Úvod do relačních databází

Jiří Fišer

Ústí nad Labem 2020

Kurz: Úvod do relačních databází

Obor: Aplikovaná informatika

Klíčová slova: databázové systémy, logický a fyzický návrh databáze, relační model, SQL.

Anotace: Úvodní kurz databázových systémů se zaměřením na návrh a implementaci efektivních databází za použití relačních databázových systémů a jazyka SQL.

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

© Katedra informatiky, PřF, UJEP v Ústí nad Labem, 2020

Autor: Jiří Fišer

Obsah

1 Základní principy databází a vývoj databázových systémů **5**
1.1 Starší modely — východisko . . . 5
1.2 Relační databázové systémy . . . 5

2 Konceptuální model dat (ER modelování) **7**
2.1 Entity a relace . . . 7
2.2 Atributy . . . 11

3 Relační model dat (logický model) **13**
3.1 Relační model . . . 13
3.2 Vztah mezi ER modelem a relačním modelem . . . 13
3.3 Kandidátní klíče . . . 14
3.4 Relační vztahy a cizí klíče . . . 14
3.5 Rozklad relací N:M . . . 15

4 Normalizace v relačním modelu dat **18**
4.1 Normalizace . . . 18
4.2 První normální forma (1.NF) . . . 19
4.3 Druhá normální forma (2.NF) . . . 21
4.4 Třetí normální forma (3.NF) . . . 23
4.5 Další normální formy . . . 25

5 SQL — základy **27**
5.1 Co je SQL? . . . 27
5.2 Standardizace SQL . . . 27
5.3 Základní pravidla syntaxe . . . 28
5.4 Domény . . . 28

6 SQL – DDL, DML **38**
6.1 Definice tabulky . . . 38
6.2 Běžné vkládání dat . . . 42
6.3 Hromadné vkládání . . . 43
6.4 Aktualizace (UPDATE) . . . 43
6.5 UPSERT (spojení vkládání s aktualizací) . . . 44
6.6 Výmaz dat . . . 45

7 SQL – DQL **47**
7.1 Základní tvar příkazu SELECT . . . 47
7.2 Sekce SELECT . . . 48
7.3 Sekce WHERE . . . 50
7.4 Sekce ORDER BY . . . 52

8 SQL – representace relačních vztahů a agregace **56**
8.1 Cizí klíče . . . 56

8.2 Vnitřní spojení (INNER JOIN) . . . 57
8.3 Křížné spojení (CROSS JOIN) . . . 63
8.4 Vnější spojení (OUTER JOIN) . . . 63
8.5 (SELECT) DISTINCT . . . 67
8.6 Vertikální spojení — množinové operace . . . 68
8.7 Agregační dotazy . . . 69

9 Transakce a triggery, zotavení z chyb 77
9.1 Transakce . . . 77
9.2 Triggery . . . 80

Seminární úkol

Seminárním úkolem je návrh a implementace databáze pro danou problémovou doménu. Problémová doména je individuální pro každého studujícího a bude stanovena v průběhu kurzu.

Navržená databáze musí splňovat minimálně tyto podmínky:

- obsahovat minimálně čtyři třídy entit (= hlavních tabulek)
- všechny entity musí být propojeni relačními vztahy
- musí obsahovat alespoň jeden rozložený relační vztah M:N
- musí splňovat alespoň 3. normální formu

Návrh musí být vyjádřen pomocí E-R diagramu.

Implementace musí obsahovat minimálně:

- vytvoření tabulek s explicitní specifikací integrálních omezení (klíče)
- datový obsah tabulek (každá tabulka by měla mít alespoň 20 řádků, alespoň jedna 50 řádků)
- výpisy spojených tabulek (podle relačních vztahů)
- kombinování data pomocí GROUP BY a agregační funkce (podle vhodně zvoleného klasifikačního výrazu)

Zadání může definovat i další dodatečné minimální podmínky, které zajistí alespoň základní využitelnost databáze v dané problémové doméně.

Klíčové pojmy

klíčový pojem

Pojmy uvedené na levém okraji textu (a v textu zvýrazněné tučně) jsou tzv. **klíčové pojmy**. Jejich správné a plné pochopení je nezbytné pro další studium. Je samozřejmou součástí zkoušek (a to i zkoušek navazujících předmětů resp. státní závěrečné) a jejich znalost se předpokládá i v rámci obhajoby seminární práce.

Většinu z nich můžete najít v doporučené literatuře a jsou popsány i v anglické *Wikipedii* (anglický překlad je uveden, s výjimkou termínů, kde je zřejmý). V oblasti informatiky jsou články anglické Wikipedii (ve většině případů) velmi kvalitní, s rozsahem přesahujícím popis uvedený v opoře a tak mohou být využity k dalšímu zpřesnění prohloubení znalostí. Navíc obsahují odkazy na další hodnotné zdroje.

1 Základní principy databází a vývoj databázových systémů

1.1 Starší modely — východisko

První databáze (včetně příslušných modelů) vznikaly v padesátých a šedesátých letech především v IBM a 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, apod.

1.2 Relační databázové systémy

databázový systém

Databázový systém (přesněji DBMS = *database management system*, česky SŘBD = systém řízení báze dat) se skládá z několika částí:

- úložiště (*data storage*, ukládají se data i metadata), úložiští může být i více a mohou být distribuována
- výkonné jádro zajišťující průběžný přístup a konzistenci databáze
- subsystém zajišťující komunikaci s klienty
- administrativní aplikace

databáze

Je nutno odlišovat pojem databázový systém (DBMS, resp. SŘBD) a databáze. **Databáze** je organizovaná kolekce dat, která je jednotně spravována a obsahuje vzájemně provázaná data resp. metadata. Databáze jsou uloženy v úložištích (v jednom úložišti může být více databází a databáze může být uložena ve více úložištích). Typicky jsou v databázi uložena data jedné aplikace nebo služby.

Relační databázové systémy (RDBMS) DBMS podporující primárně relační databázový model (okrajově však mohou podporovat i jiné modely).

Základní klasifikace databázových systémů:

databázové systémy typu klient-server

Výkonné jádro DBMS tvoří (alespoň) jeden oddělený proces (typicky na dedikovaném serveru) a klienti se připojují přes IPC (typicky TCP) pomocí proprietárního protokolu.

Příklady:

relační: PostgreSQL, MySQL, DB2, Oracle, MS SQL server,

nerelační: Mongo, Oracle NoSQL

vestavěné DBMS

Výkonné jádro je tvořeno (dynamickou) knihovnou, jež je součástí klienta. Úložištěm typicky bývají soubory (databáze = soubor).

Příklady:

relační:	SQLite, HyperSQL
nerelační:	Berkeley DB

2 Konceptuální model dat (ER modelování)

Konceptuální model databáze umožňuje definovat strukturu databáze bez zohlednění její fyzické realizace a dokonce i architektury databázového systému a použitého modelu representace,

Hlavním představitelem konceptuálního modelování je tzv. **entitně-relační modelování** (zkráceně ER modelování).

Entitně relační modelování začíná určením důležitých dat (tzv. entit)i. Další fází je určení vztahů mezi entitami tzv. *relací*. Posledním krokem je stanovení tzv atributů entit.

Entitně-relační model je typicky representován graficky v podobě spojového diagramu. Bohužel se v této oblasti plně neprosadil žádný z několika standardů. V praxi se však pravděpodobně setkáte s jednou z následujících notací:

UML E-R diagram = specializace třídního diagramu pro representaci E-R vztahů

Chenova notace = nejstarší notace

notace vraních stop = populární mezi uživateli SQL databází

Mnohé modelovací nástroje navíc podporují vlastní rozšíření či dokonce vlastní grafickou representaci. Běžné je také prolínání s logickým či dokonce fyzickým návrhem (mnohé modelovací nástroje jsou úzce propojeny s SQL databázovými systémy včetně jednocestné či dvojcestné synchronizace s SQL kódem).

V rámci této opory využívám UML notaci, která vychází z notace třídních digramů. Se základy této notace jste se seznámily již kursu PGL2. Jako alternativní zápis uvádím notaci vraních stop.

2.1 Entity a relace

entita

Entita je množina objektů se stejnými vlastnostmi.

Definice entity je blízká definici *třídy* v OOP. Klade však důraz na (statické) vlastnosti a nikoliv na (dynamické) chování. Obecně tak nemusí existovat vztah 1:1 mezi třídami a entitami.

Entity lze stejně jako třídy v OOP pojmenovat pomocí substantiva nebo substantivní skupiny v jednotném čísle (s velkým počátečním písmenem). V praxi se však dává přednost pojmenování v množném čísle (plurálu), neboť to je úzus u relačních tabulek (a entity se v relačním modelu representují pomocí tabulek).

Entity se ve všech výše uvedených notacích označují pomocí obdélníku s vepsaným názvem entity.

relace

Relace vyjadřuje vztah mezi objekty různých entit.

Relace může vyjadřovat vztah kompozice (je součástí koho/čeho), vlastnictví (je vlastněn kým), obsluhy (je obsluhován kým), umístění (je umístěn v čem), tvorby (je vyroben, vytvořen kým) apod.

V objektově orientovaném návrhu se relace označují termínem *asociace*.

Pojmenování relací: relace se běžně označují verbální frází (= slovesem + příslovcem + předložkou), jež vyjadřuje příslušný vztah mezi objekty. Tyto fráze jsou následujícím přehledu zvýrazněny tučně.

Příklady relací:

- dokument **je vydán** úředníkem (entity *Dokumenty* a *Úředníci*)
- úředník **je jmenován** dokumentem (entity *Dokumenty* a *Úředníci*)
- výrobek **je umístěn ve** skladu (entity *Výrobky* a *Sklady*)
- lékař **se (dlouhodobě) stará o** pacienty (entity *Lékaři* a *Pacienti*)
- lékař (jednou) **ošetřil pacienta** (entity *Lékaři* a *Pacienti*)
- IP adresa **má přidělenu** DNS adresu (resp. více adres), entity *IP_adresy* a *DNS_adresy*
- myš **je připojena** k počítači (entity *Počítače* a *VstupníZařízení*)
- myš je koupena spolu s počítačem a **zaplacena na** jedné faktuře (entity *Zboží*, *Faktury*)
- cestující **využívá** vlak **na trase z** X **do** Y (relace mezi třemi entitami *Cestující*, *Vlaky* a *Stanice*, přičemž stanice jsou v relaci použity dvakrát ve dvou různých rolích)

stupeň relace

Podle počtu entit zúčastněných v relaci je určena tzv. **stupeň relace** (též arita relace). Jedna entita může být použita v relaci i vícekrát v různých rolích. V tomto případě se započítává každá její role. Poslední příklad relace využívá jen tří entit (*Cestující*, *Vlaky* a *Stanice*) jedná se však o relaci se stupněm 4 (tj. je tzv. kvartérní), neboť stanice se v ní vyskytuje ve dvou rolích (počáteční a koncová stanice).

binarní relace

Nejčastěji se vyskytují tzv. **binarní relace** tj. relace mezi dvěma entitami. Relace s vyšším stupněm lze převést na binární relace zavedením pomocných entit (viz dále).

rekurzivní relace

Speciálním případem jsou binární relace mezi objekty stejné entity tzv. **rekurzivní relace**. Tato relace může být symetrická, ale častěji se role obou objektů liší.

Příklady rekurzivních relací:

- osoba **je rodičem** jiné osoby (rekurzivní v entitě *Osoby*, role jsou odlišné)
- osoba **je přítelem** jiné osoby (rekurzivní v entitě *Osoby*, role jsou symetrické)
- kniha **odkazuje** jinou knihu (rekurzivní v entitě *Knihy*)

Notace:

Binární relace jsou v případě UML notace (a i notace vraních stop) znázorněna jednoduchou hranou spojující příslušné entity. Reflexivní relace jsou representovány smyčkou. Relace by měla být pojmenována (v UML lze znázornit i směr a tím určit, co je objektem resp. subjektem dané verbální fráze).

U reflexivních relací je mnohdy vhodné kromě názvu relace uvést i role obou konců relace (to lze provést jen u UML notace).

Relace musí mít v E-R notaci povinně vyjádřenu tzv. multiplicitu tj. praktické ukázky jsou uvedeny až v následující podkapitole.

2.1.1 Multiplicita (násobnost v relací)

Důležitou charakteristikou relace je tzv. multiplicita relací tj. stanovení počtu objektů, které mohou vstupovat do do relačního vztahu na obou koncích (binární) relace.

Relace 1:1

U relací 1:1 vstupuje do relačního vztahu vždy jen jeden objekt na každé straně binární relace, tj. relace jednoznačně svazuje právě dva objekty obou entit (v matematické terminologii to je vzájemně jednoznačné zobrazení mezi množinami obou entit).

povinná participace

Jako relace 1:1 se běžně (resp. dokonce běžněji) označují i relace, které nevyžadují **povinnou participaci** objektů na relaci tj. i zobrazení z množiny resp. do množiny. V případě těchto relací mohou existovat objekty, které nejsou součástí relace (a nemají tudíž svůj obraz v množině druhé entity).

Příklady relací 1:1:

- občan a občanský průkaz (platný) — participace obou stran v relaci je povinná (nemůže existovat občan bez občanského průkazu a občanský průkaz bez občana)
- občan a cestovní pas (platný) — zde už není plná povinná participace (existují občané bez vydaného cestovního pasu, avšak neměl by existovat pas bez občana)
- stát a nejvyšší představitel — obecně to však nemusí být vztah 1:1, existovaly i státy s kolektivním vedením a stále existují např. San Marino.

Relace 1:1 se v databázových modelech vyskytují řidčeji, než by bylo lze očekávat. V mnoha případech lze sjednotit obě entity spojené relací 1:1 do jedné (entita pak sdružuje spojené objekty). V mnoha případech tak lze sjednotit občana s jeho občanským průkazem (objekt občana pak obsahuje i číslo občasného průkazu). Výjimkou je samozřejmě databáze úřadů spravujících občanské průkazy (především jejich vydávání). Podobně lze často sjednotit záznam státu se záznamem o jeho nejvyšším představiteli. je tomu tak v případě, že databáze uvádí jen jméno nejvyššího představitele. Naopak nelze očekávat implementaci relace nejvyšší představitel mezi entitou *Státy* (či jinou vyšší organizační jednotkou) a *Občané*, neboť tato relace není spojuje jen jediného občana (z mnoha tisíců či dokonce miliónů) s objektem entity *Stát* (tj. participace z této strany není jen nepovinná je přímo nepravděpodobná).

Relace 1:N

Relace typu 1:N jsou nejběžnějšími relacemi v relačních databázových modelech. Jednomu objektu jedné entity může odpovídat více objektů v druhé entitě (mají více obrazů), ale každý objekt v druhé entitě je spojen jen s jedním objektem v první (má tedy jen jeden vzor). Možný je samozřejmě i opačný pohled (N:1) — každý objekt první entity je spojen jen s jedinou instancí druhé entity, tato instance však může být spojena s více instancemi první entity (tj. je to zobrazení, které není prosté tj. obraz může mít více vzorů). Ve většině případů je tento pohled přirozenější.

Příklady entit 1:N:

- vstupní zařízení jsou připojena jen k jednomu počítači (počítač však může mít větší počet vstupních zařízení)
- vstupenka patří jen jednomu člověku, člověk však může vlastnit více vstupenek (to však neplatí pro některé vstupenky na jméno s relací 1:1)
- výrobek může být (v jednom okamžiku) umístěn jen v jediném skladu
- člověk může mít jen dva biologické rodiče ty však mohou mít více potomků (tato relace je zdánlivě typu 2:N, lze ji však snadno převést na dvě relace 1:N)
- kružnice má jen jeden střed, tento bod však může být středem více kružnic
- každé těleso ve sluneční soustavě obíhá kolem nejvýše jednoho centrálního tělesa

I v případě relací 1:N lze uvažovat o povinné či nepovinné participaci a to v obou směrech. Například vstupní zařízení v daném modelu nemusí být připojen k žádnému počítači a počítač

nemusí mít připojena žádná vstupní zařízení (např. u *headless* serverů). Dítě má ve většině případů určenou matku či otce (nemusí to být biologičtí rodiče, když i zde existují výjimky), běžnější je však situace, kdy člověk nemá žádné dítě (participace je dobrovolná). Kružnice však má vždy střed a každý bod může být středem libovolného počtu kružnic (potenciálně nekonečného, avšak v databázi je samozřejmě jen konečný počet kružnic).

Relace M:N

Všechny ostatní relace jsou relacemi typu M:N. V těchto relacích je jedna instance dané entity spojena s více instancemi druhé entity a totéž platí i v opačném gardu (více = neomezeně či alespoň ve větším počtu).

V běžném modelu světa se tyto relace vyskytují velmi často (lze dokonce říci nejčastěji). Není proto problém uvést četné příklady:

- učitel vyučuje více předmětů, jeden předmět je vyučován více učiteli
- člověk má více přátel (symetrická reflexivní relace, tj. i Váš přítel má více přátel)
- vydavatelství může vydat více knih, některé knihy mohou mít více vydavatelů (spojené edice).
- výrobek dodává více dodavatelů, kteří mohou jej mohou dodávat více odběratelům

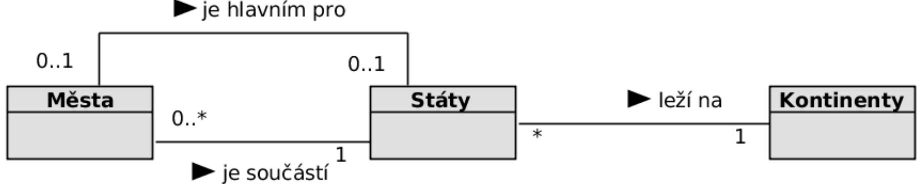
V praxi se však tyto relace vyskytují v E-R modelech jen relativně řídce, neboť se obtížně implementují (a to jak v relačních databázích tak objektových systémech). Proto se v druhé fázi návrhu (logickém) musí převést na relace typu 1:N (což lze provést téměř mechanicky). Většina návrhářů se však preferuje relace 1:N i na konceptuální úrovni (tj. už mají mírně deformovaný pohled na svět) a proto do modelu automaticky zahrnuje i příslušné mezitabulky (především pokud mají reálný podklad).

Už ve fázi konceptuálního návrhu se proto namísto relace dodavatelé vers. odběratelé uvažuje relace dodavatel : dodávka (1:N tj. dodavatel má běžně více dodávek, ale každá dodávka je spojena jen s jedním dodavatelem) a relace dodávka : dodavatel (N : 1).

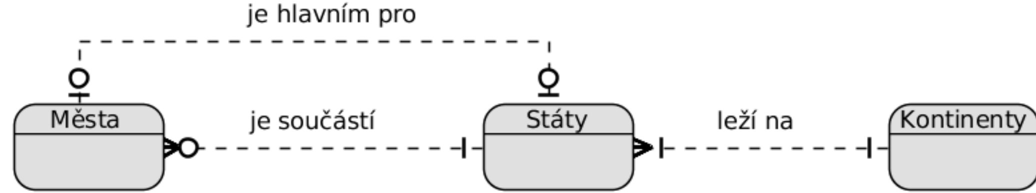
I v případě relací M:N má smysl uvažovat druh participace na vztahu (povinná vesr. nepovinná). Jako příklad uveďme vztah učitel : předmět. Učitel by měl učit alespoň jeden předmět (i když to nemusí být vždy pravda) a předmět musí být vyučován alespoň jedním učitelem (což je ve většině modelů pravděpodobně splněno). Na druhou stranu mohou existovat lidé bez přátel (alespoň v daném databázi).

Notace

V UML notaci se multiplicita označuje stejně jako multiplicita asociací v OOP tj. uvedením intervalu na příslušném konci relace (čte se ve směru k bližší entitě). Dolní mez může být 1 (povinná participace) resp. 0 (nepovinná participace). Horní mez je 1 nebo nekonečno (vyjádřená znakem „*“). Je-li dolní mez 1, pak ji lze vynechat.



V notaci vraních noh je E-R diagram obdobný:



V této notaci jsou multiplicity naznačeny grafickými symboly na konci relací. U entity s multiplicitou N (tj. entity, jejíž instance se mohou v relaci vyskytnout i vícekrát) je čára rozštěpena do vraního nohy (pařátku). Strana, jejíž instance se v relaci objevuje povinně je označena krátkou kolmicí (viz například relace *Státy* – leží na – *Kontinenty*), strana s nepovinnou participací je označena malým kroužkem (obě strany u relace *Města* – je hlavním pro – *Státy*). Lze si to snadno zapamatovat: rozdělení spojnice v pařátku jasně symbolizuje vícečetnost. Kolmá čárka evokuje číslici jedna (relace se musí zúčastnit alespoň jedna instance), kroužek pak nulu (stačí i nula instancí).

2.2 Atributy

atribut

Posledním krokem konceptuální fáze návrhu je specifikace **atributů entit**. Instance entity mají navenek stejné vlastnosti a tak musí mít i stejné atributy.

Atribut popisuje navenek viditelné vlastnosti instancí entit, jejichž hodnoty je nutno uchovávat v databázi. Atributy jednotlivé instance buď jednoznačně určují nebo je jejich znalost nutná při zpracování dat.

Atributy nemají přímou obdobu v běžných OOP metodologiích. Nejvíce se podobají *datovým členům*, neboť určují representaci objektů v databázi. Na rozdíl od datových členů *jsou však svou podstatou veřejné* a mohou být tudíž přímo využívána vnějšími operacemi (databázový koncept nezná princip zapouzdření). Z hlediska viditelnosti jsou obdobnější konceptu (veřejných) vlastností, na rozdíl o nich je však důraz kladen na přímé ukládání (některé E-R modely podporují i tzv. vypočítané atributy či atributy složené, v praxi se však téměř nepoužívají, neboť SQL je přímo nepodporuje).

Atributy jsou jsou pojmenovávány substantivními frázemi (s malým počátečním písmenem).

Příklad atributů:

entita *Studenti*

atributy: jméno, ročník, kód studenta, studovaný obor

entita *Linky_MHD*

atributy: číslo, typ dopravního prostředku (autobus/trolejbus/tramvaj), počáteční zastávka, koncová zastávka

V této fázi návrhu se běžně ještě nestanovuje doména tj. množina přípustných hodnot atributu.

V grafické notaci se v případě UML i vraních noh zapisují jako seznam v obdélníku entity pod seznamem (v UML bez specifikace viditelnosti, ta je vždy veřejná).

OTÁZKY

1. Jaký je vztah mezi entitou (konceptuální databázový model) a třídou (OOP model)?
2. Jak multiplicita relace ovlivněna povinnou participací?
3. Proč mají instance jedné entity stejné atributy?

OTÁZKY K ZAMYŠLENÍ

1. Uveďte příklady několika relací typu 1:1, 1:N a 1:M.
2. Jak jsou zapisovány relace s vyšším stupněm v E-R diagramech?
3. Uveďte příklad rozkladu relace s vyšším stupněm na relaci binární?

ÚKOLY

1. V UML notaci a notaci vraních noh vytvořte diagram databáze s třemi až pěti propojenými tabulkami (například Kniha-Autor-Nakladatel, VlakovéSpojení-Cestující-Stanice, Student-StudijníObor-Ročník, apod.)

3 Relační model dat (logický model)

3.1 Relační model

Relační model byl zaveden v roce 1970 E.F. Coddem ve článku: *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.

tabulka

Základní konstrukcí je **relace** (označení v modelu, standardní matematický význam tj. podmnožina kartézského součinu) resp. **tabulka** (označení v SQL implementaci).

Tabulka se sestává z posloupnosti uspořádaných **(datových) 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)

Jednotlivé položky n-tic (tj. **sloupce** tabulky) jsou označovány jako **atributy**. Jednotlivé atributy mohou nabývat hodnoty z určité (omezené) množiny tzv. domény (obdoba datových typů známých z jiných programovacích jazyků). Domény lze definovat i pro skupiny provázaných sloupců (obdoba záznamů či struktur), i když podpora těchto tzv. složených domén v SQL je jen omezená.

