UNIVERZITA J. E. PURKYNĚ V ÚSTÍ NAD LABEM
Přírodovědecká fakulta

RELAČNÍ DATABÁZOVÉ SYSTÉMY

Jiří Fišer

Ústí nad Labem 2020

Předmět:	Relační databázové systémy
Studijní program:	Aplikovaná informatika
Klíčová slova:	SQL, PostgreSQL
Anotace:	Rozšiřující kurz SQL a relačních databázových systémů zaměřený na moderní rysy SQL a správu databází

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

Autor: Jiří Fišer

OBSAH

PřF

1 ANALYTICKÉ FUNKCE

- analytické funkce (window function) umožňují doplňovat běžné řádky o souhrnné informace získané z ostatních řádků (se stejným klíčem).
- používají se v sekci SELECT (na rozdíl od běžných funkcí však pracují nad více řádky)
- podobají se agregacím pomocí GROUP BY a nevedou však k redukci řádků (tj. nevytvářejí jen souhrnné řádky)
- některé lze emulovat pomocí vnořených dotazů uvnitř sekcí SELECT (neefektivní) resp.pomocí samospojení (self-join).

Vytvořte si databázi Mordorský maraton.

```
CREATE TABLE mormar (
    scislo integer NOT NULL PRIMARY KEY,
    zavodnik character varying(128) NOT NULL,
    kategorie integer,
    cas interval,
    dsq boolean DEFAULT false NOT NULL,
    CONSTRAINT mormar_cas_check CHECK ((cas > '00:00:00'::interval))
);

CREATE kategorie (
    id integer NOT NULL PRIMARY KEY,
    jmeno character varying(128) NOT NULL
);

INSERT INTO mormar VALUES (3, 'Azog', 3, '03:56:11', false);
INSERT INTO mormar VALUES (4, 'Gandalf', 4, '01:34:23', false);
INSERT INTO mormar VALUES (5, 'Saruman', 4, '01:56:00', false);
INSERT INTO mormar VALUES (6, 'Sauron', 4, '00:02:00', true);
INSERT INTO mormar VALUES (7, 'Gimli', 2, '2 days 03:14:00', true);
INSERT INTO mormar VALUES (8, 'Aragorn', 1, '02:42:42', false);
INSERT INTO mormar VALUES (9, 'Eowyn', 1, '02:45:12', false);
INSERT INTO mormar VALUES (2, 'Samvěd', 5, '04:12:00', false);
INSERT INTO mormar VALUES (10, 'Bilbo', 5, '04:11:59', false);

INSERT INTO kategorie VALUES (0, 'elf');
INSERT INTO kategorie VALUES (1, 'člověk');
INSERT INTO kategorie VALUES (2, 'trpaslík');
INSERT INTO kategorie VALUES (3, 'skřet');
INSERT INTO kategorie VALUES (4, 'maia');
INSERT INTO kategorie VALUES (5, 'hobit');
```

Analytické funkce umožňují vytvářet tabulky, které obsahují původní řádky (selektované či projektované) doplněné o agregované informace z jiných řádků.

Základní syntaxe analytických výrazů (používají se v části SELECT)

```
agregační_funkce OVER specifikace_skupiny
```

Agregační funkce jsou stejně jako v případě seskupování, navíc se používají funkce RANK, LAG a LEAD.

Přípustné specifikace skupiny řádků (k níž se vztahuje agregační funkce).

`()`

skupinou jsou všechny řádky projektované databáze

`(ORDER BY specifikace)`

skupinou jsou všechny řádky databáze setříděné podle specifikace

`(PARTITION BY výraz)`

skupinou jsou řádky, pro něž výraz vrací stejnou hodnotu jako u aktuálního řádku

`(PARTITION BY výraz ORDER BY specifikace)`

skupinou jsou řádky, pro něž výraz vrací stejnou hodnotu jako u aktuálního řádku, setříděné podle specifikace

Prostudujte dokumentaci na https://www.postgresql.org/docs/9.1/tutorial-window.html

ÚKOLY

1. Vytvořte dotaz, který vypíše u každého účastníka:
   - *kdo doběhl těsně před ním*
   - *kdo doběhl těsně před ním ve stejné kategorii*
   - *kolik ztratil na vítěze*
   - *kolik ztratil na vítěze ve své kategorii*
   - *kolikátý doběhl*
   - *kolikátý doběhl ve své kategorii*
   - *kolik závodníků doběhlo za ním*

