Databázové systémy MMXIV

Jiří Fišer

Starší databázové modely

- vznikaly v padesátých a šedesátých letech především v IBM
- vycházely z běžné paměťové representace datových struktur
- základem byla/je pole struktur pevné délky – tabulka
- složitější vztahy byly modelovány skládáním (**hierarchický databázový model**) nebo odkazy na spojové seznamy (**síťový model**)
  hierarchický model je využíván i dnes: adresářové služby (LDAP), XML databáze

Relační databázový model

- byl definován v roce 1970
- Codd E.F.
  A Relational Model of Data for Large Shared Data Banks
- vychází z implementací tabulkového a síťového modelu, je však přesně definován pomocí elementárních matematických pojmů: množin, relací a operací (= funkcí) nad relacemi
- klade důraz na datovou integritu (a to jak ve statickém pohledu tak i dynamickém)
- v současnosti je to **klasický** a výrazně převažující **databázový** model, ostatní modely z něho vycházejí a/nebo se vůči němu vymezují (NoSQL)

Tabulka

- základní datová struktura
- tabulka se sestává z posloupnosti uspořádaných n-tic (tuples) neboli řádků (rows), pro něž platí:
  - **n-tice v tabulce jsou jedinečné** (= neexistují dvě stejné n-tice)
  - **všechny n-tice v tabulce jsou strukturálně shodné**, tj. obsahují stejný počet položek a odpovídající si položky jsou stejného typu (primárně: jsou stejně representované)
  - **položky jsou atomické**, tj. nelze je dále dělit na podpoložky (= nejsou to pole či podstruktury)

R-DBMS

- **DBMS** = database management systém (SŘBD = systém řízení báze dat)
  - úložiště (data + metadata)
  - výkonné jádro zajišťující průběžný přístup a konzistenci databáze
  - administrativní aplikace
  - podpora distribuovaného supersystému
- **RDBMS** = relační databázové systémy
  - DBMS podporující primárně relační databázový model
  - typické RDBMS podporují SQL

Klasifikace DBMS

- **klient-server DBMS**:
  - výkonné jádro DBMS tvoří (alespoň) jeden oddělený proces (typicky na dedikovaném serveru)
  - klienti se připojují přes IPC (typicky TCP) pomocí proprietárního protokolu
  - SQL: PostgreSQL, MySQL, DB2, Oracle, MS SQL server, NoSQL: Mongo, Oracle NoSQL
- **vestavěné DBMS**
  - výkonné jádro je tvořeno (dynamickou) knihovnou, jež je součástí klienta
  - SQL: SQLite, HyperSQL, NoSQL: Berkeley DB

SQL

Structured Query Language

- jazyk založený na relačním databázovém modelu (QL, částečně DDL, DML)
  - QL: dotazy nad databázemi (SELECT)
  - DDL: data definition language (CREATE TABLE)
  - DML: data manipulation language (INSERT, UPDATE)
  - transaction control (BEGIN, COMMIT)
  - DCL: data control language (GRANT, REVOKE)
  - DBMS administrace
- úzce propojen s vývojem relačního modelu a RDBMS (dnešní SQL však obsahuje i nerelační prvky)
- de facto standard pro RDBMS (jen v oblasti dotazů existují okrajové alternativy QBE, LINQ)

Standardizace SQL

- SQL je průběžně standardizováno organizací ISO
- nejdůležitější standardy:
  - **SQL-92** (společný základ, některé části jsou však již zastaralé)
  - SQL:2003 (v zásadě podporováno, XML, analytické funkce)
  - SQL:2008, SQL:2011 (jen dílčí podpora)

podpora standardů není ani u nejvyspělejších systémů úplná:

- vlastní (starší) rozšíření s podobnou, ale odlišnou sémantikou
- vlastní (odlišná) syntaxe
- odmítání (nekompatibilní s implementací, chybný návrh)
- není vyžadována zákazníky (nepřináší výhody)
- není čas na implementaci (SQL není vše)

Standardizace SQL II

standard navíc nespecifikuje (resp. dříve nespecifikoval) některé klíčové oblasti:

- komplexní objekty a operace nad nimi (např. 2D a 3D grafickými objekty) včetně např. tvorby indexů
- fulltextové vyhledávání
- použití BLOB (velkých binárních objektů)
- konkrétní práva a jejich mapování na role (= uživatele)
- administrace a nastavení lokálních kontextů (pouze základní syntaxe)
- procedurální rozšíření

ve všech těchto oblastech se implementace v jednotlivých RDBMS může výrazně lišit

PostgreSQL

- **PostgreSQL** = jeden ze dvou nejdůležitějších open-source relačních databázových systémů (druhým je MySQL/MariaB)
- PostgreSQL na rozdíl od MySQL podporuje i novější SQL rozšíření (analytické funkce, XML)
- standardní podpora trigerů a transakcí (MySQL jde v těchto oblastech svou vlastní cestou)
- *PL/pgSQL* (procedurální rozšíření) je obdobou *PL/SQL* z RDBMS *Oracle* (i v jiných aspektech se PostgreSQL inspirovalo u *Oraclu*)
- dobrá dokumentace

Základní pravidlo

- *veškeré zpracování by mělo být provedeno pomocí SQL (resp. jeho procedurálního rozšíření) v DBMS nikoliv v klientovi (klient pouze zobrazuje získaná data)*
  - obecně rychlejší (server je optimalizován)
  - konzistence v distribuovaném prostředí
  - menší zátěž přenosového media (sítě)
- **výjimka z pravidla**

  textové zpracování výstupů většinou provádí klient (národná špecifiká, časová pásma, apod.) i když moderní DBMS mají podporu i v této oblasti)

SQL tabulka

- řádky (n-tice/tuples) = údaje o jednom objektu/entitě
- sloupce (columns) = jednotlivé atributy objektů

- všechny objekty ve sloupci musí být stejného datového typu (datové domény)
  - representují stejné atributy u různých objektů
- standardní SQL podporuje jen elementární typy u atributů (sloupců), lze však vytvářet ad hoc *podtypy* (podmnožiny původních typu omezené pomocí predikátů)

Základní syntaxe

- SQL standardně nerozeznává malá a velká písmena (v klíčových slovech i identifikátorech)
- identifikátory jsou běžně tvořeny jen ASCII alfanumerickými znaky (první znak musí být písmeno)
- symbol v uvozovkách (*quoted identifier*) je však vždy interpretován jako identifikátor:
  - s možností použití libovolných Unicode znaků včetně mezerových znaků "oblíbený sloupec"
  - klíčových slov "select" (identifikátor)
  - s rozlišením majuskulí a minuskulí
- řádkové poznámky (komentáře) začínají dvěma pomlčkami

```
-- poznámka
```

SQL číselné datové typy

- celočíselné typy

SMALLINT	16 bitové znaménkové int	short
INT[EGER]	32 bitové znaménkové int	int
BIGINT	64 bitové znaménkové int	long

- čísla s řádovou čárkou

  NUMERIC(p, s) – dekadické číslo s *p* číslicemi celkem a *s* číslicemi za desetinnou čárkou (maximální *p* a *s* se u DBMS liší, lze však počítat minimálně s p=32 a je tak vhodný pro uložení hodnot typu *decimal*)

  podporuje jen základní aritmetické operace (včetně zaokrouhlení)

REAL	32 bit IEEE 754	float
DOUBLE [PRECISION]	64 bit IEEE 754	double

- podporují i základní transcendentální funkce (např. logaritmy)

SQL řetězcové typy

character(n), char(n)

- řetězce s fixní délkou *n*. Kratší řetězce jsou zprava doplňovány mezerami. Maximální použitelné *n* může být u některých DBMS jen 256.

  vhodné pro krátké ASCII řetězce s fixní délkou: ISBN, RČ, SPZ

character varying(n), varchar(n)

- řetězce s proměnlivou délkou (maximálně však *n*).

