PostgreSQL

Instalace PostgreSQL

V této části vám ukáži, jak naistalovat a spustit různé SQL databáze. Návody nejsou kompletní kuchařky popisující instalaci krok za krokem. V každém operačním systému se totiž postup trochu liší a s novými verzemi se může lišit ještě víc. Ale nebojte, nejde o nic složitého a určitě tu najdete dostatek informací k tomu, abyste to zvládli vlastními silami.

Instalace PostgreSQL

Nemáte-li Postgres nainstalovaný, nainstalujte jej, nebo požádejte někoho, kdo to umí :).

Každá dobrá linuxová distribuce má Postgres v repozitářích. Uživatelé Windows si můžou Postgres stáhnout a nainstalovat.

Vyberte si nejnovější verzi, x86-32 pokud máte 32-bitové Windows, x86-64 pro 64-bitové Windows.

Předpokládám, že když se chcete učit databáze, už to umíte s počítači natolik dobře, že stáhnout a spustit instalační soubor sami hravě zvládnete :)

Když instalujete z repozitářů (linuxový uživatelé), je důležité vědět, co vlastně potřebujete. Databáze totiž většinou fungují na principu **klient – server**. Server a klient jsou dva různé programy (software).

Server se stará o práci s daty a klient slouží ke komunikaci se serverem. Server i klient jsou často nainstalováni na jiném počítači, ale vy si oba můžete nainstalovat k sobě. Možná že máte přístup na školní server, takže by vám stačil jen klient. Ale s nainstalovaným serverem u sebe si užijete více legrace (budete mít superuživatelská práva, máte server po ruce i bez interentového spojení atd.).

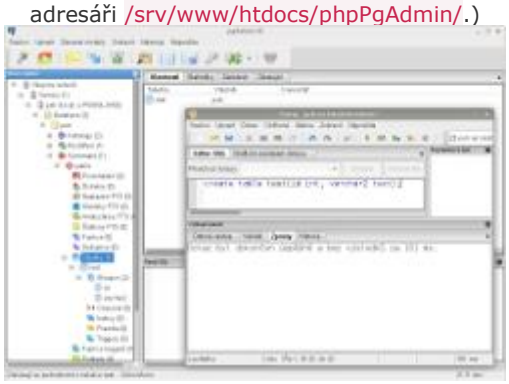
V OpenSuSE jsem instaloval balíčky **postgresql-server** a **postgresql**. V jiných linuxových distribucích se budou balíčky jmenovat určitě nějak podobně.

Balíček postgresql obsahuje klienta, který pracuje v příkazové řádce. Určitě vám doporučuji, abyste celý kurz absolvovali s tímto klientem! Jeho rozhraní popíši hned v následující kapitole. A proč je tento klient nejlepší volba?

1. Je rychlý. Když budete pracovat s databázemi často, často budete klienta spouštět a vypínat. V příkazové řádce je to otázka zlomků sekund, u grafického klienta to trvá a trvá.
2. Často pracujete s databází na vzdáleném počítači. Někdy se knému můžete připojit přímo z vašeho počítače, to umožňuje každý klient, ale někdy se musíte nejdříve připojit na vzdálený počítač (například protokolem SSH) a pak vám stejně nezbyde než pustit klienta z příkazové řádky.
3. Práce s textovým klientem je rychlejší.
4. Máte k dispozici příkazy pro nápovědu.
5. Budu popisovat pouze práci s textovým klientem.

Nebudu vám ale zatajovat, že klientů existuje víc. Pokud rozumíte Apache + PHP, můžete si nainstalovat **phpPgAdmin**. Jedná se o webové stránky, které slouží k administraci Postgres databáze. Když budete někdy používat postgres databázi na nějakém web hostingu, určitě to budete dělat přes phpPgAdmin.

Nainstalovanou aplikaci najdete na svém počítači na adrese http://localhost/phpPgAdmin/. (zdrojové soubory jsou v adresáři /srv/www/htdocs/phpPgAdmin/.)



pgAdmin III

Jestli trváte na grafickém klientovi, nainstalujte si balíček **pgadmin3**. (Při přihlášení nechte kolonku Počítač prázdnou (nedávejte localhost ani 127.0.0.1), jinak se bude pgadmin3 pokoušet přihlásit přes TCP/IP a to je obvykle zakázané.)

Uživatelé Windows si jej mohou stáhnout ze stránkywww.pgadmin.org/.

Ujistěte se, že **server běží** a že se spouští automaticky po startu systému, nebo že víte, jak jej spustit.

V OpenSuSE se po instalaci balíčku **postgresql-server** serverver automaticky nespouští. Spusťte si YaST, v části Systém spusťte Editor úrovní běhu, najděte postgresql a tlačítkem Povolit povolte jeho spouštění při startu.

Server běží jako tzv. deamon (ve Windows se tomu říká systémová služba, tuším). Server uvidíte ve výpisu procesů (např. ps aux | egrep postgres), ale jinak nemá žádné rozhraní (grafické ani textové), se kterým byste mohli komunikovat. Komunikuje se sním výhradně pomocí nějakého klienta, maixmálně ještě zasíláním signálů (uživatelé Linuxu jistě tuší, o čem mluvím).

Server samozřejmě čerpá nějaké prostředky vašeho počítače (paměť, procesor), takže jej možná nebudete chtít spouštět automaticky. V Linuxu máte několik možností, jak takový server spustit, vypnout či zapnout/vypnout jeho automatické spuštění. To se může hodně lišit distribuce od distribuce, tak jenom ukáži několik možností v OpenSuSE (většinu příkazů musíte spouštět jako root).

```
$ sudo /etc/init.d/postgresql start   # spusti server
$ sudo /etc/init.d/postgresql stop    # vypne server
$ sudo /etc/init.d/postgresql restart # restartuje server (načte znovu konfiguraci)

$ sudo systemctl stop  postgresql     # vypne server (modernejsi zpusob nez ten nahore)
$ sudo systemctl start postgresql     # spusti server
$ systemctl status postgresql         # vypise status
postgresql.service - LSB: Start the PostgreSQL master daemon
Loaded: loaded (/etc/init.d/postgresql)
Active: active (exited) since Wed, 2013-11-20 11:46:25 CET; 4s ago
Process: 29152 ExecStop=/etc/init.d/postgresql stop (code=exited, status=0/SUCCESS)
Process: 29252 ExecStart=/etc/init.d/postgresql start (code=exited, status=0/SUCCESS)
CGroup: name=systemd:/system/postgresql.service
```

Vypíše-li vám poslední příkaz řádku **Active: active (exited)...** (tak jako v příkladu), máte vyhráno!

Uživatelé Windows by se měli podívat na to, jak nastavit PATH, aby mohli snadno spouštět programy, jako je třeba klient psql.

Vytvoření uživatele

Po čerstvé instalaci PostgreSQL musíte vytvořit uživatele, který bude mít právo vytvářet a mazat databáze (popřípadě další uživatele, kteří budou mít právo do databází "jen" zapisovat).

PostgreSQL (na rozdíl od MySQL) spojuje uživatelské účty s těmi z operačního sytému. Můžete se sice připojit k databázi i pod jiným uživatelským jménem, ale chce to nějaká práva navíc. Nejjednodušší bude, když se budete držet stejného jména pro přístup k databázi jako je uživatelské jméno uživatele Linuxu.

Následující příklad je z operačního systému Linux. Pro uživatele Windodws návod (zatím?) bohužel nemám.

Řekněme, že máte v Linuxu uživatelský účet s loginem rimmer a chcete, abyste s tímto loginem mohli nejen pracovat s obsahem databáze, ale chcete mít i právo další databáze vytvářet a mazat. Pak postupujte následovně:

Poznámka:
***$** uvozuje příkazy, které se zapisují do příkazové řádky*
(ve Windows v okně programu command.com nebo cmd).
***=>** uvozuje příkazy, které se zapisují v programu psql (což je textový klient).*

1. Přihlašte se jako root.

```
$ su - root
```

2. Nyní se přihlaste jako uživatel postrgres. (Jelikož jste přihlášen jako root, nebude po vás vyžadováno heslo.) Uživatel postgres je vytvořen automaticky během instalace databáze a má neomezená práva.

```
$ su - postgres
```

3. Spusťte program psql. Program psql je textový klient, který se přihlásí k vaší lokální databázi PostgreSQL (pokud mu nepředáte žádné parametry). Uživatel postgres, pod kterým byste měli teď být přihlášeni, nevyžaduje žádné heslo.

```
$ psql
```

4. Vytvořte uživatelské konto pro uživatele (například rimmer) (nezapomeňte na středník na konci):

```
=> CREATE USER rimmer WITH CREATEDB;
CREATE ROLE
```

Všimněte si, že Postgres vypsal po zadání příkaz CREATE ROLE. Postgres má i příkaz CREATE ROLE, který se chová podobně, ale není to to samé co CREATE USER! Podrobnosti popíši v některé z příštích kapitol.

Pokud nechcete, aby uživatel rimmer mohl vytvářet a mazat databáze, zadejte místo předchozího příkazu jen:

```
=> CREATE USER rimmer;
CREATE ROLE
```

5. Opakujte krok 4. pro další uživatele.
6. Odhlašte se z databáze (klávesovou kombinací CTRL+d, nebo příkazem \q).
7. Odhlaste se z uživatelského účtu postgres i root. Přihlašte se v Linuxu jako uživatel rimmer.

*Jinou cestou, jak vytvořit uživatele, je použití programu **createuser** (v příkazové řádce shellu). Musíte být zase přihlášeni jako uživatel postgres.*

Přepínače tohoto programu získáte pomocí příkazu createuser --help nebo v manuálové stránce (důležitý přepínač je například pro uložení hesla v šifrované podobě).

Vytvoření databáze

Server už běží, vytvořili jste si už i uživatele s právem vytvářet databáze, takže zbývá už jen nějakou databázi vytvořit.

K vytváření databází slouží příkaz **createdb**. Má několik zajímavých voleb. (Všechny si můžete najít v manuálové stránce.) Za zmínku stojí určitě přepínač -E, kterým můžete určit defaultní kódování databáze, -T pro určení „template" databáze, --lc-collate pro určení jak se mají texty v databázi třídit a --lc-ctypepro určení klasifikace znaků. (Všechno bude podrobně vysvětleno někdy později).

Defaultně se pro kódování použije to kódování, které bylo vybráno při instalaci Postgresu. Přepínačem -Emůžete nastavit libovolné (Postgresem podporované) kódování, například LATIN2 (vhodné pro středoevropské jazyky v Linuxu), WIN1250 (středoevropské jakzyky ve Windows) nebo UTF8. ~~Kódování UTF8 je vhodné, pokud budete data přenášet mezi různorodými systémy, jinak bych doporučoval se držet spíše starého dobrého kódování LATIN2.~~

UTF-8 sice zabírá více místa a práce s ním je o ždibec pomalejší, ale zato umožňuje ukládat text v mnoha jazykových sadách (takže není problém uložit text v češtině i azbuce). Dnes je běžné mít např. webové stránky v různých jazycích, takže UTF8 je jasnou volbou.

V kompletním seznamu podporovaných kódování najdte další informace. Pro tuto chvíli bude stačit, když mi budete věřit, že UTF-8 je váš favorit. Totéž platí o výběru správného collate a ctype.

```
$ # createdb rimmer
$ createdb -E UTF8 -T template0 --lc-collate=cs_CZ.UTF-8 --lc-ctype=cs_CZ.UTF-8 rimmer
$ psql rimmer
psql (9.4.9)
Pro získání nápovědy napište "help".

rimmer=> \q
$
```

Zadávat jméno databáze při spouštění psql není nutné, pokud se přihlašujete k databázi stejného jména, jako je vaše uživatelské jméno.

Chcete-li vidět všechny existující databáse, použijte příkaz **psql -l**:

```
$ psql -l
                                  Seznam databází
   Jméno    | Vlastník | Kódování |  Collation  |    CType    |  Přístupová práva
------------+----------+----------+-------------+-------------+-----------------------
 postgres   | postgres | UTF8     | cs_CZ.UTF-8 | cs_CZ.UTF-8 |
 rimmer     | petr     | UTF8     | cs_CZ.UTF-8 | cs_CZ.UTF-8 |
 template0  | postgres | UTF8     | cs_CZ.UTF-8 | cs_CZ.UTF-8 | =c/postgres          +
            |          |          |             |             | postgres=CTc/postgres
```

Není nutné, abyste úplně chápali vše předešlé. Ale v dalším textu předpokládám, že již máte nainstalovaný PostgreSQL, vytvořenou databázi a získali jste příslušná přístupová práva, abyste si mohli vyzkoušet všechny příklady.

Instalace MySQL / MariaDB

Databáza MariaDB je komunitní projekt, který se oddělil od MySQl poté, co MySQL koupil Oracle a začal Mysql „dusit". První verze byla téměř totožnou kopií MySQL. Aktuální verze MariaDB 10 se už však začíná od mysql mírně lyšit (je jasný, že vývojáři MariaDB nebudou jen opisovat novinky z MySQL co vymyslí Oracle, ale chtějí přidat něco „vlastního"). Stále ale platí, že jsou tyto dvě databáze kompatibilní, téměř jedno jsou.

Tento seriál byl původně psán pro MySQL, avšak jak MariaDB přebírá otěže, budu se postupně orientovat právě na tuto databázi. V následujících kapitolách ale můžete MySQL a MariaDB brát jako synonyma, nebudu popisovat nic, co by nefungovalo v obou databázích stejně. Doporučuji vám používat communitní MariaDB.

Instalace MySQL je hodně podobná PostgreSQL. Takže jen ve stručnosti: nainstalujte si balíčky **mysql-community-server** a **mysql-community-server-client** (nebo **mysql-server** a **mysql-client**), resp.**mariadb** a **mariadb-client**. ujistěte se, že je server spuštěn, případně si nainstujte webového klienta**phpMyAdmin**, nebo grafického klienta **mysql-workbench**. (Mimochodem, mysql-workbench je o mnoho lepší než pgadmin, ale i tak vám doporučuji začít s textovým klientem mysql).

Protože se snaží MariaDB nahradit MySQL, po jejím nainstalování máte k dispozici program mysql, který ve skutečnosti spouští klienta MariaDB. I další programy z instalace MariaDB mají stejné názvy jako z MySQL.

Vytvoření databáze a uživatele

Při instalaci MySQL je vytvořen uživatel root, který má všechny práva. V MySQL nemá uživatel databáze žádnou spojitost s uživateli Linuxu, takže se může každý připojit jako root. Po instalaci nemá root (MySQL uživatel root) nastavené žádné heslo, takže se může k serveru přihlásit s tímto supermocným uživatelem každý, kdo se naloguje do počítače (je možné se připojit pouze z localhostu). Je velmi záhodno nastavit heslo uživateli root jak rychle to jen jde, ale pokud máte k počítači přístup jenom vy, tak to není zase tak horké.

Databázi a uživatele vytvoříte pomocí SQL příkazů skrze klienta a účtu root:

```
$ mysql -u root
mysql> CREATE DATABASE rimmer DEFAULT CHARSET UTF8 COLLATE utf8_czech_ci;
mysql> GRANT ALL PRIVILEGES ON rimmer.* TO rimmer@localhost;
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| rimmer             |
| test               |
+--------------------+
7 rows in set (0.00 sec)

mysql> quit
$
```

Příkazy v příkladu vytvoří databázi rimmer s defaultním kódováním UTF8 a tříděním dle českého jazyka (utf8_czech_ci). Dále vytvoří uživatele s loginem rimmer (bez hesla), který se může přihlásit k databázi z localhostu a rovnou mu nastaví všechny práva (ALL PRIVILEGES) ke všem tabulkám v databázi rimmer (i těm tabulkám, co teprve budou v databázi rimmer vytvořeny).

Nyní se můžete přihlásit k databázi rimmer jako uživatel rimmer z localhostu.

```
rimmer $ mysql rimmer -u rimmer -h localhost
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 38
Server version: 10.0.20-MariaDB openSUSE package

Copyright (c) 2000, 2015, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql:rimmer>  quit
Bye
rimmer $
```

Client mysql se snaží uhodnout jméno podle uživatelského jména Linuxu, takže pokud je stejné jako jméno databázového uživatele, nemusíte jej při spuštění clienta uvádět. Server localhost je také defaultní, takže ani to nemusíte uvádět. Stačilo by tedy pouhé mysql rimmer.

Adresa serveru která je součástí jména uživatele určuje, odkud se může uživatel přihlásit. Server **localhost** a server **127.0.0.1** nejsou pro MySQL to samé! Takže když založíte uživatele rimmer@localhost, nemůžete se přihlásit příkazem mysql -u rimmer -h 127.0.0.1.

Instalace SQLite

SQLite je trochu zvláštní databáze. Je to program, který má v sobě jakoby server i klienta. Jako databázi používá libovolný soubor, který mu určíte. Když neexistuje, vytvoří si jej, když existuje, pokusí se s ním pracovat jako s SQLite databází. Z toho vyplývá, že stačí nainstalovat jeden balíček – **sqlite3**. (Nejspíš najdete ve své distribuci ještě starší verzi sqlite2, s tou neztrácejte čas).

Uživatelé Windows is mohou stáhnout Precompiled Binaries for Windows. Dále vás odkážu zase na to, jak nastavit PATH, SQLite neřeší žádné uživatele, buď máte přístup k souboru, nebo nemáte.

```
$ sqlite3 rimmer.db3
SQLite version 3.7.13 2012-06-11 02:05:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .quit
$
```

Pokud nevytvoříte žádnou tabulku, nevytvoří se ani soubor rimmer.db3. Jak se tabulky vytvářejí budu probírat později.

Instalace Oracle XE

Nejtěžší nakonec. Oracle databázi nenajdete ve svých repozitářích. Budete si jí muset stáhnout ze stránek Oracle. (Abyste to mohli udělat, tak se budete muset nejdříve (zdarma) zaregistrovat).

Pokud se vám nechce nebo nepovede Oracle XE instalovat, můžete si zkusit online verzi. Určitě ale doporučuji nejdříve instalaci zkusit, není nad to moci pracovat na svém PC (nemluvě o možnosti využitíOracle SQL Developeru, viz dále).

Oracle databáze je velmi, velmi drahá. Naštěstí existuje **Express Edition**, která slouží pro výuku a je zdarma. Má nějaká omezení (max. možný počet připojení k databázi, max. velikost dat v databázi atp.), ale ty vás při výuce nemusí trápit. Aktuálně si můžete stáhnout XE verzi pro Windows 32 bit /64 bit, nebo Linux 64 bit. A protože máte zaručeně Linux 32 bit, mám pro vás jednu vychytralou radu: použijte VirutalBox.

K dispozici je verze Oracle Database Express Edition 11g Release 2, ačkoliv už existuje placená verze Oracle Database 12c Release 1, Express Edition 12c nejspíš nevyjde dříve, než Oracle Database 12c Release 2.

Já používám jako OS 32 bit Debian. To mi ale nezabrání v tom, abych nemohl nainstalovat do VirtualBoxu 64 bit OpenSuSE 12.3 a v něm 64 bitovou verzi Oracle XE pro Linux.

Oracle XE vyžaduje, abyste měli v počítači alespoň 1500 MiB paměti RAM.

Po stažení instalačního balíčku rozbalte tento ZIP balíček, přejděte do adresáre Disk1 a nainstalujte jej:

```
$ rpm -i oracle-xe-11.2.0-1.0.x86_64.rpm
```

Během instalace budete dotazováni na pár věcí. Ponechte defaultní volby kde to jde. Dejte si jen pozor, aby jste vybrali spouštění Oracle databáze při startu systému. Budete také požádáni o zadání hesla pro systémového uživatele SYSTEM, to rozodně nezapomeňte, jinak budete v háji.

Po instalaci Oracle musíte přidat následující řádku do /etc/hosts, jinak se Oracle odmítne spustit a vůbec neřekne proč:

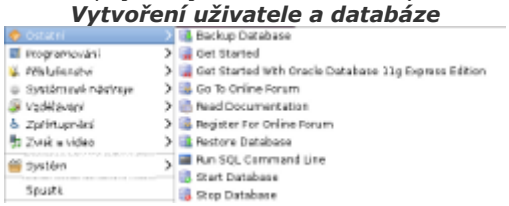
```
127.0.0.0 nazev_pocitace
```

Název počítače získáte příkazem uname -n.

Oracle XE nainstaluje jak klienta, tak server a k tomu ještě webovou aplikace **apex**. To je moc pěkné webové rozhraní pro sledování serveru, vytvoření nového uživatele, přihlášení se k databázi a k práci s ní. **Nedoporučuji používat textového klient oracle.** Já vím, že jsem to do teď doporučoval u všech databází, ale Oracle přestal aktualizovat textového klienta asi tak někdy v době temna a podle toho to vypadá. Když uděláte nějakou chybu, hlásí nepochopitelné hlášky, když se někde překlepnete, musíte celý příkaz psát znovu (žádná historie) atd.

Určitě se vyplatí, když se nejdříve seznámíte s apexem. Později ale můžete přejít na příjemnějšího grafického klienta – SQL Developer. (Tento klient vám sice neprozradí nic o stavu serveru jako **apex**, ale přeci jen je práce v desktopové aplikaci příjemnější než ve webové …)

Vytvoření uživatele a databáze



Oracle menu

Po instalaci Oracle najdete v menu aplikací několik Oracle zástupců. Ne všichni bohužel pracují jak mají. Například start a stop databáze nefunguje, nejspíš proto, že jsou k tomu potřeba práva uživatele root. Spustit příkazovou řádku se mi z menu taky nepovedlo, protože zástupce odkazuje na skript, ve kterém je chyba. (Pokud po konzoli opravdu toužíte, spustit jí můžete příkazem/u01/app/oracle/product/11.2.0/xe/config/scripts/sqlplus.sh. Ale možná budete mít štěstí a vám ten zástupce fungovat bude.)

Apex

Nejdůležitější je zástupce na spuštění apexu Get Started With Oracle Database 11g Expres Edition. Ten naštěstí funguje. Spustí webový prohlížeč a otevře stránku localhost:8080/apex/. Měli byste vidět stránku jako na obrázku *Apex*.

Vytvoření uživatele a databáze

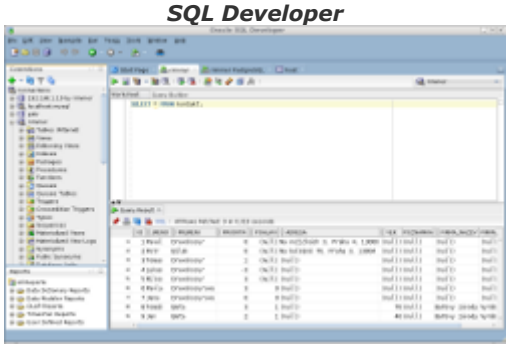
Přejděte do Application Express, kde si založíte uživatele s databází (tady se tomu říká workspace). Nejdříve budete dotázání na jméno (to je SYSTEM) a heslo (to je to, co jste vyplnili při instalaci). Pak byste se měli dostat na stránku jako je na obrázku *Vytvoření uživatele a databáze*.

Database user name a *Application user name* vyplňte stejně, ať se vám to neplete. Workspace se vytvoří automaticky se stejným jménem, jako *Database user name*.

Apex domovská stránka

Po vytvoření uživatele a jeho workspace se můžete přihlásit. Všimněte si na obrázku *Vytvoření uživatele a databáze* v pravo tlačítka Already have an account? …. Kliknutím na něj se dostanete na přihlašovací obrazovku pro uživatele.
Na stejnou obrazovku se dostanete, pokud napíšete do prohlížeče přímo adresu localhost:8080/apex/, takže se nemusíte do apexu dostávat tak krkolomě přes přihlášení uživetele SYSTEM …
Po přihlášení byste měli vidět obrazovku, jako je na obrázku *Apex domovská stránka*. Klikněte na SQL Workshop, pak na SQL Commands a můžete psát své SQL příkazy.
V textovém klientovi se přihlásíte (po spuštění sqlplus.sh skriptu viz výše) k databázi příkazem connect username/password. Ale jak už jsem psal, moc to nedoporučuji.

SQL Developer



SQL Developer

Oracle SQL Developer je grafické rozhraní určené primárně pro práci s Oracle Databází. Podporuje i práci s MySQL a od verze 4 i s Postgresem.
Aby vám fungovalo připojení do Postgresu, musíte nejdříve stáhnout JDBC driver pro Postgres. Pozor na verzi JDBC driveru. Záleží na verzi javy, kterou máte nainstalovanou. Vzhledem k tomu, že SQL DEveloper v 4 vyžaduje javu 1.8, budete potřebovat JDBC42 Postgresql Driver, Version 9.4-1203. Tedy tak to alespoň platí v době pasní tohoto odstavce.
Dále v SQL Developeru vyberete Tools → Preferences, Database → Third Party JDBC Drivers a pak přidáte stažený jar soubor.
Pak už můžete vytvořit nové databázové připojení a vybrat si záložku PostgreSQL.
Stejný postup platí i pro MySQL/MariaDB, jen stáhnete MySQL JDBC Driver.
Budete se muset přihlásit Oracle účtem a vyplnit nějaký otravný dotazník, ale je to zadarmo.
PostgreSQL připojení mi v určitých případech nezobrazuje seznam tabulek, pohledů atp. Mohu jen pouštět SQL příkazy. Myslím, že to má něco společného s nastavením práv, ale zatím jsem nepřišel na to, co. Pro MySQL funguje bezvadně. Pro SQLite neexistuje v SQL Developeru podpora.

Závěr

Hurá, to nejhorší máte zasebou. Když už máte nainstalovanou databázi a víte jak se k ní připojit, nemůže vás již nic zastavit od toho stát se SQL guru. Smyslem této kapitoly nebylo abyste něco pochopili, to příjde v dalších kapitolách. Mimo jiné jsem zde používal zmateně pojem „databáze". Co toto slovo vlastně znamená se dozvíte v další kapitole.

Úvod do jazyka SQL

Než začnu s popisem SQL jazyka a PostgreSQL, měl bych vás seznámit se základními pojmy, které se v databázích používají. Bez jejich znalosti se mezi databázisty nedorozumíte a nerozuměli byste ani tomuto tutoriálu.

- Základní pojmy
- Structured Query Language
  - ANSI/ISO SQL standard
- DBMS
  - PostgreSQL
  - MySQL a MariaDB
  - SQLite
  - Oracle

Základní pojmy

PostgreSQL

(dříve zvaná Postgres) je relační databáze fungující na bázi server – klient (jako většina databází). Na první pohled to vypadá složitě, ono kolem databází taky vznikla celá věda, ale pro domácí použití vám bude stačit pochopit pár následujících pojmů. Pokud je nepochopíte všechny hned, nevadí, jejich význam vysvětlím na příkladech v dalších kapitolách.

Databáze

je nějaký soubor uložený kdesi na disku. Struktura tohoto souboru je tvořena **tabulkami** (a relacemi mezi tabulkami). Tabulky obsahují **řádky** a **sloupce**.

Například můžete vytvořit tabulku se jménem telefonni_seznam, která bude obsahovat dva sloupce se jmény jmeno a telefon. Řádky pak budou tvořit jednotlivé záznamy v tabulce (Kolik budete mít záznamů jmeno + telefon, tolik budete mít řádků).

telefonni_seznam

jmeno	telefon
James Bond	02/12345
Karel IV.	02/012345

Jako databázi můžete považovat klidně i soubor Excelu. Ale taky můžete mít dočasnou databázi jen v paměti počítače.
Databáze je také označení pro program, který pracuje s databázovým souborem. PostgreSQL je jedním takovým programem (o dalších se dočtete dále). V tomto smyslu se o databázi hovoří jako o DBMS (viz dále).
Databáze nemusí mít vůbec nic společného s počítači. Databází je i váš papírový telefonní seznam.
Databáze je označení pro model dat (jen jakousi ideu, jak se budou data ukládat a v jakém budou vztahu).
Databázový program často pracuje s více než jedním databázovým souborem. Všechny soubory dohromady (na jednom počítači) se pak označují jako databáze, resp. **catalog** a jednotlivé „databáze" se označují jako **schema**ta nebo **workspace**. (Databáze může obsahovat několik schémat, která se liší svým jménem a místem, kam se ukládají data).
Jak vidíte, pojem databáze je hodně zneužívaný a tak není občas jasné, co se jím myslí. Většinou to poznáte z kontextu, ve kterém se o „databázi" hovoří.

Cluster, Catalog, Schéma, Tabulka
Definice dle SQL standardu:

- Databázový server je cluster. (DBMS)
- Cluster má catalogy. (Catalog = Database)
- Catalogy obsahují schemata (=workspace).
- Schemata obsahují tabulky.
- Tabulky obsahují řádky.
- Řádky obsahují data, definovaná pomocí sloupečků.

V tomto tutoriálu budete pracovat s jedním DBMS a s jedním clusterem, ze začátku pouze s jedním schématem ve kterém si budete definovat tabulky. Když budu mluvit o databázi, budu mít nejčastěji na mysli DBMS nebo schéma.

SŘBD, DBMS
systém řízení báze dat, database management system – vznešený název pro celý databázový program (např. PostgreSQL). S těmito pojmy se určitě setkáte ve škole, jinak každý říká databáze :-).

Relace
je vztah mezi tabulkami v databázi. Například můžete mít tabulku obsahující názvy hudebních CD a tabulku obsahující jména zpěváků. Vztah, který určí které CD patří kterému zpěváku, se zove relací. Až se budete učit relace používat, pochopíte a osvojíte si je rychle.

Server
Tím se zde myslí program, který se stará o soubory databáze. Zapisuje do nich, čte z nich a vrací data klientovi. Udržuje informace o přístupových právech k databázím a je jediný, kdo s daty fyzicky pracuje.
Server běžně běží na nějakém vzdáleném počítači, kde se data ukládají a k tomuto počítači mívá přístup více klientů skrze internet.
Klient a server můžou být na jednom počítači, což už víte z předchozí kapitoly.
Server běží jako proces na pozadí a vyčkává, dokud se neozve klient. V PostgreSQL se o tyto činnosti stará program postgres, který často běží ve více instancích, aby zvládal obsluhovat více klientů.

Klient
je také program. Klient říká serveru, co má udělat, jaká data z tabulky má vrátit, jaká tam má vložit, jaká smazat atd. **Klient se vůbec nestará o to, jakým způsobem a kde jsou data uložena, komunikuje pouze se serverem** pomocí SQL příkazů.
V PosgreSQL je jeden z klientů program psql. Ale klientem může být také modul do PHP nebo jiných programovacích jazyků, které umožňují komunikovat se serverem.

Primární klíč
je hodnota v sloupci, který obsahuje *jedinečné, unikátní hodnoty* (sloupec primárních klíčů se obvykle jmenuje id – id jako identifikátor). Proč se tomu říká klíč a ne identifikátor zůstává nevyřešenou záhadou (že by to byl klíč k poznání?).
Chcete-li si vést databázi zaměstnanců, může se vám stát, že dva zaměstnanci budou mít stejné jméno. Jak takové zaměstnance od sebe odlišit? Zavedením sloupce primárních klíčů! To může být sloupec s rodnými čísly, nebo prostě jen (jedinečnými) čísly. Každá tabulka, do které se odkazujete z jiné tabulky, by měla mít primární klíč.
Primárním klíčem může být nejen číslo, ale třeba i řetězec (jméno apod.). Musíte mít ovšem jistotu, že nikdy nebudete mít dva lidi se stejným jménem.
Jméno, oproti číslu, by zbytečně zabíralo místo v tabulkách, kde se na tuto hodnotu primárního klíče odkazujete. Číslo vám vždy zabere méně místa, takže se běžně používá jako primární klíč (id) – celé nenulové kladné číslo.
O to, že bude hodnota primárního klíče pro celou tabulku (v rámci sloupce) unikátní, se už postará SŘBD. Vy jen musíte říct, který sloupeček obsahuje primární klíče (o tom jak se to dělá někdy příště).

Zaměstnanci

id	jmeno	plat
1	Leoš Janáček	13000

Zaměstnanci

id	jmeno	plat
2	Tomas Mann	15000
3	Leoš Janáček	20000

Cizí klíč

je hodnota primárního klíče z jiné tabulky (ve sloupci cizích klíčů, například se jménem id_zamestnance).
Řekněme, že tabulka která bude zaznamenávat příchody vašich zaměstnanců, bude obsahovat kromě sloupce s daty příchodů ještě sloupec, do kterého se budou zapisovat primární klíče z tabulky zaměstnanců. (A **relace** je na světě :-)).
Cizí klíče v tomto sloupci se pochopitelně budou opakovat. (Odhadnete, kolik bytů ušetříte tím, že nepoužíváte jako primární klíč jméno? A nejen že tím šetříte místo, ale také zvyšujete rychlost práce s takovouto databází).

Příchody

cas	id_zamestnance
"Pondeli, 07:30"	1
"Utery, 07:00"	2
"Utery, 07:30"	1
"Utery, 08:30"	3

Pokud zvládnete říci, ve které dny a v kolik přišel do práce Leoš Janáček co bere 13000, gratuluji, můžete svůj mozek označit za relační :-).

Structured Query Language

Nyní už víte, co je to ta relační databáze, a zbývá jen říct, co je to to záhadné **SQL**.
Jak již bylo řečeno, máme tu serverovou aplikaci, která se stará o databáze a mnoho klientských aplikací, které z jedné strany komunikují se serverovou aplikací a z druhé strany s vámi. SQL je tzv. **dotazovací jazyk**, který slouží k ovládání relačních databází. Tímto jazykem se server domlouvá s klientem. To zaručuje jistou hardwarovou i softwarovou nezávislost, pokud doručíte serveru požadavek v jazyce SQL, bude mu rozumět a je mu jedno, od koho ho dostal.
Ukázka SQL jazyka:

```
SELECT * FROM phone WHERE name = 'Sinead' and surname = 'O\'Connor';
```

Jak vidíte, SQL je podobné anglickému jazyku. Příklad říká: Vyber všechno z tabulky phone kde je jméno Sinead a příjmení O'Connor. (Všimněte si, že se textové hodnoty uzavírají do uvozovek. A pokud text sám obsahuje uvozovku, musí se před ní napsat zpětné lomítko.)

ANSI/ISO SQL standard

Existuje mnoho firem, které vytvářejí aplikace na bázi jazyka SQL (PostgreSQL, MySQL, Oracle atd.). Ačkoli je tu snaha o vytvoření univerzálního jazyka a vytvářejí se normy jazyka SQL, kterým by měl rozumět každý program založený na SQL, stále existují drobné (i nedrobné) rozdíly mezi různými implementacemi SQL.
Tyto rozdíly by vás mohly vyvést z míry, pokud zde probírané příklady budte zkoušet na jiném SQL softwaru, než je PostgreSQL.

ANSI/ISO SQL standard

Rok	Označení	Podrobnosti
1986	SQL86	ANSI
	SQL87	ISO
1989	SQL89	Základní verze normy. Nedůležité.
1992	SQL92	Moderní koncepce relační DB.
1999	SQL99	Objektová orientace, rekurze, reg-výrazy, triggery …
2003	SQL2003	XML, funkce oken, sekvence, generické sloupce
2006	SQL2006	xml-sql, XQuery
2008	SQL2008	spouště INSTEAD OF, truncate, order by …

Případné nesrovnalosti byste měli dohledat v dokumentaci k vašemu systému řízení databázových dat. V dokumentaci PostgreSQL se občas dočtete, co je a co není dle normy. Normy zcela nezvládla implementovat žádná databáze a už asi nikdy ani nezvládne, protože normy vznikly až po DBMS a zpětně se chování databází mění jen těžko (kvůli zpětné kompatibilitě). Navíc některé požadavky norem jsou prostě nereálné. Takže **zapomeňte na normy, studujte dokumentaci**.

V Linuxu, distribuci OpenSuSE, existuje balíček *postgresql-docs*. Po jeho nainstalování najdete informace o PostgreSQL v adresáři /usr/share/doc/packages/postgresql/ (začněte dokumentem html/index.html)
Na internetu se můžete podívat například na http://www.pgsql.cz/index.php/PostgreSQL nebo oficiální zdroj (v angličtině) http://www.postgresql.org/docs/9.4/interactive/.
Pozn: Některé rozdíly nejsou až tak drobné. Například mohou v nějaké implementaci DBMS existovat datové typy nebo celé SQL konstrukce, které v jiné nejsou. Nebo se mohou stejné SQL příkazy chovat odlišně. Třeba zde dříve probrané vytváření uživatele a databáze v PostgreSQL, MysSQL i Oracle se liší až odstrašujícím způsobem. Ale nebojte, všechno vám vysvětlím :-).

DBMS

Zde je popis databází, které jsem vybral pro tento tutoriál. Všechny jsou to SQL relační databáze.
Existují i ne-sql databáze, ale ty se obvykle používají k jiným účelům než SQL databáze, často jako doplněk k SQL databázi. SQL databáze vládnou světu!
Existují i jiné SQL databáze, například od Microsoftu MSSQL, ale nemůžu tu popisovat úplně všechno (= nerozumím úplně všemu).

PostgreSQL

PostgreSQL je asi nejlepší free opensource databází co existuje. Říkám asi, protože se vždycky najde nějaký rýpal, co tvrdí, že MySQL je lepší. Mezi PostgreSQL a MySQL je velká rivalita. Dřív se tvrdilo, že PostgreSQL je stabilnější a víc toho umí, zatímco MySQL je jednodušší a rychlejší. Pak se tvrdilo, že to závisí na tom, jaká a kolik dat zpracováváte a dneska se tvrdí bůh ví co.
PostgreSQL je následníkem databáze Postgres (která tím zanikla), ale běžně se jí pořád říká Postgres.
Postgres dnes najdete často i na (free) webovém hostingu, ale pořád to nebývá tak často, jako MySQL. přežívá mýtus o tom, že MySQL je méně náročná na hardware a že pokročilé funkce PostgreSQL stejnak většina nevyužije.
Pokud se chcete dozvědět něco dalšího o historii či budoucnosti PostgreSQL, podívejte se třeba na wikipedii.
Dle mých zkušeností je PostgreSQL skutečně stabilnější a umí toho více než MySQL, ale osobní zkušenost se dá těžko zaměňovat za objektivní pravdu :-).

MySQL a MariaDB

MySQL je novější DBMS než PostgreSQL. Její hlavní nevýhodou je, že ji koupil Oracle. Neumím věštit budoucnost, ale nepředpokládám, že by chtěl Oracle do rozvoje MySQL nějak moc investovat a odebírat tak zákazníky své vlastní databázi.
A protože si to nemyslím jen já, a protože MySQL byla vyvíjena pod licencí GNU, vznikl projekt MariaDB.
Která z těchto databází přežije ukáže čas. Velkou nevýhodou MariaDB je absence dokumentace (ta je totiž nyní výhradním vlastnictvím Oracle). Zatím je MariaDB věrnou kopií MySQL, takže když jí budete používat, nepoznáte oproti MySQL rozdíl. To však nevydrží na věky. Poslední verze MariaDB už nekopíruje na 100% vývoj MysQL, ale tohoto kurzu se to nijak nedotýká …

SQLite

SQLite je malý relační databázový systém, který je obsažený v malé knihovně. Nefunguje na principu klient-server. Hodí se všude tam, kde chcete ukládat databázová data, ale nechcete ztrácet čas implementací. Když znáte SQL, jediné co potřebujete je malá SQLite knihovna a máte vystaráno.
SQLite je také velmi populární. Používá jej například prohlížeč Chrome, Skype, Subversion, dokonce i Apple SQLite využívá.
SQLite podprouje mnoho programovacích jazyků (C, C++, PHP, Java, Python, Ruby …).
SQLite se zkrátka hodí, když si vystačíte s jedním (relativně) malým souborem pro ukládání dat, nepotřebujete řešit přístupová práva nebo přístup mnoha uživatelům k datům najednou (= paralelní přístup), ani vzdálený přístup (přes síť).

Oracle

Oracle je jednak název firmy (Oracle Corporation) a jednak název jejího databázového systému řízení dat (Oracle Database).
Oracle je tak drahý, že si ho určitě nemůžete dovolit (ani firmu, ani databázi). Naštěstí existuje verze pro testování *Oracle Database Express Edition*, jak bylo popsáno v kapitole o instalaci.
Oracle je hodně stará databáze (první verze Oracle je z roku 1978, Postgres 1989, MySQL 1995 a SQLite 2000), proto má občas podivnou „filozofii" v tom jak zacházet s uživateli, schématy, datumy a taky si nehraje moc na SQL normy.
Tedy, podivné vám to příjde, pokud už pracujete s PostgreSQL nebo MySQL. Pokud ještě s žádnou databází pracovat neumíte, těžko poznáte, co je normální a co podivnost :-)
Oracle je bezesporu nejúspěšnější komerční databáze. A taky toho hodně umí a umí to dobře (nenechte se zmást tím, že je nejstarší). Jen ta cena …

Začínáme s PostgreSQL

V této kapitole se dozvíte jak vytvořit databázi (schéma), jak získat nápovědu, něco o metapříkazech, znakových sadách a dumpování obsahu databáze do textového souboru (souboru SQL příkazů).

Databáze

Nyní se podíváte na to, jak vytvořit a smazat databázi, tj. nějaké soubory, ve kterých budou uloženy data. V jakém souboru to konkrétně bude (jak se jmenuje a kde je uložen) není důležité. O to se stará server.
Ukáži použití klienta **psql** pro připojení k databázi a představím jednu psql vychytávku – metapříkazy.
Dozvíte se také, jak data z databáze „vydumpovat" do textového souboru ve formě SQL příkazů a jak data zase zpět do databáze nahrát.

Vytvoření databáze

Chcete-li pracovat s databází, musíte jí nejdříve vytvořit. K tomu potřebujete mít příslušná práva – musíte být uživatelem s právem CEATEDB. To jsem ukazoval v kapitole o instalaci. Nyní si na zkoušku můžete vytvořit databázi se jménem rimmer1.
Zadejte příkaz v příkazové řádce:

```
$ createdb -E UTF8 -T template0 --lc-collate=cs_CZ.UTF-8 --lc-ctype=cs_CZ.UTF-8 rimmer1
CREATE DATABASE
```

Pokud proběhne vytvoření databáze úspěšně, zobrazí se zpráva CREATE DATABASE (ale taky nemusí, záleží na nastavení).
Pokud databáze se stejným jménem již existuje, budete na to upozorněni (původní, již existující databáze se nepřemaže).

Jméno databáze můžete zvolit stejné, jako je vaše přihlašovací jméno (login) v Linuxu. Pokud spustíte psql bez zadání názvu databáze, pokusí se připojit k databázi se stejným jménem jako je váš login.
Já jsem přidal ke jménu databáze na konec jedničku, aby bylo na první pohled jasné, zda mluvím o databázi nebo o loginu uživatele.

Databázi můžete vytvořit i v psql clientovi (dále už budu psát jen psql a budu tím myslet **jakéhokoliv klienta** pro spojení s Postgresem) následujícím SQL příkazem (začínajícím za =>). Všimněte si středníku na konci příkazu:

```
=> CREATE DATABASE rimmer1 TEMPLATE template0 ENCODING 'UTF8' LC_COLLATE 'cs_CZ.UTF-8' LC_CTYPE 'cs_CZ.UTF-8';
CREATE DATABASE
```

Po provedení SQL příkazu klient vypsal CREATE DATABASE. To je takový psql zvyk, že tím oznamuje úspěch provedené akce. Při neúspěchu vypíše chybu. V zájmu stručnosti a přehlednosti nebudu vždycky vypisovat co klient po provedení úsěšného příkazu vrací.

TEMPLATE

určuje, jakou databázi použít jako šablonu pro vytvoření nové. Pokud nic nezadáte, vytvoří se databáze podle databáze template1. Ta v sobě obsahuje nějaké, pro nás aktuálně nezajímavé, databázové objekty. Pokud jsou tyto objekty v jiném kódování, než ve kterém je nově vytvářená databáze, tak se vytvoření databáze nezdaří. Používejte jako šablonu template0. Ta neobsahuje nic, takže s ní problémy nehrozí.

ENCODING

určuje znakovou sadu, v jaké se budou ukládat texty v databázi. Asi víte, že existují různé znakové sady (UTF-8, Windows-1250, ISO-8859-2 atd.). Znaková sada určuje, pod jakými čísly se znaky ukládají a kolik budou zabírat místa. Některé znakové sady zabírají více místa a umožňují používat znaky z mnoha jazyků (písmena s háčkami, čárkami, přehláskami, čínské čmáranice …), což je třeba příklad UTF-8, některé používají méně bytů, ale zase mají jen omezený repertoár znaků (například Windows CP1250 je kódování pro středoevropské jazyky od firmy Windows, zatímco ISO-8859-2 je taktéž kódování pro středoevropské jazyky, ale tentokrát se jedná o standard (který používal Linux)). **Používejte kódování UTF-8.** Možná by se vám mohlo ve Windows lépe pracovat s WIN1250, ale UTF-8 je prostě modernější a poradí si s většinou jazyků, se kterými se můžete setkat.

LC_COLLATE

určuje, jak se budou znaky řadit. Čeština má například tu specialitu, že ch je v abecedě před cd, protože ch se bere jako jedno písmeno.
Používejte pro COLLATE **cs_CZ.UTF-8**, pokud je to možné (a vhodné).
Postgres umí „tradiční C" COLLATE (normální řazení a-z), které se jmenuje C a pak POSIX (to je nějaká norma). Další podpora COLLATE je závyslá na operačním systému, takže se může taky stát, že cs_CZ.UTF-8 nebudete mít dostupné.

LC_CTYPE

určuje další informace o znacích (např. co je znak, co je číslo, co je velké a co malé písmeno atd.).
LC_CTYPE byste ani nemuseli zadávat, ono se k LC_COLLATE vybere tak nějak samo to správné.

LC_COLLATE a LC_CTYPE se po vytvoření databáze již nedá změnit!

SQL příkaz na vytvoření databáze by se dal zapsat také takto:

```
CREATE DATABASE rimmer1 WITH
    TEMPLATE = template0
    ENCODING = 'UTF-8'
    LC_COLLATE = 'cs_CZ.UTF-8'
    LC_CTYPE = 'cs_CZ.UTF-8';
```

Mezery (nebo nové řádky) nehrají roli. Slovíčko WITH, stejně jako rovnítka, je nepovinné (je jedno, jestli je uvedete nebo ne).
Při vytváření databáze se dají určit ještě další informace, o kterých jsem se nezmínil. Podívejte se do dokumentace. To, co je v hranatých závorkách, je nepovinné. Buď je to tam na okrasu nebo aby se splnil nějaký SQL standard a prakticky nemá žádný význam (to je případ slovíčka WITH), nebo se při vynechání použije nějaká defaultní hodnota (to je případ TEMPLATE, ENCODING, LC_COLLATE i LC_CTYPE).

Zrušení databáze

Pokud zrušíte databázi, už se to nedá vzít zpět. **Zrušením databáze smažete všechny tabulky, které jste v ní vytvořili!** To vás ale zatím nemusí trápit, protože jste žádnou tabulku nevytvořili :-).
Smazat databázi můžete z příkazové řádky programem dropdb.

```
$ dropdb rimmer1
```

A ještě jak se to dá udělat v klientovi:

```
DROP DATABASE rimmer1;
```

Za domácí úkol zjistěte z dokumentace, jak se dá rozšířit SQL příkaz DROP DATABASE, aby nevyhodil žádnou chybu když se pokusíte smazat neexistující databázi (to se může hodit, když budete spouštět příkazy z dávkového souboru – chyba by dávkový soubor ukončila).

V žádném případě nemažte databázi kterou jste nevytvořili, pokud si nejste jisti že víte co děláte. Postgres při své instalaci vytvoří několik databází, které potřebuje pro svůj běh!

Komunikace s databází

Pro komunikaci s databází postgres slouží program **psql**. Jedná se o textového klienta, který doporučuji používat.
V předchozí kapitole jsem vysvětlil, že program postgresql je serverem a psql klientem. Pokud se chcete připojit k databázi, musíte říci k jaké. Vytvořte databázi rimmer1 a připojte se k ní:

```
$ createdb rimmer1
CREATE DATABASE
$ psql rimmer1
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit
```

rimmer1=>

V příkazové řádce ve Windows před spuštěním psql zadejte tento příkaz: chcp 1250. Takto změníte codepage příkazové řádky na windows 1250, které využívá i psql. (Defaultní codepage příkazové řádky ve Windows je cp852.)

Nyní jsou očekávány vaše (SQL) příkazy. Krom toho můžete také zadávat **metapříkazy**. To jsou příkazy které začínají zpětným lomítkem a většinou mají jen nějaký informativní charakter (třeba \h zobrazí nápovědu).

Pomocí šipky nahoru (na klávesnici) si můžte procházet historii příkazů, příkazy upravit (opravit) a spustit znovu (nejsem si jistý, jestli to tak funguje i ve Windows).

Práci s databází ukončíte metapříkazem \q.

Všechny příkazy se spouští klávesou Enter.

Všechny SQL příkazy se ukončují středníkem (a pak se Enterem spustí).

Metapříkaz \q ukončí program psql.

```
rimmer1=> \q
$
```

Klienta psql můžete ukončit i kombinací kláves CTRL+d v Linuxu, v DOSu a Windows je to CTRL+z a Enter.

Metapříkazy

Metapříkazy ještě nejsou příkazy SQL a v jiných SQL programech než je Postgres nemusí vůbec existovat (ani v jiných postgres klientech, než je psql).

Metapříkazy začínají zpětným lomítkem a neukončují se (na rozdíl od SQL příkazů) středníkem.

Dva metapříkazy jste si již mohli vyzkoušet – příkaz na ukončení sezení \q a příkaz zobrazující stručnou nápovědu \h. Další metapříkazy, které by se vám mohly v krátké době hodit, jsou následují:

\q

Ukončí program psql.

\h

Tento metapříkaz budete pravděpodobně využívat nejčastěji. Pokud jej zadáte bez argumentu, vypíše vám všechny možné příkazy SQL. Jako argument můžete zadat jeden z SQL příkazů. Pak se vám ukáže krátký popis toho co příkaz dělá, spolu se syntaxí příkazu.

Nedozvíte se žádné podrobnosti jako v nápovědě na webu, spíš vám pomůže si vzpomenout, co všechno příkaz umí.

Tak například:

```
rimmer1=> \h DROP DATABASE
Příkaz:     DROP DATABASE
Popis:      odstraní databázi
Syntaxe:
DROP DATABASE [ IF EXISTS ] jméno
```

Dočtete se, že příkaz DROP DATABASE slouží k odstranění databáze. To co vidíte v syntaxi příkazu velkými písmeny se nemění. Vždy tedy píšete úvodní slova DROP DATABASE. To co je v hranatých závorkách [ ] je nepovinné – IF EXISTS může a nemusí být součástí SQL příkazu (pokud jej chcete použít, tak se píše bez hranatých závorek).

Za výrazy s malými písmeny se dosadí příslušná hodnota (místo jméno byste měli dosasit skutečné jméno databáze, kterou chcete smazat.

Další příklad:

```
rimmer1=>\h END
Command:     END
Description: commit the current transaction
Syntax:
END [ WORK | TRANSACTION ]
```

Znak | (svislítko) znamená „nebo". Příkaz END může mít jeden, nebo žádný argument (argumenty jsou v hranatých závorkách). Argument může být WORK nebo TRANSACTION.

*Pokud by místo hranatých závorek [] byli špičaté **{}**, znamnalo by to, že musí být jeden z argumentů uveden, tj. že příkaz musí mít jeden z vyjmenovaných argumentů.*

A další příklad:

```
rimmer1=> \h drop table
Command:     DROP TABLE
Description: remove a table
Syntax:
DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

A tady si všimněte *name [, ...]*. Tím se nám nápověda snaží naznačit, že můžete uvést více než jedno jméno tabulky (oddělená čárkou).

Tyto příklady jsem uvedl, abyste lépe porozuměli nápovědě. Vlastní význam příkazů proberu v některé z dalších kapitol.

\?

Tento metapříkaz vypíše všechny možné metapříkazy.

Pokud se výstup nevejde na obrazovku, můžete se ve výpisu pohybovat klávesou ENTER, mezerníkem a šipkami. Prohlížení ukončíte klávesou q.

\l

(Malé L) Vypíše tabulku všech existujících databází.

```
rimmer1=>\l
                                Seznam databází
   Jméno    | Vlastník | Kódování |   Collation   |    CType    |  Přístupová práva
------------+----------+----------+---------------+-------------+-----------------------
               petr      | petr     | UTF8     | cs_CZ.UTF-8 | cs_CZ.UTF-8 |
             postgres  | postgres | UTF8     | cs_CZ.UTF-8 | cs_CZ.UTF-8 |
             rimmer1   | petr     | UTF8     | cs_CZ.UTF-8 | cs_CZ.UTF-8 |
template0 | postgres | UTF8     | cs_CZ.UTF-8 | cs_CZ.UTF-8 | =c/postgres         +
```

```
           |          |          |            |            | postgres=CTc/postgres
template1 | postgres | UTF8     | cs_CZ.UTF-8 | cs_CZ.UTF-8 | =c/postgres        +
           |          |          |            |            | postgres=CTc/postgres
(5 řádek)
```

\d

Tento metapříkaz vám ukáže všechny objekty, které v databázi máte. Nyní tam nemáte nic, takže se vám ukáže, že se vám nemá co ukázat. Ukážeme si to na příkladu.

Abychom si měli co ukázat, vytvoříme si tabulku, která bude mít dva sloupce, v jednom bude text, ve druhém číslo. (Hurá, náš první SQL příkaz – ale zatím jej nechám bez hlubšího vysvětlení).

```
rimmer1=> CREATE TABLE pokus (text text, cislo integer);
CREATE TABLE
rimmer1=> \d
          Seznam relací
 Schéma | Jméno |  Typ   | Vlastník
--------+-------+--------+----------
 public | pokus | tabulka | petr
```

Metapříkaz **\d** může mít argument, kterým bude „objekt", který máte v databázi. Objektem je například tabulka (o dalších typech objektů, co můžete mit v databázi, se dozvíte později).

```
rimmer1=> \d pokus
Tabulka "public.pokus"
 Sloupec |  Typ    | Modifikátory
---------+---------+--------------
 text    | text    |
 cislo   | integer |
```

```
rimmer1=> DROP TABLE pokus;
```

Jak vaše databáze poroste a bude obsahovat více a více objektů, budete chtít vypsat například jen tabulky. K tomu slouží několik „bratříčků" metapříkazu \d; Například **\dt** vypíše jen tabulky.

Nemá cenu abych tyto bratříčky vypisoval, když ještě nevíte, jaké objekty můžete v databázi mít. Až se k jednotlivým objektům dostanu, zmíním se i o příslušném metapříkazu. Ostatně, pomocí \? tyto metapříkazy už umíte zjistit sami.

\dO

(\d a velé o) Vypíše všechna COLLATION, která máte k dispozici.

Že vám \dO nic nevypíše? No to je tím, že ve své databázi žádné definované nemáte. Pokud chcete vidět ty systémové, musíte doplnit metapříkaz písmenkem S:

```
rimmer=> \dOS
                Seznam collations
   Schéma   |      Jméno      |     Collation     |      CType
------------+------------------+-------------------+-------------------
 pg_catalog | C                | C                 | C
 pg_catalog | POSIX            | POSIX             | POSIX
 pg_catalog | aa_DJ            | aa_DJ.utf8        | aa_DJ.utf8
 pg_catalog | aa_DJ.utf8       | aa_DJ.utf8        | aa_DJ.utf8
                              ...
 pg_catalog | cs_CZ            | cs_CZ.utf8        | cs_CZ.utf8
 pg_catalog | cs_CZ.utf8       | cs_CZ.utf8        | cs_CZ.utf8
                              ...
(426 řádek)
```

Spousta metapříkazů existuje ve verzi s S na konci pro vypsání systémových informací.

Dávkové soubory (dump databáze)

Příkazy SQL také můžete zapsat do souboru ve stejné formě, jako byste je zadávali na příkazovou řádku (i se středníkem na konci SQL příkazů). Poté je na databázi aplikujete takto:

```
$ psql jmeno_databaze < jmeno_souboru_s_prikazy.sql
```

Soubor je obyčejný textový soubor a na jeho koncovce vůbec nezáleží. Někdy se mu nadává *dávkový soubor*, protože obvykle obsahuje nějakou dávku SQL příkazů (kvůli jednomu příkazu by se ho asi nevyplatilo psát).

Pokud do dávkového souboru chcete přidat nějaký komentář, uveďte na začátku řádky s komentářem -- (dvě mínus a mezeru) a vše další až do konce řádky bude ignorováno.

Chcete-li naopak data z databáze uložit do souboru (ve formě SQL příkazů), použijte k tomu program **pg_dump**.

```
$ pg_dump -O jmeno_databaze > jmeno_souboru_pro_prikazy.sql
```

Můžete si to vyzkoušet, ale protože teď v databázi nic nemáte, nic smysluplného se vám nevydumpuje.

Vydumpované soubory obvykle obsahují spoustu nestandardních věcí, které znemožňují vzít skript z jedhoho DBMS a nahrát ho do jiného DBMS. Pro začátek vám poradím toto: cokoliv se vydumpuje a vy tomu nerozumíte smažte :-).

MySQL/MariaDB

Rozdíly v dalších databázích se budu snažit popisovat velmi stručně, takže bez dalšího zdržování uvedu rovnou příklad na vytvoření a smazání databáze v MySQL:

```
CREATE DATABASE rimmer1 DEFAULT CHARACTER SET = UTF8 DEFAULT COLLATE = utf8_czech_ci;
DROP DATABASE IF EXISTS rimmer1;
```

Všimněte si slova DEFAULT. Znaková sada a COLLATE se v MySQL definuje na úrovni tabulky a ne databáze. Tímto jen definujete, jaká znaková sada a COLLATE se má použít jako defaultní, tedy u nově vytvářené tabulky, u které znakovou sadu nebo COLLATE nedefinujete explicitně.

Schizofrenie v názvosloví je krásně vidět na tom, že místo DATABASE můžete v obou příklade psát **SCHEMA** a bude to znamenat to samé. A já vám doporučuji používat název SCHEMA, protože je jasnější o čem je řeč, než používání všehobjímající slovo DATABASE.

V MySQL také existují metapříkazy, ale jsou velice odlišné. Třeba metapříkaz \h nezobrazí přehled SQL příkazů, ale přehled metapříkazů (kterých je o poznání méně než v psql).

Pokud se chcete dostat k nápovědě k SQL dotazům, použijte příkaz **help**, pak **help contents**, pak třeba **help Data Definition** a nakonec **help CREATE DATABASE**.

```
mysql> help CREATE DATABASE
Name: 'CREATE DATABASE'
Description:
Syntax:
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
[create_specification] ...

create_specification:
[DEFAULT] CHARACTER SET [=] charset_name
| [DEFAULT] COLLATE [=] collation_name

CREATE DATABASE creates a database with the given name. To use this
statement, you need the CREATE privilege for the database. CREATE
SCHEMA is a synonym for CREATE DATABASE.

URL: http://dev.mysql.com/doc/refman/5.6/en/create-database.html
```

Příkaz **mysqldump** vydumpuje databázi:

```
$ mysqldump  jmeno_databaze > jmeno_souboru_pro_prikazy.sql
```

SQLite

Program sqlite3 při spuštění očekává jako argument název souboru, se kterým bude pracovat jako s databází. Pokud soubor neexistuje, vytvoří jej. Pokud existuje, ověří, že se jedná o platný SQLite soubor a pracuje s ním.

SQLite umí pracovat jen s UTF-8 a UTF-16 kódováním. To první je defaultní, bez toho druhého se (zatím) obejdete.

Příkaz pro nápověda v programu sqlite3 je **.help** (i s tou tečkou na začátku).

Další zajímavé metapříkazy jsou **.headers ON** a **.mode column**, které vám doporučuji vždy spouštět. Zajistí hezčí výpis při výpisu obsahu tabulky.

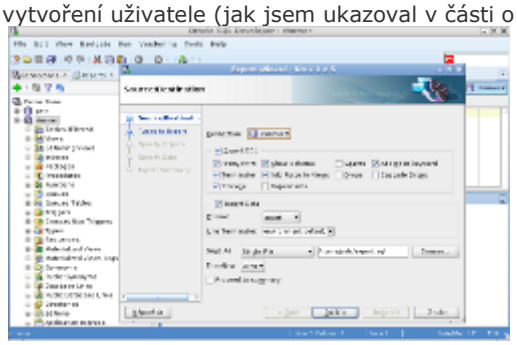
Metapříkaz .tables vypíše všechny tabulky.

Pro uživatele Linuxu: zapište metapříkazy .headers ON a .mode column do souboru ~/.sqliterc a nebudete je muset psát po každém spuštění sqlite3

Metapříkaz .dump vydumpuje databázi.

Oracle

Oracle vám vytvoří databázi rovnou při vytvoření uživatele (jak jsem ukazoval v části o instalaci). Defaultní kódování je UTF-8.



Oracle Export

Dumpování databáze můžete udělat přes Oracle SQL Developer (Tools → Database Export… spustí Export Wizard, kde si všechno naklikáte).

Apex bohužel nemá nic vhodného na dumpování. Můžete si přes něj vydumpovat tabulku po tabulce v CSV formátu, můžete si zobrazit CREATE TABLE příkazy pro jednotlivé tabulky, ale nevydumpujete si celé schéma najednou.

Apex nabízí jen možnost nahrát SQL skripty (SQL Workshop → SQL Scripts → Upoad >) a nahrané skripty spustit (případně i upravit).

Apex i SQL Workshop mají limit, který omezuje velikost skriptu, který je možný nahrát. V případě nouze tak můžete využít pro nahrání velkého skriptu příkazový řádek:

```
$ cd /u01/app/oracle/product/11.2.0/xe      # prejdu do adresare kde je oracle nainstalovano
$ . ./bin/oracle_env.sh                     # nastaveni prostredi
$ ./bin/sqlplus jmeno/heslo < script.sql    # nahrani skriptu pomoci programu sqlplus
```

První tabulka

V této kapitole se konečně naučíte pracovat s tabulkami. Naučíte se tabulku vytvořit, vložit do ní nějaká data, zobrazit si je a nakonec tabulku zase smazat. Po této kapitole už budete téměř schopni databáze využívat v reálném životě.

Datové typy

Pokud chcete vytvořit tabulku, musíte databázi (DBMS) říct, jaká data budete do sloupců tabulky ukládat (například jestli to bude text nebo číslo).

To umožňuje DBMS s tabulkami snadno pracovat. Jednak je hned jasné, které operace s hodnotami v daném sloupci lze provádět (čísla lze násobit, text ne) a také umožňuje snadnou správu databáze, neboť je (většinou) předem jasné, kolik bude zabírat jeden řádek tabulky místa na disku.

K tomu, abyste určili jaká data budete do sloupců ukládat, slouží **datové typy**. Datový typ určuje jednak jaký typ dat budete ukládat (text, celé číslo, číslo s desetinou čárkou) a také kolik bytů bude zabírat. Například typ integer je celé číslo 4 byty dlouhé, jeho rozsah je cca -2 až 2 miliardy, smalint je celé číslo 2 byty dlouhé, jeho rozsah je cca -32 až 32 tisíc, atd.).
V následující tabulce vidíte nejpoužívanější datové typy z PostgreSQL.
Pokud vás zajímají další, použijte metapříkaz **\dTS**.

Datové typy je jedna z věcí, ve kterých se DBMS od sebe dost liší (a stěžují tak přenositelnost mezi databázemi). DBMS mají pár datových typů společných (například integer nebo text, date), ale i ty se mohou lišit například v tom, kolik bytů jsou dlouhé, jak se v nich hodnoty ukládají (například datum může být ukládán s nebo bez časové zóny).
Když budete přecházet z jednoho DBMS do druhého, raději si pořádně datové typy nastudujte.

Datové typy v PostgreSQL

Typ	Význam	Popis
boolean	Logická hodnota	Může nabývat pouze dvou hodnot: **true** (pravda) a **false** (nepravda). Za true se též považuje každá nenulová hodnota, za false pak nulová hodnota. V některých DBMS je bool jen synonymem pro celé číslo dlouhé 1 bit, tj může obsahovat jen jedničku (pravdu) nebo nulu (nepravdu).
char(*n*)	Znakový řetězec	Délka řetězce je *n* znaků. Zadáte-li kratší řetězec, bude doplněn z prava mezerami. Za *n* se dosazuje celé kladné číslo. **Všechny typy znakových řetězců se v SQL příkazech uvozují jednoduchými 'uvozovkami'.** Pokud je jednoduchá uvozovka součástí řetězce, musí se před ní napsat zpětné lomítko: \'. Hodí se na řetězce o kterých předem víte, že budou mít (více méně) vždy stejnou délku (například sloupeček rodných čísel).
varchar(*n*)	Znakový řetězec	Délka řetězce může být maximálně n znaků. Zůstává ve své délce (nedoplňuje se mezerami). Varchar obsahuje informaci o délce řetězce (krom samotného řetězce). Práce s ním je o fous pomalejší, ale zabírá méně místa než char, navíc se nedoplňuje mezerami, takže varchar se používá mnohem častěji než char.
text	Znakový řetězec	Řetězec (skoro) neomezené délky. Pokud nemusíte, nepoužívejte. Zpomaluje práci s daty v databázi. Hodí se na všechny texty, kde nechcete nebo nemůžete omezit délku (například na ukládání textu článků).
integer	Celé číslo	Číslo bez desetinné čárky. Přípustné hodnoty jsou cca od -2 do 2 miliard.
float	Reálné číslo	Číslo s desetinnou čárkou. (Spíše bych měl napsat s desetinnou tečkou.) float má omezenou přesnost (občas zapomíná poslední desetinná čísla atp.). Naprosto se nehodí na ukládání informací o penězích!
numeric(*p*,*d*)	Reálné číslo	Číslo se zadanou přesností. Parametr *p* určuje celou délku čísla, parametr *d* kolik z toho bude za desetinnou čárkou. Například za numeric(5, 3) můžete dosadit maximálně [+/-]99.999. numeric v rámci svého rozsahu uloží čísla vždy „přesně", hodí se proto na ukládání informací o penězích. Na druhou stranu zabírá více paměti než float.
serial *	Automatické (celé) číslo	Do sloupce s tímto typem se automaticky ukládá nepoužité číslo. Začíná se od 1, pokračuje 2, 3, 4 atd. (není li určeno jinak). Hodí se jako primární klíč. Více o serial.
date	Datum	Zadává se jako textový řetězec ve tvaru '2002-09-26'.
time	Čas	Zadává se jako textový řetězec ve tvaru '13:00:13.440270' (Přesnost na miliontinu vteřiny), nebo jen '13:00:00' (hodiny, minuty, vteřiny) nebo jen 'hodiny:minuty'. Jakým způsobem se odděluje čas (jestli dvojtečkou, nebo tečkou) se dá v PostgreSQL nastavit příkazem SET datestyle, ale to je zatím nad rámec tohoto tutoriálu.
timestamp	Čas + datum	'2002-09-26 13:07:40'

** U typu SERIAL se ještě trochu pozastavím. Pokud jej v nějakém sloupci použijete, pro jeho správnou funkci se vytvoří v databázi objekt **sekvence**. Všechny sekvence v databázi (tedy vlastně schématu) můžete vypsat pomocí metapříkazu **\ds**. ~~Po zrušení tabulky obsahující datový typ SERIAL je třeba zrušit i tuto sekvenci a to příkazem **DROP SEQUENCE nazev_sekvence**~~. PostgreSQL už maže sekvenci automaticky. Nicméně se pořád může stát, že budete mít v databázi nějaké neopoužité sekvence. O tom ale až jindy.*
*Název sekvence se generuje automaticky a to ve tvaru nazevtabulky_nazevsloupce**_seq**.*
Podrobnosti o sekvencích popíši později v CREATE SEQUENCE.

Některé datové typy mají své zkratky, takže například místo boolean můžete psát bool, nebo místo integer jen int.
PostgreSQL je slavný tím, že má velkou škálu datových typů. Najdete v něm datový typ pro kruh, obdélník, polygon, IP adresu, MAC adresu, XML, JSON a mnoho dalších zajímavostí. Určitě se koukněte na datové typy do dokumentace.

Práce s tabulkou

Teď je ten správný čas vrhnout se na vytváření tabulek. Zde probírané příkazy budou v dalších kapitolách probírány podrobněji, nyní se naučíte jen nezbytné minimum z SQL příkazů.

Při psaní SQL příkazů nezáleží na velikostech písmen, nicméně pro přehlednost budu psát nezaměnitelné výrazy velkými písmeny, výrazy za které si dosazujete svoje jména (tabulek, sloupců…) malými.

CREATE TABLE – vytvoření tabulky

Příkaz CREATE TABLE slouží k vytvoření tabulky. Za klíčovými slovy CREATE TABLE následuje jméno vytvářené tabulky a v závorce jména sloupců následovaná datovým typem sloupce. Jména a datové typy jednotlivých sloupců se oddělují čárkou.

