Základy SQL

Nemo

8.5.2010

Ve verzi 0.01 bez odborných oprav a revize. Stávající verze je pouze orientační dokument pro budoucí opravy a dodělání.

Obsah

Úvod

Než začneme přemýšlet nad tím, co je SQL - **Structured Query Language** - a jak funguje databáze, musíme si uvědomit, jak pracují vrstvy pod ní, na čem databázi spouštíme. Kdybychom neznali náš systém, nemůžeme vůbec vědět, jak k databázi přistoupit, jak ji vůbec zapnout. Je velký rozdíl mezi SQL v Microsoft Access a v aplikovaných SQL na systému Oracle.

V této knize si jednoduše povíme základ práce s daty v databázi, jak nainstalovat databázi a jak pracovat s databází pomocí SQL. Nebudeme se vyhýbat aplikacím, kolem kterých se databáze točí. Naučíme se nakreslit relace pomocí jednoduchých diagramů. Budeme se snažit pochopit, jak jednotlivé relace vytvořit pomocí SQL. Řekneme si, jak data vybírat a jaký je rozdíl mezi daty a informacemi. Uvědomíme si, jak spravovat přístup jednotlivých uživatelů k databázi.

Vše budeme dělat se zachováním pravidel, kterým se říká normalizační a pokusíme se je pochopit tak, aby nám doopravdy pomohli. Na závěr vyzkoušíme, jak která databáze pracuje, a pro pokročilé vyzkoušíme, jak se k databázi připojit a pracovat s ní pomocí oblíbeného PHP, Visual Basicu a dalších. Nezapomene však na naprosté základy, které platí ve všech databázích, a proto se podíváme i na Openoffice databázi a právě i na Microsoft Office Access.

Nikdo by neměl ztrácet naději, protože i kdyby nepochopil, k čemu je připojení PHP k databázi, vždy se může radovat, že ví, jak databáze fungují. Není totiž jen databáze, která pracuje na vzdáleném serveru na systému Oracle, ale i malá uživatelská databáze, s kterou může pracovat každý z nás, v podobě Microsoft Office a Openoffice.

Skladování dat

Skladování dat je velmi důležitá záležitost. Ať chceme nebo ne, při naší práci na počítači neustále vytváříme a upravujeme spousty dat, ať je to souřadnice umístění našich ikon na pracovní ploše, či vytváření textových dokumentů ve formátu knih. Naše virtuální světy jsou plné nekonečných řad dat, které někdy ani nevytváříme, jen s nimi pracuje náš operační systém.

Představme si, jak postupuje nás počítač při požadavku pouhé webové stránky. Náš dotaz zformuluje vyhledávač a vyšle jej po síti. Dotaz přijde na jiný počítač, kde se zpracuje, vyhledají se data, které dotaz vyžaduje, a odešlou se zase zpět, ke zdroji. Kolik ale režijních dat je kolem? Nikdy nemůžeme přesně vědět, co všechno naše data jsou.

Data skladujeme na pevných úložištích. Můžeme je nazývat pevné disky, nebo spíše pevná datová úložiště, protože pevné disky nejsou pouhá a jediná media, které uchovávají data. Známe dynamické paměti a statické paměti. V našem případě si stačí uvědomit, že data jsou uložena uvnitř strukturovaného systému, který obsahuje systém adres a popisovačů, kde jsou data uchovávána. Systém, kterým ukládáme data pomocí operačního systému, se nazývá souborový systém.

Souborových systémů je veliké množství, v této knize zdůrazníme pouze nejpoužívanější souborové systémy, protože databáze je spíše nástavbou, než přímým souborovým systémem.

FAT – file allocation table – je jeden z prvních souborových systémů, které bylo možno použít v prastarých DOS systémech. Dodnes je ale používán a patentován firmou Microsoft. Mnohdy je jediným pojítkem mezi systémy, protože je bez problému připojitelný na systémy Linux a Microsoft bez jakéhokoliv chybného zápisu. Koncept jeho programové části je znám, proto se dá jednoduše používat.

NTFS - New Technology File system – je o mnoho pokročilejší a stále se vyvíjí. Pracuje na Windows platformách a podporuje přímé sdílení už ve svém standardu. Proto, když nastavíme sdílení a pak disk přeneseme, budou stále data sdílena. Podporuje všechny možné proprietární funkce Microsoftu, třeba jako dynamické disky a jiné věci, kterými se nemusíme zabývat už jen kvůli neustálému vývoji.

EXT – extended file system – je čistě otevřenou záležitostí UNIX like systémů, proto se s ním setkáme na všech možných distribucích Linuxu. Je velice úspěšný už jen kvůli své žurnálovací povaze, která využívá celý disk ne od začátku a pomalu přidává, ale utvoří části zapsaných dat rovnoměrně na disku.

Reiser – je s příponou FS a používá se také na Linuxových platformách. Má přelomovou technologii, která dovoluje popsat disk naprosto celý. Bohužel byl jeho vývoj pozastaven kvůli Hansy Reaseru, který byl obviněn a odsouzen za zabití své ženy. Během pobytu ve vězení nemůže dále pracovat na tomto zajímavém souborovém systému. (Dějiny informačních technologií jsou až na výjimky bez krve a násilí, proto se nebojte a čtěte dál.)

ZFS – je moderní souborový systém vyvinutý firmou Sun Microsystem. Používá se na systému Solaris, který

je jedním z posledních „čistých“ UNIXů. Proslavil se svojí stabilitou a rychlým zavedením do nového systému. Setkáme se s ním na velkých datových úložištích, proto je zde o něm zmínka, protože spousta databází pracuje na ohromných clusterech UNIXU s ZFS souborovým systémem.

Databáze je v tomto ohledu nástavba ležící na souborovém systému ale vždy bychom měli mít na paměti, na čem databáze pracuje. Někdy nám souborový systém pomůže k rychlejší práci s pevným úložištěm a databáze je pak velmi dynamická. Někdy nás může pomalý a starý souborový systém omezit svojí maximální velikostí souboru. Pokud na FAT vložíme soubor větší, než 4 GB, nebudeme s ním vůbec moci pracovat už právě kvůli omezení FAT tabulky. Ohromné databáze jsou většinou větší než 4 GB, proto je dost nevhodné pracovat s databází na FAT souborovém systému.

Hromadná úložiště

Můžeme jimi rozumět velká datová pole, na které se vejde přes TB dat. Můžeme jich dosáhnout spojením jednotlivých disků nebo spojením jednotlivých počítačů. V praxi si můžeme takové spojení představit jako více než jeden pevný disk propojený v určené funkci.

Propojení disků říkáme RAID - Redundant Array of Independent Disks – pole, které vznikne právě propojením více jak jednoho disku. Máme dvě možnosti, jak tohoto stavu docílit. Buď můžeme nastavit softwarové pole, či pomocí hardwaru, kde je nutný vnější modul, zásuvná karta nebo vestavěný integrovaný obvod na základní desce.

Softwarová pole fungují pouze za pomoci programu, který pracuje na vysoké vrstvě operačního systému, tedy práce dělení funguje až při plném běhu operačního systému. Tím ho dělá nepoužitelný pro instalaci operačního systému, tedy na Windows. Windows pracuje s logickými, primárními a rozšířenými oddíly, tedy svazky.

Dynamický svazek – může být odpojen od systému – klasický svazek

Spojitý svazek – JBOD oddíl, prodloužení jednoho oddílu na další disky

Prokládaný svazek – prodlouží svazek, ale data jsou ukládána střídavě

Další RAID určuje Windows jen pro serverové operační systémy. Linux podporuje v základním nastavení všechny možná zapojení RAID.

Hardwarové nastavení pracuje s přidáním nezávislé karty, či s podílením se dalšího procesoru na základní desce. Tím rozšíříme možnost připojení dalších disků a uzpůsobíme na RAID zapojení. Vše se děje na velmi nízké vrstvě, proto se na takové pole může nainstalovat operační systém bez jakýchkoliv problémů. Nastavení probíhá dle výrobce karty v BIOSu zásuvné karty, či BIOSu základní desky, pokud podporuje RAID.

RAID O – zřetězení, prokládání, někdy také JBOD

RAID 1 – zrcadlení mezi disky

RAID 3 – minimálně na dva disky se ukládají data a na třetí se uloží paritní data, která se dají použít při obnově jednoho vypadlého disku.

To jsou základní zapojení RAID disků. Dále je podporováno těžší zapojení RAID 2, 4, 5, 6. Další typy jsou víceméně prokládanými typy uložení dat. Tedy RAID 01 a RAID 10.

Přenosný standard

Databáze je pro nás něco, co může skladovat naše data, v čem je můžeme zachovávat bez jakéhokoliv strádání. Některé operační systémy dokonce podporují kontakt s databází pro získání dat. Dnes je daleko rychlejší uložit data ve formě obrázků na webový server, kde se rychleji budou nahrávat z databáze, než přímo ze souboru.

Databáze může být přenosná. Můžeme kopírovat celou databázi, dokonce pomocí SQL vybrat vše, co je v naší databázi, a přenést na jiný počítač. To bude pro nás něco přenosného, něco, co si můžeme vzít pomocí SQL a přenést. Tento nápad je typický pro databáze, protože jinak by neměly smysl.

Zkusme si teď představit, co vlastně je SQL a databáze. Databáze může být skříň, kde je spousta karet vedených dle stejného klíče. Takové karty mohou obsahovat údaje o pacientech s poli o jménu, příjmení a dalších. Podle naprosto stejných zásad funguje i počítačová databáze. Do ordinace přikoupí novou skříň pro rozšíření databáze, my přikoupíme nový počítač pro ukládání dalších dat naší databáze. V ordinaci zavedou novou kartu a my vytvoříme nový záznam. V ordinaci začnou hledat podle názvu pacienta

v databázi a my použijeme?

A teď přišla chvíle na naše SQL, které je dotazovacím jazykem databáze. Teď jen jednoduše ukážu, jak moc se sestřička nadře při vybírání všech karet databáze, zatímco my použijeme jeden příkaz, který vše udělá za nás.

Příkaz ve fyzické databázi: „Sestři, vyberte prosím všechny záznamy, chci se podívat, kolik tu máme pacientů!“

Nemusíme komentovat nadšení sestřičky, která začne vytahovat jednu kartu za druhou.

Příkaz v počítačové databázi: „SELECT * FROM pacienti ;“

Stačil nám jeden výraz, jeden regulérní výraz a název tabulky. Počítač se s námi nebude hádat a vybere záznamy. SELECT je prakticky to samé jako „vyber“. Výrazem FROM určíme odkud a pacienti je jméno tabulky. Hvězdička je vlastně regulérní výraz pro vše, protože znamená libovolný počet znaků a jakýkoliv znak. O regulérních výrazech ještě bude řeč.

Příklady

V úvodu berte tyto příklady jako spíše uvědomění, do čeho jste se pustili při čtení této knihy.

- Co je databáze?
- Co znamená SQL?
- Na čem je uložena databáze?
- Proč jsme si v úvodu pověděli o hromadných úložištích a souborových systémech?

(poznámka autora: Na poslední otázku neodpovídejte prosím takto: Aby autor zaplnil místo ve své knize! :-))

Historie SQL

Dříve, než se zamyslíme nad tím, jak vše vzniklo, měli bychom pochopit, co k tomu všemu lidstvo postrčilo. My sami máme od dávnověku neskonalou touhu hromadit a ukládat nekonečná množství dat a informací. Informace kde, kdy a kdo zemřel, informace i životě světců, či informace o počtu vlastněných kamenů. Číslo, které označuje počet kamenů, je pak datem, které využijeme v informaci: „Hugo má sto kamenů!“ V představě společnosti jsou nám tedy data platná, jedině pokud známe jejich informační význam anebo pokud nám jej někdo sdělí.

Postupem času se však člověk neomezoval na pouhé malé číslice, ale potřeboval také vědět, co ke které číslici přidělit. Takové Zemské desky pak obsahovaly spoustu informací o půdě a neležela v nich jen pouhá data, které by nikomu nic neřekla. Následně bylo potřeba nějakým rámcovým způsobem dělit vše, co je potřebné. Proto vznikly různé dotazníky, tedy formuláře, kterými se plnily tabulky třeba ve skříni u místního doktora. Zdravotní sestra pak jen hledala podle klíče, kterým může být křestní jméno, v databázi a následně vytáhla složku identickou rámcově se složkou jiného pacienta a podala ji doktorovi, který už věděl, kde má hledat kolonku s minulou nemocí pacienta. To byl pravý přelom ve všem, co doposud lidstvo zažilo.

Jak jsme nastínili již v úvodu, způsob dotazování je úplně jiný, než u fyzických databází, ale hodně se mu přibližuje. Naše SQL tedy vznikalo naprosto přirozenou formou, jako vznikaly databáze, jen už v počítačové podobě.

Počítačové databáze se někdy ani trochu neliší od fyzických svojí skladbou dat, ale rozdílná je celková koncepce ukládání a práce s daty. Počítače v historii uměly pracovat s menším objemem dat než dnes, ale s velikostí kolem dvou budov jsme mohli docílit uložení až milionu záznamů do počítačové databáze.

Příchod IBM

Na začátku dvacátého století našeho letopočtu v USA vznikla potřeba evidence občanů na základě jejich jednoznačného identifikačního čísla. Představme si to jako naše rodné číslo. Pokud si představíme centralizovanou databázi fyzického charakteru, musela by obsahovat přes miliony listů papíru, které by obsahovali pouhé jméno, příjmení a identifikační číslo. Prakticky šlo vše zprovoznit, ale fyzicky by budovy, které by nesly ohromné stohy papíru, zabíraly ohromné místo a koncepce složení dat by byla nepřehledná už jen kvůli hledání v nekonečných halách listů. Ráno byste přišli hledat list o uprchlém trestanci a druhý den večer byste jej konečně našli, zatímco on už by prchal do Mexika.

To byla pravá chvíle pro počátky firmy IBM, která se chopila příležitosti a začala pracovat na historicky první počítačové databázi vedenou pod vládou USA. Tenkrát se vše zapisovalo na děrný štítek, což jen potvrzuje naši teorii o datech, protože pro počítač je díra na pásku umístěná na určitém místě pouhý kus dat, ale když jej přeformuloval a my jej zasadili do informace, už dával naprostý smysl. Tato databáze obsahovala přes dvacet šest milionů záznamů v děrných štítcích. Vše bylo na rozmezí digitalizace a automatizace, nemůžeme si to vše představovat jako sněť integrovaných obvodů, protože ty v té době prostě ještě nebyly. Hlavní bylo, že práce s tímto počítačem byla efektní a poměrně rychlá. (Trestanec byl tedy chycen těsně před hranicemi a ne v Mexiku.)

IBM dodnes představuje jednoho z hlavních inovátorů počítačového trhu. Dnes můžeme znát její databázový počin DB2, který je spíše takovou stálicí, ale to není vše. Dodnes vděčíme této firmě za odhalení práce s daty. Byla vymyšlena celá teorie o databázích, do které přispěly všechny tyto začínající firmy a hlavně reálný svět, ne žádné fantasie, kterými oplývají spisovatelé, ale čirá alegorie světa tvoří databáze.

Nárůst ORACLE

ORACLE byl první název veliké databáze, která měla představovat jeden z prvních pokusů o komerční využití databází. Vše vznikalo pro pozdější sálové počítače a firma vznikala prakticky do píky. Její nárůst přišel až po velké světové krizi, kterou ORACLE přečkal už jen s tím, kde všude se dalo použít. ORACLE už od začátku byl schopen plně pracovat s prastarými UNIX platformami a následně s příchodem PC rovnou přejít ze sálů do menších počítačů. ORACLE si získal velkou oblibu u uživatelů už jen kvůli ohromné implementaci nových trendů, jako jsou třeba datové typy.

S novými verzemi jsme se vždy mohli dočkat vylepšení dalších a dalších služeb. Při vzniku programovacího jazyka Java se velmi záhy objevilo i možné připojení hned na ORACLE, proto i dnes, kdy je možná daleko přívětivější použít MySQL, které je od stejné firmy, je Java a ORACLE velmi přiblížené a kvůli rychlosti obou platforem se můžeme dočkat velice dobrého efektu. Spousta firem tak používá oblíbenou a rychlou Javu a data urychluje dobrou databází.

ORACLE si získal podporu i u nemajetných firem, protože i pro ně má některé verze databáze. Korunu na hlavu společnosti ORACLE nasazuje také fakt, že instalace na jakkoliv platformu PC vůbec je možná, proto můžeme vidět ORACLE na UNIX s dokonalým souborovým systémem, či na Microsoft Windows XP, které není ani serverový operační systém.

V neposlední řadě nabízí ORACLE vlastní akademii pro studenty, čímž si vytváří stálou klientelu uživatelů přímo vyučených na jejich výrobcích.

Tím vším se ORACLE jako společnost dostala za Microsoft v ohledu výnosů za své služby. Ať řekneme sociální sítě, či národní registry, pro svou pružnost a životaschopnost je určitě tato databáze vedena pod ORACLE.

Potřeba pro OPEN technologie

Postupem času se databáze tak moc přiblížili normálním uživatelům, že už nebylo třeba zavádět miliony zápisů, ale třeba jen stovky, či tisíce. Můžeme znát přímo uživatelsky příhodné databáze, jako jsou Microsoft Office Access, které ale nemůžeme jednoduše začlenit do běhu serveru pro webové aplikace, ale je pro normální uživatele lehce použitelné. Co třeba udělají firmy, které nemají tak silný peněžní základ, ale potřebují databázi pro vedení svých webových stránek.

Toho se velice rychle chytily ostatní firmy, které pak tvořily jednotlivé platformy pomocí ostatních programů. Příkladem si můžeme říct třeba trojici PHP + Apache + MySQL – tato trojice je v základě opensource technologie, takže s ní může pracovat každý a nemusí za to platit, i když jsou verze, kdy se platit musí, ale to už je jiná kapitola.

MySQL dokonce nabízí velice zajímavé funkce, které v jiných databázích přímo nenajdeme. Můžeme pracovat třeba s automatickým přičítáním, které se jinde řeší pomoc sekvence, ta je mnohdy mnohem složitější. MySQL se svou otevřeností někdy předstihuje ostatní databáze. Nabízí dobrou přenosnost mnohdy i pomocí jednotlivých příkazů. V praxi si to můžeme představit tak, že napíšeme jeden příkaz, kterým vytvoříme SQL soubor a ten jednoduše přeneseme jinam, tam spustíme a soubor přehraje všechna data do nové databáze. To a mnoho dalšího dělá MySQL velice zajímavé.

Další databází je PostgreSQL, které je také velice oblíbenou otevřenou alternativou, protože je také velice dobře přenosné a nabízí zajímavou možnost správy databáze. Pokud tedy databázi už poměrně rozumíme, PostgreSQL můžeme nalézt jako správci na platformách Unix až Windows.

Pokud se bavíme o open, tedy otevřených, technologiích, musíme myslet na jejich pravé využití. Software je prostě pro všechny a každý má nárok s ním bez poplatků pracovat a nakládat. Nemůžeme si ale takové kusy kódu utvořené v otevřených aplikacích nechat patentovat. Jednoduše už jen kvůli tomu, že to naše právní norma neumožňuje, ale hlavně kvůli tomu, že postup a práce je veřejná. Naším cílem by nemělo být zpoplatnění služeb, ale otevření našich služeb. Samozřejmě, že je vždy zachováno duševní vlastnictví, ale kusy skriptu našeho SQL budou vždy pod otevřenou licencí. Prakticky jsou neprodejné a jejich prodáváním bychom ubírali z práv vlastníka původního kódu. Proto, když chceme pracovat skutečně komerčně, musíme použít komerční software.

Microsoft a jeho software pro „lidi“

Microsoft přišel jako velká vlna do světa PC. Už od vytvoření DOS počítačů uživatelům spousta věcí chyběla. Můžeme zmínit pouhý fakt, že systém byl poměrně dost nestabilní a jeho závady byly v porovnání v té době drahým UNIXem dost hloupé a neřešitelné. I přesto Microsoft z počátku vyhrál boj o místo v každém počítači každé domácnosti. Byla za tím na tu dobu výborná reklama a také jisté přiblížení i uživatelům, kteří nebyli zrovna počítačově naladění.

Databáze společnosti Microsoft v tomto ohledu ale poměrně zaspaly. Přišly právě za pět minut dvanáct, kdy už na velkých webových serverech pracovaly ORACLE a IBM databáze. Navíc, když už někdo databázi od Microsoftu nakoupil, musel oželet stabilitu a omezení některých SQL příkazů. Cena, kterou tedy vložil uživatel do levnější databáze, byla vykoupena nestabilitou a nemožností některých úkonů. I přesto to bylo někdy jediné řešení pro střední firmy, které databázi potřebovaly. Pro vývojáře pracující s programovacími jazyky od Microsoftu byla zase tato možnost mnohdy jedinou, jak propojit jejich aplikaci s databází.

Postupem času se povedlo většinu závad zcela odbourat a dnes je Microsoft SQL server nesoucí databázi Microsoftu vážným konkurentem běžných ORACLE databází a otevřených MySQL, či PostgreSQL databází. Podporuje totiž mnohdy spoustu užitečných analýz, diagramů a propojení na další své produkty. Vývojáři tak mohou poskytnout mnohdy velice rychle zprávu o databázi svému vedení a stejně tak pracovat rychle s databází ve svých programech.

Microsoft otevřeně vydává omezenou verzi pro studenty, či zvědavé uživatele, takže také trochu vychází vstříc jiným. Bohužel si ale o přenosnosti můžeme nechat zdát. Pokud začneme v ORACLE, nebo Microsoft SQL serveru pracovat s čistě proprietárními věcmi, nemůžeme čekat, že je pak bez úpravy přeneseme jinam. Někdy ale můžeme být mile překvapeni, protože SQL je skutečně spojujícím standardem.

Microsoft ale přišel i s jinými databázemi, daleko bližšími běžnému uživateli než serverové verze. Mluvíme o sadách Office, ve kterých můžeme nalézt databázi Access. I v ní platí SQL, i když mnohdy velice „přitažené za vlasy“. To však nemění nic na tom, že jako databáze, je mezi uživateli ještě dost pochopitelná a jednoduchá. Když uživatel neví, jednoduše použije průvodce, kterými Microsoft překypuje ve všech příležitostech. Průvodce pak doplňuje návrhové zobrazení a různé jiné novinky. Proto se můžeme setkat s databází uloženou na přenosném mediu, se kterou pracují nadšenci, kteří ale neumí pracovat s většími databázemi. Možnost vytvářet hezké vstupy do databází pomocí formulářů, je někdy příjemná více než dost. Běžný uživatel pak může vládnout nad malou databází, která je přenosná na Microsoft SQL server a zase zpět. Můžeme se setkat s takzvanými sharepointy a jinými Microsoft věcmi, které někdy i velice pomohou.

SEQUEL a SQL

SEQUEL - Structured English Query Language – byl prakticky prvním jazykem databází. Jeho cílem bylo (i dnes v SQL je) docílit jednoduchosti, přenositelnosti a snadného naučení. English, jak můžeme číst, byl hlavně proto, že jeho termíny byly skutečně jen anglické, to však jsou v SQL i dnes. V té době představoval poměrně dost zajímavý projekt. Společnosti IBM jej velice rychle implementovala do svých databází a každý, kdo jej kdy viděl, měl poměrně dost rychlou možnost se jej naučit, už jen kvůli opravdové významové shodě výrazů SEQUEL a anglických výrazů. Cílem bylo někdy i přímo převést celý dotaz do skoro stejné anglické věty. SEQUEL pak tedy vládlo poměrně dobře nad prvními databázemi. Nevýhodami bylo možná právě to, že jazyk je dotazovací. Od ostatních jazyků té doby nedovoluje žádnou viditelnou práci s proměnnými a v jazycích strukturovaných, kterými v té době bylo C, byl spíše takovým cizákem kvůli své skladbě. Rozdíl můžeme vidět na první pohled.

SQL se jako následovník stalo skutečným vládcem právě kvůli celé otevřenosti a příchodu možnosti vložení různých relací. IBM v té době poměrně ztratilo kormidlo, protože nebylo dlouhou dobu schopné odpoutat se od starého standardu a přejít na nový. SQL mělo všechny výjimečnosti. Konečně tu bylo dokonale přenosné řešení, které mělo fungovat stejně jak na ORACLE, tak na IBM. V zásadě pak každý mohl napsat skript pro vytvoření databáze, vložení tabulky a vložení záznamu a tento skript přenést, stejným způsobem aplikovat na jinou databázi a setkat se s naprosto identickým výsledkem. Celý nápad byl poměrně dost otevřený, protože pořád byl zachován systém anglického uspořádání dotazu.

Nad celým tímto standardem bdí společnosti ISO a ANSI, které stály u zrodu SQL. I proto je samotný jazyk otevřený (databáze však nemusí), protože je spravován autoritami, které jej upravují a schvalují jeho vývoj. Takovýto trend zaručuje jistou stabilitu naší práce. Teoreticky se nám tedy nestane, že napíšeme skript pro SQL a druhý den, po instalaci nové verze SQL, se nám nepodaří zprovoznit starý skript. SQL se moc nemění, je stálé. Setkáme se spíše s PL/SQL, které je ale úplně něco jiného a my se nemusíme bát.

V zásadě tedy máme databázi, kterou si nainstalujeme a pracujeme s ní pomocí jazyka, který se nazývá SQL. Samotné SQL bez databáze nezmůže nic a databáze bez SQL není schopná jakékoliv odpovědi, tedy práce, proto tato kniha nepojednává o SQL samostatně, ale vysvětluje databázi a SQL ruku v ruce.

Příklady

- Jaký je rozdíl mezi databází a SQL?
- Proč se používají databáze?
- Jaký je rozdíl mezi SEQUEL a SQL?
- Pokud byste vytvářeli ohromnou databázi, která musí pojmout miliony záznamů, jakou firmu použijete?
- Pokud jste malá firma, která je otevřená novým názorům a chce vyniknout na novém trhu, jakou databázi zvolíte?

Základní pojmy

Než začneme pracovat s jednotlivými příkazy, musíme ujít ještě velkou cestu, na které si řekneme, co je potřeba vědět o jazyce SQL. V databázích rozlišujeme některé části, o kterých se také musíme zmínit. I samotný jazyk má určitou terminologii, kterou si musíme projít předtím, než začneme pracovat na skutečném SQL skriptu.

Příkladem si řekneme, co je skript. Prakticky je to upravovatelný ASCII soubor složený z jednoduchého textu. Skript se dá zpracovávat naprosto nezávazně, proto můžeme hodněkrát za sebou přepsat skript našeho SQL a nic se neděje. Je však potřeba něco, co náš skript pustí. Když tedy budeme potřebovat zapnout skript, musí jej zpracovat program, kterému skript patří. Pro SQL je to databázový program. Příkladem jiných skriptů je třeba .bat pro dávkové soubory Windows.

Naše SQL se nazývá deklarativní jazyk, prakticky to znamená, že v SQL definujeme, co se má přesně udělat, ale už neříkáme jakým způsobem. Většina jazyků je imperativní, které řeší pravý opak. Měli bychom mít tedy na paměti, že s SQL nezmůžeme tolik jako s jiným jazykem, ale pro náš účel to bohatě stačí.

Relační databáze

Můžeme takto rozlišovat pouhé databáze a databáze schopné relací. Když si představíme jednoduchý rozdíl, tak relační databáze je taková, která dokáže definovat vztahy mezi jednotlivými tabulkami. Můžeme si to představit tak, že v normální databázi máme jednu kartu, ve které je vše, ale pokud si takovou lékařskou databázi představíme s relacemi, najdeme na kartě s pacienty nápis: „Bydliště pacienta: umístěno v kartách s bydlišti pod identifikačním číslem této karty.“ V naší databázi by to vypadalo jako jedno číslo pod cizím klíčem, o kterém si ještě povíme.

Relační databáze poskytují velice zajímavý komfort, který si normální databáze dovolit nemůže. Pokud dobře vymyslíme vztahy, můžeme velice navýšit rychlost vyhledávání, či snad usnadnit vybírání dat. Relace jsou velice užitečné, protože nám zmenší velikost jednotlivých tabulek. Databáze pak nepracuje při vybírání s velkým objemem dat, ale jen s daty, které skutečně pro práci potřebuje.

Dokonce v některých databázích můžeme definovat jednotlivé úložiště tabulek. Může se nám tak stát, že vytížené tabulky, které potřebujeme neustále a rychle, umístíme do dynamické paměti a tabulky, které nepotřebujeme stále, položíme na pevný disk. Podle vztahů pak nadefinujeme vazby a data se pak urychlí při vyhledávání, jednoduše je máme více po ruce, když je potřebujeme a ty, které se moc nevyužívají, leží na disku. Takový systém využívá MySQL. ORACLE pracuje na jiném principu, ale je efektivnější.

Moderní databáze jsou vždy relační. Někdy se však nemůžeme dočkat veliké efektivity. Někdy nám bohužel musí stačit, že relace prostě fungují, protože databáze není tak stabilní a schopná. Nemusí to být ani tím, že by byla špatná, ale že nikdy nebyla konstruovaná pro takovou funkci. Někdy nám efektivitu pokazí naše konstrukce databáze, protože prostě neumíme dobře pracovat se vztahy v databázi.

Dělení informací a dat

Už jsme o tom mluvili, ale je nutné to jistě rozlišovat. Naše databáze je vlastně soubor objektů a jedním z objektů je tabulka, něco jako vzor, podle kterého se nanáší do databáze data. Někdy můžeme slyšet, že databáze je složena z dvourozměrných souborů zvaných tabulky. Zatím přemýšlejme jen nad tím, co se do těchto tabulek nanáší. Vkládáme do nich data, která sama o sobě nic nikomu neřeknou. Jestliže nám třeba lékař řekne: „Tři,“ nevíme jistě, co tím myslí. Sestra ví, protože zná doktora, ale mi jeho zprávu bereme jako neurčité sdělení, která nám nic samo o sobě neřekne. Je to záležitost dat, které tvoří informaci.

Pokud nám tedy řekne doktor, že potřebujeme tři léky každé odpoledne, už víme, co musíme udělat a rozhodně nevyhodnotíme data, která nám poskytl, špatně. Pak už nemůžeme říct, že máme dát sestře tři koruny za jeho služby, ale rozumíme jeho informaci. To je jednoduchý rozdíl těchto dvou termínů.

Právě proto musíme přemýšlet, co vlastně z databáze čerpáme. Řekli jsme si, že tabulka, v které jsou data, má dva rozměry, jeden rozměr je jméno pole, do kterého data ukládáme, a druhým rozměrem je klíč záznamu. Můžeme tak říct databázi pomocí SQL, že chceme dostat z pole „JMÉNO“ záznam s identifikačním číslem 2342, jednoduše jsme tedy určili, co chceme. Pokud takovýto záznam existuje, databáze nám jej zobrazí a my se nemusíme bát chyby. Náš výběr je ale stále kus neurčitých dat, informaci z něj dělá fakt, že náleží do pole „JMÉNO“ a my jej udělíme třeba do nějaké věty, ve které bude znít tento kus dat takto: „Jméno pacienta je Karel.“ Teď už rozhodně víme, co chceme.

Datové typy

Když navrhujeme tabulky v databázích, musíme si rozmyslet, jaké data budeme do databáze a jednotlivých tabulek nanášet. Databázi velice pomůže, když nadefinujeme, s čím bude pracovat. Definujeme velikost pole a jaká data budou v poli. Prakticky si to můžeme představit tak, že budeme sestře říkat, aby zapsala datum do karty pacienta. Databáze v počítači má podobné omezení. Zadáme tedy pole, které bude obsahovat datum, tím omezíme případně naší chybovost, kdybychom se pokoušeli zadávat jiný datový typ. Také můžeme velice urychlit práci databáze, protože určíme, s čím přesně bude databáze pracovat. Můžeme si to jednoduše představit tak, že budeme sdělovat: „Teď budeš pracovat s celými čísly.“ Databázi tak poskytneme údaj, že nemusí užívat algoritmy pro čísla s desetinou čárkou, a tak bude možno urychlit její práci.

Datové typy jsou více méně počítačově univerzální, proto se s těmito jednotlivými typy setkáváme nejen v databázích a SQL, ale i v jiných programovacích jazycích. Nesmíme si proto představovat, že by se kvůli databázím měnil počítačový svět.

Pokud budeme mluvit o typu datum, musíme si uvědomit, s jakou databází pracujeme. Microsoft je prostě jiný než historicky zažitý UNIX a jeho odvozeniny. V LINUXu a UNIXu jsou data nuly 1970 prvního ledna, ale na platformách Microsoft se dovídáme, že nula v datu je u roku 1990 prvního ledna. Nebudeme se zabývat tím, proč to tak je, ale je jasné, že Microsoft vždy přijde s něčím, čím naboří zajeté principy a svojí vůlí je stejně prosadí, i proto bychom se měli vyvarovat převodu mezi těmito platformami, protože Microsoft vždy může přijít s něčím, co nám doopravdy nadělá jen potíže a moc nepomůže. Berme to spíše jako rozdílnost.

Datové typy zde uvedené jsou spíše základní, protože na jiných databázích mohou být další a další možnosti pro jiné typy. Můžeme ale čekat, že standardně můžeme nalézt následující:

BIT [(velikost)]

Je datový typ pro ukládání pouze binárního obsahu. Znamená to, že v něm může být zaneseno pouhé 1 nebo 0, to z něj dělá nejmenší datový typ, který můžeme použít. V některých databázích je zjednodušen na datový typ ano/ne, protože splňuje pro toto využití základní podmínku. Hodí se tedy na zápis jednoduchých pravdivostních hodnot. Velikost znázorňuje počet možných jedniček a nul, které můžeme použít. Maximální velikost je 64 bitů, takže můžeme zaznamenat řadu 1010101 až po počet 64 míst velikosti.

TINYINT [(délka)]

Není přímo standardem, ale je výhodný, protože obsahuje zmenšený typ INT. Zapisují se v něm celá čísla od -128 do 127 (i nula je hodnota, proto 127), pokud budeme pracovat bez znamének, můžeme mít rozsah od 0 do 255. Když si uvědomíme, jak veliké číslo to je, zabírá náš záznam 1 byte.

SMALLINT [(délka)]

Je větší než TINYINT, ale ne tak veliký jako INT. Jeho velikost může být od -32768 do 32767 a bez znaménka můžeme mít rozsah od 0 do 65535. Při uložení zabírá 2 byty paměti.

MEDIUMINT [(délka)]

Je stále datovým typem pro zápis celých čísel s hodnotami od -8388608 do 8388607. Čísla bez znaménka mohou být od 0 do 16777215. V paměti zabírá 3 byty (všimněte si posloupnosti jednotlivých velikostí typu INT. Zajímavé, že? To zdravotní sestra nedokáže.)

INT [(délka)]

Je datovým typem celých čísel s hodnotami od -2147483648 do 2147483647 a s čísly bez znaménka je od 0 do 4294967295.

BIGINT [(délka)]

Je zvětšeným typem INT, který poskytuje zápis od -9223372036854775808 do 9223372036854775807 nebo bez znaménka od 0 do 18446744073709551615, což z něj dělá nástavbu už tak velkého typu. Při uložení zabírá 8 bytů.

Konečně celý typ čísel. Teď už víme, jak rozdělit číselný datový typ k největší efektivitě v naší databázi.

FLOAT [(délka,počet_číslic za_desetinou_čárkou)]

Je datový typ pro zápis čísel s plovoucí desetinou čárkou s hodnotami $-3{,}402823466*10^{38}$,0 , $-1{,}175494351*10^{-38}$ do $3{,}402823466*10^{38}$. Při uložení zabírá 4 byty.

DOUBLE [(délka,počet číslic za desetinou čárkou)]

Je datový typ pro zápis čísel s desetinou čárkou, ale ohromnou velikostí od +/- 1.79769313486231570 E+308 (15 platných číslic), což z něj dělá jeden z největších typů vůbec.

Právě jsme probrali typy pro čísla, ale mějte vždy na paměti, že délka je zadána v počtu cifer, který může záznam obsahovat.