Příklad: sloupce zeměpisná délka a zeměpisná šířka nabývají hodnot složené domény sférických souřadnic.

Funkce domén v relačním modelu:

optimalizace úložiště — specifikace domény umožní zvolit optimální formát pro uložení dat

omezení množiny přípustných operací — SQL je staticky typovaný jazyk, tj. použití operace, jež není podporována, vede k syntaktické chybě

dodateční sémantika — volba domény určuje základní interpretaci dané hodnoty

Tyto funkce jsou obdobné funkci datových typů a tříd v OOP (tam je však důraz na jednotlivé funkce právě opačný)

3.2 Vztah mezi ER modelem a relačním modelem

Vztah mezi ER modelem a relačním modelem je relativně přímočarý. Situaci však trochu komplikuje terminologie (různé použití slova *relace*).

Entity nacházejí svůj obraz v tabulkách (= relacích relačního modelu). Všechny objekty dané entity mají stejné atributy a tak jsou representovány jako záznamy příslušné tabulky.

Některé tabulky však nejsou obrazem entit, neboť vznikají až ve fázi logického návrhu: jsou to např. spojovací tabulky vzniklé při rozkladu relací typu N:M a dále tabulky vzniklé ve fázi normalizace.

Identita objektů (= řádků tabulek) vychází z existence atributů, které nabývají unikátních hodnot pro každý objekt. Tyto atributy se označují jako kandidátní klíče a hrají v relačním modelu klíčovou roli.

Relace mezi entitami jsou representovány pomocí tzv. relačními vztahy, které využívají mechanismus tzv. cizích klíčů.

3.3 Kandidátní klíče

Kandidátní klíče jsou sloupce či skupiny sloupců, které

1. jednoznačně identifikují každý záznam tabulky (tj. jsou pro každý řádek unikátní)
2. tvoří tzv. minimální množinu tj. každá podmnožina sloupců není kandidátním klíčem

Atributy, které jsou obsaženy v alespoň jednom kandidátním klíči se nazývají *klíčové*, ostatní jako *neklíčové*.

Identifikace kandidátních klíčů je relativně snadná, *musí však vycházet z modelu* nikoliv z reálných či ukázkových dat. Obě podmínky v definici kandidátního klíče mohou být narušeny při přidávání nebo změně řádků (přidáním řádku se skupina sloupců může stát kandidátním klíčem, resp. může o tento status přijít). Model naproti tomu pracuje s představou, že databáze obsahuje údaje o všech potenciálních entitách.

Již v této fázi návrhu je jeden z kandidátních klíčů označen jako **primární klíč**. V zásadě to může být libovolný kandidátní klíč, v praxi se však dává přednost jednosloupcovým kandidátním klíčům číselného typu resp. se dokonce takovýto sloupec dodá (tzv. *umělý primární klíč*). Detaily viz 6.1.2 (fyzický návrh).

3.4 Relační vztahy a cizí klíče

relační vztah

Relační vztahy (angl. *relationship* zkrácené vztah či relace[1]). Relační vztahy jsou binárními relacemi tj. jsou definována mezi dvěma tabulkami Obecně jsou to dvě různé tabulky, ale existují i relace v rámci jediné tabulky. Tyto tabulky budeme označovat symboly A a B (přičemž může platit že A=B). Některé relační vztahy jsou obrazem relací mezi entitami.

Jakákoliv binární relace je plně určena, pokud známe všechny dvojice prvků, které ji tvoří (resp. jsou v daném relačním vztahu). Prvky relačních vztahů jsou řádky (relace je tedy určena množinou dvojic řádků tj. je podmnožinou kartézského součinu řádků z obou tabulek tj. $R \subseteq A \times B$)

U relací typu 1:N je každý řádek tabulky A v relaci s nejvýše jedním řádkem tabulky B (naopak řádek v B může být v relaci s libovolným počtem řádků z A). Tyto relace lze representovat pomocí **dvojice atributů**. Prvky v tabulce B jsou jednoznačně určeny primárním klíčem (resp. jakýmkoliv jiným alternativním kandidátním klíčem). Řádky v tabulce A pak mohou odkazovat své druhy v relaci pomocí hodnoty příslušného (cizího) primárního resp. kandidátního klíče. Tento atribut (sloupec, výjimečně skupina sloupců) se označuje jako tzv. **cizí klíč** (cizí protože je primárním klíčem jiné tj. cizí tabulky).

cizí klíč

[1] český termín relace částečně koliduje s označením relace pro tabulky, I když obě konstrukce vycházejí ze stejného (matematického) principu, je jejich odlišení ve fázi logického a fyzického návrhu databáze nutné. Proto dávám přednost pojmu tabulka (pro relace mezi atributy) a relační vztah (pro relaci mezi řádky tabulek).

Aby vše fungovalo, jak má, musí platit následující omezení:

Cizí klíč může obsahovat pouze platné hodnoty odkazovaného kandidátního klíče (tj. hodnoty, jež se vyskytují v právě jednom řádku kandidátního klíče) nebo hodnotu NULL.

Relační vztahy jsou na rozdíl od relací na úrovni E-R modelu orientované (lze říci, že tabulka A odkazuje na tabulku B). Tato asymetrie má dva důsledky:

Povinná participace na relaci se projevuje jinak u odkazují a jinak u odkazované tabulky. Neúčast v relaci na straně odkazující se projevuje hodnotou NULL u cizího klíče (tj. je snadno detekovatelná), neúčast záznamu odkazované tabulky nepřítomností hodnoty jeho primárního klíče v sloupci cizího klíče (což je mnohem skrytější).

Druhý problém je závažnější pomocí jednoduchého systému odkazů nelze representovat relace typu N:M. Řešení tohoto problému si necháme do zvláštní podkapitoly.

Kvazi-relační vztahy

V některých situacích jste nuceni odkazovat tabulky, které nevznikly v rámci vašeho návrhu (platí to především u distribuovaných informačních systémů).

Typické je např. užití cizích číselníků ex-post (po naplnění původní databáze).

I když se i v tomto případě jedná povšechně o relaci 1:N může být výše uvedený model mírně narušen:

- cizí klíč se nemusí plně shodovat s klíčem odkazovaným (např. je nutné provést nějakou řetězcovou transformaci).
- odkazované klíče nemusí být zcela jedinečné (i když jde jen o výjimky)
- některé cizí klíče mohou zůstat neplatné (číselník) není úplný

kvazirelační vztah

Tyto vztahy budeme označovat jako **kvazirelační vztahy**.

I v tomto případě je možné provádět některé operace využívající tento (skoro)relační vztah jako jsou spojení. Musíte však být připraveni na některé negativní důsledky (občasné duplicity, ztráta informací, apod.) Svět není dokonalý.

3.5 Rozklad relací N:M

Relační model přímo nepodporuje relace typu N:M. Naštěstí lze každou relaci N:M rozdělit na dvě relace typu 1:N pomocí pomocné tzv. spojovací tabulky, která obsahuje dva cizí klíče odkazující primární klíče původních tabulek.

Předpokládejme, že potřebujeme relací typu N:M spojit tabulky kniha a autor. Jedná se o vztah N:M neboť autor může napsat více knih a kniha může být napsána více autory.

Předpokládejme následující (maximálně zjednodušenou tabulku) autorů:

id	jméno
1	J.R.Tolkien
2	Christopher Tolkien

A následující (minimalistickou) tabulku knih:

id	název
1	Pán prstenů
2	Hobbit
3	Húrinovy děti

Mezi těmito tabulkami existuje reálný vztah N:M neboť J.R.Tolkien je autorem všech tří knih, z nichž třetí (Húrinovy děti) spoluvytvářel jeho syn Christopher Tolkien.

Pro representaci této relace potřebujeme vytvořit propojovací tabulku spojující identifikátor autora s identifikátorem (jím napsané) knihy.

id_autora	id_knihy
1	1
1	2
1	3
2	3

Tato pomocná tabulka obsahuje jen sloupce cizích klíčů a realizuje tak dvě relace typu 1:N (obě jako odkazující tabulka). Lze ji použít pro vyhledávání v obou směrech. V SQL lze pomocí dvou spojení vypsat všechny knihy daného autora, ale i všechny autory dané knihy.

Propojovací tabulka má sice primární klíč (je jím dvojice obou cizích klíčů), tento primární klíč je však závislý na primárních klíčích obou odkazovaných tabulek. Proto se označuje jako tzv. slabá. Slabé tabulky nejsou běžně zahrnovány do konceptuálního modelu (i když některé modely tzv. slabé entity tolerují).

V některých případech, jsou však s relacemi spojeny atributy, které nepatří do žádné z propojených tabulek (entit). V případě relací M:N jsou dosti časté. Propojovací tabulka tak může obsahovat i další sloupce, z nichž některé mohou fungovat jako primární klíče. Tyto tzv. silné tabulky se běžně zahrnují již ve fázi konceptuálního návrhu, tím spíše, že často lze příslušnou entitu snadno pojmenovat (tj. příslušné objekty nají reálný základ).

Vezměme například relaci mezi absolventy VŠ a vysokými školami. Tento relační vztah je typu N:M neboť školu absolvuje více než jeden student a jeden student může absolvovat i více škol. Minimální propojovací tabulka odkazuje identifikátor studenta (např. rodné číslo) a identifikátor VŠ. Tato propojovací tabulka však má reálný podklad, je jím diplom vydaný absolventovi. Tento diplom obsahuje i další údaje (např. dosažený titul) a především jednoznačný identifikátor (tabulka je proto silná a bude pravděpodobně navržena již ve fázi konceptuálního návrhu)

OTÁZKY

1. Co je ve světě relačních databází označováno jako doména?
2. Popište vztah mezi primárním klíčem a kandidátními klíči?
3. Co je cizí klíč?
4. Odkud pocházejí tabulky v logickém návrhu relační databáze?
5. Co je slabá propojovací tabulka? Jakou má strukturu a funkce?

OTÁZKY K ZAMYŠLENÍ

1. Co je z pohledu matematiky relace? Jak to souvisí s označením tabulek jako relací resp. s relačními vztahy mezi tabulkami?
2. Jaká je maximální počet sloupců kandidátního klíče? Uveďte příklad.
3. Navrhněte alespoň tři příklady rozkladu relace M:N.

4 Normalizace v relačním modelu dat

CÍLE KAPITOLY

Normalizace neoznačuje v případě databází období represe a *doublethinku* (viz `https://cs.wikipedia.org/wiki/Normalizace`), ale užitečný formalismus, který umožňuje kontrolovat a snižovat redundanci dat a tím zvyšovat robustnost databází. Hlavním cílem by však neměla být memorizace „nesrozumitelných“ definic, ale praktické dovednosti v identifikaci nedostatků ve fyzickém návrhu databáze a jejich náprava.

Pravidla normalizace jsou ve většině případů velmi intuitivní a mnohé z nich se (téměř nevědomky) aplikují již v raných fázích návrhu. Formalizace však zjednodušuje jejich aplikaci i v případě složitějších tabulek a umožňuje objektivní zhodnocení dosaženého stavu. Splnění tzv. normálních forem však samo o sobě nezajišťuje kvalitu a především použitelnost návrhu.

4.1 Normalizace

normalizace

Jako **normalizace** se označuje proces (postupné) změny organizace databáze, tak aby se odstranila či alespoň minimalizovala redundance dat. Při normalizaci se mění tabulky, atributy tj. sloupce a relační vztahy mezi tabulkami (typicky vznikají nové tabulky a vztahy mezi nimi).

Proč je redundance dat tak problematická?

Při redundanci dat se některé údaje vyskytují na více místech tabulky. Problém však netkví v nadbytečné spotřebě pamětí. Mnohem závažnější je nebezpečí vzniku nekonzistencí při modifikaci dat. Aby byla modifikace redundantních dat provedena důsledně, musí být změněny všechny výskyty daného údaje. Nedůsledná modifikace vede k nekonzistenci.

Příklad:

Údaj o aktuálním ročníku jednoho a téhož konkrétního studenta můžeme mít být uložen ve více záznamech jedné tabulky. Viz např. tabulku stipendií (hezký motivační příklad, jména i částky jsou náhodně zvolené):

student	ročník	stipendium
Petr Novák	2	150 000
Jan Dvořák	1	20 000
Petr Novák	2	5 000

Zde je redundantní ročník u studenta P. Nováka. Pokud chceme udržovat tabulku aktuální, pak musíme zvyšovat ročník u studentů, kteří postoupili do vyššího ročníku. Želbohu, dojde k chybě, a zvýšíme jen jeden výskyt (třeba jen ten první). Je zřejmé, že výsledkem je nekonzistence; student se nám vlastně rozštěpil ve dva (každý je v jiném ročníku). Pokud by byla redundance odstraněna (viz 3.NF), nemohlo by k takové chybě vůbec dojít.

Normalizace se běžně provádí ještě ve fázi návrhu tj. v okamžiku, kdy tabulky fyzicky neexistují a neobsahují žádná data. Všechny klíčové charakteristiky: obsahy sloupců, kandidátní klíče a závislosti mezi sloupci tak vycházejí z modelu nikoliv z reálných dat. Pokud jsou tyto

modely nesprávné pak mohou vést i k chybné normalizaci! V tomto případě resp. v situaci, kdy jsme nuceni využívat data z nenormalizované databáze, je možno provést normalizaci ex post, jež musí být následována migrací dat (do nových tabulek). Při této migraci nedochází ke ztrátě dat, neboť všechny normalizace jsou svou podstatou bezeztrátové.

Při normalizace se aplikují pravidla, která po krocích mění organizaci dat v databázi, tak aby se postupně snižovala (potenciální!) redundance dat. Výsledkem jednotlivých kroků je posloupnost (či přesněji hierarchie) tzv. **normálních forem** tabulek. Čím vyšší je normální forma (všech) tabulek v databázi, tím nižší je redundance dat riziko vzniku nekonzistencí.

normální forma

Normální formy lze rozdělit do dvou skupin. V první skupině jsou normální formy, jejichž porušení (a tudíž i nutnost další normalizace) lze relativně snadno detekovat. Navíc je jejich formalismus snadno pochopitelný.

Do této skupiny patří první tři normální formy. Správně navržená databáze by měla splňovat 3. normální formu u všech svých tabulek (a tudíž i normální formu první a druhou).

Dosažení 3. normální formy však ještě nezaručuje eliminaci redundance. Existují proto i vyšší normální formy (4. NF a 5.NF). V jejich definici však neexistuje shoda a jen obtížně se detekuje jejich dosažení (formalismus už není zdaleka tak triviální), Navíc se v praxi jen zřídka vyskytují tabulky, které splňují 3.NF a nesplňují 4.NF nebo 5. NF, neboť většina protipříkladů bývá eliminována již ve fázi logického návrhu. Z tohoto důvodu se tyto normální formy v praxi téměř nepoužívají.

Zajímavým případem je tzv. 0. normální forma. Pokud už je použita, pak často splývá s elementárními požadavky na relaci (jednoznačnost řádků, shodný typ hodnot atributů). V tomto případě je však zbytečné zavádět nový pojem. Někteří autoři definují 0NF jako doplněk k 1.NF (tj. tabulka má 0.NF pokud není v 1.NF). To je však matoucí, neboť pro ostatní normální formy platí, že je-li tabulka v *n*-té normální formě, pak je i *n*-1. normální formě. Z tohoto důvodu nebudeme 0.NF zavádět.

4.2 První normální forma (1.NF)

první normální forma

Tabulka je v **první normální formě**, pokud platí:

Každý doména (množina přípustných hodnot) libovolného atributu obsahuje pouze atomické (nedělitelné) hodnoty.

Jinak řečeno: položky/buňky tabulky (hodnoty atributů v daném řádku) jsou vždy atomickými hodnotami.

Tato definice se jeví jako zcela triviální a bezesporná.

Bohužel tomu tak není. Problém je v tom, co lze označit jako atomické a co nikoliv. Ve skutečnosti je pojem atomičnosti mnohdy relativní a navíc trochu mlhavý (fuzzy).

Atomickými hodnotami par excellence jsou logické hodnoty a celá čísla. Jejich vnitřní struktura je dána jen implementací a z hlediska celkové sémantiky nemá žádný význam (je jedno zda jsou to např. binární či dekadické číslice, apod.)

Problém však už nastává s **řetězci**. Ty už mají sémanticky významnou vnitřní strukturu, neboť jsou tvořeny znaky či lépe (syntaktickými) tokeny. Pokud je řetězec tvořen jedním nedělitelným tokenem, pak je vše v pořádku. Rozklad na tokeny však není dán absolutně, ale určen použitím řetězce (což se navenek projevuje tím jak k řetězci při zpracování přistupujeme).

Příklady:

identifikátory se sémanticky významnými (pozičně určenými) znaky: např. X2L (kde X, 2 a L jsou příznaky různých kategorií) - – každý znak je token a každý by měl být ve zvláštním sloupci. Pokud však identifikátor ve většině případů využíváme v celku (jako klíč či pro výpis) je stálé

spojování pomocí zřetězení nepohodlné (a mnohdy i méně efektivní). V případě potřeby lze provést jednoduchý rozklad (pomocí indexace).

rodné číslo (bez lomítka) je tvořeno pěti tokeny: rok, měsíc+pohlaví, den, denní číslo, kontrolní číslice. Pokud používáme rodné číslo jako celek (což doporučuji) je to jeden token (a bude jen v jednom sloupci), pokud je využíváme jako příznak pohlaví a informaci o dni narození (což může být problematické) je to pět sloupců.

(české) jméno osoby: většinou chápáno jako dva nebo tři tokeny: křestní jméno (jména) + mezijmení (střední jméno) + příjmení. Rozložení na příjmení a křestní jméno se běžně provádí (proč? např. formuláře vyžadují jméno a příjmení zvlášť). Křestní jména se chápou jako celek (i když je jich více). To vychází z běžné interpretace a zjednodušuje to návrh (křestních jmen může být neomezený počet, což by vedlo ke vzniku relace 1:N). Problém však může vzniknout v okamžiku, kdy chceme poslat přání k jmeninám (pošleme 24.4 přání jen Jiřím, nebo i Jiřím-Václavům, apod.).

seznam titulů (oddělených čárkou): tokenů může být různý počet, který je v zásadě neomezený. Tyto hodnoty by se měly chápat jako neatomické (v zásadě je to neohraničené pole atomických hodnot, viz dále). Vede to však k nutnosti vzniku nové tabulky (číselníku titulů) a relace. V praxi nás běžně nezajímají jednotlivé tituly (i když dotaz najdi všechny magistry se občas hodí).

Další skupinu tvoří hodnoty, které jsou tvořeny fixním (a běžně) po malým počtem dílčích hodnot, s nimiž běžně pracujeme jako s celkem. Příkladem je např. 2D vektor (běžnými operacemi je sčítání resp. násobení) či zeměpisné souřadnice (dvě reálná čísla pro zem. šířku a délku — běžná operací je zjištění vzdálenosti mezi body). Tyto hodnoty by se měly chápat jako atomické (v případě potřeby lze jejich dílčí části extrahovat), avšak jen v případě, že databázový systém podporuje daný konkrétní typ resp. lze vytvářet složené typy. To nebývá běžné a tak nezbývá než tyto hodnoty rozložit do dvou či více sloupců.

Jiná situace nastává v případě, že by hodnota obsahovala neomezený počet hodnot stejného typu tj. pole hodnot. To podporuje jen výrazná menšina RDBMS a i u nich téměř nelze pracovat s polem jako celkem. Tento případ se chápe jako jasné porušení 1.NF (avšak vzpomeňme analogický příklad s tituly).

Speciálním případem jsou položky, jejichž hodnoty tvoří rozsáhlejší strukturovaná data, jako jsou JSON a XML dokumenty (se složitostí, která je minimálně stejná jako složitost tabulek). Pokud se přistupuje k jednotlivým prvkům, položkám či atributům strukturovaných dat, pak se jedná buď o zcela nerelační přístup (tabulka je v zásadě jen jednoduché úložiště) nebo o přístup hybridní. V tomto případě nemá smysl uvažovat o normálních formách (alespoň v nerelační části).

Normalizace do 1.NF

Postup normalizace se liší podle charakteru složených (neatomických dat).

Jednodušší situace nastává v případě, kdy platí následující dvě podmínky

1. počet částí je fixní a malý (v řádu jednotek)
2. význam částí je dán pozicí (uspořádáním) nebo ekvivalentním způsobem

V tomto případě stačí hodnotu **rozložit do dvou nebo více sloupců** (z nichž každý je atomický).

Jen o něco složitější je situace v případě, kdy je dílčích hodnot proměnné resp. větší množství, ale všechny jsou stejného typu.

V tomto případě lze **sloupec odstranit a nahradit za novou tabulku a relační vazbu 1:N resp. za vazbu M:N s dvěma novými tabulkami** (jedna bude spojovací).

Jednodušší situace nastává tehdy pokud, se každá dílčí hodnota vyskytuje jen v jednom exempláři složené hodnoty, tj. jedná se o relaci 1:N.

Nová tabulka bude v tomto případě tvořena tolika řádky, kolik existuje různých dílčích hodnot. Každý řádek pak kromě této hodnoty obsahuje i cizí klíč, odkazující kandidátní klíč původní tabulky (a realizující tak relační vztah typu 1:N).

V případě, že dílčí hodnoty vyjadřují relaci M:N (což je zcela běžné, viz náš příklad s tituly), pak je nutné přidat dvě tabulky, z nichž jedna obsahuje seznam všech (unikátních dílčích hodnot) a případný umělý primární klíč. Druhá pak tuto novou tabulku spojuje s tabulkou původní (každý řádek obsahuje dva cizí klíče, jeden odkazuje původní tabulku, druhý tabulku novou). V případě, že dílčí hodnoty mohou být použity jako primární klíč (jsou to např. celá čísla), pak je lze umístit přímo do spojovací tabulky (a tak eliminovat novou tabulku, jejíž funkci zastane tabulka spojovací).

Nejsložitější situace nastane v případě, že složené hodnoty obsahují proměnný počet hodnot různých typů (přičemž tyto hodnoty jsou do značné míry vzájemně zastupitelné). Tento případ by měl být opravdu výjimečný, neboť většina struktur tohoto typu měla být eliminována již ve fázi logického návrhu (bohužel však existují i zcela špatně navržené databáze). V tomto případě často musíte **rezignovat na representaci v relačním tvaru** a využít NO-SQL přístupu (tj. například podpory JSON hodnot).

4.3 Druhá normální forma (2.NF)

druhá normální forma

Tabulka je ve **druhá normální formě**, pokud je v 1.NF a navíc platí:

Každý neklíčový atribut je funkčně závislý pouze na celém kandidátním klíči.

Identifikace *funkční závislosti* není složitá. Atribut y je závislý na uspořádané množině atributů $(x_1, \dots, x_n)$, právě tehdy když, je jakákoliv hodnota atributů $(x_1, \dots, x_n)$ (tj. n -tice) v relaci s právě jednou hodnotou y. Volněji řečeno, hodnota závislého atributu je jednoznačně určena hodnotou atributů $(x_1, \dots, x_n)$. Tuto závislost je přirozeně nutno testovat na modelu nikoliv na ukázkových či provozních datech (závislost výběrových dat je jen nutnou nikoliv postačující podmínkou).

Příklad:

Podívejme se na klasický příklad tabulky nesplňující 2.NF (přestože splňuje 1.NF). Tabulka obsahuje informace o studentech-absolventech (výběr z řádků).

kurs	student	zakončení	datum	semestr
KI/PGL1	F1265	zápočet	2014-01-15	ZS
KI/PGL1	F5625	zápočet	2014-05-03	LS
KI/PGL2	F1265	zkouška	2015-06-03	LS
KI/PGL2	F8001	zkouška	2015-09-03	LS

Sloupec semestr obsahuje údaj o semestru v němž student daný kurs zakončil (nikoliv v jakém semestru byl kurs vyhlášen).