```
rimmer1=> CREATE TABLE telefonni_seznam (jmeno VARCHAR(20),
rimmer1(> prijmeni VARCHAR(20),telefon VARCHAR(20));
CREATE
```

Příkazy SQL se v Postgresu ukončují středníkem. První řádek jsem ukončil stiskem klávesy ENTER. Všimněte si, jak se změnil prompt. Místo rovnítka je tam kulatá závorka, která nás upozorňuje na to, že jsme v některém z předchozích řádků začali se závorkou a měli bychom jí ukončit.

Telefon, ač se to může zdát býti číslo, jsem určil jako datový typ VARCHAR, neboť s telefonem nebudu provádět žádné matematické operace (sčítání telefonů není moc užitečné). Tím se můžu vyvarovat někdy v budoucnu chyb, protože kdybych se pokusil telefony sčítat, Postgres ohlásí chybu "sčítání řetězců". A taky nebudu mít problém s telefonem, který začíná nulou.

Poslední věc o které bych se měl zmínit je hlášení, které se vám vypíše po provedení SQL příkazu. V tomto případě vidíte hlášení CREATE, které vám oznamuje, že tabulka byla vytvořena. V případě že se jí nepodaří vytvořit vypíše se chybové hlášení s popisem chyby, která neumožnila příkaz provést (podobná hlášení obdržíte i od ostatních příkazů SQL). Zkuste třeba znovu vytvořit tabulku *telefonni_seznam* (šipkou nahoru se „vyrolujete" k příkazu *CREATE TABLE*).

```
rimmer1=> CREATE TABLE telefonni_seznam (jmeno varchar(20),
rimmer1(> prijmeni VARCHAR(20),telefon VARCHAR(20));
ERROR:  relation 'telefonni_seznam' already exists
```

Nelekejte se, error není konec světa. Tabulka telefonni_seznam, která již existovala, existuje dál a stále obsahuje všechny hodnoty, které jste do ní vložily (Zatím jste nevložili žádné :-)).

O existenci tabulky se můžete přesvědčit metapříkazem \d.

```
rimmer1=> \d
                Seznam relací
 Schéma |      Jméno      |   Typ   | Vlastník
--------+-----------------+---------+----------
 public | pokus           | tabulka | petr
 public | telefonni_seznam | tabulka | petr

(1 řádka)

rimmer1=> \d telefonni_seznam
      Tabulka "public.telefonni_seznam"
  Sloupec  |         Typ          | Modifikátory
-----------+----------------------+--------------
 jmeno     | character varying(20) |
 prijmeni  | character varying(20) |
 telefon   | character varying(20) |
```

Nenechte se zmást tím, že je tabulka v schématu „public", ikdyž jste jí vytvořili v schématu rimmer1. Jak se se schématy v PostgreSQL pracuje se dozvíte někdy později.

*Nenechte se ani zmást typem sloupce character varying(20). Datový typ VARCHAR je zkratka pro character varying. Pro integer existuje zase zkratka **int** atp.*

Postgres nemá SQL příkaz ani metapříkaz pro zobrazní CREATE TABLE statementu, kterým by se dala tabulka znova vytvořit (třeba v jiném schématu). Můžete ale použít program pg_dump.

```
$ pg_dump rimmer1 --schema-only -t telefonni_seznam
```

Výstup se může lišit od vašecho původního CREATE TABLE statementu, ze stejných důvodů, jak je pospáno níže u MySQL příkazu SHOW CREATE TABLE.

INSERT – vložení dat do tabulky

Příkaz INSERT slouží ke vkládání řádků do tabulky. Za klíčovými slovy INSERT INTO následuje název tabulky, klíčové slovo VALUES a seznam hodnot oddělených čárkou.

Hodnota NULL

Pokud do některého sloupce nechcete vložit hodnotu, můžete použít speciální hodnotu **NULL**. Tato hodnota se může použít za libovolný datový typ a říká, že položka „nemá žádnou zadanou hodnotu, resp. že její hodnotu neznám". Hodnota NULL se ničemu nerovná. Pokud jí třeba použijete v sloupečku s datovým typem integer, pak NULL rozhodně není totéž jako 0 (nula je totiž hodnotou)!

Hodnota NULL není ani větší ani menší než jakákoliv jiná hodnota (tj. podmínka *promenna* > NULL bude vyhodnocena jako false (nepravda), stejně tak i *promenna* < NULL je false).

Taky platí, že NULL se nerovná NULL. Hodnotu NULL můžete brát jako **„nevím co tam je"**. A když nevíte, nemůžete ani tvrdit, že se hodnoty rovnají.

```
rimmer1=> INSERT INTO telefonni_seznam VALUES('Santa', 'Klaus', '333222111');
INSERT 0 1
rimmer1=> INSERT INTO telefonni_seznam VALUES(NULL, 'Satan', '333999666');
INSERT 0 1
```

Pokud budete mít hodně sloupců a budete chtít vytvořit řádek s hodnotami jen v některých sloupcích, můžete vyjmenovat jména sloupců, do kterých dosazujete, za jménem tabulky v závorce oddělené čárkou.

Sloupce do kterých nic nevložíte budou mít defaultní hodnotou, která je NULL (časem ukážu, jak můžete defaultní hodnotu změnit na něco smysluplnějšího).

```
rimmer1=> INSERT INTO telefonni_seznam (prijmeni, telefon)
rimmer1-> VALUES ('Lucifer', '22222223');
INSERT 0 1
```

Čísla vřádce INSERT 0 1, která se zobrazí po vložení řádku, zobrauzují nějaké OID (identifikátor objektu. Pokud ho postgres nepoužije, zobrazí se 0. Jako v příkladu.) a počet vložených řáků.

SELECT – výběr dat z baulky

Příkaz SELECT je jeden z nejpropracovanějších příkazů SQL. Slouží k vybrání řádků a sloupců z tabulky a mnoha mnohda dalším věcem. Zatím vám bude stačit vědět, jak vybrat celou tabulku, abyste se mohli podívat na to, co v tabulce máte.

SQL jazyk je hlavně a především o příkazu SELECT. Příkazem slouží k „pohledu" na data v databázi. A umí toho mnohem více, než jen vypsat data. Z toho se vám ještě zamotá hlava, co všechno SELECT umí. Všechny ostaní SQL příkazy oproti SELECTu jsou jen takové šedé myšky.

```
rimmer1=> SELECT * FROM telefonni_seznam;
 jmeno | prijmeni |  telefon
-------+----------+-----------
 Santa | Klaus    | 333222111
       | Satan    | 333999666
       | Lucifer  | 22222223
(3 řádky)
```

Všimněte si, že ve výpisu nepoznáte, jestli je ve sloupečku jmeno hodnota NULL, nebo jestli je tam prázdný řetězec ''.

Jste-li nedočkaví, můžete se nyní mrknout na kapitoly o příkazu SELECT SELECT a SELECT II. K těmto kapitolám se dostanete později, až získáte další potřebné znalosti.

DROP TABLE – smazání tabulky

Podívejte se, co vám o příkazu DROP TABLE řekne nápověda Postgresu vyvolaná pomocí metapříkazu \h.

```
rimmer1=> \h DROP TABLE
Příkaz:     DROP TABLE
Popis:      odstraní tabulku
Syntaxe:
DROP TABLE [ IF EXISTS ] jméno [, ...] [ CASCADE | RESTRICT ]
```

Nebylo těžké uhodnout, že příkaz DROP TABLE slouží ke zrušení tabulky. Pokud tabulku zrušíte, přijdete nenávratně i o všechny data v ní! Syntaxe je taková, že nejdříve napíšete klíčová slova DROP TABLE, potom jméno tabulky, popřípadě další jména tabulek určených ke zrušení oddělených od sebe čárkou.

```
rimmer1=> DROP TABLE telefonni_seznam, pokus;
DROP
```

MySQL/MariaDB

MySQL má také datové typy char, varchar, text, **mediumtext**, **longtext**, integer, float, numeric a date. Bool je jen synonymum pro integer(1) (číslo dlouhé 1 bit – může obsahovat jen jedničku nebo nulu). Nemá serial (K tomuto „typu" se ještě vrátím v některé další kapitole. Teď jen prozradím, že MySQL nemá sekvence.) Místo timestamp má **datetime**. Má i timestamp, ale ten se chová trochu speciálně. Práce s časy a datumy je vůbec v SQL databázích zajímavý oříšek, takže se tomu budu věnovat později v samostatné kapitole.

MySQL má samozřejmě i další datové typy (stejně tak MariaDB). Sice ne ty, co jsem vyjmenovával extra u PostgreSQL, ale třeba má navíc set nebo enum.

Pokud chcete přenést tabulku z jednoho DBMS do druhého, vždy se ujistěte, že existují stejné datové typy a že se i chovají stejně (mají stejný rozsach hodnot, stejnou defaultní hodnotu, chovají se stejně při sčítání atp.).

Pokud se v PostgreSQL nebo Oracle pokusíte uložit do textového typu delší řetěcec než je povoleno (třeba do typu varchar(10) 11 znaků dlouhý řetězec), způsobí to chybu a SQL příkaz se neprovede (nic se nevloží). V MySQL se naproti tomu řetězec zkrátí a SQL příkaz se provede. (Vypíše se jen jakési varování.)

Příklady použití SQL příkazů z této kapitoly jsou pro MySQL stejné. To neznamená, že by všechny u všech zmíněných SQL příkazů neexistovali rozdíly. Jen jsem vám ty rozdíly prozatím zamlčel. Zmínim se teď jen o jednom a to u CREATE TABLE. CREATE TABLE může v MySQL určit defaultní CHARSET a COLLATE pro tabulku. (Pokud jej neurčíte, převezmou se defaultní hodnoty ze schématu).

```
CREATE TABLE telefonni_seznam (
jmeno VARCHAR(20),
prijmeni VARCHAR(20),
telefon VARCHAR(20)
) DEFAULT CHARSET UTF8 COLLATE utf8_czech_ci;
```

Všimněte si, že je CHARSET (i COLLATE) zase *DEFAULT*. MySQL totiž umožňuje nastavit znakovou sadu i způsob třídění na úrovni sloupečku!

MySQL v Linuxu rozlišuje velikost písmen u názvu tabulky!

MySQL totiž používá název tabulky pro název souboru, ve kterém je tabulka uložená. A protože Linux rozlišuje velikost písmen v názvech soborů (na rozdíl od Windows), tak i v MySQL (v Linuxu) záleží název tabulky na velikosti písmen. Je to hnus, ale co s tím naděláte. Doporučuji používat pro názvy tabulek vždy jen malá písmena.

V MySQL není metapříkaz \d pro zobrazení tabulek, ale má na to speciální (nestandardní) SQL příkaz SHOW. Na zobrazení popisu tabulky zase používá DESCRIBE.

```
mysql> SHOW TABLES;
+------------------+
| Tables_in_rimmer1 |
+------------------+
| telefonni_seznam  |
+------------------+
1 row in set (0.00 sec)

mysql> DESCRIBE telefonni_seznam;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| jmeno    | varchar(20) | YES  |     | NULL    |       |
| prijmeni | varchar(20) | YES  |     | NULL    |       |
| telefon  | varchar(20) | YES  |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
```

```
3 rows in set (0.00 sec)
```

Na zobrazení CREATE TABLE příkazu, kterým by se tabulka dala znovu vytvořit, slouží příkaz SHOW CREATE TABLE.

```
mysql> SHOW CREATE TABLE telefonni_seznam;
...
```

Výstup tohoto příkazu nezobrazí váš původní SQL příkaz CREATE TABLE, ale sestaví jej na základě všech informací, které o tabulce DBMS má. Z toho důvodu výstup obsauje více informací, než jste při vytváření tabulky zadávali (přidá do CREATE TABLE explicitně i věci, které se vytvořili defaultně, protože jste je neuvedli), případně jsou zapsány jiným (ekvivalentním) způsobem.

SQLite

I SQLite má seznam svých datových typů (integer, real, text a blob). Je to o něco chudčí než v ostatních databázích, ale, vzhledem k tomu, že SQLite není určeno na nějaké velké složité databáze, určitě dostatečné.

Typ boolean je synonymum pro integer, podobně jako u MySQL, jen sqlite neumí omezit délku na 1 bit, takže můžete do sloupečku „boolean" vložit jakékoliv číslo. Na rozdíl od Postgresu a MySQL nezná SQLite ani hodnoty true a false, takže můžete pro bool hodnoty používat jen jedničky a nuly.

SQLite má například jen jeden datový typ pro text a to text. Přesto vám umožní definovat sloupce s datovými typy jako varchar(25) nebo char(10). Nicméně, takový sloupeček bude fungovat jako text, tj. nezkrátí se na zadanou maximální délku pokud je moc dlouhý, nedoplní se mezerami …

Příklady použití SQL příkazů z této kapitoly jsou pro SQLite stejné, snad jen s tím rozdílem, že DROP TABLEneumí smazat 2 tabulky najednou.

Metapříkaz pro vypsání všech tabulek je **.tables** a pro vypsání CREATE TABLE statementu je **.schema tablename**.

Pozor! Za tablename nedávejte středník. SQLite by nevyhodil žádnou chybu, ale ani by nezobrazil CREATE TABLE statement.

Oracle

I Oracle má své datové typy. Místo varchar byste měli používat varchar2 (tahle podivnost je tam z historických důvodů). Místo numeric(n,m) používá Oracle number(n,m) (numeric je ale funkční synonymum pro number). Oracle nemá datový typ „serial", ačkoliv používá pro primární klíče sekvence, podobně jako PostgreSQL.

Typ DATE uchovává jak datum tak i čas (na vteřiny), type TIMESTAMP navíc ukládá zlomky sekund. Žádný datový typ, který by uchovával pouze datum nebo pouze čas není.

Místo text má Oracle **clob** (do kterého se vejdou 4 GiB textu). Mimo další základní datové typy, jako je integer či float má i speciality jako třeba xml atp.

Datový typ boolean Oracle nezná ani jako synonymum, takže asi nepřekvapí, že nezná ani true a false. Místo boolean můžete používat integer, nebo třeba char, který omezíte integritním omezením CHECKjen na znaky 't' a 'f'. Pokud použijete integer, můžete jej omezit pomocí CHECK na hodnoty 0 a 1. Fantazii se meze nekladou. O podmínce CHECK bude řeč později, zatím ji můžete s klidným svědomím vynechat. PS: Totéž platí i pro SQLite.

SQL příkazy z příkladů pro Postgres budou v Oracle fungovat beze změny (jen místo VARCHAR používejte VARCHAR2).

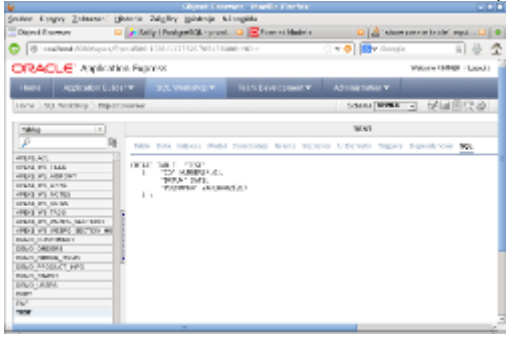
V Oracle je největší legrace s daty. Defaultní formát pro oracle vypadá tatko: '13-NOV-92' je 13 listopad 1992 a '23-NOV-13' je 23 listopad 2013. Oracle očekává, že mu budete dávat data v tomto formátu. To není úplně nejšikovnější. Očekávaný formát se dá navíc přenastavit a bývá přenastaven podle lokalizace vašeho operačního systému. Jistější je proto používat funkce TO_DATE a TO_CHAR (vždy).

```
CREATE TABLE test (id INT, datum DATE, poznamka VARCHAR2(20));
INSERT INTO test VALUES (1, '13-NOV-92', '13 listopad 1992');
ORA-01843: not a valid month
INSERT INTO test VALUES (1, TO_DATE('1992-11-13','YYYY-MM-DD'), '13 listopad 1992');
SELECT * FROM test;
        ID  DATUM       POZNAMKA
---------- ---------  --------------------
         1 11/13/1992  13 listopdad 1992

INSERT INTO test VALUES (2, '11/23/2013', '23 listopad 2013');
SELECT id, TO_CHAR(datum,'YYYY-MM-dd HH24:MI:SS'), poznamka FROM test;
        ID  TO_CHAR(DATUM,'YYYY POZNAMKA
---------- ------------------- --------------------
         2 2013-11-23 00:00:00 23 listopad 2013
         1 1992-11-13 00:00:00 13 listopad 1992
```

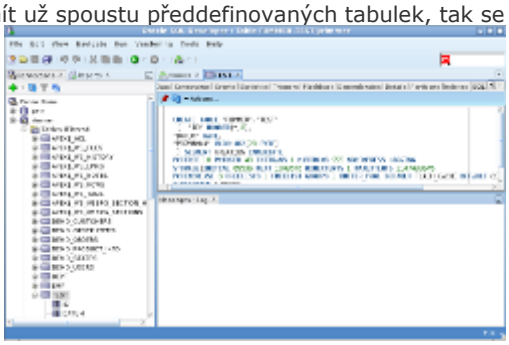
Pokud vás zajímá, jaké všechny formáty můžete s funkcemi TO_DATE a TO_CHAR používat, podívejte se na Datetime Format Elements.

Určitě jste si u posledního SELECTu v příkladu všimli, že jsem místo hvězdičky vyjmenoval sloupečky, které chci zobrazit. A dokonce jsem použil na jeden sloupeček funkci. To jste se zase naučili něco nového :-). Takto si můžete vybrat jen ty sloupečky které chcete vidět, můžete je vypsat v různém pořadí, aplikovat nějaké funkce (jako tady TO_CHAR) a ještě spoustu dalšího toho můžete. Ale o tom všem až v některé z dalších kapitol.



Apex – Object Browser

Pokud si budete chtít prohlédnout tabulky v databázi, jak tabulka vypadá a CREATE TABLE statement pro tabulku, všechno to najdete v apexu pod záložkou SQL Workshop → Object Browser.
Budete tam mít už spoustu předdefinovaných tabulek, tak se nelekněte :-).



Oracle SQL Developer

Najít si seznam tabulek a jejich detail v Oracle SQL Developeru máte za domácí úkol. (Nápověda: použijte dvojklik).
PS: SQL dotaz pro získání seznamu všech dostupných tabulek je následující (zobrazí se i různé systémové tabulky, takže těch tabulek bude hodně):

```
SELECT owner, table_name FROM all_tables WHERE owner = 'RIMMER';
```

Všiměte si, že jméno uživatele (RIMMER) je velkými písmeny.

Úprava tabulky

V úvodu minulé kapitoly jsem psal, že už budete téměř schopni používat databáze v reálném životě. Tak teď už to bude bez toho téměř.
Nejdřív vám ukážu šikovnou podmínku WHERE (česky kde), pak příkazy na úpravu dat v tabulce DELETE a UPDATE a nakonec představím příkaz na úpravu samotné definice tabulky ALTER TABLE.

Příprava

V této kapitole budu příkazy vysvětlovat na nové tabulce. Nejdřív ji tedy musíte vytvořit a vložit do ní pár řádků. K tomu využijte již známé příkazy CREATE TABLE a INSERT INTO. Přihlašte se k vaší databázi rimmer1 (nebo jak jste si ji pojmenovali) a zadejte následující příkazy:

```
CREATE TABLE dluznici (
jmeno VARCHAR(10),
prijmeni VARCHAR(15),
dluh_kc NUMERIC(8,1),
zadluzen DATE
);
```

Vytvoří se tabulka se jménem dluznici, která obsahuje sloupce se jménem a příjmením dlužníka, výší dluhu a datum, kdy se dlužník zadlužil. Jak vidíte, dluh je číslo s jedním desetinným místem (halíře) a maximální dluh může být 9 999 999,9. Můžete dokonce zadat i dluh záporný (to když si od někoho půjčíte). Věřím, že nebudete nikomu půjčovat více jak 10 miliónů :-). Pokud mou víru nesdílíte, můžete si rozšířit velikost datového typu NUMERIC u sloupečku dluh_kc dle libosti.
Naplňte tabulku těmito hodnotami:

```
INSERT INTO dluznici VALUES ('Martin', 'Doktor', 5000, '2002-09-20');
INSERT INTO dluznici VALUES ('Linus', 'Torvalds', 6030, '2002-09-21');
INSERT INTO dluznici VALUES ('Jan Jakub', 'Ryba', 10.5, '2002-09-21');
INSERT INTO dluznici VALUES ('Martin', 'Ryba', 1000, '2002-09-21');
```

*Chcete-li si hodnoty v tabulce prohlédnout (že tam opravdu jsou), použijte příkaz SELECT * FROM dluznici;.*

Podmínka WHERE

Podmínka **WHERE** slouží k omezení řádků, na které se aplikuje SQL příkaz. U kterých všech SQL příkazů se podmínka WHERE dá použít se dozvíte časem. Zatím ji ukážu na příkazu SELECT.
Řekněme, že chci vypsat jen ty dlužníky, kteří mi dluží 5000 Kč a více. Na konec příkazu SELECT přidám podmínku WHERE:

```
rimmer1=> SELECT * FROM dluznici WHERE dluh_kc >= 5000;
 jmeno  | prijmeni | dluh_kc |  zadluzen
--------+----------+---------+------------
Martin | Doktor   |  5000.0 | 2002-09-20
Linus  | Torvalds |  6030.0 | 2002-09-21
(2 rows)
```

Jestli napíšete podmínku ve tvaru sloupec >= hodnota nebo hodnota <= sloupec je jedno, ale to, kde napíšete příkaz WHERE, již jedno není. Nemůžete napsat *WHERE dluh_kc >= 5000 SELECT * FROM dluznici;*.
Jsem trochu v pokušení vám napsat, abyste se podívali na metapříkaz \h SELECT, abyste viděli co a v jakém pořadí se v tomto příkazu dá použít. Jenomže SELECT toho umí tolik, o čem jste ještě neslyšeli, že by vám z toho šla akorát hlava kolem.

Podmínky použitelné v klauzuli WHERE

Podmínka	Užití	Význam
=	sloupec = hodnota	Vyberou se ty řádky, které mají v *sloupci* zadanou *hodnotu*
<>	sloupec <> hodnota	Vyberou se ty řádky, které nemají v *sloupci* zadanou *hodnotu*
!=	sloupec != hodnota	To samé jako <>, jen jinak zapsáno.
<	sloupec < hodnota	Vyberou se ty řádky, které mají v *sloupci* menší hodnotu, než je *hodnota* v podmínce.
>	sloupec > hodnota	Vyberou se ty řádky, které mají v *sloupci* větší hodnotu, než je *hodnota* v podmínce.
<=	sloupec <= hodnota	Vyberou se ty řádky, které mají v *sloupci* menší nebo stejnou hodnotu, jako je *hodnota* v podmínce.
>=	sloupec >= hodnota	Vyberou se ty řádky, které mají v *sloupci* větší nebo stejnou hodnotu, jako je *hodnota* v podmínce.
IS NULL	sloupec IS NULL	Vyberou se ty řádky, které mají v *sloupci* hodnotu NULL
IS NOT NULL	sloupec IS NOT NULL	Vyberou se ty řádky, které nemají v *sloupci* hodnotu NULL
LIKE	sloupec LIKE 'retezec'	Vyberou se ty řádky, kde ve sloupci (s datovým typem řetězec) odpovídá hodnota řetězci za klíčovým slovem LIKE. Tento řetězec může obsahovat dva zástupné znaky: **%** (procento) zastupuje řetězec libovolné délky (i délky 0) s libovolnými znaky. _ (podtržítko) zastupuje právě jeden libovolný znak. Příklad: podmínka WHERE sloupec LIKE '_%A' vybere ty sloupce, kde je řetězec končící písmenem velké A a který má před ním minimálně jeden (libovolný) znak.
IN (ALL, ANY…)	sloupec IN (*množina hodnot*)	Tyto podmínky již nejsou tak triviální a budou vysvětleny v kapitole SELECT IV - poddotazy. Zatím se jimi nemusíte zabývat.

Pro pořádek připomínám, že hodnota NULL se ničemu nerovná. Proto jsou potřeba podmínky IS NULL a IS NOT NULL).

AND

Chcete-li pro výběr řádků použít více podmínek najednou, spojte je klíčovým slovem **AND**. Vyberou se jen ty řádky, které splňují obě podmínky.
Příklad: Vyberte všechny záznamy o dluzích všech Jakubů Rybů, které přesahují částku 10 Kč.

```
rimmer1=> SELECT * FROM dluznici WHERE jmeno LIKE '%Jakub%' AND
prijmeni = 'Ryba' AND dluh_kc > 10;
 jmeno  | prijmeni | dluh_kc |  zadluzen
----------+----------+---------+------------
Jan Jakub | Ryba     |    10.5 | 2002-09-21
(1 row)
```

Kontrolní otázka: Proč jsem použil v podmínce jmeno LIKE '%Jakub%' a ne jmeno = 'Jakub'?

OR

Pomocí klíčového slova OR vyberete ty řádky, které splňují alespoň jednu z podmínek spojených tímto klíčovým slovem.
Spojení OR a AND můžete používat spolu. Jejich prioritu je nejjistější určit závorkami.
Příklad: Vyberte všechny záznamy Rybů a Linuse Torvaldse, kteří mají dluhy vyšší než 500 Kč.

```
rimmer1=> SELECT * FROM dluznici WHERE (prijmeni = 'Ryba'
OR (jmeno = 'Linus' AND prijmeni = 'Torvalds')) AND dluh_kc > 500;
 jmeno  | prijmeni | dluh_kc |  zadluzen
--------+----------+---------+------------
Linus  | Torvalds |  6030.0 | 2002-09-21
Martin | Ryba     |  1000.0 | 2002-09-21
(2 řádky)
```

Kontrolní otázka: Co by se stalo, kdybych odstranil vnější závorky z SQL z příkladu? (Nápověda: AND má vyšší prioritu než OR a pokud nejsou závorky, rozhoduje priorita …)
Kontrolní otázka: Stalo by se něco, kdybych odstranil vnitřní závoky z SQL z příkladu?

NOT

NOT neguje logický výraz. Má ještě větší prioritu než AND.

Příklad: Vyberte všechny záznamy o dluzích všech Rybů, kteří se nejmenují Jakub, a které přesahují částku 10 Kč.

```
SELECT * FROM dluznici WHERE jmeno NOT LIKE '%Jakub%' AND
prijmeni = 'Ryba' AND dluh_kc > 10;
```

Příklad: Vyberte všechny záznamy, kde je dluh mezi 1000 a 5000 Kč včetně.

```
SELECT * FROM dluznici WHERE dluh_kc >= 1000 AND dluh_kc <= 5000;
SELECT * FROM dluznici WHERE NOT dluh_kc < 1000 AND NOT dluh_kc > 5000;
SELECT * FROM dluznici WHERE NOT (dluh_kc < 1000 OR dluh_kc > 5000);
```

Podmínka BETWEEN x AND y je snad samovysvětlující.

Příklad: Vyberte všechny záznamy, kde je dluh mezi 1000 a 5000 Kč včetně.

```
SELECT * FROM dluznici WHERE dluh_kc BETWEEN 1000 and 5000;
```

Jak jste mohli vidět z posledních dvou příkladů, v SQL můžete dojít k jednomu výsledku mnoha cestami.BETWEEN je tu jen pro to, aby se podmínka WHERE mohla zapsat trošilinku přehledněji.

Jedna dobrá rada na závěr: Pokud používáte v klauzuli WHERE více logických výrazů s AND, OR či NOT, **určujte prioritu pomocí závorek**. Nejen že tím zamezíte nechtěným chybám, ale také se to lépe čte a tak i lépe upravuje (když je potřeba).

DELETE FROM

Jistě jste si již položili otázku, jak řádky z tabulky smazat. Smazat celou tabulku a pak tam vložit zase všechno, kromě toho co tam nechceme mít, není úplně nejpohodlnější cesta.

K mazání řádků slouží příkaz DELETE. Za klíčovými slovy **DELETE FROM** následuje název tabulky, klíčové slovo WHERE s *podmínkami*. Podmínky určí, které řádky se smažou.

Pokud zadáte příkaz bez podmínky, tj. jen DELETE FROM nazev_tabulky;, smažou se všechny řádky! Bacha na to! (Tabulka však existuje dále, tu zrušíte jen příkazem DROP TABLE).

Příklad: Odpusťte dluh všem, kteří vám dluží méně jak 10 Kč.

```
rimmer1=> DELETE FROM dluznici WHERE dluh_kc < 10;
DELETE 0
```

Při úspěšném provedení příkazu DELETE se vám ukáže na obrazovce DELETE *n*, kde *n* je počet smazaných řádků.

Jelikož v tabulce nebyl nikdo s dluhem menším než 10 Kč, nebyl smazán žádný řádek.

Teď už pro vás určitě nebude problém smazat všechny dluhy Martina Ryby.

```
rimmer1=> DELETE FROM dluznici WHERE jmeno = 'Martin'
AND prijmeni = 'Ryba';
DELETE 1
```

UPDATE

Řádky můžete nejenom vytvářet (příkazem INSERT) nebo mazat (příkazem DELETE), ale také měnit hodnoty ve sloupcích řádku (nebo několika řádek najednou). K tomu slouží příkaz **UPDATE**.

Za klíčovým slovem UPDATE následuje název tabulky, klíčové slovo SET a názvy sloupců = hodnota. Potom můžete určit podmínkou WHERE kterým řádkům se mají hodnoty ve sloupcích změnit. Pokud podmínku WHERE neuvedete, změny se provedou ve všech řádcích tabulky. (Vidíte, už umíte třetí SQL příkaz, který používá WHERE. Toho byste se ještě dneska ráno nenadáli, že?).

Vaše tabulka dlužníci by měla momentálně vypadat takhle:

```
rimmer1=> SELECT * FROM dluznici;
  jmeno   | prijmeni | dluh_kc |  zadluzen
----------+----------+---------+------------
Martin    | Doktor   |  5000.0 | 2002-09-20
Linus     | Torvalds |  6030.0 | 2002-09-21
Jan Jakub | Ryba     |    10.5 | 2002-09-21
(3 rows)
```

Řekněme že Vám Martin Doktor vrátil 2200 Kč a dluží Vám tedy jen 3000 (úroky jsou sviňa). Hodnotu jeho dlužné částky změníte takto:

```
UPDATE dluznici SET dluh_kc = 3000 WHERE prijmeni = 'Doktor';
```

Pozor! Pokud byste měli více záznamů o dluhu Martina Doktora, nastavili byste je **všechny** na 3000! Proto by bylo ještě rozumné přidat podmínku:

... AND zadluzen = '2002-09-20'; nebo **... AND dluh_kc = 5000;** (nebo obě najednou).

Pokud byste z nějakého důvodu uložili do tabulky několik dluhů jednoho člověka v jeden den, a několik těchto dluhů by bylo 5000 kč, už byste byli nahraný! Jakou podmínkou určit správný řádek?

Takový problém se dá řešit buď zavedením primárho klíče nebo sloupcovým omezením UNIQUE (oboje proberu později).

SQL umí řešit i jednoduché matematické výrazy. Chcete-li například odpustit všem 20 Kč dluh, pak nejdříve smažete všechny řádky kde je dluh menší než 20Kč, ale větší než 0 (své dluhy, které si označujete zápornou hodnotou dluhu, nemůžete odpustit). Poté dosadíte do sloupce dluh_kc hodnotu **dluh_kc-20** pro řádky, kde je dluh větší než 0.

Zní to složitě, ale příklad je jednoduchý:

```
DELETE FROM dluznici WHERE dluh_kc < 20 AND dluh_kc > 0;
UPDATE dluznici SET dluh_kc = dluh_kc-20 WHERE dluh_kc > 0;
```

Následující příklad není příliš smysluplný, je tu jen pro ukázku **změny více sloupců najednou**. Co tento příkaz provede si jistě domyslíte.

```
UPDATE dluznici SET dluh_kc = dluh_kc/2+100, zadluzen = '2002-09-22';
```

ALTER TABLE

Příkaz ALTER TABLE slouží k úpravě **definice tabulky** (mění její název, název sloupečků, přidává nebo maže sloupečky atd.). Upravovat definici tabulky asi nebudete často, ale někdy se to hodí. Pamatujte, že u tabulek s velkým množstvím záznamů (milióny a milióny) může ALTER TABLE trvat dlouho. DBMS musí někdy kvůli změně definice tabulky přesouvat všechny data na disku.

Teď vám ukážu nejzajímavější možnosti využití tohoto příkazu, které se vám mohou hodit. Později se k ALTER TABLE ještě vrátím.

Změna jména tabulky

Přejmenování tabulky je jednoduché:

```
ALTER TABLE dluznici RENAME TO dluhy;
```

Tabulka dluznici se přejmenovala na dluhy.

Změna jména sloupce

V tabulce se jménem dluhy změním jméno sloupce dluh_kc na jméno dluh.

```
ALTER TABLE dluhy RENAME dluh_kc TO dluh;
```

Vytvoření nového sloupce

Řekněme, že se například rozhodnete dát svým dlužníkům časový limit, do kdy vám mají dluh vrátit. Vytvoříte tedy nový sloupec s názvem navrat_dluhu. Tabulka dluhy by teď měla vypadat takto:

```
rimmer1=> \d dluhy
                Tabulka "public.dluhy"
  Sloupec  |         Typ          | Modifikátory
-----------+----------------------+--------------
 jmeno     | character varying(10) |
 prijmeni  | character varying(15) |
 dluh      | numeric(8,1)          |
 zadluzen  | date                  |
```

Přidám nový sloupec navrat_dluhu, který bude typu DATE.

```
rimmer1=> ALTER TABLE dluhy ADD navrat_dluhu DATE;
ALTER
rimmer1=> SELECT * FROM dluhy;
 jmeno  | prijmeni |  dluh  |  zadluzen  | navrat_dluhu
--------+----------+--------+------------+--------------
 Linus  | Torvalds | 3105.0 | 2002-09-22 |
 Martin | Doktor   | 1590.0 | 2002-09-22 |
(2 řádky)
```

Jak vidíte, přibyl nám nový sloupec, který je však celý prázdný (přesněji řečeno, obsahuje pro všechny řádky hodnotu NULL).

Příkazem UPDATE dám všem dlužníkům čas 30 dní na splacení dluhu:

```
rimmer1=> UPDATE dluhy SET navrat_dluhu = zadluzen + 30;
UPDATE 2
rimmer1=> SELECT * FROM dluhy;
 jmeno  | prijmeni | dluhy  |  zadluzen  | navrat_dluhu
--------+----------+--------+------------+--------------
Linus   | Torvalds | 3105.0 | 2002-09-22 | 2002-10-22
Martin  | Doktor   | 1590.0 | 2002-09-22 | 2002-10-22
(2 rows)
```

Všimněte si, jak jsem použil hodnoty z jednoho sloupce pro nastavení hodnot v jiném sloupci. Samá kouzla, samá magie …

Poznámka pro pokročilé: pokud chcte upravit datum o jeden měsíc (ne každý měsíc má 30 dní), můžete to v PostgreSQL udělat takto:

```
UPDATE dluhy SET navrat_dluhu = zadluzen + interval '1 month';
```

Aritmetiku s časem proberu v některé z dalších kapitol.

Odstranění sloupce

Poslední, co vám chybí ke štěstí, je příkaz na odstranění sloupce z tabulky. Tím se také nenávratně ztratí všechna data ve sloupci obsažená).

```
ALTER TABLE dluhy DROP COLUMN navrat_dluhu;
```

Další informace o ALTER TABLE najdete v kapitole Úprava tabulky - pro pokročilé.

MySQL/MariaDB

V MySQL je všchno stejné, krom přejmenování sloupečku a práce s datem.

Sloupec se nedá jednoduše přejmenovat, můžete jej jen „znovuvytvořit" (změnit jeho definici včetně jména). A pokud při jeho znovuvytvoření změníte pouze jméno …

```
ALTER TABLE dluhy CHANGE COLUMN dluh_kc dluh decimal(8,1);
```

V první části SQL příkazu se říká, že chcete změnit sloupeček dluh_kc a v druhé části na co ho chcete změnit (dluh decimal(8,1);). Změněný sloupeček si ponechá své hodnoty, pokud je to možné. (Pokud změníte typ sloupce z čísla ne text tak to možné je, opačně je to už horší …)

A ano, tušíte správně, můžete takto změnit typ sloupečku (třeba na decimal(10, 1)), ale to už předbíhám o pár lekcí dopředu.

Když se pokusíte k DATE přičíst číslo, MySQL ho nechápe jako počet dní, jako PostgreSQL. Musíte na to jít trochu jinak:

```
UPDATE dluhy SET navrat_dluhu = DATE_ADD(zadluzen, INTERVAL +30 DAY);
```

INTERVAL 30 day by taky šlo, ale INTERVAL +30 DAYS už je chyba.

SQLite

U SQLite platí všechno stejně jako u PostgreSQL, krom úpravy datumu a příkazu ALTER TABLE.

Přičtení čísla k datumu na rozdíl od PostgreSQL nepřičte 30 dní. SQLite typ DATE je totiž ve skutečnosti TEXT. SQLite se při sčítání pokusí převést text na číslo, sečíst a výsledek uložit.

Pro aritmetiku s daty se v SQLite používá funkce **date**.

```
sqlite> select zadluzen + 30, date(zadluzen,'+30 days') FROM dluhy;
zadluzen + 30  date(zadluzen,'+30 days')
-------------  ------------------------
2032           2002-10-22
2032           2002-10-22
```

```
UPDATE dluhy SET navrat_dluhu = date(zadluzen, '+1 month');
```

Při letmém pohledu do dokumentace SQLite k ALTER TABLE zjistíte, že ALTER TABLE umí jen přejmenovat tabulku a přidat sloupec. (Zkuste se zamyslet nad tím obrázkem v dokumentaci, není to tak těžké pochopit.)

Přejmenování i smazání sloupece jde udělat, jen je na to potřeba takový malý trik. Ukážu vám novou verzi INSERT statemetu, a to **INSERT INTO ... SELECT**, který vloží do jedné tabulky data z jiné tabulky.

Postup bude následující:

1. Vytvořte novou tabulku, která bude mít požadované sloupce.
2. Zkopírujte data z původní tabulky do té nové pomocí **INSERT INTO ... SELECT**
3. Smaže původní tabulku
4. Přejmenujte novou tabulku na jméno původní tabulky.

```
CREATE TABLE dluhy_new (
jmeno VARCHAR(10),
prijmeni VARCHAR(15),
```

```
    dluh NUMERIC(8,1),
    zadluzen DATE,
    navrat_dluhu DATE
);
INSERT INTO dluhy_new SELECT jmeno, prijmeni, dluh_kc, zadluzen, NULL FROM dluhy;
UPDATE dluhy_new SET navrat_dluhu = date(zadluzen, '+1 month'); -- (krok navic)
DROP TABLE dluhy;
ALTER TABLE dluhy_new RENAME TO dluhy;
```

*Ve výrazu **INSERT INTO ... SELECT** musí být v SELECTu vyjmenované všechny sloupečky a ve stejném pořadí, jako jsou definované v tabulce, do které se INSERTuje. Do nového (posledního) sloupečku se proto insertuje NULL.*

Poznámka: INSERT a UPDATE by šli nahradit jedním INSERTem:

```
INSERT INTO dluhy_new SELECT jmeno, prijmeni, dluh_kc, zadluzen, date(zadluzen, '+1 month') FROM dluhy;
```

Oracle

V Oracle musíte pro práci s časem používat vám už známou funkci TO_DATE.

```
INSERT INTO dluznici VALUES ('Martin', 'Doktor', 5000, TO_DATE('2002-09-20','YYYY-MM-DD'));
INSERT INTO dluznici VALUES ('Linus', 'Torvalds', 6030, TO_DATE('2002-09-21','YYYY-MM-DD'));
INSERT INTO dluznici VALUES ('Jan Jakub', 'Ryba', 10.5, TO_DATE('2002-09-21','YYYY-MM-DD'));
INSERT INTO dluznici VALUES ('Martin', 'Ryba', 1000, TO_DATE('2002-09-21','YYYY-MM-DD'));
UPDATE dluznici SET dluh_kc = dluh_kc/2+100, zadluzen = to_date('2002-09-22','YYYY-MM-DD');
```

Přejmenování sloupce vyžaduje použití RENAME COLUMN.

```
ALTER TABLE dluhy RENAME COLUMN dluh_kc TO dluh;
```

Jak vidíte, syntaxe pro přejmenování je stejná jako v Postgresu, jen je tam navíc slovíčko COLUMN. Ale to můžete použít i v PostgreSQL! Rozdíl mezi Oracle a PostgreSQL je v tom, že Oracle slovíčko COLUMN vyžaduje, zatímco v Postgresu je nepovinné. Doporučení zní: používejte RENAME COLUMN (rázem se stane SQL příkaz přenositelnější mezi oběma DBMS).

Přičtení čísla k typu DATE se chová v Oracle stejně jako v PostgreSQL – přičte se to jako počet dní.

Pro přičtení měsíců má Oracle funkci ADD_MONTHS:

```
UPDATE dluhy SET navrat_dluhu = add_months(zadluzen, 1);
```

Sloupcová omezení, Indexy

Pokud vytváříte tabulku (příkazem CREATE TABLE), můžete mít na data, která se budou ukládat do sloupců, speciální požadavky. Nejdůležitější z nich proberu v této kapitole a v kapitole o relacích.

V této kapitole se bude vytvářet hodně tabulek, které už později nebudou potřeba, tak si je na konci můžete smazat příkazem DROP TABLE.

Integritní omezení

Integritní omezení (Integrity constraints) je správnější název než soupcová omezení. Existují totiž omezení, která se netýkají jen hodnot (jednoho) sloupce, ale i vazeb mezi sloupci. (Viz relace v další kapitole).

Integritní omezení říkají, jak mají vypadat data ukládaná do tabulky (sloupce).

Mezi integritní omezení patří (neformálně) už datový typ sloupce. Když si vytvoříte datový sloupec typu INTEGER, už do něj nemůžete vložit text. (O to se stará DBMS.)

Formálně je integritním omezením i defaultní hodnota (ikdyž to ve skutečnosti není žádné omezení). Defaultní (implicitní) hodnota říká, co se má uložit za hodnotu do sloupečku, když při INSERTU hodnotu sloupečku neurčíte explicitně.

Další **constrainty** jsou: NOT NULL, UNIQUE, CHECK, a pro relaci PRIMARY KEY (primární klíč) a FOREIGN KEY (cizí klíč).

Defaultní hodnoty

Defaultní hodnota sloupce je taková hodnota, která se do sloupce dosadí automaticky, pokud neurčíte jeho hodnotu. Pokud defaultní hodnotou neurčíte, ani neurčíte hodnotu sloupce v příkazu INSERT, je automaticky dosazena hodnota NULL (NULL je defaultní hodnota pro defaultní hodnotu :-).

Defaultní hodnoty vám mohou ušetřit trochu toho psaní a navíc vám zajistí, že ve sloupci nebudete mít tu ošklivou hodnotu NULL. (Typicky datový typ BOOL většinou není dobré nechávat nevyplněný).

Příklad: Dejme tomu, že pravidelně sázím sportku za 60 Kč. Vytvořím si tabulku, kde si budu ukládat, kolik jsem vsadil, kdy a kolik jsem vyhrál. Záznam o sázce vložím vždy ten den, kdy vsadím. Záznam o výhře budu měnit dodatečně až po tahu sportky. Vytvořím tedy defaultní hodnotu pro sázku (60 Kč), ale i pro výhru (0 Kč). Když pak nevyhraji, nemusím se obtěžovat záznam měnit a nebudu mít nikde ve sloupci výher hodnotu NULL (pokud si jí tam schválně nevložím :-)). Protože sázky i výhry budou v celých korunách, použiji jako typ celé číslo (integer).

Poznámka pro později narozené: Tenhle tutoriál vznikal v době, kdy ještě existovali halíře :-).

```
CREATE TABLE sportka (
    datum DATE,
    vsazeno INTEGER DEFAULT 60,
    vyhra INTEGER DEFAULT 0
);
```

Klíčové slovo DEFAULT s hodnotou se píší vždy až za datový typ sloupečku.

Doposud jsem při INSERTu vkládal vždy hodnoty pro všechny sloupečky tabulky. Aby měla hodnota DEFAULT nějaký smysl, musí existovat způsob, **jak vložit hodnoty jen pro některé sloupečky**. A ten teď ukážu v příkladu: za jménem tabulky se do závorek vypíší jména sloupců, do kterých chci vložit nějakou hodnotu explicitně (do ostatních se vloží DEFAULT hodnota).

Nyní vložím do tabulky *sportka* několik „sázek“:

```
INSERT INTO sportka(datum) VALUES (TO_DATE('2002-09-22','YYYY-MM-DD'));
INSERT INTO sportka(datum) VALUES (TO_DATE('2002-09-23','YYYY-MM-DD'));
INSERT INTO sportka(datum, vsazeno) VALUES (TO_DATE('2002-09-24','YYYY-MM-DD'), 120);
```

Nechme stranou, že jsem tak trochu gambler. Výsledek je:

```
rimmer1=> SELECT * FROM sportka;
   datum    | vsazeno | vyhra
------------+---------+-------
 2002-09-22 |      60 |     0
 2002-09-23 |      60 |     0
 2002-09-24 |     120 |     0
(3 rows)
```

Defaultní hodnotou může být také **výraz**. Opět ukážu na příkladě:

Představte si, že vedete malý obchod s cukrovím. Jedno ze zboží, které objednáváte, je „Sladký rohlík“. Většinou jich objednáváte 100 Kusů a jejich trvanlivost je 5 dní. Ale pokud si je objednáte ve speciálním balení, bude jejich trvanlivost delší. Musíte také vědět, kdo vám rohlíky dovezl (kvůli případné reklamaci). Váš obvyklý závozník se jmenuje Tomáš Tlustý (ale někdy se může stát, že potřebujete větší množství a tak si objednáte i u někoho jiného). Přehled o zboží si pak budete udržovat v takovéto tabulce:

```
CREATE TABLE sladky_rohlik (
zavoznik VARCHAR(30) DEFAULT 'Tomáš Tlustý',
pocet INTEGER DEFAULT 100,
dovezeno DATE DEFAULT current_date,
spotrebovat_do DATE DEFAULT current_date+5
);
INSERT INTO sladky_rohlik(pocet) VALUES(100);
```

*Poznámka: **current_date** je funkce, která vrací hodnotu aktuálního datumu. Funkcemi se budu zabývat později.*

Bohužel nemůžete ve výrazu použít hodnotu sloupce (nejde použít jako defaultní hodnotu ve sloupci spotrebovat_do dovezeno + 5). Další malou „nepříjemností“ je to, že příkaz INSERT musí obdržet jako argument hodnotu alespoň jednoho sloupce (i když mají všechny sloupce svou defaultní hodnotu). Vyzkoušejte.

Dobrá zpráva je, že můžete pro hodnotu sloupce použít klíčové slovo **DEFAULT**. Je to určitě lepší, než psát explicitně defaultní hodnotu, zvlášť když tento SQL dotaz používáte častěji (máte ho třeba někde uložený) a defaultní hodnota se v tabulce může změnit.

```
INSERT INTO sladky_rohlik(pocet) VALUES(DEFAULT);
rimmer1=> SELECT * FROM sladky_rohlik;
 zavoznik   | pocet |  dovezeno  | spotrebovat_do
------------+-------+------------+----------------
Tomáš Tlustý |   100 | 2013-11-28 | 2013-12-03
Tomáš Tlustý |   100 | 2013-11-28 | 2013-12-03
(2 řádky)
```

Více o DEFAULT.

Podmínka CHECK

Za klíčovým slovem CHECK následuje podmínka, podobně jako u WHERE (Jen si odmyslete podmínky s NULL, IN a ALL). Podmínka se uvádí za klíčovým slovem CHECK v závorce. Podmínka (constraint) určí, jaké hodnoty budete moci do tabulky vložit (pouze takové, co podmínce vyhoví).

Příklad: budete udržovat databázi zboží na skladě. Na zboží máte následující požadavky:

1. Nemůžete mít na skladě záporné množství zboží, tak si v databázi určíte podmínku, že množství zboží nesmí být menší než 0.
2. Máte také omezené místo ve skladu, proto nesmí žádné zboží přesáhnout počet 500 kusů (tolik zboží byste ani nestihli do příští objednávky prodat).

Když se pak pokusíte odečíst z databáze více zboží, než máte na skladě, nebo naopak více zboží objednat, než kolik můžete přijmout, nepůjde to.

3. Zboží musí mít také svou cenu.
4. Cena by měla být vyšší než 0.

Tabulka s takovými omezeními může vypadat následovně:

```
CREATE TABLE zbozi (
nazev VARCHAR(20),
pocet INTEGER CHECK (pocet >= 0 AND pocet <= 500),
cena NUMERIC(8,2) CHECK (cena > 0)
);
```

Teď se pokusím vložit nějaké hodnoty do tabulky:

```
rimmer1=> INSERT INTO zbozi VALUES ('Kulomety', 500, 1499.90);
INSERT 0 1
rimmer1=> INSERT INTO zbozi VALUES ('Tarasnice', 0, 500);
INSERT 0 1
rimmer1=> UPDATE zbozi SET pocet = pocet + 100;
ERROR:  new row for relation "zbozi" violates check constraint "zbozi_pocet_check"
DETAIL:  Failing row contains (Kulomety, 600, 1499.90).
rimmer1=> INSERT INTO zbozi VALUES ('Samopaly', 501, 1499.90);
ERROR:  new row for relation "zbozi" violates check constraint "zbozi_pocet_check"
DETAIL:  Failing row contains (Samopaly, 501, 1499.90).
rimmer1=> INSERT INTO zbozi(nazev, pocet) VALUES ('Samopaly', 300);
INSERT 0 1
rimmer1=> INSERT INTO zbozi(nazev, pocet, cena) VALUES ('Samopaly', 300, -1000);
ERROR:  new row for relation "zbozi" violates check constraint "zbozi_cena_check"
DETAIL:  Failing row contains (Samopaly, 300, -1000.00).
```

Všimněte si, jaké si vymyslel Postgres pro „check constraint" názvy.

Záznamy o Kulometech a Tarasnicích jsem vložil bez problémů. Splňovali všechny podmínky. Pokus o zvýšení všech položek o 100 kusů neprošel, neboť jedna položka (Kulomety) by nesplnila sloupcové omezení `pocet <= 500;`. Příkaz UPDATE se neprovedl a **nebyla změněná žádná řádka** v databázi.

```
rimmer1=> SELECT * FROM zbozi;
   nazev   | pocet |  cena
-----------+-------+---------
 Kulomety  |   500 | 1499.90
 Tarasnice |     0 |  500.00
 Samopaly  |   300 |
(3 řádky)
```

Zajímavý je předposlední příkaz INSERT. Po jeho provedení existuje položka, jejíž cena nebyla udána a je tedy NULL. Ale v požadavcích na tabulku bylo, aby byla u každého zboží cena! Tento problém vyřeší podmínka NOT NULL.

Podmínka NOT NULL

Toto sloupcové omezení zabrání vytvoření sloupce v řádku s hodnotou NULL.

Problém s cenou z předchozího příkladu se vyřeší kombinací podmínek CHECK a NOT NULL. Název zboží i počet by také neměli být nikdy prázdné, takže i k nim přidám podmínku NOT NULL.

Tabulku vytvořím znova, proto ji nejdřív smažu:

```
DROP TABLE IF EXISTS zbozi;
CREATE TABLE zbozi (
nazev VARCHAR(20) NOT NULL,
pocet INTEGER CHECK (pocet >= 0 AND pocet <= 500) NOT NULL,
cena NUMERIC(8,2) NOT NULL CHECK (cena >= 0)
);
```

Integritní omezení se píší za datovým typem, ale je jedno v jakém pořadí.

```
rimmer1=> INSERT INTO zbozi(nazev, pocet) VALUES ('Samopaly', 300);
ERROR:  null value in column "cena" violates not-null constraint
DETAIL:  Failing row contains (Samopaly, 300, null).
```

Podmínka UNIQUE

Podmínka UNIQUE zajišťuje, že v sloupci budete mít jen jedinečné hodnoty (pro všechny řádky tabulky). V sloupci tedy nebudou dvě hodnoty stejné, ale může tam být několik položek s hodnotou NULL (nepoužijete-li sloupcové omezení NOT NULL). NULL totiž znamená „nevím co je tam za hodnotu", NULL se ničemu nerovná, proto se bere jako unikátní hodnota.

Ukáži na příkladu, jak se dají sloupcové podmínky kombinovat. Řekněme, že chci mít tabulku, kam smí být proveden jen jeden zápis denně:

```
rimmer1=> CREATE TABLE zapis (
jmeno_zadatele VARCHAR(20) NOT NULL,
text_zapisu TEXT,
    datum_zapisu DATE UNIQUE NOT NULL CHECK (datum_zapisu >= current_date)
DEFAULT current_date
);
NOTICE:  CREATE TABLE / UNIQUE will create implicit index 'zapis_datum_zapisu_key' for table 'zapis'
CREATE TABLE
```

Poznámka: current_date je funkce, která vrací hodnotu aktuálního datumu. Funkcemi se budu zabývat později.

Index a DROP INDEX

Určitě jste si v předchozím příkladě všimli, že se kromě tabulky ještě vytvořil **index**zapis_datum_zapisu_key. Jak si Postgres vytvořil jeho jméno je asi jasné.

Co je index

Index je **nepovinná** datová struktura (databázový objekt) asociovaná s tabulkou. Hlavní cíle indexu jsou:

1. **Zajistit integritu dat**
   - unikátnost (unique, primary key)
   - odkazy (references – cizí klíče)
2. **Umožnit rychlejší přístup k datům**
   - optimalizace rychlosti provádění dotazů
   - snazší řazení dat dle obsahu sloupce

Zatím jediným databázovým objektem se kterým jste přímo pracovali byla tabulka. Tabulka je v databázích to nejdůležitější, kolem čeho se všechno točí. Ale existují i další databázové objekty, jako je například zmíněný index.

Index funguje podobně jako rejstřík na konci knihy. Umožní DBMS rychle najít v datech to, co potřebuje.

Index pro podmínku UNIQUE slouží k tomu, aby mohl Postgres kontrolovat, že jsou data v sloupci datum_zapisu jedinečná. Pomocí indexu DBMS rychle zjistí, zda nová vkládaná hodnota někde ve sloupci už je nebo ne. O to, jak přesně to DBMS s indexem dělá se nemusíte moc starat. Důležité je jen vědět, že index na jednu stranu urychluje práci při hledání dat, na druhou stranu zabírá místo na disku a zpomaluje INSERT a UPDATE (protože se musí aktualizovat nejen tabulka, ale i index). Z tohoto důvodu se nevytváří index u každého sloupečku, ale jenom tam, kde se dá očekávat, že bude potřeba.

Další využití indexů (krom UNIQUE) bude popsáno v dalších kapitolách.

Zrušení indexu

Příkaz pro zrušení indexu je jednoduchý:

```
DROP INDEX nazev_indexu;
```

Ze syntaxe DROP INDEX se dá uhádnout, že název indexu musí být unikátní pro celou databázi (resp. schéma).

Indexy, které máte v databázi, zjistíte metapříkazem **\di**.

```
rimmer1=> \d zapis
                    Tabulka "public.zapis"
     Sloupec     |          Typ          |              Modifikátory
-----------------+-----------------------+-----------------------------------------
 jmeno_zadatele  | character varying(20) | not null
 text_zapisu     | text                  |
 datum_zapisu    | date                  | not null implicitně ('now'::text)::date
Indexy:
```

"zapis_datum_zapisu_key" UNIQUE CONSTRAINT, btree (datum_zapisu)
Kontrolní pravidla:
"zapis_datum_zapisu_check" CHECK (datum_zapisu >= 'now'::text::date)

```
rimmer1=> \di
                        Seznam relací
 Schéma |        Jméno         | Typ  | Vlastník | Tabulka
--------+----------------------+------+----------+---------
 public | zapis_datum_zapisu_key | index | petr     | zapis
(1 řádka)
```

~~Pokud zrušíte index, podmínka UNIQUE bude z tabulky odstraněna.~~
Index nelze smazat, pokud existuje podmínka (constraint), která jej využívá.

```
rimmer1=> DROP INDEX zapis_datum_zapisu_key;
ERROR:  cannot drop index zapis_datum_zapisu_key because constraint zapis_datum_zapisu_key on table zapis requires it
DOPORUČENÍ:  You can drop constraint zapis_datum_zapisu_key on table zapis instead.
```

Jak se dá odstranit podmínka (drop constraint) proberu v kapitole o ALTER TABLE.
Smazáním tabulky se smažou i její indexy.
Další informace lze najít v odstavci o vytváření indexu.

MySQL/MariaDB

Typy tabulek

V kapitole o CREATE TABLE jsem vám zatajil o MySQL jednu důležitou věc. V MySQL existuje několik databázových „strojů" (engine), které se starají o fyziké vykonávání SQL příkazů. Mezi nejpoužívanější patří **MyISAM** a **InnoDB**.
Engine **MyISAM** byl dříve defaultní. Je velmi rychlý rychle ukládá i načítá data, ale **nevynucuje integritní omezení**. Když vytvoříte tabulku s Engine = MyISAM, můžete sice uvádět integritní omezení, ale MyISAM je potichu ignoruje.
Engine **InnoDB** je pomalejší, protože naopak integritní omezení respektuje. Dnes je to v databázi defaultní engine (tj. když při vytváení tabulky neuvedete engine, použije se InnoDB). Defaultní engine se ale dá v databázi nastavit, takže se nemůžete na 100% spolehnout na to, který je defaultní. Proto je rozumné vždycky uvádět, že chcete InnoDB.
MySQL se hodně používá na webových hostinzích, které používali amatéři na své webové stránky. Ti většinou vůbec netušili nic o integritních omezeních, proto bylo zbytečné, aby vytvářeli tabulky s engine InnoDB. Navíc chtěli poskytovatelé ušetřit nějaký ten procesorový čas, proto bylo obyklé používat MyISAM. Dneska už kvalita železa natolik postoupila, že rozdíl v rychlostech enginů není až tak podstatný a mnohem důležitější je integrita dat v databázi.

Změny

V MySQL není funkce TO_DATE, takže INSERTy vypadají jednoduše takto:

```
INSERT INTO sportka(datum) VALUES ('2002-09-22');
INSERT INTO sportka(datum) VALUES ('2002-09-23');
INSERT INTO sportka(datum, vsazeno) VALUES ('2002-09-24', 120);
```

MySQL umí jako defaultní hodnotu aktuálního času použít jen s datovým typem TIMESTAMP, který ukládá datum i čas. Navíc je omezen jen na jednu takovou defaultní hodnotu na tabulku.
Tabulku sladky_rohlik můžete vytvořit takto:

```
CREATE TABLE sladky_rohlik (
zavoznik VARCHAR(30) DEFAULT 'Tomáš Tlustý',
pocet INTEGER DEFAULT 100,
dovezeno TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
spotrebovat_do DATE DEFAULT 0
);
```

Do tabulky se pak musí hodnota do sloupce spotrebovat_do explicitně vložit, jinak v něm bude 0 ('0000-00-00').

```
mysql> INSERT INTO sladky_rohlik(pocet, spotrebovat_do)
VALUES(DEFAULT, DATE_ADD(NOW(), INTERVAL +5 DAY));
Query OK, 1 row affected, 1 warning (0.08 sec)

mysql> SHOW WARNINGS;
+-------+------+-----------------------------------------------------+
| Level | Code | Message                                             |
+-------+------+-----------------------------------------------------+
| Note  | 1265 | Data truncated for column 'spotrebovat_do' at row 1 |
+-------+------+-----------------------------------------------------+
1 row in set (0.00 sec)
```

Funkce NOW() vrací datum i čas. Při ukládání do tabulky se do sloupce spotrebovat_do typu DATE ukládá jen datum, takže čas je „oříznut". Varováním se tak netřeba znepokojovat, je to úplně normální.

TIMESTAMP má ještě jednu zvláštnost. Pokud mu neurčíte defaultní hodnotu, bude mít automaticky jako defaultní hodnotu aktuální čas. Pokud bych chtěl mít v tabulce více typů TIMESTAMP, musí všechny, až na jeden, mít definovanou „statickou" defaultní hodnotu (třeba '2015-12-24').
MySQL vám sice umožní zapsat podmínku CHECK, ale za 1. jí musíte uvést pří definici sloupce až jako poslední (to pro Postgres neplatí) a za 2. jí úplně ignoruje.

```
CREATE TABLE zbozi (
nazev VARCHAR(20),
pocet INTEGER CHECK (pocet >= 0 AND pocet >= 500),
cena NUMERIC(8,2) CHECK (cena > 0)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_czech_ci;

INSERT INTO zbozi VALUES ('Samopaly', 501, 1499.90);
```

INSERT nezpůsobí žádnou chybu.
Jak už jsem psal, CHECK se musí psát v MySQL až za ostatní integritní omezení (a MySQL ho ignoruje). (MySQL umožňuje zapsat CHECK čistě kvůli větší přenositelnosti SQL příkazů mezi různými databázemi.)

```
mysql> DROP TABLE IF EXISTS zbozi;
mysql> CREATE TABLE zbozi (
```

```
nazev VARCHAR(20) NOT NULL,
pocet INTEGER NOT NULL CHECK (pocet >= 0 AND pocet <= 500),
cena NUMERIC(8,2) NOT NULL CHECK (cena > 0)
);

mysql> INSERT INTO zbozi(nazev, pocet) VALUES ('Samopaly', 300);
Query OK, 1 row affected, 1 warning (0.03 sec)

mysql> SHOW WARNINGS;
+---------+------+---------------------------------------+
| Level   | Code | Message                               |
+---------+------+---------------------------------------+
| Warning | 1364 | Field 'cena' doesnt  have a default value |
+---------+------+---------------------------------------+
1 row in set (0.00 sec)

mysql> select * from zbozi;
+----------+-------+------+
| nazev    | pocet | cena |
+----------+-------+------+
| Samopaly |   300 | 0.00 |
+----------+-------+------+
1 row in set (0.01 sec)
```

MySQL si stěžuje, že sloupeček cena nemá definovanou hodnotu. NULL má zakázáno, tak si domyslel 0.00.

Další změna definice tabulky je kvůli omezení TIMESTAMP (jen jeden sloupeček TIMESTAMP může mít v tabulce jako defaultní hodnotu aktuální čas). A CHECK se musí napsat až na konci integritních omezení …

```
CREATE TABLE zapis (
jmeno_zadatele VARCHAR(20) NOT NULL,
text_zapisu TEXT,
datum_zapisu TIMESTAMP UNIQUE NOT NULL
DEFAULT current_timestamp
CHECK (datum_zapisu >= current_date)
);

SHOW CREATE TABLE zapis;
CREATE TABLE `zapis` (
`jmeno_zadatele` varchar(20) COLLATE utf8_czech_ci NOT NULL,
`text_zapisu` text COLLATE utf8_czech_ci,
`datum_zapisu` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
UNIQUE KEY `datum_zapisu` (`datum_zapisu`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_czech_ci;
```

SHOW CREATE TABLE ukazuje dvě zajímavé věci. Za 1. zmizela podmínka CHECK (já vám to říkal že jí MySQL ignoruje) a za 2. podmínka UNIQUE se transformovala do indexu (UNIQUE KEY) jménem datum_zapisu (nad sloupečkem datum_zapisu, který je napsaný na konci v závorkách). Shodnost názvu indexu a jména sloupečku je čistě „náhodná" (existuje způsob, jak si jméno indexu definovat, ale o tom až jindy).

V MySQL mají indexy platnost v rámci tabulky. Takže můžete mít indexy stejného jména, pokud se týkají různých tabulek. Z toho taky plyne, že když pracujete s indexy, musíte říct nejen jaký index (jeho jméno) ale i z jaké tabulky.