Dále používáme datové typy pro datum a čas:

DATE – je typ pro uchování časového formátu YYYY-MM-DD (YYYY je rok s čtyřmi číslicemi, MM je měsíc pomocí dvou číslic a DD je den v měsíci zapsaný pomocí dvou číslic), tedy v normálních databázích. Někdy se dostaneme jen k typu Datum a čas (MS ACCESS). Dalším typem může být YY-MM-DD. Někdy se rok převádí na logičtější část, protože se jednoduše čas nevrací, když tedy chceme zadat rozumný rok, který je před rokem 2000, měli bychom zadat celý formát 1991, protože 00 se převede na 2000 a 69 zase na 2069, od roku 70 do 99 se zase převede na 1970 až 1999. Měli bychom si dávat pozor na jednotlivé platformy databází.

Narazili jsme na oddělovače - , kterými oddělíme jednotlivé časové období. Můžeme ale použít i tečky, čárky, lomítka, hvězdičky a dokonce i plus. Někdy zadáváme i bez oddělovačů YYYYMMDD, délka řetězců je 6 nebo 8, kdy 6 je YYMMDD a 8 je YYYYMMDD. Kratší délka je většinou vyplněna samými nulami zleva na délku.

Při uložení zabírá 3 byty.

DATETIME – je mezikrokem mezi datem a časem, můžeme zaznamenávat YYYY-MM-DD HH:MM:SS. Když zadáváme bez oddělovačů, píšeme YYYYMMDDHHMMSS nebo YYMMDDHHMMSS. Délka je povolena 6, 8, 12 nebo 14, kde každé číslo je jiný záznam. Pro 6 je YYMMDD, pro 8 je YYYYMMDD, pro 12 je YYMMDDHHMMSS a pro 14 je YYYYMMDDHHMMSS. Platí zákon přidávání nul.

Při uložení zabírá 8 bytů.

TIME – slouží pro uchování času ve formátu HH:MM:SS nebo HHH:MM:SS. Můžeme zapsat i pomocí zkratek HH, HH:MM, či SS a bez oddělovačů jako HHMMSS a HHHMMSS.

Při uložení zabírá 3 byty.

YEAR [délka] – slouží pro zaznamenání roku ve formátu čtyř číslic, můžeme zadávat také pomocí dvou číslic, kde se musíme držet zásad rozdílů mezi 00-69 a 70-99, jak jsme již zmínili.

Při uložení zabírá 1 byte.

Někdy se můžeme setkat s typem TIMESTAMP, který nám umožní zapsat změnu záznamu. Je to spíše vlastnost, kterou umožňuje MySQL. V tabulce může být pouze jednou.

Typy pro práci s řetězci jsou jednou z nejdůležitějších věcí, kterou budeme používat. Mohou to být i typy BLOB, ale ty můžeme plnit i typy kolem 4 GB dat, proto je nemusíme používat jen na text, ale i na jiná data, která pak musíme převést na jejich formát.

CHAR (délka) – je typ pro uchovávání krátkého textu o velikosti 0 až 255, kde délka je vlastně počet znaků, které může typ obsahovat. Při ukládání se typ doplní nulami, pokud není zaplněn a při zobrazení se nuly zase vyruší, to však ukazuje, že CHAR ať je jakkoliv veliký, je stále stejný.

VARCHAR (délka) – je větší než CHAR a jeho velikost je od 0 do 65535, platí zase stejné pravidlo, jak pro celý text, tedy že délka je počtem možných znaků.

BINARY – je zvláštním typem, protože umožňuje záznam textu bezpečně binárně.

VARBINARY – je větším typem BINARY, který je vybaven binárním bezpečnostním ukládáním.

TINYBLOB – je datovým typem pro ukládání řetězců o maximální délce 255 znaků. BLOB je zvláštním typem, protože ukládá binárně a ukládaný řetězec bude uložen ve své skutečné velikosti.

BLOB – může být až 65535 v počtu znaků. Stále je ukládáno binárně a ve skutečné velikosti.

MEDIUMBLOB – je zvětšením BLOBu na 16777215 znaků, stále platí binární ukládání.

LONGBLOB – je největším typem BLOBu o velikosti 4294967295 znaků. I při této velikosti bude záznam ukládán jako binární se svou pravou velikostí.

TINYTEXT – je datovým typem pro text, který má velikost maximálně 255 znaků. Zde se liší ve způsobu ukládání, které se provádí jako text a je stále závislé na velikosti, kterou vložíme.

TEXT – je datový typ pro ukládání 65535 znaků. Vše probíhá stejně jako u TINYTEXTU.

MEDIUMTEXT – je datový typ pro text o maximální velikosti 16777215 znaků. Pořád zůstáváme na ukládání textu ve skutečné velikosti.

LONGTEXT – je největším textovým typem o maximální velikosti 4294967295 znaků a o stálém principu ukládání textu o pravé velikosti.

Tučně jsou naznačeny typy, které bychom měli mít vždy aspoň trochu v paměti, ale tím není řečeno, že by se měli ostatní vypustit. Může se nám stát, že některé zde zmíněné typy ani nenajdeme v naší databázi, což je více než pravděpodobné, proto je nutné mít aspoň obecnou povědomost.

Pokud ukládáme řetězce, musíme přemýšlet u některých databází o tom, v jaké znakové sadě se ukládají. Na UNIX platformách je to tradiční UTF, ale Microsoft často pracuje s vlastním kódování CP – 1250. Znaková sada pak přidává záznamu o něco více velikosti, než má reálně.

Příklady

- Jaký datový typ bychom měli použít pro číslo, se kterým budeme pracovat jako s číslem?
- Co dělá z relační databáze relační databázi?
- Jaký je rozdíl mezi sdělení „neděle“ a sdělením „Dnes je neděle.“?
- Co znamená zápis HH?
- Víte, kde zhruba najít datové typy v této knize?

Poslední otázku berte jako autorův návod, jak s datovými typy pracovat. Je nesmysl si je pamatovat celé, ale jen obrazně vědět, jak s nimi pracovat k dobré efektivitě. Pamatujte, že čím lépe vyberete datový typ, tím lépe databáze využije jeho velikost a pracuje s ním rychleji.

Objekty databáze

Objektem v databázi můžeme rozumět spousty jednotlivých funkcí. Není zcela jednoznačné, zdali objektem můžeme nazvat vše, co zde bude uvedeno, ale je velice důležité se u toho pozastavit, protože si musíme uvědomit, z čeho se doopravdy databáze skládá, v čem je její síla a kouzlo. Vše se pokusíme aplikovat na objektový model reálného světa a tím si určit jednoduché vazby, které nás přivedou na správnou cestu.

Je ale jasné, že nemůžeme u databází říct přesné znění definice objektové programování, protože my budeme pracovat s poněkud jinými formami programování.

Databáze

V tomto příkladě nám databáze slouží jako něco, co v sobě obsahuje spoustu vlastních databázových objektů. Můžeme ji vytvořit v nějakém jistém programu, který je určen pro databáze a ve kterém vytváříme v něm objekty. Naším objektem je zde databázová entita, kterou je právě tabulka, či jiný objekt.

Měli bychom vědět, že databází můžeme mít více než jednu, a měli bychom se naučit je rozdělovat podle toho, jaká data obsahují. Je nevhodné rozdělit data, která spolu souvisí, do více databází a je krajně nevhodné do jedné databáze vkládat entity, které spolu nemají žádnou souvislost. Nakonec se nám může stát, že máme ohromnou databázi plnou objektů, o kterých vůbec netušíme, k čemu náleží a kam patří.

Pro všechny tyto důvody by měl být název databáze naprosto směrodatný a pro nás určující, co přesně v ní můžeme nalézt.

Databázi si tedy můžeme představit fyzicky jako skříň, která obsahuje spoustu karet. V počítačové databázi jsou to právě objekty, místo karet a pracujeme s nimi pomocí SQL.

Tabulka

Jak jsme si řekli, je to pro nás takový rámec, který je shodný s normální tabulkou. My s ním však pracujeme jako se vzorem pro ukládání nových dat. Tabulky tvoří databáze a jsou jednou ze stěžejních částí. Bez tabulek bychom nebyli schopni jednotlivě pracovat se záznamy, protože naše data by neměla podle čeho se třídit.

Tabulka je právě v našem modelu tou pravou a jedinečnou entitou. Až začneme zakreslovat entity, měli bychom je psát velkými písmeny. Abychom si představili více tuto entitu, udáme si příklad.

ZVÍŘE

STROJ

AUTOMOBIL

PRODUKT

STUDENT

Z toho poznáme, že entity jsou takovým povýšeným, globálním nadhledem. Nemůžeme z nich vyčíst, kdo je zvíře, či jaký automobil myslíme, ale víme, že entity budou mít souvislost se svým jménem, a tím jednoduše stanovíme, co v ní asi bude ležet.

Někdy se setkáváme s nepřímými entitami, které nám ani zdaleka neřeknou, co v nich je a přesto data obsahují. Příkladem, entita PRODUKT může obsahovat ponožky, které jsou produktem textilní firmy, ale i tak může obsahovat auta, která jsou také produktem firmy, ale už ne textilní. Proto bychom se měli vyhýbat obecným entitám, protože i při prezentaci někomu jinému, než jsme my samotní, někdy nemusíme podat jasné vysvětlení toho, co jsme doopravdy chtěli říct.

Sloupec

Je obsažen v tabulce a je tedy součástí entity. Můžeme si ho představit jako něco, co nám má vytvořit rámec ukládaných dat. Do sloupců v tabulce vkládáme právě ta data. Určují nám, jak budou data v tabulce složena a jakého určení budou. Z nejasné entity se snaží udělat entitu, která má vlastnosti.

Sloupec se jmenuje právě proto, že když jej necháme zobrazit, je právě sloupcem, který obsahuje jednotlivá data. Můžeme jej zobrazit jako rozšíření entity:

PRODUKT jméno

	cena
	hmotnost
STROJ	jméno
	hmotnost
	velikost
	rychlost
ZVÍŘE	název
	počet končetin
	barva
STUDENT	jméno
	věk
	třída
	bydliště

Je to v pravém ohledu vlastnost entity. V našem vyjádření si ji můžeme představit jako něco, co nám stanoví, jak entita vypadá, ale neřekne, jaká je instance, tedy záznam. Nedozvíme se, jestli se student jmenuje Omáčka nebo Novák, ale víme, že se z tabulky studentů dozvíme jejich jména, věk, třídu a bydliště, tedy vlastnosti.

Řádek – záznam

Záznam je směrodatný, konečný zápis dat. Samotný nám nic neřekne, ale protože leží v databázi, která má přesný název, leží v tabulce, která má také přesný název a náš záznam je uložen podle jednotlivých vlastností, víme, jak vypadá informace, kterou získáme pomocí toho všeho.

Ukážeme si, jak vypadá tato instance.

Entita:	Vlastnost:	Instance:
PRODUKT	jméno	ponožka
	cena	40,--
	hmotnost	50 g
STROJ	jméno	rypadlo
	hmotnost	5 t
	velikost	50 m^3
	rychlost	20 km/h
ZVÍŘE	název	klokan
	počet končetin	4
	barva	hnědá

Klíč

Už jsme si ukázali, co můžeme nalézt v jednotlivých tabulkách, co jsou záznamy a podle čeho se ukládají, ale co když se stane, že uložíme nějaký údaj, který je naprosto totožný se záznamem předešlým? Jak pak odlišíme, co je náš záznam a který je který. Snažili jsme se vše dělat tak jedinečné, a teď se nám nepodaří ani uložit jednu věc vícekrát?

Ukážeme si, že můžeme data rozlišovat podle jejich jedinečnosti a pokud jedinečná nejsou, přidáme vlastnost, která je jedinečnými udělá. Nakonec uvidíme, že pomocí klíče, který dělá věci jedinečné, můžeme vytvořit druhý rozměr v poli, kde prvním rozměrem byla jiná vlastnost dané tabulky.

Někdy se stane, že nemusíme ani klíč vytvářet, že vždy vkládáme data, která jsou něčím jedinečná, ale mějme na paměti, že nám někdy pomůže si klíč vytvořit, než odkazovat na nějaký, který vyplývá z přirozeného příkladu.

Přirozený klíč by měl vypadat asi takto:

EVIDENCE_NAVSTEV

Jméno	Příjmení	Rodné číslo	Bydliště	Datum
Karel	Václavík	*900161/899*	Hodonín	21. 2. 2010
Tomáš	Neměl	*600121/655*	Pardubice	30. 1. 2008

Zde vidíme jasný přirozený klíč, protože rodné číslo je tak jedinečné, že není třeba žádného jiného, proto můžeme jasně vyhledat v tabulce záznam podle rodného čísla, příkladem 600121/655, víme, že jméno podle rodného čísla je Tomáš a příjmení Neměl. Není třeba žádný klíč vytvářet, ale co když bude tabulka bez rodného čísla?

EVIDENCE_NAVSTEV

Jméno	Příjmení	Bydliště	Datum
Karel	*Václavík*	Hodonín	21. 2. 2010
Tomáš	*Neměl*	Pardubice	30. 1. 2008

I zde můžeme jasně rozhodnout, co je jedinečný údaj v tabulce. Říkáme tomu složený klíč, protože jedinečným se stane údaj právě tehdy, když složíme dva údaje v záznamu, které vytvoří složený klíč. Zde můžeme složit jméno a příjmení, ale jen za předpokladu, že se nestane naprostou náhodou, že se do naší databáze zaregistruje jedinec, který má naprosto identické jméno s příjmením, jako někdo z předešlých návštěvníků. To je pravé kouzlo složeného klíče.

Co když ale přijde do ordinace naprostý jmenovec, jak už jsme zmínili a my, kteří se snažíme udržet osobní data na uzdě, nevezmeme za vděk s rodným číslem? Nezbude nám nic jiného, než si vytvořit vlastní klíč, úplně nezávislý na minulých datech. Někdy je daleko lepší vytvořit si vlastní klíč, než odkazovat na složený, či přirozený, protože náš umělý klíč může představovat daleko menší využití datového prostoru na disku, než třeba rodné číslo, které je moc dlouhé pro databázi deseti pacientů.

EVIDENCE_NAVSTEV

ID	Jméno	Příjmení	Bydliště	Datum
1	Karel	Václavík	Hodonín	21. 2. 2010
2	Tomáš	Neměl	Pardubice	30. 1. 2008

Zkratka ID je zde napsána zcela úmyslně, znamená zkratku z identification a právě ID je velice často v databázích pro umělé klíče. Teď jsme přidali vlastně jen další sloupec, vlastnost tabulce, která nám stanový jednoduchý druhý rozměr a my, spolu s jiným údajem, lehce určíme data, která jsou požadována.

Umělý klíč má ještě jednu výhodu. Často se v databázích setkáváme s funkcí AUTO_INCREMENT (třeba MySQL), která nám přičte k číslu minulého záznamu jedna a tím vytvoří číslo o jedna větší, než číslo minulého záznamu. Znamená to, že se nemůže stát, že vložíme stejné číslo pod ID do tabulky, protože automatické přičtení tomu zabrání. Záznamy se vždy řadí za minulé záznamy. V Microsoft Accessu se tato funkce jmenuje automatické číslo. Databáze ORACLE pracují na rozdíl od jiných databází se sekvencemi, které vytvoří řadu čísel dle zadaných parametrů, to nám může pomoci vytvořit také takové umělé klíče.

Dalším pojmem v databázi může být takzvaný cizí klíč, o kterém si více povíme v kapitole o JOIN, kde budeme spojovat relací více jak jednu tabulku. Cizí klíč připadá záznamu v jiné tabulce, stanovuje jeho jedinečnost a v tabulce, kde je cizím, stanovuje vztah k jiné tabulce, z které pomocí takového klíče, ukazuje na data, kde není cizí. Z této věty si jen tak představu neuděláme, ale na příkladu snadno pochopíme první ukázku vztahu v této knize.

EVIDENCE_NAVSTEV

ID	ID_Bydliště	Jméno	Datum
1	2	Karel	21. 2. 2010
2	2	Tomáš	30. 1. 2008

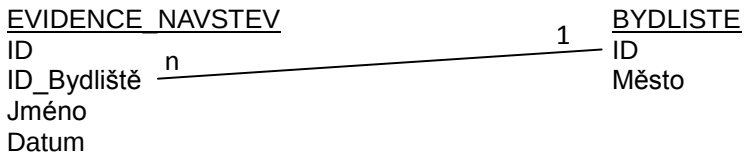
BYDLISTE

ID	Město
1	Pardubice
2	Praha

Teď jsme trochu přeskočili, ale jasně uvidíme, jak nám cizí klíč snadno ulehčí práci. Nemusíme teď už do tabulky EVIDENCE_NAVSTEV psát bydliště, ale jen vybereme číslo ID v tabulce BYDLISTE a v té už snadno najdeme, kde pacient bydlí. Ve fyzické databázi bychom tento úkon těžko zvládali, ale mnohdy jej používáme bez toho, abychom si jej uvědomili. Když si položíme otázku: „Na jeho občanském průkazu je číslo státu 3323, co je to za stát?“ a následně ji vyhledáme v databázi států (pro nás v jiné tabulce, protože

vztahy mezi databázemi nejsou normálně možné), víme hned, že je to třeba Dominikánská republika.

Pokud si tento vztah nakreslíme bez nějakých hlubších úprav, vypadá asi takto:



Tento první vztah nastiňuje k čemu je cizí klíč, tedy více návštěvníků bydlí v jednom bydlišti. Neberme to tak, že všichni bydlí v jednom bytě, ale třeba ve stejném městě. Neznamená to, že máme záznam jen o jednom bydlišti, ale že můžeme použít jeden záznam bydliště pro více uživatelů.

V dalších kapitolách si o těchto vztazích povíme více, ale pro vysvětlení cizího klíče musíme vědět, kde ho použít.

Dotaz

Dotaz je vlastně to, o čem bude tato kniha vypovídat nejvíce. Musíme si ale jasně stanovit, co takový dotaz dělá. Můžeme se dotazovat na data v databázi, v jednotlivých tabulkách. Dotaz, jako takový, neobsahuje žádná data, protože on data z tabulek vybírá. Znamená to, že pokud si uložíme dotaz, který se nám líbil, a změníme obsah tabulky, tedy vymažeme, či upravíme nějaké libovolné záznamy, nemusí výsledek dotazu vypadat stejně jako v minulosti.

Představme si to asi takto. Jdeme do restaurace a v menu se dozvíme, že husa stojí jen 100 korun. Náš dotaz je tedy: „Vyber jídlo husa z tabulky menu, kde cena odpovídá 100.“ Převedeně to takto můžeme říci, ale v SQL by to vypadalo zhruba takto: „SELECT husa FROM menu WHERE cena = '100' ;“ Může se nám stát, že v menu bude více hus, ale my chceme husu, co stojí pouhých 100 korun.

Náš dotaz si schováme a přijdeme do restaurace znovu po týdnu. Do té doby zjistí majitel restaurace, že husa je každý den vyprodaná a že vlastně tak levná není. Majitel zdraží husu. My přijdeme do restaurace a ničím nerušeni provedeme náš dotaz. Co se však stane? Nic nenajdeme, náš dotaz nám vrátí prázdnou množinu. Může se nám stát, že majitel restaurace zlevní více hus a my jich najdeme pomocí našeho dotazu více najednou, ale to jen potvrzuje, že dotaz je jen dotazem.

Příklady

- Co je umělý klíč a kde je nejvýhodnější ho použít?
- Jak poznáme, jaké vlastnosti má tabulka?
- Co je záznam v tabulce?

Obsahuje dotaz nějaká data?

Diagramy

Pro práci s databázemi jsme nuceni vytvořit si nějaký náhled, nějaký určitý nákres situace, abychom věděli, s čím skutečně pracujeme. Je těžké posuzovat vlastnosti databáze a vytvářet na ni jednotlivé dotazy, dokud netušíme, jaké vazby jsou potřeba a jaké potřeby budeme využívat.

Už jsme si říkali o cizím klíči, kterým se označují vazby jedna ku mnoha. Teď si však vysvětlíme, jak tuto vazbu zakreslit a tím poznamenat pravidlo, které tím vytvoříme. Samo sebou, že existují databáze, kde není třeba kreslit si vazby a znázorňovat náležitosti, ale ty jsou prakticky mnohem menší a netvoří je tolik tabulek, abychom o nich mohli mluvit. Proto vždy, když budeme navrhovat větší databázi, budeme si zakreslovat vazby mezi tabulkami a podle nich poznáme, jak se má databáze skládat a co od ní čekáme.

Někdy nás může zmást, jak jsou vazby nakresleny, ale vždy si musíme uvědomit jednoduché vazby, které jednoduše pochopíme. Nebudeme se zabývat vazbami typu Matrix, či jinými, které mohou být příliš specifické.

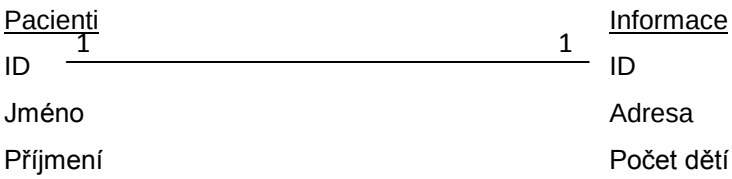
Vztahy

Vztahy jsou to, co dělá z relačních databází relační. Jak jsme již řekli, budeme potřebovat jejich pomoc při zakreslování náležitostí dat v databázi. Dělíme je na tři základní vazby a bezpočet ostatních, které ale nemusí být všude stejné. Příkladem MySQL Workbench je naplněno tolika možnostmi, že je zde ani nestačíme pokrýt, ale když půjdeme od začátku, rozšířené vazby se nám pak budou zdát jednoduché, když si o nich něco přečteme z jiné literatury. ORACLE přišlo s vlastní teorií o relacích, ta je tak rozsáhlá, že pokrývá všechny možné vazby, ale pro nás jsou tak rozsáhlé, že bychom je ani nemuseli využít.

Diagram, který budeme vytvářet, se jmenuje relační diagram, pro nás Čechy jednoduchý název, ale v anglickém jazyce se můžeme setkat s ERD – enhanced relationship diagram – pro ORACLE a pro MySQL EER – enhanced entity relations diagram. Je jasné, že obě zkratky se zabývají v zásadě tím samým, ale každé má nějaké drobné odlišnosti a přídavky vzhledem k dané databázi. My se však naučíme nejdříve znázornit diagramy co nejjednodušeji.

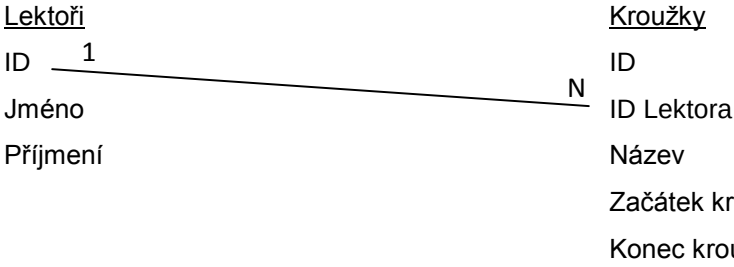
1:1 je základní vazba, která znamená jednoduše jedna ku jedné, tedy jeden záznam náleží jednomu záznamu v jiné tabulce. Praktické využití je většinou u tabulek, kde často využíváme jistou část dat a jiné data máme „schovaná“ v jiné tabulce a vybíráme je, když je jen někdy potřebujeme.

V podstatě tím urychlíme práci databáze, protože se nemusí zabývat vybíráním velkých objemů dat, které se ani celé nevyužijí a může se zaměřit na data, které jsou skutečně potřeba. Na příkladu si znázorníme, jak to zhruba vypadá.



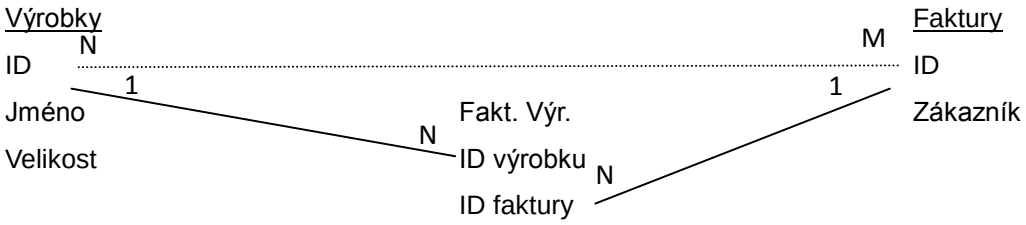
Zjednodušeně zde vidíme, že jen rozšiřujeme tabulku první o údaje z tabulky druhé. Někdy je toto jedno z elegantních řešení, jak doplnit běžící databázi bez toho, abychom nějakým způsobem měnili schéma dosavadních tabulek. Musíme uznat, že příkladem při vybírání uživatelů z databáze nám nezáleží na počtu dětí jednotlivých uživatelů, proto jsme tento údaj dali do další tabulky.

1:N je vazba, o které jsme už mluvili, a je prakticky nejpoužívanější ze všech vazeb. Mluvíme o ní jako jedna ku mnoha, tedy jeden záznam v tabulce připadá více záznamům v tabulce. To však znamená, že pokud má něco náležet více záznamům, musí mít záznam, který požaduje jiný záznam, někde zachyceno, k čemu náleží. Takhle to zní velice složitě, ale když si budete opakovat, že je to jedna ku mnoha, určitě brzy uvidíte, o co doopravdy jde. Navíc už víte, co je cizí klíč a teď jen dojde k jeho aplikaci v praxi.



Zde vidíme, že jeden lektor vede více kroužků. Při vyplňování nových záznamů ale vždy musíme myslet na to, jestli odkazujeme cizím klíčem na platného lektora, což si ukážeme u jazyka DML. Nejenže pro jedno pole v tabulce doplníme všechny údaje z jiné tabulky, ale údaje z jiné tabulky můžeme použít vícekrát, což dělá z vazby jedna ku mnoha nejpoužívanější vazbu vůbec.

M:N je zvláštní případ relace, kdy můžeme říci, že více záznamů jedné tabulky připadá více záznamů jiné tabulky, což vytváří jistou nesrozumitelnost. V praxi se to řeší přidáním další tabulky, ke které vedeme vazbu jedna ku mnoha z dvou tabulek, které chceme spojit vazbou mnoho ku mnoha.



Znázornit vazby mnoha ku mnoha je někdy velice složité a někdy velice nepotřebné. Nemusíme vždy propojit všechny tabulky, abychom získali určitá data. Už jsme si říkali o tom, že vazbu jedna ku jedné používáme pro přidaná data, ale to může platit i pro ostatní, nemusíme vždy pracovat se vším, co máme a čím méně potřebujeme, tím rychleji můžeme pracovat.

Nemusíme se omezovat na pouhé vztahy jedné tabulky k druhé, ale i k více tabulkám, propojených tak různě, jak jen to jde. Někdy se nám může ztratit přehled nad vztahy, ale měli bychom jej rychle nabýt při prohlédnutí právě druhu vazeb, které použijeme.

Normalizační zákony

Normalizační zákony jsou velice specifickou teorií okolo databáze, protože se poměrně dost vymykají ostatním programovacím jazykům. Je nutné si proto uvědomit, že normalizace nám umožňuje lepší práci s daty a určuje jasná pravidla, jak s daty zacházet. Můžeme s nimi uvažovat jednotlivé databáze a jejich relace a podle normalizace stanovit, jak databáze funguje. Můžeme je uplatnit při vytváření databáze, protože normalizační zákony určují normu, jak databázi vytvořit. Pokud překročíme nějaký ze zákonů, můžeme tím docílit špatné struktury a práce s daty se může stát nemožná.

Je nutné uvažovat aspoň první tři pravidla. Dohromady se užívá pět pravidel, ale vždy musíme pamatovat pravidla základního, že nesmíme uvažovat další pravidlo, dokud nesplníme minulé, proto je nutné uvažovat všechny a vžít si je, jako prostou pravdu pro tvorbu databáze. Nemusíme se bát, že pokud se něco nenaučíme, tak databázi nebudeme správně ovládat. Někdy je spíše potřeba si uvědomit, jestli daná databáze dává smysl a u ní uvažovat nad normalizačními zákony. Nakonec, když se nám dostane něčí databáze do rukou, můžeme ji vzít a pomocí relací stanovit běh dat, jejich relace, a jako rámec našich úvah použít pravidla normalizace.

1. normalizační forma – každý atribut obsahuje pouze atomické hodnoty. Řečeno rozumněji, můžeme uvažovat atomické, za dále nedělitelné. To znamená, že první normalizační zákon říká, že každý sloupec obsahuje data, která jsou zcela jednoznačná a dále nedělitelná.

Častou chybou je vyplnění dvou telefonních čísel do jednoho pole záznamu. Je sice dobré mít dva kontakty na jednu osobu, ale pokud je někde nějaká vazba, která pracuje čistě s datovým typem čísla a my při zápisu dvou čísel použijeme oddělovač, může se stát, že relace přestane fungovat. Nemusí to být jen relace, ale i SQL dotaz na záznam by mohl mýt jisté problémy. Představme se, že si budeme přát data z pole telefonních čísel a necháme si všechna čísla vypsat vedle sebe a jako oddělovač použijeme mezeru, jak potom odlišíme, že v jednom záznamu je více telefonních čísel? A proto používáme první normalizovanou formu/zákon.

2. normalizační forma – každý neklíčový atribut je závislý na primárním klíči. To nám samo o sobě moc neříká, ale když se zamyslíme nad celým zněním, je to prosté. Všechna data musí náležet svému správnému klíči, každý záznam má souvislost jen sám se sebou a s ničím jiným. Při vytváření tabulky tento fakt vypadá jako použití polí se stejným významem pro danou tabulku. Vysvětlíme si na příkladu.

Žáci	Žáci
ID	ID
Jméno	Jméno
Příjmení	Příjmení

Den	Rozvrh
První hodina	Den
Druhá hodina	První hodina
	Druhá hodina

Vidíme zde tři tabulky, tabulka vlevo je ukázkou špatné tabulky, protože nesplňuje druhý zákon. Když si to představíme jako sklad dat, vkládali bychom u každého žáka jeho jméno, příjmení a potom jeho první a druhou hodinu se dnem, kdy je má. Co bychom však museli udělat, kdybychom chtěli přidat další den pro stejného žáka? Nešlo by to, proto bychom nemohli pracovat s takovou databází.

Další dvě tabulky ukazují jednoduché využití druhé formy normalizace, protože rozdělí danou tabulku na dvě tabulky a pak můžeme vytvářet vazby pro tyto záznamy. Zlehčí se nám celá práce s celou databází a už nemám žádný problém s novými záznamy.

3. normalizační forma - všechny neklíčové atributy musí být vzájemně nezávislé. V zásadě nám tento zákon taky nic zprvu neřekne, ale prakticky jej používáme sami o sobě, když známe kouzlo relací. Prakticky to znamená, že bychom neměli v jedné tabulce zadávat atributy, které jsou závislé na jiných skutečnostech, než na kterých je závislá tabulka. Je krajně nevhodné přeplňovat tabulku zbytečnostmi, protože čím větší je, tím více z ní musíme vybírat a s tím více daty musíme pracovat. Když tabulku rozdělíme na více relací, stane se databáze rychlejší.

Zaměstnanci

ID

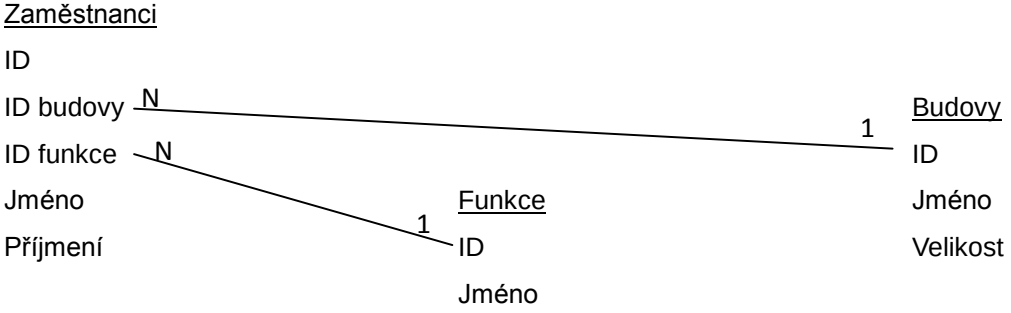
Jméno

Příjmení

Budova

Funkce

Zde vidíme, že nakládáme s příliš mnoha atributy, které se dají oddělit. Prakticky se stává, že tento zákon spousta programátorů porušuje, protože se někdy stane, že není potřeba dělit jednu tabulku na víc, když ji moc nevyužíváme. Při bližším průzkumu se nám však vyplatí třetí zákon a tabulku můžeme rozdělit takto:



BC (Boyce/Coddův) normalizační zákon - Atributy, které jsou součástí primárního klíče, musí být vzájemně nezávislé. Může se vám zdát, že tento zákon je spíše takové potvrzení třetího zákonu, ale on doopravdy je, protože jen prohlubuje členitost dat. Teď už nám nezáleží jen na nezávislosti jako takové, ale případ musíme prozkoumat poněkud hlouběji.

Tento příklad jsem vytvořil a pak zhlédl na mnoha jiných zdrojích informací o databázích, ale při vytváření příkladů se někdy doopravdy musíme zaměřit na to, aby takový příklad byl doopravdy ten pravý, proto možná vypadá stejně, jako všechny jiné.

Adresa

Město	**Ulice**	**PSČ**
Praha	Pertoldova	14300
Praha	Zápotoční	14300
Bohutín	Zápolní	12399

Zde vidíme, že se nám může stát, že u mnoha ulic bude stejné PSČ a přitom to tak být nemusí. Prakticky se pak řeší takováto závislost jako rozpad na dvě tabulky.

Adresa

PSČ	Ulice
14300	Pertoldova
14300	Zápotoční
12399	Zápolní

Město

PSČ	Město
14300	Praha
12399	Bohutín

4. normalizační forma - Tabulka popisuje pouze příčinnou souvislost mezi klíčem a atributy. Tento nic neříkající zákon je někdy až příliš nadsazován, vždy platí, že musí platit zákony předešlé a zde dvojnásob. Platí tedy, že pokud v databázi použijeme tabulku, kde se musí uplatnit složený klíč, stane se někdy, že takový klíč se stane složený spíše z povinnosti a ne proto, že by sloužil jako reference o ostatních atributech. Může se stát, že uplatním složený klíč na věci, které k sobě prostě nepasují.

Řešením takovéto závady je zase následný rozpad dle reference k danému klíči. Vždy vycházejme z přirozeného faktu, co vlastně tabulka zaznamenává a zdali je to skutečně potřeba uvádět v dané tabulce. U čtvrté formy to platí obzvláště.

5. normalizační forma - Relaci již není možno bezztrátově rozložit. Je milé, jak je tento zákon krátký, ale jak je těžký na pochopení, to už nikdo neřeší. Prakticky se nám tento zákon snaží říct, že pokud máme vytvořenou relaci mezi více atributy jedné tabulky, zní to divně, ale je to možné, když použijeme tabulku, kde tvoří klíč tři nebo více atributů, měli bychom tyto údaje rozdělit kvůli možnému vzniku cyklu mezi těmito závislostmi. Jednoduše se nám může stát, že zaměstnanec je závislý na svém působišti a na místě, kde pracuje, ale to znamená, že klíč je tvořen jeho jménem, místem práce a působišti, stejně tak se může stát, že je závislý na působišti, jménu a místě, kde pracuje. Jak jste si všimli, pořád můžeme opakovat závislost, která je prakticky stejná a stejně tak ji může opakovat databáze a nakonec se tato skutečnost stane neřešitelnou.

Řešením je rozklad těchto klíčů do vlastních tabulek, kde je nebude ohrožovat žádné zacyklení a budou zcela samostatně závislé na svých atributech, jenže to není vše. Musíme zachovat také původní tabulku, abychom na ní připojili nově vzniklé tabulky, které řeší právě ten nekonečný cyklus, tím se vyřeší pátá norma.

Jestli jste měli problém pochopit od čtvrtého zákonu po poslední, nebojte se, někdy se s takovými případy nemusíte ani setkat, ale s prvním až třetím se setkáte určitě a je velice důležité je mít stále na paměti a pracovat s nimi s velikou obezřetností a soustředěností.

Znázornění vazeb

Jak jste si všimli, pracujeme s vazbami, které mají nějaké symbolické zakreslení, ale to je vždy velice odlišné podle toho, s jakou databází pracujeme a jaký databázový model vytváříme. Jinak se kreslí relace pro ORACLE a jinak pro MySQL. Je velice moc možností, jak takové vazby zakreslit, ale co si budeme nalhávat, dokud neporozumíme tomu, jak relace funguje, nemá cenu kreslit čárkovanou relaci.

