Databázové systémy

Dotazovací jazyk SQL - III

Vnořený select: kam všude

```
SELECT (SELECT ...)
   FROM (SELECT ...) tname
   WHERE  abc > (SELECT ...)
               or abc <= ANY (SELECT ...)
               or abc IN (SELECT ...)
   GROUP BY ...
      HAVING ... (SELECT ...)
```

- často musí vracet jeden sloupec
- někdy i jedinou řádku (u aritmetického porovnávání)

Vytvoření kopie existující tabulky I

```
CREATE TABLE DBPACK
    (   PACKID      CHAR(4),
        PACKNAME    CHAR(20),
        PACKVER     NUMERIC(4,2),
        PACKCOST    NUMERIC(5,2) )

INSERT INTO DBPACK
    SELECT *
        FROM PACKAGE
        WHERE PACKTYPE = 'Database'
```

V tomto případě má cílová tabulka *DBPACK* stejnou strukturu jako vzorová tabulka *PACKAGE*,

Vytvoření kopie existující tabulky II

```
CREATE TABLE WPPACK
    (    PACKID        CHAR(4),
         PACKNAME      CHAR(20),
         PACKTYPE      CHAR(15) )

INSERT INTO WPPACK
     SELECT PACKID, PACKNAME, PACKTYPE
          FROM PACKAGE
          WHERE PACKTYPE = 'Word Processing'
          ORDER BY PACKNAME
```

V tomto případě bude množina atributů cílové tabulky podmnožinou atributů vzorové tabulky.

Stejně tak množina řádků cílové tabulky bude podmnožinou množiny řádků vzorové tabulky.

Integritní omezení schematu

```
CREATE ASSERTION A1 CHECK
    ( NOT EXISTS
        ( SELECT *
            FROM PACKAGE
            WHERE PACKCOST <
                ( SELECT MAX (SOFTCOST)
                    FROM SOFTWARE
                    WHERE PACKAGE.PACKID = SOFTWARE.PACKID
    )    )       )
```

ztratilo-li toto integritní omezení smysl, lze je odstranit:

```
DROP ASSERTION A1
```

SQL – tříhodnotová logika

Jmeno	**Prijmeni**	**Student**
Jaroslav	Novák	true
Josef	Novotný	false
Jiří	Brabenec	

```
SELECT * FROM OSOBA WHERE Student != true
```

Jaký bude výsledek?

SQL – tříhodnotová logika

Jmeno	Prijmeni	Student
Jaroslav	Novák	true
Josef	Novotný	false
Jiří	Brabenec	

```
SELECT * FROM OSOBA WHERE Student != true
```

Jaký bude výsledek?

Jmeno	Prijmeni	Student
Josef	Novotný	false

SQL – tříhodnotová logika

	A = true	**A = false**	**A = null**
A = true	true	false	null
A != true	false	true	null
A = false	false	true	null
A != false	true	false	null

- is null
- is true
- is false

	A = true	**A = false**	**A = null**
A is true	true	false	false
A is not true	false	true	true
A is false	false	true	false
A is not false	true	false	true
A is null	false	false	true
A is not null	true	true	false

SQL – tříhodnotová logika

	A and B		
	B = true	**B = false**	**B = null**
A = true	true	false	null
A = false	false	false	false
A = null	null	false	null

	A or B		
	B = true	**B = false**	**B = null**
A = true	true	true	true
A = false	true	false	null
A = null	true	null	null

	Not A
A = true	false
A = false	true
A = null	null

VIEW I

View lze chápat jako tabulku, jež neobsahuje explicitně zadaná data. Tato tabulka je „pohledem“ na jinou tabulku nebo join tabulek.

View přitom slouží nejen k získání dat z databáze ale i k jejich modifikaci.

```
CREATE VIEW PACKDATABASE AS
    SELECT PACKID, PACKNAME, PACKCOST
    FROM PACKAGE
    WHERE PACKTYPE = 'Database'
```

VIEW nemusí být materializováno – zaniká spolu s databázovým spojením (session).

