Základy SQL
v PostgreSQL

Pavel Stěhule

http://www.pgsql.cz

7. 6. 2007

Informace a podpora PostgreSQL na internetu

- http://www.postgresql.org
- http://archives.postgresql.org
- http://www.pgsql.cz
- irc://irc.freenode.net/#postgresql
- Příklady SQL jsou převzaty z prezentace Tomáše Skopala (Databázové systémy)

Realita, databáze

Zaznamenání pravdivých tvrzení

Realita

Pokud chceme popsat určitou část reality, zaznamenáme vyjádřitelné vlastnosti objektů a vzájemné vztahy objektů: Vysoký černovlasý Josef má vztah se štíhlou blondýnkou Elsou. Bílý jogurt je v lednici v druhé polici. Chleba je někde ve spíži.

Databáze

Obsahují výroky o objektech reálného světa a zachycují určité vazby mezi těmito objekty (místo termínu objekt se též používá termín entita). Laické pojetí databází se soustřeďuje pouze na údaje o objektech a pomíjí vazby. Klasický příklad: Adresář .. eviduji pouze adresu (jednotlivé specifické vlastnosti entity typu adresa). Každá dobře navržená databáze má určitý účel, nepracuje s universem, ale s malým ostře ohraničeným výsekem universa (databáze odjezdů/příjezdů vlaků, databáze telefonních čísel), tj. databáze nejsou univerzální (slouží pouze k jednomu navrženému účelu).

Databáze
Účel

- Trvalé uložení dat,
- výběr,
- řazení,
- agregace dat - získávání odvozených dat.

Rozdělení databází z hlediska účelu

- OLTP databáze - databáze určené pro ukládání dat a operace s nimi: databázi využívá zároveň velký počet uživatelů, počítá se s velkým objemem dat, většinou se zpracovává malá podmnožina dat, minimálně se využívá cache.
- OLAP databáze - databáze určené k podpoře datových analýz: s databází pracuje malý počet uživatelů, počítá se s náročnými analytickými operacemi, jejich výsledek lze uložit do cache (vyrovnávací paměti).

Rozdělení databází
a jejich specifika

Relační databáze

Data jsou uložena v sadě tzv. normalizovaných tabulek. Změny uložených hodnot provádíme vždy nad jednou konkrétní tabulkou. Pro zobrazení dat většinou spojujeme data z několika tabulek. Všechny řádky jedné tabulky obsahují stejný počet sloupců. Každý sloupec má určený význam, jméno a datový typ. SQL je standardizovaný jazyk pro manipulaci s daty v relačních databázích. Vazby mezi normalizovanými tabulkami jsou určené výskytem stejných hodnot v tabulkách.

Nerelační databáze

Data jsou uložena v datových objektech typu záznam. Každý záznam má přidělen jedinečný systémový identifikátor. Systém eviduje vazby mezi záznamy v systémové evidenci vazeb, kde je vždy uložena dvojice jedinečných systémových identifikátorů vzájemně závislých záznamů.

Kdy použít databázi
a kdy tabulkový procesor

Kdy použít databázi

- když pracujeme se strukturovanými daty (atributy zaměstnance, ...),
- když pracujeme s velkými objemy dat (mají menší hw nároky).

Kdy je lépe použít jinou aplikaci - txt editor, tabulkový procesor

- když pracujeme s nestrukturovanými daty (text),
- když pracujeme s výpočetně komplikovanými modely (analýzy WHATIF),
- pokud pracujeme s malými datovými objemy (jedna tabulka, 10 řádků).

Normalizace
Normalizované tabulky

Neobsahují duplicity, při aktualizaci nedochází k anomáliím (je nutné aktualizovat více než jeden záznam, nebo není zřejmé co se má aktualizovat). Nedochází k duplicitám.

```
jmeno a prijmeni | Kontakt
-----------------+-----------------------------------
Pavel Stěhule    | Skalice 12, Benešov;
                 | Jugoslávských partyzánů 10, Praha
```

Pokud by došlo k změně adresy, tak se bude obtížně modifikovat záznam.

Proč navrhovat tabulky v normalizovaném tvaru?

- většinou optimální z hlediska výkonu, čitelnosti a použitelnosti (variability),
- převod dat do normalizovaných tabulek je komplikovaný a nemusí být jednoznačný.

Normalizace
Normální formy

1. normální forma (1NF)

Relace (tabulka) je v první normální formě, pokud každý její atribut (sloupec) obsahuje jen atomické hodnoty. Tedy hodnoty z pohledu databáze již dále nedělitelné. Tato podmínka není splněna například u tabulky, kde je jméno a příjmení v jednom sloupci a přitom aplikace pracuje s těmito položkami jako samostatnými.

2. normální forma (2NF)

Relace se nachází v druhé normální formě, pokud splňuje podmínky první normální formy a každý neklíčový atribut je plně závislý na primárním klíči, a to na celém klíči a nejen na nějaké jeho podmnožině.

3. normální forma (3NF)

V této formě se nachází tabulka, splňuje-li předchází dvě formy a všechny její neklíčové atributy jsou vzájemně nezávislé.

Normalizace
1. normální forma (1NF)

Příkladem, kde byla byla porušena 1NF (a to v několikrát) byla předchozí ukázka. Po normalizaci bychom dostali tabulky:

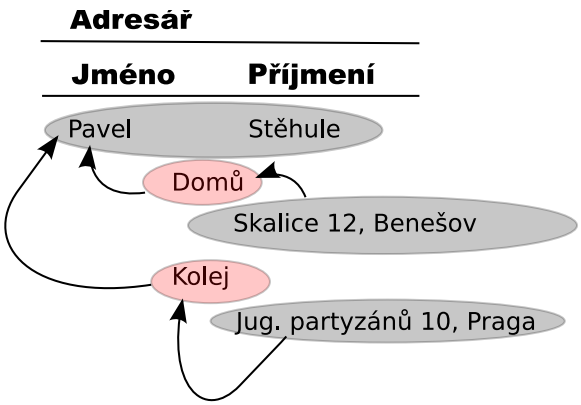
```
====== Osoby ==============
alias | jmeno  | prijmeni
------+--------+-----------
okbob | Pavel  | Stěhule
====== Adresy =============
 alias | typ      | ulice              | mesto
-------+----------+--------------------+--------
 okbob | doma     | Skalice 12         | Benešov
 okbob | kolej    | Jug. partyzánů 10  | Praha
```

Poznámka

K roztržení na dvě tabulky došlo v reakci na porušení 2.NF, kdy originální tabulka obsahovala nedefinovaný atribut *typ adresy*, který by byl částí primárního klíče.

Normalizace
Závislosti mezi atributy

Primární složený klíč (Jméno, Příjmení) není dostatečný, protože nestačí k určení adresy. Rozšířením primárního klíče na atributy (Jméno, Příjmení, Typ) zajistíme splnění 2NF normální formy.