Při vytváření kresleného modelu se držte pár jasných pravidel, která jsou podstatná:

- Piště zleva doprava
- Snažte se pokrýt zřetelně celý papír
- Tabulky, které obsahují více vazeb na více tabulek, umístěte doprostřed
- Znatelně vyznačte, která relace je která
- Stále si opakujte, jestli je to pochopitelné i pro někoho, kdo takový papír vezme poprvé do ruky

Nakonec je nutné si nastudovat, jaký diagram použít pro jakou databázi, protože spousta databází také podporuje své vlastní nástroje pro vytváření diagramů a v nich se dá nalézt více rad pro vykreslování, než v obecné příručce o SQL.

Příklady

- Jaký je první normalizační zákon a jak byste jej vysvětlili svými slovy naprostému neznalci?
- Kde používáme vazbu jedna ku jedné a proč ji používáme?

Jaké tabulky budou určitě umístěny na diagramu uprostřed?

Základy jazyka DDL

DDL znamená data definition language – už z toho můžeme odvodit důvod tohoto členění. Jazyk SQL je takto členěn pro svou různost v dané rovině. Jedna část se stará o vytvoření jednotlivých struktur, zatímco jiná o vybírání dat ze struktur. Jazyk DDL se tedy týká vytváření struktur, objektů a pravidel databáze. Je něco jiného ale vytvářet oprávnění, pravidla se v tomto jazyku týkají vytvoření takových vlastních objektů, které pak budou držet normu, s kterou budeme pracovat. Hrubě řečeno budeme vytvářet tabulky a pohledy. Tímto pro nás bude DDL jedním z nejdůležitějších jazyků, které budeme potřebovat, protože právě on nám připraví novou půdu pro naše budoucí dotazy. Proto s ním začínám, protože je asi ten nejdůležitější, na kterém všechno stojí.

Tabulky

Už víme, co jsou tabulky, kdy a kde je kreslíme, k čemu slouží, ale jak je do databáze vložit, se dozvíme právě teď. Než si ale řekneme, jak na samotný příkaz, musíme si říct, jaké znaky můžeme použít a jakou konvenci používat pro názvy tabulek:

- Název tabulky musí začínat písmenem
- Musí být od jednoho do třiceti znaků (třicet berte jako obecný limit, není vyloučeno, že u rozdílných druhů databází může být číslo nižší, nebo vyšší)
- Může obsahovat pouze znaky od A do Z, znaky od a do z, znaky od 0 do 9, dokonce může obsahovat znaky $, _ (podtržítko) a #
- Nesmí mít duplicitní jméno s jiným objektem databáze
- Nesmí být rezervovaným jménem pro danou databázi (pro ORACLE to může být příkladem tabulka DUAL, která je v systému nativně pro vytváření výběrů z databáze jako příklad, či je pomocná pro výpočty)

Hned jsme poznali, že háčky a čárky mohou být problémem. Uživatelsky jednoduché databáze jako je Microsoft Access sice tvrdí, že háčky a čárky problémem nejsou, protože s nimi umí pracovat, ale vysvětlete pobočce v Číně, jak mají napsat „ř“. Takže, i kdyby to databáze uměla, víme, že raději budeme pojmenovávat podle pravidel mezinárodních konvencí, než abychom vysvětlovali, co je „ř“. Háčkům a čárkám se jednoduše vyhneme.

Je nutné ještě zmínit, že při pojmenování tabulek si spousta administrátorů potrpí na velkých písmenech a stejně tak u polí tabulek, ale to už není součástí pravidel. Je pravda, že nás to může někdy mást, proto bychom si už od začátku práce s databází měli uvědomit, jak budeme s názvy pracovat, abychom se v nich právě my co nejlépe vyznali.

Teď už víme, jak tabulku pojmenovat, ještě si řekneme, jakou konvenci použít pro celý dotaz. Pokud budeme pracovat s dotazem, či nějakým příkazem, budeme ho psát velkými písmeny. Už víme, že je jedno, jestli je psán velkým, či malým, ale pro uchování přehlednosti bychom si měli ponechat tento zvyk i u ostatních příkazů, vše si hned ukážeme.

Tabulku vytváříme dvěma slovy: CREATE TABLE – ve významu znamená tento příkaz: vytvoř tabulku. Tabulku hned nějak nazveme, tím vznikne dotaz: CREATE TABLE prvni_tabulka; - pokud jste si všimli středníku, je to dobře, protože tím oddělujeme jednotlivé příkazy a právě středník je někdy jediný oddělovač, který můžeme použít.

Se samotnou tabulkou toho ale moc nesvedeme, protože od ní očekáváme, že ji naplníme nějakými vlastnostmi, něčím, do čeho budeme ukládat záznamy, jak jsme si již řekli. Proto, už při vytváření tabulky, můžeme určit, jak budou vypadat i pole uvnitř. Teď si ukážeme, jak takové vytvoření může vypadat:

```
CREATE TABLE druha_tabulka_s_poli
(
název_pole1 datový_typ [DEFAULT hodnota],
název_pole2 datový_typ [DEFAULT hodnota],
…,
název_polen datový_typ [DEFAULT hodnota]
);
```

Vysvětleme si teď, co jsme vytvořili. Dali jsme příkaz „vytvoř tabulku“, která se jmenuje „druha_tabulka_s_poli“ a naše tabulka obsahuje atributy název_pole1 až název_polen, které mají datový typ „datový_typ“, kdyby nějaký takový existoval. Za datovým typem obvykle pokračuje normální závorka, která obsahuje číslo vyjadřující délku datového typu, ale to si ukážeme až na dalším příkladě. DEFAULT je hodnota, která bude přirozeně vyplněna, pokud nebude vložen žádný záznam pro toto pole.

Teď si ukážeme jednoduchou tabulku, kterou můžeme takto vytvořit:

```
CREATE TABLE nase_prava_tabulka_podle_pravidel
(
    ID NUMBER (10),
    jmeno VARCHAR 100),
    narodnost VARCHAR(50)
);
```

Všimněte si, že jsme za posledním polem nenapsali čárku, kdybychom ji tam napsali, při zpracování by nastala chyba, protože by bylo očekáváno další pole. Stejně tak si povšimněme středníku za celým příkazem. Také si musíme uvědomit, že pokud překročíme velikost datového typu pro jednotlivé pole, může se daný záznam oříznout pro danou velikost, či napsat chybu, záleží na typu a nastavení databáze, proto buďme vždy trochu schopni nadsazovat. Těžko najdeme někoho, kdo bude mít jméno delší, jak padesát znaků, ale přesto s tím můžeme počítat a nechat jméno velké až sto znaků. Budeme trochu nadsazovat velikost, co se týče uložených dat, ale i tak předejdeme chybám.

Dalším příkazem, který se naučíme, je příkaz pro smazání tabulky, ten je velice podobný svou skladbou. Používáme příkaz DROP TABLE jmeno_tabulky;

Pokud si ukážeme vše na příkladu, hned uvidíme, jak je příkaz jednoduchý:

```
DROP TABLE nase_prava_tabulka_podle_pravidel;
```

Jenže, a to si musíme uvědomit, jakmile odebereme tabulku, okamžitě s ní mizí i data, všechny záznamy, které obsahovala. Na databázích ORACLE existuje něco jako koš, z kterého se dají data do jisté doby obnovit, ale spousta databází takovouto věc vůbec nepodporují, proto nakládejme s tímto příkazem obezřetně.

Integritní omezení

Tento termín je někdy velice důležitý, protože s ním můžeme omezit, jaká data do databáze vložíme a tím do jisté míry ochránit obsah databáze. Někdy se jich používá pro omezení chyb při programování, či při úpravě vkládání před samotným vložením.

Prvním z nich je NOT NULL, s kterým se setkáme velice často. Stanovíme s ní, že pokud budeme vkládat záznam pro toto pole, kde je zadáno NOT NULL, musí záznam vždy vkládat něco do takovéhoto pole. Záznam, který nebude vkládat nic do tohoto pole, bude neplatný.

Ukážeme si vše zase na jednoduchém příkladě:

```
CREATE TABLE nase_prava_tabulka_podle_pravidel
(
    ID NUMBER (10) NOT NULL,
    jmeno VARCHAR(100) CONSTRAINT JMENO NOT NULL ,
    narodnost VARCHAR(50)
);
```

V první části jsme řekli, že ID nebude nikdy nulové, tedy že vždy bude při vkládání nutné do něj vložit nějaký záznam. Pak jsme dokonce vytvořili pojmenované pravidlo JMENO, které se týká toho, že jmeno bude muset obsahovat také data při vkládání záznamu. Jenže pro ID se nám bude hodit jiné pravidlo, které je naprosto stěžejní a příhodné.

Budeme používat pravidlo PRIMARY KEY, které se týká jednoznačné definice primárního klíče nad tabulkou, což znamená, že takový záznam bude pro celou tabulku jedinečný. Na příkladu to vše pochopíme.

```
CREATE TABLE nase_prava_tabulka_podle_pravidel
```

```
(
        ID NUMBER (10) PRIMARY KEY,
        jmeno VARCHAR(100) NOT NULL ,
        narodnost VARCHAR(50)
);
```

Teď víme, že ID bude primárním klíčem a jmeno bude vždy obsahovat hodnotu. Jak jednoduše jsme tímto nadefinovali pravidla, kterými se od té doby bude muset řídit každý, kdo bude používat naši tabulku.

Po pravdě můžeme definovat ještě jedno omezení, které nám stanoví jedinečnost při vkládání záznamu, je to UNIQUE KEY, kterým můžeme říct, že záznam pro toto pole bude také jedinečný. Příkladem budeme chtít, že tabulky bude obsahovat tři pole, kde ID bude klíčem, jmeno nesmí být nulové a narodnost bude vždy jedinečná. Je to sice jen příklad pro příklad, ale i tak si můžeme potom říci, že v naší tabulce nebude nikdo, kdo má stejnou národnost, jako někdo jiný z tabulky.

```
CREATE TABLE nase_prava_tabulka_podle_pravidel
(
        ID NUMBER (10) PRIMARY KEY,
        jmeno VARCHAR(100) NOT NULL ,
        narodnost VARCHAR(50) UNIQUE
);
```

Dokonce můžeme definovat omezení pro cizí klíč nad tabulkou, to však, jak potom poznáme, nemusíme přímo používat, pokud tabulku dokážeme využít tak, jak se má. Dokonce některé databáze ani toto omezení přímo nevyžadují a nějak moc nepodporují, ale je velice úžasné, když si nadefinujeme vazbu už při vytváření tabulky. (Určitě takový postup budeme používat nad databází ORACLE.)

```
CREATE TABLE nase_prava_tabulka_podle_pravidel
(
        ID NUMBER (10) PRIMARY KEY,
        ID_zeme NUMBER (10) NOT NULL,
        jmeno VARCHAR(100) NOT NULL ,
        narodnost VARCHAR (50) UNIQUE,
        FOREIGN KEY (ID_zeme) REFENCES zeme(ID)
);
```

Teď víme, že ID_zeme je cizí klíč, který má vazbu na tabulku země a na její ID. Tím dokážeme určit i vlastnost této vazby, ta je jedna země ku více našemu n ID z „nase_prava_tabulka_podle_pravidel“.

Dalším omezením je CHECK, který tvoří opravdový komfort, co se týče databází. Někdy se ale s touto variantou setkáváme jen u ORACLE. Při vkládání dat do databáze nám toto omezení zkontroluje, jestli je záznam dle našich omezení, které stanovíme. Nejčastějším omezením je podmínka BETWEEN.CREATE TABLE nase_prava_tabulka_podle_pravidel

```
(
        ID NUMBER (10) PRIMARY KEY,
        jmeno VARCHAR(100) NOT NULL ,
        vek NUMBER(3),
        narodnost VARCHAR(50) UNIQUE,
        CHECK (vek BETWEEN 1 AND 50)
);
```

Právě jsme stanovili, že vek musí být od jednoho roku do padesáti let, takže pokud vložíme záznam, kde věk bude větší jak padesát, bude záznam neplatný.

Další funkcí, kterou můžeme dost dobře vytvořit číselnou řadu u ID, je AUTO_INCREMENT, ale

musíme si uvědomit také to, že je to striktní funkce pro MySQL. Při definici tabulky jej jednoduše začleníme dle příkladu:

```
TABLE nase_prava_tabulka_podle_pravidel
(
        ID NUMBER (10) PRIMARY KEY AUTO_INCREMENT NOT NULL,
        jmeno VARCHAR(100) NOT NULL ,
        vek NUMBER(3),
        narodnost VARCHAR(50) UNIQUE,
);
```

Zde jsme se najednou seznámili dokonce se sřetězením příkazu. Naše ID se bude samo přidávat při přidávání nových záznamů a navíc bude primárním klíčem s nenulovou hodnotou. AUTO_INCREMENT vždy začíná na jedničce. Stále však mluvíme o výhodě, kterou rozhodně nevlastní databáze ORACLE.

Úpravy tabulek

S tabulkami můžeme pracovat i jiným způsobem, než je vytvářet. Představme si, že potřebujeme do stávající tabulky přidat další pole, další atribut. Nemůže pořád jen přidávat a odebírat celou tabulku. SQL nám pomůže jednoduchou sekvencí příkazů, kterými můžeme ubírat i přidávat další pole.

Základní příkazy si hned vypíšeme, abychom je mohli používat. Jsou jimi:

ALTER – je základním příkazem, kterým začínáme upravovat

ADD – využívá se pro vkládání nového pole tabulky

MODIFY – pro změnu v poli tabulky

DROP COLUMN – odstranění pole z tabulky

Pokud tedy budeme chtít editovat tabulku, příklad by mohl vypadat zhruba takto:

```
ALTER TABLE nase_tabulka
(
ADD (novy_sloupec datovy_typ integritni_omezeni),
MODIFY(nazev_stavajiciho_sloupce datovy_typ integritni_omezeni),
DROP COLUMN nazev_stavajiciho_sloupce
);
```

Tímto jsme do tabulky nase_tabulka přidali novy_sloupec, přeměnili jsme nazev_stavajiciho_sloupce na jiné omezení a datový typ a ještě jsme odebrali nazev_stavajiciho_sloupce. Možná se vám zdá příklad poněkud matoucí, ale někdy se setkáváme s takovým příkladem až na stovky řádků, to už poněkud matoucí opravdu je.

Omezení můžeme také přidávat a odebírat, jednoduše použijeme naše známé DROP.

```
ALTER TABLE tabulka DROP NOT NULL (jmeno);
```

Zde si můžeme povšimnout, jakým způsobem pracujeme se závorkami, protože pokud máme více věcí, které potřebujeme udělat, vkládáme je všechny do závorek. Pokud máme jen jednu, můžeme příkaz zkrátit a závorky vůbec nepoužít. Je nutné si zapamatovat takováto jednoduchá pravidla, abychom měli vždy jasno.

Někdy je potřeba celou tabulku přejmenovat, to provádíme jednoduchým příkazem RENAME, tedy:

```
RENAME tabulka TO nove_jmeno_tabulky;
```

Teď si ukážeme napřed něco, co patří spíše do jazyka DML. Budeme mluvit o DELETE a TRUNCATE. Když budeme chtít někdy vymazat záznamy z tabulky, je nutné použít klauzuli DELETE a pracujeme s ní jako s normálním výběrovým dotazem, příkladem:

```
DELETE FROM tabulka;
```

Tím vymažeme všechny záznamy z tabulky, což je poměrně delší řešení, ale v databázích ORACLE nám umožní použít takový databázový koš a pak z něj můžeme obnovit nechtěně vymazaná data. Dalším

příkazem může být TRUNCATE, který „vysype tabulku“ a to doslova a v ORACLE dokonce nedělá záznam pro budoucí obnovení.

Jeho použití je jednoduché:

```
TRUNCATE TABLE tabulka;
```

Pohledy

Pohledy jsou někdy jedinou možností, jak omezit uživatele, aby se v zásadě nedívali na část databáze, kterou nechceme, aby viděli. Při správě jednoduchých databází nejsou prakticky moc potřeba, ale je nutné si uvědomit, že někdy je využít prostě musíme. Také musíme vědět, že vytvořením náhledu nevytváříme nic stávajícího, představme si to spíše jako omezení na to, kam se smíme dívat a kam ne. Pro použití pohledů si ukážeme i jiné věci, než z jazyka DDL, ale nemusíme se bát, protože to nebude nic těžkého.

Pohled vytváříme naprosto jednoduchým příkazem, který nám bude určitě moc povědomý.

```
CREATE VIEW nazev_pohledu AS dotaz_na_databazi;
```

Na příkladu jednoduše pochopíme, jak náhled funguje.

```
CREATE VIEW nahled AS

SELECT * FROM tabulka WHERE vek = 50;
```

Právě jsme vytvořili pohled, který nahlíží do tabulky na záznamy, které mají zaznamenaný vek rovný padesáti let. Pak už můžeme pracovat s náhledem, jako s normální tabulkou a stejně tak z ní vybírat data. Stále ale pracujeme s náhledem, proto si musíme uvědomit jisté omezení.

```
SELECT * FROM nahled;
```

Tedy vyber vše z nahledu, to znamená z našeho náhledu, který jsme vytvořili.

Pokud budeme chtít změnit náhled, pomůže nám výborný dotaz, který dokonce dokáže s přidáním dalšího slova i náhled vytvořit, pokud do té doby neexistoval, což z něj dělá velikou výhodu.

```
CREATE OR REPLACE VIEW pohled AS novy_dotaz;
```

Tím můžeme měnit náš náhled.

Dokonce můžeme měnit data v náhledu, když není stanoveno pouhé čtení. Pracujeme s náhledem stále tak, že je jako tabulka, i když v tabulce máme data, která měníme, zatímco náhled pouze data z tabulek zobrazuje.

Pokud si přejeme zachovat pravidlo jen a pouze pro čtení, musíme použít klauzuli WITH READ ONLY. V příkladu to vypadá asi takto.

```
CREATE VIEW nahled AS

SELECT * FROM tabulka WHERE vek = 50

WITH READ ONLY;
```

Data budou dále pouze pro čtení a nikdo je nebude moci měnit.

Sekvence

Sekvence je v podstatě příkaz pro vytvoření číselné řady od n-tého záznamu k m-tému záznamu. V našich příkladech s ním můžeme plnit pole ID, protože některé databáze neumí příkaz AUTO_INCREMENT. Sekvenci zakládáme také pomocí CREATE.

```
CREATE SEQUENCE jmeno_sekvence START WITH zacatecni_cislo INCREMENT BY
cislo_ktere_se_bude pricitat;
```

V databázích ORACLE je pro sekvence dokonce uzpůsobené prostředí, takže se nemusíme bát, že je nepochopíme. Sekvence rušíme pomocí DROP následovně.

```
DROP SEQUENCE jmeno_sequence;
```

Indexy

Indexy jsou poměrně dost důležitou součástí databází, protože vytváří jistou podstrukturu databáze.

Představme si to tak, že pole, které je zaindexováno, je pro nás pole, o kterém máme někde uvedený seznam. Když zaindexujeme pole tabulky, které je často používáno, zrychlíme tím běh databáze, protože databáze samotná o něm bude mít vždycky nějakou známou povědomost.

U lékaře příkladem zdravotní sestra indexuje jména podle počátečního písmena a řadí je dle abecedy, tím ví, že když přijde někdo se jménem na n, najde ho v databázi pod jeho písmenem. Počítačová databáze dělá něco zcela identického, prostě jen prohlíží vybraná pole a to velice rychle, proto se indexace používá velice často.

Vytváříme ji zase pomocí CREATE.

```
CREATE INDEX jmeno_indexu ON tabulky (pole1,pole2,...,poleN);
```

Jako příkladem můžeme uvést vytváření indexu v databázi knih, kde je až milion knih.

```
CREATE INDEX knihy_index ON knihovna (nazev);
```

Index mažeme zase klauzulí DROP.

```
DROP INDEX knihy_index;
```

Komentáře

V SQL, jako v každém jiném jazyce, můžeme svoje skripty doplnit o jednoduché komentáře, které nám pomohou při úpravě starých skriptů, protože nám připomenou, co daný skript znamenal. Můžeme s nimi dokonce odříznout strukturu ve skriptu, kterou už nebudeme používat, ale nechceme se jí hned vzdát.

Známe dva typy komentářů, jeden je řádkový a druhý víceřádkoví.

/* komentáře */ - tento způsob platí pro víceřádkové komentáře a právě s ním můžeme odříznout nepoužívané části skriptu.

//komentář – platí jako jednořádkový komentář, kterým si můžeme zakomentovat třeba krátké poznámky ve skriptu.

Některé databáze povolují jejich vlastní komentáře a s nimi jejich vlastní značení začátku a konce vpisování komentářů, ale všechny rozhodně musí podporovat psaní lomítek a hvězdiček, protože je to standard SQL.

Cvičení

- Vytvořte tabulku s názvem Knihovna a poli: ID, jméno autora, jméno knihy
- Jak budeme postupovat, pokud potřebujeme rychle vyhledávat v ohromném počtu záznamů?
- Ve vytvořené tabulce Knihovna budeme chtít změnit pole jméno autora na pole jméno spisovatele, jak to uděláme?
- Chceme vytvořit vazbu mezi dvěma tabulkami, kde jedna obsahuje cizí klíč, jaké omezení použijeme?

Pokud jste v této kapitole nepochopili, k čemu a jak se používají sekvence, ještě tak moc se neděje, ale pokud vás minulo vytváření tabulek, zkuste se nad touto kapitolou více zamyslet a pochopit, v čem je vytváření tabulek, protože s tabulkami budeme pracovat nadále až do konce knihy.

Jediná tabulka, která je vždy vytvořena v ORACLE databázi, je tabulka DUAL, ale tento komfort máme jen v ORACLE, v jiných databázích si tabulku musíme vytvořit a DUAL rozhodně není tabulkou pro ukládání záznamů, proto si nesmíme myslet, že bychom se bez tabulek obešli.

Základy jazyka DML

Jazyk DML je naprostým základem pro práci s databázemi. Naučili jsme se, jak vytvářet tabulky, jak určit jejich integritní omezení, jak nastavit datové typy, ale ještě zdaleka netušíme, jak data z databáze dostat. Tím se dostáváme k jazyku DML, který pro nás bude tím pravým a dokonalým řešením pro vybírání, vkládání, obnovení a mazání dat. Ačkoliv jsme stále začátečníci, kteří se jen seznamují se začátky jazyka SQL, kapitola DML pro nás musí být zcela jasná a srozumitelná, protože právě ta nám bude vždy pomáhat, když si nebudeme moci vzpomenout.

Pokusíme se proniknout do jednotlivých zákonitostí a struktury této části jazyka. Vysvětlíme si, jak jednoduše data zpracovat.

Náhled na tabulku

Někdy je nutné se podívat na tabulku, abychom mohli určit, jaké má vlastnosti, jaké ji tvoří pole. Je nutné dříve zjistit, s čím budeme pracovat, než doopravdy začneme vybírat data. Takový příkaz, který nám ukáže vlastnosti tabulky, se jmenuje DESC, je to vlastně zkratka pro describe, tedy pro popis.

Syntaxe příkazu je velice jednoduchá:

DESC tabulka;

Pokud si zobrazíme takovýto náhled na tabulku, zobrazí se většinou pole v takovémto pořadí:

jméno tabulky

pole

datový typ

délka

přesnost

váha

primární klíč

nulovost

defaultní hodnota

komentář

Jsou to vlastně seznamy jmenných vlastností, které tabulka má, seznamy polí a jejich vybavení. Často se setkáváme s takovým zápisem, když nám někdo potřebuje ukázat svůj nákres databáze s tabulkami.

Vybírání dat

Vybírání dat je zcela základní a nejdůležitější věcí, kterou od databáze očekáváme, proto u vybírání strávíme nejvíce času. Je nutné si uvědomit, že více jak padesát procent dotazů na databázi jsou dotazy s výběrem, ať je to pouhé vybrání uživatelů, či vybírání našich dat. Sama databáze pracuje s vlastními tabulkami, v kterých jsou obsaženy systémové proměnné, a vybírání praktikuje skoro neustále. Příkladem databáze ORACLE si vede všechny údaje o uživatelích v systémové tabulce DBA_USERS a pokud z ní něco potřebujeme zjistit, také musíme použít výběr.

Vybírání provádíme pomocí příkazu SELECT. Jak jsme si již uvedli předtím na příkladu pomocí lékařského záznamu v ordinaci, výběr SELECT je velice přirozený a podobný příkazu anglickému, stejně tak jeho skladby věty. Vždy bychom měli zadávat příkaz, potom co chceme vybírat a následně z čeho budeme vybírat. Také připojujeme další poddotazy, ale s těmi se zatím nezatěžujme.

Skladba příkazu SELECT je následující:

SELECT seznam_sloupců,_které_chceme_vybrat_oddělený_“,“_nebo_pouze_jeden_sloupec

FROM jméno_tabulky,_která_je_v_databázi

WHERE zde_vpisujeme_podmínky,_které_musí_dotaz_splnit

GROUP BY položky,_které_budeme_používat_k_řazení,_tedy_položky_agregace

HAVING podmínky_agregace,_tedy_položek,_které_budeme_třídit

ORDER BY položky,_které_budeme_řadit [ASC | DESC] – kde ASC znamená ascendentně a DESC je descendentně

Všechny tyto části si vysvětlíme, takže se nebojte, že nebudete nějaké z nich rozumět. Začneme s jednoduchými příklady, které určitě pochopíme. Rozhodně se nemusíme bát, že při výběru budeme muset napsat tak dlouhý příkaz se všemi klauzulemi, protože jednoduše nejsou někdy vůbec potřeba. Tím se nám ulehčuje práce o hodně času, protože můžeme praktikovat výběr i na pouhá čtyři slova, vše si ukážeme.

SELECT * FROM tabulka;

ID	jmeno	prijmeni
1	Karel	Osáhlo
2	Petr	Dobrotivský
3	Johan	Ohnutý
5	Richard	Druhý
6	Richard	Třetí

Zde vidíme, že nám stačí pouhé tři slova a jeden regulární výraz. Hvězdička je zde zcela jednoduše pro vybrání všech záznamů, tedy všechna písmena bez omezení velikosti. Můžeme takto vybrat zcela vše z tabulky „tabulka“.

Výpis by pak byl možná trochu obsáhlejší, ale přesto bychom se ke všem datům dostali. Pokud ale chceme vybrat jen určité sloupce, aby náš výběr nebyl obrovského ražení, omezíme dotaz hned na začátku po klauzuli SELECT.

Náš dotaz by pak mohl vypadat takto:

SELECT jmeno FROM tabulka;

jmeno
Karel
Petr
Johan
Richard
Richard

Teď vidíme, že jsme vybrali pouhá jména, která se nám dokonce vyrovnala naprosto stejně, jako v předchozím dotazu. Nemusíme se pouze omezovat na jednu položku, ale čárkou oddělit i další, které chceme vybrat. Nakonec, když nechceme přímo pracovat s ID, můžeme vybírat pouhé jméno a příjmení z naší cvičné tabulky.

Celý výstup můžeme navíc zmenšit s použitím klauzule LIMIT [číslo], kde číslo označuje počet záznamů. Někdy se nám takový počin hodí, když potřebujeme vybrat právě jeden záznam.

SELECT jmeno, prijmeni FROM tabulka LIMIT 4;

jmeno	prijmeni
Karel	Osáhlo
Petr	Dobrotivský
Johan	Ohnutý
Richard	Druhý

Teď jsme si ukázali, jak omezit pole, s kterými pracovat, jenže to není zdaleka vše, co bychom měli udělat. Pokud někomu ukážeme náš výpis, ve kterém je pouhé příjmení a jméno, neřekne nám asi zdaleka nic o povaze tabulky. My můžeme už ve výběru stanovit jakási zástupná jména, která budou zastupovat skutečné názvy polí. Tím zcela odstíníme původní jméno.

Proč bychom to však měli dělat, když se v tabulkách vyznáme? Důvod je zcela jednoduchý, ostatní nemusí. Někdy je nutné mít v tabulkách spousty jmen, a proto je odlišujeme jiným názvem pole, příkladem: jmeno_auta, jmeno_pes, či pomocí zkratky jm_ta, jm_es – kde bereme první dvě písmena, mezeru

zachováme a dopíšeme poslední dvě písmena. Někdy je nutné takový standard použít a některé databáze mají i vlastní teorii, jak názvy psát. Jenže, co když chceme někomu ukázat naše data a nechceme mu vždy vysvětlovat, jaké data právě vidí? Pak je naším řešením zástupný název.

Vše si ukážeme na příkladu:

```
SELECT
jmeno AS “Jméno v tabulce“,
prijmeni AS “Příjmení v tabulce“
FROM tabulka;
```

Jméno v tabulce	Příjmení v tabulce
Karel	Osáhlo
Petr	Dobrotivský
Johan	Ohnutý
Richard	Druhý
Richard	Třetí

Teď jsme vlastně jen přejmenovali pole v tabulce, ale jen zástupně, stále můžeme pracovat s normálními poli, ty jsou stále se stejnými jmény.

Někdy je potřeba náš výstup zúžit o záznamy, které jsou duplicitní. Databáze sama nepozná, že chceme jen vybrat záznam pro každé jméno jeden, že nechceme všechny záznamy, i když v tabulce určitě jsou. Řešení je naprosto jednoduché, dokonce jej některé uživatelsky nenáročné databáze používají bez vyzvání.

Někdy můžeme používat zástupné názvy daleko rychleji a mobilněji, než čekáme. Je možné zvolit zástupný název pro tabulky už v dotazu. Adresaci pomocí teček si u toho ukážeme tak, jak se nejčastěji používá.

```
SELECT v.jmeno, k.jmeno
FROM veze v, kraj k;
```

Jednoduše jsme na poměrně dost nefunkčním příkladě ukázali, že název tabulek můžeme uvést jako jeden znak, což urychlí naší práci. U tohoto zástupného značení stanovujeme pracovní název a nepoužíváme AS.

Klauzule má jméno DISTINCT a ve výběrovém dotazu ji vkládáme přímo za klauzuli SELECT, ukážeme si ji na příkladu.

```
SELECT DISTINCT jmeno FROM tabulka;
```

jmeno
Karel
Petr
Johan
Richard

Naší obětí teď bude Richard Třetí, protože má naprosto totožné jméno, jako jeho předchůdce Richard Druhý. Bohužel se nám ale může stát, že se nám neukážou ani duplicitní záznamy, které sice nemají stejné jméno, ale jiné pole, které jsme přidali do polí, které chceme vybrat. DISTINCT prostě ze všech uvedených polí vybere ty jedinečné. Měli bychom také rozlišovat, jaký záznam je stejný a jaký ne, protože záznam Karel a karel nejsou stejné, dokonce může být rozdíl mezi „Karel“ a „Karel “, kde je o jednu mezeru navíc. DISTINCT tento údaj vyhodnotí jako rozdílný a zobrazí jej a stále to není chyba databáze, ale chybně vložený záznam.

Už víme, jak vybrat data z databáze, ale co když je chceme nějak omezit? Co když nebudeme chtít všechna jména, ale jen nějaká jména a u nich záznamy z jiných polí? Tomu všemu a ještě mnohem více, pomáhá klauzule WHERE. Ta, jak nám název napovídá, určí, kde budeme vybírat.

Klauzule WHERE se dá omezovat pomocí operátorů. Z jiných programovacích jazyků můžeme znát

podmínku rovnosti, zde se uvede pouhé jednoduché „=“, to nám zadává, že oba záznamy si budou rovny.

Operátor	Význam
=	rovnost
<	menší než
>	větší než
<=	menší a zároveň rovno
>=	větší a zároveň rovno
<>, =!	různé
IS NUL	je nulový
IS BETWEEN	je mezi, kde se přidává další význam pomocí AND, tedy IS BETWEEN 1 AND 5
IN	patří do množiny

WHERE se dá dokonce zvětšit o přidání logických operátorů:

AND - logické „a“, kde obě podmínky musí platit, aby byl výraz pravdivý

OR - logické „nebo“, kde stačí, aby jeden záznam byl pravdivý, a zbytek výrazu je pravdivý

Vše si ukážeme na příkladu. Pro jednoduší pochopení vezmeme tabulku se seznamem věží, která vypadá následovně.

ID	jmeno	vyska	pocet_schodu
1	Blaník	50	234
2	blaník	50	234
3	Petřín	150	345
4	Lidojedka	300	768
5	Vytáhlice	356	980
6	Schodnatka	780	999

Zde vidíme poněkud podivné věže, kde výška je udávána v metrech. Neberme teď v potaz fakt, že jsme zapisovali čísla bezrozměrně, protože výpis bude řešit něco jiného, než databáze a my si ukážeme, jak výpis poměrně dost jednoduše zřetězit, aby ukázal, jak vypsat i značku m.

```
SELECT ID, jmeno, vyska FROM veze WHERE vyska >= 300;
```

Výpis z minulého příkladu vypadá následovně:

ID	jmeno	vyska
4	Lidojedka	300
5	Vytáhlice	356
6	Schodnatka	780

Jak jsme viděli, zbavili jsme se výpisu věží menších než 300 metrů. Teď si ještě dotaz upravíme, aby nám dosadil jednoduchou značku m.

```
SELECT ID, jmeno, vyska || ' m ' AS výška FROM veze WHERE vyska >= 300;
```

A výpis bude doplněn o malé m:

ID	jmeno	vyska
4	Lidojedka	300 m
5	Vytáhlice	356 m
6	Schodnatka	780 m

Teď už jsme vytvořili poměrně dost jednoduchý dotaz, který nám ukáže požadovaná data i s podmínkou. Přidáme si ještě do podmínky AND, které přidá další podmínku do celého výrazu, a přidáním

OR ještě více zamotáme, co jsme původně chtěli.

```
SELECT ID, jmeno, vyska || ' m ' AS výška FROM veze WHERE vyska >= 300 AND pocet_schodu > 500 OR jmeno='Blaník';
```

Tento dotaz je poněkud zamotaný, ale zamysleme se. Vyska je větší, než 300 metrů, pocet_schodu musí být více, jak 500, ale jmeno může být Blaník, takže se nám ve výpisu ukážou minulé věže, ale přibude věž Blaník, která je ve výpisu právě pro podmínku OR.

Pro vyhodnocování jednotlivých podmínek často používáme jednoduchých závorek, které nám podmínky zabalují do samostatných částí a jen když celá závorka platí, pak je podmínka pravdou.

Je nutné si uvědomit, s jakým datovým typem pracujeme, totiž, pokud požadovaná data jsou číslem, nemusíme v podmínce WHERE psát jednoduché uvozovky ' ', ale stačí pouhé číslo. Pokud používáme podmínku, která se skládá z datového typu VARCHAR, či CHAR a další, pak musíme oddělit požadovaný záznam pomocí uvozovek ' ', tím vlastně označíme, co má databáze považovat za hledaný záznam a kde jej hledat nemá. Vše si určitě ukážeme na příkladech, kterých bude velice mnoho.

Jak jsme si již řekli, není stejné uvádět záznam karel jako KaReL, protože oba Karlové mají jiná velká, či malá písmena. Měli bychom si tedy vymyslet nějaké vhodné vložení funkcí, které nám výběr okleští. Mějme na paměti, že úpravu budeme provádět v obsahu klauzule WHERE, která bude provádět vyhledávání.

```
SELECT jmeno FROM veze WHERE UPPER(jmeno) = UPPER('blaník');
```

Vycházíme z předchozího příkladu, kde jsme měli dva záznamy o věži Blaník, ale jeden z nich byl zaznamenán jako blaník, tedy s malým písmenem na začátku. Výpis by se pak týkal pouhých dvou věží, tedy našich duplicitních Blaníků.

jmeno

Blaník

blaník

Můžeme této funkce využít i jinak než na hledání duplicitních záznamů, ale většinou je používána právě na hledání takových věcí, které jsou chybně napsány. Tím ale neříkáme, že UPPER je používáno jen na těchto místech.

Nejjednodušší je ale používat klauzuli LIKE, které nám velice pomůže při hledání ve více záznamech. Můžeme pomocí LIKE rozšířit podmínku WHERE a tím vylepšit námi požadovaný výpis. Databáze ORACLE poskytuje také řešení pomocí klauzulí REG, které zde ani nebudeme uvádět, protože ve většině databází si vystačíme s LIKE.

Musíme si však napsat, jaké regulární výrazy známe a můžeme použít.