  Interpretace a omezení na *n* se u různých DBMS liší:

  nejhorší situace: *n* je v bytech a je maximálně 256 (starší MySQL)

  běžná situace: n je v dvoj-bytech a může být až 4000 (SQL Server **n**varchar)

  nejlepší: *n* je ve (vícebytových) znacích a může být až 2^{32} (PostgreSQL)

SQL datumové datové typy

- nabídka, sémantika a rozsah se v jednotlivých DBMS mírně (zde nabídka PostgreSQL)

  time denní čas

  date datum v proleptickém řehořském kalendáři

  timestamp časový okamžik (date + time)

  timestamp with time zone [timestamptz]

  interval časový interval

- datové typy označené *with time zone* ukládají čas v UTC (GMT) a server resp. klient provádí pří každém přístupu převod z UTC na klientovo aktuální časové pásmo

- volba timestamp/timestamptz závisí na požadované sémantice

Ostatní běžné datové typy

- **boolean** (boolovské hodnoty)
- **large objects (CLOB/BLOB)**
  - velká maximální velikost (běžně 2GB, někdy i více např. 128 TB u *Oracle*)
  - vhodné pro ukládání rozsáhlých textů či binárních dat (např. obrázků)

	standard	PostgreSQL
textová data	CLOB	text
binární data	BLOB	bytea

SQL literály

- **řetězcové literály** jsou v apostrofech: 'Ank Morpork'
- **časové literály** (ANSI SQL):

  TIMESTAMP '2012-08-08 16:48:00'

  TIMESTAMP WITH TIME ZONE '2012-08-08 16:48:00+02'
- **časové literály** (stručnější zápis pomocí PostgreSQL přetypování, lze využít i další formáty času)

  '2012-08-08 16:48:00+02'::timestamptz

  'January 8 04:05:06 1999 PST'::timestamptz
- **bytea** (BLOB), hexadecimální formát (PostgreSQL)

  E'\\xDEADBEEF'

Přetypování

- SQL podporuje **statické typování** (již při překladu)

  ISO standard bohužel nepopisuje **typový systém** SQL příliš detailně a tak **se** jeho pojetí a rozsah v jednotlivých implementacích **liší**

- PostgreSQL má relativně komplexní a striktní typový systém se složitými a flexibilními pravidly implicitního přetypování a přetěžování funkcí a operátorů (specialitou je implicitní přetypování z řetězcových literálů)

- mnohdy je však nutno provést **explicitní přetypování** (číslo->string, double->numeric, atd)

- ANSI přetypování: CAST(value AS type)

- PostgreSQL přetypování: value::type

Hodnota NULL

- typickým rysem SQL je podpora (pseudo)hodnoty **NULL**. Ta je podporována u všech datových typů a representuje:
- **neznámou hodnotu** (*not available, N/A*), např. neznámé jméno
- **neaplikovatelnou hodnotu** (daný atribut není aplikovatelný pro daný objekt)
  např. plat u dobrovolníka, či zlatý padák u dělníka

  použití neaplikovatelné hodnoty může být příznakem špatného návrhu: je tak vlastně realizována nepravoúhlá tabulka (= tabulka s různým počtem sloupců)
  řešení: použití více tabulek (v SQL nepříliš elegantní)
- v žádném případě by však neměla representovat (prozatím) *neinicializovanou hodnotu* (jak je tomu u ref. typů OOP jazyků)

Operace s NULL

- NULL musí být zohledněno i v operacích nad elementárními hodnotami. Jsou použity tři řešení:
- NULL je interpretováno jako **neplatní hodnota** => šíří se

  1+NULL = NULL, 2 > NULL = NULL (většina operací)
- NULL je **ignorováno**

  agregační funkce typu SUM, AVG
- hodnota „nevím“ v **trojhodnotové logice** 3VL (logické operátory)

AND	T	U	**F**
T	T	U	**F**
U	U	U	**F**
F	**F**	**F**	**F**

OR	T	U	F
T	T	**T**	T
U	**T**	U	U
F	T	U	F

Převod NULL na hodnotu a v.v.

- při převodu NULL na běžnou hodnotu se uplatní funkce COALESCE

  COALESCE(v_1, v_2, …, v_n)

  vrací první ne NULL hodnotu ze sekvence v_1, v_2, …, v_n. Jen pokud jsou všechny NULL vrací NULL.

- pro opačný převod (náhradní hodnota na NULL)

  NULLIF(a, b)

  vrací NULL pokud a == b

- pro testování zda je hodnota NULL **je nutno využít**

  IS DISTINCT FROM NULL tesp. IS NOT DISTINCT FROM NULL

  u logických hodnot IS UNKNOWN resp. IS NOT UNKNOWN

Vytvoření tabulky

- tabulka je vytvářena pomocí příkazu CREATE TABLE

```
CREATE TABLE zamestnanci
(
  jmeno varchar(32),
  prijmeni varchar(32),
  zamestnan_od date,
  interni boolean,
  zamestnan_do date,
)
```

příkaz uvádí jméno tabulky, a domény jednotlivých sloupců (atributů)

Omezení datových domén

- SQL podporuje vytváření **podtypů** pomocí tzv. *contraints*

$$MPH(podtypu) \subset MPH(typu)$$

- NOT NULL
  - v daném sloupci nejsou přípustné hodnoty NULL
  - jmeno varchar(32) NOT NULL
- CHECK (podmínka)
  - zamestnan_od date CHECK (zamestnan_od >= '2000-01-01'::date)
  - toto omezení lze pojmenovat (vhodné pro ladění)
  - zamestnan_od date CONSTRAINT min_date_from
    CHECK (zamestnan_od >= '2000-01-01'::date)
  - a lze je využívat i na úrovni tabulky (měly by být pojmenované)
  - CONSTRAINT mdf CHECK (zamestnan_od >= '2000-01-01'::date)

Tabulková omezení

- integritní omezení neomezují datovou doménu, ale zajišťují splnění jisté podmínky mezi hodnotami v jednom, či více sloupcích
- mezisloupcový **CHECK**
  - podmínka vázající hodnoty mezi sloupci (v jednom řádku)
  - CONSTRAINT date_check CHECK
    (zamestnan_od <= zamestnan_do)
- **UNIQUE**
  - hodnoty v rámci sloupce musí být unikátní
  - idp char(10) UNIQUE - - id pracovníka

omezení UNIQUE se využívá relativně zřídka, neboť sloupce s unikátními hodnotami bývají tzv. primárními klíči

Primární klíče

- řádky v rámci jedné tabulky musí být unikátní (požadavek relační algebry)
- v praxi však u vícesloupcových tabulek některé sloupce závisí na jiných a neslouží primárně k identifikaci ale jsou to atributy objektu-řádku (mohou být změněny bez změny identity objektu)
- nezávislé sloupce jednoznačně identifikující objekt/řádek se označují jako **primární klíče**
- primárním klíčem může být **jeden sloupec** (časté), *několik málo sloupců* (méně časté) nebo dokonce celý řádek (řídce u tabulek s několika málo sloupci)
- sloupce mohou být libovolného typu – nejčastěji je to však číslo nebo krátký řetězec, výjimečně časový údaj

Požadavky na primární klíč

- hodnoty musí být jedinečné (unikátní), a to nejen v aktuální tabulce, ale i při jejím (potenciálním) rozšiřování

  Je například rodné číslo jedinečné?

- pokud je vícesloupcový pak musí být všechny sloupce nezávislé (= hodnota jednoho sloupce neurčuje hodnotu jiného): 2NF
- hodnota primárního klíče musí být známa pro každý záznam (tj. nesmí být nikdy NULL)
- porovnání na shodu musí být rychlé a paměťově efektivní (unikátnost se testuje při každém vložení), to v zásadě splňují jen čísla, krátké řetězce (cca < 32 znaků)

Umělé primární klíče

- někdy **nelze ani jeden ze sloupců použít jako primární klíč** (není splněn jeden z výše uvedených požadavků)
- pak je možno přidat **dodatečný sloupec**, který jedinečnost zajistí (a slouží jako *ad hoc identifikátor*)
- typicky obsahuje posloupnost rostoucích čísel (tzv. sekvenci), která však nemusí být souvislá. Tuto posloupnost by měl zajišťovat databázový server (u klienta hrozí nekonzistence daná souběhem)
- některá prostředí si přidání umělého primárního klíče vynucují, v obecném případě by však měl být použit jen, když je nezbytný
- jednotlivé databáze se v oblasti podpory auto(sekvencí) liší (tj. **nelze vytvořit plně přenositelnou definici** UPK)

Specifikace primárního klíče

- primární klíč musí být v definici tabulky označen integritním omezením **PRIMARY KEY** (integritní omezení omezují vazby mezi tabulkami)
  - idp int PRIMARY KEY
- toto omezení (mimo jiné) implikuje omezení NOT NULL a UNIQUE na daném sloupci
- pro svou důležitost je často definováno na úrovni tabulky a pojmenováno:
  - CONSTRAINT zamestnanec_pk PRIMARY_KEY(idp)
- pokud tvoří klíč více sloupců pak musí být definováno na úrovni tabulky:
  - CONSTRAINT zamestnanec_pk PRIMARY_KEY(id_zavod, idp)

Vytvoření tabulky II

- druhá verze příkazu pro vytvoření tabulky obsahuje i omezení

  a sloupec primárního klíče

```
CREATE TABLE zamestnanci
(
  idp INT,
  jmeno varchar(32) NOT NULL,
  prijmeni varchar(32) NOT NULL,
  zamestnan_od date NOT NULL,
  interni boolean NOT NULL,
  zamestnan_do date,

  CONSTRAINT zamestnanci_pk PRIMARY_KEY(idp),
  CONSTRAINT min_date CHECK (zamestnan_od >= '2000-01-01'::date),
  CONSTRAINT date_interval CHECK (zamestnan_od <= zamestnan_do)
)
```

Implicitní hodnoty

- v definici tabulky lze u každého sloupce definovat implicitní hodnoty (default values)
- ty jsou použity, pokud není při vkládání řádku specifikována hodnota pro daný sloupec (=atribut)

  interni boolean DEFAULT true

- pro časové sloupce se často používá jako implicitní hodnota aktuální datum/čas

  d date NOT NULL current_date (podobně current_time)

- umělé primární klíče se běžně inicializují pomocí čísel z generátoru číselných sekvencí (musí být předem vytvořen)

  idp integer DEFAULT nextval('zamestnanci_idp_seq')

- zkratka: idp SERIAL (vytváří automaticky sekvenci)