Nejdříve identifikujme kandidátní klíče. Je zřejmé, že kandidátním klíčem není žádný izolovaný atribut. Atribut datum obsahuje v našem výběru jen unikátní hodnoty, ale je zřejmé že je to jen náhoda (v jednom dni může kurz absolvovat více studentů).

Otestujeme tedy všechny dvojice (10 dvojic, kombinace bez opakování). Některé z nich zjevně obsahují více stejných hodnot (*kurs+zakončení*, *kurs+semestr*), u jiných je to zřejmé z modelu (*kurs+datum*, je běžné, že v rámci jednoho termínu získá zápočet či zkoušku více studentů), podobně *student+zakončení* či *student+datum* a dvojice se semestrem. Zůstává tak jediná dvojice *kurs+student*, což je podle modelu zcela jistě kandidátní klíč (každý student může

každý kurz zakončit jen jednou). Zbývá však otestovat ještě trojice (10 trojic). Podle definice kandidátního klíče však můžeme ignorovat trojice obsahující dvojici *kurs+student*. Zbývají tak trojice *kurs+zakončení+datum*, *student+zakončení+datum*, *kurs+zakončení+semestr*, *kurs+datum+semestr*. Je zřejmé, že nejsou unikátní (např. nic studentovi nebrání udělat dva zápočty v jednom dni či dokonce semestru).

Kandidátní klíč je tak jen jediný (*kurs+student*). Neklíčové atributy jsou tři — zakončení, datum a semestr. Pro ověření 2.NF je nutné otestovat, zda žádný z nich nezávisí na libovolné prosté podmnožině kandidátního klíče. To jsou naštěstí jen jednoprvkové množiny {*kurs*} a {*klíč*}, takže kombinací příliš mnoho není. Datum na kursu a datum na studentovi zjevně nezávisí (kurs je možno ukončit ve více než jednom dni, a student obecně nemusí absolvovat všechny zápočty a zkoušky v jednom dni) Podobně to platí i o semestru. Zjevně také nezávisí zakončení na studentovi (každý student v průběhu studia absolvuje zkoušky i zápočty). Zajímavá je však závislost zakončení na kursu. Výběr ji ukazuje (PGL1 je zakončeno vždy seminářem) a PGL2 vždy zkouškou.

Je však tomu tak i v modelu? V případě naší fakulty tomu tak je, neboť zakončení je jednoznačně určeno studijním plánem, a v něm musí při změně zakončení dojít ke změně identity kursu (jen tak lze testovat zda student splnil všechny povinnosti platné v době jeho nástupu). Tabulka tak není ve 2.NF.

Ověření zda tabulka splňuje 2.NF silně závisí na počtu kandidátních klíčů, jejich délce a počtu neklíčových atributů.

Některé kombinace jsou triviální: pokud tabulka obsahuje jen jednoprvkové kandidátní klíče pak je vždy v 2.NF (mlčky předpokládíme, že je v 1.NF). Typickým příkladem je tabulka s jednoduchým primárním klíčem bez alternativních klíčů.

Pozor: Přidání jednoduchého (tzv. umělého) primárního klíče do tabulky nemění charakteristiku tabulky vůči 2. NF normální formě. Pokud byla ve 2. normální formě před přidáním klíče, tak zůstává ve 2.NF; pokud nebyla ve 2.NF, tak zůstává nenormalizována (na což se někdy zapomíná).

Pokud tabulka obsahuje mnoho sloupců, z nichž některé tvoří rozsáhlejší kandidátní klíče, pak může být ověření její 2. NF normálnosti zdlouhavé (rozhodně nestačí jen vhled). Naštěstí se takové tabulky v praxi používají jen zřídka.

Normalizace do 2.NF

Normalizace (z 1.NF) do 2.NF je jednoduchá. Z tabulky vyjmeme závislý (neklíčový) sloupec a vytvoříme novou tabulku, která obsahuje všechny (unikátní) n-tice tvořené hodnotami tohoto atributu a klíčového atributu resp. klíčových atributů, na němž (nichž) je neklíčový atribut závislý.

Hodnoty klíčových atributů v nové tabulce se stávají primárním klíčem, na něž odkazují příslušné klíčové atributy v původní tabulce (jako cizí klíč).

Příklad:

V našem příkladu se studenty odebereme sloupec *zakončení* (závislý na dílčím kandidátním klíči), tj. vznikne tabulka (ve 2.NF):

kurs	student	datum	semestr
KI/PGL1	F1265	2014-01-15	ZS
KI/PGL1	F5625	2014-05-03	LS
KI/PGL2	F1265	2015-06-03	LS
KI/PGL2	F8001	2015-09-03	LS

Abychom zachovali původní informace musíme vytvořit novou tabulku, která bude obsahovat dva sloupce. Prvním sloupcem je dílčí kandidátní klíč, druhým na něm závislý neklíčový sloupec. Tabulka bude obsahovat jen unikátní řádky (tj. v praxi jeden řádek pro každý kurs). Sloupec kurs v původní tabulce se stane cizím klíčem odkazujícím stejnojmenný (primární) klíč nové tabulky (relační vztah N:1).

kurs	zakončení
KI/PGL1	zápočet
KI/PGL2	zkouška

4.4 Třetí normální forma (3.NF)

třetí normální forma

Tabulka je ve **třetí normální formě**, pokud je v 2.NF a navíc platí:

Všechny neklíčové atributy jsou navzájem funkčně nezávislé.

Ekvivalentně řečeno: neexistuje neklíčový atribut, který by byl funkčně závislý na jiném neklíčovém atributu resp. množině neklíčových atributů.

Protože všechny použité pojmy jsme definovali a diskutovali v sekci věnované 2.NF můžeme ihned přistoupit k hodnocení naší ukázkové tabulky (která je již v 2.NF).

Příklad:

kurs	student	datum	semestr
KI/PGL1	F1265	2014-01-15	ZS
KI/PGL1	F5625	2014-05-03	LS
KI/PGL2	F1265	2015-06-03	LS
KI/PGL2	F8001	2015-09-03	LS

Tabulka obsahuje jen dva neklíčové atributy: datum (zakončení) a semestr (zakončení). Datum není funkčně závislé na semestru (známe-li semestr nemůžeme jednoznačně datum zakončení). Na druhou stranu, stejnému datu zakončení vždy odpovídá stejný semestr (neexistuje totiž den, který by spadal do dvou semestrů). Tabulka proto není ve 3.NF.

Uveďme ještě několik dalších příkladů narušení 3.NF:

- označení a základní frekvence procesoru v tabulce počítačových sestav (frekvence je jednoznačně určena označením procesoru)
- časopis a vydavatel v tabulce půjčených časopisů (vydavatel je určen časopisem, i když pozor to se může v čase měnit!)

Normalizace do 3.NF

Základní normalizační krok je obdobou normalizace do 2.NF. Závislý sloupec je odstraněn a nahrazen novou tabulkou. Tato tabulka má typicky dva sloupce: jeden (klíčový) obsahuje hodnoty (neklíčového) atributu, na němž by odstraněný atribut závislý, druhý příslušné hodnoty ze závislého sloupce (tato tabulka samozřejmě obsahuje jen unikátní dvojice). Původní sloupec, na němž závisel odstraněný sloupec, se poté stává cizím klíčem odkazujícím příslušnou dvojici v nové tabulce (přes jeho první sloupec, jenž je zde primárním klíčem).
Příklad:
Nejdříve odstraníme závislý sloupec *semestr* v původní tabulce:

kurs	student	datum
KI/PGL1	F1265	2014-01-15
KI/PGL1	F5625	2014-05-03
KI/PGL2	F1265	2015-06-03
KI/PGL2	F8001	2015-09-03

Namísto toho vytvoříme novou tabulku, která bude mapovat data na příslušný semestr.

datum	semestr
2014-01-15	ZS
2014-05-03	LS
2015-06-03	LS
2015-09-03	LS

Datum v původní tabulce je cizím klíčem odkazujícím datum v nové tabulce (ten je zde primárním klíčem).
Tím by mohla normalizace skončit. Dobrý návrhář databází však ještě nemůže být spokojen. Nová tabulka sice funguje avšak má velký počet řádků (pro každý den, kdy někdo něco ukončil) a její velikost stále rychle roste (každý rok řádově o stovku řádků).
Příslušnost daného data k semestru lze representovat i jinak. Mnohem kompaktněji. Semestry tvoří souvislou posloupnost dní a tak je lze representovat pomocí mezí:

datum_od	datum_do	semestr
2013-09-28	2014-02-10	ZS
2014-02-11	2014-09-25	LS
2014-09-26	2015-02-05	ZS
2015-02-06	2015-09-20	LS

Podle této tabulky lze snadno (a jednoznačně) převést datum na semestr. Jen se trochu zkomplikuje SQL příkaz (tzv. spojení). Něco však za toto zjednodušení zaplatíme. Vztah mezi tabulkami nelze vyjádřit pomocí běžného relačního vztahu (typu N:1), odkaz nelze označit jako cizí klíč (tj. nebude automaticky kontrolována referenční integrita tj. zda datum leží v právě jednom rozsahu) a automaticky nelze kontrolovat ani disjunktivnost intervalů (kontrolovat lze pouze jednoznačnost primárního klíče tj. např. *datumu_od*). Vše lze samozřejmě realizovat pomocí tzv. triggerů, ale to je výrazně složitější řešení.

4.5 Další normální formy

Jak již bylo řečeno, jsou ostatní (vyšší) normální formy užívány jen výjimečně. Pokud však máte zájem o hlubší pochopení normalizace doporučuji prostudovat na anglické Wikipedii alespoň dvě vyšší normální formy.

Boyce–Codd normal form — mírně striktnější verze 3.NF (někdy je označována jako 3.5NF)

Wikipedia contributors. *Boyce–Codd normal form* [Internet]. Wikipedia, The Free Encyclopedia; 2016 Jan 12, 14:41 UTC [cited 2016 Feb 4].
Availablefrom:https://en.wikipedia.org/w/index.php?title=Boyce%E2%80%93Codd_normal_form.

Domain-key normal form — zajímavý i když obtížněji detekovatelný typ redundancí

Wikipedia contributors. *Domain-key normal form* [Internet]. Wikipedia, The Free Encyclopedia; 2015 Oct 24, 06:32 UTC [cited 2016 Feb 4].
Availablefrom:https://en.wikipedia.org/w/index.php?title=Domain-key_normal_form
.

OTÁZKY

1. Proč je nutné snižovat či ještě lépe zcela eliminovat redundanci?
2. Do tabulky byl přidán jednoduchý umělý klíč? Jakou normální formu tato tabulka splňuje? Zdůvodněte!
3. Jaké domény (datové typy) narušují relační charakter RDBMS tabulek?

OTÁZKY K ZAMYŠLENÍ

1. Jaké potenciální části má registrační značka automobilů?
2. Proč nelze normalizaci jednoduše automatizovat? (pro neexistuje tlačítko: normalizuj!)
3. Kolik možných závislostí mezi atributy je nutno testovat pro 2.NF, pokud má tabulka jeden kandidátní klíč s pěti atributy a 10 neklíčových atributů?

ÚKOLY

1. Ukažte na příkladě situaci, kdy se přidáním řádku může změnit množina kandidátních klíčů.
2. Navrhněte další příklady tabulek nesplňujících 2.NF (ale splňující 1.NF)
3. Navrhněte další příklady tabulek nesplňujících 3.NF (ale splňující 2.NF)

ODKAZY NA LITERATURU

- Normalization in DBMS: 1NF, 2NF, 3NF and BCNF in Database [online]. Beginner' Book. Dostupné na `http://beginnersbook.com/2015/05/normalization-in-dbms/.`
- Mile Hillyer. *An Introduction to Database Normalization* [online]. Dostupné z `http://mikehillyer.com/articles/an-introduction-to-database-normalization/`

5 SQL — základy

CÍLE KAPITOLY

Tato kapitola obsahuje základní informace o jazyce SQL. Na jejím konci se budete orientovat v následujících oblastech:

- standardizace SQL (včetně problémů, které s ní souvisí)
- základních pravidlech syntaxe

a především se seznámíte s možnostmi datové representace elementárních hodnot prostřednictvím stručné charakteristiky podporovaných datových typů včetně možnosti jejich omezení. Budete tak připraveni využívat možnosti jednotlivých typů hodnot v procesu ukládání i zpracování dat.

5.1 Co je SQL?

SQL (*Structured Query Language*) je deklarativní jazyk založený na relačním databázovém modelu. Není však zcela kompatibilní s klasickým relačním modelem a obsahuje i části, které s relačním databázovým modelem nesouvisí.

Obsahuje několik dílčích částí (podjazyků). Nejčastěji se rozeznávají tyto (v závorce jsou typické příkazy)

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

V současnosti je to de facto standard pro RDBMS. V oblasti dotazů sice existují alternativy (QBE, LINQ), ale ty jsou jen okrajové. Toto zcela majoritní postavení SQL vede k tomu, že splývá pojem relační databázový systém a SQL databázový systém. Existuje dokonce pojem No-SQL, který se využívá pro databáze, které používají primárně jiný model než relační.

5.2 Standardizace SQL

SQL je průběžně standardizováno organizací ISO. Mezi nejdůležitější standardy patří:

- SQL-92 (společný základ, některé části jsou však již zastaralé)
- SQL:2003 (v zásadě podporováno, XML, analytické funkce)
- SQL:2008
- SQL:2011 (zatím jen dílčí podpora i u komerčních databází)

Pro SQL je typické, že mnohé části jazyka nejsou standardizovány. Mezi ty nejdůležitější patří:

- 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

Navíc mnohé části standardu jsou implementovány jen částečně nebo s mírně modifikovanou syntaxí nebo sémantikou. Mezi hlavní důvody udržování těchto nekompatibilit patří:

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

Rozdíly mezi jednotlivými dialekty jsou jen malé a ve většině oblastí je snadné přecházet z jednoho dialektu do druhého. Na druhou stranu je však téměř nemožné napsat netriviální SQL kód, který by byl plně funkční ve všech klíčových DBMS.

5.3 Základní pravidla syntaxe

SQL standardně nerozeznává malá a velká písmena (ani v klíčových slovech ani identifikátorech). Klíčová slova se často píší velkými písmeny, identifikátory jsou běžně tvořeny jen ASCII alfanumerickými znaky (první znak musí být písmeno).

Existuje však tzv. *identifikátor v uvozovkách* (*quoted identifier)*, jenž je vždy vždy interpretován jako identifikátor. Identifikátory tak mohou být:

- slova využívající libovolné Unicode znaky včetně mezerových znaků (např. ”alternativní název”)
- klíčová slova (např. ”select” ≠ SELECT)
- identifikátory lišící se pouze použitím majuskulí a minuskulí (např. ”osoby” ≠ ”Osoby”)

Řádkové poznámky (komentáře) začínají dvěma pomlčkami a končí na konci řádků

```
-- poznámka
```

5.4 Domény

doména
datový typ

Termínem **doména** (*dmain*) atributu se označuje množina přípustných hodnot, kterou mohou nabývat hodnoty atributu v rámci dané relace (tabulky). Doména je primárně určena **datovým typem**, který v zásadě odpovídá jednoduchým datovým typům ostatních programovacích jazyků. Tento typ však může být dále omezen tzv. doménovými omezeními (*constraint*).

5.4.1 Číselné datové typy

Základní nabídka číselných datových typů odpovídá typům známým i z jiných programovacích jazyků, a je zaměřena na efektivní paměťovou representaci a přímé zpracování na procesoru (pomocí ALU resp. FPU).

Většina RDBMS podporuje tyto **celočíselné datové typy**:

standardní jméno	velikost	rozsah
SMALLINT	16 bitů	-32768 … +32767
INTEGER	32 bitů	$-2^{31} \dots 2^{31} - 1$
BIGINT	64 bitů	$-2^{63} \dots 2^{63} - 1$

Téměř univerzální je i podpora čísel v IEEE 754 formátech **čísel s pohyblivou řadovou čárkou**:

standardní jmeno	velikost
REAL	4 byty (single)
DOUBLE [PRECISION]	8 bytů (double)

Specialitou SQL jsou číselný typ s téměř neomezeným dekadickým rozsahem a přesností. Jejich representace je neefektivní, operace s nimi pomalé, je však zaručeno, že **representovatelné jsou všechny dekadické hodnoty s danou přesností**.

```
NUMERIC(přesnost, měřítko)
```

kde *přesnost* je maximální počet platných číslic (počet číslic před i za desetinou čárkou) a *měřítko* počet desetinných číslic (= za desetinou čárkou).

Skutečná implementace tohoto typu včetně maximální dosažitelné přesnosti nejsou v SQL standardu definovány. Podobně nejsou definovány jaké operace má podporovat včetně pravidel zaokrouhlování.

Jednotlivé databázové systémy mohou poskytovat i další typy resp. aliasy typů. Některé databáze (např. Oracle) mají poněkud odlišý systém číselných typů (v Oracle je jádrem typ NUMBER odpovídající standardnímu typu NUMERIC).

Literály

SQL podporuje běžné dekadické číslené literály včetně semilogaritmických zápisů čísel s řadovou čárkou:

```
-42
2.56e-3
```

Operace a funkce

Všechny databáze podporují pro všechny číselné typy čtveřici základních operací (+,-,*,/) a unární minus. Podpora ostatních číselných operací závisí na konkrétním DBMS. Matematické funkce nad reálnými čísly (sqrt, log, goniometrické funkce) jsou běžně omezena jen na čísla s pohyblivou řadovou čárkou (nikoliv tedy například pro NUMERIC).

Podporovány jsou samozřejmě i běžné relační operátory. Pro testování zde je číslo v intervalu lze využít operator BETWEEN (obě meze jsou zahrnuty).

```
hodnota BETWEEN dolní_mez AND horní_mez
```

Logické hodnoty

Standard SQL nepodporuje speciální doménu logických hodnot. Logické hodnoty se klasicky representují pomocí typu SMALLINT (resp. dokonce INT), kde 0 representuje *false* a 1 *true*. Všechny funkce pracující s logickými hodnotami tuto representaci podporují.

Některé RDBMS přímo podporují typ *boolean* (např. PostgreSQL), ale z důvodů alespoň minimální přenositelnosti jej raději nepoužívejte.

5.4.2 Řetězcové datové typy

Poměrně netradiční je systém řetězcových datových typů. V případě **ASCII řetězců** (= řetězců obsahujících jen ASCII znaky) se používají tyto typy:

```
CHARACTER(n)
CHAR(n)
```

řetězce s fixní délkou n-znaků. Pokud je vložený řetězec kratší je na danou délku zarovnán výplní (typicky mezerami zleva).

Maximální hodnota použitelná pro n závisí na DBMS. Bezpečné jsou hodnoty $n < 256$.

Vhodné pro representaci identifikátorů s fixní délkou (telefonní čísla, registrační čísla automobilů, apod.)

```
CHARACTER VARYING(n)
VARCHAR(n)
```

řetězce s proměnnou délkou, avšak kratší než n. Je-li řetězec delší tak se před uložením zkrátí (není přesně definováno jak).

Maximální hodnota použitelná pro n závisí na DBMS. Bezpečné jsou hodnoty $n < 256$.

Tento typ je vhodný pro representaci kratších řetězců u nichž předem neznáme délku (jména a příjmení osob, jména měst, apod.)

V případě representace tzv. **národních řetězců** (tj. nerepresentovatelných v ASCII) neexistuje standardizované ani jednotné řešení.

Některé databáze využívají běžných typů CHAR a VARCHAR, které ukládají data v některém bytově orientovaném kódování (typicky UTF-8). Standardní kódování (pro všechny řetězce) se definuje na úrovni tabulky či databáze.

Druhým řešením je definice zvláštních typů NATIONAL CHARACTER(n) nebo NATIONAL CHARACTER VARYING(n) (zkráceně NCHAR resp. NVARCHAR). Ty podporují některé z univerzálních kódování (dnes typicky UTF-16 nebo UTF-32).

V obou případech je nutné ověřit zda limit n vyjadřuje maximální počet znaků nebo bytů (což se u některých kódování může lišit).

Pro delší řetězce (s $n \geq 256$) je vhodnější najít využít tzv CLOB tj. *Character Large Object*. Maximální počet bytů použitelných pro jeden řetězec je minimálně 2^{31} (maximální velikost se na rozdíl od VARCHAR nemusí zadávat). Pro tento typ se kromě názvu CLOB využívá i označení TEXT (PostgreSQL).

Literály

Řetězce jsou v SQL vždy mezi **apostrofy**:

```
'Hello,␣world'
```

Uvozovky mají jiný význam: ohraničují speciální identifikátory. Záměna vede ve většině případů k chybě, ale někdy pouze k podivnému chování (pokud existuje identifikátor příslušného jména). V MySQL však lze v běžné režimu používat jak uvozovky tak apostrofy (je však lepší si na to nezvykat).

Escape (únikové) sekvence nejsou ve standardu podporovány. Různé DBMS používají různou syntaxi. V PostgreSQL lze využívat běžné escape sekvence v řetězcích uvozených písmenem E.

```
E'\t\u2865\n'
```

Operace a funkce

Jediným standardním operátorem je operátor || zajišťující spojení (zřetězení) dvou řetězců. Nabídka funkcí je většinou bohatá a zahrnuje např, funkce TRIM, LEFT, RIGHT, MID resp. SUBSTRING., LOWER, UPPER, CHAR_LENGTH.

Některé RDBMS nabízejí (v souladu se standardem) funkce v němž jsou jednotlivé parametry odděleny vhodně zvolenými klíčovými slovy.

```
POSITION('a' IN 'Elánius') = -1
SUBSTRING('Ankh-Morpork' FROM 1 FOR 4) = 'Ankh'
TRIM(left 'x' from 'xAnkhxx') = 'Ankhxx'
```

5.4.3 Časové datové typy

Jedním z typických rysů SQL systému datových typů je dobrá podpora hodnot representujících časové okamžiky, kalendářní data a časové intervaly.

time denní čas (0:00 - 24:00)

time with_time_zone denní čas se zohlednění časového pásma

date datum v proleptickém[1] řehořském kalendáři

timestamp časový okamžik (tj. *date* + *time*)

timestamp with time zone časový okamžik zohledňující časová pásma (zkratka *timestamptz*)

interval časový interval (rozpětí)

Rozsah podpory kalendářních dat u jednotlivých DBMS silně kolísá. Podporovány jsou však minimálně roky 1901-2099. V PostgreSQL lze data representovat do roku 5 874 897 po Kr :).

Časový údaj zohledňující časové pásmo definuje jedinečný časový okamžik. V databázi je uložen buď se specifikací svého časového pásma nebo v univerzálním čase UTC. Při zobrazení se běžně převádí do časového pásma klienta.

Časový údaj bez údaje o časovém pásmu může representovat větší množství časových okamžiků (pro každé časové pásmo jiný). Nikdy u něj nedochází k přepočtům mezi pásmy. Hodí se jen některé specifické aplikace: např. representaci otvíracích hodin nadnárodních společností (prodejny se například otvírají v 8:00 bez ohledu na časové pásmo).

Literály:

Standard SQL podporuje následující formát časových literálů (řetězec obsahuje časový údaj v ISO formátu):

[1] proleptický = použitý i mimo období jeho reálné platnosti (tj. zde před rokem 1582)

```
DATE '2012-08-08'
TIME '18:30'
TIMESTAMP '2012-08-08␣16:48:00'
TIMESTAMP WITH TIME ZONE '2012-08-08␣16:48:00+02'
INTERVAL 3 DAYS 4 HOURS 5 MINUTES 6 SECONDS
```

Existují však i stručnější zápisy např. v PostgreSQL lze využít standardní zápis pro přetypování:

```
'2012-08-08␣16:48:00+02'::timestamptz
'3␣4:05:06'::interval
```

Operace a funkce

Většina RDBMS poskytuje alespoň základní funkce nad časovými údaji a intervaly. Bohužel se velmi výrazně liší (a to přestože pro mnohé existuje standard). Následující přehled je relevantní pro PostgreSQL.