Znak	Význam
* či %	všechny libovolné znaky v jakémkoliv množství
.	jakýkoliv jeden znak
|	alternativní operátor pro jiné možnosti
^	začátek řádku
$	konec řádku
[]	seznam jednotlivých vyhledávaných znaků
{m,n}	shoduje se nejméně mkrát a ne vícekrát, než nkrát

Regulární výrazy jsou velice používány a musíme si uvědomit, že mnohdy mají různý význam. Jiné použijeme pro databázi, trochu jiné v jiných programech. V Klauzuli LIKE použijeme nejčastěji výraz %, který nám dá libovolný počet jakýchkoliv znaků.

Teď už si jednoduše můžeme najít věž Blaník bez toho, abychom znali celé jméno a navíc, abychom věděli, jak písmena jsou malá a velká.

```
SELECT jmeno FROM veze WHERE UPPER(jmeno) LIKE UPPER('b%');
```

Výpis bude stejný, jako v minulém případě, jen teď nemusíme znát prakticky nic, než začáteční písmeno. Tento způsob se používá spíše pro vyhledávání, tedy většinou, když vyhledáváme nějaký text podle klíče, který dáme ke klauzuli LIKE.

jmeno

Blaník

blaník

Pokud ale potřebujeme nějaký výpis, který se týká více záznamů, a nevíme, kde přesně jej nalézt, můžeme využít BETWEEN, které nám vybere data od do zadaných hodnot. BETWEEN lze použít i pro datum, ale my zůstaneme u našich věží. Kdybychom však potřebovali udělat výpis dle data, nesmíme zapomenout na uvozovky ' '.

Pokud budeme chtít naše věže vybrat

```
SELECT jmeno FROM veze WHERE pocet_schodu BETWEEN 300 AND 900;
```

jmeno	Ukázka počtu schodů, který se ale ve výpisu neukáže!
Petřín	345
Lidojedka	768

Vidíme, že výpis obsahuje jen záznamy od 300 do 900 metrů, což dělá klauzule BETWEEN, v překladu ji uvádíme jako „mezi“.

Do teď jsme pracovali s výpisem dat, která se vypisují podle toho, kdy jsme je vložili. Totiž, když vkládáme do tabulky data a netřídíme je podle nějakého klíče, tak se při výpisu řadí podle toho, kdy jsme je vložili. Takže pokud vložíme záznam1 a po něm záznam2, vypíše se nejdříve záznam1, protože jsme jej vložili jako první a jeho čas vložení je menší než čas záznamu2.

Řazení uplatňujeme pomocí klauzule ORDER BY. Musíme určit, podle čeho se budou data třídit, nemusíme však řešit, že třeba číselná řada není souvislá, stačí, že se dá řadit podle své hodnoty.

Syntaxe příkazu ORDER BY je následující:

```
SELECT [ * | či_seznam_polí ] FROM tabulka ORDER BY sloupec [ASC | DESC ];
```

Pokud necháme pouhé ORDER BY, výpis se bude řadit ASC, tedy seřazen sestupně podle zadaného sloupce.

ASC..... řadí sestupně

DESC..... řadí vzestupně

Většinou se řadí podle data, ale častěji se používá právě umělý klíč ID, který se řadí sám od sebe podle vložení záznamu a čísla předchozího záznamu. Příklad bude vypadat takto:

```
SELECT ID, jmeno FROM veze ORDER BY ID DESC;
```

Výpis bude právě opačný, než normálně dle ID:

ID	jmeno	vyska	pocet_schodu
6	Schodnatka	78	999
5	Vytáhlice	356	980
4	Lidojedka	300	768
3	Petřín	150	345
2	blaník	50	234
1	Blaník	50	234

Otočit celý výpis je někdy velice důležité, protože tím otočíme výpis záznamů, které jsme vložili jako poslední, tím je zobrazíme jako první. Když si představíme, že napíšeme diskuzi, kde by se nevypisovaly poslední záznamy na začátku, museli by všichni, kteří by se chtěli podívat na aktuální příspěvek, podívat až na konec výpisu, což je dost nevhodné.

Už jsme si říkali, že pracujeme s relační databází, proto musíme uvažovat i klauzuli JOIN, která je pro databáze velice užitečná. Před příchodem klauzule JOIN se mohlo použít klauzule WHERE ve své normální formě, prostě a jednoduše dáme rovny ID záznamu s cizím klíčem v jiné tabulce. JOIN je pak jednoduší formou, protože nám pomůže stanovit vazby v horní části dotazu a tím zajistit jistou přehlednost.

Ze začátku používání JOIN se říkalo, že tato klauzule zpomaluje databázi, ale ať použijeme WHERE, či JOIN, databáze je stále stejně rychlá. Prakticky spíše záleží na záznamech samotných, než na

cestě, kterou zvolíme, protože když nastavíme špatný datový typ, uškodíme rychlosti daleko více, než použitím staršího spojení tabulek.

Pomocí WHERE určíme spojení pomocí podmínky rovnosti takto:

```
SELECT tabulka1.pole1, tabulka2.pole2, tabulka3.pole3, …

FROM tabulka1, tabulka2, tabulka3, …

WHERE tabulka1.pole1 = tabulka2.pole2
```

Všimněme si, že je použito vybírání více polí z více tabulek a jejich cestu zadáváme pomocí teček, tedy tabulka.pole vede do tabulky a jeho pole. Tímto způsobem můžeme vybírat daleko více dat, než z jedné tabulky.

Pokud příklad upravíme do nějakého případu v praxi, jednoduše pochopíme, jak to vše funguje.

```
SELECT kraj.id_veze,

    veze.jmeno

FROM kraj, veze

WHERE kraj.id_veze=veze.ID;
```

Tedy, vybíráme cizí klíč z tabulky kraj pro věž a jméno věže z tabulek kraj a věž a pak spojíme krajský cizí klíč a ID věže, takže uděláme 1:N relaci. Prakticky tento vztah dává smysl leda, že bychom měli jednu věž ve více krajích.

Jenže pomocí JOIN můžeme stanovit i jiné věci, než přímé spojení. Můžeme stanovit levé spojení, pravé spojení a mnohem více. To dělá JOIN tak obdivovaný a často používaný. Syntaxe je velice jednoduchá, přidáváme ještě klauzuli ON.

```
SELECT tabulka1.pole1, tabulka2.pole2, …

FROM tabulka1

[ INNER | FULL | LEFT OUTER | RIGHT OUTER] JOIN tabulka2

ON tabulka1.pole1 = tabulka2.pole2;
```

V poli s [ ] vidíme, kolik typů spojení JOIN je. Na rozdíl od WHERE, kde bychom tyto vlastnosti splňovali těžko, je JOIN velice regulativní. Uvedeme si, co které spojení znamená.

CROSS JOIN – moc často se nepoužívá, protože provádí kartézský součin jednotlivých záznamů v polích (každý s každým). Výpis je pak tím větší, kolik máme záznamů. V praxi se tato možnost skoro nepoužívá, protože já málo praktických situací, kde by se vůbec dal nasadit. Můžeme podle něj vytvářet jakési matematické příklady, či složitou práci s daty.

FULL JOIN – plné spojení, které nevytváří sice křížový spoj, ale zachovává ve výpisu i záznamy, které nesplňují spojovací kritérium. Takovýto spoj je vhodný při výpisech, kdy potřebujeme doopravdy všechny záznamy, ale někdy je tato věc velice nepraktická.

INNER JOIN – když vynecháme INNER, pak samotné JOIN používá toto INNER spojení, tedy vnitřní. Při použití WHERE se právě provádí INNER spojení. Prakticky se jedná o vnitřní spojení, kde cizí klíč se rovná klíči.

LEFT OUTER JOIN – méně používaný spoj, kde se spojují záznamy, které nemají splněné spojovací kriterium, ale berou se pouze záznamy z levé části.

RIGHT OUTER JOIN – také se moc nepoužívá a zpracovává záznamy, které nemají splněné kriterium, ale zprava.

Pokud si představíme tato spojení, měli bychom si říci, že není potřeba ovládat levý, či pravý spoj, ale je nutné si uvědomit, jak spojujeme pomocí INNER. Vnější spoje jsou dost zvláštní a v praxi jsou méně časté.

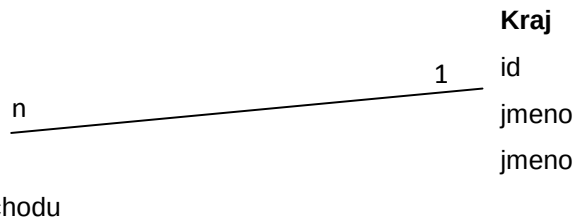
Převedeme si minulé spojení, jak jsme jej použili v případě věží a krajů pomocí INNER JOIN, teď ale správně upravíme tabulku věží, aby mohla mít cizí klíč kraje. Více věží teď může ležet v jednom kraji.

```
SELECT k.*, v.*

FROM kraj k

INNER JOIN veze v ON k.id = k.id_veze;
```

Spojení je zase 1: N a spojujeme jeden kraj k více věžím. Když si spojení rozepíšeme, pochopíme, jak je jednoduché.



Jenže databáze znají ještě jedno spojení, které se často používá. Můžeme spojit dva výpisy sloupců, které spolu vzdáleně ani nesouvisí. Takový výpis je používán klauzulí UNION. Výpis pak skládá záznamy vedle sebe bez relace.

```
SELECT sloupce FROM tabulka1
UNION
SELECT sloupce FROM tabulka2;
```

Musíme však mít stejný počet sloupců v obou dotazech shodné.

Do teď jsme vybírali data zcela bez nějakých určitých omezení. Prostě jsme je našli, maximálně určili pomoci WHERE jaká, ale nerozhodovali, jestli jsou to maxima, či suma, jen jsme je vybrali. Databáze umožňují pomocí GROUP BY a agregačních funkcí vybrat data i se sumami a dokonce vytvořit i průměr.

Ukážeme si, jak má vypadat taková skladba pro agregaci.

```
SELECT záznamy,_které_budeme_zpracovávat FROM tabulka
WHERE podmínka
GROUP BY položky_pro_seskupení
ORDER BY sloupce,_které_setřídíme;
```

O funkcích, jako je AVG a SUM si ještě povíme dále, ale je nutné vědět, že pokud začneme pracovat s GROUP BY, pak seskupujeme jednotlivé dotazy a začneme je řadit. Výhodou GROUP BY je, že můžeme použít funkci HAVING, která nám omezí výpis na menší počet záznamů. Syntaxe se nám potom může zvětšit o HAVING, které bude právě užívat AVG, či SUM, tedy agregační funkce.

```
SELECT záznamy,_které_budeme_zpracovávat FROM tabulka
WHERE podmínka
GROUP BY položky_pro_seskupení
HAVING podmínky_agregovaných_dat
ORDER BY sloupce,_které_setřídíme;
```

Na příkladu si můžeme tedy snadno omezit výpis našich věží jen na ty, které mají počet schodů větší, než je jejich průměr. Vše uvidíme v příkladu:

```
SELECT jmeno
FROM veze
GROUP BY pocet_schodu
HAVING AVG(pocet_schodu) < pocet_schodu;
```

Vložení dat

Už jsme se naučili data vybírat a dokonce pomocí podmínek seskupování, ale jak data do databáze vložit? K tomu slouží jednoduchý příkaz INSERT. Samotná klauzule má poněkud jinou skladbu, než jiné dotazy, i když ji můžeme skládat s jinými dotazy.

Syntaxe vypadá takto:

```
INSERT INTO tabulka
```

(seznam_polí)

VALUES

(data_do_sloupců);

Mějme na paměti, že počet sloupců se shoduje s počtem nově vkládaných dat do polí. Příklad na vložení nového řádku by vypadal asi takto:

INSERT INTO veze (ID , jmeno , vyska , pocet_schodu) VALUES ('7' , 'Sněžka' , '789' , '98767');

Tímto jsme jednoduše přidali nový záznam do tabulky veze. Někdy se nám počet přidaných dat navýší na ohromné množství řádků, je tedy lepší udržet jisté pravidlo a každý nový kousek dat psát na nový řádek. Pokud některé pole nechceme vyplnit a jeho integritní omezení nám to dovoluje, nemusíme jej vkládat, ale nenapíšeme jej ani do závorek.

Upravení dat

Někdy je potřeba záznamy upravit, kdybychom nevěděli, jak na to, musíme daný záznam vždy vyjmout a potom znovu vložit, což je velice náročné a poměrně dost zdlouhavé. Nejrozumnější je použít příkaz UPDATE. Pomocí UPDATE můžeme dokonce upravit více záznamů najednou, protože můžeme použít WHERE. Syntaxe příkazu je následující:

UPDATE tabulka

SET sloupec1 = hodnota, sloupec2 = hodnota, …

WHERE podmínka;

Musíme si uvědomit, že když nepoužijeme WHERE záznamy v tabulce se upraví všechny, proto vždy omezujeme kde se má UPDATE použít. Nejčastěji se setkáváme s použitím WHERE na pole s ID, protože to je vždy tak jedinečně, že nám nedovolí omylem upravit jiný záznam.

Příklad může vypadat zhruba takto:

UPDATE veze

SET jmeno = 'Blaníček' , vyska = 700

WHERE ID = 1 ;

Můžeme dokonce upravit více záznamů najednou:

UPDATE veze

SET jmeno = 'Velká věž'

WHERE vyska BETWEEN 1000 AND 1500;

Vymazání

Už víme, jak vymazat celou tabulku, ale jak vymazat pouhý jeden záznam? K tomu slouží klauzule DELETE. Syntaxe je velice podobná SELECT, ale vynecháváme regulární výraz *.

DELETE FROM tabulka WHERE podmínka;

Zase platí nepříjemná skutečnost, že když nenapíšeme podmínku, vymažou se všechny námi zadané záznamy. Při klauzuli DELETE se ale zachovává v některých záznamech možnost vrácení kroku zpět, dokud nedojde k vypršení časového limitu pro navrácení. Vytváří se tedy záznam o vymazání.

Na příkladu si ukážeme, jak budeme pracovat s tímto příkazem:

DELETE FROM veze WHERE ID=1;

Vymažeme podle klíče záznam pro věž Blaník. Vymaže se celý záznam, nenechává se z něho nic.

Jinou variantou je vysypání celé tabulky, při které se nevytváří zápis. My víme, že když použijeme DELETE bez podmínky, vymažeme záznamy z celé tabulky, ale tento krok je někdy velice zdlouhavý, protože vytvářet záznam pro případné obnovení je někdy dost náročné. Proto se používá příkaz TRRUNCATE, který doslova bez zápisu pro obnovu vyhodí všechny záznamy z tabulky.

Syntaxe je následující:

TRUNCATE TABLE veze;

A tímto se zbavíme všech věží, které jsme zapsali.

Cvičení

- Jakým způsobem spojíte dvě tabulky pomocí JOIN, když chcete všechny záznamy, které mají mezi sebou vazbu jedna ku mnoha?
- Jakým způsobem zjistíte, jaké pole tabulka obsahuje?
- Jaký příkaz použijeme, když chceme vymazat záznamy z tabulky a s nimi chceme udělat případné obnovení?
- Jak upravíme tabulku, abychom staré hodnoty přepsali a použili u toho podmínku?
- Je možné upravit více záznamu najednou a jak?
- Jak řadíme data, aby byla vzestupně uspořádána?

Základy jazyka DCL

SQL obsahuje jazyk, který podporuje práci s uživateli databáze. Představme si, že můžeme vytvořit skript, který nám najednou přidá, či odebere uživatele. Dokonce můžeme tuto variantu přidat do nějakého programu, kde budeme pracovat s databází uživatelů přímo a vybírat a odebírat uživatele dle toho, jak zrovna budeme potřebovat.

Dokonce si můžeme takto uživatele oddělit a dát jim jiná práva. Měli bychom myslet také na to, jak moc budeme nechávat uživatelům volný prostor. Co když potřebujeme uživatele, kteří pracují jen s vybíráním dat? Jako databázoví uživatelé a databázoví administrátoři bychom si měli i nad naší vlastní databází udělat jistou strukturu práv a uživatelů.

Někdy potřebujeme vytvořit nového uživatele pro někoho, kdo by se mohl připojit a podívat se, co je kde špatně, či mohl zkontrolovat naši práci. Je důležité, pokud pracujeme na něčem důležitém, mít někoho, kdo nám poradí, či pomůže.

Měli bychom také myslet na to, že pokud necháme jen jednoho uživatele, tedy toho, který je vytvořen ze začátku jako administrátor: pro ORACLE je to SYS a SYSTEM, pro MySQL root, atd, může se nám stát, že budeme pořád používat jen jeho, používat jeho heslo, používat jeho oprávnění. Ze zásady nemůžeme hlavní uživatele omezit a potom, když někdo uvidí náš skript, kde se uživatel přihlásí, může nás čekat nemilé překvapení.

Jak jsem řekl, hlavní uživatelé nemohou být omezováni, a proto jsou vládci, které nemůžeme ohlídat. Je to velice jednoduché řešení, prostě vždy potřebujeme někoho, kdo zpočátku udělá základní nastavení a nechá ostatní používat systém. Také musíme vědět, že hlavní uživatel má také hlavní prioritu při vytváření a hlavně při práci, takže pokud budeme dělat něco na databázi a připojí se administrátor, systém vyčlení prostředky hlavně pro jeho práci. Systém se nechová špatně, protože administrátor potřebuje mít vždy přednost, ale když budeme vždy pracovat jen jako on, budeme bezdůvodně omezovat ostatní, což je poněkud nevhodné.

Uživatelé a jejich vytvoření

Vytváření je podobné jako jakýkoliv jiný vytvářecí dotaz:

```
CREATE USER jméno_uživatele IDENTIFIED BY heslo [PASSWORD EXPIRE];
```

Nejdříve napíšeme jméno nového uživatele, pak vložíme heslo. Mysleme na to, že vytvářet uživatele můžeme pouze jako administrátoři. Při vytváření hesel bychom si měli říci jaké, protože ve skriptech nejsou hesla nijak chráněna, nijak zakryptovaná, proto bychom měli vědět, jaké heslo použít.

Vždy bychom měli uvažovat jen silná hesla. To jsou taková, která splňují pár bodů:

První pravidlo je, že si heslo musíme zcela pamatovat. Nesmíme si je psát na papír a už vůbec je pokládat na veřejná místa, kde se k nim někdo může dostat. U databází se to může týkat také toho, kam ukládáme naše skripty, kde hesla máme napsaná. Tím totiž otevíráme cestu naprostým neznalcům serveru, kteří měli pouhé štěstí a dostali se k heslům nějakou nevinnou cestou, kterou jim ani nemůžeme mít za zlé, protože jsme prostě neschovali, co jsme měli.

Další zásadou je nepoužívání slovníkových hesel, která jsou tvořena celým slovem, to navíc dává smysl a útočníkovi někdy stačí rozluštit jen pár písmen, aby doplnil zbytek. Připravujeme se tím o možnost jisté ochrany, protože slovníková slova jsou zcela známá a pokud nepoužijeme nějaký neznámý jazyk, dají se pomocí slovníku v počítačové formě dosazovat ve frekvenci až tisíce do vteřiny, takže takový český jazyk se vyzkouší vložit do přihlašovacího formuláře během pár dní.

Tím ale také nesmíme používat jména, která jsou zcela známá. Jména jsou častým heslem uživatelů, kteří nemohli nic vymyslet. Proto se někdy dozvídáme, že pan Karel má heslo Karel. Na světě jsou jména tak malým okruhem slov, že pro jejich celé přezkoušení, pro dosazení při zkoušení hesla, stačí ještě méně času, než zkoušet celý slovník.

Dalším krokem je nepoužívat telefonní čísla, která mají malý počet cifer a navíc se drží jednoduchých zásad. Dokonce se někdy může stát, že když použije každý své telefonní číslo, čísla se budou lišit jen posledními třemi ciframi. Když tedy uvážíme změnu pouze v třech cifrách, máme jen 999 možností

platného hesla.

Dokonce ani důležitá data nejsou to pravé, protože ta jsou omezena soustavou 30 a 12 a když přidáme rok, moc to nevylepšíme. Útočník rozhodně nebude zkoušet data, která jsou zcela známá, ale určitě zkusí data o dvacet let zpět a dvacet let kupředu, což není moc dlouhý seznam možností.

Je velice důležité kombinovat znakové sady čísel, písmen a symbolů. Příkladem může být písmeno ř, které máme jen my a africký národ, který, jak je mi známo, neví, co je to Internet. Prakticky, když budeme kombinovat zvláštní symboly, docílíme poměrně dost dobré ochrany. Za zvláštní symbol berme i mezeru, protože i ta je kus hesla. Některé programy mezery neřeší a proto je někdy pro ně velice těžké zastavit se třeba na týden u jedné pozice v heslu, která byla mezerou. Budou prakticky neustále hledat, co by mohli dosadit, ale ono tam nic není.

Když mluvíme o znacích, musíme si uvědomit, že velké písmeno není malé písmeno a naopak. Když budeme kombinovat malá a velká písmena, budeme měnit celý znak a jeho náležitost v tabulce ASCII. Prakticky existuje seznam pro znaky malých písmen a pro znaky velkých písmen. Pro nás není změna ve významu, jen přehazujeme velikost. Jednoduché a účinné.

Další vylepšení je přidání číslic do textu. Je těžké nějak logicky odůvodnit takové heslo a proto si jej potom vybavit, tak musíme volit nějakou znělou variantu. Každý zná slovo z5, kde z je písmeno a 5 má psaný význam pět, proto se dá takto napsat zpět. S podobnou úvahou si můžeme vystačit i u hesla, ne zrovna se slovem zpět, ale s nějakým jiným.

Další částí je délka hesla, která je velice důležitá. Jak jste si již všimli, případný program bude zkoušet od jedné pozice k druhé, třetí až možná sté, ale když necháme jen dva symboly a tyto symboly budou čísla, máme jen 100 možností. Některé kryptovací nástroje dokonce odděleně kryptovali jednou a druhou část hesla, ta část, která se nevešla ani do jedné, byla nekryptovaná. Pokud budeme volit délku nad osm, stane se heslo zase o něco bezpečnější. U toho se musíme zastavit při volbě jednotlivé databáze.

Úžasnou obranou je měnit heslo, když máte pochybnosti. Nevyplatí se to dělat pravidelně, ale nahodile, třeba desetkrát do roka v jiný čas. Představme si, že někdo přišel na část našeho hesla, ale pak jsme mu to heslo změnili, umíme si pak asi představit, jak dlouho mu bude trvat, než přijde na to, že došlo ke změně.

I když je těžké zapamatovat si šílené heslo s šílenými znaky, je to někdy nejlepší řešení. Vymyslet heslo vás stojí maximálně deset minut života, ale vymýšlet výmluvu pro ztrátu dat a třeba i celých tabulek, je někdy velice těžké.

Ještě je nutné vysvětlit si, k čemu slouží příklad PASSWORD EXPIRE. Pokud budeme vymýšlet uživatele, který se přihlásí a bude s databází pracovat, můžeme někdy chtít, aby si hned po přihlášení změnil heslo. Důvodů pro tuto změnu je hned několik. Pokud vytváříme uživatele hromadně, můžeme jim nastavit stejné heslo, třeba žádné, a pak jim jen říct, že si heslo mají vymyslet. Nás by stálo hodně času vymýšlení nějakých hesel, navíc bychom museli za každým z uživatelů jít a říct mu heslo, ale takhle mu jej řekneme jednou a nemusíme se bát, že by to byla nadále naše starost.

Příkladem si můžeme uvést vytvoření uživatele pokus, který bude mít tak triviální heslo, že jej bude muset hned po přihlášení změnit:

```
CREATE USER pokus IDENTIFIED BY pokus PASSWORD EXPIRE;
```

Jenže co když nemá uživatel pokus ani možnost nějak heslo změnit při přihlášení, třeba používá nějakého vzdáleného klienta databáze a ten ani neumožňuje nějaké změny. Musíme tedy vytvořit uživatele, kterému heslo řekneme a nebudeme jej měnit.

```
CREATE USER pokus IDENTIFIED BY 'L_ja23-$$kiau$ůůř ř';
```

Uživatele můžeme i nějakým způsobem měnit. Někdy se stane, že prostě uživatelé heslo zapomenou, což je poměrně obvyklé. Jako administrátoři nepotřebujeme nějaké oprávnění pro změnu hesel uživatelů, to je také výhoda administrátorů.

Změnu provádíme jednoduše pomocí příkazu ALTER:

```
ALTER USER pokus IDENTIFIED BY nove_heslo;
```

Uživatel pokus má tedy heslo nove_heslo a už nás nebude dále potřebovat.

Nakonec můžeme uživatele odebírat, když už s databází pracovat nebudou. Když si představíme, že vybudujeme ohromný databázový systém, se kterým nám budou pomáhat nějací poradci, nebudeme pak chtít, aby chodili do naší databáze na návštěvu a hledali, co se jim zlíbí, tak je odebereme.

Příkaz pro odebrání je stejný, jak jsme zvyklí, DROP:

```
DROP USER pokus;
```

Teď už nás uživatel pokus nemusí trápit vůbec, protože v databázi už není zanesen.

Práva a kde je máme použít

Bez vytvoření práv je uživatel vlastně jen prázdný objekt, nemá nadefinovaná práva. Právy můžeme omezit oblasti, kterými se bude uživatel zabývat. Můžeme uživatele dokonce omezit na jednotlivé tabulky, či pohledy a nechat jej tam, kde jen chceme.

Představme si, že někomu chceme ukázat, jak má pracovat s vybíráním dat. Takový uživatel pak nepotřebuje funkce, jako jsou CREATE TABLE, ale jen SELECT, což se dá dobře omezit a nemusíme se bát, že by se v databázi něco špatného stalo.

Dotaz provádím příkazem GRANT a skladba je následovná:

```
GRANT právo1,právo2,právo3,... ON tabulka.či_pole TO uživatel;
```

Právem v tomto příkladě rozumíme třeba SELECT, ale omezit můžeme prakticky všechno. Při použití ON můžeme definovat jak tabulky, tak pole pomocí tečkové notace, kde před tečkou je tabulka a po tečce je pole. Pokud ON nepoužijeme, oprávnění platí na celou databázi. TO se používá pro uživatele, pro které platí oprávnění.

Odebírání je naprosto stejné, jen vyměníme GRANT za REVOKE. Skladba je naprosto stejná jako u GRANT.

```
REVOKE právo1,právo2,právo3,... ON tabulka.či_pole TO uživatel;
```

Příkladem přiřadíme uživateli pokus právo na SELECT a INSERT do tabulky, ale jen na jedno pole. Uživatel pak nebude moc schopný pracovat s ničím jiným, než s danou oblastí.

```
GRANT select, insert ON veze.vyska TO pokus;
```

Je to jen příklad, ale uživatel pokus by mohl jen vkládat a vybírat výšky z tabulky veze, což není moc příjemné, ale dostatečně omezující.

Je důležité zmínit, že databáze nám poskytuje volbu role, kterou můžeme aplikovat na více uživatelů. Je možné tak vytvořit jistá schémata pro nové uživatele, které budeme vytvářet a stanovit jim, co v databázi budou skutečně dělat a co ne. Je velice praktické si takové role předpřipravit a mít je vždy schované pro příležitost, že budou potřeba. Prakticky pak aplikujeme naše jednoduché zásady a tvoření uživatelů bude daleko rychlejší, než když budeme každé právo pomalu vymýšlet a snažit se vzpomenout, co

by asi uživatel mohl a nemohl potřebovat.

Roly vytváříme pomocí známého CREATE:

```
CREATE ROLE jméno_role;
GRANT právo1,právo2,právo3,... TO jméno_role;
```

Příkladem si můžeme vytvořit profil uživatele naší databáze, který bude mít na starosti jen záznamy z tabulek stromů:

```
CREATE ROLE uzivatel_strom;
GRANT select, delete, update, insert ON stromy TO uzivatel_strom;
```

Teď jen vytvoříme uživatele a přidělíme mu jednoduchou roli:

```
CREATE USER pokus IDENTIFIED BY kouzlo;
GRANT uzivatel_strom TO pokus;
```

Pravé kouzlo je v tom, že jsme ušetřili poměrně dost řádků, které bychom museli dopisovat v případě, že nemáme vytvořenou roli.

Cvičení

- Jak bychom vytvořili uživatele pokus a přidělili mu oprávnění pro práci s tabulkou letadel?
- Jak by vypadala role, která by se jmenovala uživatel a navíc by povolovala vybírání, vytváření tabulek a mazání tabulek v celé databázi?
- Jak by mělo vypadat heslo, které by mělo být bezpečné?
- Jak můžeme odebrat oprávnění?
- Jak můžeme odebrat uživatele?

Základy jazyka TCC

Jazyk TCC, tedy transaction control commands, je velice úzkou částí jazyka SQL, protože slouží pro vytvoření takzvaných transakcí. V základech si povíme o pár příkazech, které budeme potřebovat, ale musíme si vyjasnit, co vlastně jsou transakce a proč si zaslouží mít vlastní příkazy.

Také si uvědomme, že transakce nemůžeme běžně obsluhovat bez nějakého příkazového interpretru nad databází, ale to si ukážeme později.

Transakce jsou takové operace nad databází, které potřebují pro své dokončení správný průběh, jinak se okamžitě vrací do bodu, odkud začaly a nic se nevykoná. Prakticky to znamená, že pokud budeme pracovat s transakcí a ona se nepovede, nemusíme se bát, protože bychom měli udělat transakci takovou, že se vrátí do bodu, odkud začala, a my máme zase všechna data taková, jako před operací.

Prakticky však pracujeme s takovými příkazy častěji, než si uvědomujeme. Databáze totiž každý náš skript bere jako transakci a o jejím provedení rozhoduje prakticky sama o sobě. Spustí a vybere data, spustí a upraví data a tak dále bez toho, abychom o tom věděli, a přesto pracujeme s SQL a ani neděláme žádnou chybu.

Vysvětlíme si tedy první klauzuli, kterou budeme v TCC používat. Klauzule COMMIT znamená něco jako spouštěč, pokud tedy budeme s transakcí spokojení a budeme jí chtít provést, potvrdíme ji klauzulí COMMIT a vpíšeme ještě středník.

```
COMMIT;
```

Jenže jak to udělat, abychom transakci odvolali dříve, než ji spustíme. Příkladem, když jsme udělali někde chybu, či prostě nechceme transakci spustit. Pokud ale použijeme příkaz ROLLBACK, odvoláme všechny nepotvrzené transakce, to je nutné mít na paměti.

```
ROLLBACK;
```

Vytvoření bodu obnovy

Abychom ale mohli pracovat dobře s ROLLBACK, je nutné si nějakým způsobem oddělit transakce v jedné pracovní relaci. Můžeme tak vytvářet body obnovy a pak se jen vracet tam, odkud jsme vyšli. Je to velice výhodné, když nám prostě něco nevyjde a víme, kam se chceme vrátit.

Když budeme pracovat s body obnovy, budeme používat klauzuli SAVEPOINT a musíme si ji nějak vhodně nazvat, abychom neztratili souvislost. Skladba takového příkazu je jednoduchá:

```
SAVEPOINT jmeno_bodu_obnovy;
```

Navrácení k bodu obnovy

Navrácení je potom závislé na bodu, kam se chceme vrátit. Už víme, že pokud chceme zpět, můžeme použít ROLLBACK, ale když chceme zpět na nějaké určité místo, musíme si předtím vytvořit bod obnovy.

Klauzuli pro návrat určitého bodu vytvoříme velice jednoduše pomocí upraveného ROLLBACK, tedy:

```
ROLLBACK jmeno_bodu_obnovy;
```

A celá vykonaná práce se navrátí k jmeno_bodu_obnovy.

Příkladem si můžeme ukázat, jak celé to vše probíhá. Není to nic těžkého, proto jsme si ukázali jen jednoduché příklady, které můžeme někdy použít. Pamatujme na to, že nemusíme umět všechno, ale někdy se hodí i to, o čem jsme si říkali, že už to nikdy potřebovat nebudeme.

```
INSERT INTO tabulka (id, jmeno, prijmeni) VALUES (1, 'karel', 'novak');

UPDATE tabulka SET jmeno = 'josef' WHERE id = 1;

SAVEPOINT bod_jedna;

INSERT INTO tabulka (id, jmeno, prijmeni) VALUES (2, 'petr', 'Křivý');

UPDATE tabulka SET jmeno = 'Petr' WHERE id = 2;

ROLLBACK TO bod_jedna;
```

```
COMMIT;
```

Teď se musíme zastavit a přemýšlet nad tím, co jsme vlastně chtěli. Vidíme, že přidáváme do tabulky pana karla s příjmením novak a vzápětí jej měníme na jméno josef, ale hned potom jsme vytvořili bod obnovy a za ním jsme přidali další záznam a hned ho zase pozměnili a než jsme dali příkaz COMMIT, udělali jsme ROLLBACK, což sice vrátí druhý záznam a tedy jej nezanese do tabulky, ale nechá první přidání záznamu. Uvádíme tedy tento příkaz, abychom pochopili, že je nutné vědět, co kam a jak dáváme a neplést si, co děláme.

Cvičení

- Co jsou transakce?
- Jak vytvoříme bod obnovy a jak se k němu můžeme vrátit?
- Jak spustíme celou transakci?
- K čemu je přesně ROLLBACK?

Funkce

Už víme, jak vybírat data a jak z nich čerpat jen nějaká, která potřebujeme. Naučili jsme se data vázat na relaci, udělali jsme dokonce transakce, které řeší spolehlivost podaných dat, ale co když budeme chtít víc? Většinou každý jazyk podporuje práci s proměnnými, kde vybíráme průměry, maxima a minima. SQL obsahuje také takové funkce, pomocí jichž můžeme vybírat, jen co doopravdy potřebujeme. Jak bychom třeba v SQL vytvořili pomocí jednoduchého dotazu výběr průměru? K tomu je příkladem funkce AVG.

Zde si vysvětlíme základní funkce, které budeme potřebovat při agregačním výpisu, či jen když budeme potřebovat zjistit nějaká maxima, či minima. Nebudeme zabíhat do detailů, ale řekneme si, jak bychom měli k funkcím přistupovat.

Agregační funkce jsou tedy takové, které z dat tvoří jeden údaj. Příkladem je AVG a SUM.

Aritmetické funkce jsou pak takové, které pracují s jednotlivými záznamy a vytváří z nich pole. Příkladem je ABS a SIGN.

Některé funkce je možné pustit jen z tabulek, ORACLE má k takovým příležitostem tabulku DUAL, ale my si vytvoříme nějaké data do tabulky pacientů. Naše tabulka bude vypadat asi takto:

Pacienti

ID	jmeno	prijmeni	pohlavi	vek
1	Karel	Červěňák	muž	34
2	Martin	Obtáhlo	muž	76
3	Klára	Rudá	žena	87
4	Jaruška	Holá	žena	-34
5	Tomáš	Slabý	muž	78

Funkce

ABS je první funkcí, o které si povíme. Jak jste si všimli, máme mezi záznamy jeden, který je velice chybě zadaný. Prakticky je dost nemožné, aby někdo měl záporný věk. Pokud tedy budeme chtít výpis otočit, musíme použít funkci ABS, která znamená absolutní hodnotu.

V příkladu by vypadala asi takto:

```
SELECT jmeno,ABS(vek) FROM Pacienti WHERE vek < 0;
```

Ve výpisu se nám objeví: 'jmeno: Jaruška vek:34', kde jsme jasně stanovili, že chceme vybírat z tabulky Pacienti, kde vek je menší než 0. Funkce ABS nám otočila hodnotu na kladnou.

Než si ukážeme pět hlavních agregačních funkcí, připomeneme si, že existuje klauzule DISTINCT, která nám pomůže s vybíráním jedinečných záznamů v tabulce. V příkladu jsme si ji uváděli takto:

```
SELECT DISTINCT ID FROM Pacienti;
```

Kde po jemné úpravě může zapadnout i do agregační funkce:

```
SELECT AVG(DISTINCT ID) FROM Pacienti;
```

Zde se utvoří průměr z jedinečných čísel (ID je jedinečné pro celé pole, takže se vytvoří ze všech čísel v ID)

AVG slouží pro vypočítání průměru z dat. Prakticky ale potřebujeme ještě funkci ABS, protože naše tabulka má závadu. Funkce je agregační. Syntaxe by pak vypadalo takto:

```
SELECT AVG(ABS(vek)) FROM Pacienti;
```

Všimněte si, že jsme funkce použili najednou, což je velká výhoda. Ve výpisu uvidíme 61, což ukazuje, jak moc staré máme pacienty.

