KAPITOLA 4

SQL a PL/SQL

PL/SQL je procedurální rozšíření jazyka SQL (Structured Query Language) v Oracle. SQL se používá na získávání dat z databáze, na manipulaci s daty, na návrat výsledku uživateli a případně na zápis dat do datového zdroje. Jak jsme si řekli v kapitole 1, SQL je okno do databáze a PL/SQL nám pomáhá jeho možnosti zužitkovat.

Tato kapitola se soustřeďuje na využití SQL v PL/SQL a ukazuje některé pokročilé možnosti, které zvyšují výkon aplikace a vylepšují její strukturu. V kapitole budeme mluvit o následujících tématech:

- transakční zpracování
- získávání dat pomocí základních příkazů, získávání informací Oracle Text, regulární výrazy a kurzory
- pokročilé metody výběru dat pomocí ROWID a ROWNUM
- práce s daty v SQL
- vestavěné funkce SQL

Během představování těchto vlastností zkuste přemýšlet o tom, jak byste mohli s jejich pomocí vylepšit funkcionalitu a výkon svých aplikací i jejich stavbu.

Transakční zpracování

Představte si, že nakupujete na Internetu. Vložíte zboží do košíku, napíšete číslo své platební karty a ve chvíli, kdy stisknete tlačítko, kterým se objednávka odesílá, internetový obchod spadne. Odečtou se vám peníze za zboží? Uložila se před pádem jen část informací?

V byznysu jsou to transakce, které zastřešují množství na sobě závislých operací. Ty se musí dokončit všechny, aby transakce byla úspěšná. Jestliže jedna část selže a druhá proběhne v pořádku, celá transakce je neplatná. V Oracle je *transakce* pracovní celek. Může jít o jednoduchý výraz DML či více výrazů DML, jejichž účinky ještě nejsou trvalé. Jestliže transakce doběhne v pořádku, zapíše se celá její práce do databáze. Dojde-li k chybě, celá transakce se zruší či odvolá. I když už transakce dokončila svou práci, změny se v databázi promítnou, až když k tomu transakce dostane pokyn.

Transakce zaručují konzistentnost databáze. Vlastnosti transakce, které popisují, jak by transakce měly v databázi fungovat, popisujeme zkratkou *ACID*. ACID, to je

- **A**tomicity (atomicita). Transakce buď celá uspěje, nebo selže.
- **C**onsistency (konzistentnost). Databáze je vždy v konzistentním stavu. Žádné částečné transakce neexistují.
- **I**solation (izolovanost). Změny, které transakce provede, jsou vidět pouze v relaci, která je provádí. Teprve po jejich promítnutí do databáze je mohou vidět ostatní.
- **D**urability (stálost). Dokončenou transakci nelze vzít zpět.

Vlastnosti *ACID* chrání transakční data před narušením. Zajišťují, že každá relace má úplný pohled na data.

Transakce a zámky

Transakce nejsou omezeny na bloky PL/SQL a vlastně na PL/SQL vůbec. Mohou obsahovat libovolnou změnu DML v databázi. Neexistuje žádný konkrétní výraz, který zahajuje v Oracle transakci. Místo toho začíná transakce vždy, když nějaký příkaz DML zamkne nějaký objekt. Kdykoliv použijete DML, Oracle zamkne měněné objekty až do dokončení transakce. Ukažme si to na jednoduchém příkazu `UPDATE`, který provedeme v tabulce `AUTORI`, a na dotazu do datového slovníku.

```
-- skript naleznete na DVD jako součást souboru LockSession1.sql
-- relace 1
UPDATE autori
SET jmeno = 'Ronald'
WHERE id = 44;

1 řádek aktualizován.
```

Ještě jsme příkaz nepotvrdili, takže transakce čeká, až ji potvrdíme nebo zrušíme. Dotážeme-li se do pohledů `DBA_LOCKS` nebo `V$LOCK`, uvidíme zámky transakce.

```
-- skript naleznete na DVD jako součást souboru LockSession1.sql
-- relace 1
SELECT d.session_id sid, d.lock_type, d.mode_requested,
       d.mode_held, d.blocking_others
FROM dba_locks d, v$session v
WHERE v.username = 'PLSQL'
AND d.session_id = v.sid;
```

Výsledek je následující:

```
SID  LOCK_TYPE    MODE_REQUESTED  MODE_HELD     BLOCKING_OTHERS
---  ----------   --------------  ----------    --------
204  Transaction  None            Exclusive     Not Blocking
204  DML  None    Row-X (SX)      Not Blocking
```

V naší relaci jsou dva zámky. První je zámek `Transaction`, který je v režimu `Exclusive`. V tuto chvíli neblokuje žádné další uživatele a nebrání jim v práci. Transakční zámek brání, aby ostatní relace měnily daný řádek.

Vaše SID může být odlišné, protože je specifické pro danou relaci.

Druhý zámek je na měněném objektu. V našem případě jde o zámek, který má tabulka `AUTORI` na nějakém řádku, jak je vidět z hodnoty ve sloupci `MODE_HELD` – `Row-X` (row exclusive – výhradně pro řádek). Také nikoho neblokuje. Zámek zabraňuje změnám ve struktuře objektu, který zamyká. Chcete-li vidět objekt, který zámek drží, spusťte následující dotaz:

```
-- skript naleznete na DVD jako součást souboru LockSession1.sql
-- relace 1
SELECT dbl.lock_type, dbl.mode_held, dbl.blocking_others,
       dbo.object_name object_locked, dbo.object_type
FROM dba_locks dbl, v$session v, dba_objects dbo
WHERE v.username = 'PLSQL'
AND dbl.session_id = v.sid
AND dbo.object_id = dbl.lock_id1;
```

Výsledek tohoto dotazu obsahuje název a typ drženého objektu, spolu s informací o zámku, kterou jsme již viděli.

```
LOCK_TYPE  MODE_HELD   BLOCKING_OTHERS  OBJECT_LOCKED  OBJECT_TYPE
---------  ---------   --------------   -------------  -----------
DML        Row-X (SX)  Not Blocking     AUTHORS        TABLE
```

Tyto zámky platí po dobu transakce. Během této doby zabraňují, aby jiný uživatel změnil v jiné relaci tentýž záznam.

Nyní spustíme jinou relaci SQL*Plus a první necháme otevřenou. Můžeme se dotázat do tabulky `AUTORI` a zjistit, jestli je změna vidět v jiných relacích:

```
-- skript naleznete na DVD jako součást souboru LockSession2.sql
-- relace 2
SELECT jmeno
FROM autori
WHERE id = 44;
```

Hodnota jména se nezměnila.

```
JMENO
------
Ron
```

Jestliže se pokusíme spustit tentýž aktualizační příkaz SQL, relace se zastaví (přestane odpovídat).

```
-- skript naleznete na DVD jako součást souboru LockSession2.sql
-- relace 2
UPDATE autori
SET jmeno = 'Ronald'
WHERE id = 44;
```

Nedojde k chybě, nevypíše se žádná zpráva, která by nám řekla, co se děje. Můžeme spustit následující dotaz a zjistit, co se děje:

```
-- skript naleznete na DVD jako součást souboru LockSession1.sql
-- relace 1
SELECT d.session_id sid, d.lock_type, d.mode_requested,
       d.mode_held, d.blocking_others
FROM dba_locks d, v$session v
WHERE v.username = 'PLSQL'
AND d.session_id = v.sid;
```

Spusťte tento dotaz z první relace, protože druhá neodpovídá.

Nyní vidíme, že zámek, který předtím nikoho neblokoval, nyní už blokuje.

```
SID   LOCK_TYPE     MODE_REQUESTED  MODE_HELD    BLOCKING_OTHERS
----- --------     --------------  ---------    ---------------
202   Transaction  Exclusive       None         Not Blocking
202   DML          None            Row-X (SX)   Not Blocking
204   Transaction  None            Exclusive    Blocking
204   DML          None            Row-X (SX)   Not Blocking
```

První dva záznamy jsou nové zámky. První z nich, transakční zámek, nemůže požadovaný objekt zamknout, a proto čeká ve frontě. Vidíme, že si vyžádal exkluzivní zámek, ale v tuto chvíli žádný nedrží. Jinou změnu vidíte ve třetím záznamu. Původně nikoho neblokoval, nyní již blokuje. Druhá transakce nemůže doběhnout, dokud neskončí první.

COMMIT

Hlavním cílem příkazu COMMIT je zapsat všechny informace ze zásobníku opakovacího protokolu (tzv. redo log) do aktuálních opakovacích protokolů. Příkaz nevynutí zápis dat z vyrovnávací paměti zásobníku databáze do fyzických datových souborů, – to je časté nedorozumění. Další operace, které probíhají během příkazu COMMIT, jsou generování a zápis SCN do aktuálních opakovacích protokolů.

Jestliže nevíte, co je to SCN, nebo vám není jasné, jak fungují opakovací protokoly, vyrovnávací paměť databáze, soubory opakovacích protokolů a datové soubory, a chtěli byste to vědět, podívejte se do průvodce Oracle Concepts, který naleznete na webu OTN (*http://otn.oracle.com*).

Syntaxe příkazu COMMIT je

COMMIT [*WORK*]

Volba *WORK* je nepovinná a obyčejně se nepoužívá.

Pokračujme v našem posledním příkladě a použijme následující příkaz pro provedení transakce v relaci 204:

```
-- relace 1
commit;
```

Jakmile provedeme transakci, můžeme se opětovně podívat do pohledu `DBA_LOCKS`. Zde je vidět, že zámky relace 204 jsou pryč a relace 202 nyní může mít zámek `Exclusive`.

```
SID   LOCK_TYPE     MODE_REQUESTED  MODE_HELD    BLOCKING_OTHERS
----- --------      --------        --------     --------
202   Transaction   None            Exclusive    Not Blocking
202   DML           None            Row-X (SX)   Not Blocking
```

První transakce je nyní hotová a druhá relace může dokončit požadavek.

ROLLBACK

Provedení transakce dělá změny trvalými. Chceme-li změny odvolat, použijeme příkaz ROLLBACK. Syntaxe je prostá:

ROLLBACK [*WORK*]

WORK je nepovinný příkaz a kromě čitelnosti nemá pro příkaz význam. Následující příklad ilustruje odvolání transakce:

```
-- relace 2
rollback;
```

Transakce končí stejně jako u COMMIT, ale data se nezmění. Opětovným dotazem do pohledu `DBA_LOCKS` zjistíme, že zámky zmizely.

```
-- skript naleznete na DVD jako součást souboru LockSession1.sql
-- relace 2
SELECT d.session_id sid, d.lock_type, d.mode_requested,
       d.mode_held, d.blocking_others
FROM dba_locks d, v$session v
WHERE v.username = 'PLSQL'
AND d.session_id = v.sid;

nebyly vybrány žádné řádky
```

Oba zámky byly uvolněny.

Částečné odvolání transakce pomocí bodů uložení. Jak jsme si již ukázali, odvolání ruší celou transakci. Použijeme-li příkaz *savepoint (bod uložení)*, můžeme odvolat transakci částečně. Bod uložení je návěstí pro odvolání, který říká, že se má práce zrušit jen k určitému bodu v transakci. To, co proběhlo v transakci před

bodem uložení, zůstává nedotčeno. DML po bodě uložení se ruší. Syntaxe bodu uložení je

SAVEPOINT *název*;

Název může být libovolný platný identifikátor (pravidla pro pojmenovávání identifikátorů naleznete v kapitole 3). Odvolání transakce do určitého bodu uložení probíhá takto:

ROLLBACK [*WORK*] TO SAVEPOINT *název*;

Následující příklad provádí dva příkazy `INSERT` a jeden příkaz `UPDATE`. Jsou v něm dva body uložení:

```
-- skript naleznete na DVD jako součást souboru SavePoint.sql
BEGIN
   INSERT INTO knihy (ISBN, kategorie,
                nazev, pocet_stran,
                cena, COPYRIGHT, AUTOR1)
   VALUES ('12345678', 'Oracle Server',
        'Oracle Information Retrieval with Oracle Text', 440,
        35.99, 2005, 44);

   SAVEPOINT A;

   INSERT INTO katalog (isbn, status, status_datum, mnozstvi)
   VALUES ('12345678', 'OBJEDNÁNO', null, 1100);

   SAVEPOINT B;

   UPDATE katalog
   SET status = 'NA SKLADĚ'
   WHERE isbn = '12345678';

   ROLLBACK TO SAVEPOINT B;

   COMMIT;
END;
/
```

Jestliže jste spouštěli předchozí příklady, budete patrně muset zadat příkaz na provedení svých transakcí, než spustíte tento příklad. Tím uvolníte zámky z předchozích příkladů.

Dopad příkazu `ROLLBACK TO SAVEPOINT` zjistíte následujícím dotazem:

```
-- skript naleznete na DVD jako součást souboru SavePoint.sql
SELECT k.nazev, kt.status
FROM knihy k, katalog kt
WHERE k.isbn = '12345678'
AND k.isbn = kt.isbn;
```

Vypíše se jméno knihy i status.

```
NAZEV                                     STATUS
----------------------------------        -----------------
Oracle Information Retrieval with Oracle Text  OBJEDNÁNO
```

Z dotazu je vidět, že odvolání transakce proběhlo v pořádku. První dva příkazy SQL se provedly. Třetí, který upravoval sloupec status na NA SKLADĚ, byl odvolán.

Autonomní transakce

Autonomní transakce začínají z *nadřízené* neboli *hlavní* transakce, ale pracují samostatně bez transakčního dohledu z dané nadřízené transakce. Jestliže v autonomní či v hlavní transakci použijeme potvrzení nebo odvolání či dojde-li z libovolného důvodu k chybě, druhou transakci to neovlivní.

S oblibou používáme tuto vymoženost na zápis do protokolu aplikačních událostí. Jestliže máme monitorovat aktivitu nezávisle na jejím výsledku a zároveň nemá úspěch či selhání zápisu do protokolu ovlivnit aplikaci samotnou, jsou autonomní transakce dokonalým řešením.

Chcete-li vytvořit autonomní transakci, použijte direktivu (pragma) AUTONOMOUS_TRANSACTION. Direktiva se umístí do deklarační oblasti bloku. Při spuštění kódu uvidí překladač tuto direktivu (instrukci pro překladač) a následně bere blok jako autonomní.

Rádi sdružujeme kód pro monitorování událostí a audit do balíku, ale autonomní transakční kód můžete vytvořit v procedurách, funkcích, spouštích a v objektových typech. Všechny tyto typy programů si podrobně probereme později v knize.