Operátory:

date + interval

date + integer (posun ve dnech)

timestamp + interval

time + interval

date - date (výsledek je interval)

timestamp - timestamp (rozdíl časů jako interval)

date - date (rozdíl dnů jako interval)

time - time

interval - interval

*number * interval*

Funkce:

current_date

current_timestamp

current_time

EXTRACT(field FROM source) např. `EXTRACT(DAY FROM current_timestamp)`

5.4.4 BLOB

BLOB

Pro úschovu jakýchkoliv binárních (obecně i nestrukturovaných) dat do SQL databáze lze využívat tzv. **BLOB** (*Binary Large OBjects*[2]). Typickými daty ukládanými do blobů jsou multimedia (obrázky, zvuky, videa) a binární formáty dokumentů (včetně komprimovaných dat). Bloby jsou primárně určeny pro ukládání velkých dat (*large objects*), ale lze je použít i pro data o velikosti několika málo bytů (SQL nepodporuje žádný standardní typ pro malá binární data).

Relační databáze s bloby nemanipulují ani neznají jejich vnitřní strukturu, jedinými podporovanými operacemi je zápis a čtení (obecně nelze ani testovat shodu dvou blobů). Velikost blobů je typicky omezena na 2^{32}bytů, ale některé databáze podporují i výrazně větší bloby např. Oracle až 128 TiB (problémem je u nich spíše najít dostatečně velké úložiště).

[2]ve skutečnosti je to tzv. backronym viz https://en.wikipedia.org/wiki/The_Blob

Namísto názvu BLOB (který je někdy chápán jako příliš hovorový) se používají i alternativní jména tohoto typu BYTEA (PostgreSQL = *byte array*), VARBINARY (MS SQL Server).

Obsah blobů se jen výjimečně vkládá přímo do SQL kódu nebo je vypisován na textovou konzoli (je ukládán přímo do bytových polí na straně klienta). V případě nutnosti textové representace se využije některý z mechanismů pro textovou representaci binárních data (hexedecimální zápis, base64, apod.)

5.4.5 Přetypování

SQL podporuje statické typování, tj. již při překladu jsou známy typy atributů (sloupců) dočasných symbolů apod. Bohužel ISO standard nepopisuje typový systém SQL příliš detailně a tak se jeho pojetí a rozsah v jednotlivých implementacích liší (i když rozdíly jsou jen minimální, bohužel tím více někdy překvapí).

SQL DBMS mají relativně komplexní a striktní typový systém se složitými a flexibilními pravidly implicitního přetypování a přetěžování funkcí a operátorů. Obecně lze doporučit následující řešení: spolehnout se na typový systém a problémy řešit až v okamžiku kdy se vyskytnou (např. pohledem do manuálu).

V některých situacích je však nutno provést explicitní přetypování (pokud např. nejsme spokojení s implicitním). Typickými příklady jsou konverze číslo → řetezec, resp. *double* → *numeric.*

ANSI přetypování (funguje ve všech SQL DBMS).

```
CAST(value AS type)
```

V PostgreSQL existuje stručnější a mnohými preferovaný zápis:

```
value::type
```

5.4.6 Hodnota NULL

Typickým rysem typového systému SQL jazyka je podpora (pseudo)hodnoty NULL u všech datových typů. Z hlediska teorie typů je každý typ v SQL rozšířen o hodnotu NULL (tzv. *nullable type*). Všechny funkce a operátory nad tímto typem musí být rozšířeny (angl. *lifted*), tak aby podporovali i hodnotu NULL.

Hodnota NULL může representovat:

1. **neznámou hodnotu** (not available, N/A), např. neznámé jméno
2. **neaplikovatelnou hodnotu** (daný atribut není aplikovatelný pro daný objekt), např. plat u dobrovolníka, či zlatý padák u dělníka
3. **neinicializovanou hodnotu** (tomuto použití byste se měli v SQL vyhnout)

časté použití neaplikovatelné hodnoty může být příznakem špatného návrhu. Může tak být je tak například. realizována nepravoúhlá tabulka (= tabulka s různým počtem sloupců). Výhodnější je v tomto případě použití více tabulek (a jejich sjednocení v případě potřeby) nebo operace spojení.

V jazyce SQL fungují v zásadě tři mechanismy povýšení operací, tak aby fungovali s hodnotou NULL.

1) **NULL je interpretováno jako neplatná hodnota**, tj. jakýkoliv výpočet s NULL vede opět k NULL (tj. NULL se neomezeně šíří). To je chování valné většiny operátorů a funkcí.

```
1 + NULL = NULL
2 > NULL = NULL
NULL = NULL
```

2) **NULL je ignorováno** (tj. nezapočítává se). Toto chování je typické pro statistické (agregační) funkce jako je COUNT (počet) nebo AVG (průměr)

```
COUNT(1, NULL, 2) = 2
AVG(1, NULL, 2) = 1.5
```

3) **NULL** je interpretováno jako **nejednoznačná hodnota v tříhodnotové logice** (typicky je interpretována jako UNKNOWN). Toto chování mají jen logické operátory AND a OR.

```
NULL AND FALSE = FALSE
NULL OR FALSE = NULL
```

Všimněte si, že v této logice se nemusí NULL šířit.

AND	**TRUE**	**FALSE**	**NULL**
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	**TRUE**	**FALSE**	**NULL**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Konverze hodnoty NULL

Při převodu NULL na běžnou hodnotu se uplatní funkce COALESCE.

```
COALESCE(v1, v2, …, vn)
```

Tato funkce vrací první hodnotu ze sekvence v_1, v_2, ..., v_n, která je různá od NULL. Hodnotu NULL vrací jen v případě, že všechny hodnoty v_1, v_2, ..., v_n jsou rovny NULL.

```
COALESCE(jmeno, prezdivka, "anonym")
```

Pro opačný převod funkci NULL. Používá se především tehdy, pokud je pro sémantiku hodnoty NULL využita jiná hodnota. V dobře navržené databázi by to nemělo nastat, ale ne všechny datové zdroje jsou dobře navrženy. Navíc mnohé datové protokoly a formáty nemohou NULL dobře representovat (XML, CSV).

```
NULLIF(a, b)
```

vrací NULL pokud a == b

```
NULLIF(teplota, -99) -- -99 representuje nedostupnou teplotu
```

SQL je relativně upovídaný programovací jazyk. Výrazně se to projevuje v případě testování, zda je hodnota rovna NULL či nikoliv. Podle standardu je nutno využít těchto specializovaných predikátů:

```
IS DISTINCT FROM NULL
```

resp. v negativní podobě (tj. je rovno NULL)

```
IS NOT DISTINCT FROM NULL
```

v případě, že se předpokládá sémantika tříhodnotové logiky lze využít kratších zápisů

```
IS UNKNOWN
```

resp.

```
IS NOT UNKNOWN
```

V praxi lze často použít i zápisy typu IS NULL or IS NOT NULL.

5.4.7 Doménová omezení

doménové omezení

Specifikace domén pomocí datových typů může být dále zpřesněna pomocí tzv. **doménových omezení** (*domain constraint*). Doménová omezení vytvářejí doménu, která je podtypem původního typu tj. platí.

$$MPH(C(T)) \subset MPH(T)$$

kde MPH je množina přípustných hodnota, T je původní typ a C je omezení.

Omezený typ můžete použít všude, kde je očekáván původní typ (tj. lze na něj aplikovat i operace původního typu).

5.4.8 omezení NOT NULL

Pomocí omezení NOT NULL získáváme doménu, která neobsahuje hodnotu NULL (tj. původní nerozšířený datový typ). Používá se velmi často, neboť mnohé atributy by nemohly plnit svoji funkci, pokud by neobsahovali definované hodnoty.

Příklady: jméno zaměstnance v databázi účtárny, katalogové číslo v e-shopu, jméno knihy v katalogu knihovny

Často se však stává, že u atributu, který se jeví jako NOT NULL ve fázi návrhu, může nastat (často velmi výjimečná) situace, v níž není daný údaj k dispozici.

Příklady: jméno a čas vítěze závodu (závod se nekonal resp nikdo nedorazil do cíle), autor knihy (kniha je anonymní), naměřená teplota (porucha senzoru), jméno matky (ani matka nemusí být známa), apod.

Omezení lze změnit (přidat či naopak odstranit) i po vytvoření tabulky pomocí příkazu ALTER TABLE

5.4.9 omezení CHECK

Omezení typu *check* umožňuje specifikovat přípustné hodnoty pomocí logického výrazu (predikátu). Predikát může obsahovat libovolné operace v praxi se však nejčastěji používají relační operátory.

```
CHECK (predikát)
```

příklady (užitečné):

```
CHECK (teplota < 100)
CHECK (tlak BETWEEN 950 AND 1050)
CHECK (zamestnanec_od > DATE '2000-01-01')
CHECK (splatnost > CURRENT_DATE)
CHECK (nazev <> '') -- různý od prázdného řetězce
```

Samozřejmě lze používat i komplexnější podmínky (ve většině případů však nejsou moc praktické)-

```
CHECK (vek % 2 = 0) -- pripustny je jen sudy vek
CHECK (char_length(heslo) > 10 AND position("#" in heslo)>0)
-- delší než 10 znaků a obsahující znak #
```

OTÁZKY

1. Co způsobuje nekompatibilita v implementacích SQL jazyka?
2. Proč je nutné některé idintifikátory zapisovat v uvozovkách?
3. Jaké celočíselné typy SQL podporuje? Srovnejte s typovým systémem jazyka C#.
4. Který z mezí intervalu zahrnuje operátor BETWEEN do testovaného intervalu?
5. Co může znamenat zápis VARCHAR(100) resp. NVARCHAR(100)? Zohledněte použití znakové sady Unicode a jejích různých kódování.
6. Co dělá funkce TRIM?
7. Kdy lze využívat časových údajů bez specifikace časového pásma?
8. Jaký je rozdíl mezi typem time a timestamp?
9. Jakou hodnotu má v SQL výraz NULL = NULL?

OTÁZKY K ZAMYŠLENÍ

1. Jaký datový typ je nejvhodnější pro representaci peněžních obnosů?
2. Některé databáze umožňují připojit k BLOBům tzv. MIME návěští. Jakou to má výhodu?
3. Jak jsou interně representovány časové typy?

ÚKOLY

1. Ověřte skutečnou podporu různých datových typů v PostgreSQL (resp. v jakékoliv jiném RDBMS)?
2. Definujte doménové omezení pro řetězce, které znemožní ukládání prázdných a zdánlivě prázdných řetězců (= řetězců obsahující jen mezery či tabulátory).

ODKAZY NA LITERATURU

- *PostgreSQL Documentation. Data Types.* The PostgreSQL Global Development Group. Dostupné z `http://www.postgresql.org/docs/current/static/datatype.html`.

6 SQL – DDL, DML

CÍLE KAPITOLY

Po stručném teoretickém úvodu o SQL a jeho typovém systému, přistoupíme k reálné implementaci databáze. Posupně se naučíte jak:

vytvářet tabulky

- definovat základní integritní omezení
- vkládat do tabulek data (insert)
- měnit hodnoty dat (update)
- mazat záznamy

6.1 Definice tabulky

6.1.1 Příkaz CREATE TABLE

Novou tabulku v rámci databáze vytvoříme příkazem CREATE TABLE.

Poznámka:

Při vytvoření tabulky je nutno dbát v jaké databázi bude vytvořena. V některých RDBMS jsou databáze dále členěny na tzv. schémata.

schéma

V dotazech lze využívat tabulky jen z jedné databáze (výjimkou jsou tzv. externí tabulky), ale napříč schématy. **Schémata** fungují pouze jako jmenné prostory, tj. název tabulky se skládá ze dvou částí oddělených tečkou (*schéma.tabulka*). Výjimkou je tzv. implicitní schéma, jehož entity (včetně) tabulek mohou být uváděny bez kvalifikace. jaké schéma je implicitní závisí na dané DBMS platformě a jejím nastavení. V PostgreSQL je to implicitně schéma „*public*“.

V *PgAdmin*, což je GUI nástroj pro správu PostgreSQL databází vytvoříte novou tabulku, tím že zvolíte sekci tabulky v požadovaném schématu („*public*“ pokud nechcete využívat schémata). Ještě předtím musíte vytvořit databázi.

Základem příkazu CREATE TABLE je seznam všech atributů (sloupců) tabulky se specifikací domén (= datové typy + doménová omezení).

Ukázková databáze

V rámci této opory budeme používat ukázkovou databázi států a měst. Z důvodů stručnosti je počet atributů a tabulek výrazně redukován.

Hlavní tabulkou je **tabulka států**.

```
CREATE TABLE staty (
   kod character(3) NOT NULL,
   jmeno varchar(64) NOT NULL,
   rozloha real NOT NULL CHECK (rozloha > 0.0),
   kontinent integer,
```

```
   nezavislost smallint,
   hmesto integer
);
```

kód = třípísmenný kód státu

jméno = anglický název státu (64 znaků je kvalifikovaný odhad). Musí být zadáno (je vždy aplikovatelný a vždy zjistitelný)

rozloha = rozdíly v rozloze jsou řádu $10^7 \Rightarrow$optimální representace je **plovoucí** řadová čárka, stačí přesnost 7 platných číslic (single precision). Musí být zadána.

kontinent = číselná identifikace světadílu, na němž stát leží (viz následující tabulka). Státy, které leží na více kontinentech (je jich opravdu jen pár), jsou zařazeny do světadílu, v němž leží většina území. U ostrovních států v Tichém oceánu není zadán (z důvodů neaplikovatelnosti)

nezávislost = rok získání nezávislosti

posledni_pobyt = identifikátor hlavního města (cizí klíč)

Druhou tabulkou je tabulka kontinentů. Uvádí pouze jméno kontinentu a jeho číselný identifikátor. Tabulky tohoto typu jsou běžně označovány jako číselníky.

```
CREATE TABLE kontinenty (
   id INTEGER NOT NULL,
   jmeno VARCHAR(20) NOT NULL,
)
```

pojmenovaná omezení

I když jsou předchozí definice syntakticky správné, je obecně vhodnější používat tzv. **pojmenovaná omezení**. Ty mají vlastní identifikátor a jsou ve většině případů definovány mimo definice řádků, k nímž patří (např. na konci definice tabulky). Výhodou jsou čitelnější chybové zprávy při narušení omezení a zjednodušená možnost následné manipulace (např. smazání omezení).

Jedinou výjimkou jsou omezení NOT NULL, která musí být definována přímo s atributem, jehož se týkají.

Nová a lepší definice tabulky *staty*.

```
CREATE TABLE staty (
   kod character(3) NOT NULL,
   jmeno varchar(64) NOT NULL,
   rozloha real NOT NULL,
   kontinent integer,
   nezavislost smallint,
   hmesto integer,

   CONSTRAINT staty_rozloha_check CHECK (rozloha > 0.0),
);
```

6.1.2 Primární klíč

integritní omezení

Kromě doménových omezení lze využívat i tzv. **integritní omezení**. Ty nejsou omezeny na jednotlivé hodnoty atributů v řádcích (= položku v tabulce), ale vyjadřují relace mezi sloupci či (výjimečněji) mezi řádky tabulky.

Jediným povinným integritním omezením v relačním modelu jsou tzv. **primární klíče**. Primární klíč tvoří atribut či (uspořádaná) n-tice atributů, která jednoznačně identifikuje jednotlivé řádky tabulky (n-tice v relačním modelu, entity v entitním).

Primární klíč je vybírán z množiny tzv. kandidátních klíčů. Kandidátní klíč je taková n-tice atributů, která má různou hodnotu pro každý řádek databáze a zároveň odstraněním jakéhokoliv atributu z kandidátního klíče by tato vlastnost přestala platit.

Prvotní požadavky na primární klíč

plynou přímo z relačního modelu

1. primárním klíčem může být sloupec či uspořádaná množina sloupců
2. n-tice primárního klíče musí být jedinečné v celé potenciální tabulce (to jest, při jakémkoliv potenciálním rozšíření tabulky)
3. u vícesloupcových primárním klíčů musí platit, že žádný jeho sloupec není závislý na jiném klíčovém sloupci (= minimálnost kandidátních klíčů)
4. hodnota klíče nesmí být NULL(tj. musí být vždy známa)

Hlavním problémem je zajištění jednoznačnosti klíčů v rámci celé potenciální tabulky, v případě že tabulka se může neomezeně zvětšovat v průběhu svého života. I když je aktuálně podmínka splněna, nemusí tomu tak být vždy.

Vysokou pravděpodobnost unikátnosti mají jen tzv. (neomezené) vlastní identifikátory (tj. identifikátory, jež lze bez omezení vytvářet při vkládání záznamů do tabulky), dostatečně velká (a dostatečně) náhodná čísla, a identifikátory, které vygenerovala resp. průběžně udržuje nějaká dobře navržená a spravovaná cizí databáze včetně databází distribuovaných (zděděné identifikátory).

Příklady:

vlastní identifikátory: sekvence s dostatečnou velikostí (pozor na možnost přetečení), interní řetězcové identifikátory tvořené podle pevných pravidel (opět problém s přetečením) např. spisové značky

náhodná čísla: UUID (typu 4) = 128 bitová náhodná hodnota generovaná ze dostatečně kvalitního zdroje entropie

zděděné identifikátory: SPZ, doménová jména, EAN, MAC adresy, OID

Protipříklady:

jména produktů (pokud nejsou navrhována jako jedinečná a nejsou spravována v centrální databázi)

rodná čísla: byla přidělována v době, kdy neexistovala centrální databáze (existují duplicity)

Druhotné požadavky na primární klíč

vyplývají z požadavků optimalizace operací, které se běžně s primárními klíči provádějí (testování shodnosti, porovnání, apod.)

- minimální počet sloupců (nejlépe jeden)
- datový typ s kompaktní representací (maximálně několik málo bytů) — to v zásadě splňují jen číselné typy a malé řetězce (cca do 8 bytů)
- datový typ s jednoznačnou a bezproblémovou ekvivalencí — tuto podmínku nesplňují čísla s pohyblivou řadovou čárkou (např. platí NAN ≠ NAN)
- (relativní) neměnnost hodnot v čase — změna může vyžadovat velké množství zápisů pro udržení konzistence (protipříklad: použítí čísla OP pro identifikaci osob u databází zaměstnanců)

Primární klíč se musí explicitně vyjádřit pomocí integritního omezení PRIMARY KEY. Toto omezení se běžně uvádí na zvláštní řádku a je pojmenované (jednoduché klíče lze zapsat na příslušný řádek a mohou být anonymní).

```
CONSTRAINT jméno PRIMARY_KEY(seznam_jmen_atributů)
```

Příklady:

```
CONSTRAINT zamestnanec_pk PRIMARY_KEY(idp) -- idp je jméno atributu
CONSTRAINT zamestnanec_pk PRIMARY_KEY(id_zavod, idp) -- dva atributy
```

Občas nelze v databázi nalézt žádný sloupec (nebo množinu) sloupců, která by splňovala všechny primární a (pokud možno) všechny sekundární požadavky.

umělý primární klíč

V tomto případě je možno využít tzv. **umělého primárního klíče**, což je vlastní identifikátor užívaný jen (či alespoň primárně) pro odlišení záznamů v databázi. Některé metodiky či platformy vyžadují použití umělého klíče u všech tabulek (přestože existuje dobrý přirozený primární klíč). Umělý klíč tvoří dodatečný atribut (sloupce tabulky).

Zajištění jedinečnosti a druhotné požadavky vedou k použití jediného typu umělého primárního klíče — (celočíselného) **sériového čísla**:

- sériové číslo je generováno přímo DBMS (tj. na straně serveru) pro každou tabulku zvlášť
- je to rostoucí posloupnost čísel (typicky s krokem 1), není to však garantováno
- čísla mají typ INTEGER (pokud stačí 32-bitové) nebo BIGINT (64-bitové stačí vždy)

Úkol: Naznačte u jakých databází by nemuselo stačit 32-bitové číslo (s přibližnou kvantifikací)

Syntaxe použití sériových čísel se u jednotlivých databází liší (standard neexistuje). V PostgreSQL lze použít pseudotyp SERIAL nebo BIGSERIAL. Popisuje stejnou doménu jako příslušný celočíselný typ, ale nastavuje defaultně položku pomocí generátoru sekvencí.

Ukázková databáze

V tabulce států lze jako vhodný primární klíč zvolit třípísmenný kód státu.

```
CREATE TABLE staty (
   kod character(3) NOT NULL,
   jmeno varchar(64) NOT NULL,
   rozloha real NOT NULL,
   kontinent integer,
   nezavislost smallint,
   hmesto integer,

   CONSTRAINT staty_pkey PRIMARY KEY (kod),
   CONSTRAINT staty_rozloha_check CHECK (rozloha > 0.0)
);
```

Od této chvíle DBMS kontroluje, zda jsou hodnoty atributu *kod* skutečně jedinečné. Nadbytečné je použití omezení NOT NULL u primárního klíče (v PostgtreSQL je toto omezení zahrnuto v omezení PRIMARY KEY, ale některé databázové systémy nejsou dostatečně striktní).

U kontinentů je primárním klíčem číslený identifikátor. U takto malé tabulky, u které navíc nehrozí potenciální rozšíření, doplníme potřebná jedinečná čísla sami (nemusíme používat pseudotyp SERIAL). Definice omezení však dodat musíme.

```
CREATE TABLE kontinenty (
   id INTEGER NOT NULL,
```

```
    jmeno VARCHAR(20) NOT NULL,

    CONSTRAINT kontinenty_pk PRIMARY KEY(id)
)
```

6.1.3 Další integritní omezení

Pomocí integritního omezení UNIQUE lze kontrolovat jednoznačnost u sloupců či skupin sloupců, u nichž lze jednoznačnost předpokládat (resp. kterou chceme vynutit). Z hlediska teorie jsou to alternativní kandidátní klíče.

Omezení typu CHECK mohou definovat i závislosti mezi několika sloupci. Typicky to bývá požadavek, aby hodnota v jednom sloupci byla v jednoduché relaci s jiným sloupcem.

Příklady:

```
CHECK (zamestnan_od <= zamestnan_do)
CHECK (odjezd > prijezd)
```

Mezisloupcová omezení se nepoužívají příliš často. Měla by být jednoduchá a nepříliš svazující (četná omezení výrazně omezující hodnoty jednotlivých sloupců mohou být příznakem špatného návrhu).

Ukázková databáze

Komplexnější integritní omezení lze přidat k tabulce států. Jména státu by měla být unikátní a čas posledního pobytu by měl být pozdější než čas pobytu prvního.

```
CREATE TABLE staty (
    kod character(3) NOT NULL,
    jmeno varchar(64) NOT NULL,
    rozloha real NOT NULL,
    kontinent integer,
    nezavislost smallint,
    hmesto integer,

    CONSTRAINT staty_pkey PRIMARY KEY (kod),
    CONSTRAINT staty_rozloha_check CHECK (rozloha > 0.0),
    CONSTRAINT staty_jmeno_uniq UNIQUE(jmeno)
);
```

6.2 Běžné vkládání dat

Základním příkazem pro vkládání dat je příkaz INSERT.

```
INSERT INTO table VALUES (sekvence-hodnot)
```

Hodnoty jsou nejčastěji literály (výjimečně složitější výrazy), musí být uvedeny v pořadí definice v CREATE TABLE (a to všechny včetně sloupců s implicitními hodnotami).

Bezpečnější verzí je verze s explicitním uvedením jmen sloupců.

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

O pořadí hodnot zde rozhoduje explicitní pořadí sloupců. Toto pořadí nemusí být stejné jako u CREATE TABLE. Navíc mohou být některé sloupce zcela vynechány (ty jsou jsou doplněny implicitní hodnotou nebo NULL).

```
INSERT INTO kontinenty( id, jmeno ) VALUES (1, 'Europe');
INSERT INTO kontinenty( id, jmeno ) VALUES (2, 'Asia');
INSERT INTO kontinenty( id, jmeno ) VALUES (3, 'Africa');
INSERT INTO kontinenty( id, jmeno ) VALUES (4, 'North␣America');
INSERT INTO kontinenty( id, jmeno ) VALUES (5, 'South␣America');
INSERT INTO kontinenty( id, jmeno ) VALUES (6, 'Australia');
INSERT INTO kontinenty( id, jmeno ) VALUES (6, 'Antarctica');
```

Vynechání parametrů je zcela nezbytné u atributů typu SERIAL, který by měly být vyplněny implicitně (jinak se nepoužije generátor sekvencí)!