COUNT slouží pro výpočet počtu záznamů. Můžeme jej použít prakticky třeba přitom, když budeme chtít, kolik pacientů má naše malá tabulka. Funkce je agregační. Syntaxe bude následující:

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

Jak už víme, hvězdičkou můžeme vybrat vše, co je v tabulce, v tomto případě jsou to záznamy.

Nemusíme však počítat všechny. Když určíme, v jakém sloupci má funkce pracovat, můžeme počítat jen výběr a rozdíl bude znát, když bude v poli nějaký prvek nulový. Výpis z našeho dotazu by byl 5, tedy počet pacientů.

MIN je funkce pro minima z tabulky. Můžeme tak zjistit, jaké jsou nejmenší hodnoty z pole hodnot. Tato funkce nám může prozradit, kdo je nejmladší z naší tabulky. Funkce je agregační.

```
SELECT jmeno,MIN(ABS(vek)) FROM Pacienti;
```

V našem výpisu se ukáže nejmladší pacient jménem Karel.

MAX je opakem minima, tedy ve výpisu nám ukáže, kdo je nejstarší z našeho seznamu. Využití je úplně stejné jako u minima. Funkce je agregační.

```
SELECT jmeno,MAX(ABS(vek)) FROM Pacienti;
```

Výpis takového příkladu ukáže, že nejstarší je Klára.

SUM je funkce, která je už spíše matematická. Prakticky nám spočítá sumu všech záznamů, které zadáme do funkce jako pole. Suma se velice často používá, my teď zjistíme, jak moc jsou dohromady staří naši pacienti. Funkce je agregační.

```
SELECT SUM(ABS(vek)) FROM Pacienti;
```

Výpočtem zjistíme, že pacientům je v součtu 309 let, což je poměrně dost. Teď můžeme vidět, že průměr bychom museli vypočítat vydělením sumy počtem záznamů, což jsou dohromady tři funkce, když použijeme ABS. Z toho plyne snadné poučení, tedy používat funkce, které jsou potřeba, a ušetříme si práci.

Další funkce jsou spíše matematické, protože je použijeme možná jen při vytváření nějakých matematických databází. Pamatujme, že rychlejší je si matematický výpočet naprogramovat a ne jej chtít po databázi, protože ta pracuje s daty. Přesto si probereme aspoň základní z nich, které určitě potřebovat budeme.

FLOOR je funkce pro zaokrouhlení na celá čísla směrem dolů. Využijeme ji, když budeme pracovat třeba s databází let, kde budou rozepsány i desetiny roku. Málokdo se zajímá o desetiny a tak je můžeme prostě zaokrouhlit.

```
SELECT FLOOR(ABS(vek)) FROM Pacienti;
```

V naší tabulce ale nemáme žádná čísla s desetinami, takže čísla zůstanou stejná.

ROUND řeší zaokrouhlení pomocí matematických pravidel, což je poměrně dost důležité. ROUND dokonce umí zaokrouhlit podle toho, kolik mu určíme desetinných míst. Vše si ukážeme na příkladu.

```
SELECT ROUND(ABS(vek),2) FROM Pacienti;
```

V tomto případě se nic nestane, protože naše čísla nemají desetinná místa, ale jinak by se zaokrouhlila na setiny, tedy na dvě desetinná místa. Můžeme dokonce otočit výpis zcela opačně, tedy pomocí záporných hodnot v parametru funkce.

```
SELECT ROUND(ABS(vek),-1) FROM Pacienti;
```

Výpis by byl následující:

ROUND(ABS(vek),-1)

30

80

90

30

80

Zde jasně vidíme, že jsme jen zaokrouhlili na desítky. Můžeme dokonce zaokrouhlit o -2, což nám zvedne hladinu na stovky místo desítek.

TRUNC je pak funkcí, která čísla místo zaokrouhlení pouze odejme, takže když máme číslo 99,9999, je stále po oříznutí 99. Bohužel ale tato funkce není standardně ve všech databázích, ale určitě ji najdeme v databázi ORACLE.

SIGN je velice důležitá funkce, která nám pomůže stanovit, jestli je číslo kladné, záporné, či nulové.

Pro kladná čísla vypíše 1, pro záporná -1 a pro nulová nechá 0. V našem příkladě tedy konečně uvidíme změnu při záporném číslu.

```
SELECT SIGN(vek) FROM Pacienti;
```

SIGN(vek)

1
1
1
-1
1

Zde vidíme, jak případně můžeme hned zjistit, že je nějaký zápis záporný.

Další funkce jsou ryze matematické:

POWER slouží pro povýšení čísel dle jejich kvocientu. Tedy číslo POWER(2,5) je číslo dvě na pátou. V praxi to využijeme málokdy, ale přesto si ukážeme jednoduchý příklad.

```
SELECT POWER(ID,5) FROM Pacienti;
```

Tento dotaz vypíše ID v seznamu taková, která odpovídají své hodnotě na pátou. Velice zajímavé.

EXP pracuje s n-tou mocninou s přirozeným základem logaritmu *e.* Opačná funkce k této je funkce **LN**, ta pracuje s desítkovým základem.

SIN, COS, TAN slouží pro sinus, cosinus a tangens. Prakticky se ale používají jen málokdy nad databází s pacienty. Jejich syntaxe je naprosto stejná, jako u jiných funkcí. Pamatujme však, kdy je jaká z těchto funkcí definována dle matematických zásad.

V databázích vždy platí funkce pro sčítání a odečítání. Tedy:

-	odečet
+	součet
/	děleno
*	násobek

Rozdílem je, že pokud budeme chtít modulo, musím použít funkci MOD, tedy pro zjištění zbytku po dělení. Tedy by mohl příklad vypadat takto:

```
SELECT MOD(10,3);
```

Výsledek je jedna, protože 10 děleno 3 dá 3 a 1 zbude.

Dalšími funkcemi jsou funkce okolo dat. Databáze vychází velice vstříc uživatelům v podobě těchto funkcí, protože, když si představíme, kde všude databáze pracují, bez data není, jak by databáze pracovala. Prakticky každý nově přidaný záznam má dokonce své datové razítko. My jej nevidíme, ale databáze podle něj seřazuje data, pokud nezvolíme pomocí funkce jinak. Takže když přidáme do databáze záznam jedna a po něm záznam dva, jako první se zobrazí záznam jedna.

Prakticky si některé z těchto funkcí můžeme představit jako proměnné, které databáze vytváří pro čas. Uvedeme si také, jak s daty nakládat.

Následující funkce jsou spíše pro MySQL:

CURRENT_DATE je funkce vracející přesné datum v pořadí YYYY-MM-DD. Nevyžaduje žádné parametry, a pokud ji chceme nějak uříznout, musíme pracovat s jinou funkcí navíc, ale to si ještě ukážeme. Syntaxe je poměrně dost jednoduchá. Berme to tak, že CURRENT_DATE je proměnnou z databáze a nepotřebuje tabulku pro své vyvolání.

Z jiných databází můžeme vybrat třeba proměnnou SYSDATE, která je součástí databáze ORACLE.

```
SELECT CURRENT_DATE;
```

Výpis může vypadat následovně:

2010-04-05

CURRENT_TIME je podobný, jako CURRENT_DATE, ale dává nám výpis o aktuálním času. V

příkladě tedy:

SELECT CURRENT_TIME;

Výpis je rozdělen na rozdíl od data pomocí ':', tedy:

21:32:41

První jsou hodiny, za nimi minuty a pak sekundy.

DBTIMEZONE je poměrně dost důležitá funkce, protože nám vrátí posun oproti času GMT. Když pracujeme se vzdálenou databází, je nutné tuto funkci zakomponovat do našich dotazů, aby se uživatelé neděsili, že jejich data jsou stará o hodiny, když je právě vložili. Syntaxe je stejná jako u CURRENT_DATE, tedy:

SELECT DBTIMEZONE;

Pozor však na to, kde funkci použijeme, protože patří mezi typické ORACLE funkce. Snažme se tedy raději pracovat s databázemi blíže našemu serveru. Jinak si čas musíme posunout sami dle toho, kolik ukazuje CURRENT_DATE.

Jak jsme už říkali, můžeme osekat datum z funkce CURRENT_DATE čas, který nás zajímá. Ale co když to není třeba, protože můžeme použít funkci EXTRACT. Ta nám dovolí vybrat den, měsíc i rok z data, které zadáme, budeme používat dnešní datum:

SELECT EXTRACT(DAY FROM CURRENT_DATE);

Vrátí dvoumístný den z dnešního data.

SELECT EXTRACT(MONTH FROM CURRENT_DATE);

Vrátí dvoumístný měsíc z dnešního data.

SELECT EXTRACT(YEAR FROM CURRENT_DATE);

Vrátí čtyřmístný rok z dnešního data.

Extrahovat můžeme více věcí, než dny, měsíce a roky. Pomocí klauzulí HOUR, MINUTE a SECOND. V příkladě asi takto:

SELECT EXTRACT(SECOND FROM CURRENT_TIME);

Vrátí sekundy v dvoumístném provedení.

SELECT EXTRACT(MINUTE FROM CURRENT_TIME);

Vrátí minuty v dvoumístném provedení.

SELECT EXTRACT(HOUR FROM CURRENT_TIME);

Vrátí hodiny v dvoumístném provedení.

Výhodnou funkcí je **LAST_DAY**, který nám řekne, kolikátý je poslední den z měsíce. V případě, kdy pracujeme s nějakými složitými daty, je tato funkce poměrně dost důležitá. Syntaxe vypadá takto:

SELECT LAST_DAY(CURRENT_DATE);

Výpis je zase podobě, jako datum z CURRENT_DATE.

2010-04-30

Pokud chceme zase jen jedno datum, tedy třeba jen den, můžeme použít EXTRACT:

SELECT EXTRACT(DAY FROM LAST_DAY(CURRENT_DATE));

Vrátí pouze 30.

Pro SQL, kde si nejsme jisti, zdali funguje na databázi dotaz CURRENT_..., používáme funkci NOW(), která vrací přesné datum a čas, ty pak můžeme formátovat pomocí jednoduchého FORMAT. Uvedeme si jednoduchý příklad pro přepsání NOW() na datum.

SELECT NOW();

'2010-04-25 13:26:57'

Dotaz vrátil datum a teď budeme chtít pouze datum.

SELECT FORMAT(now(),'YYYY-MM-DD');

Jenže takový postup funguje jen někde. Pamatujme tedy, že bychom měli znát dobře naší databázi a podle toho s ní pracovat.

Už jsme prošli základní funkce pro práci s daty a datem, ale co když budeme potřebovat pracovat s textem v jednotlivých záznamech? SQL poskytuje velice podobné funkce jako každý trochu rozvinutý programovací jazyk. V základě jsou to TRIM pro odstřižení textu, či SUBSTRING, ale povíme si o více z nich.

LENGTH vrací ve výpisu velikost daného řetězce. Tato funkce se velice hodí pro výpočet celkových počtů symbolů s kombinací se SUM. Vše si ukážeme:

```
SELECT LENGTH(jmeno) FROM Pacienti;
```

Vrátí seznam s počty symbolů na každém řádku.

```
SELECT SUM(LENGTH(jmeno)) FROM Pacienti;
```

Vrátí sumu ze všech spočtených velikostí záznamů.

TRIM slouží pro ostřihání mezer před a za vybraným textem. Pokud máme tedy nějaký záznam, který má před a za textem mezery, TRIM je odstraní. Variací je LTRIM pro ořezání zleva a RTRIM pro osekání zprava. Syntaxe je takováto:

```
SELECT TRIM(*) FROM Pacienti;
```

Kde výpis bude vypadat jako seznam a všechny data budou ořezána o mezery před a za textem.

LOWER a **UPPER** jsou funkce pro změnu písmen na malá a velká ve výpisu. Funkce pak upraví seznam v poli na malá a velká, podle použití. Můžeme je použít tam, kde máme jistotu, že malá či velká písmena jsou nežádoucí a chceme sjednotit výpis na jednotný formát. Syntaxe je jednoduchá:

```
SELECT UPPER(jmeno) FROM Pacienti;
```

Vrátí seznam záznamů obsahující všechna jména zapsaná velkými písmeny.

```
SELECT LOWER(jmeno) FROM Pacienti;
```

Vrátí seznam záznamů obsahující všechna jména zapsaná malými písmeny.

SUBSTRING je funkce, která dovoluje vypsat jen část záznamu pomocí jednoduché syntaxe. Můžeme takto z velkého textu vzít, co zrovna potřebujeme. Syntaxe je jednoduchá:

```
SELECT SUBSTRING(řetězec,počátek_podřetězce,počet_znaků_podřetězce) FROM Pacienti;
```

Výpis by byl tedy seznam řetězců ze záznamů a obsahovat by jen části, které by byly součástí řádku.

LOCATE je funkce, která nám pomůže vyhledat výskyt hledaného řetězce. Udá ale pouze první z výskytů. Pokud chceme hledat dál, musíme použít podřetězec ze SUBSTRING. Použití je následující:

```
SELECT LOCATE("v","text v nějakém záznamu");
```

Výpis pak ukáže 6, protože „v“ je v řetězci na šestém místě.

REPLACE je funkce, která nahrazuje text podle zadaných parametrů. Je velice jednoduchá a dá se kombinovat. Prakticky pak data můžeme upravit podle toho, jak potřebujeme, a nemusíme se bát, že by změnila originál. Syntaxe je jednoduchá:

```
SELECT REPLACE("text v nějakém záznamu","text","řetězec");
```

Výpis je pak „řetězec v nějakém záznamu“. REPLACE jednoduše vyhledá všechny možné výskyty slova text a nahradí je za řetězec.

CAST je funkce, která nám pomůže převést záznam na jiný datový typ. Prakticky jej můžeme využít třeba i na ořezání výpisu, protože víme, že jmeno je text, vezmeme z něj jen první tři písmena a uděláme to jen změnou datového typu. Nemusíme se bát, že něco pokazíme napořád, CAST mění jen výpis, originál nechává, dokud k tomu funkci neupravíme.

```
SELECT CAST(jmeno as CHAR(3)) FROM Pacienti;
```

Výpis vypadá pak takto:

CAST(jmeno as CHAR(3))

Kar

Mar

Klá

Jar

Tom

Prakticky můžeme data měnit na jakékoliv datové typy, ale nelze převádět text na číslo a tak dále. Dokonce ani při tomto ořezávání nepracujeme korektně, protože je to vlastně chyba na naší straně. CAST se spíše používá při převodu čísla na řetězec.

SELECT CAST(číslo as CHAR(3));

Tento dotaz by pak převáděl číslo na CHAR s velikostí 3, ale kdyby bylo číslo větší, než parametr, číslo by usekl.

Řídící struktury

Jak už jsme si říkali, SQL není založeno na práci s proměnnými v pravém slova smyslu, dokonce má úplně jinou skladbu, než ostatní jazyky, ale přesto poskytuje jisté možnosti pro práci s větvením jednotlivých částí skriptu. Můžeme rozhodovat o tom, kdy se jaká operace provede a kdy ne.

IF je základní podmínkovou funkcí. Jeho složení poměrně dost zásadní a jeho pochopení nám dá velikou možnost v práci nad databází. Můžeme tak vytvořit strukturu v provádění jednotlivých dotazů na databázi. Složení je vždy takovéto: IF (když) „podmínka je platná“ potom proveď „funkce“ jinak proveď „jiná funkce“.

IF je ale jednou z funkcí, která je spíše součástí PL/SQL a rozhodně bychom ji neměli používat nějak v základní části, přesto nám může trochu pomoci a naznačit, že SQL může obsahovat i IF.

Syntaxe je příkladná:

SELECT IF(podmínka,splnění_podmínky,nesplnění_podmínky) FROM tabulka;

Příkladem bychom si mohli uvést akci na naší databázi, kde budeme chtít, aby se vybralo první jméno, když průměr všech našich záznamů ve věku bude větší, než 90.

SELECT IF (AVG(vek)>90,jmeno,"ne") FROM Pacienti;

Pokud průměr nebude v IF správně, tedy dá nepravdu, vypíše se „ne“. Už zde vidíme, že pokud chceme pracovat s podmínkou IF, musíme s ní vyhodnocovat správná data, která mají být vyhodnocena, protože skript nepokračuje a nevypíše dál seznam, ale jen hodnotu na daném řádku. To nás vede k používání delších poddotazů a ty jsou poměrně pokročilé.

V základu bychom mohli vytvořit pomocí IF jednoduchý skript vypisující, jestli je naše databáze plná starců, či mladých.

SELECT IF (AVG(vek)<70,“Pacienti jsou mladí“,"Pacienti jsou pěkně staří") FROM Pacienti;

K tomu je IF v SQL poměrně dost dobře použitelné, ale jinak se v normálním SQL moc nepoužívá, takže nebuďme zneklidnění.

CASE je oproti IF více členěné. Povoluje pracovat s více možnostmi v podmínce. Prakticky jej ale zase musíme použít v nějakém rozšířeném dotazu. Složení funkce je následující:

SELECT CASE hodnota

WHEN podmínka THEN výsledek_souhlasí ELSE výsledek_nesouhlasí;

CASE pracuje s porovnáním více hodnot takto:

SELECT

CASE hodnota

WHEN podmínka1 THEN výsledek_souhlasí1

WHEN podmínka2 THEN výsledek_souhlasí2

WHEN podmínka3 THEN výsledek_souhlasí3

ELSE výsledek_nesouhlasí;

Pokud jste přímo nepochopili, k čemu použít IF a CASE nevadí, protože s SQL spíše pracujeme s WHERE podmínkou, či HAVING a nepotřebujeme tyto rozšiřující funkce. Je spíše lepší vědět, že existují, a když je jich potřeba, poradit se o jejich využití. Při zpracovávání CASE a IF mají databáze poměrně dost práce a samotné SQL k tomu ani není nějak uzpůsobeno.

Cvičení

- Jakým způsobem zaokrouhlujeme v SQL?
- Jak můžeme vybrat průměr záznamů?
- Co je funkce ABS a co dělá?
- K čemu bychom také mohli použít funkci UPPER?
- Proč a kdy použít funkci CAST?
- Vymyslete složení výběru dat z pole, kde chceme z textu „Našel jsem slona“ určit místo druhého výskytu písmena „s“ a nahradit slovo „slon“ slovem „elefant“.
- Jak byste postupovali, kdybyste měli vymyslet výpis, kde pro větší průměr než 100 vypíše dotaz „velké číslo“ a pro menší než 100 vypíše dotaz „malé číslo“ (možností, jak vymyslet odpověď, je víc).

Poddotazy

Pokud vytváříme někdy dotazy, které jsou rozsáhlé, je nutné si pomoci poddotazy, které za nás vyberou data pro podmínku, a pak spolu s nimi složíme výběr, který bychom jinak vytvořit nemohli. Po probrání poddotazů budeme schopni pracovat s výběrem dat, které bychom jinak ani nemohli použít.

Poddotaz

Poddotaz je první z poddotazů, který si vysvětlíme. Prakticky je to nejjednodušší forma, kterou použijeme nejčastěji, bez toho, abychom byli nuceni moc přemýšlet nad obsahem. Příkladem pokud potřebujeme dosadit nějaké data do výběru, který má podmínku na data z poddotazu.

Jako příklad použijeme dotaz, kde budeme chtít ID z jiné tabulky a budeme jej vybírat pomocí jména.

```
SELECT *
FROM Pacienti
WHERE id_nemocnice = ( SELECT id
                       FROM nemocnice
                       WHERE jmeno = 'Bulovka' );
```

Právě jsme vybrali všechny pacienty, kteří jsou v nemocnici Bulovka.

Korelovaný poddotaz

Korelovaný poddotaz je takový, kde se vyskytuje odkaz na tabulky, které jsou umístěny ve vnějším výběru z databáze. Pokud si uvědomíme, jak takový dotaz vypadá, jen budeme používat navíc vložený odkaz pomocí zkratky za jménem tabulky.

```
SELECT *
FROM Pacienti p
WHERE vaha = (SELECT max(vaha) FROM Pacienti WHERE id = p.id)
```

Prakticky navazujeme vztah na dotaz z venku, a proto se vnitřek vnořeného dotazu bude vyhodnocovat stále znovu pro každou řádku na rozdíl od poddotazu normálního.

Derivovaný poddotaz

S derivovanými dotazy se setkáme při vložení poddotazu za klauzuli FROM, kde ji použijeme jako část spojení pro JOIN a můžeme s ní pracovat jako s částí relace.

```
SELECT p.jmeno, p.prijmeni, vaha, n.max_vaha, n.min_vaha
    FROM Pacienti p
    JOIN (
        SELECT id, max(vaha) AS max_vaha, min(vaha) AS min_vaha
        FROM Pacienti GROUP BY id) n
    ON p.id = n.id;
```

Právě jsme tedy vytvořili výpis, kde se nám zobrazí pro každého pacienta jak jeho jméno a příjmení s váhou, tak i maximální a minimální váha.

V základě je nutné pochopit jednoduchý poddotaz, který je také nejúčinější a pro uživatele nejlehčeji pochopitelný. Pokud ale přesně nechápeme, jak na to, je jednoduší se poddotazům vyhnout anebo si je nastudovat pořádně

Použití pro práci v prostředí kanceláře

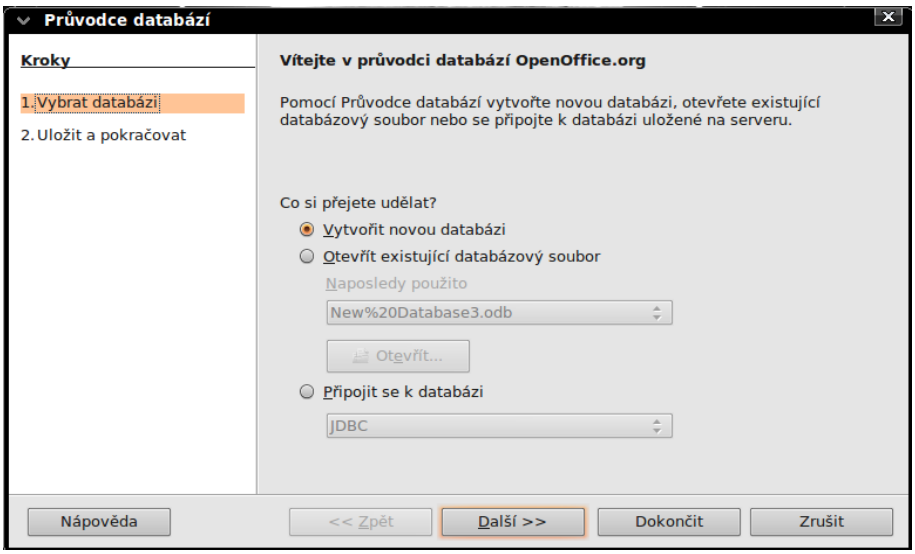
Potřeba malých databází, které by byly přenosné na malých mobilních zařízení, je v poslední době velikou výhodou moderních kancelářských programových balíčků. Jak jsme si již řekli, databáze stojí za spoustou věcí, tedy za daty, které máme na webových aplikacích, či za programy, které mají u sebe malou databázi v podobě souboru a berou z ní data. Kancelářské balíčky pracují právě se soubory jako s databázemi.

Protože jsou kancelářské programy udělány i pro uživatele, kteří nejsou moc obeznámeni s prací s SQL, spousta kancelářských programů SQL zcela vynechává a nahrazuje pomocí jiných nástrojů, jako jsou tvůrci dotazů, či průvodci. Prakticky více než polovinu práce můžeme udělat pomocí průvodců. Zaměříme se ale hlavně na části, kde budeme používat SQL.

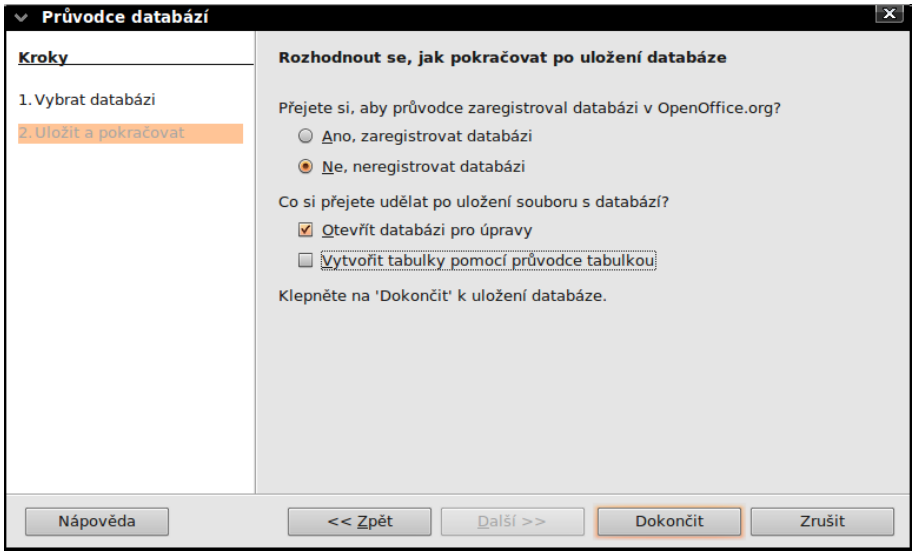
OpenOffice

Když budeme chtít používat databázový program od OpenOffice, musíme si jej stáhnout ze stránky http://www.openoffice.org/, kde najdeme http://download.openoffice.org/index.html a tam stáhneme nejnovější verzi OpenOffice. Po instalaci si ověříme, že máme i databázový software, ve kterém budeme pouštět naše databáze jako soubory.

Při spuštění nás ve verzi 3.1.1 přivítá průvodce vytvořením databáze, který nám pomůže vytvořit základní nastavení. Necháme vybranou položku „Vytvořit novou databázi“.



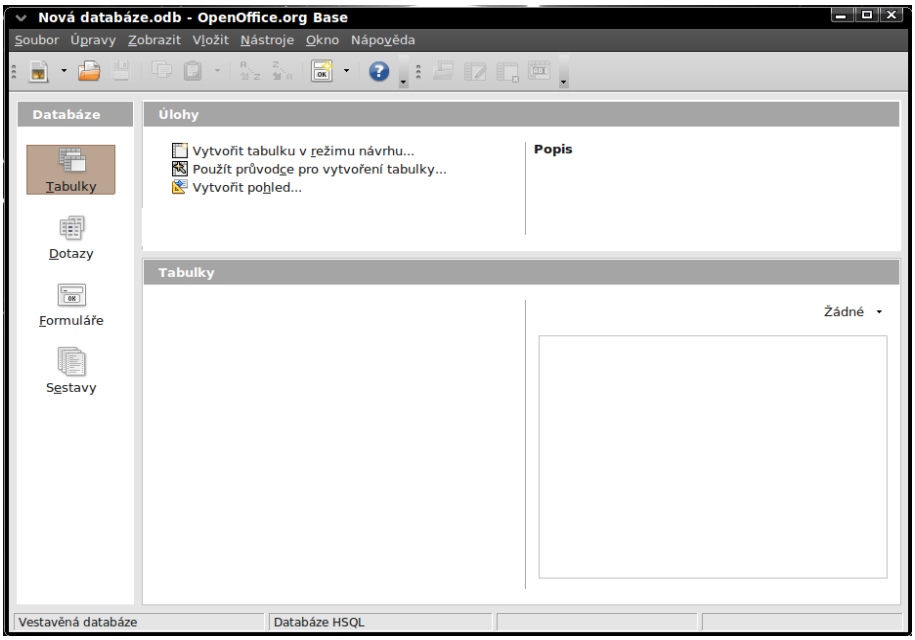
Pak nás průvodce nechá vybrat, jestli necháme zaregistrovat naší databázi, či nikoliv. Pak vybereme možnost, která je nám nejvíce příhodná a zvolíme dokončit.



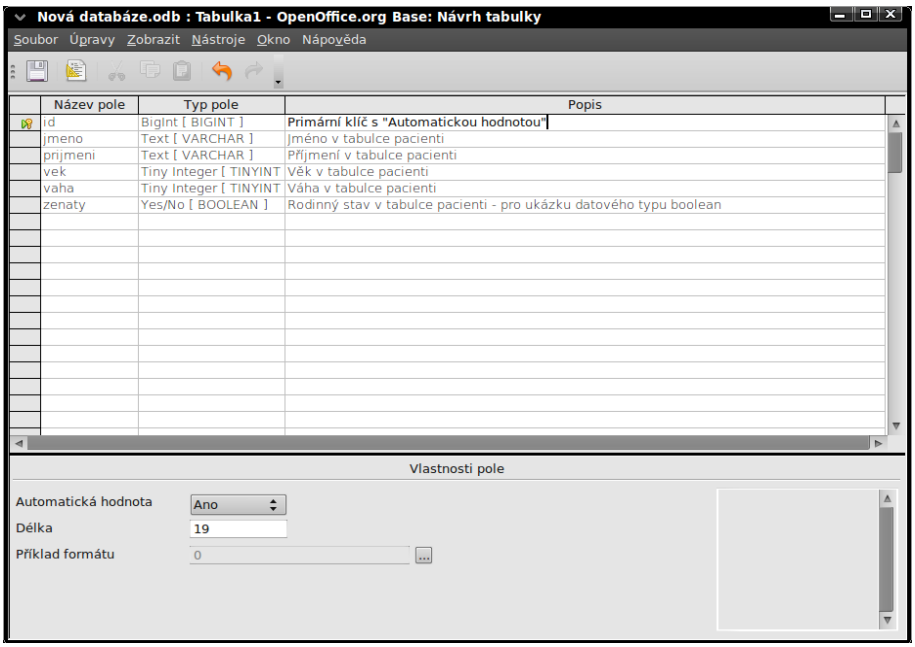
Následně se nás náš systém zeptá, kam uložit databázi a my vybereme, kam potřebujeme.

v základně není potřeba moc vybírat, ale pamatujme, že když budeme do databáze přidávat stále a stále nová data, bude databáze narůstat a my budeme nuceni databázi přemístit, takže pamatujme, jak bude databáze veliká, když ji začneme používat.

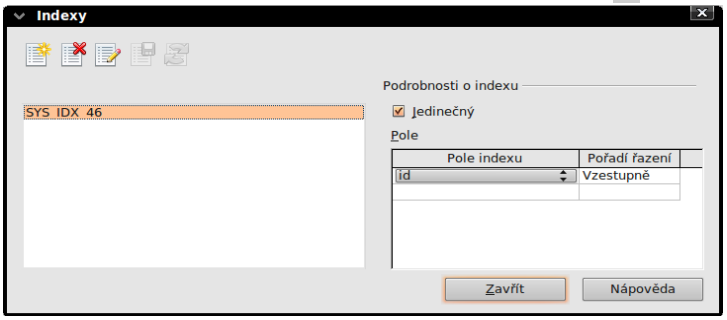
Pak už vidíme poměrně jednoduché rozhraní pro vytváření tabulek, dotazů a zvláštních nástrojů, které jsou potřeba pro práci uživatele v databázi, jako jsou formuláře a sestavy.



Pokud zvolíme vytváření tabulek pomocí režimu návrhu, začneme pracovat s tabulkou a jejím návrhem. Jak vidíme, OpenOffice nám vychází vstříc a nechává pro zvolení poměrně velký počet datových typů, kterými optimalizujeme naší databázi. Můžeme zvolit pro klíč i automatickou hodnotu. Volbu klíčového sloupce zvolíme kliknutím pravého tlačítka myši na pole a výběrem. Délku pole omezujeme pomocí délky, kterou vyplňujeme ve vlastnostech.



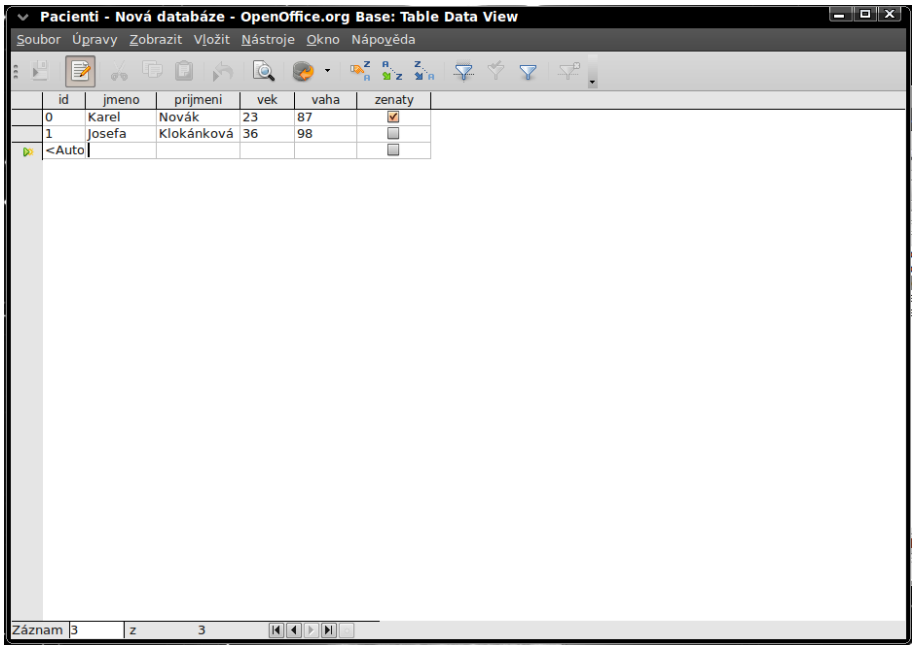
Při kliknutí na uložení se nás program zeptá, jak se má tabulka jmenovat. Nazveme ji Pacienti.



Můžeme rovnou vybrat indexování v druhé položce, znázorněnou ikonou:

Pak zvolíme v položce okno „zavřít okno“ a tím se vrátíme k práci s naší databází.

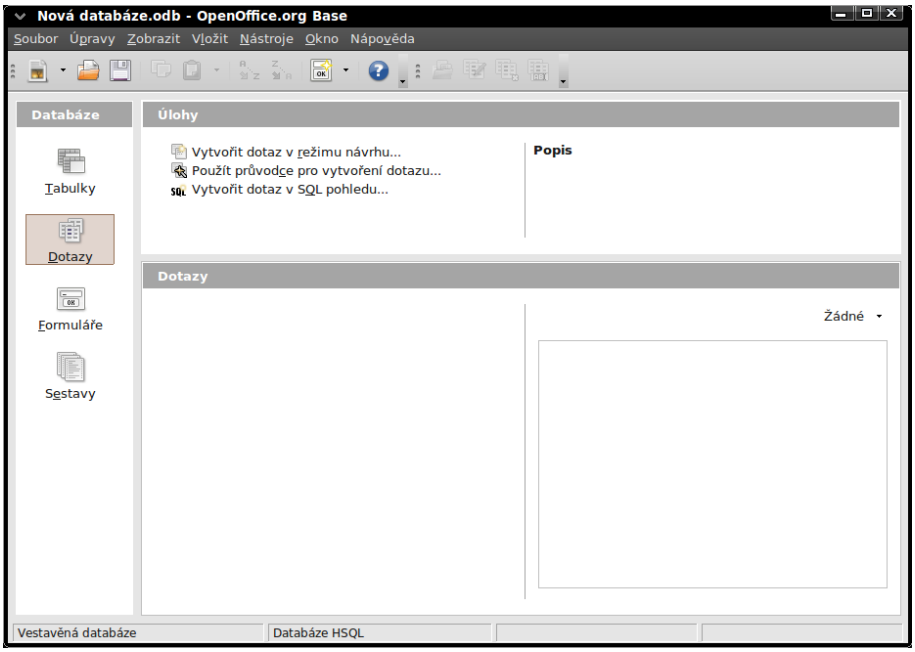
Po poklikání na naší tabulku v nabídce se dostaneme do další tabulky, kde můžeme vkládat nové



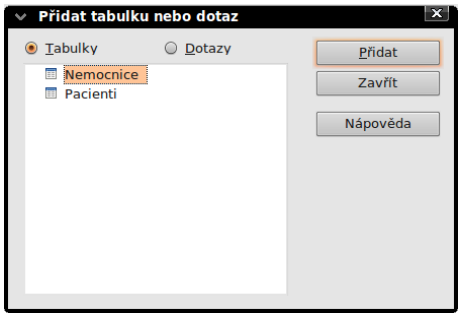
záznamy. Tím můžeme vyzkoušet, jestli vše funguje, jak má.