Materializované VIEW existuje nezávisle na databázovém spojení (session).

V PostgreSQL VIEW zaniká s databázovým spojením pokud je označený jako TEMPORARY. Jinak i nematerializovaný VIEW ve schématu zůstává.

VIEW II

PACKAGE

PACKID	PACKNAME	PACKVER	PACKTYPE	PACKCOST
AC01	Boise Accounting	3.00	Accounting	725.83
DB32	**Manta**	1.50	Database	**380.00**
DB33	**Manta**	2.10	Database	**430.18**
SS11	Limitless View	5.30	Spreadsheet	217.95
WP08	Words & More	2.00	Word Processing	185.00
WP09	Freeware Processing	4.27	Word Processing	30.00

Obsahem View PACK*DATABASE* budou buňky se žlutým pozadím.

Při definici view můžeme omezit jeho přístup pouze k vyjmenovaným atributům.

```
CREATE VIEW PACKDATABASE ( PACKID, PACKNAME, PACKCOST ) AS
   SELECT PACKID, PACKNAME, PACKCOST
   FROM PACKAGE
   WHERE PACKTYPE = 'Database'
```

Na view se lze obracet stejně jako na tabulku.

```
SELECT * FROM PACKDATABASE
WHERE PACKCOST > 400;
```

PACKID PACKNAME PACKCOST

DB33 Manta 430.18

VIEW III

Atributy view mohou mít jiná jména než atributy zdrojové tabulky.

```
CREATE VIEW DATABASE ( PKID, NAME, COST ) AS
   SELECT PACKID, PACKNAME, PACKCOST
   FROM PACKAGE
   WHERE PACKTYPE = 'Database'
```

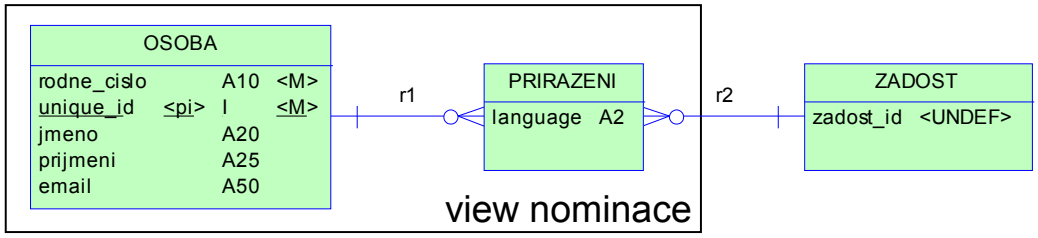
Význam view:

1. Datová nezávislost.
   Změna struktury databáze u atributů neúčastnících se view neovlivní práci s view.
2. Různé pohledy na tatáž data. Uživatel nevidí, co nemá.

Problémy při update view:

- pokud view nezahrnuje všechny sloupce původní tabulky a přidáme větu do view, jaká hodnota se v původní tabulce přiřadí atributům neúčastnícím se view? NULL
- pokus o přidání řádku ('AC01', 'DATAQUICK', 250.00) musí selhat, protože v tabulce *PACKAGE* věta s primárním klíce 'AC01' existuje. To ovšem může uživatele view překvapit, protože on vidí jen věty obsažené ve view.

VIEW IV – Updatable view in PosgreSQL



Definice view:

```
CREATE OR REPLACE VIEW nominace AS
SELECT osoba.rodne_cislo, osoba.jmeno, osoba.prijmeni, osoba.email, prirazeni.language, prirazeni.zadost_id
FROM osoba JOIN prirazeni ON osoba.unique_id = prirazeni.osoba_unique_id;
```

1. pravidlo:

```
CREATE OR REPLACE RULE "_INSERT_A_FIRST" AS
ON INSERT TO nominace
WHERE
NOT (EXISTS (SELECT 1 FROM osoba WHERE osoba.rodne_cislo = new.rodne_cislo))
DO INSTEAD
INSERT INTO osoba (rodne_cislo, jmeno, prijmeni, email)
        VALUES (new.rodne_cislo, new.jmeno, new.prijmeni, new.email);
```

2. pravidlo:

```
CREATE OR REPLACE RULE "_INSERT_Z_LAST" AS
ON INSERT TO nominace
DO INSTEAD
INSERT INTO prirazeni (zadost_ id, language,      osoba_unique_id)
        VALUES (new.zadost_id, new.language, ( SELECT osoba.unique_id
                                                FROM osoba
                                                WHERE osoba.rodne_cislo = new.rodne_cislo)));
```

Pravidla se uplatňují v ABECEDNÍM pořadí !