Vytvoření tabulky finální verze

- finální příkaz pro vytvoření tabulky obsahuje i implicitní hodnoty u atributů, u nichž může mít smysl.

```
CREATE TABLE zamestnanci
(
  idp SEARIAL,
  jmeno varchar(32) NOT NULL,
  prijmeni varchar(32) NOT NULL,
  zamestnan_od date NOT NULL DEFAULT current_date,
  interni boolean NOT NULL DEFAULT true,
  zamestnan_do date,

  CONSTRAINT zamestnanci_pk PRIMARY_KEY(idp),
  CONSTRAINT min_date CHECK (zamestnan_od >= '2000-01-01'::date),
  CONSTRAINT date_interval CHECK (zamestnan_od <= zamestnan_do)
)
```

Vkládání dat do tabulky

- základním příkazem pro vkládání dat je příkaz INSERT

  INSERT INTO table VALUES (sekvence-hodnot)

  hodnoty jsou literály (výjimečně statické výrazy), musí být uvedeny v pořadí definice v CREATE TABLE

- bezpečnější verze s explicitním uvedením sloupců

  INSERT INTO TABLE (jména-sloupců) VALUES (sekvence-hodnot)

  o pořadí hodnot rozhoduje explicitní pořadí sloupců

  tento tvar je jediným standardním s podporou vkládání menšího počtu hodnot (zbývající jsou doplněny implicitní hodnotou nebo NULL)

Rychlé vkládání

- vkládání po jednom řádků je neefektivní pro vkládání velkého počtu řádků (tisíce a více) – *bulk loading*
- řádově rychlejší je vícenásobné vkládání:

```
INSERT INTO table (jména-sloupců)
    VALUES (hodnoty-řádku), (hodnoty-řádku), ...;
```

- ještě rychlejší je kopírování z textového CSV souboru:

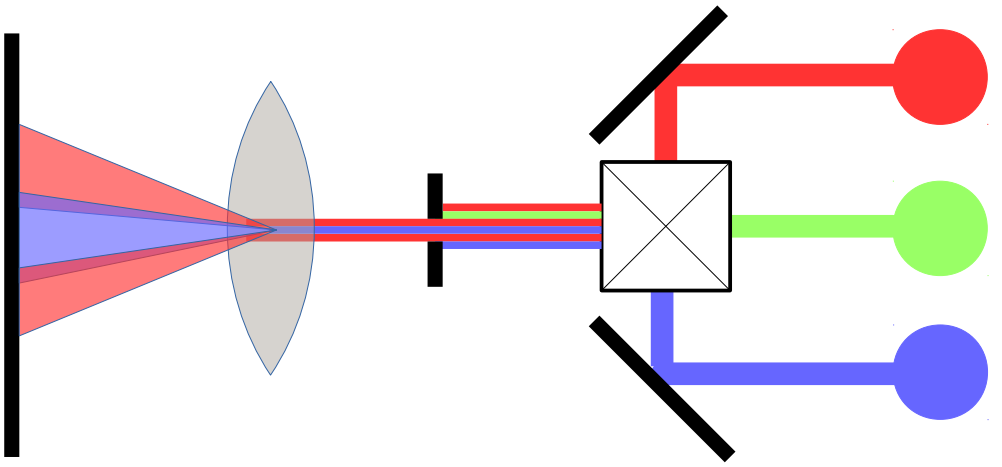
```
COPY table FROM filename|STDOUT FORMAT csv
DELIMITER ',' NULL'' QUOTE '"';
```

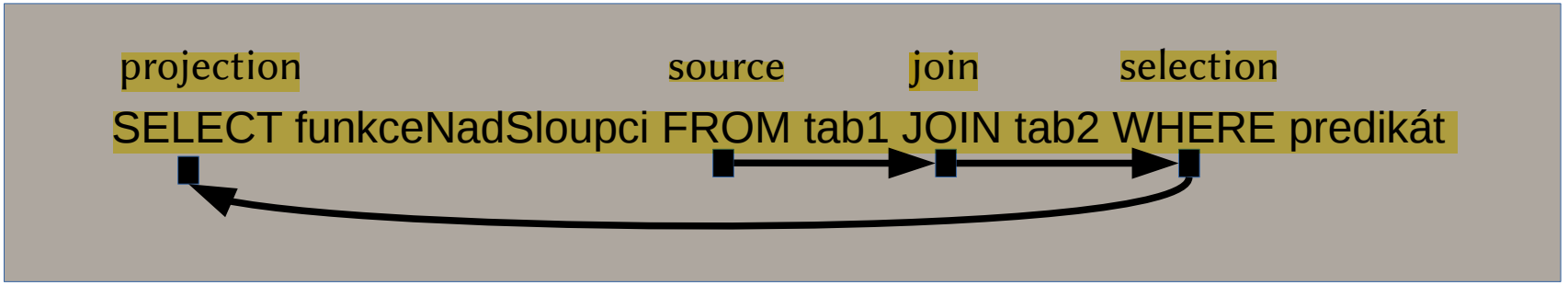
- podporován je ještě holý *text* (implicitní oddělovač tabulátor) a binární formát *binary* (běžně získaný zápisem pomocí COPY TO)

Příkaz SELECT

- příkaz SELECT je klíčovou konstrukcí jazyka SQL
- primárně slouží pro dotazy nad databází tj. pro získání části dat podle explicitně stanovených požadavků
- současné využití příkazu SELECT však není omezeno jen na dotazování: používá se pro výpočty (skalárních) hodnot, rozsáhlé transformace, kontingenční výpočty i jako podklad mechanismu pohledů.
- základní podoba příkazu SELECT (SELECT-FROM(JOIN)-WHERE-ORDER BY) je sdílena všemi moderními RDBMS
- tento tvar příkazu vychází ze základních operací relační algebry

SELECT (a relační algebra)





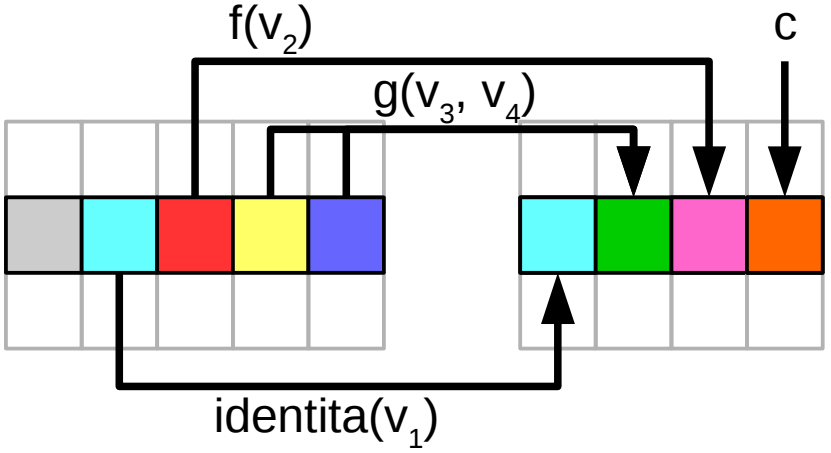
Sekce SELECT

- sekce SELECT provádí **transformaci** - tj. zobrazení každého **řádku (vstupní) tabulky na řádek výstupní tabulky** (obecně různé od vstupní), přičemž toto zobrazení je u všech řádků stejné

  výsledkem dotazu je vždy tabulka!

- toto zobrazení lze dále rozložit na **dílčí zobrazení**, které zobrazují sloupec či sloupce vstupní tabulky na (jeden) sloupec tabulky výstupní

- SELECT tak funguje jako n-tice funkcionálů *map* a *zip*, jež jsou volány nad seznamy (sloupci tabulky) a vrací seznam n-tic



SELECT bez zdroje

- nejejdnodušší (ale netypické) je využití příkazu bez uvedení zdroje tj. jen pro projekci (= vyhodnocení) skalárního výrazu nebo sekvence skalárních výrazů

  skalární výraz = výraz vracející jednu elementární hodnotu = skalár

  SELECT 2 < 3;

  vrací tabulku obsahující jeden řádek a jeden sloupec s hodnotou true::boolean

  SELECT now();

  vrací 1×1 tabulku s aktuálním časem (typu timestamp with tz)

  SELECT 'Hello year'::varchar(10), extract(year from now());

  vrací 1×2 tabulku: "Hello year" 2014 (2014 je typu double!)

Ukázková databáze

- databáze všech hvězd viditelných na obloze pouhým okem
  cca 8 884 řádků (limitní magnituda 6,5)
  extrahováno z **SKY2000 Catalog, Version 4 (Myers+ 2002)**

```
CREATE TABLE vstars
(
  recno integer,                    -- identifikátor v katalogu
  ra double precision NOT NULL, -- rektascenze ve stupních
  de double precision NOT NULL, -- deklinace ve stupních
  hd character(6),                  -- označení v HD katalogu
  sao character(6),                 -- označení v SAO katalogu
  pm_ra double precision,           -- změna RA arcsec/year
  pm_de double precision,           -- změna DE arcsec/year
  dist numeric,                     -- vzdálenost
  mag double precision,             -- jasnost
  sp character(3),                  -- spektrální typ
  CONSTRAINT stars_pkey PRIMARY KEY (recno)
)
```

SELECT – projekce řádků tabulky

- speciální tvar

  SELECT * FROM vstars;

  zobrazeny je celá tabulka (všechny řádky a všechny sloupce)

- výběr omezené podmnožiny sloupců:

  SELECT ra,de,mag FROM vstars;

- hodnoty ze sloupců lze spojovat pomocí operací, funkcí apod.