2 OPTIMALIZACE DOTAZŮ

Při optimalizaci dotazů se musí zohledňovat velikost zpracovávaných tabulek (včetně dočasných tabulek vzniklých během provádění komplexnějších dotazů)

- velikost tabulky je primárně dána počtem řádků (i když rozsah sloupců může také hrát roli, je však v praxi omezený)

microtabulky = tabulky s jedním nebo několika málo sloupci

- optimalizace je kontraproduktivní

mezotabulky = rozsah desítek až tisíců řádku (pomocné struktury se do jednoho diskového bloku resp. stránky paměti)

- bitmap scan, hash join

macrotabulky

- index scan, merge join

2.1 Prováděcí plán

Rozsah průběžně vytvářených tabulek lze zjistit jen obtížně.

výjimka: agregační funkce, dotazy se sekcí LIMIT

proto se běžně jen odhaduje:

- z velikosti fyzických tabulek
- z empiricky zjištěných rozdělení pravděpodobnosti
- z výsledků předchozích obdobných dotazů

Na základě těchto údajů se stanovuje prováděcí plán

Dobré odhady vyžadují průběžnou analýzu data a provádění reálně využívaných dotazů (tzv. postupná akomodace).

Jak zjistit jaký plán byl použit: EXPLAIN [ANALYZE] dotaz

Vypisuje detailní plán s uvedením použitých mechanismů, odhadovanou cenou a počtem řádků tabulek

ANALYZE = plán je vykonán, údaje o reálném provedení jsou doplněny do výstupu pro srovnání

ÚKOLY

Diskutujte s pomocí dokumentace https://www.postgresql.org/docs/9.4/using-explain.html následující výpis plánu:

```
HashAggregate  (cost=39.53..39.53 rows=1 width=8) (actual time=0.661..0.672
rows=7 loops=1)

   ->  Index Scan using test_pkey on test  (cost=0.00..32.97 rows=1311 width=8)
```

```
(actual time=0.050..0.395 rows=99 loops=1)

         Index Cond: ((id > $1) AND (id < $2))

 Total runtime: 0.851 ms

(4 rows)
```

2.2 Indexy

Indexy jsou pomocné datové struktury na disku, které umožňují výrazně urychlit některé operace (vyhledávání, třídění) u mezotabulek a makrotabulek.

Sami však vyžadují jistou režii pro své uložení a správu (pokud nejsou nezbytné, neměly by být vytvářeny). Indexy se musí měnit při každém vložení řádku v tabulce pro níž jsou vytvořeny.

```
CREATE INDEX jméno_indexu ON tabulka (výraz);
```

Prostudujte https://www.postgresql.org/docs/9.1/indexes-intro.html.

Nejjednodušší je vytvoření indexu nad jedním sloupcem (výrazem je jméno sloupce). Indexy optimalizují selekci (konstrukce typu WHERE sloupec = hodnota) a spojení.

2.2.1 Typy indexů

b-tree — založený na b-tree. Využitelný pro všechny relační operace a po určité konfiguraci i pro vzory se shodou na počátku řetezců.

hash — založený na hashovací tabulce. Vhodný jen pro testování shody. V současnosti není u PostgreSQL doporučován (není zcela up to date)

složené indexy (GIN, GIST) -- využívány pro urychlení komplexních relací nad prostorovými daty a pro fulltext

ÚKOLY

Podívejte se na výhody b-tree na Wikipedii (https://cs.wikipedia.org/wiki/B-strom) a porovnejte s využitím hashovací tabulky.

2.2.2 Vícesloupcové a vypočítané indexy

PostgreSQL podporuje i indexy nad vypočítanými hodnotami a vícesloupcové indexy.