```
SHOW INDEXES FROM zapis;
-- výsledkem je jedna dlouhá řádka, takže jí sem celou nevypíšu.
-- zajímavý je jen sloupeček Key_name, kde najdete jméno indexu (datum_zapisu)
+-------+------------+-------------+--------------+-------------+-- ...
| Table | Non_unique | Key_name    | Seq_in_index | Column_name | C ...
+-------+------------+-------------+--------------+-------------+-- ...
| zapis |          0 | datum_zapisu |            1 | datum_zapisu | A ...
+-------+------------+-------------+--------------+-------------+-- ...
```

Smazáním UNIQUE indexu zrušíte integritní omezení UNIQUE.

```
DROP INDEX datum_zapisu ON zapis;

SHOW CREATE TABLE zapis;
CREATE TABLE `zapis` (
`jmeno_zadatele` varchar(20) COLLATE utf8_czech_ci NOT NULL,
`text_zapisu` text COLLATE utf8_czech_ci,
`datum_zapisu` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_czech_ci
```

Všimněte si, že DROP INDEX zmiňuje nejen jaký index, ale i z jaké tabulky smazat.

SQLite

SQLite je na tom kupodivu o mnoho lépe než MySQL. Sice také nemá funkci TO_DATE, ale nedělá mu problém vytvořit dva sloupce s defaultní hodnotou "aktuální čas" a dokonce rozumí podmínce CHECK! (Styď se MySQL, styď se!)

```
INSERT INTO sportka(datum) VALUES ('2002-09-22');
INSERT INTO sportka(datum) VALUES ('2002-09-23');
INSERT INTO sportka(datum, vsazeno) VALUES ('2002-09-24', 120);
```

Podmínka DEFAULT která obsahuje nějaký početní výraz musí být uzavřená v závorkách (ikdyž je to jedna funkce). (To se týká poslední DEFAULT podmínky v následujícím příkladě.)

```
CREATE TABLE sladky_rohlik (
zavoznik VARCHAR(30) DEFAULT 'Tomáš Tlustý',
```

```
    pocet INTEGER DEFAULT 100,
    dovezeno DATE DEFAULT current_date,
    spotrebovat_do DATE DEFAULT (date(current_date,'+5 days'))
);
```

SQLite nerozumí klíčovému slovu DEFAULT při INSERTu (nebo UPDATu):

```
sqlite> INSERT INTO sladky_rohlik(pocet) VALUES(DEFAULT);
Error: near "DEFAULT": syntax error
sqlite> INSERT INTO sladky_rohlik(pocet) VALUES(100);
sqlite> select * FROM sladky_rohlik;
zavoznik        pocet       dovezeno    spotrebovat_do
--------------  ----------  ----------  --------------
Tomáš Tlustý    100         2013-11-28  2013-12-03
```

K zobrazení indexů z tabulky se používá metapříkaz **.indices**:

```
sqlite> .indices zapis
sqlite_autoindex_zapis_1
```

Neukončujte metapříkaz .indices středníkem, SQLite by si myslelo že je součást názvu tabulky.

K mazání indexů se používá DROP INDEX, ale nelze smazat index svázaný s integritním omezením UNIQUE (nebo primarním klíčem):

```
sqlite> DROP INDEX sqlite_autoindex_zapis_1;
Error: index associated with UNIQUE or PRIMARY KEY constraint cannot be dropped
```

Oracle

U Oracle Database moc rozdílů tentokrát není. První drobností je, že DROP TABLE nemá část IF EXISTS.

```
DROP TABLE zbozi;
```

Jak si s tím poradit ukáži příště.

V Oracle neexistuje datový typ TEXT, místo něj použijte **CLOB**. A také byste všude měli používat datový typ **VARCHAR2** místo VARCHAR, který je zastaralý.

Hlavním překvápkem je nemožnost použití current_date v podmínce CHECK. Oracle fylozofie je totiž taková, že **podmínka CHECK musí pro data v tabulce být neustále pravdivá**. Jenže když dáte do podmínky, že musí být datum větší než „dnešní" datum, tak vložená hodnota může dnes podmínce vyhovět, ale za týden už ne.

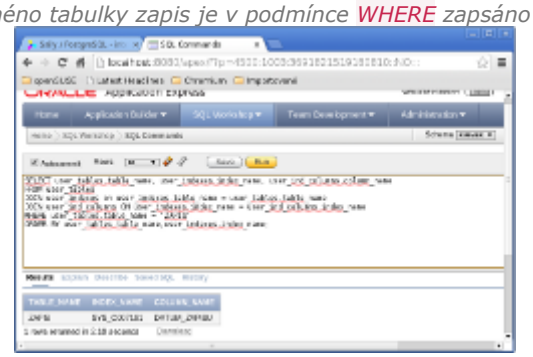
Je to vcelku logický a správný požadavek, jen pro nás uživatele je to trošku pruda. Řešení sice existuje (TRIGGERY mohou automaticky vkládat do sloupečků hodnotu aktuálního času (resp. jakoukoliv hodnotu)), ale triggery jsou zatím nad rámec tohoto kurzu.

```
CREATE TABLE zapis (
    jmeno_zadatele VARCHAR2(20) NOT NULL,
    text_zapisu CLOB,
    datum_zapisu DATE UNIQUE NOT NULL
);
```

Vypsání indexů tabulky taky není v Oracle žádný med. Dá se to udělat následujícím SQL příkazem, který zatím nebudu vysvětlovat a vy se jej nesnažte pochopit (na JOIN příjde řada v některé z dalších kapitol).

```
SELECT user_tables.table_name, user_indexes.index_name,
    user_ind_columns.column_name
FROM user_tables
JOIN user_indexes on user_indexes.table_name = user_tables.table_name
JOIN user_ind_columns ON user_indexes.index_name = user_ind_columns.index_name
WHERE user_tables.table_name = 'ZAPIS'
ORDER BY user_tables.table_name,user_indexes.index_name;
```

Všimněte si, že jméno tabulky zapis je v podmínce WHERE zapsáno velkými písmeny.



Zobrazení indexu v Oracle

Indexy mají platnost v rámci schématu, takže jejich mazání je stejné jako v PostgreSQL. A taky platí, že index pro UNIQUE omezení ani primární klíč nemůžete smazat (můžete odstranit podmínku UNIQUE nebo primární klíč a tím smazat i index, ale o tom zase až později).

```
oracle> DROP INDEX SYS_C007191;
ORA-02429: cannot drop index used for enforcement ofunique/primary key
```

Oracle interně převádí prázdný řetězec '' na hodnotu NULL. Lze sice definovat sloupec takto:

```
sloupec VARCHAR2(255) DEFAULT '' NOT NULL,
```

Ale protože se '' převádí na NULL, nemůže se defaultní hodnota u NOT NULL sloupečku '' nikdy použít, čili je na prd. Nemůžete ani INSERTnout ''. Takže se v Oracle u textových sloupečků (CHAR, VARCHAR2, CLOB) podmínce NOT NULL raději vyhněte.

Oracle je přecitlivělý na pořadí integritních pomínek a defaultních hodnot. Takže:

```
-- toto funguje
sloupec VARCHAR2(255) DEFAULT '-' NOT NULL,
-- toto zahlásí chybu ORA-00907: missing right parenthesis
sloupec VARCHAR2(255) NOT NULL DEFAULT '-',
```

Vytváření relací

V této lekci se konečně dozvíte co jsou to ty **relace** a proč se SQL databázím říká **relační databáze**. Nastudováním této lekce se vaše znalosti o relačních databázích posunou na zcela novou úroveň. Bude to trochu teoretická a docela ukecaná kapitola, ale myslím že nic těžkého na pochopení.

Oprašte si znalost některých pojmů, jako jsou **relace**, **primární klíč** a **cizí klíč**. Budete je teď hodně potřebovat. Spojování tabulek pomocí klíčů je jedna z velkých předností (relačních) databází. V této kapitole se na příkladech pokusím vysvětlit, jak a proč se tabulky spojují.

Většina práce se spojováním tabulek je na bedrech příkazu SELECT. Tento příkaz budu probírat podrobně v dalších a dalších kapitolách, v této kapitole se zatím budu věnovat jen **vytváření relací** mezi tabulkami.

*Příkaz SELECT jste zatím používali k vrácení obsahu celé tabulky, nebo její části – vybírali jste **sloupce** (jejich vyjmenováním za klíčovým slovem SELECT), nebo, pomocí podmínky WHERE, jen některé **řádky**. Brzo vás čeká získávání dat **z více tabulek** najednou jedním SELECTem.*

K čemu je to dobré

Protože jste si zopakovali základní pojmy, jako je relace, primární klíč a cizí klíč, nemusím asi moc vysvětlovat.

Zatím jesm ukazoval jen návrhy jednoduché databáze, kde jsem si vystačil s jednou tabulkou o několika sloupcích. Teď si představte, že jste ve velkém podniku, kde si musíte vést záznamy o svých zaměstnancích, jak dlouho u vás pracují, kolik berou, kdy dostali prémie, co je náplní jejich práce, v jakém jsou oddělení, atd., atp. Musíte si také vést záznamy o pracovních odděleních, kolik tam pracuje lidí, jací lidé, záznamy o zakázkách, které vaše firma dostala, kdo a kdy, na zakázce pracoval, kdo jí zadal …

Nacpat to všechno do jedné tabulky je pochopitelně nepraktické, ne-li nemožné. A asi je vám už jasné, že mezi tabulkami, které navrhnete, jsou různé vztahy. Například vztahem mezi tabulkou oddělení a zaměstnanci bude třeba to, kdo pracuje v jakém oddělení, nebo třeba taky kdo šéfuje danému oddělení.

Trochu konrétněji: První tabulka bude obsahovat **informace o zaměstnancích** (tabulku pojmenuji zamestnanci), se sloupečky pro jméno a příjmení zaměstnance, věk, **v jakém oddělení pracuje**, jeho rodné číslo a cokoli vás ještě bude zajímat), v další tabulce budou **informace o odděleních** (tabulka oddeleni), sloupečky budou např. název oddělení, telefon do oddělení atp.).

Další tabulka by mohla obsahovat **informace o zakázkách** (tabulka zakazky) se sloupečky zákazník, cena zakázky, datum objednávky, datum vyhotovení zakázky, které oddělení na zakázce pracovalo atd. atd.

Ale co to povídám, žádný sloupečky o zákaznících v tabulce o zakázkách nebudou. Pro zákazníky si přece vytvoříte tabulku zakaznici …

Proč to všechno? Abyste si v informacích udrželi pořádek. Zkuste si představit, že byste chtěli mít informace o oddělení a zaměstnancích v jedné tabulce. Každý řádek by musel obsahovat informace o zaměstanci a informace o oddělení ve kterém pracuje a taky informace o oddělení, kterému šéfuje (pokud nějakému). Sloupečků by tam bylo požehnaně a informace o odděleních zduplikované nepočítaněkrát.

Další informace o tom jak správně navrhovat tabulky a udržovat pořádek v datech se dozvíte v povídání o **normalizaci databáze** v nějaké příští kapitole. Teď opustím teorii a ukáži, jak se ty vztahy mezi tabulkami řeší v SQL databázi.

PRIMARY KEY

Jak vložit informace do tabulky zamestnanci o tom, ve kterém oddělení zaměstnanec pracuje? Mohli byste tam vkládat název oddělení. To má ovšem své nevýhody. Může se stát, že bude v podniku více oddělení se stejným názvem. Potom se také může název oddělení změnit a pak je třeba upravit nejen tabulku oddeleni, ale také tabulku zamestnanci a vůbec všechny tabulky, kde se na dané oddělení odkazují.

Tyto problémy se řeší pomocí primárního klíče – evidenčních čísel. Každé oddělení bude mít své **jedinečné**číslo a tím se všechny problémy odstraní. Toto číslo bude primárním klíčem tabulky oddeleni.

Primárním klíčem nemusí být jen číslo. Klidně by to mohl být název oddělení (pokud by byl unikátní), ale práce s čísly je mnohem rychlejší. (Zvláště proto, že se na primární klíč odkazujete z jiné tabulky toutéž hodnotou.)

*Protože se do tabulky zavádí jinak nic neříkající a s oddělením nijak nesouvisející číslo, říká se mu někdy **umělý klíč**. To už ale zase moc ~~terorizuju~~ teoretizuju :-).*

Sloupcové omezení PRIMARY KEY zajišťuje dvě věci. Jednak bude klíč jedinečný, a také (navíc od UNIQUE) se na tento sloupec může odkazovat z jiné tabulky (a všichni skandujeme: **relace relace relace**!) pomocí REFERENCES (cizího klíče). A za třetí, primární klíč nemůže být NULL :-).

Poznámka: Při vytvoření primárního klíče se vytvoří index, který DBMS používá k zajištění výše zmíněných vlastností PK.

Poznámka: tabulka může mít jenom jeden primární klíč (jen na jednom sloupci v tabulce může být integritní omezení PRIMARY KEY).

REFERENCES

REFERENCES je integritní omezení, které říká, že hodnoty ve sloupci budou primárními klíči z jiné tabulky (a z jaké tabulky). Jinak řečeno to říká, že jde o sloupec **cizích klíčů** (foreign keys).

Cizí klíč může být NULL (samosebou jen pokud k sloupečku nepřidáte integritní omezení NOT NULL).

Jak se takové integritní omezení zapisuje? Za klíčovým slovem REFERENCES následuje název tabulky, do které se budeme odkazovat (např. do *oddeleni*) a v závorce za názvem tabulky bude název sloupce s primárními klíči této tabulky:

název a typ sloupečku … **REFERENCES** oddeleni (prim_klic),

Z toho vyplývá, že nejdříve musí existovat tabulka, do které se cizí klíč odkazuje. A musí mít sloupec s primárním klíčem.

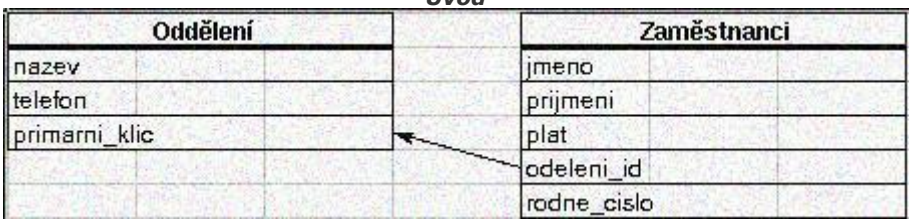
Při vytvoření reference se vytvoří index.

V sloupci s cizími klíči nemusí být jedinečné hodnoty. Samosebou jen pokud nepřidáte podmínku UNIQUE. Určitě bude více zaměstnanců, kteří pracují v jednom oddělení, takže to dělat nebudu, ale jsou situace, kdy se to hodí.

Příklad

Kdybych ukazoval příklady pro jednotlivé odstavce této kapitoly, nebyly by vidět potřebné souvislosti. Všechno tedy osvětlím na jednom „velkém" příkladu. S tímto příkladem pak budu pracovat i v dalších lekcích.

Úvod



Příklad obsahuje dvě tabulky. První, oddeleni, obsahuje primární klíč s názvem prim_klic, druhá, zamestnanci, obsahuje odkaz na sloupec primárních klíčů první tabulky v sloupečku s názvemoddeleni_id.

Všimněte si, že relace mezi tabulkou zaměstnanců a oddělení neříká nic o podstatě této relace. Je zaměstnanec vedoucím daného oddělení? Nebo v něm jen pracuje? Nebo jej chodí uklízet? V tomto příkladě budu relaci považovat za relaci *„zaměstanec pracuje v"*.

*Název prim_klic je dost ujetý název, normálně se používá pro sloupeček s primárním klíčem název **id**. Někdo používá název jako oddeleni_id, takže by se primární klíč v tabulce oddeleni jmenoval stejně jako v tabulce, která se na primární klíč odkazuje (třeba v tabulce zamestnanci). To ale dost prudí při JOINování tabulek (o kterém budu povídat později), takže také někteří chytráci používají název ve tvaru id_oddeleni, což považuju za nejzvrhlejší možnost :-). V zásadě je ale nejdůležitější jen jedno pravidlo: **pojmenovávejte primární a cizí klíče ve všech tabulkách konzistentně** (ne že jednou použijete id, pak id_tabulky, pak id_tabulka a pak tabulka_id …).*

Typ SERIAL a SEQUENCE

Typ SERIAL není vlastně žádný typ. Pokud pro sloupeček použijete SERIAL, ve skutečnosti bude mít sloupeček typ INTEGER a **navíc se vytvoří SEQUENCE**, která se použije pro automatické vyplňování sloupečků číselnou řadou.

SEQUENCE je databázový objekt, který si pamatuje, jaké číslo „vydal" naposledy. Můžete jí vytvořit pomocí příkazu CREATE SEQUENCE.

Sekvenci můžete požádat o další číslo v řadě explicitně, nebo můžete nastavit sloupeček, aby používal sekvenci jako svou defaultní hodnotu.

Když použijete typ SERIAL, vytvoří se sloupeček s typem INTEGER, vytvoří se sekvence a použije se u sloupečku jako defaultní hodnota. Spousta koláčů bez práce :-)

SEQUENCE se používá hlavně s primárními klíči. Když vkládáte řádek a pro primární klíč nevložíte žádnou hodnotu (použijete DEFAULT, nebo sloupeček vynecháte), primární klíč si vezme hodnotu ze sekvence. Tím je zaručeno, že **každý řádek bude mít unikátní hodnotu**.

Představte si, jak by to fungovalo bez sekvence. Dva uživatelé se přihlásí k databázi. Oba se ve stejný okamžik podívají, jaké je maximální ID v databázi (třeba 10). Oba se pokusí vložit řádek s ID o jedno větší (11). První uspěje, druhý selže.

Nebo jiný příklad. Někdo smaže z databáze řádek s nejvyšším ID (třeba 10). K tomuto ID se váže spousta (papírových) dokumentů. Někdo další příjde k databázi, podívá se na nejvyšší ID (které bude 9), zvýší ho o 1 a vloží řádek s ID již smazaného řádku (s ID 10). A všechny papírové dokumenty se najednou vztahují k novému řádku, který za to chudák nemůže. No a to je průs…

Doufám, že už chápete, jak jsou sekvence důležité. Někdy příště se vrátím k tomu, jak vytvářet a používat sekvence jinak, než automaticky s „typem" SERIAL.

Vytvoření tabulek a vložení dat

Všechny SQL příkazy jsem pro vás uložil do souboru psql6.sql, abych vám ušetřil trochu psaní. (Jak jej můžete použít se dočtete v kapitole Začínáme s PostgreSQL).

Skript můžete spouštět kolikrát chcete. Vždycky vám znovu vytvoří tabulky a naplní je daty. Aby se mohli tabulky (znovu) vytvořit, nesmí v databázi existovat (před prvním spuštěním skriptu zřejmě nebudou, ale při druhém už ano). Proto se na začátku souboru psql6.sql mažou tabulky příkazem DROP TABLE **IF EXISTS** ~~a stejně tak se musí smazat sekvence, které vzniknou použitím datového typu SERIAL~~. Indexy primárních klíčů a referencí se smažou automaticky při smazání tabulky, nově i sekvence, které vzniknou použitím SERIAL.

```
DROP TABLE IF EXISTS  zamestanci;
DROP TABLE IF EXISTS oddeleni;
DROP SEQUENCE IF EXISTS oddeleni_prim_klic_seq;
```

DROP SEQUENCE je s novou verzí PostgreSQL zbytečné, protože se sekvence smaže už příkazem DROP TABLE. Ale ani neuškodí :-).

Všiměnte si, že nejdříve musí být smazaná tabulka zamestnanci, která se odkazuje do tabulky oddeleni. Nemůžete smazat tabulku oddeleni, když se na ní nějaká jiná tabulka odkazuje. Jinak by takový odkaz odkazoval do nikam a tím by se porušila integrita databáze.

Následuje vytvoření tabulky oddeleni, která bude obsahovat jen název oddělení, telefon do daného oddělení a primární klíč se jménem prim_klic.

Postgres od verze 9.3 nezobrazí NOTICE (viz příklad níže), pokud nemá nastaven log level na DEBUG1.

```
rimmer1=> CREATE TABLE oddeleni (
nazev VARCHAR(20),
telefon VARCHAR(20),
prim_klic SERIAL NOT NULL PRIMARY KEY
);
NOTICE:  CREATE TABLE will create implicit sequence "oddeleni_prim_klic_seq" for serial column "oddeleni.prim_klic"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "oddeleni_pkey" for table "oddeleni"
CREATE TABLE
```

Možná Vás napadlo, že byste mohli použít jako primární klíč pro tabulku oddělení telefonní číslo. Možné to je, ale mohli by nastat problémy v případě přečíslování. Všechny dokumenty, všechny odkazy, cokoliv co se vztahuje k primárnímu klíči byste museli změnit. Navíc je telefonní číslo zbytečně dlouhé (těžko budete mít v podniku milión oddělení), takže práce s ním by byla

pomalejší než s umělým primárním klíčem. To jsou všechno věci, které musíte při návrhu tabulek brát v úvahu. Mezi vašimi úkoly je i (kvalifikované) věštění budoucnosti (jak moc se databáze může rozrůst)!

Druhá tabulka, zamestnanci, bude obsahovat jméno a příjmení zaměstnance, jeho plat, taktéž primární klíč (rodné číslo) a referenci do oddělení, ve kterém zaměstnanec pracuje, oddeleni_id.

Sloupec oddeleni_id tabulky zamestnanci se odkazuje do tabulky oddeleni. Proto je nutné vytvořit nejdříve tabulku oddeleni a až pak tabulku zamestnanci.

Tabulka zamestnanci využívá rodné číslo jako svůj primární klíč, ale to teď není podstatné.

```
rimmer1=> CREATE TABLE zamestnanci (
jmeno VARCHAR(20),
prijmeni VARCHAR(20),
plat INTEGER CHECK (plat >= 0),
oddeleni_id INTEGER REFERENCES oddeleni(prim_klic),
rodne_cislo CHAR(10) NOT NULL PRIMARY KEY
);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "zamestnanci_pkey" fortable "zamestnanci"
CREATE TABLE
```

Teď už můžete vkládat data. Zase platí, že nejdříve se musí vložit data do tabulky s primárním klíčem a až pak do tabulky (tabulek), které se na tento primární klíč odkazují. Nemůžete vložit do tabulky zaměstnanců záznam s odkazem na neexistující řádek v tabulce oddělení.

Tohle není zbytečná buzerace, naopak, je to způsob, jak zajistit **integritu dat**, jak snížit možnost chyb. (Je lepší zaznamenat chybu už při vkládání dat do databáze, než aby vám DBMS umožnil vložit chybné data a přišlo se na to náhodou někdy za pár měsíců). Nikdo vás nemůže nutit primární klíče a refence používat, mohli byste používat prostě sloupečky typu INT bez integritních omezení a pak byste mohli mazat i vytvářet tabulky v libovolném pořadí, vkládat a mazat data v libovolném pořadí, ale přišli byste o tuto důležitou kontrolu. (Obdobně by se to dalo napsat o dalších integritních omezeních, jako je UNIQUE, NOT NULL, CHECK). Jejich použití je dobrovolné, ale velice, velice doporučované (všude tam, kde dávají smysl).

```
INSERT INTO oddeleni(nazev, telefon, prim_klic) VALUES ('Sekretariat', '2 123 123 12', DEFAULT);
INSERT INTO oddeleni(nazev, telefon) VALUES ('Pravni oddeleni', '2 123 123 11');
INSERT INTO oddeleni(nazev, telefon) VALUES ('Pravni oddeleni', '2 123 123 13');
```

Přišel čas vložit do databáze záznamy o zaměstnancích. Čísla primárních klíčů jednotlivých oddělení jsou (díky sekvenci) postupně 1, 2 a 3. Přesvědčit se o tom můžete příkazem SELECT.

Pokud se pokusíte vložit do tabulky záznam s již existujícím primárním klíčem (v případě tabulky zaměstnanců se stejným rodným číslem), nepodaří se vám to (p.k. je UNIQUE). Stejně tak, pokud se pokusíte vložit záznam s cizím klíčem, který neexistuje v tabulce do které se cizí klíč odkazuje (v případě tabulky zaměstnanců je cizý klíč číslo oddělení).

```
INSERT INTO zamestnanci VALUES ('Lenka',  'Pavova',10000,1,'8001010601');
INSERT INTO zamestnanci VALUES ('Jana',   'Pavova',10000,1,'8001010602');
INSERT INTO zamestnanci VALUES ('Jana',   'Mala',  12000,1,'8001010603');
INSERT INTO zamestnanci VALUES ('Lenka',  'Pavova',15000,2,'8001010604');
INSERT INTO zamestnanci VALUES ('Tom',    'Jerry', 15000,2,'8001010605');
INSERT INTO zamestnanci VALUES ('Martin', 'Luter', 12000,2,'8001010606');
INSERT INTO zamestnanci VALUES ('Leopold','King',  13000,2,'8001010607');
INSERT INTO zamestnanci VALUES ('Tomas',  'Mann',  22000,3,'8001010608');
INSERT INTO zamestnanci VALUES ('Vasek',  'Trn',   16000,3,'8001010609');
INSERT INTO zamestnanci VALUES ('Stary',  'Osel',   9000,3,'8001010610');
```

Teď už nemůžete smazat žádný záznam z tabulky oddělení, protože na každou řádku se odkazuje nějaký řádek z tabulky zaměstnanců. Pokud byste chtěli oddělení smazat, musíte nejdříve „přesunout" zaměstnance do jiného oddělení (změnit jejich oddeleni_id), nebo je vyhodit :-).

```
rimmer=> DELETE FROM oddeleni WHERE prim_klic = 1;
ERROR:  update or delete on table "oddeleni" violates foreign key constraint "zamestnanci_oddeleni_id_fkey"
on table "zamestnanci"
DETAIL:  Key (prim_klic)=(1) is still referenced from table "zamestnanci".
```

Tabulky z této kapitoly budu používat pro výklad v dalších kapitolách.

Proč relační databáze

Už víte z čeho je název relační databáze odvozen? Ne, nevíte :-P. Relační databáze totiž nemají nic společného s relacemi, které jsem popisoval v této kapitole (ač si to hodně lidí myslí). Existují i relační databáze, které takovéto relace nepodporují.

Termín relace přišel z matematiky, z teorie množin. Relací je myšlena samotná tabulka, která je vlastně *podmnožinou kartézkého součinu množin všech přípustných hodnot všech sloupců* – což je relace.

Řádky tabulky jsou **záznamy**, které mají **atributy** ve formě sloupečků. Představte si tabulku, která by obsahovala všechny záznamy všech možných kombinací atributů – to je kartézký součin. Tabulka ale obsahuje většinou jen některé (smysluplné) záznamy, tedy relace.

Schválně se zeptejte svého učitele databází, jestli ví, proč se relačním databázím říká relační. Nejspíš to vědět bude (doufám), ale vy se aspoň blýsknete, že to víte taky :-).

MySQL

Všechny SQL příkazy jsem pro vás uložil do souboru psql6-mysql.sql.

Co se teorie primárních klíčů, referencí a integritních omezení týče, to platí pro všechny databáze stejně. Jak je to uděláno prakticky, to už je jiná káva.

Předně, MySQL nemá, nezná a neumí SEQUENCE. Místo toho můžete na sloupečku s primárním klíčem přidat tzv. **AUTO_INCREMENT**, který dělá v podstatě to samé jako sekvence. (V MySQL nemůžete sekvence vytvářet sami (třeba pro jiné sloupečky, které nejsou primární klíče), mazat je atd. MySQL prostě databázový objekt SEQUENCE nezná).

Za další je pro MySQL důležité, abyste použili ENGINE InnoDB. Jinak MySQL nebude kontrolovat vztah mezi primárním a cizým klíčem (můžete vložit cizý klíč, který odkazuje na neexistující primární klíč atd.)

```
CREATE TABLE oddeleni (
nazev VARCHAR(20),
telefon VARCHAR(20),
prim_klic INT NOT NULL PRIMARY KEY AUTO_INCREMENT
) ENGINE=InnoDB;
```

A ještě jeden chyták v MySQL. Pokud vytvoříte tabulku takto:

```
CREATE TABLE zamestnanci (
    jmeno VARCHAR(20),
    prijmeni VARCHAR(20),
    plat INTEGER CHECK (plat >= 0),
    oddeleni_id INTEGER REFERENCES oddeleni(prim_klic),
    rodne_cislo CHAR(10) NOT NULL PRIMARY KEY
) ENGINE=InnoDB;
```

Tak jednak nebude fungovat CHECK (ten nefunguje v MySQL nikdy, jak už víte), ale ani REFERENCES. Proč ne? To fakt nevím. Ale naštěstí existuje alternativní zápis integritního omezení (constraintu) REFERENCES, který už v MySQL (s engine InnoDB) funguje:

```
CREATE TABLE zamestnanci (
    jmeno VARCHAR(20),
    prijmeni VARCHAR(20),
    plat INTEGER CHECK (plat >= 0),
    oddeleni_id INTEGER,
    rodne_cislo CHAR(10) NOT NULL PRIMARY KEY,
    FOREIGN KEY (oddeleni_id) REFERENCES oddeleni(prim_klic)
) ENGINE=InnoDB;
```

Prvnímu způsobu zápisu se říká **definice na úrovni sloupce**, druhému **definice na úrovni tabulky**. První v MySQL nefunguje (je ignorován), druhý funguje. **Druhý způsob zápisu funguje i v PostgreSQL**.

K čemu je vůbec dobrý druhý způsob zápisu? Možná už jsem se zmínil, že primární klíč se může skládat z více sloupců. V takovém případě jej nelze definovat na úrovni sloupečku. Taktéž cizí klíč, který se odkazuje na více sloupečků (na primární klíč skládající se z více sloupečků) se musí definovat na úrovni tabulky. Proto jsou za klíčovými slovy FOREIGN KEY sloupce, které tvoří cizí klíč, uzavřené v závorkách (a oddělené čárkou, když jich je více).

V příkladu nahoře se skládá cizí klíč jen z jednoho sloupce (oddeleni_id), ale i tak musí být uzavřen v závorkách. Pamatujte si to!

SQLite

Všechny SQL příkazy jsem pro vás uložil do souboru psql6-sqlite.sql.

SQLite také nezná sekvence. Umožní vám sice vytvořit tabulku s typem SERIAL, ale to je jen synonymum pro INTEGER. SQLite má, podobně jako MySQL, AUTOINCREMENT (na rozdíl od MySQL se píše bez podtržítka).

AUTOINCREMENT se dá použít jen s primárním klíčem nad sloupečkem s datovým typem INTEGER (kupodivu s jeho synonymem INT to nejde.)

```
CREATE TABLE oddeleni (
    nazev VARCHAR(20),
    telefon VARCHAR(20),
    prim_klic INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT
);
```

Jak už jsem psal dříve, SQLite neumí použít při insertu DEFAULT, takže první INSERT do tabulky oddělení, kde jsem jeho použití schválně ukázal, fungovat v SQLite nebude (tak ho musíte přepsat jako následující dva INSERTy z příkladu).

V SQLite můžete definovat REFERENCES jak na úrovni sloupce, tak na úrovni tabulky, oboje bude fungovat. (Hanbi sa, MySQL!).

SQLite cizí klíče defaultně nepodporuje. **Kontrolu cizích klíčů musíte zapnout** následujícím příkazem:

```
PRAGMA foreign_keys = ON;
```

Samozřejmě to znamená určité spomalení práce, ale integrita dat je určitě důležitější. Doporučuji vám tento příkaz zapsat do ~/.sqliterc, aby se vám kontrola cizích klíčů zapínala automaticky.

Oracle

Všechny SQL příkazy jsem pro vás uložil do souboru psql6oracle.sql. (Poznámka: v Apexu si musíte nahrát soubor pod jménem „psql6oracle" nebo tak nějak. Název nemůže obsahovat nic jiného než písmena a čísla (takže ani tečku, ani „mínus")).

Oracle tentokrát moc nepotěší. Sice má sekvence, ale nemá „typ" SERIAL, takže si sekvence musíte vytvořit explicitně. Ale co hůř, nemůžete hodnotu ze sekvence použít jako defaultní hodnotu pro sloupeček. Musíte jí tak předávat při každém insertu. Oracle také neumí klauzuli IF EXISTS, takže se na to musí přes *programování*.

```
BEGIN
EXECUTE IMMEDIATE 'DROP TABLE zamestnanci';
EXCEPTION
WHEN OTHERS THEN
IF SQLCODE != -942 THEN
RAISE;
END IF;
END;
/
```

O co tu jde? Program začíná slovem BEGIN, pak se spustí okamžitě SQL příkaz DROP TABLE zamestnanci(který v uvozovkách nesmíte ukončovat středníkem). Pak se zachytí jakákoliv výjimka (exception), ke které dojde a zkontroluje se její číselný kód. Kód -942 odpovídá chybě *mazání tabulky, která neexistuje*. Když se nejedná o tento kód, výjimka se znovu vyvolá (RAISE), když se jedná o tento kód, neudělá se nic (chyba se tiše ignoruje).

Celý program se pak ukončí lomítkem /.

Pokud spustíte program bez závěrečného lomítka v Apexu v okně "SQL Commands", tak vám to projde. Ale běda vám, pokud byste na něj zapomněli ve skriptu!

O programování v Oracle by se dala napsat celá kniha, to tady popisovat nebudu. Berte proto prosím předchozí (a následující) příklad jako dogma.

Totéž platí při mazání sekvence, jen kód výjimky je jiný: -2289.

```
BEGIN
EXECUTE IMMEDIATE 'DROP SEQUENCE oddeleni_prim_klic_seq';
EXCEPTION
WHEN OTHERS THEN
IF SQLCODE != -2289 THEN
RAISE;
END IF;
```

```
END;
/
```

Tabulka oddeleni se vytvoří s typem INTEGER pro primární klíč a sekvence se vytvoří následne SQL příkazem CREATE SEQUENCE. Na vytváření tabulky zamestnanci už není nic zajímavého.

```
CREATE TABLE oddeleni (
    nazev VARCHAR2(20),
    telefon VARCHAR2(20),
prim_klic INTEGER NOT NULL PRIMARY KEY
);

CREATE SEQUENCE oddeleni_prim_klic_seq;

CREATE TABLE zamestnanci (
    jmeno VARCHAR2(20),
    prijmeni VARCHAR2(20),
    plat INTEGER CHECK (plat >= 0),
oddeleni_id INTEGER REFERENCES oddeleni(prim_klic),
    rodne_cislo CHAR(10) NOT NULL PRIMARY KEY
);
```

Zajímavé jsou ještě příkazy pro vložení dat do tabulky oddeleni. Protože prim_klic nemá defaultní hodnotu, musí se při insertu vkládat explicitně. Další hodnota v řadě se ze sekvence získá pomocí NEXTVAL.

```
INSERT INTO oddeleni(nazev, telefon, prim_klic) VALUES
('Sekretariat','2 123 123 12', oddeleni_prim_klic_seq.NEXTVAL);
INSERT INTO oddeleni(nazev, telefon, prim_klic) VALUES
('Pravni oddeleni','2 123 123 11', oddeleni_prim_klic_seq.NEXTVAL);
INSERT INTO oddeleni(nazev, telefon, prim_klic) VALUES
('Pravni oddeleni','2 123 123 13', oddeleni_prim_klic_seq.NEXTVAL);
```

Na vkládání do tabulky zamestnanci už nic zajímavého není.

SELECT I a VIEW

Jak už jsem dříve psal, nejdůležitější SQL příkaz je SELECT Toto je jedna z prvních kapitol, kde se mu budu věnovat podrobně. Navíc zbyde ještě trochu času na použití SELECTu v kombinaci s jinými SQL příkazy (CREATE TABLE a INSERT INTO) a na nový databázový objekt VIEW, který, zjednodušeně řečeno, ze SELECTu vytvoří virtuální tabulku.

- Výběr sloupců
  - Tabulka dual
- Spojování tabulek
- Pohledy (CREATE VIEW)
- CREATE TALBE AS
- INSERT INTO
- MySQL /MariaDB
- SQLite
- Oracle

Výběr sloupců

O příkazu SELECT jsem se poprvé zmínil ve třetí kapitole. Tento příkaz se používá pro získávání dat z databáze. Umí toho ale mnohem více, než jenom zobrazit data z tabulky tak, jak jsou uložená. Ukazoval jsem například, jak z databáze získat jen některé řádky, pomocí podmínky WHERE.

Teď ukáži, jak vybrat jen některé sloupce. Jak se to dělá jste už mohli vidět v lekci o úpravách tabulek, v části o SQLite, kde se plnila tabulka daty pomocí INSERT INTO … SELECT, který v této kapitole ještě připomenu.

Budu vycházet z příkladu z předchozí lekce.

Pokud za klíčovým slovem SELECT uvedete hvězdičku, vypíší se všechny sloupce v tabulce. Pokud chcete vypsat jen některé sloupce, zapište jejich jména místo hvězdičky. Sloupce se budou vypisovat v tom pořadí, v jakém jste je zapsali. Názvy sloupců můžete navíc uzavřít do **dvojitých uvozovek**. To pro případ, že by název sloupce obsahoval nějaké nestandardní znaky, které by SQL parser nerozpoznal jako součást identifikátoru (názvu sloupce), například mezery.

Dvojité uvozovky vlastně můžete používat i k označení dalších identifikátorů, jako je třeba název tabulky.

```
rimmer1=> SELECT * FROM "oddeleni";
        nazev       |   telefon    | prim_klic
--------------------+--------------+-----------
 Sekretariát        | 2 123 123 12 |         1
 Právní oddělení    | 2 123 123 11 |         2
 Právní oddělení    | 2 123 123 13 |         3
(3 rows)

rimmer1=> SELECT "prim_klic", nazev FROM oddeleni;
 prim_klic |       nazev
-----------+-------------------
         1 | Sekretariát
         2 | Právní oddělení
         3 | Právní oddělení
(3 rows)
```

Pokud se vám nelíbí název nějakého sloupce, můžete ho nechat vytisknout pod jiným jménem pomocí klauzule **AS**. (Jméno sloupce v tabulce to nijak neovlivní.)

```
rimmer1=> SELECT prijmeni AS "potřebují přidat", rodne_cislo
FROM zamestnanci WHERE plat <= 10000;
 potřebují přidat | rodne_cislo
------------------+-------------
```

```
Pavova          | 8001010601
Pavova          | 8001010602
Osel            | 8001010610
(3 řádky)
```

V tomto příkladu už jsem musel použít dvojité uvozovky pro název sloupce za AS, protože název obsahuje mezeru. Bez uvozovek by se to SQL parser snažil interpretovat jako dva identifikátory zasebou, což by vedlo k nepochopení SQL příkazu a následné chybě.

Díky uvozovkám můžete vytvořit tabulku s názvem obsahujícím mezeru, nebo sloupce s mezerou. ***Nedělejte to.*** *Jen si tím zavaříte na problémy a ostatní programátoři, co budou pracovat s vaší databází, vás nebudou mít rádi. Pokud chcete mít název tabulky nebo sloupce složený z více slov,* ***spojujte je podtržítky****(nazev_tabulky). Raději pro žádné identifikátory* ***nepoužívejte ani českou diakritiku*** *(kdo ví, třeba budete mít jednou úspěch a s vaším produktem budou pracovat i cizinci, kteří nebudou mít českou diakritiku k dispozici).*

Místo názvu sloupce můžete za SELECT uvést i nějaký výraz, který se může a nemusí odkazovat na nějaký sloupeček z tabulky.

```
SELECT 'Zaměstnanec:', prijmeni, jmeno, plat, plat-(plat/100)*20
FROM zamestnanci;
 ?column?   | prijmeni |  jmeno  | plat  | ?column?
--------------+----------+---------+-------+----------
Zaměstnanec: | Pavova   | Lenka   | 10000 |     8000
Zaměstnanec: | Pavova   | Jana    | 10000 |     8000
Zaměstnanec: | Mala     | Jana    | 12000 |     9600
Zaměstnanec: | Pavova   | Lenka   | 15000 |    12000
                                ...
Zaměstnanec: | Osel     | Stary   |  9000 |     7200
(10 řádek)
```

Všimněte si, že první sloupeček je obyčejná řetězcová konstanta. Řetězce se uvozují do **jednoduchých uvozovek**.

Všimněte si taky názvu sloupců nad řetězcovou konstantou (první sloupec) a nad matematickým výrazem (poslední sloupec). Nejsou moc hezké, že? Od toho tu je AS! Pomocí AS můžete dokonce přejmenovat i tabulku, ačkoliv v následujícím příkladu je to zcela zbytečné.

```
SELECT 'Zaměstnanec:' AS " ", prijmeni, jmeno,
plat AS "hrubá mzda", plat-(plat/100)*20 AS "čistá mzda"
FROM zamestnanci AS nevolnici;
              | prijmeni |  jmeno  | hrubá mzda | čistá mzda
--------------+----------+---------+------------+------------
Zaměstnanec: | Pavova   | Lenka   |      10000 |       8000
Zaměstnanec: | Pavova   | Jana    |      10000 |       8000
Zaměstnanec: | Mala     | Jana    |      12000 |       9600
Zaměstnanec: | Pavova   | Lenka   |      15000 |      12000
                                ...
Zaměstnanec: | Osel     | Stary   |       9000 |       7200
(10 řádek)
```

Identifikátor nemůže být prázdný řetězec, proto jsem pojmenoval první sloupec jako mezeru " ".

Tabulka dual

Asi vás napadlo, že můžete použít SELECT jako kalkulačku. Kolik je třeba 3 na třetí?

```
rimmer1=> select 3^3 as "3^3" FROM oddeleni;
select 3^3 as "3^3" from oddeleni;
 3^3
-----
  27
  27
  27
(3 řádky)
```

Jupí, ono to funguje! Akorát se výsledek vypsal pro každý řádek z tabulky oddeleni. Chudák tabulka, takhle zneužitá. Chtělo by to nějakou univerzální tabulku s jedním řádkem.

A ejhle, taková tabulka existuje. Ale ne v PostgreSQL. MySQL a Oracle k tomuto účelu poskytují tabulku **dual**. Jedná se o „virtuální" tabulku, kterou nenajdete ve výpisu mezi svými tabulkami, ale přesto nad ní můžete provádět SELECTy.

Takovou tabulku si samozřejmě můžete vytvořit („nevirtuálně") i v PostgreSQL …

```
CREATE TABLE dual (dummy CHAR(1) NOT NULL UNIQUE CHECK (dummy = 'X'));
INSERT INTO DUAL VALUES('X');
```

Tabulku jsem vytvořil schválně tak, aby do ní šel vložit jen jeden řádek. Né že by to bylo nezbytně nutné, ale případným čtenářům mé databáze rychleji dojde, k čemu že ta tabulka slouží. (Nebo jim to třeba nedojde vůbec, když neznají dual z jiných DBMS, za nic neručím :-)

```
rimmer1=> SELECT 3^3 AS "3^3" FROM dual;
 3^3
-----
  27
(1 řádka)
```

A jedna dobrá (?) zpráva na konec. PostgreSQL dokáže udělat **nestandardní** SELECT, který neobsahuje klauzuli FROM. Takže se vlastně bez tabulky dual obejdete.

```
rimmer1=> SELECT 3^3;
?column?
----------
      27
(1 řádka)

rimmer1=> select pi() as "PI";
PI
```

```
------------------
 3.14159265358979
(1 řádka)
```

Smyslem databází není nahrazovat matlab nebo excel, ale i tak má PostgreSQL docela slušnou výbavičkumatematických operátorů a funkcí. Funkce, jako je třeba ta pi(), proberu v kapitole o funkcích.

Spojování tabulek

V minulé kapitole jsem vytvořil dvě tabulky: oddeleni a zamestnanci. Nyní se pokusím vybrat informace z obou tabulek naráz. To se dělá tak, že se v příkazu SELECT za klíčovým slovem FROM zapíší jména obou tabulek oddělených čárkou. Výsledek pak bude **kartézský součin** obou tabulek.

To znamená, že se vytvoří řádky obsahující sloupce z obou tabulek (resp. ze všech, které vyjmenujete zaFROM) a řádky se pospojují **každý s každým** (to je kartézký součin).

V našem případě jsou v tabulce oddělení tři řádky a v tabulce zaměstnanců 10 řádků. Takže ve výsledku bude každý řádek z tabulky oddělení spojení s každým řádkem z tabulky zaměstnanci (3 * 10 = 30 řádků).

```
rimmer1=> SELECT jmeno,prijmeni,oddeleni_id,prim_klic,nazev AS oddeleni
          FROM oddeleni, zamestnanci;
  jmeno  | prijmeni | oddeleni_id | prim_klic |    oddeleni
---------+----------+-------------+-----------+-----------------
 Lenka   | Pavova   |           1 |         1 | Sekretariat
 Jana    | Pavova   |           1 |         1 | Sekretariat
 Jana    | Mala     |           1 |         1 | Sekretariat
 Lenka   | Pavova   |           2 |         1 | Sekretariat
 Tom     | Jerry    |           2 |         1 | Sekretariat
 Martin  | Luter    |           2 |         1 | Sekretariat
 Leopold | King     |           2 |         1 | Sekretariat
 Tomas   | Mann     |           3 |         1 | Sekretariat
 Vasek   | Trn      |           3 |         1 | Sekretariat
 Stary   | Osel     |           3 |         1 | Sekretariat
 Lenka   | Pavova   |           1 |         2 | Pravni oddeleni
 Jana    | Pavova   |           1 |         2 | Pravni oddeleni
 Jana    | Mala     |           1 |         2 | Pravni oddeleni
                                ....
 Stary   | Osel     |           3 |         3 | Pravni oddeleni
(30 rows)
```

Výpis je zkrácen. První tři sloupce jsou vybrány z tabulky zamestnanci, poslední dva z tabulky oddeleni.

Možná vás už napadlo, jak vybrat tabulku s hodnotami, které budou dávat smysl. Využijte primárního klíče z tabulky oddělení a cizího klíče z tabulky zaměstnanci, abyste spojili řádky zaměstnanců jen s odpovídajícím oddělením.

Takto se využívá **relace** mezi tabulkami.

```
rimmer1=> SELECT jmeno, prijmeni, oddeleni_id, prim_klic, nazev AS oddeleni
          FROM oddeleni, zamestnanci
          WHERE oddeleni_id = prim_klic;
  jmeno  | prijmeni | oddeleni_id | prim_klic |    oddeleni
---------+----------+-------------+-----------+-----------------
 Lenka   | Pavova   |           1 |         1 | Sekretariat
 Jana    | Pavova   |           1 |         1 | Sekretariat
 Jana    | Mala     |           1 |         1 | Sekretariat
 Lenka   | Pavova   |           2 |         2 | Pravni oddeleni
 Tom     | Jerry    |           2 |         2 | Pravni oddeleni
 Martin  | Luter    |           2 |         2 | Pravni oddeleni
 Leopold | King     |           2 |         2 | Pravni oddeleni
 Tomas   | Mann     |           3 |         3 | Pravni oddeleni
 Vasek   | Trn      |           3 |         3 | Pravni oddeleni
 Stary   | Osel     |           3 |         3 | Pravni oddeleni
(10 rows)
```

Teď máte krásný přehled o tom, kdo pracuje v jakém oddělení. Jistě už pro vás nebude problém zjistit, v jakém oddělení pracuje zaměstnanec Vasek Trn:

```
rimmer1=> SELECT nazev FROM oddeleni, zamestnanci
WHERE oddeleni_id = prim_klic AND jmeno='Vasek' AND prijmeni='Trn';
     nazev
-----------------
 Pravni oddeleni
(1 row)
```

V případě že se ve dvou tabulkách schodují názvy sloupců, můžete se na ně odkazovat přes tzv. **tečkovou notaci**:

```
SELECT nazev_tabulky1.nazev_sloupce, nazev_tabulky2.nazev_sloupce
FROM nazev_tabulky1, nazev_tabulky2;
```

Například:

```
SELECT oddeleni.nazev
FROM oddeleni, zamestnanci
WHERE oddeleni_id = prim_klic AND jmeno='Vasek' AND prijmeni='Trn';
```

A ještě něo o AS. Když přejmenujete tabulku pomocí AS, už se na ní musíte v SELECTu odkazovat novým jménem.

```
rimmer1=> SELECT o.nazev FROM oddeleni AS o;
     nazev
-----------------
 Sekretariat
 Pravni oddeleni
 Pravni oddeleni
(3 řádky)
```

```
rimmer1=> SELECT oddeleni.nazev FROM oddeleni AS o;
ERROR:  invalid reference to FROM-clause entry for table "oddeleni"
ŘÁDKA 1: SELECT oddeleni.nazev FROM oddeleni AS o;
                ^
DOPORUČENÍ:  Perhaps you meant to reference the table alias "o".
```

Jo a ještě něco. Klíčové slovo AS je nepovinné, takže ho nemusíte uvádět (ani u tabulek, ani u sloupců). Následující dva SELECTy dělají úplně to samé.

```
SELECT o.nazev  AS "oddeleni" FROM oddeleni AS o;
SELECT o.nazev "oddeleni" FROM oddeleni o;
```

Pohledy (CREATE VIEW)

Všimli jste si, že když provedete nějaký SELECT, že jeho výsledkem je vlastně tabulka? Nebylo by skvělé, kdyby jste s takovou tabulkou mohli pracovat jako s jakoukoliv jinou?

Pomocí příkazu **CREATE VIEW** můžete uložit **pohled** na data vytvořený příkazem SELECT. Tento pohled se chová podobně jako skutečná tabulka. Za určitých, hodně specifických, okolností se do něj dají i vkládat data, ale důležitější je, že z něj můžete data získávat pomocí dalšího příkazu SELECT.

Syntaxe příkazu je následující:

```
CREATE VIEW nazev_pohledu AS  SELECT ...
```

Až tak jednoduché to je. Stačí před SELECT napsat CREATE VIEW nazev_pohledu AS a je hotovo.

Na příkaz SELECT neexistují při vytváření pohledů žádná omezení. Můžete používat klauzule jako WHERE, můžete tabulky spojovat atd.

Pohled (VIEW) není skutečná tabulka, ve které by byly data. Pohled je spíše jako uložený SELECT pod nějakým jménem. Změníte-li tedy hodnoty tabulek ze kterých jste pohled vytvářeli, změní se i hodnoty v pohledu, která tato data zobrazuje.

```
CREATE VIEW pracovni_pozice AS
SELECT rodne_cislo, prijmeni, jmeno, nazev AS oddeleni
FROM oddeleni, zamestnanci
WHERE oddeleni_id = prim_klic;
```

Nyní máte vytvořen pohled pracovni_pozice a můžete s ním pracovat skoro stejně, jako s jakoukoliv jinou tabulkou.

```
rimmer1=> SELECT * FROM pracovni_pozice WHERE oddeleni = 'Sekretariat';
 rodne_cislo | prijmeni | jmeno |  oddeleni
-------------+----------+-------+-------------
 8001010601  | Pavova   | Lenka | Sekretariat
 8001010602  | Pavova   | Jana  | Sekretariat
 8001010603  | Mala     | Jana  | Sekretariat
(3 řádky)
```

Všimněte si, že můžete z tohoto pohledu zjistit, kdo pracuje v právním oddělení, ale nemůžeme zjistit, v kterém (v databázi jsou dvě právní oddělení), protože pohled neobsahuje prim_klic právního oddělení.

Pohledy můžete vypsat metapříkazem \dv (nebo \d). Definici pohledu (SELECT, který view definuje), zobrazíte metapříkazem \d+ nazev_pohledu.

Pohled smažete příkazem DROP VIEW nazev_pohledu;

```
DROP VIEW pracovni_pozice;
```

Kdykoliv provedete SELECT nad pohledem, musí se provést nejdříve SELECT, který definuje pohled. Vytvořením pohledu tedy rozhodně získávání dat z databáze nezrychlíte (spíš naopak). DBMS se snaží SELECTy optimalizovat, s větším či menším úspěchem. Když uděláte SELECT nad SELECTem, který přidává nějakou podmínku do WHERE, DBMS někdy dokáže vytvořit z obou SELECTů jen jeden (který spojí všechny podmínky do jedné klauzule WHERE) a ten pak provede. Použití SELECTu nad SELECTem pak **není pomalejší**, než kdybyste si vytvořili jeden SELECT sami. Ale to je to nejlepší, co se o „zlepšování výkonu" použitím pohledů dá říct.

V pohledu se může objevit opravdu jakýkoliv SELECT, takže i SELECT z jiného pohledu. Takovéto řetězení pohledů DBMS optimalizaci určitě neulehčuje, takže si to raději 2x rozmyslete, než SELECTnete.

Postgres od verze 9.3 umožňuje nad jednoduchými pohledy provádět DML operace (INSERT, UPDATE, DELETE).

CREATE TABLE AS

Jako můžete vytvořit pohled do tabulek pomocí CREATE VIEW, můžete obdobně vytvořit i novou tabulku. Do nové tabulky přenesete data z původní tabulky (nebo tabulek) pomocí SELECT.

CREATE TABLE AS vytvoří skutečnou tabulku, která nebude mít s původní tabulkou (tabulkami) již nic společného. Pokud původní tabulku aktualizujete, této nové tabulky se to nijak nedotkne! Taky proč? Už je to jiná tabulka.

Vytvářet novou tabulku tímto způsobem se hodí například v případě, kdy chcete uložit nějaký pohled do databáze v čase, aby se už neměnil – prostě taková slabší záloha.

```
rimmer1=> CREATE TABLE platy2013 AS SELECT oddeleni_id, prijmeni,
jmeno, plat AS hruba_mzda, plat-(plat/100)*20 AS cista_mzda
FROM zamestnanci;
rimmer1=> SELECT * FROM platy2013;
 oddeleni_id | prijmeni |  jmeno  | hruba_mzda | cista_mzda
-------------+----------+---------+------------+------------
           1 | Pavova   | Lenka   |      10000 |       8000
           1 | Pavova   | Jana    |      10000 |       8000
           1 | Mala     | Jana    |      12000 |       9600
                                ...
           3 | Osel     | Stary   |       9000 |       7200
(10 řádek)
```

Pokud vytvoříte tabulku tímto způsobem, žádná integritní omezení ani indexy se v notvé tabulce z té staré nepřenesou. Pokud je chcete, musíte je k tabulce přidat. O tom jak se to dá udělat zase až někdy příště.

INSERT INTO

Použití INSERT INTO … SELECT jsem už ukazoval v SQLite části o ALTER TABLE (jak přejmenovat nebo smazat sloupec). Nevýhoda nutnosti přidávat dodatečně všechna integritní omezení a indexy do tabulky u příkazu CREATE TABLE AS se dá obejít tak, že nejdřív vytvoříte tabulku pomocí CREATE TABLE tak jak chcete a pak jí naplníte daty z jiné tabulky (nebo jiných tabulek) pomocí příkazu INSERT INTO … SELECT.

```
DROP TABLE platy2013;
CREATE TABLE platy2013 (
oddeleni_id INTEGER REFERENCES oddeleni(prim_klic),
prijmeni varchar(20),
jmeno varchar(20),
hruba_mzda integer CHECK (hruba_mzda >= 0),
cista_mzda integer CHECK (cista_mzda >= 0)
);
INSERT INTO platy2013 SELECT oddeleni_id, prijmeni,
jmeno, plat, plat-(plat/100)*20
FROM zamestnanci;
```

Při vkládání hodnot do tabulky můžete vyjmenovat sloupečky, do kterých chcete něco vložit. Předchozí příklad by se dal přepsat takto (schválně jsem pomíchal pořadí sloupečků, na výsledek to ale nemá vliv):

```
INSERT INTO platy2013(prijmeni,jmeno,oddeleni_id,cista_mzda,hruba_mzda)
SELECT prijmeni, jmeno, oddeleni_id, plat-(plat/100)*20, plat
FROM zamestnanci;
```

Musíte si dávat pozor jen na to, aby počet a pořadí sloupečků vyjmenovaných za názvem tabulky do které se vkládá a sloupečky v SELECTu byli totožné (to dá rozum).

MySQL / MariaDB

Asi největší opruz z novinek v této kaptile jsou překvapivě uvozovky. Každý DBMS má trochu jinou strategii uvozování textových řetězců a identifikátorů, což je další hřebíček do rakve přenositelnosti SQL dotazů.

MySQL používá primárně pro uvozování identifkátorů zpětné uvozovky (`), pro uvozování řetězců jednoduché (') nebo dvojité (") uvozovky.

```
SELECT * FROM `oddeleni`;
```

Při přejmenovávání sloupce nebo tabulky pomocí AS se může v MySQL použít libovolná uvozovka ze všech tří možností (zpětná, jednoduchá i dvojitá).

V MySQL se v názvu sloupců výrazů bez použití AS nepoužije ?column?, ale MySQL použije samotný výraz (nebo jeho část).

```
mysql> SELECT 'Zaměstnanec:', prijmeni, jmeno, plat, plat-(plat/100)*20
FROM zamestnanci;
+---------------+----------+---------+-------+--------------------+
| Zaměstnanec:  | prijmeni | jmeno   | plat  | plat-(plat/100)*20 |
+---------------+----------+---------+-------+--------------------+
| Zaměstnanec:  | Pavova   | Lenka   | 10000 |          8000.0000 |
...
```

Pokud v MySQL použijete pro název sloupce jméno " " (mezeru), MySQL zobrazí varování Name ' ' has become ''. Tohle varování (warning) ničemu nevadí, ale někoho by to mohlo vylekat, tak se o tom raději zmiňuji. (Jiná varování můžou znamenat opravdový problém). Pokud vám to varování vadí, použijte jako název ''.

V MySQL existuje „virtuální" tabulka dual. Nemůžete z ní vybrat žádnou hodnotu (sloupeček), ale můžete jí použít v SELECTu s aritmetickými výrazy. Můžete použít i nestandardní formu SELECTu, kde se vypouští FROM (jako v Postgresu).

```
mysql> SELECT * FROM dual;
ERROR 1096 (HY000): No tables used

mysql> SELECT 3^3 FROM dual;
+-----+
| 3^3 |
+-----+
|   0 |
+-----+
mysql> SELECT pi();
+----------+
| pi()     |
+----------+
| 3.141593 |
+----------+
```

V příkladu výše si můžete všimnout, že MySQL má jiné aritmetické operátory a funkce. Zatímco v PostgreSQL je 3^3 tři na třetí, tj. 9, v MySQL je to logická operace XOR (pro kterou používá PostgreSQL operátor #). MySQL má na exponenty zase metodu POWER(). Zkrátka, čtěte dokumentaci.

SQLite

V SQLite můžete pro uvození jména tabulky použít libovolné uvozovky (zpětné uvozovky (`),jednoduché (') nebo dvojité ("), stejně tak pro přejmenování sloupce (za AS). Ale pro jméno sloupce můžete použít jen dvojitou nebo zpětnou uvozovku. (V jednoduchých uvozovkách je uzavřený řetězec).

Pokud nepřejmenujete sloupec s aritmetickým výrazem nebo řetězcem, nepoužije SQLite ?column?, ale jako MySQL použije výraz, který sloupec tvoří.

```
sqlite> SELECT 'Zaměstnanec:', prijmeni, jmeno, plat, plat-(plat/100)*20
FROM zamestnanci;
'Zaměstnanec:'  prijmeni  jmeno  plat  plat-(plat/100)*20
--------------  --------  ------  -----  ------------------
Zaměstnanec:    Pavova    Lenka   10000  8000
...
```

SQLite podporuje tyto aritmetické operátory: +, -, *, /, % (modulo)
a bitové operátory: <<, >>, &, |

V dokumentaci k základním funkcím najdete i nějaké matematické funkce.

Oracle

V Oracle můžete používat pro uvozování identifkátorů jen dvojité uvozovky. Ale pozor! Jak jsem dříve psal, že na velikosti písmen v identifikátorech nezáleží (jen u jmen tabulek v MySQL a to ještě jen v Linuxu), tak v Oracle na velikosti záleží.

Dokud nepoužijete uvozovky, Oracle automaticky převádí identifikátory na velká písmena. Když například napíšete AS oddeleni, ve výsledku se sloupeček bude jmenovat ODDELENI, ale AS "oddeleni" zobrazíoddeleni.
Pokud jste při definici tabulky nepoužili pro názvy sloupců či tabulek uvozovky, pak jsou názvy velkými písmeny. Pokud použijete uvozovky, musíte napsat identifikátory ve správné velikosti!

```
SELECT * FROM "ODDELENI";
SELECT "PRIM_KLIC", nazev FROM oddeleni;
```

Pokud nepřejmenujete sloupec s aritmetickým výrazem nebo řetězcem, nepoužije Oracle?column?, ale jako MySQL či SQLite použije výraz, který sloupec tvoří.
I Oracle má svou sadu aritmetických operátorů a funkcí. (Nemá operátor ^, ale má funkci POWER()).
Oracle nepodporuje nestandardní SELECT bez FROM, ale vždy má k dispozici tabulku dual.

```
SELECT POWER(3, 3) FROM dual;
POWER(3, 3)
-----------
         27
```

V Oracle při přejmenování tabulek **nemůžete** použít AS, ale bez AS to funguje:

```
SELECT o.nazev FROM oddeleni o; -- nemůže být ...oddeleni AS o;
```

Oracle nepodporuje IF EXISTS klauzuli u příkazu DROP VIEW. Vyřešit se to dá zase trochou programování (jako řeba u DROP TABLE – rozdíl je jen v SQLCODE).

```
BEGIN
EXECUTE IMMEDIATE 'DROP VIEW pracovni_pozice';
EXCEPTION
WHEN OTHERS THEN
IF SQLCODE != -00942 THEN
RAISE;
END IF;
END;
/
```

Opakování
V této kapitole se nenaučíte nic nového, ale protože opakování jest matkou moudrosti, zopakujete si co ste se už naučili. V příkladech vytvořím tabulky, vložím nějaké data, upravím je, zase smažu atd. Pokusím se ukázat příkazy SQL v plné síle, někdy možná na úkor účelnosti a smyslnosti takových příkazů. Příkazy nebudu moc komentovat, sami určitě snadno poznáte co je jejich účelem.

- CREATE TABLE
- INSERT INTO
- ALTER TABLE
- UPDATE
- SELECT
- CREATE VIEW
- DELETE FROM
- DROP TABLE, VIEW
- MySQL/MariaDB, SQLite, Oracle

CREATE TABLE

```
rimmer1=> CREATE TABLE zaci (
id SERIAL NOT NULL PRIMARY KEY,
pohlavi BOOL NOT NULL DEFAULT true,
jmeno VARCHAR(40) CHECK (jmeno != '') NOT NULL,
prijmeni VARCHAR(40) CHECK (prijmeni != '') NOT NULL
);
NOTICE:  CREATE TABLE will create implicit sequence "zaci_id_seq" for serial column"zaci.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "zaci_pkey" for table"zaci"
CREATE TABLE

rimmer1=> CREATE TABLE znamky (
id SERIAL NOT NULL PRIMARY KEY,
zaci_id INTEGER REFERENCES zaci(id) NOT NULL,
kod_predmetu CHAR(4) NOT NULL CHECK (kod_predmetu != ''),
znamka NUMERIC(1,0) CHECK (znamka >=1 AND znamka <= 5) DEFAULT 5,
datum DATE DEFAULT current_date
);
NOTICE:  CREATE TABLE will create implicit sequence "znamky_id_seq" for serial column"znamky.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "znamky_pkey" for table "znamky"
CREATE TABLE
```

*Jenom na okraj bych podotknul, že primární klíč by měl být jako první sloupec. Urychluje to práci s databází[zdroj?]. Také je zvykem jej pojmenovat **id**. A není od věci, aby měla každá tabulká svůj primární klíč.*
Poznámka: current_date je funkce Postgresu vracející aktuální datum v době použití funkce (tj. v době vložení hodnot do tabulky).

INSERT INTO

```
INSERT INTO zaci VALUES (DEFAULT, DEFAULT, 'Jan', 'Maly');
INSERT INTO zaci VALUES (DEFAULT, FALSE, 'Jana','Mala');
INSERT INTO zaci(jmeno, prijmeni, pohlavi) VALUES ('Jana', 'Velká', false);

INSERT INTO znamky VALUES (DEFAULT, 1, 'MA10', 5, DEFAULT);
INSERT INTO znamky VALUES (DEFAULT, 1, 'FY10', 3, DEFAULT);
INSERT INTO znamky VALUES (DEFAULT, 2, 'FY10', 3, TO_DATE('2014-12-24','yyyy-mm-dd'));
```

```
INSERT INTO znamky(zaci_id, kod_predmetu, znamka) VALUES (2, 'FY10', 1);
INSERT INTO znamky(znamka, kod_predmetu, zaci_id) VALUES (2, 'FY10', 1);
rimmer1=> SELECT *  FROM znamky;
 id | zaci_id | kod_predmetu | znamka |   datum
----+---------+--------------+--------+------------
  1 |       1 | MA10         |      5 | 2013-12-05
  2 |       1 | FY10         |      3 | 2013-12-05
  3 |       2 | FY10         |      3 | 2014-12-24
  4 |       2 | FY10         |      1 | 2013-12-05
  5 |       1 | FY10         |      2 | 2013-12-05
(5 řádek)
```

ALTER TABLE

```
ALTER TABLE znamky RENAME TO hodnoceni;          -- prejmenovani tabulky
ALTER TABLE hodnoceni RENAME zaci_id TO id_zaci;  -- prejmenovani sloupce
ALTER TABLE zaci ADD rodne_cislo CHAR(10) UNIQUE; -- pridani sloupce
NOTICE:  ALTER TABLE / ADD UNIQUE will create implicit index 'zaci_rodne_cislo_key' for table 'zaci'
ALTER
ALTER TABLE zaci RENAME TO studenti;
```

Vše za -- (dvěmi mínus) až do konce řádku je považováno za komentář.

Poznámka: Po přejmenování tabulky zaci na studenti tabulka hodnoceni (dříve znamky) odkazuje sloupecid_zaci do tabulky studenti (čili žádný problém, bro!).

UPDATE

```
UPDATE studenti SET rodne_cislo = '7951010611', prijmeni = 'Vdana'
WHERE id = 2;
UPDATE studenti SET rodne_cislo = '7901220611'
WHERE studenti.id = 1;
UPDATE hodnoceni SET znamka = 1*1+2 WHERE (znamka > 2 AND
znamka < 2+2) OR znamka = 3; -- dokážete to zjednodušit? :-)
UPDATE 2
```

SELECT

```
rimmer1=> SELECT * FROM studenti, hodnoceni WHERE id = id_zaci;
ERROR:  column reference "id" is ambiguous
```

Sloupec se jménem id je v obou tabulkách. Tak takhle to nepůjde. S tím se musíte nějak vypořádat (přest tzv. tečkovou notaci).