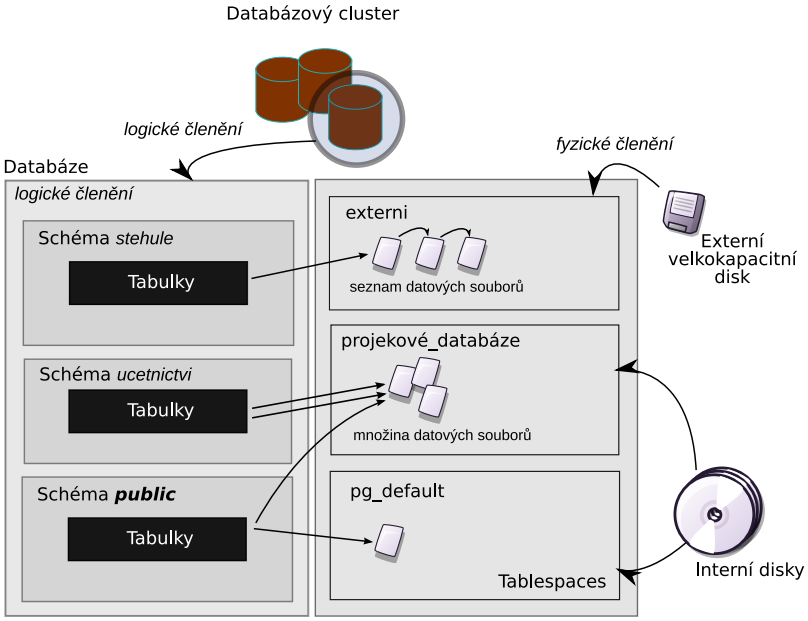
Primární klíč o třech typu `varchar` není praktický, proto přidáváme sloupec alias. Po přidání sloupce již musíme data rozdělit do dvou tabulek, abychom zajistili 2NF (neboť, *typ* bude vždy částí PK, a *Jméno* a *Příjmení* na něm nebude nikdy závislé.

Normalizace
Ukázky porušení 2NF

```
Název výrobku   | Firma             | Název kat. | Telefon
----------------+-------------------+------------+---------------
Aniseed Syrup   | Pasta Buttini     | Koření     | (089) 6547665
                | s.r.l.            |            |
----------------+-------------------+------------+---------------
Chef Anton's    | New Orleans       | Koření     | (100) 555-4822
Cajun Seasoning | Cajun Delights    |            |
----------------+-------------------+------------+---------------
Chef Anton's    | New Orleans       | Koření     | (100) 555-4822
Gumbo Mix       | Cajun Delights    |            |
```

Primárním klíčem je (Název výrobku, Firma). Atribut *Telefon* je ale závislý pouze na atributu *Firma*, nikoliv na celém složeném klíči.

Návrh datového schématu
Logické a fyzické členění databáze



Návrh datového schématu
Základní datové typy v SQL

integer - celá čísla (uloženo v 4bytech),
bigint - celá čísla (uloženo v 8bytech),
numeric - číslo uložené s deklarovanou přesností (nedochází k chybě při uložení čísla),
char - řetězec s deklarovanou délkou (nevyužité znaky se nahradí mezerou),
varchar - řetězec o maximální deklarované délce,
text - řetězec bez omezení délky,
date - dny,
time - čas,
timestamp - časové razítko, obsahuje určení dne a času,
boolean - logická hodnota,
serial - automaticky zvyšující se celočíselná hodnota.

Doménová a referenční integrita
Zajištění konzistence databáze

Referenční integrita

R.I. míníme stav databáze, kde ke každé hodnotě použité jako cizí klíč existuje odpovídající hodnota ve významu primárního klíče. Příklad: Rodiče (PK) - děti (FK) - porušením referenční integrity je dítě bez rodiče (buďto zadáme dítě s chybným identifikátorem rodiče, nebo odstraníme rodiče a zapomeneme odstranit dítě z databáze).

Doménová integrita

D.I. míníme stav databáze, kdy každá uložená hodnota vyhovuje kritériím přiřazeným k danému sloupci. Příklad: počet přijatých kusů je vždy kladné nenulové číslo. Jedná se o ochranu před neočekávanými vstupy, které by mohli vést k nesprávnému chování navazujících aplikací.

Pokud je databáze nekonzistentní, nelze se spolehnout na výsledky SQL příkazů.

Doménová a referenční integrita
Zajištění konzistence databáze

```
01 CREATE TABLE rodic (
02   id serial PRIMARY KEY,
03   jmeno varchar(20),
04
05   prijmeni varchar(20)
06 );
07
```

```
08 CREATE TABLE dite (
08   id serial PRIMARY KEY,
10   rodic_id integer
11            REFERENCES rodic(id),
12   jmeno varchar(20),
13   prijmeni varchar(20)
14 );
```

Je nezbytné udržet konzistenci databáze

Dodatečné opravy jsou daleko náročnější než okamžitá oprava po odmítnutí vstupu (ten kdo data zadává, nejlépe ví, co zadává). V databázi, která nesplňuje podmínky referenční integrity se objevují "anomálie". Podle tvaru dotazu se objevují a mizí záznamy.

Přihlášení k databázi a zápis SQL příkazu
Použití psql

```
[pavel@k153stehule ~]$ psql postgres
Welcome to psql 8.3devel, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

postgres=# SELECT CURRENT_DATE;
    date
------------
 2007-06-07
(1 row)

postgres=#
```


Přihlášení k databázi a zápis SQL příkazu
Použití psql

```
postgres=# CREATE TABLE test (a integer, b date, c varchar);
CREATE TABLE
postgres=# INSERT INTO test VALUES(10,CURRENT_DATE, 'Žlutý kůň');
INSERT 0 1
postgres=# SELECT * FROM test;
 a  |     b      |     c
----+------------+-----------
 10 | 2007-06-14 | Žlutý kůň
(1 row)

postgres=# INSERT INTO test VALUES(20,CURRENT_DATE+1, 'Červený kůň');
INSERT 0 1
postgres=# SELECT * FROM test;
 a  |     b      |      c
----+------------+-------------
 10 | 2007-06-14 | Žlutý kůň
 20 | 2007-06-15 | Červený kůň
(2 rows)

postgres=#
```

Přihlášení k databázi a zápis SQL příkazu
Použití psql

```
postgres=# UPDATE test SET c = 'Zelený kůň' WHERE a = 20;
UPDATE 1
postgres=# SELECT * FROM test;
 a  |     b      |     c
----+------------+------------
 10 | 2007-06-14 | Žlutý kůň
 20 | 2007-06-15 | Zelený kůň
(2 rows)

postgres=# DELETE FROM test WHERE a = 10;
DELETE 1
postgres=# SELECT * FROM test;
 a  |     b      |     c
----+------------+------------
 20 | 2007-06-15 | Zelený kůň
(1 row)

postgres=#
```

