MySQL

Databázové systémy

Distanční opora pro předmět Databázové systémy 1 a Databázové systémy 2

Sestavil: Jan Kubrický – poslední editace: 15.9.2021

OBSAH

OBSAH **1**

1 ÚVOD **5**

2 ZÁKLADY DATABÁZÍ **6**

Součásti databáze 6
Nejrozšířenější modely databází 9
Shrnutí 12

3 RELAČNÍ DATABÁZE **13**

Databázová terminologie 13
Vztahy neboli relace 14
Získávání dat 16
Systém pro správu relační databáze RDBMS 16

4 LOGICKÝ A FYZICKÝ NÁVRH DATABÁZE **17**

Tabulky 17
Sloupce a datové typy 17
Klíče 18
Indexy 19
Hodnota NULL 19
Integrita dat 19

5 NORMALIZACE DATABÁZÍ **21**

První normalizovaná forma 22
Druhá normalizovaná forma 23
Třetí normalizovaná forma 24
Boyce-Coddova normalizovaná forma 25
Čtvrtá normalizovaná forma 26
Pátá normalizovaná forma 28

6 MYSQL **30**

Vlastnosti MySQL 30
Připojení k serveru MySQL 31
Po instalaci MySQL 32
Přihlášení k PHPMyAdmin (PMA) 32
Nastavení účtu 33
Vytvoření první databáze 35

7 SQL V MYSQL **36**

Vkládání nových záznamů do tabulky 36
Získávání informací z tabulky 37
Vyhledávání vzorů 38
Řazení záznamů 38
Omezení počtu výsledků 39
Vracení odlišných záznamů 40
Agregační funkce 40
Odstraňování záznamů 42
Změna záznamu 42
Další funkce 44
Používání funkcí data 45

8 VYTVÁŘENÍ POKROČILEJŠÍCH DOTAZŮ **47**

Přiřazení nového záhlaví sloupcům pomocí AS ... 47
Spojování sloupců pomocí CONCAT() ... 47
Práce s více tabulkami ... 48
Seskupení v dotazu ... 50
Klauzule HAVING ... 51
SOUHRN ... 52

9 DATOVÉ TYPY A TYPY TABULEK ... 53

Datové typy ... 53
Číselné typy sloupců ... 53
Kategorie celočíselných typů ... 54
Kategorie s pohyblivou desetinnou čárkou ... 55
Další číselné typy ... 56
Zásady volby číselného typu ... 56
Řetězcové typy ... 57
Sloupce typu datum a čas ... 60
Typy tabulek ... 61
Tabulky MyISAM ... 61
Tabulky InnoDB ... 62
Dočasné tabulky ... 63
Tabulky MRG_MYISAM ... 64
Tabulky MEMORY (HEAP) ... 65
Tabulky ARCHIVE ... 66
Tabulky CSV ... 66
Optimalizace tabulek ... 66

10 POKROČILÝ JAZYK SQL ... 67

Operátory ... 67
Proměnné a další konstrukce ... 69

11 SPOJENÍ JOIN ... 72

Typy spojení ... 73
Vnitřní spojení ... 74
Vnější spojení ... 75
Přirozená spojení ... 77
Pro zájemce - Spojení STRAIGHT_JOIN ... 79

12 VNOŘENÉ DOTAZY SELECT ... 82

13 ZAMYKÁNÍ A TRANSKACE ... 84

Zamykání ... 84
Transakce ... 86

14 INDEXY ... 94

Vytvoření indexu ... 94
Primární klíč ... 95
Ordinální (obvyklý) index ... 95
Fulltextový index ... 96
Jedinečný index ... 99

15 POHLEDY ... 101

16 ULOŽENÉ PROCEDURY A TRIGGERY ... 104

Uložené procedury (UP) – primárně pro předmět Databázové systémy 2 ... 104
Tvorba UP ... 104
Interní uložení UP ... 106

SYNTAXE A PRVKY JAZYKA 107
VOLÁNÍ ULOŽENÝCH PROCEDUR 108
PARAMETRY A NÁVRATOVÉ HODNOTY 110
ZAPOUZDŘENÍ PŘÍKAZŮ 110
PROMĚNNÉ 111
DEKLAROVÁNÍ 112
VĚTVENÍ KÓDU 112
CYKLY 113
ZPRACOVÁNÍ CHYB 115
PŘÍKLADY UP 116
TRIGGERY – PŘEDMĚT DATABÁZOVÉ SYSTÉMY 1 119

17 ZÁLOHOVÁNÍ DATABÁZE 123

ZÁLOHOVÁNÍ TABULEK POMOCÍ BACKUP 123
OBNOVENÍ TABULEK MYISAM POMOCÍ RESTORE 123
ZÁLOHOVÁNÍ DATABÁZE POMOCÍ NÁSTROJE MYSQLDUMP 124

18 PROGRAMOVÁNÍ S MYSQL 125

DATOVÁ SPOJENÍ 125
DOTAZY 126

19 PŘÍLOHA 1 137

PŘÍKLAD NORMALIZACE DATABÁZE 137

20 PŘÍLOHA 2 141

21 PŘÍLOHA 3 145

PŘÍKLADY SPOJENÍ POMOCÍ JOIN 145
SQL PŘÍKAZY PRO VYTVOŘENÍ DATABÁZE S TABULKAMI 152
DISKUZE K DATOVÝM TYPŮM 155

22 PŘÍLOHA 4 159

ODKAZY NA FUNKCE MYSQL 159

23 PŘÍLOHA 5 160

MYSQL WORKBENCH 160

LITERATURA 161

1 Úvod

Ještě před pár desítkami let byly databáze spjaty pouze se špičkovými specializovanými laboratořemi. Dnes jsou databáze neodmyslitelnou součástí světa informačních technologií. S databázemi pracujeme denně. Ať už se jedná o bankovní transakce, cestovní rezervace, vyhledávání na webu atd.

Podobně jako je tomu u mnoha jiných rychle se vyvíjejících technologií, i v oblasti databází existuje nápor nových produktů, které se snaží usměrnit sada standardů a norem. Navíc vzniklo také několik různých databázových modelů, mezi nimiž má zatím hlavní roli tzv. **relační model**.

Tento studijní text je primárně zaměřen na 3 části:

- Základní teorie databází.
- Normalizace databáze.
- Databázový systém MySQL.

Než začneme využívat konkrétního systému pro řízení a správu databáze, je potřeba seznámit se základními pojmy a teorií relačního modelu databáze. Tomu je věnována první část. V druhé části si představíme pojmy a činnosti, související s tzv. normalizací databáze. Tedy správnou technikou samotného návrhu databáze. V následující části se budeme věnovat databázovému systému MySQL.

Problematika okolo databázového systému MySQL je velmi rozsáhlá, stejně jako je tomu i u jiných databázových systémů a proto se do tohoto studijního materiálu všechno nevešlo. Primárně se zaměříme na implementaci standardů SQL, spojení JOIN a tvorbu procedur. Administrace a správa MySQL je probírána jen okrajově a pokud se budete chtít více ponořit do této problematiky, budete muset dále sáhnout po některé z příruček či odborných publikací.

Studijní materiál z velké části čerpá ze zdrojů, které jsou uvedeny v závěru. A vznikl tedy sestavením teoretických poznatků ve spojení s praktickými příklady. Příklady uvedené v textu jsou rovněž vloženy do externích souborů.

Přeji Vám úspěšné studium, které by ideálně mělo kombinovat Vaši účast na seminářích s doplněním o četbu tohoto studijního textu.

2 Základy databází

Tato část je věnována základním pojmům, stavbě databáze a definicím. Rozebereme nedůležitější součásti databáze a podíváme se na nejrozšířenější modely databází.

Součásti databáze

Databáze je kolekce vzájemně souvisejících dat, s nimiž pracujeme jako s ucelenou jednotkou. To, co databáze odlišuje od obyčejných seznamů, jsou jejích charakteristické vlastnosti a součásti, které obyčejný soubor nemá. Konkrétně se jedná o:

- Správa databázového systému (DBMS)
- Vrstvy datové abstrakce
- Fyzická datová nezávislost
- Logická datová nezávislost

Databázový systém (DBMS)

Je SW dodávaný příslušným výrobcem databází. Mezi nejznámější a nejpoužívanější se řadí:

- MS Access
- ORACLE
- MS SQL Server
- PostgreSQL
- MySQL

Databázový systém zajišťuje všechny základní služby, nezbytné pro organizaci databáze a udržení v chodu, jako například:

- Přesouvání dat do fyzických souborů a naopak.
- Správa současného přístupu uživatelů k datům, včetně provádění takových opatření, které zabrání vzájemným konfliktům
- Správa transakcí, které znamenají současné vykonávání několika změn v databázi v rámci jedné nedělitelné jednotky.
- Podpora dotazovacího jazyka, který tvoří množina příkazů pro načítání dat z databáze.
- Mechanizmy pro zálohování databáze.
- Bezpečnostní mechanizmy, které zabraňují v neoprávněném přístupu k datům a v neoprávněných modifikacích.

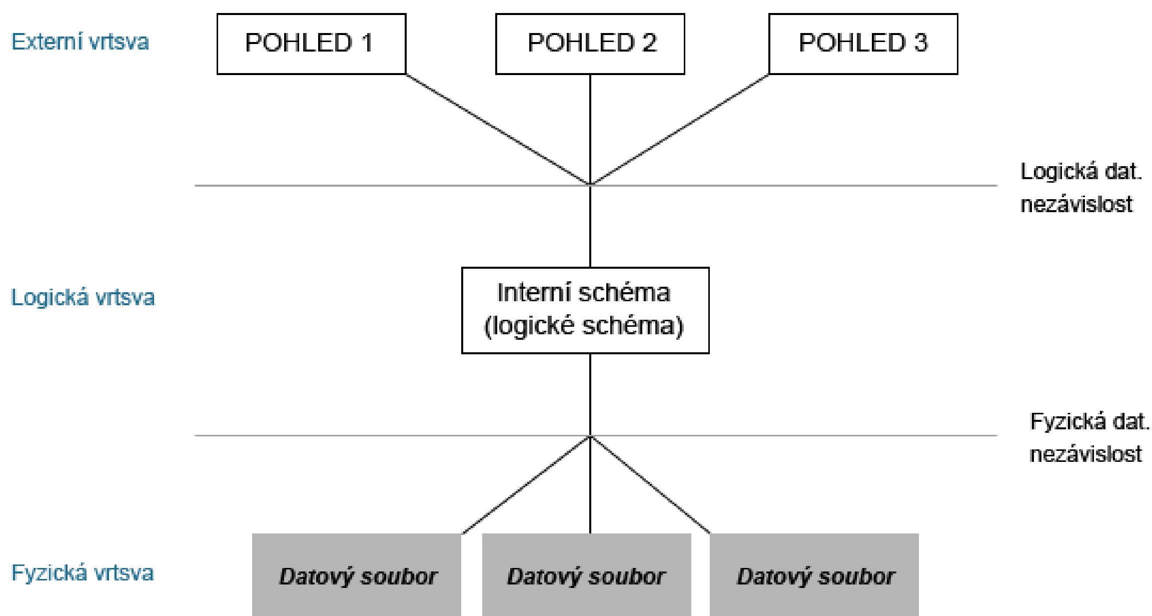
Vrstvy datové abstrakce

Databáze mají jedinečnou schopnost nabízet různým uživatelům jedněch stejných a pouze jedenkrát uložených podkladových dat různé, samostatné pohledy na tato stejná data. Těmto pohledům se říká

uživatelské pohledy. Za uživatele můžeme považovat libovolnou osobu nebo aplikaci, která se může přihlásit do databáze za účelem načítání nebo ukládání dat.

Jestliže data uložíme do tabulkového kalkulátoru, jako je MS Excel, musí všichni uživatelé pracovat s jedním společným pohledem a tento pohled se musí shodovat se způsobem fyzického uložení dat v podkladovém datovém souboru.

V databázovém systému můžeme oproti tomu každému jednotlivému uživateli nabídnout jiný pohled na stejná data, přičemž tyto pohledy mohou být každému přizpůsobené, přestože pracují nad jednou společnou uloženou kopií dat. Protože v pohledech nejsou uložena žádná skutečná data, odrážejí se v nich automaticky veškeré změny provedené v podkladových databázových objektech. To je možné díky tzv. **vrstvám abstrakce**, které zachycuje následující obrázek:



Obr: Vrstvy abstrakce

Fyzická vrstva

Fyzická vrstva obsahuje datové soubory, do nichž se ukládají veškerá data příslušné databáze. Podle konkrétního systému může být jedna databáze uložena v několika souborech, rozmístěných často na různých fyzických diskových jednotkách. *Jednou z výjimek je MS Access, pod nímž se celá databáze ukládá do jediného fyzického souboru*. Toto uspořádání ovšem znamená omezení možnosti škálování databázového systému, který se nedokáže přizpůsobit většímu počtu současně pracujících uživatelů a který tak není vhodným řešením pro velké podnikové systémy - na druhé straně se tím ale zjednodušuje práce s databází na osobním počítači jednotlivého uživatele.

Uživatel vůbec nemusí tušit, jakým způsobem jsou data v souborech rozmístěna. Databáze ve spolupráci s OS automaticky zajišťuje správu datových souborů. Serverově orientované databázové systémy jako je ORACLE, MySQL nebo MS SQL Server zajišťují správu fyzických souborů automaticky a uživatel se při práce nemusí výslovně odvolávat.

Logická vrstva

Logická vrstva představuje první ze dvou **vrstev abstrakce** v databázi. Je tomu tak proto, že fyzická vrstva skutečně existuje a je realizována konkrétními soubory, zatímco logická vrstva je pouze součástí abstraktních logických struktur, které se podle potřeby skládají z objektů fyzické vrstvy. Tato vrstva se někdy označuje pojmem schéma, jenž označuje kolekci všech datových položek uložených v příslušné databázi. Podle konkrétního DS může být schéma tvořeno množinou dvojrozměrných tabulek, hierarchickou strukturou nebo jinou strukturou. Struktury – viz modely databází.

Externí vrstva

Externí vrstva je druhou z vrstev abstrakce v databázi. Tuto vrstvu tvoří již zmíněné uživatelské pohledy, kterým se souhrně říká subschéma. **V této vrstvě se k databázi připojují uživatelé a aplikační programy, které v ní dále pracují a zadávají v ní dotazy.** Do přímého styku s fyzickou a logickou vrstvou vstupuje v ideálním případě jen databázový administrátor. DBMS pak zajišťuje transformaci vybraných položek z jedné nebo více datových struktur v logické vrstvě do konkrétního uživatelského pohledu. Protože se uživatelské pohledy vytvářejí v této externí vrstvě, mohou být předem definovány a uloženy do databáze, kde je kdokoli může znovu využít, nebo mohou být vytvořeny jen jako dočasné položky, ve kterých si DBMS ukládá výsledky jednorázového dotazu a které po použití „zahodí“.

Fyzická datová nezávislost

Možnost změny fyzické souborové struktury v databázi bez narušení činnosti stávajících uživatelů a procesů se označuje jako **fyzická datová nezávislost.** Spočívá v oddělení fyzické vrstvy od logické. Určitý stupeň fyzické datové nezávislosti je součástí všech počítačových systémů. Tabulkový sešit bude stejně dobře pracovat, když jej z HD zkopírujeme na Flash disk nebo vypálíme na CD. Rozdíly jsou poté patrné jen v rychlosti přístupu (a lidské paměti, kde je daný soubor uložen).

DBMS jsou v tomto ohledu pokročilejší. Jakýkoli uživatel databáze může pracovat s databázovými objekty i bez znalosti fyzického umístění datových souborů a bez jejich zadávání. Místa fyzického uložení těchto objektů si databázový systém hlídá sám v tzv. **katalogu**. Jaké změny můžeme provést datově nezávislým způsobem?

- Přesunutí datového souboru databáze z jednoho zařízení do jiného, nebo z jednoho adresáře do jiného.
- Rozdělení nebo sloučení datových souborů databáze.
- Přejmenování databázových souborů.
- Přemístění databázového objektu z jednoho souboru do jiného.
- Přidání nových datových souborů.

Pochopitelně jsme nezmiňovali odstraňování souborů, což by vedlo k okamžité ztrátě daných dat a nefunkčnosti databáze.

Logická datová nezávislost

Možnost provádění změn v logické vrstvě bez narušení činnosti stávajících uživatelů a procesů označujeme jako **logickou datovou nezávislost**. Důležité je uvědomit si, že většina logických změn s sebou přináší také nějakou fyzickou změnu. Do databáze např. můžeme těžko přidat nějaký nový objekt (např. tabulku v relačním DBMS), aniž bychom jeho data někam uložili: to skutečně znamená odpovídající změnu ve fyzické vrstvě. Zde jsou změny, které můžeme provést právě díky logické datové nezávislosti:

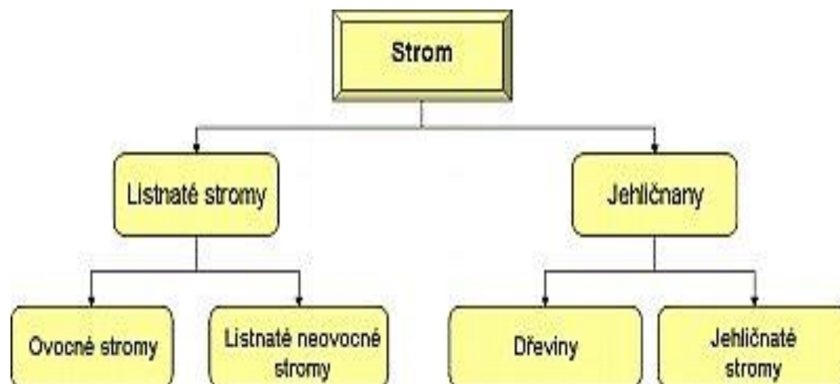
- Přidání nového databázového objektu
- Přidání datových položek ke stávajícímu objektu

Nejrozšířenější modely databází

Databázový model je v podstatě architektura, podle které DS ukládá objekty do databáze a podle které je vzájemně provazuje. V době před existencí relačního databázového modelu se obvykle používaly dva modely – *hierarchický databázový model* a *síťový databázový model*.

Hierarchický databázový model

Data jsou strukturována hierarchicky a obvykle se znázorňují v podobě obráceného stromu. Přičemž jedna z tabulek slouží jako tzv. **kořen** tohoto obráceného stromu a ostatní tabulky jako **větve** vycházející z kořene, což můžete vidět na obrázku níže ve vysvětlující analogii.



Obr: Hierarchický databázový model

Je zřejmé, že nejvyšším prvkem jsou stromy, které se dělí na 2 základní druhy a každý druh má zase své podruhy a ty zase své podruhy, až se dostanete k jednotlivým instancím. Například u ovocných stromů lze uvést jabloň, u jehličnatých stromů smrk, u dřevin tis apod.

Vztah je v hierarchické databázi reprezentován termíny *rodič* a *potomek*. V tomto typu vztahu může být tabulka rodiče přidružena k jedné, nebo více tabulkám potomků, ale tabulka potomka může být přidružena pouze k jedné tabulce rodiče. Tyto tabulky jsou zřetelně propojeny šipkami, nebo

prostorovým rozvržením záznamů v tabulce. Uživatel pak může k záznamům přistupovat ve směru hierarchie, tedy od kořenové tabulky, přičemž postupuje dále přes stromovou strukturu až ke hledaným datům (v našem příkladě jednotlivým stromům).

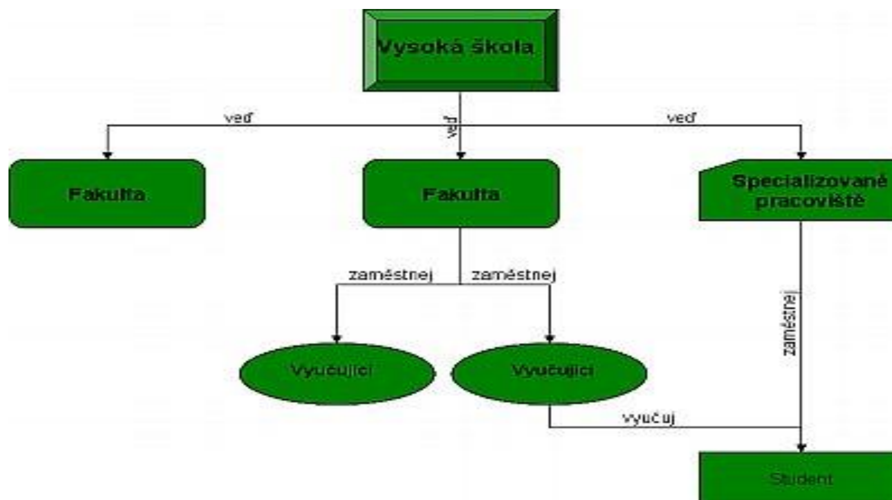
Výhodou hierarchické databáze je, že uživatel může získat data velmi rychle, protože mezi tabulkami existuje přímé propojení. Další výhodou je zabudování a **automatické prosazování referenční integrity**, což zajišťuje, že záznamy v tabulce potomka musí být napojeny na již existující záznamy v tabulce rodiče. Tím se docílí toho, že pokud odstraníte záznam v tabulce rodiče, budou smazány i propojené záznamy v tabulkách potomků.

Hierarchická databáze nepodporuje tvorbu komplexních vztahů. Můžeme se tedy setkat s **redundantními (nadbytečnými) daty**. Například v praxi se běžně vyskytují vztahy mezi jehličnatými stromy a listnatými neovocnými stromy, vytvářejí společně smíšené lesy. Toto nelze v hierarchické struktuře modelovat přímo, ale lze to například vyřešit přidáním další tabulky (tabulek), kde se však již budou vyskytovat redundantní data.

Hierarchická databáze byla hojně využívaná zejména v době ukládání dat na magnetické pásky, zejména proto, že přístup k datům byl pouze sekvenční. S příchodem magnetických médií a narůstajícím počtem redundantních dat se od používání hierarchického modelu značně upustilo.

Síťový databázový model

Síťová databáze byla vyvinuta hlavně jako pokus o vyřešení problémů hierarchické databáze. Struktura síťové databáze je vyjádřena v pojmech *uzlů* (někdy také označovaných jako záznamy) a *množinových struktur*.



Obr: Síťový databázový model

Na obrázku můžete vidět, že vysoká škola (VŠ) zaštiťuje několik fakult a specializovaných pracovišť případně ústavů. Každá fakulta má libovolný počet zaměstnanců, kteří učí žáky dané fakulty. Pod VŠ, ale také patří specializovaná pracoviště, či ústavy, kde se provádí výzkumy, kde jsou mimo zaměstnanců i studenti VŠ.

Uzel reprezentuje soubor záznamů a množinová struktura reprezentuje a zřizuje vztah v síťové databázi. Je to snadno pochopitelná konstrukce, která vytváří vztah mezi dvěma uzly tak, že jeden uzel je definován jako **vlastník** a druhý jako **prvek** (tato metoda je značným vylepšením vztahu rodič/potomek). Množinová struktura podporuje vztah 1:N, neboli jeden záznam v uzlu vlastník může být v relaci k jednomu, nebo více záznamům v uzlu člen. Na druhé straně, jeden záznam v uzlu člen je ve vztahu pouze k jednomu záznamu typu vlastník. Záznam v uzlu typu člen navíc nemůže existovat, aniž by byl ve vztahu k nějakému záznamu v odpovídajícím uzlu typu vlastník.

Mezi uzly může být definována jedna, nebo více spojení (množin) a libovolný počet může být součástí dalších množin s jinými uzly v databázi. Například uzel *učitel* je ve vztahu k uzlu *student* prostřednictvím množinové struktury *vyučuj*. *Specializované pracoviště* je také ve vztahu k uzlu studenti, ale prostřednictvím množinové struktury *zaměstnávej*.

Uživatel má možnost k datům v síťové databázi přistupovat pomocí procházení odpovídajících množinových struktur. Na rozdíl od hierarchické databáze, ve které musí k datům přistupovat z kořenové tabulky, může uživatel v síťové databázi začít přistupovat k datům z libovolného uzlu a procházet přidruženými množinami. Vztaženo k příkladu - chcete zjistit, jakou vysokou školu studuje daný student. Začnete od uzlu *student*, až se dostanete k uzlu *Vysoká škola*.

Výhodou síťové databáze je rychlý přístup k datům. Umožňuje uživatelům vytvářet dotazy, které jsou mnohem komplexnější než dotazy v hierarchickém modelu. Hlavní nevýhodou síťové databáze je, že uživatel musí znát strukturu databáze, aby mohl pracovat s množinovými strukturami. Další nevýhodou je, že je velmi nesnadné změnit strukturu databáze, aniž by bylo nutno změnit aplikace, které s ní pracují.

Relační databázový model

Relační model je v současnosti bezesporu nejpoužívanějším modelem při správě databází. Historie relačních databází je vlastně úzce spjatá s historií samotných počítačů. Ovšem přelomu a základního návrhu dosáhnul až **Dr. Edgar F. Codd**. Tento zaměstnanec IBM navrhl implementaci relačního datového modelu. Jednou z mnoha změn, díky které se odlišoval (model) od svých předchůdců bylo použití srozumitelných příkazů, které vycházejí z běžné angličtiny. Tím se stal tímto model přijatelnějším pro budoucí potencionální netechnicky vzdělané uživatele.

Vývoj kódu pro relační databáze probíhal jak u IBM, tak i na Kalifornské univerzitě v Berkley. V IBM vycházeli z původního kódu pro hierarchickou strukturu, System-R SEQUEL (Structured English Query Language). V roce 1976 IBM představilo jazyk SEQUEL2, jehož název byl později změněn na nám již známou zkratku **SQL**. **Poté byl tento jazyk uznán jako standard.**

Další modely databází

Kromě tří výše zmíněných modelů se dnes využívají také **objektově orientované** a **objektově relační** modely databází. Jejich podstata vychází z objektového principu přistupování k datům. Pro zájemce:

Objektově orientované databázové systémy

Díky rozvoji **objektově orientovaného programování** (OOP) začaly vznikat i objektově orientované databázové systémy. Prakticky vznikly následkem pokusu odstranit chyby relačních systémů. Základem takto objektově orientovaných databází není tabulka, jako u relačních, ale data jsou sdružována to tzv. objektů (ty korespondují s objekty reálného světa, tak jako u OOP). Tyto objekty

jsou pak uspořádány do objektové struktury dat. Komunikace mezi objekty a zbytkem systému je realizována pomocí zpráv.

Každý objekt má svou identitu (tím je míněno, že je jednoznačně systémově identifikovatelný). Je charakterizován množinou vlastností objektu, množinou zpráv, na které objekt reaguje, množinou metod, z nichž každá je tvořena částí programového kódu, který je implementací reakce na zprávu. Objekty, které obsahují stejné typy hodnot a metody jsou sdružovány do tříd (typ objektu).

Velikou výhodou je, že definice dat se mohou dědit (a to i mnohonásobně), objekty všelijak propojovat, podporovat zapouzřenost a polymorfismus objektů.

Práci s databází lze rozdělit do dvou částí: těmi jsou **tvorba databáze** (definice objektů) a **pokládání dotazů** (pomocí příkazů SELECT). Pokládání dotazů v OO databázi je mnohem jednodušší než v relačních databázích, bohužel však na druhou stranu je mnohem složitější tvorba celé objektové struktury.

Objektově orientované databáze (založené na perzistentních jazycích)

Takové databáze umí manipulovat s daty, která jsou „trvalá“ (perzistentní). Nepracuje přímo s SQL, ale lze jej vnořit do jazyka, mezi ostatní příkazy. S perzistentním jazykem pak sdílí datové typy. Výhodou je, že programátor nemusí znát jazyk SQL (pokud jej samozřejmě nevnoří) a vysoký výkon. Objektový přístup je zabudován už přímo do jazyka. Avšak velikou nevýhodou je velmi malá možnost tvorby dotazů běžným uživatelem, perzistentní jazyky jsou totiž jazyky procedurální.

Objektově relační databáze

Jedná se vlastně o nadstavbu relačních databází, kdy se vychází z relační struktury databáze. Dotazovací jazyky (např. SQL) jsou doplněny o další příkazy a funkce, což umožňuje vznik tzv. složených typů (opakují se atributy, skupinové atributy, ...). Vytvářejí se sice složitější datové typy, ale díky použití „silných“ dotazovacích jazyků je dosaženo vysoké úrovně ochrany dat.

Shrnutí

Tento studijní text je ve zbytku plně věnován **relačním databázím**. Důvodem je nejen to, že relační model je ze všech nejrozšířenější, ale také několik dalších níže uvedených vlastností:

- Definice, údržba i manipulace s datovými strukturami je velmi snadná.
- Data je možné načítat prostřednictvím jednoduchých dotazů.
- Data jsou v systému dobře chráněna.
- Konkrétní produkty je možné vybírat od mnoha různých výrobců.
- Převody mezi různými implementacemi od různých výrobců jsou poměrně snadné.
- Dnešní relační databáze jsou stabilní a stále aktuální.

3 Relační databáze

Tabulka **Produkty**

Kod_zasob	Popis	Cena
A416	Hřebíky, krabice	10 Kč
C923	Připínáčky, krabice	8 Kč

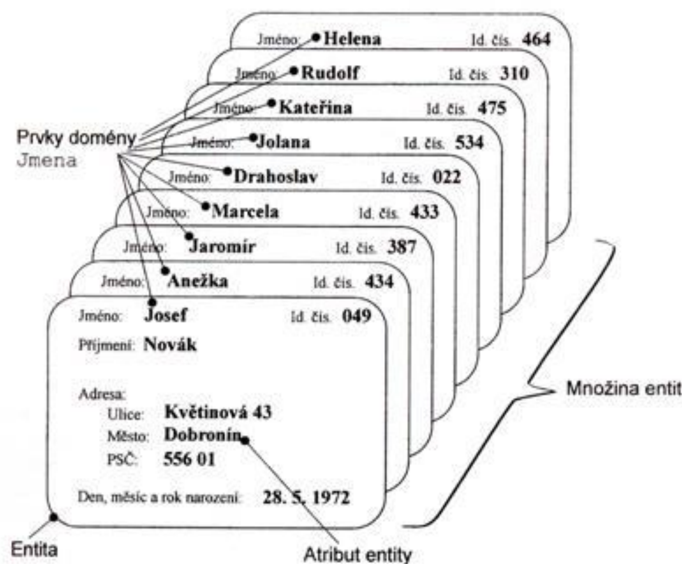
Tabulka **Faktury**

Kod_faktury	Kod_zasob	Mnozstvi
3804	A416	15
3804	C923	20

Databázová terminologie

Podívejme se nejprve na dvě předchozí tabulky a jejich uspořádání:

- Každá tabulka se skládá z několika **řádků** a **sloupců**.
- Každý řádek obsahuje data o jediné **entitě** (jako je produkt nebo jedna objednávka). Označuje se za **záznam.** Například první řádek v tabulce Produkty je záznamem: popisuje produkt A416, což je krabice hřebíků s cenou 10 Kč. Termíny řádek a záznam lze vzájemně zaměňovat.
- Každý sloupec obsahuje jednu část dat související s daným záznamem, která se nazývá **atribut.** Příklady atributů jsou množství prodaných položek nebo cena produktu. Atribut se při popisu databázové tabulky označuje jako **pole.** Termíny atribut a pole lze opět zaměňovat.



Obr: Množina entit

Vztahy neboli relace

Relace je velmi důležitým pojmem v relačních databázích. Je to způsob propojení jednotlivých tabulek tak, aby spolu mohly komunikovat a aby jejich propojení umožňovalo svázat vzájemně související data.

Při návrhu databáze je totiž běžné, že jsou data rozdělena do jednotlivých tabulek, které jsou propojeny pomocí vybraných údajů ve vybraných sloupcích. **Důležité je zejména to propojení tabulek**. Tím jsou totiž označovány relace. Sloupec, podle kterého se tabulky propojují, je označen jako *primární klíč*. Primární klíč je atribut, jehož hodnota je pro každý záznam jedinečná. **V relačních databázích existují 3 typy relací:**

Relace typu 1 : 1

Význam relace 1:1 je takový, že právě jednomu záznamu v jedné tabulce odpovídá právě jeden záznam v tabulce druhé. Jednotlivé záznamy v obou tabulkách jsou spojeny přímo. Tento typ relace se málo kdy používá, protože takto spojené údaje lze vlastně umístit přímo do jedné tabulky. Relace samotná pak slouží zejména pro lepší přehlednost.

Relace 1:1 má význam zejména u velmi rozsáhlých tabulek s mnoha sloupci. V takovém případě slouží druhá propojená tabulka jako „odlehčení“ té první, zejména v případě že se s hodnotami v druhé tabulce příliš často nepracuje.

Následující příklad ukazuje typickou ukázku relace 1:1.

Tabulka **zamestnanci**:

id	jmeno	prijmeni	telefon
1	František	Louka	607 878 XXX
2	Lucie	Smetanová	607 878 XXX
3	Markéta	Palacká	607 878 XXX

Tabulka **nahrady**:

id_zamestnance	plat_hod	procento_odmeny
1	250	5.0
2	250	5.0
3	350	7.5

Pole (Atribut) **id** v tabulce *zaměstnanci* a **id_zamestnance** v tabulce *nahrady* jsou primárními klíči v těchto tabulkách. Pro každý záznam v jedné tabulce existuje pouze jeden odpovídající záznam v tabulce druhé.

Relace typu 1 : N

Relace 1: N je nejpoužívanějším typem. Vyjadřuje, že právě jednomu záznamu v první tabulce odpovídá více záznamů v tabulce druhé. Důležité je zvolit si primární klíč v tabulce, která bude tvořit relaci 1, na sloupec který bude propojen s druhou tabulkou (N).

Tabulka **fotogalerie**:

id_fotogalerie	nazev	popis	datum
1	Brouci	...	xx.x.xxxx
2	Motýli	...	xx.x.xxxx
3	Včely	...	xx.x.xxxx

Tabulka **fotky**:

id_fotogalerie	foto
1	foto1.jpg
1	foto2.jpg
2	foto1.jpg

Jednomu záznamu v tabulce *fotogalerie* přísluší více záznamů (N) v tabulce *fotky*. **Primárním klíčem v tabulce *fotky* je kombinace obou atributů (složený primární klíč).**

Relace typu M : N (někdy také označována jako N : M)

Relace tohoto typu definuje, že několika záznamům v první tabulce odpovídá několik záznamů v tabulce druhé. V případě používání takové relace, je pro její vytvoření nutná tzv. dekompozice vztahů neboli jejich rozdělení vytvořením tzv. **vazební (spojovací) tabulky**. Vytvoří se pomocná tabulka vzniklá spojením dvou primárních klíčů spojovaných tabulek. Ta je pak definována relací typu 1:N.

Také ale může nastat situace, kdy v tabulkách nejsou vzájemně související údaje, a proto mezi nimi není definován žádný vztah.

Tabulka **studenti**:

id	jmeno	prijmeni	rok_narozeni
1	Jan	Střihavka	1988
2	Andrea	Martinová	1990
3	Matěj	Spáčil	1989

Vazební tabulka:

id_student	id_jazyk
1	1
1	2
2	2

Tabulka **jazyky**:

id	jazyk
1	Anglický jazyk
2	Německý jazyk
3	Španělský jazyk

Více záznamům z tabulky ***studenti*** odpovídá více záznamů z tabulky ***jazyky*** a naopak. Příslušné spojení realizuje vazební tabulka, která má složený primární klíč z primárních klíčů spojovaných tabulek.

Získávání dat

Data jsou z relační databáze získávána pomocí **dotazovacího jazyka SQL** (Structured Query Language – strukturovaný dotazovací jazyk). SQL je standardní jazyk používaný k vytvoření, modifikaci a udržování databáze, a vytváření dotazů.

Tři základní části SQL dotazu jsou příkaz SELECT...FROM, WHERE a klauzule ORDER BY. Klauzule SELECT se používá k specifikování položek, které se mají v dotazu použít, klauzule FROM určuje tabulky, do kterých tato pole patří. Záznamy vrácené dotazem se dají dále filtrovat podle jednoho, nebo více polí v klauzuli WHERE, a poté setřídit vzestupně či sestupně pomocí klauzule ORDER BY.

Většina současných velkých relačních databází obsahuje nějakou implementaci jazyka SQL.

Systém pro správu relační databáze RDBMS

Systém pro správu relační databáze (**RDBMS** – Relational Database Management System) je program, který se používá k vytvoření, správě a modifikaci relačních databází. Některé RDBMS mimo tohoto také zajišťují nástroje pro vytváření aplikací pro koncové uživatele, kteří pracují s daty uloženými v databázi (např. ORACLE). Je důležité si při výběru RDBMS dávat pozor, jaký druh relačního databázového modelu podporuje a zdali má plnou implementaci databázového modelu.

V počátcích svého zrodu (od r. 1970) byly RDBMS používány na sálových počítačích, přičemž mezi dva nejčastěji používané systémy byly System-R uvedený na trh společností IBM, a Interactive Graphics Retrieval System (INGRES) vyvinutý na Kalifornské univerzitě v Berkley. Poté co uživatelé objevili přínosy relačních databází, došlo k jejich velkému rozšíření a k vývoji nových komerčních RDBMS. V 80. letech spatřily světlo světa systémy, jako jsou **Oracle** (Oracle Corporation) nebo DB2 (IBM), ovšem stále jen pro sálové počítače.

S příchodem osobních počítačů se zrodily RDBMS určené i pro běžná PC. Některé počáteční produkty v této kategorii, jako například dBase (Aston-Tate) a Fox (Fox Software), byly pouze jednoduché systémy pro správu databáze založené na souborech. RDBMS jaké známe dnes, se začaly objevovat až po uvedení R:BASE (Microrim) či Paradox (Ansa Software) na trh. Oba tyto systémy přinesly revoluční myšlenku, která pomohla přenést správu databází ze sálových počítačů na běžná PC.

Ovšem tím stále vývoj nekončil, jelikož vyvstala potřeba tyto databáze nějakým způsobem sdílet. Vývojáři přišli s myšlenkou centrálně sdílené databáze, která následně dostala pojmenování **RDBMS Klient/Server**. Server (nebo také databázový server) zde označuje právě onen centrálně sdílený prvek, ke kterému mají přístup všichni klienti (databázový klienti) v dané síti. Vývojáři zajistili implementaci integrity a bezpečnosti dat na databázovém serveru, čímž umožnili založení množství různých uživatelských aplikací na jedné množině dat, aniž by byla ohrožena bezpečnost nebo integrita dat. Tyto typy databází se využívají zejména pro správu velkého objemu sdílených dat. Mezi nejnovější produkty v této oblasti patří zejména Microsoft SQL Server 2019 (Microsoft Corporation) nebo Oracle Database 19c (Oracle Corporation).