  SELECT dist, mag, round((**mag**-5*(log(**dist**/3.2616)-1))::numeric, 2) FROM vstars;

  vypíše tři sloupce vzdálenost, jasnost a absolutní jasnost

  8.60 -1.44 1.45
  313 -0.63 -5.54
  4.39 -0.01 4.34

ORDER BY — řazení

- řádky lze uspořádat podle libovolného sloupce

  SELECT mag FROM vstars ORDER BY stars;

- implicitně se řadí vzestupně, pro sestupné řazení se ke sloupci připojí DESC (z descending) [opakem je ASC z ascending]

- lze řadit i vícestupňově podle několika sloupců (primárním klíčem je sloupec vlevo, shodné řádky se pak řadí podle druhého sloupce atd.)

  SELECT sp, mag FROM vstars ORDER BY sp DESC, mag;

- řadit lze i podle výrazů (zde podle vlastního pohybu)

  SELECT ra, de, mag, sqrt(pm_de^2 + (pm_ra * cos(de))^2) AS pm FROM vstars ORDER BY pm DESC;

$$\mu^2 = {\mu_\delta}^2 + {\mu_\alpha}^2 \cdot \cos^2 \delta$$

Konstrukce AS s výrazem

- konstrukce AS s výrazem umožňuje **pojmenovat sloupec výstupní tabulky**

  výraz AS identifikátor

- identifikátor sloupce se využije při textovém či formátovaném výstupu, a lze jej použít i při programovém zpracování výstupní tabulky
- využívá se hlavně tehdy, není-li výrazem jméno sloupce a výstupní sloupec by byl bezejmenný
- lze jej jej použít na místě jména sloupce v sekci **ORDER BY** ta **se** totiž **provádí** až **po sekci SELECT** tj. již na výstupní tabulce
- klíčové slovo AS lze vynechat (je to však méně přehledné)

Sekce WHERE

- pomocí sekce WHERE jsou filtrovány jen ty řádky, které splňují logickou podmínku (predikát)
- je to obdoba funkcionálu **filter**, jehož filtrovací funkce mapuje n-tici (řádek) na boolovskou hodnotu

  $D_1 \times D_2 \times \ldots \times D_n \longrightarrow \{true, false\}$, kde D_i je doména *i*-tého sloupce

- jednoduchá podmínka:

```
SELECT * FROM vstars WHERE dist < 25 ORDER BY dist;
```

- složená podmínka:

```
SELECT * FROM vstars
    WHERE dist < 25 AND mag BETWEEN 5.0 and 6.0
    ORDER BY dist;
```

Řetězcové vzory

pro filtrování podle řetězcových dat lze kromě relačních operátorů využít i testování řetězcových vzorů. PostgreSQL podporuje tři syntaxe (z nichž dvě jsou SQL standardem)

- **operátor LIKE** (resp. NOT LIKE), ILIKE (case insensitive)
- _ = jeden znak, % - sekvence nula nebo více znaků

  SELECT hr, name FROM starnames WHERE name LIKE 'A%';
  (26 hvězd z 79!, proč?)

  SELECT hr, nameFROM starnames WHERE name ILIKE '%Y%';
  Alcyone, Procyon, Taygete

- **operátor SIMILAR TO** (NOT SIMILAR TO, SQL:1999)
  are a curious cross between LIKE notation and common regular expression notation

- **operátor** ~ (case sensitive), ~* (case insensitive) s POSIX regexp

Řetězcové vzory II

SQL patterns	POSIX regexp
_	.
%	.*
P | P	RE | RE
P *	RE *
P +	RE +
P ?	RE ?
P {m,n}	RE {m,n}
[character-set]	[character-set]

```
SELECT hr, name FROM starnames WHERE name SIMILAR TO '_{4,5}';
```

ekvivalentní zápis pomocí POSIX RE

```
SELECT hr, name FROM starnames WHERE name ~ '^.{4,5}$';
```

Agregační funkce

- agregační funkce se používají v sekci SELECT, na rozdíl od běžných funkcí však výsledky agregují ze všech řádků dané (restringované) tabulky
  - agregační funkce odpovídají funkcionálu *fold*.
- typickým příkladem jsou statistické operace (počet, součet, průměr, apod).

```
SELECT avg(dist) FROM vstars;
-- 410 ly výsledek je nereprezentativní, neboť jen u 87% okem viditelných hvězd je známa vzdálenost

SELECT count(dist) / (count(*)::numeric) FROM vstars;
```

- COUNT(*) počítá počet řádků, COUNT(column) jen řádky, které nejsou NULL

Agregační funkce II

- další statisticky orientované agregační funkce

  sum, max, min

- boolovské agregace

  bool_and (every), bool_or

  SELECT bool_and(mag <= 6.5) FROM vstars;

- kumulativní agregace

  SELECT string_agg(name, ',') FROM starnames WHERE name LIKE 'M%';

  -- "Maia,Markab,Megrez,Menkar,Merak,Merope,Mintaka,Mira,Mirach,Mirfak,Mizar"

- SELECT json_agg(starnames) FROM starnames WHERE name LIKE 'N%';

  "[{"hr":"3165","name":"Naos"}, {"hr":"1829","name":"Nihal"}{"hr":"7121","name":"Nunki"}]"

Agregační funkce III

- komplexnější kumulativní funkce často vyžadují specifikaci uspořádání (běžná sekci ORDER BY nelze použít) a explicitní přetypování (zde na uživatelsky definovaný typ)

  -- CREATE TYPE maghd AS (mag real, hd text);

  SELECT json_agg((mag, hd)::maghd ORDER BY mag ASC)
  FROM vstars WHERE mag < 0;

  [{"mag":-1.44,"hd":" 48915"}, {"mag":-0.63,"hd":" 45348"},
  {"mag":-0.01,"hd":"128620"}]

Spojení tabulek

- nejdůležitější funkcí příkazu SELECT je **spojení dat z více tabulek do jediné (dočasné) tabulky** (na níž může být následně aplikována restrikce a projekce)
- spojovat lze v zásadě jakékoliv dvě tabulky, ale nejčastějším typem **spojení realizující relaci 1:N či N:1** pomocí společného klíče
  - *tabulky vznikly jednotným návrhem* resp. normalizací = společný klíč je explicitně primárním klíčem v odkazované tabulce a cizím klíčem v tabulce odkazující, spojení se děje na základě přesné shody
  - *tabulky vznikly nezávisle* avšak sdílejí společný identifikátor = společný klíč nemusí být primárním či cizím klíčem, měl by však být v jedné tabulce unikátní), shoda nemusí být úplná

Ukázkové tabulky

```
CREATE TABLE "Historie"."Zeme"
(
  id serial NOT NULL,
  jmeno text NOT NULL,
  rozloha real NOT NULL,
  CONSTRAINT "Zeme_pkey" PRIMARY KEY (id)
)
```

1	"Čechy"	52065
2	"Morava"	22349
3	"Slezsko"	4459

```
CREATE TABLE "Historie"."HlavniMesta"
(
  id serial NOT NULL,
  jmeno text NOT NULL,
  id_zeme integer NOT NULL,
  CONSTRAINT "HlavniMesta_pkey" PRIMARY KEY (id),
  CONSTRAINT "HlavniMesta_id_zeme_fkey"
      FOREIGN KEY (id_zeme)
      REFERENCES "Historie"."Zeme" (id)
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

1	"Praha"	1
2	"Brno"	2
3	"Olomouc"	2
4	"Opava"	3

Cizí klíče

- cizí klíče umožňují kontrolu tzv. **referenční integrity** (reference -- použití primárního klíče pro odkaz na záznam/řádek v libovolné tabulce cizí, ale i vlastní)
- *referenční integrita zajišťuje, že v databázi nejsou reference na neexistující záznamy*
- **vkládání odkazujícího záznamu** — reference musí odkazovat na již existující záznam (= musí existovat záznam s daným klíčem)
- **při výmazu nebo úpravě** odkazovaného záznamu musí být upraveny i záznamy odkazující
  - nastavením implicitní hodnoty u odkazujících/závislých záznamů (typicky NULL, jenž zde symbolizuje neexistenci odkazu)
  - výmazem závislých záznamů

Cizí klíče II

odkazující/závislý sloupec (cizí klíč)

FOREIGN KEY (id_zeme) odkazovaná tabulka

REFERENCES "Historie"."Zeme" (id) odkazovaný sloupec

ON UPDATE NO ACTION ON DELETE NO ACTION

akce při změně odkazovaného atributu řádku akce při výmazu odkazovaného záznamu

možné akce:

NO ACTION nebo RESTRICT
akce se neprovede (s možnou produkcí chyby)

CASCADE
závislé záznamy se vymažou

SET NULL
u závislých záznamů (referencí se nastaví NULL)

SET DEFAULT
nastaví se implicitní hodnota (ta musí opět splňovat referenční omezení)

Výmaz a aktualizace záznamu

- při výmazu lze uvést libovolnou podmínku, pokud však má být vymazán jen jeden určitý záznam pak se využívá test na hodnotu primárního klíče

  DELETE FROM "Historie"."Zeme" WHERE id=1;

  ERROR: update or delete on table "Zeme" violates foreign key constraint "HlavniMesta_id_zeme_fkey" on table "HlavniMesta"

  DETAIL: Key (id)=(1) is still referenced from table "HlavniMesta".

- aktualizace má podobnou strukturu (jen je nutné specifikovat daný sloupec/sloupce a jeho/jejich novou hodnotu)

  UPDATE "Historie"."Zeme"
  SET id=5, jmeno='Valašsko', rozloha=2350 WHERE id = 1;

  stejná chyba

INNER JOIN

- základním mechanismem spojení je tzv. vnitřní JOIN podle klíčů (primárního v jedné tabulce a sekundárního ve druhé)
- vytváří tabulku která vznikne spojením řádků obou tabulek (pokud je spojení 1:N pak se údaje jedné z tabulek opakují)