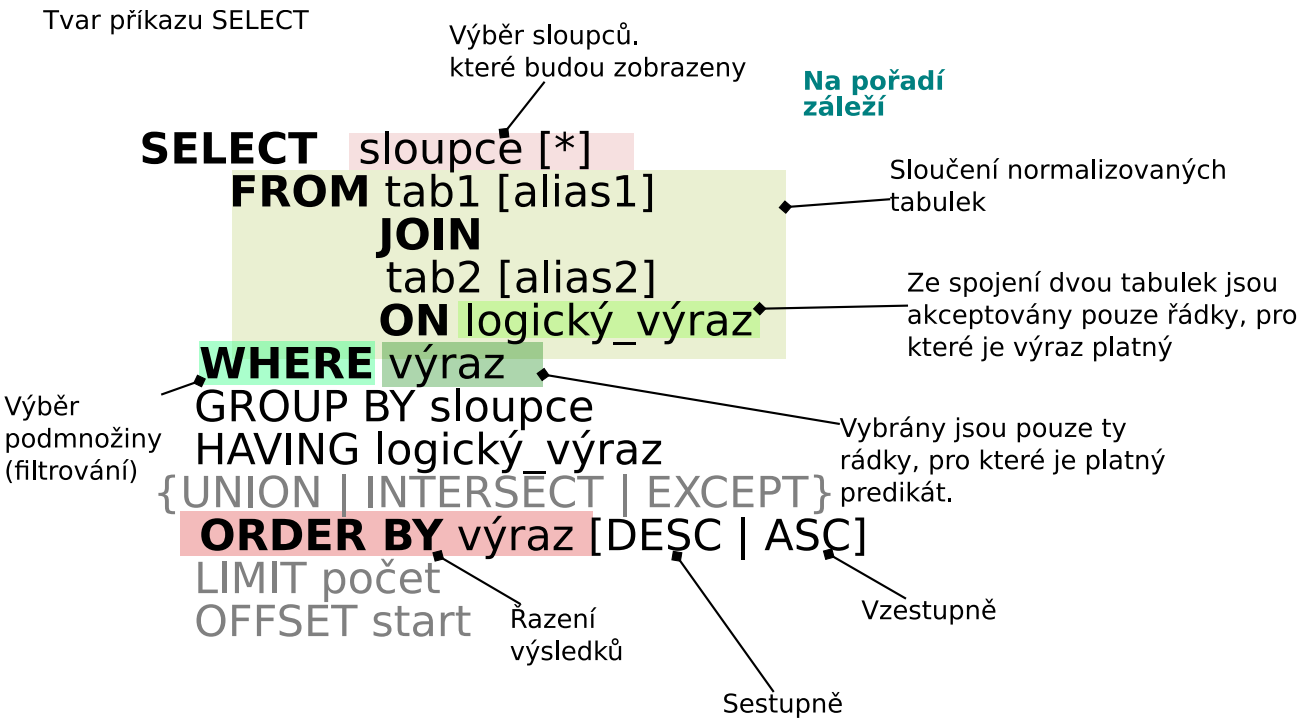

Přihlášení k databázi a zápis SQL příkazu
Použití psql

```
postgres=# \dt
           List of relations
 Schema |     Name     | Type  | Owner
--------+--------------+-------+-------
 public | test         | table | pavel
(1 rows)

postgres=# \d test
          Table "public.test"
 Column |       Type        | Modifiers
--------+-------------------+-----------
 a      | integer           |
 b      | date              |
 c      | character varying |

postgres=# DROP TABLE test;
DROP TABLE
postgres=#
```

Čtení obsahu databáze
Příkaz SELECT



Ukázky použití příkazu SELECT
Vytvoření ukázkové databáze

```
Postgres=# CREATE TABLE Osoby(
  id serial,
  jmeno varchar(10),
  prijmeni varchar(10),
  zamestnani integer,
  vek integer);
NOTICE:  CREATE TABLE will create implicit
sequence "osoby_id_seq" for serial column "osoby.id"
CREATE TABLE

postgres=# CREATE TABLE Zamestnani(id integer, profese varchar(10));
CREATE TABLE

postgres=# INSERT INTO Zamestnani VALUES(1,'Vývojář'),(2,'Dispečer');
INSERT 0 2
```

Ukázky použití příkazu SELECT
Vytvoření ukázkové databáze

```
postgres=# INSERT INTO Osoby
  VALUES (DEFAULT, 'Pavel','Stěhule',1,34),
         (DEFAULT, 'Zdeněk','Stěhule',2,29);
INSERT 0 2
postgres=# SELECT * FROM Osoby ORDER BY vek;
 id | jmeno  | prijmeni | zamestnani | vek
----+--------+----------+------------+-----
  2 | Zdeněk | Stěhule  |          2 |  29
  1 | Pavel  | Stěhule  |          1 |  34
(2 rows)

postgres=# SELECT * FROM Zamestnani;
 id | profese
----+----------
  1 | Vývojář
  2 | Dispečer
(2 rows)
```


Ukázky použití příkazu SELECT
První dotaz, sloučení tabulek

```
postgres=# SELECT jmeno, prijmeni, profese
postgres-#    FROM Osoby
postgres-#         JOIN
postgres-#         Zamestnani
postgres-#         ON Osoby.zamestnani = Zamestnani.id;
 jmeno  | prijmeni | profese
--------+----------+----------
 Pavel  | Stěhule  | Vývojář
 Zdeněk | Stěhule  | Dispečer
(2 rows)

postgres=# SELECT jmeno, prijmeni, profese
postgres-#    FROM Osoby
postgres-#         JOIN
postgres-#         Zamestnani
postgres-#         ON Osoby.zamestnani = Zamestnani.id
postgres-#   WHERE vek > 30;
 jmeno | prijmeni | profese
-------+----------+---------
 Pavel | Stěhule  | Vývojář
(1 row)
```

Ukázky použití příkazu SELECT
Význam predikátu při spojení tabulek

```
postgres=# SELECT jmeno, prijmeni, profese
postgres-#   FROM Osoby
postgres-#        JOIN
postgres-#        Zamestnani
postgres-#        ON true;
 jmeno  | prijmeni | profese
--------+----------+----------
 Pavel  | Stěhule  | Vývojář
 Zdeněk | Stěhule  | Vývojář
 Pavel  | Stěhule  | Dispečer
 Zdeněk | Stěhule  | Dispečer
(4 rows)
```

Díky chybnému predikátu spojení výsledek obsahuje řádky, které neodpovídají datům. SQL vám nijak nebrání udělat chybu použitím nesprávného predikátu.

Ukázky použití příkazu SELECT
Zastupitelnost spojení tabulek a poddotazů

```
-- spojení tabulek
SELECT jmeno, prijmeni
   FROM Osoby
        JOIN
        Zamestnani
        ON Osoby.zamestnani = Zamestnani.id
  WHERE profese = 'Vývojář';

-- použití tzv. poddotazů
SELECT jmeno, prijmeni
   FROM Osoby
  WHERE Zamestnani = (
                       SELECT id
                          FROM Zamestnani
                         WHERE profese = 'Vývojář'
                     );
```

Ukázky použití příkazu SELECT
Použití operátoru IN (je prvkem)

```
-- všechny vývojáře a dispečery
SELECT jmeno, prijmeni
   FROM Osoby
  WHERE Zamestnani IN (
                        SELECT id
                           FROM Zamestnani
                          WHERE profese IN ('Vývojář','Dispečer')
                       );

-- všechny nevývojáře
SELECT jmeno, prijmeni
   FROM Osoby
  WHERE Zamestnani NOT IN (
                            SELECT id
                               FROM Zamestnani
                              WHERE profese = 'Vývojář'
                           );
```


Ukázky použití příkazu SELECT
Zastupitelnost spojení tabulek a poddotazů

```
-- všechny vývojáře a dispečery
SELECT jmeno, prijmeni
   FROM Osoby
        JOIN
        Zamestnani
        ON Osoby.zamestnani = Zamestnani.id
  WHERE Zamestnani.profese = 'Vývojář'
     OR Zamestnani.profese = 'Dispečer';

-- všechny nevývojáře
SELECT jmeno, prijmeni
   FROM Osoby
        JOIN
        Zamestnani
        ON Osoby.zamestnani = Zamestnani.id
  WHERE Zamestnani.profese <> 'Vývojář';
```