4 Logický a fyzický návrh databáze

Tabulky

Po vlastní databázi je primární jednotkou ukládání dat v relačním modelu **tabulka**. **Každý řádek reprezentuje jeden výskyt entity**, kterou tabulka modeluje, **a každý sloupec reprezentuje jeden atribut této entity**. Proces optimalizace entit z konceptuálního návrhu do tabulek v logickém návrhu nazýváme *normalizací*. Entita v konceptuálním návrhu se často promítá do jedné (obsáhlé) tabulky, což v logickém návrhu nemusí platit. Z důvodů, které si vysvětlíme později, se entity často rozdělují do několika tabulek.

Pamatujte si, že relační tabulka je jen logickou záznamovou strukturou a že obvykle ve fyzické vrstvě neexistuje ve skutečné tabulkové formě. Data jsou rozděleny do souborů OS z fyzické vrstvy, kterým se ve většině RDBMS říká **tabulkový prostor**.

Každá tabulka musí v okamžiku svého vytvoření dostat jednoznačný (jedinečný) název. Min. a max. počet znaků v názvu se u jednotlivých RDBMS liší (v MySQL je to max. 65 znaků a to jak pro databáze, tabulky a sloupce). Název tabulky by měl být výstižný a měl by označovat entitu reálného světa, kterou tabulka reprezentuje. **V názvech tabulek je nejvhodnější vyvarovat se mezer a diakritiky.**

Sloupce a datové typy

Sloupec (atribut, pole) je v relační databázi nejmenší pojmenovanou jednotkou dat, na kterou se můžeme odkazovat. Každému sloupci musíme přiřadit jedinečný název (v rámci jedné tabulky) a **datový typ**. Datový typ je přitom jakási kategorie formátu konkrétního sloupce. Definice datového typu má několik výhod:

- Omezuje množinu dat ve sloupci jen na takové znaky, které mají pro daný datový typ smysl (např. jen číslice nebo jen platné kalendářní datum).
- Cílovému uživateli databáze předkládá jistou množinu dat. Pokud např. odečteme číslo od jiného čísla, dostaneme ve výsledku opět číslo.
- Napomáhá relačnímu databázovému systému v efektivním ukládání dat sloupce. Čísla se např. dají ukládat v interním, prostorově úsporném číselném formátu, a nikoli jen v neefektivním formátu řetězce znaků.

Každé pole ve správně navržené databázi obsahuje právě jednu hodnotu a jeho jméno identifikuje typ údaje, který je v něm uložen. Ve špatně navržené databázi obvykle najdeme následující 3 typy polí:

1. **Vícesložkové pole**, které ve své hodnotě obsahuje dvě, nebo více součástí.
2. **Vícehodnotové pole**, jež obsahuje několik hodnot toho samého typu.
3. **Vypočítané pole**, které obsahuje spojené textové řetězce, nebo výsledky matematických operací.

Pro příklad uvedeme nevhodně navrženou tabulku **zakaznici:**

	Vypočítané pole	*Vícesložkové pole*	*Vícehodnotové pole*
id	**cele_jmeno**	**mesto_stat_psc**	**poradce**
1	Radim Motyka	Olomouc, ČR, 771 40	Novák, Potměšil
2	Martina Šviehlová	Nové Zámky, SR, 587 44	Beňo

Klíče

Klíče jsou speciálními poli, které mají v tabulce velmi významnou roli. Typ klíče určuje jeho význam v tabulce. Existuje několik typů klíčů, ale nejvýznamnější jsou **primární klíč** a **cizí klíč**.

Primární klíč je pole, nebo i skupina polí, které jednoznačně identifikuje každý záznam v tabulce. Je-li primární klíč složen ze dvou, nebo více polí, tak se označuje jako **složený primární klíč**. Každá tabulka by měla mít primární klíč.

Primární klíč

id	jmeno	prijmeni	rok_narozeni
1	Jan	Střihavka	1988
2	Andrea	Martinová	1990
3	Matěj	Spáčil	1989

Primární klíč	*Cizí klíč*		
id	**id_student**	**nazev**	**mistnost**
10	1	Anglický jazyk	P6
11	1	Německý jazyk	P43
12	2	Španělský jazyk	P22

Pole **id** jednoznačně identifikuje každého studenta, zabraňuje duplicitě a je tedy Primárním klíčem tabulky *studenti*. Může se také použít pro zřízení vztahu mezi tabulkou *studenti* a tabulkou *skupiny*, jak je uvedeno v příkladu.

Vztah (1:N) je realizován existencí primárního klíče tabulky *studenti* (id) v tabulce *skupiny*, kde je reprezentován sloupcem id_student a představuje v ní cizí klíč.

Vlastnosti primárního klíče

- Nesmí se jednat o vícesložkové pole.
- Musí obsahovat jedinečné hodnoty.
- Nesmí obsahovat hodnotu NULL.
- Skládá se z nejmenšího možného počtu polí, která zajišťují jedinečnost.
- Hodnota primárního klíče musí jednoznačně identifikovat hodnoty všech polí daného záznamu.
- Jeho hodnota se mění jen ve velmi výjimečných případech.

Indexy

Index je struktura, kterou poskytují RDBMS, aby zlepšily zpracování dat. Konkrétní RDBMS stanoví, jak má index pracovat a jak jej budeme používat. Index se stanovuje pro konkrétní sloupce tabulky a může plnit různé úlohy. Více se na téma indexů budeme bavit v navazující části věnované MySQL.

Hodnota NULL

NULL reprezentuje chybějící či neznámou hodnotu. Je potřeba hned od začátku vědět, že NULL nereprezentuje nulu, ani prázdný textový řetězec nebo textový řetězec „null“. Je to hodnota reprezentující doslova nic. Důvod její existence je prostý:

1. Nula může mít širokou škálu významů. Může reprezentovat stav účetní rovnováhy, aktuální počet místenek nebo určitou cenu.
2. Textový řetězec tvořen jednou nebo více mezerami je pro nás bezvýznamný, nicméně v databázi a příslušných dotazech zaujímá své místo. Řetězec nulové délky v daném poli může např. znamenat, že dodací adresa je shodná s adresou bydliště (tento stav ale lze reprezentovat hodnotou NULL).

Hodnota NULL je také důležitá pro šetření místa. Pokud existuje sloupec, který může obsahovat skutečné hodnoty jen sporadicky (druhé křestní jméno), je lepší neukládat do záznamu nic, než např. prázdný textový řetězec. Pokud to tedy lze, volte u sloupců možnost NULL.

Naopak v případě sloupce, který by nikdy neměl zůstat nevyplněn, volte volbu **NOT NULL**. V případě příjmení klienta bychom neradi očekávali hodnotu NULL.

Integrita dat

Integrita dat označuje platnost, konzistentnost a přesnost dat v databázi. Integrita dat je jedním z nejdůležitějších aspektů procesu logického návrhu databáze a nesmí se podceňovat, přehlížet nebo dokonce opomíjet.

Existují 4 typy integrity, které se implementují v průběhu procesu návrhu databáze. Tři z nich jsou založeny na různých aspektech struktury databáze a jsou označeny podle oblasti, kde se objevují. Poslední typ je založen na způsobu jakým organizace chápe a užívá svá data.

1. **Integrita na úrovni tabulky** – označovaná jako **integrita entity**, zajišťuje, že neexistují duplicitní záznamy a že pole, které identifikuje každý záznam v tabulce je jedinečné a nikdy neobsahuje NULL.
2. **Integrita na úrovni pole** – označovaná jako **doménová integrita**, zajišťuje, že struktura každého pole je spolehlivá – že hodnoty v každém poli jsou platné, konzistentní a přesné a že pole stejného typu jsou v celé databázi definovány shodně.
3. **Integrita na úrovni vztahů** – označovaná jako **referenční integrita**, zajišťuje, že vztah mezi dvěma tabulkami je spolehlivý a že záznamy v obou tabulkách jsou synchronizovány kdykoli jsou data v kterékoli z tabulek zadávána, opravována nebo mazána.

4. **Business pravidla** – zavádějí omezení na jisté aspekty databáze založené na způsobu jakým organizace získává a používá data. Tato omezení mohou ovlivnit některé aspekty návrhu databáze, jako rozsah a typ hodnot uložených v poli, typ účasti a stupeň účasti každé tabulky ve vztahu, či typ synchronizace používaný pro integritu na úrovni vztahů v některých vztazích.

5 Normalizace databází

Normalizace databází je technika, která nám pomůže vyhnout se datovým anomáliím a dalším problémům se správou dat. **Existují tři druhy datových anomálií:**

Anomálie při vkládání

Tato situace může nastat v případě, kdy jsou chybně v jedné tabulce směšovány atributy dvou různých entit. Např. v tabulce (fakulta, datum založení, kurz) můžeme zaznamenat jen data pro fakulty, které mají kurzy.

Anomálie při odstraňování

Anomálie při odstraňování je přesným protikladem té předchozí. Označujeme tak situaci, kdy odstranění dat o jedné konkrétní entitě způsobí nežádoucí ztrátu dat, která charakterizuje jinou entitu. Tak např. odstraněním jediného kurzu na fakultě X, bychom ztratili také všechny data tuto fakultu reprezentující (fakulta, datum založení).

Anomálie při aktualizaci

Anomálie při aktualizaci znamená, že při aktualizaci jediné datové hodnoty je potřeba aktualizovat současně několik řádků dat. Tak například na fakultě X by probíhalo celkem *n* kurzů a tomu by odpovídali příslušných *n* řádků v dané tabulce. V případě změny názvu fakulty X na fakultu Y by se tento název musel změnit ve všech odpovídajících záznamech.

Při normalizaci se postupuje transformováním tabulky různými fázemi:

1. První normalizovanou formou.
2. Druhou normalizovanou formou.
3. Třetí normalizovanou formou.
4. Boyce Coddovou normalizovanou formou.
5. Čtvrtou normalizovanou formou.
6. Pátou normalizovanou formou.

Normalizace se zaměřuje na:

- Eliminaci nadbytečnosti dat.
- Usnadnění změny dat a tedy zabránění vzniku anomálií při této činnosti.
- Ulehčení zavádění omezení referenční integrity.
- Vytváření snadno pochopitelné struktury, která se úzce podobá situaci reprezentované daty a umožňuje další růst.

První normalizovaná forma

Tabulky v první normalizované formě naplňují tato pravidla:

- Jsou eliminovány sloupce se stejným obsahem.
- Jsou definovány všechny klíčové atributy.
- Všechny atributy závisejí na primárním klíči.

To znamená, že data musejí být schopna převedení do tabulkového formátu, kde každé pole obsahuje jednu hodnotu. Jedná se rovněž o fázi, kde se definuje primární klíč.

Poznámka

Třebaže není vždy chápán jako součást definice první normalizované formy, v této fázi se obvykle aplikuje princi **atomičnosti.** To znamená, že všechny sloupce musejí obsahovat své nejmenší součásti, neboli musejí být nedělitelné. Obvyklým příkladem tohoto principu je situace, kdy někdo vytvoří pole jmeno (např. Petra Nováková) a nikoli pole jmeno (Petra) a prijmeni (Nováková).

Mějme tabulku **OSOBY**:

Jmeno	Prijmeni	Bydliste	Telefony
Petr	Bříza	Praha	125789654; 602589875
Roman	Brtník	Olomouc	369852147; 357951456
Alena	Nová	Brno	546789123; 123456789

S takovouto tabulkou by byla spousta problémů, například by se dost špatně prováděly změny čísel, případně vyhledávání podle telefonního čísla.

Aby tabulka byla v **1NF** musíme buďto rozdělit atribut telefon do více atributů - nové sloupce Telefon 1 a Telefon 2 (a to pouze za předpokladu, že jsme si jisti, že se množství telefonních čísel nezvýší!!! – špatné řešení odporující 1. NF), nebo oddělit telefonní čísla do samostatné tabulky – **správné řešení**:

Nová tabulka **OSOBY**:

ID	Jmeno	Prijmeni	Bydliste
1	Petr	Bříza	Praha
2	Roman	Brtník	Olomouc
3	Alena	Nová	Brno

Nová tabulka **TELEFONY**:

ID_osoby	Telefon
1	125789654
1	601258987
2	369852147

2	357951456
3	546789123
3	123456789

Aplikací první normalizované formy nám v tomto případě vznikly dvě nové tabulky, které jsou v relaci 1:N.

Druhá normalizovaná forma

Tabulka je v druhé normalizované formě, pokud naplňuje tato pravidla:

- Nachází se v první normalizované formě.
- Nezahrnuje žádné částečné závislosti (kdy nějaký atribut závisí jen na části primárního klíče) resp. každý neklíčový atribut[1] je plně závislý na primárním klíči. *Prakticky tedy platí, že pokud je tabulka v 1. Norm. formě a nemá složený primární klíč, tak je automaticky také v 2. Norm. formě.*
- Kdykoli se opakují identické obsahy některých sloupců v různých záznamech, tabulka by měla být rozdělena. *Důsledek předchozího.*

Mějme tabulku **SKLAD**:

Nazev	Vyrobce	Telefon	Cena	Mnozstvi
Čokoláda	Milka	+420123456789	30Kč	1200
Oplatky	Orion	+420987654321	20 Kč	1500
Bonbóny	Milka	+420123456789	18 Kč	1700

Je nám na první pohled jasné, že zde jaksi schází jasně definovaný primární klíč. Nicméně jeho úlohu sehrává kombinace atributů (složený primární klíč) **Název** a **Výrobce**. (Čokoláda x Milka, nebo nový možný řádek Čokoláda x Orion).

Ovšem atribut **Telefon** je závislý jen na části primárního klíče, tj. závislý na **Výrobce**. Všimněte si, že část primárního klíče v poli **Nazev** v řádku 1 a 3 se různí, ale **Telefon** je shodný. Tj. existuje částečná závislost. Tudíž tabulka se nenachází v druhé normalizované formě a je potřeba s ní dále pracovat.

Při stávajícím řešení by docházelo k anomáliím! Odstraněním všech výrobků od výrobce Milka by došlo i ke ztrátě příslušného tel. čísla výrobce. Taktéž při změně tel. čísla výrobce by se musely upravit všechny odpovídající záznamy. Řešením je opět rozklad na dvě tabulky:

[1] Neklíčový atribut je takový atribut, který neplní úlohu primárního ani cizího klíče.

Nová tabulka **ZBOZI**:

Nazev	ID_Vyrobce	Cena	Mnozstvi
Čokoláda	1	30Kč	1200
Oplatky	2	20 Kč	1500
Bonbóny	1	18 Kč	1700

Nová tabulka **VYROBCE**:

ID	Vyrobce	Telefon
1	Milka	+420123456789
2	Orion	+420987654321

Tabulka **VYROBCE** má nyní jasný primární klíč, pole **ID**. Tabulka **ZBOZI** má *složený primární klíč* z polí **Nazev** a **ID_Vyrobce**. Přičemž pole **ID_Vyrobce** je zároveň cizím klíčem. Mezi tabulkami existuje relace 1:N.

Třetí normalizovaná forma

Tabulka je v třetí normalizované formě, pokud naplňuje tato pravidla:

- Je ve druhé normalizované formě.
- **Neobsahuje žádné tranzitivní neboli přechodné závislosti** (tj. když nějaký neklíčový atribut závisí na primárním klíči prostřednictvím jiného neklíčového atributu). Jiné vyjádření téhož říká, že všechny neklíčové atributy jsou navzájem nezávislé, ostatní musíme odstranit (přesunout do samostatné tabulky).

Vše nejlépe pochopíme na následující ukázce. Mějme tabulku **ZAMESTNANCI**:

ID	Jmeno	Prijmeni	Funkce	Mesto	PSC
1	Jan	Malina	Grafik	Olomouc	772 00
2	Jiří	Novák	Programátor	Prostějov	654 00
3	Andrea	SIkorová	CEO	Olomouc	772 00
4	Petra	Nováková	Účetní	Prostějov	654 00

V této tabulce je patrné, že neklíčový atribut **PSC** je závislý na primárním klíči prostřednictvím jiného neklíčového atributu a to atributu **Mesto**. Toto způsobuje existenci datových anomálií a je potřeba opět přistoupit k rozdělení tabulek a rozpad na více relací.

Nová tabulka **ZAMESTNANCI**:

ID	Jmeno	Prijmeni	ID_Mesto	Funkce
1	Jan	Malina	1	Grafik
2	Jiří	Novák	2	Programátor

3	Andrea	SIkorová	1	CEO
4	Petra	Nováková	2	Účetní

Nová tabulka **MESTO**:

ID	Mesto	PSC
1	Olomouc	772 00
2	Prostějov	654 00

Poznámka

Pokud nějaká tabulka obsahuje jen jeden neklíčový atribut, je samozřejmě nemožné, aby byl tento závislý na jiném neklíčovém atributu.

Třetí normalizovaná forma je obvykle dostačující pro většinu tabulek, protože vylučuje nejčastější typy datových anomálií. Další normalizační formy jsou pro obvyklé aplikace použitelné jen zřídkakdy, ovšem jejich základy si vysvětlíme.

Boyce-Coddova normalizovaná forma

Platí, že tabulka je v BC normalizované formě, když naplňuje tyto podmínky:

- Nachází se v třetí normalizované formě.
- Každý determinant je kandidátním klíčem.

Determinant - atribut, který určuje hodnotu jiného atributu.

Kandidátní klíč - je buď klíč, nebo alternativní klíč (jinými slovy, daný atribut může být klíčem pro danou tabulku).

Mějme tabulku **STUDIUM**:

Student	Kurz	Instruktor
Jitka Nosková	Freeware	Josef Minarčík
Markéta Hrubá	Freeware	Jiří Dostál
Ondřej Dvořák	Tvorba WWW	Milan Klement
Ondřej Dvořák	Freeware	Josef Minarčík

Tato tabulka nemá na první pohled jasně definovaný primární klíč. Tedy bude se muset použít určité kombinace. Např. kombinace **Student x Kurz**, která jasně určí Instruktora, nebo **Student x Instruktor**, protože ta vám pomůže určit Kurz. Určeme do role klíče dvojici Student x Kurz.

V jaké normalizované formě se tedy tato tabulka nachází? Je v první normalizované formě, protože má klíč a nevyskytují se v ní opakující se skupiny. Je také v druhé normalizované formě, jelikož

instruktor závisí na obou zbývajících polích (studenti mají více kurzů a tedy i instruktorů). Je rovněž v třetí normalizované formě, protože se vyskytuje jen jeden neklíčový atribut (Instruktor).

Přesto tu však nastávají některé datové anomálie. V tomto případě jeden klíčový atribut závisí na neklíčovém atributu. Tj. Instruktor (determinant) určuje Kurz. Ale podle BC normalizované formy by tak měl být determinant Instruktor také kandidátním klíčem – což NENÍ, protože sám nedokáže jednoznačně identifikovat záznam. Tabulka tak není v BC normalizované formě.

Řešení spočívá opět v rozdělení tabulky.

Nová tabulka **STUDENT_INSTRUKTOR:**

Student	Instruktor
Jitka Nosková	Josef Minarčík
Markéta Hrubá	Jiří Dostál
Ondřej Dvořák	Milan Klement
Ondřej Dvořák	Josef Minarčík

Po odstranění pole Kurz musí primární klíč zahrnovat obě zbývající pole, aby byly záznamy jednoznačně identifikované.

Nová tabulka **INSTRUKTOR_KURZ:**

Instruktor	Kurz
Josef Minarčík	Freeware
Jiří Dostál	Freeware
Milan Klement	Tvorba WWW

V této tabulce je primárním klíčem Instruktor, který jasně označuje příslušný kurz.

Čtvrtá normalizovaná forma

Tabulka je ve čtvrté normalizované formě, když naplňuje tato kritéria:

- Je v BC normalizované formě.
- Neobsahuje více než jednu vícehodnotovou závislost.

Vícehodnotová závislost existuje mezi dvěma atributy, když každé hodnotě prvního atributu odpovídá jedna nebo více přiřazených hodnot druhého atributu.

Takový případ by mohl nastat např., pokud by jeden student měl více instruktorů na jediný kurz. Nebo jiný případ pro níže uvedenou tabulku **STUDIUM**. Možná tabulka by vypadala následovně:

Mějme tabulku **STUDIUM**:

Student	Schopnosti	Jazyky
Jitka Nosková	Ovládání PC	Angličtina
Jitka Nosková	Management	Španělština
Ondřej Dvořák	Ovládání PC	Angličtina
Ondřej Dvořák	Ovládání PC	Francouzština
Ondřej Dvořák	Management	Angličtina

Pro hodnotu Studenta tu máme více schopností (to je jedna vícehodnotová závislost). Následně pro hodnotu Studenta tu máme jednu nebo více přiřazených hodnot jazyků (to je druhá vícehodnotová závislost).

Data by se dala různě interpretovat, jako např. že Jitka Nosková Ovládá PC jen v angličtině, či hovoří španělsky jen v sekci Managementu.

Možným klíčem v této tabulce je kombinace všech tří atributů (Student x Schopnosti x Instruktor). Žádná kombinace z pouhých dvou atributů nemůže záznam jednoznačně identifikovat.

Skutečnost že Ondřej Dvořák má schopnosti Ovládání PC je zde uložena vícekrát, podobně jako fakt, že Ondřej Dvořák ovládá jazyk Angličtina. Reálným problémem je tak fakt, že tato tabulka ukládá více druhů skutečností:

- Vztah Studentů ke schopnostem
- Vztah Studentů k jazykům

Tomu se lze opět vyhnout rozdělením dat do dvou tabulek:

Nová tabulka **STUDENT_SCHOPNOSTI:**

Student	Schopnosti
Jitka Nosková	Ovládání PC
Jitka Nosková	Management
Ondřej Dvořák	Ovládání PC
Ondřej Dvořák	Management

Nová tabulka **STUDENT_JAZYKY:**

Student	Jazyky
Jitka Nosková	Angličtina
Jitka Nosková	Španělština
Ondřej Dvořák	Angličtina
Ondřej Dvořák	Francouzština

Pátá normalizovaná forma

- Tabulka je v páté normalizované formě, pokud ji nelze rozložit na žádné menší tabulky s odlišnými klíči.

Mějme např. tabulku **PRODEJCE**:

Prodejce	Společnost	Produkt
Ondřej Malík	DecaTrade	Matrace
Ludvík Spilka	HoldMaster	Akcie
Ondřej Malík	DecaTrade	Akcie

Tato data by se normálně uložila do jedné tabulky, protože potřebujeme všechny tři záznamy ke zjištění platných kombinací. Ludvík Spilka prodává akcie pod firmou HoldMaster, nemusí však prodávat matrace. Ondřej Malík prodává Akcie i Matrace pro DecaTrade.

Přidejme ještě další data, která rozšíří naši tabulku.

Prodejce	Společnost	Produkt
Ondřej Malík	DecaTrade	Matrace
Ondřej Malík	DecaTrade	Akcie
Ludvík Spilka	HoldMaster	Akcie
Ondřej Malík	HoldMaster	Akcie
Ondřej Malík	HoldMaster	Matrace

S touto závislostí lze tabulku dále normalizovat na tři samostatné tabulky, aniž by došlo ke ztrátě nějakých skutečností.

Nová tabulka **PRODEJCE_PRODUKT**:

Prodejce	Produkt
Ondřej Malík	Matrace
Ondřej Malík	Akcie
Ludvík Spilka	Akcie

Nová tabulka **PRODEJCE_SPOLECNOST**:

Prodejce	Společnost
Ondřej Malík	DecaTrade
Ondřej Malík	HoldMaster
Ludvík Spilka	HoldMaster

Nová tabulka **SPOLEČNOST_PRODUKT**:

Společnost	**Produkt**
DecaTrade	Matrace
Decatrade	Akcie
HoldMaster	Matrace
HoldMaster	Akcie

Další příklady

V **příloze 1** tohoto studijního textu si můžete projít další příklady k normalizaci databáze a to zejména k aplikaci prvních tří normalizačních forem.

6 MySQL

MySQL je systém řízení relační databáze (RDBMS). Je to systém schopný ukládat obrovská množství dat a vracet je zpět pro naplňování potřeb rozličných aplikací. Od nejmenších až po ty největší. MySQL je konkurenceschopným free produktem mezi dobře známými světovými giganty, jako jsou ORACLE či SQL Server.

Obr: Logo databázového systému MySQL

Vlastnosti MySQL

Následující seznam odhaluje nejdůležitější vlastnosti MySQL:

- **Relační databázový systém.**
- **Architektura klient/server.** Systém sestává z databázového serveru a libovolného množství klientů (aplikačních programů), které komunikují se serverem.
- **Kompatibilita s SQL.** MySQL dodržuje současný standard (SQL:2008), avšak s některými omezeními.
- **Vnořené dotazy.** Podpora od verze 4.1
- **Pohledy.** Podpora od verze 5.0
- **Uložené procedury.** Podpora od verze 5.0
- **Triggery.** Podpora v rozšířené podobě od verze 5.1
- **Full-textové vyhledávání.**
- **Unicode.** MySQL podporuje od verze 4.1 všechny myslitelné znakové sady.
- **Transkace.**
- **Omezení cizího klíče.** Podporuje pouze u tabulek InnoDB.
- **GIS funkce.** MySQL podporuje od verze 4.1 ukládání dvojdimenzionálních geografických dat.
- **Programovací jazyky.** Existuje velká spousta programovacích platforem s API a knihoven určených pro vývoj aplikací s MySQL.
- **Rychlost.** Jeden z nejrychlejších RDBMS na trhu.
- **A další.**

Většinu těchto vlastností byste po prostudování tohoto studijního textu měli v základu zvládnout.

Aktuální verze

Stabilně se dnes setkáváme s verzemi MySQL s číselným označením 8.n. Nové verze s sebou přinášejí řadu vylepšení a nové možnosti, které jsou již adaptované v komerčních databázích. Zejména MSSQL a ORACLE. Příklady probírané v tomto studijním textu jsou kompatibilní s verzemi MySQL Community Edition 5.0.n a vyšších.

Jednotlivé verze MySQL se označují atributy alfa, beta, gama a GA.

Alfa – vývojová verze určená pro vývojáře a testery.

Beta – verze téměř kompletní, ale ještě ne zcela otestovaná.

Gama – je víceméně stabilní verze. Cílem je nalézt a opravit zbývající chyby.

GA – je produkční verze pro všeobecné použití, u které vývojáři dospěli k závěru, že může být určena pro spolehlivé nasazení v kritických aplikacích.

V praxi to tedy znamená, že nová verze MySQL (tj. n.n.0) mívá status alfa. Vyšší čísla verzí znamení postupně zvyšující se status na beta, gama…

Připojení k serveru MySQL

Pro správu a cvičné příklady budeme v dalším textu používat dobře známou webovou aplikaci správy MySQL, s označením **PHPMyAdmin (nebo případně aplikaci Adminer)**. Umožňuje v intuitivním grafickém uživatelském prostředí téměř kompletní správu vašich databází a nabízí tak alternativní variantu pro ty, kteří nechtějí využívat prostředí příkazového řádku (Příloha 2). PHPMyAdmin na rozdíl od příkazového řádku nabízí další grafické funkce, které vám umožní lépe se orientovat v prostředí vašich databází, ale také efektivněji zpracovat jejich návrh a následnou optimalizaci. Zároveň uchovává řízení uživatelů a jejich práv na řízení databáze. A hlavně: můžeme jej také používat pro správu serveru MySQL, který neběží na lokálním počítači, ale např. na hostingu našeho webu.

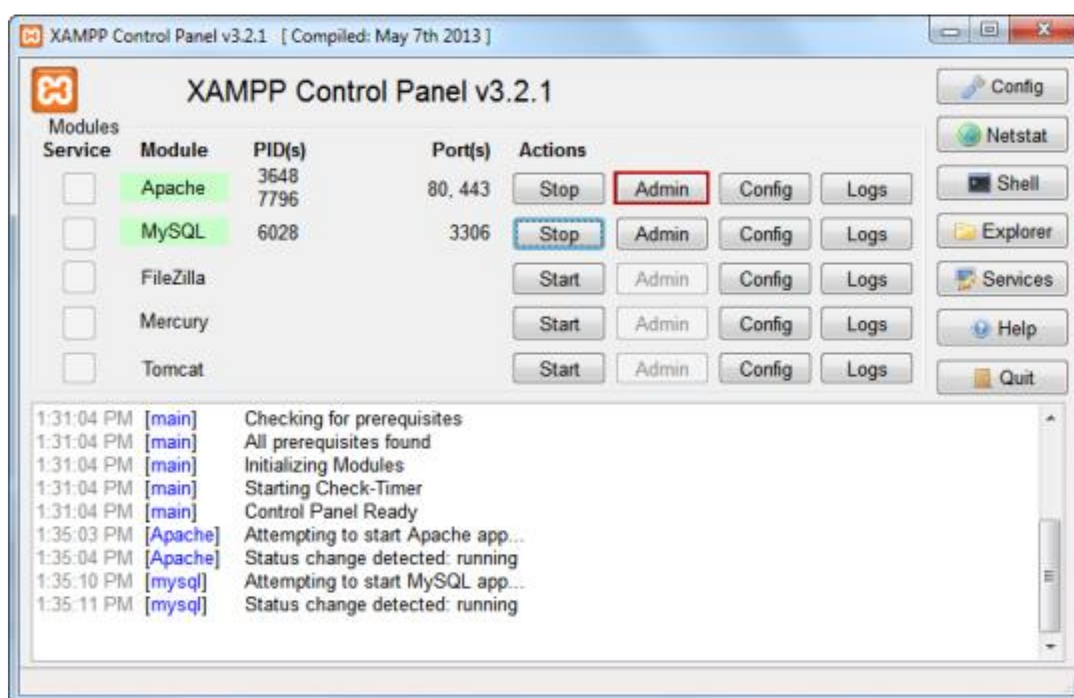
Existují i další, např. grafické konzoly správy MySQL, jako je **MySQL WorkBench**. I tento vám vřele doporučuji, a to zejména pro případ, že nechcete pracovat v prostředí internetového prohlížeče s nutností chodu webového serveru. Více v příloze 5.

Obr: Logo aplikace PHPMyAdmin

Aplikace PHPMyAdmin je rovněž v neustálém vývoji, nicméně verze jsou funkčně stálé a nové doplňky se týkají především optimalizace a programové efektivity. Pro ideální práci vám doporučuji verzi s označením od 5.0.n[2].

Po instalaci MySQL

MySQL byste měli mít nainstalován již z předchozího předmětu Tvorba webových apikací v instalačním balíku **XAMPP**, který je dostupný jak ve verzi pro LINUX, tak Windows. Spolu s MySQL (alternativně MariaDb) jste si v tomto balíku instalovali také webový server **Apache**, skriptovací prostředí **PHP** a také aplikaci **PHPMyAdmin**. Tato varianta je mnou doporučována, a pokud jste tedy tak ještě neučinili, **stáhněte a nainstalujte si balík XAMPP**[3].



Obr: Control Panel systému XAMPP

Přihlášení k PHPMyAdmin (PMA)

Servery MySQL běží v reálných podmínkách jako služby příslušných stanic. V případě, že je u webového serveru umístěna na dané stanici webová aplikace PHPMyAdmin, připojíte se k serveru (resp. k PHPMyAdmin) pomocí internetového prohlížeče adresou URL ve tvaru:

http://www.adresa-serveru-mysql.cz/phpmyadmin

V testovacích podmínkách máme server MySQL na vlastní stanici. V Control panelu systému XAMPP nastartujte servery APACHE a MySQL (pokud již neběží jako služby) a spusťte PHPMyAdmin:

[2] Více o PHPMyAdmin – www.phpmyadmin.net (oficiální), www.phpmyadmin.cz (český web o PHPMyAdmin)
[3] http://www.apachefriends.org/en/xampp.html

http://localhost/phpmyadmin

Pokud jste ještě neučinili jinak, tak vás rovnou uvítá základní nabídka PHPMyAdmin. V opačném případě budete vyzváni k zadání jména a hesla pro vstup do uživatelského prostředí.

Nastavení účtu

Po klasické instalaci MySQL je vytvořen jeden nebo více uživatelských účtů. Defaultní (mající veškerá práva na správu celé MySQL) má jméno **root** (bez hesla). **Toto je pochopitelně nutné okamžitě změnit.** Změnu hesla provedete pod záložkou **Oprávnění** klepnutím na ikonu editace u daného účtu. Pokud jsou k dispozici další, defaultně vytvořené účty, vymažte je. Zabráníte tak budoucím možným bezpečnostním rizikům spojeným s neoprávněnou manipulací s databází. Pro začátek budeme využívat pouze hlavní účet *root*.

Server: localhost

Databáze | SQL | Stav | Proměnné | Znakové sady | Úložiště | Oprávnění | Procesy | Export | Import

Přehled uživatelů

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [Zobrazit vše]

	Uživatel	Počítač	Heslo	Globální oprávnění [1]	Přidělování	
☐	pma	localhost	Ne	RELOAD, SHUTDOWN, PROCESS	Ne	
☐	root	127.0.0.1	Ne	ALL PRIVILEGES	Ano	
☐	root	localhost	Ano	ALL PRIVILEGES	Ano	

Zaškrtnout vše / Odškrtnout vše

Přidat nového uživatele

Odstranit vybrané uživatele

(Odebrat uživatelům veškerá oprávnění a poté je odstranit z tabulek.)

☐ Odstranit databáze se stejnými jmény jako uživatelé.

Poznámka: phpMyAdmin získává oprávnění přímo z tabulek MySQL. Obsah těchto tabulek se může lišit od oprávnění, která server právě používá,

[1] Poznámka: názvy oprávnění v MySQL jsou uváděny anglicky

Obr: PHPMyAdmin – Přehled uživatelů

Se změnou hesla uživatele MySQL souvisí také příslušná změna v konfiguračním skriptu PHPMyAdmin. Nepleťte si prosím MySQL s PHPMyAdmin. MySQL je databázový systém a PHPMyAdmin pouze webová aplikace umožňující správu tohoto systému pracující na rozhraní externí a logické vsrtvy databázového systému.

Aplikace PHPMyAdmin je napsána v PHP. V adresáři /xampp/phpmyadmin/ si v textovém editoru (nejlépe PSPad) otevřete soubor *config.sample.inc.php* nebo pokud je vytvořen - otevřete přímo soubor *config.inc.php*.

V prvním případě (platí pro nové distribuce PHPMyAdmin) uložte soubor do stejného adresáře (/phpmyadmin/) pod novým názvem *config.inc.php*. Původní nemažte a ani v něm nic neměňte.

V nově vytvořeném souboru *config.inc.php* změňte hodnotu (pokud tak již není učiněno) autentizace z *config* na ***cookie***.

```
$cfg['Servers'][$i]['auth_type'] = 'cookie';
```

V druhém případě, kdy soubor *config.inc.php* již existuje, proveďte v případě potřeby stejnou změnu a soubor uložte.

Ověření (autentizace) uživatele

V této části si stručně popíšeme, jaké metody používá PHPMyAdmin k ověřování uživatele:

Metoda config

Nebo-li metoda ověření konfiguračním souborem. Heslo a uživatelské jméno je přímo v textové podobě vloženo v konfiguračním souboru. Tato varianta je vhodná pouze pro lokální testovací systém. Navíc neumožňuje přístup pro více uživatelských účtů než jeden.

Metoda http

Tato metoda spolu s metodou cookie spočívá v nutnosti uživatele se nejprve přihlásit v přihlašovacím okně. Přihlašování je bezpečnější a navíc umožňuje využít pro PHPMyAdmin více uživ. účtů.

Ověřování pomocí http má ovšem také své nevýhody. Přihlašovací údaje se přenáší protokolem HTTP od klienta k serveru jako prostý text. Větší bezpečnosti dosáhneme s protokolem HTTPS. S tímto jsou data přenášena v zašifrované podobě.

Metoda cookie

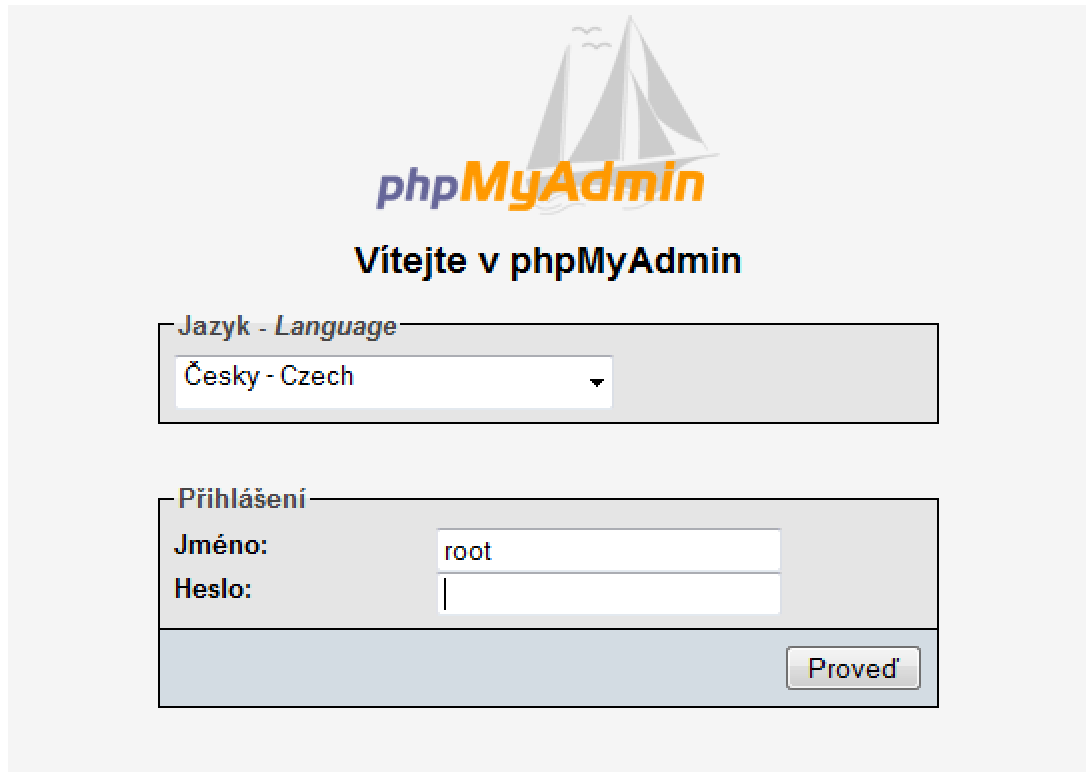
Kromě nastavení autentizace na hodnotu cookie musíme ještě nastavit šifrovací řetězec, který zašifruje přihlašovací údaje, jenž budou poté uloženy v klientském prohlížeči. **Proto v souboru *config.inc.php* nastavte také klíč šifrování cookies:**

```
$cfg['blowfish_secret'] = 'libovolny888KLIC999sifrovani###';
```

Po změnách nezapomeňte soubor *config.inc.php* uložit a nyní se znovu přihlaste do PHPMyAdmin.

http://localhost/phpmyadmin

Bude na vás čekat následující či podobný (moderněji vyhlížející) formulář autentizace:



Obr: PHPMyAdmin – Přihlášení

Vytvoření první databáze

Vytvořit databázi je v PHPMyAdmin triviální záležitost a mnozí z vás si jistě z předchozího semestru pamatují jak na to. Novou databázi jsme vytvářeli pro potřeby instalace redakčního systému WordPress. V horním menu PHPMyAdmin vyberte záložku *Databáze* a v kolonce **Vytvořit novou databázi** zadejte jméno nové databáze (jedinečný název) např. **prvnidb** a nastavte kódování znaků na UTF-8 Czech_CI.