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

u vícesloupcového indexu by měl být nejvíce rozlišující sloupec uveden jako první (tj. např. příjmení před jménem a to před pohlavím)

Index nad vypočítanou hodnotou lze vytvořit nad libovolným řádkovým výrazem:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

nebo

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

2.2.3 Částečné indexy

Nalezení rovnováhy mezi režií přípravy a režií provedení mohou usnadnit částečné indexy, které indexují jen určitý rozsah hodnot:

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
     client_ip < inet '192.168.100.255');
```

Jiný praktický příklad:

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
     WHERE billed is not true;
```

ÚKOLY

1. Jaký rozsah mají IP adresy indexované v prvním příkladě? Co tento rozsah pravděpodobně znamená.

2. Použije se či nepoužije index pro následující dotaz?

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

3. Diskutujte nasazení posledního praktického příkladu (kdy se vyplatí nasadit tento částečný index).

3 HIERARCHICKÉ DATOVÉ STRUKTURY

SQL není primárně navrženo pro representaci datových struktur, lze je však relativně snadno implementovat pomocí sebe-odkazujících tabulek, kde prvky odkazují rodičovský prvek (mechanismem cizích klíčů)

Vytvořte následující hierarchickou tabulku administrativních jednotek v ČR (můžete doplnit svá oblíbená místa a jednotky):

```
CREATE TABLE a_jednotky
(
  id serial NOT NULL,
  name text NOT NULL,
  parent integer,
  CONSTRAINT a_jednotky_pkey PRIMARY KEY (id),
  CONSTRAINT a_jednotky_parent_fkey
      FOREIGN KEY (parent)
      REFERENCES a_jednotky (id)
      ON UPDATE NO ACTION ON DELETE CASCADE
```

id	name	parent
1	ČR	
2	Ústecký kraj	1
3	Karlovarský kraj	1
4	okres Ústí nad Labem	2
5	okres Litoměřice	2
6	okres Karlovy Vary	3
7	Ústí nad Labem	4
8	Chlumec	4
9	Dobříň	5
10	Horní Blatná	6
11	Bukov	7
12	Klíše	7

```
)
```

Dotazy s fixní úrovní odkazů = tj. např. nalezení sourozenců, dětských prvků atd. jsou v SQL snadné:

```
SELECT p.name as "nadřízená jednotka", c.name as "podřízená jednotka"
  FROM a_jednotky AS p JOIN a_jednotky AS c ON p.id = c.parent;
```

ÚKOLY

Vytvořte dotaz, který pro každou administrativní jednotku zobrazí její sesterské jednotky tj. jednotky, které mají společnou nadřízenou jednotku.

3.1 Rekurzivní dotazy

- problém však přinášely dotazy, které potřebovali pracovat s proměnlivou resp. neomezenou (tj. předem neznámou) úrovní odkazů — ty musely být realizovány jen pomocí externích jazyků
- SQL nepodporovalo rekurzivní volání dotazů resp. jiné alternativní nástroje pro prohledávání stromů
- nejčastěji se používalo procedurální rozšíření SQL (nestandardní a zbytečně pomalé)
- příklad: vytvořte tabulku všech administrativních jednotek s hloubkou jejich zanoření (tj. řádem 1=stát, 2=kraj, 3=okres atd.)
- dnes je možno tento dotaz zapsat pomocí rekurzivní varianty CTE (standardní, ale nikoliv všeobecně podporované)

```
WITH RECURSIVE subareas AS (
  SELECT name, 1 AS depth, id
    FROM a_jednotky
    WHERE parent IS NOT DISTINCT FROM NULL
  UNION ALL
  SELECT c.name, depth+1, c.id
   FROM subareas AS p
        JOIN a_jednotky AS c
        ON(c.parent = p.id)
)
SELECT * FROM subareas
```

První část rekurzivního dotazu vrací kořenový prvek (zde je to stát) s hloubkou zanoření 1. K němu se připojuje prvky s hloubkou o jednu vyšší, pak o dvě vyšší atd.

ÚKOLY

Vytvořte dotaz, který vypíše všechny vnořené administrativní jednotky bez ohledu na hloubku zanoření.