Ukázky použití příkazu SELECT
Zastupitelnost IN za ANY, ALL

```
-- všechny vývojáře a dispečery
SELECT jmeno, prijmeni
   FROM Osoby
  WHERE zamestnani = ANY (
                          SELECT id
                             FROM Zamestnani
                           WHERE profese IN ('Vývojář','Dispečer')
                         );

-- všechny nevývojáře
SELECT jmeno, prijmeni
   FROM Osoby
  WHERE zamestnani <> ALL (
                           SELECT id
                              FROM Zamestnani
                            WHERE profese = 'Vývojář'
                           );
```


Ukázky použití příkazu SELECT
Správné použití predikátu ALL v poddotazů

```
-- nalezení nejstarší osoby
SELECT jmeno, prijmeni
   FROM Osoby
  WHERE vek >= ALL (
                     SELECT vek
                        FROM Osoby
                    );

-- optimální řešení nalezení nejstarší osoby
SELECT jmeno, prijmeni
   FROM Osoby
  WHERE vek = (
                SELECT MAX(vek)
                   FROM Osoby
               );
```

Ukázky použití příkazu SELECT
Variace příkazu SELECT (více sloupcový predikát)

```
-- nalezení nejstarší osoby podle oddělení
SELECT jmeno, prijmeni, vek, profese
   FROM Osoby
        JOIN
        Zamestnani
        ON Osoby.zamestnani = Zamestnani.id
  WHERE (zamestnani, vek) = ANY (
                                  SELECT zamestnani, MAX(vek)
                                     FROM Osoby
                                    GROUP BY zamestnani
                                 );
```


Ukázky použití příkazu SELECT
Variace příkazu SELECT (korelovaný poddotaz)

```
-- nalezení nejstarší osoby podle oddělení
SELECT jmeno, prijmeni, vek, profese
   FROM Osoby o1
        JOIN
        Zamestnani
        ON o1.zamestnani = Zamestnani.id
  WHERE vek = (
                SELECT MAX(vek)
                   FROM Osoby o2
                  WHERE o2.zamestnani = o1.zamestnani
               );
```

Ukázky použití příkazu SELECT
Variace příkazu SELECT (derivovaná tabulka)

```
-- nalezení nejstarší osoby podle oddělení
SELECT jmeno, prijmeni, o1.vek, profese
   FROM Osoby o1
        JOIN
        Zamestnani
        ON o1.zamestnani = Zamestnani.id
        JOIN
        (
         SELECT zamestnani, MAX(vek) AS vek
            FROM Osoby
           GROUP BY zamestnani
        ) dt
        ON o1.vek = dt.vek AND o1.zamestnani = dt.zamestnani;
```


Ukázky použití příkazu SELECT
Variace příkazu SELECT (ORDER BY LIMIT)

```
-- dohledání dvou nejstarších osob z každého oddělení
SELECT jmeno, prijmeni, vek, profese
   FROM Osoby o1
        JOIN
        Zamestnani
        ON o1.zamestnani = Zamestnani.id
  WHERE vek = ANY (
                   SELECT vek
                      FROM Osoby o2
                     WHERE o2.zamestnani = o1.zamestnani
                     ORDER BY vek DESC LIMIT 2
                  )
  ORDER BY o1.zamestnani, o1.vek DESC;
```

Ukázky použití příkazu SELECT
Zastupitelnost korelovaných poddotazů

```
-- zjisteni vyplacene castky celkem ke kazde osobe
-- narocne na zpracovani, pokud mozno nepouzivat
SELECT jmeno, prijmeni, (
                          SELECT SUM(castka)
                             FROM Mzdy
                            WHERE osoba_id = Osoby.id
                         ) AS celkem
   FROM Osoby;

-- zjisteni vyplacene castky celkem ke kazde osobe (JOIN)
SELECT jmeno, prijmeni, celkem
   FROM Osoby
        JOIN (
              SELECT osoba_id, SUM(castka) AS celkem
                 FROM Mzdy
                GROUP BY osoba_id
             ) dt
        ON Osoby.id = dt.osoba_id;
```


Ukázky použití příkazu SELECT
Náhrada NOT IN spojením

```
-- zjisteni osob, ktere jeste nedostali mzdu
SELECT jmeno, prijmeni
   FROM Osoby
  WHERE id NOT IN (
                    SELECT osoba_id
                       FROM Mzdy
                   );

-- zjisteni osob, ktere jeste nedostali mzdu
SELECT jmeno, prijmeni
   FROM Osoby
        LEFT JOIN
        Mzda
        ON Osoby.id = Mzdy.osoba_id
  WHERE Mzdy.osoba_id IS NULL;
```

Ukázky použití příkazu SELECT
Komentář

Proč variace příkazu SELECT?

- Příkaz SELECT může mít pro jednu úlohu řadu tvarů lišících se efektivitou zpracování. Hledáme takový tvar, který je co nejčitelnější a zároveň dostatečně rychlý (0.2s interaktivní dotazy).
- Pokud možno, nepoužívejte korelované subdotazy (proto tu nebyl zmíněn operátor EXISTS).
- V řadě případů je spojení tabulek rychlejší než použití množinového operátoru IN.
- Optimální tvar je závislý na zadání úlohy a skutečných datech. Vždy existují výjimky, pro které neplatí výše uvedená doporučení. Optimální tvar příkazu může být jiný v různých SQL systémech, a také se může lišit podle verze jednoho SQL systému.

Ukázky použití příkazu SELECT
Odstranění duplicit

SELECT DISTINCT

Klíčové slovo `DISTINCT` způsobí eliminaci duplicitních řádků z výsledné tabulky. DISTINCT se vždy musí používat s rozvahou. Duplicitní řádky ve výsledku dotazu mohou signalizovat chybně sestavený SQL dotaz.

```
-- zjisteni unikatnich prijmeni
SELECT DISTINCT prijmeni
   FROM Osoby;
```

Cvičení
Tabulky užívané v příkladech - tab. Lety

```
postgres=# SELECT * FROM Lety;
  let    | spolecnost | destinace | pocet cestujicich
---------+------------+-----------+-------------------
 OK251   | CSA        | New York  | 276
 LH438   | Lufthansa  | Stuttgart | 68
 OK012   | CSA        | Milano    | 37
 OK321   | CSA        | London    | 156
 AC906   | Air Canada | Torronto  | 116
 KL7621  | KLM        | Rotterdam | 75
 KL1245  | KLM        | Amsterdam | 130
(7 rows)
```


Cvičení
Tabulky užívané v příkladech - tab. Letadla

```
postgres=# SELECT * FROM Letadla;
   letadlo    | spolecnost | kapacita
--------------+------------+----------
 Boeing 717   | CSA        | 106
 Airbus A380  | KLM        | 555
 Airbus A350  | KLM        | 253
(3 rows)
```

Cvičení
Procvičování příkazu SELECT

Cvičení č.1

1. Jaké společnosti přepravují cestující?
2. Jaké páry letadel mohu vytvořit (bez ohledu na vlastníka) a jaká bude celková kapacita párů?