V následujícím příkladu vytvoříme proceduru s direktivou AUTONOMOUS_TRANSACTION.

```
-- skript naleznete na DVD jako součást souboru Autonomous.sql
CREATE OR REPLACE PROCEDURE protokol_zapis(
   i_uzivatel IN VARCHAR2,
   i_datumcas IN TIMESTAMP)
IS
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN

   INSERT INTO protokol (uzivatel, datumcas)
   VALUES (i_uzivatel, i_datumcas);

   commit;

END;
/
```

Procedura bude při volání pracovat nezávisle na volající nadřízené transakci. Nyní vytvoříme proceduru, která vloží záznam do tabulky knih, zavolá právě vytvořenou proceduru PROTOKOL_ZAPIS a provede odvolání.

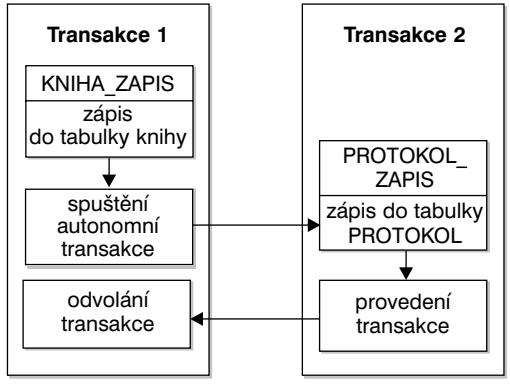
```
-- skript naleznete na DVD jako součást souboru Autonomous.sql
CREATE OR REPLACE PROCEDURE kniha_zapis(
   i_isbn IN KNIHY.ISBN%TYPE,
   i_kategorie IN KNIHY.KATEGORIE%TYPE,
   i_nazev IN KNIHY.NAZEV%TYPE,
   i_pocet_stran IN KNIHY.POCET_STRAN%TYPE,
   i_cena IN KNIHY.CENA%TYPE,
   i_copyright IN KNIHY.COPYRIGHT%TYPE,
   i_autor1 IN KNIHY.AUTOR1%TYPE,
   i_autor2 IN KNIHY.AUTOR1%TYPE,
```

```
   i_autor3 IN KNIHY.AUTOR1%TYPE)
IS
BEGIN

   INSERT INTO knihy (ISBN, KATEGORIE,
                NAZEV, POCET_STRAN,
                CENA, COPYRIGHT, AUTOR1,
                AUTOR2, AUTOR3)
   VALUES (i_isbn, i_kategorie,
        i_nazev, i_pocet_stran,
        i_cena, i_copyright, i_autor1,
        i_autor2, i_autor3);

   -- Voláme proceduru, která je autonomní transakcí
   PROTOKOL_ZAPIS('PLSQL', systimestamp);

   -- Odvolání platí pouze pro nadřízenou transakci
   ROLLBACK;

EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/
```

Kód otestujeme zavoláním procedury `KNIHA_ZAPIS`:

```
-- skript naleznete na DVD jako součást souboru Autonomous.sql
BEGIN
 KNIHA_ZAPIS('12345678',
 'Oracle Server',  'Oracle Information Retrieval with Oracle Text',
 440,  35.99,  2005,  44,  null,  null);
END;
/
```

Procedura `KNIHA_ZAPIS` vložila záznam do tabulky `KNIHY` a hodnota `ISBN` je `12345678`. Dále pomocí volání autonomní transakce ve formě procedury `PROTOKOL_ZAPIS` zapsala událost do tabulky PROTOKOL. Na závěr byl zpět v nadřízené proceduře proveden příkaz `ROLLBACK`. Vše vidíte na obrázku 4.1.



Obrázek 4.1 Autonomní transakce

Chceme-li otestovat, jak autonomní transakce fungovala, podívejme se na záznamy v tabulkách KNIHY a PROTOKOL.

```
-- skript naleznete na DVD jako součást souboru Autonomous.sql
COL uzivatel FORMAT A10
COL datumcas FORMAT A30
SELECT *
FROM protokol;
```

Dostáváme

```
UZIVATEL       DATUMCAS
------------   ---------
PLSQL          30.03.07 11:52:10,916000
```

Záznam, který vkládala autonomní transakce, v tabulce PROTOKOL existuje, tato transakce tedy proběhla. Dále můžeme spustit dotaz do tabulky KNIHY:

```
-- skript naleznete na DVD jako součást souboru Autonomous.sql
SELECT *
FROM knihy
WHERE isbn = '12345678';

nebyly vybrány žádné řádky
```

Dostáváme prázdnou množinu řádků, transakce tedy byla opravdu odvolána.

Nikdy nezapomeňte autonomní transakci ukončit příkazem provedení či odvolání! Zanedbávání příkazů pro řízení transakcí v základních objektech PL/SQL je lajdáctví. Uděláte-li to v objektech, které deklarujete jako autonomní transakce, dojde k chybě. Vytvořme proceduru, která je stejná jako procedura PROTOKOL_ZAPIS, ale vynechejme příkaz provedení či odvolání:

```
-- skript naleznete na DVD jako součást souboru Autonomous.sql
CREATE OR REPLACE PROCEDURE protokol_zapis_chyba(
   i_uzivatel IN VARCHAR2,
   i_datumcas IN TIMESTAMP)
IS
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN

   INSERT INTO protokol (uzivatel, datumcas)
   VALUES (i_uzivatel, i_datumcas);

   -- Chybí COMMIT nebo ROLLBACK

END;
/
```

Při překladu nedojde k *žádné* chybě. Zkusme proceduru spustit a uvidíme, co se bude dít.

```
-- skript naleznete na DVD jako součást souboru Autonomous.sql
EXEC PROTOKOL_ZAPIS_CHYBA('PLSQL', systimestamp)
```

Dostaneme následující chybové hlášení:

```
BEGIN PROTOKOL_ZAPIS_CHYBA('PLSQL', systimestamp); END;

*
ERROR na řádku 1:
ORA-06519: zjištěna a odvolána (rolled back) aktivní autonomní transakce
ORA-06512: na "PLSQL.PROTOKOL_ZAPIS_CHYBA", line 13
ORA-06512: na line 1
```

Předcházejte této chybě patřičným používáním příkazů `COMMIT` a `ROLLBACK`.

Nastavení transakce

I když většinu vlastností transakcí měnit nelze, Oracle přesto nabízí určité možnosti nastavení. V tabulce 4.1 je seznam dostupných příkazů včetně popisu jejich funkce.

Tabulka 4.1 Nastavení transakce příkazem SET TRANSACTION

Příkaz	Popis
SET TRANSACTION READ ONLY	Při tomto nastavení pracuje následující transakce v podstatě na snímku databáze ve chvíli, kdy byl příkaz vydán. Obzvláště se to hodí tehdy, když se provádí v transakci více dotazů a data musí být konzistentní.
SET TRANSACTION READ WRITE	Toto nastavení dává transakci možnost zápisu. Jde o základní nastavení transakce.
SET TRANSACTION ISOLATION LEVEL READ COMMITTED	Stejně jako dříve v příkladu v oddíle „Transakce a zámky“ i zde platí, že když se budeme snažit změnit záznam, který již na sobě má zámek typu DML Row Exclusive, kód bude čekat, dokud se zámky neuvolní.
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE	Nastavení totožné jako READ COMMITTED, pouze ve druhé relaci dojde místo čekání na uvolnění zámků k chybě.
SET TRANSACTION USE ROLLBACK SEGMENT	V Oracle do verze 9*i* používají všechny výrazy DML odvolávací segmenty, aby byla zachována konzistentnost databáze. Musejí být dost velké na to, aby v nich mohla být celá transakce, protože prostor, který se používá v odvolávacím segmentu, není uvolněn, dokud nedojde k provedení či odvolání transakce. Příkaz umožňuje specifikovat, který odvolávací segment se bude používat pro danou transakci. V Oracle Database 9*i* se objevuje Automatic Undo Management. Jestliže jej použijete, není již toto nastavení obecně potřeba.

Příkaz SET TRANSACTION musí být v transakci na prvním místě a ukončuje se středníkem. Jestliže to zkusíte a dojde k chybě, dejte před příkaz SET příkaz COMMIT, abyste měli jistotu, že již v nějaké transakci nejste.

Získávání dat

Možnosti získávání dat jdou od základních příkazů SELECT po vyhledávání vzorů přes regulární výrazy a získávání informací (Information Retrieval – IR) pomocí Oracle Textu. O základním výrazu SELECT si zde sice něco povíme, ale předpokládáme, že již máte s SQL zkušenost. Rychle přejdeme k pokročilým vlastnostem, které by jinak bylo obtížné pochopit.

Oddíl zahájíme příkazem SELECT. Pak prozkoumáme použití prvku `LEVEL`, což je pseudosloupec, s jehož pomocí můžeme získávat hierarchické datové množiny spolu s odpovídající pozicí každého řádku ve vzájemných vztazích. Ukážeme si novou vlastnost v Oracle Database 10*g*, jíž jsou regulární výrazy, které umožňují pokročilé práce se vzory v datech. Na závěr si popíšeme získávání informací (Information Retrieval, IR) s použitím Oracle Textu. Při čtení tohoto oddílu přemýšlejte, jak by mohly tyto různé možnosti získávání dat a informací prospět vaší práci.

Příkaz SELECT

Standard ANSI pro SQL vyžaduje, aby výraz SELECT měl dvě části – klauzule SELECT a FROM. Další klauzule jsou nepovinné. Základní syntaxe příkazu je následující:

SELECT *seznam_vybíraných_položek*
[INTO *seznam_proměnných*]
FROM *seznam_tabulek*
[WHERE *seznam_podmínek*]
[ORDER BY *seznam_sloupců*]

Seznam_vybíraných_položek se může skládat ze sloupců, řetězců, vestavěných funkcí SQL nebo znaku *, který vrací celý záznam. V seznamu také můžete použít aritmetické operace. Dále je možné položky přejmenovat (použít tzv. aliasy), čímž se uživateli zobrazí jiné jméno, než je zadané v seznamu, ale hodnoty zůstanou stejné.

Seznam_proměnných v klauzuli INTO je proměnná nebo sada proměnných, která odpovídá počtu proměnných v *seznamu_vybíraných_položek* a jejich datovým typům. Proměnné je možné deklarovat jako jednoduché datové typy, například `VARCHAR2` nebo `NUMBER`, nebo jako vázaný typ pomocí `%TYPE`. Je také možné proměnnou deklarovat jako celý záznam a ta pak může sama obsáhnout celý výběr `SELECT *`.

Seznam_tabulek může být jedna nebo více tabulek, pohledů nebo vnitřních pohledů (poddotazů v klauzuli FROM). Zadáte-li v *seznamu_vybíraných_položek* názvy sloupců, musí tyto sloupce existovat v objektech, které jsou v *seznamu_tabulek*.

Seznam_podmínek omezuje výsledkovou množinu a umožňuje provázat nebo spojit objekty v *seznamu_tabulek*. Operátory srovnání, které se používají v klauzuli WHERE, si probereme později.

Tento výpis dostupných klauzulí není úplný. Celý seznam naleznete v Oracle Database SQL Reference na webu *http://otn.oracle.com*.

V následujícím příkladě použijeme příkaz SELECT a vložíme název knihy do (INTO) proměnné, kterou pak zobrazíme pomocí vestavěného balíku DBMS_OUTPUT na obrazovku:

```
-- skript naleznete na DVD jako součást souboru BasicSelect.sql
SET SERVEROUTPUT ON
DECLARE
   v_nazev KNIHY.NAZEV%TYPE;
BEGIN

   SELECT nazev
   INTO v_nazev
   FROM knihy
   WHERE isbn = '72230665';

   -- zobrazíme výsledky na obrazovku
   DBMS_OUTPUT.PUT_LINE(v_nazev);

EXCEPTION
   WHEN OTHERS
   THEN
      -- zobrazíme případnou chybu
     DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/
```

Tento anonymní blok nám dá následující výstup:

```
Oracle Database 10g PL/SQL Programming
```

Když vybíráte hodnotu do proměnné, ujistěte se, že se vrátí jedna a pouze jedna hodnota. Jestliže se nevrátí žádná hodnota, dojde k následující výjimce:

```
ORA-01403: nenalezena žádná data
```

Pokud se v příkazu SELECT ... INTO vrátí víc než jeden záznam, dostaneme tuto chybu:

```
ORA-01422: přesné načtení vrací více než požadovaný počet řádek
```

Metody, jak tyto chyby s pomocí předdefinovaných výjimek zachytávat, naleznete v kapitole 7.

Výběr hierarchických dat

Data v předchozím příkladě byla v jediné tabulce a při získávaní požadovaného záznamu nebylo potřeba nic složitého. Ale některá data se ukládají hierarchickým způsobem. Například genealogický strom vašeho rodu je hierarchický. Začíná nahoře v počátečním bodě a větví se dále, každá větev je závislá na předchozí. Takováto data vidíte denně, kupříkladu organizační struktura ve firmě či vztahy manažer/zaměstnanec mají rovněž hierarchickou strukturu.

Jak se dá reprezentovat hierarchická datová množina v dotazu tak, abychom znali každou úroveň a mohli ji zobrazit? S pomocí pseudosloupce s názvem LEVEL můžeme vidět, do kterého místa stromu zapadá každý záznam.

V následujícím příkladě jsme přidali do tabulky knih sloupec PARENT_ISBN. Sloupec odkazuje na knihu, která ji v řadě předchází. Struktura tabulky je tedy tato:

```
DESC knihy
Název            Nezadáno?          Typ
--------------   -----------------  ---------------
ISBN             NOT NULL           VARCHAR2(10)
NADRIZENE_ISBN   VARCHAR2(10)
EDICE            VARCHAR2(20)
KATEGORIE        VARCHAR2(20)
NAZEV            VARCHAR2(100)
POCET_STRAN      NUMBER
CENA             NUMBER
COPYRIGHT        NUMBER(4)
```

První kniha v edici má PARENT_ISBN prázdné (hodnota NULL), protože nemá žádného předchůdce. Následující knihy v edici mají ISBN knihy, která v dané edici vyšla před nimi. Zdrojová data obsahují dvě edice. První edice se týká Oracle PL/SQL a obsahuje 3 tituly. Druhá ediční řada obsahuje pouze jedinou knihu. Záznamy jsou v tabulce řazeny náhodně a demonstrují tak, že pořadí dat nehraje roli.