```
rimmer1=> SELECT * FROM studenti, hodnoceni
WHERE studenti.id = id_zaci;
 id | pohlavi | jmeno | prijmeni | rodne_cislo | id | id_zaci | kod_predmetu | znamka |   datum
----+---------+-------+----------+-------------+----+---------+--------------+--------+------------
  1 | t       | Jan   | Maly     | 7901220611  |  1 |       1 | MA10         |      5 | 2013-12-05
  2 | f       | Jana  | Vdana    | 7951010611  |  4 |       2 | FY10         |      1 | 2013-12-05
  1 | t       | Jan   | Maly     | 7901220611  |  5 |       1 | FY10         |      2 | 2013-12-05
  1 | t       | Jan   | Maly     | 7901220611  |  2 |       1 | FY10         |      3 | 2013-12-05
  2 | f       | Jana  | Vdana    | 7951010611  |  3 |       2 | FY10         |      3 | 2014-12-24
(5 řádek)
```

Zapamatujte si, že pokud vytváříte nějaký SQL příkaz, který budete používat opakovaně, je lepší používat tečkovou notaci. Nikdy totiž nevíte, kdy vám někdo přídá do jedné tabulky pomocí ALTER TABLE sloupeček se stejným názvem, jako je v jiné tabulce a z fungujícího SQL příkazu vám udělá nefungující.

```
rimmer1=> SELECT s.id AS studenti_id, s.jmeno, s.prijmeni,
h.datum, h.kod_predmetu "Kód předmětu", h.znamka
FROM studenti AS s, hodnoceni h
WHERE s.id = h.id_zaci AND s.prijmeni = 'Vdana';
 studenti_id | jmeno | prijmeni |   datum    | Kód předmětu | znamka
-------------+-------+----------+------------+--------------+--------
           2 | Jana  | Vdana    | 2013-12-05 | FY10         |      1
           2 | Jana  | Vdana    | 2014-12-24 | FY10         |      3
(2 řádky)
```

CREATE VIEW

```
rimmer1=> CREATE VIEW zaci AS
SELECT s.id, s.pohlavi, s.jmeno, s.prijmeni, s.rodne_cislo,
h.kod_predmetu AS predmet, h.znamka
FROM studenti s, hodnoceni h
WHERE s.id = h.id_zaci;

rimmer1=> SELECT * FROM zaci WHERE prijmeni = 'Vdana';
 id | pohlavi | jmeno | prijmeni | rodne_cislo | predmet | znamka
----+---------+-------+----------+-------------+---------+--------
  2 | f       | Jana  | Vdana    | 7951010611  | FY10    |      1
  2 | f       | Jana  | Vdana    | 7951010611  | FY10    |      3
(2 řádky)
```

DELETE FROM

```
rimmer1=> DELETE FROM hodnoceni WHERE znamka >= 2*2 AND id_zaci = 2;
DELETE 0
rimmer1=> DELETE FROM hodnoceni; -- bacha, bez WHERE se smaze vsechno!
DELETE 5
```

DROP TABLE, VIEW

```
rimmer1=> DROP VIEW IF EXISTS zaci;
DROP VIEW
```

```
rimmer1=> DROP TABLE studenti;
ERROR:  cannot drop table studenti because other objects depend on it
DETAIL:  constraint znamky_zaci_id_fkey on table hodnoceni depends on table studenti
rimmer1=> DROP TABLE IF EXISTS hodnoceni;
DROP TABLE
rimmer1=> DROP TABLE studenti;
DROP TABLE
```

MySQL/MariaDB, SQLite, Oracle

Protože byla tahle kapitola tak extrémě nenáročná, přepsat SQL příkazy do ostatních databází vám dám za domácí úkol. Trochu napovím: MySQL nemá funkci TO_DATE a ignoruje CHECK, který navíc musí být na konci definice sloupce. MySQL, SQLite ani Oracle nemají typ SERIAL. Oracle navíc nemá ani AUTO_INCREMENT, takže se musí hodnoty v primárním klíči vždy INSERTovat pomocí sequence.NEXTVAL. Oracle nezná klauzuli IF EXISTS. SQLite neumí používat DEFAULT v INSERTu. Oracle nepoužívá AS pro přejmenování tabulky (ale bez AS to umí). SQLite má jen omezený příkaz ALTER TABLE. V Oracle byste měli používat typ VARCHAR2místo typu VARCHAR.

Na zbytek už určitě přijdete sami.

SELECT II

V této kapitole se dozvíte o dalších kouzlech, co umí příkaz SELECT. Dosud jste se naučili selektovat jen řádky a sloupce, tak jak jsou uložené v databázi. Teď se naučíte, jak zpracovávat souhrné (statistické) informace. Level vašich znalostí o SQL zase stoupne o jednu laťku nahoru.

- WHERE
- ORDER BY
  - NULLS FIRST
- COUNT, AVG, MIN, MAX, SUM, STDDEV
- GROUP BY
- HAVING
- LIMIT a OFFSET
- MySQL/MariaDB
  - SQLite
  - Oracle
  - LIMIT A OFFSET

WHERE

Podmínku WHERE jste se naučili používat již v kaptiole Úprava tabulky. Tuto podmínku, na rozdíl od následujících, využívají i příkazy jako DELETE FROM a UPDATE. Odkaz na tuto podmínku je tady jen pro úplnost.

V této kapitole využiji tabulky z příkladu kapitoly Vytváření relací. Dozvíte se tu spoustu zajímavých a potřebných věcí, které byste měli velmi dobře znát.

ORDER BY

ORDER BY způsobí setřídění řádků podle nějakého sloupce (abecedně nebo podle velikosti čísel). ORDER BYse píše až za podmínku WHERE (pokud ji použijete).

Na začátek ještě vložím jeden řádek do databáze, který se bude v této lekci později hodit.

```
INSERT INTO zamestnanci VALUES ('Pan', 'Novy', NULL, 1, 8001010612);
```

A teď už příklad k ORDER BY.

```
rimmer1=> SELECT * FROM zamestnanci ORDER BY prijmeni;
  jmeno  | prijmeni | plat  | oddeleni_id | rodne_cislo
---------+----------+-------+-------------+-------------
 Tom     | Jerry    | 15000 |           2 | 8001010605
 Leopold | King     | 13000 |           2 | 8001010607
 Martin  | Luter    | 12000 |           2 | 8001010606
                    ...
(11 rows)

rimmer1=> SELECT * FROM zamestnanci WHERE plat >=15000 ORDER BY prijmeni;
 jmeno | prijmeni | plat  | oddeleni_id | rodne_cislo
-------+----------+-------+-------------+-------------
 Tom   | Jerry    | 15000 |           2 | 8001010605
 Tomas | Mann     | 22000 |           3 | 8001010608
 Lenka | Pavova   | 15000 |           2 | 8001010604
 Vasek | Trn      | 16000 |           3 | 8001010609
(4 rows)
```

Setřídění probíhá implicitně vzestupně – **ASC**, nebo můžete třídit sestupně – **DESC**.

```
rimmer1=> SELECT * FROM zamestnanci WHERE plat >=15000 ORDER BY plat DESC;
 jmeno | prijmeni | plat  | oddeleni_id | rodne_cislo
-------+----------+-------+-------------+-------------
 Tomas | Mann     | 22000 |           3 | 8001010608
 Vasek | Trn      | 16000 |           3 | 8001010609
 Lenka | Pavova   | 15000 |           2 | 8001010604
 Tom   | Jerry    | 15000 |           2 | 8001010605
(4 rows)
```

Třídit lze podle více sloupců než podle jednoho. V následujícím SELECTu setřídím výsledek nejdříve podle platu (setstupně) a tam kde je plat stejný setřídím záznam podle jména (vzestupně).

Priorita třídění je dána pořadím sloupců za klíčovými slovy ORDER BY.

```
rimmer1=> SELECT * FROM zamestnanci WHERE plat >=15000
          ORDER BY plat DESC, prijmeni;
 jmeno | prijmeni | plat  | oddeleni_id | rodne_cislo
-------+----------+-------+-------------+-------------
```

```
Tomas | Mann    | 22000 |         3 | 8001010608
 Vasek | Trn    | 16000 |         3 | 8001010609
 Tom   | Jerry  | 15000 |         2 | 8001010605
Lenka | Pavova | 15000 |         2 | 8001010604
(4 rows)
```

V klauzuli ORDER BY můžete použít nejen jméno sloupce, ale v podstatě jakýkoliv výraz, který se může použít i za SELECTem. Můžete zkrátka třídit podle jakéhokoliv sloupce, který se objeví ve výsledku. A co víc, můžete dokonce třídit i podle sloupců (nebo výrazů), které se neobjeví ve výsledku!

```
rimmer1=> SELECT * FROM zamestnanci WHERE plat >=15000
ORDER BY plat % 5000, plat DESC, prijmeni;
 jmeno | prijmeni | plat  | oddeleni_id | rodne_cislo
-------+----------+-------+-------------+-------------
 Tom   | Jerry    | 15000 |         2 | 8001010605
Lenka | Pavova   | 15000 |         2 | 8001010604
 Vasek | Trn      | 16000 |         3 | 8001010609
Tomas | Mann     | 22000 |         3 | 8001010608
(4 řádky)
```

NULLS FIRST

Podívejte se na následující příklad:

```
rimmer1=> SELECT * FROM zamestnanci WHERE oddeleni_id = 1 ORDER BY plat;
 jmeno | prijmeni | plat  | oddeleni_id | rodne_cislo
-------+----------+-------+-------------+-------------
Lenka | Pavova   | 10000 |         1 | 8001010601
 Jana  | Pavova   | 10000 |         1 | 8001010602
 Jana  | Mala     | 12000 |         1 | 8001010603
  Pan  | Novy     |       |         1 | 8001010612
(4 řádky)
```

Dá se nějak rozumě vysvětlit, proč je Pan Novy poslední, když jeho plat je NULL? Jestli vás napadlo, že je NULL menší než všechny čísla, tak jste nedávali pozor. NULL znamená, že nevím jaký má Pan Novy plat. Důvot je ten, že Postgres prostě implicitně hodnoty s NULL řadí na konec.

To se dá změnit pomocí klauzule NULLS FIRST:

```
rimmer1=> SELECT * FROM zamestnanci WHERE oddeleni_id = 1
ORDER BY plat NULLS FIRST, prijmeni, jmeno;
 jmeno | prijmeni | plat  | oddeleni_id | rodne_cislo
-------+----------+-------+-------------+-------------
  Pan  | Novy     |       |         1 | 8001010612
 Jana  | Pavova   | 10000 |         1 | 8001010602
Lenka | Pavova   | 10000 |         1 | 8001010601
 Jana  | Mala     | 12000 |         1 | 8001010603
(4 řádky)
```

Pozor! Pokud použijete NULLS FIRST, je jedno jestli řadíte sloupec sestupně (DESC) nebo vzestupně (ASC), hodnoty s NULL budou vždy první.

Za domácí úkol zjistětě z dokumentace, jak se dá explicitně říct, že mají být hodnoty NULL na konci. (Buďte raději paranoidní a implicitnímu nastavení nevěřte. Nikdy nevíte, kdy se může změnit!)

Pokud nepoužijete ORDER BY, řádky se vracejí v náhodném pořadí. Možná by se vám z výsledků mohlo někdy zdát, že se vracejí v pořadí v jakém jste je vložili do tabulky, ale to je jen „náhoda". Bez ORDER BYse opravdu nemůžete na pořadí spolehnout.

COUNT, AVG, MIN, MAX, SUM, STDDEV

Příkaz SELECT může získat některé „statistické" údaje pomocí následujících agregačních funkcí: Průměr AVG, minimum MIN, maximum MAX, součet SUM, počet položek COUNT, variace VARIACE a směrodatná odchylka STDDEV. Celý výčet agregačních funkcí najdete v dokumentaci PostgreSQL.

```
rimmer1=> SELECT COUNT(plat) FROM zamestnanci;
 count
-------
    10
(1 řádka)
```

Příkaz vrátil počet řádků, kde je uveden plat, tj. kde plat není NULL. Pokud vás zajímá počet řádků (tj. bez ohledu na to, zda je nebo není v nějakém sloupci NULL), použijte funkci COUNT s hvězdičkou *.

Zkusím přidat zaměstnance, kde neuvedu jeho plat. Nezapomeňte, že 0 je také číslo, takže jej COUNTzapočítává. Proto je třeba uvést NULL. Počet „platů" v tabulce se tím nezmění (přestože počet řádků ano).

```
rimmer1=> SELECT COUNT(plat) AS platů, COUNT(*) AS řádek FROM zamestnanci;
 platů | řádek
-------+-------
    10 |    11
(1 řádka)
```

Další příklad ukazuje funkci MAX:

```
rimmer1=> SELECT MAX(plat) FROM zamestnanci;
  max
-------
 22000
(1 row)
```

Všimli jste si něčeho zajímavého? Když jsem popisoval speciální hodnotu NULL, psal jsem, že NULLznamená, že nevím, co tam je. Jak teda můžu vědět, když mám v sloupečku plat jednu hodnotu NULL, jaká je maximální hodnota? Neměl by být výsledek taky NULL (že nevím, jaký je výsledek)?

Věc se má tak, že MAX prostě NULL ignoruje. Jediný případ, kdy by MAX vrátilo NULL by bylo, kdyby bylo NULL ve všech řádcích. Obdobně to funguje i s ostatními agregačními funkcemi.

Další příklad už asi nepotřebuje komentář:

```
rimmer1=> SELECT
MAX(plat) AS "maximalni plat",
MIN(plat) AS "minimalni plat",
COUNT(plat) AS "pocet platu",
COUNT(*) AS radku,
AVG(plat) AS prumerny_plat
FROM zamestnanci;
 maximalni plat | minimalni plat | pocet platu | radku |    prumerny_plat
----------------+----------------+-------------+-------+---------------------
          22000 |           9000 |          10 |    11 | 13400.000000000000
(1 řádka)
```

```
rimmer1=> SELECT
MAX(plat) AS "maximalni plat",
MIN(plat) AS "minimalni plat",
COUNT(plat) AS "pocet platu",
COUNT(*) AS radku,
AVG(plat) AS "prumerny plat"
FROM zamestnanci
WHERE oddeleni_id = 1;
 maximalni plat | minimalni plat | pocet platu | radku |     prumerny plat
----------------+----------------+-------------+-------+-----------------------
          12000 |          10000 |           3 |     4 | 10666.6666666666666667
(1 řádka)
```

```
rimmer1=> SELECT STDDEV(plat) FROM zamestnanci;
        stddev
----------------
 3835.5066303047
(1 row)
```

GROUP BY

Předchozí (agregační) funkce pracují s celou tabulkou a vrátí jeden souhrný řádek. Pomocí podmínky WHEREmůžete některé řádky ze zpracování vyřadit, ale výsledek pak bude stále jeden souhrný řádek pro všechny zbylé řádky.

Představte si, že chcete zjistit výsledky pro každé oddělení zvlášť. Jednou z možností je pro každé oddělení použít podmínku WHERE oddeleni_id = číslo oddělení. Tak ale budete muset spustit pro každé oddělení extra SELECT. Samozřejmě to jde jednodušeji (tak si nestěžujte že se musíte učit zase něco nového, ušetří vám to spoustu práce :-).

K tomu, abyste zjistili souhrnná data (maximum, minimum atd.) pro skupiny řádků, slouží klauzule **GROUP BY**, která řádky seskupí (vytvoří skupiny z řádků obsahujících stejnou hodnotu v seskupovaném sloupci).

V příkladu zobrazím minima, maxima, počet vyplněných platů a počet záznamů oddělení.

```
rimmer1=> SELECT MAX(plat), MIN(plat), COUNT(plat), COUNT(*) AS "řádek"
FROM zamestnanci GROUP BY oddeleni_id;
  max  |  min  | count | řádek
-------+-------+-------+-------
 12000 | 10000 |     3 |     4
 22000 |  9000 |     3 |     3
 15000 | 12000 |     4 |     4
(3 řádky)
```

Z předchozího příkladu nepoznáte, kterému oddělení patří který řádek. Naštěstí sloupce podle kterých se seskupuje (jsou uvedeny v GROUP BY) můžete přidat do výpisu.

Pokud použijete GROUP BY, smíte vypsat pouze sloupce podle kterých se seskupuje (jsou za GROUP BY), nebo agregované sloupce.

Výsledný řádek je totiž vždy výsledkem zpracování několika řádků. Kdybyste chtěli vypsat sloupec bez použití agregační funkce, nebylo by jasné, z jakého řádku se má použít hodnota do výsledku. Tento problém se sloupci z GROUP BY odpadá – každý řádek výsledku je seskupením řádků pro **stejnou hodnotou** groupovacího sloupečku.

```
rimmer1=> SELECT oddeleni_id, MAX(plat), MIN(plat), COUNT(plat), STDDEV(plat)
FROM zamestnanci GROUP BY oddeleni_id ORDER BY oddeleni_id;
 oddeleni_id |  max  |  min  | count |      stddev
-------------+-------+-------+-------+------------------
           1 | 12000 | 10000 |     3 | 1154.700538379252
           2 | 15000 | 12000 |     4 | 1500.000000000000
           3 | 22000 |  9000 |     3 | 6506.407098647712
(3 řádky)
```

Ještě pár příkladů:

```
rimmer1=> SELECT oddeleni_id, MAX(plat), MIN(plat), COUNT(plat)
FROM zamestnanci WHERE oddeleni_id !=3 GROUP BY oddeleni_id
ORDER BY MAX(plat) DESC;
 oddeleni_id |  max  |  min  | count
-------------+-------+-------+-------
           2 | 15000 | 12000 |     4
           1 | 12000 | 10000 |     3
(3 rows)
```

```
rimmer1=> SELECT max(plat), min(plat), plat FROM zamestnanci;
ERROR:  column "zamestnanci.plat" must appear in the GROUP BY clause or be used in an aggregate function
ŘÁDKA 1: SELECT max(plat), min(plat), plat FROM zamestnanci;
                                      ^
```

I agregační funkce lze použít v klauzuli ORDER BY.

Pokud přeci jen chcete vypsat sloupec, který není v GROUP BY, protože víte že i tak je ve všech řádcích použitých pro výsledný řádek hodnota stejná, použijte třeba funkci MAX. Když víte, že jsou hodnoty stejné, tak MAX nemůže vrátit nic jiného, než tu hodnotu (stejně tak MIN).

```
rimmer1=> SELECT oddeleni_id,
          MAX(o.nazev) AS nazev,
          max(plat), min(plat)
          FROM oddeleni o, zamestnanci z
          WHERE o.prim_klic = z.oddeleni_id
          GROUP BY oddeleni_id
          ORDER BY oddeleni_id;
 oddeleni_id |      nazev      |  max  |  min
-------------+-----------------+-------+-------
           1 | Sekretariat     | 12000 | 10000
           2 | Pravni oddeleni | 15000 | 12000
           3 | Pravni oddeleni | 22000 |  9000
(3 řádky)
```

V klauzuli GROUP BY můžete uvést více sloupců. Řekněme třeba, že mě zajímá jak často se která výše platu objevuje v závislosti na oddělení:

```
rimmer1=> SELECT plat,oddeleni_id,count(plat)
          FROM zamestnanci
          GROUP BY oddeleni_id, plat
          ORDER BY plat DESC, oddeleni_id;
 plat  | oddeleni_id | count
-------+-------------+-------
       |           1 |     0
 22000 |           3 |     1
 16000 |           3 |     1
 15000 |           2 |     2
 13000 |           2 |     1
 12000 |           1 |     1
 12000 |           2 |     1
 10000 |           1 |     2
  9000 |           3 |     1
(9 řádek)
```

Z výsledku se třeba dočtu, že plat 12000 se vyskytuje jednou v oddělení id 1 a jednou v oddělení id 2 a že plat 10000 se objevuje jen dvakrát v oddělení id 1. Na prvním řádku je plat NULL. Objevuje se v oddělení s id 1, ale COUNT vrací 0, protože hodnotu NULL funkce COUNT(plat) nezapočítá. (Zkuste místo toho COUNT(*).)

Poznámka: Pořadí klauzulí WHERE … GROUP BY … ORDER BY nelze zaměňovat.

HAVING

Podmínka HAVING je ekvivalentní podmínce WHERE jen s tím rozdílem, že podmínka WHERE se aplikuje před provedením příkazu GROUP BY (odstraní řádky před zgroupováním) a HAVING se aplikuje až na řádky vytvořené po zgroupování (tedy na řádky výsledku SELECT … GROUP BY).

V prvním příkladu vyberu z tabulky zamestnanci jen ty řádky, které neobsahují oddělení č. 1 a z nich získám souhrnné informace pro ostatní oddělení. V druhém příkladu získám souhrnné informace ze všech oddělení, ale výslednou tabulku omezím na oddělení, které nemají oddeleni_id č. 1.

Dvěma různými cestami tak získám stejný výsledek.

```
rimmer1=> SELECT oddeleni_id, MAX(plat), MIN(plat), COUNT(plat)
FROM zamestnanci WHERE oddeleni_id <> 1 GROUP BY oddeleni_id;
 oddeleni_id |  max  |  min  | count
-------------+-------+-------+-------
           2 | 15000 | 12000 |     4
           3 | 22000 |  9000 |     3
(2 rows)

rimmer1=> SELECT oddeleni_id, MAX(plat), MIN(plat), COUNT(plat)
FROM zamestnanci GROUP BY oddeleni_id HAVING oddeleni_id != 1;
 oddeleni_id |  max  |  min  | count
-------------+-------+-------+-------
           2 | 15000 | 12000 |     4
           3 | 22000 |  9000 |     3
(2 rows)
```

Mezi operátorem <> a != není rozdíl, oboje znamená „nerovná se" a oboje můžete libovolně zaměňovat.

Zajímavější bude následující příklad, kde zobrazím jen ta oddělení, kde je maximální plat větší jak 15000. Vzhledem k tomu, že se omezení týká výsledku agregační funkce, lze toho dosáhnout pouze pomocí HAVING(WHERE se aplikuje před zjištěním tohoto výsledku).

```
rimmer1=> SELECT oddeleni_id, MAX(plat),MIN(plat), COUNT(plat)
FROM zamestnanci GROUP BY oddeleni_id HAVING MAX(plat) > 15000;
 oddeleni_id |  max  | min  | count
-------------+-------+------+-------
           3 | 22000 | 9000 |     3
(1 row)
```

Všimněte si, že zatímco WHERE se píše před GROUP BY, HAVING se píše až za. To není náhoda :-).

PS: Nic vám nebrání používat WHERE i HAVING v jednom SQL dotazu :-).

LIMIT a OFFSET

LIMIT a OFFSET jsou poslední dvě věci, které vám chyběli, abyste mohli pracovat s tabulkami o milionech řádků.
LIMIT omezí počet vrácených řádků a OFFSET přeskočí zadaný počet prvních řádek.
Jejich použití je tak jednoduché a jasné, že bych snad ani nemusel dělat příklad. Ale pro jistotu:

```
rimmer=> SELECT * FROM zamestnanci ORDER BY prijmeni LIMIT 3;
 jmeno  | prijmeni | plat  | oddeleni_id | rodne_cislo
--------+----------+-------+-------------+-------------
 Tom    | Jerry    | 15000 |           2 | 8001010605
 Leopold| King     | 13000 |           2 | 8001010607
 Martin | Luter    | 12000 |           2 | 8001010606
(3 rows)

rimmer=> SELECT * FROM zamestnanci ORDER BY prijmeni LIMIT 3 OFFSET 1;
 jmeno  | prijmeni | plat  | oddeleni_id | rodne_cislo
--------+----------+-------+-------------+-------------
 Leopold| King     | 13000 |           2 | 8001010607
 Martin | Luter    | 12000 |           2 | 8001010606
 Jana   | Mala     | 12000 |           1 | 8001010603
(3 řádky)
```

LIMIT i OFFSET musí být kladná celá čísla.
Poznámka: příkaz OFFSET lze použít i bez klauzule LIMIT.

MySQL/MariaDB

Výčet agregačních funkcí MySQL najdete v dokumentaci. Asi vás nepřekvapí, že se od PostgreSQL lyší.
MySQL má trochu jinou strategii pro pojmenování sloupců kde použijete agregační funkce než PostgreSQL. Raději vždy používejte vlastní jména pomocí AS.
MySQL také umožní přidat do SELECTu sloupce, které nejsou součástí GROUP BY. Hodnota v takovém sloupci se vybere náhodně z řádků, které jsou seskupené (v příkladě je hodnota třetího sloupce náhodně vybrána ze všech řádků tabulky). Že je to fuj a že byste měli používat trik s MAX (nebo MIN) asi nemusím dodávat.

```
SELECT max(plat), min(plat), plat FROM zamestnanci;
+-----------+-----------+-------+
| max(plat) | min(plat) | plat  |
+-----------+-----------+-------+
|     22000 |      9000 | 10000 |
+-----------+-----------+-------+
1 row in set (0.00 sec)
```

MySQL nemá pro ORDER BY klauzuli NULL FIRST, ale může si pomoci jiným trikem: Nejdříve se seřadí sloupec plat podle toho, jestli je nebo není NULL (použije se funkce ISNULL) a pak už podle samotného sloupce plat.

```
mysql> SELECT * FROM zamestnanci WHERE oddeleni_id = 1
ORDER BY plat IS NULL DESC, plat, prijmeni, jmeno;
+-------+----------+-------+-------------+-------------+
| jmeno | prijmeni | plat  | oddeleni_id | rodne_cislo |
+-------+----------+-------+-------------+-------------+
| Pan   | Novy     | NULL  |           1 | 8001010612  |
| Jana  | Pavova   | 10000 |           1 | 8001010602  |
| Lenka | Pavova   | 10000 |           1 | 8001010601  |
| Jana  | Mala     | 12000 |           1 | 8001010603  |
+-------+----------+-------+-------------+-------------+
4 rows in set (0.00 sec)
```

Tento trik funguje i v PostgreSQL a SQLite, ale ne v Oracle (tam alespoň funguje NULLS IF).
Příjdete na to, co z SQL dotazu odstranit, aby byli hodnoty s NULL na konci?

MySQL umí LIMIT a OFFSET jako Postgres (jenom vyžaduje toto pořadí), ale umí ještě něco navíc, co je (bohužel) hojně používáno.
Pokud zapíšete za LIMIT dvě čísla (oddělená čárkou), první se bere jako offset a druhé jako limit.
Typický MySQLák píše takto:

```
mysql> SELECT * FROM zamestnanci ORDER BY prijmeni LIMIT 1,3;
+---------+----------+-------+-------------+-------------+
| jmeno   | prijmeni | plat  | oddeleni_id | rodne_cislo |
+---------+----------+-------+-------------+-------------+
| Leopold | King     | 13000 |           2 | 8001010607  |
| Martin  | Luter    | 12000 |           2 | 8001010606  |
| Jana    | Mala     | 12000 |           1 | 8001010603  |
+---------+----------+-------+-------------+-------------+
3 rows in set (0.00 sec)
```

Je to kratší, ale nestandardní.

SQLite

Agregační funkce SQLite také najdete v nápovědě. Zjistíte, že SQLite nemá STDDEV (standardní odchylku) a není moc, co byste stím mohli udělat (leda si tuto funkci doprogramovat – to nežertuji).
SQLite umožňuje přidat sloupeček do SELECTu, který není součástí GROUP BY stejně jako MySQL.
SQLite neumí NULLS FIRST. Můžete ale použít stejný trik, jako u MySQL.
SQLite umí, stejně jako MySQL, mimo standardního LIMIT a OFFSET (pouze v tomto pořadí, pokud použijete obé) i nestandardní offset s limitem pomocí LIMIT offset,limit.

Oracle

Agregační funkce najdete, jako vždy, v dokumentaci.
Oracle neumožňuje SELECTovat sloupeček, který není součástí GROUP BY. Trik s MAX samozřejmě funguje.
Oracle umí NULLS FIRST. Pokud byste ale chtěli použít trik s ORDER BY plat IS NULL, tak to neumí, protože neumí použít plat IS NULL jako součást klauzule SELECT (na rozdíl od ostatních databází).

Můžete použít jiný trik, který bude fungovat všude. Používá ale CASE, který budu vysvětlovat až v příští kapitole. Takže teď ten trik ukáži prozatím bez vysvětlení:

```
oracle> SELECT * FROM zamestnanci WHERE oddeleni_id = 1
ORDER BY CASE WHEN plat IS NULL THEN 0 ELSE 1 END,
plat, prijmeni, jmeno;
```

JMENO	PRIJMENI	PLAT	ODDELENI_ID	RODNE_CISL
Pan	Novy		1	8001010612
Jana	Pavova	10000	1	8001010602
Lenka	Pavova	10000	1	8001010601
Jana	Mala	12000	1	8001010603

LIMIT a OFFSET

Oracle nemá LIMIT ani OFFSET. To je bezesporu šokující zjištění. Abyste mohli omezit výstupní řádky, budete muset použít trik, který využívá dvě novinky: pseudosloupeček ROWNUM a něco co už umíte, jen o tom nevíte – SELECT ze SELECTu.

Tak nejdřív co je to ten SELECT ze SELECTu. V povídání o CREATE VIEW jste se dozvěděli, že výsledkem každého SELECTu je tabulka. Pomocí CREATE VIEW jste si tuto „tabulku" pojmenovali a mohli jste z ní SELECTovat. Ono to jde ale i přímo ze SELECTu!

```
SELECT * FROM  (
SELECT * FROM zamestnanci ORDER BY prijmeni
) z;
```

Toto funguje ve všech DBMS. Rozdíl je jen v pojmenování výsledku vnitřního SELECTu. MySQL a PostgreSQL vyžadují, aby byl pojmenovaný. Ostatní databáze to umožňují, ale nevyžadují. Oracle to také umožňuje (viz příklad), ale nemůžete použít AS nazev_tabulky (slovo AS tam nesmí být). Všechny ostatní DBMS AS povolují (ale nevyžadují).

ROWNUM je pseudosloupeček, který obsahuje číslo řádku **výsledné tabulky** SELECTu. ROWNUM umí ze zde zmiňovaných databází jen Oracle.

ROWNUM je právě to, co se dá použít pro omezení počtu řádků.

Následuje příklad LIMITu:

```
oracle> SELECT * FROM  (
SELECT * FROM zamestnanci ORDER BY prijmeni
)
WHERE ROWNUM <= 3;
```

JMENO	PRIJMENI	PLAT	ODDELENI_ID	RODNE_CISL
Tom	Jerry	15000	2	8001010605
Leopold	King	13000	2	8001010607
Martin	Luter	12000	2	8001010606

Pokud by vás napadlo, že by to šlo jednodušeji a vystačili byste si s jedním příkazem, tak nevystačili. Problém je v tom, že při použití ORDER BY se řádky SELECTují v zatím neseřazeném pořadí, už se jim přiděluje ROWNUM a porovnává se ve WHERE podmínce.

```
oracle> SELECT * FROM zamestnanci WHERE ROWNUM <= 3 ORDER BY prijmeni;
```

JMENO	PRIJMENI	PLAT	ODDELENI_ID	RODNE_CISL
Jana	Mala	12000	1	8001010603
Lenka	Pavova	10000	1	8001010601
Jana	Pavova	10000	1	8001010602

Výsledekem je, že sice vyberete 3 řádky seřazené podle příjmení, ale ne první 3 řádky. Vyberete náhodné 3 řádky, které projdou podmínkou WHERE. Až na nich se provede seřazení pomocí ORDER BY.

Jak simulovat OFFSET? Možná vás napadlo toto:

```
SELECT * FROM (
SELECT * FROM zamestnanci ORDER BY prijmeni
) z  WHERE ROWNUM BETWEEN 2 AND 4;
```

To ale nevrátí žádný řádek. Proč? ROWNUM se přiděluje řádkům, které jsou součástí výsledku dotazu. První řádek má číslo 1, druhý řádek č. 2 atd. Jenže, když první řádek neprojde podmínkou WHERE, stává se z dalšího řádku v řadě řádek první. Takže ten taky neprojde podmínkou WHERE atd. atd.

SELECT s podmínkou WHERE ROWNUM > 1 nikdy nevrátí žádný řádek.

Kvůli **OFFSETu** se musí provést ještě jeden vnořený SELECT navíc.

```
oracle> SELECT * FROM (
SELECT z.*, ROWNUM rnum FROM (
SELECT * FROM zamestnanci z ORDER BY prijmeni
) z WHERE ROWNUM <= 4
)
WHERE rnum >= 2;
```

JMENO	PRIJMENI	PLAT	ODDELENI_ID	RODNE_CISLO	RNUM
Leopold	King	13000	2	8001010607	2
Martin	Luter	12000	2	8001010606	3
Jana	Mala	12000	1	8001010603	4

Funguje to asi takhle: Dva vnořené SELECTy fungují jako LIMIT. Prostřední SELECT navíc udělá z ROWNUM (toho nejvíce vnořeného SELECTu) regulérní non-pseudo sloupec **rnum**. V posledním, vnějším SELECTu už se s tímto sloupcem může pracovat jako s jakýmkoliv jiným reálným sloupcem.

Je to hnus, ale je to Oraclem doporučovaný oficiální postup pro LIMIT a OFFSET. Dobrá zpráva aspoň je, že ten úplně vnitřní SELECT nevybere úplně všechny data z disku do paměti, kde by se teprve vybrali požadované řádky. Díky optimalizaci dotazů dokáže Oracle takovéto SELECTy provádět opravdu rychle (jako jakákoliv jiná databáze s LIMITem a OFFSETem).

V Oracle verze 12c už můžete použít top N queries klauzuli.

Funkce

S funkcemi jste se již určitě setkali. Na každé kalkulačce máte funkce, například sínus, mocnina, odmocnina atp. V této kapitole se budu věnovat chápání a využívání funkcí v SQL databázích.
Funkce si můžete dokonce naprogramovat vlastní, ale tomu se v této kaptiole věnovat nebudu.

Co jsou funkce

Funkce jsou vlastně databázové objekty, které však neslouží k uchovávání dat jako tabulky, ale k manipulaci s daty nebo k získávání (výpočtu) dat. S funkcemi jste se již setkali. Například statistické funkce. Funkce mohou mít argumenty a návratovou hodnotu, nebo obojí.
Argumenty jsou data, se kterými se manipuluje nebo které se používají pro získání výsledků funkce. Například funkce MAX(argument) má jeden argument (kterým je název sloupce). Argumenty se uvádějí v závorce za jménem funkce. Pokud jich je více, oddělují se čárkou.
Funkcí která nemá argumenty je například funkce RANDOM, která vrací náhodné číslo v intervalu <0,1>. **I funkce bez argumentu se musí volat se závorkami za svým jménem.**
Návratová hodnota funkce může být číslo, řetězec, tabulka atp. V případě agregační funkce MAX je to největší hodnota ze sloupce, který je jí předán jako argument.
Pokud chcete funkce vyzkoušet „nanečisto", pak je můžete spustit pomocí příkazu SELECT (viz tabulka dual).

```
rimmer=> SELECT LENGTH('sallyx') FROM dual;
 length
--------
      6
(1 řádka)

rimmer=> SELECT RANDOM() FROM dual;
       random
--------------------
 0.572437878232449
(1 řádka)
```

Druhy funkcí

Skalární funkce

Skalární funkce mají jednu hodnotu v jednom argumentu (jedno číslo, jeden řetězec, nikoliv celý sloupec) a jednu hodnotu na výstupu.
Například funkce LENGTH(retezec). Jejím argumentem je jeden řetězec a návratovou hodnotou číslo (délka řetězce předaného jako argument). Skalární funke může být součástí libovolného výrazu.

```
rimmer1=> SELECT
prijmeni, jmeno, LENGTH(prijmeni)+LENGTH(jmeno) AS "Délka jména"
FROM zamestnanci;
 prijmeni |  jmeno  | Délka jména
----------+---------+-------------
 Pavova   | Lenka   |          11
 Pavova   | Jana    |          10
 Mala     | Jana    |           8
 Pavova   | Lenka   |          11
 Jerry    | Tom     |           8
 Luter    | Martin  |          11
 King     | Leopold |          11
 Mann     | Tomas   |           9
 Trn      | Vasek   |           8
 Osel     | Stary   |           9
 Novy     | Pan     |           7
(11 rows)
```

Agregační funkce

Výstupem (návratovou hodnotou) agregační funkce je také jedna hodnota. Na vstupu je však hodnotou množina hodnot (sloupec).
Množinou hodnot jsou buď všechny řádky ve sloupci, nebo jejich část seskupená pomocí GROUP BY. Tyto funkce se mohou použít jednak v seznamu sloupců za klíčovým slovem SELECT, nebo v podmínce pro skupinu řádků za klíčovým slovem HAVING, i v klauzuli LIMIT.
Příklady použití agregačních funkcích jste viděli v předešlé kapitole.

```
rimmer1=> SELECT MAX(plat) FROM zamestnanci;
  max
-------
 22000
(1 row)

rimmer1=> SELECT MAX(plat) FROM zamestnanci GROUP BY oddeleni_id;
```

```
 max
-------
 12000
 15000
 22000
(3 rows)

rimmer1=> SELECT oddeleni_id, MAX(plat) FROM zamestnanci GROUP BY oddeleni_id
HAVING AVG(plat) >= 11000 ORDER BY MAX(plat) DESC;
 oddeleni_id |  max
-------------+-------
           3 | 22000
           2 | 15000
(2 rows)
```

Funkce SQL

Kompletní seznam funkcí najdete v oficiální dokumentaci. Já jsem pro vás vybral pár těch nejzajímavějších.

Agregační funkce

Většinu agregačních funkcí jsem ukázal v kapitole SELECT II. Pro úplnost je zde vypíšu v tabulce.

Název	Význam	Typ agrumentu
AVG()	Vrátí průměrnou hodnotu.	smallint, int, bigint, real, double precision, numeric, interval
COUNT()	Vrátí počet položek v argumentu (které nejsou NULL)	Všechny typy
MAX()	Vrátí největší hodnotu.	jakékoliv pole, číslo, řetězec, nebo datum/čas
MIN()	Vrátí nejmenší hodnotu.	viz MAX
SUM()	Vrátí součet hodnot v množině.	bigint,double precision, integer, interval, money, numeric, real, smallint
STDDEV()	Spočte směrodatnou odchylku.	smallint, int, bigint, real, double precision, or numeric
VARIANCE()	Rozptyl hodnot v množině.	smallint, int, bigint, real, double precision, or numeric

Funkce pro práci s čísly

Název	Návratová hodnota
ABS(x)	Absolutní hodnota.
CINT(x)	Převede jakýkoliv číselný datový typ nebo řetězec (ve kterém je zapsáno číslo) na celé číslo.
CBRT(x)	Třetí odmocnina x.
COS(x)	Cosínus zadaného úhlu v radiánech.
EXP(x)	e na x-tou, kde e=2.7182 a x je reálné číslo (nikoliv celé číslo!). Celé číslo můžete na reálné převést například pomocí funkce ROUND takto: EXP(ROUND(x,0)) (x nesmí být příliš velké číslo).
FACTORIAL(x)	faktoriál x, kde x je celé číslo.
FLOOR(x)	Vrátí nejbližší celé číslo menší nebo rovno argumentu.
MOD(x,y)	Zbytek po celočíselném dělení x/y (x a y jsou celá čísla).
POWER(x,y)	x na y
RANDOM()	Náhodné číslo v intervalu <0,1> (Bez argumentu, závorky jsou přesto nutné!).
ROUND(x,y)	Číslo x se zaokrouhlí na y desetinných míst.
SIN(x)	Sínus zadaného úhlu v radiánech.
SQRT(x)	Druhá odmocnina x, kde x je reálné číslo (nikoliv celé číslo! viz funkce EXP výše).
TAN(x)	Tangens zadaného úhlu v radiánech.

Funkce pro práci s řetězci

Název	Návratová hodnota
retezec || retezec	|| - toto není funkce ale operátor, který spojí dva řetězce v jeden. V jiných databázích (než PostgeSQL) se pro spojování řetězců používá funkce CONCAT, nebo operátor +. Třeba MySQL používá || jako logický operátor „nebo".
CONCAT(str,str,...)	Spojení řetězců
INITCAP(x)	Převede u všech slov v řetězci *x* první písmeno na velké a všechna další ve slově na malé.
LOWER(x)	Převede všechna písmena v řetězci *x* na malá.
LENGTH(x)	Vrací počet znaků v řetězci.
TRIM([leading | trailing | both] [characters] from string)	Ořízne všechny bílé znaky (mezery, tabulátory, nová řádky) nebo charactersz leva, nebo z prava, nebo zleva i prava z řetězce string.
UPPER(x)	Převede všechna pímsena v řetězci x na velká.

```
rimmer1=> SELECT
CONCAT('*', TRIM (' xAx '),                '*'),
CONCAT('*', TRIM (leading ' xAx '),        '*'),
CONCAT('*', TRIM (trailing ' x' from ' xAx '), '*')
FROM dual;
 concat | concat | concat
--------+--------+--------
 *xAx*  | *xAx * | * xA*
(1 řádka)
```

Funkce pro práci s časem

Název	Návratová hodnota
CURRENT_TIMESTAMP	Vrací aktuální datum s časem i časovou zónou (např. 2002-12-29 20:22:31.610797+00). Toto vlastně není funkce, ale **makro**, proto se nepoužívají závorky! Může se použít jako DEFAULT hodnota.
CURRENT_DATE	Vrací akutální datum.
CURRENT_TIME	Vrací aktuální čas.
DATE(x)	Převede *x* na datum. Argument *x* může být typu date, abstime, nebo text. (např. date('25.6.2002') vrátí datový typ date s hodnotou 2002-06-25).
- + * /	Toto nejsou funkce, ale operátory. PostgreSQL umožňuje data odečítat, sčítat, a násobit a dělit časové intervaly. Detaily uvidíte v dokumentaci.

```
rimmer=> SELECT DATE('7.7.2002') - DATE('9.7.2002') FROM dual;
 ?column?
----------
       -2

rimmer=> SELECT CURRENT_TIME - '21:05' FROM dual;
    ?column?
-----------------
 18:57:19.676169+01
(1 řádka)


rimmer=> SELECT INTERVAL '1 hour' / 1.5;
 ?column?
----------
 00:40:00
(1 řádka)
```

Funkce pro konverzi datových typů

Název	Návratová hodnota
TO_CHAR(x, vzor)	Převede datum (s časem), interval nebo číslo *x* na řetězec podle vzoru v argumentu *vzor*. Jak může vypadat vzor si najdete v dokumentaci. A to jak pro data, tak pro čísla.

Název	Návratová hodnota
TO_DATE(text, vzor)	Převede řetězec *text* na datum podle vzoru.
TO_TIMESTAMP(x, vzor)	Převede řetězec nebo číslo *x* na timestamp s časovou zónou podle vzoru.
TO_NUMBER(text, vzor)	Převede řetězec *text* na číslo podle vzoru.

PostgreSQL má ještě jiný mechanismus pro konverzi datových typů. A to dvě dvojtečky **::**. Pokud napíšete nějakou hodnotu (nebo sloupeček), za ní dvě dvojtečky a název datového typu, PostgreSQL se pokusí hodnotu převést na tento datový typ. Tento zápis asi nebudete často potřebovat, protože PostgreSQL se pokusí převést datový typ sám od sebe všude tam, kde je to potřeba. Ale občas se s tímto zápisem můžete někde setkat (třeba v dokumentaci).

```
rimmer1=> SELECT
TO_CHAR(CURRENT_TIMESTAMP, 'dd.mm.YYYY HH12:MI:SS'),
TO_CHAR(interval '15h 2m 12s', 'HH24:MI:SS'),
TO_CHAR(-125.8, '999D99S'),
TO_NUMBER('12,454.8-', '99G999D9S'),
TO_TIMESTAMP('05 Dec 2000', 'DD Mon YYYY'),
123.5::integer
FROM dual;
      to_char       | to_char  | to_char | to_number |      to_timestamp      | int4
--------------------+----------+---------+-----------+------------------------+------
07.12.2013 04:29:58 | 15:02:12 | 125,80- |  -12548.8 | 2000-12-05 00:00:00+01 |  124
```

```
-- implicitně převede '3' na int
rimmer1=> SELECT '3' + 3 from dual;
 ?column?
----------
        6

-- pokusí se převést '3.3' na int
rimmer1=> SELECT '3.3' + 3 from dual;
ERROR:  invalid input syntax for integer: "3.3"

-- implicitně převede '3.3' na float
rimmer1=> SELECT '3.3' + 3.0 from dual;
 ?column?
----------
      6.3

-- explicitně převede '3.3' na float
rimmer1=> SELECT '3.3'::float + 3 from dual;
 ?column?
----------
      6.3
```

Všimněte si, jak se snaží PostgreSQL uhodnout datový typ pro převod textového řetězce podle typu čísla ve výrazu.

Ostatní funkce

PostgreSQL obshuje celou řadu dalších funkcí a operátorů. Vzpomeňte si na to, kolik obsahuje PostgreSQL zvláštních datových typů. K nim se vážou i funkce. Takže máte funkce pro práci s XML, funkce pro práci s IP adresami, geometrické funkce, funkce pro práci s JSONem atd. Všechny samozřejmě najdete v dokumentaci.

MySQL/MariaDB, SQLite, Oracle

Z předchozích kapitol už víte, že s funkcemi a operátory je to v každé DBMS jinak. Například víte, že MySQL nemá funkci TO_DATE. Nebo že operátor ^ se používá v Postgresu na umocňování, zatímco v MySQL jako logická operace XOR. V této kapitole jste se už dozvěděli, že MySQL používá operátor || jako logické OR. Pro spojování řetězců umí používat funkci CONCAT, kterou ale zase neumí SQLite.

Kromě toho, že DBMS podporují různé sady funkcí a operátorů a toho, že některé operátory občas dělají něco úplně jiného, existuje ještě jedna záludnost. Někdy funkce, i když se jmenují a dělají totéž, nedělají to stejně!

Tak třeba funkce CONCAT může mít v Oracle jen dva argumenty (spojí jen dva řetězce do jednoho). SQLite pro jistotu CONCAT vůbec nemá. MySQL i PostgreSQL umožňují zadat do CONCAT libovolný počet argumentů (které spojí všechny v jeden řetězec).

Podívejte se na následující tabulku, která ukazuje výsledky spojování řetězců, pokud je jeden ze spojovaných výrazů NULL.

Operátor/Funkce	PostgreSQL	MySQL	SQLite	Oracle
CONCAT('text',NULL)	'text'	NULL		'text'
'text' || NULL	NULL		NULL	'text'

O tomhle chování se ani v dokumentaci nedočtete. Podle standardu a logiky věci by měl být výsledek vždy NULL. Jak je vidět, není …

Nebo funkce TRIM. Pokud budete chtít odebrat všechny mezery a hvězdičky ze začátku a konce řetězce, dopadnete asi takto:

```
-- PostgreSQL: vše dopadne jak má
rimmer1=> SELECT CONCAT('*',TRIM(' x' from ' xAx '),'*') FROM dual;
 concat
--------
```

```
*A*
(1 řádka)
-- MySQL: z prava se nic neodebere, MySQL bere ' x' jako celý řetězec
mysql> SELECT CONCAT('*',TRIM(' x' from ' xAx '),'*') as con FROM dual;
+--------+
| con    |
+--------+
| *Ax *  |
+--------+
-- MySQL: z leva se nic neodebere, MySQL bere 'x ' jako celý řetězec
mysql> SELECT CONCAT('*',TRIM('x ' from ' xAx '),'*') as con FROM dual;
+--------+
| con    |
+--------+
| * xA*  |
+--------+
```

V SQLite pro jistotu nejde určovat, které znaky se mají odstranit.

```
-- Oracle neumožňuje zadat víc jak jeden znak
oracle> SELECT CONCAT(CONCAT('*',TRIM(' x' from ' xAx ')),'*') FROM dual;
SELECT CONCAT(CONCAT('*',TRIM(' x' from ' xAx ')),'*') FROM dual
                                *
ERROR at line 1:
ORA-30001: trim set should have only one character
oracle> SELECT CONCAT(CONCAT('*',TRIM(' ' from ' xAx ')),'*') FROM dual;

CONCA
-----
*xAx*
```

Co k tomu říct závěrem? Snad jen, že když používátě nějakou novou funkci, nebo „starou" známou funkci (nebo operátor) ale v jiném DBMS, přečtěte si k ní pozorně dokumentaci!

A ještě jedna blbůstka: MySQL vyžaduje, aby mezi jménem funkce a úvodní závorkou před argumenty nebyla mezera. Jinak dostanete na první pohled nepravdivé chybové hlášení:

```
mysql> select max (plat) from zamestnanci;
ERROR 1630 (42000): FUNCTION rimmer1.max does not exist.
Check the 'Function Name Parsing and Resolution' section in the Reference Manual
```

Podmíněné výrazy

Tato kapitola měla být původně součástí kapitoly o funkcích. Ale protože by byla už neúměrně dlouhá, vytvořil jsem pro podmíněné výrazy vlastní kapitolu. Naučíte se tu další důležité funkce a direktivu CASE.

Podmíněné výrazy

Podmíněný výraz (conditional expression) by se dal zjednodušeně popsat jako *„Když něco, tak udělej toto, když něco jiného, tak udělej tamto, když ani to ne, udělej tohleto"*.

Podmíněné výrazy reprezenuje především direktiva **CASE**.

Kromě direkrvy CASE se v PostgreSQL ještě k podmíněným výrazům počítají některé funkce – COALESCE, NULLIF, GREATEST, a LEAST.

CASE

Direktiva CASE vypadá takto:

```
CASE WHEN condition THEN vyraz
[WHEN ...]
[ELSE vyraz]
END
```

Obšlehnuto z dokumentace PostgreSQL

Popis directivy CASE v dokumentaci SQLite

Direktiva vždy začíná klíčovým slovem CASE, vždy končí klíčovým slovem END a vždy má alespoň jednu WHEN … THEN … část.

Podmínka condition se vyhodnotí jako booleovský výraz (pravda nebo nepravda). Pokud je pravda, výsledkem celého CASE je vyraz.

Pokud je nepravda, zkusí se to v další WHEN … THEN … části.

Když není splněna žádná condition, výsledkem je vyraz za ELSE.

Když není ELSE vyraz definováno, tak už zbývá jako výsledek jenom NULL.

Možná to zní složitě, ale z příkladu se to dá pochopit snadno:

```
rimmer1=> SELECT CASE
WHEN 1 < 0 THEN 'jedna je mensi jak 0'
WHEN 1 > 0 THEN 'jedna je vetsi jak 0'
ELSE 'jedna se rovna nula'
```

```
END
FROM dual;
case
----------------------
jedna je vetsi jak 0
(1 řádka)
```

Nezapomeňte na ukončení CASE klíčovým slovem END. Na to se rádo zapomíná.

Ještě jeden příklad:

```
rimmer1=> SELECT CASE
WHEN NULL = NULL THEN 'NULL se rovna NULL'
WHEN NULL IS NULL THEN 'NULL je NULL'
WHEN 1 = 1 THEN 'jedna je jedna'
END AS "Co je co"
FROM dual;
      Co je co
--------------
NULL je NULL
(1 řádka)
```

*Jedna se sice rovná jedné, ale výsledkem CASE je **první** WHEN … THEN …, kde je podmínka (condition) pravda.*

Pokud porovnáváte jednu hodnotu s nějakými dalšími hodnotami, dá se zápis CASE zjednodušit. Porovnáním myslím použití operátoru = ve všech podmínkách.

Pozor! Nelze, resp. nemá smysl, porovnávat cokoliv s NULL, protože NULL se ničemá nerovná, ani sama sobě (viz předchozí příklad).

Následující dva příkazy dělají totéž.

V druhém příkazu je za CASE hodnota, která se porovnává s hodnotami za WHEN (může to být sloupeček, nebo nějaký výraz, který se vyhodnotí a pak porovnává):

```
rimmer1=> SELECT nazev,
CASE
WHEN nazev = 'Sekretariat' THEN 3000
WHEN nazev = 'Pravni oddeleni' THEN 2000
ELSE 0
END AS "prémie"
FROM oddeleni;
     nazev      | prémie
----------------+--------
 Sekretariat    |   3000
Pravni oddeleni |   2000
Pravni oddeleni |   2000
(3 řádky)

rimmer1=> SELECT nazev,
CASE nazev
WHEN 'Sekretariat' THEN 3000
WHEN 'Pravni oddeleni' THEN 2000
ELSE 0
END AS "prémie"
FROM oddeleni;
     nazev      | prémie
----------------+--------
 Sekretariat    |   3000
Pravni oddeleni |   2000
Pravni oddeleni |   2000
(3 řádky)
```

Výhoda druhého zápisu je čistě v tom, že vám ušetří trochu psaní.

CASE můžete používat v SELECTu, v podmínce WHERE, v ORDER BY – prostě všude tam, kde se dají používat aritmetické výrazy, funkce nebo sloupečky.

```
rimmer1=> SELECT jmeno, prijmeni, plat,
CASE plat/10000 WHEN 1 THEN 1 ELSE 0 END
FROM zamestnanci
ORDER BY CASE plat/10000 WHEN 1 THEN 1 ELSE 0 END DESC, plat ASC;
 jmeno  | prijmeni | plat  | case
--------+----------+-------+------
 Jana   | Pavova   | 10000 |    1
 Lenka  | Pavova   | 10000 |    1
 Martin | Luter    | 12000 |    1
 Jana   | Mala     | 12000 |    1
 Leopold| King     | 13000 |    1
 Lenka  | Pavova   | 15000 |    1
 Tom    | Jerry    | 15000 |    1
 Vasek  | Trn      | 16000 |    1
 Stary  | Osel     |  9000 |    0
 Tomas  | Mann     | 22000 |    0
 Pan    | Novy     |       |    0
(11 řádek)
```

plat/10000 je celočíselné dělení, tj. výsledek je beze zbytku (bez desetinného čísla).
Výsledek je seřazen podle platu s tím, že ti co mají plat mezi 10000 a 19999 jsou mezi prvními.

COALESCE

COALESCE je funkce, která vrátí svůj první argument, který není NULL. Když jsou NULL všechny argumenty, pak vrátí NULL. Všechny argumenty musí být stejného typu (není možné, aby ve výsledné tabulce v jednom sloupci byli někde čísla a někde text. Hodnoty sloupce v tabulce musí být vždy jednoho datového typu).

```
rimmer1=> SELECT jmeno, prijmeni, plat, COALESCE(plat,'-')
FROM zamestnanci WHERE CASE plat/10000 WHEN  1 THEN 1 ELSE 0 END = 0;
ERROR:  invalid input syntax for integer: "-"
ŘÁDKA 2: COALESCE(plat,'-')
```

Sloupec plat je typu integer, proto očekává COALESCE že další argumenty už taky budou integer.

Naštěstí se dá sloupec vždycky přetypovat na text:

```
rimmer1=> SELECT jmeno, prijmeni, plat, COALESCE(plat::text, NULL, '-')
FROM zamestnanci WHERE CASE plat/10000 WHEN  1 THEN 1 ELSE 0 END = 0;
 jmeno | prijmeni | plat  | coalesce
-------+----------+-------+----------
 Tomas | Mann     | 22000 | 22000
 Stary | Osel     |  9000 | 9000
 Pan   | Novy     |       | -
(3 řádky)
```

NULL jsem přidal do příkladu jen proto, abyste viděli, že COALESCE může mít víc než jenom 2 argumenty. Podmínka WHERE by se dala přepsat trochu čitelnějším způsobem takto: WHERE plat NOT BETWEEN 10000 and 19999 OR plat IS NULL;.

To co dělá COALESCE by šlo napsat i pomocí CASE (vyzkoušejte za domácí úkol), COALESCE je ale stručnější a tak i čitelnější.

NULLIF

Funkce NULLIF vrací NULL, když se její první argument rovná jejímu druhému argumentu, jinak vrátí první argument. Může se použít jako inverzní funkce k COALESCE. Taktéž platí, že se její funkce dá přepsat pomocí CASE, jen by to bylo o dost ukecanější.

```
rimmer1=> SELECT jmeno, prijmeni, plat,
COALESCE(plat::text, '-'),
NULLIF(COALESCE(plat::text, '-'), '-')::integer
FROM zamestnanci
WHERE plat NOT BETWEEN 10000 and 19999 OR plat IS NULL;
 jmeno | prijmeni | plat  | coalesce | nullif
-------+----------+-------+----------+--------
 Tomas | Mann     | 22000 | 22000    |  22000
 Stary | Osel     |  9000 | 9000     |   9000
 Pan   | Novy     |       | -        |
(3 řádky)
```

Výsledek NULLIF jsem schválně přetypoval na integer, aby se sloupeček zarovnal do prava, jak je u čísel zvykem.

GREATEST a LEAST

Funkce GREATEST a LEAST vrací největší, resp. nejmenší hodnotu ze svých argumentů. Hodnoty NULLignoruje, takže když se mezi argumenty objeví, nebude výsledek NULL (ač by k tomu logika významu NULLsváděla).

```
rimmer1=> SELECT GREATEST(3, 5,-100, NULL, 50, 3.7) FROM dual;
 greatest
----------
       50
(1 řádka)
```

MySQL/MariaDB

MySQL nedělí dvě celá čísla celočíselně, ale výsledkem je číslo s desetinnou částí. Proto se musí z výsledku dělení odstanit desetinná část pomocí funkce FLOOR. Jinak příklad s CASE funguje jako v Postgresu.

```
SELECT jmeno, prijmeni, plat,
CASE FLOOR(plat/10000) WHEN 1 THEN 1 ELSE 0 END
FROM zamestnanci
ORDER BY CASE FLOOR(plat/10000) WHEN 1 THEN 1 ELSE 0 END DESC, plat ASC;
```

MySQL automaticky převede datový typ na text, když je potřeba, takže následující příkaz funguje.

```
mysql> SELECT jmeno, prijmeni, plat, COALESCE(plat,'-')
FROM zamestnanci WHERE CASE FLOOR(plat/10000) WHEN  1 THEN 1 ELSE 0 END = 0;
+-------+----------+-------+--------------------+
| jmeno | prijmeni | plat  | COALESCE(plat,'-') |
+-------+----------+-------+--------------------+
| Tomas | Mann     | 22000 | 22000              |
| Stary | Osel     |  9000 | 9000               |
| Pan   | Novy     |  NULL | -                  |
+-------+----------+-------+--------------------+
3 rows in set (0.00 sec)
```

Napoak MySQL neumí přetypovávání datových typů pomocí dvou dvouteček :: (na přetypování používá něco jiného). Takže musíte uvádět příkazy bez nich:

```
mysql> SELECT jmeno, prijmeni, plat,
COALESCE(plat, '-') AS coalesce,
NULLIF(COALESCE(plat, '-'), '-') AS nullif
FROM zamestnanci WHERE plat NOT BETWEEN 10000 and 19999 OR plat IS NULL;
+-------+----------+-------+----------+--------+
| jmeno | prijmeni | plat  | coalesce | nullif |
+-------+----------+-------+----------+--------+
| Tomas | Mann     | 22000 | 22000    | 22000  |
| Stary | Osel     |  9000 | 9000     | 9000   |
| Pan   | Novy     |  NULL | -        | NULL   |
```

```
+-------+----------+-------+----------+--------+
3 rows in set (0.00 sec)
```

MySQL neignoruje NULL hodnoty ve funkcích GREATEST či LEAST:

```
SELECT GREATEST(3, 5,-100, NULL, 50, 3.7) FROM dual;
+-----------------------------------+
| GREATEST(3, 5,-100, NULL, 50, 3.7) |
+-----------------------------------+
|                              NULL |
+-----------------------------------+
1 row in set (0.00 sec)
```

IF

MySQL má navíc funkci IF, která vypadá takto:

IF(condition, result1, result2)

Funkce IF vrací result1, pokud se condition vyhodnotí jako pravda, jinak result2.

```
mysql> SELECT IF(1 < 0, 'jedna je mensi jak 0', 'jedna neni mensi jak 0') AS result FROM dual;
+------------------------+
| result                 |
+------------------------+
| jedna neni mensi jak 0 |
+------------------------+
1 row in set (0.00 sec)
```

IFNULL

Funkce IFNULL (neplést s NULLIF) dělá něco podobného jako COALESCE. Má jen dva argumenty. Pokud je první NULL, vrací druhý argument, jinak první.

```
mysql> SELECT jmeno, prijmeni, plat, IFNULL(plat,'-') AS ifnull
FROM zamestnanci
WHERE CASE FLOOR(plat/10000) WHEN  1 THEN 1 ELSE 0 END = 0;
+-------+----------+-------+--------+
| jmeno | prijmeni | plat  | ifnull |
+-------+----------+-------+--------+
| Tomas | Mann     | 22000 | 22000  |
| Stary | Osel     |  9000 | 9000   |
| Pan   | Novy     |  NULL | -      |
+-------+----------+-------+--------+
3 rows in set (0.00 sec)
```

SQLite

SQLite, stejně jako MySQL, neumí přetypovávat pomocí dvou dvouteček ::, takže všechny příklady se musí psát bez tohoto přetypování (a fungují bez něj).

SQLite dělí celá čísla celočíselně, takže jako v PostgreSQL se nemusí ořezávat pomocí funkce FLOOR(kterou SQLite stejně nemá :-).

SQLite nezná funkce GREATEST ani LEAST, ani podmínku IF, zato umí IFNULL (i NULLIF).

Vše ostatní funguje dle očekávání.

Oracle

Oracle dělí celá čísla neceločíselně, takže se musí desetinná část oddělit pomocí funkce FLOOR, stejně jako v MySQL.

Oracle ve funkci COALESCE automaticky nepřetypuje hodnoty arguemtnů na text, když je potřeba, ani neumí použít ::, takže se musí použít funkce TO_CHAR:

```
oracle> SELECT jmeno, prijmeni, plat, COALESCE(TO_CHAR(plat),  '-')
FROM zamestnanci
WHERE CASE FLOOR(plat/10000) WHEN  1 THEN 1 ELSE 0 END = 0;

JMENO      PRIJMENI      PLAT      COALESCE(TO_CHAR(PLAT),'-')
Stary      Osel          9000      9000
Tomas      Mann          22000     22000
Pan        Novy          -         -
```

Totéž platí pro další funkce:

```
oracle> SELECT jmeno, prijmeni, plat,
COALESCE(TO_CHAR(plat), '-') AS coalesce,
TO_NUMBER(NULLIF(COALESCE(TO_CHAR(plat), '-'), '-')) AS nullif
FROM zamestnanci
WHERE plat NOT BETWEEN 10000 and 19999 OR plat IS NULL;

JMENO      PRIJMENI      PLAT      COALESCE      NULLIF
Stary      Osel          9000      9000          9000
Tomas      Mann          22000     22000         22000
Pan        Novy          -         -             -
```

Oracle neignoruje NULL ve funkcích GREATEST a LEAST.