Cvičení
Procvičování příkazu SELECT WHERE

Cvičení č.2

1. Kterými lety cestuje více než 100 cestujících?
2. Kterými letadly by mohli letět cestující dané společnosti, pokud chceme mít naplněnost (počet cestujících / kapacita letadla) letadla alespoň 30%?
3. Do kterých destinací se dá letět Airbusem?
4. Cestující kterých letů se vejdou do libovolného letadla (bez ohledu na vlastníka)?
5. Vrať dvojice let-letadlo uspořádané vzestupně podle volného místa v letadle po obsazení cestujícími (uvažujeme pouze ta letadla, kam se cestující z letu vejdou)?
6. Jak se jmenuje největší letadlo?

Ukázky použití příkazu SELECT
Agregační funkce

Pokud se objeví agregační funkce v příkazu SELECT, tak výsledkem nejsou původní data, nýbrž odvozená data. Hodnoty agregačních funkcí se počítají na podmnožinách určených predikátem v klauzuli GROUP BY. Vstupní množinou je výsledek SQL dotazu, pokud by nedošlo k použití agregačních funkcí. Pokud chybí GROUP BY, vstupní množina dat se nedělí. Základní agregační funkce jsou: SUM, MIN, MAX, AVG a COUNT.

Pravidlo pro použití agregačních funkcích

Pokud se v dotazu objeví agregační funkce, pak platí, že atribut je parametrem agregační funkce nebo je zahrnut v GROUP BY.

GROUP BY může obsahovat název atributu, výraz nebo pořadové číslo atributu v seznamu zobrazovaných sloupců. Tím se zabrání opakovanému vyhodnocování složitějších výrazů (totéž u ORDER BY).

Ukázky použití příkazu SELECT
Agregační funkce COUNT

Tato funkce je specifická svými parametry. Pokud je parametrem symbol * pak je výsledkem pocet řádků. Pokud je parametrem název sloupce, pak je výsledkem počet ne NULL hodnot v tomto sloupci. Poslední možností je DISTINCT a název sloupce. Pak je výsledkem počet unikátních hodnot v daném sloupci.

```
SELECT COUNT(*)
   FROM ...  -- počet řádků

SELECT COUNT(prijmeni)
   FROM ...  -- počet zadaných hodnot v sloupci

SELECT COUNT(DISTINCT prijmeni)
   FROM ...  -- počet unikátních příjmení
```

Ukázky použití příkazu SELECT
Klauzule HAVING

Pro filtrování agregovaných dat slouží klauzule HAVING. Je obdobou klauzule WHERE (ta slouží k filtraci originálních dat). Predikát v klauzuli HAVING musí obsahovat agregovanou hodnotu.

Cvičení
Procvičování příkazu agregačních funkcí

Cvičení č.3

1. Kolik společností přepravuje cestující?
2. Kolika lety se přepravují cestující?
3. Kolik je letadel, jakou mají maximální, minimální, průměrnou a celkovou kapacitu?
4. Pro společnosti vlastnící letadla vrať součet všech cestujících a kapacit letadel.
5. Kteří jsou dva největší dopravci (podle součtu jejich cestujích)?
6. Jakou (kladnou) přepravní kapacitu mají jednotlivé dopravní společnosti?
7. Kterým společnostem se vejdou najednou všichni cestující do letadel (bez ohledu na destinaci, tj. v jednom letadle mohou být cestující více letů)?

Ukázky použití příkazu SELECT
Sloučení tabulek klauzulí UNION

Obsah dvou relací (tabulek, výsledků SQL dotazů) lze spojit za sebe klauzulí UNION. Musí platit, že spojované tabulky mají stejný počet sloupců, a datové typy jsou konvertibilní.

```
postgres=# SELECT *
postgres-#    FROM (VALUES(1,'Pavel','Stěhule')) v1
postgres-# UNION
postgres-# SELECT *
postgres-#    FROM (VALUES(2,'Zdeněk','Stěhule')) v2;
 column1 | column2 | column3
---------+---------+---------
       1 | Pavel   | Stěhule
       2 | Zdeněk  | Stěhule
(2 rows)
```


Konstanta NULL
Univerzální hodnota pro neznámé

Hodnotu NULL používáme, pokud nemáme žádnou rozumnou hodnotu, kterou bychom mohli dosadit do záznamu. Vyjadřujeme tak svou neznalost hodnoty atributu. Výsledkem všech výrazů, kde se NULL objeví je také NULL. Výjimkou jsou agregační funkce a operátory *IS* a *IS DISTINCT FROM*.
Příkladem může být například použití NULL na místě rodného čísla u osob, které rodné číslo nemají vydané (nezletilí, cizinci). Dalším příkladem jsou například audit sloupce modifikováno, zrušeno. Dokud nedošlo k první modifikaci nebo ke zrušení, tak přirozeným obsahem je NULL. V řadě případů je NULL vhodnější variantou než mystické konstanty (např. 0::date). Dynamickou substituci NULL řeší funkce *COALESCE*.

```
SELECT COALESCE(jmeno || ' ' || prijmeni,
                prijmeni, 'nezadano')
   FROM ...
```

Cvičení
Procvičování testu na NULL

Cvičení č.4

1. Které lety nemohou být uskutečněny (protože společnost nevlastní vhodné/žádné letadlo)?

Příkaz INSERT
Přidávání nových řádků do tabulky

Používá se pro vkládání nových záznamů do tabulky (vždy do jedné). Vložit můžeme jeden řádek (klasický INSERT), více řádků (vícenásobný INSERT) nebo výsledek dotazu (INSERT SELECT). Příkazy INSERT, UPDATE, DELETE podporují klauzuli RETURNING. Pokud není použita, pak výsledkem SQL příkazu je počet ovlivněných řádků. Pokud použita je, pak výsledkem je tabulka obsahující nové, upravené nebo odstraněné řádky.

Poznámka

Pro plnění tabulek ještě existuje nestandardní příkaz COPY. Jelikož obchází aparát SQL je mnohem rychlejší a méně náročný na paměť. Tento příkaz by se měl preferovat, pokud dochází k přidání více než deset tisíc nových záznamů. Provádění příkazů značně urychlíte, pokud všechny příkazy soustředíte do jedné transakce.

Pro vložení defaultní hodnoty vyřaďte sloupec ze seznamu vkládaných atributů nebo použijte klíčové slovo DEFAULT.