Nebo přímo pod záložkou SQL vložte následující příkaz SQL:

```
CREATE DATABASE `prvnidb` DEFAULT CHARACTER SET utf8 COLLATE
utf8_czech_ci;
```

PHPMyAdmin vás okamžitě vybídne k vytvoření první tabulky a zadání počtu sloupců. Prozatím vše vytvoříme automaticky. Do okna SQL opět vložte následující příkaz:

```
CREATE TABLE `prvnidb`.`studenti` (
`id` INT( 1 ) NOT NULL ,
`jmeno` VARCHAR( 20 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL ,
`prijmeni` VARCHAR( 40 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL ,
`rok_narozeni` SMALLINT UNSIGNED NOT NULL ,
PRIMARY KEY ( `id` )
) ENGINE = MYISAM CHARACTER SET utf8 COLLATE utf8_czech_ci;
```

7 SQL v MySQL

V následujících částech se naučíte základům užití **dotazovacího jazyka SQL** v prostředí databáze MySQL. Věnujte pečlivou pozornost jednotlivým ukázkám a vše si testujte v lokálním provozu MySQL na své stanici. Jednotlivé praktické ukázky budeme popisovat pro práci v PHPMyAdmin.

Vkládání nových záznamů do tabulky

INSERT

Protože nyní máme k dispozici databázi a v ní tabulku, potřebujeme do ní vložit nějaká data. Chceme, aby naše tabulka vypadala následovně:

id	jmeno	prijmeni	rok_narozeni
1	Marek	Stejskal	1987
2	Diana	Zouharová	1990
3	Šimon	Tomík	1989
4	Jakub	Stejskal	1987

Tyto údaje můžete velmi snadno vložit do tabulky pod záložkou **Vložit**, kde pro každý řádek vyplníte příslušná data. ALE. Od této chvíle se již musíte učit striktní syntax SQL v případě RDBMS MySQL.

Zvykneme si zapisovat příkazy jazyka SQL velkými písmeny a zbytek malými. Výrazně nám to usnadní orientaci.

```
INSERT INTO studenti (id, jmeno, prijmeni, rok_narozeni) VALUES (1,
'Marek', 'Stejskal', 1987);

INSERT INTO studenti (id, jmeno, prijmeni, rok_narozeni) VALUES (2,
'Diana', 'Zouharová', 1990);

INSERT INTO studenti (id, jmeno, prijmeni, rok_narozeni) VALUES (3,
'Roman', 'Tomík', 1989);

INSERT INTO studenti (id, jmeno, prijmeni, rok_narozeni) VALUES (4,
'Jakub', 'Stejskal', 1987);
```

Řetězcové pole znaků vyžaduje kolem své hodnoty apostrofy, což jak vidíte, není případ číselných polí.

Existuje rovněž **zkrácená varianta příkazu INSERT**. Mohli byste použít toto:

```
INSERT INTO studenti VALUES (1, 'Marek', 'Stejskal', 1987);
```

Při zadávání příkazů tímto způsobem musíte zadat pole v témže pořadí, v němž jsou definována v databázi!!!

Získávání informací z tabulky

SELECT

Získávání informací z tabulky je v systému MySQL velmi jednoduché. Můžeme použít příkaz **SELECT**. Příkaz SELECT a jeho rozšířenou syntax si budeme ukazovat postupně na jednoduchých příkladech. Vždy stačí, když si kód okopírujete do okna pod záložkou SQL u dané databáze, a stisknete tlačítko **Proveď**:

```
SELECT jmeno FROM studenti;
```

jmeno
Diana
Marek
Roman
Jakub

Příkaz SELECT má několik součástí. První část bezprostředně za SELECT je seznamem polí. Následujícím způsobem byste mohli vrátit jak jmeno, tak prijmeni.

```
SELECT jmeno, prijmeni FROM studenti;
```

Rovněž můžete použít zástupný znak (*) a vrátit všechna pole tabulky:

```
SELECT * FROM studenti;
```

id	jmeno	prijmeni	rok_narozeni
1	Marek	Stejskal	1987
2	Diana	Zouharová	1990
3	Roman	Tomík	1989
4	Jakub	Stejskal	1987

Klauzule WHERE

Tato klauzule je velmi pružná a může obsahovat mnoho podmínek různých druhů:

```
SELECT jmeno, prijmeni FROM studenti WHERE id = 1;

SELECT jmeno, prijmeni FROM studenti WHERE id = 1 AND rok _narozeni
 = 1987;

SELECT jmeno, prijmeni FROM studenti WHERE jmeno = 'Marek' OR jmeno
 = 'Roman';
```

jmeno	prijmeni
Marek	Stejskal
Roman	Tomík

Význam logických **AND** a **OR** by vám pevně doufám měl být již dávno jasný. Můžete pochopitelně využívat i jejich kombinace za pomocí jednoduchých závorek.

```
SELECT jmeno, prijmeni FROM studenti WHERE (jmeno = 'Marek' AND
 rok_narozeni = 1987) OR rok_narozeni  > 1988;
```

jmeno	prijmeni
Diana	Zouharová
Marek	Stejskal
Roman	Tomík

Vyhledávání vzorů

LIKE a %

Podíváme se na další doplňky příkazu **SELECT**. Možná jste někdy v nějakém vyhledávacím formuláři, v textovém políčku viděli předvyplněný znak procent (%). Možná jste ho před zadáním hledaného výrazu smazali, možná ne.

Dejme tomu, že jste zapomněli, jak se jmenuje student Roman. Toman nebo Tomek či Tomík. Existuje možnost SQL v příkazu **LIKE**. Pokud si pamatujete, že příjmení začíná na Tom, můžete použít následující:

```
SELECT * FROM studenti WHERE prijmeni LIKE 'Tom%';
```

id	jmeno	prijmeni	rok_narozeni
3	Roman	Tomík	1989

% je zástupný znak určený pro specifické použití uvnitř podmínky příkazu SELECT. Znamená **nulový nebo větší počet znaků**. Tento zástupný znak můžete použít, kolikrát chcete, což umožňuje vytváření podobných dotazů:

```
SELECT * FROM studenti WHERE prijmeni LIKE '%o%';
```

id	jmeno	prijmeni	rok_narozeni
2	Diana	Zouharová	1990
3	Roman	Tomík	1989

Vrátí se dva záznamy, s příjmením obsahující písmeno „o“.

Řazení záznamů

ORDER BY

Klauzule **ORDER BY**, umožňuje řazení výsledků. Např. abecední seznam studentů z tabulky *studenti* bychom získali následujícím příkazem:

```
SELECT * FROM studenti ORDER BY prijmeni;
```

id	jmeno	prijmeni	rok_narozeni
1	Marek	Stejskal	1987
4	Jakub	Stejskal	1987
3	Roman	Tomík	1989
2	Diana	Zouharová	1990

Pokud chceme zapojit do parametrů řazení více polí a docílit tak efektivnějšího řazení i podle jména, doplníme příkaz následovně:

```
SELECT * FROM studenti ORDER BY prijmeni, jmeno;
```

id	jmeno	prijmeni	rok_narozeni
4	Jakub	Stejskal	1987
1	Marek	Stejskal	1987
3	Roman	Tomík	1989
2	Diana	Zouharová	1990

Pokud bychom rádi řadili výsledky opačně (sestupně), použijeme klíčový výraz **DESC**.

```
SELECT * FROM studenti ORDER BY prijmeni DESC, jmeno DESC;
```

id	jmeno	prijmeni	rok_narozeni
2	Diana	Zouharová	1990
3	Roman	Tomík	1989
1	Marek	Stejskal	1987
4	Jakub	Stejskal	1987

Řazení výsledků sestupně či vzestupně lze rovněž kombinovat, jak ukazuje následující příkaz:

```
SELECT * FROM studenti ORDER BY prijmeni DESC, jmeno ASC;
```

Omezení počtu výsledků

LIMIT

Prozatím jsme vždy získali úplnou sadu výsledků naplňujících zadaná kritéria. V reálných podmínkách to ale může znamenat tisíce záznamů a my si např. v daný moment vystačíme s pouhými třemi. Naštěstí je na toto v SQL myšleno a my můžeme využít služeb klauzule **LIMIT.**

Např. chceme vypsat první dva studenty dle abecedního seznamu:

```
SELECT * FROM studenti ORDER BY prijmeni, jmeno LIMIT 2;
```

id	jmeno	prijmeni	rok_narozeni
4	Jakub	Stejskal	1987
1	Marek	Stejskal	1987

Je-li za klauzulí LIMIT pouze jedna číslice, určuje počet vrácených výsledků.

LIMIT ale umožňuje nejen vracet omezený počet záznamů s počátkem na začátku nebo konci tabulky. MySQL můžete rovněž sdělit potřebné odsazení – jinými slovy od jakého záznamu začít s limitováním.

Jsou-li za klauzulí LIMIT dvě číslice, pak první představuje odsazení a druhé omezení počtu záznamů. Následující příklad vrací prostřední dva záznamy z naší zkušební tabulky *studenti*:

```
SELECT * FROM studenti ORDER BY prijmeni, jmeno LIMIT 1,2;
```

id	jmeno	prijmeni	rok_narozeni
1	Marek	Stejskal	1987
3	Roman	Tomík	1989

Poznánka

Klauzule LIMIT se často používá pro využití tzv. ***stránkování*** – kdy se má na stránku zobrazit např. pouze 10 záznamů a na další záznamy pak vedou odkazy. První stránka výsledků tak používá LIMIT 0,10, druhá následně 10,10, třetí pak 20,10 atd…

Vracení odlišných záznamů

DISTINCT

Někdy nechcete vracet duplicitní záznamy. Jako třeba roky narození našich studentů. Dejme tomu že vás zajímají roky narození, ale nechcete, aby se záznamy opakovaly. Pak použijete následující příkaz:

```
SELECT DISTINCT rok_narozeni FROM studenti;
```

rok_narozeni
1990
1987
1989

Agregační funkce

MAX()

MAX je funkce MySQL, která umožňuje vracet maximální hodnotu daného sloupce. V naší zkušební tabulce by např. bylo možno hledat rok narození nejmladšího ze zaznamenaných studentů.

```
SELECT MAX(rok_narozeni) FROM studenti;
```

MAX(rok_narozeni)
1990

Všimněte si závorek při použití funkce. Funkce se aplikuje na to, co se nachází v závorkách.

COUNT()

Funkce **COUNT()** umožňuje vrátit hodnotu počtu záznamů, odpovídajících zadanému příkazu. Tak např. následující příkaz vrátí počet neduplicitních záznamů let narození našich studentů:

```
SELECT COUNT(DISTINCT rok_narozeni) FROM studenti;
```

COUNT(DISTINCT rok_narozeni)
3

AVG()

Chcete-li vrátit průměrnou hodnotu, např. roku narození našich studentů, příkaz by vypadal následovně:

```
SELECT AVG(rok_narozeni) FROM studenti;
```

AVG(rok_narozeni)
1988.2500

MIN()

Potřebujete-li znát minimální hodnotu:

```
SELECT MIN(rok_narozeni) FROM studenti;
```

MIN(rok_narozeni)
1987

SUM()

Další z řady velmi užitečných funkcí. Ačkoli náš příklad není přímo vypovídající, jistě na něm snadno pochopíte, že funkce **SUM**() umožňuje sečíst hodnoty daného sloupce podle zadaných kritérií:

```
SELECT SUM(rok_narozeni) FROM studenti WHERE prijmeni = 'Stejskal';
```

SUM(rok_narozeni)
3974

Výpočty

SQL nám rovněž dává možnost počítání nad příslušnými dotazy. Příkaz uvedený níže vrací naše studenty s rokem narození v předminulém století:

```
SELECT jmeno, prijmeni, rok_narozeni - 100 FROM studenti;
```

jmeno	prijmeni	rok_narozeni – 100
Diana	Zouharová	1890
Marek	Stejskal	1887

Roman	Tomík	1889
Jakub	Stejskal	1887

Odstraňování záznamů

DELETE

K odstraňování záznamů se používá příkaz **DELETE**. Odstraňuje se vždy celý záznam (řádek) podle zadaných kritérií. Pokud by např. Jakub Stejskal ukončil studium, odstranění příslušného záznamu byste provedli následujícím příkazem:

```
DELETE FROM studenti WHERE id = 4;
```

Jako podmínku byste mohli uvést např. i příjmení. **Ale pozor.** Toto není jednoznačné pole. Příkaz

```
DELETE FROM studenti WHERE prijmeni = 'Stejskal';
```

odstraní jak Marka, tak Jakuba. S využitím příjmení byste museli podmínku dále kombinovat.

```
DELETE FROM studenti WHERE prijmeni = 'Stejskal' AND jmeno =

'Jakub';
```

V případě že studují dva Jakubové Stejskalové, máte opět problém. Proto využívejte k odstraňování záznamů vždy jednoznačné (unikátní) pole nebo jejich unikátní kombinaci.

Všimněte si rozdílu oproti příkazu SELECT. Při odstraňování se za příkaz DELETE neuvádějí žádná jména sloupců – záznam se maže vždy celý.

POZOR

V případě chybějící podmínky vymažete všechna data v tabulce a způsobíte v nejhorším případě nevratnou ztrátu dat.

```
DELETE FROM studenti;
```

Změna záznamu

UPDATE

Již umíme záznam vložit (INSERT), umíme jej i vybrat a vypsat (SELECT), a naposled jsme se jej naučili i odstranit (DELETE). V dalším kroku si předvedeme, jak lze existující záznam pohodlně upravit.

Řekněme například, že Diana Zouharová učinila „neprozřetelný krok“ a vstoupila do stavu manželského se spolužákem Romanem Tomíkem. No, stává se. Důvody, které je k tomu vedly, nechme stranou. Pro nás je teď důležité, že si Diana Zouharová vzala příjmení svého chotě a my tak potřebujeme její příjmení v databázi změnit. Příkaz bude vypadat následovně:

```
UPDATE studenti SET prijmeni = 'Tomíková' WHERE id = 2;
```

Příkaz **UPDATE** využívá další příkaz **SET**, za kterým pomocí názvu sloupce a rovnítka přidělujeme nový název hodnoty. Editace více údajů záznamu v tabulce je jednoduchá, stačí oddělit čárkou zbylé parametry. Tak např.:

```
UPDATE studenti SET prijmeni = 'Tomíková', rok_narozeni = 1989 WHERE
 id = 2;
```

POZOR

Chybějící podmínka WHERE způsobí (podobně jako v případě DELETE) editaci všech záznamů.

Další funkce

Vytvořme si nyní následující tabulku **zbozi**: (Můžete opět jednoduše využít přiložených SQL kódů)

id	polozka	datum_nakupu	datum_spotreby
1	jogurty	2011-07-24	2011-08-24
2	sýry	2011-07-25	2011-07-30
3	uzeniny	2011-07-26	2011-07-29

```
CREATE TABLE `prvniDB`.`zbozi` (
`id` INT( 1 ) NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`polozka` VARCHAR( 50 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL ,
`datum_nakupu` DATE NOT NULL ,
`datum_spotreby` DATE NOT NULL

) ENGINE = MYISAM CHARACTER SET utf8 COLLATE utf8_czech_ci;
```

```
INSERT INTO zbozi (id, polozka, datum_nakupu, datum_spotreby) VALUES
('', 'jogurty', '2011-07-24', '2011-08-24');

INSERT INTO zbozi (id, polozka, datum_nakupu, datum_spotreby) VALUES
('', 'sýry', '2011-07-25', '2011-07-30');

INSERT INTO zbozi (id, polozka, datum_nakupu, datum_spotreby) VALUES
('', 'uzeniny', '2011-07-26', '2011-07-29');
```

AUTO_INCREMENT

Při vytváření pole s primárním klíčem - identifikátoru **id,** jsme využili vestavěné funkce MySQL – **AUTO_INCREMENT**. Tato funkce si sama hlídá unikátní číselné pole a vkládá identifikátor (celé číslo) automaticky. Proto identifikátor při vkládání neuvádíme tzv. natvrdo, ale nahradíme danou hodnotu prázdnými apostrofy nebo hodnotou *NULL*. MySQL si už hodnotu vloží a inkrementuje automaticky.

Automaticky inkrementované pole (zvyšované o jedničku) je užitečný prvek, který dovoluje automatické inkrementování hodnoty zvoleného pole po každém vložení nového záznamu.

Pouze jedno pole v záznamu může být automaticky inkrementované a toto pole musí být číselným primárním klíčem nebo číselným jedinečným indexem.

Resetování hodnoty A_I

Poslední vloženou hodnotu automatické inkrementace můžeme zjistit pomocí funkce MySQL, funkce **LAST_INSERT_ID()**. Příkaz ale musíte spouštět v jednom sledu s příkazem vložení. Samotný bude vracet hodnotu 0.

```
SELECT LAST_INSERT_ID() FROM zbozi LIMIT 1;
```

Chceme-li resetovat čítač automatické inkrementace tak, aby začínal na určité hodnotě, např. zpět na jedničce po vymazání všech záznamů, můžeme použít následující:

```
ALTER TABLE nazev_tabulky AUTO_INCREMENT = hodnota_a_i;

ALTER TABLE zbozi AUTO_INCREMENT = 10;
```

Změnu můžete provést také ručně v PHPMyAdmin. U dané tabulky pod záložkou **Úpravy**.

LAST_INSERT_ID()

Tato funkce má své opodstatnění. Její efektivní využití najdete např. v části věnované spojení PHP a MySQL. Ale mohou s ní nastat také některé problémy:

Hodnota vrácená funkcí LAST_INSERT_ID() se neresetuje na tutéž hodnotu, na kterou se resetuje čítač automatické inkrementace. Místo toho se vrací na 1.
Dané číslo je udržováno u jednotlivých připojení, takže dojde-li k novým vložením z jiného připojení, hodnota vrácená touto funkcí nebude aktualizovaná!

Používání funkcí data

DATE_FORMAT()

Všimněte si, že MySQL si defaultně ukládá datum (datový typ DATE) ve formátu **RRRR-MM-DD**. Tento tvar není pro nás vždy nejsrozumitelnější. Požadujeme-li vrácení data nákupu ve formátu DD.MM.RRRR, můžeme využít funkci DATE_FORMAT():

```
SELECT DATE_FORMAT(datum_nakupu, '%d.%m.%Y') FROM zbozi;
```

DATE_FORMAT (datum_nakupu, '%d.%m.%Y')
24.07.2011
25.07.2011
26.07.2011

Část uvozenou apostrofy označujeme jako formátovací řetězec. Uvnitř tohoto formátovacího řetězce používáme **specifikátor** k přesnému zadání vráceného formátu. Specifikátorů formátování data je spousta a můžete si je prohlédnout na odkazu v zápatí této stránky[4]:

Vracení aktuálního data a času

Chcete-li zjistit, jaké je aktuální datum podle nastavení hodin serveru, můžete použít funkci **CURDATE**(). Nebo funkci **NOW**(), která vrací i čas:

[4] http://dev.mysql.com/doc/refman/5.5/en/date-and-time-functions.html#function_date-format

```
SELECT CURDATE(), NOW();
```

CURDATE()	NOW()
2011-08-17	2011-08-17 18:06:07

Funkce YEAR(), MONTH() a DAY()

Tyto funkce můžem použít pouze na atributy, které mají v databázi definován datový datum nebo některo z jeho variant. (viz kapitola 9 Sloupce typu datum a čas)

```
SELECT YEAR(datum_nakupu), MONTH(datum_nakupu), DAY(datum_nakupu)
 FROM zbozi;
```

YEAR(datum_nakupu)	MONTH(datum_nakupu)	DAY(datum_nakupu)
2011	7	24
2011	7	25
2011	7	26

Zjištění dne v roce

Chcete-li zjistit jaký je den v roce (od 1 do 366) v dané datum, stačí použít funkci **DAYOFYEAR()**.

```
SELECT DAYOFYEAR(datum_spotreby) FROM zbozi WHERE id = 1;
```

8 Vytváření pokročilejších dotazů

Ještě předtím než se pustíme do pokročilejších dotazů, ukážeme si dva užitečné doplňky SQL.

Přiřazení nového záhlaví sloupcům pomocí AS

Pomocí klíčového slova **AS** můžeme vytvářet *aliasy*, které dávají sloupcům námi zvolený název:

```
SELECT polozka, MONTH(datum_spotreby) AS mesic FROM zbozi;
```

polozka	mesic
jogurty	8
sýry	7
uzeniny	7

Spojování sloupců pomocí CONCAT()

Spojování sloupců je užitečná funkce, která nám dovolí spojit dva či více sloupců do jednoho. Typickým příkladem může být spojení jména a příjmení do jednoho údaje:

```
SELECT CONCAT(jmeno, ' ', prijmeni) FROM studenti;
```

CONCAT(jmeno, ' ', prijmeni)
Diana Tomíková
Marek Stejskal
Roman Tomík

Druhou možností je rozšířená funkce **CONCAT_WS**, která vrací výsledek se zvoleným separátorem.

Práce s více tabulkami

V této chvíli již ovládáte jednoduché dotazy. V mnoha případech (např. jednoduché webové aplikace) si takto vystačíte. Ovšem nastanou situace, kdy požadujete více seskupení – nejčastějším případem bývá spojení dvou či více tabulek dohromady (takový typ dotazu se označuje za spojení **JOIN**). A je vlastně principem činnosti relační databáze.

Nejprve si v naší databázi vytvoříme dvě nové tabulky. Tabulka **autori** bude obsahovat identifikátor, jméno a příjmení autorů knih, a tabulka **knihy** bude obsahovat identifikátor, identifikátor autora, název a ISBN knihy:

Nová tabulka **autori**:

id	jmeno	prijmeni
1	Agatha	Christie
2	Dan	Brown
3	Andrew	Oppel

```
CREATE TABLE `prvniDB`.`autori` (
`id` INT( 1 ) NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`jmeno` VARCHAR( 20 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL ,
`prijmeni` VARCHAR( 40 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL

) ENGINE = MYISAM CHARACTER SET utf8 COLLATE utf8_czech_ci;
```

```
INSERT INTO autori (id, jmeno, prijmeni) VALUES ('', 'Agatha',
'Christie');

INSERT INTO autori (id, jmeno, prijmeni) VALUES ('', 'Dan', 'Brown');

INSERT INTO autori (id, jmeno, prijmeni) VALUES ('', 'Andrew',
'Oppel');
```

Nová tabulka **knihy**:

id	id_autor	nazev	isbn
1	1	Vražda v Orient-Expressu	978-80-242-2097-0
2	1	Oznamuje se vražda	978-80-242-2161-8
3	2	Andělé a démoni	80-7203-787-0
4	2	Šifra mistra Leonarda	80-86518-62-0
5	3	Databáze bez předchozích znalostí	80-251-1199-7

```
CREATE TABLE `prvniDB`.`knihy` (
`id` INT( 1 ) NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`id_autor` INT( 1 ) NOT NULL ,
`nazev` VARCHAR( 255 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL ,
`isbn` VARCHAR( 20 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NULL

) ENGINE = MYISAM CHARACTER SET utf8 COLLATE utf8_czech_ci;
```

```
INSERT INTO knihy (id, id_autor, nazev, isbn) VALUES ('', 1, 'Vražda
v Orient-Expressu', '978-80-242-2097-0');

INSERT INTO knihy (id, id_autor, nazev, isbn) VALUES ('', 1, 'Oznamuje
se vražda', '978-80-242-2161-8');

INSERT INTO knihy (id, id_autor, nazev, isbn) VALUES ('', 2, 'Andělé a
démoni', '80-7203-787-0');

INSERT INTO knihy (id, id_autor, nazev, isbn) VALUES ('', 2, 'Šifra
mistra Leonarda', '80-86518-62-0');

INSERT INTO knihy (id, id_autor, nazev, isbn) VALUES ('', 3, 'Databáze
bez předchozích znalostí', '80-251-1199-7');
```

Na těchto tabulkách je již patrné spojení. **Primární klíč** v tabulce *autori* (identifikátor ID) je **cizím klíčem** v tabulce *knihy* (id_autor). Vytvořme tedy dotaz, který bude vracet název díla, ISBN a také celé jméno autora:

```
SELECT nazev, isbn, CONCAT(jmeno, ' ', prijmeni) AS cele_jmeno
FROM knihy, autori
WHERE knihy.id = 3 AND autori.id = id_autor;
```

nazev	isbn	cele_jmeno
Andělé a démoni	80-7203-787-0	Dan Brown

První část dotazu za příkazem SELECT uvádí pole, které chcete vrátit. V případě že vybíráte pole z dvou různých tabulek a tato pole nemají v tabulkách shodný název, nemusíte před ně umisťovat název tabulky a doplňovat tečku, jak vidíte u podmínek za WHERE.

Druhá část příkazu za FROM říká MySQL, které tabulky se mají použít.

Třetí část za WHERE obsahuje příslušnou podmínku. Zde si všimněte, že již musíme uvádět název tabulky před identifikátor (knihy.id), protože pole s názvem id obsahují obě tabulky. Je tak bezpodmínečně nutné to MySQL přesně určit (vybírám knihu s klíčem 3). Další část činí z dotazu spojení. Právě zde říkáme MySQL, podle kterých polí se má spojit.

Protože jsme podmínkou WHERE knihy.id = 3 zároveň určili (ač to není v příkazu viditelné) id_autor=2 (cizí klíč), tak spojením autori.id = id_autor jsme získali hodnotu primárního klíče daného záznamu v tabulce autori. Odtud jsme již získali (resp. MySQL) příslušné jméno a příjmeni.

Mnozí programátoři si striktně potrpí na přesném dodržování celé syntaxe příkazů spojení, který by vypadal takto. Výsledek je pochopitelně stejný:

```
SELECT knihy.nazev, knihy.isbn, autori.jmeno, autori.prijmeni
FROM knihy, autori
WHERE knihy.id = 3 AND autori.id = knihy.id_autor;
```

V dalším příkazu si ukážeme, jak vypsat všechny díla autorky A. Christie.

```
SELECT jmeno, prijmeni, nazev
FROM autori, knihy
WHERE autori.id = 1 AND id_autor = autori.id;
```

jmeno	prijmeni	nazev
Agatha	Christie	Vražda v Orient-Expressu
Agatha	Christie	Oznamuje se vražda

Seskupení v dotazu

GROUP BY

Dejme tomu, že chceme zjistit, kolik knih má v tabulce zastoupen ten který autor. V rámci jednoho autora by to byla hračka. Použili bychom funkci COUNT(). Například takto:

```
SELECT id_autor, COUNT(id) FROM knihy WHERE id_autor = 1;
```

id_autor	COUNT (id)
1	2

Pokud bychom ale chtěli celou tabulku pro všechny autory, musíme počty k určitému identifikačnímu poli seskupit. Pro tyto účely se používá klauzule **GROUP BY**:

```
SELECT id_autor, COUNT(id) FROM knihy GROUP BY id_autor;
```

id_autor	COUNT(id)
1	2
2	2
3	1

Můžeme samozřejmě použít i spojení:

```
SELECT jmeno, prijmeni, COUNT(knihy.id) AS pocet_knih
FROM autori, knihy
WHERE id_autor = autori.id
GROUP BY id_autor
ORDER BY jmeno;
```

jmeno	prijmeni	pocet_knih
Agatha	Christie	2
Andrew	Oppel	1
Dan	Brown	2

Vybrali jsme jméno a příjmení autora, a pomocí AS jsme vybrali počet knih. Označili jsme příslušné tabulky (FROM), které budeme spojovat. Vytvořili jsme podmínku spojení mezi tabulkami (WHERE) a provedli seskupení GROUP BY. Nakonec jsme výsledek seřadili klauzulí ORDER BY podle jména autora.

Klauzule GROUP BY se musí použít vždy, když spojujeme tabulky a používáme některou z agregačních funkcí!

TIP

GROUP BY je klauzule, která je pro MySQL nepostradatelná. Ve spojení s funkcí **CONCAT(),** kterou jsme si představili dříve, utváří další funkci, tzv. **GROUP_CONCAT()**. Její využití si můžete prohlédnout v příloze 3 na praktickém příkladu.

Tato funkce vrací řetězec, který je složen z hodnot skupiny.

Klauzule HAVING

Klauzule HAVING zastává v jazyku SQL jednu velmi podstatnou úlohu. Pomocí ní můžeme do dotazů vkládat podmínky s agregačními funkcemi, což za klíčovým výrazem WHERE nelze. Pomocí klasicky položeného dotazu s podmínkou nemůžeme např. vypsat autory, kteří v naší knihovně mají více jak jednu knihu. S klauzulí **HAVING** je to hračka:

```
SELECT jmeno, prijmeni, COUNT(knihy.id) AS pocet_knih
FROM autori, knihy
WHERE id_autor = autori.id
GROUP BY id_autor
HAVING COUNT(knihy.id) > 1
ORDER BY jmeno;
```

jmeno	prijmeni	pocet_knih
Agatha	Christie	2
Dan	Brown	2

SOUHRN

Máme za sebou nezbytný úvod do problematiky dotazů databázového systému MySQL. Jeho moc a síla spočívá ve schopnosti strukturovat data a přebírat je pro široké množství vysoce specifických požadavků. Průmyslovým standardem manipulování a definování dat je jazyk SQL. Mezi nejdůležitější příkazy patří:

- Příkaz **CREATE** vytváří databáze a tabulky v databázích.
- Příkaz **INSERT** umisťuje záznamy do tabulky.
- Příkaz **SELECT** vrací výsledky dotazu.
- Příkaz **DELETE** odstraňuje záznamy z tabulky.
- Příkaz **UPDATE** mění data v tabulce.
- Příkaz **ALTER** mění strukturu tabulky, přičemž používá klauzule jako **ADD** k přidání nového sloupce, **CHANGE** ke změně názvu či definice existujícího sloupce, **RENAME** k přejmenování tabulky, nebo **DROP** k odstranění tabulky.

Sílu MySQL dále zvyšují funkce, které poznáme podle závorek bezprostředně za názvem funkce.

Dále se seznámíme se zásadními otázkami strukturování dat, přesuneme se k pokročilejšímu dotazování SQL a podíváme se na různé typy tabulek, které MySQL používá pro různá řešení problémů.

9 Datové typy a typy tabulek

Datové typy

V MySQL existují tři hlavní typy sloupců:

- Číselné
- Řetězcové
- A typ datum.

Třebaže existuje spousta dalších specifických typů, všechny lze zařadit do jednoho ze tří hlavních typů. Obecně vždy volíme nejmenší možný typ sloupce pro dané hodnoty, protože databáze pak bude šetřit místo a nabízet rychlejší přístup i aktualizaci. Jestliže ale naopak zvolíme příliš malý typ sloupce, může docházet ke ztrátě dat nebo k jejich ořezávání při vkládání.

Číselné typy sloupců

Existují dva hlavní druhy číselných typů: **celočíselné typy** (celá čísla bez desetinných nebo zlomkových částí), a **typy s pohyblivou desetinnou čárkou**. Všechny číselné typy nabízejí dvě možnosti:

1. **UNSIGNED** – znepřístupňuje záporná čísla (přičemž tak rozšiřuje kladný rozsah u celočíselných typů)
2. **ZEROFILL** – doplňuje danou hodnotu nulami namísto obvyklých mezer a zároveň ji automaticky činí UNSIGNED.

Např.:

```
CREATE TABLE testA(id TINYINT ZEROFILL);

INSERT INTO testA VALUES(2);

INSERT INTO testA VALUES(-1);

INSERT INTO testA VALUES(256);

SELECT id  FROM testA;
```

id
002
000
255

Jelikož je pole UNSIGNED všimněte si, že záporné číslo se přizpůsobilo dolnímu rozsahu a protože 256 přesahuje horní rozsah, upraví se na 255, což je max. povolená hodnota.

U specifikací datových typů budeme používat tato označení[5]:

- **hranaté závorky „[ ]“** – jakákoli informace
- **M** – maximální zobrazovací šířka; když není uvedeno jinak, je to celé číslo od 1 do 255
- **D** – počet číslic za desetinnou čárkou (pokud datový typ desetinnou čárku podporuje); musí to být celé číslo od 0 do 30; D nesmí být větší než M − 2, jinak se M upraví na D + 2
- **CC, YY, MM, DD** – označení u časových a datových typů; označení znamenají: století, rok, měsíc, den
- **hh, mm, ss** – označení u časových a datových typů; označení znamenají: hodina, minuta, sekunda

Kategorie celočíselných typů

TINYINT [(M)]

Použití: velmi malá celá čísla
Atributy: AUTO_INCREMENT, UNSIGNED, SIGNED, ZEROFILL
Rozsah čísel: −128 až 127 nebo 0 až 255 pro UNSIGNED
Výchozí hodnota: NULL, pokud může být sloupec NULL; 0 je-li NOT NULL
Místo v paměti: 1 bajt

SMALLINT [(M)]

Použití: malá celá čísla
Atributy: AUTO_INCREMENT, UNSIGNED, SIGNED, ZEROFILL
Rozsah čísel: −32768 až 32767 nebo 0 až 65535 pro UNSIGNED
Výchozí hodnota: NULL, pokud může být sloupec NULL; 0 je-li NOT NULL
Místo v paměti: 2 bajty

MEDIUMINT [(M)]

Použití: středně velká celá čísla
Atributy: AUTO_INCREMENT, UNSIGNED, SIGNED, ZEROFILL
Rozsah čísel: −8388608 až 8388607 nebo 0 až 16777215 pro UNSIGNED
Výchozí hodnota: NULL, pokud může být sloupec NULL; 0 je-li NOT NULL
Místo v paměti: 3 bajty

INT [(M)]

Použití: běžně velká celá čísla
Atributy: AUTO_INCREMENT, UNSIGNED, SIGNED, ZEROFILL
Rozsah čísel: −2147483648 až 2147483647 nebo 0 až 4294967295 pro UNSIGNED
Výchozí hodnota: NULL, pokud může být sloupec NULL; 0 je-li NOT NULL
Místo v paměti: 4 bajty

[5] Převzato a doplněno z http://programujte.com/clanek/2007052903-prehled-datovych-typu-v-mysql/

BIGINT [(M)]

Použití: velká celá čísla
Atributy: AUTO_INCREMENT, UNSIGNED, SIGNED, ZEROFILL
Rozsah čísel: −9223372036854775808 až 9223372036854 nebo 0 až 18446744073709551615 pro UNSIGNED
Výchozí hodnota: NULL, pokud může být sloupec NULL; 0 je-li NOT NULL
Místo v paměti: 8 bajtů

Kategorie s pohyblivou desetinnou čárkou

FLOAT (p)

Použití: čísla v pohyblivé řádové čárce s bity přesnosti, které jsou minimálně požadované a určené hodnotou *p*. Pro hodnoty *p* od 0 do 24 je datový typ *FLOAT* ekvivalentní datovému typu *float* bez označení *M* nebo *D*. Pro hodnoty od 25 do 53 je tento datový typ ekvivalentní pro *DOUBLE* bez označení *M* nebo *D*.
Atributy: UNSIGNED, SIGNED a ZEROFILL
Rozsah čísel: viz dále v popisu *FLOAT* a *DOUBLE*
Výchozí hodnota: NULL, pokud může být sloupec NULL; 0 je-li NOT NULL
Místo v paměti: 4 bajty pro hodnoty 0 až 24 (= *FLOAT*), 8 bajtů pro hodnoty 24 až 53 (= *DOUBLE*)

FLOAT [(M,D)]

Použití: malá čísla v pohyblivé řádové čárce. Je-li *D* rovno 0, hodnoty sloupce nemají desetinnou čárku a ani desetinnou část. Pokud není *M* ani *D* uvedeno, není stanovena velikost zobrazení ani přesnost.
Atributy: UNSIGNED, SIGNED a ZEROFILL
Rozsah čísel: nejmenší nenulové hodnoty jsou ±1,175494351E−38; největší nenulové hodnoty jsou −±3,402823466E+38. Je-li sloupec UNSIGNED, jsou záporné hodnoty zakázané.
Výchozí hodnota: NULL, pokud může být sloupec NULL; 0 je-li NOT NULL
Místo v paměti: 4 bajty

Double [(M,D)] – shodné s Real

Použití: velká čísla v pohyblivé řádové čárce; větší přesnost než u *FLOAT*. Je-li *D* rovno nule, hodnoty sloupce nemají desetinnou čárku a ani desetinnou část. Pokud není *M* ani *D* uvedeno, není stanovena velikost zobrazení ani přesnost.
Atributy: UNSIGNED, SIGNED a ZEROFILL
Rozsah čísel: nejmenší nenulové hodnoty jsou ±2,2250738585072014E-308; největší nenulové hodnoty jsou −±1,17976931348623157E+308. Je-li sloupec UNSIGNED, jsou záporné hodnoty zakázané.
Výchozí hodnota: NULL, pokud může být sloupec NULL; 0 je-li NOT NULL
Místo v paměti: 8 bajtů

Real [(M,D)] – shodné s Double

Použití: velká čísla v pohyblivé řádové čárce; větší přesnost než u *FLOAT*. Je-li *D* rovno nule, hodnoty sloupce nemají desetinnou čárku a ani desetinnou část. Pokud není *M* ani *D* uvedeno, není stanovena velikost zobrazení ani přesnost.
Atributy: UNSIGNED, SIGNED a ZEROFILL
Rozsah čísel: nejmenší nenulové hodnoty jsou ±2,2250738585072014E-308; největší nenulové hodnoty jsou −±1,17976931348623157E+308.

Výchozí hodnota: NULL, pokud může být sloupec NULL; 0 je-li NOT NULL
Místo v paměti: 8 bajtů

Decimal [(M,D)]

Použití: velká čísla v pohyblivé řádové čárce ukládané jako řetězec (1 bajt na číslici, desetinnou čárku nebo znaménko „–“. Je-li *D* rovno nule, hodnoty sloupce nemají desetinnou čárku a ani desetinnou část. Pokud není *M* ani *D* uvedeno, jsou výchozí hodnoty 10 a 0.
Atributy: UNSIGNED, SIGNED a ZEROFILL
Rozsah čísel: rozsah čísel je stejný jako u *DOUBLE*; rozsah typu *DECIMAL* určují označení *M* a *D*
Výchozí hodnota: NULL, pokud může být sloupec NULL; 0 je-li NOT NULL
Místo v paměti: za normálních okolností *M+2* bajtů (dva bajty navíc jsou pro znaky znaménka a desetinné čárky). Je-li sloupec *UNSIGNED*, velikost se zmenšuje o jeden bajt (nemusí se ukládat znaménko). Je-li *D* rovno nule, velikost se zmenšuje také o jeden bajt (nemusí se ukládat desetinná čárka).