```
oracle> SELECT GREATEST(3, 5,-100, NULL, 50, 3.7) AS greatest FROM dual;
GREATEST
-
```

DECODE, NVL, NVL2

Oracle neumí jako MySQL funkce IF ani IFNULL. Zato má pár vlastních: DECODE, NVL (totéž co IFNULL) aNVL2.

Z příkladů z dokumentace je jasné co dělají a jak se používají, tak vám jejich studium nechám za domácí úkol.

A mimochodem, jako vždy, všechny tyto funkce lze nahradit pomocí CASE.

Telefonní databáze

V této kapitole vás provedu návrhem trochu „složitější" databáze. Pokusím se ukázat, nad čím vším byste se během návrhu databáze měli zamyslet.
Navrženou databázi pak budu využívat i v dalších lekcích.
Půjde hodně o povídací kapitolu a jedinou novou věcí z SQL, kterou se naučíte, bude, **jak vytvořit unikátní či primární klíč z více jak jednoho sloupce**.

Všechny příkazy pro vytvoření databáze jsem pro vás uložil do souboru psql10.sql.

Návrh telefonní databáze

Než začnete nějakou databázi vytvářet, je třeba se zamyslet nad tím, co všechno od ní budete požadovat a k čemu má sloužit. To vám může v budoucnu ušetřit spoustu práce.
Navrhováním tabulek vlastně **modelujete** reálný svět. Model je zdejdnodušený pohled na skutečné objekty z reálného světa. Model by měl obsahovat pouze to, co vás o reálném světě skutečně zajímá. Nemusíte ukládat informaci o tom, jakou barvu očí má váš zákazník – ledaže byste navrhovali databázi pro očního chirurga.
V této kapitole se pokusím navrhnout databázi kontaktů s telefonními čísli pro rize soukromé, domácí použití. Vědět, pro jaký účel je databáze navrhována, je klíčové. Z toho se pak odvozuje, jaké údaje budou v „modelu" potřeba a jaké jsou zbytečné. Jak by tedy měla vypadat taková telefonní databáze? Nejdřív vás možná napadne, že si uděláte jednu tabulku, kde budete mít jméno, telefonní číslo a možná nějakou poznámku ke kontaktu.
Co byste ještě mohli potřebovat dál? Například by vás mohlo zajímat, jakého operátora dané číslo využívá. Taky vás může zajímat, kolik vás bude stát hovor za minutu na dané číslo. Pak si možná budete chtít do databáze uložit nějaké další informace o člověku, kterému číslo patří, abyste si za čas nelámali hlavu nad tím, co že je to za člověka.
Pro zjednodušení budu předpokládat, že telefonní čísla, která budete ukládat, budou jen z české republiky, proto se nebudu zabývat předvolbami typu "+420". Taky mě nebude zajímat kolik stojí poslání SMS, ani to, že někteří operátoři účtují po sekundách a někteří po minutách.

Diagram tabulek

Z výše uvedených předpokladů budu vycházet při návrhu databáze. Vytvořím celkem 5 tabulek (kdo by se toho nadál :-), které můžete vidět na obrázku.
Na začátku skriptu se sluší tabulky smazat (pokud existují). A to v opačném pořadí, než se budou vytvářet (kvůli referencím mezi tabulkami).

```
DROP TABLE IF EXISTS telefon;
DROP TABLE IF EXISTS kontakt;
DROP TABLE IF EXISTS tarif;
DROP TABLE IF EXISTS predvolba;
DROP TABLE IF EXISTS operator;
```

Tabulka kontakt

Základem telefonní databáze bude jistě tabulka telefonů. Je ale dobré trochu předvídat potřeby budoucnosti. Je možné, že si někdy v budoucnu budete chtít uchovávat další informace, které se budou týkat stejných lidí, které budete mít uložené v tabulce „telefonů". Pak budete potřebovat takovéto informace propojit. Proto bude lepší, když budou informace o lidech oddělené (v jiné tabulce) než informace o telefonech.
Výhodou rozdělení kontaktů a telefonů do dvou tabulek je i to, že když budete zadávat ke kontaktu více telefonů, nemusíte všechny inormace o něm duplikovat v každém řádku. Z tabulky telefonů se prostě odkážete do tabulky kontaktů (všimněte si šipky z tabulky telefonů do tabulky kontaktů na obrázku výše).
Tabulku jsem pojmenoval kontakt, protože člověk je příliš obecný název. Proč je název v jednotém čísle a ne v množném? V zásadě je to jedno, jestli budete používat jednotné nebo množné číslo. Důležité je, abyste byli v tomto konzistentní a používali pro všechny tabulky buď jen jednotné, nebo jen množné číslo.
Tabulka kontakt obsahuje nejvíce sloupečků, ale její návrh je velmi jednoduchý. Obashuje základní informace o kontaktu (jméno, příjmení, priorita, pohlaví, adrsesa kontaktu, věk a poznámka). Kontakt nemusí být jen váš známý, může to být nějaký obchodní kontakt, proto obsahuje dva sloupečky, kam si můžete zapsat z jaké firmy člověka znáte a nějakou poznámku o dané firmě.

- **id** je primární klíč, na který se bude odkazovat tabulka telefon.
- **jmeno**, **prijmeni** a **poznamka** asi nemusím vysvětlovat. Rozhodl jsem se je udělat povinné (na obrázku označeno hvězdičkou *). Myslím, že není potřeba rozlišovat mezi tím, kdy třeba příjmení neznám, nebo nemám žádnou poznámku a tím, kdy je poznámka nebo příjmení prázdné (obsahuje prázdný řetězec). Sice tak přijdu o informaci jestli osoba příjmení nemá (třeba Madona), nebo ho neznám, ale to mě u mé soukromé databáze netrápí a ušetřím si starosti s hodnotou NULL. Stejně tak nepotřebuju rozlyšovat u poznámky, jestli je prázdná, nebo jí neznám (co by to vůbec znamenalo?).
- **pohlavi**, **adresa** a **vek** jsou asi taky jasné. Nechal jsem je nepovinné, protože je jednak nemusím vždycky znát, nebo mě nemusí zajímat. (Pravda, sloupec adresa by také mohl být povinný a nemusel bych rozlišovat mezi tím, kdy je adresa prázdná a kdy je NULL, stejně jako u příjmení nebo poznámky. A asi by to bylo i lepší – konzistentnější rozhodnutí :-) V zájmu příkladů pro výuku se mi ale hodí, aby mohla být adresa NULL.)

- **priorita** určuje nakolik mám daného člověka rád (3 = nejlepší přítel, 0 = defaulně bez priority, -3 = nepřítel na život a na smrt atd.). Pokud sem se pro žádnou prioritu nerozhodl, je 0, takže hodnota NULL by znamenala totéž. Mít dvě hodnoty pro totéž nevede nikdy k ničemu dobrému, proto je NULL zakázané (priorita je povinná).
- Sloupce **firma_nazev** a **firma_poznamka** se hodí pro kontakty, které znám z nějakého firmy, obchodu, úřadu atp. Pochopitelně jsou nepovinné. U firmy chci vědět, jestli je člověk z nějaké firmy nebo ne, proto může být firma_nazev NULL. A když už může být NULL název firmy, měla by být NULL i poznámka k firmě. Vypadalo by dost divně, kdyby byla firma NULL, ale poznámka ne.

Výsledné SQL pro vytvoření tabulky kontakt vypadá takto:

```
CREATE TABLE kontakt (
id SERIAL PRIMARY KEY NOT NULL,
jmeno VARCHAR(20) DEFAULT '' NOT NULL,
prijmeni VARCHAR(20) DEFAULT '' NOT NULL,
priorita INTEGER NOT NULL DEFAULT 0 CHECK (priorita >= -3 AND priorita <= 3),
pohlavi BOOLEAN DEFAULT NULL,
adresa VARCHAR(100) DEFAULT NULL,
vek INTEGER DEFAULT NULL CHECK (vek > 0),
poznamka TEXT DEFAULT '' NOT NULL,
firma_nazev VARCHAR(50) DEFAULT NULL,
firma_poznamka TEXT DEFAULT NULL
);
```

Všimněte si integritních omezení, která se starají o to, aby se nedostali do databáze omylem nějaké blbosti.

Proč jsou sloupce s datovým typem VARCHAR tak dlouhé, jak jsou? Jednoduše kvalifikovaný odhad :-). Někdo používá v takovém případě vždy VARCHAR(255) (což je max. velikost tohoto datového typu v MySQL). Zpomalení způsobené takto velkým datovým typem je, s dnešními počítači, většinou zanedbatelné, takže proč ne. Snad jen, že přijdete o „nápovědu" o tom, co asi bude sloupeček obsahovat za hodnoty.

Pokud vám kvalifikovaný odhad nestačí (u profiesionálních databází by neměl), není zase tak těžké si vygooglit, že nejdelší české příjmení má 17 znaků (Červenokostelecký) a nejdelší jméno z českého kalendáře má 11 znaků. Pravda, pořád hrozí, že bude mít někdo nějaké cizokrajné jméno či příjmení, na které bude 20 znaků málo.

Tabulka operator

Zatím by to mohlo vypadat, jako by nám stačila tabulky kontakt a telefon. Pojďme si to ale trošku zkomplikovat – teda zjednodušit.

Pro operátory jsem navrhl extra tabulku. Bude obsahovat jen umělý klíč (id) a jméno operátora.

Jméno by samo o sobě asi mohlo být primárním klíčem. Operátorů budou jednotky a né statisíce, takže s výkonem byste si starosti dělat nemuseli. Ale co když se nějaký operátor přejmenuje (někdo ho koupí)? Díky primárnímu klíči stačí upravit jen sloupeček jmeno v tabulce operátora a ne ve všech tabulkách, které se na operátora odkazují.

Nicméně asi nebude od věci udělat sloupeček jmeno unikátní. Jméno zůstává **přirozeným primárním klíčem**.

Důvod proč jsem nenechal operátora součástí tabulky telefon je asi zřejmý z diagramu výše. Bude se na něj odkazovat více tabulek. Mít jméno operátora v tabulce telefonů i tarifů, při přejmenování by se muselo jméno opravovat na více místech. A pak taky, kdyby ste se rozhodli že chcete ukládat další informace o operátorovi, kam byste je dali? Do tabulky telefonů nebo tarifů?

SQL pro vytvoření tabulky operator je velmi jednoduché:

```
CREATE TABLE operator (
id SERIAL PRIMARY KEY NOT NULL,
jmeno VARCHAR(20) NOT NULL UNIQUE
);
```

Tabulka predvolba

Tabulka předvoleb bude obsahovat předvolby a referenci do tabulky operátorů. Předvolby tentokrát nechám jako primární klíč. Předvolba se bude těžko měnit jako jméno společnosti. Navíc je krátká, má vždy max. 3 čísla, takže se vyplatí jí udělat typu CHAR, se kterým se pracuje poměrně rychle. Maximálně (teoreticky) můžete mít 1000 předvoleb, takže o nějakém zrychlování zavedením umělého klíče taky nemůže být řeč. Zavedením umělého klíče by se ani neušetřilo moc místa.

```
CREATE TABLE predvolba (
predvolba CHAR(3) NOT NULL PRIMARY KEY CHECK(LENGTH(predvolba) >= 1),
operator_id INTEGER REFERENCES operator(id) NOT NULL
);
```

Napadá vás, proč je předvolba typu CHAR a ne INTEGER? Správná odpověď je – ze sémantických důvodů – předvolba není číslo, nebudete s ní dělat žádné matematické operace (násobit jí, sčítat atd). Telefonní číslo je číslo asi jako rodné číslo nebo číslo občanky – vlastně to číslo není :-).

Tabulka tarif (unikátní klíče)

V této tabulce se bude udržovat informace o tarifu. Stejný tarif bude mít mnoho telefonních čísel, takže, aby se informace neduplikovali, budou ve vlastní tabulce.

Nejde o tabulku tarifů, které vám nabízí váš operátor. Jde o tarif, za který voláte na dané telefonní číslo. I v rámci jednoho operátora můžete volat za různé ceny (můžete mít zvýhodněné volání v rámci firmy, existují různá „drahá" telefonní čísla atp.). Všechny tarify jsou tarify vašeho operátora (se kterým voláte), ale vztahují se k nějakému operátorovi (do kterého voláte). Jeden tarif může znamenat různé ceny podle operátora, ne kterého voláte (jiná cena je pro vašeho operátora, jiná na pevnou linku, jiná na ostatní mobilní operátory). Název tarifu proto není unikátní, ale název tarifu + operátor už ano.

Je možné, že nebudete znát cenu tarifu, proto je cena nepovinná. Stojí za zvážení, jestli takový tarif bez ceny má cenu ukládat do databáze. Kdybych se rozhodl že ne, udělal bych cenu povinnou položkou.

Tarif bude mít umělý klíč id. Bude se na něj snadněji odkazovat, než na id operátora + jméno. Ale dvojice sloupců operator_id a jmeno by měla být unikátní. Omezení (constraint) UNIQUE nad více než jedním sloupcem se musí definovat na úrovni tabulky (a ne na úrovni sloupce), takže SQL pro vytvoření tabulky tarif bude vypadat takto:

```
CREATE TABLE tarif (
id SERIAL PRIMARY KEY NOT NULL,
operator_id INTEGER REFERENCES operator(id) NULL,
nazev VARCHAR(20) NOT NULL,
cena NUMERIC(3, 1) NULL CHECK(cena >= 0),
UNIQUE (operator_id, nazev)
);
```

Vytvořením omezení UNIQUE se vytvoří i index, který DBMS využívá pro hlídání unikátnosti. Tento index má své jméno, které mu vymyslí DBMS, nebo mu ho můžete explicitně určit:

```
rimmer1=> CREATE TABLE tarif (
id SERIAL PRIMARY KEY NOT NULL,
operator_id INTEGER REFERENCES operator(id) NULL,
nazev VARCHAR(20) NOT NULL,
cena NUMERIC(3, 1) NULL CHECK(cena >= 0),
CONSTRAINT operator_nazev_uix UNIQUE (operator_id, nazev)
);
NOTICE:  CREATE TABLE will create implicit sequence "tarif_id_seq" for serial column"tarif.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "tarif_pkey" for table"tarif"
NOTICE:  CREATE TABLE / UNIQUE will create implicit index "operator_nazev_uix" for table "tarif"
CREATE TABLE
```

Tabulka telefon (složený primární klíč)

Konečně se dostane na tabulku telefon. Tabulka obsahuje předvolbu a zbytek čísla (telefon), povinný odkaz na operátora, nepovinný odkaz na tarif (ne vždy ho budu znát, tak jej nevyžaduji), nepovinný odkaz na kontakt (můžu mít uložen telefon bez kontaktu – třeba na požárníky nebo na erotickou linku :-). Přidal jsem ještě poznámku, která se může hodit pro telefony bez kontaktu, nebo pro kontakty s více telefony (je to telefon domů, do práce, nebo jaký teda?).

Poznámka může mít až 100 znaků, což by mělo na bližší identifikaci telefonu stačit (kvalifikovaný odhad :-). Pro takovou domácí databázi by asi nevadilo použít i typ TEXT, ikdyž je s ním práce o malinko pomalejší (zas tak moc telefonů mít doma určitě nebudete). Jiná záležitost by to ale byla při navrhování databáze pro velkou (nadnárodní) firmu. Tam už by stálo za zvážení, zda je rychlost důležitá, zda omezit max. délku poznámky (která většinou bude stejně prázdná). Takové rozhodnutí by už ste ale neměli dělat sami, ale po poradě s klientem (opravdu mu bude stačit 100 znaků, nebo raději obětuje trochu toho místa a rychlosti?)

Při definici tabulky telefon bude jako primární klíč sloužit dvojice sloupců predvolba a telefon. Na tabulku se odnikud nebudu odkazovat, takže umělý klíč (id) by byl zbytečný. Stejně jako když jsem vytvářel unikátního klíče nad více sloupci v tabulkce tarif, i pro primární klíč nad více sloupci platí, že se musí definovat na úrovní tabulky (nemůžete ho definovat na úrovni sloupce, když se to týká dvou sloupců).

```
CREATE TABLE telefon (
predvolba CHAR(3) NOT NULL REFERENCES predvolba(predvolba),
telefon VARCHAR(6) NOT NULL,
operator_id INTEGER REFERENCES operator(id) NOT NULL,
tarif_id INTEGER REFERENCES tarif(id) DEFAULT NULL,
kontakt_id INTEGER REFERENCES kontakt(id) DEFAULT NULL,
poznamka VARCHAR(100) NOT NULL DEFAULT '',
PRIMARY KEY(predvolba, telefon)
);
```

Primární klíč musí být vždy unikátní. Takže při vytvoření primárního klíče se vytvoří i index, který DBMS využívá pro kontrolu unikátnosti. Indexu vytvoří DBMS jméno automaticky, nebo ho můžete určit explicitně:

```
rimmer1=> CREATE TABLE telefon (
predvolba CHAR(3) NOT NULL REFERENCES predvolba(predvolba),
telefon VARCHAR(6) NOT NULL,
operator_id INTEGER REFERENCES operator(id) NOT NULL,
tarif_id INTEGER REFERENCES tarif(id) DEFAULT NULL,
kontakt_id INTEGER REFERENCES kontakt(id) DEFAULT NULL,
poznamka VARCHAR(100) NOT NULL DEFAULT '',
CONSTRAINT telefon_pk PRIMARY KEY(predvolba, telefon)
);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "telefon_pk" for table"telefon"
CREATE TABLE
```

V našem případě je unikátní dvojice predvolba + telefon. To znamená, že se předvolba i telefon mohou v tabulce libovolněkrát opakovat, ale dvojice predvolba + telefon musí být unikátní. Což je přesně to, co se od tabulky telefonů očekává.

Využití databáze

Vyplnění tabulek hodnotami

Než budu databázi používat v příkladech, musím do ní vložit nějaké hodnoty. Začnu od nejjednoduššího - tabulky operátorů. Předvoleb je dneska už tolik, že se vám snadno stane, že narazíte na předvolbu, u které nebudete vědět, ke kterému operátoru patří. A protože je u předvolby povinný odkaz na operátora, vytvořím si tzv. *Neznámého operátora*, který mi dovolí uložit předvolbu do databáze do doby než zjistím, kterému operátoru předvolba vlastně patří.

Jasně, taky jsem nemusel dělat v tabulce předvoleb odkaz na operátora povinný. Ale zase tak často novou neznámou předvolbu ukládat nebudu. A když už jí uložím, tak jen dočasně, než zjistím, komu patří (to je kvalifikovaný odhad :-). Povinnost mít vyplněného operátora (to, že nesmí být NULL) mi ale na druhou stranu velmi zjednoduší SELECTy! (Nebudu muset dávat speciální pozor na řádky s NULL hodnotou u operator_id.)

```
INSERT INTO operator(jmeno) VALUES ('?'); -- id=1
INSERT INTO operator(jmeno) VALUES ('Pevná linka'); -- id=2
INSERT INTO operator(jmeno) VALUES ('O2'); -- id=3
INSERT INTO operator(jmeno) VALUES ('T-Mobil'); -- id=4
INSERT INTO operator(jmeno) VALUES ('Vodafone'); -- id=5
INSERT INTO operator(jmeno) VALUES ('U:fon'); -- id=6
```

Teď se můžete vyplnit tabulka předvoleb (která se odkazuje na operátory). Předvoleb je hodně, tak nebudu vypisovat všechny INSERTy, najdete je ve skriptu psql10.sql.

```
INSERT INTO predvolba VALUES (312,2);
INSERT INTO predvolba VALUES (318,2);
INSERT INTO predvolba VALUES (601,3);
INSERT INTO predvolba VALUES (602,3);
...
```

```
INSERT INTO predvolba VALUES (799,1);
INSERT INTO predvolba VALUES (908,1);
```

Ještě něco do tabulky kontaktů:

```
INSERT INTO kontakt(jmeno,prijmeni,priorita,adresa) VALUES ('Pavel','Drevokocur',0,'Na nožičkách 3, Praha 4, 13000');
INSERT INTO kontakt(jmeno,prijmeni,priorita,adresa) VALUES ('Petr','Bílek', 0,'Na balkáně 70, Praha 3, 13004');
INSERT INTO kontakt(jmeno,prijmeni,priorita) VALUES ('Tomas','Drevokocur', 1);
INSERT INTO kontakt(jmeno,prijmeni,priorita) VALUES ('Lukas','Drevokocur',-3);
INSERT INTO kontakt(jmeno,prijmeni,priorita) VALUES ('Milos','Drevokocur', 3);
INSERT INTO kontakt(jmeno,prijmeni,priorita,pohlavi) VALUES ('Pavla','Drevokocurova', 3, false);
INSERT INTO kontakt(jmeno,prijmeni,priorita,pohlavi) VALUES ('Jana', 'Drevokocurova', 0, false);
INSERT INTO kontakt(jmeno,prijmeni,priorita,pohlavi,vek,firma_nazev,firma_poznamka) VALUEs ('Tomáš','Baťa',3,true,70,'Baťovy závody','Vyrábí boty');
INSERT INTO kontakt(jmeno,prijmeni,priorita,pohlavi,vek,firma_nazev,firma_poznamka) VALUEs ('Jan','Baťa',2,true,40,'Baťovy závody','Vyrábí boty');
```

Ještě vložím pár příkladů tarifů:

```
INSERT INTO tarif(operator_id,nazev,cena) VALUES (1, 'Za 30', NULL); -- id 1
INSERT INTO tarif(operator_id,nazev,cena) VALUES (1, 'Za 60', NULL); -- id 2
INSERT INTO tarif(operator_id,nazev,cena) VALUES (1, 'Za 90', NULL); -- id 3
INSERT INTO tarif(operator_id,nazev,cena) VALUES (2, 'Pevna', 10); -- id 4
INSERT INTO tarif(operator_id,nazev,cena) VALUES (3, 'O2 zaklad',9); -- id 5
INSERT INTO tarif(operator_id,nazev,cena) VALUES (3, 'O2 kamarad',6); -- id 6
INSERT INTO tarif(operator_id,nazev,cena) VALUES (4, 'zaklad', 8); -- id 7
INSERT INTO tarif(operator_id,nazev,cena) VALUES (4, 'kamarad', 5); -- id 8
INSERT INTO tarif(operator_id,nazev,cena) VALUES (5, 'Vodafone zaklad', 3.5); -- id 9
INSERT INTO tarif(operator_id,nazev,cena) VALUES (5, 'Vodafone kamarad', 0); -- id 10
INSERT INTO tarif(operator_id,nazev,cena) VALUES (6, 'zaklad', 7); -- id 11
INSERT INTO tarif(operator_id,nazev,cena) VALUES (6, 'kamarad', 5); -- id 12
```

A teprve teď, po té dlouhé a strastiplné cestě, se mohou začít ukládat telefonní čísla.

```
INSERT INTO telefon (telefon,predvolba,operator_id, tarif_id, kontakt_id) VALUES ('555555', '737',4, 7, 1);
INSERT INTO telefon (telefon,predvolba,operator_id, tarif_id, kontakt_id) VALUES ('555555', '608',5, 10,  2);
INSERT INTO telefon (telefon,predvolba,operator_id, tarif_id, kontakt_id) VALUES ('555555', '776',5, 9,  4);
INSERT INTO telefon (telefon,predvolba,operator_id, tarif_id, kontakt_id, poznamka) VALUES ('555555', '312', 2, 4, 4, 'Telefon domu');
INSERT INTO telefon (telefon,predvolba,operator_id, tarif_id, kontakt_id) VALUES ('555555', '604',4, 7, 6);
INSERT INTO telefon (telefon,predvolba,operator_id, tarif_id, kontakt_id, poznamka) VALUES ('300300','908',1,1, NULL, 'Půjčka za 30/min');
```

Výběr hodnot z tabulek

Všechny tabulky jsou vytvořeny, data vložena, takže si můžeme začít hrát :-).

```
rimmer1=> SELECT prijmeni, jmeno, CONCAT(predvolba,'/', telefon) AS telefon
FROM telefon, kontakt WHERE kontakt_id = kontakt.id;
   prijmeni    | jmeno |  telefon
---------------+-------+------------
 Drevokocur    | Pavel | 737/555555
 Bílek         | Petr  | 608/555555
 Drevokocur    | Lukas | 776/555555
 Drevokocur    | Lukas | 312/555555
 Drevokocurova | Pavla | 604/555555
(5 rows)
```

Ale ejhle, jen 5 řádek? V databázi je přece 6 telefonních čísel a 9 kontaktů! Jenomže ne každý kontakt má telefon a ne každý telefon má kontakt. O tom, jak do výsledku předešlého dotazu zahrnout i kontakty bez telefonu, nebo telefony bez kontaktu, bude následující kapitola :-).

Jak již bylo zmíněno v kapitole SELECT I a VIEW, v případě že máte ve dvou tabulkách sloupce stejného jména, použijete pro jejich rozlišení tzv. tečkovou notaci. Například sloupec poznamka je v tabulce telefon i kontakt.

```
rimmer1=> SELECT prijmeni, jmeno, predvolba, telefon, telefon.poznamka
FROM kontakt, telefon
WHERE kontakt_id = kontakt.id AND prijmeni = 'Drevokocur' AND jmeno = 'Lukas';
   prijmeni    | jmeno | predvolba | telefon |   poznamka
---------------+-------+-----------+---------+--------------
 Drevokocur    | Lukas | 776       | 555555  |
 Drevokocur    | Lukas | 312       | 555555  | Telefon domu
(2 rows)
```

To by myslím jako ukázka stačilo. Za domácí úkol si vyzkoušejte další SELECTy, přidejte například do výsledku informace o ceně volání (z tarifu), název operátora atd.

Vytvoření pohledu do databáze

Data máme rozdělené do mnoha tabulek. Aby se s nimi dalo dobře pracovat, tak je teď pro změnu zase spojíme. Nejzajímavější určitě bude spojení tabulek kontakt a telefon. Proto se bude hodit vytvořit **pohled**na spojení těchto dvou tabulek. Vytvořím jej pomocí příkazu CREATE VIEW. Pohled setřídím podle příjmení a vyberu do něj jen ty sloupce, které mě budou zajímat nejvíce. Pohled by mohl vypadat takto:

```
rimmer1=> SELECT prijmeni, jmeno, telefon, predvolba, telefon.poznamka AS "O telefonu",
priorita, pohlavi, vek, adresa, kontakt.poznamka
FROM kontakt, telefon
WHERE kontakt_id = id
ORDER BY prijmeni;
 prijmeni | jmeno | telefon | predvolba | O telefonu | priorita | pohlavi | vek |          adresa           | poznamka
----------+-------+---------+-----------+------------+----------+---------+-----+---------------------------+----------
 Bílek    | Petr  | 555555  | 608       |            |        0 |         |     | Na balkáně 70, Praha 3, 13004 |
```

```
Drevokocur    | Pavel | 555555 | 737      |            |       0 |       |    | Na nožičkách 3, Praha 4, 13000 |
Drevokocur    | Lukas | 555555 | 776      |            |      -3 |       |    |                                |
Drevokocur    | Lukas | 555555 | 312      | Telefon domu |      -3 |       |    |                                |
Drevokocurova | Pavla | 555555 | 604      |            |       3 | f     |    |                                |
(5 rows)
```

Teď to ještě chce zobrazit jméno operátora. Všimněte si, že operátor telefonu se dá zjistit dvěma cestami. Buď přímo z operator_id z tabulky telefon, nebo přes tabulku predvolba. To ukazuje, že nebyla databáze navržena úplně košer, protože k takovéto duplicitě dat by nemělo nikdy dojít. K tomu se ještě vrátím.

Spojení tabulky operator přímo s tabulkou telefon by bylo jednodušší, proto to udělám (ze cvičných důvodů) oklikou přes tabulku predvolba (později stejně operator_id z tabulky telefon vykopnu, z důvodů popsaných výše).

```
CREATE VIEW telefonni_seznam AS
SELECT prijmeni, kontakt.jmeno, telefon.predvolba, telefon,
operator.jmeno as "Operator", telefon.poznamka AS "O telefonu",
priorita, pohlavi, vek, adresa,
kontakt.poznamka
FROM kontakt, telefon, predvolba, operator
WHERE kontakt_id = kontakt.id AND predvolba.predvolba = telefon.predvolba AND predvolba.operator_id = operator.id
ORDER BY prijmeni, jmeno;
```

Teď už se může pracovat s telefonním seznamem jako s jednou tabulkou:

```
rimmer1=> SELECT prijmeni, jmeno, CONCAT(predvolba,'/',telefon) as telefon, "O telefonu", "Operator"
FROM telefonni_seznam ORDER BY "Operator" DESC;
  prijmeni    | jmeno |  telefon  |  O telefonu  |  Operator
--------------+-------+-----------+--------------+------------
 Bílek        | Petr  | 608/555555 |              | Vodafone
 Drevokocur   | Lukas | 776/555555 |              | Vodafone
 Drevokocur   | Pavel | 737/555555 |              | T-Mobil
 Drevokocurova | Pavla | 604/555555 |              | T-Mobil
 Drevokocur   | Lukas | 312/555555 | Telefon domu | Pevná linka
(5 rows)
```

Všimněte si, že název sloupečku Operator musí být v uvozovkách. Jinak se Postgres tváří, jako by pohled sloupeček "Operator" neměl (ikdyž víme, že má). Je to proto, že při vytváření pohledu byl název sloupečku definován v uvozovkách. Kdyby byl definován bez uvozovek, pak by se v SELECTu taky musel použít bez uvozovek. No není to na hlavu? (Je to na hlavu. V ostatních DBMS tento problém není, vyjma Oracle, který k tomu ale má důvod).

Za domácí úkol můžete přidat do pohledu ještě informace o tarifech.

Závěr

V této kapitole jsem připravil tabulky a data pro další kapitoly. Ukázal jsem, jak vytvořit unikátní index nebo primární klíč nad více jak jedním sloupečkem.

Návrh databáze není dokonalý. Obsahuje několik „nedokonalostí“, o kterých bude řeč v kapitole o **normalizaci databáze**. Už jsem se zmínil o tom, že není dobré, aby bylo možné zjistit operátora telefonního čísla dvěmi cestami. Napadají vás další chybičky v návrhu, které by vám mohli v budoucnu znepříjemnit používání databáze?

MySQL/MariaDB

Jediné novinky v této lekci byly vytvoření unikátního a primárního klíče na více sloupcích. To se v MySQL dělá stejně jako v PostgreSQL. Rozdíly oproti PostgreSQL v SQL příkazech z této lekce byli již popsány v předchozích lekcích.

V MySQL je také bug, který znemožňuje použití defaultní hodnoty s datovým typem TEXT. Existuje sice určité nastavení MySQL, které použití defaultní hodnoty povoluje, ale na to se nedá moc spoléhat. Proto jsem ve skryptu pro MySQL odstranil z tabulky kontakt defaultní hodnotu pro sloupeček poznamka a přidal do insertů do tabulky kontakt sloupec poznamka s hodnotou ''.

SQL skript pro MySQL najdete tady: psql10-mysql.sql.

Ještě připomenu, že je nutné definovat cizí klíč (foreign key) na úrovni tabulky (protože na úrovni sloupce nefunguje), nefunkčnost integritního omezení CHECK (ač jej lze definovat na konci definice sloupce), používání AUTO_INCREMENT místo sekvencí, potřeba definice typu tabulky (Engine), znakové sady (CHARSET) a COLLATE (způsob řazení znaků).

SQLite

V SQLite se unikátní a primární klíč na více sloupcích také vytváří stejně jako v PostgreSQL. Změny oproti PostgreSQL z této lekce byli již probrány v předchozích lekcích.

SQL skript pro SQLite najdete tady: psql10-sqlite.sql.

Zmíním jen používání AUTOINCREMENT místo sekvencí, neexistenci true a false (místo nich se používá 1 a 0). SQLite také nezná funkci CONCAT, takže se místo ní používá operátor ||:

```
SELECT prijmeni, jmeno, predvolba || '/' || telefon AS telefon
FROM telefon, kontakt WHERE kontakt_id = kontakt.id;
```

V klauzuli ORDER BY musí být název sloupce jednoznačně identifikován, takže u sloupce jmeno, který se nachází ve více tabulkách, musí být pomocí tečkové notace jasně řečeno, z jaké je tabulky. (Ostatní DBMS „uhodnou“, že máme na mysli sloupeček, který je v klauzuli SELECT, kde je sloupec jmeno jednoznačně identifikován. Což asi není úplně dobře, třídit se dá i podle sloupců které nejsou v klauzuli SELECT, tak proč si to DBMS takto domýšlí?)

```
CREATE VIEW telefonni_seznam AS
SELECT prijmeni, kontakt.jmeno, telefon.predvolba, telefon,
operator.jmeno as "Operator", telefon.poznamka AS "O telefonu",
priorita, pohlavi, vek, adresa,
kontakt.poznamka
FROM kontakt, telefon, predvolba, operator
WHERE kontakt_id = kontakt.id AND
predvolba.predvolba = telefon.predvolba AND predvolba.operator_id = operator.id
ORDER BY prijmeni, kontakt.jmeno;
```

A ještě jednou problém s funkcí CONCAT:

```
SELECT prijmeni, jmeno, predvolba || '/' || telefon as telefon,
"O telefonu", operator
FROM telefonni_seznam ORDER BY "Operator" DESC;
```

Oracle

V Oracle se unikátní a primární klíč na více sloupcích také vytváří stejně jako v PostgreSQL. Změny oproti PostgreSQL z této lekce byli již probrány v předchozích lekcích.

SQL skript pro Oracle najdete tady: psql10oracle.sql.

Z rozdílů zmíním neexistenci IF EXISTS, nemožnost použít hodnotu sekvence jako DEFAULT (takže se musí explicitně psát v klauzuli INSERT), neexistenci typu BOOLEAN (ani hodnot true a false), datové typy VARCHAR2 a CLOB (namísto VARCHAR a TEXT), či nemožnost vložení prázdného řetězce '' (konvertuje se automaticky na hodnotu NULL).

V Oracle je problém s funkcí CONCAT, která může mít jen dva argumenty. Buď můžete použít operátor ||, nebo vnořený CONCAT.

```
SELECT prijmeni, jmeno, CONCAT(CONCAT(predvolba,'/'), telefon) AS telefon
FROM telefon, kontakt WHERE kontakt_id = kontakt.id;

SELECT prijmeni, jmeno, predvolba || '/' || telefon as telefon,
"O telefonu", "Operator" FROM telefonni_seznam ORDER BY "Operator" DESC;
```

V Oracle se převádějí jména sloupečků automaticky na velká písmena, pokud nepoužijete uvozovky, a navíc je Oracle case-sensitive (rozlišuje velikosti písmen). Proto, když se při vytváření pohledu telefonni_seznamdefinovalo jméno sloupečku "Operator", musíte tento identifikátor používat vždy s uvozovkami (jinak se převede na OPERATOR a to už není totéž co Operator).

Takto se Oracle chová nejen u pohledů, ale i u tabulek, takže, když nic jiného, jedná se alespoň o konzistentní chování.

Slučování tabulek

Slučování/spojování tabulek jsem ukazoval už v kapitole SELECT I a VIEW v odstavciSpojování tabulek. V této kapitole proberu další možnosti spojování dat z více tabulek. Především pak konstrukce s **JOIN**, které jsou podporovány až v novějších verzích PostgreSQL.(Tahle věta už je trochu pasé.)

Používání aliasů

Aliasy jsem popisoval již v kapitole SELECT I a VIEW. Pomocí klíčového slova **AS** můžete vytvořit nové jméno pro sloupec v SELECTu.

Aliasy se dají použít i pro přejmenování tabulky. Hodí se to na zkrácení zápisu při používání tečkové notace, nebo při vytváření korelovaných dotazů (viz následující kapitola). Aliasem zde myslím jiný název pro tabulku. Alias pro tabulku se vytvoří přidáním nového názvu za název tabulky a AS (které je ovšem nepovinné) v klauzuli FROM:

```
... FROM tabulka [AS] alias_tabulky1, tabulka2 [AS] alias_tabulky2 ...
rimmer1=> SELECT prijmeni, jmeno, telefon, t.poznamka AS "O telefonu",
k.poznamka AS "O kontaktu"
FROM telefon AS t, kontakt k
WHERE kontakt_id = k.id;
  prijmeni   | jmeno | telefon |  O telefonu  | O kontaktu
-------------+-------+---------+--------------+------------
 Drevokocur  | Pavel | 555555  |              |
 Bílek       | Petr  | 555555  |              |
 Drevokocur  | Lukas | 555555  |              |
 Drevokocur  | Lukas | 555555  | Telefon domu |
 Drevokocurova | Pavla | 555555  |              |
(5 řádek)
```

Použití AS je nepovinné, proto ilustraci jsem ho jednou použil a jednou ne.

Protože je sloupec poznamka v obou tabulkách, musela se použít tečková notace pro jejich odlišení. Sloupec id je jen v jedné tabulce, takže se tečková notace použít nemusí, ale pro ilustraci jsem jí v klauzuli WHEREpoužil.

Pokud nějakou tabulku přejmenujete, musíte používat všude v SELECTu nové jméno tabulky (alias).

Ušetřil jsem si trochu práce tím, že místo telefon.poznamka jsem mohl (teda už musel) psát t.poznamka. Snad to i trochu zlepšilo čitelnost příkazu, ne?

Poznámka: Všimněte si, že se v příkazu SELECT nejdříve alias použije (v tečkové notaci t.poznamka a k.poznamka) a až poté se definuje (telefon t, kontakt k).

CROSS JOIN

Spojování tabulek tím, že je vyjmenujete za FROM je, řekněme, zastaralý způsob. Pořád se hojně používá (je to pohodlné), ale SQL standard přišel s klíčovým slovem JOIN.

JOIN má několik verzí. První, kterou představím, je nepříliš používaný CROSS JOIN. Není příliš používaný, protože dělá přesně to samé, co vyjmenování tabulek za FROM – udělá kartézský součin všech řádků z tabulek, tj. spojí každou řádku z první tabulky s řádkou z druhé tabulky. Podmínkou WHERE pak můžete vybrat jen ty řádky, které vás zajímají:

```
rimmer1=> SELECT prijmeni, jmeno, telefon, t.poznamka AS "O telefonu",
k.poznamka AS "O kontaktu"
FROM telefon AS t CROSS JOIN kontakt k
```

```
WHERE kontakt_id = k.id;
 prijmeni    | jmeno | telefon |  O telefonu  | O kontaktu
-------------+-------+---------+--------------+------------
 Drevokocur  | Pavel | 555555  |              |
 Bílek       | Petr  | 555555  |              |
 Drevokocur  | Lukas | 555555  |              |
 Drevokocur  | Lukas | 555555  | Telefon domu |
 Drevokocurova | Pavla | 555555  |              |
(5 řádek)
```

Jak vidíte, jediný rozdíl oproti příkladu z odstavce o Používání aliasů je ten, že se místo čárky mezi tabulkami napsalo CROSS JOIN. A protože čárka zabere míň psaní, používá se většinou ona, namísto CROSS JOIN.

INNER JOIN

INNER JOIN („vnitřní spojení") dělá totéž co CROSS JOIN. Má však navíc klauzuli **ON**, která říká, jaké řádky se mají spojit. Když do klauzule ON napíšete stejnou podmínku, jako do klauzule WHERE z příkladu k CROSS JOIN, výsledek bude stejný. Je tu jen drobný rozdíl v teorii:

CROSS JOIN nejdříve spojí všechno se vším a pak pomocí podmínky WHERE profiltruje jen vyhovující řádky.

Naproti tomu INNER JOIN spojí jen to, co vyhovuje podmínce v ON.

Interně jsou ovšem DBMS optimalizované, takže CROSS JOIN ve skutečnosti netahá všechna data z databáze, aby je pak profiltroval. Rozdíl mezi CROSS JOIN + WHERE a INNER JOIN + ON není fakticky žádný.

Jediná výhoda INNER JOIN + ON je sémantická: klauzulí ON říkáte, podle čeho spojujete a pak můžete přidat ještě WHERE na odfiltrování záznamů. U CROSS JOIN byste všechno psali do WHERE a o tuto sémantiku byste přišli:

```
rimmer1=> SELECT prijmeni, jmeno, telefon, t.poznamka AS "O telefonu",
k.poznamka AS "O kontaktu"
FROM telefon t CROSS JOIN kontakt k
WHERE kontakt_id = k.id AND prijmeni = 'Drevokocur' AND jmeno = 'Lukas';
 prijmeni  | jmeno | telefon |  O telefonu  | O kontaktu
-----------+-------+---------+--------------+------------
 Drevokocur | Lukas | 555555  |              |
 Drevokocur | Lukas | 555555  | Telefon domu |
(2 řádky)

rimmer1=> SELECT prijmeni, jmeno, telefon, t.poznamka AS "O telefonu",
k.poznamka AS "O kontaktu"
FROM telefon t INNER JOIN kontakt k
ON kontakt_id = k.id
WHERE prijmeni = 'Drevokocur' AND jmeno = 'Lukas';
 prijmeni  | jmeno | telefon |  O telefonu  | O kontaktu
-----------+-------+---------+--------------+------------
 Drevokocur | Lukas | 555555  |              |
 Drevokocur | Lukas | 555555  | Telefon domu |
(2 řádky)

rimmer1=> SELECT prijmeni, jmeno, telefon, t.poznamka AS "O telefonu",
k.poznamka AS "O kontaktu"
FROM telefon t CROSS JOIN kontakt k
ON kontakt_id = k.id
WHERE prijmeni = 'Drevokocur' and jmeno = 'Lukas';
ERROR:  syntax error at or near "ON"
ŘÁDKA 3: ON kontakt_id = k.id WHERE prijmeni = 'Drevokocur' and jmeno...
```

Už vás asi nepřekvapí, že výsledek prvních dvou SELECTů je stejný. V příkladu s INNER JOIN je jen trochu čitelnější, která podmínka slouží ke spojení řádků a která pro vyfiltrování výsledku.

Třetí SELECT skončil chybou, protože CROSS JOIN nemá klauzuli ON.

Klíčové slovo INNER je nepovinné. Pokud napíšete jen JOIN, DBMS to bude chápat jako INNER JOIN.

LEFT OUTER JOIN

Jestli jsem vás ještě nepřesvědčil o důležitosti ON, tak teď vám ukážu jeho praktický význam.

LEFT OUTER JOIN vybere **úplně všechny záznamy** z první tabulky, **a k nim připojí** druhou tabulku podle podmínky **ON**. Jinak řečeno, z první (levé) tabulky uvidíte všechny záznamy (v příkladě to budou všechny kontakty), ať už k nim existuje odpovídající záznam z druhé (pravé) tabulky nebo ne. Tam, kde k záznamu levé tabulky existuje záznam z pravé tabulky (v našem případě telefon) bude tento záznam připojen, jinak bude ve sloupcích pro druhou tabulku NULL.

Klíčové slovo OUTER je nepovinné.

```
rimmer1=> SELECT jmeno, prijmeni, predvolba, telefon
FROM kontakt
LEFT JOIN telefon ON kontakt_id = kontakt.id;
 jmeno |  prijmeni   | predvolba | telefon
-------+-------------+-----------+---------
 Pavel | Drevokocur  | 737       | 555555
 Petr  | Bílek       | 608       | 555555
 Lukas | Drevokocur  | 776       | 555555
 Lukas | Drevokocur  | 312       | 555555
 Pavla | Drevokocurova | 604     | 555555
 Milos | Drevokocur  |           |
 Tomáš | Baťa        |           |
 Tomas | Drevokocur  |           |
 Jan   | Baťa        |           |
 Jana  | Drevokocurova |         |
(10 řádek)
```

Všimněte si, že z tabulky kontaktů teď vidíte všechny záznamy. Jeden záznam vidíte 2x (Lukas Drevokocur), protože k němu existují 2 telefony.
Kdybyste teď chtěli vyfiltrovat jen kontakty s příjemním Drevokocur, museli byste použít podmínku WHERE. Podmínka ON totiž nevyfiltruje z levé tabulky (tabulky kontaktů) vůbec nic! A to je ten podstatný rozdíl mezi ON a WHERE.
Zatímco u INNER JOIN ON filtruje z obou tabulek, takže oproti WHERE žádný faktický rozdíl.

RIGHT OUTER JOIN

RIGHT OUTER JOIN je totéž co LEFT OUTER JOIN, jen podmínka ON nevyfiltruje nic z pravé tabulky. (Respektive, psal jsem přece že ON se používá na spojování, takže, přesněji řečeno, RIGHT OUTER JOINvypíše vše z pravé tabulky, plus připojí všechny existující záznamy z levé tabulky odpovídající podmínce ON).
Klíčové slovo OUTER je nepovinné.

```
rimmer1=> SELECT jmeno, prijmeni, predvolba, telefon
          FROM kontakt RIGHT JOIN telefon
            ON kontakt_id = kontakt.id;
 jmeno |   prijmeni    | predvolba | telefon
-------+---------------+-----------+---------
 Pavel | Drevokocur    | 737       | 555555
  Petr | Bílek         | 608       | 555555
 Lukas | Drevokocur    | 776       | 555555
 Lukas | Drevokocur    | 312       | 555555
 Pavla | Drevokocurova | 604       | 555555
       |               | 908       | 300300
(6 řádek)
```

Ve výsledku jsou všechny záznamy z pravé tabulky (z telefonů) a všechny záznamy z kontaktů odpovídající podmínce ON.
A ano, předchozí příklad by se dal přepsat pomocí LEFT OUTER JOIN se stejným výsledkem, stačí jen přehodit pořadí tabulek:

```
rimmer1=> SELECT jmeno, prijmeni, predvolba, telefon
          FROM telefon LEFT JOIN kontakt
            ON kontakt_id = kontakt.id;
 jmeno |   prijmeni    | predvolba | telefon
-------+---------------+-----------+---------
 Pavel | Drevokocur    | 737       | 555555
  Petr | Bílek         | 608       | 555555
 Lukas | Drevokocur    | 776       | 555555
 Lukas | Drevokocur    | 312       | 555555
 Pavla | Drevokocurova | 604       | 555555
       |               | 908       | 300300
(6 řádek)
```

Ještě jeden příklad na závěr: jak zjistit, ke kterým telefonům nemáte žádný kontakt? Využiji toho, že spojuji tabulku telefon přes sloupec kontakt_id. Když bude ve výsledku tento sloupec NULL, tak je jasené, že není s ničím spojen:

```
rimmer1=> SELECT jmeno, prijmeni, predvolba, telefon
FROM telefon LEFT JOIN kontakt ON kontakt_id = kontakt.id
          WHERE kontakt_id IS NULL;
 jmeno | prijmeni | predvolba | telefon
-------+----------+-----------+---------
       |          | 908       | 300300
(1 řádka)
```

FULL OUTER JOIN

FULL OUTER JOIN vybere všechny záznamy z obou tabulek. Kde si záznamy odpovídají (odpovídají podmínce ON), budou spojeny v jeden řádek. Pokud pro řádek neexistuje odpovídající řádek v jedné či druhé tabulce, bude nakonec vypsán a hodnoty z druhé tabulky nahrazeny hodnotou NULL.
V případě tabulek kontakt a telefon znamená, že ve výsledku uvidíte všechna jména (i ty, u kterých není telefonní číslo) i všechna telefonní čísla (i ty, ke kterým nepatří žádné jméno).
Klíčové slovo OUTER je nepovinné.

```
rimmer1=> SELECT jmeno, prijmeni, predvolba, telefon
          FROM telefon FULL JOIN kontakt
            ON kontakt_id = kontakt.id;
 jmeno |   prijmeni    | predvolba | telefon
-------+---------------+-----------+---------
 Pavel | Drevokocur    | 737       | 555555
  Petr | Bílek         | 608       | 555555
 Lukas | Drevokocur    | 776       | 555555
 Lukas | Drevokocur    | 312       | 555555
 Pavla | Drevokocurova | 604       | 555555
       |               | 908       | 300300
 Milos | Drevokocur    |           |
 Tomáš | Baťa          |           |
 Tomas | Drevokocur    |           |
  Jan  | Baťa          |           |
 Jana  | Drevokocurova |           |
(11 řádek)
```

USING, NATURAL JOIN

Jen telegraficky zmíním, že místo ON můžete použít USING, pokud používáte podmínku rovnosti nad sloupci stejného jména. Prostě to znamená, že místo ON table1.sloupec = table2.sloupec můžete psát USING(sloupec).
To bych ale musel mít jinou politiku pojmenovávání primárních a cizích klíčů, aby se mi USING hodilo.
Třeba v tabulce kontakt bych musel mít místo id kontakt_id a pak bych mohl použít kontakt JOIN telefon USING(kontakt_id).
NATURAL JOIN je ještě větší zkratka. NATURAL JOIN je vlastně INNER JOIN, který se spojuje pomocí sloupců stejného jména z obou tabulek.

Kdybych měl v tabulce kontakt místo id kontakt_id, pak by následující tři JOINy dělali to samé*:

```
... kontakt JOIN telefon ON kontakt.kontakt_id = telefon.kontakt_id WHERE ...
... kontakt JOIN telefon USING(kontakt_id) WHERE ...
... kontakt NATURAL JOIN telefon WHERE ...
```

* NATURAL JOIN je velice náchylný na změnu definice tabulek. Náhodou přidáte do tabulky sloupec stejného jména jako je v té druhé a NATURAL JOIN jej potichoučku zahrne do slučovací podmínky (na tom, jestli jde o primární a cizí klíč vůbec nesejde!). Protože tabulka telefon i kontakt mají sloupeček stejného jména (poznamka), byl by třetí JOIN z příkladu výše ve skutečnosti ekvivalentní těmto:

```
... kontakt JOIN telefon ON kontakt.kontakt_id = telefon.kontakt_id AND kontakt.poznamka = telefon.poznamka WHERE ...
... kontakt JOIN telefon USING(kontakt_id, poznamka) WHERE ...
```

Traduje se, že je z tohoto důvodu v některých zemích použití NATURAL JOIN trestán smrtí :-). Já vám tedy rozhodně doporučuji NATURAL JOIN nepoužívat nikdy, nikde a za nic. Ani pod dohledem profesionálů.

UNION

UNION (sjednocení) slouží ke spojování tabulek „nad sebe". Tabulky, které chcete nad sebe spojit, musí mít**stejný počet sloupců** a sloupce musí mít **stejný datový typ** (nebo alespoň „kompatibilní" datový typ).

Jak UNION funguje? Napíšete libovolný SELECT, pak klíčové slovo UNION a pak zase SELECT, který vrací stejný počet sloupců …

Třídění (ORDER BY) nemůže být součástí jednotlivých SELECTů v UNIONu. Setřídit můžete až celou výslednou tabulku!

Využití příkazu UNION ukážu na tabulace zamestnanci z příkladu z kapitoly Vytváření relací.

V kapitole SELECT II jsem popisoval funkce pro získávání statistických údajů. Umíte je vypsat v tabulce s jedním řádkem:

```
rimmer1=> SELECT 'Průměr:' AS "Funkce", AVG(plat) as "Výsledek"
          FROM zamestnanci;
   Funkce  |      Výsledek
-----------+--------------------
Průměr:    | 13400.000000000000
(1 řádka)

rimmer1=> SELECT 'Maximum:' AS "Funkce", MAX(plat) AS "Výsledek"
          FROM zamestnanci;
   Funkce  | Výsledek
-----------+----------
Maximum:   |    22000
(1 řádka)
```

Poznámka: Všimněte si, že Průmer, jakožto konstantní hodnota textu, je v jednoduchých uvozovkách, kdežto Funkce, jakožto název sloupce, ve dvojitých. Jelikož název sloupce neobsahuje mezeru, jsou zde dvojité uvozovky zbytečné, v případě hodnoty, která bude ve sloupci, jsou (jednoduché) uvozovky povinné. To jen tak na připomenutí.

Určitě pro vás nebude problém tyto hodnoty vypsat jedním SELECTem v jednom řádku. Ale co když je chcete vypsat jedním příkazem pod sebou? Na to je tu UNION.

Pomocí UNION můžete spojit předchozí dvě tabulky vytvořené příkazy SELECT do jedné, neboť mají stejný počet sloupců (dva) stejných datových typů (řetězec a číslo).

```
rimmer1=> SELECT 'Průměr:' AS "Funkce", AVG(plat) as "Výsledek"
          FROM zamestnanci
          UNION
          SELECT 'Maximum:' AS "Funkce2", MAX(plat) AS "Výsledek2"
          FROM zamestnanci;
   Funkce  |      Výsledek
-----------+--------------------
Maximum:   |          22000
Průměr:    | 13400.000000000000
(2 řádky)
```

V druhém SELECTu jsem schválně změnil názvy sloupců, abyste viděli, že se berou v potaz názvy jen z prvního SELECTu.

Spojovat můžete kolik chcete tabulek vytvořených příkazem SELECT (nejenom tabulky se statistickými hodnotami, které jsme si ukazovali). Jediné (pochopitelné) omezení je v počtu sloupců a v nutnosti stejných datových typů těchto sloupců:

```
rimmer1=> SELECT 'Průměr:' AS "Funkce", AVG(plat) as "Výsledek"
          FROM zamestnanci
          UNION
          SELECT 'Maximum:' AS "Funkce", MAX(plat) AS "Výsledek"
          FROM zamestnanci
          UNION
          SELECT 'Minimum:', MIN(plat)
          FROM zamestnanci
          ORDER BY "Výsledek";
   Funkce  |      Výsledek
-----------+--------------------
Minimum:   |          9000
Průměr:    | 13400.000000000000
Maximum:   |          22000
(3 řádky)
```

Pokud nepoužijete ORDER BY, seřazení řádků ve výsledků není definováno. To znamená, že může být jakékoliv. To znamená, že mohou být i proházené řádky ze SELECTů v UNIONu. Nenechte se zmýlit tím, že se vám při vašich pokusech vrací vždy nejdříve řádky z jedné tabulky a pak z té druhé. Jakmile začnete pracovat s vekým množstvím dat, do hry začnou vstupovat indexy atp., najednou se to začne chovat jinak!

Pokud chcete mít ve výsledku nejdříve řádky z jedné tabulky a pak z druhé, jde to udělat třeba takto:

```
rimmer1=> SELECT prijmeni, jmeno, 1 as "tab" FROM kontakt
          UNION
          SELECT prijmeni, jmeno, 2 as "tab" FROM zamestnanci
          ORDER BY tab, prijmeni, jmeno;
```

```
  prijmeni    | jmeno  | tab
--------------+--------+-----
 Baťa         | Jan    |   1
 Baťa         | Tomáš  |   1
              ...
 Drevokocurova | Jana  |   1
 Drevokocurova | Pavla |   1
 Jerry        | Tom    |   2
 King         | Leopold |  2
              ...
 Pavova       | Lenka  |   2
 Trn          | Vasek  |   2
(19 řádek)
```

Co jsem myslel tím, že musí být datové typy v sloupcích stejné, nebo **kompatibilní**? DBMS je musí umět převést na stejný datový typ:

```
rimmer1=> SELECT '2013' AS "datum" UNION SELECT 333 AS "integer";
 datum
-------
   333
  2013
(2 řádky)
rimmer1=> SELECT '2013-12-31' AS "datum" UNION SELECT 333 AS "integer";
ERROR:  invalid input syntax for integer: "2013-12-31"
ŘÁDKA 1: SELECT '2013-12-31' AS "datum" UNION SELECT 333 AS "integer"...
                                                ^
```

V prvním příkladu převedl DBMS text '2013' na datový typ INTEGER, v druhém příkladu už to nešlo.
Dobrá rada: Když to nejde jinak, převeďte sloupce explicitně na TEXT. K tomu slouží funkce **CAST**, nebo postgresovské ::.

```
rimmer1=> SELECT '2013-12-31' AS "text"
UNION
SELECT CAST(333 AS TEXT) AS "taky text";
    text
------------
 2013-12-31
 333
(2 řádky)
rimmer1=> SELECT '2013-12-31' AS "text"
UNION
SELECT 333::text AS "taky text";
    text
------------
 2013-12-31
 333
(2 řádky)
```

Ještě lepší rada: u uložených SQL příkazů (v nějakém programu) raději používejte přetypování vždy. Mohlo by se stát, že si to vyzkoušíte bez přetypování s daty, které postgres přetypuje (jako '2013'), ale později do tabulky vložíte data, která přetypovat nejdou ('2013-12-31') a SQL příkaz vám umře.

UNION ALL

UNION má jednu zákeřnou vlastnost, a to, že eliminuje duplicitní řádky z výsledku. To většinou není to, co chcete. Zabránit tomu se dá použitím **ALL**:

```
rimmer1=> SELECT 'A' UNION SELECT 'B' UNION SELECT 'A';
 ?column?
----------
 B
 A
(2 řádky)
rimmer1=> SELECT 'A' UNION ALL SELECT 'B' UNION SELECT 'A';
 ?column?
----------
 B
 A
(2 řádky)

rimmer1=> SELECT 'A' UNION ALL SELECT 'B' UNION ALL SELECT 'A';
 ?column?
----------
 A
 B
 A
(3 řádky)

rimmer1=> SELECT 'A' UNION SELECT 'B' UNION ALL SELECT 'A';
 ?column?
----------
 A
 B
```

```
A
(3 řádky)
```
Z příkladu, doufám, vše jasné :-)
Prioritu UNION (i INTERSECT a EXCEPT, viz níže) můžete určit závorkami:
```
rimmer1=> SELECT 'A' UNION (SELECT 'B' UNION ALL SELECT 'A');
?column?
----------
B
A
(2 řádky)
```
Používejte UNION s ALL, pokud nemáte dobrý důvod k eliminaci duplicitních řádků.

INTERSECT, EXCEPT

INTERSECT [ALL] (průnik) funguje podobně jako UNION, ale vrací jen řádky, které jsou v obou tabulkách. Bez ALL eliminuje duplicitní řádky. (Řádky jsou stejné, pokud mají všechny sloupečky stejnou hodnotu).
EXCEPT [ALL] (rozdíl) vrátí všechny řádky z první tabulky, kromě těch, které se vyskytují v druhé tabulce.
Tabulkami myslím výsledky SELECTů v INTERSECTu nebo EXCEPTu.

MySQL

MySQL implementuje CROSS JOIN jako synonymum k INNER JOIN, takže jej můžete používat s ON.
MySQL neumí FULL [OUTER] JOIN. Můžete ho nasimulovat pomocí UNION. (Tentokrát se zrovna hodí UNIONbez ALL).
```
mysql> SELECT jmeno, prijmeni, predvolba, telefon
FROM telefon LEFT JOIN kontakt ON kontakt_id = kontakt.id
UNION
SELECT jmeno, prijmeni, predvolba, telefon
FROM telefon RIGHT JOIN kontakt ON kontakt_id = kontakt.id;
+---------+--------------+-----------+---------+
| jmeno   | prijmeni     | predvolba | telefon |
+---------+--------------+-----------+---------+
| NULL    | NULL         | 908       | 300300  |
| Pavel   | Drevokocur   | 737       | 555555  |
| Petr    | Bílek        | 608       | 555555  |
| Lukas   | Drevokocur   | 312       | 555555  |
| Lukas   | Drevokocur   | 776       | 555555  |
| Pavla   | Drevokocurova | 604      | 555555  |
| Tomas   | Drevokocur   | NULL      | NULL    |
| Milos   | Drevokocur   | NULL      | NULL    |
| Jana    | Drevokocurova | NULL     | NULL    |
| Tomáš   | Baťa         | NULL      | NULL    |
| Jan     | Baťa         | NULL      | NULL    |
+---------+--------------+-----------+---------+
11 rows in set (0.04 sec)
```
MySQL je trochu chytřejší při konverzi datových typů, takže tento SQL dotaz projde, protože se hodnoty zkonvertují automaticky na TEXT.
```
SELECT '2013' AS "datum" FROM dual UNION SELECT 333 AS "integer" FROM dual;
```
U více UNIONů nemůžete jednoduše určovat prioritu závorkami. Můžete si ale pomoci tatko:
```
mysql> SELECT 'A' UNION (SELECT 'B' UNION ALL SELECT 'A');
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your
MySQL server version for the right syntax to use near 'UNION ALL SELECT 'A')' at line 1

mysql> SELECT 'A'
UNION
SELECT * FROM (SELECT 'B' UNION ALL SELECT 'A') AS temp;
+---+
| A |
+---+
| A |
| B |
+---+
2 rows in set (0.00 sec)
```
*SELECT * FROM je SELECT ze SELECTu, který musí být pojmenován, proto to AS temp na konci.*
MySQL neumí INTERSECT [ALL] ani EXCEPT [ALL].

SQLite

SQLite, stejně jako MySQL, implementuje CROSS JOIN jako synonymum k INNER JOIN.
SQLite nepodporuje RIGHT [OUTER] JOIN. Ale už víte, že se dá snadno nahradit pomocí LEFT [OUTER] JOIN.
SQLite nepodporuje ani FULL [OUTER] JOIN. Můžete ho nahradit pomocí UNION a dvou LEFT JOIN.
```
SELECT jmeno, prijmeni, predvolba, telefon
FROM telefon LEFT JOIN kontakt ON kontakt_id = kontakt.id
UNION
SELECT jmeno, prijmeni, predvolba, telefon
FROM kontakt LEFT JOIN telefon ON kontakt_id = kontakt.id;
```
V prvním LEFT JOINu je telefon nalevo a kontakt napravo, v druhém LEFT JOINu je to naopak.
SQLite je stejně chytrý jako MySQL při konverzi datových typů v UNIONu. Takže tento příklad funguje:
```
SELECT '2013' AS "datum" FROM dual UNION SELECT 333 AS "integer" FROM dual;
```
Stejně jako v MySQL nemůžete prioritu UNION určit jednoduše závorkami, ale musíte si pomoci SELECTem ze SELECTu.
SQLite **podporuje** INTERSECT a EXCEPT, ale obojí bez ALL.

Oracle

V Oracle nemůžete používat AS pro vytvoření aliasu tabulky. Bez AS to funguje:

```
SELECT prijmeni, jmeno, telefon, t.poznamka AS "O telefonu",
k.poznamka AS "O kontaktu" FROM telefon t, kontakt  k
WHERE kontakt_id = k.id;

SELECT prijmeni, jmeno, telefon, t.poznamka AS "O telefonu",
k.poznamka AS "O kontaktu" FROM telefon  t CROSS JOIN kontakt k
WHERE kontakt_id = k.id;
```

Další drobný problém je s tím, že Oracle převádí automaticky identifikátory sloupců na velká písmena. Takže se musí používat uvozovky, když je to potřeba:

```
SELECT prijmeni, jmeno, 1 as "tab" FROM kontakt
UNION
SELECT prijmeni, jmeno, 2 as "tab" FROM zamestnanci
ORDER BY "tab", prijmeni, jmeno;
```

"tab" musí být v uvozovkách i v ORDER BY, protože je v uvozovkách v prvním SELECTu.

Všechny SELECTy, které jsem v této lekci používal bez FROM, musí mít v Oracle FROM (nejlépe z tabulky dual).

Oracle je v automatickém převodu datových typů v UNIONu nejhloupější, takže nefunguje ani jeden z příkladů bez CAST. Takhle to funguje:

```
SELECT '2013-12-31' AS "datum" FROM dual
UNION
SELECT CAST(333 AS VARCHAR2(20)) AS "text" FROM dual;

SELECT '2013-12-31' AS "text" FROM dual
UNION
SELECT CAST(333 AS VARCHAR2(20)) AS "taky text" FROM dual;
```

V Oracle můžete použít závorky k určení priority UNIONů:

```
SELECT 'A' FROM dual
UNION
(SELECT 'B' FROM dual UNION ALL SELECT 'A' FROM dual);
```

Oracle podporuje INTERSECT (bez ALL). Namísto EXCEPT používá **MINUS** (dělá to to samé, jen si to jinak říká).

SELECT IV - poddotazy

Tato kapitola bude trochu náročnější mozkové cvičení. Proberu zde poddotazy a korelované dotazy, které se vám budou v praxi hodit častěji, než by se mohlo zdát.

~~Protože je toto poslední kapitola ze základního kurzu o SQL~~, využiju teď chvilku k tomu, abych udělal reklamu svým dalším tutoriálům. Například byste se mohli podívat na programovací jazyk Python, který umožňuje pomocí modulu psycopg2 s databází PostgreSQL komunikovat. Pokud nevíte o programování zhola nic, pak to budete mít trochu těžší. Programování je vysvětleno od úplných začátků v článku o programovacím jazyku C.
Konec reklamního bloku. A teď zpět k Postgresu :-)

Poddotazy

Poddotazem se myslí příkaz SELECT, jehož výsledek se použije buď jako tabulka pro další SELECT v rámci jednoho SQL dotazu, nebo se vrácená hodnota (hodnoty) použijí třeba v podmínce WHERE.

Podívejte se nejdříve na využití SELECTu vracejícího jednu hodnotu. Takovým příkazem bude typicky nějaká statistická hodnota, například průměrný plat. Pokud vás například budou zajímat všichni zaměstnanci s průměrným a nadprůměrným platem, docílíte toho následujícím poddotazem (tabulku zamestnanci jsem vytvořil v kapitole Vytváření relací). Pro zajímavost jsem výsledek setřídil sestupně dle výše platu.

```
rimmer1=> SELECT * FROM zamestnanci
WHERE plat >= (SELECT AVG(plat) FROM zamestnanci)
ORDER BY plat DESC;
 jmeno | prijmeni | plat  | oddeleni_id | rodne_cislo
-------+----------+-------+-------------+-------------
 Tomas | Mann     | 22000 |           3 | 8001010608
 Vasek | Trn      | 16000 |           3 | 8001010609
 Lenka | Pavova   | 15000 |           2 | 8001010604
 Tom   | Jerry    | 15000 |           2 | 8001010605
(4 řádky)
```

Poddotaz musí být v závorkách a musí vrátit jen jednu hodnotu (jeden řádek s jedním sloupcem), jinak by tato podmínka nedávala smysl.

Teď řekněme, že mě zajímá průměrný plat z předchozího výsledku. Jednou z možností by bylo udělat z předchozího SELECTu pohled (VIEW) a nad ním zavolat AVG. Nebo se s vytvářením pohledu nebudu zdržovat a udělám rovnou SELECT z předchozího SELECTu.