Vkládání lze výrazně urychlit použitím rozšířené podoby příkazu INSERT, který podporuje více řádků (není ve standardu, ale je všeobecně podporováno).

```
INSERT INTO staty VALUES ('AFG', 'Afghanistan', 652090, 2, 1919, 1),
                         ('NLD', 'Netherlands', 41526, 1, 1581, 5),
                         ('ALB', 'Albania', 28748, 1, 1912, 34),
                         ('DZA', 'Algeria', 2381741, 3, 1962, 35);
```

6.3 Hromadné vkládání

Při vkládání velkého množství dat (tisíce i více řádků) může být pomalé i vkládání pomocí rozšířeného příkazu INSERT. Navíc jsou data jen málokdy dostupná ve formátu příkazu INSERT.

Většina databází naštěstí podporuje import dat z jednoduchých textových formátů jako je CSV. Syntaxe příkazu na jednotlivých platformách se však může bohužel radikálně lišit. Jako ukázku si uveďme formát importu v PostgreSQL (jak je možno vidět, konfigurovat lze základní oddělovače i representaci NULL hodnot).

```
COPY table FROM filename FORMAT csv DELIMITER ',' NULL'' QUOTE '“';
```

Příklad:

```
COPY mesta (id, jmeno, kod_statu, obyvatel) FROM stdin;
1	Kabul	AFG	1780000
2	Qandahar	AFG	237500
3	Herat	AFG	186800
```

Většina databází je schopna do CSV formátu i exportovat, což umožňuje jednoduchou migraci dat.

Import dat lze využít i pro jednoduché zálohování tabulek. V tomto případě je vhodný export do binárního tvaru (ten se samozřejmě u jednotlivých RDBMS platforem liší). Tento přístup k zálohování stačí jen pro malé nesdílené databáze. U komplexnějších databází existují řešení založená na mechanismu synchronizace mezi uzly distribuované databáze (zálohou je v tomto případě v čase zamrzlý dílčí server).

6.4 Aktualizace (UPDATE)

Pro změnu údajů v již existujících řádcích se používá příkaz UPDATE.

```
UPDATE tabulka SET atribut=hodnota[,atribut=hodnota]* WHERE predikát
```

Příkaz má dvě hlavní sekce:

Sekce **WHERE** obsahuje predikát určující jakých řádků se bude změna týkat. Pokud je vynechán pak se změna týká všech řádků (což se příliš často nehodí).

Sekce **SET** obsahuje přiřazení, které definují nové hodnoty atributů (alespoň jedno přiřazení musí být uvedeno).

Běžně se aktualizace aplikují na jedinou položku, která je identifikována svým primárním klíčem (resp. obecně jakýmkoliv kandidátním klíčem) v sekci WHERE.

```
UPDATE zamestnanci SET zamestnan=0, zamestnan_do=current_date WHERE id=4237;
-- propuštění na hodinu zaměstnance s id rovným 4237
UPDATE zbozi SET kusy=kusy-1 WHERE prod_id=75582;
UPDATE staty SET rozloha=rozloha+26860 WHERE jmeno='Russia';
-- anexe za pomoci alternativního kandidátního klíče
```

Hromadné aktualizace

Hromadné aktualizace mění hodnoty u celé skupiny položek (resp. mohou potenciálně změnit více položek).

```
UPDATE zamestnanci SET plat = 0.8*plat WHERE zamestnan_od >= DATE '2015-01-01'
UPDATE studenti SET splneno = 1 WHERE kreditni_body >= 180;
-- v určité interpretaci může signalizovat narušení 3NF (splneno zavisle na KB)
UPDATE staty SET kontinent = 2 WHERE kontinent = 1;
-- všechny evropské státy jsou přesunuty do Asie (nelze jednoduše vrátit zpět)
```

Hromadné aktualizace tvoří tzv. transakci, tj. buď se provedou všechny nebo žádná (dojde-li k chybě např. při pokusu o narušení integritního omezení).

6.5 UPSERT (spojení vkládání s aktualizací)

upsert

Termínem **upsert** se označuje spojení operace vkládání a aktualizace.

Motivační příklad:

Předpokládejme, že máme tabulku kursů se dvěma sloupci: jedinečný kód měny a kurs (jednotka měny v Kč). Pro vkládání lze použít jednoduchý příkaz INSERT.

```
INSERT INTO kursy ( kod, kurs ) VALUES ('USD', 24.2);
```

Pro změnu kursu (prováděnou typicky každý den) použijeme (opět triviální) příkaz:

```
UPDATE kursy SET kurs = 24.1 WHERE kod = 'USD';
```

Předpokládejme, že kursy získáváme z Internetu (např. prostřednictvím webové služby). Při hromadné aktualizaci všech měn musíme rozlišovat mezi aktualizací kursů, pro než již máme záznam (volíme příkaz UPDATE, INSERT se neprovede, neboť by vedl k duplicitě kódu měny) a vložením nové měny (příkaz INSERT, UPDATE nelze použít, neboť podmínka sekce WHERE nemůže být splněna). Přidání nové měny není příliš časté, ale nikoliv nemožné.

I když je odlišení mezi oběma operacemi navenek zřejmé, na úrovni SQL přináší problémy, neboť předběžné testování existence záznamu je pomalé (musí být provedeny dva SQL příkazy a mezi serverem a klientem musí být přeneseno několik bloků dat).

Většina RDBMS (PostgreSQL však až od verze 9.5) poskytuje konstrukci, která provádí výše zmíněný UPSERT. Nejdříve se pokusí záznam vložit. Pokud se to nepodaří (existuje záznam se stejným primárním klíčem) provede náhradní operaci, jíž je běžně změna tohoto záznamu.

```
INSERT INTO kursy (kod, kurs)
 VALUES ('USD', 24.1)
 ON CONFLICT (kod) -- při konfliktu (již existujicím kódu měny)
 DO UPDATE SET kurs = 24.1
```

6.6 Výmaz dat

Pro výmaz dat slouží příkaz DELETE.

Ve většině případů se maže jen podmnožina záznamů s tabulky, tj. využívá se sekce WHERE.

```
DELETE FROM zamestnanci WHERE external = 1;
```

Pro výmaz konkrétního prvku se využije primární klíč:

```
DELETE FROM knihy WHERE signatura = 'A500';
```

Některé databázové modely či aplikace výmaz řádku nepřipouštějí. Namísto toho se záznam označí jako neaktuální či neplatný (pro tento účel musí obsahovat speciální atribut/sloupec). Díky tomu zůstanou v platnosti všechny odkazy na daný záznam včetně odkazů mimo databázi.

Jako příklad si uveďme výmaz zaměstnance. Co se stane s dalšími dokumenty či záznamy, které na něj odkazovali?

Pokud jsou uloženy v dané databázi a jsou správně označeny (jako cizí klíče viz níže), pak buď budou automaticky smazány (tj. mohou být smazány např. všechny faktury, které daný zaměstnanec vytvářel) nebo nastaveny na NULL (mnohé aktivity tak budou bez jakéhokoliv odkazu na zodpovědnou osobu). Ostatní odkazy (včetně papírových dokumentů) budou odkazovat neznámou osobu (bez jakékoliv informace o ní). Zaměstnavatel může jen doufat, že má k dispozici i jiné informace o daném exzaměstnanci.

OTÁZKY

1. Jaký je rozdíl mezi databází a schématem v SQL databázích?
2. Proč jsou výhodnější pojmenovaná omezení oproti anonymním?
3. Co je CSV a jak souvisí s databázemi?

OTÁZKY K ZAMYŠLENÍ

1. V jakých situacích lze využít čísla OP pro identifikaci osob?
2. Jaké jsou výhody globálních unikátních identifikátorů oproti sériovým klíčům? Zhodnoťte v různých případech užití.
3. Alternativně lze rozlohu všech států v tabulce státy z km^2 změnit m^2. (zkuste vytvořit odpovídající příkaz, ale neprovádějte ho!). Argumentujte pro resp. proti této změně. (úspora paměti, využití v různých kontextech)
4. Nově nastavovaná hodnota atributu v části UPDATE příkazu INSERT/ON CONFLICT (UPSERT) je fixní. Jak lze nastavit hodnotu, která závisí na původní hodnotě atributu resp. na hodnotě v nově vkládaném záznamu? (konzultujte dokumentaci svého DBMS).
5. Representace (virtuálního) výmazu prostřednictvím atributu platnosti (s hodnotami 0/1 resp. false/true) většinou postačuje. Ještě expresivnější však je využití časového razítka. Navrhněte příslušný scénář použití.

ÚKOLY

1. Vytvořte tabulku v databázi *MyData* ve schématu *main*. Konzultujte dokumentaci využívané databáze. (MySQL nepodporuje schémata).
2. Připravte si ukázky dalších příkladů a protipříkladů primárních klíčů (z různých oblastí) a zhodnoťte je podle primárních i sekundárních požadavků.

ODKAZY NA LITERATURU

PostgreSQL – Vytvoření databáze a tabulky [online]. ITNetwork.cz. Dostupné z `http://www.itnetwork.cz/postgresql/postgresql-vytvoreni-databaze-a-tabulky`.

PostgreSQL – Vkládání a mazání dat v tabulce [online]. ITNetwork.cz. Dostupné z `http://www.itnetwork.cz/postgresql/postgresql-vkladani-a-mazani-dat-v-tabulce`

7 SQL – DQL

CÍLE KAPITOLY

Příkaz SELECT je centrální konstrukcí jazyka SQL. Poté co se s tímto příkazem seznámíte, budete moci:

- vyhledávat data v databázi
- transformovat dat v databázi do jiné podoby (vhodnější pro daný případ užití)
- vyhodnocovat výrazy s konstantními hodnotami (funkce SQL kalkulačky)

To samozřejmě není vše. SELECT se využívá i pro další činnosti jako je:

- kopírování dat (včetně archivace, vytváření *snapshotů*)
- vypisování metadata (údajů o tabulkách, omezeních a dalších objektech v databázi)

Tyto operace případy užití SELECTu nejsou o nic složitější, vyžadují však studium dokumentace jednotlivých databázových systémů.

V této kapitole se seznámíme s tzv. základním tvarem příkazu SELECT. Složitější dotazy jsou diskutovány v následující kapitole.

7.1 Základní tvar příkazu SELECT

Příkaz SELECT je nejen nejužitečnějším, ale i nejsložitějším příkazem jazyka SQL. Z tohoto důvodu vyjdeme z tzv. základního tvaru příkazu SELECT, který poskytuje základní operace relačního kalkulu.

```
SELECT projekce FROM zdroj | spojení-zdrojů WHERE selekce
```

SELECT transformuje datový zdroj, do dočasné tabulky, která se stává výstupem příkazu (tato výstupní tabulka je předána klientovi, který ji může zpracovat, např. ji může zobrazit). Zdrojem může být tabulka (včetně dočasné tabulky) resp. množina tabulek (z nichž některé mohou být opět dočasné).

Tato transformace se sestává ze tří základních fází, popsaných třemi sekcemi (klauzulemi), prováděných v pořadí.

1. sekce FROM (získání a spojení dat)
2. sekce WHERE (selekce záznamů-řádků)
3. sekce SELECT (projekce atributů-sloupců)

V sekci FROM jsou spojována data z několika zdrojů, tak aby vytvořily jedinou souvislou relaci tj. dočasnou tabulku s n řádky a m sloupci. Zdrojem jsou především tabulky, ale mohou to být i výsledky jiných (vnořených) příkazů SELECT (včetně tzv. pohledů) resp. tzv. kurzory. V nejjednodušší podobě (která nám prozatím postačí) zdrojem jediná tabulka.

selekce řádků

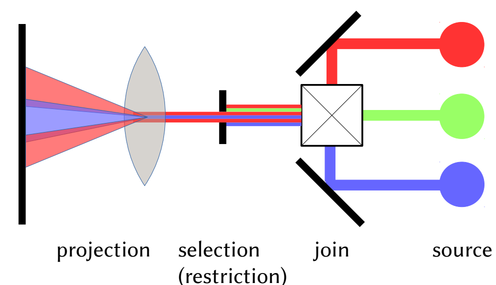
V sekci WHERE se provádí tzv. **selekce řádků**, tj. výběr řádků, které splňují nějaký predikát (= logický výraz). Výsledkem je nová zdánlivá tabulka, která obsahuje menší resp. stejný počet řádků (počet řádků se nemůže zvýšit). Počet sloupců zůstává nezměněn.

projekce sloupců

V sekci SELECT se provádí tzv. **projekce sloupců**. Některé sloupce mohou být vynechány, naopak mohou být

doplněny nové (vypočítané) sloupce, které vznikají kombinací původních sloupců. Počet řádků se v této fázi neměnní.

Jednotlivé fáze příkazu SELECT lze ilustrovat pomocí základních funkcí dataprojektoru:



Nejdříve se spojují tři oddělené zdroje světla, do jediného proudu světla (~ sekci FROM a operaci *join*). Z tohoto proudu se vezme centrální výřez (~ sekci WHERE a operaci *selekce*), který je následně promítán pomocí čočky na plátno (přičemž dochází k částečnému míchání barev, ~sekce SELECT a operace projekce).

Pozor na dvě matoucí okolnosti související se SELECT:

- operaci, jež se v relačním modelu jmenuje selekce provádí sekce WHERE (nikoliv SELECT) — to je naštěstí jen akademický problém
- příkaz se neprovádí zleva doprava, ale v pořadí FROM ⟶ WHERE ⟶ SELECT — to už není jen akademický problém, neboť ovlivňuje viditelnost symbolů

7.2 Sekce SELECT

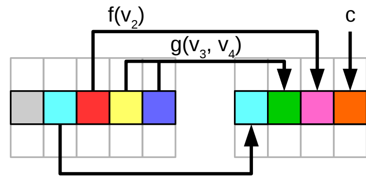
Sekce SELECT provádí transformaci záznamů tj. zobrazení každého řádku (vstupní) tabulky na řádek výstupní tabulky přičemž toto zobrazení je u všech záznammů/řádků stejné.

Výsledkem transformace je vždy tabulka se stejným počtem řádků.

Projekci lze rozložit na několik dílčí zobrazení, které zobrazují sloupec či sloupce vstupní tabulky na (jeden) sloupec tabulky výstupní.

Speciální případy:

1. zobrazení zobrazuje sloupec beze změny (identita) [na obrázku označeny azurovou barvou]
2. některé vstupní sloupce nemusí být v projekci využity tj. ve výstupu zcela chybí [na obrázku šedě]
3. vstupní sloupec může být zobrazen na výstupní sloupce pomocí unárního zobrazení (bude ve výstupu přítomen v modifikované podobě) [červená ⟶ fialová]
4. několik vstupních sloupců může být zobrazeno na jediný výstupní [modrá + žlutá ⟶ zelená]
5. zobrazení může být nulární, tj. do nového sloupce vkládá fixní hodnotu nebo výsledek bezparametrické funkce [oranžový]



Speciální tvar pro projekci neměnící sloupce (projekce je identita)

```
SELECT * FROM staty;
```

zobrazeny je celá tabulka (všechny řádky a všechny sloupce)

Výběr omezené podmnožiny sloupců:

```
SELECT jmeno, rozloha FROM staty;
```

Výstup (zkrácený):

```
                 jmeno                   |   rozloha
-----------------------------------------+-------------
 Afghanistan                             |      652090
 Netherlands                             |       41526
 Albania                                 |       28748
 Algeria                                 | 2.38174e+06
 Andorra                                 |         468
...
 Zimbabwe                                |      390757
(192 rows)
```

Pro výstup byl použit standardní konzolový klient *psql*. Je dostupný ihned instalaci PostgreSQL a lze ho využít (nejen) pro ladění jednoduchých SQL příkazů.

Výstup s vypočteným řádkem (rozloha je vyjádřena relativně vzhledem k ČR).

```
SELECT jmeno, round((rozloha / 78866)::numeric, 2) FROM staty;
```

Funkce *round* zaokrouhluje svůj první parametr (zde podíl rozlohy státu vůči rozloze ČR) na dvě desetinná místa (druhý parametr). První parametr musí být typu NUMERIC, zatímco podíl rozlohy (typu REAL) a celého čísla (literál 78866) je typu REAL. REAL se na NUMERIC nepřevede automaticky, proto musí být využito explicitní přetypování.

Přetypování lze provést i standardní funkcí CAST.

```
SELECT jmeno, round(CAST(rozloha / 78866 as NUMERIC), 2) FROM staty;
```

Ale ani to bohužel nemusí zajistit absolutní přenositelnost.

Výstup (zkrácený):

```
                 jmeno                   | round
-----------------------------------------+--------
 Afghanistan                             |   8.27
 Netherlands                             |   0.53
 Albania                                 |   0.36
 Algeria                                 |  30.20
 Andorra                                 |   0.01
...
```

Vypočtené sloupce jsou implicitně bezejmenné. To může zkomplikovat další zpracování výstupní tabulky a částečně i jejich kontrolní výstupy v nástrojích typu *psql* (psql musí využít náhradní jméno, zde je to jméno vnější funkce).

Pro pojmenování (resp. přejmenování) sloupce v sekci AS lze využít následující konstrukci :

```
výraz AS identifikátor
```

Tj. chceme-li vypočtený sloupec s relativní rozlohou pojmenovat rel_rozloha, použijeme zápis:

```
SELECT jmeno, round(CAST(rozloha / 78866 as NUMERIC), 2) AS rel_rozloha FROM staty;
```

Prozatím nezískáme nic víc než lepší jméno sloupce:

```
                jmeno                 | rel_rozloha
--------------------------------------+-------------
 Afghanistan                          |        8.27
 Netherlands                          |        0.53
```

Klíčové slovo AS lze ve většině kontextů vynechat. Zápis je však poté poněkud nepřehledný takže to nelze příliš doporučit.

Pozor: klíčové slovo AS má v SQL více významů. I v našem jednoduchém dotazu je použito ve dvou významech.

7.2.1 Příkaz SELECT bez zdroje

Speciálním (degenerovaným) případem je využití příkazu bez uvedení zdroje tj. jen pro projekci (= vyhodnocení) skalárního výrazu nebo sekvence skalárních výrazů.

skalární výraz = výraz vracející jednu elementární hodnotu

```
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" 2016 (2016 je typu *double*!)

7.3 Sekce WHERE

Pomocí sekce WHERE jsou filtrovány jen ty řádky, které splňují logickou podmínku (predikát). Počet sloupců se nemění.

Je to obdoba funkcionálu *filter*, jehož filtrovací funkce mapuje n-tici (řádek) na boolovskou hodnotu.

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

Příklad: Všechny státy nezávislé již od středověku (před rokem 1492).

```
SELECT jmeno, nezavislost FROM staty WHERE nezavislost < 1492;
```

Výsledek ukazuje, že datace nezávislosti v tomto období je někdy spíše zbožným přáním (např. Čína).

```
    jmeno      | nezavislost
---------------+-------------
 Andorra        |        1278
 United Kingdom |        1066
 Ethiopia       |       -1000
```

```
 Japan          |       -660
 China          |      -1523
 Portugal       |       1143
 France         |        843
 Sweden         |        836
 San Marino     |        885
 Denmark        |        800
 Thailand       |       1350
```

Podmínky mohou být samozřejmě i složitější (včetně logických spojek)

Příklad: miliónové metropole v USA

```sql
SELECT jmeno FROM mesta WHERE obyvatel > 1000000 AND kod_statu='USA';
```

I zde může být výsledek překvapivý (otázkou je jak se počítá počet obyvatel).

```
 jmeno
--------------
 New York
 Los Angeles
 Chicago
 Houston
 Philadelphia
 Phoenix
 San Diego
 Dallas
 San Antonio
(9 rows)
```

Pro filtrování lze využít i složitější operace resp. funkce.

Příklad: Výpis měst se jmény o délce tři a méně znaků.

```sql
SELECT jmeno, length(jmeno) as dj FROM mesta WHERE length(jmeno) <= 3;
```

```
  jmeno   | dj
----------+----
 Ede      |  3
 Itu      |  3
 Jaú      |  3
...
 Ufa      |  3
 Hue      |  3
(31 rows)
```

V příkazu je výraz pro zjištění dálky jména dvakrát (v sekci SELECT i WHERE). Je to bohužel nutné, neboť jméno (alias) sloupce zavedený v sekci SELECT nejde využít v sekci WHERE.

```
ERROR: column "dj" does not exist
LINE 1: SELECT jmeno, length(jmeno) as dj FROM mesta WHERE dj <= 3;
```

Nově zavedené jméno *dj* není v sekci WHERE (ještě) viditelné, neboť WHERE (selekce) se provádá **před** sekcí SELECT (projekce).

Řetězcové vzory

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

1) operátor LIKE (resp. NOT LIKE), ILIKE (case insensitive)

- jednoduché vzory typické pro SQL. Jen dva speciální znaky: „_“ : shoduje se s libovolným (jedním) znakem, „%“ : shoduje se s libovolnou posloupností znaků

2) operátor ~ (case sensitive), ~* (case insensitive)

- POSIXovské rozšířené regulární výrazy (EGREP)

3) operátor SIMILAR TO (NOT SIMILAR TO, SQL:1999)

- kříženec mezi rozšířenými regulárními výrazy a SQL vzory

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

Příklad: města začínající písmenem Q:

```
SELECT jmeno FROM mesta WHERE jmeno LIKE 'Q%';
```

Příklad: města začínající Q, za nímž následují tři libovolné znaky (Qina, Qods)

```
SELECT jmeno FROM mesta WHERE jmeno LIKE 'Q___';
```

Příklad: jména měst tvořená jen samohláskovými písmeny (všimněte si nutnosti ukotvení)

```
SELECT jmeno FROM mesta WHERE jmeno ~* '^[aeiouy]+$';
```

s pomocí SQL vzorů (nemusí být kotveno):

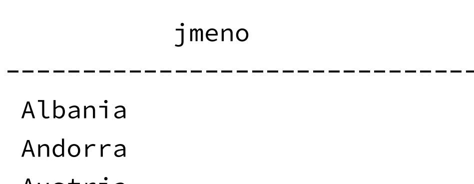
```
SELECT jmeno FROM mesta WHERE jmeno SIMILAR TO '[AEIOUYaeiouy]+';
```

7.4 Sekce ORDER BY

Pomocí sekce ORDER BY lze řádky uspořádat (setřídit) podle libovolného sloupce.
Příklad: jména evropských států uspořádaných podle abecedy.

```
SELECT jmeno FROM staty WHERE kontinent=1 ORDER BY jmeno;
```

výstup:



```
          jmeno
------------------------------
 Albania
 Andorra
 Austria
```

```
 Belarus
 Belgium
 Bosnia and Herzegovina
 Bulgaria
 Croatia
 Czech Republic
...
 Yugoslavia
(42 rows)
```

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

Příklad: největší města (sestupně podle počtu obyvatel):

```
SELECT jmeno, obyvatel FROM mesta ORDER BY obyvatel DESC;
```

A výsledek (v Korei vědí o každém jednotlivém obyvateli, což o Indii nelze říci):

```
              jmeno                | obyvatel
-----------------------------------+----------
 Mumbai (Bombay)                   | 10500000
 Seoul                             |  9981619
```

Třídit lze samozřejmě i podle vypočtené hodnoty. V naší ukázkové databázi je však obtížné najít praktický příklad.

Příklad: státy setříděné podle délky jména (od nejdelšího).

```
SELECT jmeno FROM staty ORDER by length(jmeno) DESC
```

Státy se stejnou délkou jména jsou ve výstupu uspořádány náhodně. Pokud bychom skupiny jmen se stejnou délkou chtěli uspořádat abecedně, můžeme využít vícestupňové řazení. Druhým (sekundárním) klíčem bude jméno.