Další číselné typy

V praxi můžeme narazit ještě nejméně na tři další číselné typy. Jedním z nich je typ **SERIAL**, který je v podstatě shodný s **BIGINT**. Dle manuál MySQL platí:

```
SERIAL ⇔ BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE
```

Dalším typem je **BIT(M)**, což je typ pro bitovou sekvenci 1 a 0, v maximální šířce M = 64.

Posledním typem je typ **BOOL,** který je ekvivalentní s typem TINYINT(1).

Zásady volby číselného typu

Při volbě číselného typu se držte vždy následujících zásad:

- Vyberte nejmenší použitelný typ.
- V případě celých čísel volte celočíselný typ.
- Pro vysokou přesnost používejte raději celočíselné typy a nikoli typy s pohyblivou desetinnou čárkou (na čísla s pohyblivou desetinnou čárkou působí chyby při zaokrouhlování).

Řetězcové typy

Řetězcové typy sloupců k ukládání libovolných znakových dat, jako jsou jména, adresy, kombinace čísel a znaků atd. Poradí si téměř se vším, ovšem je chybou je zaměňovat s daty nebo určitými číselnými údaji.

CHAR [(M)]

Použití: řetězec pevně dané délky dlouhý 0 až *M* (*M* musí být celé číslo v rozmezí od 0 do 255). Není-li *M* uvedeno, je výchozí hodnota nastavena na 1. Pokud je řetězec ukládaný do databáze delší než *M*, tak se zkrátí na délku *M* a zbytek se ořeže. Je-li naopak ukládaný řetězec kratší než *M*, doplní se mezerami na délku *M*, ta je však při načítání dat z databáze ignorována, takže nemusíme mít obavy z mezer navíc.
Atributy: BINARY, CHARACTER SET
Povolená délka: 0 až *M* bajtů
Výchozí hodnota: NULL, pokud může být sloupec NULL; '' (= prázdný řetězec), pokud je NOT NULL
Místo v paměti: *M* bajtů

VARCHAR [(M)]

Použití: řetězec pevně dané délky dlouhý 0 až *M* (*M* musí být celé číslo v rozmezí od 0 do 65 535 (pozor – do verze MySQL 5 pouze 0 až 255). Pokud je řetězec ukládaný do databáze delší než *M*, tak se zkrátí na délku *M* a zbytek se „ztratí“. Je-li naopak ukládaný řetězec kratší než *M*, koncové mezery se při ukládání odstraní.
Atributy: BINARY, CHARACTER SET
Povolená délka: 0 až *M* bajtů
Výchozí hodnota: NULL, pokud může být sloupec NULL; '' (= prázdný řetězec), pokud je NOT NULL
Porovnávání: není-li uveden atribut *BINARY*, nerozlišuje velikost písmen
Místo v paměti: délka hodnoty plus 1 bajt pro zaznamenání délky

TINYBLOB

Použití: malá hodnota BLOB (binární objekt)
Atributy: jen globální atributy
Povolená délka: 0 až 255 bajtů
Výchozí hodnota: NULL, pokud může být sloupec NULL; '' (= prázdný řetězec), pokud je NOT NULL
Porovnávání: rozlišuje velikost písmen
Místo v paměti: délka hodnoty plus 1 bajt pro zaznamenání délky

BLOB

Použití: normální hodnota BLOB
Atributy: jen globální atributy
Povolená délka: 0 až 65535 bajtů
Výchozí hodnota: NULL, pokud může být sloupec NULL; '' (= prázdný řetězec), pokud je NOT NULL
Porovnávání: rozlišuje velikost písmen
Místo v paměti: délka hodnoty plus 2 bajty pro zaznamenání délky

MEDIUMBLOB

Použití: středně velká hodnota BLOB
Atributy: jen globání atributy
Povolená délka: 0 až 16777215 bajtů

Výchozí hodnota: NULL, pokud může být sloupec NULL; '' (= prázdný řetězec), pokud je NOT NULL
Porovnávání: rozlišuje velikost písmen
Místo v paměti: délka hodnoty plus 3 bajty pro zaznamenání délky

LONGBLOB

Použití: velká hodnota BLOB
Atributy: jen globální atributy
Povolená délka: 0 až 4294967295 bajtů
Výchozí hodnota: NULL, pokud může být sloupec NULL; '' (= prázdný řetězec), pokud je NOT NULL
Porovnávání: rozlišuje velikost písmen
Místo v paměti: délka hodnoty plus 4 bajty pro zaznamenání délky

TINYTEXT

Použití: malá hodnota TEXT
Atributy: CHARACTER SET
Povolená délka: 0 až 255 bajtů
Výchozí hodnota: NULL, pokud může být sloupec NULL; '' (= prázdný řetězec), pokud je NOT NULL
Porovnávání: nerozlišuje velikost písmen
Místo v paměti: délka hodnoty plus 1 bajt pro zaznamenání délky

TEXT

Použití: normální hodnota TEXT
Atributy: CHARACTER SET
Povolená délka: 0 až 65535 bajtů
Výchozí hodnota: NULL, pokud může být sloupec NULL; '' (= prázdný řetězec), pokud je NOT NULL
Porovnávání: nerozlišuje velikost písmen
Místo v paměti: délka hodnoty plus 2 bajty pro zaznamenání délky

MEDIUMTEXT

Použití: středně velká hodnota TEXT
Atributy: CHARACTER SET
Povolená délka: 0 až 16777215 bajtů
Výchozí hodnota: NULL, pokud může být sloupec NULL; '' (= prázdný řetězec), pokud je NOT NULL
Porovnávání: nerozlišuje velikost písmen
Místo v paměti: délka hodnoty plus 3 bajty pro zaznamenání délky

LONGTEXT

Použití: velká hodnota TEXT
Atributy: CHARACTER SET
Povolená délka: 0 až 4294967295 bajtů
Výchozí hodnota: NULL, pokud může být sloupec NULL; '' (= prázdný řetězec), pokud je NOT NULL
Porovnávání: nerozlišuje velikost písmen
Místo v paměti: délka hodnoty plus 4 bajty pro zaznamenání délky

ENUM ('hodnota1', 'hodnota2',…)

Použití: výčet hodnot; hodnoty ze sloupce mohou mít přiřazeny právě jednu hodnotu ze seznamu hodnot
Atributy: jen globální atributy

Výchozí hodnota: NULL, pokud může být sloupec NULL; první hodnota výčtu, pokud je NOT NULL
Porovnávání: nerozlišuje velikost písmen
Místo v paměti: 1 bajt pro výčty s členy 1 až 255; 2 bajty pro výčty s členy 256 až 65535

SET ('hodnota1', 'hodnota2',...)

Použití: množina; hodnotám ze sloupců lze přiřadit nula nebo více členů ze seznamu hodnot
Atributy: jen globální atributy
Výchozí hodnota: NULL, pokud může být sloupec NULL; '' (= prázdná množina), pokud je NOT NULL
Porovnávání: nerozlišuje velikost písmen
Místo v paměti: 1 bajt pro množiny s 1 až 8 členy; 2 bajty pro množiny s 9 až 16 členy; 3 bajty pro množiny s 17 až 24 členy; 4 bajty pro množiny s 25 až 32 členy; 8 bajtů pro množiny s 33 až 64 členy

Zásady volby řetězcového typu

Při volbě řetězcového typu pracujte podle následujících zásad:

- Čísla nikdy neukládejte do řetězcových typů. Každá číslice v řetězcovém poli zabírá celý bajt, kdežto v číselném poli je uložena v bitech.
- Požadujete-li rychlost, volte sloupce s pevnou šířkou, jako např. CHAR.
- Chcete-li ušetřit místo, používejte dynamické sloupce, jako např. VARCHAR.
- Chcete-li omezit obsah sloupce na jednu z možností, použijte ENUM.
- Pro text, který chcete prohledávat bez rozlišování velikosti znaků, použijte typ TEXT.
- Pro text, který chcete prohledávat s rozlišováním velikosti znaků, použijte typ BLOB.
- V případě obrázků a dalších binárních objektů využívejte ukládání v systému souborů a nikoli databázi. Ta je vhodná v případě potřeby ukládání menších souborů, které vyžadují vyšší zabezpečení.

Sloupce typu datum a čas

Sloupce typu datum a čas jsou určeny pro speciální podmínky nutné pro práci s časovými údaji a lze je používat k ukládání takových údajů, jako je čas dne nebo libovolné datum vložení, editace atp.

DATE

Použití: datum ve formátu '*RRRR-MM-DD*'
Atributy: jen globální atributy
Rozsah: '1000-01-01' až '9999-12-31'
Nulová hodnota: '0000-00-00'
Výchozí hodnota: NULL, pokud může být sloupec NULL; '0000-00-00', je-li NOT NULL
Místo v paměti: 3 bajty

TIME

Použití: čas ve formátu '*hh:mm:ss*'; pro záporné hodnoty '-hh:mm:ss'; reprezentuje uplynulý čas, nebo se dá chápat jako čas dne
Atributy: jen globální atributy
Rozsah: '-838:59.59' až '838:59:59'
Nulová hodnota: '00:00:00'
Výchozí hodnota: NULL, pokud může být sloupec NULL; '00:00:00', je-li NOT NULL
Místo v paměti: 3 bajty

DATETIME

Použití: datum a čas ve formátu '*RRRR-MM-DD hh:mm:ss*'
Atributy: jen globální atributy
Rozsah: '1000-01-01 00:00:00' až '999-12-31 23:59:59'
Nulová hodnota: '000-00-00 00:00:00'
Výchozí hodnota: NULL, pokud může být sloupec NULL; '0000-00-00 00:00:00', je-li NOT NULL
Místo v paměti: 8 bajtů

TIMESTAMP [(M)]

Použití: časová značka ve formátu '*RRRR-MM-DD hh:mm:ss*' (obsahuje datum i čas). Vložíte-li do sloupce, kde je nastaven *TIMESTAMP*, NULL, do databáze se uloží aktuální datum a čas ve formátu popsaném výše. Upravujete-li jakýkoli sloupec v řádku, kde je nastaven také *TIMESTAMP*, upraví se i sloupec *TIMESTAMP*, a to na aktuální datum a čas, kdy byla změna provedena.
Atributy: jen globální atributy
Rozsah: 19700101000000 až někdy do roku 2037
Nulová hodnota: 00000000000000
Výchozí hodnota: aktuální datum a čas pro první sloupec *TIMESTAMP* tabulky, 0 pro ostatní sloupce
Místo v paměti: 4 bajty
Poznámka: příkaz *DESCRIBE* a *SHOW COLUMNS* vrací *NULL*, i když prakticky nelze *NULL* do tohoto typu sloupce uložit, nastavíte-li však sloupec jako *NULL*, uloží se do sloupce *TIMESTAMP* aktuální datum a čas; pokud je nastaven atribut *NOT NULL*, ignoruje se

Typy tabulek

Existuje více typů tabulek MySQL, jejichž správná volba má zásadní vliv na výkonnost a rychlost.

Tabulky MyISAM

Typ tabulek v systému MySQL, který nahradil starší typ ISAM. Poprvé se objevil v MySQL 3.23.0. Tyto tabulky nepodporují transakce, ale na druhou stranu jsou velmi rychlé a **typ MyISAM je optimalizován pro dotazy SELECT**.

Datové soubory MyISAM mají přípony .MYD a indexy .MYI. Databáze MyISAM jsou uloženy v určitém adresáři, takže pokud jste si zkoušeli dané příklady, tyto soubory najdete v adresáři ../xampp/mysql/data/prvnidb/

Existují 3 podtypy tabulek MyISAM:

- Statické
- Dynamické
- Komprimované – nebudeme dále rozebírat.

MySQL rozhoduje o tom, zda se má použít dynamická nebo statický tabulka již při jejím vytváření. Statické tabulky představují výchozí formát, který se aplikuje, když se v tabulce nevyskytují žádné sloupce typu VARCHAR, BLOB ani TEXT. Pokud je v ní některý z uvedených typů, stane se tabulka dynamickou.

Statické tabulky

Statické tabulky mají pevnou délku. Podívejme se na tabulku níže, která znázorňuje znaky uložené v jedné statické tabulce. Pole je typu CHAR(10).

J	A	N							
J	I	N	D	Ř	I	CH			
M	A	R	K	É	T	A			

Pro každý záznam je určeno přesně 10 bajtů. Pokud některý údaj obsahuje méně znaků, pak se zbytek sloupce doplní mezerami.

Mezi charakteristické znaky statických tabulek patří:

- Jsou velmi rychlé (protože MySQL „ví“, že další data začínají vždy na n+1 pozici).
- Každý údaj v tabulce má pevně vyhrazené místo a nedochází ke fragmentaci.
- Snadno se ukládají do mezipaměti.

- Vyžadují větší diskový prostor (30 znaků je potřeba pro 3 záznamy, ačkoli skutečná data zabírají jen 17) viz příklad.

Dynamické tabulky

Sloupce v dynamických tabulkách mají různé délky. Pokud vložíme do dynamické tabulky stejná data, uloží se podle níže uvedené tabulky:

J	A	N				
J	I	N	D	Ř	I	CH
M	A	R	K	É	T	A

Třebaže tento formát šetří prostor, je složitější. Každý záznam má určitou hlavičku, jež indikuje jeho délku. **Mezi typické znaky těchto tabulek patří:**

- Všechny řetězcové sloupce jsou dynamické (pokud nejsou menší než 4 bajty – v takovém případě by byl ušetřený prostor zanedbatelný a přidaná složitost by vedla k výkonnostní ztrátě).
- Obvykle zabírají mnohem méně diskového prostoru než tabulky statické.
- Tabulky vyžadují pravidelnou údržbu, aby nedocházelo k fragmentaci. (Kdybyste např. aktualizovali jméno Jan na Honza, pak by se písmena „za“ nemohly objevit v prostoru bezprostředně za písmeny „Hon“, protože tento prostor je již obsazen začátkem dalšího záznamu.
- V případě fragmentovaných sloupců znamená každý nový odkaz dodatečných 6 bajtů a bude mít velikost přinejmenším 20 bajtů (a může mít dokonce i své vlastní odkazy, pokud další aktualizace překročí uvedenou délku).

Tabulky InnoDB

Tabulky InnoDB jsou transakčně bezpečným typem tabulek. V případě tabulky MyISAM je při vkládání zamčená celá tabulka. Pro ten okamžik nelze na tabulce vykonávat žádné příkazy. **InnoDB umožňuje uzamykání na úrovni řádků, takže je uzamčen pouze jeden řádek a nikoli celá tabulka.** (o transakcích později)

Z výkonnostních důvodů by se měly používat tabulky InnoDB, pokud data vykonávají velké množství příkazů INSERT nebo UPDATE, relativně k příkazům SELECT. MyISAM bude lepší volbou, vykonává-li databáze velké počty příkazů SELECT vzhledem k operacím UPDATE nebo INSERT.

Tabulky InnoDB také podporují **omezení cizího klíče**. Pokud jsou definovány vztahy mezi tabulkami, ovladač InnoDB hlídá referenční integritu po provedení příkazu DELETE, INSERT a UPDATE. Je tak

nemožné, aby se záznam v jedné tabulce odkazoval na neexistující záznam v tabulce druhé. Což se u MyISAM při špatném návrhu a špatné programové podpoře může snadno stát.[6]

Nevýhody a omezení InnoDB

Velikost záznamu

Maximální velikost datového záznamu je 8kB. Tento limit neplatí pro sloupce TEXT a BLOB, kde se do databáze uloží jen prvních 512 bajtů. Zbytek se uloží do zvláštních stránek.

Paměťové požadavky

U tabulek InnoDB je vznikají mnohem větší nároky na paměť než u tabulek MyISAM.

Fulltextové indexy

V InnoDB nelze požívat fulltextové indexy.

MyISAM nebo InnoDB?

V jedné databázi je možná i kombinace tabulek MyISAM a InnoDB. Je tedy na zvážení, u které tabulky použít ten či onen ovladač. MyISAM lze doporučit v případě, kdy chceme ušetřit paměťové místo a čas. InnoDB budou lepší v případech, kdy je potřeba transakčních konstrukcí, větší bezpečnost a pokud je pravděpodobnost, že více uživatelů bude najednou provádět změny.

Dočasné tabulky

Jsou typem tabulek, které lze vytvářet dočasně se všemi tabulkovými typy. Tyto tabulky jsou automaticky smazány po odpojení k serveru MySQL. Dočasné tabulky nejsou viditelné ostatním uživatelům.

Dočasné tabulky vlastně nejsou samostatným typem, ale jakousi dočasnou variantou ostatních. Sám server MySQL je vytváří skrytě jako pomocné články při komplikovaných dotazech SELECT. Dočasné tabulky jsou ukládány do adresáře pro dočasné soubory. Dočasnou tabulku vytvoříme přidáním klíčového slova **TEMPORARY**:

```
CREATE TEMPORARY TABLE nazev_tabulky (…) ENGINE = typ_tabulky;
```

Na tabulce můžeme klasicky používat příkazy INSERT, DELETE, UPDATE. Tabulku odstraníme příkazem **DROP**.

[6] Problematika cizích klíčů a jejich nastavení není v této studijní opoře zatím zakomponována. Více informací se dozvíte na semináři nebo např. zde: https://dev.mysql.com/doc/refman/5.7/en/create-table-foreign-keys.html

Tabulky MRG_MYISAM

Tento typ je jakási nadstavba nad typ MyISAM a vznikne sloučením identických MyISAM. Slouží k rozložení zátěže tím, že data rozdělíme do více identických tabulek a každou můžeme např. umístit na jiný pevný disk. Každá tato tabulka má pak svá vlastní data a ke všem najednou se přistupuje pod jejich společným a*liasem.* Nebo můžeme samozřejmě obejít MERGE tabulku a pracovat s jednotlivými tabulkami přímo. Tento typ také může řešit problém s limitem velikosti souborů v některých operačních systémech. Nevýhodou jsou např. nemožnost použití FULLTEXT indexu nad všemi tabulkami.

- Tabulky MERGE musí používat uložiště MyISAM, takže nejsou bezpečné pro transakce.
- Všechny podtabulky musí mít stejnou strukturu.
- Protože jsou informace rozděleny do podtabulek, existují situace, kdy musí databáze MySQL při dotazu provádět značné zpracování indexů.

Ukázkově si vytvoříme dvě shodné tabulky, na nichž poté vytvoříme tabulku **MERGE**:

```
CREATE TABLE `prvnidb`.`prodejce_1` (
`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`cislo` INT( 1 ) UNSIGNED NOT NULL ,
`jmeno` VARCHAR( 20 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL ,
`prijmeni` VARCHAR( 40 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL

) ENGINE = MYISAM CHARACTER SET utf8 COLLATE utf8_czech_ci;
```

```
CREATE TABLE `prvnidb`.`prodejce_2` (
`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`cislo` INT( 1 ) UNSIGNED NOT NULL ,
`jmeno` VARCHAR( 20 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL ,
`prijmeni` VARCHAR( 40 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL

) ENGINE = MYISAM CHARACTER SET utf8 COLLATE utf8_czech_ci;
```

Dále vytvoříme tabulku MERGE:

```
CREATE TABLE `prvnidb`.`prodejce1_2` (
`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`cislo` INT( 1 ) UNSIGNED NOT NULL ,
`jmeno` VARCHAR( 20 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL ,
`prijmeni` VARCHAR( 40 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL

) ENGINE = MERGE UNION(prodejce_1, prodejce_2);
```

Nyní do tabulek vložíme nějaká data:

```
INSERT INTO prodejce_1 (id, cislo, jmeno, prijmeni) VALUES (NULL, 198,
'Alena', 'Morávková');
INSERT INTO prodejce_2 (id, cislo, jmeno, prijmeni) VALUES (NULL,
19887, 'Andrea', 'Kroupová');
```

Pokud se nyní dotážeme sloučené tabulky, budeme mít k dispozici všechny sloučené záznamy z tabulek prodejce_1 a prodejce_2.

```
SELECT jmeno, prijmeni FROM prodejce1_2;
```

jmeno	prijmeni
Alena	Morávková
Andrea	Kroupová

Na základě tohoto výsledku nevíme, ve které z používaných tabulek se záznamy nacházejí. Avšak jak pro aktualizaci, tak pro odstranění záznamu na tom nezáleží. Vyzkoušejme:

```
UPDATE prodejce1_2 SET prijmeni = 'Morávek' WHERE cislo = 198;
DELETE FROM prodejce1_2 WHERE cislo = 19887;
```

Tabulky MEMORY (HEAP)

Zvláštností tohoto typu je, že struktura tabulky je uložena běžným způsobem v souborech na disku, ale samotná data se uchovávají pouze v paměti. Z toho plyne, že slouží pouze k dočasnému uchovávání mezivýsledků, cache apod. **Co se týče rychlosti, jednoznačně vítězí**. Ale protože jsou data jen v paměti, můžeme o ně vypnutím či pádem systému přijít. Je zde však více omezení:

- Nelze použít sloupce typu BLOB ani TEXT, protože data tabulky jsou uložena vždy v pevném formátu (statickém).
- Pokud není správně nastaven parametr max_heap_table_size, vystavujeme se riziku spotřebování příliš velkého množství paměti. Velikost tabulky MEMORY můžete také nastavovat po mocí volby MAX_ROWS při vytváření tabulky.

Pro příklad použití si můžeme představit menší online chat, který umožňuje komunikaci zákazníků s operátory.

```
CREATE TABLE `prvnidb`.`konverzace` (
`id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`start` DATETIME NOT NULL ,
`konec` DATETIME NOT NULL
) ENGINE = MEMORY;
```

```
CREATE TABLE `prvnidb`.`chat` (
`id` INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`id_konverzace` INT NOT NULL ,
`odesilatel` VARCHAR( 100 ) CHARACTER SET utf8 COLLATE utf8_czech_ci
NOT NULL ,
`text` VARCHAR( 255 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL ,
`datum` DATETIME NOT NULL ,
INDEX ( `id_konverzace` )
) ENGINE = MEMORY;
```

Tabulky ARCHIVE

Podle názvu nám dojde, že se jedná o uložiště archivních dat, ke kterým budeme jen velice zřídka přistupovat a potřebujeme, aby měly co nejmenší objem. Veškerá data jsou komprimována, lze provádět pouze příkazy SELECT a INSERT (data se vkládají na konec), nelze použít DELETE, REPLACE a UPDATE, nepoužívají se žádné indexy (dotaz SELECT má za následek průchod celou tabulkou a dekompresi zkoumaných dat).

Tabulky CSV

Data jsou ukládána v textovém formátu CSV, každý záznam je na jednom řádku, sloupce odděleny čárkami. Nejsou podporovány indexy. Tyto tabulky se nepoužívají typicky pro zvýšení výkonu a mají některá závažná omezení. Podporovány jsou pouze operace **SELECT** a **INSERT**.

Optimalizace tabulek

Mezi nevýhody typů sloupců s proměnlivou délkou (VARCHAR, TEXT, …) patří potenciální riziko snížení výkonu kvůli **fragmentaci**. Bylo popsáno u dynamických tabulek MyISAM. MySQL obsahuje nástroj, který umí tyto tabulky defragmentovat pomocí příkazu **OPTIMIZE TABLE**, který obnovuje efektivnější strukturu a zároveň řadí nezbytné indexy.

```
OPTIMIZE TABLE nazev_tabulky;
```

10 Pokročilý jazyk SQL

Dále se blíže seznámíme s následujícími tématy:

- Logické, aritmetické a porovnávací operátory.
- Proměnné.
- Spojení JOIN.
- Spojování výsledků pomocí UNION.
- Vnořené příkazy SELECT.
- Vykonávání transakcí pomocí BEGIN a COMMIT, zamykání tabulek.

Operátory

Logické operátory

Logické operátory se omezují na pravdu – logická 1, a nepravdu – logická 0.

Operátor	Syntaxe	Popis
AND, &&	P1 AND P2, P1 && P2	Pravda, pouze jsou-li obě podmínky označené jako P1 a P2 pravdivé.
OR, ||	P1 OR P2, P1 || P2	Pravda, pokud je P1 nebo P2 pravdivá, příp. obě pravdivé.
NOT, !	NOT P1, ! P1	Pravda, pokud je P1 nepravda, nepravda, když je P1 pravda.

Příklad

```
SELECT !((1 OR 0) AND (0 OR 1));
```

Výsledek 0.

Pamatujte, že podmínky v nejvnitřnějších závorkách se vyhodnocují nejprve. To znamená, že MySQL zjednodušuje komplexní výraz následujícím způsobem:

```
!((1 OR 0) AND (0 OR 1))

!((1) AND (1))

!(1)

0
```

Aritmetické operátory

Aritmetické operátory se používají k vykonávání základních matematických operací

Operátor	Syntaxe	Popis
+	a + b	Součet
-	a - b	Rozdíl
*	a * b	Součin
/	a / b	Podíl
%	a % b	Celočíselně vydělí a hodnotou b a vrátí zbytek

Porovnávací operátory

Jednoduché části, které jistě znáte, vynecháme (=, !=, <>, >, <, >=, <=). Oproti JS a PHP je rovnost určena pouze jedním rovnítkem.

Operátor	Syntaxe	Popis
IS NULL	a IS NULL	Pravda, pokud a obsahuje hodnotu NULL
IS NOT NULL	a IS NOT NULL	Pravda, pokud a neobsahuje hodnotu NULL
BETWEEN	a BETWEEN b AND c	Pravda, jestliže a je mezi hodnotami b a c, včetně těchto hodnot
NOT BETWEEN	a NOT BETWEEN b AND c	Pravda, jestliže a není mezi hodnotami b a c, včetně těchto hodnot
LIKE	a LIKE b	Pravda, pokud a odpovídá b podle shody vzorů SQL
NOT LIKE	a NOT LIKE b	Pravda, jestliže a neodpovídá b podle shody vzorů SQL
IN	a IN (b1, b2, b3, bn)	Pravda, jestliže a se rovná čemukoli v seznamu.
NOT IN	a NOT IN (b1, b2, b3, bn)	Pravda, jestliže a se nerovná čemukoli v seznamu.

Proměnné a další konstrukce

Proměnné

MySQL umožňuje ukládat jednoduché hodnoty do proměnných, které jsou důležitým prvkem tzv. **uložených procedur**, o kterých později.

Existují tři typy proměnných:

Běžné proměnné: Tyto poznáme podle předpony @. Obsah těchto proměnných se ztrácí po odpojení od serveru MySQL.

Systémové a serverové proměnné: Tyto proměnné ukládají stav a vlastnosti serveru MySQL. Poznáme je podle předpony @@.

Lokální proměnné a parametry uložených procedur: Tyto proměnné jsou uloženy přímo v uložené proceduře a jsou dostupné jen v ní. Nemají žádné předpony, které by je odlišily, ale jejich názvy se musí lišit od názvů tabulek a sloupců.

V této části nás budou zajímat především běžné proměnné.

Načtení hodnoty do proměnné

Existuje více způsobů přiřazení a každý má mírně odlišný syntax:

```
SET @nazevpromenne = 3;

SELECT @nazevpromenne := 3;

SELECT @nazevpromenne := COUNT(pole) FROM tabulka;

SELECT COUNT(pole) FROM tabulka INTO @nazevpromenne;

SELECT jmeno, prijmeni FROM autori WHERE id = … INTO @jmeno,
 @prijmeni;
```

Poslední varianta přiřazení hodnot více sloupců do více proměnných funguje jen v případě, že dotaz SELECT vrací přesně jeden záznam. Je tedy potřeba použít vhodné podmínky **WHERE** či omezení počtu výsledků klauzulí **LIMIT**.

Použití proměnných

Jakmile proměnnou definujeme, můžeme ji použít v libovolném příkazu SQL. V názvech proměnných MySQL od verze 5 nerozlišuje velikost písmen.

Příklad

```
SET @id_knihy = 3;
SELECT nazev FROM knihy WHERE id = @id_knihy;
```

Proměnné lze používat též pro výpočty. Následující příklad počítá, kolik procent z celkového počtu knih zaujímají díla Agathy Christie.

```
SELECT @id_christie := id FROM autori WHERE jmeno = 'Agatha' AND
prijmeni = 'Christie';

SELECT @christie_pocet_knih := COUNT(id) FROM knihy WHERE id_autor
= @id_christie;

SELECT @christie_pocet_knih / COUNT(id) * 100 AS procent FROM
knihy;
```

Funkce IF()

Pomocí funkce **IF** můžeme vrátit jeden ze dvou výsledků podle toho, jak byla vstupní podmínka vyhodnocena:

```
IF(podmínka, splneni_podminky, nesplneni_podminky)
```

Podmínky lze do sebe i vnořovat:

```
IF(podmínka, vysledek1, IF(podmínka2, vysledek2a, vysledek2b))
```

Příklady

```
SELECT IF(1<2, 'podmínka platí', 'podmínka neplatí');

SELECT nazev, IF(ISNULL(isbn), 'nemá ISBN', isbn) AS isbn FROM knihy
 WHERE id = 1;
```

V druhém příkladu jsme použili porovnávací operátor **ISNULL(),** který zjišťuje, zda-li je hodnota v daném sloupci nulová. Ekvivalentním je k tomuto konstrukce **IFNULL:**

```
SELECT nazev, IFNULL(isbn, 'nemá ISBN') AS isbn FROM knihy WHERE id
 = 1;
```

Větvení CASE

Konstrukci **CASE** lze zapsat dvěma způsoby. První z nich vrací výsledek1, jestliže vyraz = hodnota1 a naopak vysledek2, pokud vyraz = hodnota2, atd:

```
CASE vyraz
	WHEN hodnota1 THEN vysledek1
	WHEN hodnota2 THEN vysledek2
```

```
    …
    ELSE vysledekn
END
```

Druhá varianta nevyhodnocuje jen jednu podmínku, ale několik podmínek. Vrací tedy výsledek1, jestliže platí podmínka1, výsledek2, pokud platí podmínka2 atd.

```
CASE
    WHEN podminka1 THEN vysledek1
    WHEN podminka2 THEN vysledek2
    …
    ELSE vysledekn
END
```

Ukážeme si relativně jednoduchý příklad, který využívá vestavěných funkcí MySQL pro práci s řetězci, kterých je celá řada a můžete je prozkoumat na adrese uvedené v zápatí této stránky[7].

Příklad

```
SELECT
CASE CHAR_LENGTH(nazev) > 20
    WHEN TRUE THEN LEFT(INSERT(nazev, 21, 1, '…'), 21)
    WHEN FALSE THEN INSERT(nazev, CHAR_LENGTH(nazev) + 1, 1,
'…')
END AS nazev_knihy
FROM knihy;
```

Funkce **CHAR_LENGTH()** jak její název napovídá, vrací počet znaků hodnoty v daném sloupci. V případě, že počet znaků je větší než 20 (pravda TRUE) funkce **LEFT()** ořízne zleva řetězec na 21 znaků. Zároveň funkce **INSERT()** (neplést s klauzulí INSERT pro vložení záznamu) nahradí 21 znak třemi tečkami.

Pokud řetězec není delší než 20 znaků, funkce INSERT() vloží tři tečky ihned za poslední znak názvu knihy, jehož pozici jsme zjistili funkcí **CHAR_LENGTH().**

[7] http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_left

11 Spojení JOIN

Se základním spojením tabulek jsme se seznámili již dříve. Spojení však mohou být mnohem komplikovanější a špatně napsaná spojení mohou být strůjcem výkonnostních potíží.

Pro následné procvičování byste měli mít vytvořeny tyto tabulky:

Tabulka **autori**:

id	jmeno	prijmeni
1	Agatha	Christie
2	Dan	Brown
3	Andrew	Oppel

Tabulka **knihy**, kterou doplníme o nový řádek:

id	id_autor	nazev	isbn
1	1	Vražda v Orient-Expressu	978-80-242-2097-0
2	1	Oznamuje se vražda	978-80-242-2161-8
3	2	Andělé a démoni	80-7203-787-0
4	2	Šifra mistra Leonarda	80-86518-62-0
5	3	Databáze bez předchozích znalostí	80-251-1199-7
6	1	Pozvánka pro Hercula Poirota	NULL

```
INSERT INTO knihy (id, id_autor, nazev, isbn) VALUES('', 1,
 'Pozvánka pro Hercula Poirota', NULL);
```

A třetí, nová tabulka **stav**, která bude monitorovat, zda je kniha zapůjčena či nikoliv:

Id_knihy	zapujceno
1	1
2	1
3	0
4	1
5	0
6	0
7	1

SQL příkaz:

```
CREATE TABLE `prvnidb`.`stav` (
`id_knihy` INT( 9 ) NOT NULL ,
`zapujceno` TINYINT( 1 ) NOT NULL ,
PRIMARY KEY ( `id_knihy` )

) ENGINE = MYISAM;
```

```
INSERT INTO stav (id_knihy, zapujceno) VALUES (1,1);
INSERT INTO stav (id_knihy, zapujceno) VALUES (2,1);
INSERT INTO stav (id_knihy, zapujceno) VALUES (3,0);
INSERT INTO stav (id_knihy, zapujceno) VALUES (4,1);
INSERT INTO stav (id_knihy, zapujceno) VALUES (5,0);
INSERT INTO stav (id_knihy, zapujceno) VALUES (6,0);
INSERT INTO stav (id_knihy, zapujceno) VALUES (7,1);
```

Typy spojení

Existuje více typů spojení JOIN. Jejich syntaxe je podobná, ale přece se v detailech a funkčnosti něčím liší. Začneme krátkým přehledem:

	Syntax zápisu	U všech variant můžeme podmiňovat pomocí WHERE
1	FROM tab1, tab2 WHERE tab1.xx = tab2.yy	
2	FROM tab1 JOIN tab2 ON tab1.xx = tab2.yy	
3	FROM tab1 CROSS JOIN tab2 ON tab1.xx = tab2.yy	Ekvivalentní předchozímu
4	FROM tab1 INNER JOIN tab2 ON tab1.xx = tab2.yy	Ekvivalentní předchozímu
5	FROM tab1 STRAIGHT_JOIN tab2	*Není standardem SQL*
6	FROM tab1 LEFT JOIN tab2 ON tab1.xx = tab2.yy	
7	FROM tab1 LEFT JOIN tab2 USING xy	
8	FROM tab1 NATURAL JOIN tab2	

U spojení 1 až 4 se MySQL snaží sám najít správné pořadí pro přístup k datům. STRAIGHT_JOIN má tu výhodu, že MySQL přistupuje k tabulkám v tom pořadí, v jakém jsou uvedeny.

Varianty 6 a 7 pro každý záznam z první tabulky generují záznam ve výsledku, a to i v případě, kdy v druhé tabulce neexistuje odpovídající hodnota. V tom případě bude ve výsledku hodnota NULL. To není případ prvních 4 variant a spojení NATURAL JOIN, které zobrazí vždy jen záznamy mající odpovídající hodnotu spojení i v druhé tabulce.

Vnitřní spojení

INNER JOIN

Tento typ spojení vrací pouze ty záznamy, jež mají odpovídající záznam v obou tabulkách. Je to nejběžnější typ spojení. Klíčové slovo INNER není povinné, ale zvykneme si jej uvádět.

```
SELECT nazev, jmeno, prijmeni
FROM knihy
INNER JOIN autori ON knihy.id_autor = autori.id;
```

nazev	jmeno	prijmeni
Vražda v Orient-Expressu	Agatha	Christie
Oznamuje se vražda	Agatha	Christie
Andělé a démoni	Dan	Brown
Šifra mistra Leonarda	Dan	Brown
Databáze bez předchozích znalostí	Andrew	Oppel
Pozvánka pro Hercula Poirota	Agatha	Christie

První dva řádky jsou jasné. Vybíráme sloupce, které nás zajímají a za klauzulí FROM název první spojované tabulky. Poté vnitřním spojením připojujeme druhou spojovanou tabulku a pomocí klíčového slova **ON** označíme samotné spojení.

Nyní bychom chtěli vypsat knihy, které jsou zapůjčeny a nejsou tedy k dispozici:

```
SELECT nazev
FROM knihy
INNER JOIN stav ON knihy.id = stav.id_knihy
WHERE zapujceno = 1;
```

nazev
Vražda v Orient-Expressu
Oznamuje se vražda
Šifra mistra Leonarda

Všimněte si, že v tabulce **stav**, je záznam (id_knihy = 7), který nemá odpovídající záznam v tabulce **knihy**. Proto nebude ve vnitřním spojení akceptován.

Vnější spojení

OUTER JOIN

Vnější spojení vrací všechny záznamy. MySQL podporuje stabilně dva druhy vnějších spojení:

Levé vnější spojení (LEFT OUTER JOIN) – vrací všechny odpovídající záznamy a všechny další záznamy z levé tabulky - z tabulky nalevo od klíčového slova JOIN.

Pravé vnější spojení (RIGHT OUTER JOIN) – vrací všechny odpovídající záznamy a všechny další záznamy z pravé tabulky - z tabulky napravo od klíčového slova JOIN.

---

*Vzhledem ke kompatibilitě si zvykneme používat pouze **levé vnější spojení**. Usnadní nám to přechod např. na databáze MSSQL nebo jiné databáze, kde RIGHT JOIN není podporován.*

Levé (vnější) spojení (LEFT JOIN), slovo OUTER není povinné, nám tedy dobře poslouží v případě, kdy v druhé tabulce není odpovídající záznam spojení. Uvedeme příklad:

Zakoupili jsme novou knihu, ale ještě předtím než jsme ji zaevidovali si ji někdo vypůjčil. Tedy tato kniha nemá odpovídající záznam v tabulce **Knihy**. V předchozím příkladě jsme se o ní tedy nemohli nic dozvědět. „Pomůže“ nám levé vnější spojení:

```
SELECT id_knihy, nazev
FROM stav
LEFT JOIN knihy ON id_knihy = id
WHERE zapujceno = 1;
```

id_knihy	nazev
1	Vražda v Orient-Expressu
2	Oznamuje se vražda
4	Šifra mistra Leonarda
7	NULL

Tabulka nalevo od klauzule LEFT JOIN (tabulka stav), vrací všechny odpovídající záznamy (1,2,4) a všechny další záznamy (7). Protože v druhé tabulce neexistuje příslušný záznam, doplní se hodnotou NULL.