```
rimmer1=> SELECT AVG(xxx.plat) FROM (
SELECT * FROM zamestnanci
WHERE plat >= (SELECT AVG(plat) FROM zamestnanci)
ORDER BY plat DESC
) xxx;
```

```
        avg
--------------------
 17000.000000000000
(1 řádka)
```

Všimněte si, že jsem vnitřní SELECT pojmenoval xxx. Respektive jsem tak pojmenoval výsledek vnitřního SELECTu, kterým je vždy **tabulka**. Takže jsem „stvořil" tabulku xxx, ze které jsem provedl SELECT AVG(xxx.plat)
Pokud používám takto poddotaz jako tabulku, **vždycky musí být pojmenována**.
Použití tečkové notace s názvem tabulky (xxx.plat) už v příkladu nebylo nezbytné, použil jsem ho jen pro příklad.
SELECTů jako tabulek můžete v jednom příkazu použít více, můžete je pak JOINovat a dělat s nimi cokoliv, co s pohledem (VIEW). Poddotaz není vlastně nic jiného, než ad-hoc VIEW.

Porovnávání hodnoty s polem hodnot

V následujících odstavcích popíšu operátory, které se používají k porovnávání s polem hodnot. Můžete je používat s poddotazy i bez nich, ale teprve s poddotazy oceníte jejích sílu :-)

IN

IN je operátor, který říká, zda je proměnná v nějaké množině hodnot nebo ne (alespoň jednou), a používá se v podmínce WHERE (stejně jako třeba = atp.). Množinou můžou být hodnoty oddělené čárkou uzavřené v závorkách, nebo výsledek poddotazu > **1 sloupcec** s libovolným počtem řádků.
Například budu chtít zjistit zaměstnance, kteří pracují ve všech právních odděleních.
Můžu na to jít takto:

```
rimmer1=> SELECT prim_klic FROM oddeleni WHERE nazev = 'Pravni oddeleni';
 prim_klic
-----------
         2
         3
(2 řádky)

rimmer1=> SELECT * FROM zamestnanci WHERE oddeleni_id IN (2, 3);
 jmeno  | prijmeni | plat  | oddeleni_id | rodne_cislo
--------+----------+-------+-------------+-------------
 Lenka  | Pavova   | 15000 |           2 | 8001010604
 Tom    | Jerry    | 15000 |           2 | 8001010605
 Martin | Luter    | 12000 |           2 | 8001010606
 Leopold| King     | 13000 |           2 | 8001010607
 Tomas  | Mann     | 22000 |           3 | 8001010608
 Vasek  | Trn      | 16000 |           3 | 8001010609
 Stary  | Osel     |  9000 |           3 | 8001010610
(7 řádek)
```

Nebo rovnou takto:

```
rimmer1=> SELECT * FROM zamestnanci WHERE oddeleni_id IN (
  SELECT prim_klic FROM oddeleni WHERE nazev = 'Pravni oddeleni'
);
 jmeno  | prijmeni | plat  | oddeleni_id | rodne_cislo
--------+----------+-------+-------------+-------------
 Lenka  | Pavova   | 15000 |           2 | 8001010604
 Tom    | Jerry    | 15000 |           2 | 8001010605
 Martin | Luter    | 12000 |           2 | 8001010606
 Leopold| King     | 13000 |           2 | 8001010607
 Tomas  | Mann     | 22000 |           3 | 8001010608
 Vasek  | Trn      | 16000 |           3 | 8001010609
 Stary  | Osel     |  9000 |           3 | 8001010610
(7 řádek)
```

Jak vidíte, použití poddotazu vám ušetří ruční přepisování výsledku prvního SELECTu do podmínky druhého.
Opakem IN je **NOT IN**. NOT IN vyhoví tehdy, pokud hodnota není v množině ani jednou.

ALL

ALL spolupracuje s dalším operátorem (dle vašeho výběru), který aplikuje na celou množinu. Řádek je do výsledku zahrnut pouze tehdy, vrátí-li operátor **pro všechny hodnoty v množině** true.
Příklad: vypište všechny zaměstnance, kteří mají maximální plat.
Pouze maximální plat je roven nebo větší než všechny platy:

```
rimmer1=> SELECT * FROM zamestnanci
WHERE plat >= ALL (SELECT plat FROM zamestnanci WHERE plat IS NOT NULL);
 jmeno | prijmeni | plat  | oddeleni_id | rodne_cislo
-------+----------+-------+-------------+-------------
 Tomas | Mann     | 22000 |           3 | 8001010608
(1 řádka)
```

Všimněte si podmínky WHERE plat IS NOT NULL, bez které by ALL použil operátor na NULL hodnotu (která se vyskytuje v jednom řádku tabulky zamestnanci) a tím pádem by bez milosti vrátil pro každý řádek NULL. Výsledkem by bylo, že podmínka nikdy nevyhovuje. (Nic není >= NULL.)

S ALL můžete použít jakýkoliv operátor (v příkladu >=), který vrací booleovskou hodnotu (true nebo false).
Opakem ALL není *výraz operátor* NOT ALL (…), ale NOT *výraz operator* ALL (…).

```
SELECT * FROM zamestnanci
WHERE NOT plat >= ALL (SELECT plat FROM zamestnanci WHERE plat IS NOT NULL);
...
```

ANY/SOME

ANY je podobné jako ALL, jen s tím rozdílem, že vrací true, když operátor vrátí true **alespoň pro jednu hodnotu z množiny**.
V příkladu vyberu všechny platy, které jsou větší nebo rovny dvojnásobku nejmenšího platu. (Nejmenší plat je 9000. Takže podmínku splní každý plat který je větší nebo roven 18000).

Podmínka ANY požaduje splnění podmínky (>=) alespoň u jednoho záznamu. V příkladě tedy stačí, aby byl plat větší než dvojnásobek toho nejmenšího.

```
rimmer1=> SELECT * FROM zamestnanci
WHERE plat >= ANY (SELECT plat*2 FROM zamestnanci);
 jmeno | prijmeni | plat  | oddeleni_id | rodne_cislo
-------+----------+-------+-------------+-------------
 Tomas | Mann     | 22000 |           3 | 8001010608
(1 row)
```

Všimněte si, že ve vnořeném SELECTu není podmínka NOT NULL. ANY vrací true, když vyhoví alespoň jedna hodnota z množiny. Když jsou některé hodnoty NULL, tak sice nevyhoví, ale to nevadí.

SOME je jen synonymum pro ANY.

Operátor IN je vlastně jen synonymum pro = ANY.

```
rimmer1=> SELECT * FROM zamestnanci
WHERE oddeleni_id = ANY (
SELECT prim_klic FROM oddeleni WHERE nazev = 'Pravni oddeleni'
);
 jmeno  | prijmeni | plat  | oddeleni_id | rodne_cislo
--------+----------+-------+-------------+-------------
 Lenka  | Pavova   | 15000 |           2 | 8001010604
                    ...
 Stary  | Osel     |  9000 |           3 | 8001010610
(7 řádek)
```

Porovnávání řádků

V SQL existuje něco, čemu se říká **řádkový konstruktor**.

Řádkový konstruktor vypadá takto: ROW(hodnota, hodnota, …)

Klíčové slovo ROW není povinné, pokud konstruujete řádek s více jak jedním sloupcem. Takže jako řádkový konstruktor slouží i jen hodnoty oddělené čárkou uzavřené do závorek.

Příklad použití:

```
rimmer1=> SELECT ROW('a', 3, 5.2) as priklad;
 priklad
-----------
 (a,3,5.2)
(1 řádka)
```

A teď k čemu je to dobré: řádky můžete porovnávat. Porovnávají se z leva do prava po jednotlivých sloupcích (hodnotách). Aby se dva řádky rovnali, musí být všechny hodnoty stejné. Aby byl jeden řádek větší než druhý, musí mít větí první nestejnou hodnotu (z leva).

```
rimmer1=> SELECT ('a', 3, 5.2) = ('a', 3, 5.2) as rovnost,
('a', 3, 5.3) > ('a', 3, 5.2) as vetsi;
 rovnost | vetsi
---------+-------
 t       | t
(1 řádka)
```

Praktičtější využití může být toto (oba SELECTy dělají totéž):

```
SELECT * FROM telefon WHERE predvolba = '908' AND telefon = '300300';
SELECT * FROM telefon WHERE (predvolba, telefon) = ('908', '300300');
```

Zábava příjde s poddotazy. Jako příklad zkusím najít všechny telefony, které mají předvolbu a operatora jinou, než je v tabulce operator. Abych něco našel, tak si nejdřív něco takového do tabulky vložím:

```
INSERT INTO telefon (predvolba, telefon, operator_id) VALUES ('908','666666',2);
```

Teď onen SELECT s poddotazem:

```
rimmer1=> SELECT * FROM telefon t
WHERE (t.predvolba,t.operator_id) NOT IN (SELECT p.predvolba, p.operator_id FROM predvolba p);
 predvolba | telefon | operator_id | tarif_id | kontakt_id | poznamka
-----------+---------+-------------+----------+------------+----------
 908       | 666666  |           2 |          |            |
(1 řádka)
```

Předchozí SELECT se dá přepsat pomocí JOINu. Není z něho možná tak „čitelné" o co se snažím, ale JOIN verze SELECTů bývají v praxi o dost rychlejší než SELECTy s poddotazy. (V tabulce s pár tisíci záznami rozdíl obvykle nepoznáte.)

```
SELECT t.* FROM telefon t
LEFT JOIN predvolba p
ON (t.predvolba, t.operator_id) = (p.predvolba, p.operator_id)
WHERE p.predvolba is null;
```

Pozor! Následující SQL dotaz dělá něco jiného a nevrátí žádnou řádku:

```
SELECT t.* FROM telefon t WHERE
t.predvolba  NOT IN (SELECT p.predvolba   FROM predvolba p) AND
t.operator_id NOT IN (SELECT p2.operator_id FROM predvolba p2);
```

Korelované dotazy

Při vytváření korelovaného dotazu se využívá jednak poddotazů a jednak aliasů. Korelované dotazy jsou dotazy několika SELECTů do jedné a té samé tabulky (pomocí poddotazů) v rámci jednoho SQL příkazu. Aliasy slouží k tomu, aby šlo rozeznat z jakého SELECTu (poddotazu) pochází sloupec, se kterým se pracuje.

Korelované dotazy nejsou opravdu nic jiného než poddotazy nad stejnou tabulkou a několik příkladů jste už vyděli s tabulou zamestnanci výše. Jen se navíc navzájem mezi se bou odkazují a ovlivňují tak svůj výsledek.

Pomocí korelovaného dotazu můžu například vypsat všechny zaměstnance, kteří mají maximální plat v oddělení, ve kterém pracují.

```
rimmer1=> SELECT prijmeni, plat, oddeleni_id
FROM zamestnanci z1
WHERE z1.plat = (
```

```
SELECT MAX(z2.plat) FROM zamestnanci z2
WHERE z1.oddeleni_id = z2.oddeleni_id
);
 prijmeni | plat  | oddeleni_id
----------+-------+-------------
 Mala     | 12000 |           1
 Pavova   | 15000 |           2
 Jerry    | 15000 |           2
 Mann     | 22000 |           3
(4 rows)
```

Podívejte se, kde je zakopaný pes. První a druhá řádka příkazu je obyčený SELECT s aliasem tabulky zamestnanci z1. V tuto chvíli by se vybrali všechny řádky z tabulky zamestnanci (alias z1). Mě však zajímají jen ty řádky z tabulky z1, kde je maximální plat pro řádky se stejnou hodnotu v sloupci oddeleni_id.

Tato podmínka je podstatou tohoto korelovaného dotazu. Moc dobře se to pochopit nedá, že? Ale jde to :-).

Ve výsledku to funguje asi takto: Pro každý řádek z tabulky z1 se udělá vnořený SELECT nad tabulkou z2, do jehož podmínky se za z1.oddeleni_id dosadí hodnota aktuálně zpracovávaného řádku z z1, pro který se korelovaný dotaz provádí.

Teoreticky to znamená, že se pro každý řádek ze z1 provede extra poddotaz, což není zrovna nejvýkonější záležitost. V našem příkladě by si mohl optimalizátor DBMS zapamatovat výsledky pro stejná z1.oddeleni_id a nedělat stejný dotaz znovu, ale i tak je lepší se korelovaným dotazům vyhýbat (pokud je to teda možné).

Není důvod, proč si to ještě trochu nezkomplikovat. Místo oddeleni_id bude jistě lepší mít ve výsledku název oddělení. Název oddělení je uložen v tabulce oddeleni, takže se musí předcházející korelovaný dotaz propojit (JOINovat) s touto tabulkou.

```
rimmer1=> SELECT prijmeni, plat, nazev
FROM zamestnanci z1, oddeleni
WHERE z1.oddeleni_id = oddeleni.prim_klic AND
z1.plat = (
SELECT max(z2.plat) FROM zamestnanci z2
WHERE z1.oddeleni_id = z2.oddeleni_id
);
 prijmeni | plat  |      nazev
----------+-------+-----------------
 Mala     | 12000 | Sekretariat
 Pavova   | 15000 | Pravni oddeleni
 Jerry    | 15000 | Pravni oddeleni
 Mann     | 22000 | Pravni oddeleni
(4 rows)
```

MySQL

MySQL zachází poněkud podivně s hodnotou NULL u operátoru ALL. Následující SELECT vrátí maximální plat, i když existuje řádek, kde je plat NULL.

```
SELECT * FROM zamestnanci WHERE plat >= ALL (SELECT plat FROM zamestnanci);
```

V PostgreSQL i Oracle je potřeba přidat do poddotazu podmínku plat IS NOT NULL.

MySQL neumí zacházet s řádkem jako by to byl sloupec (nemá datový typ „řádek“), takže tento příklad nefunguje:

```
SELECT ROW('a', 3, 5.2) as priklad;
```

Jinak funguje vše jako v Postgresu, včetně tohoto:

```
rimmer1> SELECT ('a', 3, 5.2) = ('a', 3, 5.2) as rovnost, ('a', 3, 5.3) > ('a', 3, 5.2) as vetsi;
+---------+-------+
| rovnost | vetsi |
+---------+-------+
|       1 |     1 |
+---------+-------+
1 row in set (0,00 sec)
```

SQLite

SQLite nemá operátory ALL a ANY/SOME.

Nemá ani konstruktor řádku (ROW), takže celou část kapitolu o něm můžte, pokud jde o SQLite, zapomenout. (Což není taková hrůza, protože víte, jak všechny SQL dotazy přepsat bez použití řádkového konstruktoru.)

Vše ostatní z této kapitoly funguje tak, jako v PostgreSQL.

Oracle

Oracle umí řádkový konstruktor tak trochu. Umí ho jen v části WHERE a jen s poddotazy. Proto toto v Oracle nefunguje:

```
SELECT ROW('a', 3, 5.2) as priklad FROM dual;
SELECT ('a', 3, 5.2) = ('a', 3, 5.2) as rovnost, ('a', 3, 5.3) > ('a', 3, 5.2) as vetsi;
SELECT * FROM telefon WHERE (predvolba, telefon) = ('908', '300300');
```

Zatímco první dva SELECTy byste museli nějak přepsat bez použití řádkového konstruktoru, ten třetí jde za použití poddotazu. Toto v Oracle funguje:

```
SELECT * FROM telefon WHERE (predvolba, telefon) = (SELECT '908', '300300' FROMdual);
```

Nicméně, asi by bylo lepší i toto přepsat bez řádkového konstruktoru (viz výše, kde je to ukázáno).

Protože funguje řádkový konstruktor jen v klauzuli WHERE, nefunguje ani toto:

```
SELECT t.* FROM telefon t
LEFT JOIN predvolba p
ON (t.predvolba, t.operator_id) = (p.predvolba,p.operator_id)
WHERE p.predvolba is null;
```

A musí se to přepsat takto:

```
SELECT t.* FROM telefon t
LEFT JOIN predvolba p
ON (t.predvolba = p.predvolba AND t.operator_id = p.operator_id)
WHERE p.predvolba is null;
```

Vše ostatní z této kapitoly funguje tak, jako v PostgreSQL.

Schémata

Cluster, Catalog, Schéma, Tabulka

Už víte, že tabulky musíte vytvořit v nějakém schématu. Na obrázku můžete vidět, jaký je vztah cluseteru (DBMS), databáze (catalog) a schématu dle normy SQL. Jak je to ve skutečnosti implementováno v různých DBMS se dozvíte teď a tady. V této kapitole ukážu, jak používat schémata a některá speciální schémata, které máte v databázi k dispozici.

Public schéma

Schémata se používají k seskupování logicky souvisejících tabulek, k nastavování přístupových práv (nad schématem), nebo k zabránění kolize jmen v různých aplikacích, které využívají tabulky stejného jména k různým účelům. Nic světoborného v tom nehledejte.

V PostgreSQL vytvoříte databázi (catalog) příkazem CREATE DATABASE nebo skriptem createdb, jak bylo popsáno v kapitole Začínáme s PostgreSQL.

V PostgreSQL se v každé nové databázi automaticky vytvoří schéma **public**. Všechny databázové objekty, pokud nevytvoříte a neurčíte jiné schéma, se vytvoří automaticky v tomto schématu. Všimněte si sloupečku Schéma v náseldujícím příkladu.

```
rimmer1=> \dt zamestnanci
                Seznam relací
 Schéma |    Jméno    |  Typ   | Vlastník
--------+-------------+--------+----------
 public | zamestnanci | tabulka | petr
(1 řádka)
```

Přesněji řečeno, vytvoří se v schématu, které je první v proměnné **search_path**. Defaultně je tam nastaveno nejdříve schéma stejné jako je vaše uživatelské jméno a za ním public. Protože jsem ale nikdy nevytvořil schéma schodného jména jako je mé uživatelské jméno (petr), postgres toto jméno ignoruje a použije první existující schéma – public.

K zobrazení hodnoty proměnné slouží v Postgresu příkaz SHOW, k nastavení příkaz SET. Postgres má spousty a spousty proměnných, search_path je jen jedna z nich. Proměnným se budu ještě věnovat v kapitole o konfiguraci.

```
SHOW search_path;
  search_path
----------------
 "$user",public
(1 řádka)
```

Změnit search_path můžete pomocí příkazu SET:

```
SET search_path TO schema1, schema2, ... ;
```

Objekty můžte **kvalifikovat** pomocí jména schématu (můžete přesně určit, z jakého schématu vás objekt zajímá, nebo v jakém ho chcete vytvořit). Dělá se to podobně jako se kvalifikují sloupce tabulkami – pomocí tečkové notace.

Následující dva příkazy dělají totéž (předpokládám že schéma public je první existující schéma v search_path).

```
CREATE TABLE operator ( id SERIAL PRIMARY KEY NOT NULL, jmeno VARCHAR(20) NOT NULL UNIQUE);
CREATE TABLE public.operator ( id SERIAL PRIMARY KEY NOT NULL, jmeno VARCHAR(20) NOTNULL UNIQUE);
```

Pokud by nebylo schéma public první existující schéma, například by existovalo schéma petr, tedy stejného jména jako mé uživatelské jméno, pak by první příkaz vytvořil tabulku operator v schématu petr a ne ve schématu public.

První uvedené (a existující) schéma v search_path je tzv. **běžné schéma**. Používá se všude tam, kde nekvalifikujete objekty pomocí schéma.

V Postgresu můžete ještě kvalifikovat schéma pomocí jména databáze. Předchozí příkaz by se dal napsat takto:

```
CREATE TABLE rimmer1.public.operator ( id SERIAL PRIMARY KEY NOT NULL, jmeno VARCHAR(20) NOT NULL UNIQUE);
```

Ovšem pozor. V Postgresu není možné pracovat s jinou databází, než ke které jste připojeni (takže kvalifikace pomocí jména databáze je na prd):

```
rimmer1=> CREATE TABLE rimmer2.public.operator (id
SERIAL PRIMARY KEY NOT NULL, jmeno VARCHAR(20) NOT NULL UNIQUE);
ERROR:  cross-database references are not implemented: "rimmer2.public.operator"
```

Tabulky z různých schémat můžete bez problémů JOINovat, provádět INSERT INTO … SELECT, vytvářet mezi nimi reference atp.

Vytvoření a smazání schéma

Schéma z aktuální databáze zobrazíte metapříkazem \dn.

```
rimmer1=> \dn
  Seznam schémat
  Jméno  | Vlastník
---------+----------
 public  | postgres
(1 řádka)
```

Vytvoření nového schéma pomocí CREATE SCHEMA:

```
rimmer1=> CREATE SCHEMA petr;
rimmer1=> \dn
  Seznam schémat
  Jméno  | Vlastník
---------+----------
 petr    | petr
 public  | postgres
(2 řádky)
```

Všimněte si, co se stane, když vytvoříte tabulku stejného jména, ale v jiném schématu:

```
rimmer1=> CREATE TABLE petr.operator ( id
SERIAL PRIMARY KEY NOT NULL, jmeno VARCHAR(20) NOT NULL UNIQUE);
NOTICE:  CREATE TABLE will create implicit sequence "operator_id_seq" for serial column "operator.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "operator_pkey" for table "operator"
NOTICE:  CREATE TABLE / UNIQUE will create implicit index "operator_jmeno_key" for table "operator"
CREATE TABLE
rimmer1=> \dt operator
                Seznam relací
 Schéma |  Jméno   |   Typ   | Vlastník
--------+----------+---------+----------
  petr  | operator | tabulka | petr

rimmer1=> SELECT * FROM operator;
 id | jmeno
----+-------
(0 řádek)

rimmer1=> SELECT * FROM public.operator;
 id |    jmeno
----+-------------
  1 | ?
  2 | Pevná linka
  3 | O2
  4 | T-Mobil
  5 | Vodafone
  6 | U:fon
(6 řádek)
```

Metapříkaz \dt zobrazuje jen tabulky, které můžete použít bez kvalifikace schématem. Tváří se, jako by (nová prázdná) tabulka operator existovala jen ve schématu petr.

Tabulka petr ale dál existuje ve schématu public. Všechny relace mezi touto tabulkou a ostatními DB objekty (tabulkami a sekvencemi) zůstávají nezměněné (nová tabulka petr nepřebírá žádné relace).

Schéma smažete příkazem DROP SCHEMA. Pokud mazané schéma obsahuje nějaké databázové objekty, musí se použít ještě CASCADE.

```
rimmer1=> DROP SCHEMA petr;
ERROR:  cannot drop schema petr because other objects depend on it
DETAIL:  table operator depends on schema petr
DOPORUČENÍ:  Use DROP ... CASCADE to drop the dependent objects too.
rimmer1=> DROP SCHEMA petr CASCADE;
NOTICE:  drop cascades to table operator
DROP SCHEMA
```

Informační schéma

Informační schéma (information_schema) je **standardní** schéma, které vám poskytuje metadata o databázi (metadata = data o datech). Protože se jedná o standard, měl by toto schéma implementovat každý DBMS (stejně).

O tom, že existuje, se můžete přesvědčit metapříkazem \dnS.

Tabulky v informačním schématu jsou obvykle implementovány jako pohledy (views). Můžete si znich přečíst (vyselectovat) různé informace o databázi.

Například, když se chcete podívat na to, jaké máte v databázi tabulky, můžete to udělat třeba takto:

```
rimmer1=> SELECT table_catalog, table_schema, table_name, table_type
FROM information_schema.tables
WHERE table_schema = 'public' and table_name like 'zam%';
 table_catalog | table_schema | table_name  | table_type
---------------+--------------+-------------+------------
 rimmer1       | public       | zamestnanci | BASE TABLE
(1 řádka)
```

To je odpověď na to, jak obejít omezení metapříkazu \dt, který zobrazí z tabulek stejného jména jen tu z prvního schématu, kterou najde.

Popis všech tabulek z informačního schéma najdete v dokumentaci The Information Schema. Spousta metapříkazů nedělá nic jiného, než že se podívá do tabulek informačního schématu a zobrazí z nich požadované informace.

PG katalog schéma

Zatímco Informational schéma je standardní pohled na data, v systémovém katalogu (pg_catalog) jsou (meta)data skutečně uložena. Jedná se o regulérní tabulky, které můžete updatovat, ale většinou byste to neměli dělat. Tyto tabulky jsou updatovány automaticky, když je to potřeba (například CREATE TABLEautomaticky vloží záznam do tabulky pg_tables).

Seznam tabulek a jejich popis najdete v dokumentaci System Catalogs.

Cluster (DBMS)

Cluster je to, co vám běží na počítači – serverová část postgresu. Ta se stará o všechny databáze, která máte na svém počítači. Pokud máte více počítačů, na kterých vám běží DBMS, dá se zařídit, aby spolu komunikovali. Jeden DBMS může být používán např. pro zálohu, tj. může udržovat kopii všech dat z primárního CLUSTERu. Konfigurace clusterů je už ale nad rámec tohoto kurzu.

MySQL/MariaDB

V MySQL nejsou schéma implementované. Databázové objekty patří konkrétní databázi. Fakticky se ovšem databáze chová spíš jako schéma – můžete například přistupovat k jiným tabulkám z jiné databáze, než ke které jste připojeni.

Schizofrenii MySQL dokládá i to, že CREATE DATABASE a CRAETE SCHEMA dělají to samé – vytvoří databázi.

V MySQL neexistuje nic jako search_path. Běžné schéma je to, ke kterému jste připojeni. Běžné schéma, tedy vlastně databázi, můžete změnit příkazem USE jmeno_database.

```
USE rimmer1;
```

Aktuální databázi můžete zjistit pomocí funkce DATABASE(), respektive SCHEMA() (oboje vrací název běžného schématu, tedy databáze :-).

```
mysql> SELECT SCHEMA(), DATABASE();
+----------+------------+
| SCHEMA() | DATABASE() |
+----------+------------+
| rimmer1  | rimmer1    |
+----------+------------+
1 row in set (0.00 sec)
```

V MySQL najdete databázi **information_schema**, jejíž význam je stejný jako v PostgreSQL. Dále databázi **mysql**, jejíž význam odpovídá (zhruba) významu schématu pg_catalog z Postgresu.

SQLite

SQLite nepodporuje schémata. Jeden soubor – jedna databáze a hotovo.
Můžete ale pomocí příkazu ATTACH DATABASE 'filename' AS dbname připojit další databázi (dalšímu souboru).
S tabulkami z takto připojené databáze pracujete pomocí kvalifkátoru dbname.
Běžné schéma je první otevřená databáze. Kvalifikátor pro běžné schéma je main.

```
$ sqlite3 rimmer1.db3
sqlite> CREATE TABLE t1(id INTEGER PRIMARY KEY AUTOINCREMENT);
sqlite> CREATE TABLE main.t2(t1_id INTEGER REFERENCES t1(id));
sqlite> insert into main.t1 values(1);
sqlite> insert into main.t1 values(2);
sqlite> insert into t2 values (1);
sqlite> ATTACH DATABASE 'rimmer2.db3' AS rimmer2;
sqlite> CREATE TABLE rimmer2.t2(t1_id integer references main.t1(id));
Error: near ".": syntax error
sqlite> CREATE TABLE rimmer2.t2 AS SELECT * FROM t2;
sqlite> SELECT * FROM rimmer2.t2;
t1_id
----------
1
sqlite> DROP TABLE rimmer2.t2;
```

Na příkladu můžete vidět, že není možné vytvořit reference mezi dvěmi databázemi. (Error se týká main.t1).

Oracle

V Oracle každé schéma vlastní jeden DB uživatel. Schéma nelze měnit, ani neexistuje žádná search_pathproměnná. Jiná schémata můžete oslovovat (pouze) kvalifikovaně.
DBMS oracle spravuje jen jednu (nepojmenovanou) „databázi", která obsahuje schémata všech uživatelů dané instance DBMS. Schéma se vytvoří se založením uživatele a má stejné jméno jako uživatel. Toto schéma je pro uživatele běžné schéma, takže se objekty z něj nemusí kvalifikovat jménem schématu.
Oracle neimplementuje information_schema. Místo toho můžete najít metadata v tabulkách a pohledech (nejen) ze schématu SYS. Na tyto tabulky existují v každém schématu synonyma (podívejte se na Public Synonyms v Oracle SQL Developeru), takže je můžete používat bez kvalifikace.

```
SELECT * FROM user_tables;
SELECT * FROM SYS.user_tables;
```

Metadata jsou uložena v tzv. **data dictionary view**. Seznam všech těchto pohledů získáte následujícím příkazem:

```
oracle> SELECT * FROM dict;
TABLE_NAME          COMMENTS
------------------- ------------------------------------------------------------
USER_CONS_COLUMNS   Information about accessible columns in constraint definitions
USER_TABLES         Description of the user's own relational tables
...
```

Normalizace

Nebojte se, nebudeme se bavit o násilném potlačení Pražského jara 1968 armádami Varšavské smlouvy. Normalizace databáze je proces, při kterém se tabulky převedou do takového tvaru, aby vyhověli tzv. **normálním formám**. Normální formy jsou pravidla, jak správně navrhnout tabulky, aby se zamezilo různým anomáliím při spravování databáze, jak o jsou třeba duplicity, nebo nekonzistentní data. Zní to hrozně vědecky, ale vlastně jsou to jen takové „jednoduché" návody, jak si ulehčit život.

- Normální formy
  - 1NF
  - 2NF
  - 3NF
  - 3.5NF (BCNF)
  - 4NF
  - 5NF

Normální formy

Normalizace databáze patří čistě do databázové teorie. Pro tentokrát tedy můžete zapomenout na PostgreSQL, MySQL či jiné DBMS. Vlastně můžete zapomenout na celé SQL, normalizace se týká čistě návrhu tabulek.
Existuje 6 normálních forem (NF), které jsou očíslované od 1 do 5. Boyce-Coddova normální forma (BCNF) má číslo 3.5 (protože je hodně podobná 3NF).
Platí, že aby byla tabulka v nějaké NF, musí být i ve všech předešlích NF (tak to bylo definováno a tak to prostě je).
Nejdůležitější jsou první 3NF, možná ještě ta 3.5, takže se budu věnovat hlavně jejich vysvětlování.

Návrh telefonní databáze

Pokud by vás NF zaujaly natolik, že byste se chtěli dozvědět i jejich přesné matematické definice, historii, nebo něco více o 4NF a 5NF, doporučuji se podívat na anglickou wikipedii.
NF budu vysvětlovat na návrhu databáze z kapitoli o telefonní databázi, viz diagram.

1NF

Atributy obsahují pouze atomické hodnoty a každá hodnota obsahuje pouze jednu hodnotu.

Atributem se, zjednodušeně řečeno, myslí sloupeček (Dle teorie sloupeček obsahuje atribut objektu, který je reprezentován tabulkou).
1NF je na pochopení asi nejjednodušší. V příkladu ji porušuje tabulka kontakt. Sloupeček adresa totiž neobsahuje atomickou (dále nedělitelnou) hodnotu – adresu můžete rozdělit na město, městksou část, ulici, číslo ulice a PSČ (našli by se i další věci, co by se k adrese mohli hodit, ale o to tu teď nejde).

Telefonní databáze 1NF

Ačkoliv na takové to domácí použití ničemu nevadí, že mám adresu jako jeden sloupeček, přináší porušení 1NF několik ošklivých praktických problémů. Jak třeba zjistíte, kolik lidí pochází z Prahy 1? Mohli byste zkusit použít operátor LIKE, nebo regulární výrazy, ale výsledek by byl nejistý (tak to může být v adrese napsáno cokoliv a jakkoliv). A taky by to bylo dost pomalý.
Řešení je vidět na obrázku – sloupec adresa jsem rozdělil na
sloupce adresa_mesto, adresa_mestska_cast, adresa_ulice,adresa_ulice_cislo, adresa_psc.
Pravda je, že původní návrh (s jedním sloupečkem adresa) by se vám pro soukromé účely asi používal snadněji. Čas od času i profesionálové porušují normálni formy při návrhu databáze, ale většinou se to krutě nevyplatí. **Pokud chcete porušit NF, musíte k tomu mít sakra dobrý důvod a dokázat si to obhájit**.
Ta část NF, která se zmiňuje o pouze jedné hodnotě, znamená, že byste neměli například do sloupečkufirma_nazev vkládat názvy dvou firem (oddělené třeba středníkem), pokud někdo náhodou tak moc pracuje. Ono to k tomuto řešení svádí, když v návrhu nepočítáte s tím, že by někdo mohl pracovat ve více firmách a dodatečně přemýšlíte, jak to vyřešit …). Problém je stejný jako v předchozím příkladě – jak vyhledat lidi z jedné firmy? Jak odebrat ze sloupečku jednu firmu, ale další tam nechat?
Místo toho byste měli vytvořit extra tabulku pro firmy a odkazovat se z této tabulky na kontakt. Problém vyřešen.

2NF

Tabulka (relace) musí být v 1NF a každý neklíčový atribut tabulky musí být plně závislý na celém primárním klíči.

2NF, ač se to třeba nezdá, je vlastně taky jednoduché pravidlo. Už víte, že PK se může skládat z více sloupečků. Proto se v 2N hovoří o *celém primárním klíči*. Neklíčový atribut je prostě sloupeček, který není součástí PK. Nedá se rovnou napsat, že jde o sloupeček, který není PK právě proto, že PK se může skládat z více sloupečků (proto zní 2NF hrozně učeně).
V příkladu porušuje 2NF tabulka telefon. Primární klíč se skládá z předvolby a telefonu. Neklíčový atribut operator_id je ovšem závislý pouze na předvolbě. Pro každou stejnou předvolbu musí být stejný operátor (to je ta závislost).*
** Tento příklad byl psán v době, kdy nebylo možné odejít se stejným číslem k jinému operátorovi. Dnes už můžete mít telefony se stejnou předvolbou ale s jiným operátorem. Pro teď prosím předstírejme, že to tak není a že je operátor závislý na předvolbě (a vice versa).*
Tato závislost může vést k akutalizační anomálii – můžete změnit operátora u některých řádků s určitou předvolbou, ale u jiných řádků ne.
Ostatní atributy jsou už závislé na celém PK. Předvolba, ani zbytek čísla (telefon) vám nestačí k tomu, abyste určili, kterému kontaktu, nebo kterému tarifu telefon patří.
Řešením pouršení 2NF je vytvoření extra tabulky pro závislost mezi částí PK a závislým atributem. V příkladu už taková tabulka je – predvolba je relací mezi předvolbou a operátorem. Takže vlastně stačí sloupeček operator_id z tabulky telefon smazat a je vystaráno.

Telefonní databáze 2NF

Teď ale zpět do reality. Závislost mezi předvolbou a operátorem už neexistuje. Takže co s tím? Sloupeček operator_id v tabulce telefon ponechám, abych si mohl udržovat informaci o tom, který operátor dané číslo obsluhuje. Takovou informaci ale nebudu mít asi vždycky k dispozici, takže si ho změním na nepovinný.
Zajímavější je to s relací predvolba. Tato relace měla říkat, jaká předvolba patří jakému operátorovi. Protože tato relace už neexistuje, zdá se tabulka predvolba zbytečná. Protože existují jen určitá čísla jako předvolba, mohl bych si ponechat tabulkupredvolba pouze se sloupečkem predvolba.
Díky odkazování se z tabulky telefon do tabulky predvolba bude zajištěno, že se do tabulky telefon nedostane nějaká neexistující předvolba.
Ale vlastně, proč si neudržovat informaci o tom, kterému operátorovi byly původně předvolby přiřazeny? Takže si ponechám i sloupeček operator_id, jen už nebudu tvrdit, že se jedná o operátora, který obsluhuje všechny telefony s těmito předvolbami, ale o operátora, který vydává telefony s těmito předvolbami (a obsluhuje většinu z nich). Hlavní je umět si to okecat :-).

3NF

Tabulka (relace) musí být v 2NF a všechny neklíčové atributy vzájemně nezávislé.

Protože je tabulka v 2NF, všechny atributy jsou závislé na celém PK. Může se ovšem stát, že jsou závislé mezi sebou a tím porušují 3NF. V Příkladu to porušují sloupce firma_nazev a firma_poznamka z tabulky kontakt. Firma je závislá na PK tabulky, protože nás zajímá název firmy, ve které daný kontakt pracuje – na základě kontaktu (člověka) se dá **jednoznačně** říct, jaká firma má v sloupečku firma_nazev být.
Stejně tak to platí, ač to nemusí být úplně zřejmé, i pro firma_poznamka. Stačí mi znát kontakt a vím, co by mělo být v poznámce – tak nějak díky tomu, že vím, v jaké firmě kontakt pracuje. A ejhle, máme tu nějakou další závislost. Ze sloupečku firma_nazev můžu jednoznačeně říct, co má být ve sloupečku firma_poznamka.
Tohle samozřejmě platí, pokud se poznámka o firmě skutečně týká firmy. Pokud byste do poznámky psali třeba na jaké pozici kontakt pracuje, pak už ze sloupce firma_nazev nepoznáte, co by mělo být v sloupci firma_poznamka …

Telefonní databáze 3NF (pokus 1)

3NF také řeší problém akutalizační anomálie – mohli byste mít různé řádky se stejnou firmou, ale jinou poznámkou –. Což, jak jsem vysvětlil v poznámce výše, nechcete. kdybyste to chtěli, nejednalo by se o porušení 3NF.
Obdobný problém je s adresou. Třeba městská část závisí na městě.
Problém s 3NF se řeší vytvořením extra tabulky. Vytvořím tabulky pro firmu a pro adresu a z tabulky kontakt se na ně budu odkazovat pomocí cizího klíče. Řešení můžete vidět na obrázku.
Všiměte si vedlejšího efektu normalizace. Přibyly nám tabulky, ale hlavně máme v databázi více integritních omezení, což zlepšuje konzistenci dat. Například jsem mohl určit, že když už zadávám firmu, požaduji její název (v předchozím návrhu jste mohli zadat poznámku k firmě, aniž byste zadali název firmy). Adresa má přesně daný formát a navíc vyžaduji, aby obsahovala alespoň město a ulici. Pořád ale platí, že ke kontaktu adresu ani firmu mít nemusím (adresa_id i firma_id jsou nepovinné).

Telefonní databáze 3NF (pokus 2)

Další tabulka, která nesplňuje 3NF, je tabulka telefon. Sloupec operator_id se totiž dá odvodit ze sloupce tarif_id. Přes tarif_id se dostanete do tabulky tarif, kde je operator_id, který by měl být stejný, jako v tabulce telefon (nemůžete přece k telefonu přiřadit tarif, který se vztahuje na jiného operátora).
Nejjednodušší řešení by bylo odstranit sloupec tarif_id z tabulky telefon a zjišťovat id tarifu přes tabulku tarif. Jenomže odkaz do tabulky tarif není povinný. Kam byste pak uložili informaci o operátorovi, když nebudete znát tarif?
Další možností je nepoužívat umělý klíč tarif_id, ale místo něho přidat do tabulky telefon tarif_nazev. Odkazem do tabulky tarif by pak byla dvojice operator_id a tarif_nazev. Jestli si vzpomínáte, tak tato dvojice je v tabulce tarif unikátní – mohla by sloužit jako primární klíč místo uměle vytvořeného PK id. Takto se vám už nebude moci stát, že byste updatovali tarif_id v tabulce telefon, ale pořád jste se odkazovali na tarif pro jiného operátora. DBMS vám dokonce umožní vložit jen operator_id a tarif_nazev nechat NULL, ikdyž tvoří oba sloupce cizí klíč. Takže krásný sluníčkový den.
Blbý je, že tabulka může mít jen jeden PK, takže každá další tabulka, co se bude chtít odkazovat na tabulku tarif, bude muset používat tuto dvojici. Je na zvážení (stejně jako u tabulky adresa), zda se to prakticky vyplatí. Možná by stálo za úvahu vrátit se k prvnímu řešení. Udělat tarif povinný s tím, že pro každého operátora vložíte do tabulky tarif „neznámý tarif". Cena u takového tarifu by byla NULL.
Všimněte si, že s tabulkou predvolba tento problém není. V tabulce predvolba znamená operator_id něco jiného než operator_id v tabulce telefon, takže je úplně vpořádku, když budou mít různou hodnotu.

3.5NF (BCNF)

Tabulka (relace) musí být v 3NF a všechny determinanty funkční závislosti jsou zároveň kandidátním klíčem.

Je tabulka adresa skutečně v 3NF? Je a není. Už číslo ulice porušuje 1NF, protože by mělo správně být rozdělené na číslo popisné a číslo orientační. Pro zjednodušení to teď ale zanedbám. Dělám si soukromou databázi, ne databázi pro poštovní úřad, takže mě takto rozdělené číslo nebude nikdy zajímat.
2NF tabulka neporušuje (teda porušuje, protože číslo ulice porušuje 1NF, ale víte co chci říct).
Nejzajímavější je problém 3NF. Totiž, nedá se náhodou z městské části jednoznačně říct, v jakém jsme městě? A z ulice v jaké jsme městské části?
Z názvu městské části Praha 1 snadno poznáte, že jste v Praze. Kdybychom si u městské části zaznamenávali i GPS souřadnice, tak i pro stejně se jmenující městské části z různých měst můžeme jednoznačně říct, do jakého města patří (obdobně to platí i pro ulice). Zůstaňme ale při tom, že názvy ulic a městských částí nejsou jedinečné, takže z nich nepoznáme, kam patří. 3NF tak neporušujeme.
Co ale PSČ? Z atributů (město, ulice) lze PSČ zjistit. Je tu tedy závislost **(město, ulice) → PSČ**.
Zdálo by se, že se porušuje 3NF, protože PSČ není součástí PK a je závislý na něčem, co není PK. Jenže 3NF říká, že všechny **neklíčové** atributy musí být vzájemě nezávislé. PSČ je ale klíčový atribut (je částí kandidátního klíče).
Tabulka adresa má následující **kandidátní** klíče (tj. atributy, nebo skupiny atributů, které by mohli sloužit jako PK. Tabulka může mít více kandidátních klíčů, ze kterých se pak jeden libovolně vybere jako PK):
(město, ulice, cislo_ulice) → (mestska_cast, psc)
(PSČ, ulice, cislo_ulice) → (mestska_cast, město)
V tabulce jsou tedy 2 kandidátní klíče – 2 skupiny atributů, které budou v tabulce vždy unikátní (zapomeňte teď na umělý klíč id). Všechny atributy, které jsou součástí některého z kandidátních klíčů, jsou **klíčové atributy**. Není tedy možné, aby porušili 3NF.
A od toho je tu právě BCNF, která je trochu silnější variantou 3NF. Rozdíl mezi BCNF a 3NF je natolik formální, že si lidi často myslí, že tabulka porušuje 3NF, když ve skutečnosti porušuje BCNF.
Determinanty funkční závislosti v tabulce adresa jsou:
(mesto, ulice, cislo_ulice) → (mestska_cast, psc)
(PSC, ulice, cislo_ulice) → (mestska_cast, město)
(město, ulice) → (PSČ)
(PSC) → (město)
To co je na levé straně **determinuje** (jednoznačně určuje) to, co je na pravé straně. Ovšem poslední dvě závislosti porušují BCNF – determinanty nejsou kandidátním klíčem. Tyto dvojice nejsou pro tabulku adresaunikátní. Nejde ovšem o porušení 3NF, protože PSČ i město nejsou neklíčové atributy.

Telefonní databáze v BCNF

Řešením je, jak jinak, rozdělením tabulky adresa do více tabulek. Tabulka pošta bude obsahovat závislost **(PSC)** → (město), v tabulce adresa zbytek. Viz diagram.
Teď už nemůže dojít k anomálii, kdy byste měli stejné PSČ pro různá města. Ale musel jsem změnit v tabulce adresa psc na povinný atribut. Je otázka k zamyšlení, zda to stojí za to. A jestli taková anomálie natolik nebezpečná, že se vyplatí přidávat další tabulku. (Obvykle ano :-).
Pokud umíte trochu anglicky, doporučuju se podívat ještě na příklad na anglické wikipedii o Boyce–Codd normal form. Najdete tam další jednoduchý příklad a taky se dozvíte, že ne vždy lze beztrestně dosáhnout BCNF.

4NF

Tabulka (relace) musí být v BCNF a musí obsahovat pouze příčinné souvislosti (mezi klíčem a atributy).

Příklad a vysvětlení najdete na anglické wikipedii Fourth normal form.

5NF

Tabulku (relaci) již není možno bezestrátově rozložit.

Transakce

Databáze jsou obvykle navržené pro práci více uživatelů. Stává se proto, že se snaží několik uživatelů naráz editovat stejná data. Jak se s tím vyrovnat? O tom pojednává tato kapitola (teoreticky). Jak na to prakticky popisuje kaptiola Transakce prakticky.
Pochopením transakcí vaše znalosti o DBMS poskočí o další levely nahoru (a přitom je to tak jednoduché :-)).

- Transakce
- Začátek a konec transakce
- Souběh transakcí
  - Řešení souběhu transakcí
- ACID

Transakce

Transakce, zjednodušeně řečeno, je povel nebo sada povelů, které se buď provedou všechny a celé, nebo se neprovede vůbec nic. Transakce je nedělitelná (atomická) jednotka.

Každý jednotlivý SQL příkaz je transakcí. Tj. buď se provede celý, nebo nic. Toto platí, pokud jsou transakce podporovány. MyISAM tabulka z MySQL transakce nepodporuje, takže vám toto nezajišťuje.

Když spustíte nějaký příkaz, například UPDATE, musí se provést celý nebo nic. To znamená, že když například updatujete 10 řádek, musí se změnit všech 10, nebo žádná. A to, i kdyby průběh příkazu přerušil výpadek proudu. DBMS si při provádění loguje změny (transakční protokol) a v případě, že dojde k nějaké havárii, je schopný z logu vrátit nedokončený příkaz (zruší všechny nepotvrzené operace).

Pokud jde o transakce s více SQL příkazy, představte si následující problém: Do banky přijdou dva požadavky na stržení peněz z účtu jednoho klienta. První požadavek si načte zůstatek klienta (a zjistí, že peněz má dost). O milisekundu později si načte zůstatek druhý požadavek. První požadavek odečte částku z načteného zůstatku a uloží ji. Totéž udělá o milisekundu i druhý požadavek. Výsledkem je, že je uložený výsledek druhého požadavku a stržení částky prvního požadavku zmizí. Tak takhle ne!

Nebo jiný příklad: chcete převést peníze z jednoho účtu na druhý. Nejdříve peníze z prvního účtu smažete. Pak se chystáte je připsat na druhý účet, ale v tom dojde k nějaké chybě a už se vám to nepodaří. Peníze nenávratně zmizely. Tak tak by to taky nešlo. Buď se musí provést obojí (smazání z prvního účtu a připsání na druhý), nebo nic!

Začátek a konec transakce

Každá transakce má svůj začátek a konec. Transakce končí svým potvrzením (COMMITem).

Pokud pracuje databáze v tzv. **auto commit** režimu, pak je každý jednotlivý příkaz transakcí. Jeho spuštěním transakce začne a jeho dokončením se transakce automaticky ukončí (UPDATE začne updatováním prvního řádku a skončí updatem posledního řádku.)

Zajímavější je možnost vytvořit transakci z více příkazů. Prakticky se to dělá zahájením transakce příkazem **BEGIN**, pak spuštěním všech SQL příkazů, které mají být v rámci transakce (které se mají buď všechny provést, nebo žádný) a nakonec potvrzením transakce příkazem **COMMIT**.

Když nejste v **auto commit** režimu, pak (dle ANSI/ISO normy) transakce začne prvním SQL příkazem (po přihlášení k databázi) a automaticky se potvrdí (COMMITne) při bezchybném ukončení programu (když ho ukončíte sami, neukončí se nějakou chybou nebo výpadkem proudu atd.).

Když databáze selže, provede se automaticky zrušení celé transakce (ROLLBACK). Mimo to můžete samozřejmě potvrdit transakci kdykoliv příkazem COMMIT. (Následujícím SQL příkazem po COMMITu začíná nová transakce …).

ANSI/ISO norma po DBMS vyžaduje, aby byli součástí transakce všechny SQL příkazy. Realita je obvykle taková, že se **v DBMS potvrzuje transakce jakýmkoliv SQL příkazem, který mění strukturu databáze (CREATE TABLE, ALTER TABLE …)**. (Takže nejen COMMITem). Důvodem je zřejmě velká náročnost, kterou by splnění požadavku ANSI/ISO normy vyžadovalo.

Souběh transakcí

Problémy, které mohou nastat při práci více uživatelů se stejnými daty lze rozdělit do následujících kategorií:

1. Problém ztracené aktualizace

Aktualizace jsou provedené bez vzájemného ohledu. V takovém případě platí poslední provedený UPDATE (DELETE …). Jde třeba o problém, který jsem popisoval v příkaldu se dvěma požadavky na stržení peněz z účtu jednoho klienta.

2. Problém nepotvrzených dat

Jde o tzv. nečisté čtení (dirty reading). Jde o problém, kdy si někdo přečte změny v datech, které jste ještě nepotvrdili (necommitli). Vy potom změny nepotvrdíte, ale ten kdo si je přečet je má za platné a pracuje s nimi.

3. Problém nekonzistentních dat

Jde o tzv. neopakovatelné čtení (nonrepeatable read). Během transakce provedete nějaký SELECT. Pak dál něco provádíte a mezi tím někdo provede (a commitne) změnu v databázi. Vy pak provedete (během jedné transakce) stejný SELECT a on vám vrátí jiný výsledek.

4. Problém přízračných dat

Aneb problém vložení přeludu (phantom read). Jde o podobný problém, jako u neopakovatelného čtení. Týká se však SELECTů s agregačními funkcemi.

Problém nekonzistentních datj lze technicky ošetřit například tak, že si DBMS pamatuje obsah řádků, na které jste se (v transakci) už ptali (které četl). Dotaz na stejný řádek vrátí stejné data.

Pokud se zeptáte na SUMu řádků z nějaké tabulky, DBMS si zapamatuje hodnoty v řádcích, ze kterých dělal SUMu. Když ale někdo mimo transakci vloží nový řádek a vy znovu provedete SUMu, bude tento nový řádek součástí výsledku – stejný SELECT vrátí jinou hodnotu.

Řešení souběhu transakcí

Izolace transakcí

- v průběhu transakce je pohled na DB zcela konzistentní
- v transakci nejsou vidět nepotvrzené změny jiných uživatelů
- v transakci nejsou vidět potvrzené změny jiných uživatelů (potvrzené v průběhu izolované transakce)

Serializace transakcí

- transakce A, B jsou spuštěny souběžně
- DBMS zajistí, že výsledky budou stejné, jako by transakce A proběhla první a za ní transakce B (NEBO nejdřív B a pak A). To znamená, že druhá transakce v pořadí musí ovykle čekat na dokončení té první

Odstínění transakcí

- skrytí nepotvrzených transakcí
- skrytí transakcí potvrzených po zahájení aktuální transakce

Úrovně izolací definovaných standardem

Následující tabulka ukazuje úrovně izolací definovaných standardem a jaké konflikty daná úroveň povoluje:

	1.	2.	3.	4.
READ_UNCOMITED	NE	ANO	ANO	ANO
READ_COMITED	NE	NE	ANO	ANO
REPEATABLE_READ	NE	NE	NE	ANO
SERIALIZABLE	NE	NE	NE	NE

Všimněte si, že žádná z úrovní izolací nedovoluje přepsat data dvěma souběžnými transakcemi. Důsledkem je, že pokud se jakýkoliv program v jakékoliv úrovni izolace pokusí přepsat/smazat data již přepsaná/smazaná, dojde k odvolání transakce (celá se zruší).

Vždycky musíte počítat s tím, že každá transakce může být odvolána! Když budete psát nějaký program, musíte takovou situaci ošetřit a pokusit se transakci provést znovu, nebo zobrazit uživeteli chybovou hlášku atd.

ACID

ACID je zkratka pro vlastnosti, které by měla databázová transakce splňovat:

- A – Atomicity – buď se provede celá transakce, nebo nic.
- C – Consistency – před a po dokončení transakce není porušeno žádné integritní omezení a data jsou konzistentní.
- I – Isolation – výsledky příkazů uvnitř transakce jsou pro okolí neviditelné.
- D – Durability – trvalost – pokud byla transakce potvrzena, pak jsou změny dat skutečně uložené a už nemohou být ztracené.

To by bylo k teorii transakcí asi tak všechno. V kapitole Transakce prakticky se dozvíte, jak se to všechno dělá v Postgresu (a dalších DBMS) prakticky. Tato kapitola ale není teoretická, takže je zařazena až v sekci pro pokročilé. Pokud jste ale nedočkaví, můžete na tu kapitolu přeskočit hned.

Entity-relationship diagram

Při návrhu tabulek vám pomůže si rozkreslit, jaká data potřebujete uložit a v jakém jsou vzájemném vztahu. K tomu se většinou používá tzv. ERD diagram.

ERD je velice jednoduchý a oblíbený nástroj pro popis objektů a jejich vztahů. Nemůžete tvrdit, že jste databázový specialista a přitom neumět nakreslit jednoduchý ERD diagram :-).

O ERD

ERD (Entity-relationship diagram) je **konceptuální model**. Konceptuální model popisuje data, která jsou potřeba pro váš byznis, pomáhá vám (a vašemu klientovi) popsat, jaká data a v jakém vztahu budete potřebovat. V ERD diagramu by měli být pouze ta data, která vás skutečně zajímají – data, která budete ukládat, měnit, zobrazovat a mazat.

Konceptuální model by měl být zcela nezávyslý na tom, jak budou data fyzicky uložená. Jestli v PostgreSQL databázi, v MySQL databázi, v textovém souboru, nebo je budete tesat do kamene. Naproti tomu **fyzický model** už popisuje jak budou data uložena, například jaký použijete datový typ (například v PostgreSQL pro text použijete VARCHAR, v Oralce VARCHAR2 atd.).

Diagram tabulek

ERD diagram vypadá podobně jako diagram na obrázku v pravo. V této kapitole se dozvíte, jak má vypadat opravdový ERD diagram (a proč obrázek v pravo není úplný ERD diagram).

Neexistuje žádný mezinárodně uznávaný standard, jak ERD diagrami kreslit, takže se můžete setkat s mnoha „dialekty", tedy s různými grafickými prvky pro vyjádření stejné myšlenky (různými konvencemi, chcete-li). Já zde budu popisovat Oracle konvence pro zápis ERD. Nicméně, rozdíly mezi ERD diagrami jsou většinou zanedbatelné (někdo používá obdélníky s kulatými rohy, někdo se zaoblenými atp.). Nakonec si stejně budete čmárat ERD diagrami nejčastěji na papír podle svého, takže co :-).

Základní pojmy

Entita popisuje nějaký objekt z reálného světa. Popisuje, z čeho se objekt skládá (jaké jeho vlastnosti nás zajímají). Pokud umíte OO programování, můžete si entitu představit jako třídu. Z entity se stane ve fyzickém modelu tabulka.

Instance entity je už popis skutečného objektu popsaného entitou. Instance se stane ve fyzickém modelu řádkem tabulky. (V OO programování je instanci třídy objekt).

Atribut entity popisuje jednu vlasnost entity. Entita se obvykle zkládá z několika atributů. Ve fyzickém modelu se stane z atributu sloupec.

Identifikátor (UID) je atribut, nebo skupina atributů, které jednoznačně identifikují instanci entity (řádek v tabulce). Z identifikátorů se stane v databázi primární/unikátní klíč.

Optionality (volitelnost) je vlastnost atributu, která říká, zda je atribut povinný (musí mít hodnotu) nebo ne (může být NULL). Optionality se také týká relací. Musí se řádek odkazovat na jiný řádek v tabulce (musí existovat relace), nebo nemusí? (Například položka faktury se určitě musí odkazovat na fakturu.)

Kardinalita říká, v jakém množství se něco vyskytuje. Jestli se vyskytuje právě jednou, nebo se může vyskytovat vícekrát. (Jestli se vyskytuje alespoň jednou se dozvíte z Optionality). Kardinalita se vztahuje na relace mezi entitami – můžete se (dává smysl) na řádek tabulky odkazovat jen jedním řádkem, nebo mnoha řádky?

Konvence pro kreslení ERD

Entity

Entity se kreslí do obdélníků se zakulacenými rohy. Jméno entity je v jednotném čísle a napsané velkými písmenami.

ERD - Entity

Atributy

Atributy jsou vypsány pod názvem entity malými písmeny. Mezi atributy nepatří cizí klíč – ten se týká fyzického modelu a v některých implementacích ERD modelu (třeba v textovém souboru) se cizí klíč vůbec nemusí použít. Zato umělé klíče (id), které budete používat, do ERD diagramu patří.

Umělé klíče, pokud je použijete, bude váš business používat pro identifikaci instancí (řádek v tabulce). Cizí klíče bude využívat váš program, ale uživatelé programu o nich nebudou (v ideálním případě) nic vědět. Cizí klíč je jen technikálie.

ERD - Atributy

Optionality atirbutů a UID

Povinné atributy se označují hvězdičkou *, nepovinné kroužkem o. Atributy, ze kterých se skládá unikátní klíč (vzpomeňte si, že unikátní klíč se může skládat z více jak jednoho sloupečku) se označují hash tagem #.
Pokud má tabulka více jak jeden unikátní klíč, přidává se za # číslo, kterým se jednotlivé klíče odlišují. Takže můžete v entitě například 2x # (unikátní klíč zkládající se ze dvou atirbutů), nebo #1 a 2x #2 (dva unikátní klíče, druhý se skládá ze dvou atirbutů) atd.

ERD - Optionality

Z ERD diagramu nahoře vyčtete, že entita FIRMA má dva unikátní klíče (ale nevyčtete, který bude primárním klíčem – to vás v konceptuálním modelu nezajímá).
Zajímavá je taky entita TARIF. Tabulka tarif má jako primární klíč dvojici sloupců operator_id a nazev. Jenže cizý klíče se v konceptuálním modelu neuvádí. Evidentně nám tu chybí důležitá informace. O tom, jak se tato informace do ERD modelu zapíše, se dozvíte o pár odstavců dále.

Relace

Relace mezi entitami se zobrazují tak, že se entity spojí čárou. Čára může být plná, čárkovaná, nebo napůl plná a čárkovaná. Tím se vyjadřuje volitelnost (optionality) relace – zda relace musí nebo nemusí existovat.
Volitelnost relace si můžete představit tak, že v sloupečku cizýho klíče může být null.

ERD - Optionality relací

ERD - Relace

Z diagramu teď můžete vyčíst například to, že nemůže existovat tarif bez odkazu na operátora.To se dá zařídit tím, že cizí klíč v tabulce tarif do tabulky operator bude povinný (NOT NULL). Operátor bez odkazu na tarif existovat může.
Dále ERD diagram vyjadřuje podmínku, že by neměla existovat adresa, na kterou se neodkazuje žádný kontakt (stejně tak firma nemůže existovat bez kontaktu). Databáze by taky neměla obsahovat poštu, ke které neexistuje adresa. To je trochu diskutabilní podmínka, ale já, jakožto zákazník, nechci mít v databázi „smetí".
Všimněte si, že ne všechny z těchto podmínek je možné vynutit pomocí databázových omezení (constraints). Například, co se týče relace adresa - kontakt, tak ta bude implementována cizím klíčem v tabulce kontakt (který bude nepovinný). Na jednu adresu se může odkazovat více kontaktů, takže v tabulce adresa cizí klíč být nemůže – to by se adresa mohla odkazovat jen na jeden kontakt. (To, že se kontakt může odkazovat jen na jednu adresu, berte jako součást zadání klienta.) No a protože v tabulce adresa není cizí klíč, není možné k čemu dát omezení NOT NULL. A tak se může stát, že bude v databázi adresa, ke které nebude existovat kontakt.

Kardinalita

Kardinalita relace může nabývat jedné tří možností:

1:1

Jedné entitě A může odpovídat jen jedna entita B a naopak.

1:M

Jedné entitě A může odpovídat více jak jedna entita B, entita B odpovídá jen jedné entitě A.

M:M

Entitě A může odpovídat více jak jedna entita B, entitě B může odpovídat více jak jedna entita A.
Standardně se používají opravdu jen tyto vztahy. I když víte, že entitě A odpovídají právě 3 entity B, neoznčuje se kardinalita 1:3, ale stále 1:M. Ne že by existoval nějaký zákon, který by vám to zakazoval. Vztah 1:M se v databázi řeší tak, že tabulka pro entitu B má cizí klíč odkazující na entitu A. Neexistuje žádný standardní způsob jak zajistit, že budete mít v tabulce právě (nebo maximálně) 3 řádky odkazující se do tabulky entity A. Možná proto se používá 1:M (což je ale trochu zvláštní vysvětlení, když je ERD konceptuální model a o nějakou databázovou implementaci by se vůbec neměl starat, že :-).
Entita, která se může vyskytnout v relaci M-krát má na konci čáry vyznačující relaci „vydličku".

ERD - Kardinalita relací

Aplikováno na příklad vypadá ERD diagram takto:

ERD - Kardinalita

Všechny vztahy v ERD diagramu mají kardinalitu 1:M.

Slabé entitní typy

Teď je ta správná chvíle vyrovnat se s entitou TARIF. Tarif by měl mít unikátní název, ale jen v rámci operátora. Proto by měl být součástí primárního klíče nejen název, ale i odkaz na operátora. To zatím z ERD diagramu není vidět.
Protože k určení jedinečné instance tarifu je zapotřebí znát operátora, ke kterému tarif patří, označuje se entita TARIF jako **slabý entitní typ**.
To, že je součástí unikátního identifikátoru odkaz na jinu entitu se označuje přeškrtnutím relace. Takto označená relace se anglický nazývá barred relationship (nevím jak to rozumě přeložit do češtiny).
Tarif ale není jediný slabý entitní typ v příkladu. Druhou takovou entitou je TELEFON. Když se na diagram podíváte, všimnete si, že součástí primárního klíče je odkaz na předvolbu. Protože se nejedná o umělý klíč, nemusí to být na první pohled zřejmé, ale předvolba je také odkaz, který by v ERD diagramu neměl být v seznamu atributů. Předvolba se teď zobrazuje v ERD diagramu duplicitně, což je nežádoucí. Řešením je odstranit atribut předvolba z entity TELEFON a z relace telefon - předvolba udělat „barred relationship".

ERD - Barred relationship

Teď už to vypadá, že ERD diagram obsahuje všechny potřebné informace. Když si budete kreslit ERD sami pro sebe, většinou skončíte s takovýmto diagramem. Něco důležitého ale přeci jen chybí. A nejen o tom bude řeč v další kapitole.

Sekvecne a indexy

V této kapitole se dozvíte něco praktického o databázovém objektu **sequence** a o ještě důležitějších databázových objektech – indexech. O indexech byla již řeč v kapitole o integritních omezeních, takže teď je ta správná chvíle si zopakovat jejich význam.

Sekvence

O sekvencích jsem psal při popisování typu SERIAL. Jde o databázový objekt, který vám generuje nějakou posloupnost čísel. Jméno sekvence je většinou tvořeno jménem tabulky, podtržítkem, jmenénem sloupce, podtržítkem a slovem*seq*. Takto se jména sekvencí tvoří při vytváření tabulky automaticky. Je to docela chytré, protože může existovat jen jedna tabulka stejného jména a v ní jen jeden sloupec stejného jména, nikdy nedojde ke kolizi jmen. Pokud vytváříte sekvneci sami, můžeme jí dát jméno téměř libovolné (bez mezer, české diakritiky atp.).

Vytvářet sekvneci se hodí například ve chvíli, kdy chcete z typu INTGETER vytvořit typ SERIAL (pokud vytváříte tabulku se sloupcem typu SERIAL rovnou, vytvoří se sekvence automaticky). Taky se to hodí, když chcete používat jednu sekvenci pro více tabulek (třeba když chcete mít jednu sekvenci primárních klíčů pro všechny tabulky v databázi. Někdo to tak má rád, protože pak jen podle ID poznáte libovolný záznam ze všech tabulek).

A teď rovnou k příkladu:

```
rimmer1=> CREATE SEQUENCE mutant_c_seq;
CREATE
rimmer1=> CREATE TABLE mutant (slovo VARCHAR(10), c1 INTEGER, c2 INTEGER);
CREATE
```

V příkladu jsem schválně vytvořil nejdřív sekvenci, aby bylo jasné, že je to objekt, který není nijak s tabulkou či sloupcem svázán. Proto si musíte dát pozor při rušení tabulky, která nějakou sekvneci využívá, abyste z databáze smazali i sekvenci (pokud jí teda už nebudete potřebovat), Jinak vám bude v databázi zůstávat „smetí".

~~Zkuste si vytvořit tabulku s typem SERIAL, poté tabulku zrušit (sekvneci ponechat) a stejným příkazem jako poprvé tabulku vytvořit. Kvůli existující sekvneci se to nepodaří.~~

To co jsem psal v předchozím odstavci už neplatí. Automaticky vytvořené sekvence při vytvoření tabulky se už při smazání tabulky automaticky mažou (v nových verzích Postgresu).

Podívejte se, co se stane, když přiřadíte sekvenci dvěma sloupcům. (Změna defaultní hodnoty viz další kapitola ALTER TABLE - DEFAULT).

```
rimmer1=> ALTER TABLE mutant ALTER c1 SET DEFAULT nextval('"mutant_c_seq"'::text);
ALTER
rimmer1=> ALTER TABLE mutant ALTER c2 SET DEFAULT nextval('mutant_c_seq');
ALTER
rimmer1=> INSERT INTO mutant VALUES ('jedna');
INSERT 17065 1
rimmer1=> INSERT INTO mutant VALUES ('dva');
INSERT 17066 1
rimmer1=> INSERT INTO mutant VALUES ('tri');
INSERT 17067 1
rimmer1=> SELECT * FROM mutant;
 slovo | c1 | c2
-------+----+----
 jedna |  1 |  2
 dva   |  3 |  4
 tri   |  5 |  6
(3 rows)
```

Jak vidíte, při insertování jsem využil defaultních hodnot pro sloupce c1 a c2.

Chtěl bych jen tak na okraj podotknout, že to nezaručuje unikátní hodnoty pro oba sloupce, jak by se mohlo zdát. Unikátní hodnoty zajišťuje jen podmínka UNIQUE, která dokáže zajistit unikátnost jen pro jeden sloupec (nebo kombinaci sloupců), ale ne unikátnost hodnoty v rámci více sloupců (tj. aby se číslo mohlo objevit pouze jednou v rámci obou sloupců).

Při vytváření sekvence můžete nastavit její minimální hodnotu (implicitně 1), maximální hodnotu (implicitně maximální hodnota typu INTEGER) krok (increment), tj. o kolik se bude další hodnota zvyšovat (implicitně 1), počáteční hodnotu (implicitně 1) či zda se má sekvence cyklit (pokud dojde na konec sekvence – k maximální hodnotě, jestli má začít vracet čísla od začátku, nebo chybu).

Jak se to udělá snadno vyčtete z nápovědy, nebo se můžete podítat na online nápovědu k CREATE SEQUENCE.

