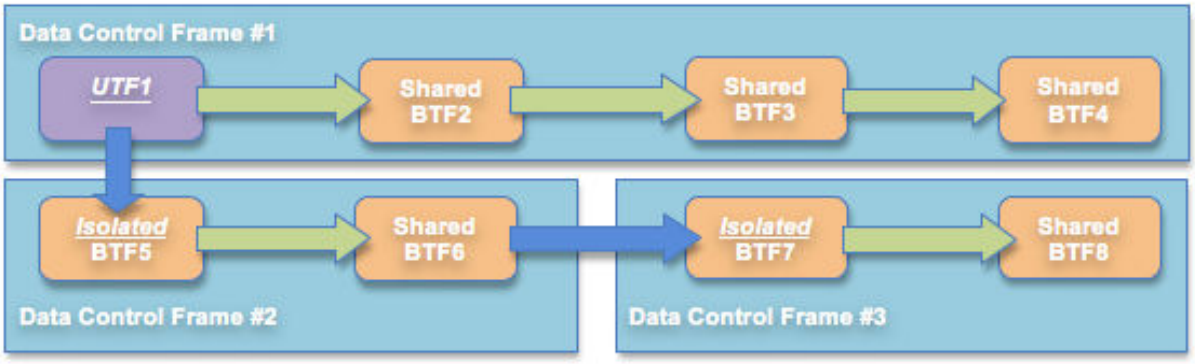
Svaki TF može imati svoj vlastiti tzv. Data Control Frame. Unbounded TF uvijek ima svoj vlastiti data control frame. Bounded TF može imati svoj vlastiti data control frame, ili ga dijeliti s TF-pozivateljem, a to se definira na pozvanom bounded TF (a ne na TF-pozivatelju), kroz stranicu Overview -> granu Behavior -> dio Transaction -> Share data controls with calling task flow. Ako se odabere, onda se u odgovarajuću XML datoteku zapisuje vrijednost Shared za svojstvo data-control-scope, a inače se zapisuje vrijednost Isolated.

Kada bounded TF na izlazu izvršava TF return aktivnost, provjerava se da li je taj TF konfiguriran za commit tekuće transakcije (to se određuje drugim svojstvom, o čemu će biti govora kasnije). Ako jeste, ADF runtime traži njegov data control frame – bilo njegov vlastiti, bilo onaj koji dijeli s (nekim) TF-pozivateljem. Data control frame tada delegira commit poziv rukovatelju transakcije (transaction handler instance) na daljnje procesiranje. Rukovatelj transakcije iterira kroz sve data controle koje postoje u data control frameu i poziva naredbu commitTransaction na svakom vršnom (root) data controlu. Data controle dalje delegiraju commit poziv transakcijskim objektima (transaction object) koji su povezani za AM. Napomena: ako TF-dijete participira u transakciji koju je startao pozivatelj, ili je AM ugniježđen u drugi AM, oni dijele zajednički transakcijski objekt. Commit poziv, koji je bio delegiran transakcijskom objektu, sada će napraviti commit svih promjena u svim AM-ovima koji su vezani za taj transakcijski objekt. Navedena zbivanja prikazuje slika 3.22.



Slika 3.22. Zbivanja kod "aplikacijskog" commita u bounded task flowu; Izvor [11]

Slika 3.23. prikazuje jedan primjer koji objašnjava kada se kreiraju, odnosno kada se ne kreiraju data control frameovi. Prvo unbounded TF UTF1 kreira novi data control frame #1, koji može dijeliti s drugim TF-ovima koje poziva, a koji imaju definirano data-control-scope = shared. Na slici su to bounded TF BTF2, BTF3 i BTF4. Za razliku od njih, BTF5 ima definirano data-control-scope = isolated, tj. kreira svoj vlastiti data control frame #2. Njega dijeli BTF6, dok BTF7 opet kreira svoj vlastiti data control frame #3, kojega dijeli i BTF8.