POZOR

Kdybychom pořadí tabulek obrátili, dojdeme ke stejnému výsledku jako v případě vnitřního spojení:

```
SELECT id_knihy, nazev
FROM knihy
LEFT JOIN stav ON id_knihy = id
```

```
WHERE zapujceno = 1;
```

id_knihy	nazev
1	Vražda v Orient-Expressu
2	Oznamuje se vražda
4	Šifra mistra Leonarda

Tabulka nalevo je nyní **knihy** a v ní logicky žádný neodpovídající záznam není.

Vyzkoušejme nyní něco komplikovanějšího. Chceme vypsat všechny nevypůjčené knihy, včetně celého jména autora. Budeme tedy potřebovat data ze všech tří tabulek a příkaz bude vypadat následovně:

```
SELECT nazev, CONCAT(jmeno, ' ', prijmeni) AS cele_jmeno
FROM knihy
LEFT JOIN autori ON autori.id = knihy.id_autor
      LEFT JOIN stav ON stav.id_knihy = knihy.id
WHERE stav.zapujceno = 0;
```

nazev	cele_jmeno
Andělé a démoni	Dan Brown
Databáze bez předchozích znalostí	Andrew Oppel
Pozvánka pro Hercula Poirota	Agatha Christie

Příkaz není vůbec složitý. Je z něj patrné, že můžeme v jednom sledu kombinovat více spojení najednou. Popišme si jednotlivé kroky pěkně postupně[8]:

1. Za příkazem SELECT jsme vybrali kýžené sloupce. Potřebujeme znát název z tabulky knihy a celé jméno autora z tabulky autoři.

2. Jako první vybíráme tabulku **knihy**. (mohli bychom i obráceně využít jako první tabulku autoři)

3. Pomocí levého vnějšího spojení připojujeme tabulku autoři a spojujeme na sloupcích autori.id a knihy.id_autor. Zde byste si mohli představit, že vlastně máte určitou dočasnou tabulku, která vypadá následovně:

nazev	cele_jmeno
Vražda v Orient-Expressu	Agatha Christie
Oznamuje se vražda	Agatha Christie
Andělé a démoni	Dan Brown
Šifra mistra Leonarda	Dan Brown

[8] Optimalizátor MySQL v dotazu volí svůj vlastní sled výběru.

Databáze bez předchozích znalostí	Andrew Oppel
Pozvánka pro Hercula Poirota	Agatha Christie

4. Nyní tuto myšlenou tabulku propojíme z opravdovou tabulkou stav, dalším levým spojením na sloupcích stav.id_knihy a knihy.id. U každého díla si tak myšlenkově můžete představit 1 nebo 0, která reprezentuje, zda je dílo volné či zapůjčené.

5. Posledním je podmínka WHERE, která nám prostě jen filtruje volná díla.

Pokud si dovedete takto myšlenkově jednotlivé kroky rozložit, nebude pro vás komplikované spojení představovat žádný problém.

Pro uvedený příklad byste mohli použít i vnitřní spojení, protože tabulky jsou v příkazu seřazeny tak, že jednotlivá spojení mají ve všech tabulkách odpovídající záznamy. Čili správně by fungoval i následující zápis příkazu:

```
SELECT nazev, CONCAT(jmeno, ' ', prijmeni) AS cele_jmeno
FROM knihy
INNER JOIN autori ON autori.id = knihy.id_autor
     INNER JOIN stav ON stav.id_knihy = knihy.id
WHERE stav.zapujceno = 0;
```

Přirozená spojení

NATURAL JOIN

Pole **id** v tabulce autoři a pole **id_autor** v tabulce knihy spolu navzájem souvisejí. Pokud bychom jim přiřadili *shodný název*, mohli bychom využívat zkratkových zápisů SQL, které omezují množství zadání u konstrukce JOIN. Přejmenujme tedy pro tuto chvíli autori.id na autori.id_autor:

```
ALTER TABLE `autori` CHANGE `id` `id_autor` INT(9) NOT NULL
AUTO_INCREMENT;
```

id_autor	jmeno	prijmeni
1	Agatha	Christie
2	Dan	Brown
3	Andrew	Oppel

Protože nyní mají naše dvě tabulky pole se shodnými názvy, můžeme vykonat NATURAL JOIN, které hledá shodně nazvaná pole, na nichž vykoná spojení:

```
SELECT nazev, CONCAT(jmeno, ' ', prijmeni) AS cele_jmeno
```

```
FROM knihy NATURAL JOIN autori;
```

Pokud bychom opět chtěli získat pouze volné knihy, stačí přidat další 2 řádky kódu:

```
SELECT nazev, CONCAT(jmeno, ' ', prijmeni) AS cele_jmeno
FROM knihy NATURAL JOIN autori
INNER JOIN stav ON stav.id_knihy = knihy.id
WHERE stav.zapujceno = 0;
```

nazev	cele_jmeno
Andělé a démoni	Dan Brown
Databáze bez předchozích znalostí	Andrew Oppel
Pozvánka pro Hercula Poirota	Agatha Christie

Klíčové slovo USING

Klíčové slovo **USING** nabízí trochu větší možnost řízení než NATURAL JOIN. Pokud ve dvou tabulkách existuje více shodných polí, toto klíčové slovo vám dovoluje zadat, která z těchto polí se použijí jako spojovací podmínky. Máme-li kupříkladu tabulky alfa a beta s identickými poli a, b, c, d, pak jsou následující podmínky ekvivalentní:

```
SELECT * FROM alfa LEFT JOIN beta USING (a,b,c,d)
```

```
SELECT * FROM alfa NATURAL JOIN beta
```

Klíčové slovo USING nabízí větší pružnost, protože nám dovoluje použít jen některé ze čtyř shodných polí. Například:

```
SELECT * FROM alfa LEFT JOIN beta USING (a,c)
```

Sjednocování výsledků pomocí UNION

UNION je klauzule, která umožňuje kombinaci výsledků různých příkazů SELECT do jednoho. Každý příkaz musí mít stejný počet sloupců.

Pro potřeby ukázky tohoto příkazu si vytvořme následující tabulku:

```
CREATE TABLE `prvniDB`.`autori2` (
`id` INT( 9 ) NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`jmeno` VARCHAR( 20 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL ,
`prijmeni` VARCHAR( 40 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL

) ENGINE = MYISAM CHARACTER SET utf8 COLLATE utf8_czech_ci;
```

```
INSERT INTO autori2 (id, jmeno, prijmeni) VALUES ('', 'Jules',
'Verne');

INSERT INTO autori2 (id, jmeno, prijmeni) VALUES ('', 'Jiří',
'Kosek');
```

id	jmeno	prijmeni
1	Jules	Verne
2	Jiří	Kosek

Nyní vypíšeme seznam všech autorů z obou tabulek v jednom příkazu:

```
SELECT CONCAT(jmeno, ' ', prijmeni) AS cele_jmeno_autora FROM
 autori
UNION
SELECT CONCAT(jmeno, ' ', prijmeni) AS cele_jmeno_autora FROM
 autori2;
```

cele_jmeno_autora
Agatha Christie
Dan Brown
Andrew Oppel
Jules Verne
Jiří Kosek

Pro zájemce - Spojení STRAIGHT_JOIN

Se spojením typu **STRAIGHT_JOIN** se v praxi moc nesetkáváme, nicméně pro určité druhy dotazů je z hlediska optimalizace směrem k lepšímu výkonu velmi užitečným typem spojení.

Spojíme-li typem JOIN (INNER) **n** množství tabulek, optimalizátor MySQL si sám určí pořadí, v jakém bude tabulky spojovat a automaticky volí tak, aby spojení bylo co nejefektivnější. Ovšem nemusí tomu tak být vždy.

Při zadání **STRAIGHT_JOIN** před seznam vybíraných sloupců nebo přímo jako forma spojení bude optimalizátor nucen spojit tabulky v tom pořadí, v jaké jsou v rámci dotazu uvedeny.

Možnosti zápisu – můžeme použít STRAIGHT_JOIN jako spojovací konstrukci:

```
SELECT nazev
FROM knihy
STRAIGHT_JOIN stav ON id = id_knihy
WHERE zapujceno = 1;
```

Toto je ideální pořadí, protože MySQL nejprve prohledává tabulku **knihy**. Čte řádek po řádku a pak provádí spojení s tabulkou **stav** pomocí jejího primárního klíče *id_knihy* a zároveň filtruje záznamy podmínkou WHERE.

Možnosti zápisu – můžeme použít STRAIGHT_JOIN jako prvek určení pořadí:

```
SELECT STRAIGHT_JOIN nazev
FROM stav
INNER JOIN knihy ON id = id_knihy
WHERE zapujceno = 1;
```

Tato možnost použití STRAIGHT_JOIN předepisuje dotazu s vnitřním spojením dodržet určené pořadí spojení tabulek. Výše uvedený dotaz je ale příkladem neefektivního použití (ne z hlediska syntaxe, ale z hlediska nesprávného pořadí spojení tabulek).

MySQL nejprve prochází tabulku **stav**. Čte řádek po řádku a užívá podmínku za WHERE. Poté prochází celou tabulku **knihy.** Provádí spojení a opět užívá podmínku WHERE. Systém není schopen užít primárního klíče a dotaz je komplikovanější.

Doplněk

S výkonnostní problematikou dotazů MySQL úzce souvisí příkaz **EXPLAIN**. Ten popisuje detailně to, co jsem popisoval v předchozím odstavci. Tedy jak optimalizátor MySQL přistupuje k jednotlivým tabulkám, jak využívá klíčů atd. Pokud bychom chtěli analyzovat nějaký dotaz, stačí příkaz EXPLAIN použít před zkoumaným dotazem.

Porovnejte následující výsledky:

```
EXPLAIN
SELECT nazev
FROM knihy
STRAIGHT_JOIN stav ON id = id_knihy
WHERE zapujceno = 1;
```

```
EXPLAIN
SELECT STRAIGHT_JOIN nazev
FROM stav
INNER JOIN knihy ON id = id_knihy
WHERE zapujceno = 1;
```

Co znamenají jednotlivé sloupce tabulky, která se vám zobrazí po vyhodnocení příkazem EXPLAIN je vaším samostatným úkolem. Já jej zde dále popisovat nebudu. Vyhodnocení příkazu EXPLAIN

bude součástí závěrečného zkoušení! Proto si jej z příbuzné literatury či zdrojů na internetu pečlivě nastudujte. Příkaz EXPLAIN bude rovněž vysvětlen v rámci prezenčních přednášek!

12 Vnořené dotazy SELECT

Od verze 4.1 podporuje MySQL použití vnořených příkazů SELECT. Existuje několik způsobů, jak formulovat vnořené dotazy. Ukážeme si základní čtyři možnosti:

1. možnost zápisu

SELECT … WHERE pole = [ANY/ALL] (SELECT…)

U tohoto způsobu musí druhý dotaz vracet jednoduchou hodnotu (průsečík jeden řádek/jeden sloupec). Tato hodnota se použije pro porovnávání pole = …

Pro porovnávání můžeme použít klíčová slova ANY, SOME a ALL. V tom případě může druhý dotaz SELECT vracet více než jednu hodnotu. ANY/SOME znamená, že se budou brát v úvahu všechny vhodné hodnoty. Dotaz pak může vrátit několik výsledků. Výraz pole = ANY… odpovídá výrazu pole IN…

Příklad

```
SELECT prijmeni FROM autori
WHERE id = (SELECT id_autor FROM knihy WHERE isbn = '80-7203-787-
0');
```

Příklad

```
SELECT prijmeni FROM autori
WHERE id > ALL(SELECT id_autor FROM knihy);
```

2. možnost zápisu

SELECT … WHERE pole [NOT] IN (SELECT …)

V tomto zápisu může druhý dotaz SELECT vracet několik hodnot. Seznam je potom zpracován ve tvaru SELECT … WHERE pole IN (n1,n2,n3). Místo IN můžeme také použít NOT IN.

Příklad

```
SELECT prijmeni FROM autori
WHERE id NOT IN (SELECT id_autor FROM knihy);
```

3. možnost zápisu

SELECT … WHERE [NOT] EXISTS (SELECT …)

V tomto případě je druhý dotaz spouštěn pro každý záznam, který nalezl první dotaz SELECT. Jenom v případě, že druhý dotaz vrací nějaký výsledek (alespoň jeden záznam), zůstane záznam z výsledku prvního dotazu ve výsledném seznamu. U této varianty můžeme použít také operaci NOT.

Příklad

```
SELECT prijmeni FROM autori
WHERE EXISTS (SELECT id FROM knihy WHERE knihy.id_autor =
autori.id);
```

Jistě vás v tento moment napadne, že mnohem elegantněji by tento dotaz šel vyřešit pomocí spojení INNER JOIN. Nicméně i tento vnořený příkaz najde své opodstatnění.

4. možnost zápisu

SELECT … FROM (SELECT …) AS nazev WHERE …

U tohoto způsobu je nejprve spuštěn dotaz SELECT v závorce. Ten vrací tabulku, která slouží jako základ vnějšího dotazu SELECT. Jinak řečeno, vnější dotaz nepřistupuje k běžné tabulce, ale k tabulce, která je výsledkem jiného dotazu. Takové tabulky se nazývají **odvozené**. Syntaxe SQL stanovuje, že tuto tabulku musíme nazvat pomocí klauzule AS.

Příklad

```
SELECT nazev FROM (SELECT nazev FROM knihy INNER JOIN stav ON
 stav.id_knihy = knihy.id  WHERE stav.zapujceno = 1) AS zapujcene;
```

Ve většině příkazů lze vnořené dotazy nahradit spojením JOIN, které je z hlediska návrhu efektivnější a jeho syntax přehlednější. Ovšem existují situace, které použití vnořených dotazů vyžadují. Dotazy SELECT lze také vnořovat do podmínek WHERE příkazů UPDATE a DELETE.

Vyzkoušejme následující příklad:

```
DELETE FROM stav WHERE id_knihy NOT IN (SELECT id FROM knihy);
```

13 Zamykání a transkace

Zamykání

MySQL je databázový systém klient/server. Je tedy jasné, že více klientů může v současnou chvíli pracovat s týmiž daty. Odesílat dotazy, vkládat data, aktualizovat nebo mazat záznamy. A samozřejmě může nastat situace, kdy dva uživatelé ve stejný moment chtějí měnit týž záznam. Je tedy potřeba být na tuto situaci připraven.

Příkladem může být dotaz dvou dealerů automobilů do skladu na určitý typ vozu, zda-li je k dispozici ten či onen model. Oba dealeři obdrží ve stejný moment informace, že ano, ikdyž je k dispozici pouze jeden vůz. Oba dealeři pak objednají jeden vůz. Jestliže je databázový systém na tuto situaci dobře připraven, jeden dealer vůz úspěšně objedná, druhý má ale smůlu.

Databázové dotazy se provádějí jeden po druhém. V případě jednoduchých webových aplikací nezáleží na tom, zda se na jedné stránce nejprve zobrazí seznam jmen, či telefonních čísel. Důležitá je rychlost zobrazení. Ovšem určité druhy dotazů se musejí vykonat v konkrétním pořadí.

Dotazy, které závisejí na výsledcích předchozího dotazu, nebo skupiny aktualizací, jež je zapotřebí vykonat jako celek musí být ošetřeny speciální konstrukcí. Všechny typy tabulek mohou využívat **zamykání**, avšak pouze tabulky **InnoDB** mají zabudované transakční schopnosti.

Tabulky MySQL poskytují následující úrovně zámků:

Zámky tabulek

Jsou podporovány MyISAM, MEMORY a InnoDB, a omezují přístup k celé tabulce. Jejich dopad může být částečně regulován klauzulemi LOCAL a LOW_PRIORITY u zámků READ a WRITE.

Zámky řádků

Zamykají konkrétní záznam a umožňují přístup a úpravy všech ostatních řádků tabulky. Tento druh zámků poskytují pouze tabulky InnoDB (můžeme pro ně použít i zámky tabulek).

Zámky tabulek

Existují 2 druhy zámků tabulek:

- zámky čtení - READ
- zámky zápisu - WRITE.

Zámky čtení (READ) zaznamenají, že v tabulce lze vykonávat pouze operace čtení a zápis je uzamčen. Zámek uvolníme příkazem UNLOCK TABLES. READ LOCK můžeme použít pouze tehdy, když tabulka není uzamčena žádným zámkem WRITE LOCK.

Lokální zámky čtení **(READ LOCAL)** mají proti svému předchůdci tu možnost, že příkazy INSERT jsou dovoleny, pokud však nezmění žádné existující záznamy.

Zámky zápisu (WRITE) zaznamenají, že na tabulce nelze vykonávat žádné operace čtení ani zápisu po dobu existence zámku. Zadání direktivy (WRITE) LOW_PRIORITY sdělujeme MySQL, aby před aplikováním zámku WRITE čekal na dokončení všech ostatních činností, které si vyžádaly zámky READ.

Syntaxe uzamčení tabulky vypadá následovně:

```
LOCK TABLE nazev_tabulky {READ|WRITE}
```

Chceme-li tabulku odemknout, použijeme příkaz **UNLOCK TABLES**.

Zamykání bychom měli použít tehdy, když spouštíme několik na sobě závislých příkazů, a nechceme, aby zpracovávaná data ovlivnil jiný uživatel. To je běžný případ, kdy nejprve vybereme dotazem SELECT data, která pak chceme zpracovat příkazy UPDATE nebo DELETE.

Vyzkoušíme si snadný příklad:

Zamkneme tabulku **autori** zámkem pro čtení a následně se pokusíme vložit nový záznam. Příkaz je následující.

```
LOCK TABLE autori READ;
INSERT INTO autori VALUES(NULL, 'Karel', 'May');
```

Ve výsledku MySQL vrátí následující hlášení:

```
Table 'autori' was locked with a READ lock and can't be updated.
```

Pokud tedy budeme dále potřebovat záznam vložit, musíme tabulku před vložením odemknout:

```
LOCK TABLE autori READ;
UNLOCK TABLES;
INSERT INTO autori VALUES(NULL, 'Karel', 'May');
```

Zámky zápisu mají větší prioritu než zámky čtení. Pokud jeden uživatel aplikace čeká na zámek čtení a druhý požaduje zámek zápisu, musí zámek čtení čekat, dokud nebude obdržen a následně uvolněn druhý zámek zápisu.

Zámky tabulek se používají na transakčně nezabezpečených tabulkách a mnohdy je nejrozumnější se jim úplně vyhnout. V případě zámku zápisu nelze z tabulek číst, což je např. u webových aplikací používajících MySQL velmi bolestné.

Používat se by se měly tzv. inkrementální aktualizace. Např.:

```
UPDATE provize SET hodnota = hodnota + 300 WHERE id = 1;

UPDATE provize SET hodnota = hodnota + 100 WHERE id = 1;
```

Nezáleží na tom, v jakém pořadí jsou uvedené příkazy vykonány, protože aktualizovat se bude vždy aktuální hodnota.

Transakce

Síla tabulek InnoDB pramení z používání **transakcí** neboli příkazů SQL, které jsou seskupeny do jednoho celku. Typickým příkladem je bankovní transakce – např. převod peněz z jednoho účtu na druhý.

Obvykle se vykonají 2 příkazy:

```
UPDATE ucet1 SET stav = stav - prevadene_mnozstvi;

UPDATE ucet2 SET stav = stav + prevadene_mnozstvi;
```

Pokud by tyto příkazy proběhly nerušeně v pořádku, není o čem debatovat. Ovšem v případě kdy po vykonání prvního příkazu dojde k selhání, nastává problém. Peníze zmizely z účtu 1, ale nepřibyly na účtu 2. V takovém případě je zásadní, aby byly oba příkazy vykonány společně, nebo nebyl vykonán ani jeden. Za tímto účelem se zabalí oba příkazy dohromady do mechanismu, který se označuje za **transakci**.

Příkaz **BEGIN (nověji START TRANSACTION)** označuje začátek takové transakce a příkaz **COMMIT** zase její konec. Pouze po zpracování příkazem COMMIT se všechny dotazy stanou trvalými.

Vytvoříme si příslušnou tabulku, na níž transakce vyzkoušíme:

```
CREATE TABLE `prvnidb`.`innotest` (
`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`hodnota` CHAR( 10 ) NOT NULL

) ENGINE = InnoDB CHARACTER SET utf8 COLLATE utf8_czech_ci;
```

Nyní se pokusíme o naši první transakci. Obalíme příkaz do START TRANSACTION/COMMIT. Vzhledem ke skutečnosti, že PHPMyAdmin je webová aplikace a protokol http je bezestavový, musíte zkoušet příkazy z této a následující části pohromadě. Nebo si vše můžete rovnou zkoušet v řádkovém klientu **mysql.exe**:

```
START TRANSACTION;
INSERT INTO innotest (id, hodnota) VALUES (NULL , 'test1');
COMMIT;
```

Pakliže proběhlo vše v pořádku, následující příkaz vrátí níže uvedený záznam.

```
SELECT * FROM innotest;
```

id	hodnota
1	test1

Nyní si ukážeme případ, kdy transakce nebude moci proběhnout kvůli programátorské chybě. Budeme chtít vložit dva nové záznamy, přičemž v druhém příkazu záměrně vytvoříme chybu.

```
START TRANSACTION;
INSERT INTO innotest (id, hodnota) VALUES ('', 'test2');
INSERT INTO innotest (id, hodnta) VALUES ('', 'test3');
COMMIT;
```

Výsledkem bude chybové hlášení. Ovšem pozor, nedošlo ani ke vložení prvního záznamu, jehož příkaz je uveden správně.

Pokud bychom nepoužili transakci a neobalili příkazy do **START TRANSACTION/COMMIT**, došlo by ke vložení dat z prvního příkazu a výsledek by byl následující.

id	hodnota
1	test1
2	test2

Důležité vlastnosti transakcí

Příkazy se v transakci provedou vždy jako celek – všechny, nebo žádný. A to i v případě, že vypadne elektrický proud nebo nastane kritická chyba serveru, či přesušení datového spojení. Transakce také zajišťují, že data nemohou měnit dva uživatelé současně. U tabulek MyISAM toho lze docílit zámky celé tabulky, kdežto u InnoDB jsou blokované jen jednotlivé záznamy.

Transakce v MySQL splňují tzv. ACID pravidla:

Atomicita (Atomicity)

Atomicita znamená, že transakce je nedělitelná. Buď se provedou všechny operace nebo ani jedna. To musí platit za všech okolností.

Konzistence (Consistency)

Po provedení transakce musí být databáze v konzistentním stavu. Kdyby transakce měla porušit referenční integritu, potom musí být zrušena (ROLLBACK).

Izolace (Isolation)

Několik transakcí může být zpracováno současně bez toho, aby se vzájemně ovlivnily. To znamená, že každá transakce vidí databázi ve stejném stavu jako na začátku provádění transakce (kromě změn, které sama provedla). Jestliže nějaká transakce vkládá, mění nebo maže záznamy, druhá souběžná transakce tyto změny nevidí.

V tomto smyslu existují tzv. čtyři stupně izolace o kterých se zmíníme později.

Trvanlivost (Durability)

Transakce musí být chráněna před výpadky dokonce i chvíli po svém vlastním provedení. Tabulkový ovladač InnoDB změny zaznamenává do zvláštního souboru. Pokud by došlo k výpadku ještě předtím, než se změny zaznamenají do databáze, budou uloženy při dalším spuštění MySQL.

Příkazy transakcí

Příkaz **START TRANSACTION** (nebo BEGIN – také funguje, ale brzo již podporován nebude) zahájí transakci. Příkaz **COMMIT** ji zase ukončí a uloží všechny změny. Jestliže se v průběhu zpracování stane něco neočekávaného, můžeme použít příkaz **ROLLBACK** k vrácení nedokončené transakce.

Nastavení automatického potvrzování

MySQL běžně pracuje v režimu automatického potvrzování. Automatické potvrzování deaktivujeme příkazem

```
SET AUTOCOMMIT = 0;
```

Poté všechny příkazy na tabulkách, které podporují transakce, budou automaticky považovány za transakci, dokud nejsou ukončeny pomocí COMMIT nebo ROLLBACK. Automaticky pak bude zahájena další transakce. Nastavení deaktivace automatického potvrzování má za následek, že všechny nepotvrzené příkazy po výpadku nebudou uloženy.

Příklad

Vyzkoušíme si transakce na triviálním příkladu dvou spojení. Demonstrovat jej budeme v řádkovém klientu **mysql.exe**. (pokud jste tak ještě neučinili, projděte si Přílohu 2)

Spustíme si řádkového klienta pro spojení (okno), které si nazveme jako ***spoj1***. Nejprve vymažeme všechny hodnoty z tabulky a nastavíme hodnotu automatické inkrementace na 1:

```
USE prvnidb;

DELETE FROM innotest;

ALTER TABLE innotest AUTO_INCREMENT = 1;
```

Stále pracujeme v režimu automatického potvrzování mimo transakci:

```
C:\Windows\system32\cmd.exe - mysql -u root -p
mysql> USE prvnidb;
Database changed
mysql> DELETE FROM innotest;
Query OK, 0 rows affected (0.01 sec)

mysql> ALTER TABLE innotest AUTO_INCREMENT = 1;
Query OK, 0 rows affected (0.18 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql>
```

Obr: spoj1

Nyní si vložíme nový řádek:

```
INSERT INTO innotest VALUES(NULL, '12345abcde');
```

Zahájíme transakci

Tato transakce bude jednoduchou ukázkou aktualizace záznamu:

```
START TRANSACTION;
```

Nechme nyní okno ***spoj1*** chvíli být a otevřeme si nové okno příkazového řádku. To nazveme jednoduše ***spoj2***. Připojíme se k serveru MySQL, stejně jako u prvního okna.

```
C:\Windows\system32\cmd.exe - mysql -u root -p
Microsoft Windows [Verze 6.0.6002]
Copyright (c) 2006 Microsoft Corporation. Všechna práva vyhrazena.

C:\Users\Honza>cd c:\xampp\mysql\bin

c:\xampp\mysql\bin>mysql -u root -p
Enter password: ***********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 188
Server version: 5.1.30-community MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE prvnidb;
Database changed
mysql>
```

Obr: spoj2

Nyní se opět vrátíme k první oknu, kde máme započatou transakci. Aktualizujeme náš jediný záznam:

```
UPDATE innotest SET hodnota = 'uvwxy98765' WHERE id = 1;
```

A vypíšeme si obsah tabulky ve spoj1:

```
SELECT * FROM innotest;
```

```
C:\Windows\system32\cmd.exe - mysql -u root -p
Query OK, 0 rows affected (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> USE innotest;
ERROR 1049 (42000): Unknown database 'innotest'
mysql> USE prvnidb;
Database changed
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE innotest SET hodnota = "uvwxy98765" WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM innotest;
+----+------------+
| id | hodnota    |
+----+------------+
|  1 | uvwxy98765 |
+----+------------+
1 row in set (0.00 sec)

mysql> _
```

Obr: spoj1 – aktualizovaná hodnota

Můžeme vidět, že se hodnota v těle transakce aktualizovala. Ale uvidí stejné změny spoj v okně ***spoj2*** ještě před dokončením transakce ve ***spoj1***?

Přepneme se do okna spoj2 a vykonáme stejný příkaz:

```
C:\Windows\system32\cmd.exe - mysql -u root -p
c:\xampp\mysql\bin>mysql -u root -p
Enter password: **********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 188
Server version: 5.1.30-community MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE prvnidb;
Database changed
mysql> SELECT * FROM innotest;
+----+------------+
| id | hodnota    |
+----+------------+
|  1 | 12345abcde |
+----+------------+
1 row in set (0.00 sec)

mysql> _
```

Obr: spoj2 – původní hodnota

Jak je vidno, jelikož změny transakce ještě nebyly potvrzeny, ***spoj2*** nemůže vidět aktualizovanou hodnotu. Dokončeme tedy transakci v okně ***spoj1*** příkazem COMMIT, aby se změny staly trvalými. Poté uvidíme aktualizovanou hodnotu v případě obou spojení. Změnu zachycuje další obrázek z okna ***spoj2***.

```
C:\Windows\system32\cmd.exe - mysql -u root -p
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 188
Server version: 5.1.30-community MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> USE prvnidb;
Database changed
mysql> SELECT * FROM innotest;
+----+------------+
| id | hodnota    |
+----+------------+
|  1 | 12345abcde |
+----+------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM innotest;
+----+------------+
| id | hodnota    |
+----+------------+
|  1 | uvwxy98765 |
+----+------------+
1 row in set (0.00 sec)

mysql>
```

Obr: spoj2 – po dokončení transakce

Stupně izolace

Moderní relační databázové systémy jsou navrženy pro práci mnoha uživatelů současně. Musí tedy existovat nějaká pravidla pro to, aby mohli všichni pracovat bez vzniku chaosu. Tato pravidla byla zpracována do stupňů izolace diktujících souběžné chování.

Než zahájíme novou transakci, můžeme definovat její izolační úroveň.

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL
  {
       READ UNCOMMITTED
     | READ COMMITTED
     | REPEATABLE READ
     | SERIALIZABLE
  }
```

Příkaz SET lze spustit třemi způsoby:

SET bez SESSION a GLOBAL: V takovém případě bude vybrané nastavení stupně izolace jen pro následující transakci.

SET SESSION: Zde nastavení platí až do ukončení spojení nebi další změny stupně izolace.

SET GLOBAL: Toto nastavení bude platit pro všechny nová připojení.

Jednotlivé stupně izolace si podrobně představíme:

READ UNCOMMITTED

SELECT čte data se všemi změnami, které udělaly jiné transakce, aniž by ještě byly ukončeny COMMIT. SELECT tedy není vůbec izolovaný od ostatních transakcí. V češtině se také nazývá jako **nekonzistentní čtení**. Pokud bychom tedy v dřívějším příkladu v okně ***spoj2*** zadali před výběrem stupeň izolace READ UNCOMMITTED, viděli bychom aktualizovaný řádek tabulky ještě před dokončením transakce v okně ***spoj1***.

Pamatujte, že READ UNCOMMITTED nezajišťuje izolaci pro dotazy SELECT. V případě, že by ***spoj2*** chtěl provést příkaz UPDATE na daném řádku, musí čekat na dokončení transakce ve ***spoj1***.

READ COMMITTED

Dotaz SELECT akceptuje jen ty změny z jiných transakcí, které byly dokončeny pomocí COMMIT. Je tedy restriktivnější než předchozí READ COMMITTED.

REPEATABLE READ

Dotaz SELECT v tomto případě nebere v úvahu změny z jiných transakcí a to i přesto, že tyto byly dokončeny COMMIT. Jeden a ten samý dotaz v rámci aktivního připojení **v transakci** vrátí jeden a ten samý výsledek (pokud samozřejmě příslušná data nezmění vlastní transakce).

SERIALIZABLE

Tento stupeň izolace funguje podobně jako REPEATABLE READ. Jediným rozdílem je, že běžné dotazy SELECT budou spouštěny jako SELECT … LOCK IN SHARE MODE, tudíž všechny přečtené záznamy vybaví sdíleným zámkem. **SERIALIZABLE se liší v uvedené definici od REPEATABLE READ pouze v případě, že je deaktivováno automatické potvrzování**.

Poznámka

Pokud není nastaveno jinak, InnoDB použije stupeň izolace REPEATABLE READ.

SELECT … LOCK IN SHARE MODE

V tabulkách InnoDB jsou dotazy SELECT spouštěny i na blokované záznamy. V předchozím příkladu se tak stalo. Transakce v okně ***spoj1*** aktualizovala daný řádek. A ačkoli byl touto transakcí blokován, mohlo k němu spojení v okně ***spoj2*** přistoupit a zobrazit jeho obsah. Pochopitelně v daném okamžiku to byla původní hodnota – tedy stará hodnota (12345abcde).

Jestliže bychom spustili SELECT s klíčovým slovem **LOCK IN SHARE MODE** na konci dotazu, pak bude ***spoj2*** trpělivě čekat, až bude transakce ve ***spoj1*** ukončena COMMIT. Poté zobrazí aktuální novou hodnotu daného řádku.

```
SELECT * FROM innotest WHERE id = 1 LOCK IN SHARE MODE;
```

SELECT … FOR UPDATE

Klauzule **FOR UPDATE** vybaví všechny výsledné záznamy *exkluzivním zámkem*. Stejně jako u předchozího sdíleného zámku, nemohou ostatní uživatelé (aplikace) měnit uzamčené záznamy. Zamčená data mohou všichni číst dotazem SELECT, ale ne dotazem SELECT … LOCK IN SHARE MODE.

Rozdíl mezi sdíleným a exkluzivním zámkem tedy spočívá pouze v tom, zda ostatní klienti mohou spustit SELECT … LOCK IN SHARE MODE:

Doporučení k transakcím

Potřebujeme-li používat transakce, je potřeba sáhnout po tabulkách InnoDB. Vzhledem k celkovému výkonu je ale potřeba počítat se ztrátami, větším zatížením paměti pro potřeby sledování interních zámků atd. Všechny činnosti v tabulkách InnoDB totiž probíhají jako transakce, ať už jsou ukončovány automaticky či nikoli. Transakce by měly být ve většině případů z logických důvodů co nejkratší.

Pokud vaše aplikace přímo nevyžaduje transakce, měli byste pro datová úložiště volit tabulky MyISAM. Shodují se tak dokonce ve většině autoři odborných publikací věnujících se MySQL. Pokud je potřeba transakcí, měli byste si osvojit následující pravidla volby stupňů izolace:

Zásady volby stupně izolace

Každý ze stupňů izolace transakcí se hodí pro nějaké konkrétní podmínky. Následující přehled je určitým návodem:

READ UNCOMMITTED

Tento stupeň používá nejméně zámků a proto je značně nebezpečný pro víceuživatelská dynamická prostředí. Ovšem může být s úspěchem použit, je-li uživatel jen jeden.

READ COMMITTED

Tento stupeň je vhodný pro také aplikace, které nemohou tolerovat nekonzistentní čtení (fantómové záznamy), a zároveň nepotřebují bezpečnější variantu REPEATABLE READ.

REPEATABLE READ

Defaultní stupeň izolace transakce. Hodí se pro aplikace vyžadující ochranu před nekonzistentním čtením, ochranu před jinými uživateli měnícími informace, které jsou upravovány aktuální aplikací, spolehlivost dat a data poskytovaná konzistentním čtením ke konkrétnímu okamžiku.

SERIALIZABLE

Maximální stupeň izolace, jenž by měl být použit pouze v případech, kdy se nemohou tolerovat žádné změny v datech ani pro pouhé načítání dat beze změn.

14 INDEXY

Dejme tomu, že máme za úkol vrátit záznamy s příjmením May z tabulky autori. Bez doplňujících informací nemůže MySQL vědět, kde nalézt záznamy naplňující toto kritérium. Skenování tabulky záznam po záznamu se označuje za **plný průchod tabulkou**. Když jsou tabulky objemné a obsahují třeba tisíce záznamů, může tato činnost probíhat velmi pomalu. Chceme-li tento problém překonat, pomůže nám setřídění záznamů.

Podívejme se na nově seřazenou tabulku podle příjmení:

id	jmeno	prijmeni
2	Dan	Brown
1	Agatha	Christie
4	Karel	May
3	Andrew	Oppel

Pokud máme takto seřazenou tabulku, hledání podle příjmení může probíhat mnohem rychleji. MySQL „ví“, že má hledat příjmení May v záznamech pouze po iniciálu M a následující tak může vynechat. Nicméně pokud bychom opětovně chtěli vyhledávat podle identifikátoru, vidíme, že hodnoty jsou přeházeny a zase bychom se dostali k pomalým a nevýkonným dotazům.

Řešením je vytvořit samostatné seznamy pro každé pole, které potřebujeme řadit. To nezahrnuje všechna pole tabulky, ale jenom ta, která potřebujeme seřazená. Seznamy řazení se označují za **indexy** a jsou jedním nejdůležitějších rozšíření MySQL.

Vytvoření indexu

V MySQL existují 4 druhy indexů:

- primární klíč,
- jedinečný index,
- fulltextový index a
- ordinální index.

Pokud chceme získat cenné informace o existujících indexech použitých na konkrétní tabulce, stačí vykonat příkaz:

```
SHOW INDEX FROM nazev_tabulky;
```

V případě PHPMyAdmin můžete indexy prohlížet u dané tabulky pod záložkou Struktura a odkazem +Podrobnosti.

Primární klíč

Primární klíč již známe z dřívějších kapitol. Je indexem na poli, kde je každá hodnota jedinečná a žádná z hodnot není NULL. Funkci primárního klíče může pochopitelně zároveň plnit více polí, jejichž kombinace tak vždy musí být jedinečná. Pokud nemůžeme vybrat jeden nebo více sloupců, jež mohou sloužit jako primární klíč, je dobré jednoduše vytvořit jeden numerický sloupec a nechat MySQL, aby jej naplnila unikátními daty (AUTO INCREMENT).

Primární klíč na poli vytvoříme při tvorbě tabulky. V PHPMyAdmin stačí ve výběrovém rozbalovacím poli Index zvolit hodnotu **PRIMARY**. Pozor, toto pole nesmíte volit nikdy jako nulové.

Nejprve vytvoříme jednoduchou tabulku, která bude využívat primární index na obou polích a pak do ní zkusíme vložit nějaká data:

```
CREATE TABLE `prvnidb`.`tb_pi` (
`alfa` INT NOT NULL ,
`beta` INT NOT NULL ,
PRIMARY KEY ( `alfa` , `beta` )

) ENGINE = MYISAM ;

INSERT INTO tb_pi VALUES(1,1);

INSERT INTO tb_pi VALUES(1,2);
```

alfa	beta
1	1
1	2

Data jsou úspěšně vložena, oba záznamy jsou svou kombinací jedinečné. Pokud by Primary index existoval pouze na poli alfa, náš druhý záznam by vložit nešel, protože hodnota 1 již byla vložena v prvním záznamu.

Ordinální (obvyklý) index

Index, který není unikátní, povoluje duplicitní hodnoty. Vždy je dobré jej vytvořit již při vytváření tabulky. **Ordinální index by se měl volit v těch případech, kdy se počítá s velkým vytížením tabulky na mnoho záznamů a u pole, které bude často využíváno za podmínkou WHERE.**

Takovýto index bychom např. využili v tabulce Studenti pro pole rok_narozeni. Uvažujme, že tato tabulka bude obsahovat tisíce záznamů a dotazy SELECT budou často hledat studenty s konkrétním rokem narození. Např.:

```
SELECT * FROM studenti WHERE rok_narozeni > 1988;
```

Pokud by nebyl použit index na poli **rok_narozeni**, dojde k plnému průchodu tabulkou a tedy velkým časovým ztrátám. V případě použití indexu budou data seřazeny a MySQL svůj ukazatel posune na rok 1989 a vybere všechny následující záznamy.

Upravme tedy tabulku Studenti a přidejme Ordinální index na pole **rok_narozeni**. V PHPMyAdmin stačí u tabulky Studenti klepnout na záložku Struktura a u pole rok_narozeni v sekci Akce klepnout na Index (ikona s bleskem). Dojde k vytvoření Indexu.