Potom zase zavřeme okno v položce okno a pracujeme dále s databází.

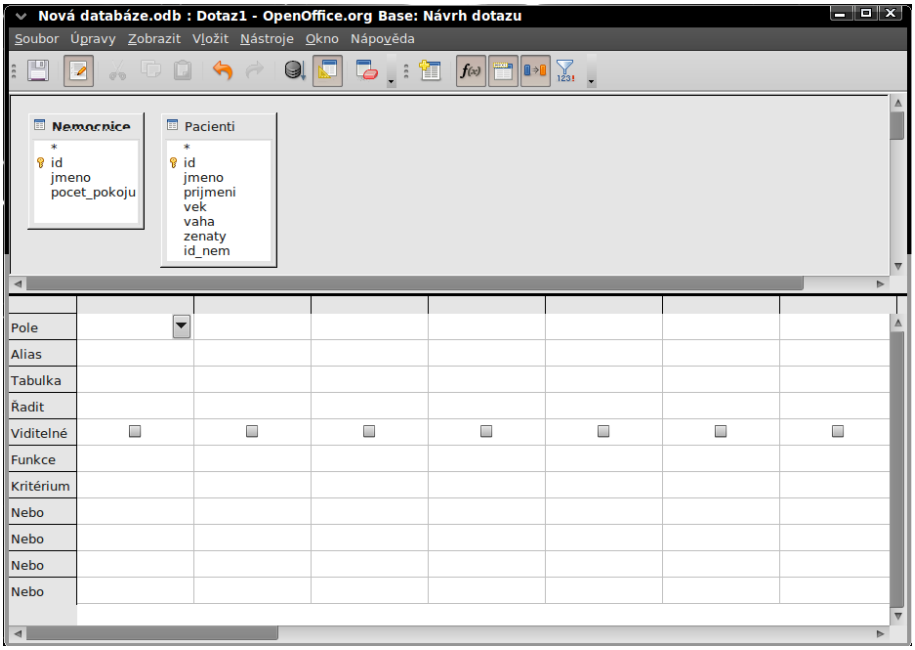
V položce dotazů si můžeme vybrat práci pomocí režimu návrhu, či pomocí průvodce a pomocí SQL, kde SQL je volba pro nás ta nejrychlejší. Podíváme se ale i na volbu režimu návrhu.



Při zvolení režimu návrhu se nás program zeptá, s jakými tabulkami budeme pracovat. Přidáme i tabulku Nemocnic, která už je vytvořená a ukážeme si na ní naznačení vazby.

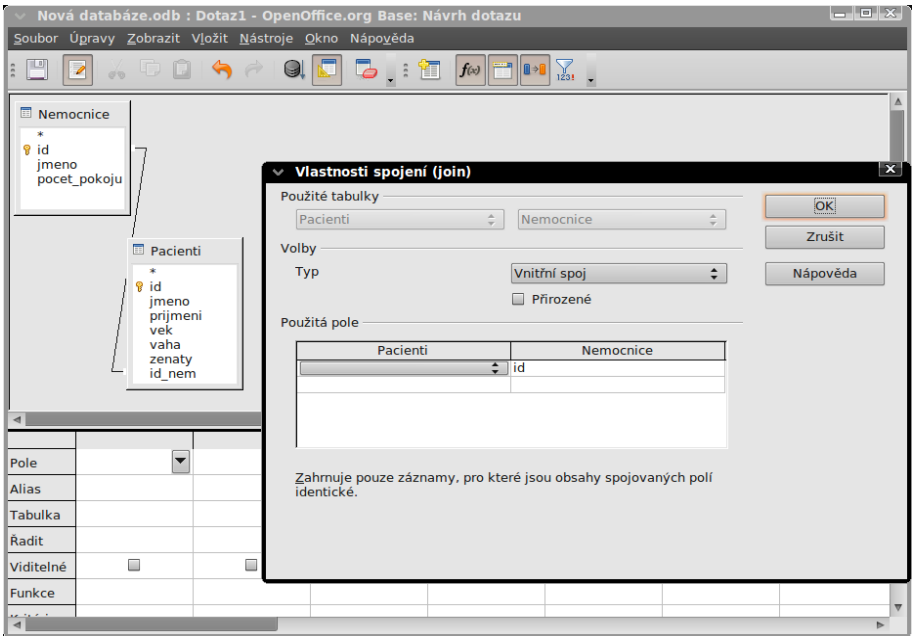


Následně se nám ukážou naše dvě tabulky, které používáme pro vytvoření dotazu.



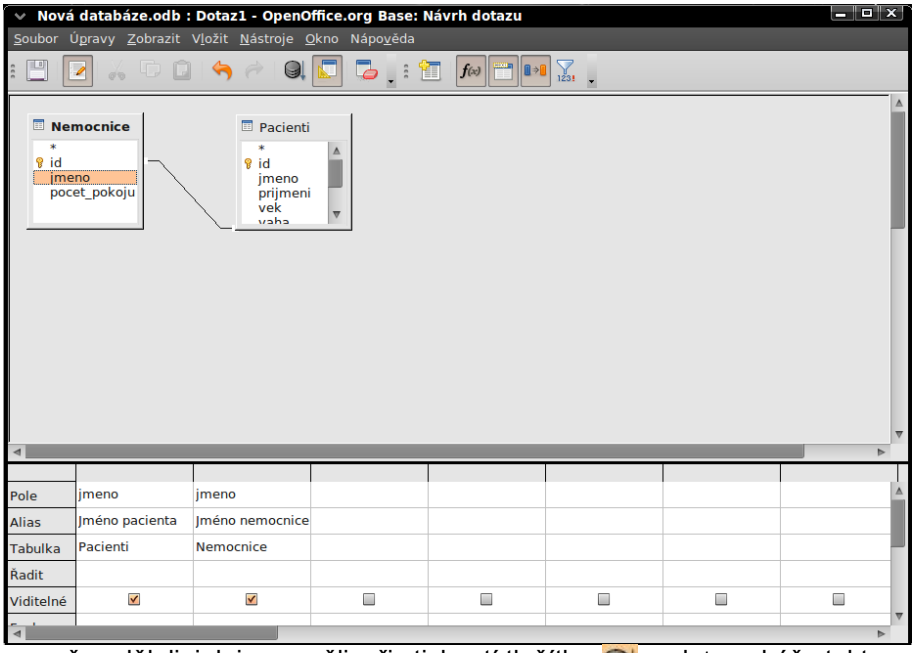
Následnou vazbu znázorníme pomocí tažení myši s přidrženým polem klíče na pole klíče cizího, čímž se vytvoří vazba jedna ku mnoha a my víme, že jedna nemocnice může mít více pacientů.

Všimněme si, že vlastnosti spojení si můžeme volit při kliknutí na čáru vazby pravým tlačítkem a zvolení možnosti „upravit“.

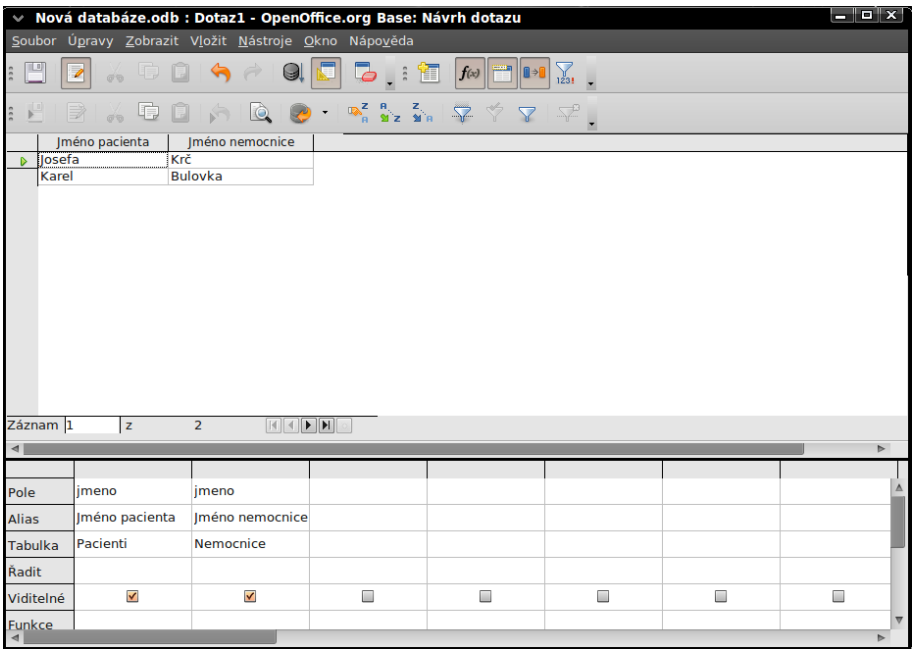


Pak jen vložíme jména polí, které chceme použít, v našem případě jsme ale předtím museli vložit záznam do tabulky pacientů a našim dvěma pacientům přidat do cizího klíče položku pro číslo nemocnice a potom ještě předtím vytvořit samotné záznamy pro nemocnice. Potom jsme schopni vytvořit třeba takovýto

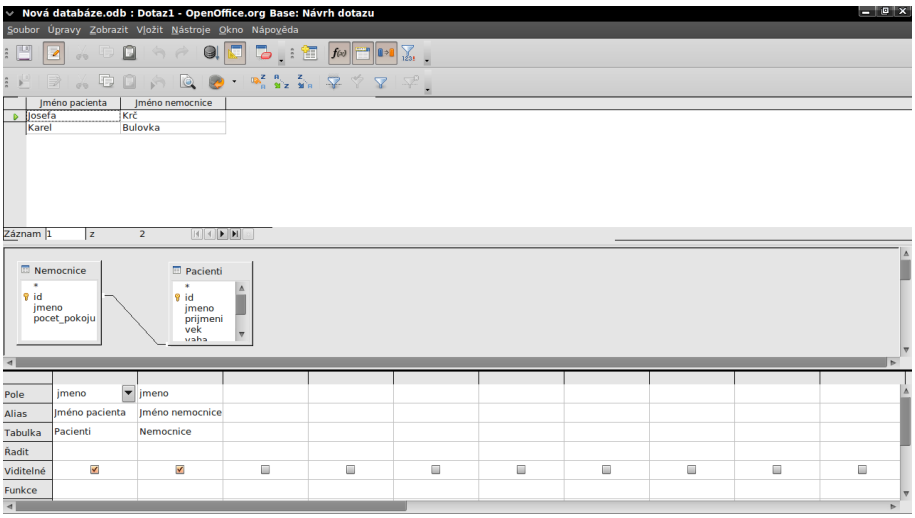
dotaz.



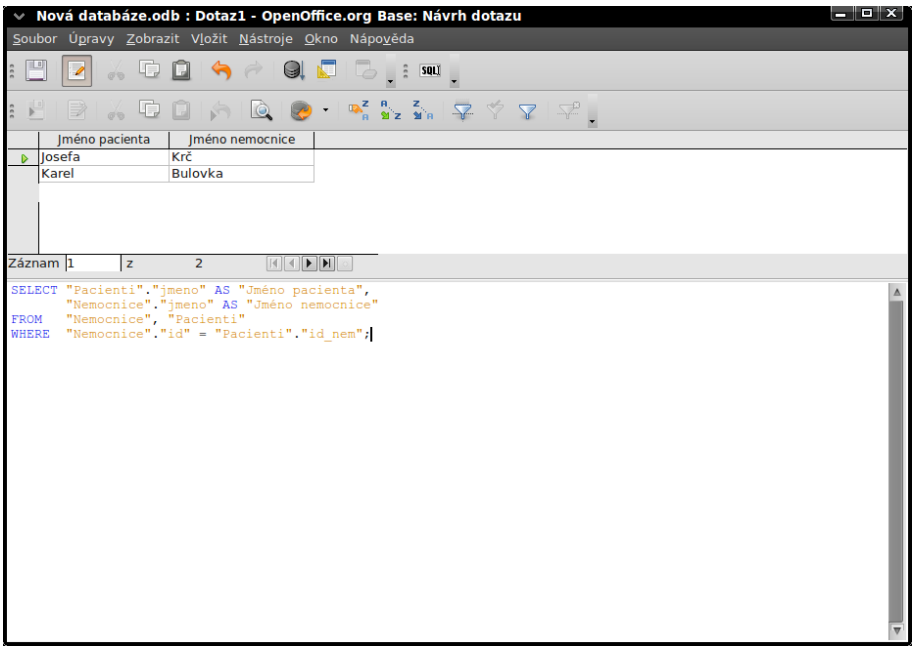
Pokud jsme vše udělali, jak jsme měli, při stisknutí tlačítka  se dotaz ukáže takto:



Pokud budeme chtít udělat náhled na náš dotaz pomocí SQL, můžeme použít ikonu pro vypnutí  náhledu . Dokonce, když použijeme větší zobrazení a roztáhneme pole pro záznamy a pole pro úpravy, vidíme i pole pro vytváření dotazů a potom dotazy znovu upravovat.

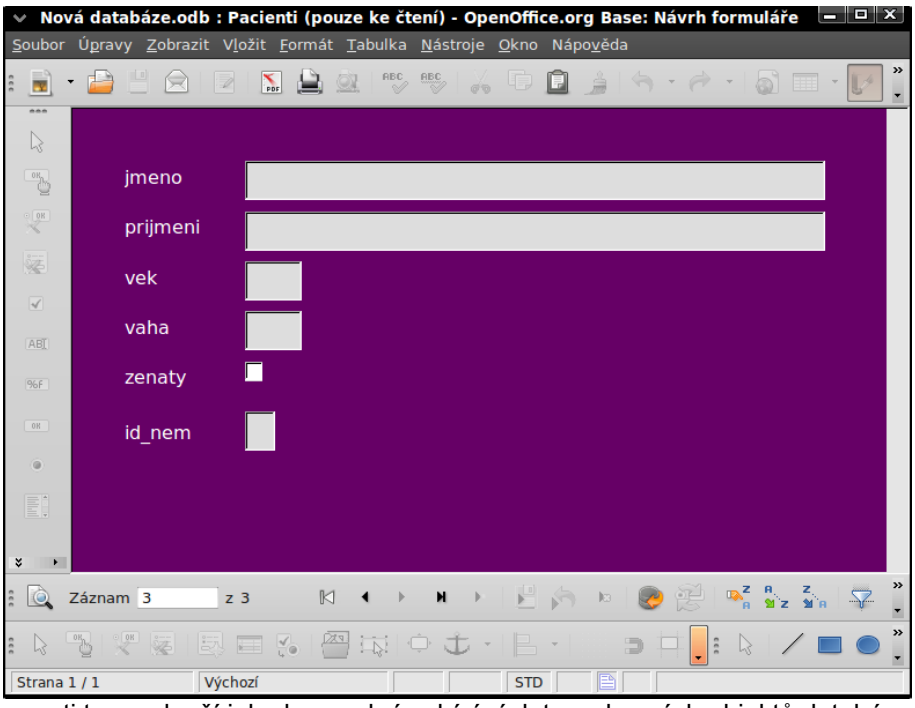


Vrátíme se teď do databáze a zkusíme vše udělat pomocí SQL. Po napsání SQL skriptu zase vše spustíme stejnou ikonou, jakou v minulém případě.  Jak vidíme, zvolili jsme přirozený spoj pomocí WHERE, ale můžeme použít i JOIN.

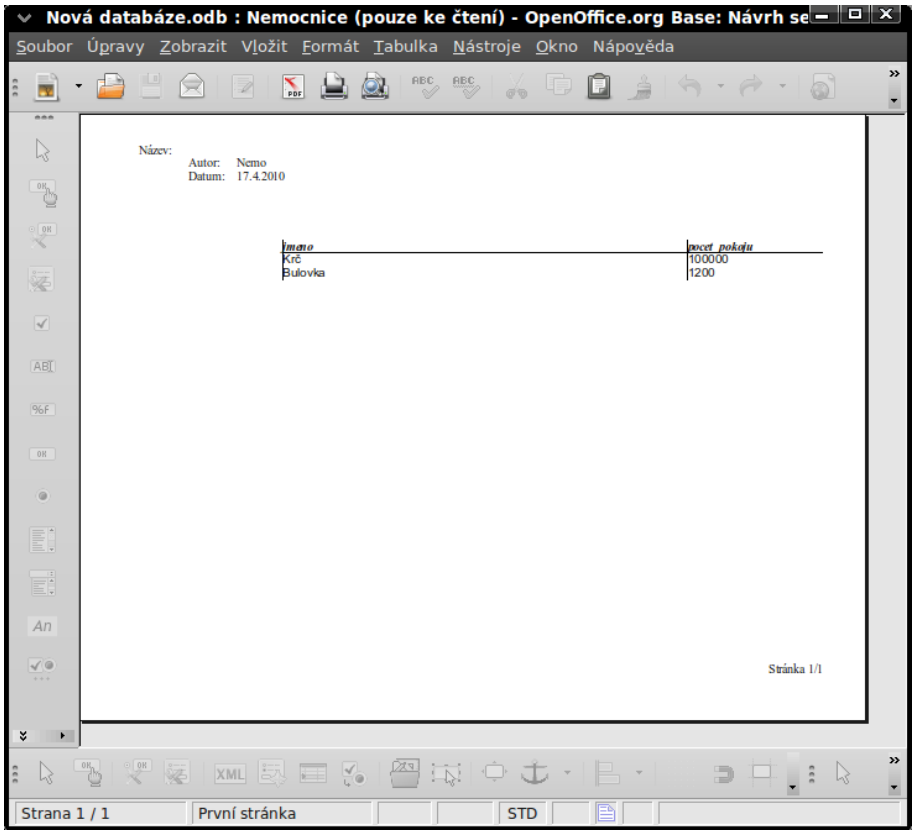


Takto můžeme pracovat s databázemi pomocí otevřeného databázového programu OpenOffice. Další dvě možnosti se týkají formulářů a sestav. Formuláře spíše slouží pro nějaké rozumné řešení přístupu uživatelů k datům, aby je vkládali v nějakém rozumném formátu. Nemůžeme chtít, aby si normální uživatel otevřel tabulku a vkládal údaje, a proto jsou tu formuláře pro svou přívětivost. Prakticky tak můžeme vytvořit jistý obraz naší databáze, ale to už není součást SQL. Berme to jako nástavbu.

Pokud budeme pracovat s formuláři, používejme průvodce, který nás provede jednoduše vším, co potřebujeme. Pro ukázku si předvedeme jeden formulář, který už je pomocí průvodce vytvořený.



Sestavy oproti tomu slouží jako hromadné vybírání dat z vybraných objektů databáze. Můžeme tak zobrazit všechny údaje, které potřebujeme, a není potřeba pracovat přímo s dotazy. Je to spíše takové ulehčení práce a zase slouží pro jistou prezentaci pro koncové uživatele.



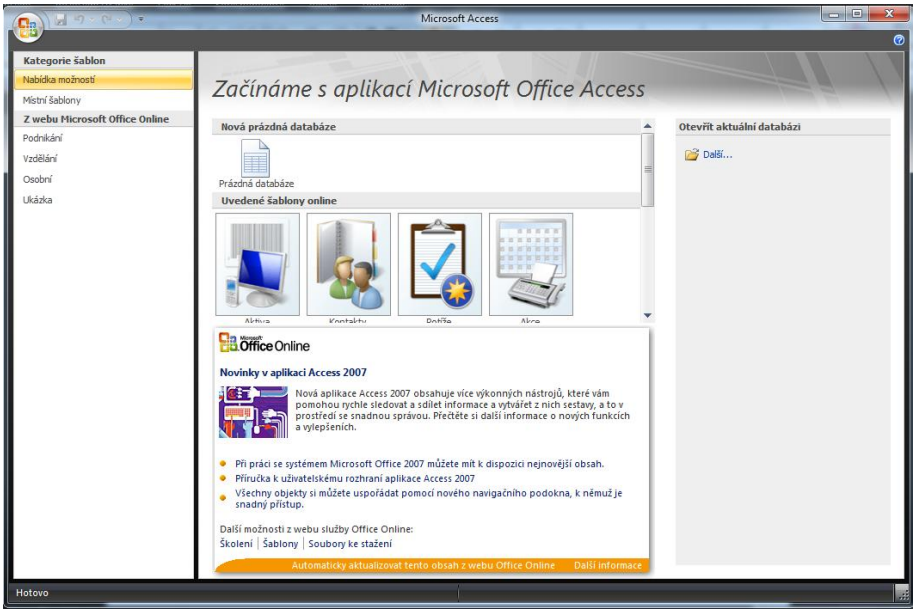
Microsoft Office

Microsoft přišel se jednoduchým a velice přívětivým prostředím pro koncové uživatele, jak pracovat s databází. Vše nám pomůže vypracovat koncept naší databáze a na vše jsou průvodce, ale pro databázi v normálním formátu je velice složité nějakým způsobem pojmout celou kapacitu Microsoft Access. Access sám o sobě umožňuje propojení na datová místa Microsoft programů. OpenOffice oproti tomu umožňuje připojení přímo na databázi na serveru od Mysql, až po ORACLE.

Pokud zůstaneme u stejných základů, ukážeme si jak na tabulky a dotazy, ale pamatujme, že princip

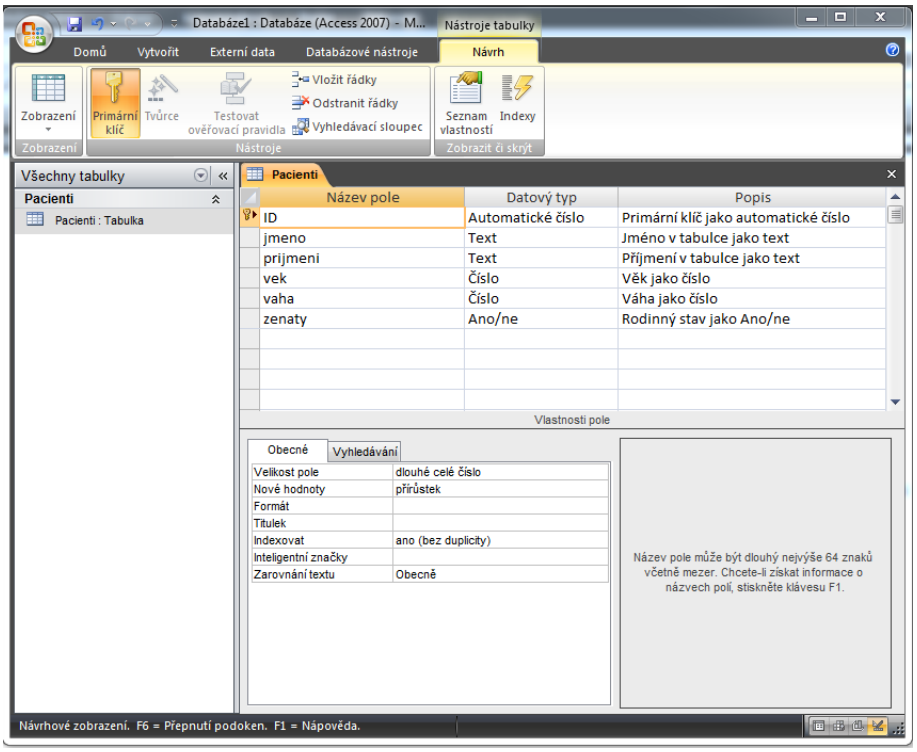
je stále stejný. Mysleme také na to, že Microsoft Access je komerční, a proto si musíme program zaplatit. Navíc Access povoluje jisté věci, které jsou vykročením ze standardního SQL, jako jsou kontingenční tabulky, či zápis jiné syntaxe pro práci s daty.

Takto vypadá úvodní obrazovka databáze pro Microsoft Office Access 2007. Můžeme si zvolit už vytvořené databáze, ale my zvolíme Prázdnou databázi. Necháme ji vytvořit podle zásad, o kterých jsme mluvili a začneme pracovat.

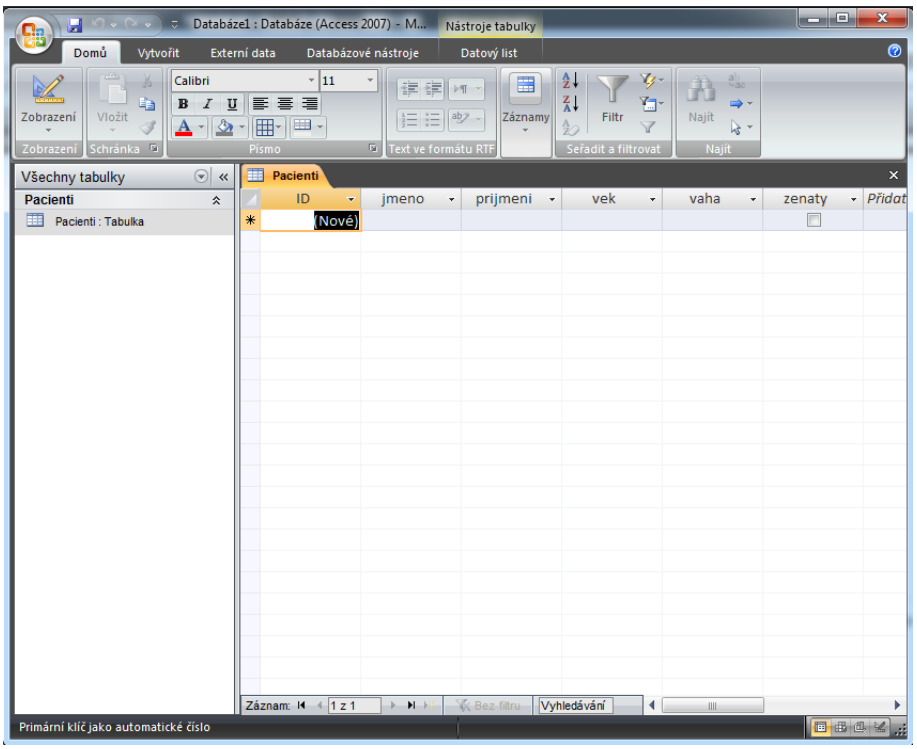


Databázový program nás uvede do vkládání nových dat do tabulky, ale my klikneme vlevo na Tabulka1 a pravým tlačítkem myši vybereme Návrhové zobrazení. To se velice podobá návrhu v OpenOfficu. Všimněme si, že chybí spousta datových typů, které jsou nahrazeny zjednodušenou formou v podobě text, číslo, či automatické číslo a datum.

Zde ve vlastnostech můžeme nastavit i indexaci, takže není potřeba žádná další vyplňující tabulka.

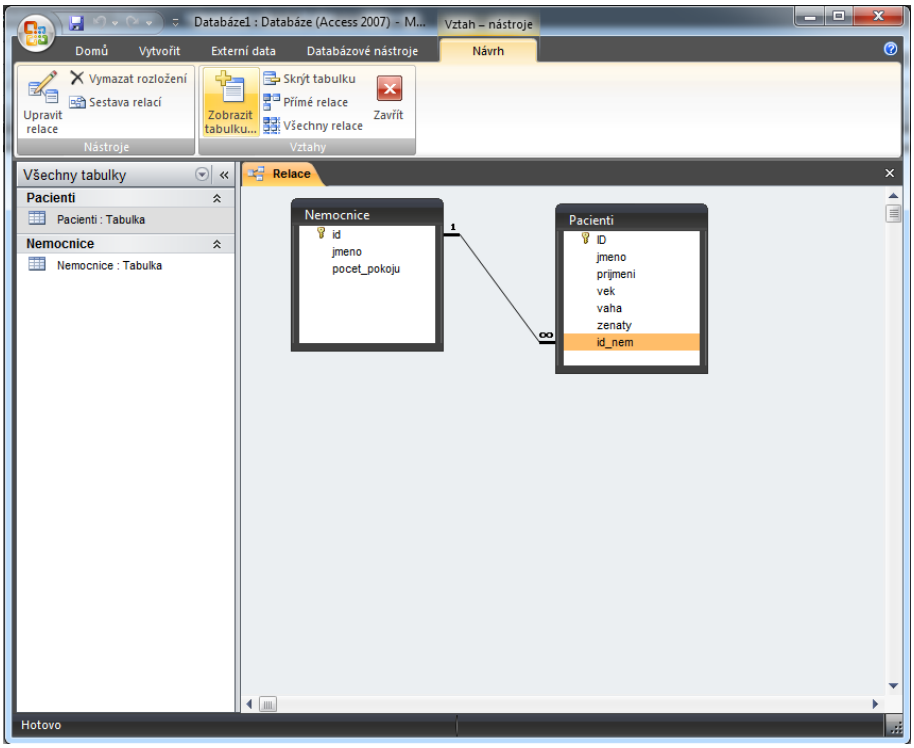


Po zavření tabulky vpravo a jejím uložení, můžeme do ní hned vkládat nová data a pracovat s nimi. Práce je pak usnadněna když hned vidíme, kde zrovna pracujeme, protože seznam objektů databáze máme hned vlevo od našeho pracovního prostoru.

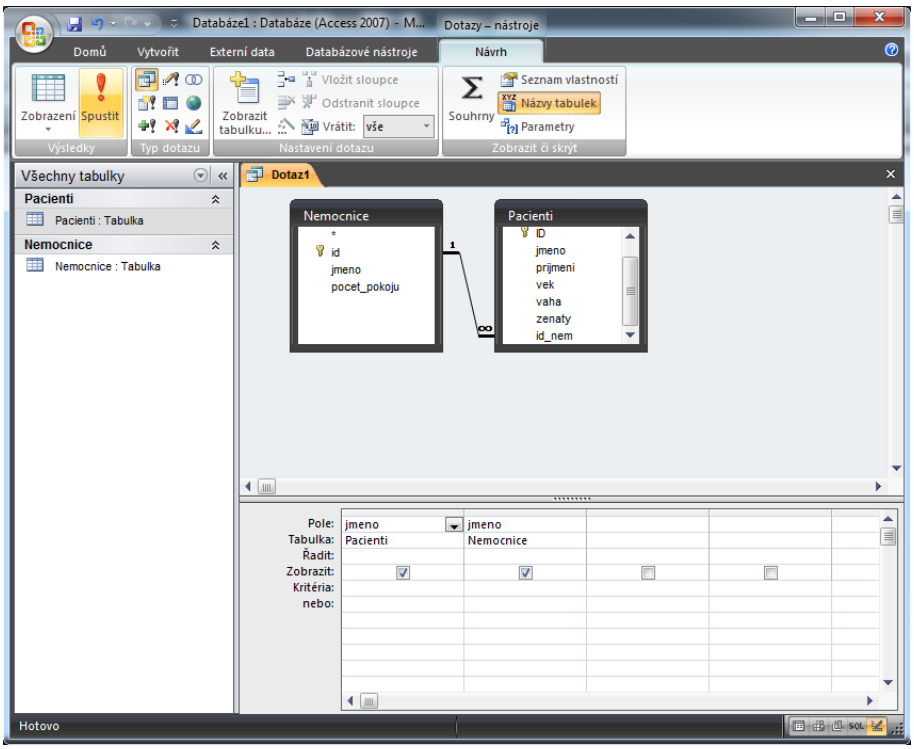


Vytvoříme si ještě jednu tabulku a do tabulky pacienti přidáme náš cizí klíč pro nemocnici. Editace proběhne stejně, jako jsme tabulku vytvářeli. Tabulky se jinak vytváří v položce „Vytvořit“ a „Návrh tabulky“. Následně přejdeme do položky „Databázové nástroje“ a zvolíme položku „Vztahy“, nejedná se o nic jiného, než o relace mezi tabulkami, které budeme potřebovat pro práci s dotazy.

Relaci vytvoříme tažením cizího klíče na klíč jedinečný, tedy jedna nemocnice připadne na více pacientů. Program Access se nás zeptá tabulkou na jisté detaily a jediné, co musíme je zaškrtnout políčko pro „Zajištění integrity“, relace se nám pak vykreslí dle vazby.

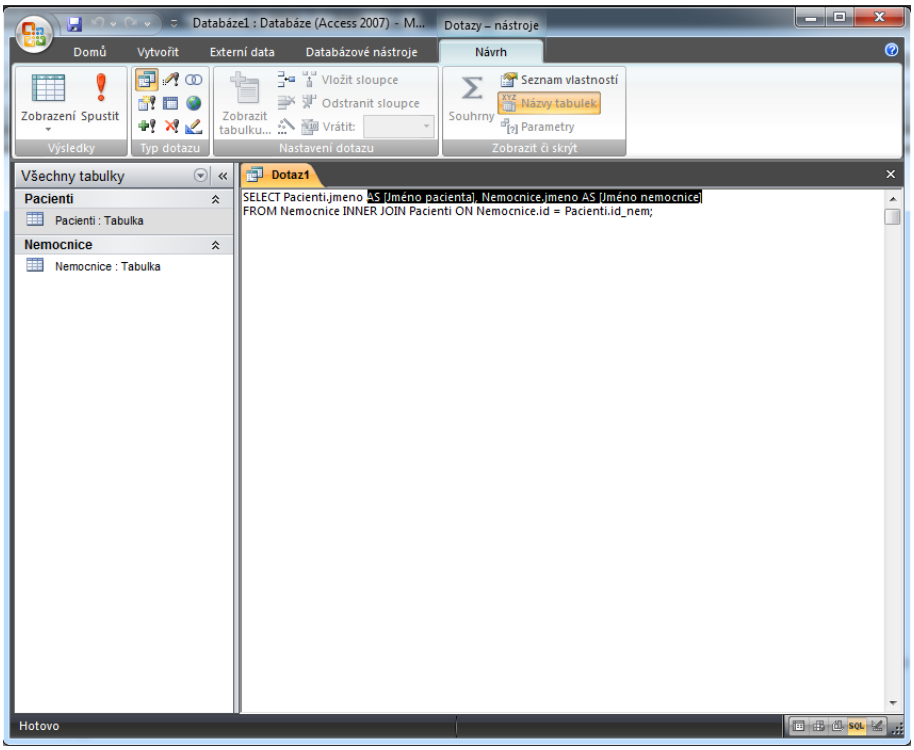


Vztahy zavřeme a přejdeme do položky „Vytvořit“, kde najdeme v kartě „Jiné“ položku „Návrh dotazu“. Musíme vložit naše dvě tabulky, které hned vykreslí náš vztah. Práce je pak velice jednoduchá. Pole, se kterými budeme pracovat, jen přesouváme do polí budoucího dotazu.



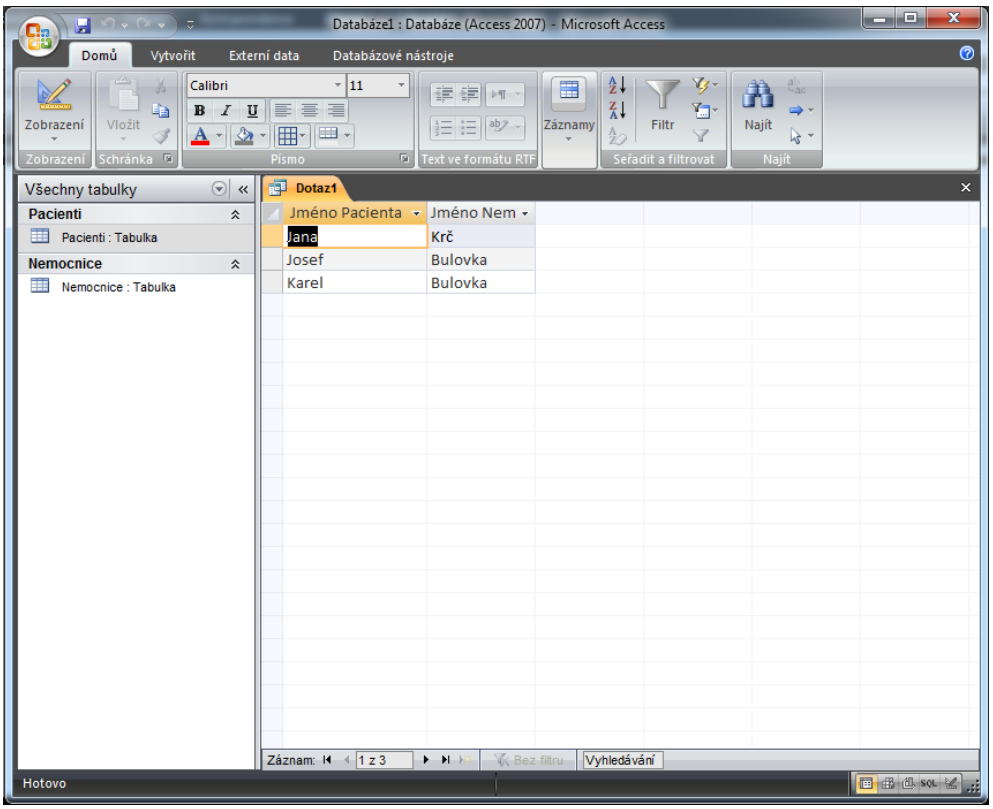
Vše je jednoduše na první pohled vidět, ale pokud chceme pracovat s SQL, musíme se přepnout do jiného režimu, který je v položce „Zobrazení“ viditelný pomocí šipky dolů, kdy na ní klikneme, zvolíme „Zobrazení SQL“, tam už vidíme, jaký dotaz jsme vytvořili a pomocí AS políčka přejmenujeme na vlastní názvy.