V našem příkladě chceme vybrat všechny knihy v tabulce a zobrazit jejich pozici v hierarchii. Všimněte si, že tabulka nemá sloupec, v němž by bylo pořadí vydání, datum vydání či pozice ve stromě. Tuto pozici chceme určit pouze na základě vztahů nadřízená/podřízená položka, které jsou ve sloupci NADRIZENE_ISBN. V tomto příkladě používáme pseudosloupec LEVEL spolu s klauzulemi START WITH a CONNECT TO PRIOR:

```
-- skript naleznete na DVD jako součást souboru Level.sql
SET SERVEROUTPUT ON
DECLARE
   v_uroven PLS_INTEGER;
   v_nazev KNIHY.NAZEV%TYPE;

   -- kurzor použijeme na odkaz na data, která chceme v dotazu použít
   CURSOR cur_strom
   IS
      SELECT isbn, nazev, edice
      FROM knihy;
BEGIN

-- smyčka přes kurzor, postupně po jednotlivých záznamech
FOR l IN cur_strom
LOOP

   -- získáme úroveň každého záznamu relativně ve stromě
   -- a uložíme ji do proměnné v_uroven
   SELECT max(LEVEL)
   INTO v_uroven
   FROM knihy
   START WITH isbn = l.isbn
   CONNECT BY PRIOR nadrizene_isbn = isbn;

   -- zobrazíme název a úroveň na obrazovku
   DBMS_OUTPUT.PUT_LINE(l.nazev||' je '
                    ||v_uroven||'. kniha v edici '||l.edice||'.');
```

```
-- konec smyčky přes kurzor, když již neexistují další záznamy
END LOOP;


EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/
```

Soustřeďte se v tomto oddíle na **tučně** psaný text, který ukazuje použití funkce LEVEL. Tentýž příklad použijeme na předvedení kurzorů a kurzorových smyček později v této kapitole.

Ze spuštění tohoto anonymního bloku nám vyplynou následující výsledky:

```
Oracle9i PL/SQL Programming je 2. kniha v edici Oracle PL/SQL.
Oracle8i Advanced PL/SQL Programming je 1. kniha v edici Oracle PL/SQL.
Oracle Database 10g PL/SQL Programming je 3. kniha v edici Oracle PL/SQL.
Oracle E-Business Suite Financials Handbook je 1. kniha v edici Oracle
   Ebusiness.
```

Úroveň je vytištěna tučně, protože jde o správné umístění každého titulu v hierarchii.

Jestliže potřebujete zadat úroveň napevno, přidejte jednoduše do tabulky sloupec pozice/úroveň. Předchozí ukázkový blok můžeme změnit tak, že bude obsahovat příkaz, který vloží nebo aktualizuje záznamy tak, aby obsahovaly správnou úroveň. To umožňuje udržovat při změnách v datech aktuální napevno zadané hodnoty pro každou pozici záznamu v hierarchii. Kompletní příklad naleznete v souboru `LevelUpdate.sql`.

Vyhledávání vzorů

Jakýkoliv dotaz SELECT, jehož klauzule WHERE srovnává textový sloupec s řetězcem, provádí *vyhledávání vzorů*. Může jít o přesné srovnání, když srovnáváme pomocí rovnosti, nebo pouze o část řetězce, když zadáme operátor `LIKE`. Vyhledávání vzorů se od získávání dat liší v tom, že nelze interpretovat smysl či závažnost dat. Buď vzor v textu nalezneme, nebo ne.

LIKE

Operátor `LIKE` vyhledávání vzorů značně ulehčuje, povoluje totiž neznámé znaky (použijeme znak podtržítka, _) a neúplné řetězce (použijeme zástupný znak %). Operátor `LIKE` se používá v klauzuli WHERE. Jako vstup slouží libovolný řetězec či neúplný řetězec. Operátor se snaží najít tento vzor v prohledávaných datech. Vyhledávání vzorů pomocí `LIKE` je vhodné pro krátké řetězce – jména, města, země atd.

Vytvořme proceduru na prohledávání tabulky `AUTORI`:

```
-- skript naleznete na DVD jako součást souboru Level.sql
CREATE OR REPLACE PROCEDURE autori_vyber (
   i_prijmeni IN AUTORI.PRIJMENI%TYPE,
   cv_autor IN OUT SYS_REFCURSOR)
```

```
IS
   v_prijmeni AUTORI.PRIJMENI%TYPE;
BEGIN

   /* Na obě strany řetězce přidáme zástupný znak
      a převedeme jej na velká písmena*/
   v_prijmeni := '%'||UPPER(i_prijmeni)||'%';

   OPEN cv_autor FOR
   SELECT id, jmeno, prijmeni
   FROM autori
   WHERE UPPER(prijmeni) LIKE v_prijmeni;

EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/
```

Procedura má jako vstupní parametr řetězec a snaží se nalézt shodu v příjmení v tabulce `AUTORI`. Otestujeme ji vložením neúplného řetězce příjmení:

```
-- skript naleznete na DVD jako součást souboru Level.sql
COL jmeno FORMAT A20
COL prijmeni FORMAT A20

VARIABLE x REFCURSOR
EXEC autori_vyber('rin', :x)

print x
```

Výsledek je:

```
ID     JMENO    PRIJMENI
------ ------   -------------
28     Sumit    Sarin
54     Cheryl   Riniker
```

Řetězec se vyskytuje na různých místech v příjmení, ale v obou případech souhlasí s řetězcem '`rin`', který jsme vložili do procedury.

Regulární výrazy

V Oracle Database 10*g* se objevuje nová vlastnost vyhledávání vzorů, která se nazývá regulární výrazy (Regular Expressions). Jestliže pracujete na systémech Unix nebo Perl, asi již regulární výrazy znáte. Regulární výrazy byly po dlouhou dobu součástí skriptovacích jazyků Perlu a Unixu a nyní i Oracle podporuje standard IEEE POSIX (Portable Operating System Interface – rozhraní operačního systému) ERE (Extended Regular Expressions – rozšířené regulární výrazy).

Základní princip je překvapivě podobný operátoru `LIKE`. Řetězce či znaky se porovnávají s datovým zdrojem a vyhledávají se stejné vzory v textu. To, co povyšuje regulární výrazy nad veškeré možnosti operátoru LIKE, jsou metaznaky a vestavěné funkce. Metaznaky mají v Oracle speciální význam, podobný vyhrazeným slovům,

o nichž jsme mluvili v kapitole 3. Určují, jak Oracle analyzuje řetězce a jak srovnává nebo nahrazuje vzory v textu. Vestavěné funkce jsou:

- REGEXP_LIKE
- REGEXP_INSTR
- REGEXP_REPLACE
- REGEXP_SUBSTR

Úkoly, které tyto funkce plní, se velmi podobají jejich původním funkcím, resp. operátorům – LIKE, INSTR, REPLACE a SUBSTR.

Regulární výrazy pracují s těmito datovými typy:

- CHAR
- VARCHAR2
- NCHAR
- NVARCHAR
- CLOB
- NCLOB

Všechny tyto typy můžete prohledávat pomocí regulárních výrazů.

Metaznaky Tabulka 4.2 obsahuje seznam běžně používaných metaznaků i jejich popis. Jestliže jste nikdy s funkcemi regulárních výrazů nepracovali, může být pro vás setkání s nimi poněkud odstrašující. Pamatujte si, že jsou prostě souhrnem svých samostatných komponent. Rozložte si příkaz po jednotlivých písmenech a metaznacích, vyhledejte význam každého z nich a určitě celý řetězec pochopíte.

Tabulka 4.2 Metaznaky regulárních výrazů (část)

Znak	Význam
*	Odpovídá žádné nebo více hodnotám.
.	Platný znak.
^	Vyhledává vzory na začátku.
[]	Skupiny znaků. Jednotlivé znaky se berou, jako by byly odděleny operátorem OR.
$	Vyhledává vzory na konci.
\	Znak, který se používá, když se metaznak má brát jako skutečné písmeno – literál.
()	Sdružuje řetězce. Často se používá se znakem |.
|	Odděluje výrazy, které jsou součástí skupiny. Má význam logického součtu OR.

Když jsem já (Ron Hardman) začínal pracovat s regulárními výrazy (v Perlu), zjistil jsem, že je mnohem jednodušší vypsat si význam každého příkazu pěkně rukou, ne v kódu. Složitost příkazů se zmenšila, až se mi nakonec dostaly do krve.

V této kapitole se zaměříme na funkci REGEXP_LIKE. Funkce je podobná operátoru `LIKE`. Může se vám zdát, že o něm mluvíme jako o operátoru odlišném od ostatních funkcí, ale to platí hlavně v SQL. V PL/SQL funguje REGEXP_LIKE jako funkce.

Syntaxe REGEXP je

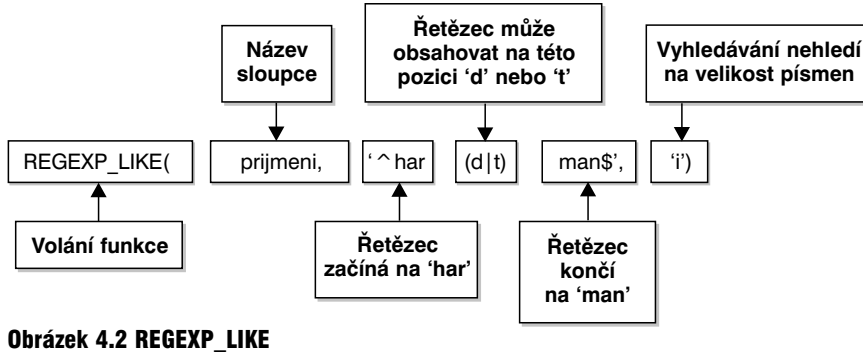
REGEXP_LIKE(*sloupec*, *řetězec* [,*parametr*])

Sloupec je prohledávaný databázový sloupec. *Řetězec* je výraz, v němž jsou literály a metaznaky, které hledáme, a *parametr* je nepovinný parametr.

Následující příklad prohledává sloupec `AUTORI.PRIJMENI`:

```
-- skript naleznete na DVD jako součást souboru RegexpLike.sql
CREATE OR REPLACE PROCEDURE autor_vyber_regexp_like (
   cv_autor IN OUT SYS_REFCURSOR)
IS
BEGIN

   -- hledáme příjmení hardman nebo hartman
  OPEN cv_autor FOR
   SELECT id, jmeno, prijmeni
   FROM autori
   WHERE REGEXP_LIKE(prijmeni, '^har(d|t)man$', 'i');

EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/
```

Vyhledávací řetězec říká, že hledaná hodnota musí souhlasit s řetězcem '`hartman`' nebo '`hardman`', před ani za příjmením nemá být žádný text a vyhledávání má probíhat bez ohledu na to, jestli jsou písmena malá či velká. Podívejte se na obrázek 4.2, na němž je rozbor příkazu REGEXP_LIKE.



Obrázek 4.2 REGEXP_LIKE

Proceduru otestujeme zavoláním:

```
-- skript naleznete na DVD jako součást souboru RegexpLike.sql
COL jmeno FORMAT A20
```

```
COL prijmeni FORMAT A20

VARIABLE x REFCURSOR
EXEC autor_vyber_regexp_like(:x)

print x
```

Výsledek je následující:

```
ID      JMENO       PRIJMENI
------  ----------  ---------------
44      Ron         Hardman
55      Robert      Hartman
```

Vyhledávání bylo úspěšné a splnilo naše očekávání. Dostali jsme do výsledku `Hardmana` i `Hartmana` a při vyhledávání se nehledělo na to, jestli jsou písmena malá, či velká.

Získávání informací

Technologie na *získávání informací (Information Retrieval)* mají za cíl vrátit data, která potřebujeme, a odstranit ta, která se nám nehodí. Následující definice nám pomohou odlišit pojmy „data“ a „informace“:

- **Data:** Znaky, řetězce nebo čísla uložená v databázi. Data vrací jednoduchý příkaz SELECT.
- **Informace:** Data, která jsou filtrovaná podle smyslu a která spolehlivě a přesně odpovídají vyhledávacím kritériím. Jsou ihned použitelná v byznysu. Na proces filtrování dat lze použít určitou úroveň inteligence, která určí „význam“ uživatele nebo dat.

Jádrem řešení získávání informací v Oracle je Oracle Text. Ve verzi 8*i* se nazýval interMedia, ve verzi 8.0 ConText. Oracle Text je plnohodnotným řešením na získávání informací.

Oracle Text

Oracle Text je řešení na indexování dokumentu nebo textu. Může indexovat více než 150 různých textových formátů a poskytuje možnost full-textového vyhledávání. Indexuje také obsah všech typů velkých objektů LOB. Viz kapitola 16, kde naleznete příklad indexování LOB pomocí Oracle Textu. Některé z jeho vyhledávacích možností jsou:

- třídění podle relevance (relevance ranking),
- neurčitá (fuzzy) vyhledávání,
- podobnostní vyhledávání (stemming),
- vyhledávání pomocí zástupných znaků,
- v základním nastavení se nehledí na velikost písmen,
- podpora více jazyků uložených a indexovaných v téže tabulce, možnost vícejazyčného vyhledávání v jediné tabulce,
- a ještě mnohem více!

Oracle Text se hodí pro vývojáře PL/SQL, kteří pracují na aplikacích s datovými sklady, katalogy, databázemi znalostí i na jakýchkoliv aplikacích, v nichž uživatelé vyhledávají. Může indexovat text uložený v databázi, dokumenty uložené v databázi nebo dokumenty v souborovém systému, na něž existují odkazy.

Pro vývojáře je to výhodné, protože se tak dostanou k datům, k nimž by se tradičními metodami získávání dat nedostali. Navíc je vyhledávání v základním nastavení necitlivé na velikost písmen a vždy se používá index.

Oracle Text má čtyři různé typy indexů, které jsou určeny speciálně na různé aplikace. V tabulce 4.3 jsme vám tyto čtyři typy indexů vypsali. Existují od prvního vydání Oracle Database 9*i*. V Oracle Database 8*i* chyběl index `CTXXPATH`.

Tabulka 4.3 Indexy Oracle Textu

Typ indexu	Popis
CONTEXT	Tradiční full-textové vyhledávání. Ideální pro statické katalogy a datové sklady, kde se data příliš nemění, ale požadavky na vyhledávání jsou vysoké.
CTXCAT	Speciálně uzpůsobený index pro elektronické katalogy, který je ideální pro aplikace, jejichž data se často mění. Vyhledávací funkce jsou omezenější než u indexu CONTEXT.
CTXRULE	Index, který se nejlépe hodí pro databáze znalostí či jiné klasifikační aplikace. Je možné použít směrování na dokumenty na základě předdefinovaných pravidel.
CTXXPATH	První full-textový index vytvořený speciálně pro XML. Zvyšuje výkon a vylepšuje vyhledávací vlastnosti v případě prohledávání dokumentů XML.