```
ALTER TABLE `studenti` ADD INDEX (`rok_narozeni`);
```

Ano, index vyrázně urychlí operace při filtrování dat, ovšem nic není zadarmo. Indexy vyžadují další místo na disku. Čas při editaci či mazání záznamů při použití mnoha indexů značně narůstá, protože MySQL musí při každé změně tabulky aktualizovat také indexy. Tabulky s mnoha indexy znamenají pomalejší úpravy dat, takže je vždy nutno pečlivě analyzovat jejich výhody i nevýhody.

Mohutnost indexu

Jak jste si mohli všimnout u vyhodnocení příkazu **SHOW INDEX,** příkaz vrací mimojiné informace o mohutnosti indexu (tj. o unikátnosti hodnot sloupců tvořících index). Mohutnost by měla být vždy co možná největší. Vysoce duplicitní index s mnoha opakujícími se daty má pouze několik klíčových hodnot. V mnoha případech je lepší index nepoužít vůbec, než vytvářet index vysoce duplicitní. Jako třeba v případě, kdy tabulka s více jak tisíci řádky může na poli pohlaví obsahovat hodnotu M nebo F.

Když přidáváme řádek do tabulky s vysoce duplicitním indexem, musí MySQL projít mnoho záznamů duplicitního indexu, aby našla správné místo pro nový řádek. To samozřejmě negativně ovlivňuje výkon.

Fulltextový index

V tabulkách MyISAM můžeme vytvářet fulltextové indexy na polích typu **CHAR, VARCHAR** nebo **TEXT**. Fulltextový index slouží k jednoduchému hledání klíčových slov v textových polích rozsáhlých tabulek.

Fulltextový index vytváříme opět velmi jednoduše. V PHPMyAdmin stejným způsobem jako v případě ordinálního indexu, tentokrát klepnutím na ikonu indexu s písmenem T. V tabulce knihy vytvoříme fulltextový index na poli **nazev**. SQL příkaz pro již vytvořenou tabulku je následující:

```
ALTER TABLE `knihy` ADD FULLTEXT(`nazev`);
```

Používání fulltextového indexu

K vrácení výsledků fulltextového hledání se používá funkce **MATCH()**, kdy porovnáváme shodu v poli vzhledem (AGAINST()) k nějaké hodnotě, jak znázorňuje následující příklad výskytu slova „vražda“.

```
SELECT nazev FROM knihy WHERE MATCH(nazev) AGAINST('vražda');
```

nazev
Oznamuje se vražda
Vražda v Orient-Expressu

Všimněte si, že hledání nerozlišuje velká a malá písmena. Je tomu tak u všech polí typu TEXT, VARCHAR a CHAR.

Princip hledání

- MySQL využívá princip, který se označuje za **50 procentní práh**. Všechna slova, která se objevují ve více než 50 procentech polí se považují za **často používaná slova**, což znamená, že se při vyhledávání ignorují.
- Všechna slova se třemi nebo méně znaky se ignorují.
- Existuje rovněž seznam předdefinovaných často používaných slov, které zahrnuje např. anglický určitý člen THE, které nebudou v hledání reflektovány.

Poznámka

Máme-li tabulku jen s jedním záznamem, všechna jeho slova budou často používaná, takže fulltextové hledání vůbec nic nevrátí! Tabulky s velmi malým počtem záznamů rovněž zvyšují pravděpodobnost výskytu často používaných slov.

Relevance

V podmínce WHERE můžeme vracet následující hodnoty relevance pro daná hledání:

```
SELECT nazev, (MATCH(nazev) AGAINST('vražda')) FROM knihy;
```

nazev	(MATCH(nazev) AGAINST('vražda'))
Vražda v Orient-Expressu	0.670031070709229
Oznamuje se vražda	0.67756325006485
Andělé a démoni	0
Šifra mistra Leonarda	0
Databáze bez předchozích znalostí	0
Pozvánka pro Hercula Poirota	0

Hledání ve více polích zároveň

Pokud bychom měli tabulku s více textovými poli a chtěli bychom prohledávat v těchto polích zároveň, dotaz by vypadal následovně:

```
SELECT * FROM tabulka WHERE MATCH(pole1, pole2, ..., poleN)
 AGAINST('hledaný výraz');
```

Ovšem je zde nutná podmínka. Fulltextový index musí být na těchto polích sjednocen. V PHPMyAdmin musíte vybrat (zaškrtávací pole) požadované sloupce a dole pod tabulkou vybrat fulltextový index.

Součástí fulltextového vyhledávání je možnost logického (booleovského) fulltextového hledání. Lze využívat sady prvků pro hledání slov, kombinaci slov, částí slov atd. Navíc logická fulltextová hledání nedodržují 50 procentní limit.

Chceme-li vykonat logická fulltextová hledání, použijeme klauzuli **IN BOOLEAN MODE**:

```
SELECT * FROM tabulka WHERE MATCH(pole1) AGAINST('hledaný výraz' IN BOOLEAN MODE);
```

Logické operace hledání

Operátor	Popis
+	Následující slovo je povinné a musí se nacházet na každém vráceném řádku
-	Následující slovo je zakázané a nesmí se nacházet na žádném vráceném řádku
<	Následující slovo má nižší relevanci než jiná slova.
>	Následující slovo má vyšší relevanci než jiná slova.
()	Používá se k seskupení slov v podvýrazech.
~	Následující slovo přispívá záporným způsobem k relevanci řádku.
*	Zástupný znak indikující nulový nebo větší počet znaků. Může se objevit pouze na konci slova.
"	Vše uzavřené do uvozovek se bere jako celek.

Ukážeme si dva příklady:

```
SELECT nazev FROM knihy WHERE MATCH(nazev) AGAINST('+vražda -oznamuje' IN BOOLEAN MODE);
```

nazev
Vražda v Orient-Expressu

V tomto příkladu je slovo *oznamuje* vyřazeno, takže nedojde k vrácení záznamu **Oznamuje se vražda**.

Všimněte si rozdílu mezi těmito dvěma sadami výsledků:

```
SELECT nazev FROM knihy WHERE MATCH(nazev) AGAINST('Oznamuje vražda' IN BOOLEAN MODE);
```

nazev
Vražda v Orient-Expressu
Oznamuje se vražda

```
SELECT nazev FROM knihy WHERE MATCH(nazev) AGAINST('+Oznamuje
```

```
vražda' IN BOOLEAN MODE);
```

nazev
Oznamuje se vražda

Slovo *oznamuje* je v druhém případě povinné, takže nedojde k vrácení titulu Vražda v Orient-Expressu.

Jedinečný index

Jedinečný (UNIQUE) index stejně jako PRIMARY **nepovoluje žádné duplikáty**. Oproti primárnímu klíči, může být v tabulce více unikátních indexů. Chceme-li vytvořit unikátní index, stačí v PHPMyAdmin u daného pole kliknout na ikonu indexu s písmenem U.

```
ALTER TABLE `knihy` ADD UNIQUE(`isbn`);
```

Vytváření ordinálních indexů z části pole

V případě sloupců typu VARCHAR, CHAR, BLOB a TEXT nám MySQL umožňuje vytvořit index, který nepoužívá celé takové pole. Třebaže příjmení studenta může obsahovat až 40 znaků je pravděpodobné, že všechna příjmení se budou lišit již v deseti prvních znacích. Když pro index použijeme tak jen prvních deset znaků, index bude mnohem menší. Tím by se zrychlila aktualizace i vkládání a navíc to nebude mít žádný vliv na rychlost příkazu SELECT.

Chcete-li vytvořit index z části pole, prostě uveďte požadovanou velikost. Kupříkladu při vytváření 10znakového indexu na poli přijmení:

```
ALTER TABLE `prvnidb`.`studenti` ADD INDEX(`prijmeni`(10));
```

Volba indexů

Jak už bylo naznačeno, volba indexů je vždy otázkou pečlivé analýzy. Existují základní pravidla volby indexů, které je ale potřeba vždy důkladně zvážit vzhledem k rozsahu tabulek a návrhu aplikace pro něž je databáze určena.

- Indexy vytvářejte na polích, kde se budou vykonávat nějaké dotazy (nejčastěji na polích v podmínce WHERE).
- Vytvářejte indexy na polích, které slouží pro spojení s jinými tabulkami.
- Vytvářejte indexy, které vracejí co nejmenší možný počet řádků (zde je nejlepší primární klíč, protože primární klíč je přiřazen jednomu záznamu). Jinak řečeno, vytvářejte indexy s co možná největší mohutností.
- Pokud je to možné, lépe volit numerické indexy (ve srovnání s řetězcovými) umožňují rychlejší přístup k datům.

- Používejte krátké indexy (indexujte např. jen prvních 10 znaků jména a nikoli celé pole).
- Nevytvářejte příliš mnoho indexů. Index zvětšuje čas, potřebný pro aktualizaci a vložení záznamu. Pokud daný index slouží jen zřídkakdy používanému dotazu, který klidně může běžet trochu pomaleji, zvažte možnost takový index vůbec nevytvářet.

15 POHLEDY

MySQL od verze 5 podporuje velmi užitečný nástroj, kterým jsou tzv. **pohledy**. Pohled je jakýmsi přechodem (uloženou definicí výběrových dotazů) mezi trvalou základní tabulkou a odvozenou tabulkou. Pro jednoduchost si lze pohled představit jako tabulku, se kterou lze za jistých okolností i stejně pracovat. Pohled může vycházet z jedné či více tabulek a může obsahovat i odvozené atributy, které se přímo ve zdrojových tabulkách nevyskytují.

Pohledy jsou užitečné při přístupu k různým druhům dat. Jednou z hlavních výhod pohledu je to, že můžeme definovat složité dotazy a uložit je v definici pohledu. Takto můžeme podle potřeby volat pohledy místo opětovného vytváření dotazů.

Kromě toho mohou být pohledy ideálním způsobem prezentace dat uživatelům bez potřeby poskytování nadbytečných nebo nežádoucích dat. To znamená vytvořit **pohled** takový, který vrací pouze ty sloupce, které jsou nezbytné pro aktuální zobrazení.

Vytvoření pohledu

Syntaxe vytvoření pohledu:

```
CREATE [OR REPLACE] VIEW jmeno_pohledu
[(seznam_atributu)]
AS dotaz_vybirajici_data
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

Seznam atributů představuje výčet sloupců, které chceme do pohledu zařadit. Za klauzulí **AS** následuje výběrový dotaz, který může obsahovat i složitá spojení z více tabulek.

Vytvoříme si nejjednodušší možný pohled, který načítá data pouze z jedné tabulky:

```
CREATE VIEW nazvy_knih
(nazev)
AS SELECT nazev FROM knihy;
```

V případě, že chceme pohled odstranit, použijeme příkaz **DROP**, kterým lze odstranit jak tabulky, tak celé databáze MySQL.

```
DROP VIEW nazvy_knih;
```

Nyní si vytvoříme užitečnější pohled, který bude vracet názvy knih spolu se jménem autora:

```
CREATE VIEW seznam_literatury
(nazev, autor)
AS SELECT nazev, CONCAT(jmeno, ' ', prijmeni)
      FROM knihy
      INNER JOIN autori ON autori.id = knihy.id_autor
WITH CHECK OPTION
```

Dotaz

```
SELECT * FROM seznam_literatury;
```

nazev	autor
Vražda v Orient-Expressu	Agatha Christie
Oznamuje se vražda	Agatha Christie
Andělé a démoni	Dan Brown
Šifra mistra Leonarda	Dan Brown
Databáze bez předchozích znalostí	Andrew Oppel

Aktualizované pohledy

V MySQL jsou některé pohledy **aktualizovatelné**. Jinými slovy můžeme pohled používat k úpravám dat v základní tabulce, z níž je pohled odvozen. Za jistých okolností tak můžeme používat i příkazy INSERT, UPDATE a DELETE.

Aktualizovatelné pohledy nesmí obsahovat:

- agregační funkce (min, max, sum, avg, count, concat ..)
- klauzuli DISTINCT pro výpis jedinečných záznamů
- klauzuli UNION pro slučování výsledků dotazů
- GROUP BY pro seskupování hodnot
- poddotaz v podmínce WHERE, který se odkazuje na tabulku uvedenou za klíčovým slovem FROM našeho pohledu

Příklad

```
UPDATE seznam_literatury SET nazev = 'Vražda v Orient Expressu'
 WHERE nazev = 'Vražda v Orient-Expressu';
```

Provedete-li výše vypsaný příkaz pro aktualizaci záznamu, bude úspěšně proveden. A to i přesto, že definice pohledu obsahuje spojení tabulek a agregační funkci.

Pokud bychom však chtěli pomocí pohledu smazat některý záznam (případně vložit nový), příkaz skončí chybou. To kvůli tomu, že záznamy v uvedeném pohledu **seznam_literatury** jsou složeny z dat dvou tabulek, které jsou spojeny pomocí INNER JOIN. Nelze tedy určit, jak by se data vkládala.

```
DELETE FROM seznam_literatury WHERE autor = "Agatha Christie";
```

WITH CHECK OPTION

Přidání klauzule **WITH CHECK OPTION** na konec příkazu zajistí, aby aktualizace pohledů prošly kontrolou platnosti před jejich přijetím do databáze. To znamená, že aktualizace budou provedeny v tom případě, že nezpůsobí vyřazení záznamu z pohledu (tj. jsou splněny podmínky za WHERE v dotazu SELECT v definici pohledu.)

- Možnost **WITH LOCAL CHECK OPTION** hlídá splnění podmínky WHERE pouze aktuální pohledu. Další pohledy, z nichž může být tento pohled odvozen hlídány nejsou. Používá se málo.
- **WITH CASCADED CHECK OPTION** hlídá splnění podmínky WHERE v pohledu samotném i pohledy z nichž je definovaný pohled odvozen. Používá se častěji.

Jestliže neuvedeme LOCAL ani CASCADED, je nastaveno defaultně **CASCADED**.

Doplněk

V některých publikacích se možná dozvíte, že pohledy, které obsahují data z více než jedné tabulky nejsou nikdy aktualizovatelné. Jak jsme se před chvíli mohli přesvědčit, platí tomu tak napůl. Jelikož jsme upravovali název knihy a v podmínce jsme uváděli sloupec téže tabulky, aktualizace se zdařila. V tabulce knihy byl záznam úspěšně aktualizován. **Ovšem při vložení nového záznamu nebo pokusu o smazání dojde k chybě**. Zde již narážíme na spojení tabulek a také na použití funkce CONCAT() u jména autora. **Operace INSERT a DELETE tak nemohou být provedeny.**

Ale i při určitém pokusu o aktualizaci může dojít k nezdaru. Pakliže je dotaz formulován v rozporu s definicí pohledu, skončí stejně jako v případě mazaní a vložení:

```
UPDATE seznam_literatury SET autor = 'A.Christie' WHERE autor

 = 'Agatha Christie';
```

Pole autor v pohledu **seznam_literatury** vzniklo spojením jména a příjmení autora z tabulky **autori**. Není tak možná jeho editace.

16 Uložené procedury a triggery

Uložené procedury (UP) – primárně pro předmět Databázové systémy 2

Uložené procedury jsou novinkou od verze MySQL 5. Jedná se o vlastní programy nebo funkce SQL, které jsou uloženy a spouštěny přímo serverem MySQL. Díky uloženým procedurám máme k dispozici programovací jazyk vycházející z SQL a můžeme do nich ukládat část logiky aplikace klient/server.

Výhody UP

Vyšší rychlost

V rámci komunikace klient/server se stává, že mezi aplikací a databázovým serverem se přenáší velké množství dat. Vybírají se hodnoty tabulky, na základě toho se vkládají nová data, … atd. Pokud by se toto dalo sloučit a provést na straně serveru, můžeme značně ulehčit zátěži sítě.

Další výhodou UP je, že MySQL může jejich kód kompilovat a tím opět zvýšit rychlost provádění.

Používání UP ale samo o sobě není zárukou zvýšení rychlosti či zvýšení efektivity. Toho lze dosáhnout opravdu jen dobrým návrhem. Programovací jazyky, jako např. PHP jsou mnohem lépe vyvinutější než jednoduchá platforma SQL, ve které se optimální kód uložené procedury někdy napsat nedá.

Při užití UP je více zatěžován databázový server a je tedy potřeba zvážit, zda toto zatížení je přínosné, vzhledem k snížení zátěže sítě a např. webového serveru.

Odpadá redundance kódu

UP mohou napomoci odstranit stálé používání identických příkazů, které se v programové části aplikace opakují (kontrolují vstupní hodnoty, vkládají nové záznamy atd). Pokud lze takový kód přesunout do UP, odpadá nejen redundance kódu, ale značně se zlepšuje správa aplikací.

Vylepšení zabezpečení databáze

V mnoha oborech, kde bezpečnost je na prvním místě, je potřeba, aby aplikace nemohly přistupovat k databázovým tabulkám přímo, ale pouze volaly UP, které provedou všechny operace samostatně. Na serveru jsou proto uloženy všechny databázové operace, přes dotazy SELECT, INSERT atd. Správce databáze tím získává velkou výhodu: každý přístup k datům má pod kontrolou a v případě potřeby může vše potřebné protokolovat.

Nevýhody UP

UP mají odlišný syntax pro různé RDBMS. A proto je většinou nelze přenášet na jiné systémy. Tedy nelze bez komplikace převádět UP např. z ORACLE na MySQL či naopak.

Tvorba UP

Existují dva typy UP:

- Procedury.
- Funkce.

Navzájem se mezi sebou liší. Prozatím stačí říci, že funkce vrací hodnoty, zatímco procedury podporují parametry formou odkazů a bohatší paletu příkazů SQL.

Nejprve si vyzkoušíme definici konkrétní funkce (než budete kód zkoušet, čtěte odstavec níže):

```
CREATE FUNCTION zkratit(s VARCHAR(255), n INT)
RETURNS VARCHAR(255)
BEGIN
   IF ISNULL(s) THEN
                RETURN NULL;
   ELSEIF n < 15 THEN
                 RETURN LEFT(s , n);
   ELSE
     IF CHAR_LENGTH(s) <= n THEN
                           RETURN s;
     ELSE
         RETURN CONCAT(LEFT(s, n-10), '…', RIGHT(s, 5));
     END IF;
   END IF;
END
```

Před vytvořením této funkce prostřednictvím PHPMyAdmin věnujte pozornost následujícímu:

Pod oknem pro vložení příkazu SQL je pole pro vložení oddělovače. Standardně jsme prozatím tuto hodnotu měnit nemuseli, protože středník opravdu ukončoval jednotlivé příkazy. Ovšem nyní je středník součástí jednoho bloku (funkce) a jeho výskyt v těle funkce by skončil chybou denice funkce.

Proto nyní musíme oddělovač nahradit zástupným znakem $$

Po úspěšném definování funkce si můžeme vyzkoušet její funkčnost. Pakliže jste si prošli postupně celý studijní materiál, je vám teoreticky jasné, jak tato funkce funguje:

```
SELECT zkratit('tento text bude zkracen podle funkce zkratit', 25)
 AS zkraceny_text;
```

Aplikace funkce na tabulku:

```
SELECT nazev, zkratit(nazev, 15) AS zkraceny_nazev FROM knihy;
```

nazev	zkraceny_nazev
Vražda v Orient-Expressu	Vražd…ressu
Oznamuje se vražda	Oznam…ražda
Andělé a démoni	Andělé a démoni

Šifra mistra Leonarda	Šifra…narda
Databáze bez předchozích znalostí	Datab…lostí

Interní uložení UP

MySQL ukládá **UP** (ang. stored procedure) do tabulky **proc** v databázi s názvem **mysql**. Ve sloupcích této tabulky jsou uloženy tyto údaje: název podkladové databáze, název a typ procedury, parametry, skutečný kód a další atributy:

```
SELECT * FROM mysql.proc;
```

Správa UP

Vytváření UP

UP se v MySQL vytvářejí pomocí klíčového slova **CREATE FUNCTION** nebo **CREATE PROCEDURE**. Syntaxe obou příkazů je shrnuta v následujících řádcích:

```
CREATE FUNCTION nazev ([seznam parametru]) RETURNS datovy typ

[volby] kod SQL


CREATE PROCEDURE nazev ([seznam parametru])

[volby] kod SQL
```

V obou příkazech se vytvoří asociace mezi vytvořenou UP a aktuální databází. Jednotlivé příkazy uvnitř UP jsou odděleny středníky, které mohou činit problémy v případě, že používáte interpret příkazů mysql. Tedy je potřeba nastavit DELIMITER nebo-li oddělovač ze středníku na jiný zástupný znak.

Odstranění UP

Pro odstranění slouží opět nám již dobře známý příkaz **DROP**. Volitelná klíčová slova **IF EXISTS** při svém přidání způsobí, že příkaz bude vykonán bez vyvolání chyby, pokud by snad daná rutina neexistovala:

```
DROP FUNCTION [IF EXISTS] nazev

DROP PROCEDURE [IF EXISTS] nazev
```

Změna UP

V případě potřeby změny kódu UP je potřeba UP nejprve odstranit a poté znovu vytvořit.

Zjištění údajů o UP

Dva příkazy **SHOW PROCEDURE STATUS** a **SHOW FUNCTION STATUS** vrací seznam všech definovaných procedur a funkcí.

Syntaxe a prvky jazyka

UP sestávají z klasických příkazů jazyka SQL, které již znáte. Kromě toho mohou být ale použity ještě další příkazy, např. příkazy podmínečných bloků nebo cyklů. Na tyto se zaměříme níže. Nejprve se v následující tabulce podíváme na hlavní rozdíly mezi procedurami a funkcemi:

	Procedury	**Funkce**
Volání	Pomocí klíčového slova CALL	Možné ve všech příkazech SQL
Návratové hodnoty	Může vrátit jednu či více výsledných sad SELECT	Vrací jedinou hodnotu (příkazem RETURN). Datový typ návratové hodnoty musí být určen v deklaracích spolu s RETURNS
Parametry	Povoleny jsou parametry předávané hodnotou i odkazem (IN, OUT, INOUT)	Povoleny jsou parametry předávané hodnotou, proto není přípustné jejich označení IN.
Příkazy přípustné v kódu	Všechny příkazy SQL	Nejsou přípustné příkazy přistupující k tabulkám.
Volání jiných funkcí a procedur	Je možné volat jiné procedury a funkce	Je možné volat jen jiné funkce.

Obecná pravidla syntaxe

Středník

UP může být složena z většího počtu příkazů, které musejí být odděleny středníkem. Rovněž řídící struktury.

BEGIN/END

Kód každé UP, která se skládá z více než jedné instrukce, musí začínat příkazem **BEGIN** a končit **END**. Konstrukce BEGIN/END lze rovněž vnořovat.

Proměnné

Lokální proměnné a parametry jsou používány bez předpony @. Uvnitř UP můžeme přistupovat i k běžným proměnným, ale na ně je potřeba se odkazovat i s předponou @.

NON CASE – SENSITIVE

V definici nebo volání UP se nerozlišují malá a velká písmena. Zajímavostí je, že lze vytvořit proceduru a funkci téhož jména. Při jejich volání umí MySQL rozlišit, zda-li se jedná o proceduru či funkci

Komentáře

-- Komentáře začínají dvojí pomlčkou a končí s koncem řádku.

Volání uložených procedur

Všechny UP jsou asociovány s určitou databází. Pokud bychom chtěli spustit UP patřící do jiné databáze, musíme název této databáze připojit před název uložené procedury. Např.

```
CALL druhaDB.nazev_up()
```

Funkce

Funkce jsou stejně jako vestavěné funkce SQL integrovány do běžných příkazů SQL:

```
SELECT zkratit('textový řetězec, který je potřeba ořezat', 25);

SELECT zkratit(nazev, 10) FROM knihy;

UPDATE knihy SET nazev = zkratit(nazev, 10) WHERE CHAR_LENGTH(nazev)
 > 10;
```

V příkazech SET nebo SELECT INTO pro načtení hodnoty do proměnné můžeme také použít funkce:

```
SET @str = 'textový řetězec';

SET @str_z = zkratit(@str, 10);

SELECT zkratit(@str, 10) INTO @str_z2;
```

Procedury

Procedury je nutné volat příkazem **CALL**. Jako výsledek volání lze vrátit i tabulku (stejně jako v příkazu SELECT). Jiným způsobem se procedury na příkazy napojit nedají.

```
CREATE PROCEDURE nacist_zaznam(IN id_titulu INT)
BEGIN
   SELECT nazev, CONCAT(jmeno, ' ', prijmeni) AS autor
   FROM knihy INNER JOIN autori ON autori.id = knihy.id_autor
   WHERE knihy.id = id_titulu;
END
```

Tato triviální UP se chová úplně stejně jako SELECT se spojením tabulek knihy a autori. Vyzkoušejme:

```
CALL nacist_zaznam(1);
```

nazev	autor
Vražda v Orient-Expressu	Agatha Christie

Bohužel PHPMyAdmin podporuje na 100% pouze správu procedur. V případě jejich vykonávání v rámci vracení setu výsledků nebývá již tak úspěšný. Proto předchozí příklad v PHPMyAdmin nebude vykonán. Můžeme proceduru vyzkoušet z příkazového řádku:

```
C:\Windows\system32\cmd.exe - mysql -u root -p
c:\xampp\mysql\bin>mysql -u root -p
Enter password: **********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 49
Server version: 5.1.30-community MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use prvnidb;
Database changed
mysql> CALL nacist_zaznam(1);
+--------------------------+-----------------+
| nazev                    | autor           |
+--------------------------+-----------------+
| Vražda v Orient-Expressu | Agatha Christie |
+--------------------------+-----------------+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.02 sec)

mysql>
```

Obr: Volání UP v řádkovém klientu MySQL

Parametr předávaný hodnotou (v příkladu je to hodnota 1) je deklarován při tvorbě procedury volbou IN. V případě procedur s **parametry předávanými odkazem** (OUT nebo INOUT) lze výsledek vyhodnotit jen v případě, kdy je jako parametr předána proměnná SQL.

Vše dobře pochopíte na následujícím příkladu procedury s názvem *polovina*:

```
CREATE PROCEDURE polovina (IN a INT, OUT b INT)
BEGIN
   SET b = a/2;
END
```

A nyní proceduru zavoláme a využijeme:

```
CALL polovina (10, @vysledek);
SELECT @vysledek AS polovina_zadani;
```

V tomto případě procedura odpovídá i v aplikaci PHPMyAdmin. Můžeme vyzkoušet.

Parametry a návratové hodnoty

Parametry v procedurách

Seznam parametrů v procedurách je nepovinný, stejně jako např. v případě funkcí v PHP. Ale stejně tak je potřeba uvést pár závorek.

```
CREATE PROCEDURE nazev ([seznam parametru])

[volby] kod SQL
```

Dva a více parametrů je nutné v definici oddělit čárkami. Každý parametr je uváděn v tomto tvaru:

```
[IN nebo OUT nebo INOUT] nazev_parametru datovy_typ
```

Klíčová slova IN, OUT a INOUT stanoví, zda lze parametr použít jen jako vstupní, výstupní, nebo pro přenos dat v obou směrech.

Povoleny jsou všechny datové typy MySQL, např. INT, VARCHAR(n) atd. **Je důležité dávat dobrý pozor, aby se názvy parametrů lišily od názvů tabulek a sloupců, jinak může v kódu SQL dojít k chybám interpretace.**

Návratové hodnoty procedur

Na rozdíl od funkcí procedury nevrací striktně jednu hodnotu. V procedurách je však možno běžně používat příkazy SELECT, a to i vícekrát za sebou. Procedura tak může vracet (zobrazovat) více výsledných sad dotazu SELECT.

Parametry funkcí

Významný rozdíl oproti procedurám spočívá v tom, že funkce nepodporují parametry předávané odkazem. Z tohoto důvodu není možné v seznamu parametrů používat klíčová slova IN, OUT a INOUT.

```
CREATE FUNCTION nazev ([seznam parametru]) RETURNS datovy typ

[volby] kod SQL
```

Návratové hodnoty funkcí

Funkce vracejí hodnotu pomocí příkazu RETURN, který zároveň končí provádění kódu funkce. Příkaz RETURN lze použít jen u funkcí, nikoli v procedurách. Datový typ návratové hodnoty musí být určen v seznamu parametrů pomocí klauzule RETURNS.

Zapouzdření příkazů

Každá procedura nebo funkce, která sestává z více jak jednoho příkazu SQL, musí začínat klíčovým slovem **BEGIN** a končit slovem **END**. Konstrukce BEGIN – END jsou přípustné také uvnitř kódu, např.

v podmínečném bloku **IF** nebo v **cyklu**, pokud je nutné deklarovat místní proměnné, podmínkové výrazy, zpracování chyby nebo kurzory.

Uvnitř bloku BEGIN-END musí být dodrženo následující pořadí:

```
BEGIN
	DECLARE proměnné;
	DECLARE kurzory;  // Nebudeme probírat
	DECLARE podmínky;
	DECLARE zpracovani chyby;
	jiné příkazy SQL;
END;
```

Před klíčovým slovem BEGIN je možné uvést nepovinné návěští. Jeho název se pak může zopakovat za slovem END. Pojmenování takového bloku má význam v případě, pokud je možné blok předčasně ukončit příkazem LEAVE. Syntaxi této konstrukce můžete vidět níže:

```
nazev_bloku: BEGIN
příkazy;
IF podmínka THEN LEAVE nazev_bloku; END IF;
další příkazy;
END nazev_bloku;
```

Proměnné

V souvislosti s procedurami a funkcemi je třeba brát v úvahu dva typy proměnných:

1. **Běžné proměnné** – obsahují předponu @.
2. **Lokální proměnné a parametry** – nemají předponu @ a musí být v UP deklarovány příkazem DECLARE. Obsah lokálních proměnných je ztracen ihned poté, co blok ve kterém byly deklarovány končí. Lokálním rozsahem proměnných se rozumí vnitřek bloku BEGIN/END, ve kterém jsou definovány.

Obecně platí, že lépe je v UP používat lokální proměnné, a to kvůli vyloučení negativních vedlejších účinků.

Deklarování

Deklarace lokálních proměnných musí být umístěna uvnitř bloku BEGIN-END a před dalšími příkazy v tomto bloku. Syntaxe deklarace proměnné vypadá takto:

```
DECLARE nazev_promenne1, nazev_promenne2, ... datový typ [DEFAULT hodnota]
```

U všech lokálních proměnných je nutné zadat jejich datový typ. Lokální proměnné obsahují implicitně hodnotu NULL, pokud neprovedete jejich inicializaci jinou hodnotou.

Následující ukázka předvádí úrovně definice lokálních proměnných. Obsahuje celkem tři proměnné s názvem x, každá je však deklarovaná na jiné úrovni kódu, takže jsou na sobě navzájem nezávislé. Když tuto proceduru zavoláme, budou vráceny tři výsledné hodnoty (2, 1, 0).

```
CREATE PROCEDURE zkouska_promenne()
BEGIN
   DECLARE x INT DEFAULT 0;
   BEGIN
      DECLARE x INT DEFAULT 1;
      IF TRUE THEN
      BEGIN
         DECLARE x INT DEFAULT 2;
         SELECT x;
         END;
         END IF;
      SELECT x;
     END;
   SELECT x;
END
```

Větvení kódu

Vzpomenete si ještě na syntax funkce **IF**, kterou jsme si popisovali dříve v rámci pokročilejších struktur SQL? Vypadá takto:

```
IF(podmínka, splneni_podminky, nesplneni_podminky)
```

Z PHP či JavaScriptu jsme ale zvyklí na trochu jiný syntax. Jeho podobu nabývá konstrukce IF v rámci uložených procedur. Ačkoli opět s menšími změnami.

Syntaxe SQL pro rozhodovací bloky je následující:

```
IF podminka THEN
      příkazy;
```

```
[ELSE IF podminka THEN
      příkazy;] ...
[ELSE
      příkazy;]
END IF;
```

Syntax je velmi podobná, ale oproti PHP či JavaScriptu je tu přeci jen několik změn.

1. Podmínka se nezapisuje do klasických závorek.
2. Sled příkazů po splnění podmínky je odstartován klíčovým slovem **THEN**.
3. Více příkazů po splnění podmínky se nezapisuje do složených závorek.

Cykly

REPEAT – UNTIL

Tento cyklus je ekvivalentním k cyklu DO-WHILE, který využívají mnohé programovací jazyky. Instrukce uvedené mezi klíčovými slovy cyklu jsou prováděny do té doby, **dokud není poprvé splněna zadaná podmínka**. Protože tato podmínka se vyhodnotí až na konci cyklu, provede se tělo cyklu minimálně jednou.

Cyklus může mít přiděleno nepovinné označení, jehož název se musí opakovat na konci cyklu. Pojmenování cyklu má význam v tom případě, pokud ho chcete předčasně ukončit příkazem LEAVE nebo chcete opakovat iteraci cyklu pomocí příkazu ITERATE.

```
[nazev_cyklu:] REPEAT
příkazy;
UNTIL podmínka
END REPEAT [nazev_cyklu];
```

WHILE

Instrukce uvedené mezi klíčovými slovy DO a END WHILE se provádí, **dokud je splněna zadaná podmínka cyklu**. Protože se tato podmínka vyhodnotí na začátku cyklu, může se stát, že se příkazy v cyklu neprovedou ani jednou. Pokud chceme doplnit LEAVE nebo ITERATE musíme cyklus doplnit o označení:

```
[nazev_cyklu:] WHILE podmínka DO
příkazy;
END WHILE [nazev_cyklu];
```

LOOP

Instrukce mezi LOOP a END LOOP se provádí do té doby, kdy cyklus ukončí příkaz LEAVE nazev_cyklu.

```
[nazev_cyklu:] LOOP
příkazy;
END LOOP [nazev_cyklu];
```

Příklad

```
CREATE FUNCTION vrat_pomlcky (n INT)
      RETURNS VARCHAR(255)
BEGIN
      DECLARE i INT DEFAULT 0;
      DECLARE s TEXT DEFAULT '';
      cyklus: LOOP
            IF i > n THEN LEAVE cyklus;
            END IF;
            SET i = i + 1;
            SET s = CONCAT(s, '-');
            END LOOP cyklus;
      RETURN s;
END
```

Funkce po zavolání s celočíselným parametrem vrátí díky cyklu týž počet pomlček. Můžeme vyzkoušet:

```
SELECT vrat_pomlcky(5);
```

LEAVE a ITERATE

Příkaz **LEAVE** nazev_bloku ukončí provádění bloku nebo cyklu. Příkaz LEAVE lze také použít pro předčasné ukončení bloku BEGIN-END.

Příkaz **ITERATE** nazev_cyklu znamená, že se cyklus začne novou smyčku od začátku.

Zpracování chyb

Během provádění příkazů SQL uvnitř UP může dojít k chybám. SQL proto poskytuje mechanismus, jak na tyto chyby reagovat. Zpracování chyby musí být v bloku BEGIN-END definováno za deklarací proměnných a podmínek, ale před samotné příkazy SQL. Její syntaxe vypadá následovně:

```
DECLARE typ HANDLER FOR podminka1, podmínka2, ... příkaz;
```

V současnosti jsou povolené typy CONTINUE a EXIT. První znamená, že program v případě výskytu chyby pokračuje následujícím příkazem. Druhý definuje, že při výskytu chyby je ukončen daný blok BEGIN/END a program pokračuje za ním.

Podmínky určují, kdy se má rutina aktivovat. Dají se formulovat několika způsoby:

1. **SQLSTATE 'kod_chyby'**: Určuje konkrétní chybový kód.
2. **SQLWARNING**: Zahrnuje všechny *01nnn* stavy SQLSTATE.
3. **NOT FOUND**: Zahrnuje všechny ostatní chyby (tedy ty stavy SQLSTATE, které nezačínají 01 nebo 02).
4. **číslochybyMySQL**: Stanoví číslo chyby MySQL namísto kódu SQLSTATE.
5. **nazevpodminky:** Odkazuje na podmínku, která byla formulována příkazem DECLARE CONDITION.

Příkaz v deklaraci zpracování chyby se provede v případě výskytu chyby. Obvykle se zde použije proměnná, vyhodnocená v následujícím kódu. Příkaz musí být rovněž definován v případě použití DECLARE EXIT HANDLER.

Příklad

```
DECLARE promenna_chyba VARCHAR(50);

DECLARE duplicit_klic CONDITION FOR SQLSTATE '23000';

DECLARE CONTINUE HANDLER FOR duplicit_klic SET promenna_chyba =

'duplicita klíče';
```

Nejprve jsme na prvním řádku deklarovali proměnnou promenna_chyba. Poté jsme vytvořili podmínku duplicit_klic, která vlastně dává srozumitelný název chybovému stavu duplicity primárního klíče v tabulce (kód 23000). Na závěr jsme deklarovali zpraování pro tuto chybu (duplicit_klic).

Příklady UP

Následující procedura přidá to tabulky *knihy* nový záznam. Proceduře je nutné předat název knihy a isbn, a jméno a příjmení autora. O zbytek se procedura postará sama:

```
CREATE PROCEDURE nova_kniha (IN nazevNoveKnihy VARCHAR(255), IN
isbnNoveKnihy VARCHAR(20), IN jmenoAutora VARCHAR(20), IN prijmeniAutora
VARCHAR(40), OUT zprava VARCHAR(40))
proc: BEGIN
        -- Deklarujeme lokální proměnné
        DECLARE pocet, id_autoraKnihy, id_noveKnihy INT;
        DECLARE promenna_chyba VARCHAR(50);
        -- Deklarujeme chybovou podmínku a zprac. chyby duplicitního klíče
        DECLARE duplicit_klic CONDITION FOR SQLSTATE '23000';
        DECLARE EXIT HANDLER FOR duplicit_klic SET promenna_chyba =
        'duplicita klíče';
        -- Ověřujeme, zda-li kniha v db neexistuje - v případě pokud nemá ISBN
        IF ISNULL(isbnNoveKnihy) OR TRIM(isbnNoveKnihy) = '' THEN
           SELECT COUNT(knihy.id) FROM knihy
           INNER JOIN autori ON autori.id = knihy.id_autor
           WHERE knihy.nazev = nazevNoveKnihy AND autori.prijmeni =
           prijmeniAutora AND ISNULL(knihy.isbn) INTO pocet;
        -- Ověřujeme existenci knihy v případě že má ISBN
        ELSE
           SELECT COUNT(id) FROM knihy
           WHERE isbn = isbnNoveKnihy INTO pocet;
        END IF;
        -- Akce v případě chyby zadání, či existence knihy v db
        IF ISNULL(nazevNoveKnihy) OR TRIM(nazevNoveKnihy) = '' OR
        pocet !=0
        THEN
           SET zprava = 'Pozor. Záznam již existuje.';
           LEAVE proc;
        END IF;
        -- Zkontrolováno, můžeme zahájit proces vkládání
        -- Nejprve zjistíme, zda daný autor již v db existuje
        -- Pokud ano, načteme jeho id do proměnné id_autoraKnihy
        SELECT id FROM autori WHERE jmeno = jmenoAutora AND prijmeni =
        prijmeniAutora INTO id_autoraKnihy;
        -- Pokud autor ještě v db neexistuje, musíme jej vytvořit
```

```
        IF ISNULL(id_autoraKnihy) THEN
           INSERT INTO autori (id, jmeno, prijmeni) VALUES(NULL,
           jmenoAutora, prijmeniAutora);
        -- Pro nový záznam do tabulky knihy potřebujeme znát id autora
           SET id_autoraKnihy = LAST_INSERT_ID();
        END IF;
        -- Vložíme nový záznam do tabulky knihy
        INSERT INTO knihy (id, id_autor, nazev, isbn)
        VALUES(NULL, id_autoraKnihy, nazevNoveKnihy, isbnNoveKnihy);
        -- Pro nový záznam do tabulky stav potřebujeme znát id nové knihy
        SET id_noveKnihy = LAST_INSERT_ID();
        -- Vložíme nový záznam do tabulky stav
        INSERT INTO stav (id_knihy, zapujceno)
        VALUES(id_noveKnihy, 0);
        SET zprava = 'Vše proběhlo v pořádku';
END proc
```

Až budete proceduru vytvářet, nezapomeňte do pole oddělovače vložit zástupný znak $$.