```
rimmer1=> CREATE SEQUENCE test_cycle_seq MINVALUE 5 MAXVALUE 7 START WITH 6 CYCLE;
CREATE SEQUENCE
rimmer1=> SELECT nextval('test_cycle_seq'), nextval('test_cycle_seq'),
nextval('test_cycle_seq'), nextval('test_cycle_seq');
 nextval | nextval | nextval | nextval
---------+---------+---------+---------
       6 |       7 |       5 |       6
(1 řádka)
```

Smazání sekvence

Sekvnece se zruší pomocí DROP SEQUENCE. Můžete smazat pouze takovou sekvenci, která není využívána žádnou tabulkou.

```
rimmer1=> DROP SEQUENCE mutant_c_seq;
ERROR:  cannot drop sequence mutant_c_seq because other objects depend on it
DETAIL:  default for table mutant column c2 depends on sequence mutant_c_seq
```

DOPORUČENÍ: **Use DROP** ... CASCADE **to drop** the dependent objects too.
rimmer1=> **DROP TABLE** mutant;
DROP TABLE
rimmer1=> **DROP SEQUENCE** mutant_c_seq;
DROP SEQUENCE

Funkce

Funkce nextval není jediná funkce, kterou můžete na sekvenci aplikovat. Existuje ještě pár dalších funkcí, které například umožní nastavit aktuální hodnotu sekvence. Viz Sequence Manipulation Functions.

Indexy

K čemu slouží indexy jsem popisoval v souvislosti s podmínkou UNIQUE. Pokud vytvoříte v tabulce nějaký sloupec a až později se rozhodnete ke sloupci přidat podmínku UNIQUE, uděláte to prostě tak, že vytvoříte jeho „unikátní" index.

Ke jménu indexu se dá říct téměř to samé, jako pro jméno sekvence.

Index neslouží jen k „hlídání" unikátní hodnoty, ale taky k rychlejšímu vyhledávání v sloupci (sloupcích), nebo k rychlejšímu řazení podle zaindexovaného sloupce (sloupů).

Index můžete vytvořit příkazem **CREATE INDEX**. Na úplný syntax příkazu CREATE INDEX se můžete podívat pomocí metapříkazu \h CREATE INDEX.

Název indexu musí být v rámci databázového schématu unikátní.

Unikátní indexy

Příklad vytvoření (unikátního) indexu:
rimmer1=> **CREATE TABLE** cisla (jedna **INTEGER**, dva **FLOAT**);
CREATE
rimmer1=> **INSERT INTO** cisla **VALUES** (1,1);
rimmer1=> **INSERT INTO** cisla **VALUES** (1,2);
rimmer1=> **CREATE UNIQUE INDEX** cisla_jedna_key **ON** cisla (jedna);
ERROR: Cannot **create unique index**. **Table** contains non-**unique values**
rimmer1=> **CREATE UNIQUE INDEX** cisla_dva_key **ON** cisla (dva);
CREATE INDEX
rimmer1=> **INSERT INTO** cisla(jedna) **VALUES** (1);
INSERT 0 1
rimmer1=> **INSERT INTO** cisla (dva) **VALUES** (2);
ERROR: duplicate **key value** violates **unique constraint** "cisla_dva_key"
DETAIL: **Key** (dva)=(2) already **exists**.

První pokus o vytvoření unikátního klíče nezdařil, protože sloupec jedna neobsahoval unikátní hodnoty. To jste asi pochopili z chybového hlášení.

Sloupec dva po vytvoření unikátního indexu již nedovoluje vložit dvě stejné hodnoty.

Unikátní index umožňuje hlídat unikátnost řádků v několika sloupcích. Jinak řečeno, zabrání vložení řádku který by měl ve (všech) vybraných sloupcích stejné hodnoty jako jiný řádek. Vybranými sloupci myslím ty sloupce, které byli vybrány při vytváření indexu.

rimmer1=> **CREATE TABLE** vektor (jmeno **CHAR**(10), i **INTEGER**, j **INTEGER**);
CREATE
rimmer1=> **CREATE UNIQUE INDEX** vektor_key **ON** vektor(i, j);
CREATE
rimmer1=> **INSERT INTO** vektor **VALUES** ('jedna', 3, 1);
rimmer1=> **INSERT INTO** vektor **VALUES** ('dva', 0, 5);
rimmer1=> **INSERT INTO** vektor **VALUES** ('dva', 1, 0);
rimmer1=> **INSERT INTO** vektor **VALUES** ('dva', 1, 0);
ERROR: duplicate **key value** violates **unique constraint** "vektor_key"
DETAIL: **Key** (i, j)=(1, 0) already **exists**.

Pokud byste chtěli, aby, kromě jiného, sloupec i nemohl obsahovat dvě stejné čísla (hodnoty v něm byly unikátní), mohli byste navíc vytvořit unikátní index jen pro tento sloupec.

Zrychlení selektů

Psal jsem, že indexy pomáhají urychlovat vyhledávání a třídění dat. K tomu pomáhají jak unikátní indexy (bonus k unikátním indexům zdarma), tak „obyčejné" indexy (vytvořené bez klíčového slova UNIQUE). Otázka zní, proč se nevytvoří pro každý sloupeček index automaticky?

Protože indexy něco stojí. Sice ne peníze, ale určitě místo na disku a pak hlavně čas aktualizace tabulky. Kdykoliv totiž vložíte do tabulky řádek, smažete ho nebo upravíte, musí se upravit každý index, kterého se změna týká. A to je poměrně náročná operace. Proto vytvářejte indexy jen tam, kde je budete potřebovat.

Použitím indexů se opravdu zrychlí SELECTy velmi výrazně. Proto na vytváření indexů nezapomínejte! Vytvářejte indexy nad sloupečky, které se objevují v klauzuli WHERE, ORDER BY a ON (pokud už nejsou vytvořené implicitní díky primárním a cizím klíčům).

Pokud už vytvoříte unikátní index, nemusíte vytvářet „obyčejný" neunikátní index. Ten unikátní vám poslouží pro zrychlení také. Další index by byla pro postgres jen práce navíc.

Řekněme například, že budu často SELECTovat řádky z tabulky vektor podle hodnoty sloupce jmeno. Pokud bych v tuto chvíli udělal takový SELECT, bude muset DBMS projít celou tabulku a zkontrolovat každý řádek, jestli je požadovaná hodnota v sloupci jmeno. Index si ale udržuje setříděné hodnoty sloupce (nad kterým je vytvořený). Najít něco v setříděných datech už je otázka chvilky, proto se díky indexu SELECT obvykle mnohonásobně zrychlí.

rimmer1=> **CREATE INDEX** vektor_jmeno_ix **ON** vektor(jmeno);
CREATE INDEX
rimmer1=> **SELECT** * **FROM** vektor **WHERE** jmeno = 'jedna';
```
 jmeno  | i | j
--------+---+---
 jedna  | 3 | 1
(1 řádka)
```

Výsledek SELECTu nijak nepřekvapil. Ani nijak nepoznáte, jestli se index skutečně použil, nebo ne. Poznat to můžete snad jen z toho, jak rychle se vám vrátí výsledek. A to ještě jen u tabulek, které obsahují hodně dat. U tabulky, kde moc dat naní, se může Postgres rozhodnout index nepoužít, protože koukání „stranou" by v takovém případě byla jen práce navíc. Stejně tak, pokud

selektujete většinu dat z tabulky, může si Postgres usmyslet nepoužít indexy, protože přečtení celé tabulky bez koukání na indexy bude rychlejší. Chytrej je ten Postgres, moc chytrej. (Jiné databáze samozřejmě taky :-)
V tabulce vektor existuje už unikátní index nad sloupečky i a j. Postgres umí tento index použít i při prohledávání jednoho sloupce, takže by bylo zbytečné vytvářet index nad sloupcem i nebo j.
Stejně jako urychlují indexy SELECTy, tak mohou urychlovat i příkazy UPDATE a DELETE, pokud se v nich objeví klauzule WHERE, která může index využít.

Indexy nad výrazy (expression)

Index můžete vytvářet nejen nad sloupcem (nebo sloupci), ale i nad výrazy. Do indexu se pak ukládají seřazené hodnoty výsledků výrazu. Aby to mělo cenu, musí být výsledek výrazu neměnný (immutable). To znamená, že pro stejné zadané hodnoty do výrazu musí výraz vrátit vždy stejné výsledky. Ve výrazu nemůžete pracovat s náhodnými čísly, s časem, s hodnotami z extra tabulky atp.
Výraz by měl být uzavřený v závorkách. Proto jsou v následujícím příkladu závorky 2x.

```
CREATE INDEX vektor_sum ON vektor ((i + j));
```

Od této chvíle, když použijete v SELECTu výraz i+j (tentokrát už nemusí být v závorkách), použije Postgres index vektor_sum.
Ovšem pozor. Výraz musí být stejný jako při vytváření indexu. Jakákoliv malinká odchylka může způsobit (a způsobí), že se index nepoužije.

```
SELECT * FROM vektor WHERE i+j   = 3333; -- index se použije
SELECT * FROM vektor WHERE j+i   = 3333; -- index se nepoužije
SELECT * FROM vektor WHERE i+j+1 = 3333; -- index se nepoužije
SELECT * FROM vektor WHERE i+j = 3333-1; -- index se použije
```

Smazání indexů

K mazání indexů slouží příkaz **DROP INDEX**.

```
rimmer1=> DROP INDEX vektor_sum;
```

V Postgresu musí být jméno indexu unikátní pro celou databázi, proto stačí zadat jen jeho jméno.
V Postgresu jde použít s DROP INDEX i iF EXISTS, což zajistí, že pokud náhodou index zadaného jména neexistuje, nevyhodí se žádná chyba, která by mohla třeba zastavit provádění skriptu.

```
rimmer1=> DROP INDEX IF EXISTS vektor_sum;
NOTICE:  index "vektor_sum" does not exist, skipping
DROP INDEX
```

EXPLAIN

O tom, jestli se použijí indexy nebo ne, rozhoduje „plánovač". Ten naplánuje jak se SELECT skutečně provede na základě SQL dotazu, struktury tabulky a indexů a obsahu tabulky.
Příkaz **EXPLAIN** vám ukáže, jak si to plánovač naplánoval.
V následujících příkladech předpokládám, že je vytvořená tabulka vektor, unikátní index nad sloupečky i a jvektor_key, index vektor_jmeno_ix nad sloupečkem jmeno. Naopak (zatím) neexistuje index vektor_sumnad výrazem (i+j).
Podívejte se, jak vypadá výstup z **EXPLAIN** pro SELECT všech hodnot z tabulky vektor.

```
rimmer1=> EXPLAIN SELECT * FROM vektor;
                        QUERY PLAN
-----------------------------------------------------------
Seq Scan on vektor  (cost=0.00..1.03 rows=3 width=52)
(1 řádka)
```

První co se dočtete je, že se provede sekvenčí čtení tabulky vektor. Sekvenční čtení znamená, že se bude číst každý řádek tabulky, jeden za druhým. Indexy se vůbec nepoužijí. Taky proč, chcete přece vypsat všechny data z tabulky, ani je nijak neřadíte, tak k čemu indexy?
V závorce pak vidíte **cost**. To určuje jak dlouho (zhruba – je to jen odhad) bude příkaz trvat. První číslo říká, jak dlouho to bude trvat, než bude možné něco začít vracet na výstup a druhé číslo celkový čas příkazu.
Protože SELECT v příkladu nemusí nic JOINovat ani třídit, může rovnou začít posílat výstup. Proto je první číslo 0. Druhé číslo udává, celkový čas. Jednotkou tohoto čísla ale není vteřina, ani nic podobného. Celková doba běhu příkazu bude nějakým násobkem tohoto čísla, ale těžko říct jakým. To záleží na nastavení Postgresu, na prostředí ve kterém běží (operační systém, souborvý systém atp.), na hardware atd. Číslo můžete použít k porovnávání různých selektů, který bude rychlejší, jinak vám moc neřekne.
Následuje **rows**, který odhaduje počet řádků, který se vrátí. Poslední je **width**, který odhaduje, kolik bajtů bude mít průměrný vrácený řádek.
Teď zkusím EXPLAIN na SELECTu, který by měl využít index vektor_jmeno_ix.

```
rimmer1=> EXPLAIN SELECT * FROM vektor WHERE jmeno = 'dva';
                        QUERY PLAN
-----------------------------------------------------------
Seq Scan on vektor  (cost=0.00..1.04 rows=1 width=52)
  Filter: (jmeno = 'dva'::bpchar)
(2 řádky)
rimmer1=>  SELECT * FROM vektor WHERE jmeno = 'dva';
 jmeno  | i | j
--------+---+---
 dva    | 0 | 5
 dva    | 1 | 0
(2 řádky)
```

A ejhle, on se index zase nepoužívá. To proto, že plánovač plánuje i na základě obsahu tabulky. A protože tabulka obsahuje málo řádek, nevyplatilo by se koukat bokem na index. Takže sekvenční čtení je pořád nejlepší volba. Všimněte si, že plánovač špatně odhadl počet řádek, který příkaz SELECT vrátí. Odhaduje, že se vrátí 1, ale ve skutečnosti se vrátí dva. Možná, že když si to budete zkoušet u sebe, tak se strefí. Je to jen odhad.
Abyste si mohli EXPLAIN vyzkoušet, připravil jsem pro vás dávkový soubor s 111000 INSERTy do tabulky vektor vektor.sql.zip.
Je to velký soubor, takže jsem jej zazipoval a obsahuje jen příkaz pro smazání všech řádků tabulky vektor a pak INSERTy. Neobashuje CREATE TABLE ani vytváření INDEXů. Chtěl jsem, aby byl skript kompatibilní se všemi DBMS, které tu probírám. Nechci tak velký soubor nabízet v několika verzích, aby ste mi nezahltili server :-).
Před použitím tohoto dávkového souboru doporučuji smazat a znovu vytvořit tabulku vektor – tím se smažou i indexy. Indexy znovu vytvořte až po nahrání dat. Je výrazně rychlejší vytvořit index až nad naplněnou tabulkou, než při každém INSERTu index

aktualizovat. Při 111000 řádcích už to pocítíte.
Během nahrávání skriptu se můžete připojit k databázi a zkusit si příkaz SELECT count(*) from vektor; Uvidíte, jak daleko s nahráváním jste.

```
DROP TABLE vektor;
CREATE TABLE vektor (jmeno CHAR(10), i INTEGER, j INTEGER);
-- nahrani dat z vektor.sql
CREATE UNIQUE INDEX vektor_key ON vektor(i, j);
CREATE INDEX vektor_jmeno_ix ON vektor(jmeno);
```

A teď zkusím EXPLAIN znovu.

```
rimmer1=> EXPLAIN SELECT * FROM vektor WHERE jmeno = 'dva';
                                QUERY PLAN
---------------------------------------------------------------------------
Index Scan using vektor_jmeno_ix on vektor  (cost=0.00..8.34 rows=4 width=19)
  Index Cond: (jmeno = 'dva'::bpchar)
(2 řádky)

rimmer1=>  SELECT * FROM vektor WHERE jmeno = 'dva';
  jmeno    | i | j
-----------+---+---
 dva       | 0 | 5
 dva       | 1 | 0
(2 řádky)
```

No vida, tentokrát už se použije index vektor_jmeno_ix. Odhad výsledného počtu řádků je zase špatně, ale na to, že je v tabulce 111000 řádků je docela blízko :-).

Teď zkusím další EXPLAIN. Připomínám, že neexistuje index vektor_sum.

```
rimmer1=> EXPLAIN SELECT * FROM vektor WHERE i + j BETWEEN 50000 and 500000;
                                QUERY PLAN
---------------------------------------------------------------
Seq Scan on vektor  (cost=0.00..2929.00 rows=555 width=19)
  Filter: (((i + j) >= 50000) AND ((i + j) <= 500000))
(2 řádky)

rimmer1=> SELECT * FROM vektor WHERE i + j BETWEEN 50000 and 500000;
  jmeno    |   i    |   j
-----------+--------+--------
 test      | 349191 |  57865
 test      |  20151 | 222418
 test      |   2246 | 323118
 test      | 122626 | 156366
 test      | 401753 |   2168
 test      | 155143 | 258160
 test      |  95297 | 303726
 test      |  93011 | 200915
 test      |   9690 | 119219
(9 řádek)
```

Odhad celkové doby je 2929 (něčeho). To je docela dost :-). Odhadovaný počet vrácených řádek je 555. A skutečnost? Pouze 9.

Teď vytvořím index pro výraz (i+j) a zkusím EXPLAIN ještě jednou:

```
rimmer1=> CREATE INDEX vektor_sum ON vektor ((i + j));
rimmer1=> explain SELECT * FROM vektor WHERE i + j BETWEEN 50000 and 500000;
                                QUERY PLAN
---------------------------------------------------------------------------
Bitmap Heap Scan on vektor  (cost=13.96..723.10 rows=555 width=19)
  Recheck Cond: (((i + j) >= 50000) AND ((i + j) <= 500000))
->  Bitmap Index Scan on vektor_sum  (cost=0.00..13.82 rows=555 width=0)
      Index Cond: (((i + j) >= 50000) AND ((i + j) <= 500000))
(4 řádky)
```

Tentokrát se plán rozdělil do dvou kroků. Poslením krokem je *Bitmap Heap Scan*. Jeho součástí je Bitmap Index Scan. Všimněte si, že *Bitmap Heap Scan* začíná až po index scanu, takže jeho první *cost* je o malilinko větší než druhá *cost* index scanu. Index Scan se použije na vyfiltrování řádků a odhaduje se, že jich vrátí 555. Bitmap Heap Scan pak už jen řádky vypíše, žádný neodfiltruje, takže odhad počtu řádků je pořád 555. Index scan neposílá svůj výsledek na výstup. Až skončí, předá svůj výsledek Bitmap Heap Scanu.

```
rimmer1=> explain SELECT * FROM vektor WHERE i + j BETWEEN 50000 and 500000 ORDER BYi;
                                QUERY PLAN
---------------------------------------------------------------------------
Sort  (cost=748.39..749.78 rows=555 width=19)
  Sort Key: i
->  Bitmap Heap Scan on vektor  (cost=13.96..723.10 rows=555 width=19)
      Recheck Cond: (((i + j) >= 50000) AND ((i + j) <= 500000))
->  Bitmap Index Scan on vektor_sum  (cost=0.00..13.82 rows=555 width=0)
      Index Cond: (((i + j) >= 50000) AND ((i + j) <= 500000))
(6 řádek)
```

Výsledek tohoto příkladu je stejný jako toho předchozího, jen se jako poslední provede sortování podle sloupce i. (Sortování se provede na výsledku Bitmap Heap Scan, který se provede na výsledku Bitmap Index Scan.)

```
rimmer1=> explain SELECT * FROM vektor ORDER BY i DESC;
                                QUERY PLAN
---------------------------------------------------------------------------
Index Scan Backward using vektor_key on vektor  (cost=0.00..5733.26 rows=111000 width=19)
(1 řádka)
```

A tohle už je poslední příklad. Vidíte, že se vrátí všechny řádky tabulky, že to bude trvat dlouho a že se použije pro třídění index vektor_key, který se bude číst pozpátku (protože je ORDER v sestupném pořadí).
Pokud vás EXPLAIN nadchnul, podívejte se do dokumentace na Using EXPLAIN, kde se dozvíte více podrobností, třeba něco o EXPLAIN ANALYZE.

MySQL

MySQL nemá sekvence. Můžete používat AUTO_INCREMENT, ale tomu nemůžete nastavit vlastnosti jako INCREMENT BY, CYCLE či MAXVALUE. Největším omezením je nemožnost použití sekvence na více jak jednu tabulku. Existuje způsob, jak si vytvořit vlastní funkci nextval, ale to je zatím nad rámec toho, co jsem vás tu naučil.
MySQL neumí vytvořit index nad výrazem. Jediný zatím dostupný způsob jak to obejít je vytvořit extra sloupeček, kam si uložíte výsledky výrazu a ten zaindexuje. Sice tím porušíte 3NF, ale co už.
V MySQL jsou indexy součástí tabulky, takže můžete mít více indexů stejného jména, pokud se vztahují na jiné tabulky. Důsledkem toho je, že když index mažete, musíte říct nejen jeho jméno, ale i jméno tabulky.

```
mysql> DROP INDEX cisla_dva_key ON cisla;
```

EXPLAIN

Výstup tohoto příkladu se v MySQL od Postgresu dost liší.

```
mysql> EXPLAIN SELECT * FROM vektor WHERE jmeno = 'dva';
+----+-------------+--------+------+---------------+-----------------+---------+-------+------+-------------+
| id | select_type | table  | type | possible_keys | key             | key_len | ref   | rows | Extra       |
+----+-------------+--------+------+---------------+-----------------+---------+-------+------+-------------+
|  1 | SIMPLE      | vektor | ref  | vektor_jmeno_ix | vektor_jmeno_ix | 31      | const |    2 | Using where |
+----+-------------+--------+------+---------------+-----------------+---------+-------+------+-------------+
```

Z výstupu můžete vyčíst, že se MySQL může použít vektor_jmeno_ix a také se rozhodl použít index vektor_jmeno_ix. Očekávaný počet **prohledávaných** řádků jsou 2. (Tabulka je před nahráním skriptu vektor.sql.)

```
mysql> explain SELECT * FROM vektor ORDER BY i DESC;
+----+-------------+--------+------+---------------+------+---------+------+------+----------------+
| id | select_type | table  | type | possible_keys | key  | key_len | ref  | rows | Extra          |
+----+-------------+--------+------+---------------+------+---------+------+------+----------------+
|  1 | SIMPLE      | vektor | ALL  | NULL          | NULL | NULL    | NULL |    3 | Using filesort |
+----+-------------+--------+------+---------------+------+---------+------+------+----------------+
mysql> explain SELECT i,j FROM vektor ORDER BY i DESC;
+----+-------------+--------+-------+---------------+------------+---------+------+------+-------------+
| id | select_type | table  | type  | possible_keys | key        | key_len | ref  | rows | Extra       |
+----+-------------+--------+-------+---------------+------------+---------+------+------+-------------+
|  1 | SIMPLE      | vektor | index | NULL          | vektor_key | 10      | NULL |    3 | Using index |
+----+-------------+--------+-------+---------------+------------+---------+------+------+-------------+
```

V dalším příkladu vydíte, že se MySQL se chová oproti Postgresu zase trochu rozdílně. Rozhodl se využít index vektor_key pouze v druhém případě, kdy se tahají z tabulky jen sloupce, které jsou v indexu (v takovém případě se ani nemusí dívat do tabulky, všechna data vysosá z indexu). (Viz ORDER BY Optimization.)
Pro podrobnější vysvětlení výstupu z EXPLAIN se podívejte na EXPLAIN Output Format.

SQLite

SQLite nemá sekvence. Používá jen AUTOINCREMENT. Nevýhody jsou stejné jako u MySQL.
SQLite neumí vytvořit index nad výrazem. Řešit se to dá stejně jako v MySQL.
Indexy v SQLite musí být unikátní v rámci celého schématu, takže jako v Postgresu.
V SQLite se používá pro zobrazení exekučního plánu příkaz **EXPLAIN QUERY PLAN** (existuje i verze bez QUERY PLAN, ale ta zobrazí trochu něco jiného).

```
sqlite> EXPLAIN QUERY PLAN SELECT * FROM vektor WHERE jmeno = 'dva';
selectid    order       from        detail
----------  ----------  ----------  ----------------------------------------------------------------
0           0           0           SEARCH TABLE vektor USING INDEX vektor_jmeno_ix (jmeno=?) (~10 rows)
```

Více viz The EXPLAIN QUERY PLAN Command.

Oracle

Oracle sekvence má, ale nemůže je použít jako defaultní hodnoty. O tom jsem už psal v kapitole o vytváření relací. Existuje ještě další podivnost, oproti Postgresu:

```
oracle> CREATE SEQUENCE mutant_c_seq;
oracle> INSERT INTO mutant VALUES('jedna', mutant_c_seq.NEXTVAL,mutant_c_seq.NEXTVAL);
oracle> INSERT INTO mutant VALUES('dva', mutant_c_seq.NEXTVAL, mutant_c_seq.NEXTVAL);
oracle> INSERT INTO mutant VALUES('tri', mutant_c_seq.NEXTVAL, mutant_c_seq.NEXTVAL);
oracle> SELECT * FROM mutant;
SLOVO   C1      C2
jedna   1       1
dva     2       2
tri     21      21
```

První čeho si všimnete je, že NEXTVAL vrací pořád stejné číslo v rámci jednoho SQL dotazu. To se může někdy hodit a někdy ne. Těžko říct, jestli je lepší chování Postgresu nebo Oracle. Rozhodněte si to sami :-).
Druhá zajímavá věc je, že na třetím řádku sekvence poskočila o cca 20 záznamů. To je způsobené tím, že sekvence v Oracle pužívají defaultně tzv. CACHE pro 20 čísel. Funguje to tak, že se ze sekvence uzme 20 čísel, která se pak jedno za druhým používají. Pak se vezme dalších 20 čísel atd. Příkazy pro insert jsem spouštěl v Apexu, tj. přes webové rozhraní a mezi druhým a třetím příkzem došlo k expiraci sezení, takže se cache smazala a pro třetí příkaz se vytvořilo nové spojení s novým cachováním sekvence. Číslo tak poskočilo až za dvacítku.
Co z toho plyne? Sekvence, kvůli cachování, nejsou bez „děr". Výhodou je, že se čísla generují několikanásobně rychleji. Pokud INSERTujete hodně řádků, kde sekvenci používáte, je to znát.
Pokud nechcete mít sekvence s dírama, uveďte při jejich vytváření NOCACHE. Pokud chcete nastavit vlastní velikost CACHE (čím větší, tím rychlejší, ale taky možnost vzniku větších děr), uveďte CACHE n, kde n je číslo větší než 1.
V Postgresu taky můžete použít CACHE. Jen místo NOCACHE se používá CACHE 1, což je taky v Postgresu defaultní nastavení.

```
oracle> CREATE SEQUENCE test_cycle_seq MINVALUE 5 MAXVALUE 7 START WITH 6 CYCLE NOCACHE;
oracle> SELECT test_cycle_seq.NEXTVAL, test_cycle_seq.NEXTVAL,
```

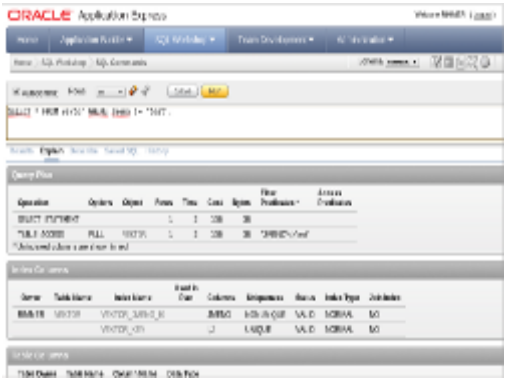
```
test_cycle_seq.NEXTVAL, test_cycle_seq.NEXTVAL from dual;
NEXTVAL NEXTVAL NEXTVAL NEXTVAL
      6       6       6       6
oracle> SELECT test_cycle_seq.NEXTVAL, test_cycle_seq.NEXTVAL,
test_cycle_seq.NEXTVAL, test_cycle_seq.NEXTVAL from dual;
NEXTVAL NEXTVAL NEXTVAL NEXTVAL
      7       7       7       7
-- dál už je to asi jasné
```

Oracle umí vytvořit indexy nad výrazy a index má platnost v rámci databáze, v tom se chová stejně jako Postgres. Jen neumí použít při mazání indexu IF EXISTS. Místo toho to jde nasimulovat takto: (podobně jako při mazání tabulky).

```
BEGIN
EXECUTE IMMEDIATE 'DROP INDEX vektor_sum';
EXCEPTION
WHEN OTHERS THEN
IF SQLCODE != -1418 THEN
RAISE;
END IF;
END;
/
```

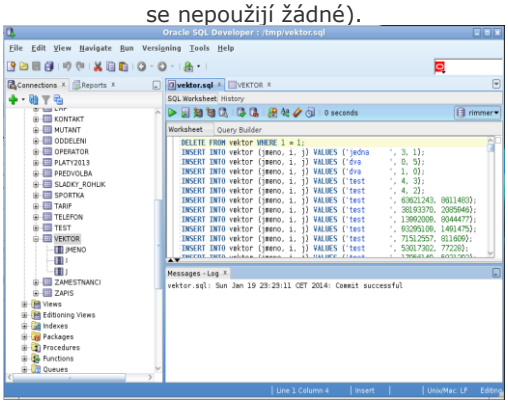
Plánovač dotazu je v Oracle o ždibec chytřejší, takže dokáže použít index vektor_sum jak s výrazem (i+j), tak (j+i), ale s (i+j+1) si už taky neporadí. K tomu, abyste poznali kdy se index bude používat a kdy ne slouží už dobře známý příkaz EXPLAIN.

EXPLAIN PLAN FOR



Explain v Apexu

Exekuční plán (Query Plan) se může zobrazit v Apexu pomocí záložky explain (viz obrázek). Ve výsledku uvidíte 3 tabulky. První popisuje samotný exekuční plán, ve druhém uvidíte indexy které jsou k dispozici a indexy, které se použijí (v příkadu na obrázku se nepoužijí žádné).



Import vektor.sql

Pokud budete chtít nahrát skript vektor.sql (viz výše),tak přes Apex to bohužel nepůjde kvůli jeho velikosti. Kupodivu jej ale můžete bez větších problémů otevřít a spustit v Oracle SQL developeru.

Po spuštění skriptu (to druhé tlačítko se zeleným trojúhelníkem a listem papíru) nezapomeňte kliknout na tlačítko pro potvrzení transakce (tlačítko se zelenou fajfkou), jinak se provedené změny do databáze neuloží.

Pokud chcete použít SQL příkaz, pak se jmenuje EXPLAIN PLAN FOR. Tento příkaz ovšem exekuční plán nevypíše, ale uloží jej do systémové tabulky PLAN_TABLE, ze které ho dostanete příkazem SELECT * FROM TABLE(dbms_xplan.display); (Proč to musí být v Oracle vždycky tak složitý?)

```
oracle> EXPLAIN PLAN FOR SELECT * FROM vektor WHERE jmeno != 'test';

Explained.

oracle> SELECT * FROM TABLE(dbms_xplan.display);

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 3437271467

------------------------------------------------------------------------------
| Id  | Operation         | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
```

```
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |           |    1 |    26 |    2   (0)| 00:00:01 |
|*  1 |  INDEX RANGE SCAN| VEKTOR_KEY |    1 |    26 |    2   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------

   1 - access("I"=4)

Note
-----
   - dynamic sampling used for this statement (level=2)

17 rows selected.


Execution Plan
----------------------------------------------------------
Plan hash value: 2137789089

-------------------------------------------------------------------------------------------
| Id  | Operation                         | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                  |         |  8168 | 16336 |    29   (0)| 00:00:01 |
|   1 |  COLLECTION ITERATOR PICKLER FETCH| DISPLAY |  8168 | 16336 |    29   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Další informace můžete najit na wiki, nebo v dokumentaci.

Úprava tabulky - defaultní hodnoty a sekvence

Tato kapitola navazuje na povídání o ALTER TABLE. Probírá další možnosti tohoto příkazu, ale určitě tím nevyčerpává všechny jeho možnosti, ani se tu nedozvíte nic extra světoborného.

Už ani vlastně nevím, proč tahle kapitola vznikla a proč probírá zrovna tyto varianty příkazu ALTER TABLE. Asi jsem jí napsal v době, kdy jsem tyto varianty potřeboval. Znát ALTER TABLE se vám ale určitě vyplatí.

- Přidání defaultní hodnoty
- Přidání typu serial
- MySQL
- SQLite
- Oracle

Přidání defaultní hodnoty

Defaultní hodnotu sloupce můžete určit už při vytváření tabulky, nebo jí ke sloupci přidat později. Pokud existuje, můžete jí taky smazat pomocí příkazu ALTER TABLE. Jak se to dělá zjistíte v nápovědě k tomuto příkazu.

O vytváření nového sloupce jsem již povídal v kapitole o úpravě tabulky, odstavci o ALTER TABLE. Dozvěděli jste se, jak přidat do tabulky sloupec s datovým typem, ale nic víc.

Ukážu příklad, kde do tabulky dluhy přidám sloupec navrat_dluhu typu DATE jehož defaultní hodnota bude deset dní od aktuálního času (current_date+10). V Postgresu se to dělá tak, že se nejdříve přidá sloupec a teprve poté k němu DEFAULT hodnota.

Udělá se to pomocí ALTER TABLE. Příklad předvedu na tabulce navrat_dluhu, kterou jsem vytvořil v kapitole o Úprava tabulky.

Pro připomenutí vám znovu ukážu SQL pro vytvoření této tabulky:

```
rimmer1=> CREATE TABLE dluhy (
jmeno VARCHAR(10),
prijmeni VARCHAR(15),
dluh NUMERIC(8,1),
zadluzen DATE
);
rimmer1=> INSERT INTO dluhy VALUES ('Linus','Torvalds','3105.0', '2002-09-22');
rimmer1=> INSERT INTO dluhy VALUES ('Martin','Doktor',1590.0,'2002-09-22');
rimmer1=> ALTER TABLE dluhy ADD navrat_dluhu DATE;
rimmer1=> ALTER TABLE dluhy ALTER navrat_dluhu SET DEFAULT current_date+10;
rimmer1=> SELECT * FROM dluhy;
 jmeno  | prijmeni | dluhy  |  zadluzen  | navrat_dluhu
--------+----------+--------+------------+--------------
 Linus  | Torvalds | 3105.0 | 2002-09-22 |
 Martin | Doktor   | 1590.0 | 2002-09-22 |
(2 rows)
```

Jak vidíte, přibyl nový sloupec, který je celý prázdný i přes to, že má defaultní hodnotu. Defaultní hodnota se dosazuje pouze při vkládání nového řádku. Sloupec navrat_dluhu je nutné vyplnit tak, jak jsem to dělal v odstavci o ALTER TABLE.

Sloupec lze přidat do tabulky rovnou s nastavenou defaultní hodnotou. V takovém případě se do sloupece hned defaultní hodnota vloží.

```
rimmer1=> ALTER TABLE DROP COLUMN navrat_dluhu;
rimmer1=> ALTER TABLE dluhy ADD navrat_dluhu DATE DEFAULT current_date+10;
rimmer1=> select * from dluhy;
 jmeno  | prijmeni |  dluh  |  zadluzen  | navrat_dluhu
```

```
--------+----------+--------+------------+-------------
Linus   | Torvalds | 3105.0 | 2002-09-22 | 2014-01-31
Martin | Doktor   | 1590.0 | 2002-09-22 | 2014-01-31
(2 řádky)
```

Přidání typu serial

Pokud mluvím o typu serial, dopouštím se tím jisté nepřesnosti. Serial je ve své podstatě typ INTEGER, který má jako defaultní hodnotu funkci nextval nějaké sekvence a integritní omezení NOT NULL ~~a UNIQUE~~.

```
rimmer1=> CREATE TABLE tabulka (cislo serial);
NOTICE:  CREATE TABLE will create implicit sequence 'tabulka_cislo_seq' for SERIAL column 'tabulka.cislo'
CREATE
rimmer1=> \d tabulka
                Table "tabulka"
 Column |  Type   |                    Modifiers
-------+---------+-------------------------------------------------
 cislo  | integer | not null default nextval('"tabulka_cislo_seq"'::text)
```

Z výpisu tabulky tabulka (jak originální název, že :-) je patrné, že sloupec cislo je typu integer (!) s výše jmenovanými vlastnostmi. Funkce nextval obsahuje jako argument sekvenci tabulka_cislo_seq. Tato sekvence se vytvořila automaticky při vytvoření tabulky.

Pokud chci „sestrojit" typ SERIAL, vytvořím nejdřív sloupec s typem INTEGER a k němu přidám defaultní hodnotu s funkcí nextval. Sekvenci si budu muset vytvořit sám. K tomu slouží příkaz CREATE SEQUENCE. Pak ještě omezím nový sloupec pouze na unikátní hodnoty (dřív byl typ SERIAL automaticky unikátní, ale to už neplatí) pomocí příkazu CREATE INDEX. Nový sloupec bude prázdný, proto jej nakonec vyplním nějakými hodnotami.

```
CREATE TABLE lide (jmeno VARCHAR(10));
INSERT INTO lide VALUES('Petr');
INSERT INTO lide VALUES('Petr');
INSERT INTO lide VALUES('Pavel');
ALTER TABLE lide ADD id INTEGER;
ALTER TABLE lide ALTER id SET DEFAULT nextval('"lide_id_seq"'::text);
CREATE UNIQUE INDEX lide_id_key ON lide (id);
rimmer1=> INSERT INTO lide values ('Martin');
ERROR:  pg_aclcheck: class "lide_id_seq" not found
rimmer1=> CREATE SEQUENCE lide_id_seq;
CREATE SEQUENCE
rimmer1=> INSERT INTO lide(jmeno) values ('Martin');
INSERT 0 1
rimmer1=> SELECT * FROM lide;
 jmeno  | id
--------+----
 Petr   |
 Petr   |
 Pavel  |
 Martin |  1
(4 rows)
```

První insert s Martinem se nepovedl, protože ještě nebyla vytvořená sekvence lide_id_seq, kterou využívá defaultní hodnota pro id.

Jelikož již v řádku se jménem Matin hodnota id je, a já ji z nějakého důvodu nechci změnit, použiji při vyplňování podmínku WHERE.

```
rimmer1=> UPDATE lide SET id = nextval('"lide_id_seq"'::text)
WHERE id IS NULL;
rimmer1=> SELECT id, jmeno FROM lide ORDER BY id;
 id | jmeno
----+--------
  1 | Martin
  2 | Petr
  3 | Petr
  4 | Pavel
(4 řádky)
```

A nakonec přidám NOT NULL podmínku:

```
rimmer1=> ALTER TABLE lide ALTER id SET NOT NULL;
rimmer1=> INSERT INTO lide(id,jmeno) VALUES(NULL,'Ruprt');
ERROR:  null value in column "id" violates not-null constraint
DETAIL:  Failing row contains (Ruprt, null).
```

MySQL

MySQL umí použít jako defaultní hodnotu aktuální čas jen s datovým typem TIMESTAMP (který navíc může být v tabulce jen jednou). Takže pro vytvoření defaultní hodnoty aktuální čas se musí změnit typ sloupce.

```
mysql> ALTER TABLE dluhy ADD navrat_dluhu DATE;
Query OK, 2 rows affected (0.19 sec)
Records: 2  Duplicates: 0  Warnings: 0
mysql> ALTER TABLE dluhy CHANGE navrat_dluhu navrat_dluhu TIMESTAMP DEFAULT CURRENT_TIMESTAMP;
mysql> SELECT * FROM dluhy;
+--------+----------+--------+------------+---------------------+
| jmeno  | prijmeni | dluh   | zadluzen   | navrat_dluhu        |
+--------+----------+--------+------------+---------------------+
| Linus  | Torvalds | 3105.0 | 2002-09-22 | 2014-01-21 12:41:03 |
| Martin | Doktor   | 1590.0 | 2002-09-22 | 2014-01-21 12:41:03 |
```

```
+--------+----------+--------+------------+--------------------+
2 rows in set (0.00 sec)
```

Všimněte si, že je navrat_dluhu vyplněn defaultní hodnotou.

MySQL neumí ALTER TABLE … SET DEFAULT, takže přidání defaultní hodnoty se musí vždy dělat znovuvytvořením definice sloupce, obdobně jako v tomto příkladu. MySQL neumí použít pro defaultní hodnotu výrazy, takže defaultní hodnotu s časem +10 dní nevytvoříte.

MySQL nemá sekvence, takže si v něm ani nevytvoříte „typ" SERIAL. Můžete ovšem dodatečně přidat AUTO_INCREMENT, což jde ovšem jen na primární klíč.

```
mysql> ALTER TABLE lide CHANGE COLUMN id id INTEGER PRIMARY KEY AUTO_INCREMENT NOT NULL;
mysql> INSERT INTO lide(jmeno) values ('Martin');
mysql> SELECT * FROM lide;
+--------+----+
| jmeno  | id |
+--------+----+
| Petr   |  1 |
| Petr   |  2 |
| Pavel  |  3 |
| Martin |  4 |
+--------+----+
4 rows in set (0.00 sec)
```

SQLite

ALTER TABLE toho v SQLite moc neumí. Už je to popsáno v kapitole o úpravě tabulky. Prostě budete muset tabulku vytvořit znova …

SQLite také nezná sekvence, čímž se mi výklad zjednodušuje.

Oracle

Oracle neumí ALTER TABLE … SET DEFAULT. Místo toho se používá toto:

```
oracle> ALTER TABLE dluhy MODIFY(navrat_dluhu DEFAULT CURRENT_TIMESTAMP);
oracle> SELECT * FROM dluhy;
JMENO    PRIJMENI        DLUH    ZADLUZEN        NAVRAT_DLUHU
Linus    Torvalds        3105    09/22/2002      -
Martin   Doktor          1590    09/22/2002      -
```

Toto v Oracle funguje stejně jako v Postgresu:

```
oracle> ALTER TABLE dluhy ADD navrat_dluhu DATE DEFAULT CURRENT_DATE+10;
oracle> SELECT * FROM dluhy;
JMENO    PRIJMENI        DLUH    ZADLUZEN        NAVRAT_DLUHU
Linus    Torvalds        3105    09/22/2002      01/31/2014
Martin   Doktor          1590    09/22/2002      01/31/2014
```

Oracle nemá ani AUTO_INCREMENT, ani neumí použít sekvence jako defaultní hodnoty, čímž se mi výklad zase zjednodušuje :-).

Transakce prakticky

Tato kapitola navazuje na teoretickou kapitolu o transakcích. Zde ukážu několik praktických příkladů, jak transakce začínat, ukončovat a odvolávat, jak nastavit úroveň izolace transakcí atd. Bez znalosti transakcí se žádný opravdový databázista neobejde, tak dávejte dobrý pozor :-).

BEGIN, COMMIT, ROLLBACK

Transakci zahájíte SQL příkazem **BEGIN**. Za BEGIN může následovat ještě klíčové slovo WORK nebo TRANSACTION, ale ty nemají žádný efekt (jsou tam jen kvůli kompatibilitě s jinými DBMS nebo standardem).

Následují SQL příkazy, které se neuloží, dokud nepotvrdíte transakci příkazem **COMMIT** (a zase může následovat WORK nebo TRANSACTION).

Pokud si transakci rozmyslíte (třeba proto, že jste udělali nějaký nechtěný UPDATE), můžete celou transakci odvolat pomocí **ROLLBACK** [WORK|TRANSACTION].

Příklady budu ukazovat na tabulce trest (zkratka *tr*ansakční t*est* :-). A hned do ní vložím nějaká data.

```
CREATE TABLE trest (
    id SERIAL,
    uzivatel VARCHAR(255),
    penize NUMERIC(10,0)
    );
INSERT INTO trest(uzivatel,penize) VALUES
    ('Franta', 30000),
    ('Tonda',  40000),
    ('Pepa',   11000),
    ('Marie',  35000);
```

Fungování transakce si ověříte pomocí dvou spojení do databáze. Abych je od sebe nějak rozlyšil, budou v příkladech tentokrát místo názvu databáze (rimmer1) jména připojení (conn1 a conn2).

```
conn1=> BEGIN WORK;
conn1=>UPDATE trest SET penize = penize - 1000 WHERE uzivatel = 'Franta';
conn2=> SELECT * FROM trest;
 id | uzivatel | penize
----+----------+--------
  1 | Franta   |  30000
  2 | Tonda    |  40000
  3 | Pepa     |  11000
  4 | Marie    |  35000
(4 řádky)
conn1=> UPDATE trest SET penize = penize + 1000 WHERE uzivatel = 'Pepa';
conn1=> SELECT * FROM trest ORDER BY id;
 id | uzivatel | penize
----+----------+--------
  1 | Franta   |  29000
  2 | Tonda    |  40000
  3 | Pepa     |  12000
  4 | Marie    |  35000
(4 řádky)
conn2=> SELECT * FROM trest;
 id | uzivatel | penize
----+----------+--------
  1 | Franta   |  30000
  2 | Tonda    |  40000
  3 | Pepa     |  11000
  4 | Marie    |  35000
(4 řádky)
conn1=> COMMIT WORK;
conn2=> SELECT * FROM trest ORDER BY id;
 id | uzivatel | penize
----+----------+--------
  1 | Franta   |  29000
  2 | Tonda    |  40000
  3 | Pepa     |  12000
  4 | Marie    |  35000
(4 řádky)
```

A teď se podívejte, co se stane, když se dvě transakce pokusí upravit stejný řádek.

```
conn1=> BEGIN;
conn1=> UPDATE trest SET penize = penize - 5000 WHERE uzivatel = 'Marie';
UPDATE 1
conn2=> UPDATE trest SET penize = penize - 5000 WHERE uzivatel = 'Marie';
-- prikaz "zamrzne" a ceka na potvrzeni/odvolani transakce conn1
-- pokud conn1 nepotvrdi/neodvola transakci dost rychle, prikaz po nejake dobe selze
conn1=> COMMIT;
-- ted se teprve provede UPDATE z conn2
conn2=> SELECT * FROM trest WHERE uzivatel = 'Marie';
 id | uzivatel | penize
----+----------+--------
  4 | Marie    |  25000
(1 řádka)
```

Jak vidíte, odečetlo se celkem 10000, žádné „odečtení“ se neztratilo.

Dlouho trvající transakce může způsobit zbytečné selhání jiné transakce. Proto by měli být transakce vždy co nejkratší. Každá transakce musí počítat s tím, že bude odvolána!

Příklad 3:

```
conn1=> BEGIN;
conn1=> SELECT * FROM trest WHERE uzivatel = 'Tonda';
 id | uzivatel | penize
----+----------+--------
  2 | Tonda    |  40000
(1 řádka)
conn2=> UPDATE trest SET penize = '35000' WHERE uzivatel = 'Tonda';
UPDATE 1
conn1=> UPDATE trest SET penize = penize - 1000 WHERE uzivatel = 'Tonda';
UPDATE 1
conn1=> SELECT * FROM trest WHERE uzivatel = 'Tonda';
 id | uzivatel | penize
----+----------+--------
  2 | Tonda    |  34000
(1 řádka)
conn1=> COMMIT;
COMMIT
conn1=> SELECT * FROM trest WHERE uzivatel = 'Tonda';
 id | uzivatel | penize
----+----------+--------
  2 | Tonda    |  34000
```

Postgres databáze má (defaultně) izolační level *READ-COMMITED*. To znamená, že uvnitř transakce se čtou změny provedené (commitnuté) v jiných transakcích. Proto druhý SELECT vrátil 34000 na místo 39000.
Nezapomeňte, že příkazy na změnu struktury databáze (CREATE TABLE, ALTER TABLE atd.) nemohou být součástí transakce. Pokud je provedete a máte zrovna nějakou transakci započatou, tak se automaticky potvrdí (nebo odvolá, když nejde potvrdit).

SAVEPOINT

Příkaz ROLLBACK zruší celou transakci (od jejího počátku). Naštěstí existuje příkaz **SAVEPOINT**, kterým si můžete v transakci označit místa, ke kterým se můžete vrátit pomocí ROLLBACK TO savepoint (takže nemusíte rušit celou transakci).
Místo si označíte libovolně zvoleným jménem (které by nemělo začínat číslem, obsahovat mezery atp.).

```
rimmer1=> SELECT * FROM trest ORDER BY id;
 id | uzivatel | penize
----+----------+--------
  1 | Franta   |  29000
  2 | Tonda    |  34000
  3 | Pepa     |  12000
  4 | Marie    |  25000
(4 řádky)
rimmer1=> BEGIN WORK;
rimmer1=> UPDATE trest SET penize = 30000 WHERE uzivatel = 'Franta';
UPDATE 1
rimmer1=> SAVEPOINT A;
SAVEPOINT
rimmer1=> UPDATE trest SET penize = 30000 WHERE uzivatel = 'Pepa';
UPDATE 1
rimmer1=> SAVEPOINT B;
SAVEPOINT
rimmer1=> UPDATE trest SET penize = 30000 WHERE uzivatel = 'Marie';
UPDATE 1
rimmer1=> ROLLBACK TO A;
ROLLBACK
rimmer1=> UPDATE trest SET penize = 30000 WHERE uzivatel = 'Tonda';
UPDATE 1
rimmer1=> COMMIT;
COMMIT
rimmer1=> SELECT * FROM trest ORDER BY id;
 id | uzivatel | penize
----+----------+--------
  1 | Franta   |  30000
  2 | Tonda    |  30000
  3 | Pepa     |  12000
  4 | Marie    |  25000
(4 řádky)
```

Není možné zrušit jen příkazy mezi dvěma SAVEPOINTy (například zrušit jen příkaz mezi A a B, ale ponechat příkaz za B (před ROLLBACK TO). Vždycky se ruší všechny příkazy od SAVEPOINTu až po poslední provedený příkaz před ROLLBACK TO příkazem.
Používat SAVEPOINT se vyplatí. Pokud v Postgresu uděláte nějakou chybu (stačí překlep v SQL příkazu), tak Postgres celou transakci zneplatní. Vrácením se k poslednímu SAVEPOINTu si tak může zachránit spoustu práce.

```
rimmer1=> BEGIN;
BEGIN
rimmer1=> SELECT * FROM trest WHERE id = 2;
 id | uzivatel | penize
----+----------+--------
  2 | Tonda    |  30000
(1 řádka)

rimmer1=> UPDATE trest SET penize = 15000 WHERE id = 2;
UPDATE 1
rimmer1=> SAVEPOINT A;
SAVEPOINT
rimmer1=> UPDATE trest SET penize = 15000 WHERE id  1;
ERROR:  syntax error at or near "1"
ŘÁDKA 1: UPDATE trest SET penize = 15000 WHERE id  1;
                                                 ^
rimmer1=> UPDATE trest SET penize = 15000 WHERE id = 1;
ERROR:  current transaction is aborted, commands ignored until end of transaction block
rimmer1=> ROLLBACK TO A;
ROLLBACK
rimmer1=> COMMIT;
COMMIT
rimmer1=> SELECT * FROM trest WHERE id = 2;
 id | uzivatel | penize
----+----------+--------
  2 | Tonda    |  15000
(1 řádka)
```

Nastavení
Úrovně izolace transakcí

Úroveň izolace **aktuální transakce** se nastavuje v PostgreSQL pomocí SET TRANSACTION ISOLATION LEVEL. Nastaví se tím úroveň izolace pouze pro začatou transakci. Další transakce bude mít zase nastavenu úroveň podle defaultního natavení, takže, pokud to potřebujete, musíte jí nastavit ISOLATION LEVEL znovu.

SET TRANSACTION ISOLATION LEVEL { SERIALIZABLE | REPEATABLE **READ** | **READ** COMMITTED | **READ** UNCOMMITTED }

Můžete nastavit jednu ze čtyř nabízených možností, ale READ UNCOMMITTED se v Postgresu chová stejně jako READ COMMITTED. Což znamená, že tak nízkou úroveň izolace jako je READ UNCOMMITTED si v Postgresu nevyzkoušíte.

Před verzí 9.1 byla úroveň SERIALIZABLE synonymum pro REPEATABLE READ.

Aktuálně nastavenou úroveň zobrazíte příkazem **SHOW TRANSACTION ISOLATION LEVEL;**.

Zkusím znovu příklad 3, tentokrát s ISOLATION LEVEL REPEATABLE READ:

```
conn1=> UPDATE trest SET penize = 40000 WHERE uzivatel = 'Tonda';
conn1=> BEGIN;
conn1=> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
conn1=> SHOW TRANSACTION ISOLATION LEVEL;
 transaction_isolation
-----------------------
 repeatable read
(1 řádka)
conn1=> SELECT * FROM trest WHERE uzivatel = 'Tonda';
 id | uzivatel | penize
----+----------+--------
  2 | Tonda    |  40000
(1 řádka)
conn2=> UPDATE trest SET penize = '35000' WHERE uzivatel = 'Tonda';
conn2=> UPDATE trest SET penize = '4444' WHERE uzivatel = 'Marie';
conn1=> SELECT * FROM trest WHERE uzivatel IN ('Tonda', 'Marie');
 id | uzivatel | penize
----+----------+--------
  2 | Tonda    |  40000
  4 | Marie    |  25000
(2 řádky)
conn1=> UPDATE trest SET penize = penize - 1000 WHERE uzivatel = 'Tonda';
ERROR:  could not serialize access due to concurrent update
conn1=> COMMIT;
ROLLBACK
```

Tentokrát se update nepovedl a celá transakce byla Postgresem odvolána. Kdybych provedl během transakce nějaké změny, všechny by se vrátili zpět – databáze by zůstala ve stavu před začátkem transakce, tedy v konzistentním stavu.

Všimněte si, že Postgres je trochu přísnější v REPEATABLE READ než vyžaduje norma (chová se trochu jako SERIALIZABLE). Norma ale nezakazuje, aby DBMS byli v různých úrovních izolace přísnější. Norma říká, jak „minimálně přísná" má být úroveň izolace. Takže když se v PostgreSQL READ UNCOMMITTED chová stejně jako READ COMMITTED, normu to taky neporušuje :-).

Úroveň izolace **v rámci celého sezení** (spojení s databází) se nastavuje příkazem:

SET SESSION CHARACTERISTICS **AS TRANSACTION** ISOLATION LEVEL { SERIALIZABLE | REPEATABLE **READ** | **READ** COMMITTED | **READ** UNCOMMITTED }

Autocommit

Autocommit je nastavení systému, které způsobí, že každý příkaz je defaultně brán jako (potvrzená transakce). Tedy každý SQL příkaz je vlastně BEGIN; SQL příkaz; COMMIT;.

Pokud je autocommit vypnutý (off), pak první příkaz po přihlášení k databázi a každý příkaz po ukončení předchozí transakce začíná novou transakci, tj BEGIN; SQL příkaz; . Transakce se musí ukončit pomoci COMMIT (nebo ROLLBACK).

Navzdory tomu co tvrdí dokumentace, nelze v Postgresu nastavit AUTOCOMMIT na OFF.

```
rimmer1=> SET AUTOCOMMIT TO OFF;
ERROR:  SET AUTOCOMMIT TO OFF is no longer supported
rimmer1=> SHOW AUTOCOMMIT;
 autocommit
------------
 on
(1 řádka)
```

Autocommit si můžete vypnout pro klienta psql nastavením autocommit na off v jeho konfiguračním souboru. V Linuxu je to soubor ~/.psqlrc.

Zadejte do něj tuto řádku:

set AUTOCOMMIT off

Toto nastavení platí samozřejmě jen pro klienta psql. Od této chvíle v něm budete muset každou změnu potvrzovat příkazem COMMIT. Pozor! Bezchybné ukončení psql poslední transakci nepotvrdí (jak by podle standardu mělo).

SERIALIZABLE vs REPEATABLE READ

Jaký že je rozdíl v **PostgreSQL** (v 9.1 a novějším) mezi SERIALIZABLE a REPEATABLE READ?

Stručně řečeno, pokud je úroveň izolace SERIALIZABLE, musí celá transakce začít a skončit tak, že všechny SQL příkazy v rámci serializované transakce nemohli být ovlivněné příkazy z jiné transakce.

Následující příklad by v úrovni izolace REPEATABLE READ prošla (obě transakce by se podařilo COMMITnout), ale při SERIALIZABLE projde úspěšně jen ta první COMMITnutá.

```
conn1=> BEGIN;
conn1=> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
conn2=> BEGIN;
conn2=> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
conn2=> SELECT * FROM trest ORDER BY id;
 id | uzivatel | penize
----+----------+--------
```

```
 1 | Franta   |  30000
 2 | Tonda    |  35000
 3 | Pepa     |  12000
 4 | Marie    |   4444
(4 řádky)
conn1=> UPDATE trest SET penize = 22000 WHERE id = 1;
conn2=> UPDATE trest SET penize = 32000 WHERE id = 2;
conn1=> COMMIT;
COMMIT
conn2=> COMMIT;
ERROR:  could not serialize access due to read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.
DOPORUČENÍ:  The transaction might succeed if retried.
```

Proč že druhá transakce selhala? Protože conn1 sáhla na tabulku trest a tak conn2 už nemá jistotu, že provedené změny neovlivní aktuální transakci. Konkrétně SELECT v conn2 by dopadl jinak, kdyby transakce conn2 proběhla celá až po conn1. Platí to i obráceně. Kdybych nejdřív ukončil druhou transakci, commitla by se. Selhal by ale následný commit první transakce – řádek s id 2 by byl během transakce změněn.

Jak už jsem psal, při REPEATABLE READ by se obě transakce odeslali vpořádku.

Chyba by při REPEATABLE READ nastala, pokud by se transakce pokusili přepsat stejný řádek. Při READ COMMITTED by pokus o přepsání stejného řádku v druhé transakci byl zablokován, dokud by neskončila první transakce a pak by proběhl (platil by výsledek po ukončení druhé transakce).

Aby to fungovalo jak jsem pospal, musí mít obě transakce nastaven transaction isolation level na SERIALIZABLE. Je to divný, že level jedné transakce ovlivní funkci jiné transakce, ale je to tak. Pokud by conn1 neměl level SERIALIZABLE, neúčastil by se "serializable" transakcí a neovlivnil by tak conn2.

Problém vejce a slepice

Představte si, že potřebujete vytvořit dvě tabulky (třeba slepice a vejce), které se na sebe navzájem odkazují. Problém nastane už při vytváření první tabulky, která by se měla odkázat na druhou tabulku, která ovšem ještě neexistuje.

To se dá vyřešit přidáním integritního omezení FOREIGN KEY až po vytvoření tabulek.

Pak je tu ale ještě jeden zakopaný pes (nebo slepice?). Když budete chtít vložit do tabulky slepici a odkazovat se na vejce (ze kterého třeba pochází), musí vejce už být uloženo (když je odkaz na vejce NOT NULL). Jenže když chcete uložit vejce, musíte se odkazovat na slepici …

Tento problém lze řešit pomocí odkládání kontroly integritních omezení na konec transakce. Postup je následující: začnete transakci a řeknete, že nechcete kontrolovat integritní omezení. Provedete insert do obou tabulek a transakci ukončíte. Až při jejím ukončení DBMS zkontrolue, zda odložená integritní omezení jsou platná a pokud ne, transakci zruší.

Aby šlo kontrolu integritního omezení odložit, musí se už při vytváření integritního omezení označit za **deferrable**. Implicitně je totiž každé integritní omezení not deferrable – nejde odložit. Odkládání kontroly na konec transakce totiž stojí nějakou tu námahu pro DBMS navíc.

Pokud označíte IO za deferrable, musíte ještě určit jestli je **initially deferred** nebo **initially immediate** (tedy jestli se má implicitně odložit až na konec transakce, nebo se má implicitně kontrolovat hned).

To, jestli mají být IO odloženy na konec transakce nebo ne se dá nastavit explicitně během transakce příkazy:

```
SET CONSTRAINTS [MODE] ALL DEFERRED;
SET CONSTRAINTS [MODE] ALL IMMEDIATE;
```

Můžete to nastavit i pro konkrétní IO dle jejich jmen:

```
SET CONSTRAINTS [MODE] io1, io2,... {DEFERRED|IMMEDIATE};
```

Nastavení platí pouze pro IO, které byli vytvořeny jako **deferrable** a pouze do konce aktuální transakce.

Pokud chcete, aby platila změna pro celé sezení (aktuální připojení k databázi), můžete to udělat takto:

```
ALTER SESSION SET CONSTRAINT {DEFERRED|IMMEDIATE};
```

Odsunutí nelze aplikovat na CHECK a NOT NULL.

A teď tedy praktický příklad. Nejdřív vytvoření tabulek:

```
CREATE TABLE slepice (id INT NOT NULL PRIMARY KEY, vejce_id INT NOT NULL);
CREATE TABLE vejce (id INT NOT NULL PRIMARY KEY, slepice_id INT NOT NULL);
ALTER TABLE slepice ADD FOREIGN KEY (vejce_id)
REFERENCES vejce(id) DEFERRABLE INITIALLY IMMEDIATE;
ALTER TABLE vejce ADD FOREIGN KEY (slepice_id)
REFERENCES slepice(id) DEFERRABLE INITIALLY IMMEDIATE;
```

Vložení záznamů:

```
rimmer1=> INSERT INTO vejce VALUES(1,1);
ERROR:  insert or update on table "vejce" violates foreign key constraint "vejce_slepice_id_fkey"
DETAIL:  Key (slepice_id)=(1) is not present in table "slepice".
rimmer1=> BEGIN;
rimmer1=> SET CONSTRAINTS ALL DEFERRED;
rimmer1=> INSERT INTO vejce VALUES(1,1);
INSERT 0 1
rimmer1=> INSERT INTO slepice VALUES(1,1);
INSERT 0 1
rimmer1=> COMMIT;
COMMIT
```

Zamykání tabulek

Pokud jste pročetli poctivě kapitolu od zhora až sem, tak už asi tušíte, že to s těma transakcema není zase tak jednoduchý, jak jsem se vám na začátku snažil namluvit :-).

Existuje způsob, jak si nelámat hlavu úrovní izolace transakce. A to je zamykání (LOCK) tabulek.

Můžete si zamknout celou tabulku (nebo i více tabulek) pro zápis, nebo i pro čtení. Do zamknutých taublek pak nikdo nesmí zapisovat nebo znich ani číst. Pokud se o to někdo pokusí, příkaz zamrzne, dokud tabulku neodemknete. Nebo příkaz po nějaké době selže.

Existuje mnoho různých typů zámků, které můžete v Postgresu na tabulku aplikovat. Jejich popis najdete v dokumentaci Explicit Locking. Pokud žádný typ neurčíte, bere se implicintě ACCESS EXCLUSIVE, který ostatním neopovolí vůbec nic (ani zápis do tabulky, ani její čtení).
Představte si situaci, kdy váš program chce vložit do tabulky trest řádek s id 3, uživatelem Tonda a částkou 0 Kč. Pokud ale řádek s tímto ID už existuje, tak ho chce jen updatovat.
ID je primární klič, tak je dobré míst jistotu, že se nikdo jiný nepokusí vložit do tabulky to samé, pokud zjistím, že řádek s daným ID neexistuje. Mohlo by se stát, že dva konkurenční programi si v (téměř) stejný okamžik zjistí že takový řádek v tabulce není a oba se pokusí ho vložit. Když si ale zamknu tabulku, tak se to stát nemůže.
Tabulka se zamyká příkazem **LOCK TABLE** a jde jí zamknout jen v rámci transakce. Tabulky se uvolní po ukončení transakce (Postgres nemá *UNLOCK TABLES*).
Postup vypadá tedy nějak následovně:

```
BEGIN;
LOCK TABLE trest;
SELECT * FROM trest WHERE id = 3;
-- kdyz radek neexistuje
INSERT ...;
-- kdyz radek existuje
UPDATE ... WHERE id = 3;
COMMIT;
```

Zamykání tabulek má samozřejmě výrazně negativní vliv na efektivitu konkurenční práce. Zamykejte tabulky na co nejkratší dobu, nebo raději vůbec.

SELECT FOR UPDATE

Příkaz **SELECT ... FOR UPDATE** zablokuje (v transakci) všechny řádky, které SELECT vrátí, až do konce transakce. Výhodou oproti LOCK table je, že jsou zablokované jen řádky, které vyhovují podmínce WHERE a ne celá tabulka. Nevýhodou je, že nemůžete zablokovat řádek, který neexistuje. Takže v příkladu z části o Zamykání tabulek by bylo SELECT … FOR UPDATE bezmocné.
Řádky jsou zablokované pouze pro další **SELECT ... FOR UPDATE**, UPDATE, DELETE … „Obyčejný" SELECT můžete bez problémů provést.
Příklad použití:

```
SET SESSION TRANSACTION ISOLATION LEVEL ...
-- nejdřív malá inicializace dat v tabulce
conn1=> UPDATE trest SET penize = 10000;
UPDATE 4
conn1=> BEGIN;
BEGIN

conn1=> SELECT * FROM trest WHERE id = 2 FOR UPDATE;
 id | uzivatel | penize
----+----------+--------
  2 | Tonda    |  10000
(1 řádka)

conn2=> SELECT * FROM trest WHERE id = 2;
 id | uzivatel | penize
----+----------+--------
  2 | Tonda    |  10000
(1 řádka)

conn2=> SELECT * FROM trest WHERE id = 2 FOR UPDATE;
-- teď se příkaz zablokoval a čeká na dokončení transakce v conn1
-- buď se dočká, nebo po vypršení nějaké doby umře
```

SELECT FOR UPDATE druhého spojení se zablokoval, ačkoliv jsem nezačal transakci pomocí BEGIN. Vtip je v tom, že jsem v autocommit režimu, takže je to jako bych BEGIN před příkazem zapsal. Po skončení příkazu se automaticky provede COMMIT, což znamená (a modří už tuší), že se v conn2 žádný řádky nezablokují (resp. se zablokují jen na čas v průběhu SELECTu).
Pozor! V PostgreSQL je bug, který uvolní zámek získaný pomocí SELECT FOR UPDATE, pokud SELECTprovedete před SAVEPOINTem, po SAVEPOINTu provedete nějaký UPDATE/DELETE… a pak se k SAVEPOINTu ROLLBACKnete. Viz žlutý rámeček Caution. (Od verze Postgres 9.3 už tato chyba není.)