Ukažme si použití indexu `CONTEXT` při vývoji v PL/SQL. Více informací o Oracle Textu a jeho vlastnostech naleznete na OTN v průvodci *Oracle Text Application Developer's Guide*.

CONTAINS. Při práci s indexem `CONTEXT` musíme v dotazech používat operátor `CONTAINS`. Ten říká Oracle, že pro data existuje index Oracle Textu. V následujícím příkladě máme zdrojovou tabulku `KNIHA_POPIS`. Index Oracle Textu jsme vytvořili na sloupci `POPIS` a v něm také zkusíme vyhledávat.

Syntaxe použití operátoru `CONTAINS` je následující:

```
SELECT [score(návěstí),] seznam_sloupců
[INTO seznam_proměnných]
FROM seznam_tabulek
WHERE CONTAINS (název_sloupce, 'vyhledávací_řetězec'[, návěstí]) > 0;
```

Tučně vytištěné části odlišují tuto syntaxi od běžného dotazu SELECT. Definujme si tyto rozdíly:

- ***score(návěstí):*** se vztahuje na třídění podle relevance. *Návěstí* musí odpovídat návěstí v klauzuli WHERE. Je potřeba pouze tehdy, když třídění podle relevance

požadujete. Zadaná hodnota návěstí nemá žádný dopad na výsledné hodnocení (score).

- Operátor `CONTAINS` vyžaduje ***název_sloupce***, na němž musí být platný index typu `CONTEXT`. Je také potřeba ***vyhledávací_řetězec***, podle nějž se bude vyhledávat. Jestliže zadáte ***návěstí***, musí odpovídat návěstí v klauzuli SELECT. A na závěr je potřeba srovnávací operátor. Výraz > 0 znamená, že výsledek dostanete tehdy, je-li jeho hodnocení větší než nula. Hodnocení se vytváří vždy bez ohledu na to, jestli je zadáte v klauzuli SELECT.

V prvním příkladě si ukážeme dotaz, který využívá operátor `CONTAINS` a nehledí na velikost písmen:

```
-- skript naleznete na DVD jako součást souboru TextIndex.sql
SET SERVEROUTPUT ON
DECLARE
   v_isbn KNIHA_POPIS.ISBN%TYPE;
   v_hodnoceni NUMBER(10);
BEGIN

   SELECT score(1), isbn
   INTO v_hodnoceni, v_isbn
   FROM kniha_popis
   WHERE CONTAINS (popis, '10G or oracle', 1) > 0;

   DBMS_OUTPUT.PUT_LINE('Hodnocení: '||v_hodnoceni||' a ISBN: '||v_isbn);

EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/
```

Řetězec '`10G`' je v databázi s malým *g*, ale v příkladě je velké písmeno. Stejně tak v '`oracle`' je ve vyhledávacím řetězci malé *o*, avšak v databázi je uloženo velké *O*. Dotaz vrací očekávaný výsledek:

```
Hodnocení: 3 a ISBN: 72230665
```

Další příklad předvádí přibližné dotazy, kde jeden z pojmů je v textu poblíž (`NEAR`) druhého:

```
-- skript naleznete na DVD jako součást souboru TextIndex.sql
SET SERVEROUTPUT ON
DECLARE
   v_isbn KNIHA_POPIS.ISBN%TYPE;
   v_hodnoceni NUMBER(10);
BEGIN

   SELECT score(1), isbn
   INTO v_hodnoceni, v_isbn
   FROM kniha_popis
   WHERE CONTAINS (popis, '10g near Oracle', 1) > 0;

   DBMS_OUTPUT.PUT_LINE('Hodnocení: '||v_hodnoceni||' a ISBN: '||v_isbn);

EXCEPTION
```

```
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/
```

I když obsahuje vyhledávací řetězec stejné pojmy jako v předchozím příkladě, hodnocení je výrazně vyšší:

```
Hodnocení: 14 a ISBN: 72230665
```

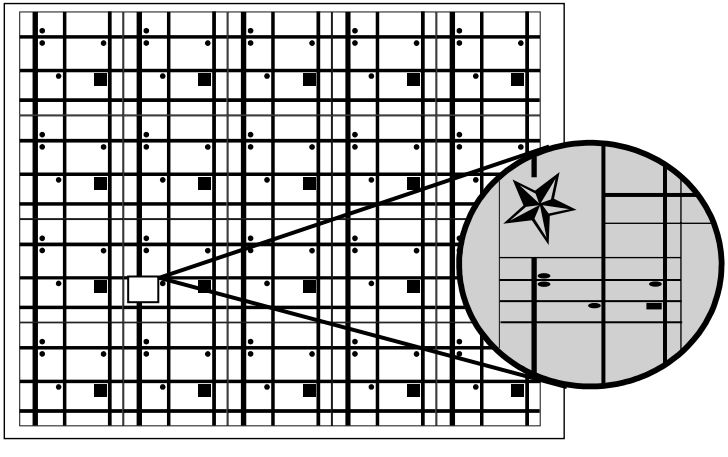
Důvodem je přidání výrazu přibližnosti `NEAR`. `10g` je blízko pojmu `Oracle`, a proto má velký stupeň relevance – vysoké hodnocení.

Zatím jsme se na horu zvanou Oracle Text podívali pouze zdálky. Hrejte si s příklady a hledejte cesty, jak použít tuto mocnou technologii ve svých aplikacích. Možná jste si všimli, že v posledních létech staví Oracle sám více na Oracle Textu – vytváří UltraSearch, IFS, XML DB a Oracle Applications, kde Oracle Text je základní komponenta.

Kurzory

Mám-li řídit, jsem na rozpacích. Vysaďte mě v cizím městě a ztratím se, než objedu jeden blok. Když už musím řídit, odmítám odjet z letiště bez podrobné mapy oblasti.

Představte si, že vás vysadí v neznámém městě a dají vám mapu, na níž je každá ulice. Není to však mapa pouze tohoto města, je na ní celá země, každá dálnice, každá ulička i alej. A vy z toho 99,9999 procent nebudete nikdy potřebovat, protože chcete jen najít cestu do hotelu! Jak dlouho vám to bude trvat, než cestu najdete? Jak efektivní bude vaše hledání? Taková mapa celé země je jako SQL v databázi Oracle a celá záplava informací v ní.



Podrobné mapy měst nabízí mnohem cílenější data. Poskytují detaily, které odpovídají vaší situaci, a informace, které nepotřebujete, v nich nejsou. Vedou vás oblastí

města a jsou efektivním zpracováním informací. Totéž platí pro kurzory. Snižují množství dat, které musí transakce zpracovat, a umožňují přímý přístup k požadované informaci, což zvyšuje efektivitu.

Jak kurzor pracuje

Kurzor je podmnožina dat, kterou definujeme dotazem. Obsah kurzoru se při jeho otevření uloží do paměti a je v ní až do jeho uzavření. Když jsem se začal učit PL/SQL, popisoval můj učitel kurzor jako ukazatel na záznamy v databázi. Jak jsem PL/SQL používal stále více, začal mě tento popis mást. Pro mě není přímým ukazatelem na data v tabulkách nic jiného než index. Kdyby byl kurzor jako index, kde se změny v datech promítají do výsledkové množiny kurzoru již při zpracování, byl by s konzistentností databáze pro čtení konec.

Kurzor není jen ukazatelem na data v tabulce. Ukazuje na oblast v paměti v kontextové oblasti (Process Global Area, PGA), v níž se nachází:

- řádky, které vrátil dotaz,
- počet řádků, které dotaz zpracoval,
- ukazatel na analyzovaný dotaz ve sdíleném poolu (Shared Pool).

Kurzor je tedy ukazatel do paměti, ne přímo do dat. Protože záznamy se do paměti vkládají v okamžiku otevření kurzoru, máme zajištěn konzistentní pohled na data v průběhu celé transakce.

Když dojde k přidání, smazání či úpravě dat až po otevření kurzoru, nová či změněná data se ve výsledkové množině kurzoru neobjeví. Otevření kurzoru se vlastně podobá snímku databáze, tak jak v danou chvíli vypadá. Podívejte se na následující příklad:

```
-- skript naleznete na DVD jako součást souboru ContextArea1.sql
SET SERVEROUTPUT ON
DECLARE

   v_id_radku ROWID;
   v_pocet_radku NUMBER := 0;

   CURSOR autor_cur1
   IS
      SELECT rowid
      FROM autori
      WHERE id > 50;

   CURSOR autor_cur2
   IS
      SELECT rowid
      FROM autori
      WHERE id > 50;

BEGIN

   OPEN autor_cur1;

   DELETE FROM autori
```

```
   WHERE id > 50;

   OPEN autor_cur2;

   -- ověřme kurzor 1
   FETCH autor_cur1 INTO v_id_radku;
   IF autor_cur1%ROWCOUNT > 0
   THEN
      DBMS_OUTPUT.PUT_LINE('Kurzor 1 obsahuje smazané řádky');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Kurzor 1 neobsahuje smazané řádky');
   END IF;

   v_pocet_radku := 0;
   -- ověřme kurzor 2
   FETCH autor_cur2 INTO v_id_radku;
   IF autor_cur2%ROWCOUNT > 0
   THEN
      DBMS_OUTPUT.PUT_LINE('Kurzor 2 obsahuje smazané řádky');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Kurzor 2 neobsahuje smazané řádky');
   END IF;

   CLOSE autor_cur1;
   CLOSE autor_cur2;

   ROLLBACK;

EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/
```

V bloku probíhají tyto kroky:

- Vytváří se dva stejné kurzory. Každý z nich vybírá rowid všech záznamů s `ID` větším než 50.
- Otevře se první kurzor a do paměti se vkládají hodnoty rowid pro autory 51, 52 a 53.
- Dále dochází ke smazání všech záznamů s `ID` větším než 50 z fyzické tabulky.
- Nyní se otevře druhý kurzor a vkládá do paměti všechny záznamy s `ID` větším než 50.
- Pokus o získání záznamu z obou kurzorů. Atribut `%ROWCOUNT` je použit ke zjištění, jestli existují v kurzoru záznamy.
- Zobrazí se zpráva, která říká, zda v kurzorech byly záznamy.

Blok vrací následující výpis:

```
Kurzor 1 obsahuje smazané řádky
Kurzor 2 neobsahuje smazané řádky
```

Z toho vidíme, že kurzor po otevření udržuje obraz dat takový, jaký byl, a nefunguje jako dynamický ukazatel na živá data. Kurzory, které otevřeme po změně dat, tuto

změnu obsahují, i když jsou součástí téhož bloku. Drastičtějším příkladem, který máte na DVD k dispozici, může být situace, kdy otevřeme kurzor, tabulku odstraníme a stále můžeme záznamy v tabulce ve smyčce procházet. Soubor se nazývá `ContextArea2.sql`.

Nyní si ukážeme následující čtyři odlišné typy kurzorů a pohovoříme o atributech kurzorů, smyčkách a parametru OPEN_CURSORS:

- **Explicitní kurzory:** Kurzor se deklaruje pomocí dotazu SELECT v deklarační oblasti libovolného bloku. Vývojář může ovládat v podstatě všechny kurzorové operace.
- **Implicitní kurzory:** Implicitní kurzory jsou řízeny PL/SQL a vytváří se vždy, když spustíte libovolný příkaz DML nebo SELECT ... INTO.
- **Kurzorové proměnné:** Kurzorová proměnná je deklarovaný typ, který je možné spojit s více dotazy v jediném bloku PL/SQL.
- **Kurzorové poddotazy:** Kurzorové poddotazy, kterým se někdy říká vnořené kurzorové výrazy, nabízí možnost vložit kurzory do příkazů SQL.

Explicitní kurzory

Explicitní kurzory nabízí vládu nad průběhem svého zpracování, kterou nemáte u žádného jiného typu kurzoru. Jejich účelem je práce s dotazy SELECT, které vrací více než jeden záznam. Tyto kurzory vám sice dávají větší možnosti než implicitní kurzory, ale vyžadují přídavné kroky. Srovnání použití implicitních a explicitních kurzorů provedeme v oddíle s názvem „Implicitní kurzory".

Chcete-li používat explicitní kurzor, musíte jej deklarovat, otevřít, načíst z něj a uzavřít jej.

Deklarace kurzoru

Kurzor musíte pojmenovat a zadat dotaz SELECT, který se v kurzoru bude používat. Tyto operace se provádí v deklarační oblasti bloku. Syntaxe je:

```
CURSOR název_kurzoru [seznam_parametrů]
[RETURN návratový_typ]
IS dotaz
[FOR UPDATE [OF (seznam_sloupců)][NOWAIT]];
```

Název_kurzoru může být libovolný platný identifikátor, ale kvůli konzistentnosti vám doporučujeme dodržovat názvovou konvenci. *Seznam_parametrů* je nepovinný a může v něm být libovolný platný parametr, který se používá pro provádění dotazu. V nepovinné klauzuli RETURN zadáváte typ návratových dat definovaný jako *návratový_typ*. *Dotazem* může být libovolný dotaz SELECT. Poslední klauzulí je FOR UPDATE, která zamyká záznamy v době, kdy je kurzor otevřen. Záznamy jsou ostatním relacím dostupné pouze pro čtení – READ ONLY. Klauzule FOR UPDATE vám zajistí:

- Když program prochází ve smyčce záznamy z kurzoru, nejsou na nich zámky od ostatních relací a vy je můžete aktualizovat.
- Data souhlasí s tím, co je v kontextové oblasti.

- Zadáte-li NOWAIT, program skončí ihned při otevření, není-li možné získat exkluzivní zámek.