Indexy I

výhody:
- zvýšení efektivnosti vyhledávání (záleží na kvalitě optimalizce dotazu)
- třídění

nevýhody:
- nároky na kapacitu média
- index musí být updatován při každém update databáze

Ačkoliv standard SQL92 nedefinuje následující příkazy pro vytvoření a odstranění indexu, ve většině databázových systémů jsou k disposici v téže syntaktické podobě.

CREATE **INDEX** *CUSTIND2* ON *EMPLOYEE* (*COMPID*)

Vytvoří index pojmenovaný *CUSTIND2* pro tabulku *EMPLOYEE.* Indexačním výrazem bude jednoprvková množina sloupců { *COMPID* }.

Indexy II

CREATE INDEX *SOFTIND* ON *SOFTWARE* (*PACKID, TAGNUM*)

Index může být vytvořen nad více než jedním atributem (sloupcem).

Indexy II

CREATE INDEX *SOFTIND* ON *SOFTWARE* (*PACKID, TAGNUM*)

Index může být vytvořen nad více než jedním atributem (sloupcem).

CREATE INDEX *PACKIND3* ON *PACKAGE* (*PACKNAME, PACKVER* **DESC**)

Indexu nastavit vzestupné nebo sestupné třídění.

Indexy III

Odstranění nepotřebného klíče:

DROP INDEX *PACKIND*

Indexy IV

CREATE **UNIQUE** INDEX *PACKIND* ON *PACKAGE* (*PACKID*)

Správa indexu nedovolí, aby se v tabulce vyskytlo více vět s touž hodnotou atributu(ů) nad nímž (nimiž) je postaven daný index.

SQL92 umožňuje předepsat tuto jednoznačnost v definici tabulky popř. dalšími prostředky pro definici integritních omezení.

To je správnější, neboť tak se tato důležitá podmínka stává součástí definice schematu (fyzického datového modelu) a není ponechána na vůli správce databáze, zda vytvoří vhodný index.

(Ne)Existence indexu má mít vliv pouze na efektivitu (výkon) databáze a nikoliv na její funkci (a definice jednoznčnosti některého atributu je funkční záležitostí).

Rozšíření SQL
Domény – uživatelsky definované datové typy

```
CREATE DOMAIN LOCATIONS CHAR(12)
              CHECK ( VALUE = 'Accounting' OR
                      VALUE = 'Sales' OR
                      VALUE = 'Info Systems' OR
                      VALUE = 'Home' )
```

... potom použiji takto:

```
CREATE TABLE PC
(   ...
    ...
    LOCATION LOCATIONS
    ...
    ... )
```

Deklarace atributu *LOCATION* pomocí domény *LOCATIONS* .

Rozšíření SQL
Uživatelsky definované funkce

```
CREATE FUNCTION one() RETURNS integer AS $$
    SELECT 1 AS result;
$$ LANGUAGE SQL;

CREATE FUNCTION one() RETURNS integer AS '
    SELECT 1 AS result;
' LANGUAGE SQL;

SELECT one();
one
-----
  1
```

Rozšíření SQL
Uživatelsky definované funkce

```
CREATE FUNCTION clean_emp() RETURNS void AS '
  DELETE FROM emp
      WHERE salary < 0;
' LANGUAGE SQL;

SELECT clean_emp();
```