Příkaz INSERT
Syntaxe

```
INSERT INTO tabulka [ ( sloupec [, ...] ) ]
    { DEFAULT VALUES
    | VALUES ( { vyraz | DEFAULT } [, ...] ) [, ...]
    | query }
  [ RETURNING * | vyraz [ AS vystupni_sloupec ] [, ...] ]
```


Příkaz UPDATE
Modifikace hodnot v tabulce

Příkaz UPDATE slouží k úpravě existujících záznamů. V případě použití nestandardní klauzule FROM lze záznamy upravovat na základě hodnot z připojených tabulek. Jedním příkazem UPDATE lze modifikovat pouze jednu tabulku.

```
UPDATE [ ONLY ] tabulka [ [ AS ] alias ]
    SET { sloupec = { vyraz | DEFAULT } |
          ( sloupec [, ...] ) = ( { vyraz | DEFAULT } [, ...] ) }
             [, ...]
    [ FROM seznam_tabulek ]
    [ WHERE condition | WHERE CURRENT OF jmeno_kurzoru ]
    [ RETURNING * | vyraz [ AS vystupni_sloupec ] [, ...] ]
```

Příkaz UPDATE
Příklad

```
CREATE TABLE Test(a integer, b integer);

CREATE TABLE Nove_hodnoty(a integer, b integer);

-- aktualizace tabulky test na zaklade obsahu tabulky nove_hodnoty
UPDATE Test t SET b = n.b
   FROM Nove_hodnoty n
  WHERE t.a = n.a RETURNING t.a, t.b;
```


Příkaz DELETE
Odstranění záznamů

Příkazem DELETE odstraníme všechny záznamy, které vyhovují zadanému predikátu. Pokud není predikát zadaný, odstraní se všechny záznamy. Nestandardní klauzule USING má stejný význam jako klauzule FROM v příkazu UPDATE.

```
DELETE FROM [ ONLY ] table [ [ AS ] alias ]
    [ USING usinglist ]
    [ WHERE condition ]
    [ RETURNING * | output_expression [ AS output_name ] [, ...] ]
```

Poznámka

Pokud potřebujete rychle odstranit veškeré záznamy v tabulce, použijte příkaz TRUNCATE. Jelikož při něm nedochází ke spuštění triggerů a rušení řádky po řádce, je mnohem rychlejší než příkaz DELETE.

Příkaz DELETE
Příklad

```
-- nestandardní forma (pouziti USING)
DELETE FROM Osoby
  USING Zamestnani
  WHERE Osoby.zamestnani = Zamestnani.id
    AND Zamestnani.profese = 'Vývojář';

-- standardni zpusob (pouziti poddotazů)
DELETE FROM Osoby
  WHERE zamestnani IN (
                        SELECT id
                           FROM Zemestnani
                          WHERE profese = 'Vývojář'
                       );
```


SQL funkce a operátory
Pro řetězce

```
SELECT 'abc' || 'def'; --> 'abcdef', spojeni retezcu
SELECT 'Stehul' LIKE 'Stehul%'; --> true, test na shodu ze vzorem
SELECT 'char_length('abcde'); --> 4, pocet znaku v retezci
SELECT lower('ABCDE'); --> abcde, prevod na mala pismena
SELECT upper('abcde'); --> ABCDE, prevod na velka pismena
SELECT position('x' IN 'abcdxabcd'); --> 5, pozice podretezce
SELECT substring('abcdxabcd' FROM 5 FOR 1); --> x, vraci podretezec
SELECT trim('  aaa  '); --> 'aaa', odstrani krajni mezery

--formatovany vystup
SELECT to_char(current_timestamp, 'HH12:MI:SS'); --> 12:28:09
```

SQL funkce a operátory
Pro čísla

```
SELECT random(); --> 0.626262, nahodnehe cislo z intervalu (0,1)
SELECT sign(-292); --> -1, pro kladne vraci 1, pro nulu 0
SELECT trunc(8.7); --> 8, cela cast cisla
SELECT round(8.7); --> 9, zaokrouhlovani
```


SQL funkce a operátory
Pro datum

```
SELECT date '2007-10-10' + 1; --> 11.10.2007 pricteni n dnu
SELECT date '2007-10-10' + interval '1month'; --> 11.11.2007
SELECT date '2007-10-10' - date '2007-10-08'; --> 2
SELECT age('2007-10-10', '2007-10-08'); --> '2 days'
SELECT 'epoch'::date;     --> 1970-01-01
SELECT 'now'::date;       --> 2007-06-20
SELECT 'tomorrow'::date;  --> 2007-06-21
SELECT 'yesterday'::date; --> 2007-06-19
SELECT EXTRACT(dow FROM CURRENT_DATE); --> 3 den v tydnu
SELECT EXTRACT(year FROM CURRENT_DATE); --> 2007
SELECT date_trunc('month', date '2007-10-10'); --> 2007-10-01
SELECT justify_days('200 days') -->  '6 mons 20 days'

-- formatovy vstup a vystup
SELECT to_date('05 Dec 2000', 'DD Mon YYYY'); --> 2007-12-05
SELECT to_char(CURRENT_DATE,'day'); --> wednesday
```

SQL funkce a operátory
Pro čas

```
-- pricteni intervalu vysledek  2007-10-10 13:12:00
SELECT timestamp '2007-10-10 12:00:00' + interval '1hour 12 min';

-- praktictejsi pricteni intevalu z hlediska generovani SQL
-- lze resit jako klasicke SQL bez nutnosti dynamickeho SQL
SELECT timestamp '2007-10-10 12:00:00' + 1*interval '1hour'
                                       + 12*interval '1min';
```


SQL funkce a operátory
Pro pole

```
SELECT ARRAY[1,2,3]; -- konstruktor pole
SELECT ARRAY(SELECT a FROM Foo); -- konstruktor pole s poddotazem
SELECT '{1,2,3}'::int[]; -- vytvoreni pole pretypovanim
SELECT ARRAY[1,2,3] || 1; --> [1,2,3,1]
SELECT ARRAY[1,2,3] || ARRAY[1,2,3]; --> [1,2,3,1,2,3]
SELECT array_lower(ARRAY[1,2],1); --> 1, dolni mez pole
SELECT array_upper(ARRAY[1,2],1); --> 2, horni mez pole
SELECT array_to_string(ARRAY[1,2,3],':'); --> '1:2:3'
SELECT string_to_array('1:2:3',':'); --> [1,2,3]

-- predikaty
SELECT 1 = ANY (ARRAY[1,2,3]); --> t, test na clenstvi
SELECT 6 > ALL (ARRAY[1,2,3]); --> t, 6 je vetsi nez vsechny prvky
```

Řešení příkladů
Cvičení č.1

```
- Jaké společnosti přepravují cestující?
SELECT DISTINCT spolecnost
   FROM Lety;

-- Jaké páry letadel mohu vytvořit (bez ohledu na vlastníka) a
-- jaká bude celková kapacita párů?

-- pokud nam nevadi opakujici se pary
SELECT l1.letadlo, l2.letadlo, l1.kapacita + l2.kapacita
   FROM Letadla l1
        JOIN
        Letadla l2
        ON l1.letadlo <> l2.letadlo;

-- pokud chceme zabranit opakovani
SELECT l1.letadlo, l2.letadlo, l1.kapacita + l2.kapacita
   FROM Letadla l1
        JOIN
        Letadla l2
        ON l1.letadlo < l2.letadlo;
```


Řešení příkladů
Cvičení č.2

```
-- Kterými lety cestuje více než 100 cestujících?

SELECT let, "pocet cestujicich"
   FROM Lety
  WHERE "pocet cestujicich" > 100;

-- Kterými letadly by mohli letět cestující dané společnosti, pokud
-- chceme mít naplněnost (počet cestujících / kapacita letadla)
-- letadla alespoň 30%?

SELECT lt.let, ld.letadlo,
       round(100.0 * lt."pocet cestujicich" / ld.kapacita, 2) AS Naplnenost
   FROM Lety lt
        JOIN
        Letadla ld
        ON    lt.spolecnost = ld.spolecnost
          AND "pocet cestujicich" < kapacita
          AND round(100.0 * lt."pocet cestujicich" / ld.kapacita, 2) > 30.0;
```