Slika 3.23. Kada se (ne) kreiraju data control frameovi - primjer; Izvor [9]

Osim navedenog svojstva Share data controls with calling task flow (kako piše na ekranu), odnosno data-control-scope (kako piše u XML datoteci), bounded TF ima još jedno važno svojstvo. Isto kao prethodno svojstvo, može se odabrati kroz stranicu Overview -> granu Behavior -> dio Transaction, ili kroz XML svojstvo Transaction. To svojstvo ima četiri vrijednosti, koje se malo drugačije zovu na ekranu i u XML datoteci. Vrijednosti na ekranu bolje objašnjavaju o čemu je riječ (nego one u XML datoteci):

- "Always Begin New Transaction" vrijednost određuje da BTF starta novu transakciju. Uobičajeno se ova vrijednost koristi zajedno sa isolated data control scope. Naime, ako se "Always Begin New Transaction" koristi sa shared data control scope, tada se data controle prethodnog TF koriste i u tekućem TF, i tada se, ako prethodni TF (odnosno njegove data controle) ima otvorenu transakciju (tj. isTransactionDirty() == true), kod izvođenja javlja greška "ADFC-00020 + Task flow '<name>' requires a new transaction, but a transaction is already open on the frame". BTF koji ima "Always Begin New Transaction" mora na kraju pozivati TF flow return activity, koja poziva commit() ili rollback(). Ti pozivi commit() ili rollback() operacija nad data control frame, zapravo pozivaju commit ili rollback na svim data controlama koje su povezane za taj frame. Drugačije je ponašanje kada se commit ili rollback operacije pozivaju iz Data Control Palette – one rade commit ili rollback samo na određenoj data controli, a ne nad svim data controlama koje su povezane za frame.

- "Always Use Existing Transaction" vrijednost određuje da pozvani BTF dijeli transakciju sa pozivajućim BTF. Ako se kod izvođenja desi da pozivajući BTF nije imao transakciju, pozvani BTF javlja grešku "ADFC-00006: Existing transaction is required when calling task flow <task flow name>". Inače, BTF sa ovom opcijom ne može se tokom dizajna vezati za isolated data control scope. Također, takav BTF ne može tokom dizajna kod izlaska raditi commit ili rollback (JDeveloper nas sprečava u tome). A ako se commit ili rollback i pokrenu programski, tokom izvođenja će se te operacije jednostavno zanemariti (neće se desiti greška, ali se te operacije neće izvršiti).

- "Use Existing Transaction if Possible" je najfleksibilnija od svih opcija. Ona je kombinacija "Always Begin New Transaction" i "Always Use Existing Transaction" opcija. Ako se "Use Existing Transaction if Possible" koristi sa isolated data control scope, ponaša se na isti način kao "Always Begin New Transaction" sa isolated data control scope. Ako se "Use Existing Transaction if Possible" koristi sa shared data control scope, njeno ponašanje ovisi o tome da li je BTF-pozivatelj otvorio transakciju. Ako je BTF-pozivatelj otvorio transakciju, onda je ponašanje isto kao "Always Use Existing Transaction" sa shared data control scope. Ako BTF-pozivatelj nije otvorio transakciju, onda je ponašanje kao "Always Begin New Transaction" sa shared data controls scope. Ovu opciju odabiremo kad nismo sigurni kako će se BTF koristiti.

- "<No Controller Transaction>" je najkompleksnija za razumijevanje.

"<No Controller Transaction>" opcija ima sljedeće osobine:

- ne starta novu transakciju;
- ne provjerava i ne traži da je transakcija otvorena na data control frame;
- na kraju TF ne poziva finalizaciju za data control frame transakciju, tj. ne poziva DataControlFrame commit() ili rollback() operacije.

No, prije spominjana pravila ipak vrijede i kod ove opcije. Ako se "<No Controller Transaction>" opcija kombinira sa isolated data control scope, ipak će se instancirati novi data control frame i nove data controle. Ako se kombinira sa shared data control scope, dijelit će data control frame i data controle iz prethodnog TF. Sa "<No Controller Transaction>" imamo veću slobodu programske (ne-deklarativne) intervencije, ali sa slobodom dolazi i odgovornost za ispravno ponašanje programskog koda.

Kako kaže autor u [9], postoje 64 teoretske kombinacije, jer 4 (opcije za kontrolu transakcije) puta 2 (opcije za dijeljenje data control frame) daje 8 kombinacija, ali onda imamo 8 * 8 kombinacija za pozivajući TF i pozvani TF zajedno. Istina, neke od tih kombinacija ipak nisu teoretski moguće, jer ih sprečava JDeveloper tokom razvoja (npr. "Always Use Existing Transaction" i isolated data control scope).