Deklarace kurzor z oddílu „Jak kurzor pracuje“ je:

```
CURSOR autor_cur1
   IS
      SELECT rowid
      FROM autori
      WHERE id > 50;
```

Kdybychom chtěli vytvořit kurzor, který bude přijímat hodnotu ID jako parametr, mohli bychom deklaraci přepsat takto:

```
CURSOR autor_cur1 (i_id IN NUMBER)
   IS
      SELECT rowid
      FROM autori
      WHERE id > i_id;
```

Otevření kurzoru

Kurzory otevíráme v oblastech bloku EXECUTION nebo EXCEPTION. Syntaxe je:

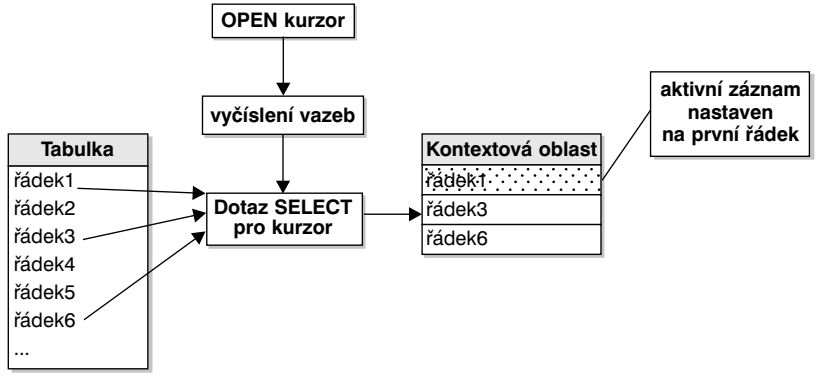
OPEN *název_kurzoru* [(hodnoty_parametrů)];

Příkaz OPEN připravuje kurzor k použití. Při jeho spuštění dojde k analýze dotazu, vyčíslí se vazebné proměnné, záznamy se načtou do kontextové oblasti a nachystá se výsledková množina.

V kurzoru může být v jednom okamžiku pouze jeden aktivní záznam. V okamžiku otevření je aktivním záznamem první řádek, který vrátil dotaz kurzoru. Viz obrázek 4.3, kde je zobrazeno, co se děje při otevření kurzoru.

Následující řádek kódu otevírá kurzor AUTOR_CUR1 bez parametrů:

```
OPEN autor_cur1;
```



Obrázek 4.3 Proces OPEN

A je to! Kdybychom chtěli použít tentýž kurzor s parametrem, jak jsme to viděli v části „Deklarace kurzoru“, příkaz `OPEN` by vypadal takto:

```
OPEN autor_cur1(50);
```

V tomto případě se do `OPEN` vkládá hodnota a vyčíslí se vazebná proměnná. Tato hodnota ani výsledná množina se nezmění, dokud kurzor nezavřete a znovu neotevřete.

Načtení záznamů z kurzoru

Příkazem FETCH vkládáte záznamy z kurzoru do proměnné, abyste je mohli dále používat. Tento příkaz pracuje pouze s aktuálním záznamem a výsledkovou množinu prochází po jednotlivých záznamech. Výjimkou je použití klauzule BULK COLLECT, která může získat všechny záznamy v kurzoru v jednom okamžiku. Více informací o této vlastnosti naleznete v kapitole 6.

Syntaxe příkazu FETCH je:

FETCH *název_kurzoru* INTO *název_proměnné(proměnných)* | *záznam_PL/SQL;*

Název_kurzoru je název otevřeného kurzoru a *název_proměnné(proměnných)* může být názvem jedné nebo více proměnných oddělených čárkou, které odpovídají počtu a typu sloupců ve výsledkové množině. *Záznam_PL/SQL* lze použít jako alternativu k seznamu proměnných v případě, že každý řádek výsledkové množiny obsahuje kompletní záznam.

Načtení výsledkové množiny o jednom sloupci do proměnné může vypadat takto:

```
FETCH autor_cur1 INTO v_id_radku;
```

Načtení dat, když kurzor vrací více sloupců a je potřeba více proměnných, by mohl vypadat asi takto:

```
FETCH autor_cur INTO v_jmeno, v_prijmeni;
```

V tomto případě musí příkaz SELECT pro kurzor `AUTOR_CUR` obsahovat sloupce `jmeno` a `prijmeni`, a to v tomto pořadí. Kurzor nesmí obsahovat žádné další sloupce.

Jestliže je v kurzoru celý záznam, můžete použít záznam PL/SQL jako náhradu jednotlivých proměnných.

```
DECLARE
   v_autor autori%ROWTYPE;
BEGIN
...
FETCH autor_cur INTO v_autor;
...
```

Proměnná `v_autor` obsahuje kompletní záznam. Chcete-li se odkazovat na jednotlivé hodnoty, použijte následující syntaxi:

```
název_proměnné.název_sloupce
```

Například na ID se můžete odkazovat takto:

```
v_autor.id
```

Uzavření kurzoru

Vždy, opravdu vždy své explicitní kurzory zavírejte! Často se říká, že nezavírání kurzorů se rovná černé díře na paměť ukryté v kódu. Miluji toto přirovnání. Pamatujte si, že kontextová oblast (část PGA) je paměť vyhrazená pro kurzor. Dokud kurzor nezavřete, paměť se neuvolní.

Oracle po dokončení posledního bloku prochází *opuštěné* kurzory a automaticky je uzavírá ve chvíli, kdy je poslední blok hotov. Ale *nespoléhejte se* na to!

Kurzor zavřete následovně:

CLOSE *název_kurzoru;*

Název_kurzoru je název otevřeného kurzoru. Jestliže se pokusíte zavřít kurzor, který není právě otevřen, dostane se vám následující chyby.

```
ORA-01001: neplatný kurzor
```

V dalším oddíle vám ukážeme, jak otestovat, jestli je kurzor v danou chvíli otevřen, aby k této chybě nedocházelo.

Atributy kurzoru

Oracle poskytuje pro práci s kurzory šest atributů. Máte je v tabulce 4.4 i s jejich popisy.

Tabulka 4.4 Atributy kurzoru

Název atributu	Popis
%BULK_EXCEPTIONS	Tento atribut se používá pro operace s poli nebo operace Bulk Collect. Jsou v něm informace o chybách, ke kterým došlo v průběhu těchto operací.
% BULK_ROWCOUNT	Také tento atribut je určen pro operace Bulk Collect a je v něm počet řádků, které byly v průběhu operace změněny.
%FOUND	Atribut testuje, zdali příkaz FETCH navrátil záznam. Návratová hodnota je typu Boolean. Je-li rovna TRUE, příkaz FETCH řádek vrátil. V případě, že je rovna FALSE, nedošlo k vrácení řádku.
%ISOPEN	Atribut testuje, je-li kurzor již otevřen. Hodnota TRUE říká, že kurzor je otevřen. FALSE značí, že otevřen není.
%NOTFOUND	Opačný atribut k %FOUND. *Nebyl-li* příkazem FETCH vrácen řádek, je atribut roven TRUE, byl-li vrácen jeden řádek, má atribut hodnotu FALSE.
%ROWCOUNT	V libovolném okamžiku testuje počet řádků načtených z kurzoru a vrací číslo.

Atributy, které souvisí s operací bulk collect, si popíšeme v kapitole 6.

Následující příklad ukazuje použití atributů (někdy hovoříme i o tzv. pseudoatributech) %FOUND, %ISOPEN, %NOTFOUND a %ROWCOUNT:

```
-- skript naleznete na DVD jako součást souboru ExplicitAttribute.sql
SET SERVEROUTPUT ON
DECLARE
   v_jmeno AUTORI.JMENO%TYPE;
   v_prijmeni AUTORI.PRIJMENI%TYPE;
   v_pocet_radku PLS_INTEGER := 0;
   v_pocet_knih PLS_INTEGER := 0;

   CURSOR aut_cur IS
      SELECT a.jmeno, a.prijmeni, count(k.nazev)
      FROM autori a, knihy k
      WHERE a.id = k.autor1
      OR a.id = k.autor2
      OR a.id = k.autor3
      GROUP BY a.jmeno, a.prijmeni
      HAVING count(k.nazev) > 0
      ORDER BY a.prijmeni;

BEGIN
   DBMS_OUTPUT.ENABLE(1000000);

   OPEN aut_cur;
   LOOP
      FETCH aut_cur INTO v_jmeno, v_prijmeni, v_pocet_knih;
      EXIT WHEN aut_cur%NOTFOUND;
      -- Jiná možnost je EXIT WHEN NOT aut_cur%FOUND;

      v_pocet_radku := aut_cur%ROWCOUNT;
      DBMS_OUTPUT.PUT_LINE('Počet dosud zpracovaných řádků:'
|| v_pocet_radku);

      DBMS_OUTPUT.PUT_LINE(v_prijmeni
                           ||', '
                           ||v_jmeno
                           ||' napsal '
                           ||v_pocet_knih
                           ||' knih(u/y).');
   END LOOP;
   CLOSE aut_cur;

   IF aut_cur%ISOPEN = FALSE
   THEN
      DBMS_OUTPUT.PUT_LINE('Kurzor zavřen.');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Kurzor stále otevřen.');
   END IF;

EXCEPTION
   WHEN OTHERS
      THEN
      DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```

Průchod kurzory ve smyčkách

Kurzory se nejčastěji používají ve smyčkách (jak jste si mohli všimnout v posledním příkladě), s jejichž pomocí je možné procházet aktivní sadu záznamů. Jak uvidíte dále, u implicitních kurzorů to není nutné, ale u ostatních druhů kurzorů se to velmi hodí.

Jednoduchá smyčka. Jednoduchá smyčka má syntaxi

```
LOOP ... END LOOP;
```

Uvnitř smyčky se získávají a používají všechny záznamy z aktivní sady záznamů. Nepoužíváte-li postup s Bulk Collect, který si popíšeme v kapitole 6, každá iterace posunuje ukazatel v množině záznamů o jeden záznam dále.

Následuje příklad, který takovou jednoduchou smyčku předvádí:

```
-- skript naleznete na DVD jako součást souboru SimpleLoop.sql
SET SERVEROUTPUT ON
DECLARE
   v_autor AUTORI%ROWTYPE;

   CURSOR aut_cur IS
      SELECT *
      FROM autori;
BEGIN
   OPEN aut_cur;
   LOOP
      FETCH aut_cur INTO v_autor;
      EXIT WHEN aut_cur%NOTFOUND;

      DBMS_OUTPUT.PUT_LINE(v_autor.prijmeni);
   END LOOP;

   CLOSE aut_cur;
END;
/
```

Příkaz EXIT WHEN je ve smyčce nutné použít, protože zajišťuje, že po načtení posledního záznamu se smyčka ukončí.

Smyčka s WHILE. Smyčka s klíčovým slovem WHILE je funkčně podobná jednoduché smyčce, ale metoda provedení se poněkud liší. Přepišme předchozí příklad na použití smyčky typu WHILE:

```
-- skript naleznete na DVD jako součást souboru WhileLoop.sql
SET SERVEROUTPUT ON
DECLARE
   v_autor AUTORI%ROWTYPE;

   CURSOR aut_cur IS
      SELECT *
      FROM autori;
BEGIN
   OPEN aut_cur;

   FETCH aut_cur INTO v_autor;
```

```
   WHILE aut_cur%FOUND LOOP

      DBMS_OUTPUT.PUT_LINE(v_autor.prijmeni);

      FETCH aut_cur INTO v_autor;

   END LOOP;

   CLOSE aut_cur;
END;
/
```

Tento blok sice používá jinou syntaxi, ale výsledky vrací stejné jako příklad `SimpleLoop.sql`. Není také potřeba používat výraz `EXIT WHEN`, protože kontrola `%FOUND` je již v syntaxi smyčky `WHILE`.

Kurzor se smyčkou typu FOR. Kurzorová smyčka typu FOR je jedinečná v tom, že nevyžaduje, abyste uváděli příkazy jako OPEN, FETCH či CLOSE. I když kurzor definujeme jako explicitní, PL/SQL řídí jeho zpracování. Navíc smyčka typu FOR používá proměnnou, která se nedeklaruje v deklarační části bloku. Použijeme tentýž příklad jako u ostatních smyček a přepíšeme jej do podoby smyčky typu FOR.

```
-- skript naleznete na DVD jako součást souboru WhileLoop.sql
SET SERVEROUTPUT ON
DECLARE
   CURSOR aut_cur IS
      SELECT *
      FROM autori;
BEGIN

   FOR v_autor IN aut_cur
   LOOP
      DBMS_OUTPUT.PUT_LINE(v_autor.prijmeni);
   END LOOP;

END;
/
```

Vidíte, že tento typ smyčky je naprosto nejkompaktnější. Vrací tytéž výsledky jako dva předchozí anonymní bloky, ale nemusíte vypisovat příkazy `OPEN`, `FETCH` ani `CLOSE`.

Implicitní kurzory

Implicitní kurzory Oracle otevírá a zavírá automaticky. Ve skutečnosti každý prováděný příkaz DML má kontextovou oblast v PGA, a tedy i kurzor. Vývojář nemusí pro použití implicitních kurzorů vyvíjet žádnou aktivitu. Příkazy OPEN, FETCH a CLOSE se nepoužívají, ale stejných 6 atributů, jako pro explicitní kurzory, můžete použít i pro implicitní kurzory (viz dříve, tabulka 4.4).

Příklad provádí aktualizaci a atributy kurzoru používá na test výsledku:

```
-- skript naleznete na DVD jako součást souboru ImplicitAttribute.sql
SET SERVEROUTPUT ON
BEGIN
```

```
   DBMS_OUTPUT.ENABLE(1000000);

   UPDATE knihy
   SET cena = cena * .90
   WHERE isbn = '78824389';

   DBMS_OUTPUT.PUT_LINE('Upraveno řádků: '||SQL%ROWCOUNT);

   IF SQL%NOTFOUND
   THEN
      DBMS_OUTPUT.PUT_LINE('Nelze upravit isbn 78824389');
   END IF;

   COMMIT;

EXCEPTION
   WHEN OTHERS
      THEN
      DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```

Použijete-li u implicitních kurzorů atribut %ISOPEN, dostanete vždy hodnotu FALSE, protože kurzory se zavírají automaticky. I když nedojde k chybě, přesto se u implicitních kurzorů atribut %ISOPEN celkem k ničemu nehodí.

Kurzorové proměnné

Kurzorové *proměnné* nabízí dynamickou a stálou alternativu ke statickým explicitním kurzorům, které jsme si již ukázali. Kurzorové proměnné se vyčíslují za běhu, nikoli během překladu. Je možné je otevřít pro více dotazů SELECT v jediném bloku.

Kurzorové proměnné je možné implementovat různě podle vašich potřeb. Většinou se ovládají stejně jako explicitní kurzory. To znamená, že je není potřeba explicitně uzavírat, a používat příkaz FETCH také nemusíte, abyste získali záznamy, i když to není vyloučeno. Lepší vlastností kurzorových proměnných je, že nabízí způsob, jak mohou procedury (viz kapitola 8) vracet sadu záznamů.