4 FULLTEXTOVÉ VYHLEDÁVÁNÍ

je speciálním typem vyhledávání dat v textově orientovaných dokumentech

Cílem je nalezení slova (sousloví, množiny slov) v textovém dokumentu s ohledem na (přirozený) jazyk dokumentu

- slovo se hledá bez ohledu na gramatický tvar
- hledají se i synonyma
- fuzzy hledání (překlepy)
- ohodnocuje se i pozice slov (u hledání množin slov i jejich vzájemná pozice)
- nezohledňují se běžná a sémanticky neplnohodnotná slova (pomocná slovesa, předložky, spojky)
- shodu lze kvantifikovat pomocí koeficientu (možný ranking)

PostgreSQL podporuje fulltextové vyhledávání a to pro řadu přirozených jazyků (včetně češtiny). Úroveň podpory jazyků se však liší (nejlepší je samozřejmě v angličtině)

dokument = textový sloupec
spojení textových sloupců title || content || caption
agregovaná textová data (string_agg(name, ' ')

dokument musí být předzpracován

tokenizace = rozdělení do tokenů a klasifikace (slova, čísla)

lematizace = transformace slov na základní tvar

nalezení synonym

to vše se děje pomocí vestavěné podpory využívající externích nástrojů (cspell, apod.) podle konfigurace (per jazyk)

4.1 Tsvector

výsledkem předzpracování je tzv. tsvector (základní tvary slov + pozice + metadata)

```
select to_tsvector('cs', 'Jiří Fišer, Dukelských hrdinů 35');
```

"'35':5 'dukelský':3 'fišer':2 'hrdina':4 'jiří':1"

cs = jediná dostupná konfigurace pro český jazyk
http://postgres.cz/wiki/Instalace_PostgreSQL#Instalace_Fulltextu

4.2 Tsquery

předzpracovaný fulltextový dotaz, který se tokenizuje a lematizuje (odstranění stopwords, interpunkce)

```
plainto_tsquery('cs', 'kočka a pes')
```

lze přímo využít i dotazovací jazyk (mezery mezi slovy nelze použít):

```
to_tsquery('cs', 'kočka | pes')
```

podporované operátory &, *, ! a lze vyjádřit i hledaný prefix:

```
to_tsquery('cs', 'krok:*')
```

najde i krok, kroky apod.

4.3 Dotazy

základem fulltextových dotazů je aplikace dotazu *tsquery* na vektor textových dat *tsvector.*

```
to_tsvector('cs',...) @@ to_tsquery('cs',...)
```

tato konstrukce se využívá nejčastěji v sekci WHERE:

```
SELECT chapter, paragraph, content
  FROM svejk
  WHERE to_tsvector('cs',content) @@ to_tsquery('cs','zeman');
```

Nainstalujte si databázi svejk.sql, která obsahuje všechny odstavce Haškova Švejka,

ÚKOLY

Vytvořte dotaz, který vypíše pro každou kapitolu počet odstavců, které obsahují (v různých tvarech) jméno hlavního hrdiny.

komplexnější fulltextové dotazy mohou kvantifikovat shodu na základě indexů:

- ts_rank -- funkce počtu výskytů
- ts_rank_cd -- funkce tzv. cover density

a vizuálně vyznačovat nalezené shody (ts_headline)

```
SELECT ts_headline('cs', content, query, 'StartSel={, StopSel=}'),
          ts_rank_cd(to_tsvector('cs', content), query) AS rank
FROM svejk, to_tsquery('baloun & maso') AS query
WHERE query @@ to_tsvector('cs', content)
ORDER BY rank DESC
LIMIT 4;
```

ÚKOLY

Najděte ty části Švejka, které se nejrelevantněji věnují problematice česko-uherských stavů (kvalifikujte pomocí funkcí rank *a porovnejte ze skutečností)*

Pro urychlení fulltextových dokumentů je téměř nezbytné (v PostgreSQL však nikoliv nutné) používat specializované indexy.

```
CREATE INDEX svejk_idx ON svejk USING gin(to_tsvector('cs',
content));
```

při častém využívání dokumentu (včetně složených) je vhodné vytvořit pomocný sloupec s předpřipraveným fulltextovým vektorem.

```
ALTER TABLE pgweb ADD COLUMN textsearchable_index_col tsvector;
UPDATE pgweb SET textsearchable_index_col =
     to_tsvector('english', coalesce(title,'') || ' '
                          || coalesce(body,''));
```

ÚKOLY

Vyzkoušejte jaký vliv na urychlení komplexnějších fulltextových dotazů má zavedení indexu (za pomoci příkazu EXPLAIN ANALYZE).