Komentář k UP

Na první pohled je kód této UP možná důvodem zvýšeného tepu a pocitu zmaru. To v horším případě. V lepším si stačí řádek po řádku kód projít a zjistíte, že se vlastně jedná o velmi jednoduchou a celkem nenáročnou proceduru. Strukturovaný jazyk SQL vás povede kousek po kousku, takže logika věci vám bude jasná.

Musíme pochopitelně provést zkoušku. Takže zavoláme naší proceduře:

```
CALL nova_kniha('Vinetou', NULL, 'Karel', 'May', @zprava);
SELECT @zprava;
```

```
CALL nova_kniha('Zlo pod sluncem', '80-242-0985-3', 'Agatha',
 'Christie', @zprava);
SELECT @zprava;
```

Kód procedury obsahuje pouze vám známé konstrukce. Snad jen vestavěná funkce **TRIM()** je pro nás nová, ale její účel je naprosto stejný jako u funkce téhož jména v jazyce PHP. Odstraňuje mezery ze začátku a konce textového řetězce[9].

[9] Seznam řetězcových funkcí MySQL - http://dev.mysql.com/doc/refman/5.0/en/string-functions.html

Další procedura bude mít za úkol odstranění záznamu z tabulky knihy a další referenční záznam z tabulky stav. Ovšem pouze v tom případě, kdy kniha není zapůjčená:

```
CREATE PROCEDURE odstran_knihu (IN nazevKnihy VARCHAR(255), OUT oznameni
VARCHAR(255))
proc: BEGIN
        -- Deklarujeme lokální proměnné
        DECLARE knihaID, knihaZapujcena INT;
        -- Zjišťujeme, zda-li není kniha zapůjčena
        SELECT id, zapujceno FROM stav
        INNER JOIN knihy ON knihy.id = stav.id_knihy
        WHERE knihy.nazev = nazevKnihy INTO knihaID, knihaZapujcena;
        -- Pokud je zapůjčena načteme stav do proměnné a UP končí (LEAVE)
        IF knihaZapujcena = 1 THEN
           SET oznameni = 'Záznam o knize nyní nelze vymazat. Kniha je
           zapůjčena.';
           LEAVE proc;
        -- Kniha není zapůjčena, můžeme začít s vymazáním
        ELSE
           DELETE FROM knihy WHERE id = knihaID;
           DELETE FROM stav WHERE id_knihy = knihaID;
           SET oznameni = 'Záznam byl úspěšně vymazán';
        END IF;
END proc
```

A samozřejmě vyzkoušíme:

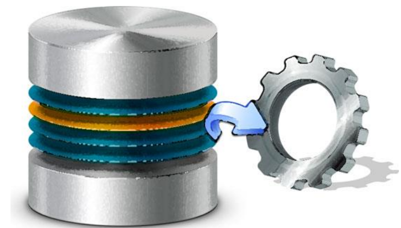
```
CALL odstran_knihu('Vinetou', @oznameni);
SELECT @oznameni;
```

TRIGGERY – předmět Databázové systémy 1

Triggery jsou velmi užitečnou pomůckou a řadí se mezi procedury spouštěné z MySQL. Oproti UP je tu však jeden podstatný rozdíl: **UP jsou spuštěny explicitním voláním (příkazem CALL). Triggery jsou spouštěny samočinně (implicitně), na základě stanovené akce.**

Triggery tak slouží pro automatické volání příkazů SQL nebo uložených procedur[10]. Podle akce, která trigger implicitně spouští je dělíme na:

1. Triggery vkládání.
2. Triggery aktualizace.
3. Triggery odstraňování.



Ještě než se pustíme do vytváření a popisu příkazu triggerů, zapamatujte si následující dva příkazy:

```
SHOW TRIGGERS;
```

Tento příkaz vám ukáže vytvořené triggery na dané databázi.

```
DROP TRIGGER jmeno_trigeru;
```

Tímto vyvoláte akci odstranění triggeru daného jména.

Tvorba triggerů

Na začátek si ukážeme první jednoduchý příklad. Ještě předtím si z tabulky **stav**, kterou jsme používali pro identifikaci zapůjčených knih, vymažte řádek s **id_knihy** 7.

```
DELETE FROM stav WHERE id_knihy = 7;
```

Náš první trigger bude mít následující úlohu:

Po vložení nového záznamu do tabulky *knihy,* trigger automaticky vloží také záznam o této knize do tabulky *stav* a nastaví hodnotu zapujceno na 0. Tj. že kniha je nezapůjčena. Úkol je tedy velmi podobný dříve vytvořené UP, ale nyní jej bude mít na starost trigger.

Při tvorbě triggeru v PHPMyAdmin postupujte následovně:

Příkaz triggeru vložte pod danou databází klasicky pod záložkou SQL do připraveného okna. Pozor ale, ještě předtím než příkaz SQL provedete, změňte Oddělovač ze středníku (;) na dvě klasická lomítka //.

[10] Pozor – triggery nemohou vracet žádné výsledky, tj. použití UP je omezené.

Trigger vložení

```
CREATE TRIGGER after_insert_knihy
AFTER INSERT ON knihy
FOR EACH ROW
BEGIN
INSERT INTO stav VALUES(NEW.id, 0);
END;
```

A nyní můžeme funkčnost triggeru otestovat. Do tabulky knihy vložíme nový záznam:

```
INSERT INTO knihy (id, id_autor, nazev, isbn) VALUES ('', 1, 'Vánoce
Hercula Poirota', '978-80-242-2255-4');
```

Náš trigger měl zajistit, že se také vložil nový záznam o této knize do tabulky **stav**. Můžeme se o tom přesvědčit:

```
SELECT nazev, CONCAT(jmeno, ' ', prijmeni) AS cele_jmeno
FROM knihy
LEFT JOIN autori ON autori.id = knihy.id_autor
      LEFT JOIN stav ON stav.id_knihy = knihy.id
WHERE stav.zapujceno = 0;
```

nazev	cele_jmeno
Andělé a démoni	Dan Brown
Databáze bez předchozích znalostí	Andrew Oppel
Pozvánka pro Hercula Poirota	Agatha Christie
Vánoce Hercula Poirota	Agatha Christie

Výborně. Náš trigger funguje. Do tabulky stav byl automaticky vložen nový záznam o naší nové knize.

Syntaxe tvorby triggeru

```
CREATE TRIGGER < nazev triggeru >
{BEFORE | AFTER}
{INSERT | DELETE | UPDATE [OF < seznam sloupcu > ] }
ON < nazev tabulky >
FOR EACH ROW
< provadene prikazy SQL >
```

Za klíčovými slovy CREATE TRIGGER následuje název triggeru.

Na druhém řádku musíme zadat, zda má být trigger proveden po úpravách (AFTER) nebo před úpravami (BEFORE) v předmětné tabulce.

Ve třetím řádku vkládáme informaci o tom, zda jde o trigger **vkládání, odstraňování** nebo **aktualizace**. Pokud jde o trigger **aktualizace**, máme možnost aplikovat trigger na jeden nebo více konkrétních sloupců. Když je zadán více než jeden sloupec, musíme jejich názvy oddělovat čárkami.

Na dalším řádku syntaxe musíte zadat klauzuli **ON** obsahující název předmětné tabulky. Tedy tabulky, pro kterou je trigger definován. **Trigger může být definován pouze pro jednu tabulku**.

Jakmile definice triggeru obsahuje více než jeden příkaz SQL, **musí být příkazy uzavřeny do bloku** BEGIN/END. V případě potřeby deklarace lokální proměnné je syntax shodná s UP.

Uvnitř SQL kódu triggeru, můžeme přistupovat jen ke sloupcům aktuálního záznamu:

OLD.nazevsloupce – vrací obsah existujícího záznamu před jeho změnou či smazáním (operace UPDATE a DELETE).

NEW.nazevsloupce – vrací obsah nového nebo změněného záznamu (operace INSERT a UPDATE).

Klauzule **FOR EACH ROW** určuje, že trigger je vyvolán při každém vložení, aktualizaci nebo odstraňování, dle své funkčnosti.

TRIGGER odstraňování

Zkusíme si další konkrétní příklad. Ještě předtím si vytvoříme tabulku *zaloha*, do níž se budou ukládat data o vyřazených knihách z tabulky knihy.

```
CREATE TABLE `prvniDB`.`zaloha` (
`id` INT( 9 ) NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`id_autor` INT( 9 ) NOT NULL ,
`nazev` VARCHAR( 255 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NOT
NULL ,
`isbn` VARCHAR( 20 ) CHARACTER SET utf8 COLLATE utf8_czech_ci NULL

) ENGINE = MYISAM CHARACTER SET utf8 COLLATE utf8_czech_ci;
```

Při odstranění záznamu o knize z tabulky knihy se vytvoří duplicitní záznam do tabulky zaloha:

```
CREATE TRIGGER before_delete_knihy
BEFORE DELETE
ON knihy
FOR EACH ROW
BEGIN
INSERT INTO zaloha VALUES(NULL, OLD.id_autor, OLD.nazev,
```

```
OLD.isbn);
END;
```

V této definici triggeru odstranění jsou oproti prvnímu triggeru vložení dva významné rozdíly. Akce vložení nového záznamu do tabulky *zaloha* musí být provedena ještě před smazáním původních dat v tabulce *knihy*. Proto jsme použili klauzuli **BEFORE**.

Druhým podstatným rozdílem je, že se odkazujeme na staré hodnoty, tj. hodnoty aktuální v tabulce *knihy* před smazáním záznamu. Proto místo aliasu **NEW**, používáme alias **OLD**. Můžeme vyzkoušet. Když odstraníme některý ze záznamů (nebo i více) z tabulky *knihy*, vytvoří se nám duplikáty v tabulce *zaloha*. Navíc konstrukce BEGIN / END hlídá, jestli k vložení zálohy došlo. Pokud by se tak nestalo, záznam se z předmětné tabulky knihy neodstraní.

Náš trigger odstranění ještě malinko vylepšíme. Určitě bychom nechtěli odstranit některý záznam o knize, a přitom ponechat odpovídající záznam v tabulce stav:

```
DROP TRIGGER before_delete_knihy;
CREATE TRIGGER before_delete_knihy
BEFORE DELETE
ON knihy
FOR EACH ROW
BEGIN
   DELETE FROM stav WHERE id_knihy = OLD.id;
   INSERT INTO zaloha VALUES(NULL, OLD.id_autor, OLD.nazev,
   OLD.isbn);
END
```

Pokud bychom chtěli dále ohlídat, zda je tato kniha zapůjčená, pak trigger není pro tento účel ideální nástroj. Sám o sobě nevrací žádné hodnoty, takže by bylo nemožné uživateli sdělit, proč záznam nelze vymazat. Pro tuto úlohu se lépe hodí předchozí konstrukce UP.

17 Zálohování databáze

Ačkoli tento studijní text není zaměřen na správu databázového systému MySQL, měli byste bezpodmínečně znát základy zálohování, což je prvořadý úkol administrátora databázového systému. Bez zálohování můžete přijít o velké množství dat z rozsáhlých tabulek. Ukážeme si dva z možných způsobů zálohování:

- Příkazy BACKUP a RESTORE
- Nástroj mysqldump

Zálohování tabulek pomocí BACKUP

Jednou z nejsnazších možností zálohování je použití příkazu **BACKUP**. Ten má nevýhodu, že funguje pouze u tabulek MyISAM. Jeho syntaxe je následující:

```
BACKUP TABLE nazev_tabulky [, nazev_dalsi_tb] TO '/cesta_zalohy';
```

Cesta zálohy musí být cestou k adresáři, do kterého chceme uložení uskutečnit, a nesmí se jednat o název souboru. Tímto způsobem se vytvoří kopie souborů .frm (definice) a .MYD (data), nikoli však souboru .MYI (index). Indexy lze znovu sestavit po obnovení databáze.

Příklad

```
BACKUP TABLE studenti TO 'c:\\zaloha_tab_studenti';
```

Na Windows musíme použít dvě zpětná lomítka, protože zpětné lomítko jako takové slouží v MySQL k zadávání speciálních znaků.

Obnovení tabulek MyISAM pomocí RESTORE

Opakem BACKUP je příkaz **RESTORE**, což je příkaz obnovující tabulky MyISAM dříve vytvořené pomocí BACKUP. Rovněž znovu vytváří indexy. Syntaxe je následující:

```
RESTORE TABLE nazev_tabulky FROM '/cesta_zalohy';
```

Příklad

```
RESTORE TABLE studenti FROM 'c:\\zaloha_tab_studenti';
```

Zálohování databáze pomocí nástroje mysqldump

Předchozí metoda je čistě založena na kopírování zdrojových souborů a navíc funguje jen u tabulek MyISAM. Tato metoda je založena na vygenerování sledu SQL příkazů potřebných k vytvoření zálohovaných tabulek.

V PHPMyAdmin můžete tohoto nástroje využít velice jednoduše. Stačí pod danou databází zvolit záložku **Export** a poté stiskem jediného tlačítka vygenerovat příslušné SQL příkazy. A to za vás vykoná nástroj **mysqldump**.

V praxi je tohoto nástroje využíváno automaticky a jsou díky němu prováděny rutinní intervalové zálohy.

Vygenerovaná data vložená do vámi vybraného souboru, nejčastěji textového s příponou SQL, má zhruba následující podobu:

```
-- phpMyAdmin SQL Dump
-- version 3.1.1
-- http://www.phpmyadmin.net
--
-- Počítač: localhost
-- Vygenerováno: Úterý 13. srpna 2011, 18:34
-- Verze MySQL: 5.1.30
-- Verze PHP: 5.2.8

SET SQL_MODE="NO_AUTO_VALUE_ON_ZERO";

--
-- Databáze: `prvnidb`
--

-- --------------------------------------------------------

--
-- Struktura tabulky `autori`
--

CREATE TABLE IF NOT EXISTS `autori` (
  `id` int(9) NOT NULL AUTO_INCREMENT,
  `jmeno` varchar(20) COLLATE utf8_czech_ci NOT NULL,Příkazy, díky nimž
jsou databáze zálohovány automaticky bývají v pravidelnou dobu spouštěny
např. s dávkových souborů .bat, v niž jsou uloženy připojovací informace a
příkazy zálohování:
...
```

18 Programování s MySQL

Skutečnou sílu databáze poznáme až v okamžiku, kdy se stane součástí nějakého informačního systému či webové aplikace. Ve spojení s MySQL se nejčastěji používají následující programovací technologie:

- Java
- C#
- PHP
- Perl
- Python
- Visual Basic
- a další, které mají dobře vyvinutá aplikační programová rozhranní pro komunikaci s MySQL.

My si dále ukážeme příklady s použitím jazyka PHP, který byste měli v základu ovládat z předchozího studia. Kombinace PHP a MySQL se staly základním stavebním kamenem většiny webových aplikací.

Zaměříme se především na rozhranní **mysqli** (mysql improved), které nahradilo starší verzi **mysql** a které je součástí dnes aktuální verze PHP5, resp. PHP7 a je podporována již od verze MySQL 4.1.2.

V našich příkladech si ukážeme základy jak procedurálního, tak objektového přístupu programování s databází MySQL. **U objektového přístupu se o funkcích hovoří jako o metodách**.

Datová spojení

Procedurální metoda:

```
<?php
$conn = mysqli_connect('localhost', 'jmeno_uzivatele', 'heslo',
 'nazev_databaze');
if (empty($conn)) {
      die ('Nepodařilo se spojit s MySQL: '.mysqli_connect_error());
}
mysqli_set_charset($conn, "utf8");
?>
```

Tento krátký skript slouží k připojení k databázovému serveru. Funkce **mysqli_connect()** zajistí spojení s databázovým serverem MySQL na základě parametrů - adresy serveru (pokud naše aplikace běží na stejném stroji jako MySQL můžeme psát localhost), jména uživatele (např. root), hesla, které má uživatel přiděleno a jména konkrétní databáze (např. prvnidb).

Objektová metoda:

V případě objektově orientovaného rozhranní, můžeme parametry připojení definovat při tvorbě objektu **mysqli** pomocí jeho konstruktoru:

```
<?php
$mysqli = new mysqli('localhost', 'jmeno_uzivatele', 'heslo',
 'nazev_databaze');
if (mysqli_connect_errno()) {
      die ('Nepodařilo se spojit s MySQL: '.mysqli_connect_error());
}
$mysqli -> set_charset("utf8");
?>
```

mysqli_connect_errno ()	Načte chybový kód posledního neuskutečněného spojení.
mysqli_connect_error ()	Načte chybovou zprávu, vztahující se k poslednímu neuskutečněnému spojení.
mysqli_set_charset()	Nastavená znaková sada pro odesílání dat klien – db, db - klient

Připojovací skript je vždy nejrozumnější uložit do zvláštního souboru a při potřebě spojení s databází jej vložit do těla dokumentu. Můžeme si tak vytvořit např. soubor *spojeni.inc.php(spojeni-oop.inc.php)*, který budeme dále připojovat do těla dalších skriptů. Jelikož je heslo uvedeno přímo v textové podobě, mělo by se vyskytovat maximálně v jednom souboru, abychom eliminovali riziko jeho zcizení.

Dotazy

SELECT

Webové aplikace jsou převážně postaveny kolem tabulek **MyISAM**, tj. nejčastěji využívají operace zobrazení výsledků. Jak už víme, k tomuto účelu je v MySQL připraven příkaz **SELECT**. Ukážeme si příklad, který vypíše seznam knih a jmen autorů z tabulky *knihy* a *autori*.

Procedurální metoda:

```
<?php
require_once "spojeni.inc.php";
$result = mysqli_query($conn, "SELECT nazev, CONCAT(jmeno, ' ',
 prijmeni) AS cele_jmeno
FROM knihy
      INNER JOIN autori ON autori.id = knihy.id_autor ORDER BY nazev")
```

```
or die ("<p>Spojení se serverem MySQL selhalo.</p>");
while ($row = mysqli_fetch_assoc($result)) {
       echo  $row['nazev']. ': '.$row['cele_jmeno']. '<br />';
mysqli_free_result($result);
mysqli_close($conn);
?>
```

Sled příkazů je velmi jednoduchý. Popíšeme si je krok za krokem:

1. Pomocí funkce **require_once** jsme do těla dokumentu vložili soubor *spojeni.inc.php*, který obsahuje připojovací údaje k MySQL a provede spojení se serverem. Vzpomenete si ještě jaká je další funkce PHP umožňující vložit do dokumentu obsah jiného souboru[11]?

2. V proměnné $result jsme odeslali SQL dotaz k databázovému serveru. Tento úkon zajišťuje funkce **mysqli_query()**.

3. Prostřednictvím cyklu *while* vypisujeme řádek po řádku, z datového typu pole. Toto pole vytváří z dat získaných z dotazu (krok 2), funkce **mysqli_fetch_assoc()**.

4. Po ukončení cyklu (výpisu všech možných jmen z tabulky autori) uvolňujeme místo v paměti – **mysqli_free_result()** a uzavíráme spojení se serverem MySQL – **mysqli_close()**.

mysqli_query ()	Odešle dotaz do databáze. Má tyto argumenty: • prvek (objekt) spojení • dotaz (řetězec SQL)
mysqli_fetch_row ()	Načítá celý výsledek naposledy spuštěného dotazu a ukládá ho v paměti v poli, pouze s číselnými indexy.
mysqli_fetch_assoc ()	Načítá celý výsledek naposledy spuštěného dotazu a ukládá ho v paměti v poli, pouze s asociativními klíči.
mysqli_fetch_array ()	Načítá celý výsledek naposledy spuštěného dotazu a ukládá ho v paměti v poli, jaks asociativními klíči, tak číselnými klíči.
mysqli_fetch_object ()	Vrací datový záznam jako objekt.
mysqli_close ()	Uzavírá spojení se severem MySQL.

Společným prvkem funkcí (mysqli_fetch_...) je, že při každém jejich volání je automaticky vrácen další záznam (nebo FALSE, pokud bylo dosaženo konce seznamu dat). Proto je možné použít cyklus WHILE, který volá tyto funkce do vymezení počtu načtených řádků.

[11] Je to funkce include().

Objektová metoda:

```
<?php
require_once "spojeni-oop.inc.php";
$result = $mysqli -> query("SELECT nazev, CONCAT(jmeno, ' ', prijmeni)
 AS cele_jmeno
FROM knihy
      INNER JOIN autori ON autori.id = knihy.id_autor ORDER BY nazev ")
or die ("<p>Spojení se serverem MySQL selhalo.</p>");
while ($row = $result -> fetch_assoc()) {
      echo  $row['nazev']. ': '.$row['cele_jmeno']. '<br />';
$result -> free();
$mysqli -> close();
?>
```

INSERT

Nyní si ukážeme, jak naopak data do tabulky vložit. Vyzkoušíme si vložit nový záznam do tabulky knihy:

Procedurální metoda:

```
<?php
require_once "spojeni.inc.php";
$result = mysqli_query($conn, "INSERT INTO knihy VALUES(NULL, 1, 'Zlo
    pod sluncem', '80-242-0985-3')")
or die ("<p>Spojení se serverem MySQL selhalo.</p>");
mysqli_close($conn);
?>
```

Objektová metoda:

```
<?php
require_once "spojeni-oop.inc.php";
$result = $mysqli -> query("INSERT INTO knihy VALUES(NULL, 1, 'Zlo pod
 sluncem', '80-242-0985-3')")
or die ("<p>Spojení se serverem MySQL selhalo.</p>");
$mysqli -> close();
?>
```

Syntaxe je velmi jednoduchá a není potřeba ji rozebírat. Stejně tomu bude také v případě odstraňování a aktualizace záznamu.

Pokud si zkoušíte uvedené příklady, určitě vás napadne zkontrolovat, zda-li také správně funguje trigger **after_insert_knihy**, který po vložení nového záznamu do tabulky knihy, vytvoří také nový záznam v tabulce stav.

DELETE

Procedurální metoda:

```
<?php
require_once "spojeni.inc.php";
$result = mysqli_query($conn, "DELETE knihy WHERE nazev = 'Zlo pod
 sluncem'")
or die ("<p>Spojení se serverem MySQL selhalo.</p>");
mysqli_close($conn);
?>
```

Objektová metoda:

```
<?php
require_once "spojeni-oop.inc.php";
$result = $mysqli -> query("DELETE knihy WHERE nazev = 'Zlo pod
 sluncem'")
or die ("<p>Spojení se serverem MySQL selhalo.</p>");
$mysqli -> close();
?>
```

Opět můžeme ověřit, jak tentokrát zafungoval trigger **before_delete_knihy**.

Další tipy

Někdy je zapotřebí z databáze pouze zjistit, kolik řádků tabulky odpovídá našemu dotazu. K tomuto účelu můžeme použít dva přístupy. První z nich využívá funkce **mysqli_num_rows().**

Procedurální

```
$result = mysqli_query($conn, "SELECT * FROM knihy");
$pocet_radku = mysqli_num_rows($result);
```

Objektový

```
$result = $mysqli -> query("SELECT * FROM knihy");

$pocet_radku = $result -> num_rows;
```

Do proměnné `$pocet_radku` byl načten počet záznamů v tabulce knihy. Docílili jsme tedy kýženého výsledku. **ALE POZOR**.

Tuto metodu byste měli používat pouze v případě, kdy chcete s daty získanými funkcí **mysql_query()** dále pracovat, např. vypisovat příkazem SELECT. Jak totiž víte, byla do paměti načtena data z celé tabulky. Zahlcuje se tak paměť a plýtvá drahocenným časem. V případě obsáhlých tabulek ani nemluvě.

Pokud potřebujete opravdu jen zjistit počet záznamů v tabulce a přitom neplýtvat časem a pamětí, použijte následující:

Procedurální

```
$result = mysqli_query($conn, "SELECT COUNT(id) FROM knihy'");

$pocet_radku = mysqli_fetch_row($result);
```

Objektový

```
$result = $mysqli -> query("SELECT COUNT(id) FROM knihy");

$pocet_radku = $result -> fetch_row();
```

Pozor ale, výsledná hodnota se nenachází v proměnné `$pocet_radku` . **Tato proměnná je datový typ pole!** Tj. kýžená hodnota je v hodnotě s indexem 0 - `$pocet_radku[0]`. Využili jsme vestavěnou agregační funkci MySQL pro počet záznamů COUNT() a jednoduše jsme tedy získali potřebný údaj.

Komplikace s triggery

Nevýhodou v používání triggerů na webu je skutečnost, že spousta poskytovatelů webhostingových služeb je z bezpečnostních důvodů na svých databázových serverech znepřístupňuje. Značně se tak komplikuje situace pro vývojáře, kteří na toto musí myslet.

Vezmeme si například náš trigger **after_insert_knihy**, který musíme nahradit. Vezmeme to pěkně popořadě:

Do tabulky knihy vkládáme nový záznam:

```
$result = mysqli_query($conn, "INSERT INTO knihy VALUES(NULL, 1, 'Zlo

pod sluncem', '80-242-0985-3')");
```

Po provedení tohoto příkazu přišel na řadu trigger **after_insert_knihy**, který automaticky vložil odpovídající záznam do tabulky *stav*. Bohužel triggery nejsou zpřístupněny a my si musíme poradit jinak.

Potřebujeme znát **id** nového záznamu, abychom jej mohli použít do tabulky *stav*. Máme dvě možnosti:

Příkazem SELECT si zjistíme hodnotu nového id v tabulce knihy a poté provedeme nové vložení INSERT do tabulky stav. Toto je samozřejmě funkční možnost, ale zbytečně zdlouhavá a vzhledem k možnostem MySQL značně amatérská.

Druhá možnost tkví ve využití nám již známe funkce **LAST_INSERT_ID()**. Celý sled bude tedy vypadat následovně:

```
$result = mysqli_query($conn, "INSERT INTO knihy VALUES(NULL, 1, 'Zlo
 pod sluncem', '80-242-0985-3')");
$result = mysqli_query($conn, "INSERT INTO stav VALUES(LAST_INSERT_ID(),
 0)");
```

Poslední vygenerovanou hodnotu vrací též funkce PHP **mysqli_insert_id()**.

Procedurální: `$last_ID = mysqli_insert_id($conn);`

Objektové: `$last_ID = $mysqli -> insert_id;`

Provedení několika příkazů SQL najednou

S novým rozhranním **mysqli** se v PHP objevila také nová funkce **multi_query()**. Jednotlivé příkazy SQL se v ní oddělují středníkem. V následujících řádcích si ji představíme v objektově orientovaném přístupu, pro nějž byla také navržena. Ovšem lze ji využívat také v procedurálním přístupu.

```
<?php
require_once "spojeni-oop.inc.php";
$sql = "SET @a=5; SET @b=10; SELECT @a + @b; SELECT nazev FROM knihy
 WHERE id=1";
$result = $mysqli -> multi_query($sql);
if($result)
      do {
            $multi = $mysqli -> store_result();
            if($multi) {
                  $radek = $multi -> fetch_row();
                  echo $radek[0];
                  $multi -> close();
            }
      }
while($mysqli -> next_result());
?>
```

Protože každý z příkazů SQL může vrátit výsledky, mění se u této metody také jejich vyhodnocení. Výsledek prvního dotazu lze načíst jednou ze dvou metod:

1. **use_result** – výsledek dotazu zůstává na straně serveru a je na klienta přenášen postupně, řádek po řádku.

2. **store_result** – výsledek je v celku okamžitě načten na klienta, což je ve většině případů efektivnější metoda.

Pro zpracování dalšího výsledku je potřeba zavolat metodu **next_result**, která vrátí TRUE nebo FALSE v závislosti na tom, zda-li jsou k dispozici ještě další záznamy. Pokud ano, je nutné je opět načíst metodou **use_result** nebo **store_result**.

V příkladu, který jsme uvedli výše, jsme do proměnné $sql načetli více příkazů SQL. Prostřednictvím proměnné $result jsme metodou **multi_query()** odeslali dotazy k serveru MySQL. V případě že proces uspěl, použili jsme cyklus d**o/while** u kterého se podmínka vyhodnocuje až na konci. To znamená, že příkazy v těle cyklu budou provedeny minimálně jednou.

V prvním a druhém kroku inicializujeme proměnné @a a @b. V třetím kroku vracíme jejich součet. Podmínka v cyklu while nás díky metodě posouvá k poslednímu příkazu, který vrací jeden ze záznamů z tabulky knihy.

Výsledek v prohlížeči

15
Vražda v Orient-Expressu

Připravené příkazy

S technikou objektového přístupu k programování se začala významně prosazovat strategie, kterým se programátoři snaží eliminovat bloky podobných programových konstrukcí na různých místech kódu. Stejně tak v případě dotazů MySQL, které bývají často shodné a liší se pouze v parametrech.

Pro tyto případy nabízí MySQL tzv. **připravené příkazy**, kdy úplný příkaz je na server zasílán pouze jednou: poté se liší už jen jeho parametry. Zmíněný postup značně eliminuje množství přenášených dat a dovoluje účinnější zpracování příkazů, protože MySQL musí analyzovat strukturu příkazu jen jednou.

Opět se zaměříme pouze na objektový přístup k věci:

```
<?php
require_once "spojeni-oop.inc.php";
$order = $mysqli -> prepare("INSERT INTO autori (id, jmeno, prijmeni
) VALUES (NULL, ?, ?)");
$order -> bind_param("ss", $jmeno, $prijmeni);

$jmeno = "Jakub";
$prijmeni = "Vrána";
$order -> execute();

$jmeno = "John";
$prijmeni = "Grisham";
$order -> execute();

$order -> close();

$last_ID = $mysqli -> insert_id;
echo $last_ID;
?>
```

Po rutinním vložení připojovacího skriptu je nejprve potřeba specifikovat příkaz SQL (metodou **$mysqli -> prepare**). Do příkazu se místo parametrů umístí otazníky. Výsledkem bude objekt *mysqli_stmt*, základna pro všechny další operace. Tento objekt jsme uložili do proměnné $order.

Dalším krokem je pomocí metody **bind_param()** navázání parametrů na proměnné PHP. Ještě předtím je potřeba uvést jejich datový typ. Datový typ vyjádříme prvním písmenem daného typu. Viz následující tabulka:

Identifikátor	Datový typ
i	integer (celé číslo)
d	double (desetinné číslo)
s	string (textový řetězec)
b	BLOB (binární data)

Vlastní spuštění příkazu SQL provedeme tak, že uložíme požadované hodnoty do příslušných proměnných a spustíme dotaz metodou **$order -> execute()**. V momentě, kdy už nebude objekt *mysqli_stmt* potřeba, uvolníme ho z paměti voláním metody **close**(). Tímto také dáme serveru jasnou informaci o tom, že již nebudou následovat žádné další příkazy a proto MySQL může smazat připravený dotaz.

Připravené příkazy SELECT

Ihned si ukážeme konkrétní příklad:

```
<?php
require_once "spojeni-oop.inc.php";
$order = $mysqli -> prepare("SELECT nazev, isbn FROM knihy");
$order -> execute();
$order -> store_result();
$order -> bind_result($nazev, $isbn);
while($order -> fetch()) {
      echo $nazev. ": " .$isbn. "<br />";
}
$order -> close();
?>
```

Po provedení připraveného příkazu metodou **execute()** a načtení dat do paměti metodou **store_result()**, navážeme pomocí metody **bind_result()** sloupce výsledku do proměnných PHP. Cyklus *while* provede příkazy ve svém těle tolikrát, kolikrát metoda **fetch()** načte jeden řádek po druhém z výsledků dotazu. V prohlížeči poté uvidíme přibližně následující:

```
Vražda v Orient-Expressu: 978-80-242-2097-0
Oznamuje se vražda: 978-80-242-2161-8
Andělé a démoni: 80-7203-787-0
Šifra mistra Leonarda: 80-86518-62-0
Databáze bez předchozích znalostí: 80-251-1199-7
```

Metody **bind_result()** a **bind_param()** je možné také kombinovat. A to například pro příkazy typu SELECT … FROM … WHERE sloupec = ?:

```
<?php
require_once "spojeni-oop.inc.php";
$order = $mysqli -> prepare("SELECT nazev, isbn FROM knihy WHERE
 id_autor = ?");
$order -> bind_param("i", $id_autor);
// První příkaz SQL
$id_autor = 1;
$order -> execute();
$order -> store_result();
$order -> bind_result($nazev, $isbn);
echo "Počet záznamů pro autora s id ".$id_autor. " je ". $order ->
 num_rows. "<br />";
while($order -> fetch()) {
      echo $nazev. ": " .$isbn. "<br />";
}
// Druhý příkaz SQL
$id_autor = 2;
$order -> execute();
$order -> store_result();
$order -> bind_result($nazev, $isbn);
echo "Počet záznamů pro autora s id ".$id_autor. " je ". $order ->
 num_rows. "<br />";
while($order -> fetch()) {
      echo $nazev. ": " .$isbn. "<br />";
}
$order -> close();
?>
```

Neformátovaný výstup v prohlížeči:

```
Počet záznamů pro autora s id 1 je 2
Vražda v Orient-Expressu: 978-80-242-2097-0
Oznamuje se vražda: 978-80-242-2161-8
Počet záznamů pro autora s id 2 je 2
Andělé a démoni: 80-7203-787-0
Šifra mistra Leonarda: 80-86518-62-0
```

Krátké shrnutí

Představili jsme si základ propojení PHP aplikací s databází MySQL. V praxi můžete využívat prakticky všech technik, které jsme se naučili v rámci jednoduchých dotazů, přes spojení, až po zamykání tabulek či transakce na tabulkách InnoDB. Nikdy ale nezapomínejte, že dobrá aplikace vždy závisí na dobře navržené struktuře databáze. Teprve poté přichází na řadu programování uživatelského rozhraní. Nicméně i programátor se musí držet specifik navržené databáze a navrhovat dotazy takové, které budou efektivně využívat možností databáze a indexů.

19 Příloha 1

Příklad normalizace databáze

Mějme tabulku Knihy, která má následující podobu a počítá se dále s jejím rozšířením o další záznamy. Proveďte její analýzu a případně úpravu, podle prvních tří normalizačních forem:

nazev	vydavatel	rok	autor_1	autor_2	autor_3
Mistrovství v PHP 5	Computer Press	2008	Andi Gutmans	Stig S. Bakken	Derick Rethans
Myslíme v MySQL 4	Grada	2003	Ian Gilfillan		
MySQL 5	Computer Press	2007	Michael Kofler		
Návrh databází	Grada	2003	Michael J. Hernandez		
SEO v PHP	Computer Press	2008	Jamie Sirowich	Cristian Darie	

První normalizační forma nám říká:

- Jsou eliminovány sloupce se stejným obsahem.
- Jsou definovány všechny klíčové atributy.
- Všechny atributy závisejí na primárním klíči.

Otázka: Jsou eliminovány sloupce se stejným obsahem?

Odpověď: Nejsou. Sloupce autor_1, autor_2, autor_3 jsou sloupce se stejným obsahem.

Otázka: Jsou definovány všechny klíčové atributy?

Odpověď: Nejsou. Vzhledem ke skutečnosti, že se tabulka bude rozšiřovat o další záznamy, můžou se objevit duplicitní názvy knih. Název knihy jako klíč tedy nestačí. Dokonce se můžou objevit aktualizované vydání knih u stejného vydavatele a ještě v témže roce vydání.

Otázka: Všechny atributy závisejí na primárním klíči?

Odpověď: Ne. Primární klíč není jasně stanoven.

Vytvoříme tedy tabulku odpovídající první normalizované formě.

id_knihy	nazev	vydavatel	rok	autor
1	Mistrovství v PHP 5	Computer Press	2008	Andi Gutmans
2	Mistrovství v PHP 5	Computer Press	2008	Stig S. Bakken
3	Mistrovství v PHP 5	Computer Press	2008	Derick Rethans
4	Myslíme v MySQL 4	Grada	2003	Ian Gilfillan
5	MySQL 5	Computer Press	2007	Michael Kofler
6	Návrh databází	Grada	2003	Michael J. Hernandez
7	SEO v PHP	Computer Press	2008	Jamie Sirowich
8	SEO v PHP	Computer Press	2008	Cristian Darie

Nyní má tabulka knihy přijatelnější podobu a můžeme přistoupit k druhé normalizované formě.

Druhá normalizační forma nám o tabulce říká:

- Nachází se v první normalizované formě.
- Nezahrnuje žádné částečné závislosti (kdy nějaký atribut závisí jen na části primárního klíče, resp. každý neklíčový atribut je plně závislý na primárním klíči).
- Kdykoli se opakují identické obsahy některých sloupců v různých záznamech, tabulka by měla být rozdělena.

Otázka: Nachází se v první normalizované formě?

Odpověď: Ano, tabulka se nachází v první normalizované formě.

Otázka: Nezahrnuje žádné částečné závislosti, kdy nějaký atribut závisí jen na části primárního klíče?

Odpověď: Nezahrnuje částečné závislosti, protože tabulka nemá složený primární klíč.

Otázka: Opakují se identické obsahy týž sloupců v různých záznamech?