```
SELECT * FROM "Historie"."Zeme" as zeme
                INNER JOIN "Historie"."HlavniMesta" as mesta
                    ON zeme.id = mesta.id_zeme;
```

1	"Čechy"	52065	1	"Praha"	1
2	**"Morava"**	**22349**	2	"Brno"	2
2	**"Morava"**	**22349**	3	"Olomouc"	2
3	"Slezsko"	4459	4	"Opava"	3

INNER JOIN

- zobrazení hodnot klíčů je zbytečné (jsou to jen umělé interní klíče), proto je vhodné je v SELECTU eliminovat
- navíc lze vynechat i klíčové slovo INNER (ze syntaxe je zřejmé, že se jedná o vnitřní spojení)

```
SELECT zeme.jmeno, zeme.rozloha, mesta.jmeno
        FROM "Historie"."Zeme" as zeme
        JOIN "Historie"."HlavniMesta" as mesta
            ON zeme.id = mesta.id_zeme;
```

v případě, shodných jmen klíčových sloupců existují kratší zápisy:

- T1 JOIN T2 USING(jméno sloupce)
  sloupec bude ve výsledku jen jednou
- T1 NATURAL JOIN T2
  spojení přes všechny stejně pojmenované klíče (nikoliv přes cizí klíč!)

CROSS JOIN

- výsledek vnitřního spojení lze interpretovat jako podmnožinu kartézského součinu obou tabulek

  $J \subset T1 \times T2$

- vnitřní spojení je tudíž relací mezi dvěma tabulkami
- v relaci jsou ty řádky z obou tabulek pro které je splněna spojovací podmínka (typicky shoda primárního a cizího klíče)
- proto existuje i alternativní (starší) zápis pro spojení dvou tabulek:

```
SELECT zeme.jmeno, zeme.rozloha, mesta.jmeno
  FROM Zeme as zeme, HlavniMesta" as mesta
  WHERE zeme.id = mesta.id_zeme;
```

FROM sekce se dvěma tabulkami provádí tzv. **CROSS JOIN = kartézský součin tabulek**

id	jmeno	rozloha	id1	jmeno1	id_zeme
1	Čechy	52065	1	Praha	1
1	Čechy	52065	2	Bmo	2
1	Čechy	52065	3	Olomouc	2
1	Čechy	52065	4	Opava	3
2	Morava	22349	1	Praha	1
2	Morava	22349	2	Bmo	2
2	Morava	22349	3	Olomouc	2
2	Morava	22349	4	Opava	3
3	Slezsko	4459	1	Praha	1
3	Slezsko	4459	2	Bmo	2
3	Slezsko	4459	3	Olomouc	2
3	Slezsko	4459	4	Opava	3

CROSS JOIN II

- doporučuji používat raději explicitní syntaxi s použitím operátoru JOIN
  - je přehlednější (především v případě složitějších spojení)
  - jasně odděluje spojení od restrikce
  - pokud není optimalizována, pak může být výrazně pomalejší a náročnější na paměť (ve skutečném spojení nikdy nevzniká celý kartézský součin tabulek s n × m řádky)
  - je dnes podporována všemi databázemi (včetně těch odlehčených jako je SQLite)
- pokud je (výjimečně) potřeb provést křížové spojení, pak použijte explicitní operátor:

```
SELECT *  FROM "Historie"."Zeme" as zeme
          CROSS JOIN "Historie"."HlavniMesta" as mesta;
```

Složitější vnitřní spojení

- úkol -> zobrazit 10 nejjasnějších hvězd, jenž mají vlastní jména
- je potřeba propojit tři tabulky přes různá katalogová čísla

recno	ra	de	hd	sao	pm_ra	pm_de	dist	mag	sp
63	0,08	-44,29	224750	231888	0,08	-0,11	237	6,29	K0
78	0,1	26,92	224758	91648	0,05	-0,05	256	6,44	F8
97	0,13	59,56	224784	35983	-0,16	-0,02	427	6,18	K0
139	0,18	45,25	224801	53568	0,02	0	678	6,36	A0p

hr	hd	sao
1	3	36042
2	6	128569
3	28	128572
4	87	91701
5	123	21085
6	142	214963

hr	name
897	Acamar
472	Achernar
4730	Acrux
2618	Adara

```
SELECT name, mag
    FROM starnames
        NATURAL JOIN hr
        JOIN vstars USING(hd)
    ORDER BY mag
    LIMIT 10;
```

name	mag
Sirius	-1,44
Canopus	-0,63
Vega	0,03
Capella	0,08
Arcturus	0,16
Rigel	0,28
Procyon	0,4
Achernar	0,54
Betelgeuse	0,57
Agena	0,64

Omezení vnitřního spojení

- formulace předchozího úkolu se může jevit jako poněkud přeformalizovaná:

  zobrazit 10 nejjasnějších hvězd, ***jenž mají vlastní jména***

- není to totéž jako zobrazení deseti nejjasnějších hvězd na obloze?
- bohužel nikoliv: třetí nejjasnější hvězda oblohy α Centauri A nemá v naší databázi jmen jméno (existuje sice jméno Toliman, ale to se vztahuje na celou celou dvojhvězdu)
- jak zobrazit deset nejjasnějších hvězd spolu s jménem (pokud ho mají)?

Omezení vnitřního spojení

Na tento úkol **vnitřní spojení nestačí**

- k řádku s katalogovým číslem HR pro třetí nejjasnější hvězdu neexistuje řádek se stejným HR v databázi jmen a **nemůže tak být splněna spojovací podmínka**.

recno	ra	de	hd	sao	pm_ra	pm_de	dist	mag
175304	219,9	-60,83	128620	252838	-7,55	0,48	4,39	-0,01

hr	hd	sao
5459	128620	252838

?

hr	name
5340	Arcturus
5506	Izar

OUTER JOIN

- jediným řešením je použití tzv. vnějšího spojení (OUTER JOIN)
- vnější spojení zahrne do výsledku všechny řádky požadované tabulky bez ohledu na zda mají v druhé tabulce odpovídající řádek (tj. řádek se stejným klíčem)

  řádky, které nemají obraz v druhé tabulce však nemohou být úplné, chybí jim hodnoty ze sloupců druhé tabulky. Tyto sloupce **jsou nastaveny na hodnotu NULL**
- vnější spojení je asymetrické tj. je nutno uvádět jeho směr:

  levé vnější spojení (LEFT OUTER JOIN) vkládá všechny řádky levé tabulky (v pořadí zápisu), preferujte

  pravé vnější (RIGHT OUTER JOIN) spojení vkládá všechny řádky z pravé tabulky

OUTER JOIN

```
SELECT mag,hr, name

FROM vstars
LEFT JOIN hr USING(hd)
NATURAL LEFT JOIN starnames

ORDER BY mag;
```

mag	hr	name
-1,44	2491	Sirius
-0,63	2326	Canopus
-0,01	5459	**NULL**
0,03	7001	Vega
0,08	1708	Capella
0,16	5340	Arcturus
0,28	1713	Rigel
0,4	2943	Procyon
0,54	472	Achernar
0,57	2061	Betelgeuse
...	...	...
6,5	7366	NULL
6,5	7815	NULL
6,5	**NULL**	NULL
6,5	**NULL**	NULL

FULL OUTER JOIN

- v některých případech je nutno ve výsledném spojení zahrnout všechny řádky z obou tabulek tj. sjednotit levé a pravé vnější spojení = úplné spojení
- i v tomto případě jsou sloupce, u kterých neexistuje řádek dané tabulky vyplněny hodnotou NULL
- SELECT mag,vstars.hd, hr.hd, hr
  FROM vstars
  FULL JOIN hr USING(hd)
  WHERE vstars.hd is NULL
  OR hr.hd is NULL;

mag	hd	hd1	hr
NULL	NULL	224960	9090
NULL	NULL	225233	9106
5,53	NULL	NULL	NULL
6,47	NULL	NULL	NULL
5,62	NULL	NULL	NULL
5,48	NULL	NULL	NULL
6,46	NULL	NULL	NULL
6,44	NULL	NULL	NULL
4,4	5516	NULL	NULL
6,44	137785	NULL	NULL

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

Prováděcí plán

Rozsah zpracovávané tabulky 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 -- postupná akomodace

Jak zjistit jaký plán byl použit:

EXPLAIN [ANALYZE] dotaz

Prováděcí plán II

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