Následující anonymní blok deklaruje kurzorovou proměnnou a pak kurzor otevírá, načítá z něj a zavírá jej:

```
-- skript naleznete na DVD jako součást souboru CursorVariable1.sql
SET SERVEROUTPUT ON
DECLARE

   TYPE kniha_typ IS REF CURSOR RETURN KNIHY%ROWTYPE;
   cv_knihy kniha_typ;
   v_knihy KNIHY%ROWTYPE;

BEGIN

   DBMS_OUTPUT.ENABLE(1000000);

   OPEN cv_knihy FOR
   SELECT *
```

```
   FROM knihy
   WHERE isbn = '78824389';

   FETCH cv_knihy INTO v_knihy;

   DBMS_OUTPUT.PUT_LINE(v_knihy.nazev||' stojí '||v_knihy.cena);

   CLOSE cv_knihy;
END;
/
```

V tomto příkladě jsme deklarovali typ se jménem `kniha_typ` jako REF CURSOR. Pak jsme deklarovali kurzorovou proměnnou tohoto typu a také lokální proměnnou, do níž se budou načítat záznamy pomocí příkazu FETCH. A na závěr kurzor uzavíráme.

Příklad ovšem nepředvedl jednu z mých oblíbených vlastností u kurzorových proměnných. Následující ukázka je uložená procedura, která vrací do okna SQL*Plus výsledkovou množinu. V příkladu se v deklaraci kurzoru používá vestavěný typ SYS_REFCURSOR (dostupný v Oracle Database 9*i*):

```
-- skript naleznete na DVD jako součást souboru CursorVariable2.sql
SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE autori_vyber (
   cv_vysledky IN OUT SYS_REFCURSOR)
IS
BEGIN
   OPEN cv_vysledky FOR
   SELECT id, jmeno, prijmeni
   FROM autori;
END;
/
```

V proceduře jsme deklarovali kurzorovou proměnnou a propojili ji při otevření s příkazem SELECT. Z kurzoru jsme nenačítali ani jsme jej nezavírali, protože chceme, aby klientská aplikace měla výsledkovou množinu k dispozici i po dokončení programu. Spustíme jej následovně:

```
-- skript naleznete na DVD jako součást souboru CursorVariable2.sql
COL jmeno FORMAT A12
VARIABLE x REFCURSOR
EXEC autori_vyber(:x)
PRINT x
```

Výsledky jsou podle očekávání:

```
ID    JMENO    PRIJMENI
----- -------- ----------
1     Marlene  Theriault
2     Rachel   Carmichael
3     James    Viscusi
```

Kdybychom bývali kurzorovou proměnnou zavřeli uvnitř procedury, přesto by bývala šla přeložit. Ale nebyli bychom bývali schopni z ní v okně SQL*Plus získat výsledky.

Kurzorové poddotazy

Kurzorové *poddotazy*, kterým se někdy také říká vnořené kurzory, se objevily v Oracle Database 9*i* spolu se svou integrací do analyzátorů SQL*Plus a PL/SQL. Existovaly sice už v SQL v Oracle Database 8*i*, ale bez spolupráce s PL/SQL bylo jejich použití výrazně limitováno.

Kurzorové poddotazy používají výraz pro kurzor uvnitř dotazu SQL SELECT. Je možné použít všechny až dosud definované typy kurzorů, jediná výjimka platí pro implicitní kurzory. Návratový typ je vždy REF CURSOR.

Následující příklad využívá kurzorový poddotaz v explicitním kurzoru:

```
-- skript naleznete na DVD jako součást souboru CursorSubquery.sql
SET SERVEROUTPUT ON
DECLARE

   cv_autor SYS_REFCURSOR;
   v_nazev KNIHY.NAZEV%TYPE;
   v_autor AUTORI%ROWTYPE;
   v_pocet PLS_INTEGER := 0;

   CURSOR kniha_cur
   IS
      SELECT k.nazev,
         CURSOR (SELECT *
                 FROM autori a
                 WHERE a.id = k.autor1
                 OR a.id = k.autor2
                 OR a.id = k.autor3)
      FROM knihy k
      WHERE isbn = '78824389';

BEGIN

   DBMS_OUTPUT.ENABLE(1000000);

   OPEN kniha_cur;

   LOOP
      FETCH kniha_cur INTO v_nazev, cv_autor;
      EXIT WHEN kniha_cur%NOTFOUND;

      v_pocet := 0;

      DBMS_OUTPUT.PUT_LINE('Název z hlavního kurzoru: '||v_nazev);

      LOOP
         FETCH cv_autor INTO v_autor;
         EXIT WHEN cv_autor%NOTFOUND;

         v_pocet := v_pocet + 1;

         DBMS_OUTPUT.PUT_LINE('Autor'||v_pocet||': '
                              ||v_autor.jmeno||' '
                              ||v_autor.prijmeni);
      END LOOP;
   END LOOP;
```

```
   CLOSE kniha_cur;

END;
/
```

Po spuštění vrací blok:

```
Název z hlavního kurzoru: Oracle PL/SQL Tips and Techniques
Autor1: Brad Brown
Autor2: Rich Niemic
Autor3: Joe Trezzo
```

Otevřené kurzory

Počet povolených otevřených kurzorů v libovolném okamžiku se řídí parametrem OPEN_CURSORS v `init.ora`. Chcete-li vědět, jaký je maximální počet, spusťte následující dotaz:

```
SELECT value
FROM v$parameter
WHERE name = 'open_cursors';
```

Naše nastavení je velmi nízké, na hodnotu 20, abychom si ukázali, co se stane, když toto číslo překročíme. Otestujeme to tak, že v předchozím příkladě uděláme změnu, aby vracel všechny záznamy. Kurzor vypadá takto:

```
-- skript naleznete na DVD jako součást souboru OpenCursor.sql
CURSOR kniha_cur
   IS
      SELECT k.nazev,
         CURSOR (SELECT *
                 FROM autori a
                 WHERE a.id = k.autor1
                 OR a.id = k.autor2
                 OR a.id = k.autor3)
      FROM knihy k;
```

Vše, co jsme odstranili, je klauzule `WHERE`. Spustíte-li tento anonymní blok, bude probíhat smyčka přes každou knihu a budou se tisknout její autoři, ale nakonec skončí následující chybou:

```
DECLARE
*
ERROR na řádku 1:
ORA-01000: překročen maximální počet otevřených kurzorů
ORA-06512: na line 25
```

DML a DDL

Jazyk pro práci s daty (Data Manipulation Language, DML) obsahuje příkazy INSERT, UPDATE a DELETE, pomocí nichž můžete upravovat data. PL/SQL podporuje příkazy DML přímo. Následující příklad je anonymní blok, který aktualizuje tabulku dual (no, ve skutečnosti vlastně ne... podívejte se na klauzuli WHERE).

```
-- skript naleznete na DVD jako součást souboru UpdateDual.sql
BEGIN
   UPDATE dual
   SET dummy = 'x'
   WHERE 1=2;
END;
/
```

Nikdy, skutečně nikdy neaktualizujte tabulku dual! Toto je běžný způsob, jak spustit transakci, která nic neudělá. Klauzule `WHERE` nikdy není vyhodnocena jako pravdivá, protože 1 se nemůže rovnat 2.

Blok se přeloží bez chyb. Nyní si promyslete další jednoduchý příklad, který používá DDL. Příklad vytváří tabulku s jedním sloupcem:

```
-- skript naleznete na DVD jako součást souboru DDL.sql
BEGIN
   CREATE TABLE ddl_tabulka (
      id NUMBER(10));
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/
```

Skript selže!

```
ERROR na řádku 2:
ORA-06550: řádka 2, sloupec 4:
PLS-00103: nalezen symbol "CREATE" v situaci, kdy se předpokládala
    jedna z následujících možností:
begin case declare exit for goto if loop mod null pragma
raise return select update while with <identifikátor>
...
```

Proč selhal?

Prekompilace

Objekty PL/SQL jsou předpřeloženy. Před spuštěním se ověřují všechny závislosti, což spouštění programu podstatně zrychluje. Závislosti nesouvisí s daty. Existují na jiných databázových objektech, jako jsou tabulky, pohledy, synonyma a ostatní programové struktury. DML, které běží v PL/SQL, nemá jako takové žádnou možnost změnit nějakou závislost a vyvolat chybu v programu. Na druhé straně ovšem DDL, v němž jsou příkazy CREATE, DROP či ALTER i příkazy ohledně oprávnění GRANT a REVOKE, může v průběhu spuštění změnit závislosti, je-li to povoleno.

Pokud bychom měli například blok, který by nejprve odstranil nějakou tabulku a pak by se v ní pokusil provést aktualizaci, tak by samozřejmě neproběhl v pořádku. Takovou závislost ovšem nelze ověřit dopředu. Až do chvíle spuštění by aktualizační příkaz UPDATE vypadal správně, protože v tu chvíli tabulka existuje. Selhal by až ve chvíli spuštění bloku, protože tabulka už by neexistovala.

Výrazy DDL proto nejsou v PL/SQL přímo povoleny. Jak si ukážeme později v tomto oddíle i v kapitole 13, nabízí Oracle možnost, jak tento zákaz obejít.

Práce s daty pomocí DML

Na začátku této kapitoly v části „Transakční zpracování" jsme si řekli, že příkazy DML vyžadují explicitní příkaz `COMMIT`, než se změny promítnou natrvalo. DML obsahuje také příkazy `ROLLBACK` a `SAVEPOINT`, s nimiž můžete změny zrušit před jejich trvalým zapsáním.

Ne všechny klauzule, které můžete v těchto příkazech DML použít, vám zde popíšeme. Celý seznam dostupných klauzulí naleznete v nápovědě Oracle Database SQL na adrese *http://otn.oracle.com*.

Vkládání (INSERT)

Příkaz INSERT vkládá záznamy do tabulky. Základní syntaxe vypadá takto:

INSERT INTO *Název_tabulky* [(*seznam_sloupců*)]
VALUES *dotaz* | (*seznam_hodnot*);

Název_tabulky může být tabulka, synonymum nebo aktualizovatelný pohled. *Seznam_sloupců* je nepovinný, ale velice vám doporučujeme jej používat a předcházet tak problémům s hodnotami, které se vloží do špatných sloupců. Také to napomáhá čitelnosti a údržbě. V klauzuli `VALUES` může být dotaz `SELECT`, jenž načte stejný počet sloupců stejného typu, jako má cílová tabulka, nebo seznam hodnot. Použijete-li *seznam_hodnot*, musí být v závorkách a může obsahovat literály či proměnné. *Seznam_hodnot* může být libovolný platný výraz (viz definice v kapitole 3).

V následujícím příkladu jsou v seznamu hodnot literály i proměnné:

```
-- skript naleznete na DVD jako součást souboru Insert.sql
SET SERVEROUTPUT ON
DECLARE

   v_isbn KNIHY.ISBN%TYPE := '12345678';
   v_kategorie KNIHY.KATEGORIE%TYPE := 'Oracle Server';
   v_nazev KNIHY.NAZEV%TYPE := 'Oracle Information Retrieval';

BEGIN

   INSERT INTO knihy (ISBN,KATEGORIE,NAZEV,POCET_STRAN,CENA,
                      COPYRIGHT,AUTOR1)
   VALUES (v_isbn, v_kategorie, v_nazev, 450, 39.95,
           2005, 44);

   COMMIT;

EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE(SQLERRM);
      ROLLBACK;
END;
/
```

Všimněte si příkazů řízení transakce (COMMIT a ROLLBACK), které se používají ve všech ukázkách DML. Podrobněji to probereme v kapitole 7.

Aktualizace (UPDATE)

Příkaz UPDATE mění existující data a platí pro něj stejná pravidla řízení transakcí jako pro INSERT. Syntaxe je tato:

UPDATE *Název_tabulky*
SET *název_sloupce* = *dotaz* | *hodnota* [, název_sloupce = hodnota]
[WHERE *klauzule_where* | WHERE CURRENT OF *kurzor*];

Název_tabulky může být libovolná tabulka, synonymum nebo aktualizovatelný pohled. *Název_sloupce* je libovolný sloupec z tabulky určené *názvem_tabulky*. V klauzuli SET může být více než jeden *název_sloupce*, navzájem se oddělují čárkou. Sloupcům lze přiřazovat celá čísla, proměnné nebo jakékoli platné výrazy. Je také možné jim přiřadit výsledek poddotazu. Nepovinná klauzule WHERE CURRENT OF se hodí na práci s kurzorem, který jste deklarovali jako FOR UPDATE. V *klauzuli_where* může být srovnání jakéhokoliv sloupce dané tabulky s libovolným výrazem. Klauzule WHERE CURRENT OF pracuje s příkazy UPDATE a INSERT a říká, že práce se má provést na aktuálním záznamu v *kurzoru*.

V prvním příkladě provedeme úpravu v tabulce a hodnotu odvodíme z proměnné téhož typu, jaký má upravovaný sloupec.

```
-- skript naleznete na DVD jako součást souboru Update.sql
SET SERVEROUTPUT ON
DECLARE

   v_pocet_stran KNIHY.POCET_STRAN%TYPE;
   v_isbn KNIHY.ISBN%TYPE := '72230665';

BEGIN

   SELECT pocet_stran
   INTO v_pocet_stran
   FROM knihy
   WHERE isbn = v_isbn;

   DBMS_OUTPUT.PUT_LINE('Počet stran před: '||v_pocet_stran);

   v_pocet_stran := v_pocet_stran + 200;

   UPDATE knihy
   SET pocet_stran = v_pocet_stran
   WHERE isbn = v_isbn;

   DBMS_OUTPUT.PUT_LINE('Počet stran po: '||v_pocet_stran);

   COMMIT;

EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE(SQLERRM);
      ROLLBACK;
END;
/
```

Ve druhém příkladě použijeme klauzuli WHERE CURRENT OF.

```
-- skript naleznete na DVD jako součást souboru WhereCurrentOf.sql
SET SERVEROUTPUT ON
DECLARE

   v_isbn KATALOG.ISBN%TYPE;
   v_mnozstvi KATALOG.MNOZSTVI%TYPE;

   CURSOR katalog_cur
   IS
      SELECT isbn, mnozstvi
      FROM katalog
      WHERE status = 'NA SKLADĚ'
      AND isbn IN (SELECT isbn
                   FROM knihy
                   WHERE cena > 40)
      FOR UPDATE OF mnozstvi;

BEGIN

   FOR y IN katalog_cur
   LOOP
      FETCH katalog_cur INTO v_isbn, v_mnozstvi;
      EXIT WHEN katalog_cur%NOTFOUND;

      DBMS_OUTPUT.PUT_LINE(v_isbn||'množství NA SKLADĚ před: '||v_mnozstvi);

      v_mnozstvi := v_mnozstvi + 250;

      UPDATE katalog
      SET mnozstvi = v_mnozstvi
      WHERE CURRENT OF katalog_cur;

      DBMS_OUTPUT.PUT_LINE(v_isbn||'množství NA SKLADĚ po: '||v_mnozstvi);

   END LOOP;

   COMMIT;

EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE(SQLERRM);
      ROLLBACK;
END;
/
```

Příkaz UPDATE aktualizuje pouze aktuální záznam v kurzoru.