```
SELECT jmeno FROM staty ORDER by length(jmeno) DESC
```

Sekce ORDER BY je prováděna až po selekci, což umožňuje využívat jmen dočasných sloupců zavedených v části SELECT.

```
SELECT jmeno, length(jmeno) as dj FROM staty ORDER by dj DESC, jmeno;
```

Pro kontrolu uveďme výstup:

```
                jmeno                | dj
--------------------------------------+----
 Congo, The Democratic Republic of the | 37
 Saint Vincent and the Grenadines      | 32
 Micronesia, Federated States of       | 31
...
 Cuba                                  |  4
 Chad                                  |  4
 Iran                                  |  4
 Iraq                                  |  4
 Laos                                  |  4
 Mali                                  |  4
```

```
 Oman                                     |  4
 Peru                                     |  4
 Togo                                     |  4
(192 rows)
```

OTÁZKY

- Co může být zdrojem dat v příkazu SELECT? Tato studijní opora diskutuje jen hlavní a obecně podporované možnosti. Na speciálnější možnosti se podívejte do dokumentace RDBMS.
- Co je selekce? Jaká část příkazu SELECT ji popisuje?
- V jakém pořadí se provádí jednotlivé sekce základního tvaru příkazu SELECT.

OTÁZKY K ZAMYŠLENÍ

- K čemu můžete použít příkaz SELECT bez části FROM?
- Podle jakých pravidel se řadí řetězce v sekci ORDER BY? Rada: podívejte se na pojem *collation* a jeho využití ve Vaší RDBMS.

ÚKOLY

- Vypište jména měst ve zkrácené podobě. Jména kratší než 10 znaků boudou vyspána celá, u delších se vypíše jen 7 znaků následovaných třemi tečkami (např. místo 'Buenos Aires' se vypíše 'Buenos ...'). Rada: využijte řetězcové funkce a konstrukci CASE.
- Vypište jména všech měst, která obsahují alespoň dvě slova.

ODKAZY NA LITERATURU

- HERNANDEZ, Michael J a John VIESCAS. *Myslíme v jazyku SQL: tvorba dotazů.* 1. vyd. Praha: Grada, 2004, 378 s. Knihovna programátora (Grada). ISBN 80-247-0899-x. Část II: Základy SQL

8 SQL – representace relačních vztahů a agregace

Tato kapitola pokračuje v popisu možností příkazu SELECT.

Po dokončení této kapitoly získáte znalosti a praktické zkušenosti pro:

- zajištění referenční integrity mezi klíči representujícími vztahy mezi tabulkami
- využití vnitřního spojení pro realizaci relačních vztahů
- využití vnějších spojení pro ošetření neúplných relačních vztahů
- spojování tabulek prostřednictvím množinových operací mezi řádky
- kombinování hodnot mezi řádky (včetně seskupování)

8.1 Cizí klíče

Relační databázové systémy podporují kontrolu integrity atributů reprezentujících relační vztahy mezi atributy.

Zatímco unikátnost v rámci odkazovaného atributu zajišťuje integritní omezení PRIMARY KEY resp. UNIQUE (alternativní kandidátní klíč), pak platnost odkazu u atributu/sloupce v roli cizího klíče (tzv. **referenční integritu**) zajišťuje omezení FOREIGN KEY s relativně komplexní sémantikou:

referenční integrita

```
CONSTRAINT jméno_omezení
FOREIGN KEY (sloupec_cizího_klíče)
REFERENCES odkazovaná_tabulka (odkazovaný_klíč)
ON UPDATE akce ON DELETE akce
```

Sekce uvozené klíčovým slovem FOREIGN KEY a REFERENCE určují representaci relačního vztahu 1:N. Odkazující tabulku (= tabulka, v níž je omezení definováno), jméno odkazujícího sloupce (sloupec_cizího_klíče), jméno odkazované tabulky a jméno odkazovaného sloupce (nejčastěji primárního klíče, může to být i vícesloupcový primární klíč). Omezení zajistí pro tyto sloupce referenční identitu (tj. platnost hodnoty ve sloupci cizího klíče) a to po provedení všech modifikujících operací nad oběma tabulkami:

- přidání klíče do odkazující tabulky (A). Pokud obsahuje neplatný odkaz (hodnotu cizího klíče) tak se neprovede a skončí s chybou
- změna hodnoty cizího klíče v tabulce A (pokud je nastaven neplatný odkaz pak se neprovede a skončí s chybou)
- změna odkazovaného klíče v tabulce B (provede se akce v sekci UPDATE)
- výmaz odkazovaného řádku v tabulce B (provede se akce v sekci DELETE)

V sekci UPDATE resp. DELETE lze uvést tyto akce:

NO ACTION nebo **RESTRICT**

akce se neprovede (s možnou produkcí chyby)

CASCADE

kaskádní výmaz

všechny závislé záznamy se vymažou (= všechny odkazující záznamy z tabulky A). To může způsobit další výmazy, pokud je použit celý řetězec relací tzv. **kaskádní výmaz**.

SET NULL

odkaz se nastaví na NULL

SET DEFAULT

odkaz se nastaví na implicitní hodnotu (je určena v CREATE TABLE a po nastavení musí opět splňovat referenční omezení)

8.2 Vnitřní spojení (INNER JOIN)

Klíčovou operací v sekci FROM příkazu SELECT je spojení více tabulek (relací) do jediné společné tabulky pomocí operace označované JOIN (vnitřní spojení).

vnitřní spojení

Existuje několik typů spojení, z nichž základní a nejdůležitější je vnitřní spojení.
Vnitřní spojení spojení spojuje dvojici tabulek (A,B) do jedné tabulky výsledné (C):

$$A \otimes B \longrightarrow C$$

Vnitřní spojení lze nejlépe popsat jako posloupnost tří kroků (v realitě jsou nejsou tyto kroky odděleny a provádí se proudově)

1. vytvoří se kartézský součin řádků obou tabulek tj. vznikne množina všech uspořádaných dvojic (a, b). kde a je libovolný řádek tabulky A a b je libovolný řádek tabulky B.
2. v každé dvojici se řádky a a b spojí (zřetězí) do jediné řádky ab (jednoduše se přidají sloupce/atributy řádku b za konec řádku a). Vznikne tak tabulka s $r_A \cdot r_B$ řádky a $c_A + c_B$ sloupci (kde r je počet řádků a c počet sloupců).
3. filtrují se ty řádky výše vzniklé, pro něž platí tzv. spojovací predikát (*join predicate*). Je to predikát mezi některými sloupci/atributy záznamů v tabulce A a některými atributy v tabulce B

Podle charakteru *join*-predikátu můžeme rozlišovat tři typy vnitřních spojení

1. obě tabulky vznikly v rámci společného návrhu a jsou spojeny relací 1:N (resp. N:1). Spojovacím predikátem je **rovnost** (shoda, ekvalita) mezi hodnotou primárního klíče v jedné tabulce a tzv. cizího klíče v druhé tabulce. Sloupce klíčů by měly být svázány integritním omezením — odkazující sloupec je označen jako cizí klíč odkazující příslušný sloupec v cílové tabulce, odkazovaný je označen jako primární klíč (či alespoň jako unikátní tj. kandidátní klíč).

   přirozené spojení

   Tento typ vnitřního spojení je jednoznačně nejčastější a označuje se jako **přirozené spojení** (NATURAL JOIN).
2. tabulky vznikly nezávisle a jsou spojeny kvazi-relačním vztahem. Spojovacím predikátem je shoda či jen přibližná shoda mezi dvěma atributy. Integritní omezení neexistuje a tudíž se mohou vyskytovat (mírná) narušení (např. více shodných primárních klíčů). To často vyžaduje úpravu výsledků (odstranění nejednoznačností).
3. spojení dvou nezávislých sloupců. Ani jeden ze slupců nemusí mít charakter identifikátoru (primárního či alternativního klíče). Predikát by však měl alespoň vyjadřovat relaci

ekvivalence. Tento typ spojení se používá jen zřídka, může však výrazně zjednodušit některé typy vyhledávání (a navíc díky výrazné optimalizaci operace JOIN v relačních databázových systémech může být značně efektivní).

8.2.1 Přirozené vnitřní spojení

Spojení využívající relaci mezi tabulkami A a B, přičemž v tabulce A je cizí klíč odkazující na primární klíč tabulky B.

Zápis:

```
A [INNER] JOIN B ON join-predikát
```

Slovo INNER je nepovinné a běžně se vynechává. Na pořadí tabulek de facto nezávisí (INNER JOIN je symetrickou operací), avšak většinou se používá pořadí (1) odkazující tabulka, (2) odkazovaná tabulka.

Příklad: Vypište jména zemí včetně jejich hlavních měst.

Tabulka *státy* neobsahuje jména hlavních měst, pouze jejich číselné identifikátory, které však úzce souvisí s identifikátory měst v tabulce *města*. Není to přirozeně náhoda: obě tabulky vznikly v rámci jednoho návrhu a oba atributy implementují relaci 1:1 mezi tabulkami. Identifikátor hlavního města v tabulce států je cizí klíč (není tak sice prozatím explicitně definován, ale jeho funkce je jasná). Odkazuje primární klíč tabulky města (je definován jako primární klíč, v zásadě postačuje jakýkoliv kandidátní klíč s unikátními hodnotami). Je zřejmé, že se jedná o přirozené spojení.

Nejdříve se podívejte na nezpracovaný výsledek spojení:

```
SELECT * FROM staty JOIN mesta ON hmesto = id;
```

Výstup má velký počet sloupců, přičemž klíč, přes nějž došlo ke spojení je ve výstupu dokonce dvakrát ve sloupcích *hmesto* a *id* (je v obou tabulkách):

```
 kod |    jmeno     |  rozloha   | kontinent | nezavislost | hmesto |  id  |   jmeno    | kod_statu | obyvatel
-----+--------------+------------+-----------+-------------+--------+------+------------+-----------+----
 AFG | Afghanistan  |     652090 |         2 |        1919 |      1 |    1 | Kabul      | AFG       |  1780000
 NLD | Netherlands  |      41526 |         1 |        1581 |      5 |    5 | Amsterdam  | NLD       |  1245000
```

Spojená tabulka proto bývá většinou ještě upravena. Rozhodně je nutné eliminovat umělé klíče (včetně těch, přes něž došlo ke spojení).

```
SELECT staty.jmeno as stat, mesta.jmeno as hlavni_mesto
FROM staty JOIN mesta ON staty.hmesto = mesta.id;
```

Všimněte si, že jméno každého atributu je kvalifikováno jménem tabulky. Pokud je jméno atributu jednoznačné (mezi všemi zdrojovými tabulkami) pak je to zbytečné, avšak užitečné (lepší orientace, jaký atribut patří kam). Vhodné je přejmenování výstupních sloupců.

Kontrolní výstup:

```
                 stat                  |           hlavni_mesto
---------------------------------------+-----------------------------------
 Afghanistan                           | Kabul
 Netherlands                           | Amsterdam
 Albania                               | Tirana
 Algeria                               | Alger
 Andorra                               | Andorra la Vella
```

```
 Angola                                  | Luanda
 Antigua and Barbuda                     | Saint John´s
 United Arab Emirates                    | Abu Dhabi
 Argentina                               | Buenos Aires
```

Někdy je pro dosažení cíle nutné provést celý řetězec spojení, tj. výstup předchozího spojení je spojen s další tabulkou.

Příklad: Vypište jména všech měst evropských států v databázi (setříděna podle obyvatel sestupně).

Databáze měst neobsahuje informace o kontinentu, na němž město leží. Obsahuje však odkaz na stát, v němž město leží (kód státu jako cizí klíč relace 1:N). Stát, pak obsahuje odkaz na kontinent, na němž se (převážně) rozkládá (relace 1:N). To umožňuje vypsat všechna města, která odkazují stát, který odkazuje daný kontinent (u nás Evropu).

Užitečnější by bylo vypsat evropská města (= města ležící v Evropě), na to však nemáme dostatek informací. Město totiž může ležet v Evropě (např. *Edirne*, nebo *Petrohrad*) i když stát leží převážně na jiném kontinentě (zde v Asii).

Nejdříve si vypišme výsledek spojení (ovšem jen s atributy, které jsou pro nás klíčové)

```
SELECT mesta.jmeno, kontinenty.jmeno, mesta.obyvatel
FROM mesta JOIN staty ON mesta.kod_statu = staty.kod
        JOIN kontinenty ON staty.kontinent = kontinenty.id;
```

Nejdříve se přirozeně spojí tabulka měst s tabulkou států přes klíče *mesta.kod_statu* (cizí klíč) a *staty.kod* (primární klíč). Výsledná tabulka se následně spojí s tabulkou *kontinenty.* Pro přirozené spojení se využije klíčů *staty.kontinent* (cizí klíč) a *kontinenty.id* (primární klíč).

Výsledek má tento tvar:

```
              jmeno                |     jmeno      | obyvatel
-----------------------------------+----------------+----------
 Kabul                             | Asia           |  1780000
 Qandahar                          | Asia           |   237500
 Herat                             | Asia           |   186800
 Mazar-e-Sharif                    | Asia           |   127800
 Amsterdam                         | Europe         |   731200
 Rotterdam                         | Europe         |   593321
...
 Mutare                            | Africa         |   131367
 Gweru                             | Africa         |   128037
(3994 rows)
```

Tou už je dobrý základ.

Nyní odfiltrujeme evropská města (učiníme tak podle jména kontinentu, nikoliv jeho číselného identifikátoru, ten je jen pomocný a může se změnit) a setřídíme je podle populace (sestupně). Jméno kontinentu navíc nemusíme vypisovat (je zřejmé, že je to Evropa).

```
SELECT mesta.jmeno, mesta.obyvatel
FROM mesta JOIN staty ON mesta.kod_statu = staty.kod
     JOIN kontinenty ON staty.kontinent = kontinenty.id
WHERE kontinenty.jmeno = 'Europe'
ORDER BY mesta.obyvatel DESC;
```

Tou už je relativně komplexní SQL dotaz, vracející zajímavé výsledky:

```
              jmeno               | obyvatel
----------------------------------+----------
 London                           |  7285000
 Berlin                           |  3386667
 Madrid                           |  2879052
...
 Ústí nad Labem                   |    95491
...
 Monaco-Ville                     |     1234
 Città del Vaticano               |      455
(649 rows)
```

SQL poskytuje i zkrácené zápisy vnitřního spojení:

```
tab1 [INNER] JOIN tab2 USING (jméno-sloupce)
```

Spojení se provede jako přirozené (shoda klíčů) přes stejně pojmenovaný sloupec. Bohužel to znamená, že cizí klíč a odkazovaný primární musí míst stejné jméno, což nemusí platit (záleží na dohodě při pojmenování klíčových atributů).
Nejkratší (i když nejméně) použitelný je zápis:

```
tab1 NATURAL JOIN tab2
```

Spojení se provede, přes všechny sloupce, které mají stejné jméno v obou tabulkách. To znamená, že se může provést přes sloupce, které mají náhodně stejné jméno (např. jméno státu a jméno města) a nikoliv přes klíče relace (ty mohou mít a často mívají různá jména). Bohužel se nikdy nezohledňují případná omezení cizích i vlastních klíčů. **Jen výjimečně tak dostaneme skutečné přirozené spojení** podle výše uvedené definice! Proto tento zápis raději aktivně nepoužívejte.

```
SELECT staty.jmeno, mesta.jmeno FROM staty NATURAL JOIN mesta;
```

Vrací (doufám, že pro Vás nikoliv překvapivě) státy, se stejnojmenným hlavním městem. (hlavní město Arménie, však není Armenie).

```
   jmeno    |   jmeno
------------+------------
 Djibouti   | Djibouti
 Mexico     | Mexico
 Armenia    | Armenia
 Kuwait     | Kuwait
 San Marino | San Marino
 Singapore  | Singapore
(6 rows)
```

8.2.2 Speciální (nepřirozená) vnitřní spojení

Dotaz uvedený v předchozím příkladě není přirozený dotaz, neboť spojuje pomocí atributů, které nejsou navrženy jako dvojice implementující relaci 1:N mezi tabulkami (jméno státu je sice alternativní kandidátní klíč, ale jméno města, na něj rozhodně neodkazuje a shoda je pouze náhodná).

Existují však ještě podivnější dotazy (z nichž jen některé jsou skutečně užitečné).

První příklad mírně zobecňuje příklad měst, jejichž jméno odpovídá názvu státu. Jméno státu může totiž tvořit jen část jména města.

```
SELECT staty.jmeno, mesta.jmeno FROM staty JOIN mesta ON mesta.jmeno LIKE ('%' || staty.j
```

Výsledek ukazuje, že dotaz není příliš užitečný.

```
   jmeno    |              jmeno
------------+-----------------------------------
 Andorra    | Andorra la Vella
 Argentina  | Malvinas Argentinas
 Belize     | Belize City
 Cuba       | Cubatão
 Chile      | Santiago de Chile
 Djibouti   | Djibouti
 Mexico     | Mexico
 Mali       | Malita
 Guatemala  | Ciudad de Guatemala
 Peru       | Perugia
...
 Singapore  | Singapore
 Sudan      | Port Sudan
 Spain      | Port-of-Spain
 India      | Indianapolis
(23 rows)
```

Nedostatky jsou dva:

- jméno určitého státu může být součástí jména města ležícím v jiném státě (např. Port-of-Spain je hlavní město Trinidadu a Tobaga)
- jedná se o náhodnou grafickou shodu na začátku jména např. Purugia a Peru spolu nesouvisí, v názvech India a Indianapolis však už určitá, i když nepřímá závislost je)

Další dva příklady využívají možnosti provést spojení tabulky se sebou samou (tj. provést výběr z kartézského součinu $A \times A$). Cílem obou příkladů je vytvoření dvojic (zde dvojic států) splňujících nějaké kritérium (to je použito jako spojovací predikát).

Příklad: výpis všech dvojic států ležících na stejném kontinentu.

```
SELECT a.jmeno , b.jmeno
FROM staty as a JOIN staty as b ON a.kontinent = b.kontinent
ORDER by a.jmeno, b.jmeno;
```

Při (reflexivním spojení tabulky se sebou samou vzniká problém, jak rozlišit atributy převzaté z prvního a druhého výskytu tabulky ve spojení. V běžném spojení stačí kvalifikace jmen atributů jménem příslušné tabulky. Zde jsou však obě jména stejná.

Proto je nutné použít tzv. aliasy tj. dočasné přezdívky tabulek, zavedené pomocí konstrukce AS (tentokrát v sekci FROM). V našem případě je první výskyt tabulky státy dočasně překřtěn na identifikátor *a* a pravý na identifikátor *b*. Tato dočasná jména jsou použita jak ve spojovacím predikátu (kontinent prvního členu dvojice je roven kontinentu druhého členu dvojice), tak v sekci SELECT (odlišíme tak jména v dané dvojici států). Navíc se využije i v závěrečném řazení (řadí se primárně podle prvního jména ve dvojici a pak podle druhého).

Výsledek má relativně velký počet řádků (7551 možných dvojic):

```
                jmeno                   |                jmeno
----------------------------------------+---------------------------------------
 Afghanistan                            | Afghanistan
 Afghanistan                            | Armenia
 Afghanistan                            | Azerbaijan
...
 Afghanistan                            | Yemen
 Albania                                | Albania
 Albania                                | Andorra
...
 Zimbabwe                               | Zambia
 Zimbabwe                               | Zimbabwe
(7551 rows)
```

ekvivalence

Výsledkem aplikace tohoto spojení je relace mezi státy. Tato relace je reflexivní (stát je v relaci sám se sebou), symetrická a tranzitivní. Jinak řečeno jedná se o **ekvivalenci**, která rozděluje množinu států na několik disjunktních podmnožin vzájemně ekvivalentních států (zde jej tvoří státy na stejném kontinentu).

Vnitřní rekurzivní spojení často popisují ekvivalenci (včetně přirozených, kde je rozkladem množina záznamů se stejným primárním klíčem). Není to však povinné.

Příklad: výpis dvojic států s podobnou rozlohou (tj. musí platit, že jsou buď shodné, nebo rozloha většího státu převyšuje rozlohu menšího o méně než 5%).

Porovnání rozlohy komplikuje asymetrie výpočtu vzhledem k většímu či menšímu státu tj. predikát $a.rozloha \leq b.rozloha \cdot (1+p)$ platí jen jen tehdy, je-li $a.rozloha \geq b.rozloha$. Naštěstí lze tento predikát převést do symetričtějšího tvaru (nezáleží na tom zde je větší a nebo b):

$$\frac{|a.rozloha - b.rozloha|}{a.rozloha + b.rozloha} \leq \frac{\delta}{2+\delta}$$

tento výraz lze využít jako spojovací predikát. Navíc vyloučíme i dvojice se shodnými státy, neboť nejsou zajímavé (podmínku splní triviálně).

```
SELECT a.jmeno, a.rozloha, b.jmeno, b.rozloha
FROM staty as a JOIN staty as b
   ON abs(a.rozloha-b.rozloha) / (a.rozloha+b.rozloha) <= 0.05/(2+0.05)
     AND a.kod <> b.kod
ORDER BY a.jmeno, b.jmeno
```

výstup:

```
        jmeno       |   rozloha   |           jmeno            |   rozloha
-------------------+-------------+----------------------------+-------------
 Afghanistan       |      652090 | Central African Republic   |      622984
 Afghanistan       |      652090 | Myanmar                    |      676578
 Afghanistan       |      652090 | Somalia                    |      637657
 Albania           |       28748 | Armenia                    |       29800
...
 Cuba              |      110861 | Liberia                    |      111369
 Czech Republic    |       78866 | Panama                     |       75517
 Denmark           |       43094 | Estonia                    |       45227
...
```

```
 Poland              |       323250 | Côte duIvoire                |       322463
 Poland              |       323250 | Finland                      |       338145
 Poland              |       323250 | Malaysia                     |       329758
 Poland              |       323250 | Norway                       |       323877
 Poland              |       323250 | Oman                         |       309500
 Poland              |       323250 | Vietnam                      |       331689
...
```

Jak je vidět tak naším jediným (podle rozlohy s odchylkou 5%) sesterským státem je (o něco menší) Panama. Státy mají maximálně 6 sesterských států (nejkompaktnější je skupina, jejímž zástupce je Polsko).

Pro zajímavost těsných dvojic (poměr < 0.1%) je jen 8 s nejmenším relativním rozdílem u Rumunska a Ghany.

8.3 Křížné spojení (CROSS JOIN)

křížné spojení

Speciální typem spojení je tzv .**křížné spojení**(*cross join*). Je to ve skutečnosti alternativní zápis pro kartézský součin dvou tabulek (se zřetězením řádků).

Kartézský součin není ve skutečnosti příliš užitečný, neboť vzniká velké množství řádků, které obsahují navzájem nesouvisející atributy.

Ve starších databázích však byl jediným podporovaným typem spojení. Koncový výběr řádků (podle spojovacího predikátu) se prováděl pomocí sekce WHERE.