```
QUERY PLAN
------------------------------------------------------------------------------------------
 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)
```

Indexy

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

sami však vyžadují jistou **režii** pro své uložení a správu (pokud nejsou nezbytné, neměly by být vytvářeny)

```
CREATE INDEX test1_id_index ON test1 (id);
```

Po vytvoření indexu je jeho používání včetně aktualizace automatické (včetně např. sdružených indexů)

Typy indexů

b-tree - založený na binárních stromech. 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

Vícesloupcové a vypočítané indexy

PostgreSQL podporuje i ne zcela běžné 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));
```

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');
```

Použije se či nepoužije index pro následující dotaz?

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

Jiný praktický příklad:

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

Sekce GROUP BY

- sekce GROUP BY umožňuje seskupovat řádky tabulek podle společných hodnot ve sloupci (sloupcích)
- v základní verzi slouží k vypsání těchto společných hodnot:

```
SELECT sp FROM vstars GROUP BY sp;
```
-- 64 řádků

s
A
B
F
G
K
M
N
O
P
S

- lze samozřejmě použít i funkce nad sloupci:

```
SELECT left(sp,1) FROM vstars GROUP BY left(sp,1);
```

- resp.

```
SELECT left(sp,1) as s FROM vstars
GROUP BY s ORDER BY s;
```

Agregační funkce v GROUP BY

- ve většině případů však potřebujeme další informace z tabulky
- v části SELECT však lze odkazovat **jen seskupované sloupce**
- jedinou (o to však důležitější výjimkou) jsou **agregační funkce,** které v jsou v tomto případě volány jen přes seskupené řádky

```
SELECT left(sp,1) as s, count(*) as pocet,
round(avg(dist)) as prum_vzdalenost
FROM vstars
GROUP BY s
ORDER BY s;
```

s	pocet	prum_vzdalenost
	6	332
A	1938	344
B	1562	624
F	1178	191
G	952	313
K	2785	463
M	413	646
N	17	879
O	31	1194
P	1	
S	1	498

Komplexnější GROUP BY

- SELECT 100*(dist/100)::integer AS distclass, left(sp,1)
  AS sclass, count(*) FROM vstars
  WHERE dist is not NULL and sp is not NULL and sp <> ''
  GROUP BY (dist/100)::integer, left(sp, 1)
  ORDER BY distclass, sclass;

```
WITH ctable AS (
    SELECT 100*(dist/100)::integerAS distclass,
           left(sp, 1) as sclass
    FROM vstars
WHERE dist is not NULL and sp is not NULL
        and sp <> ''
)
SELECT distclass, sclass, count(*)
 FROM ctable
 GROUP BY  distclass, sclass
 ORDER BY distclass, sclass;
```

distclass	sclass	count
0	A	9
0	F	29
0	G	61
0	K	41
100	A	165
100	B	8
100	F	523
100	G	219
100	K	119
100	M	2
200	A	457
200	B	39
200	F	338
200	G	101
200	K	268
200	M	12

Rozklad komplexních dotazů

- rozklad komplexních dotazů do několika úrovní vnořených dotazů je jednou z možností **zpřehlednění** jejich **zápisu** (a někdy i zvýšení jejich přenositelnosti)

  Existují dvě základní možnosti: **CTE** (*Common Table Expression*) a **pojmenované pohledy** (view).

  CTE: umožňují zavádět lokální jména pro výsledky dotazů platné jen v rámci daného výrazu (obdoba konstrukce LET v Lispu):

  ```
  WITH name1 AS (query1), name2 AS (query2), ...
  SELECT ...   -- hlavní dotaz
  ```

- nevýhoda: nejsou všeobecně podporovány (MySQL, SQLite)

Použití CTE pro víceúrovňový SELECT

```
WITH tbl as
    (SELECT left(sp, 1) as s, min(mag) as m
    FROM vstars
    WHERE de > -30.0
    GROUP BY left(sp,1)
    ORDER BY left(sp,1))
SELECT s, m,
    (SELECT name FROM vstars
       JOIN hr USING(hd)
      NATURAL JOIN starnames
    WHERE mag = m)
FROM tbl;
```

s	m	name
	5,63	
A	-1,44	Sirius
B	0,28	Rigel
F	0,4	Procyon
G	0,08	Capella
K	0,16	Arcturus
M	0,57	Betelgeuse
N	4,92	
O	2,78	

- nepřehledné a asi neefektivní, existuje alternativa

Alternativa
(s vlastní agregační funkcí)

```
CREATE OR REPLACE FUNCTION first_agg ( anyelement, anyelement )
RETURNS anyelement LANGUAGE sql IMMUTABLE STRICT AS $$
        SELECT $1;
$$;

CREATE AGGREGATE first (
        sfunc    = public.first_agg,
        basetype = anyelement,
        stype    = anyelement
);

SELECT left(sp, 1) as s, min(mag) as m, first(name ORDER BY mag)
  FROM vstars LEFT JOIN hr USING(hd) NATURAL LEFT JOIN starnames
  WHERE de > -30
  GROUP BY left(sp,1)
  ORDER BY left(sp,1)
```

Pohledy

Pohledy (views) umožňují vytvářet **virtuální tabulky**, které nejsou přímo uloženy v úložišti ale vznikají jako výsledky dotazů.

Pohledy jsou **živé**, tj. reflektují změny v podkladových tabulkách a v některých databázích umožňují skrze sebe zápis do podkladových tabulek (tj. navenek je mnohdy nelze odlišit od skutečných = **persistentních tabulek**)

Vytvoření pohledu spojuje zápis podobný vytvoření tabulky s generujícím dotazem:

CREATE VIEW name (columns) AS SELECT ...

Read-only pohledy podporují všechny běžné SQL databázové systémy. Seznam sloupců je nepovinný (jména mohou být odvozena z výsledku dotazu).

Použití pohledů

Pohledy lze stejně jako CTE použít pro **formální rozklad složitějších dotazů (**je to nástroj **abstrakce** podobně jako procedury v procedurálních jazycích**)**.

Pohledy jsou však trvalé databázové objekty a jsou tudíž od svého vzniku stále viditelné a to ve všech sezeních (výjimkou jsou tzv. dočasné pohledy s prefixem TEMP[ORAL]).

- **Nové možnosti použití:**

  omezení viditelnosti dat tabulky resp. spojení tabulek pro určité uživatele (role) — uživatel vidí jen pohled nikoliv podkladové tabulky

  definice alternativních pohledů na tabulku, tabulky (vhodných např. pro jinou cílovou skupinu)

Modifikovatelné pohledy

- v některých databázích (v PostgreSQL od verze 9.3) lze prostřednictvím pohledů i zapisovat do podkladových databází.

Některá základní pravidla:

- **nelze zapisovat do vypočítaných sloupců**, či výsledků příkazů GROUP BY (není možné rozložit zapisovaná data do tabulek)
- není vhodné zapisovat do sdílených sloupců (např. primárních/cizích klíčů)
- v některých databázích lze zakázat zápis i do ostatních sloupců (PostgreSQL 9.4+)
- **někdy je dost obtížné splnit resp. testovat** všechna **integritní omezení**, která lze zápisem narušit

Materializovaný pohled

- materializované pohledy jsou speciálním typem pohledů, které poskytují **pohled na data v okamžiku vzniku pohledu** (tj. nezobrazují všechny změny) [PostgreSQL 9.3+, Oracle]
- navenek se podobají vzniku persistentních tabulek na základě dotazu:

  CREATE TABLE name AS SELECT ....
- jsou však většinou efektivněji representovány (nemusí vůbec dojít ke kopírování dat: COW) a mohou být updatovány:

  CREATE MATERIALIZED VIEW name AS SELECT ...

  REFRESH MATERIALIZED VIEW name
- **využití**: snapshots

Sekce HAVING

- **HAVING** je poslední klauzule klasického příkazu SELECT
- je využitelná pouze spolu se sekcí **GROUP BY** (v PostgreSQL neplatí, ale HAVING bez GROUP BY není moc užitečné)
- slouží k filtraci výstupů z příkazu GROUP BY (tj. k filtraci seskupených řádků)
- je funkčně obdobná sekci WHERE, tj. i ona obsahuje **predikáty**
- tyto predikáty však musí být buď nad sloupci, podle nichž se seskupuje (méně často) nebo **nad výsledky agregací** (ty nejsou sdíleny se sekcí SELECT)

FROM -> WHERE -> (SELECT + GROUP BY)
-> HAVING -> ORDER BY

HAVING

```
SELECT sp, count(*) as count  FROM vstars
GROUP BY sp
HAVING count(*) > 300
ORDER BY sp
```

sp	count
A0	809
A2	506
B3	361
B8	330
B9	368
F0	377
F5	391
G5	644
K0	2008
K2	374
K5	394

Window function

- **window function** (česky **analytické funkce**) umožňují doplňovat běžné řádky o souhrné 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 agregačních funkcí nevedou však k redukci řádků (tj. nevytvářejí jen souhrnné řádky)
- některé lze emulovat pomocí vnořených dotazů uvnitř sekcí SELECT resp. pomocí samospojení (self-join)

Agregační funkce ve window konstrukci