Mazání dat (DELETE)

Příkazy DELETE odstraňují data. Platí pro ně tatáž pravidla jako pro vkládání a aktualizace. Syntaxe příkazu DELETE je:

DELETE FROM *název_tabulky*
[WHERE *klauzule_where* | WHERE CURRENT OF *kurzor*]

Název_tabulky může být libovolná tabulka, synonymum nebo aktualizovatelný pohled, kde uživatel má oprávnění DELETE. Jestliže vynecháte klauzuli WHERE, smažou se všechny záznamy. V *klauzuli_where* může být srovnání libovolného sloupce v tabulce s libovolným výrazem. Klauzule WHERE CURRENT OF funguje v příkazech UPDATE a DELETE a říká, že proces má proběhnout na aktuálním záznamu v kurzoru.

Následující příklad provádí akci `DELETE` na tabulce `AUTORI`:

```
-- skript naleznete na DVD jako součást souboru Delete.sql
SET SERVEROUTPUT ON
DECLARE

   v_autor AUTORI%ROWTYPE;

BEGIN

   SELECT *
   INTO v_autor
   FROM autori
   WHERE id = 54;

   DELETE FROM autori
   WHERE id = v_autor.id;

   DBMS_OUTPUT.PUT_LINE('Autor '||v_autor.jmeno
                        ||' '||v_autor.prijmeni
                        ||' byl odstraněn.');

   COMMIT;

EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE(SQLERRM);
      ROLLBACK;
END;
/
```

Mazání proběhlo úspěšně:

```
Autor Charles Moffett byl odstraněn.
```

Úvod do dynamického SQL

Až dosud byly výrazy SQL, které jsme vám ukazovali, statické. Byly předpřeložené spolu s kódem a nebylo možné je měnit. Dynamické SQL vkládáme a spouštíme během exekuce bloku. Používáme buď vestavěný balík DBMS_SQL, nebo nativní dynamické SQL (Native Dynamic SQL, NDS).

Už jsme si řekli, že PL/SQL nepodporuje přímo DDL. Vestavěný balík DBMS_SQL nabízí několik desítek procedur a funkcí, které umožňují používat dynamické SQL včetně DDL. Balík byl už v Oracle Database 7.1, a i když není moc efektivní, obsahuje několik málo vlastností, které nejsou v NDS.

V následném vývoji se již patrně využívá nativní dynamické SQL (NDS). Objevilo se v Oracle Database 8*i* a ke spuštění je potřeba mnohem méně kroků. Následující oddíl obsahuje přehled NDS. V kapitole 13 pohovoříme o DBMS_SQL a NDS podrobně.

Nativní dynamické SQL

NDS využívá jednoduchého příkazu EXECUTE IMMEDIATE, s jehož pomocí spouští výrazy dynamicky v bloku PL/SQL. Neumím ani vypovědět, jak jsem byl nadšený, když jsem tuto vlastnost v Oracle Database 8*i* uviděl. DBMS_SQL je sice užitečný balík, ale s jednoduchostí a výkonem NDS pro většinu operací se nedá srovnávat.

Příklad ukazuje, jak dynamicky vytvořit výraz a spustit jej v PL/SQL pomocí EXECUTE IMMEDIATE.

```
-- skript naleznete na DVD jako součást souboru NDS.sql
SET SERVEROUTPUT ON
DECLARE

   v_vyraz VARCHAR2(500);

   CURSOR trigger_cur
   IS
      SELECT trigger_name
      FROM user_triggers;

BEGIN

   FOR y IN trigger_cur
   LOOP
         -- vytvoříme výraz
         v_vyraz := 'ALTER TRIGGER '||y.trigger_name||' DISABLE';

         -- spustíme výraz
         EXECUTE IMMEDIATE v_vyraz;
   END LOOP;

END;
/
```

Blok provádí následující operace:

- Prochází ve smyčce všechny názvy spouští v aktuálním schématu.
- Vytváří výraz DDL, který vypíná spoušť získanou v kurzoru.
- Tento výraz ve smyčce provádí, jedenkrát pro každý záznam v kurzoru.

Další příklad naleznete ve skriptu `CreateUser.sql`.

ROWID a ROWNUM

Stejně jako sloupec `LEVEL`, o němž jsme již mluvili, i `ROWID` a `ROWNUM` jsou pseudosloupce, které lze použít při vývoji aplikací. Už ze jmen je zřejmé, že souvisí s jednotlivými řádky, resp. záznamy v tabulce. Mají však výrazně odlišnou strukturu a účel.

ROWID

ROWID je systémem generovaný jedinečný identifikátor, který se vytváří pro každý záznam v databázi. Tato binární hodnota je adresa, případně pozice dat v systému. ROWID může být fyzické, stejně jako záznamy v běžné databázové tabulce. Jak jsme si řekli v kapitole 3, může být ROWID i logické, jako je tomu v případě řádků v indexově organizované tabulce.

V mém systému vrací ROWID pro záznam s mým jménem v tabulce AUTORI následující:

```
SELECT rowid
   FROM autori
   WHERE jmeno = 'Ron';
```

Výsledek:

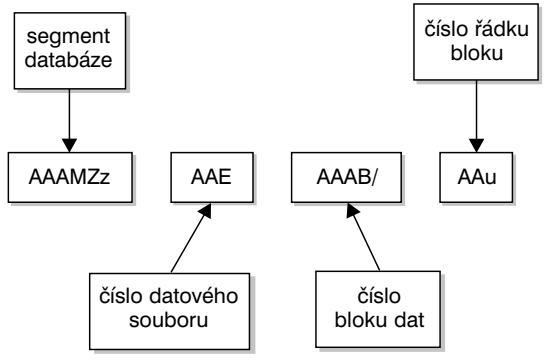
```
ROWID
---------
AAAMZzAAEAAAAB/Aau
```

Vaše rowid se bude od mého lišit. Je to číslo generované systémem, a proto není možné jej předpovědět.

Jde o fyzické umístění záznamu s mým jménem v tabulce AUTORI. Pro tento záznam v databázi je jedinečné. Rozbor struktury máte na obrázku 4.4.

ROWID používají šifrování ve formátu 64 a při dotazu v SQL*Plus vracejí desetibajtovou hodnotu jako řetězec. Struktura ROWID je sice snadno pochopitelná, ale dešifrovat při pohledu na něj skutečné umístění řádku je podobné situaci, v níž byste měli poznat člověka podle jeho DNA.

Naštěstí nemusíme hodnotu chápat, stačí, že to umí Oracle. Využijeme pouze výkonové výhody, kterou nám její použití při přístupu k datům nabízí.



Obrázek 4.4 Struktura ROWID

ROWID a výkon

Jednou z největších výhod ROWID je zisk výkonu, kterého dosáhneme, když je budeme používat při odkazování na záznam. Není potřeba žádný index, není nutné určovat, je-li lepší procházet celou tabulku, a není třeba se starat o kardinalitu. ROWID je přímou adresou záznamu, není tedy nutné nic překládat.

Následující příklad ilustruje použití ROWID s příkazem UPDATE. Nejprve vybereme stejné záznamy jako v později používaném kurzoru, abychom si ukázali jejich aktuální stav:

```
-- skript naleznete na DVD jako součást souboru RowID.sql
COL jmeno FORMAT A7
COL prijmeni FORMAT A10
SELECT a.rowid, a.jmeno, a.prijmeni
FROM autori a, knihy k
WHERE k.isbn = '72230665'
AND (
         a.id = k.autor1
         OR
         a.id = k.autor2
         OR
         a.id = k.autor3);
```

Výstupem je ROWID, JMENO a PRIJMENI autorů této knihy:

```
ROWID              JMENO   PRIJMENI
------------------ -----   -----
AAAMaHAAEAAAAIHAAZ Scott   Urman
AAAMaHAAEAAAAIHAAu Ron     Hardman
AAAMaHAAEAAAAIHAAv Mike    McLaughlin
```

Jména jsou uložena tak, že první písmeno je velké a zbytek jsou malá písmena, my je však chceme převést na samá velká písmena. Přesně tento úkol dělá následující blok:

```
-- skript naleznete na DVD jako součást souboru RowID.sql
SET SERVEROUTPUT ON
DECLARE
/* Vložíme rowid, jména a příjmení
   autorů této knihy do kurzoru */

   CURSOR autor_rowid_cur
   IS
      SELECT a.rowid
      FROM autori a, knihy k
      WHERE k.isbn = '72230665'
      AND (
         a.id = k.autor1
         OR
         a.id = k.autor2
         OR
         a.id = k.autor3);

BEGIN

   /* Budeme ve smyčce procházet záznamy v kurzoru
      a převedeme jména a příjmení na velká písmena */
```

```
    FOR y IN autor_rowid_cur
    LOOP

       UPDATE autori
       SET jmeno = UPPER(jmeno),
           prijmeni = UPPER(prijmeni)
       WHERE rowid = y.rowid;
    END LOOP;

    COMMIT;

EXCEPTION
    WHEN OTHERS
    THEN
       DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/
```

Blok provádí následující úkony:

- Získává `ROWID` třech autorů této knihy v kurzoru `autor_rowid_cur`.
- Prochází kurzor (ukazatele na tři odpovídající řádky).
- Provádí aktualizaci, která používá `ROWID` v podmínce.

Jde o malou datovou množinu, takže jsme mohli klidně použít sloupec `AUTORI.ID` a snížení výkonu bychom nepostřehli. Ovšem ve větších aplikacích už výkon při použití této logiky vzroste výrazněji, obzvláště v případě, kdy na dané tabulce neexistuje primární klíč, který je možné použít jako alternativu.

Ověřme data ještě jednou. Výsledek je podle našich očekávání.

```
ROWID              JMENO   PRIJMENI
------------------ -----   -----
AAAMaHAAEAAAAIHAAZ SCOTT   URMAN
AAAMaHAAEAAAAIHAAu RON     HARDMAN
AAAMaHAAEAAAAIHAAv MIKE    MCLAUGHLIN
```

Chcete-li se přesvědčit, že došlo k úpravě pouze v těchto záznamech, spusťte následující dotaz, který vybírá pouze první řádek v tabulce:

```
SELECT rowid, jmeno, prijmeni
    FROM autori
    WHERE rownum = 1;
```

Dostanete

```
ROWID              JMENO    PRIJMENI
------------------ -----    -----
AAAMaHAAEAAAAIHAAA Marlene  Theriault
```

Protože jméno není velkými písmeny, můžeme si být jisti, že úprava dat se omezila podle `ROWID`. Co se týče použití `ROWNUM` v posledním dotazu, podíváme se na ně nyní.

ROWNUM

Pseudosloupec `ROWNUM` vrací číslo řádku záznamu. V posledním příkladě jsme použili `ROWNUM` a s jeho pomocí získali zcela první záznam v tabulce `AUTORI`. Jde o logické

číslo, které se určuje v okamžiku spuštění dotazu. Příkazy vložení či mazání řádků mohou způsobit jiné přiřazení ROWNUM. Čísla řádků nejsou svázaná s konkrétními záznamy, a proto se na ně nikdy nespoléhejte tak, jako na fyzická ROWID.

Jedno z častých použití ROWNUM je omezení počtu navrácených záznamů. Můžeme spustit následující dotaz a omezit počet navrácených záznamů z tabulky na horních deset:

```
SELECT nazev
FROM knihy
WHERE ROWNUM <= 10;
```

Tento typ dotazu je možné použít ve funkci či proceduře (s typem REFCURSOR) a omezit tak případné rozsáhlé návratové množiny.

O procedurách a funkcích si více povíme v deváté kapitole.

Použití ROWNUM spolu s klauzulí ORDER BY

Použijete-li ROWNUM se seřazenou výsledkovou množinou, nemusíte dostat očekávané výsledky. Chcete-li například dostat jména autorů v abecedním pořádku, ale požadujete-li pouze horních 10 záznamů se seřazeného seznamu, nemůžete použít základní dotaz s ORDER BY a ROWNUM.

Podívejte se na proceduru AUTOR_VYBER, která vybírá jména autorů, řadí je podle příjmení a omezuje pomocí ROWNUM:

```
CREATE OR REPLACE PROCEDURE autor_vyber (
   cv_autori IN OUT SYS_REFCURSOR)
IS
BEGIN
   OPEN cv_autori FOR
   SELECT id, prijmeni||', '||jmeno JMENO
   FROM autori
   WHERE rownum <= 10
   ORDER BY prijmeni;
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/
```

Možná si myslíte, že procedura seřadí všechny výsledky a vybere prvních deset seřazených vzestupně podle příjmení, ale není to pravda. Proceduru spustíte následovně:

```
VARIABLE x REFCURSOR
EXEC autor_vyber(:x)
```

Výsledek je uložen v proměnné x, kterou pomocí následujícího příkazu vypíšeme na obrazovku:

```
COL jmeno FORMAT A30
PRINT x
```

Výsledek vrací horních deset záznamů v tabulce a pak je řadí podle příjmení. Ale my jsme chtěli, aby napřed seřadil řádky a pak vybral horních deset.

```
ID     JMENO
------ -----------
4      Abbey, Michael
9      Abramson, Ian
2      Carmichael, Rachel
5      Corey, Michael
7      Deshpande, Kirtikumar
8      Kostelac, John
10     Smith, Kenny
1      Theriault, Marlene
6      Vaidyanatha, Gaja
3      Viscusi, James
```

To tedy není požadovaný výsledek. Přemýšlejte v souvislostech obchodních aplikací, kde by se objednávky patrně měly zobrazovat podle data, ale jen posledních deset. Naše výsledky by však obsahovaly prvních deset řádků v tabulce seřazených podle data, a nikoli deset posledních objednávek.