5 ULOŽENÉ PROCEDURY A FUNKCE

funkce a procedury uložené (a prováděné) na straně serveru umožňují výrazným způsobem rozšířit funkčnost PostgreSQL a řídit přístup k databázím

lze je využívat mnoha způsoby:

- jako funkce v různých sekcích SQL příkazů (typicky SELECT) — PostgreSQL podporuje i uživatelské agregační funkce a operátory
- jako producenty vypočtených tabulek (tj. ve funkci read-only pohledů)
- skripty pro komplexnější vkládání hodnot
- dávkové soubory pro automatickou údržbu databáze
- obslužné rutiny triggerů (aktivovány při vkládání, změně a výmazu tabulek)

Uložené funkce lze vytvářet pomocí několika programovacích jazyků

standardní SQL

— bezpečné a rychlé, bohužel s omezenými možnostmi

procedurální rozšíření SQL (PL/pgSQL)

— je třeba se učit novou syntaxi, skvěle integrované

procedurální skriptovací jazyky (PL/Python , PL/Tcl, PL/Perl)

— využití předchozích znalostí + jednoduché API

programovací jazyk C

— nízkoúrovňový kód, super rychlé

ÚKOLY

Vestavěný Python je tzv. untrusted jazyk. Co to znamená? (konzultujte https://www.postgresql.org/docs/current/plpython.html)

5.1 Definice nové uložené funkce

definice nové funkce v PostgreSQL

```
CREATE FUNCTION function_name()
     RETURNS  return_type AS '
          function-body
     ' LANGUAGE programing_language;
```

všimněte si, že tělo funkce je v řetězci (jen tak lze podporovat více programovacích jazyků)

namísto apostrofů se často jako omezovač řetězce používá dvojznak $$

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

Předávání parametrů se děje hodnotou pomocí syntaxe známé z ostatních programovacích jazyků

```
CREATE FUNCTION add(x integer, y integer)
     RETURNS integer AS $$
          SELECT x + y;
     $$ LANGUAGE SQL;
```

I když je tělo funkce řetězcem nedochází k jednoduché textové substituci (text je předparserován). Parametry lze používat pouze na místech, kde jsou očekávány hodnoty nikoliv například identifikátory či dokonce klíčová slova.

ÚKOLY

Vytvořte funkci, která spočítá vzdálenost dvou bodů (parametry jsou reálné hodnoty x1, y1, x2, y2).

Funkce vracející jednoduché hodnoty (čísla, řetězce, …) lze volat v rámci sekce SELECT (včetně bezezdrojové verze)

```
SELECT  add(1,1);
```

tento zápis lze využít i v případě funkcí bez návratové hodnoty s postranním efektem (všimněte si odlišení parametru od stejnojmenného sloupce).

```
CREATE FUNCTION debit_account (accountno integer, debit numeric) RETURNS
integer AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = debit_account.accountno;
    SELECT 1;
$$ LANGUAGE SQL;

SELECT debit_account(42, 1.0);
```

ÚKOLY

V tabulce 2D bodů (sloupce x1, y1, x2, y2 a id_bodu) nalezněte dva nejbližší body (rada: použijte funkci z předchozího úkolu a CROSS JOIN).

5.2 Funkce nad složenými hodnotami

Důležitou vlastností PostgreSQL funkcí je možnost zpracování složených hodnot (tzv. záznamů)

Složené hodnoty lze používat přímo (pak musí být daný složený typ definován) nebo lze využívat řádky tabulky (pak roli definice typu hraje definice tabulky)

```
-- použití SELECT s více sloupci
CREATE OR REPLACE FUNCTION mul_frac(x fraction, y fraction)
RETURNS fraction AS $$
        SELECT x.nom * y.nom, x.denom * y.denom
$$ LANGUAGE SQL;

-- použití funkce ROW a vrácení jednoho (složeného sloupce)
CREATE OR REPLACE FUNCTION mul2_frac(x fraction, y fraction)
RETURNS fraction AS $$
        SELECT ROW(x.nom * y.nom, x.denom * y.denom)::fraction
$$ LANGUAGE SQL;
CREATE TYPE FRACTION AS (nom INTEGER, denom INTEGER);
```

Bez ohledu na způsob definice, lze funkce vracející složenou hodnotu volat dvěma mechanismy:

- v sekci SELECT : výsledkem je jedna složená hodnota (tabulka 1×1) — obtížnější následné zpracování
  ```
  SELECT mul_frac(ROW(2,3), ROW(1,4));
  ```
- v sekcu FROM : výsledkem je jednořádková tabulka s více sloupci 1×N
  ```
  SELECT * FROM mul2_frac(ROW(2,3), ROW(1,4));
  ```

ÚKOLY

Vytvořte nový typ representující 3D vektory a funkci realizující vektorový součin. Předpokládejte tabulku rovin, určených třemi body neležícími na přímce. Pomocí této funkce vytvořte tabulku normál.