```
WITH tbl AS (
SELECT name, mag, dist, min(mag) OVER (PARTITION BY CASE
                                                    WHEN de > 30 THEN 1
                                                    WHEN de > -30 THEN 2
                                                    ELSE 3
                                                    END )
FROM  vstars LEFT JOIN hr USING(hd) NATURAL LEFT JOIN starnames)
SELECT * FROM tbl where mag < 1.0 ORDER BY mag;
```

name	mag	dist	min
Sirius	-1,44	8,6	-1,44
Canopus	-0,63	313	-0,63
	-0,01	4,39	-0,63
Vega	0,03	25,3	0,03
Capella	0,08	42,2	0,03
Arcturus	0,16	36,7	-1,44
Rigel	0,28	773	-1,44
Procyon	0,4	11,4	-1,44
Achernar	0,54	144	-0,63
Betelgeuse	0,57	427	-1,44
Agena	0,64	525	-0,63
Altair	0,93	16,8	-1,44
Aldebaran	0,99	65,1	-1,44

RANK, LAG, LEAD

```
WITH tbl AS (
  SELECT name, mag, dist,
      RANK() OVER (ORDER BY mag) AS mag_rank,
      RANK() OVER (ORDER BY
            round((mag-5*(log(dist/3.2616)-1))::numeric)) AS amag_rank,
      RANK() OVER (ORDER BY dist) AS dist_rank,
      LAG(name, 1, '---') OVER (ORDER by mag) as prev,
      LEAD(name, 1, '---') OVER (ORDER by mag) as next
   FROM  vstars LEFT JOIN hr USING(hd) NATURAL LEFT JOIN starnames)
SELECT * FROM tbl where mag < 1.0 ORDER BY mag;
```

name	mag	dist	mag_rank	amag_rank	dist_rank	prev	next
Sirius	-1,44	8,6	1	3803	3	---	Canopus
Canopus	-0,63	313	2	2	3158	Sirius	
	-0,01	4,39	3	7354	1	Canopus	Vega
Vega	0,03	25,3	4	3803	33		Capella
Capella	0,08	42,2	5	1870	100	Vega	Arcturus
Arcturus	0,16	36,7	6	1870	74	Capella	Rigel
Rigel	0,28	773	7	1	6966	Arcturus	Procyon
Procyon	0,4	11,4	8	6799	5	Rigel	Achemar
Achemar	0,54	144	9	49	1096	Procyon	Betelgeuse
Betelgeuse	0,57	427	10	5	4701	Achemar	Agena
Agena	0,64	525	11	5	5650	Betelgeuse	Altair
Altair	0,93	16,8	12	5885	12	Agena	Aldebaran
Aldebaran	0,99	65,1	13	549	250	Altair	Spica

DISTINCT

- použitím klíčového slova DISTINCT v dotazu je možno odstranit **duplicitní řádky**
- použití:
  po komprimační projekci mohou vznikat duplicitní řádky

  ty mohou být ze sémantického hlediska nutné, tolerovatelné nebo nepřípustné

  nepřípustné jsou v případě, kdy řádky identifikují objekty tj. tvoří relační tabulku

  použití klauzule DISTINCT interně vyžaduje třídění, proto ji nepoužívejte není-li nutná

DISTINCT v korelační dotazech

- korelační dotazy obsahují poddotaz SELECT odkazující na sloupec aktuálnho řádku
- jsou relativně pomalé a je lepší se jim vyhnout (window function, spojení)

```
SELECT [DISTINCT] dept_id
FROM departments
WHERE dept_id NOT IN (SELECT dept_id FROM students)
```

- vrací identifikátory kateder, které nemají žádného studenta
- DISTINCT je nutné jen v případě, že dept_id není v tabulce departments unikátní (málo pravděpodobné)

DISTINCT v alternativách korelačních dotazů

- dotaz na neexistenci řádku s cizím klíčem lze provést i pomocí vnějšího spojení (DISTINCT jen není li dept_id unikátní)

```
SELECT [DISTINCT] d.dept_id
FROM depertments AS d
    LEFT JOIN students AS s USING dept_id
WHERE  s.dept_id IS NOT DISTINCT FROM NULL;
```

- opačný dotaz = zobrazující záznamy, které jsou odkazovány z druhé tabulky lze řešit pomocí vnitřního spojení s povinnou sekcí DISTINCT

```
SELECT DISTINCT d.dept_id
FROM departments AS d JOIN students AS s USING dept_id;
```

DISTINCT v jiných kontextech

- DISTINCT lze využít i na několika dalších místech SQL dotazu
- u agregačních funkcích (agreguje se jen přes unikátní řádky)
- při kombinování dotazů pomocí UNION, INTERSECT, EXCEPT

- klazule však není podporována u windowing funkcí!

DISTINCT ON

- PostgreSQL podporuje i klauzuli DISTINCT ON, která odstraní až na jeden všechny řádky, které sdílejí hodnoty ve specifikovaném sloupci nebo sloupcích (resp. použitý výraz pro ně vrací stejnou hodnotu)
- musí být použita sekce ORDER BY se stejným primárním třídícím klíčem
- zůstává vždy první řádek, což je buď řádek náhodný resp. první v uspořádání daném sekundárním klíčem v sekci ORDER by
- funkce je obdobná sekci GROUP BY -- skupina je však representována jedním ze svých členů (prvním v sekundárním uspořádání)

DISTINCT ON — příklad

```
SELECT DISTINCT ON (s) left(sp, 1) as s, mag as m, name
FROM vstars LEFT JOIN hr USING(hd)
        NATURAL LEFT JOIN starnames
WHERE de > -30
ORDER BY s, m
```

třetí možné řešení zobrazení údajů o extrémním objektu ve skupině (po vnořeném dotazu a agregační funkci FIRST)

s	m	name
	5,63	
A	-1,44	Sirius
B	0,28	Rigel
F	0,4	Procyon
G	0,08	Capella
K	0,16	Arcturus
M	0,57	Betelgeuse
N	4,92	
O	2,78	

Kombinace výsledků dotazů

- jednotlivé výsledky dotazů (dočasné tabulky) lze kombinovat pomocí:
- **UNION**

  výsledek vznikne sjednocením obou dotazů (nejdříve řádky prvního a pak druhého)
- **INTERSECT**

  výsledek je průnikem obou dotazů
- **EXCEPT**

  hodnoty druhého výsledku jsou odečteny od prvního (množinový rozdíl)
- u všech může být uvedeno DISTINCT (z výsledku jsou odstraněny duplikáty)

Použití kombinací

- kombinační klauzule lze použít jen pro dotazy se stejným počtem sloupců, které mají navíc stejné domény (jména se mohou lišit)
- kombinace dvou výsledků nad stejnou tabulkou nebo spojení, lze ve většině případů napsat jednodušeji
- spojení dvou nezávislých zdrojů — hodí se tehdy jsou li data ve více tabulkách např. z důvodů rozdělení podle časových úseků

```
SELECT *
FROM sales2014q1
UNION
SELECT *
FROM sales2014q2;
```

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)
- příklad tabulka administrativních jednotek v ČR:

```
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	Homí Blatná	6
11	Bukov	7
12	Klíše	7

Dotazy s fixní úrovní vnořených odkazů

- 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;
```

nadřízená jednotka	podřízená jednotka
ČR	Ústecký kraj
ČR	Karlovarský kraj
Ústecký kraj	okres Ústí nad Labem
Ústecký kraj	okres Litoměřice
Karlovarský kraj	okres Karlovy Vary
okres Ústí nad Labem	Ústí nad Labem
okres Ústí nad Labem	Chlumec
okres Litoměřice	Dobříň
okres Karlovy Vary	Horní Blatná
Ústí nad Labem	Bukov
Ústí nad Labem	Klíše

Rekurzivní dotazy

- problém však přinášely dotazy, které potřebovali pracovat s proměnlivou resp. neomezenou (tj. předem neznámou) úrovní odkazů — ty museli 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)

Rekurzivní CTE

```
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
```

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

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)

Fulltext a PostgreSQL

- 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)

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
- podrobnější informace poskytuje funkce **ts_debug**:

(word,"Word, all letters",Jiří,{simple},simple,{jiří})
(blank,"Space symbols"," ",{},,)
(word,"Word, all letters",Fišer,{simple},simple,{fišer})
(blank,"Space symbols",", ",{},,)
(word,"Word, all letters",Dukelských,{simple},simple,{dukelských})
(blank,"Space symbols"," ",{},,)
(word,"Word, all letters",hrdinů,{simple},simple,{hrdinů})
(blank,"Space symbols"," ",{},,)
(uint,"Unsigned integer",35,{simple},simple,{35})

Tsquery

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

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

  "('kočka' | 'kočko') & 'pes'"::tsquery
- 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', 'kroko:*')

  najde i krok, kroky apod.

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');
```

dotaz je proveden nad tabulkou obsahující všechny odstavce Haškova Švejka:

```
CREATE TABLE svejk
(
  chapter integer NOT NULL,
  paragraph integer NOT NULL,
  content text,
  CONSTRAINT svejk_pkey PRIMARY KEY (chapter, paragraph)
)
```

Komplexnější fulltext dotazy

- 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;
```

ts_headline	rank
Kuchař Pavlíček, když okoušel {maso} s {Balounem}, vylomil si přední zub a {Baloun}	0,05
{maso} z kostí a těšili se pohledem na uvázaného {Balouna}, který stál sice pevně opř	0,03
{Baloun}. „Než mně ho přinesly, zařvala dvě housata, ale to není žádný {maso}, to je	0,01
{Balouna}, aby jí polovičku nesežral. Kromě toho musí Vaněk se Švejkem koupit prase p	0,01

Urychlení fulltextových dotazů

- 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,''));
```

Transakce

- transakce umožňují pro definovanou databázovou operaci (= množinu elementárních databázových příkazů) zajistit:
- **atomičnost**

  Atomicity requires that each transaction be "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged

- konzistenci

  The **c**onsistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof.