Řádkový pohled (inline view). Problém s `ROWNUM` a `ORDER BY` můžeme vyřešit pomocí řádkového pohledu. Řádkové pohledy jsou poddotazy v klauzuli `FROM`, které slouží jako pohled v okamžiku provádění. Nejde o pojmenované pohledy, které se ukládají do databáze.

Následující úprava procedury `AUTOR_VYBER` řeší náš problém:

```
CREATE OR REPLACE PROCEDURE autor_serazeny_vyber(
   cv_autori IN OUT SYS_REFCURSOR)
IS
BEGIN
   OPEN cv_autori FOR
   SELECT *
   FROM (
         SELECT id, prijmeni||', '||jmeno JMENO
         FROM autori
         ORDER BY prijmeni) SERAZENI_AUTORI
   WHERE rownum <= 10;
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE(sqlerrm);
END;
/
```

Proceduru spustíme jako v předcházejícím příkladě, jen musíme použít její správný název:

```
VARIABLE x REFCURSOR
EXEC autor_serazeny_vyber(:x)
COL jmeno FORMAT A30
PRINT x
```

Nyní procedura vrací přesně to, co potřebujeme: seřazený seznam dat, který jsme omezili na horních deset záznamů.

```
ID     JMENO
------ -----------
4      Abbey, Michael
9      Abramson, Ian
33     Adkoli, Anand
14     Allen, Christopher
30     Armstrong-Smith, Michael
31     Armstrong-Smith, Darlene
12     Bo, Lars
51     Boudreaux, Scott
36     Brown, Brad
35     Burleson, Donald
```

Seřazený řádkový pohled vyřešil problém a umožnil nám použít `ROWNUM` k dosažení potřebných výsledků.

Vestavěné funkce SQL

Kromě příkazů SQL podporuje PL/SQL i většinu vestavěných funkcí SQL. I když je kompletní popis těchto funkcí nad možnosti této knihy, rozdělili jsme nejpoužívanější funkce do kategorií.

Tyto vestavěné funkce jsou ve skutečnosti součástí balíku PL/SQL, jenž se jmenuje `STANDARD`. Tento balík, který patří uživateli `SYS`, sdružuje funkce dohromady, aby se snáze používaly a udržovaly. Pokud někdy budete potřebovat strukturu funkce, jednoduše napište `DESC STANDARD` a zobrazí se názvy všech parametrů a datové typy. Balíky si podrobněji popíšeme v kapitolách 8 a 9.

Další informace o všech funkcích SQL naleznete v nápovědě Oracle Database SQL na webu OTN (*http://otn.oracle.com*).

Znakové funkce

Do znakových funkcí vstupují hodnoty typu `VARCHAR2` nebo `CHAR` a výstupem je opět znakový typ nebo číslo. Příkladem takové znakové funkce je LOWER, která přijímá vstupní řetězec a vrací tentýž řetězec převedený na malá písmena.

```
-- skript naleznete na DVD jako součást souboru Lower.sql
SET SERVEROUTPUT ON
BEGIN
   DBMS_OUTPUT.PUT_LINE(LOWER('Na VeLiKoStI nĚkDy NeZáLeŽí.'));
END;
/
```

Znakové funkce naleznete v tabulce 4.5.

Tabulka 4.5 Znakové funkce

ACII	INSTRC	NLS_LOWER	SUBSTR
ASCIISTR	LENGTH	NLS_UPPER	SUBSTR2
CHR	LENGTH2	NLSSORT	SUBSTR4

Tabulka 4.5 Znakové funkce (pokračování)

COMPOSE	LENGTH4	REGEXP_LIKE	SUBSTRB
CONCAT	LENGTHB	REGEXP_INSTR	SUBSTRC
DECOMPOSE	LENGTHC	REGEXP_REPLACE	TRANSLATE
INITCAP	LOWER	REGEXP_SUBSTR	TRIM
INSTR	LPAD	REPLACE	UNISTR
INSTR2	LTRIM	RPAD	UPPER
INSTR4	NCHR	RTRIM	
INSTRB	NLS_INITCAP	SOUNDEX	

Číselné funkce

Číselné funkce mají v argumentu číslo a číslo také vrací ve výsledku. Například funkce ROUND přijímá číslo a zaokrouhluje jej na zadaný počet míst. Syntaxe je následující:

ROUND(*a* [, *b*])

kde *a* je zaokrouhlované číslo a *b* je počet desetinných míst, na něž se má zaokrouhlit. Následující příklad ukazuje tuto funkci v akci:

```
-- skript naleznete na DVD jako součást souboru Round.sql
SET SERVEROUTPUT ON
DECLARE
   v_zaokrouhleni NUMBER (10,4) := 12345.6789;
BEGIN

   DBMS_OUTPUT.PUT_LINE('Základní nastavení: '||ROUND(v_zaokrouhleni));
   DBMS_OUTPUT.PUT_LINE('+2: '||ROUND(v_zaokrouhleni, 2));
   DBMS_OUTPUT.PUT_LINE('-2: '||ROUND(v_zaokrouhleni, -2));

END;
/
```

Výsledek je následující:

```
Základní nastavení: 12346
+2: 12345,68
-2: 12300
```

Další číselné funkce máte v tabulce 4.6.

Tabulka 4.6 Číselné funkce

ABS	CEIL	LOG	SIN
ACOS	COS	MOD	SINH
ASIN	COSH	POWER	SQRT
ATAN	EXP	REMAINDER	TAN
ATAN2	FLOOR	ROUND	TANH
BITAND	LN	SIGN	TRUNC

Funkce data a času

Funkce data a času berou jako parametr datum a vrací opět datum nebo číslo. Funkce v této kategorii jdou od SYSDATE či SYSTIMESTAMP, které vrací aktuální datum a čas z instance, až po aritmetické funkce, jako je ADD_MONTHS, které datum a čas počítají.

Příklad ukazuje použití funkcí `SYSDATE`, `SYSTIMESTAMP`, `MONTHS_BETWEEN` a `LAST_DAY`:

```
-- skript naleznete na DVD jako součást souboru DateTime.sql
SET SERVEROUTPUT ON
DECLARE
   v_systemove_datum DATE := SYSDATE;
   v_systemove_datum_a_cas TIMESTAMP := SYSTIMESTAMP;
   v_datum DATE;
   v_pocet NUMBER(10);
BEGIN
   -- vypíše aktuální datum
   DBMS_OUTPUT.PUT_LINE('Dnešní datum: '||v_systemove_datum);

   -- Vypíše aktuální datum a čas
   DBMS_OUTPUT.PUT_LINE('Dnešní datum: '||v_systemove_datum_a_cas);

   -- vypočítá počet měsíců mezi dvěma daty
   v_pocet := MONTHS_BETWEEN('13.6.1973', '13.1.1973');
   DBMS_OUTPUT.PUT_LINE('Počet měsíců mezi daty: '||v_pocet);

   -- Určuje počet dní, které ode dneška zbývají v tomto měsíci
   v_datum := LAST_DAY(v_systemove_datum);
   DBMS_OUTPUT.PUT_LINE('Poslední den tohoto měsíce: '||v_datum);
END;
/
```

Výsledek je následující:

```
Dnešní datum: 04.04.07
Dnešní datum: 04.04.07 14:07:15,554000
Počet měsíců mezi daty: 5
Poslední den tohoto měsíce: 30.04.07
```

Další funkce naleznete v tabulce 4.7.

Tabulka 4.7 Funkce data a času

ADD_MONTHS	MONTHS_BETWEEN	SYSTIMESTAMP
CURRENT_DATE	NEW_TIME	TO_DSINTERVAL
CURRENT_TIME	NEXT_DAY	TO_TIME
CURRENT_TIMESTAMP	NUMTODSINTERVAL	TO_TIME_TZ
DBTIMEZONE	NUMTOYMINTERVAL	TO_TIMESTAMP
EXTRACT	ROUND	TO_TIMESTAMP_TZ
FROM_TZ	SESSIONTIMEZONE	TO_YMINTERVAL
LAST_DAY	SYS_EXTRACT_UTC	TRUNC
LOCALTIMESTAMP	SYSDATE	TZ_OFFSET

Převodní funkce

Převodní funkce se dělí na dvě kategorie: implicitní a explicitní. Většina datových typů PL/SQL se v Oracle v případě potřeby převádí implicitně, automaticky. V případě, že výstup potřebuje specifický formát, použije se základní. Explicitní převody vyžadují speciální volání jedné z těchto funkcí, ale je pak možné určit výstupní formát.

Dvě nejčastěji používané funkce pro převody jsou TO_DATE a TO_CHAR, jež pracují s daty. Funkce TO_DATE má na vstupu řetězec a vrací výstup v datovém typu `DATE`. Když pracujete s daty, funkce `TO_CHAR` přijímá datum a vrací řetězec ve formátu `VARCHAR2`.

Následuje příklad, který převádí mezi TO_DATE a TO_CHAR a mění postupně formát:

```
-- skript naleznete na DVD jako součást souboru Conversion.sql
SET SERVEROUTPUT ON
DECLARE
   v_systemove_datum DATE := SYSDATE;
   v_datum DATE;
   v_retezec VARCHAR2(20);
BEGIN

   -- Vypíše aktuální datum
   DBMS_OUTPUT.PUT_LINE('Dnešní datum: '||v_systemove_datum);

   -- vypíše aktuální datum a čas jako řetězec a změní formát
   v_retezec := TO_CHAR(v_systemove_datum,
'DD:MM:YYYY HH24:MI:SS');
   DBMS_OUTPUT.PUT_LINE('Zobrazení jako znakový řetězec ve formátu
DD:MM:YYYY HH24:MI:SS: '||v_retezec);

   -- Převádí znakový řetězec zpět do formátu data
   v_datum := TO_DATE(v_retezec, 'DD:MM:YYYY HH24:MI:SS');
   DBMS_OUTPUT.PUT_LINE('Převedeno zpět do formátu data: '||v_datum);

END;
/
```

Výstup vypadá takto:

```
Dnešní datum: 04.04.07
Zobrazení jako znakový řetězec ve formátu DD:MM:YYYY HH24:MI:SS:
    04:04:2007 14:24:57
Převedeno zpět do formátu data: 04.04.07
```

V tabulce 4.8 jsou další převodní funkce.

Tabulka 4.8 Převodní funkce

CASE	RAWTONHEX	TO_CHAR	TO_NCLOB
CHARTOROWID	ROWIDTOCHAR	TO_CLOB	TO_NUMBER
CONVERT	TO_BINARY_DOUBLE	TO_DATE	TO_SINGLE_BYTE
HEXTORAW	TO_BLOB	TO_MULTI_BYTE	
RAWTOHEX	TO_BINARY_FLOAT	TO_NCHAR	

Chybové funkce

Chybové funkce jsou výjimečné tím, že je není možné použít v SQL. Vývojářům PL/SQL nabízí metody, jak zobrazovat chyby, ke kterým dojde v průběhu programu. SQLERRM vrací text chybové zprávy a SQLCODE vrací kód chyby. Následující blok vám to předvede:

```
-- skript naleznete na DVD jako součást souboru Error.sql
SET SERVEROUTPUT ON
DECLARE
   v_chyba VARCHAR2(10);
BEGIN
   SELECT dummy
   INTO v_chyba
   FROM dual
   WHERE 1=2;
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.PUT_LINE('SQLERRM: '||SQLERRM);
      DBMS_OUTPUT.PUT_LINE('SQLCODE: '||SQLCODE);
END;
/
```

Výsledek bloku ukazuje rozdíl mezi těmito dvěma funkcemi:

```
SQLERRM: ORA-01403: nenalezena žádná data
SQLCODE: 100
```

Podrobněji si tyto chybové funkce probereme v kapitole 7.

Další funkce

Tyto funkce nelze snadno kategorizovat. Mnoho z nich si ukážeme podrobně v dalších kapitolách knihy, protože se vztahují ke konkrétním technologiím. Například funkce DEREF, REF, TREAT a VALUE souvisí s objekty a povíme si o nich v kapitolách 14 a 15. Funkce BFILENAME, EMPTY_BLOB a EMPTY_CLOB souvisí s velkými objekty (LOB) a předvedeme si je v kapitole 16.

Následující příklad ukazuje, jak se v bloku PL/SQL dají použít funkce GREATEST a LEAST:

```
-- skript naleznete na DVD jako součást souboru GreatestLeast.sql
SET SERVEROUTPUT ON
DECLARE
   v_znak VARCHAR2(10);
   v_cislo NUMBER(10);
BEGIN

   v_znak := GREATEST('A', 'B', 'C');
   v_cislo := GREATEST(1,2,3);

   DBMS_OUTPUT.PUT_LINE('Znak nejdále v abecedě: '||v_znak);
   DBMS_OUTPUT.PUT_LINE('Nejvyšší číslo: '||v_cislo);

   v_znak := LEAST('A', 'B', 'C');
   v_cislo := LEAST(1,2,3);
```

```
    DBMS_OUTPUT.PUT_LINE('Znak nejdříve v abecedě: '||v_znak);
    DBMS_OUTPUT.PUT_LINE('Nejmenší číslo: '||v_cislo);

END;
/
```

Výstup ze znakové funkce GREATEST je 'C', tedy nejzazší písmeno v abecedě ze zadaných, a číselná funkce GREATEST vrací hodnotu '3', tedy nejvyšší zadanou hodnotu. Funkce LEAST funguje opačně.

```
Znak nejdále v abecedě: C
Nejvyšší číslo: 3
Znak nejdříve v abecedě: A
Nejmenší číslo: 1
```

V tabulce 4.9 jsou všechny funkce SQL, které nebyly v ostatních tabulkách.

Tabulka 4.9 Další funkce

BFILENAME	EMPTY_CLOB	NLS_CHARSET_NAME	TREAT
COALESCE	GREATEST	NULLIF	UID
DECODE	LEAST	NVL	USER
DEREF	NANVL	REF	USERENV
DUMP	NLS_CHARSET_ DECL_LEN	SYS_CONTEXT	VALUE
EMPTY_BLOB	NLS_CHARSET_ID	SYS_GUID	VSIZE

Souhrn

V této kapitole jsme si ukázali, jak se v PL/SQL používá SQL, včetně

- získávání dat pomocí příkazu SELECT, regulárních výrazů a Oracle Textu,
- využití kurzorů,
- práce s daty pomocí DML,
- vestavěných funkcí,
- ROWID a ROWNUM,
- dynamického SQL.

V další kapitole se zaměříme na záznamy v PL/SQL.