Zato je u [9] prikazao četiri "čiste" kombinacije (koje će biti kratko prikazane u nastavku):

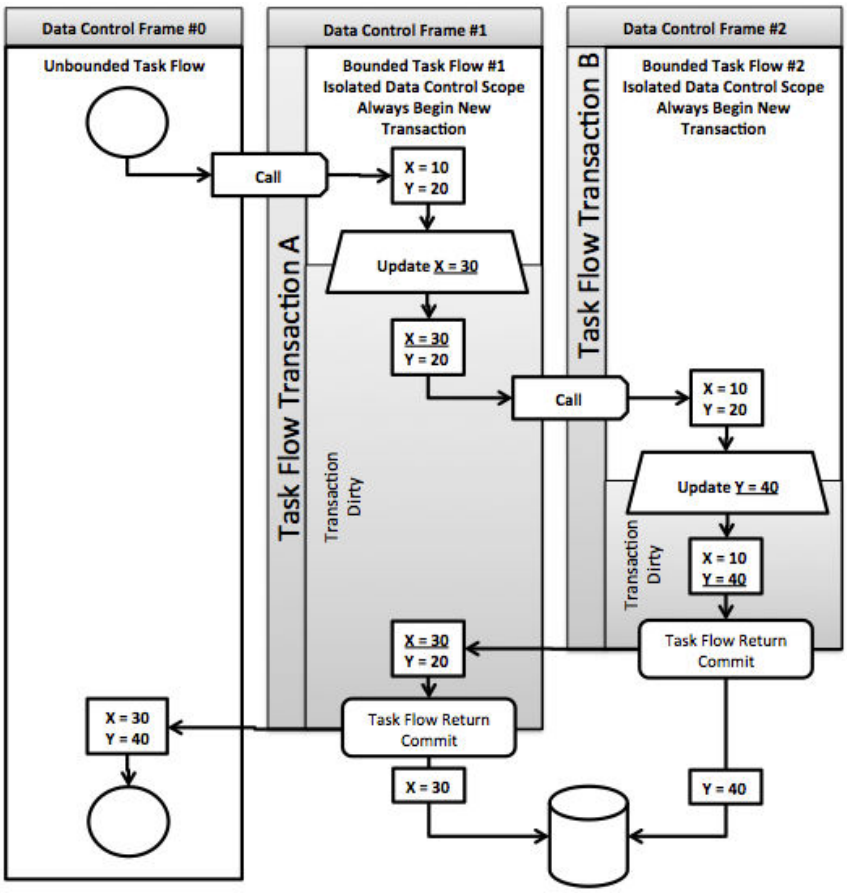
- potpuno odvojene transakcije: dva BTF su potpuno izolirana, tako da svaki za sebe radi commit ili rollback;

- garantirano povezane transakcije: ili u potpunosti dijele transakciju i podatke, ili se javlja greška;

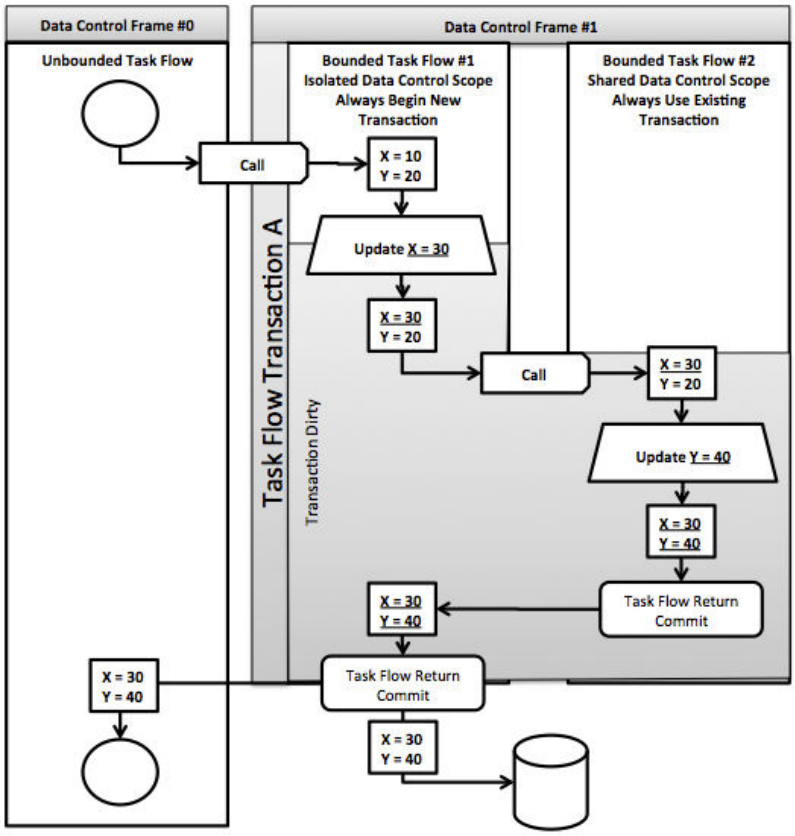
- fleksibilni režim transakcija: pozvani BTF će dijeliti transakciju sa BTF-pozivateljem, ako je moguće, a inače će kreirati vlastitu transakciju;

- "<No Controller Transaction>" opcija: autor kaže da ova opcija, korištena sa prethodne tri opcije, stvara najkompleksniju situaciju, i da bi ju trebalo koristiti sa dužnom pažnjom.