Odpověď: Ano, jsou zde opakující se identické záznamy ve sloupci **nazev**. Tabulka by měla být rozdělena.

id_knihy	nazev	vydavatel	rok
1	Mistrovství v PHP 5	Computer Press	2008
2	Myslíme v MySQL 4	Grada	2003
3	MySQL 5	Computer Press	2007
4	Návrh databází	Grada	2003
5	SEO v PHP	Computer Press	2008

Vznikla nám nová tabulka knihy, která se oprostila od identických záznamů, ale také ztratila autora. Pro autory musíme vytvořit novou, samostatnou tabulku.

id_autora	jmeno	prijmeni
1	Andi	Gutmans
2	Stig S.	Bakken
3	Derick	Rethans
4	Ian	Gilfillan
5	Michael	Kofler
6	Michael J.	Hernandez
7	Jamie	Sirowich
8	Cristian	Darie

V nové tabulce jsme rozdělili jména autorů na jmeno a prijmeni, abychom eliminovali chybné **vypočítané pole.** Primárním klíčem je identifikátor **id_autora**.

Jelikož se jedná o vztah M:N, tj. více různých knih může mít stejného autora a naopak, více různých autorů může být spoluautorem jedné knihy, musíme vytvořit vazební tabulku:

id_knihy	id_autora
1	1
1	2
1	3
2	4
3	5
4	6
5	7
6	8

Primárním klíčem je kombinace obou atributů.

Třetí normalizační forma nám o tabulce říká:

- Je ve druhé normalizované formě.
- **Neobsahuje žádné tranzitivní neboli přechodné závislosti** (tj. když nějaký neklíčový atribut závisí na primárním klíči prostřednictvím jiného neklíčového atributu). Jiné vyjádření téhož říká, že všechny neklíčové atributy jsou navzájem nezávislé, ostatní musíme odstranit (přesunout do samostatné tabulky).

Otázka: Nacházejí se všechny tabulky v druhé normalizované formě?

Odpověď: Ano, všechny tabulky se nachází v druhé normalizované formě.

Otázka: Jsou všechny atributy přímo závislé na primárním klíči?

Odpověď: Ne. Existuje zde závislost atributu vydavatel. Ten se v různých záznamech opakuje.

V praxi nepředpokládáme, že jedna kniha bude mít více vydavatelů. Proto opakující se řetězcové hodnoty ze sloupce vydavatel nahradíme cizími číselnými klíči k nové tabulce **vydavatele**.

Naše normalizovaná databáze tak bude po aplikaci tří hlavních normalizačních forem vypadat následovně:

Tabulka **knihy**:

id_knihy	nazev	id_vydavatele	rok
1	Mistrovství v PHP 5	1	2008
2	Myslíme v MySQL 4	2	2003
3	MySQL 5	1	2007
4	Návrh databází	2	2003
5	SEO v PHP	1	2008

Vazební tabulka:

id_knihy	id_autora
1	1
1	2
1	3
2	4
3	5
4	6
5	7
5	8

Tabulka **autori**:

id_autora	jmeno	prijmeni
1	Andi	Gutmans
2	Stig S.	Bakken
3	Derick	Rethans
4	Ian	Gilfillan
5	Michael	Kofler
6	Michael J.	Hernandez
7	Jamie	Sirowich
8	Cristian	Darie

Tabulka **vydavatele**:

id_vydavatele	nazev_vydavatele
1	Computer Press
2	Grada

20 Příloha 2

Součástí MySQL pro Linux i pro Windows je řádkový klient **mysql.exe** a řádkový administrátor **mysqladmin.exe**. Oba soubory se nacházejí v podadresáři /bin/ instalace MySQL serveru.

Chceme-li je využit, otevřeme konzolu příkazového řádku. Následující ukázky provedu pro Windows, ovšem v systému Linux je postup téměř shodný.

Spustíme řádkového klienta mysql.exe

Nejprve se přemístíme do adresáře s programem. Je to adresář /bin/

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Verze 6.0.6002]
Copyright (c) 2006 Microsoft Corporation. Všechna práva vyhrazena.

C:\Users\Honza>cd c:\xampp\mysql\bin

c:\xampp\mysql\bin>_
```

V dalším kroku zadáme název řádkového klienta a parametry připojení k serveru MySQL:

```
> mysql.exe -h nazev_hostitele -u uživatelské_jmeno -p
```

```
C:\Windows\system32\cmd.exe - mysql.exe -u root -p
Microsoft Windows [Verze 6.0.6002]
Copyright (c) 2006 Microsoft Corporation. Všechna práva vyhrazena.

C:\Users\Honza>cd c:\xampp\mysql\bin

c:\xampp\mysql\bin>mysql.exe -u root -p
Enter password: _
```

Pokud se připojujete k lokálnímu serveru, není potřeba zadávat parametr **–h.**

Po potvrzení příkazu budete vybídnuti k zadání hesla. Jakmile tak provedete, jste úspěšně připojeni k serveru MySQL jako klient.

```
C:\Windows\system32\cmd.exe - mysql.exe -u root -p
Microsoft Windows [Verze 6.0.6002]
Copyright (c) 2006 Microsoft Corporation. Všechna práva vyhrazena.

C:\Users\Honza>cd c:\xampp\mysql\bin

c:\xampp\mysql\bin>mysql.exe -u root -p
Enter password: **********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 18
Server version: 5.1.30-community MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> _
```

Zadávání příkazů SQL

V řádkovém klientu můžete přímo zadávat příkazy nebo SQL dotazy. Příkazy odešlete ke zpracování odklepnutím řádku. SQL dotazy mohou být samozřejmě delší, než na jeden řádek a proto poslední řádek prostě ukončete středníkem.

Zobrazení databází

```
SHOW DATABASES;
```

Výběr databáze

```
USE prvnidb;
```

```
C:\Windows\system32\cmd.exe - mysql -u root -p
mysql> USE prvnidb;
Database changed
mysql>
```

Zobrazení tabulek v aktuální databázi

Zobrazíme tabulky v aktuální databázi – vybrané příkazem USE:

```
SHOW TABLES;
```

```
C:\Windows\system32\cmd.exe - mysql -u root -p

mysql> USE prvnidb;
Database changed
mysql> SHOW TABLES;
+--------------------+
| Tables_in_prvnidb  |
+--------------------+
| autori             |
| autori2            |
| knihy              |
| stav               |
| studenti           |
| zbozi              |
+--------------------+
6 rows in set (0.00 sec)

mysql> _
```

V režimu příkazového řádku je pochopitelně nutné znát perfektní syntaxi jazyka SQL. Můžeme se tak přímo dotazovat.

Vypíšeme obsah a strukturu tabulky knihy

```
SELECT * FROM knihy;
```

```
C:\Windows\system32\cmd.exe - mysql -u root -p
mysql> SELECT * FROM knihy;
+----+-----------+-------------------------------------+-------------------+
| id | id_autor  | nazev                               | isbn              |
+----+-----------+-------------------------------------+-------------------+
|  1 |         1 | Vražda v Orient-Expressu            | 978-80-242-2097-0 |
|  2 |         1 | Oznamuje se vražda                  | 978-80-242-2161-8 |
|  3 |         2 | And?lé a démoni                     | 80-7203-787-0     |
|  4 |         2 | Šifra mistra Leonarda               | 80-86518-62-0     |
|  5 |         3 | Databáze bez p?edchozích znalostí   | 80-251-1199-7     |
+----+-----------+-------------------------------------+-------------------+
5 rows in set (0.00 sec)

mysql> _
```

Struktura tabulky

```
DESCRIBE TABLE knihy;
```

Můžete zkoušet různé dotazy. Od jednoduchých, přes spojení, až po užití uložených procedur. Klient příkazového řádku může mít problémy se zobrazením některých znaků. V tom případě vyzkoušejte změnit použití sady znaků příkazem **SET names**.

Např. SET names UTF8;

21 Příloha 3

Příklady spojení pomocí JOIN

V této části si procvičíme spojení pomocí konstrukce JOIN. Využijeme tabulky z **přílohy 1**. K tabulce **knihy** ještě přidáme sloupec o počtu knih, které jsou v knihovně k dispozici (počet volných knih): (SQL kód pro vytvoření a naplnění tabulek je v závěru této přílohy)

Tabulka **knihy**:

id_knihy	nazev	id_vydavatele	rok	volne
1	Mistrovství v PHP 5	1	2008	2
2	Myslíme v MySQL 4	2	2003	1
3	MySQL 5	1	2007	3
4	Návrh databází	2	2003	1
5	SEO v PHP	1	2008	0

Vazební tabulka **kniha_autor**:

id_knihy	id_autora
1	1
1	2
1	3
2	4
3	5
4	6
5	7
5	8

Tabulka **autori**:

id_autora	jmeno	prijmeni
1	Andi	Gutmans
2	Stig S.	Bakken
3	Derick	Rethans
4	Ian	Gilfillan
5	Michael	Kofler
6	Michael J.	Hernandez
7	Jamie	Sirowich
8	Cristian	Darie

Tabulka **vydavatele**:

id_vydavatele	nazev_vydavatele
1	Computer Press
2	Grada

Dále si databázi doplníme o tabulku **klienti**, která bude obsaovat seznam klientů naší knihovny a k ní vazební tabulku, která bude spojovat v relaci M:N tabulku **knihy** a **klienti**.

Tabulka **klienti**:

id_klienta	jmeno	prijmeni	telefon	email
10	Roman	Tyčka	654876987	romantycka@web.cz
11	Alena	Mourková	987789456	alca@wed.cz
12	Olga	Rounová	345654876	rounova@zkolaven.cz

Vazební tabulka **kniha_klient**:

id_knihy	id_klienta
1	10
2	11
4	10
5	10
5	12

Primárním klíčem v tabulce **klienti** je očividně identifikátor **id_klienta.** Ve vazební tabulce je to jako obvykle složený primární klíč z obou polí.

V tabulce **knihy** je pole volne vždy přizpůsobeno aktuálnímu počtu kusů knih, které jsou k dispozici. Takže např. celkový počet titulu SEO v PHP je 2, ale oba jsou momentálně zapůjčené.

Úkol 1

Ke každému vydavatelství vypište počet knih, které mu náleží z tabulky knihy.

Řešení

Jednoduchá úloha, která vyžaduje spojení pouze dvou tabulek (knihy a vydavatele). Abychom získali počet kusů pro jednotlivá vydavatelství, použijeme klauzuli pro seskupení - **GROUP BY**.

```
SELECT nazev_vydavatele, COUNT(id_knihy) AS pocet_knih
FROM vydavatele
INNER JOIN knihy ON knihy.vydavatel = vydavatele.id_vydavatele
GROUP BY knihy.vydavatel
ORDER BY pocet_knih;
```

nazev_vydavatele	pocet_knih
Grada	2
Computer Press	3

Úkol 2

Vypište kompletní záznamy o knihách (nazev, vydavatel, rok, celé jméno autorů a počet volných kusů).

Řešení

Zde bude potřeba provést spojení více než dvou tabulek. Budeme potřebovat tabulky **knihy**, **vydavatele** a **autori**, a musíme využít údajů z vazební tabulky **kniha_autor**:

Vzhledem k tomu, že máme tabulky propojeny klíči se shodnými názvy, můžeme použít přirozené spojení **NATURAL JOIN**, které samo hledá sloupce stejného názvu, na kterých by tabulky spojilo. Tím se nám rovněž zpřehlední syntax příkazu:

```
SELECT nazev, nazev_vydavatele, rok, CONCAT( jmeno, ' ', prijmeni ) AS
 autor, volne
FROM knihy
      NATURAL JOIN vydavatele
      NATURAL JOIN kniha_autor
      NATURAL JOIN autori
ORDER BY nazev;
```

Je vidno, že dotaz je celkem jednoduchý, ale s výsledkem bude potřeba ještě pracovat.

nazev	nazev_vydavatele	rok	autor	volne
Mistrovství v PHP 5	Computer Press	2008	Andi Gutmans	2
Mistrovství v PHP 5	Computer Press	2008	Stig S. Bakken	2
Mistrovství v PHP 5	Computer Press	2008	Derick Rethans	2
Myslíme v MySQL 4	Grada	2003	Ian Gilfillan	1
MySQL 5	Computer Press	2007	Michael Kofler	3
Návrh databází	Grada	2003	Michael J. Hernandez	1
SEO v PHP	Computer Press	2008	Jamie Sirowich	0
SEO v PHP	Computer Press	2008	Cristian Darie	0

Asi byste neradi takovou tabulku zobrazili klientům k prohlížení či objednávce knih. Je potřeba předchozí dotaz ještě vylepšit.

Podívejme se třeba na katalog vědecké knihovny v Olomouci. Při hledání knihy „PHP6 : Programujeme profesionálně“. Autoři knihy jsou uvedeni všichni přímo v názvu knihy, za titulem a lomítkem. V poli autor je uveden pouze první autor.

Název

PHP 6 : programujeme profesionálně / Ed Lecky-Thompson, Steven D. Nowicki ; [překlad Ondřej Gibl]

Autor

Lecky-Thompson, Ed, 1981-

Náš dotaz tedy formulujeme podobně. Zobrazíme pouze hlavního (prvního) autora. Stačí, když přidáme klauzuli seskupení podle názvu knihy:

```
SELECT nazev, nazev_vydavatele, rok, CONCAT( jmeno, ' ', prijmeni ) AS
 autor, volne
FROM knihy
      NATURAL JOIN vydavatele
      NATURAL JOIN kniha_autor
      NATURAL JOIN autori
```

```
GROUP BY nazev
ORDER BY nazev;
```

nazev	nazev_vydavatele	rok	autor	volne
Mistrovství v PHP 5	Computer Press	2008	Andi Gutmans	2
Myslíme v MySQL 4	Grada	2003	Ian Gilfillan	1
MySQL 5	Computer Press	2007	Michael Kofler	3
Návrh databází	Grada	2003	Michael J. Hernandez	1
SEO v PHP	Computer Press	2008	Jamie Sirowich	0

Nicméně můžete oponovat, že i toto řešení je omezené a MySQL přece musí mít něco, čím zobrazí i ostatní autory. A máte pravdu. O této funkci jsme se zmínili už dříve. Je to funkce **GROUP_CONCAT()**. Ta nám umožní spojit hodnoty skupiny autorů do jednoho pole:

```
SELECT nazev, nazev_vydavatele, rok, GROUP_CONCAT( CONCAT( jmeno, ' ',
prijmeni ) SEPARATOR ', ' ) AS autor, volne
FROM knihy
      NATURAL JOIN vydavatele
      NATURAL JOIN kniha_autor
      NATURAL JOIN autori
GROUP BY nazev
ORDER BY nazev;
```

nazev	nazev_vydavatele	rok	autor	volne
Mistrovství v PHP 5	Computer Press	2008	Andi Gutmans, Stig S. Bakken, Derick Rethans	2
Myslíme v MySQL 4	Grada	2003	Ian Gilfillan	1
MySQL 5	Computer Press	2007	Michael Kofler	3
Návrh databází	Grada	2003	Michael J. Hernandez	1
SEO v PHP	Computer Press	2008	Jamie Sirowich, Cristian Darie	0

Úkol 3

V třetí úloze tohoto cvičení máte pomocí příkazu SELECT, vypsat všechny knihy, které má zapůjčeny klient Roman Tyčka. Podoba výpisu záznamu má být stejná, jako v předchozím příkladu.

Řešení

Je jasné, že použijeme předchozí příkaz, který ale budeme muset doplnit o další spojení a výběrovou podmínku WHERE.

```
SELECT nazev, nazev_vydavatele, rok, GROUP_CONCAT( CONCAT( jmeno, ' ',
 prijmeni )
SEPARATOR ', ' ) AS autor
FROM knihy
      NATURAL JOIN kniha_autor
      NATURAL JOIN autori
      NATURAL JOIN vydavatele
NATURAL JOIN kniha_klient
WHERE id_klienta = 10
GROUP BY nazev
ORDER BY nazev;
```

nazev	nazev_vydavatele	rok	autor
Mistrovství v PHP 5	Computer Press	2008	Andi Gutmans, Stig S. Bakken, Derick Rethans
Návrh databází	Grada	2003	Michael J. Hernandez
SEO v PHP	Computer Press	2008	Jamie Sirowich, Cristian Darie

Úkol 4

V této úloze máme vypsat všechny knihy v knihovně (včetně autorů a jména vydavatele), uvést jejich celkový počet a počet volných exemplářů.

Řešení

Úkol není jednoduchý a tak vše provedeme pěkně popořadě. Na začátek se pokusíme vypsat celkové počty knih. Bude potřeba získat údaje z tabulky **kniha_klient**, abychom zjistili počty zapůjčených titulů. Tyto poté přičteme k volným, abychom získali celkový počet knih:

```
SELECT knihy.nazev, (COUNT( kniha_klient.id_knihy ) + volne) AS celkem,
 volne
FROM knihy
      LEFT JOIN kniha_klient USING (id_knihy)
GROUP BY knihy.id_knihy
ORDER BY knihy.nazev;
```

nazev	celkem	volne
Mistrovství v PHP 5	3	2
Myslíme v MySQL 4	2	1
MySQL 5	3	3
Návrh databází	2	1
SEO v PHP	2	0

Sami se pokuste objasnit, z jakého důvodu jsme v předchozím spojení nemohli použít přirozené spojení NATURAL JOIN, ačkoli v obou tabulkách existují sloupce shodného jména.

Dostali jsme se k zdárnému výsledku. Nyní je potřeba doplnit dotaz o další prvky, které doplní jména autorů a název vydavatelství:

```
SELECT knihy.nazev, vydavatele.nazev_vydavatele, GROUP_CONCAT(CONCAT(
autori.jmeno, ' ',autori.prijmeni) SEPARATOR ', ') AS autor, (
COUNT( kniha_klient.id_knihy ) + knihy.volne
) AS celkem, knihy.volne
FROM knihy
      LEFT JOIN kniha_klient ON knihy.id_knihy = kniha_klient.id_knihy
      NATURAL JOIN vydavatele
      INNER JOIN kniha_autor ON knihy.id_knihy = kniha_autor.id_knihy
      NATURAL JOIN autori
GROUP BY knihy.id_knihy
ORDER BY knihy.nazev;
```

V případě spojení na tabulce **kniha_autor** nelze použít přirozené spojení NATURAL JOIN, protože sloupec na kterém bychom spojovali tabulky (id_knihy) je použit také v tabulce **kniha_klient**. Optimalizátor MySQL by tedy nevěděl, které z tabulek má spojit. Museli bychom volit vhodné aliasy nebo přímo určit sloupce pomocí spojení INNER JOIN, jak jsme učinili. A nebo úplně obrátit pořadí spojení:

```
SELECT knihy.nazev, vydavatele.nazev_vydavatele, GROUP_CONCAT(CONCAT(
autori.jmeno, ' ',autori.prijmeni) SEPARATOR ', ') AS autor, (
COUNT( kniha_klient.id_knihy ) + knihy.volne
) AS celkem, knihy.volne
FROM knihy
      NATURAL JOIN vydavatele
      NATURAL JOIN kniha_autor
      NATURAL JOIN autori
LEFT JOIN kniha_klient ON knihy.id_knihy = kniha_klient.id_knihy
GROUP BY knihy.id_knihy
ORDER BY knihy.nazev;
```

Výsledek bude v obou případech následující:

nazev	nazev_vydavatele	autor	celkem	volne
Mistrovství v PHP 5	Computer Press	Andi Gutmans, Stig S. Bakken, Derick Rethans	5	2
Myslíme v MySQL 4	Grada	Ian Gilfillan	2	1
MySQL 5	Computer Press	Michael Kofler	3	3
Návrh databází	Grada	Michael J. Hernandez	2	1
SEO v PHP	Computer Press	Jamie Sirowich, Cristian Darie	4	0

Určitě jste si v tabulce všimli nesrovnalostí. Ano, celkové údaje jsou spočteny špatně (chybu nedělá MySQL, ale nevhodně položený dotaz). Chyby jsou v těch řádcích, kde kniha obsahuje více než jednoho autora. Chyba je zapříčiněna údaji z vazební tabulky **kniha_autor,** kterou jsme při první spojení nepoužili.

Když MySQL pomocí optimalizátoru spojuje jednotlivé tabulky, v určité fázi dojde k následujícímu dočasnému řešení:

nazev	nazev_vydavatele	rok	autor	volne
Mistrovství v PHP 5	Computer Press	2008	Andi Gutmans	2
Mistrovství v PHP 5	Computer Press	2008	Stig S. Bakken	2
Mistrovství v PHP 5	Computer Press	2008	Derick Rethans	2
Myslíme v MySQL 4	Grada	2003	Ian Gilfillan	1
MySQL 5	Computer Press	2007	Michael Kofler	3
Návrh databází	Grada	2003	Michael J. Hernandez	1
SEO v PHP	Computer Press	2008	Jamie Sirowich	0
SEO v PHP	Computer Press	2008	Cristian Darie	0

Údaje z této tabulky slouží ke spojení s tabulkou **kniha_klient**. Pro **každý z řádků** (jsou 3) titulu *„Mistrovství v PHP5“* je přidělena hodnota z tabulky **kniha_klient** odpovídající počtu vypůjčených knih tohoto titulu. MySQL si tak v mezipaměti uloží pro každý řádek titulu jedničku. Ale jelikož v závěru seskupí sloupce pomocí funkce GROUP_CONCAT() (ikdyž se jedná o sloupce jmen autorů), sečte i hodnoty počtu vypůjčených knih. S volnými knihami tak vypočte číslo 5. Stejně tak v případě „SEO v PHP“. Pro každý záznam tohoto titulu určí 2 vypůjčené knihy. Proto je výsledek 4.

Samostatný úkol

Jak asi tušíte, vaším úkolem bude vytvořit dotaz, který vrátí tabulku ve formátu, jaký požadujeme:

nazev	nazev_vydavatele	autor	celkem	volne
Mistrovství v PHP 5	Computer Press	Andi Gutmans, Stig S. Bakken, Derick Rethans	3	2
Myslíme v MySQL 4	Grada	Ian Gilfillan	2	1
MySQL 5	Computer Press	Michael Kofler	3	3
Návrh databází	Grada	Michael J. Hernandez	2	1
SEO v PHP	Computer Press	Jamie Sirowich, Cristian Darie	2	0

Další

1. Vypište všechny tituly, které jsou zapůjčené.

2. Vypište všechny knihy (včetně autorů), které vydalo nakladatelství GRADA.

3. Vypište všechny zapůjčené knihy, které jsou vydány po roce 2004.

4. Vypište všechny knihy, které mají pouze jednoho autora.

SQL příkazy pro vytvoření databáze s tabulkami

Vytvořte si databázi **druhadb**. Na této databázi spusťte následující příkazy:

```
CREATE TABLE IF NOT EXISTS `autori` (
  `id_autora` int(11) NOT NULL AUTO_INCREMENT,
  `jmeno` varchar(20) COLLATE utf8_czech_ci NOT NULL,
  `prijmeni` varchar(40) COLLATE utf8_czech_ci NOT NULL,
  PRIMARY KEY (`id_autora`)
) ENGINE=MyISAM  DEFAULT CHARSET=utf8 COLLATE=utf8_czech_ci
AUTO_INCREMENT=9 ;

--
-- Vypisuji data pro tabulku `autori`
--

INSERT INTO `autori` (`id_autora`, `jmeno`, `prijmeni`) VALUES
(1, 'Andi', 'Gutmans'),
(2, 'Stig S.', 'Bakken'),
(3, 'Derick', 'Rethans'),
(4, 'Ian', 'Gilfillan'),
(5, 'Michael', 'Kofler'),
(6, 'Michael J.', 'Hernandez'),
(7, 'Jamie', 'Sirowich'),
(8, 'Cristian', 'Darie');


--
-- Struktura tabulky `klienti`
--

CREATE TABLE IF NOT EXISTS `klienti` (
  `id_klienta` int(11) NOT NULL AUTO_INCREMENT,
  `jmeno` varchar(20) COLLATE utf8_czech_ci NOT NULL,
  `prijmeni` varchar(40) COLLATE utf8_czech_ci NOT NULL,
  `telefon` int(11) NOT NULL,
  `email` varchar(50) COLLATE utf8_czech_ci NOT NULL,
  PRIMARY KEY (`id_klienta`)
) ENGINE=MyISAM  DEFAULT CHARSET=utf8 COLLATE=utf8_czech_ci
AUTO_INCREMENT=13 ;

--
-- Vypisuji data pro tabulku `klienti`
--
```

```
INSERT INTO `klienti` (`id_klienta`, `jmeno`, `prijmeni`, `telefon`,
`email`) VALUES
(10, 'Roman', 'Tyčka', 654987321, 'romantycka@web.cz'),
(11, 'Alena', 'Mourková', 654789123, 'alca@wed.cz'),
(12, 'Olga', 'Rounová', 456321456, 'rounova@zkolaven.cz');


--
-- Struktura tabulky `kniha_autor`
--

CREATE TABLE IF NOT EXISTS `kniha_autor` (
  `id_knihy` int(11) NOT NULL,
  `id_autora` int(11) NOT NULL,
  PRIMARY KEY (`id_knihy`,`id_autora`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_czech_ci;

--
-- Vypisuji data pro tabulku `kniha_autor`
--

INSERT INTO `kniha_autor` (`id_knihy`, `id_autora`) VALUES
(1, 1),
(1, 2),
(1, 3),
(2, 4),
(3, 5),
(4, 6),
(5, 7),
(5, 8);

--
-- Struktura tabulky `kniha_klient`
--

CREATE TABLE IF NOT EXISTS `kniha_klient` (
  `id_knihy` int(11) NOT NULL,
  `id_klienta` int(11) NOT NULL,
  PRIMARY KEY (`id_knihy`,`id_klienta`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_czech_ci;

--
-- Vypisuji data pro tabulku `kniha_klient`
--

INSERT INTO `kniha_klient` (`id_knihy`, `id_klienta`) VALUES
(1, 10),
(2, 11),
(4, 10),
(5, 10),
(5, 12);


--
-- Struktura tabulky `knihy`
--

CREATE TABLE IF NOT EXISTS `knihy` (
  `id_knihy` int(11) NOT NULL AUTO_INCREMENT,
  `nazev` varchar(255) COLLATE utf8_czech_ci NOT NULL,
```

```
  `id_vydavatele` tinyint(4) NOT NULL,
  `rok` smallint(6) NOT NULL,
  `volne` tinyint(4) NOT NULL,
  PRIMARY KEY (`id_knihy`),
  KEY `vydavatel` (`id_vydavatele`)
) ENGINE=MyISAM  DEFAULT CHARSET=utf8 COLLATE=utf8_czech_ci
AUTO_INCREMENT=6 ;

--
-- Vypisuji data pro tabulku `knihy`
--

INSERT INTO `knihy` (`id_knihy`, `nazev`, `id_vydavatele`, `rok`, `volne`)
VALUES
(1, 'Mistrovství v PHP 5', 1, 2008, 2),
(2, 'Myslíme v MySQL 4', 2, 2003, 1),
(3, 'MySQL 5', 1, 2007, 3),
(4, 'Návrh databází', 2, 2003, 1),
(5, 'SEO v PHP', 1, 2008, 0);


--
-- Struktura tabulky `vydavatele`
--

CREATE TABLE IF NOT EXISTS `vydavatele` (
  `id_vydavatele` int(11) NOT NULL AUTO_INCREMENT,
  `nazev_vydavatele` varchar(50) COLLATE utf8_czech_ci NOT NULL,
  PRIMARY KEY (`id_vydavatele`)
) ENGINE=MyISAM  DEFAULT CHARSET=utf8 COLLATE=utf8_czech_ci
AUTO_INCREMENT=3 ;

--
-- Vypisuji data pro tabulku `vydavatele`
--

INSERT INTO `vydavatele` (`id_vydavatele`, `nazev_vydavatele`) VALUES
(1, 'Computer Press'),
(2, 'Grada');

/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
```

Diskuze k datovým typům

Nebudeme nyní uvažovat nad typem tabulek. Pochopitelně našly by se důvody proč u některých tabulek volit InnoDB, ale tyto přeskočíme a shodneme se na MyISAM. Diskutujme k datovým typům a případné volbě indexů v tabulce **knihy**:

id_knihy	nazev	id_vydavatele	rok	volne
1	Mistrovství v PHP 5	1	2008	2
2	Myslíme v MySQL 4	2	2003	1
3	MySQL 5	1	2007	3
4	Návrh databází	2	2003	1
5	SEO v PHP	1	2008	0

Primární klíč

Tabulka má jasný **primární klíč** v podobě atributu **id_knihy**. Jedná se tedy o číselné pole, pro něž je potřeba datový typ odpovídající jejímu možnému rozsahu. V závislosti na něm vybíráme z možností TINYINT, SMALLINT, MEDIUMINT nebo INTEGER. Hodnota identifikátoru bezesporu nebude záporná, proto přidáme volbu UNSIGNED.

Z hlediska zajištění integrity je pochopitelně nezbytné, aby všechna pole v jiných tabulkách, kde primární klíč tabulky knihy plní roli cizího klíče (nebo část primárního klíče – vazební tabulky), měla shodný datový typ!

[12]**Ideální volba:** Pro rozsáhlou tabulku se jeví jako ideální INTEGER UNSIGNED.

Název – CHAR nebo VARCHAR

Rozhodovat se jistě budeme mezi typy **CHAR** a **VARCHAR**. Rozhodnutí volby datového typu u sloupce **nazev** ovlivní, zda tabulka bude formátu statického či dynamického. Sloupce typu CHAR jsou omezeny maximální délkou řetězce 255 znaků. U VARCHAR je rozsah podstatně delší. S CHAR získáme rychlejší statickou tabulku, s VARCHAR ušetříme více místa na disku. Kdyby názvy některých knih mohly překročit délku 255 znaků, museli bychom se přiklonit k typu VARCHAR. Pokud by dlouhé názvy nehrozily a dospějeme k závěru, že diskové prostory uložení naší databáze jsou dostatečné, vybereme volbu CHAR. Získáme mnohem rychlejší přístup k datům v poli nazev.

Pokud umožníme uživatelům hledat tituly podle názvu, bude pochopitelně nezbytné vytvořit na poli nazev **index**. Je na zvážení, zda vystačíme s ordinálním indexem, či bude potřeba fulltextový index. Toto rozhodnutí musí vycházet z podrobných zátěžových testů, rozsahu tabulky a způsobu možného vyhledávání.

Ideální volba: Budeme-li uvažovat nad rozsáhlou tabulkou, s dostatečným diskovým prostorem, jeví se jako ideální CHAR(M) a Ordinální index.

ID vydavatele

Datový typ tohoto sloupce musí vycházet z datového typu primárního klíče v tabulce **vydavatele**. V tabulce knihy má toto pole roli **cizího klíče**. Jelikož vydavatelů nebude taková spousta jako knih, měli bychom se přiklonit k číselnému typu menšího rozsahu. Nejlépe asi SMALLINT.

[12] **Ideální volba**: Bereme v úvahu konkrétní příklad s dostatečným diskovým prostorem

Vytvořit na tomto poli také ordinální index? To je správná otázka. Pokud by nastala situace, že bychom potřebovali vypisovat knihy podle vydavatelů velmi často (dotazy SELECT … WHERE id_vydavatele = …), pak ano. Vytvoříme ordinální index. Navíc tento cizí klíč slouží pro spojení s tabulkou vydavatele, takže vše hraje pro jeho užití.

Ovšem pokud by k takovým výpisům docházelo jen sporadicky, např. pouze pro kontrolu správce a uživatelé by viděli název vydavatele pouze v detailu knihy, pak se raději indexu vyhneme.

Ideální volba: SMALLINT UNSIGNED.

Rok

Jasné **číselné pole**. Volba **SMALLINT UNSIGNED**.

Volné knihy

Opět číselné pole. Určitě nepředpokládáme, že bychom k jednomu titulu měli více jak 128 exemplářů, takže volba **TINYINT**.

Dále se pokusme diskutovat k tabulce **klienti**:

id_klienta	jmeno	prijmeni	telefon	email
10	Roman	Tyčka	654876987	romantycka@web.cz
11	Alena	Mourková	987789456	alca@wed.cz
12	Olga	Rounová	345654876	rounova@zkolaven.cz

Primární klíč

Primární klíč je opět jasný. Číselný typ v potřebném rozsahu. Uživatelů knihovny můžou být tisíce. V závislosti na tom vybíráme z možností SMALLINT, MEDIUMINT nebo INTEGER. Bude-li nám kupříkladu jasno, že uživatelů bude více než 1000, ale nepřesáhne počet 60.000, můžeme zvolit hodnotu SMALLINT UNSIGNED (0 – 65 535). **Ale pozor.** Pokud budeme některé uživatele mazat a jiné zase přidávat, hodnota automatické inkrementace může max. hranici překročit. S tímto je potřeba počítat a proto volit datový typ větší. Tedy MEDIUMINT.

Z hlediska zajištění integrity je pochopitelně nezbytné, aby všechna pole v jiných tabulkách, kde primární klíč tabulky klienti plní roli cizího klíče (nebo část primárního klíče – vazební tabulky), měla shodný datový typ!

Ideální volba: Pro rozsáhlou tabulku se jeví jako ideální MEDIUMINT UNSIGNED.

Jméno a Příjmení – CHAR nebo VARCHAR

Tabulka **klienti** obsahuje více polí řetězcových typů. Je opět na zvážení, zda použít typ **CHAR** nebo **VARCHAR**. Opět bychom se měli rozhodovat na základě faktorů diskového prostoru, variability délky řetězců a frekvence jejich editace. Jelikož jména a příjmení klientů se patrně nebudou objevovat v podmínkách vyhledávání (určitě ne často) a editace těchto údajů prakticky nehrozí, můžeme se přiklonit k typu VARCHAR. V opačném případě je lépe volit typ CHAR.

Ať už zvolíme tak či onak, maximální počet znaků musíme nastavit na spolehlivou hodnotu. Například **VARCHAR(8)** pro jméno, by nebylo dobrou volbou. Klientku se jménem Miroslava bychom v databázi uložili s mužským Miroslav.

Ideální volba: Jméno – VARCHAR(30), příjmení VARCHAR(50).

Telefon

Číselné pole. Asi nejlépe INTEGER, nebo BIGINT. Pokud byste chtěli ukládat i státní předvolbu, např. + 420, bylo by lépe pro toto zřídit vlastní pole.

Ideální volba: INTEGER UNSIGNED.

Email

Řetězcový typ. A opět dilema CHAR nebo VARCHAR. Rozhodnutí už musí vycházet z celkového konceptu našeho návrhu a z kritérií, které jsou uvedeny v závěru.

Budeme-li navíc uvažovat, že email bude pro administrátory knihovny mimo primárního klíče prvotním identifikačním znakem klienta, zřídíme na tomto poli ordinální index.

Ideální volba: Vzhledem k tomu, že v tabulce jsme již použili sloupce VARCHAR, přikloníme se opět k VARCHAR(70). Volba ordinální indexu je také jasná, protože email bude patrně i sloužit jako základní přihlašovací údaj klienta do naší knihovny[13].

Jako poslední se podíváme na tabulku **autori**:

id_autora	jmeno	prijmeni
1	Andi	Gutmans
2	Stig S.	Bakken
3	Derick	Rethans
4	Ian	Gilfillan
5	Michael	Kofler
6	Michael J.	Hernandez
7	Jamie	Sirowich
8	Cristian	Darie

Tato tabulka může být opět velmi rozsáhlá. Zamysleme se nad sloupci **jmeno** a **prijmeni.** Je více než pravděpodobné, že necháme uživatele vyhledávat knihy podle jejich autorů. Ovšem neradi byste se spoléhali pouze na příjmení. Ideální volbou by v tomto případě byl vícesloupcový index, který vytvoříte na poli **prijmeni** a **jmeno** v uvedeném pořadí.

```
CREATE INDEX prijmeni_jmeno ON autori (prijmeni, jmeno);
```

Ovšem pozor. Aby byl index využit, musí se mu přizpůsobit také dotazy.

Pokud programátor zpřístupní dotazy typu

```
SELECT … FROM autori WHERE jmeno = …
```

[13] V případě ukládání hesla klienta do databáze, bychom měli využívat hešovacích funkcí (např. MD5)

index se nebude moci využít, protože jako první je uveden v indexu (prijmeni_jmeno) sloupec ***prijmeni***. Pro všechny dotazy, které nevyužívají **prijmeni** jako hodnotu pro filtrování, spojování nebo řazení, bude tento index nepoužitelný. Aby byl index využíván, musíme se odkazovat na sloupec uvedený úplně vlevo.

Kritéria volby

Volba datových typů je vždy velmi důležitým faktorem při návrhu databáze a mělo by ji předcházet detailní testování. Zejména v případě návrhu tabulek, u nichž je pravděpodobnost velkého vytížení. V základu se řídíme dle následujících kriterií:

- Diskový prostor.
- Maximální rozsah hodnot (jak číselné, tak řetězcové).
- Frekvence editace údajů (u řetězců – bude-li docházet ke fragmentaci).
- Využití polí ve vyhledávacích podmínkách (rychlejší MyISAM statické tabulky než dynamické).

Snažte se rovněž nekombinovat v záznamech různé datové typy. Místo tel. čísla ve +420 111 222 333, pro které byste volili např. typ VARCHAR, číslo rozdělme na samostatné pole SMALLINT pro předvolbu, a INTEGER pro samotné číslo.

22 Příloha 4

Manuál MySQL:

http://dev.mysql.com/

Bez manuálu MySQL se neobejdete. Studijní text neobsahuje všechny důležité aspekty návrhu databáze a využití SQL v databázovém systému MySQL. Vše potřebné naleznete tam.

Odkazy na funkce MySQL

Funkce pro práci s řetězci:

http://dev.mysql.com/doc/refman/5.0/en/string-functions.html

Numerické funkce:

http://dev.mysql.com/doc/refman/5.0/en/numeric-functions.html

Funkce větvení kódu:

http://dev.mysql.com/doc/refman/5.0/en/control-flow-functions.html

23 Příloha 5

MySQL WorkBench

MySQL Workbench je grafická konzola pro práci se serverem MySQL a databázemi. Kompletní manuál (v angličtině) najdete na linku níže. Doporučuji vám si tento nástroj minimálně vyzkoušet.

http://dev.mysql.com/doc/workbench/en/index.html

Literatura

OPPEL, A. *Databáze bez předchozích znalostí.* Brno : Computer Press, 2006. ISBN 80-251-1199-7.

HERNANDEZ, M. J. *Návrh databází.* Praha : Grada, 2006. ISBN 80-247-0900-7.

SHELDON, R. *SQL začínáme programovat.* Praha : Grada, 2005. ISBN 80-247-0999-6.

SCHNEIDER, R. S. *MySQL. Oficiální průvodce tvorbou, správou a laděním databází.* Praha : Grada, 2006. ISBN 80-247-1516-3.

KOFLER, M. *Mistrovství v PHP5.* Brno : Computer Press, 2007. ISBN 978-80-251-1502-2.

GILFILLAN, I. *Myslíme v MySQL 4.* Praha : Grada, 2003. ISBN 80-247-0661-X.

HERNANDEZ, M. J., VIESCAS, J. L. *Myslíme v jazyku SQL.* Praha : Grada, 2004. ISBN 80-247-0899-X.