```
SELECT staty.jmeno as stat, mesta.jmeno as hlavni_mesto
FROM staty, mesta -- křížné spojení
WHERE staty.hmesto = mesta.id;
```

Tento způsob se považuje za zastaralý. Moderní zápis za pomocí vnějších či vnitřních spojení má mnoho výhod:

- je přehlednější (především v případě složitějších spojení)
- jasně odděluje spojení od restrikce/selekce
- pokud není optimalizována, pak může být výrazně pomalejší a náročnější na paměť (ve skutečném spojení nikdy nevzniká celý kartézský součin tabulek s $n \times m$ řádky)
- přímo podporuje vnější spojení
- je dnes podporována všemi databázemi (včetně těch odlehčených jako je SQLite)

Občas se může vyskytnout situace, kdy lze využít přímo křížné spojení (ale nic skutečně rozumného mne nenapadá). V tomto případě použijte modernější (explicitní zápis).

Umělý příklad:

```
SELECT a.jmeno, b.jmeno FROM lide as a CROSS JOIN lide as b;
```

Vrací všechny možné kombinace jmen lidí z tabulky *lidé*. (včetně dvojic se shodnými položkami)

8.4 Vnější spojení (OUTER JOIN)

Vnitřní spojení je základním a nejčastěji používaným spojením. Nepříliš dobře však pracuje s těmi řádky tabulek, které nejsou součástí relace/vztahu v případě, kdy je participace tabulky na relačním vztahu nepovinná.

Pokud cizí klíč obsahuje hodnotu NULL, pak se příslušný řádek odkazující tabulky neobjeví ve výsledné tabulce, neboť není splněna spojovací podmínka (NULL není roven žádné ne-NULL hodnotě ve sloupci s primárním klíčem). Podobně se neobjeví řádek s primárním klíčem, který není nikdy odkazován (není roven žádnému cizímu).

Základní schéma vnitřního spojení:

tabulka A

	FK
A1	1
A2	2
A3	NULL

tabulka B

PK	
1	B1
2	B2
3	B3

A INNER JOIN B

	FK	PK	
A1	1	1	B1
A2	2	2	B2

Motivační příklad

Předpokládejme například relaci 1:N mezi zaměstnanci a místnostmi, v nichž sídlí. Tabulka zaměstnanců proto obsahuje cizí klíč odkazující tabulku místností (každý zaměstnanec sídlí jen v jedné místnosti).

Někteří zaměstnanci však nikde nesídlí (jsou to například externí zaměstnanci, prodavači apod.). Ti musí mít ve sloupci cizího klíče hodnotu NULL (odkaz na místnost není u nich aplikovatelný).

Nyní dostaneme za úkol vypsat všechny zaměstnance včetně jejich působišť. Použijeme přirozené spojení, tj. SQL příkaz bude mít následující:

```
SELECT z.jmeno, z.prijmeni, m.oznaceni
FROM zamestnanci as z
    JOIN mistnosti as m ON z.id_mistnosti = m.id;
```

Vypíší se přirozeně jen zaměstnanci, kteří mají určenou sídelní místnost.

Ve výpise se však neobjeví:

- zaměstnanci, kteří nikde nesídlí. To může být problém, pokud jde primárně o seznam zaměstnanců (např. pro účely účtárny) a místnost je jen doplňující údaj.
- místnosti, v nichž nikdo nesídlí. To může být problém, pokud se jedná primárně o adresář místností (a zaměstnanci tvoří jen pomocný údaj)

vnější spojení

Řešením jsou obecněji definované operace tzv. **vnějších spojení** (*outer join*). Existují tři podtypy podle toho jaké mimorelační záznamy zohledňuje (levé, pravé a úplné).

8.4.1 LEFT OUTER JOIN

Levé vnější spojení zahrne do výsledné tabulky:

1. všechny řádky odpovídajícího vnitřního spojení (tj. vnitřního spojení se stejným spojovacím klíčem)
2. řádky vzniklé zřetězením
    a) těch řádků první (levé) tabulky, které nebyly propojeny s alespoň jedním řádkem druhé (pravé) tabulky [v jedné kopii]
    b) a hodnot NULL na místě atributů pravé tabulky (neboť tyto atributy nejsou pro nepropojené řádky definované)

V případně přirozeného vnějšího levého spojení to znamená, že do výsledku budou kromě řádků spojených v relaci zahrnuty i ty řádky levé (odkazující) tabulky , které mají na místě cizího klíče hodnotu NULL (jiná možnost nemůže nastat). Atributy odkazované tabulky mají v tomto případě hodnotu NULL.

Schéma levého vnějšího spojení:

tabulka A

	FK
A1	1
A2	2
A3	NULL

tabulka B

PK	
1	B1
2	B2
3	B3

A LEFT OUTER JOIN B

	FK	PK	
A1	1	1	XXX
A2	2	2	XXX
A3	NULL	NULL	NULL

Zápis:

```
A LEFT [OUTER] JOIN B ON spojovací-predikát
```

Slovo OUTER je nepovinné a běžně se nepoužívá.

Použití:

Používá se nejčastěji v případě přirozených nebo kvazi-přirozených spojeních, pokud má výstup obsahovat všechny řádky odkazující tabulky (tj. včetně řádků, pro něž není příslušná relace definována).

Příklad: Všechny státy světa doplněné kontinentem, na němž leží. V případě, že tento kontinent není definován, je uvedena hodnota NULL (což platí pro státy v Oceánii). Jádrem je výpis států tedy odkazující tabulky.

```
SELECT staty.jmeno, kontinenty.jmeno
FROM staty LEFT JOIN kontinenty ON staty.kontinent = kontinenty.id
ORDER by staty.jmeno;
```

```
                jmeno                  |     jmeno
---------------------------------------+---------------
 Afghanistan                           | Asia
 Albania                               | Europe
...
 Namibia                               | Africa
 Nauru                                 |
 Nepal                                 | Asia
 Netherlands                           | Europe
 New Zealand                           |
 Nicaragua                             | North America
...
```

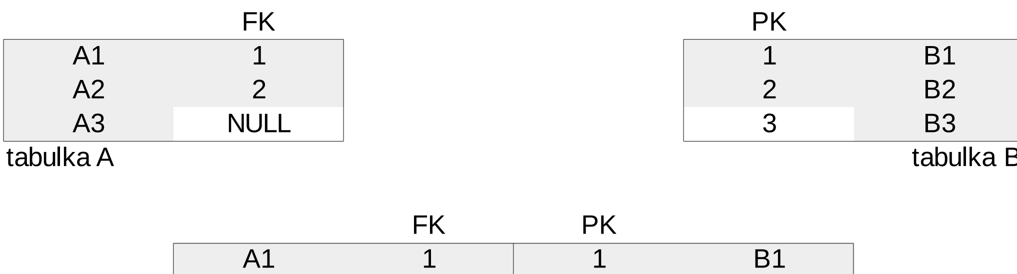
8.4.2 RIGHT OUTER JOIN

Pravé vnější spojení zahrne do výsledné tabulky:

1. všechny řádky odpovídajícího vnitřního spojení (tj. vnitřního spojení se stejným spojovacím klíčem)
2. řádky vzniklé zřetězením
   a) hodnot NULL na místě atributů levé tabulky (neboť tyto atributy nejsou pro nepropojené řádky definované)
   b) těch řádků druhé (pravé) tabulky, které nebyly propojeny s alespoň jedním řádkem první (levé) tabulky [v jedné kopii]

V případně přirozeného vnějšího pravého spojení to znamená, že do výsledku budou kromě řádků spojených v relaci zahrnuty i ty řádky pravé (odkazované) tabulky, které nejsou ani jednou odkazovány. Atributy odkazující tabulky mají v tomto případě hodnotu NULL.

Schéma pravého vnějšího spojení:



	FK	PK	
A1	1	1	B1
A2	2	2	B2
NULL	NULL	NULL	B3

A RIGHT OUTER JOIN B

Zápis:

```
A RIGHT [OUTER] JOIN B ON spojovací-predikát
```

Slovo OUTER je i zde nepovinné a běžně se nepoužívá.

Použití:

Používá se nejčastěji v případě přirozených nebo kvazi-přirozených spojeních, pokud má výstup obsahovat všechny řádky odkazované (tj. včetně řádků, které nejsou odkazovány).

Příklad: Vypište všechny kontinenty a státy na nich ležící (včetně kontinentů beze států tj. Antarktidy). Jádrem je tedy výpis kontinentů (odkazované tabulky).

```
SELECT kontinenty.jmeno, staty.jmeno
FROM staty RIGHT JOIN kontinenty ON staty.kontinent = kontinenty.id
ORDER BY kontinenty.jmeno;
```

```
     jmeno     |           jmeno
---------------+----------------------------
 Africa        | Togo
 Africa        | Tanzania
...
 Africa        | Malawi
 Antarctica    |
 Asia          | Georgia
...
```

8.4.3 FULL OUTER JOIN

Úplné vnější spojení v sobě zahrnuje jak řádky z levého vnitřního spojení (tj. řádky levé tabulky, nemající obraz v pravé), tak pravého vnitřního spojení (řádky z pravé nemající obraz resp. vzor v levé). Obsahuje tak samozřejmě i celé vnitřní spojení (jen v jednom exempláři).

Používá se relativně zřídka. Některé RDBMS jej ani nepodporují (lze jej pak získat sjednocením levého a pravého spojení a odstraněním vícenásobných výskytů řádků).

Zápis:

```
A FULL [OUTER] JOIN B ON spojovací-predikát
```

Slovo OUTER je i zde nepovinné a běžně se nepoužívá.

Schéma plného vnějšího spojení:

tabulka A

	FK
A1	1
A2	2
A3	NULL

tabulka B

PK	
1	B1
2	B2
3	B3

A FULL OUTER JOIN B

	FK	PK	
A1	1	1	B1
A2	2	2	B2
A3	NULL	NULL	NULL
NULL	NULL	NULL	B3

Naše ukázková databáze sice umožňuje využít FULL JOIN (státy a kontinenty včetně států bez kontinentů a kontinentů bez států), není to však příliš praktické (a to ani jako domácí úkol).

8.5 (SELECT) DISTINCT

Specifikace (SELECT) DISTINCT nesouvisí přímo se spojeními, ale nejčastěji se používá nad výsledky spojení (dobře navržené tabulky neobsahují duplicitní řádky).

Uvedením specifikace DISTINCT na začátku sekce SELECT se z výsledku **odstraní všechny duplicity řádků**, tj. pokud se nějaký řádek vyskytuje vícekrát, pak se zachová jen jedna jeho kopie.

Poznámka: DISTINCT se sice zapisuje v sekci SELECT, ale teoreticky tam nepatří. SELECT totiž provádí běžně provádí jen projekci, což je operace neměnící počet řádků. Nemůže však být ani v sekci WHERE (selekce), neboť se provádí až po projekci (tj. na tabulce, jejíž sloupce projekce zavedla).

DISINCT lze například použít pro dotazy, které vypisují kategorie objektů, obsahující alespoň jeden objekt splňující určitý predikát.

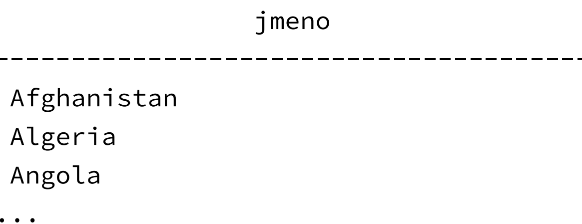
Příklad: výpis všech zemí, ve kterém je alespoň jedno velkoměsto s více než milionem obyvatel.

```
SELECT DISTINCT staty.jmeno
FROM mesta JOIN staty ON mesta.kod_statu = staty.kod
WHERE mesta.obyvatel >= 1000000
ORDER BY staty.jmeno
```

Postup zpracování:

1. vnitřní spojení tabulky měst a států podle relace „město je v daném státě“, obsahuje informace o městě spolu s informacemi o zemi, v němž leží
2. jsou filtrovány jen řádky o městech s alespoň milionem obyvatel (WHERE)
3. zobrazen je však jen sloupec se jménem státu, v němž tato města leží (projekce v SELECT)
4. některé státy jsou však v tomto seznamu několikrát (mají více než jedno milionové město). Proto odstraníme duplikáty (DISTINCT).
5. seznam je uspořádán podle abecedy (ORDER BY), tento krok může být proveden před odstraněním duplikací, neboť jej může výrazně urychlit (optimalizace)

Výsledkem je seznam 75 států:



```
                  jmeno
---------------------------------------
 Afghanistan
 Algeria
 Angola
...
```

V SQL mnohdy existuje více cest jak získat kýžený výsledek. V tomto případě lze využít funkci funkci EXISTS v rámci sekce WHERE. Funkce očekává poddotaz a vrací *true*, pokud tento dotaz vrací alespoň jeden prvek. Nicméně toto řešení (stejně jako většina poddotazů) může být extrémně pomalé (pro každý stát se hledá velkoměsto). Rychlejším řešením je využití operátoru IN na předpřipravený (seřazený) seznam zkratek států (ty lze získat jediným dotazem). I toto řešení bývá pomalejší oproti optimalizovanému spojení (většina RDBMS nejlépe optimalizuje spojení). Na druhou stranu lze jednodušeji získat doplňkový seznam (státy bez velkoměsta), neboť existuje i operátor NOT IN.

Doplňkový seznam lze získat i pomocí (vnitřního) spojení a to dokonce bez (explicitního) DISTINCT. Musíme se však seznámit ještě s jedním mechanismem spojení tabulek.

8.6 Vertikální spojení — množinové operace

Operace JOIN spojují tabulky pomocí kartézského součinu. SQL však podporuje ještě jeden mechanismus spojení tabulek, tentokrát jen na úrovni jejich řádků (proto vertikální spojení).

Popis těchto spojení není složitý, neboť se jedná o běžné množinové operace sjednocení, průnik a rozdíl množin (množina = tabulka jejímiž prvky jsou řádky).

Všechny tyto operace lze použít pouze pro tabulky, které mají stejný počet sloupců a všechny odpovídající sloupce mají stejný resp. kompatibilní datový typ (doménová omezení zde nehrají žádnou roli). Toto omezení je zřejmé, neboť jinak by množinové operace vedly k tabulkám, jejichž řádky by měly různý počet sloupců resp. by sloupce s několika doménami (sjednocení), nebo k prázdným tabulkám s nedefinovaným počtem sloupců (průnik) resp. by nic nevykonávaly (rozdíl).

Operace se aplikují mezi dva dotazy jejichž výsledek spojují (nelze je psát přímo mezi tabulky, ale to lze snadno obejít).

Zápis:

```
query1 UNION [ALL] query2 -- sjednocení výsledků dotazů
query1 INTERSECT [ALL] query2 -- průnik výsledků dotazů
query1 EXCEPT [ALL] query2 -- (množinový) rozdíl výsledků dotazů
```

Popis:

UNION vrací všechny řádky z dotazu *query1* i *query2*.

INTERSECT vrací jen řádky, které jsou ve výsledku dotazu *query1* a zároveň i ve výsledku dotazu *query2*.

EXCEPT vrací jen ty řádky z výsledku dotazu *query1*, které se nevyskytují ve výsledku dotazu *query1*.

U žádného dotazu není garantováno pořadí ve výsledné tabulce. Pokud je uvedeno klíčové slovo ALL, pak se neodstraňují duplicitní řádky (většina RDBMS podporuje tuto specifikaci jen u sjednocení, neboť jen zde mohou duplicity vzniknout jak důsledek množinové operace).

Využití

UNION

a) spojení dvou nezávislých zdrojů — např. jsou-li jednotná data ve více tabulkách např. z důvodů rozdělení podle časových úseků

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

b) logické spojení OR dvou predikátů v případě, kdy nepostačuje použití OR v sekci WHERE (výjimečné)

INTERSECTION

a) společné prvky více databází

```
SELECT product_id FROM vyrobky
INTERSECTION
SELECT product_id FROM prodej
```

b) logické spojení AND dvou predikátů v případě, kdy nepostačuje použití AND v sekci WHERE (výjimečně)

EXCEPT

tvorba doplňkových seznamů (tj. seznamů entit, nesplňujících nějaký predikát) v případě, kdy nepostačuje negace v sekci WHERE

Příklad: seznam států, v nichž se nenachází milionové velkoměsto

```
SELECT staty.jmeno FROM staty
EXCEPT
SELECT staty.jmeno
 FROM mesta JOIN staty ON mesta.kod_statu = staty.kod
 WHERE mesta.obyvatel > 1000000;
```

Otázka: Proč nestačí jen negovat podmínku ve WHERE (v základním tj. druhém dotazu)?

8.7 Agregační dotazy

Termínem agregační dotazy lze označovat ty dotazy SELECT, které produkují řádky (resp. řádek) shrnující (seskupující, agregující) informace z více řádků původního zdroje.

Současné SQL nabízí hned několik prostředků pro kombinační dotazy:

globální agregační funkce — výsledky se agregují do jediného řádku (východisko všech agregačních dotazů)

sekupování GROUP BY — skupiny řádků se agregují do jediného řádku (skupina je definována společnou hodnotou) ve sloupci

analytické funkce — řádky jsou rozšířeny o agregované hodnoty z okolních řádků

konstrukce DISTINCT ON — skupina řádku je nahrazena jedním z této skupiny zástupcem

8.7.1 Agregační funkce

agregační funkce

Agregační funkce jsou jádrem agregačních dotazů.

Agregační funkce přijímá seznam hodnot (typicky celý sloupec v tabulce či část sloupce) a vrací jedinou hodnotu (atomickou z hlediska SQL).

Agregační funkce lze rozdělit do dvou základních typů:

redukující agregace: redukují data do výrazně jednodušší podoby (například jediného čísla)

restruktulizační agregace: vrací všechna původní data, avšak v alternativní representaci (tj. nikoliv podobě sloupce či tabulky)

Nejjednodušším případem užití agregačních funkcí je **totální agregace hodnot**. V tomto případě se agregační funkce umisťují do sekce SELECT (nahrazují tak běžnou projekci).

```
SELECT COUNT(stat.kod) FROM staty;
```

Agregační funkce COUNT očekává seznam jakýchkoliv hodnot (zde je to sloupec s kódy všech zemí) a vrací jejich počet. Do počtu však nezahrnuje hodnoty NULL. V našem případě však tato situace nenastane, neboť atribut *stat.kod* je primárním klíčem tabulky.

I když je de facto vrácena jen jediná hodnota, formálně se jedná o tabulku s jedním řádkem a jedním sloupcem (stále platí, že SELECT vždy vrací tabulku).

```
 count
-------
   192
(1 row)
```

Jiný počet vrací počet odkazů na kontinenty, neboť některé státy/řádky obsahují v tomto sloupci NULL (výsledek: 179 nepočítají se státy Oceánie):

```
SELECT COUNT(staty.kontinent) FROM staty;
```

V případě agregační funkce COUNT existuje i speciální zápis počítající všechny záznamy (bez ohledu na nějaká NULL)

```
SELECT COUNT(*) FROM source;
```

Agregovat lze samozřejmě i dočasné tabulky získané filtrací (počítají se jen filtrované řádky) a/nebo spojením. Snadno lze například zjistit počet států s rozlohou vyšší než 1 000 000 km^2.

```
SELECT COUNT(staty.kod)
 FROM staty
 WHERE staty.rozloha >= 1000000;
-- výsledek: 29
```

Resp. počet českých měst v databázi:

```
SELECT COUNT(*)
 FROM mesta JOIN staty ON mesta.kod_statu = staty.kod
 WHERE staty.kod='CZE';
-- výsledek: 10
```

Statistické agregační funkce

Statistické agregační funkce patří mezi klasické redukující agregáty a jsou podporovány na všech RDBMS (u četných uživatelů jsou to také jediné agregační funkce, které znají a i v literatuře často platí agregační funkce = statistika).

AVG (průměr), SUM (součet), MIN (minimální hodnota), MAX (maximální hodnota)

Použití je zřejmé, proto uvádím jen jediný příklad:

```
SELECT AVG(staty.rozloha) FROM staty;
-- výsledek: 693549.54
```

Logické agregační funkce

Logické hodnoty lze agregovat pomocí dvou funkcí:

bool_and (alternativně *every*) = logický součin všech hodnot (vrací *true*, jsou-li všechny hodnoty pravdivé)

bool_or = logický součet všech hodnot ((vrací *true*, je-li alespoň jedna hodnota pravdivá)

Přímá použitelnost těchto funkcí je ovlivněna skutečností, že tabulky s boolovskými atributy nejsou příliš časté (v naší databázi není ani jediná).

Standard SQL však umožňuje volat agregační funkce i na složitější výrazy. Výraz se provede na každém řádku dočasné tabulky a agreguje se jeho výsledek.

Příklad: Zjistěte zda je v databázi město s méně než 100 obyvateli.

```
SELECT bool_or(obyvatel < 100) from mesta;
```

Výraz by měl vracet hodnotu true resp. 1 (podle representace logických hodnot).

Restruktulizační agregace nejsou prozatím příliš rozšířené a tím i standardizované. PostgreSQL podporuje agregace do řetězce, formátu JSON a XML.

Nejjednodušší je agregace řetězcových hodnot.

Příklad: Vytvořte řetězec obsahující jména všech měst v ČR (z databáze měst). Jména budou oddělena čárkou.

```
SELECT string_agg(mesta.jmeno, ', ')
FROM mesta
WHERE kod_statu='CZE';
```

Výstup: *Praha, Brno, Ostrava, Plzen, Olomouc, Liberec, Ceské Budejovice, Hradec Králové, Ústí nad Labem, Pardubice.*

Restruktulizační agregace mohou být i komplexnější:

```
SELECT string_agg(mesta.jmeno || ' in ' || staty.jmeno, ', '
             ORDER BY mesta.obyvatel DESC)
FROM mesta JOIN staty ON mesta.kod_statu = staty.kod
WHERE mesta.obyvatel > 8000000;
```

Přehledné zobrazení megalopolí vychází ze spojení tabulky státy a města (vztah 1:N). Agregační funkce vypisuje řetězec, který vznikne spojením řetězců, které sami vznikají zřetězením jména města, anglické předložky a (anglického) jména státu. Navíc jsou města ve výsledném řetězci uspořádány sestupně podle počtu obyvatel. Sekce ORDER BY však nemůže být umístěna na svém obvyklém místě na konci SELECT, neboť by se aplikovala až na výsledný agregovaný řetězec (uspořádání tabulky s jediným řádkem a jediným sloupcem nemá příliš smysl). PostgreSQL proto podporuje zápis klauzule ORDER BY přímo uvnitř agregační funkce.

Fáze výpočtu agregační funkce:

- spojení tabulek města a státy (JOIN)
- filtrování (selekce měst s počtem obyvatel > 8000000), WHERE
- výsledná dočasná tabulka je vstupem do agregační funkce
  - setřídění rozšířené tabulky měst podle počtu obyvatel
  - zřetězení atributů *mesta.jmeno* a *staty.jmeno* pro každý záznam
  - spojení výsledných řetězců do jediného seznamu

Výstup: *Mumbai (Bombay) in India, Seoul in South Korea, São Paulo in Brazil, Shanghai in China, Jakarta in Indonesia,*

Karachi in Pakistan, Istanbul in Turkey, Ciudad de México in Mexico, Moscow in Russian Federation, New York in United States.

8.7.2 Seskupování

Sekce GROUP BY umožňuje seskupovat řádky tabulek podle společných hodnot ve sloupci (sloupcích).

Každá skupina řádků je nahrazena jediným řádkem, který obsahuje jen dva typy informací:

1. společná hodnota sloupce resp. sloupců, podle nichž se seskupuje, nebo klasifikační funkce
2. agregované hodnoty z ostatních sloupců

Nejjednodušší je seskupování hodnoty podle jednoho sloupce.

Příklad:

Počet měst v jednotlivých státech (samozřejmě jen měst se záznamem v naší databázi)

```
SELECT kod_statu, COUNT(*)
  FROM mesta
  GROUP BY kod_statu
  ORDER BY kod_statu;
```

Příkaz opravdu vypíše seznam států s počtem měst, ale identifikuje je je jen zkratkou. Proto příkaz vylepšíme využitím relace mezi tabulkou státy a města (tj. s využitím JOIN).

```
SELECT staty.jmeno, COUNT(*)
  FROM mesta JOIN staty ON mesta.kod_statu = staty.kod
  GROUP BY staty.jmeno
  ORDER BY staty.jmeno;
```

Seskupování se nyní provádí podle jména státu. To není optimální, neboť jednodušší a bezpečnější by bylo využití primárního klíče *staty.kod* (je jistě unikátní a lépe se porovnává), avšak pak by jej nebylo možno vypsat (vypsat lze pouze hodnotu klíče, podle něhož se seskupovalo, přestože víme, že každému kódu odpovídá právě jeden stát).

Výsledek:

```
              jmeno              | count
---------------------------------+-------
 Afghanistan                     |     4
 Albania                         |     1
 Algeria                         |    18
...
 Czech Republic                  |    10
```

```
...
```

Seskupovat však lze i podle více sloupců (víceúrovňová klasifikace).

Příklad:

Příklad zobrazte počet obyvatel největšího města ve státě, setříděný primárně podle kontinentu, sekundárně podle obyvatel (takže lze snadno zjistit státy s největším městy na kontinentě).