Transakce II

- **izolovanost**

  The **i**solation property ensures that the *concurrent execution* of transactions result in a system state that would be obtained if transactions were executed serially, i.e. one after the other.

- **trvalost**

  Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors.

- společně se tyto vlastnosti označují akronymem ACID.

Transakce v SQL

Transakce začíná klíčovým slovem BEGIN:

BEGIN [transaction_mode]

po ukončení transakce je nutno transakci buď potvrdit (všechny změny se projeví)

COMMIT

- nebo ji celou zrušit (žádná změna se neprojeví)

  ROLLBACK

  stejná situace nastane v případě neošetřené chyby (výjimky)

- u příkazů, které nejsou uvnitř explicitní transakce se uplatňuje autotransakční mód — každý příkaz tvoří vlastní transakci

Izolační úrovně

- dosažení plné izolovanosti není zadarmo, zvyšuje se režie a při vyšším zatížení databáze (hlavně zápisy) může dojít ke zdánlivému uváznutí (deadlocku). Transakce na sebe čekají tak dlouho, že doba odezvy přestává být akceptovatelná (i když je vždy, alespoň teoreticky konečná)
- standardní SQL proto nabízí tzv. izolační úrovně, které nabízejí nižší míru izolace za cenu časové a prostorové efektivity
- tyto režimy jsou definovány na základě negativních projevů, kterým daná úroveň (a úrovně nižší) nezabraňuje
- detailní chování však závisí na implementaci. Dvě hlavní:
  - dvojfázové zamykání (klasické řešení předpokládané ve standardu)
  - Multiversion concurrency control (použité v PostgreSQL)

Read uncommitted

- de facto bez izolace transakcí
- není prováděna žádná serializace
- lze číst i nepotvrzené změny jiné transakce (*dirty read*)
- v PostgreSQL s MCC není podporována

Transaction 1

```
/* Query 1 */
SELECT age FROM users WHERE id = 1;
/* will read 20 */



/* Query 1 */
SELECT age FROM users WHERE id = 1;
/* will read 21 */
```

Transaction 2

```


/* Query 2 */
UPDATE users SET age = 21 WHERE id = 1;
/* No commit here */



ROLLBACK; /* DIRTY READ */
```

Read committed

- základní úroveň izolace (nejnižší v PostgreSQL)
- lze vidět jen potvrzená (commited) data
- lze však vidět změny, které jsou výsledkem transakcí, které byly spuštěny později (tj. čtení se mohou lišit) — **nekonzistentní opakovaná čtení**
- efektivní a snadno se používá, vhodné pro jednodušší transakce

Transaction 1

```
/* Query 1 */
SELECT * FROM users WHERE id = 1;



/* Query 1 */
SELECT * FROM users WHERE id = 1;
COMMIT;
```

Transaction 2

```


/* Query 2 */
UPDATE users SET age = 21 WHERE id = 1;
COMMIT;
```

Repeatable reads

- nepříliš často používaná úroveň
- eliminuje nekozistentní vícenásobná čtení u jednotlivých řádků, stále však neposkytuje plně konzistentní stav
- problém: fantómové čtení při provádění příkazů nad více řádky

Transaction 1

```
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;



/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
COMMIT;
```

Transaction 2

```
/* Query 2 */
INSERT INTO users VALUES ( 3, 'Bob', 27 );
COMMIT;
```

- PostgreSQL eliminuje na této úrovni fantómové čtení, nezajišťuje nicméně plnou serializovatelnost

Serializable

- **úplná izolace transakcí**
- transakce jsou provedeny jako by byly provedeny v pořadí svých aktivací jedna za druhou (sériově)
- z důvodů efektivity jsou však prováděny, pokud možno, paralelně
- to může vést:
  - k dlouhému čekání (falešnému deadlocku) při 2-phase locking
  - k vyvolání výjimky, pokud se nenajde prováděcí plán (MCC); transakce je anulována a musí být provedena znova (explicitně)

```
while True:
  try:
    cursor.execute("UPDATE ...")
    break
  except TransactionRollbackError:
    continue #restartování transakce
```

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 automatické údržbu databáze
  - obslužné rutiny triggerů (aktivovány při vkládání, změně a výmazu tabulek)

Programové prostředky uložených funkcí

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 2 i 3 , PL/Tcl, PL/Perl)
  — využití předchozích znalostí + jednoduché API
- **programovací jazyk C**
  — nízkoúrovňový kód, super rychlé

Definice nové 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;
```

Parametry funkcí

- 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.

- alternativně lze namísto pojmenovaných parametrů používat číselně-poziční (před verzí 9.0 to byla jediná možnost)

```
CREATE FUNCTION add(integer, integer)
    RETURNS integer AS $$
        SELECT $1 + $2;
    $$ LANGUAGE SQL;
```

Volání funkcí

- 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);
```

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)

```
CREATE TYPE FRACTION AS (nom INTEGER, denom INTEGER);
```

```
-- 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;
```

Volání funkcí nad složenými objekty

- 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));

Funkce vracející množinu hodnot

- 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;
```

- ale množiny lze generovat i dynamicky:

```
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);
```

Funkce vracející tabulku

- pokud je definován typ cílové tabulky (datovým typem 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;
```

Použití funkcí vracející množinu/tabulku

- základní použití je v sekci FROM dotazů (resp. jako zdroj u jiných SQL příkazů)
- pokud je funkce použita spolu s dalším zdrojem a závisí na hodnotě sloupce = tj. funkce produkuje více řádků lze využít konstrukci LATERAL

  (konstrukci lze použít i u jiných zdrojů, ale zde je většinou vhodnější použít spojení)

```
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;
```

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 (i 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 spojit do jediné funkce. Patří sem i časové funkce typu current_time()
- IMMUTABLE = vždy vrací stejné hodnoty pro stejné parametry

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)

Ukázka PL/pgSQL

- typické je explicitní oddělení deklarační části a těla funkce
- funkci v ukázce lze representovat i v čistém SQL

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

Řídící konstrukce

- i když jsou řídící konstrukce podobné jiným procedurálním jazykům, přece jen se projevuje zaměření na zpracování tabulek nikoliv skalárů

  lze například postupně vracet jednotlivé části výstupní tabulky pomocí RETURN NEXT (po řádku) nebo RETURN QUERY (po segmentech).

```
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;
```

Klíčové řídící konstrukce

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

```
CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
      statements
  [ WHEN expression [, expression [ ... ]] THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

```
EXIT [ label ] [ WHEN boolean-expression ];
CONTINUE [ label ] [ WHEN boolean-expression ];
```

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

Kursory

- kursory jsou jednou z možností jak procházet výsledky dotazů
- na rozdíl od ostatních vždy vracejí řádky postupně (jeden po druhém) a to i tehdy, jsou-li využity na straně klienta

```
DECLARE
    curs1 refcursor; -- unbound cursor
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR 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
CLOSE curs1;
```

Výjimky (chyby)

- mechanismus výjimek je obdobný ostatním jazykům, chybí však společná abstrakce objektu výjimky, informaci o výjimce může nést:
  - řetězec
  - předdefinované *condition name*
  - předdefinované SQLSTATE 'číslo'

```
RAISE unique_violation
   USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

- některé výjimky lze v PL/pgSQL zachycovat

```
BEGIN
   statements
EXCEPTION
   WHEN condition [ OR condition ... ] THEN
      handler_statements
   WHEN ...
END;
```

Ostatní skriptovací jazyky

- je nutné je zavést v rámci každé databáze, kde jsou použity

  CREATE EXTENSION plpythonu;
- pak už je použití podobné jako u SQL či pgSQL
- jednoduché hodnoty jsou mapovány do obdobných pythonských typů, záznamy (řádky) do slovníků

```
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;
```

```
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;
```

Přístup k tabulce

```
CREATE FUNCTION count_odd_iterator() RETURNS integer AS $$
odd = 0
for row in plpy.cursor("select num from largetable"):
    if row['num'] % 2:
        odd += 1
return odd
$$ LANGUAGE plpythonu;

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;
```

Transakce

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
try:
    with plpy.subtransaction():
        plpy.execute(
            "UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
        plpy.execute(
            "UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

Klientské knihovny