5.3 Funkce vracející množinu hodnot resp. tabulku

- funkce může vracet i množinu hodnot (= tabulku s více řádky)
- stačí použít specifikace **SETOF typ** (kde typ určuje typ každého řádku)
- zdrojem bývá nejčastěji tabulka resp. dotaz/pohled

```
CREATE FUNCTION fun() RETURNS SETOF int AS $$
SELECT id FROM table
$$ LANGUAGE SQL;
```

Množiny lze generovat i dynamicky:

Pokud chceme vrátit tabulku (= množina řádků), pak je-li definován typ řádků (datovým typem

```
CREATE OR REPLACE FUNCTION squares(max int)
RETURNS SETOF INT AS $$
        SELECT s * s FROM generate_series(1, max) as s
$$ LANGUAGE SQL;

SELECT * FROM  squares(10);
```

nebo tabulkou) pak je možné využít konstrukci SETOF

U tabulek s jednorázovým typem je použit návratovou hodnotu typu TABLE:

```
CREATE FUNCTION  crop(limit int)
     RETURNS TABLE(name string, salary numeric) AS $$
          SELECT name, salary FROM staff WHERE salary >= limit;
$$ LANGUAGE SQL;
```

ÚKOLY

1. Uložené procedury (funkce) vracející tabulku mají podobné využití jako pohledy. V čem se liší?
2. Následující volání funkce využívá specifikaci LATERAL? K čemu slouží?

```
CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL STABLE;

SELECT name, child
FROM nodes, LATERAL listchildren(name) AS child;
```

5.4 Volatilita funkcí

Při definici funkcí je vhodné uvádět tzv. volatilitu (česky – proměnlivost), explicitní uvedení volatility umožňuje lepší optimalizaci při volání funkce

VOLATILE = funkce může měnit svět (typicky tabulky v databázi) a může při každém volání vracet jinou hodnotu (při stejných parametrech!). Volání těchto funkcí nelze optimalizovat

STABLE = funkce nemění databázi a vrací stejné hodnoty při shodě parametrů a nad stejným řádkem (v rámci jedné transakce). Několik funkcí nad řádkem lze tak spojit do jediné funkce. Patří sem i časové funkce typu *current_time*()

IMMUTABLE = vždy vrací stejné hodnoty pro stejné parametry

ÚKOLY

Ohodnoťte volatilitu dříve uvedených funkcí.

6 PROCEDURÁLNÍ ROZŠÍŘENÍ

6.1 PL/PGSQL

- procedurální rozšíření SQL
- není přenositelné (i když je odvozeno PL/SQL Oraclu a je mu tudíž velmi podobné)
- je vždy k dispozici (u každé PostgreSQL databáze a na každé PostgreSQL platformě)
- je bezpečné (neumožňuje provádět změny mimo databázi)
- syntaxe je obdobná jiným procedurálním jazykům (i když nepatří do rodiny C-jazyků)

V následující ukázce jsou procedurální konstrukce označeny červeně.

Všimněte si, že:

1. BEGIN má v PIL/PGSQL jiný význam než v SQL (i když tělo funkce tvoří mimo jiné i transakční blok)
2. konstrukce INTO je přiřazením (přiřazení se nezapisuje operátorem)

```
CREATE FUNCTION get_userid(username text) RETURNS int
AS $$
DECLARE
     userid int;
BEGIN
    SELECT users.userid INTO userid
        FROM users WHERE users.username = get_userid.username;
    RETURN userid;
END
$$ LANGUAGE plpgsql;
```

Řídící konstrukce jsou podobné jiným procedurálním jazykům.

Prostudujte si dokumentaci k PL/PGSQL na https://www.postgresql.org/docs/current/plpgsql.html. Zaměřte se na následující konstrukce:

IF, WHILE, FOR (ve dvou podobách)

Výjimkou jsou konstrukce zaměřené na zpracování tabulek (tj, nikoliv běžných skalárů).