Rozšíření SQL
Uživatelsky definované funkce

```
CREATE FUNCTION tf1(integer, numeric) RETURNS integer AS $$
    UPDATE bank
        SET balance = balance - $2
        WHERE accountno = $1;
    SELECT balance FROM bank WHERE accountno = $1;
    -- Nebo pouze:     RETURNING balance;
$$ LANGUAGE SQL;


SELECT tf1(17, 100.0);
```

Rozšíření SQL
Uživatelsky definované funkce

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;


SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper-fooname
     1 |        1 |     Joe |     JOE
```

Rozšíření SQL
Uživatelsky definované funkce

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;
```

```
SELECT *, upper(fooname) FROM getfoo(1) AS t1;
```

```
 fooid | foosubid | fooname | upper-fooname
     1 |        1 | Joe     | JOE
     1 |        2 | Ed      | ED
```

Uložené procedury (Stored procedures)

PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python
(available in the standard PostgreSQL distribution)

Additional procedural languages:
PL/Java
PL/PHP
PL/Py
PL/R
PL/Ruby
PL/Scheme
PL/sh

Uložené procedury (Stored procedures)

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  --> 30
    quantity := 50;
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  --> Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  --> 50
    END;

    RAISE NOTICE 'Quantity here is %', quantity;  --> 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

Uložené procedury (Stored procedures)

```
-- PostgreSQL < 8.0
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;

-- PostgreSQL >= 8.0
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Uložené procedury (Stored procedures)

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);

CREATE FUNCTION getAllFoo() RETURNS SETOF foo AS
$$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN SELECT * FROM foo WHERE fooid > 0
    LOOP
        -- provedeme operace nad r
        RETURN NEXT r; -- vracíme upravený řádek SELECTu
    END LOOP;
    RETURN;
END
$$
LANGUAGE plpgsql;
```

Uložené procedury (Stored procedures)

IF ... THEN … END IF
IF ... THEN ... ELSE … END IF
IF ... THEN ... ELSIF ... THEN ... ELSE … END IF

CASE ... WHEN ... THEN ... ELSE ... END CASE (parametrizovaný)
CASE WHEN ... THEN ... ELSE ... END CASE

LOOP … END LOOP (může mít <<label>>)
WHILE … LOOP … END LOOP
FOR … IN … LOOP … END LOOP (IN query: iteruje přes výsledky)

EXIT (break, může vést na <<label>>)
EXIT WHEN …

CONTINUE
CONTINUE WHEN ...

Uložené procedury (Stored procedures)

```
SELECT * FROM emp WHERE empname = myname;
IF NOT FOUND THEN
   RAISE EXCEPTION 'employee % not found', myname;
END IF;

INSERT INTO mytab (firstname, lastname) VALUES ('Tom', 'Jones');
BEGIN
   UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
   x := x + 1;
   y := x / 0;
   RETURN y;
EXCEPTION
   WHEN division_by_zero THEN
      RAISE NOTICE 'caught division_by_zero';
      RETURN x;
END;
```

Kurzory (Cursors)

```
DECLARE
 curs1 refcursor;
 curs2 CURSOR FOR SELECT * FROM tenk1;
 curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;

OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
OPEN curs2;
OPEN curs3(42);

FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

Kurzory (Cursors)

… DECLARE, OPEN ...

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;

UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
(podobně DELETE)

CLOSE curs1;
CLOSE curs2;
CLOSE curs3;
```

Kurzory (Cursors)

```
CREATE FUNCTION myfunc(refcursor, refcursor)
  RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- musíme být v transakci, abychom mohli používat cursor
BEGIN;
SELECT * FROM myfunc('a', 'b');
FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

Triggery

```
CREATE TRIGGER check_update
    BEFORE UPDATE OF balance ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();

CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
    EXECUTE PROCEDURE check_account_update();

CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE PROCEDURE log_account_update();
```

Triggery

```
CREATE TABLE emp (empname text, salary integer, last_date timestamp,
last_user text);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $$
    BEGIN
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF TG_OP = 'UPDATE' AND OLD.salary > 1000000 THEN
            RAISE EXCEPTION '% had too much', OLD.empname;
        END IF;

        -- Remember who changed the payroll and when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```