```
SELECT kontinenty.jmeno, staty.jmeno, max(mesta.obyvatel) as ob
                  FROM mesta JOIN staty ON mesta.kod_statu = staty.kod
                  LEFT JOIN kontinenty ON staty.kontinent = kontinenty.id
 GROUP BY kontinenty.jmeno, staty.jmeno
 ORDER BY kontinenty.jmeno, ob DESC;
```

Pro kontinenty je použito vnější spojení, aby se zobrazily i státy bez kontinentů. Pořadí seskupování je klíčové (opačné

seskupení by bylo nezajímavé, neboť každý stát leží jen na jednom kontinentě). Třídění se provádí až po seskupení (a projekci) takže můžete řadit podle klasifikačních sloupců resp. podle agregovaných hodnot (jako je tomu i zde).

```
     jmeno     |                jmeno                 |    ob
---------------+--------------------------------------+----------
 Africa        | Egypt                                |  6789479
 Africa        | Congo, The Democratic Republic of the |  5064000
...
 Australia     | Australia                            |  3276207
 Europe        | United Kingdom                       |  7285000
 Europe        | Germany                              |  3386667
...
               | New Zealand                          |   381800
               | Papua New Guinea                     |   247000
...
```

Některé sloupce nejsou pro seskupení vhodné, neboť obsahují příliš mnoho různých hodnot. Nejsou to jen sloupce s unikátními hodnotami (např. kandidátní klíče), ale i sloupce, kde je pravděpodobnost shody malá (rozlohy států, počet obyvatel, či rok vzniku).

Příklad:

Příkaz, který zjistí počet nově vzniklých států v každém roce je jednoduchý:

```
SELECT nezavislost, COUNT(*) as pn
 FROM staty
 GROUP BY nezavislost
 ORDER BY pn DESC;
```

I když zjistíme zajímavé údaje (např. v letech 1991 resp. 1960 vzniklo pokaždé 18 států, jistě víte proč), je většina roků rovna jedné či nule. Zajímavější by bylo zjištění počtu vzniklých států podle století:

```
SELECT nezavislost / 100 + 1 as stoleti, COUNT(*) as pn
FROM staty
GROUP BY nezavislost / 100
ORDER BY pn DESC;
```

Využíváme skutečnosti, že dělíme celá čísla (tj. výsledkem je celé číslo). Po přepočet na století přičtěme jedničku (v tomto případě PostgreSQL správně pochopilo, že vypisovaný výraz závisí na výraze klasifikačním). Přičtení jedničky nefunguje u letopočtů před Kristem, ale to nevadí, neboť všechny údaje před Kristem jsou nerelevantní.

Výstup:

```
20 | 149
19 |  27
 9 |   4
15 |   2
18 |   2
16 |   1
```

Výstup není překvapivý. Státy jsou dost krátkodeché objekty (i když v 9. století byl jakýsi country-boom).

klasifikaní funkce

Funkce, která převádí hodnoty na klasifikační skupiny (tzv. **klasifikaní funkce**) může být i komplexnější.

Příklad:

Zobrazte počet států, které jsou větší či naopak menší než ČR (v jedné tabulce).

```
SELECT CASE
       WHEN rozloha > 78866 THEN 'vetsi'
       WHEN rozloha < 78866 THEN 'mensi'
       ELSE 'CR'
       END , COUNT(*) as pn
FROM staty
GROUP BY CASE
       WHEN rozloha > 78866 THEN 'vetsi'
       WHEN rozloha < 78866 THEN 'mensi'
       ELSE 'CR'
       END
ORDER BY pn DESC;
```

Klasifikační funkce mapuje rozlohu na pouhé tři hodnoty. Řetězec „větší“ pro rozlohy větší než rozloha ČR, řetězec „menší“ pro rozlohy menší a „ČR“ pro rozlohy stejné. Využívá k tomu konstrukci CASE. Dotaz funguje (113 států je větších, 78 menších, požadovanou rozlohu má jen ČR = rozloha České republiky je pod mediánem).

Zásadní nevýhodou je nutnost dvojí specifikace funkce, jednou v sekci GROUP BY, podruhé v části SELECT. Řešení není jednoduché, protože se seskupování (GROUP BY) provádí před projekcí (SELECT). Výslednou hodnotu proto nejde pojmenovat (to lze jen v části SELECT a to už je pozdě). Nejjednodušším (přenositelným) řešením je využití vnořeného SELECT, kdy je výstup jednoho dotazu zdrojem dat pro dotaz edruhý (vnější).

```
SELECT typ, COUNT(*)
FROM
 (SELECT CASE
        WHEN rozloha > 78866 THEN 'vetsi'
        WHEN rozloha < 78866 THEN 'mensi'
        ELSE 'CR'
       END AS typ FROM staty) AS source
GROUP BY typ;
```

Vnitřní dotaz vrací pro každý stát jen jeden sloupec s řetězcem klasifikujícím stát („větší“, „menší“, „ČR“). Teto dočasná tabulka je pojmenována *source* (jméno není důležité, ale PostgreSQL jej vyžaduje). Tato tabulka (fyzicky nikdy nevznikne) slouží jako zdroj vnějšího dotazu. Tento dotaz může používat jména sloupců dočasné tabulky a tak může využít jméno *typ* pro seskupování a zobrazení (seskupuje se podle jediného sloupce a počítá se počet řádků se stejnou hodnotou).

Sekce HAVING

Relativně častým požadavkem je filtrace výsledků ze seskupené tabulky (zajímají nás například jen skupiny s dostatečně velkým počtem členů). Pro filtraci nelze použít sekci WHERE, neboť ta se provádí ještě před seskupením.

Řešením by byl vnitřní dotaz podobného typu jako v předchozím příkladu (vnitřní cyklus by seskupoval a vnější poté filtroval), ale standard jazyka pro tento účel podporuje zvláštní sekci s názvem HAVING.

Příklad:

Zjistěte jakým písmenem nejčastěji začínají jména hlavních měst. Za město budeme pro účel tohoto dotazu považovat jen sídla s více než 5000 obyvatel.

```
SELECT left(mesta.jmeno,1) AS pznak, COUNT(*) as pocet
  FROM staty JOIN mesta ON staty.hmesto = mesta.id
  WHERE mesta.obyvatel > 5000
  GROUP BY left(mesta.jmeno,1)
  HAVING COUNT(*) >= 10
  ORDER BY pocet DESC
```

Dokaz se provádí v tomto pořadí:

1. FROM = zdrojem spojení tabulek měst a států (podle vztahu hlavní město)
2. WHERE = odfiltrování měst s více než 5000 obyvatel
3. GROUP BY = seskupení podle prvního písmene jména města (funkce *left(s,n)* vrací *n* prvních znaků řetězce *s*)
4. HAVING = odfiltrují se jen ty řádky, které seskupují více než 10 měst
5. SELECT = vypíše se seskupovací hodnota a počet seskupených řádků
6. ORDER BY = a setřídí se sestupně podle počtu řádků

Výsledek je následující:

```
 pznak | pocet
-------+-------
 B     |    24
 A     |    16
 S     |    15
 M     |    15
 P     |    15
 K     |    13
 L     |    12
 T     |    11
(8 rows)
```

Jak lze vidět, naše republika je v tomto případě zcela v hlavním proudu.

OTÁZKY

1. Na kolik sloupců může odkazovat cizí klíč?
2. Jakou aritu má operace vnitřního spojení?
3. Jaký rozměr má výsledná tabulka u kartézského součinu (tj. CROSS JOIN)? (= počet jejích sloupců s řádků)
4. Proč není spojení označené slovem NATURAL skutečně přirozené?
5. Co je ekvivalence?
6. DISTINCT by se neměl používat přímo nad jednotlivými tabulkami (dotazy jejichž zdrojem je jediná tabulka). Proč?
7. U jakého typu množinového spojení je přínosné využívání verze se specifikací ALL a bez ní (tj. s explicitním DISTINCT)?
8. Jaký je rozdíl mezi COUNT(*) a COUNT(atribut)?
9. Podle jakých sloupců (atributů) se téměř nikdy přímo neseskupuje?
10. Jaký je rozdíl mezi sekcí WHERE a HAVING?

OTÁZKY K ZAMYŠLENÍ

1. Otázka k zamyšlení: Promyslete automatické akce pro zajištění referenční integrity v databázi lékařů a pacientů, pokud je z databáze odebrán záznam obvodního lékaře? Jak by se to mělo projevit na záznamech pacientů, kteří tento záznam odkazovali.
2. Otázka k zamyšlení: Speciálním případem relace 1:N je relace 1:1. Jak ji poznáte v databázi. Můžete zajistit její referenční integritu?
3. Některé DBMS nepodporují úplná vnější spojení. Jakým způsobem je můžete nahradit.
4. Jak rychlá může být operace odstranění duplikátů (= jakou má asymptotickou časovou složitost). Jaké pomocné datové struktury lze použít pro její urychlení?
5. Jak můžete zjistit medián hodnoty v nějaké sloupci?
6. Při seskupování do dvou skupin (jak tomu bylo v případě hledání států větší a menších než ČR) existuje alternativní řešení nevyužívající GROUP BY, ale množinovou operaci. Navrhněte jej?

ÚKOLY

1. Navrhněte příkaz SELECT vypisující jména států společně s kontinenty, na nichž leží. Jsou vypsány všechny státy?
2. Navrhněte příkaz SELECT vypisující jména všech měst, která ve svém jméně obsahují jméno svého státu (a tak odstranili první nedostatek ukázkového příkladu). Rada: spojovací predikát může obsahovat i logické spojky.
3. Navrhněte příkaz SELECT vypisující všechna města, která leží ve státě s neurčeným kontinentem (tj. v Oceánii). Zohledněte situaci, kdy je (a) znám stát pro každé město, ale nikoliv kontinent pro každý stát (skutečný stav) nebo (b) existují města ležící mimo známé státy a zároveň státy ležící mimo kontinenty (nejobecnější stav).
4. Vyzkoušejte obě alternativní řešení pro výpis států s alespoň jedním velkoměstem (včetně získání doplňkového seznamu, tj. států bez velkoměst). Alternativní řešení by měla využívat operátorů EXISTS resp. IN a vnořeného dotazu.

ODKAZY NA LITERATURU

- HERNANDEZ, Michael J a John VIESCAS. *Myslíme v jazyku SQL: tvorba dotazů*. 1. vyd. Praha: Grada, 2004, 378 s. Knihovna programátora (Grada). ISBN 80-247-0899-x. Část III: Práce s více tabulkami a Část IV: Shrnutí a seskupování dat.

9 Transakce a triggery, zotavení z chyb

CÍLE KAPITOLY

Jedním z klíčových rysů moderních relačních databázových systémů je důraz na integritu (neporušitelnost) dat. V této kapitole se dozvíte jaké záruky Vám databáze dává a jak je můžete co nejlépe využít.

Prakticky se seznámíte především se dvěma mechanismy

- transakce — explicitní označení těch částí kódu, které atomicky přecházejí z jednoho konzistentního stavu do druhého
- triggery — procedurální specifikace integrity databáze (co vše je třeba provést, aby

9.1 Transakce

9.1.1 Záruky ACID

transakce

Transakce umožňují pro určitou databázovou operaci (= množinu databázových příkazů) zajistit splnění čtyř záruk: atomičnosti, konzistence, izolovanosti a trvanlivosti (ang. atomicity, consistency, isolation, durability = **ACID**).

atomičnost

Atomičnost zajišťuje, že každá transakce buď skončí úspěšně nebo se vůbec neprovede (princip vše nebo nic). Jestliže dojde uvnitř transakce k chybě, pak skončí s chybou celá transakce a stav databáze se nezmění,

konzistenci

Konzistence zaručuje, že transakce převede databázi z jednoho konzistentního (platného) stavu do druhého konzistentního stavu. Databáze je v konzistentním stavu, pokud platí všechna integritní omezení (a samozřejmě i omezení doménová). Navíc musí být provedeny všechny závislé činnosti jako jsou kaskádní mazání či spouštění triggerů (a jejich kombinací) a to opět bez narušení integritních omezení.

izolovanost

Záruka izolovanosti zaručuje, že případné paralelní provádění více transakcí povede ke stavu, kterého by bylo dosaženo postupným (tj. sériovým) prováděním těchto transakcí (tj. jako kdyby byly transakce provedeny jedna po druhé v pořadí daném jejich příchodem).

trvalost (výsledků)

trvalost zaručuje uchování všech výsledků potvrzených transakcí (tj. transakcí, jejichž ukončení bylo signalizováno zprávou) a to i v případě hardwarových chyb, výpadků napájení nebo komunikačních problémů (u distribuovaných databází). Trvalost nemůže být nikdy absolutní, týká se jen prostředků pod absolutní správou DBMS tj. např. závisí na trvalosti dat v nevolatilních zařízeních (pevných discích).

Každý jednotlivý SQL příkaz je transakcí a to i tehdy, pokud vykonává více dílčích činností resp. činností závislých. Tj. transakcí je i výmaz více prvků (smažou se všechny nebo žádný, provedou se všechny triggery a kaskádní nastavení či výmazy, výsledek je trvalý).

9.1.2 Explicitní transakce

V některých případech je však nutné zajistit provedení transakce složené s více příkazů.

Klasický příklad: převod peněz mezi účty v rámci jedné databáze. Atomicky (trvale, izolovaně) se musí provést snížení zůstatku u výdejce a zvýšení zůstatku u příjemce.

Nevzniknou tak tyto nepřípustné situace:

- částka byla odečtena, ale nebyla přičtena (okradení výdejce)
- jiná aplikace (např. kontrola celkové bilance) viděla mezistav (a objeví tak dočasnou nerovnováhu)
- transakce byla potvrzena, ale nebyla uložena (tj. může být vytištěno potvrzení o transakci, která v databázi není)

Transakce složená z více příkazů musí začínat příkazem:

```
BEGIN [TRANSACTION]
```

a končí buď příkazem

```
COMMIT
```

pokud ji chceme dokončit (= potvrdit)

nebo

```
ROLLBACK
```

pokud se ji chceme zrušit a vrátit se do původního stavu (např. pokud zjistíme, že nemůžeme dojít ke konzistentnímu stavu a tento stav nelze kontrolovat pomocí integritních omezení).

Identifikace explicitních transakcí může být náročná (a závislá na zvoleném modelu). Musíme se přitom vyvarovat dvou chyb:

příliš úzká transakce — do transakce nezahrneme příkazy, které do ní patří. Pak neplatí žádná ze záruk a databáze resp. celá naše aplikace se může dostat do nedefinovaného stavu (přitom selhání má většinou jen velmi malou ale nikoliv nulovou pravděpodobnost)

příliš široká transakce — do transakce zahrneme příkazy, které do ní nepatří. Výsledkem může být výrazné zpomalení celé databáze resp. zvýšené požadavky na prostředky (paměť, apod.)

9.1.3 Úrovně izolovanosti

uváznutí

Dosažení plné izolovanosti není zadarmo, zvyšuje se režie a při vyšším zatížení databáze (hlavně zápisy) může dojít ke zdánlivému (*deadlock*). Transakce na sebe čekají tak dlouho, že doba odezvy přestává být akceptovatelná (i když je vždy, alespoň teoreticky konečná). U skutečného deadlocku je doba čekání potenciálně nekonečná (a lze to matematicky dokázat).

RDBMS proto nabízí tzv. úrovně izolovanosti , které nabízejí **lepší časové a prostorové efektivity za cenu nižší míru izolace**.

Tyto režimy jsou definovány na základě negativních projevů, kterým daná úroveň (a úrovně nižší) nezabraňuje. detailní chování však závisí na implementaci.

Existují dvě hlavní implementace izolace:

- dvojfázové zamykání (klasické řešení předpokládané ve standardu)
- *Multiversion Concurrency Control* (MCC, použité v PostgreSQL)

Uncommited read

De facto bez izolace transakcí tj. není prováděna žádná serializace.

Lze číst i nepotvrzené změny jiné transakce (tzv. *dirty read*). V PostgreSQL s MCC není podporována.

```
Transakce 1                             Transakce 2
/* dotaz */
SELECT age FROM users WHERE id = 1;
/* vraci 20 */
                                        /* zmena */
                                        UPDATE users SET age = 21 WHERE id = 1;
                                        /* transakce neukoncena */
/* opakovaný dotaz */
SELECT age FROM users WHERE id = 1;
/* vraci (chybne) 21 -- dirty read
                                       ROLLBACK;
```

Read commited

Základní úroveň izolace (nejnižší v PostgreSQL)

Lze vidět jen **potvrzená** (*commited*) data. Lze však také vidět změny, které jsou výsledkem transakcí, které byly spuštěny později (tj. čtení se mohou lišit) — nekonzistentní opakovaná čtení tzv. fantómová čtení.

Efektivní a snadno se používá, vhodné pro jednodušší transakce.

```
Transakce 1                             Transakce 2
/* dotar */
SELECT * FROM users WHERE id = 1;
                                        /* zmena */
                                        UPDATE users SET age = 21 WHERE id = 1;
                                        COMMIT;
/* opakovany dotaz 1 */
SELECT * FROM users WHERE id = 1;
/* vraci (chybne) 21 -- fantomove cteni
COMMIT;
```

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:

1. k dlouhému čekání (zdánlivému deadlocku) při dvojfázovém zamykání
2. k vyvolání výjimky, pokud se nenajde prováděcí plán (MCC); transakce je anulována a musí být provedena znova (explicitně)

9.2 Triggery

Trigger (česky spouštěč) je procedurální kód, který je automaticky aktivován jako reakce na určitou událost v databázi. Událost se typicky týká tabulek či záznamů.

Hlavní funkcí triggerů je udržování integrity databáze. Zajišťují například vložení všech závislých záznamů při vložení záznamu hlavního, nebo aktualizaci všech závislých hodnot.

Triggery jsou velmi užitečné, ale jejich nadměrné využívání může být příznakem chybného návrhu nebo použití berliček obcházejících standardní nástroje udržování integrity.

9.2.1 Definice triggerů

Podpora triggerů v jednotlivých databázích se liší. Zvláště nepříjemná je okolnost, že se může mírně lišit i sémantika, tj. daný trigger funguje ve více databázích, ale výsledek se může v některých (často jen hraničních) situacích mírně lišit.

Při definici triggerů nad řádky tabulky (což je nejtypičtější použití) je nutno zvolit:

A) při jaké operaci se trigger aktivuje — podporovány jsou INSERT, UPDATE a DELETE.

B) v jakém pořadí se povede trigger a hlavní činnost dané události — existují triggery vykonávané před (BEFORE) i po (AFTER) hlavní akci. Speciálním případem je vykonání namísto (INSTEAD OF) původní akce.

Tělem triggeru může být obecně jakýkoliv SQL příkaz s vedlejším efektem, jednotlivé RDBMS však mají různá omezení. PostgreSQL podporuje pouze tzv. uložené procedury (které však mohou obsahovat jakékoliv rozumné příkazy).

Příklad:

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

Tento trigger se spustí pokud je aktualizován sloupec *balance* v tabulce *accounts* (nové volání pro každý řádek tabulky).

Uložená procedura může provést několik činností (jejichž hlavním cílem je udržení konzistentního stavu).

Základní možností je kontrola aplikačního kontraktu, který nelze vyjádřit pomocí běžných omezení, zde např. maximálního výběru:

```
CREATE OR REPLACE FUNCTION account_body()
 RETURNS trigger AS
$$
BEGIN
IF OLD.balance - NEW.balance > 10000 THEN
      RAISE 'withdrawal␣limit'
     USING ERRCODE = 'integrity_constraint_violation';
END IF;

RETURN NEW;
END;
$$
```

Funkce triggeru musí být definovány jako bezparametrická a vracet pseudotyp trigger. Tělo funkce je (jak je v PostgreSQL zvykem) tvořeno dlouhým řetězcem (ten je ohraničen oddělovači $$).

Interní kód je jasný. Pokud je pokles zůstatku větší než 10000, pak je vyvolána výjimka. Ta ukončí nejen trigger, ale i celou transakci (tj. účet se nezmění) a může být zachycena v klientském programu (a uživatel tak může být informován proč se bankovní operace nezdařila).

K záznamu se přistupuje pomocí identifikátoru OLD a NEW (OLD obsahuje stav před změnou a NEW, pokud by se změna provedla).

Akce však může být složitější:

```
CREATE OR REPLACE FUNCTION account_body()
  RETURNS trigger AS
$$
BEGIN
IF NEW.balance <> OLD.balance THEN
INSERT INTO transaction_audits
          (account_id,old_balance, new_balance, time)
     VALUES(OLD.id, OLD.balance, NEW.balance, now());
END IF;

RETURN NEW;
END;
$$
```

V tomto případě je v reakci na změnu zůstatku přidána záznam do tabulky *transaction_audit* (míní se bankovní transakce). Ten obsahuje informaci o účtu, zůstatcích a času provedení.

Poslední možností je změna aktualizované hodnoty, resp. libovolného atributu v novém záznamu (NEW).

9.2.2 Použití triggerů

Jak již bylo řečeno triggery slouží primárně k zajištění integrity v případě, že standardní prostředky (omezení, referenční integrita) nedostačují (resp. pokud nejsou k v daném RDBMS dispozici). Výhodou je skutečnost, že se provádějí automaticky a v rámci jediné transakce (a to i v případě, že se volají rekurzívně). Pokud tedy dojde při použití triggeru k chybě (např. narušení jiného integritního omezení) pak je revokována celá transakce (včetně původního příkazu), tj. systém zůstává ve všech případech konzistentní.

Nebezpečí: při nadměrném použití triggerů může dojít k neřízenému kaskádovému efektu se spuštěním stovek triggerů, resp. k podivným vedlejším efektům (záznam je změněn několikrát, přičemž cílový stav je téměř neodhadnutelný). Někdy může dokonce dojít k zacyklení (výsledkem je buď nekonečný cyklus změn nebo vyhození výjimky).

Z tohoto důvodu lze doporučit jen střídmé využívání triggerů se jednoduchou funkcí. pokud možno bez složitějších závislostí. Vždy musíte vědět jaké triggery a v jakém pořadí se volají.

OTÁZKY

1. K jakým chybám dochází při specifikaci transakcí?
2. Co je (zdánlivé) uváznutí?
3. Jaké úrovně izolovanosti se nejčastěji používají? Uveďte jejich výhody a nevýhody.
4. Jak souvisí triggery s transakcemi?

OTÁZKY K ZAMYŠLENÍ

1. Co se stane pokud transakci explicitně neukončíte?
2. Jak se transakce používají u read-only databází?
3. Jak se můžete eliminovat negativní efekt netolerovatelného uváznutí?
4. V jakém pořadí se triggery spouštějí? (konzultujte dokumentaci RDBMS)

ÚKOLY

1. Vytvořte minimalistické tabulky *accounts* a *transactions_audits* a vyzkoušejte funkci triggerů.
2. Vytvořte trigger, který sníží zůstatek pro provedení změny o -1 (nepříliš dobrá implementace poplatku za změnu stavu).
3. Vytvořte trigger, který zabrání vymazání účtu pokud je zůstatek nenulový.

ODKAZY NA LITERATURU

- Pavel Stěhule. *Transakce a izolace transakcí v databázích* [online]. Root.cz. 13.5.2009. Dostupné na `http://www.root.cz/clanky/transakce-a-izolace-transakci-v-databazich/`
- *Triggers* [online]. Database concepts, Oracle Help Center. Dostupné na `https://docs.oracle.com/cd/B28359_01/server.111/b28318/triggers.htm`