Lze tak například postupně vracet jednotlivé části výstupní tabulky pomocí RETURN NEXT (po řádku) nebo RETURN QUERY (po segmentech). [částečně odpovídá konstrukci *yield* v Pythonu].

```
CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS $$
DECLARE
    r foo%rowtype; -- r je stejného typu jako řádek tabulky
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP
        -- zde může být nějaké zpracování
        RETURN NEXT r;
    END LOOP;
    RETURN;
END $$ LANGUAGE plpgsql;
```

ÚKOLY

1. Vytvořte PL/PGSQL funkci, která vrací vrcholy pravidelné n-cípé hvězdy jako tabulky se dvěma sloupci (x-ová a y-ová souřadnice).

2. Vytvořte PL/PGSQL (resp. SQL) funkce, které pracují s tzv. temporální tabulkou To je tabulka, ve které může být několik záznamů se stejnou identitou (tj. například shodným identifikátorem), jejichž platnost je však časově omezena, Časovou platnost určují dva přidané sloupce, které určují, od kdy záznam platí a do kdy (mají typ TIMESTAMP). Pro jednoduchost pracujte s tabulkou o třech sloupcích, kde první je původní číselný primární klíč.

3. Implementujte funkci pro vkládání (údaj platí od aktuálního časového okamžiku, dokud se nevloží údaj se stejným klíčem) a výpis hodnot platných v určitý okamžik.

6.2 Python

Podporu je nutné zavést v rámci každé databáze, kde je použita:

```
CREATE EXTENSION plpythonu;
```

pakk už je použití podobné jako u SQL či pgSQL

Jednoduché hodnoty jsou mapovány do obdobných pythonských typů:

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpython3u;
```

záznamy jsou mapovány do slovníků (klíčem je jméno sloupce)

```
CREATE FUNCTION overpaid (e employee)
```

```
  RETURNS boolean
AS $$
  if e["salary"] > 200000:
    return True
  if (e["age"] < 30) and (e["salary"] > 100000):
    return True
  return False
$$ LANGUAGE plpython3u;
```

Množiny (posloupnosti) mohou být vráceny jako pole, nebo iterátory (včetně generátorů s příkazem yield).

Pro přístup k databázi je možné využít automaticky importovaný modul *plpy*.

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

rv je indexovatelná kolekce záznamů tj. slovníků. rv[i][column_name]

lepší je však využití kurzoru:

```
CREATE FUNCTION count_odd_fetch(batch_size integer) RETURNS integer AS $$
odd = 0
cursor = plpy.cursor("select num from largetable")
while True:
    rows = cursor.fetch(batch_size)
    if not rows:
        break
    for row in rows:
        if row['num'] % 2:
            odd += 1
return odd
$$ LANGUAGE plpythonu;
```

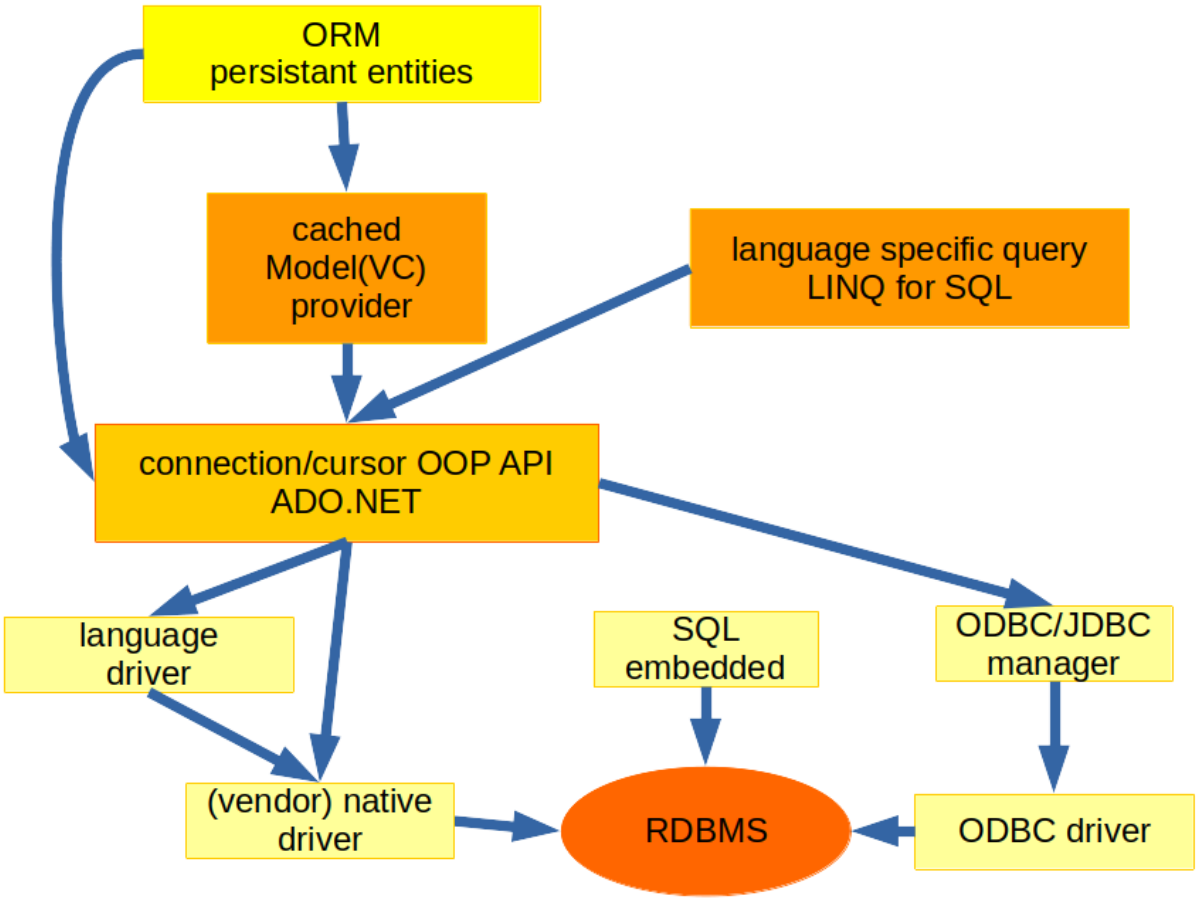
ÚKOLY

1. Implementujte příklady uvedené v předchozím úkolu, ale tentokrát v Pythonu.
2. Porovnejte časovou efektivitu obou řešení.

7 PROGRAMOVÁNÍ KLIENTŮ

Vytváření kódu klientů tj. vně serveru závisí na zvolené vývojové platformě, která je dána primárně použitým programovacím jazykem a úrovní abstrakce.

Základní úrovně abstrakce jsou uvedeny na následujícím (zjednodušeném) diagramu:



Python podporuje rozhraní založené na přímém volání SQL kódu, dotazy vracejí kurzory, které se chovají jako iterátory přes záznamy (n-tice). Podobné rozhraní podporují téměř všechny běžně používané programovací jazyky (známé je např. v PHP)

Toto rozhraní je v Pythonu standardizované tj. nezávisí na použitém DBMS (jedná se o vnější rozhraní, vnitřní rozhraní tj. SQL standardizované není.)

Prostudujte toto rozhraní ve standardním modulu https://docs.python.org/3.9/library/sqlite3.html (tento používá známou vestavěnou databázi SQLite, a je dostupný na většině platforem).

Pozornost věnujte interpolaci parametrů na straně serveru, což je nezbytný přístup, pokud jsou data v dotazech z nezabezpečených zdrojů (viz SQL Injection).

V případě PostgreSQL doporučuji modul *psycopg2*. Prostudujte dokumentaci na stránkách https://www.psycopg.org/docs/.

Vysokoúrovňový přístup tzv. ORM (Object Relational Mapping) je pro Python k dispozici v podobě několika desítek modulů. Doporučuji SqlAlchemy.

Prostudujte tutoriál: https://docs.sqlalchemy.org/en/13/orm/tutorial.html

ÚKOLY

*1. Pomocí nízkoúrovňového kódu (*sqlite3 resp. pyscopag2*) vytvořte tabulku representující část souborového systému (např. domovský adresář s podadresáři), kde jsou u každého souboru uvedeny základní atributy (adresář/soubor/link, velikost, čas poslední modifikace) a je zohledněn hierarchický relační vztah soubor/adresář.*
2. Totéž implementujte pomocí SQL Alchemy.