Všimněme si, jaký typ spoje zvolil Access oproti OpenOfficu a jakou jinou syntaxi mají vložené nové nápisy po klauzuli AS. Zápis není chybný, ale SQL je prostě velice upravitelné a v různých ohledech různými autoritami modulováno.



Náš výběr by pak vypadal jako v minulém příkladě, ale zde máme jiná data.

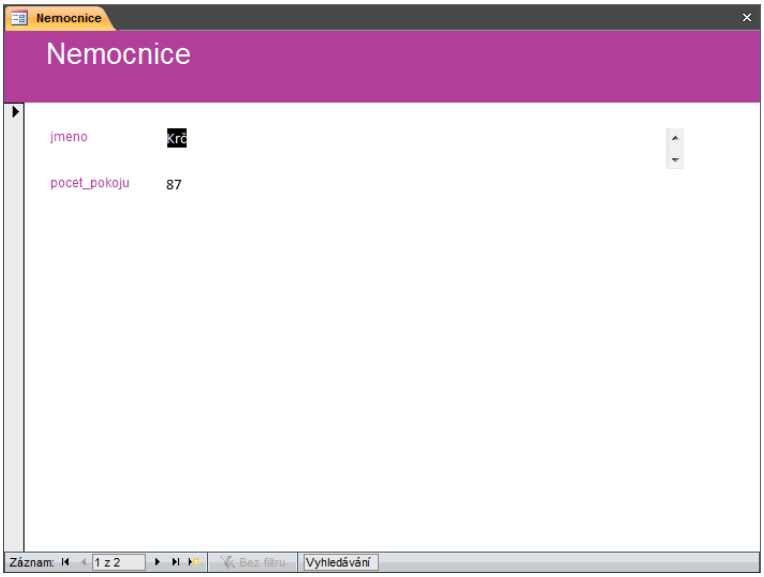
Pokud se chceme vracet, pracujeme zase s položkou „Zobrazení“.



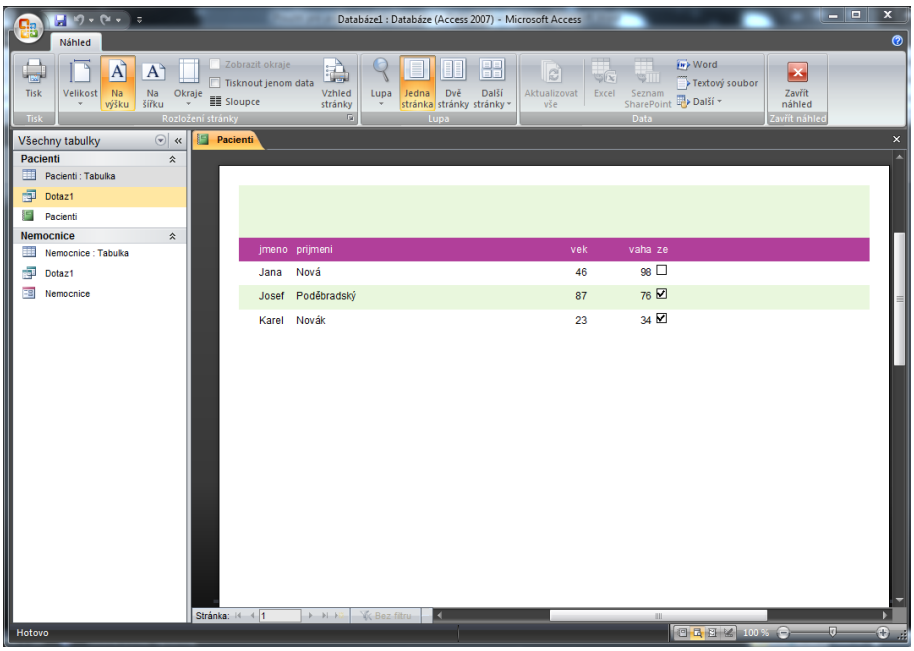
Stejně jako OpenOffice pracuje s formuláři a sestavami, tak i v Microsoft Office Access je možnost pro tyto volby.

Pokud chceme vytvářet formuláře a sestavy, Microsoft Office nám vyjde vstříc velikým počtem průvodců, které odvedou celou práci za nás. Uvedeme zde jen dvě ukázky. Zásady platí naprosto stejné jako u sestav a formulářů v OpenOfficu. Princip obou těchto objektů je stejný.

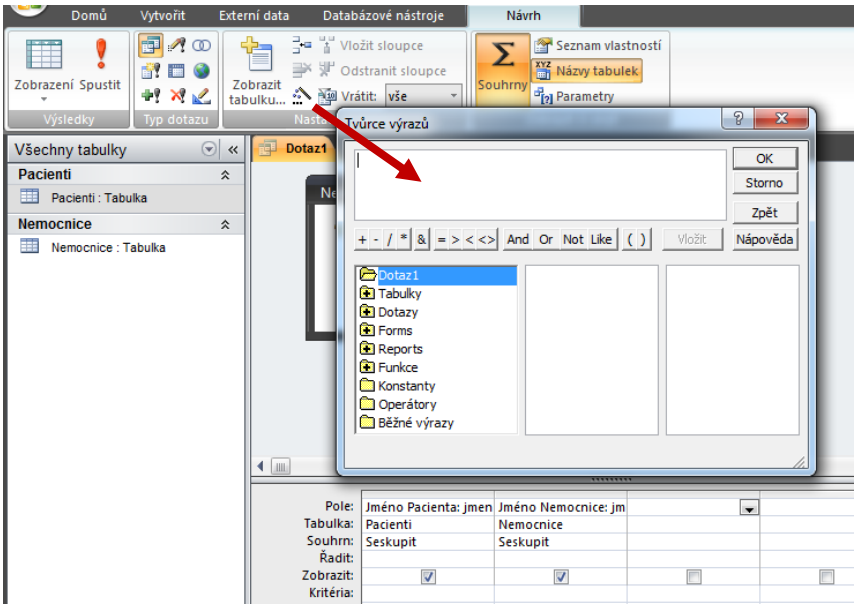
Jak vidíme, zobrazení je stejné a stejná je i práce. Dolní lišta pro posouvání záznamu je prakticky identická.



Sestavy jsou připraveny pro úpravy, aby vypadaly, co možná nejlépe mohou. Vše je stylizováno pro uživatelské pohodlí.



Pokud jste postrádali rychlou volbu pro agregační funkce, je jednoduché je v obou platformách najít. Microsoft Access dokonce podporuje malého pomocníka, který se jmenuje „Tvůrce výrazů“, můžeme v něm vytvořit jak složené výrazy pomocí AND a OR, tak různé spojení, které bychom jinak se syntaxí SQL v Access dělali těžko.



OpenOffice pak nechává pro tuto práci pole „Funkce“ a pole „Řadit“, kde vše můžeme vytvořit rychleji, protože nejsme nuceni otvírat nové okno. Práce je ale trochu jinačí než v Access.

Pokud jste si zrovna neoblíbili použití SQL v kancelářských programech, nezoufejte. Je to sice nejlehčí implementace SQL vůbec, ale není to přeci jen pravá databáze, s kterou pracují normální servery. Je poměrně dost jednoduché se vše ale doučit, protože jak OpenOffice, tak Microsoft Office se snaží o uživatelskou přívětivost, což je dělá velice jednoduché, a pro pokusy a lehké databázové experimenty jsou tyto databáze k nezaplacení. k práci v malé firmě, kde je potřeba vypracovat malou databázi, třeba pro evidenci malých záznamů, je velice snadná volba sáhnout k malé databázi, která neběží nějak závisle na nějakém serveru.

Jak si ukážeme dále, můžeme databáze Microsoftu využít i jinak, tedy spojením třeba s Visual Basicem.

Použití a připojení na databázi

Prakticky jsme už viděli, jak na SQL pomocí malých databází, které si můžeme uchovávat jako soubor, ale kde doopravdy SQL běží neustále a my se s ním setkáme určitě, jsou velké databázové servery. Pak už ani nezáleží na tom, jestli z databáze čerpá místní webová aplikace, či s ní pracuje program, který z ní data přebírá po dobu svého běhu a pracuje s nimi až do vypnutí. Takovéto databáze obvykle zabírají celý jeden server a na jistém portu poskytují svoje připojení, přihlášení a následnou práci. Musíme tedy vědět, že databáze běží na serveru jako jeden celistvý program schopný pracovat na vlastních síťových portech.

My jako uživatelé ale nemusíme ani znát porty, na kterých běží, protože my jsme správci a budeme se snažit do databáze dostat tak, abychom ji mohli použít. Na datovém výstupu nám nezáleží, protože ten bude programovat někdo jiný, či se k němu připojí aplikace po nastavení sama. Naším úkolem je se dotazovat na databázi, což se dělá pomocí příkazového interpretru. Sami ale víme, že nemusíme použít jen příkazový řádek, ale i jiné možnosti. Některé databáze od svého nainstalování provádí připojení přes webové rozhraní přístupné na portu buď 80, či 8080, což si také ukážeme.

Někde je bohužel nutné nějaké drobnosti doinstalovat, ale to nemůže tato kniha řešit, protože to se dozvíte vždy na stránkách výrobce. My si ukážeme, jak se jednoduše na serveru k databázi připojit a pracovat s ní. Pamatujme si, že se připojujeme na serveru ke službě, která takto běží neustále, v reálném čase, na rozdíl od databází, které se užívají v kancelářském prostředí. Pokud je taková databáze už v chodu s jiným programem, který z ní data čerpá, pamatujme, že je někdy velice obtížné získávat data zpět, když je náhodou vymažeme, proto před každým krokem přemýšlejme, co vlastně chceme dělat.

MySQL

MySQL nabízí velké množství možných přístupů. V základě ale vždy podporuje nízkoúrovňový přístup pomocí příkazového řádku.

Základní příkazy si vysvětlíme na příkladě:

Nejdříve se musíme přes náš příkazový řádek přihlásit k databázi. Pamatujme, že když není databáze přímo zavedena do systému, musíme spustitelný soubor mysql najít ve své složce. Pak už se přihlásíme dle příkazu:

```
mysql -u uživatelské_jméno --password=heslo
```

Zde se přihlásíme pomocí přepínače -u jako uživatel root, tedy administrátor databáze.

```
create database pokus;
```

Zde jsme vytvořili databázi pokus. Nemusíme se bát problémů s oprávněními, proto jako root nemáme žádná omezení.

```
show databases;
```

Tento příkaz nám ukáže, jaké databáze můžeme použít. Pamatujme, že na velkém serveru to může být mnoho databází běžících vedle sebe, ale pracujících v jednom programu.

```
use database pokus;
```

Tento příkaz nás přesunul do databáze pokus. To znamená, že teď jsme schopni vidět všechny tabulky a pracovat s databází tak, jak jsme si již říkali.

```
show tables;
```

Tento příkaz nám ukáže všechny tabulky v databázi.

Následně už postupujeme podle SQL. Pamatujme, že pokud píšeme příkaz, můžeme jej psát na více řádků a zakončíme jej symbolem pro středník, tedy „;“. Pak se příkaz spustí a vypíše, za jak dlouho byl proveden a co se přesně stalo.

Ukážeme si teď příklad, při kterém vybereme data z již běžící databáze a tabulky uziv:

```
mysql> select * from uziv;

+----+-------+----------------------------------+

| id | jmeno | heslo                            |

+----+-------+----------------------------------+

|  1 | nemo  | e587f6146ebfbdefdc028c591643f220 |

|  2 | jj    | bf2bc2545a4a5f5683d9ef3ed0d977e0 |

|  3 | hh    | 5e36941b3d856737e81516acd45edc50 |

|  4 | gg    | 73c18c59a39b18382081ec00bb456d43 |

|  5 | ff    | 633de4b0c14ca52ea2432a3c8a5c4c31 |

|  6 | dd    | 1aabac6d068eef6a7bad3fdf50a05cc8 |

|  7 | ss    | 3691308f2a4c2f6983f2880d32e29c84 |

+----+-------+----------------------------------+

7 rows in set (0.02 sec)
```

Vidíme, že formátování podléhá možnostem příkazového řádku a není moc komfortní.

Příkazy, kterými ovládáme databázi MySQL jsou rozdílné od příkazů SQL v použití přístupu k objektům. Uvedeme si výpis jednotlivých příkazů. Už jsme se setkali příkladem s příkazem USE, který není přímo definován jako SQL klauzule. Potřebujeme je k práci s databází na základě přístupu a režie kolem, ale ne k práci s daty. K takovýmto příkazům se dostaneme pomocí příkazu help.

?	(\?) Synonym pro ‘help'.
clear	(\c) Vyčistí právě se ukazující výpis dotazů.
Connect	(\r) Znovu se připojí k databázi, používá se nejdříve s názvem_databáze a servem, tedy connect pokus localhost.
delimiter (\d)	Nastaví oddělovací znak, kterým spouštíme příkazy SQL, defaultně je nastaven středník „;“.
edit	(\e) Pomůže editovat příkazy pomocí editoru, který je zadán v proměnné $EDITOR.
ego	(\G) Pošle příkaz databázi.

Exit (\q) Odejde z databáze a je stejný jako příkaz quit.

Go (\g) Odešle příkaz databázi.

.

Help (\h) Zobrazí pomoc.

Nopager (\n) Odnastaví pager.

Notee (\t) Nezapisuje do vnějšího souboru.

.

Pager (\P) Nastaví PAGER [nový_pager]. Zobrazí výstup pomocí nastaveného PAGERu. Dávejme si pozor při nastavování, abychom nenastavili symbol, který databáze nepřijme.

print (\p) Vypíše běžící příkaz.

Prompt (\R) Změní prompt, tedy nápis před polem pro příkaz.

Quit (\q) Odejde z databáze.

rehash (\#) Přenastaví hash.

Source (\.) Provede SQL skript, který zadáme jako argument s jeho celou cestou k němu v našem souborovém systému.

status (\s) Vypíše status databáze.

Systém (\!) Provede systémový příkaz.

tee (\T) Nastaví vnější soubor pro výpis do něj. Argumentem udáme, kam se bude ukládat: tee soubor_výpisu.

Use (\u) Použije jinou databázi.

Charset (\C) Přepne na jinou znakovou sadu.

warnings (\W) Ukazuje varování za příkazy.

nowarning (\w) Neukazuje varování za příkazy.

Pokud jste nenarazili na nic, co by vám mohlo pomoci, zkuste se obrátit přímo na portál MySQL, které je právě v přechodu pod jiného vlastníka a určitě se bude hodně měnit. Je nutné si uvědomit, že většina programů podléhá základům prodeje, a proto nemůžeme přesně říct, jak bude vypadat ovládání takové databáze za dalších pár let.

Postgresql

Do databáze Postgresql se přistupuje pomocí souboru psql, který nás navede přímo do databáze. Postqresql ale musí být nastaveno pomocí svých konfiguračních souborů k práci s našimi lokálními účty na našem počítači. Návod najdeme na stránkách Postgresql. Problém je v odlišnostech nastavení na jiné operační systémy.

V zásadě musíme vytvořit uživatele pomocí **su postgresql**, následně vepíšeme **createuser pokus** a zvolíme, které atributy bude mít uživatel, tedy zda bude administrátor a jestli bude moc vytvářet tabulky, či je normálním uživatelem.

```
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) y
Shall the new role be allowed to create more new roles? (y/n) y
```

Vlastně vytvoříme předem definované role. Postgresql vlastní navíc rozšíření funkcí rolí, které dovoluje schéma jednoho uživatele přidělit více uživatelům. Následně se přihlásíme, ale musíme vyplnit nějakou databázi, ke které se chceme přihlásit. Standardně je v systému databáze postgres.

psql postgres

Databáze nás uvítá a vypíše, jak se máme dostat k nápovědě.

psql (8.4.3)

Type "help" for help.

postgres=# help

You are using psql, the command-line interface to PostgreSQL.

Type: \copyright for distribution terms

\h for help with SQL commands
- pomoc s SQL dotazy

\? for help with psql commands
- pomoc s psql databází

\g or terminate with semicolon to execute query

\q to quit

postgres=#

Nás nebude zajímat nápověda SQL, protože o té máme celou knihu, ale podíváme se na nápovědu k psql, tedy k databázi Postgresql. Výpis příkazů je zkrácený.

\g [FILE] or ;	proveď příkaz a pošli výsledek do souboru, kde [FILE] argumentem je soubor
\h [NAME]	pomoc pro syntaxi SQL příkazů v souboru, kde [NAME] je cesta k souboru
\q	opuštění psql
\copy	proveď SQL kopii a převeď do souboru
\echo [STRING]	vypiš text do standardního výstupu
\i FILE	proveď příkaz ze souboru, kde FILE je argument pro soubor
\o [FILE]	pošli všechny výsledky dotazů do souboru, kde [FILE] je argument pro soubor

Informační:

(options: S = show system objects, + = additional detail)

\d[S+]	list všech tabulek, náhledů,...
\d[S+] NAME	popis objektů, tabulek, pohledů,…, kde NAME je argument pro objekt
\da[+] [PATTERN]	výpis agregací
\db[+] [PATTERN]	list tabulátorů
\dc[S] [PATTERN]	list konverzí
\dC [PATTERN]	list rolí

\dd[S] [PATTERN]	výpis komentářů na objekt
\dD[S] [PATTERN]	list domén
\des[+] [PATTERN]	list cizích serverů
\deu[+] [PATTERN]	list uživatelských map
\df[antw][S+] [PATRN]	list [jen agg/normal/trigger/window] funkcí
\dF[+] [PATTERN]	list textových hledacích funkcí
\dg[+] [PATTERN]	list rolí (skupin)
\di[S+] [PATTERN]	list indexů
\dl	list velkých objektů, podobné jako \lo_list
\dn[+] [PATTERN]	list schémat
\do[S] [PATTERN]	list operátorů
\dp [PATTERN]	list tabulek, pohledů a sekvencí s jejich přístupovými právy
\ds[S+] [PATTERN]	list sekvencí
\dt[S+] [PATTERN]	list tabulek
\dT[S+] [PATTERN]	list datových typů
\du[+] [PATTERN]	list rolí
\dv[S+] [PATTERN]	list pohledů
\l[+]	list všech databází
\f [STRING]	ukaž nebo nastav [STRING] separátor polí pro nevyrovnaný výstup dotazu
\H	následuj HTML výstup
\pset NAME [VALUE]	nastav výstup tabulky (NAME := {format|border|expanded|fieldsep|footer|null| numericlocale|recordsep|tuples_only|title|tableattr|pager})
\T [STRING]	nastav HTML výstup pro tag <table> atribut, či jej odnastav

Spojení:

\c[onnect] [DBNAME|- USER|- HOST|- PORT|-] připoj se k nové databázi

\encoding [ENCODING]	ukaž nebo nastav [ENCODING] kódování
\password [USERNAME]	bezpečně změň heslo pro uživatele

Operační systém:

\cd [DIR]	změň právě používaný adresář
\timing [on|off]	následuj časování řádků v příkazovém řádku

Pokud jste nějaké funkce přímo nepochopili, najdete vysvětlení na stránkách výrobce, protože může mít novější manuál. Není třeba znát je, ale vědět, kde je najít a kdy je použít. Správa databází na tak nízké úrovni je velice účelová. Nesmíme zapomínat na středník, a že přístup na databázi je vždy poněkud jiný.

Připojení na databáze přes webové a jiné rozhraní

Databáze, které jsou velké a na jejich skriptech pracují stovky až tisíce zaměstnanců, mají vlastní rozhraní, ke kterému se dá připojit pomocí nějakého uživatelsky příjemného vstupu. Nejčastěji je takový přístup vyřešený pomocí webového rozhraní na portu 80, tedy přes protokol http. Můžeme si to tedy představit tak, že nainstalujeme databázový server a k němu přiinstalujeme součást pro ovládání z webu, někdy se ani nemusí přiinstalovávat, stačí jen nainstalovat server a rozhraní už se přidá samo. Následně se přihlásíme na webový server na databázovém serveru, tedy otevřením webového prohlížeče a napsáním adresy localhost, či 127.0.0.1. Přihlášení se liší podle toho, jaké rozhraní používáme.

ORACLE APEX

ORACLE přišel s rozumnou alternativou připojení se na databázi, kterou neustále rozvíjí a upravuje. Nabízí nám náhled na databázi, připojení se pomocí SQL dotazů a také interaktivní správu databáze pomocí průvodců a různých zařízení pro jednoduchou zprávu. Výhodou takového rozhraní je, že i s malou znalostí SQL můžeme pracovat velice efektivně a rychle. Nevýhodou je cena za každou distribuci ORACLE databáze, která se dělí na ceny podle toho, jak moc dokáže se systémem pracovat.

ORACLE databázi můžeme získat na webových stránkách ORACLE, kde budou i poslední verze programu.

V zásadě můžeme rozdělit jednotlivé edice podle ceny:

ORACLE Database Express Edition

Express Edition je řešení pro všechny vývojáře, či začínající programátory, protože je zcela zdarma. Naproti tomu je ale také schopná využít jen jedno jádro procesoru a pracovat jen s jedním GB ran paměti a také s pouze čtyřmi GB pevné paměti disku, což z ní dělá databázi neschopnou pracovat na velkých úlohách. V zásadě je to ale výborné řešení pro rozšíření vlivu ORACLE nad trhem, kde se naučíte pracovat na slabé databázi, abyste pak ve velké firmy pracovali dobře na drahé databázi, kterou, kvůli zkušenostem, budete doporučovat více, než databáze, které jsou levnější.

ORACLE Database Standard Edition

Standard edice je už určena pro malé firmy, kde nemáme žádná omezení kolem práce nad paměťmi, ale stále jsme schopni pracovat jen s dvěma jádry. Taková databáze někdy bohatě stačí, ale pro vrcholné zatížení při doopravdy velkých projektech, je zcela nevhodná a neodpovídající.

ORACLE Database Enterprise Edition

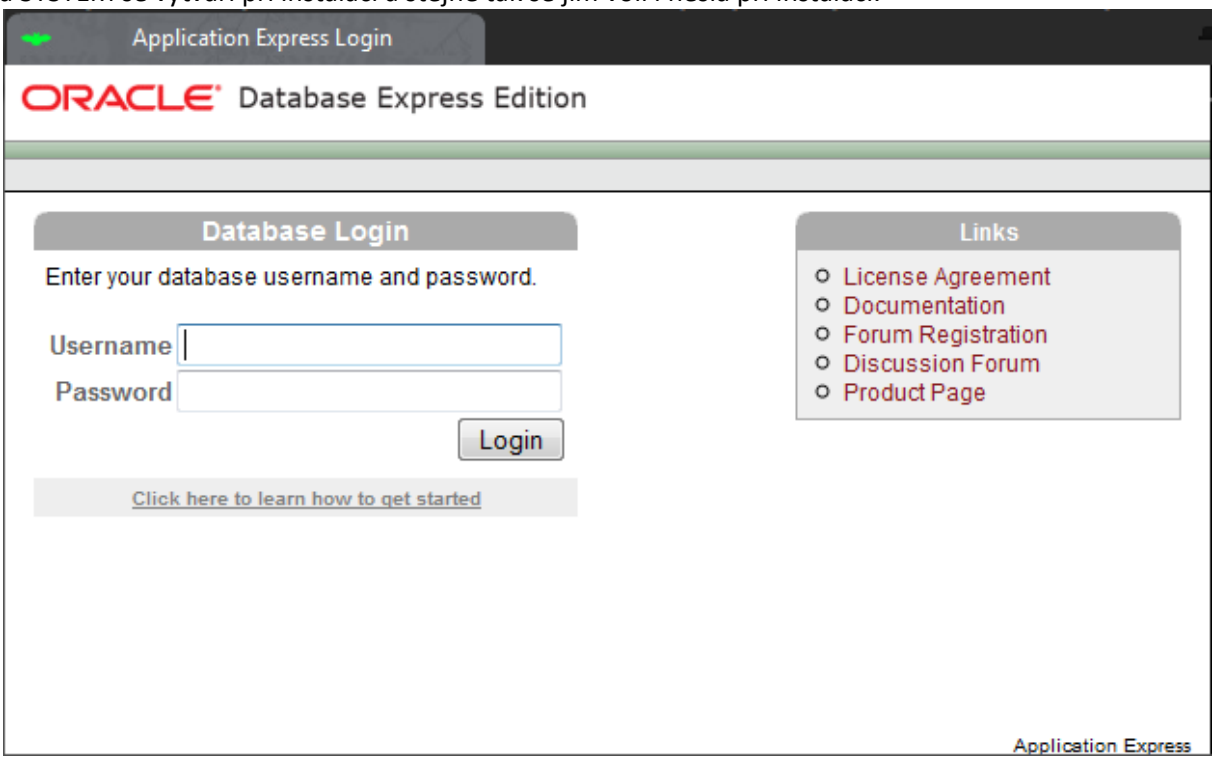
Enterprise edice je jedna z nejvýkonnějších databází vůbec, kde se počítá s nasazením v nadnárodním celku a na ohromných projektech. Enterprise edice dovoluje přidání dalších komponentů, práci s bezpočtem jader a ohromným datovým úložištěm (v rámci efektivity uvažujme i souborový systém). Nevýhodou takové edice je její ohromná cena, která se ale vyplatí, když si představíme, kde taková databáze obvykle běží.

APEX, o kterém se budeme bavit u databází ORACLE, je právě takové webové rozhraní pro správu databáze. Ukážeme si, jak se do něj dostat, jak jej používat a kde najdeme jednotlivé části, které nám pomohou efektivně databázi používat. APEX je přenesený význam slov Application Express.

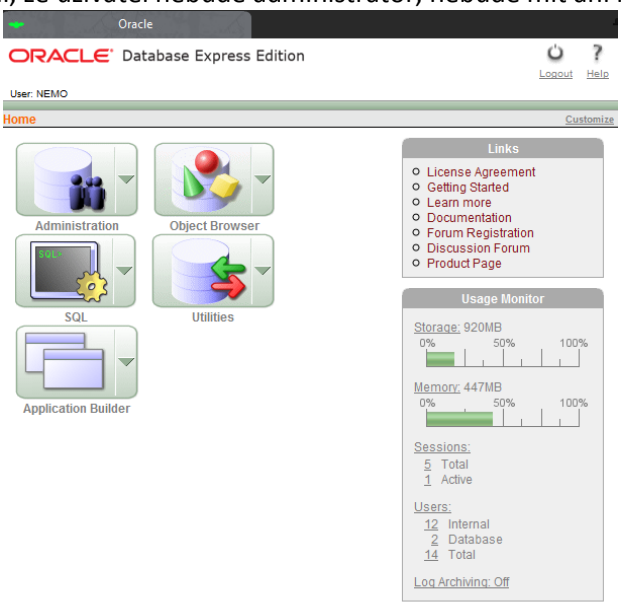
Pamatujme, že do APEXu se dostaneme pomocí připojení se na webový server databáze, tedy buď na localhost, kde je naše databáze (na lokálním počítači, kam jsme ji instalovali), nebo na server, který jsme použili, ale pak musíme vědět, na kterém serveru je a adresu použít pro připojení. Vše můžeme zvolit při instalaci ORACLE databáze.

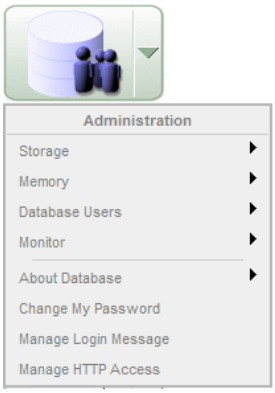
Zde vidíme uvítací okno APEXu. Přihlásíme se pomocí uživatele, který je předem vytvořený od nějakého administrátora, či jsme jej vytvořili při instalaci. ORACLE pracuje s uživateli SYS a SYSTEM, kde oba mohou provádět celou správu databáze, ale neměli bychom je používat jako normální uživatele. Po

přihlášení na systémový účet tedy hned vytvoříme nového uživatele a přiřadíme mu heslo s omezeními. SYS a SYSTEM se vytváří při instalaci a stejně tak se jim volí i hesla při instalaci.

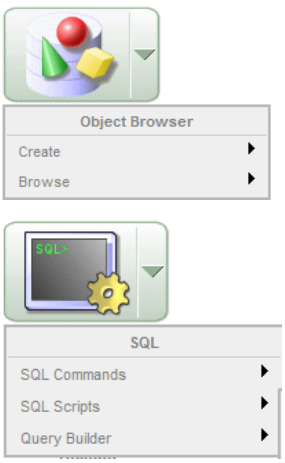


Po přihlášení už vidíme nabídku databáze, ve které se můžeme pohybovat a pracovat s ní. Přihlásili jsme se jako uživatel nemo, který je už v databázi vytvořený (ukazatel vlevo v horním rohu). Vpravo vidíme odkazy na dokumentaci a využití databáze. Podle nastavení oprávnění se nám pak ukazují jednotlivé části nástrojů. Pokud jsme řekli, že uživatel nebude administrátor, nebude mít ani kolonku Administration.





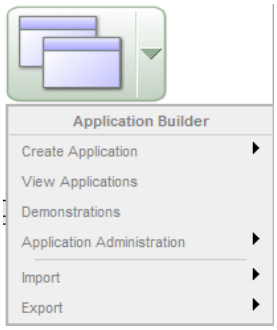
Projdeme si všechny kolonky, abychom věděli, co v nich můžeme hledat. Nemusíme se bát, že bychom někdy nevěděli, jak na to, protože APEX nás rychle navede.



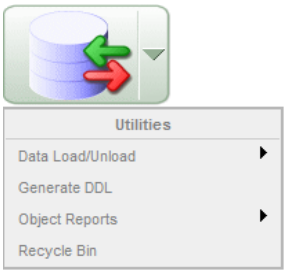
Nejdříve uvidíme Administration položku, kde můžeme změnit doopravdy vše kolem omezení nad databází až po nastavení ukládání a používání paměti. Výhodou takové správy je optimalizace databáze na nás server. Zajímavé je, že ORACLE všechno, co používá, APEX

použil i do svého SQL, které sice není standardní, ale dovoluje ovládání databáze za použití jen a pouze SQL.

Můžeme použít také prohlížeč objektů, který nám pomůže s vytvářením tabulek, či jiných součástí databáze, bez znalosti SQL:



Dále vidíme položku pro SQL, kterou si více popíšeme, abychom věděli, kde SQL máme nasadit, ale už předem vidíme, že nemusíme vkládat pouze příkazy ale také vytvářet skripty a dokonce je v APEXu něco jako stavitel dotazů, který nám pomocí průvodce pomůže dotaz vytvořit.

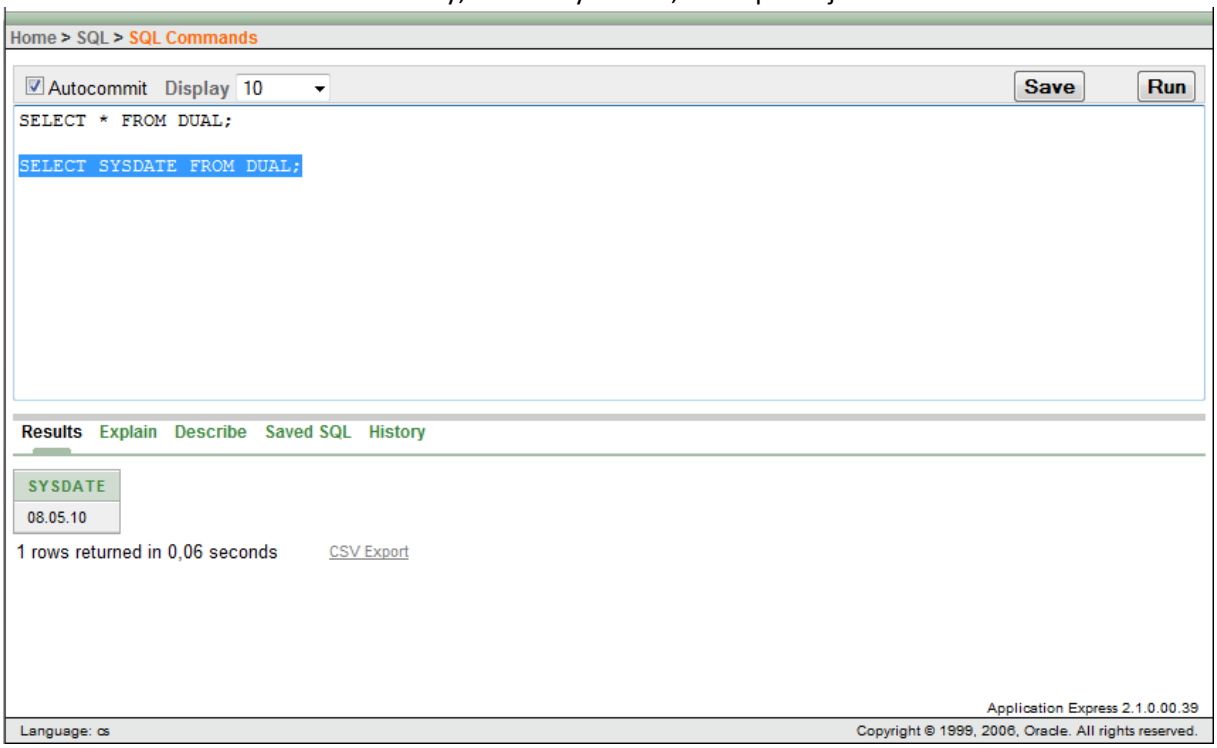


Dále máme položku pro součásti, které dovolují nahrávat či exportovat data. Můžeme také prohlížet jakési zprávy o objektech. Navíc ORACLE obsahuje koš, kde jsou data uchovávána na chvíli před úplným vymazáním.

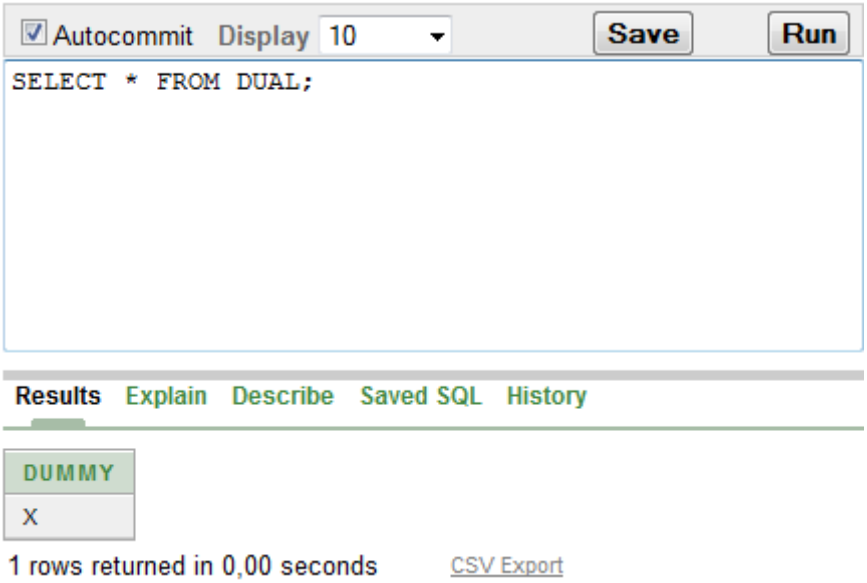
Poslední částí je stavitel aplikace, ve kterém se můžeme sjednotit naší celou práci do jakési aplikace.

Pro práci s SQL tedy rozklikneme položku SQL a v ní SQL Commands, kde můžeme vkládat dotazy, o kterých jsme si už přečetli.