U nastavku su prikazane prva, druga i četvrta varijanta (slike 3.24. - 3.26.). Treća varijanta bila bi kombinacija prve i druge. U sva tri slučaja UTF poziva BTF1, koji nakon toga poziva BTF2.

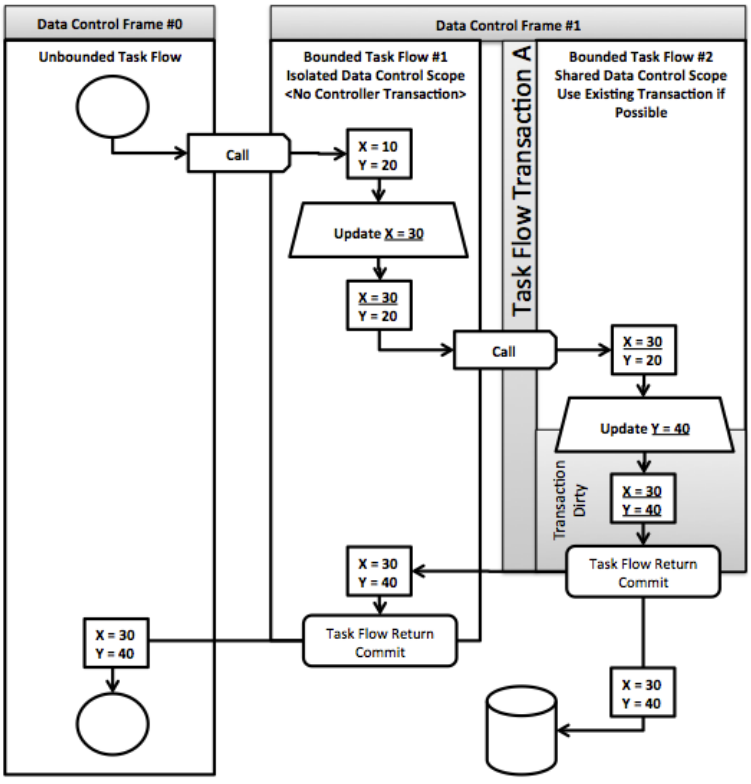


Slika 3.24. Potpuno odvojene transakcije između BTF1 i BTF2; Izvor [9]



Slika 3.25. Garantirano povezane transakcije – BTF2 koristi istu transakciju kao BTF1; Izvor [9]

Prethodne dvije varijante su bile relativno jednostavne za razumijevanje. Sljedeća varijanta (na slici 3.26.) je složenija. Sliči na prethodnu varijantu, ali ovdje BTF1 ima "<No Controller Transaction>" opciju. Iako rezultat izgleda kao u prethodnoj varijanti, treba napomenuti da ovdje commit radi BTF2, a ne BTF1. U slučaju kada bi BTF1 radio neke izmjene nakon poziva BTF2, to bi se moralo programirati korištenjem commit ili rollback operacija iz Data Control Palette, jer BTF1 nema TF return activity.



Slika 3.26. Složenija varijanta – BTF1 ima "<No Controller Transaction>" opciju; Izvor [9]

ZAKLJUČAK

Ispravno rukovanje transakcijama u bazama podataka vrlo je bitno za poslovne aplikacije.

Nažalost, greške u rukovanju transakcijama obično se kod testiranja teže uoče, jer često ovise o spletu događaja, broju korisnika koji istovremeno rade i sl. Najčešće se puno lakše uoče npr. greške u korisničkom sučelju (što ne znači da su te greške nevažne, ili da ih je uvijek lako ispraviti). Greške u transakcijama mogu uzrokovati npr.:

- pogrešne podatke; primjer: derivirani iznos u zaglavlju dokumenta nije izračunat u istoj transakciji u kojoj su mijenjani podaci iz stavaka dokumenta, na temelju kojih se taj iznos računa; zanemarimo pitanje da li (ni)je dobro imati derivirani iznos na razini zaglavlja dokumenta;

- gubljenje dokumenata; primjer: račun je štampan, COMMIT slijedi iza štampanja, ali transakcija se prekinula i u bazi nema dokumenta, a štampan je;

- pojavu "rupa" u rednim brojevima dokumenata; primjer: za punjenje brojeva dokumenata koristi se sekvenca sa baze, što ne garantira da neće biti rupa ...