Řešení příkladů
Cvičení č.2

```
-- Do kterých destinací se dá letět Airbusem?

SELECT destinace
   FROM Lety
  WHERE spolecnost IN (
                        SELECT spolecnost
                           FROM Letadla
                          WHERE letadlo LIKE 'Airbus %'
                       );
```


Řešení příkladů
Cvičení č.2

```
-- Cestující kterých letů se vejdou do libovolného letadla (bez
-- ohledu na vlastníka)(kapacita každého letadla je větší)?

SELECT *
   FROM Lety
  WHERE "pocet cestujicich" < ALL(
                                  SELECT kapacita
                                     FROM letadla
                                 );

-- Vrať dvojice let-letadlo uspořádané vzestupně podle volného
-- místa v letadle po obsazení cestujícími (uvažujeme pouze ta
-- letadla, kam se cestující z letu vejdou)?

SELECT lt.let, ld.letadlo, ld.kapacita - lt."pocet cestujicich" as "volnych mist"
   FROM Lety lt
        JOIN Letadla ld
        ON    lt.spolecnost = ld.spolecnost
          AND "pocet cestujicich" < "kapacita"
  ORDER BY 3;
```

Řešení příkladů
Cvičení č.2

```
-- Jak se jmenuje největší letadlo?

SELECT letadlo
   FROM Letadla
  ORDER BY kapacita DESC
  LIMIT 1;

SELECT letadlo
   FROM Letadla
  WHERE kapacita >= ALL (
                          SELECT kapacita
                             FROM Letadla
                        );

SELECT letadlo
   FROM Letadla
  WHERE kapacita = (
                     SELECT MAX(kapacita)
                        FROM Letadla
                   );
```


Řešení příkladů
Cvičení č.3

```
-- Kolik společností přepravuje cestující?

SELECT COUNT(DISTINCT spolecnost)
   FROM Lety;

-- Kolika lety se přepravují cestující?

SELECT COUNT(*)
   FROM Lety;

-- Kolik je letadel, jakou mají maximální, minimální, průměrnou
-- a celkovou kapacitu?

SELECT COUNT(*), MAX(kapacita), MIN(kapacita),
       ROUND(AVG(kapacita),2), SUM(kapacita)
   FROM Letadla ;
```

Řešení příkladů
Cvičení č.3

```
-- Pro společnosti vlastnící letadla vrať součet všech
-- cestujících a kapacit letadel.

SELECT pc.spolecnost, "pocet cestujicich", kapacita
   FROM (
         SELECT spolecnost,
                SUM("pocet cestujicich") AS "pocet cestujicich"
            FROM Lety
           GROUP BY spolecnost
        ) pc
        JOIN
        (
         SELECT spolecnost, SUM(kapacita) AS kapacita
            FROM Letadla
           GROUP BY spolecnost
        ) k
        ON pc.spolecnost = k.spolecnost;
```


Řešení příkladů
Cvičení č.3

```
-- Kteří jsou dva největší dopravci (podle součtu jejich cestujích)?

SELECT spolecnost
   FROM Lety
  GROUP BY spolecnost
  ORDER BY SUM("pocet cestujicich") DESC
  LIMIT 2;

-- Jakou (kladnou) přepravní kapacitu mají jednotlivé dopravní
-- společnosti?

SELECT spolecnost, SUM(kapacita)
   FROM Letadla
  GROUP BY spolecnost;
```

Řešení příkladů
Cvičení č.3

```
-- Kterým společnostem se vejdou najednou všichni cestující do
-- letadel (bez ohledu na destinaci, tj. v jednom letadle mohou být
-- cestující více letů)? pozn. ukazka korelovaneho dotazu

SELECT spolecnost, SUM("pocet cestujicich")
   FROM Lety lt
  GROUP BY spolecnost
  HAVING SUM("pocet cestujicich") < (
                                     SELECT SUM(kapacita)
                                        FROM Letadla ld
                                       WHERE lt.spolecnost
                                             = ld.spolecnost
                                    );
```


Řešení příkladů
Cvičení č.4

```
-- Které lety nemohou být uskutečněny (protože společnost
-- nevlastní vhodné/žádné letadlo)?

SELECT lt.let, lt.destinace, lt.spolecnost
   FROM Lety lt
        LEFT JOIN
        Letadla ld
        ON lt.spolecnost = ld.spolecnost
           AND lt.''pocet cestujicich''<= ld.kapacita
  WHERE ld.letadlo IS NULL;
```

Řešení příkladů
Cvičení č.4

```
-- Variace predchoziho SQL prikazu
-- Poddotaz (mene efektivni nez LEFT JOIN)

SELECT *
   FROM Lety
  WHERE let NOT IN (
                     SELECT let
                        FROM Lety
                             JOIN
                             Letadla
                             ON lety.spolecnost = letadla.spolecnost
                                AND lety."pocet cestujicich" < letadla.kapacita
                    );

-- ukazka korelovaneho dotazu (nejhorsi varianta)
SELECT *
   FROM Lety lt
  WHERE NOT EXISTS(
                    SELECT *
                       FROM Letadla
                      WHERE lt.spolecnost = spolecnost
                        AND lt."pocet cestujicich" < kapacita
                   );
```


Indexy
Akcelerace vyhledání údajů v databázi

Urychlují výběr (vyhledávání) - udržují seřazená data z vybraných sloupců (na seřazená data lze aplikovat algoritmus vyhledávání metodou půlení intervalu). Bez indexů je vyhledávání v tabulkách nad 10000 řádků pomalé.

- jednoduché indexy obsahují pouze data z jednoho sloupce → úspora paměti, rychlejší přístup,
- indexy lze sestavit nad vybranou množinou → úspora paměti, rychlejší přístup,
- klasický B-tree index urychluje operace menší, větší, rovno,
- GiST indexy slouží pro urychlení operací obsahuje, neobsahuje, ...,
- vyhledávání pomocí LIKE vždy na úzkých sloupcích (max. 100-200 znaků). U delších řetězců používat FULLTEXT a FT operace (vyhledávání po slovech).

```
postgres=# \h CREATE INDEX
CREATE [ UNIQUE ] INDEX name ON table [ USING method ]
    (  column | ( expression )
    [ WHERE predicate ]
```

Indexy
Poznámky

- Indexy se automaticky vytváří s primárními klíči,
- V případě větších tabulek (nad 1000 řádek) má smysl vytvářet index i nad sloupci obsahující cizí klíče.
- Indexy zásadně vytváříme až ve chvíli, kdy víme, že jsou nezbytné.

Důležité

Nezapomínat na aktualizaci statistik (v PostgreSQL příkaz `ANALYZE`)

```
postgres=# EXPLAIN SELECT * FROM f WHERE a BETWEEN 1 AND 100;
                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on f  (cost=13.31..40.72 rows=494 width=4)
   Recheck Cond: ((a >= 1) AND (a <= 100))
   ->  Bitmap Index Scan on fa  (cost=0.00..13.19 rows=494 width=0)
         Index Cond: ((a >= 1) AND (a <= 100))
(4 rows)
```


Pohledy
fiktivní tabulky

Slouží k vytváření fiktivních tabulek zprostředkovávajících obsah, který je díky normalizaci uložený v několika tabulkách. Umožňuje zpřístupnit data určité skupině uživatelů, kterým nechceme zpřístupnit samotné tabulky (příkazy `GRANT` a `REVOKE`).