Zajímavá je funkce, kde můžeme pracovat jen s SQL příkazem, který je zrovna vybraný. Zde jsme vybrali SYSDATE, tedy systémové datum, a proto je výstup jen datum. U ORACLE vždy vybíráme z tabulky, nesmíme nechávat SELECT bez tabulky, z které vybíráme, takže proto je zde tabulka DUAL.



DUAL je zvláštností ORACLE databází, ale poskytuje nám možnosti, které bychom jinak těžko hledali. Tabulka DUAL obsahuje jedno pole a jeden zápis:

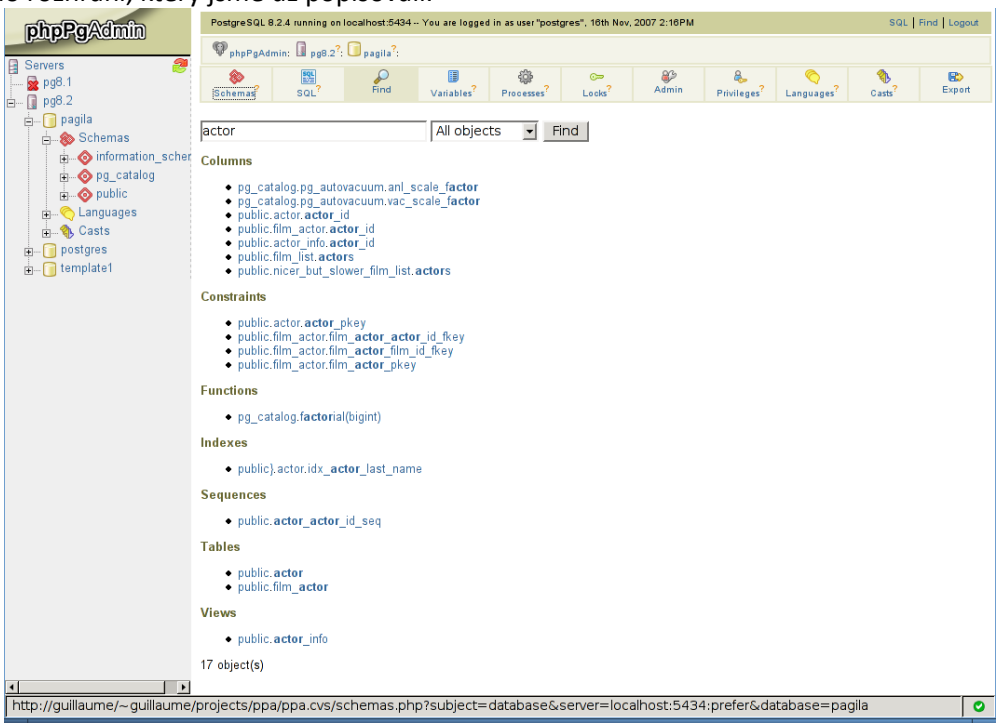


ORACLE je velikou společností, která má vládu nad trhem s databázemi, proto se můžeme dočkat dalšího rozrůstání ORACLE databází, ale vždy mysleme na to, že všechno něco stojí. ORACLE nepatří mezi nejlevnější, ale své služby dokáže vyvinout k výborné obslužnosti, proto je také tak oblíbený.

PHPPgAdmin

PHPPgAdmin je jedna z alternativ pro připojení na databázi přes webové rozhraní pro Postgresql databáze. Vidíme ale, že tato aplikace je složena z PHP, které ale potřebuje vlastní webový server Apache. Prakticky pak musíme nainstalovat databázi, webový server a pak ještě PHPGrAdmin. Je to vlastně takový chybějící článek mezi uživatelem a databází. To vše zní velice složitě, ale pamatujme, že takovéto aplikace vyvíjí komunita kolem otevřených aplikací, a tím určuje neustálou pomoc na fórech a nekonečně ohromnou zásobu aktualizací.

Přesto je takovéto řešení někdy daleko elegantnější, než používat drahou databázi. Samotné rozhraní je velice jednoduché a stabilní. Pracuje na základě PHP funkcí. Všechny příkazy pro SQL jsou zde implementovány také v prohlížečích. Můžeme postupovat i v okně SQL, kde vpisujeme rovnou dotazy, a rozhraní nám říká, jak je databáze provedla. Můžeme zde také vidět výstup, který připomíná výstup z příkazového rozhraní, který jsme už popisovali.

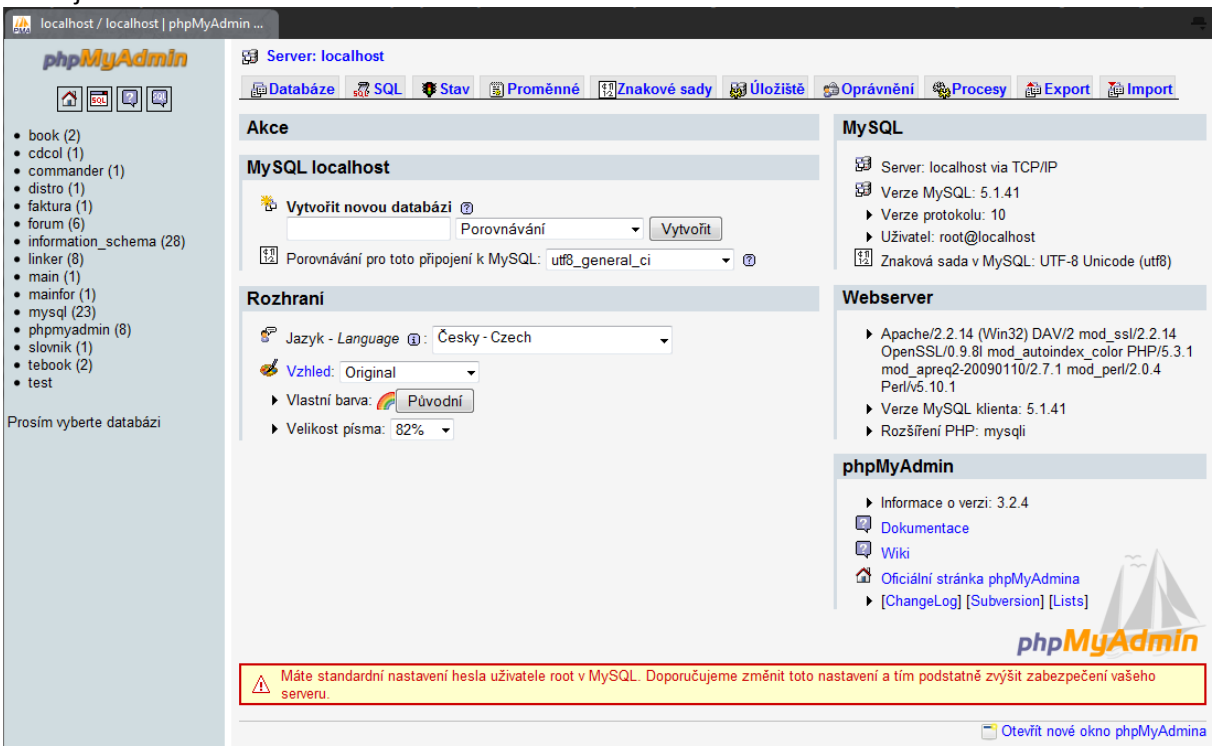


S instalací a používáním vždy pomůže a poradí komunita, která je rozrostlá všude na světě. V zásadě stačí nějakou vyhledat přes webový vyhledávač a přečíst si, co je potřeba. I když, pokud nainstalujete správně rozhraní, není už třeba nic jiného. Rozhraní je velice intuitivně založené a, na rozdíl od ORACLE i jiných firem, je většinou lokalizované.

PHPMyAdmin

Kvůli rozšíření MySQL databází bylo navrženo webové rozhraní i pro MySQL databáze a zase pracuje pomocí PHP, tedy patří do rodiny takovýchto rozhraní. Prakticky je takové rozhraní stejně přístupné a otevřené, jako je PHPPgAdmin.

Jak vidíme, k naší spokojenosti si můžeme měnit i vzhled rozhraní. Také nalezneme různé průvodce, jak vytvořit databázi a tabulky, včetně ostatních objektů. I když PHPMyAdmin je ve správě komunity, přináší velice jednoduché ovládání a intuitivní rozhraní.



Příklad

Pokud jsme již pochopili, jak na SQL, musíme se seznámit s tím, jak vytvořit nějakou databázi v praxi. Bohužel necháme pouze a jen na vás, jakou databázi použijete, ale zachováme takový postup, aby bylo prakticky jedno, na kterém databázovém stroji příklad budete provádět, protože postup musí být, dle SQL, vždy stejný. Pro ilustraci můžeme použít některé náznaky z různých databází a celou úlohu pak schematicky shrnout v SQL, které je stále stejné.

Zadání

Představme si situaci, kdy přijde zákazník a bude si přát vytvořit databázi, které bude zaznamenávat studenty, lektory, učebny a kurzy, na které pak budou studenti docházet. Je nutné, aby u každého studenta a každého profesora bylo zaznamenáno vše, co by kdy bylo potřeba.

Zákazník nepotřebuje, aby byla databáze nějak graficky upravena (formuláře, sestavy a jiné nástroje Access či OpenOffice), ale vyžaduje dotazy na databázi, které pak použijí ostatní programátoři jako schéma pro svou další práci.

Navíc potřebuje, aby byla databáze snadno upravovatelná a byla relační, protože o tom slyšel v televizi (takové důvody bývají většinou slyšet u zákazníků, protože málokdo doopravdy ví, co znamená slovo jako „relační“, či „atomární“). Také si přeje, aby věděl, jak databáze vypadá dříve, než se bude zadávat do databázového programu.

Zákazník předem neví, kolik dat bude muset databáze pojmout, ale přeje si, aby byla rozšiřitelná a zaznamenávala vše velice podrobně.

Úvaha nad řešením

Pokud se zamyslíme nad základním přáním zákazníka, určitě budeme muset převzít iniciativu a navrhnout databázi více velkoryse, než třeba zákazník sám plánoval. Pokud mluvil zákazník o programátorech, pak se určitě zajímejte, o jaké programátory jde a jak je případně kontaktovat.

Po přečtení druhého odstavce je snadné nabýt přesvědčení, že zákazník nás nechá pracovat na tom, čemu skutečně rozumíme a nebude rušit s grafikou, ale to je velký kámen úrazu, protože v předposledním odstavci se dovídáme, že musíme navrhnout i nějaký diagram, který bychom mohli hned prezentovat jako výsledek prozatímní práce. Nikdy nepodléhejte dojmu, že stačí jen programovat, protože tomu většina zákazníků nerozumí. Nejlepší je vše nějakým způsobem dokumentovat, protože výstup může být dobrý, ale bez průvodních zpráv vše vypadá tak, že nás to stálo až moc času, za který zákazník může odmítnout zaplatit. Pokud jej budeme neustále zaplavovat zprávami a diagramy, bude mít pocit, že stále pracujeme, i když jen třeba řešíme problém se vztahy a vytváříme každý den lepší řešení.

Nadále bychom se měli snažit zjistit, na jakém asi databázovém stroji bude databáze běžet. Mysleme objektivně, nemůžeme vědět, jak se naše ORACLE SQL může chovat na MySQL databázi. Nikdo nám nemůže upřít informaci, která je důležitá k budoucímu běhu firmy. Pokud zákazník ale řekne, že určitě počítá s nějakou webovou stránkou, pak dříve, či později bude potřebovat databázi běžící jako službu na nějakém serveru, což znamená, že bude volit buď nějakou dražší variantu, jakou je třeba ORACLE, či levnější MySQL. To už je pro nás snadný impulz k tomu, abychom s tím počítali, protože převod hezky vypadající databáze Access na webové rozhraní bývá někdy velice složitý.

Ze základu pro nás platí stále normalizační zákony, tedy rozklad dat na atomární data (naprosté základy dat, příkladem místo pole adresa uděláme pole město, ulice, PSČ, ...). Také navrhneme vazby a vypracujeme, jaké vazby jsou potřeba, takže ze začátku nakreslíme nějaký počeštěný databázový diagram. Pro tento účel můžeme také použít ERD nástroj od MySQL, tedy program MySQL Workbench, či jiný (Dia,...). Pokud ale chceme, aby to bylo doopravdy v jazyce srozumitelné, je nutné vše lokalizovat. ERD diagramy vypadají velice hezky z Workbench, ale vztahy jsou psány anglicky a my nevíme, jak na tom zákazník je s angličtinou, takže uděláme diagram po svém a hlavně, aby byl srozumitelný.

Diagram

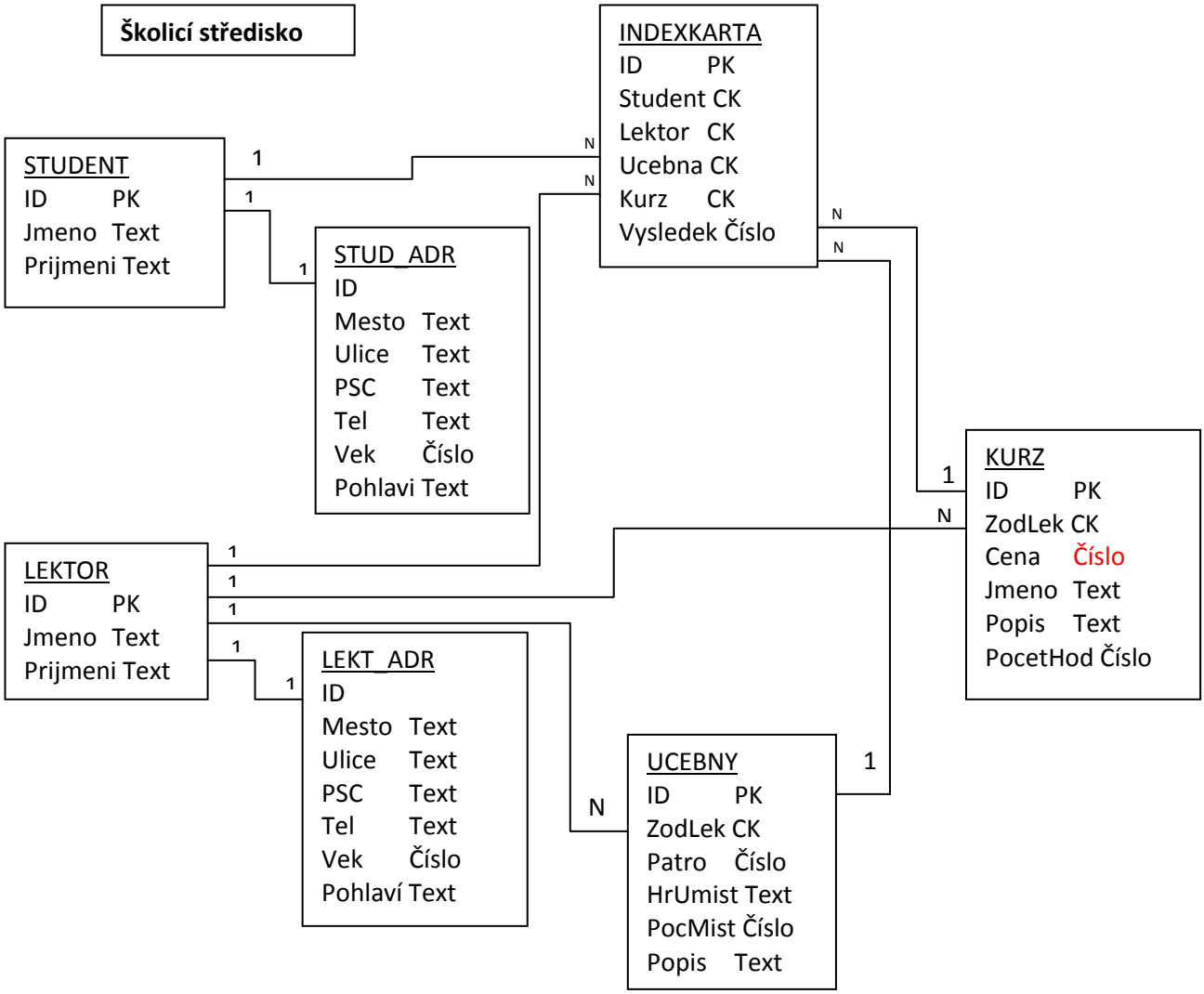
V diagramu znázorníme naprosto vše, ať je to primární klíč, či cizí, a budeme se snažit naznačit i datové typy. Zkusme se ale držet při zemi a jako datové typy volit třeba jen takové, které jsou schematické a pak budou jiné. Jako CHAR() napíšeme raději text, jako INT() napíšeme číslo.

Pamatujme stále na to, že diagram musí být přehledný a urovnaný. Na první pohled by měl zákazník vědět, co budeme zaznamenávat a co v databázi bude. Mnohdy je diagram jediná možnost, jak udělat

nějaký momentální výstup naší práce. Vykresleme nejdříve takové entity, které jsou zcela jasné a nebudou se moc rozšiřovat, tedy student, či lektor.

Mysleme také na odbourání vazby m:n, tedy přidáním nějaké další tabulky. Nakonec takové řešení uděláme i v praktickém životě, ale zákazník jej ani nemusí pocítit, naopak jej může přivítat. Zde to bude pomocí tabulky INDEXKARTA, tedy nějaká shrnující tabulka, co se týče studenta v nějakém kurzu.

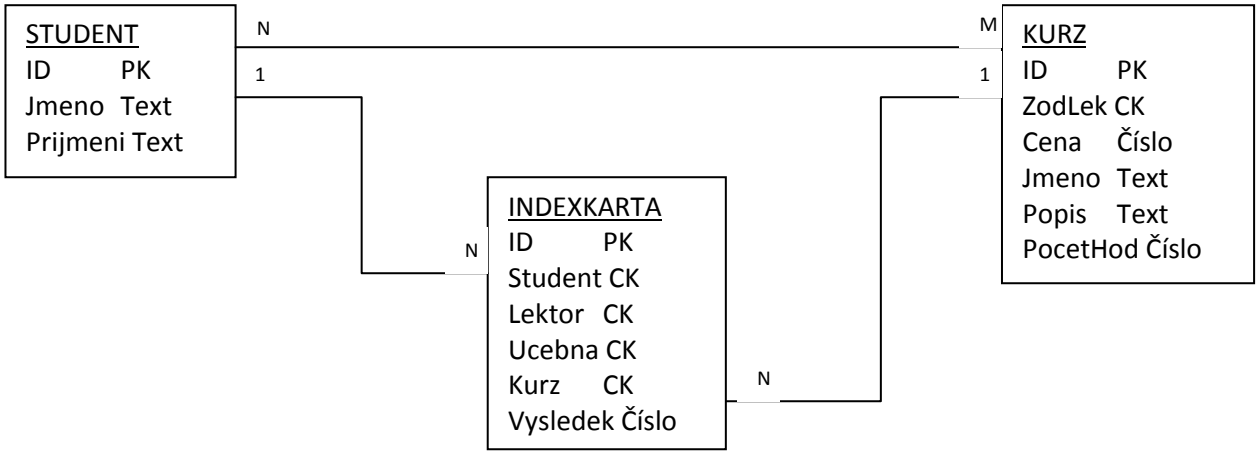
PK je zde zkratka pro primární klíč a CK pro cizí klíč.



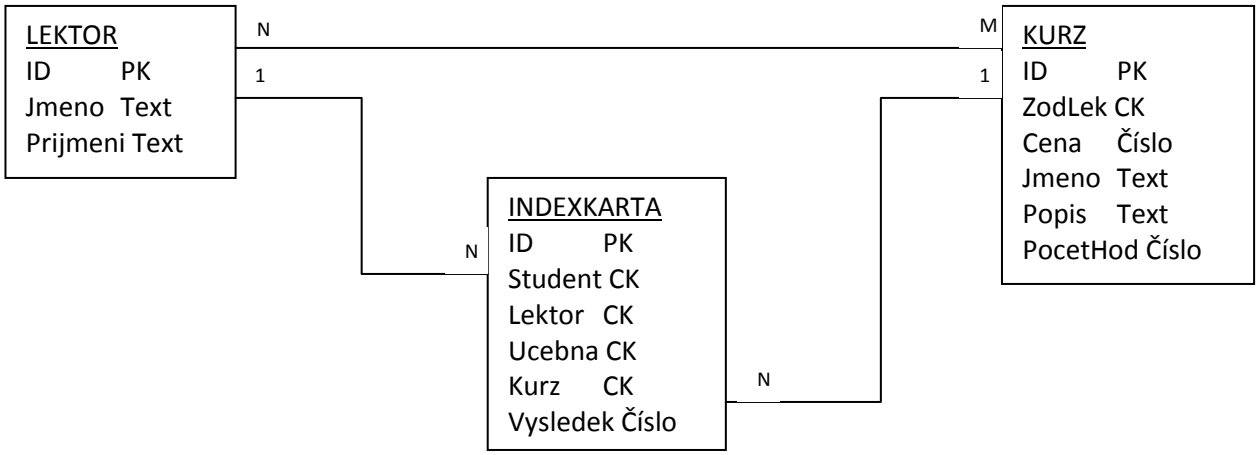
Teď jsme si rozložili nějaký hrubý diagram toho, co naše databáze bude obsahovat. Byli jsme velice otevření a přidali jsme i tabulky, které mají vazby jedna ku jedné, tím si ukládáme data, která by nemusela být na první pohled vidět, a tak jsme do jisté míry zaručili malou ochranu citlivých dat. Pak jsme vytvořili vazby jedna ku mnoha.

Prakticky můžeme jednotlivé části rozebrat na vazby N:M, do kterých se hodí hned čtyři segmenty diagramu.

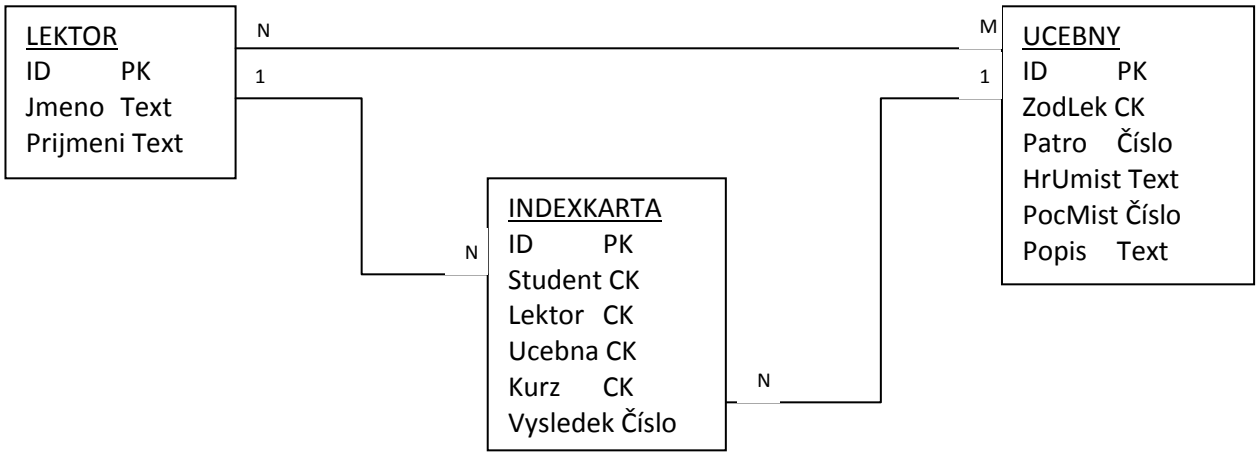
Nejdříve více studentů může docházet na více kurzů.

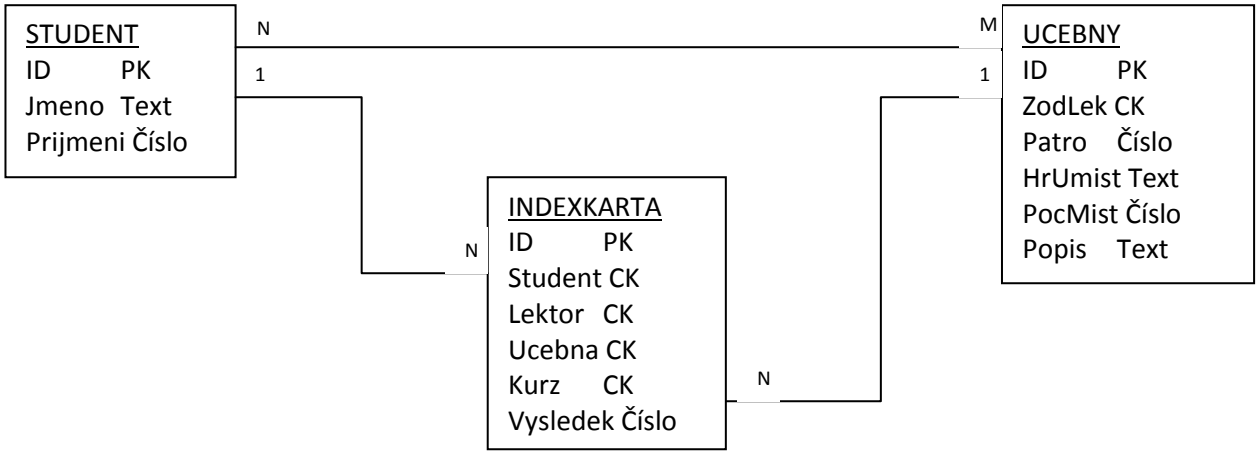


A potom vztah, kde více lektorů může učit více kurzů.



Následně víme, že více studentů dochází do více učeben a více lektorů učí ve více učebnách. Všechny tyto vztahy si znázorníme.





Vynechali jsme naprosto známé vazby, že více lektorů může učit více studentů a stejně tak naopak. Takové vazby rozebereme s lehkou znalostí normalizačních zákonů do malé chvíle bez toho, abychom nad nimi přemýšleli, ale pokud chcete ohromit zákazníka, určitě je rozkreslete.

Následně jsme zmenšili některé názvy do menší formy, aby byl diagram kompaktnější a trochu čitelnější, podle toho víme, že jeden lektor může být ZodLek, tedy Zodpovědný Lektor, více učeben. Navíc jeden lektor může být ZodLek, tedy Zodpovědný Lektor, pro více kurzů.

Pokud je doopravdy zákazník ochotný slyšet, co jste vymysleli, překvapte jej takovýmto diagramem a výkladem. Pokud je vše v pořádku, můžete začít s SQL

SQL DDL navržení databáze

Teď už diagramy převedeme do databáze. Nejdříve zjistěte, jak se bude databáze jmenovat. Myslete na to, že by měl být název naprosto věrohodný. Podle databázového stroje vložte novou databázi. Příkladem vytvoříme určitě databázi takto:

```
CREATE DATABASE SkoliciStredisko;
```

Stále se vyhýbáme použití háčků a čárek, ty si dovolíme až v databázi samotné, tedy ve vkládání záznamů. Přechod do databáze zajistíme příkazem USE pro MySQL, či jiným, dle databáze.

Následně začneme vytvářet jednu tabulku za druhou a dáváme pozor, zdali se tabulky shodují s diagramem. Vytváříme také odpovídající datové typy.

```
CREATE TABLE STUDENT
(
    ID INT (10) PRIMARY KEY AUTO_INCREMENT,
    Jmeno VARCHAR(100) NOT NULL,
    Prijmeni VARCHAR(100) NOT NULL
);
```

Už víme, že AUTO_INCREMENT neplatí všude, ale pro ilustraci jej použijeme, v jiné databázi bychom museli najít jiný ekvivalent tomuto výrazu.

```
CREATE TABLE STUD_ADR
(
    ID INT (10) AUTO_INCREMENT PRIMARY KEY,
    Mesto VARCHAR(100) NOT NULL,
    Ulice VARCHAR(100) NOT NULL,
    PSC VARCHAR(100) NOT NULL,
    Tel TEXT NOT NULL,
    Vek INT(3) NOT NULL,
    Pohlavi VARCHAR(10) NOT NULL
```

);

Pokud se divíte, že všude vkládáme NOT NULL, myslete na to, že záznam, který není konzistentní, může vypadat velice podivně, a proto právě NOT NULL, které zaručí, že záznam bude muset mít vyplněna nějakým způsobem nulová pole.

Další poznámka je nutná k Tel, tedy telefonnímu číslo, protože je zapsáno datovým typem TEXT(). Datový typ TEXT() zaručí malou ochranu při nějakém nedovoleném nahlížení do databáze, ale také je mnohem jasnější, že nebudeme pracovat s telefonními čísly jako s čísly, tedy je určitě nebudeme sčítat a odčítat.

```
CREATE TABLE LEKTOR
(
    ID INT (10) PRIMARY KEY AUTO_INCREMENT,
    Jmeno VARCHAR(100) NOT NULL,
    Prijmeni VARCHAR(100) NOT NULL
);
```

Stále nedefinujeme vazby, i když v některých databázích můžeme. Držme se toho, že vazba se vytvoří v dotazu, tedy tak, jak podporují určitě všechny databáze.

```
CREATE TABLE LEKT_ADR
(
    ID INT (10) PRIMARY KEY AUTO_INCREMENT,
    Mesto VARCHAR(100) NOT NULL,
    Ulice VARCHAR(100) NOT NULL,
    PSC VARCHAR(100) NOT NULL,
    Tel TEXT NOT NULL,
    Vek INT(3) NOT NULL,
    Pohlavi INT(10) NOT NULL
);
```

Zde vidíme další tabulku z vazby jedna ku jedné.

```
CREATE TABLE INDEXKARTA
(
    ID INT (10) PRIMARY KEY AUTO_INCREMENT,
    Student INT (10) NOT NULL,
    Lektor INT (10) NOT NULL,
    Ucebna INT (10) NOT NULL,
    Kurz INT (10) NOT NULL,
    Vysledek INT(4) NOT NULL
);
```

Tabulka INDEXKARTA pak nebude mít žádnou zmínku o cizích klíčích, ale přesto na ně bude obsahovat pole. Podle toho bude taky vypadat možnost přidání záznamů, kterou budou muset ohlídat programátoři nějakého aplikačního rozhraní. V případě Access a OpenOffice bychom se potýkali s formuláři a sestavami.

```
CREATE TABLE UCEBNY
(
    ID INT (10) PRIMARY KEY AUTO_INCREMENT,
    ZodLek INT (10) NOT NULL,
```

```
    Patro INT (2) NOT NULL,
    HrUmist VARCHAR (256) NOT NULL,
    PocMist INT (3) NOT NULL,
    Popis TEXT NOT NULL
);
```

Další tabulky vytvoříme dle již známých příkazů.

```
CREATE TABLE KURZY
(
    ID INT (10) PRIMARY KEY AUTO_INCREMENT,
    ZodLek INT (10) NOT NULL,
    Cena INT (30) NOT NULL,
    Jmeno VARCHAR (256) NOT NULL,
    Popis TEXT NOT NULL,
    PocetHod INT (8) NOT NULL
);
```

Právě jsme vytvořili potřebné tabulky pro celou databázi. Nepracovali jsme s ničím tak složitým, aby to nešlo více rozšířit a zapracovat na tom, ale postavili jsme jednoduchý základ.

Jak dál upravit databázi

Nejdříve si položme otázku, se kterou tabulkou budou programátoři asi nejvíce pracovat, a podle toho musíme připravit dotazy. Je těžké říci, které příkazy přímo použijí, a proto zákazníka spíše přemluvte k delší spolupráci, tím se tedy snažte být více napřed a na databázi zaberte rovnou místo administrátora. Pomocí GRANT připravte uživatele, který bude programátor. Rozhodně omezte přístup k ovládání databáze z jazyka DDL.

CREATE USER programator IDENTIFIED BY ‘pass‘ PASSWORD EXPIRE; Pro ORACLE, kde chceme hned změnit heslo, ale to můžeme udělat pouze přes APEX.

CREATE USER ‘programator‘ IDENTIFIED BY ‘pass‘; Pro kteroukoliv jinou databázi.

Následně programátorům upravíme možnost pracovat jen s DML příkazy. Pak se nemusíme bát, že omylem programátoři upraví databázi a ta se stane chybnou.

GRANT SELECT,UPDATE,INSERT,DELETE ON *.* TO ‘programator‘;

Tím jsme stanovili práva na naší databázi (pomocí tečky). Nadále práva můžeme upravovat dle potřeby a programátoři nás určitě budou dále zásobovat prosbami na změny v databázi, což nás dělá neustálými zaměstnanci až do té doby, než se databáze vyladí dle potřeb zákazníka.

Závěr

Závěrem bychom mohli shrnout, co je potřeba znát. Nesmíme mluvit na zákazníka jazykem SQL, ani jiným, počítačovým, kterému nerozumí, protože prostě není zvědavý na to, jak to provedete, ale jak to bude vypadat. Což je pro nás veliký problém, protože my, jako SQL programátoři, těžko vytvoříme grafické rozhraní, proto se už od začátku snažte najít a kontaktovat programátory, kteří budou pracovat s vaší databází dál.

Snažte se vše vyjádřit diagramy, aby byla databáze přehledná a snadno rozšiřitelná. Vytvořte uživatele, kteří budou pracovat s databází, a v případě potřeba vysvětlete a načrtněte, jak se mají programátoři starat o příkazy, jako jsou SELECT, UPDATE, INSERT a DELETE. Snažte se být při ruce. Navíc, pokud jste postupovali, jak je nastíněno výše, buďte administrátorem až do vyladění databáze, pak klidně předejte jméno a heslo administrátora dál a už se o databázi nestarejte. Neustále se snažte zákazníkovi vysvětlit, že zajetí databáze chvíli trvá a že mu nedokážete objektivně vytvořit databázi na papír, pokud nebudete znát, na jakém databázovém programu bude běžet.

V případě, že se dozvíte, na jakém programu máte databázi stavět, prostudujte si kontingenční práva, jako jsou CHECK a REFERENCE pro cizí klíče. Snažte se případně doporučit databázi, se kterou

máte zkušenosti.

Cvičení

- Pomocí typických databázových klauzulí označte primární klíče a cizí klíče (pokud najdete vhodnou variantu pro vaši databázi).
- Vymyslete INNER JOIN příkaz pro výběr z dvou a více tabulek v databázi.
- Nakreslete diagram spoje mnoha ku mnoha z databáze.
- Zopakujte si normalizační zákony a zkuste najít chyby v databázi, následně je odstraňte, či vymyslete, kdy mohou chyby nastat.
- Navrhněte, jak by se mohla databáze dále rozvíjet a předělejte podle toho diagram.

Závěr

Pokud jste přímo nepochopili, jak na programy okolo SQL, nemusíte se strachovat, protože ty se dají najít všude a k nim je většinou i velice silná zákaznická služba, která vám poradí. U otevřených databází je to fórum, u uzavřených je to výrobce, kdo vám poradí.

Pokud vám dělali problém některé SQL příkazy, pamatujte, kde jste o nich četli a v případě potřeby je vyhledejte a použijte. Není nic horšího, než se splést, protože jste v dané chvíli měli určitě jistotu, že to bude fungovat.

Spíše tuto knihu berte jako malý úvod do světa databází a SQL, kde je nutné se pomalu rozkoukávat a stále znalosti prohlubovat.

Pokud jste nenašli správné použití nějaké klauzule, či vám nefungovala tak, jak jste si mysleli, zkuste vyhledat pomoc komunity okolo dané databáze, protože je veliký rozdíl mezi databází ORACLE a databází Access ze sady Microsoft Office. Navíc, je možné, že popsaná klauzule nepatří zrovna do databáze, kterou používáte. ORACLE má třeba jiné funkce pro práci s daty než má Mysql.

Přesto by vám tento malý průvodce měl poradit s běžnými dotazy na databázi a neměli by vás překvapit ani vazby JOIN, či agregační funkce.

Rejstříky

Použitá literatura

Ing.Ivanka Koníčková, I. Y. (2006). *Databáze.* Praha: Střední smíchovská průmyslová škola.
James R. Groff, P. N. (2009). *SQL - Kompletní průvodce.* CPRESS.
Komunita. (19. 3 2010). *www.wikipedia.org*. Získáno 8. 5 2009, z Wikipedia.org: http://www.wikipedia.org
Procházka, D. (2009). *ORACLE průvodce správou, využitím a programováním nad databázovým systémem.* Havlíčkův Brod: GRADA Publishing a.s.
Spurná, I. (25. 6 2006). *www.ivasp.info*. Získáno 4. 5 2010, z ivasp: http://www.ivasp.info