Kako često naglašava Tom Kyte (između ostalog, i izvrsnoj knjizi [7]), često aplikacijski programeri gledaju na DBMS sustav kao na "crnu kutiju". Nemaju vremena (ili motivacije) za dublje upoznavanje s mogućnostima konkretnog DBMS sustava, drže da su svi DBMS sustavi isti (ili vrlo slični), žele pisati generički kod (neovisan o DBMS-u) itd.

Naravno, postoje slučajevi kada je pisanje generičkog koda (vrlo) opravdano, npr. kada je riječ o softveru koji zaista mora raditi na većem broju različitih DBMS sustava. No, često se potreba za neovisnošću od DBMS sustava postavlja kao zahtjev iako se u stvarnosti radi samo s jednom vrstom DBMS sustava. A, kako Tom Kyte naglašava, različiti DBMS sustavi prilično se razlikuju baš po upravljanju transakcijama, zaključavanju redaka i sl.

Npr. do prije nekoliko godina, Oracle je bio jedan od rijetkih "velikih" DBMS sustava koji je čitateljima podataka dozvoljavao čitanje podataka, bez obzira što ih je neki pisac (DML naredbom) zaključao (u Oracle bazi čitatelji ne zaključavaju retke drugim čitateljima i piscima, a pisci ne zaključavaju retke čitateljima).

Nije onda čudno da su se u drugim DBMS sustavima preporučivale (pre)kratke transakcije, ili to što JDBC sustav ima za parametar AutoCommit podrazumijevanu (default) vrijednost AutoCommit = true (što označava da svaka uspješna SQL naredba na kraju radi implicitni COMMIT), pa se u JDBC literaturi kao jedna od najvažnijih stvari za ispravno rukovanje transakcijama navodi to da na početku programskog koda treba staviti:

```
connection.setAutoCommit(false);
```

Aplikacijski programeri ponekad ne nalaze vremena za upoznavanje problematike rukovanja transakcijama niti unutar alata s kojima rade, npr. Forms i ADF alata, već se pouzdaju u podrazumijevano (default) ponašanje, ponekad i ne znajući kakvo je to ponašanje.

Istina, često se transakcije i u bazi (npr. Oracle), i u alatima (npr. Forms i ADF), mogu koristiti na "standardan" način, i često je tako i najbolje. No u praksi se neminovno javljaju slučajevi kada treba posegnuti za nekim rješenjem koje nije "standardno".

LITERATURA

1. Basham, B., Sierra, K., Bates, B. (2008): Head First Servlets and JSP (2. izdanje), O'Reilly Media

2. Bernstein, P.A., Newcomer, E. (2009): Principles of transaction processing (2. izdanje), Elsevier / Morgan Kaufmann Publishers

3. Date, C.J. (2004): An introduction to Database Systems (8. izdanje), Addison-Wesley

4. Haralabidis, N. (2012): Oracle JDeveloper 11gR2 Cookbook, Packt Publishing

5. Harris, T. (2010): Transactional Memory (2. izdanje), Morgan & Claypool

6. Herlihy, M., Shavit, N. (2012): The Art of Multiprocessor Programming (2. izdanje), Elsevier / Morgan Kaufmann Publishers

7. Kyte, T. (2009): Expert Oracle Database Architecture, Apress

8. Mills, D., Koletzke, P., Roy-Faderman, A. (2010): Oracle JDeveloper 11g Handbook, Oracle Press

9. Muir, C. (2012): Oracle ADF Task Flow Transaction Fundamentals, tekst iz serije ADF Architecture Square, Oracle

10. Nimphius, F., Munsinger, K., (2010): Oracle Fusion Developer Guide, Oracle Press

11. Purushothaman, J., (2012): Oracle ADF Real World Developer's Guide, Packt Publishing

Oracle priručnici za bazu:

12. Oracle Database Advanced Application Developer's Guide 11g Release 2, E17125-03, August 2010

13. Oracle Database Development Guide 12c Release 1, E17620-11, June 2013

Oracle priručnici za Forms:

14. Oracle Forms Services & Oracle Forms Developer 11g Technical Overview, An Oracle White Paper, 2009.

15. Forms Services Deployment Guide 11g Release 2, E24477-03, November 2012

Oracle priručnici za ADF:

16. Fusion Developer's Guide for Oracle Application Development Framework 11g Release 2, E16182-05, March 2013

17. Developing Fusion Web Applications with Oracle Application Development Framework 12c, E23132-01, June 2013