Poznámka

Až na malé výjimky pohledy nemají vliv na rychlost provádění dotazu. Definice pohledu se rozbalí v SQL příkazu, který obsahuje dotaz na pohled. Rozumným návrhem mohou přispět "user-friendly" databáze a i větší čitelnosti aplikací nad databází vytvářených.

```
postgres=# \h CREATE VIEW
Command:     CREATE VIEW
Description: define a new view
Syntax:
CREATE [ OR REPLACE ] VIEW name [ ( column_name [, ...] ) ]
    AS query
```

Triggery
jinak také spouště

Trigger je příkaz, který se spustí, pokud dojde ke změně obsahu tabulky. V případě PostgreSQL se jedná vždy o uloženou proceduru. Triggery používáme k zajištění konzistence údajů v tabulkách, které nedokážeme zajistit jinak (podmínkami referenční a doménová integrity).

```
01 CREATE OR REPLACE FUNCTION audit()
02 RETURNS TRIGGER AS $$
03 BEGIN
04   -- doslo skutecne ke zmene
05   IF ROW(new.*) IS DISTINCT FROM ROW(old.*) THEN
06     INSERT INTO audit VALUES(old.*);
07     new.changed := CURRENT_TIMESTAMP;
08     new.changed_by := CURRENT_USER;
09     RETURN new;
10   ELSE
11     RETURN NULL; -- ignoruj zmenu
12   END IF;
13 END $$ LANGUAGE plpgsql;
```


Triggery
jinak také spouště

```
postgres=# \h CREATE TRIGGER
Command:     CREATE TRIGGER
Description: define a new trigger
Syntax:
CREATE TRIGGER name  BEFORE | AFTER   event [ OR ... ]
    ON table [ FOR [ EACH ]  ROW | STATEMENT  ]
    EXECUTE PROCEDURE funcname ( arguments )
```

Poznámka

Triggery ovlivňují rychlost provádění příkazů INSERT, UPDATE, DELETE a nemají vliv na provádění příkazu SELECT. Při hromadných importech a změnách je lze proto dočasně blokovat (pouze superuser). Jedná se o poslední instanci, nic je nedokáže obejít. Neměly by obsahovat pomalé SQL příkazy.

Spojení tabulek je podmnožinou kartézského součinu
kartézský součin

```
postgres=# SELECT *
              FROM (VALUES(10),(20)) a(a)
                   JOIN
                   (VALUES(20),(30)) b(b)
                   ON true;
 a  | b
----+----
 10 | 20
 10 | 30
 20 | 20
 20 | 30
(4 rows)
```


Spojení tabulek je podmnožinou kartézského součinu
Podmnožina kartézského součinu

```
postgres=# SELECT *
              FROM (VALUES(10),(20)) a(a)
                   JOIN
                   (VALUES(20),(30)) b(b)
                   ON a.a = b.b;
 a  | b
----+----
 20 | 20
(4 rows)
```

Transakce
Zajištění konzistence změn dat

Upozornění

Jakákoliv databázová operace může selhat!. A to ať z důvodu pokusu o provedení nepovolené operace, nebo z důvodu výpadku systému (sw, hw). Pokud k tomu dojde, databázový systém musí dokázat vrátit obsah databáze do stavu před zahájením nedokončené operace (nevíme, kolik se stihlo z operace provést, a co se nestihlo dokončit).

- Každý SQL příkaz je spouštěn pod implicitní transakcí. Je zaručené, že bude dokončen úspěšně a nebo neprovede žádné změny dat (v PostgreSQL).
- V případě, že k modifikaci dat použijeme více SQL příkazů, musíme použít explicitní transakci. Ta je zahájena příkazem `BEGIN` a ukončena příkazem `COMMIT`.
- Nepotvrzené změny dat nejsou viditelné ostatním uživatelům. Pro ostatní uživatele je obsah databáze stále konzistentní.

Transakce
Zajištění konzistence změn dat

```
01  CREATE TABLE ucet (
02    uid integer,
03    ucet numeric(10,2)
04           CHECK (ucet > -kontokorent),
05    kontokorent numeric(10,2));
06
07  -- bezpecny presun z uctu 10 na ucet 11 castka 10 000
08  BEGIN;
09    UPDATE ucet SET ucet=ucet+10000 WHERE uid = 11;
10    UPDATE ucet SET ucet=ucet-10000 WHERE uid = 10;
11  COMMIT;
12  -- pripadne ROLLBACK, pokud na uctu nebyla dostatecne vysoka
13  -- castka
```

Konstrukce CASE
Konverze hodnot

Proveďte agregaci dat po 15 minutových intervalech

```
...
GROUP BY
      date_trunc('hour', vlozeno)
    + CASE WHEN EXTRACT(minutes FROM vlozeno) BETWEEN  0 AND 14
           THEN interval  '0 min'
           WHEN EXTRACT(minutes FROM vlozeno) BETWEEN 15 AND 29
           THEN interval '15 min'
           WHEN EXTRACT(minutes FROM vlozeno) BETWEEN 30 AND 44
           THEN interval '30 min'
           ELSE interval '45 min' END;
```

Díky konstrukci CASE můžeme realizovat určité složitější úlohy, aniž bychom museli opouštět SQL.

Konstrukce CASE
Syntaxe

```
-- jednoducha forma
CASE výraz
     WHEN hodnota1 THEN výsledek1
     WHEN hodnota2 THEN výsledek2
     ELSE výsledek_v_případě_že_hodnota_nebyla_nalezena END

-- vyhledavaci forma
CASE
     WHEN logický_výraz1 THEN výsledek1
     WHEN logický_výraz2 THEN výsledek2
     ELSE výsledek_v_případě_že_hodnota_nebyla_nalezena END
```

Generování testovacích dat
Náhodná čísla, náhodné řetězce

```
-- nahodne cislo z intervalu (0, n)
SELECT random()*n;

-- nahodne cele cislo z intervalu (0, n-1)
SELECT trunc(random()*n);

-- generovani posloupnosti nahodnych retezcu
SELECT trim(array_to_string(
     ARRAY(
        SELECT chr(CASE WHEN random() > 0.1
                        THEN (trunc(random()*25)+65)::integer
                        ELSE 32 END)
           FROM generate_series(1,(random()*20*sign(i)+5)::int)),
    ''))
   FROM generate_series(1,3) g(i);
```


Generování testovacích dat
Naplnění tabulky obsahující pětiprvkové celočíselné pole

```
INSERT INTO test_tabulka
       SELECT ARRAY(
                    SELECT trunc(random()*10*sign(i))::integer)
                       FROM generate_series(1,5)
                   )
          FROM generate_series(1,10000) g(i);
```

Poznámka

SQL příkazy je nutné testovat nad databází naplněnou testovacími daty, které odpovídají alespoň dvou-tříletému provozu databáze. Po naplnění nezapomenout na příkaz ANALYZE, kterým aktualizujeme statistiky

Vlastní funkce
Univerzální řazení pole

```
CREATE OR REPLACE FUNCTION array_sort(anyarray)
RETURNS anyarray AS
$$
  SELECT ARRAY(SELECT $1[i]
                  FROM generate_series(array_lower($1,1),
                                       array_upper($1,1)) g(i)
                 ORDER BY 1
               )
$$ LANGUAGE SQL;
```