MySQL

První odlišnost od PostgreSQL je v nastavování úrovně izolace transakcí. MySQL rozlyšuje mezi READ COMMITTED a READ UNCOMMITTED, Defaultní úroveň je (pro tabulky InnoDB) REPEATABLE READ, která není tak přísná jako v Postgresu (kde se víc blíží SERIALIZABLE).
Úroveň transakce se nastavuje **před** jejím začátekem (a ne až po BEGIN).
K nastavení úrovně pro všechny budoucí transakce v aktuálním sezení stačí přidat kličové slovo SESSION:

```
SET SESSION TRANSACTION ISOLATION LEVEL ...
```

Zjištění aktuálního nastavení:

```
mysql> SHOW VARIABLES LIKE "%tx_isolation%";
+---------------+-----------------+
| Variable_name | Value           |
+---------------+-----------------+
| tx_isolation  | REPEATABLE-READ |
+---------------+-----------------+
1 row in set (0.00 sec)
```

V MySQL lze nastavit autocommit na ON nebo OFF:

```
mysql> SET AUTOCOMMIT = Off;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> show variables like 'autocommit';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | OFF   |
+---------------+-------+
```

Řešení problému vejce a slepice

V MySQL nelze odkládat kontrolu integritních omezení (neumí **deferrable**). Problém s vejcem a slepicí lze vyřešit třeba tak, že se vzájemné odkazy umístí do extra tabulky, třeba se jménem slepice_vejce_spoj. Odkazy v této tabulce se udělají unikátní, aby bylo zajištěno to co v původním zadání – slepice může mít jen jeden odkaz na vejce a opačně.

V MySQL také můžete zamykat tabulky, ale funguje to trochu jinak. Máte na výběr jiné druhy zámků (jen WRITE pro zápis a READ pro čtení). Zamykání se také neprovádí uvnitř transakce a zámky se musí uvolnit příkazem **UNLOCK TABLES;**. Uvolní se všechny zámky tabulek, nemůžete uvolnit zámek jen na některé tabulce.

```
LOCK TABLES tabulka1, tabulka2 READ;
-- teď si z tabulek nemůže nikdo nic přečíst, natož zapisovat
... -- nějaká práce
UNLOCK TABLES;
```

LOCK TABLES se může v MySQL používat namísto transakcí. Jako vždy platí, že byste měli zamykat tabulky na co nejkratší dobu. SELECT FOR UPDATE funguje obdobně jako v Postgresu (jen v rámci transakce atp.).

SQLite

SQLite zvládá BEGIN [TRANSACTION], SAVEPOINT, ROLLBACK a COMMIT, ale transakce umí jen v režimu SERIALIZABLE. Pokud se pokusíte upravit tabulku se kterou pracuje jiná transakce, příkaz na nic nečeká a selže. SQLite navíc blokuje (pro UPDATE) celou databázi, nejen tabulku nebo řádek v tabulce.

```
conn1> BEGIN;
conn1> UPDATE trest SET penize = 15000 WHERE uzivatel = 'Franta';
conn2> UPDATE trest SET penize = 10000 WHERE uzivatel = 'Marie';
Error: database is locked
```

SQLite neumí LOCK TABLE, SELECT FOR UPDATE ani deferrable integritní omezení.

SQLite prostě není vhodné do prostředí, kde se hodně uživatelů snaží databázi upravovat. Hodí se ale tam, kde se hodně uživatelů snaží z databáze číst (s tím problém není), třeba na webu, nebo pro jednouživatelské aplikace, které chtějí využít sílu SQL.

Oracle

Tabulku pro testování s daty můžete v Oracle vytvořit třeba takto:

```
CREATE TABLE trest (
id integer primary key,
uzivatel VARCHAR2(255),
penize NUMERIC(10,0)
);
INSERT INTO trest(id, uzivatel, penize) VALUES (1, 'Franta', 30000);
INSERT INTO trest(id, uzivatel, penize) VALUES (2, 'Tonda',  40000);
INSERT INTO trest(id, uzivatel, penize) VALUES (3, 'Pepa',   11000);
INSERT INTO trest(id, uzivatel, penize) VALUES (4, 'Marie',  35000);
COMMIT;
```

Skrz webové rozhraní APEX si bohužel transakce v Oracle moc nevyzkoušíte. Každý příkaz se totiž posílá z prohlížeče na server k vykonání, každý znamená navázání spojení s databází, provedení příkazu a ukončení spojení. Žádná spojení nepřežije více než jeden SQL dotaz.

Použijte buď spojení přes příkazovou řádku, nebo Oracle SQL Developer, nebo kombinaci obojího.

Oracle má defaultně autocommit vypnutý. Takže první příkaz po připojení k databázi nebo po ukončení transakce začíná transakci novou. Pokud se z databáze normálně odhlásíte, provede se automaticky COMMIT.

Ať vás ani nenapadne začínat transakci příkazem ~~BEGIN~~. Tak to v Oracle nefunguje. Jak už jsem psal, v Oracle, když si nezapnete autocommit, začne transakce prvním příkazem.

Tímto začnete novou transakci a přiřadíte jí jméno. COMMIT, SAVEPOINT a ROLLBACK už fungují normálně.

Znovu upozorňuji na autocommit :-). Pokud budete zkoušet příklady z této lekce, tak musíte u conn2 zadávat za každým příkazem (kde jsem nezačínal transakci pomocí BEGIN) COMMIT. Nebo si u conn2 zapněte autocommit.

Autocommit se dá nastavit příkazem **SET AUTOCOMMIT { ON|OFF };**

Aktuální stav se zobrazí příkazem **SHOW AUTOCOMMIT;**

```
oracle> SET AUTOCOMMIT ON;
oracle> SHOW AUTOCOMMIT;
autocommit IMMEDIATE
```

Oracle má jako defaultní úroveň izolace transakcí READ COMMITTED. Nastavení úrovně pro celé sezení (aktuální přihlášení) se provádí příkazy:

```
ALTER SESSION SET ISOLATION_LEVEL = SERIALIZABLE;
ALTER SESSION SET ISOLATION_LEVEL = READ COMMITTED;
```

Je to tak. V Oracle můžete nastavit jen SERIALIZABLE, nebo READ COMMITTED. Tedy můžete nastavit ještě jeden, ale to jde jen pro aktuální transakci. Pro tu se nastavuje úroveň izolace takto:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET TRANSACTION READ ONLY;
```

Úroveň izolace pro aktuální transakci musíte nastavit hned na jejím začátku. READ ONLY, jak název napovídá, umožňuje během transakce jen číst (v transakci bude vidět databáze celou dobu v takovém stavu, kdy transakce začala). Inserty, updaty ani dylíty dělat nemůžete.

Bohužel se mi nepodařilo zjistit žádný hezký způsob, jak zjistit aktuální nastavení úrovně izolací. Jediný způsob je pomocí následujícího SELECTu, který ovšem tahá data ze systémových tabulek, ke kterým má přístup jenom administrátorský účet, jako je například SYSTEM, kterému jste zadávali heslo během instalace (já vám říkal, abyste si ho zapamatovali).

Tento SELECT navíc zobrazí úroveň izolace pouze pro aktivní transakce, tedy takové transakce, které provedly nějaký insert, update, delete …

Takže asi takhle:

```
oracle1> connect system/tajneheslo
oracle2> connect rimmer/tajneheslo
oracle2> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE NAME 'libovolne_jmeno';
oracle2> INSERT INTO trest VALUES (5,'Krasomila',0);
oracle1> SELECT s.sid, s.serial#,t.name,
CASE BITAND(t.flag, POWER(2, 28))
WHEN 0 THEN 'READ COMMITTED' ELSE 'SERIALIZABLE' END
AS isolation_level
FROM v$transaction t, v$session s WHERE t.addr = s.taddr;
       SID    SERIAL#   NAME              ISOLATION_LEVE
----------------------------------------------------
        32         39       libovolne_jmeno     SERIALIZABLE
oracle2> ROLLBACK;
```

Pokud jde o odkládání IO (deferrable), funguje to jako v Postgresu. Jen příkazy pro nastavení pro SESSION vypadají malilinko jinak:

```
ALTER SESSION SET CONSTRAINT = DEFERRED;
ALTER SESSION SET CONSTRAINT = IMMEDIATE;
```

Při zamykání tabulky musíte v Oracle explicitně uvést, v jakém módu (lockmode Clause) ji chcete zamknout.

```
LOCK TABLE trest IN EXCLUSIVE MODE;
```

Teď když v jiné transakci provedete SELECT … FOR UPDATE, tak se SELECT zablokuje, dokud neukončíte transakci (a tím neuvolníte zamknutí tabulky). Čímž jsem i prozradil, že v Oracle funguje SELECT … FOR UPDATE :-).

Závěr

Teď už víte, jak šedá je teorie a barevný je strom života. Možná jste trochu frustrovaní z toho, že pořádně nevíte, co vlastně která úroveň izolace v různých DBMS dovoluje, co nedovoluje, kdy automaticky stornuje transakci, kdy čeká na dokončení té druhé atd. Vypsat všechny varianty by bylo asi na hodně tlustý sešit. Ale tím se nenechte znechutit, obyčejně si vystačíte s defaultním nastavením izolace a tam, kde potřebujete vědět jak to přesně funguje, si to prostě vyzkoušíte :-).

Datum a čas

Při práci s časem a datumem si musíte dávat pozor ně několik věcí. Jendnak na časovou zónu (časová pásma, chcete-li), na letní a zimní čas, na konverzi typů s a bez časovou zónou, na nastavení časové zóny v konfiguraci DBMS atd. O tom všem je tato kapitola.

Kapitola je nechutně dlouhá, ale to proto, že se v každé databázi řeší práce s datumy dost odlišně. Nastudujete si určitě část o Postgresu, ostatní kapitoly už nepopisují podrobně všechny problémy, na které lze narazit.

O datu a čase

Jak asi víte, ne všude na zeměkouli je stejný čas. Když se řekne, že se něco stalo v 11 hodin, ale neřekne se v jaké časové zóně, tak vlastně nevíte, jak už je to dlouho. Časová zóna ale nestačí. Musíte ještě vědět, jestli se to stalo v letním (daylight-saving time) nebo zimním čase.

To, kde platí jaká časová zóna či letní/zimní čas je politické rozhodnutí, které se navíc v čase mění! Takže 13 hodin prvního června roku 2014 a tentýž čas a datum roku 1914 může být posunutý o hodinu. Aby toho nebylo málo, některé státy prochází několika časovými pásmy.

Letní čas začíná poslední březnový víkend (v noci ze soboty na neděli) a končí poslední víkendu v říjnu. Takže rok od roku jindy. Časové pásmo se udává v posunu od UTC (Coordinated Universal Time), též nazývaného GMT, Greenwich Mean Time. (Nultý poledník prochází Královskou observatoří v Greenwichi v Londýně). V ČR platí časové pásmo +1 hodina, nazývané 'Europe/Prague'.

A to jsem se ještě nezmínil o tom, že taky existuje přestupný rok. Navíc, během století došlo k několika reformám kalendáře. Juliánský kalendář zavedl v Římě Gaius Julius Caesar roku 46 př. n. l. V r. 1582 vznikl reformou papeže Řehoře (Gregor) XIII. Gregoriánský kalendář, když rozdíl mezi kalendářním a slunečním počátkem roku dosáhl 10 dní. Počet přestupných let byl zredukován tak, že roky dělitelné stem jsou přestupné pouze jsou-li zároveň dělitelné čtyřmi sty. Gregoriánskou reformu kalendáře z roku 1582 přijala Velká Británie až o dvě století později a Rusko teprve po revoluci 1918. (Viz wikipedie). Ono to s tím datováním je vůbec velice zapeklité.

Taky je potřeba počítat s tím, že každý měsíc má jiný počet dní.

Jak vidíte, vůbec se nedá spolehnout na to, že den má 24 hodin, nebo rok 365 dní. Tyhle problémy jsou naštěstí známy a existují knihovny, které obsahují všechny potřebné informace o změnách času napříč státy i historií. Funkce a operátory v DBMS pro práci s časem by měli všechny problémy reflektovat. Vy jen musíte vědět, jakým způsobem je reflektují, abyste se nedivili, co že vám to vrací za výsledky.

Věděli jste, že existují i přestupné sekundy?

Datové typy

Datový typ	Význam	Popis
timestamp [without time zone]	Datum a čas bez časové zóny	014-01-22 12:00:00
timestamp with time zone	Datum a čas s časovou zónou	2014-01-22 12:00:00+01
date	Datum (bez času)	2014-01-22
time [without time zone]	Čas (bez data)	23:00:00
time with time zone	Čas (bez data) s časovou zónou	Postgres jej podporuje jen proto, že jej vyžaduje standard. Nedoporučuje se používat, protože čas s časovou zónou bez datumu je na prd.
interval	Časový interval.	Používá se pro aritmetiku s časem (přičítání, odčítání časového intervalu, výsledek rozdílu dvou času atp.).

Pro vytvoření datové či časové hodnoty existuje několik způsobů. Jejich kompletní popis najdete v dokumentaci k datovým typům. Nejbezpečnější je používat vždy ISO formát (viz ukázky v tabulce výše).
Interval můžete zadat nějak takto: INTERVAL '1 day 12 hours 59 min 10 sec'. Existují i další alternativní zápisy intervalu, které najdete v dokumentaci, ale tento je nejčitelnější a úplně si s ním vystačíte :-).
Interval se dá omezit, například:
INTERVAL '1 day 12 hours 59 min 10 sec' **HOUR TO MINUTE**
je totéž jako
INTERVAL '1 day 12 hours 59 min'
(ořezává se jen pravá – nejméně významná část z intervalu, takže se ořežou jen vteřiny a dny zůstávají, ikdyž bylo specifikováno „ponech hodiny až minut").
Omezení intervalu může být: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, YEAR TO MONTH, DAY TO HOUR, DAY TO MINUTE, DAY TO SECOND, HOUR TO MINUTE, HOUR TO SECOND, MINUTE TO SECOND.

Konfigurace

V některých zemích je běžné psát datum v pořádí měsíc, den, rok (MDY). U nás je to den, měsíc, rok (DMY). Toto nastavení se týká formátů datumu, který není jednoznačný (např. 1/2/1999 může být 1. února, nebo 2. ledna). V Postgresu si můžete nastavit jednu z těchto možností v jeho konfiguračním souboru volbou**datestyle**. Stejnou volbou se nastavuje i výstupní formát data (jak se datum zobrazí při SELECTu z databáze).
Ideálně byste měli mít nastaveno:
datestyle = 'iso, dmy'.

Výstupní formáty

Nastavení	Příklad
iso	1997-12-17 07:37:16-08
sql	12/17/1997 07:37:16.00 PST
postgres	Wed Dec 17 07:37:16 1997 PST
german	17.12.1997 07:37:16.00 PST

PTS a -08 je označení časové zóny.

Umístění konfiguračního souboru PostgreSQL zjistíte SQL příkazem **SHOW config_file;**. Musíte k tomu mít oprávnění superuživatele. Ty získáte nejsnadněji tak, že se přihlásíte k databázi jako uživatel postgres. (V Linuxu se pomocí příkazu sudo su postgres přepněte na účet postgres (zadejte heslo roota) a spusťte psql.)
V mé distribuci je konfigurační soubor Postgresu /var/lib/pgsql/data/postgresql.conf.
V konfiguračním souboru taky můžete nastavit defaultní časovou zónu (volbou timezone). Obvykle je nastavená tak, že se bere časová zóna z nastavení vašeho operačního systému.
Časovou zónu můžete nastavit SQL příkazem:

```
rimmer1=> SET TIMEZONE TO 'Europe/London';
SET
rimmer1=> SET TIMEZONE TO 'Europe/Prague';
SET
```

A zobrazit aktuální časovou zónu:

```
rimmer1=>  SHOW TIMEZONE;
  TimeZone
---------------
 Europe/Prague
(1 řádka)
```

Seznam časových zón získáte z pohledu pg_timezone_names ze systémového schématu pg_catalog.

```
rimmer1=> SELECT * FROM pg_catalog.pg_timezone_names ORDER BY utc_offset;
              name               | abbrev | utc_offset | is_dst
```

```
-------------------------------------+--------+------------+--------
 Etc/GMT+12                          | GMT+12 | -12:00:00  | f
 posix/Etc/GMT+12                    | GMT+12 | -12:00:00  | f
 posix/Pacific/Niue                  | NUT    | -11:00:00  | f
                                    ...
 GMT                                 | GMT    | 00:00:00   | f
 GMT+0                               | GMT    | 00:00:00   | f
 Europe/London                       | GMT    | 00:00:00   | f
                                    ...
 CET                                 | CET    | 01:00:00   | f
 Europe/Bratislava                   | CET    | 01:00:00   | f
 Europe/Prague                       | CET    | 01:00:00   | f
                                    ...
```

Použití datových typů

Jak už jsem psal, při použití datových typů času a data je nejdůležitější dávat si pozor na časové pásmo a letní/zimní čas. Důležité je také vědět, že timestamp s časovou zónou se převede při ukládání na čas podle aktuálně nastavné časové zóny do UTC a při SELECTu zase zpět z UTC dle aktuální časové zóny. To má jeden neblahý(?) důsledek. Pokud se váš DBMS řídí časovou zónou operačního systému a nějaký nezkušený administrátor vám dočasně časovou zónu změní, může to vést k nemilému důsledku:

```
rimmer1=> CREATE TABLE zonetest (id serial, datumz TIMESTAMP WITH TIME ZONE, datum TIMESTAMP);
rimmer1=> SET TIMEZONE TO 'Europe/Prague';
rimmer1=> INSERT INTO zonetest(datumz,datum) VALUES('2014-01-21 12:00:00+01','2014-01-21 12:00:00');
rimmer1=> SET TIMEZONE TO 'Europe/London';
rimmer1=> INSERT INTO zonetest(datumz,datum) VALUES('2014-01-21 12:00:00+01','2014-01-21 12:00:00');
rimmer1=> SET TIMEZONE TO 'Europe/Prague';
rimmer1=> SELECT * FROM zonetest ORDER BY id;
 id |        datumz         |       datum
----+------------------------+---------------------
  1 | 2014-01-21 12:00:00+01 | 2014-01-21 12:00:00
  2 | 2014-01-21 12:00:00+01 | 2014-01-21 12:00:00
(2 řádky)
rimmer1=> SET TIMEZONE TO 'Europe/London';
rimmer1=> SELECT * FROM zonetest ORDER BY id;
 id |        datumz         |       datum
----+------------------------+---------------------
  1 | 2014-01-21 11:00:00+00 | 2014-01-21 12:00:00
  2 | 2014-01-21 11:00:00+00 | 2014-01-21 12:00:00
```

Zatím vypadá všechno sluníčkově. Datum bez časové zóny zobrazuje vždy stejný čas. Datum s časovou zónou taky, jen v různém formátu podle aktuální časové zóny. V praze zobrazí '12:00:00+01', v Londýně '11:00:00+00', což je to samé zapsané jinak.

Ošklivě ale narazíte, pokud budete převádět čas bez časové zóny na ten s časovou zónou, protože se v takovou chvíli použije aktuální časová zóna:

```
rimmer1=> SET TIMEZONE TO 'Europe/Prague';
rimmer1=> INSERT INTO zonetest(datumz, datum) VALUES('2014-01-21 12:00:00','2014-01-21 12:00:00+01');
rimmer1=> SET TIMEZONE TO 'Europe/London';
rimmer1=> INSERT INTO zonetest(datumz, datum) VALUES('2014-01-21 12:00:00','2014-01-21 12:00:00+01');
rimmer1=> SET TIMEZONE TO 'Europe/Prague';
rimmer1=> SELECT * FROM zonetest;
 id |        datumz         |       datum
----+------------------------+---------------------
  1 | 2014-01-21 12:00:00+01 | 2014-01-21 12:00:00
  2 | 2014-01-21 12:00:00+01 | 2014-01-21 12:00:00
  3 | 2014-01-21 12:00:00+01 | 2014-01-21 12:00:00
  4 | 2014-01-21 13:00:00+01 | 2014-01-21 12:00:00
(4 řádky)
```

Překvapení! Čtvrtý řádek obsahuje jiný čas. Bylo totiž vloženo 12 hodin v Londýně, což je v Praze 13 hodin. Datum bez časové zóny časovou zónu ignoruje, takže ukazuje stabilně 12 hodin.

Pokud budete někdy něco programovat a budete pracovat s datumy s časovou zónou, nespoléhejte na to, že máte vždy nastavené pásmo 'Europe/Prague' a informaci o časovém posunu neignorujte (ani při čtení, ani při zápisu). Dříve nebo později na to dojedete.

Práce s intervaly je jednoduchá. Jen se nesmí zapomínat na letní/zimní čas. Mimochodem, pro zadání časové zóny můžte použít kromě '+01' i jméno (Europe/Prague) nebo zkratku (CET).

```
rimmer1=> SELECT
TIMESTAMP WITH TIME ZONE '2014-03-29 12:00:00 Europe/Prague' + INTERVAL '13 hour' AS"+13",
TIMESTAMP WITH TIME ZONE '2014-03-29 12:00:00 Europe/Prague' + INTERVAL '14 hour' AS"+14";
          +13          |          +14
-----------------------+-----------------------
 2014-03-30 01:00:00+01 | 2014-03-30 03:00:00+02
(1 řádka)

rimmer1=> SELECT
TIMESTAMP '2014-03-29 12:00:00' + INTERVAL '13 hour' AS "+13",
TIMESTAMP '2014-03-29 12:00:00' + INTERVAL '14 hour' AS "+14";
        +13        |        +14
-------------------+--------------------
```

```
2014-03-30 01:00:00 | 2014-03-30 02:00:00
(1 řádka)
```

Při použití časové zóny správně Postgres započítal přechod na letní čas a posunul hodiny o jednu dopředu (a změnil časovou zónu na +02 hodiny). Bez časové zóny prostě přičetl 14 hodin.

Všimněte si, že přičíst 1 den a 24 hodin není vždy totéž. To platí i o roku a 365 dnech (kvůli přestupnému roku), ale neplatí o týdnu a 7 dnech (to je vždy to samé).

```
rimmer1=> SELECT
TIMESTAMP WITH TIME ZONE '2014-03-29 12:00:00+01' + INTERVAL '24 hour' AS "24 hodin",
TIMESTAMP WITH TIME ZONE '2014-03-29 12:00:00+01' + INTERVAL '1 day' AS "1 den";
        24 hodin        |         1 den
------------------------+------------------------
2014-03-30 13:00:00+02 | 2014-03-30 12:00:00+02
(1 řádka)
```

Speciální hodnoty

Při zadávání data nebo času můžete použít některou z následujících speciálních hodnot. Hodnoty se při použití uzavírají do uvozovek.

Hodnota	Použitelné s	Popis
epoch	date, timestamp	1970-01-01 00:00:00+00 (UNIXový systémový čas se ukládá jako počet vteřin od tohoto data)
infinity	date, timestamp	"Nejpozdější" čas
-infinity	date, timestamp	"Nejdřívější" čas
now	date, time, timestamp	Začátek aktuální transakce
today	date, timestamp	dnešní půlnoc
tomorrow	date, timestamp	zítřejší půlnoc
yesterday	date, timestamp	včerejší půlnoc
allballs	time	00:00:00.00 UTC

Tyto hodnoty není vhodné používat jako defaultní hodnoty. Viz funkce.

```
rimmer1=> SELECT
TIMESTAMP 'yesterday' as včera,
TIMESTAMP 'today' AS dnes,
DATE 'tomorrow' AS zítra;
        včera        |        dnes        |   zítra
---------------------+--------------------+------------
2014-01-23 00:00:00 | 2014-01-24 00:00:00 | 2014-01-25
(1 řádka)
```

Formátování a konverze

Postgres mnohdy dokáže převést text na TIMESTAMP implicitně. Takže můžete insertovat hodnotu jako '2014-01-24 13:30:00 CET' bez explicitní konverze na TIMESTAMP.

Postgres dokáže automaticky převést jen některé způsoby zápisu data a jak datum převede je navíc závyslé na konfiguraci. Nejbezpečnější je použít ISO formát (viz příklad v předchozím odstavci), ale někdy můžete potřebovat převést i jiný formát zápisu.

K převodu textu na datum/čas a obráceně slouží funkce TO_CHAR a TO_TIMESTAMP. Pomocí TO_TIMESTAMP taky můžete převést unixový čas (počet vteřin od 1970-01-01 00:00:00+00) na TIMESTAMP.

```
rimmer1=> SELECT
TO_CHAR(TIMESTAMP WITH TIME ZONE '2014-01-24 13:30:01.15+01',
'Day, DD Month YYYY HH24:MI:SS.MS') AS text;
                   text
-----------------------------------------
Friday   , 24 January   2014 13:30:01.150

rimmer1=> SELECT
TO_TIMESTAMP('Friday, 24 January   2014 13:30:01.150',
'Day, DD Month YYYY HH24:MI:SS.MS') AS timestamp;
          timestamp
---------------------------
2014-01-24 13:30:01.15+01
(1 řádka)
```

Druhý argument TO_CHAR a TO_TIMESTAMP můžete poskládat z různých zástupných zkratek pro formát data a času. Všechny je najdete v dokumentaci v tabulce Template Patterns for Date/Time Formatting.

Všimněte si, že TO_TIMESTAMP vrací i časovou zónu, ikdyž jsem jí v prvním argumentu nijak nespecifikoval. On si jí prostě vzal z aktuálního nastavení DBMS. Takže, když budete mít špatně nastavenou časovou zónu a budete jí používat …

Když napíšu `TIMESTAMP WITH TIME ZONE '2014-01-24 13:30:01.15+01'` tak Postgresu říkám, aby ten text (co je v uvozovkách) implicitně převedl na `TIMESTAMP` s časovou zónou. Kdybych napsal jen `TIMESTAMP '2014-01-24 13:30:01.15+01'`, bude Postgres časovou zónu ignorovat.

Funkce **`EXTRACT`** se používá pro získání části data. Můžete jí použít pro získání unixového času (počtu vteřin od začátku „epochy").

```
rimmer1=> SELECT TO_TIMESTAMP(3600*24+3600 + 65); -- 90065
      to_timestamp
------------------------
 1970-01-02 02:01:05+01

rimmer1=> SELECT EXTRACT('epoch' FROM TIMESTAMP WITH TIME ZONE '1970-01-02 02:01:05+01');
 date_part
-----------
     90065
(1 řádka)
```

Funkce `EXTRACT` dokáže vyextrahovat z TIMESTAMPu téměř cokoliv. Podívejte se do dokumentace na to, co všechno můžete extrahovat. Máte tam i spoustu příkladů.

Ke konverzi mezi datovými typy můžete použít funkci **`CAST`**. Ta se používá na konverzi všemožných datových typů, teď jí ale ukáži při práci s datovými typy pro čas a datum. A nezapomeňte (zase) na vliv časové zóny.

```
rimmer1=> SELECT CAST(TIMESTAMP '2014-01-24 13:35:12' as TIME),
CAST(TIMESTAMP '2014-01-24 13:35:12' as TIMESTAMP WITH TIME ZONE);
   time   |      timestamptz
----------+------------------------
 13:35:12 | 2014-01-24 13:35:12+01
(1 řádka)
```

K povídání o konverzi asi ještě patří funkce **`DATE_TRUNC`**. Ta ořeže z data „méně významnou část" (vteřiny jsou méně významné než minuty atd.).

```
rimmer1=> SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40') AS hodiny,
date_trunc('year', TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40+01') AS roky;
       hodiny       |         roky
--------------------+------------------------
 2001-02-16 20:00:00 | 2001-01-01 00:00:00+01
(1 řádka)
```

Co všechno můžete ořezat najdete, jako vždy, v dokumentaci.

Pokud jde o ořezávání, tak ořezat můžete i interval, jak jsem už psal v části o datových typech.

```
rimmer1=> SELECT INTERVAL '3 years 2 months 5 days 20 hours' DAY;
      interval
---------------------
 3 years 2 mons 5 days
(1 řádka)
```

Poslední možností konverze je zobrazení TIMESTAMPu v zadané časové zóně pomocí **`AT TIME ZONE`**.

```
rimmer1=> SET TIMEZONE TO 'Europe/Prague';
rimmer1=> SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'Europe/London';
        timezone
------------------------
 2001-02-16 21:38:40+01
(1 řádka)

rimmer1=> SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40+01' AT TIME ZONE 'GMT'; -- totez co 'Europe/London'
      timezone
---------------------
 2001-02-16 19:38:40
(1 řádka)
```

Všimněte si, jak se v příkladech změní čas bez časové zóny na ten s časovou zónou a naopak.

Funkce a operátory

V Postgresu můžete k datům přičítat a odečítat intervaly, odečítat data od sebe (výsledkem je interval), násobit nebo dělit intervaly (3 * 1 vteřina jsou 3 sekundy). Postgres také poskytuje sadu zajímavých funkcí pro práci s časem a datumem. Všechno najdete i s příklady v dokumentaci.

Pro ty zvás, co se bojí kliknout, nebo mají fobii z angličtiny (ale ty příklady pochopíte i bez ní, fakt), jsem sem obšlehnul pár zajímavých příkladů:

```
rimmer1=> SELECT
DATE '2001-09-28' + INTEGER '7' AS soucet,
TIMESTAMP '2001-09-29 03:00' - TIMESTAMP '2001-09-27 12:00' AS rozdil,
21 * INTERVAL '1 day' AS nasobeni;
   soucet   |     rozdil     | nasobeni
------------+----------------+----------
 2001-10-05 | 1 day 15:00:00 | 21 days
(1 řádka)

rimmer1=> SELECT age('1979-05-14 04:30:00',now()), CURRENT_TIMESTAMP, LOCALTIMESTAMP;
                  age                   |              now              |         timestamp
----------------------------------------+-------------------------------+---------------------------
 -34 years -8 mons -10 days -09:15:26.609082 | 2014-01-24 13:45:26.609082+01 | 2014-01-24 13:45:26.609082
(1 řádka)
```

Zajímavé je i to, co je to aktuální čas. Někdy je to čas, kdy začala transakce (funkce v rámci jedné transakce vrací stále stejný čas, bez ohledu na to, kolik už času uběhlo od začátku transakce), někdy je to čas kdy byl spuštěn SQL příkaz (v rámci transakce každé spuštění příkazu bude mít vlastní čas (stejný v rámci jednoho SQL příkazu)), nebo skutečný aktuální čas (pak i v jednom SQL příkazu může vrátit funkce různý čas).
Tyto funkce vrací vždy čas začátku transakce:

CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
LOCALTIME
LOCALTIMESTAMP

U těchto funkcí na to přijdete z jejich jména (až na NOW() – ta je ekvivalentní s transaction_timestamp()):

now()
transaction_timestamp()
statement_timestamp()
clock_timestamp()

Všechny funkce a datové typy můžete používat jako DEFAULTNí hodnotu. Jen pozor na rozdíl v tomto:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';  -- incorrect for use with DEFAULT
```

Všechny 3 řádky vrací to samé. Třetí řádek ovšem není funkce, ale speciální hodnota. Takže, když použijete TIMESTAMP 'now', budete mít jako defaultní hodnotu datum a čas z doby vytvoření tabulky a ne aktuální čas v době provedení INSERTu (nebo UPDATU).

MySQL

Tahle kapitola už je dost dlouhá a aby toho nebylo málo, tak v MySQL (a nejen v MySQL) je všechno jinak. Snažit se popsat jen to, co je v MySQL jinak, by teď nebylo zrovna přehledný – skoro všechno je jinak. Proto popíšu jak to v MySQL je a abych kapitolu moc nenatahoval, pokusím se MySQL popsat tak stručně, jak to jen jde. Vy už víte, na co si musíte při práci s datem a časem dávat pozor. Když si nebudete něčím v MySQL jistí, vyzkoušejte si to, nebo si to nastudujte z dokumentace.
Toto platí i pro ostatní DBMS (SQLite a Oracle) samozřejmě taky.

Datové typy

V MySQL jsou pro datum a čas k dispozici tři datové typy: **DATE**, **DATETIME** a **TIMESTAMP**. První dva jsou bez časové zóny, TIMESTAMP s časovou zónou. (Ale na rozdíl od PostgreSQL se neuvádí WITH TIME ZONE.)
Datumy mohou obsahovat zlomky sekund na 6 desetinných míst (v Postgresu jen na 3), mohou se zadávat v ISO formátu, třeba takto: '2010-12-10 14:12:09.019473'. Existují i další způsoby zápisu dat, které MySQL dokáže rozšifrovat, doporučuju ale držet se tohoto způsobu.
U MySQL můžete před datumem/časem zadat datový typ DATE, TIME (ikdyž datový typ TIME nezná) a TIMESTAMP, protože to vyžaduje SQL standard. MySQL tento datový typ ignoruje, takže ho nepoužívejte, ať vás to nemate.

```
mysql> SELECT
    -> DATE '2010-12-10 14:12:09.019473' AS "date",
    -> TIME '2010-12-10 14:12:09.019473' AS "time",
    -> TIMESTAMP '2010-12-10 14:12:09.019473' AS "timestamp";
+----------------------------+----------------------------+----------------------------+
| date                       | time                       | timestamp                  |
+----------------------------+----------------------------+----------------------------+
| 2010-12-10 14:12:09.019473 | 2010-12-10 14:12:09.019473 | 2010-12-10 14:12:09.019473 |
+----------------------------+----------------------------+----------------------------+
1 row in set (0.00 sec)
```

Datovou zónu zadat nemůžete. Bere se z aktuálně nastavené datové zóny (pro TIMESTAMP, ostatní datové typy (DATE a DATETIME) datovou zónu ignorují).

Konfigurace

MySQL bere časovou zónu z operačního systému. Můžete si jí v konfiguračním souboru i pomocí SQL příkazu nastavit, ale nejdřív musíte do MySQL časové zóny nahrát. MySQL má sice v schématu mysqlpřipravené tabulky (time_zone*), ale ty jsou prázdné.
K naplnění těchto tabulek slouží program mysql_tzinfo_to_sql. Pokud používáte Linux, předáte mu na příkazové řádce cestu k adresáři s informacemi o časových zónách (obvykle /usr/share/zoneinfo) a jeho výstup přesměrujete do klienta mysql.

```
$ mysql_tzinfo_to_sql /usr/share/zoneinfo | mysql -u root mysql
```

Tento program by měl být ve vaší linuxové distribuci. V OpenSuSE jsem musel doinstalovat balíček mysql-community-server-tools, v Debianu byl součástí balíčku mysql-server, takže jsem nic doinstalovávat nemusel.
Pokud používáte Windows, můžete si stáhnout soubory s časovými zónami zde.
Po nahrání časových zón si je můžete zobrazit třeba takto:

```
mysql> SELECT n.Name, t.Offset, t.is_DST, t.Abbreviation
    -> FROM time_zone_transition_type t JOIN time_zone_name n
    -> ON t.Time_zone_id = n.Time_zone_id
    -> WHERE Name = 'Europe/Prague';
+---------------+--------+--------+--------------+
| Name          | Offset | is_DST | Abbreviation |
+---------------+--------+--------+--------------+
| Europe/Prague |   7200 |      1 | CEST         |
| Europe/Prague |   3600 |      0 | CET          |
| Europe/Prague |   7200 |      1 | CEST         |
| Europe/Prague |   3600 |      0 | CET          |
| Europe/Prague |   7200 |      1 | CEST         |
| Europe/Prague |   3600 |      0 | CET          |
+---------------+--------+--------+--------------+
6 rows in set (0.00 sec)
```

Offset je posun od UTC ve vteřinách a is_DST říká, jestli je časová zóna letní čas (daylight-saving time).

Zobrazení a nastavení časové zóny aktuálního spojení se v MySQL provede takto:

```
mysql> SELECT @@session.time_zone;
+--------------------+
| @@session.time_zone |
+--------------------+
| SYSTEM             |
+--------------------+
-- SYSTEM znamená, že se bere zóna z operačního systému
mysql> SET time_zone = 'Europe/Prague';
ERROR 1298 (HY000): Unknown or incorrect time zone: 'Europe/Prague'
-- první pokus před nahráním časových zón selhal
mysql> SET time_zone = 'Europe/Prague';
Query OK, 0 rows affected (0.00 sec)
-- po nahrání časových zón (pomocí mysql_tzinfo_to_sql)
```

Další informace o časových zónách v MySQl viz MySQL Server Time Zone Support.

TIMESTAMP

TIMESTAMP má v MySQL několik zajímavých vlastností. Předně si dávejte pozor na časovou zónu:

```
mysql> CREATE TABLE zonetest (id integer primary key auto_increment, datumz TIMESTAMP, datum DATETIME);
mysql> SET time_zone = "Europe/Prague";
mysql> INSERT INTO zonetest(datumz, datum) VALUES('2014-01-21 12:00:00', '2014-01-21 12:00:00');
mysql> SET time_zone = "Europe/London";
mysql> INSERT INTO zonetest(datumz, datum) VALUES('2014-01-21 12:00:00', '2014-01-21 12:00:00');
mysql> SELECT * FROM zonetest ORDER BY id;
+----+---------------------+---------------------+
| id | datumz              | datum               |
+----+---------------------+---------------------+
|  1 | 2014-01-21 11:00:00 | 2014-01-21 12:00:00 |
|  2 | 2014-01-21 12:00:00 | 2014-01-21 12:00:00 |
+----+---------------------+---------------------+
2 rows in set (0.00 sec)
```

Myslím, že výsledek už dokážete interpretovat sami. První řádek vložil čas v časové zóně 'Europe/Prague', ale při SELECTu byla nastavená zóna Europe/London …

Časová zóna se nedá v MySQL zadat explicitně, vždy se bere z aktuálního nastavení:

```
mysql> SET time_zone = "Europe/London";
mysql> INSERT INTO zonetest(datumz, datum) VALUES('2014-01-21 12:00:00+01:00', '2014-01-21 12:00:00+01:00');
SHOW warnings;
+---------+------+-----------------------------------------------------+
| Level   | Code | Message                                             |
+---------+------+-----------------------------------------------------+
| Warning | 1265 | Data truncated for column 'datumz' at row 1         |
| Warning | 1264 | Out of range value for column 'datum' at row 1      |
+---------+------+-----------------------------------------------------+
2 rows in set (0.00 sec)
mysql> SELECT * from zonetest WHERE id = 3;
+----+---------------------+---------------------+
| id | datumz              | datum               |
+----+---------------------+---------------------+
|  3 | 2014-01-21 12:00:00 | 2014-01-21 12:00:00 |
+----+---------------------+---------------------+
1 row in set (0.00 sec)
```

Kdyby se časová hodnota (12 hodin) uložila s časovou zónou +01, pak by se musela v nastavené časové zóně "Europe/London" zobrazit jako 11 hodin. Ve skutečnosti se uložilo 12 hodin v nastavené časové zóně ("Europe/London").

TIMESTAMP jako jediný datový typ může mít jako defaultní hodnotu aktuální čas. Dokonce se automaticky aktualizuje na aktuální čas i v případě UPDATE řádku (pokud jsou aktualizovány ostatní sloupce). Toto chování můžete nastavit explicitně. **Když ale nenastavíte ani defaultní hodnotu, ani automatický update, tak se tak chová TIMESTAMP stejně**.

Následující definice tabulek jsou totožné:

```
CREATE TABLE t1 (id INT, datum TIMESTAMP);
CREATE TABLE t2 (id INT, datum TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP);
```

Pouze jeden slopec v tabulce může mít nastavenou defaultní hodnotu nebo automatický UPDATE na aktuální čas. To neznamená, že byste nemohli mít více sloupečků s datovým typem TIMESTAMP. Pokud se nepokusíte uvést něco jinak, tak automatický update má pouze první sloupeček s TIMESTAMP, ostatní ne. Pokud chcete mít defaultní hodnotu nebo automatický UPDATE na jiném než prvním sloupečku, musíte u toho prvního explicitně automatické hodnoty vypnout a u druhého zase zapnout. Například, když nastavíte defaultní hodnotu na 0, nebude fungovat ani ON UPDATE CURRENT_TIMESTAMP.

```
mysql> CREATE TABLE t3(cislo INTEGER, datum1 TIMESTAMP DEFAULT 0, datum2 TIMESTAMP DEFAULT CURRENT_TI
MESTAMP ON UPDATE CURRENT_TIMESTAMP);
mysql> INSERT INTO t3 (cislo) VALUES (1);
mysql> SELECT * FROM t3;
+-------+---------------------+---------------------+
| cislo | datum1              | datum2              |
+-------+---------------------+---------------------+
|     1 | 0000-00-00 00:00:00 | 2014-01-24 20:18:10 |
+-------+---------------------+---------------------+
1 row in set (0.00 sec)
mysql> UPDATE t3 SET datum1 = CURRENT_TIMESTAMP, cislo = 2;
mysql> SELECT * FROM t3;
```

```
+-------+---------------------+---------------------+
| cislo | datum1              | datum2              |
+-------+---------------------+---------------------+
|     2 | 2014-01-24 20:20:21 | 2014-01-24 20:20:21 |
+-------+---------------------+---------------------+
1 row in set (0.00 sec)
```

Pokud u datového typu `TIMESTAMP` neurčíte explicitně že může obsahovat `NULL`, tak `NULL` obsahovat nemůže. Pokud budete do takového sloupce vkládat `NULL`, vloží se místo něj defaultní hodnota (aktuální čas).

```
mysql> INSERT INTO t3 VALUES (3, NOW(), NULL);
mysql> SELECT * FROM t3;
+-------+---------------------+---------------------+
| cislo | datum1              | datum2              |
+-------+---------------------+---------------------+
|     2 | 2014-01-24 20:21:07 | 2014-01-24 20:21:07 |
|     3 | 2014-01-24 20:23:04 | 2014-01-24 20:23:04 |
+-------+---------------------+---------------------+
```

Podrobné pravidal o tom, kdy se používají autmoatické hodnoty a kdy ne najdete v dokumentaci Automatic Initialization and Updating for TIMESTAMP.

Funkce

MySQL obsahuje hodně funkcí pro práci s časem a datem. Chybí jí snad jen nějaké funkce na výpočet věku.
V tabulce je seznam těch nejdůležitějších. Další najdete v dokumentaci Date and Time Functions.

Funkce	Význam
CONVERT_TZ	Konvertuje z jedné časové zóny do jiné.
NOW(), CURRENT_TIMESTAMP	Vrací čas začátku transakce (v aktuální časové zóně)
SYSDATE()	Vrací čas ve chvíli spuštění funkce (takže může vrátit v rámci jedno SQL dotazu různé časy).
UTC_TIMESTAMP()	Vrací aktuální čas v UTC zóně
DATE_ADD()	Přičítá interval k datu
DATE_SUB()	Odečítá interval od data
DATE()	Extrahuje pouze datovou část z DATETIME nebo TIMESTAMPu
DATEDIFF()	Odečte dvě data a vrátí rozdíl v počtu dní
EXTRACT()	Extrahuje zadanou část z data. Výrazy pro extrahované části data jsou stejné, jako pro interval, viz DATE_ADD()
FROM_UNIXTIME()	Převede číslo na datum (jako počet vteřin od 1970-01-01)
UNIX_TIMESTAMP()	Převede datum na číslo (jako počet vteřin od 1970-01-01)
DATE_FORMAT()	Naformátuje datum podle zadaného formátu.
STR_TO_DATE()	Inverzní funkce k DATE_FORMAT() (převede řetězec na datový typ DATETIME, DATE nebo TIME)

Příklady použití funkcí

```
mysql> SET time_zone = "Europe/Prague";
mysql> SELECT
CONVERT_TZ('2014-01-01 12:00:00','SYSTEM','Europe/London') as PrgToLon;
+---------------------+
| PrgToLon            |
+---------------------+
| 2014-01-01 11:00:00 |
+---------------------+

mysql> SELECT NOW(), UTC_TIMESTAMP;
+---------------------+---------------------+
| NOW()               | UTC_TIMESTAMP       |
+---------------------+---------------------+
| 2014-01-24 21:27:24 | 2014-01-24 20:27:24 |
+---------------------+---------------------+

mysql> SELECT
DATE_ADD('2014-03-29 12:00:00', INTERVAL 13 hour) as "+13",
```

```
DATE_ADD('2014-03-29 12:00:00', INTERVAL 14 hour) as "+14";
+---------------------+---------------------+
| +13                 | +14                 |
+---------------------+---------------------+
| 2014-03-30 01:00:00 | 2014-03-30 02:00:00 |
+---------------------+---------------------+

mysql> SELECT
DATE_ADD('2014-01-24 12:00:00', INTERVAL -11 hour) as "-11",
DATE_SUB('2014-01-24 12:00:00', INTERVAL 11 hour) as "-11";
+---------------------+---------------------+
| -11                 | -11                 |
+---------------------+---------------------+
| 2014-01-24 01:00:00 | 2014-01-24 01:00:00 |
+---------------------+---------------------+

mysql> SELECT
DATE('2014-01-24 12:00:00') as date,
DATEDIFF('2014-01-24 11:00:00', '1989-11-17 23:00:00') as datediff;
+------------+----------+
| date       | datediff |
+------------+----------+
| 2014-01-24 |     8834 |
+------------+----------+

mysql> SELECT
EXTRACT(YEAR FROM '2014-01-24 12:00:00') AS year,
EXTRACT(DAY_MINUTE FROM '2014-01-24 12:30:00') AS "day_minute";
+------+------------+
| year | day_minute |
+------+------------+
| 2014 |     241230 |
+------+------------+

mysql> SELECT
UNIX_TIMESTAMP('2014-03-30 02:00:00') as "02",
UNIX_TIMESTAMP('2014-03-30 03:00:00') as "03",
FROM_UNIXTIME(1396141200) as "from_unixtime";
+------------+------------+---------------------+
| 02         | 03         | from_unixtime       |
+------------+------------+---------------------+
| 1396141200 | 1396141200 | 2014-03-30 03:00:00 |
+------------+------------+---------------------+

mysql> SET time_zone = 'UTC';
mysql> SELECT
UNIX_TIMESTAMP('2014-03-30 02:00:00') as "02",
UNIX_TIMESTAMP('2014-03-30 03:00:00') as "03",
FROM_UNIXTIME(1396141200) as "from_unixtime";
+------------+------------+---------------------+
| 02         | 03         | from_unixtime       |
+------------+------------+---------------------+
| 1396144800 | 1396148400 | 2014-03-30 01:00:00 |
+------------+------------+---------------------+

mysql> SELECT DATE_FORMAT('2009-10-04 22:23:00', '%W %M %Y') as format;
+--------------------+
| format             |
+--------------------+
| Sunday October 2009 |
+--------------------+

mysql> SELECT
STR_TO_DATE('04/31/2004 13:35:10.1275', '%m/%d/%Y %H:%i:%s.%f') as "date";
+----------------------------+
| date                       |
+----------------------------+
| 2004-04-31 13:35:10.127500 |
+----------------------------+
```

SQLite

SQLite poskytuje jen 2 datové typy pro práci s časem a datumem: **DATE** a **DATETIME**. Oba se ukládají interně jako číslo. Funkce pro práci s časem je v zásadě jen jedna: **stftime**. Existují ještě další funkce, které jsou ale jen zkrácený zápis stftime.

funkce	Ekvivalent pomocí strftime
strftime(format, timestring, modifier, modifier, ...)	
date(timestring, modifier, modifier, ...)	strftime('%Y-%m-%d',timestring, modifier, modifier, ...)
time(timestring, modifier, modifier, ...)	strftime('%H:%M:%S',timestring, modifier, modifier, ...)
datetime(timestring, modifier, modifier, ...)	strftime('%Y-%m-%d %H:%M:%S',timestring, modifier, modifier, ...)

timestring je nejjednodušší zadávat ve formátu 'YYYY-MM-DD HH:MM:SS.SSS', tedy například '2014-01-30 13:30:25.155', kde poslední 3 čísla jsou zlomky sekund.
modifier je něco podobného jako interval v Postgresu. Třeba následující příklad zobrazí poslední den aktuálního měsíce:

```
sqlite> SELECT DATE('now','start of month','+1 month','-1 day') as "posledni den";
posledni den
------------
2014-01-31
```

Kompletní seznam modifikátorů najdete v dokumentaci.
format pro funkci strftime je stejný, jako pro funkci strftime z jazyka C. Možnosti substitučních formátů najdete i v dokumentaci SQLite. Funkce strftime z SQLite má navíc substituční formát pro frakci sekund %f.
V SQLite se na časové zóny moc nehraje. Můžete připsat na konec času posun od UTC času ve formátu +-HH:MM. Hodnota se k datu a času přičte (či odečte), výsledek se vždy ukládá v UTC. Letní/zimní čas už se vůbec nebere v potaz.

```
sqlite> SELECT DATETIME('2014-06-01 08:00:00+01:00') AS UTC;
UTC
-------------------
2014-06-01 07:00:00
```

8 hodin v našem časovém pásmu je 7 hodin v UTC.
Rozdíl oproti Postgresu (i MySQL) je i při přičítání 1 měsíce. SQLite prostě zvýší hodnotu měsíce o jedna a pak datum normalizuje. Takže 2001-03-31 + 1 měsíc je 2001-04-31, ale protože duben nemá 31 dní, výsledkem je 2001-05-01. V Postgresu (i MySQL) by byl výsledek 2001-04-30. (Postgres i MySQL vrací tento výsledek pro dva různé datumy, všimli jste si?)
Pokud jde o extrakci části datumu, nebo konverzi datumu, tak na to si musíte vystačit s funkcí strftime.

Oracle

Datové typy

Datový typ	Význam
DATE	Datum **i čas** bez časové zóny. Rozsah od 1. ledna 4712 B.C. do 31 prosince 9999
TIMESTAMP	Datum i čas bez časové zóny, oproti DATE ukládá i zlomky sekund
TIMESTAMP WITH TIME ZONE	Ukládá se i časová zóna.
TIMESTAMP WITH LOCAL TIME ZONE	Čas se převádí do UTC a z UTC jako v Postgresu u TIMESTAMP WITH TIME ZONE. (Takže původní časovou zónu už nejde zjistit).
INTERVAL YEAR TO MONTH	Interval obsahující rok a měsíc. Jedná se skutečně o datový typ (můžete vytvořit tabulku se sloupečkem s tímto datovým typem).
INTERVAL DAY TO SECOND	Interval obsahující den, hodinu, minutu, vteřinu a zlomky vteřin.

V Oracle se defaultně zobrazuje a zadává DATE ve formátu DD-MON-RR, tj. například 24-JAN-14, přestože uchovává i čas.
Další možnosti zadání data jsou:

1. **DATE** '2014-01-24'
2. TO_DATE('1998-DEC-25 17:30:00.50','YYYY-MON-DD HH24:MI:SS.FF','NLS_DATE_LANGUAGE=AMERICAN')
3. **TIMESTAMP** '1997-01-31 09:26:50.123'
4. **TIMESTAMP** '1999-01-15 11:00:00 -5:00'
5. **TIMESTAMP** '1999-01-15 8:00:00 America/Los_Angeles'

První řádek je dle ANSI normy. Druhý řádek využívá funkce TO_DATE. Třetí parametr je tam kvůli určení jazyka, ve kterém je název měsíce (DEC je americká zkratka pro prosinec).
Poslední tři řádky jsou taky zadání dle ANSI normy. Třetí řádek ukazuje zadání TIMESTAMP s přesností na destinu vteřiny, čtvrtý a pátý dva způsoby, jak je možné zadat časovou zónu.
Intervaly můžete zadat takto:

```
INTERVAL '123-2' YEAR(3) TO MONTH
INTERVAL '4 5:12:10.222' DAY TO SECOND(3)
```

Pravidla pro zadávání roku dvěma číslicemi jsou:

	Pokud je zadaný rok v rozsahu:	
	0-49	50-99

Pokud aktuální rok končí na:	0-49	Vrácený datum je v aktuálním století	Vrácený datum je v předházejícím století
	50-99	Vrácený datum je v následujícím století	Vrácený datum je v aktuálním století

```
-- spuštěno v roce 2014
oracle> SELECT
TO_CHAR(TO_DATE('21-BŘE-14'), 'YYYY-MM-DD') AS "2014",
TO_CHAR(TO_DATE('21-BŘE-99'), 'YYYY-MM-DD') AS "1999"
FROM dual;
2014       1999
---------- ----------
2014-03-21 1999-03-21
```

Konfigurace

Podporované časové zóny zjistíte tímto příkazem:

```
oracle> SELECT * FROM V$TIMEZONE_NAMES WHERE TZNAME = 'Europe/Prague';
TZNAME          TZABBREV
Europe/Prague   LMT
Europe/Prague   PMT
Europe/Prague   CET
Europe/Prague   CEST
```

CET	Central Europe Time
CEST	Central Europe Summer Time (letní čas)

Defaulní časový formát (pro typ DATE je to 'DD-MM-RR') lze nastvit pro aktuální sezení takto:

```
ALTER SESSION SET NLS_DATE_FORMAT='DD-MON-YYYY HH24:MI:SS'; --  '01-JAN-2003'
ALTER SESSION SET NLS_TIMESTAMP_FORMAT='DD-MON-YY HH:MI:SSXFF';
ALTER SESSION SET NLS_TIMESTAMP_TZ_FORMAT='DD-MON-RR HH:MI:SSXFF AM TZR';

ALTER SESSION SET NLS_TIMESTAMP_FORMAT='YYYY-MM-DD HH24:MI:SS.FF6'; -- '2015-10-02 13:10:10.123456'
...
```

Nastavené formáty lze zjisit takto:

```
SELECT value FROM nls_session_parameters WHERE parameter = 'NLS_TIMESTAMP_FORMAT';
...
```

Změna časové zóny pro aktuální sezení(vyberte si dle libosti):

```
ALTER SESSION SET TIME_ZONE='-7:00';
ALTER SESSION SET TIME_ZONE='Asia/Hong_Kong';
ALTER SESSION SET TIME_ZONE=dbtimezone; -- podle nastavení databáze
```

V Oracle se časová zóna nastavuje na úrovni databáze. Při vytvoření databáze se převezme z operačního systému, nebo jí můžete určit explicitně:

```
CREATE DATABASE db01
...
SET TIME_ZONE='Europe/London';
```

Později se dá změnit (pokud na to máte dostatečná práva):

```
ALTER DATABASE SET TIME_ZONE='Europe/Prague';
```

Nastavenou časovou zónu databáze (ne platnou pro aktuální sezení) zjistíte takto:

```
oracle> SELECT dbtimezone FROM DUAL;
DBTIME
------
+00:00
```

Konverze

Konverze datových typů

Funkce pro konverzi můžete vidět na obrázku. Pro práci s datem jsou důležité TO_DATE a TO_CHAR. Co můžete použít pro formát data najdete v dokumentaci.

Například: DD je den v měsíci (1-31), MON je zkratka názvu měsíce, YYYY je nezkrácený rok.

```
SELECT
TO_DATE('01-LED-2003', 'DD-MON-YYYY','NLS_DATE_LANGUAGE=CZECH') as "2003",
TO_DATE('01-BŘE-03', 'DD-MON-YYYY','NLS_DATE_LANGUAGE=CZECH') as "Rok 03"
FROM dual;
2003       Rok 03
---------- ----------
01/01/2003 03/01/0003
```

Třetí parametr nemusíte uvádět, pokud se spolehnete na aktuální nastevní sezení. To zjistíte takto:

```
oracle> SELECT value FROM nls_session_parameters WHERE parameter = 'NLS_DATE_LANGUAGE';
VALUE
------
CZECH
```

Pro další datové typy existují obdobné funkce TO_TIMESTAMP a TO_TIMESTAMP_TZ.

Stejně jako v Postgresu můžete použít AT TITME ZONE:

```
oracle> SELECT TIMESTAMP '2014-07-11 12:00:00 Europe/Prague'
AT TIME ZONE 'Europe/London' AS Londyn
FROM dual;
```

LONDYN

11.07.14 11:00:00,000000000 EUROPE/LONDON
Funkce **FROM_TZ** umožňuje připojit k TIMESTAMPu datovou zónu (a vytvořit tak TIMESTAMP WITH TIME ZONE).

```
oracle> SELECT FROM_TZ(TIMESTAMP '2000-03-28 08:00:00', '3:00')
FROM DUAL;

FROM_TZ(TIMESTAMP'2000-03-2808:00:00','3:00')
---
28-MAR-00 08.00.000000000 AM +03:00
```

Funkce **TO_CHAR** umožňuje vypsat datum v zadaném formátu.

```
oracle> SELECT TO_CHAR(DATE '2014-03-21','YY-MON-DD') FROM dual;
TO_CHAR(DA
---
14-BŘE-21

oracle> SELECT TO_CHAR(TIMESTAMP '2014-03-21 12:30:50.5', 'YYYY-MM-DD HH24:MI:SS.FF')FROM dual;
TO_CHAR(TIMESTAMP'2014-03-211
---
2014-03-21 12:30:50.500000000
```

Funkce a operátory

Přičtení čísla k datu se chápe jako přičtení počtu dnů:

```
oracle> SELECT TO_DATE('28-ÚNO-12') + 1 AS "29.02.2012" FROM dual;

29.02.20
---
29.02.12
```

Můžete přičítat i intervaly, podobně jako v Postgresu:

```
oracle> SELECT
TIMESTAMP '2014-03-29 12:00:00 Europe/Prague' + INTERVAL '13' hour AS "+13",
TIMESTAMP '2014-03-29 12:00:00 Europe/Prague' + INTERVAL '14' hour AS "+14"
FROM dual;
+13                                +14
30-MAR-14 01.00.00.000000000 AM EUROPE/PRAGUE   30-MAR-14 03.00.00.000000000 AM EUROPE/PRAGUE
```

Všimněte si, že číselný údaj v intervalu musí být v uvozovkách.

Funkce pro práci s časem

Funkce	Popis
SYSDATE	SYSDATE vrátí aktuálni čas počítače, na kterém běží databáze. Časovou zónu bere dle nastavení počítače v době spuštění databáze (takže ne časovou zónu nastavenou pro aktuální spojení).
CURRENT_DATE	Vrátí aktuální čas dle aktuálně nastavené časové zóny typu DATE.
CURRENT_TIMESTAMP	Jako CURRENT_DATE, ale vrací TIMESTAMP WITH TIME ZONE
LOCALTIMESTAMP	Jako CURRENT_DATE, ale vrací TIMESTAMP
EXTRACT (xxx FROM datetime)	Extrahuje část data. xxx může být: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, TIMEZONE_MINUTE, TIMEZONE_REGION, TIMEZONE_ABBR).

Příklady

```
oracle> ALTER SESSION SET TIME_ZONE='Asia/Hong_Kong';
oracle> SELECT
TO_CHAR(SYSDATE,'YYYY-MON-DD HH24:MI:SS') "Pocitac",
TO_CHAR(CURRENT_DATE, 'YYYY-MON-DD HH24:MI:SS') "Hong_Kong"
FROM dual;
Pocitac                    Hong_Kong
-------------------------- --------------------------
2014-LED-26 16:07:11       2014-LED-26 23:07:11
```

Další funkce

Funkce	Popis
MONTHS_BETWEEN	Počet měsíců mezi dvěmi daty
ADD_MONTHS	Přidá měsíce k datu
NEXT_DAY	Další den od zdaného data
LAST_DAY	Poslední den měsíce

Funkce	Popis
ROUND	Zaokrouhlí datum dle zadané „přesnosti“
TRUNC	Vrátí z data pouze rok, měsíc a den.

Příklady

```
SELECT
  NEXT_DAY('21-DUB-14','Čtvrtek') as next,
  ROUND (TO_DATE('24-DUB-14'), 'MONTH') AS round_month,
  ROUND (TO_DATE('24-SRP-14'), 'YEAR') AS round_year,
  TRUNC (TO_DATE('24-DUB-14'), 'MONTH') AS trunc_month,
  TRUNC (TO_DATE('24-DUB-14'), 'YEAR') AS trunc_year
FROM dual;
NEXT      ROUND_MO ROUND_YE TRUNC_MO TRUNC_YE
--------- -------- -------- -------- --------
24.04.14  01.05.14 01.01.15 01.04.14 01.01.14
```

Přístupová práva

V této kapitole budu popisovat jak se připojit k databázi a jak nastavovat uživatelům práva. Jsou to dvě rozdílné, ale úzce související témata. Tato kapitola není ani tak o jazyku SQL, jako spíše o administraci.

Připojení se k databázi

První problém, který budete řešit po nainstalování Postgresu určitě bude, jak se připojit k databázi a jak si vytvořit nějakého uživatele a pro něj schéma. Teď budu popisovat tu část o tom, jak se mohou uživatelé připojit k databázi. Co to je vlastně uživatel, o tom bude řeč v části O uživatelích, skupinách a rolích. Teď vám jen prozradím, abychom si rozuměli, že v Postgresu se uživatelům i skupinám (uživatelů) souhrně říká **role**.

Konfigurační soubor pg_hba.conf

Možnosti připojení k DBMS se řídí nastavením v souboru pg_hba.conf. Jeho umístění najdete v konfiguračním souboru postgresql.conf pod volbou pg_hba.conf.

V Debianu jsem našel oba soubory v adresáři /etc/postgresql/9.1/main/, v OpenSuSE pro změnu v /var/lib/pgsql/data/. Jestli nenajdete konfiguraci ani tam, tak můžete umístění konfiguračního souboru PostgreSQL zjistit SQL příkazem **SHOW config_file;**.

K těmto konfiguračním souborům bude mít povolený přístup pravděpodobně jen superuživatel (uživatel postgres).

Obsah souboru pg_hba.conf

Soubor pg_hba.conf vypadá docela jednoduše. Obsahuje řádky, kde každý řádek popisuje nějaký způsob přihlášení k databázi. Popisuje, kdo se takto může přihlásit, odkud (podle IP adresy), jestli je vyžadováno heslo … Nejlépe se to asi vysvětlí na příkladu:

```
local   all             postgres                                peer
```

Tento řádek říká, že se může k databázi připojit přes unix socket (local), může se připojit ke všem databázím (all), tento přístup je umožněn jen uživateli postgres a jako autentizace se použije jen ověření, že ten kdo se chce přihlásit pod nějakým uživatelským jménem (v tomto případě to může být jenom postgres) má stejné uživatelské jméno v operačním systému (peer).

Socket je speciální soubor, kterým můžete komunikovat s programem, v našem případě s PostgreSQL. Tímto souborem lze komunikovat samozřejmě jen tehdy, pracujete-li na stejném počítači, jako je tento soubor. To platí pro klienta i DBMS, takže to umožňuje jen lokální spojení (klient se připojuje ze stejného počítače na kterém běží DBMS). To je i pro autentizaci typu peer nezbytné.

Spojení přes socket je rychlejší než přes internet pomocí TCP/IP.

Spojení přes socket nefunguje ve Windows!

Takovýto řádek buduete mít určitě v pg_hba.conf a to je přesně důvod, proč si po instalaci zakládáte nové uživatele tak, že se přihlásíte příkazem su jako uživatel postgres a pak se můžete k databázi přihlásit bez zadávání hesla (DBMS díky nastavení peer pouze zkontroluje, že se k ní přihlašuje Linuxový uživatelpostgres).

Uživatel **postgres** má superuživatelská práva (může cokoliv). To jen tak na okraj. A teď na další příklad:

```
host    all             all             127.0.0.1/32            md5
host    all             all             0.0.0.0/0               md5
```

Tentokrát se jedná o připojení skrz TCP/IP (host) ke všem databázím (all), pro všechny uživatele (all), z IP adresy 127.0.0.1. Číslo 32 za lomítkem je maska adresy. Na vysvětlování jak funguje TCP/IP tu není dost místa, takže vám jen povím, že 32 vyžaduje, aby připojení přišlo z úplně stejné IP adresy, 24 znamená, že musí odpovídat první 3 části IP adresy a 0 znamená, že nemusí odpovídat žádná část. To je případ té druhé řádky – proto je IP adresa samá nula. Kdybyste tam napsali jakékoliv čísla, je to fuk, protože kvůli masce 0 se prostě ignorují (ale něco tam být naspáno musí).

Autentiakace md5 vyžaduje, aby klient poslal heslo zašifrované algoritmem MD5.

Toto nastavení není jediné, které je potřeba pro připojení z interentu. V souboru postgresql.conf je volba **listen_addresses**, která určuje z jakých IP adres bude Postgres přijímat připojení. Defaultně je nastavena na localhost, tedy pouze z lokálního

počítače. Můžete jí nastavit na *, což znamená, že bude akceptovat připojení odkudkoliv. Dále se už přístup řídí řádky v pg_hba.conf.

Řádka v pg_hba.conf

1) Nejdříve se na řádku určuje způsob připojení:

Typ připojení

typ	význam
local	Spojení přes socket
host	Spojení přes internet (TCP/IP). Může být SSL, ale nemusí
hostssl	Spojení přes internet (TCP/IP), musí být SSL.
hostnossl	Spojení přes internet (TCP/IP), nesmí být SSL.

2) Databáze může být buď konkrétní jméno schématu (nebo schémat oddělených čárkou (ale bez mezery)), nebo **all**, což znamená libovolnou databázi, nebo **samerole**, což znamená, že jméno databáze je stejné jako jméno přihlašovaného uživatele, nebo že je uživatel členem role tohoto jména.

3) Uživatel může být buď konkrétní jméno role (nebo rolí oddělených čárkou), nebo **all**, což znamená, že to platí pro všechny role.

Pokud k jménu role připojíte +, pak se nemusí přihlašovat přímo uživatel tohoto jména, ale každý kdo je přímo nebo nepřímo člen této role (o tom co to znamená bude řeč v části o rolích).

4) Adresa je IP adresa, odkud se může uživatel připojit. Jak může vypadat jsem už popisoval výše. Adresa se samozřejmě nezadává pro spojení local.

5) Způsobů autentizace je mnoho, ty nejdůležitější viz tabulka:

Způzoby autentizace

Název metody	Popis
trust	Žádná autentizace není potřeba. To si můžete dovolit například na svém osobním notebooku, ke kterému nemá nikdo jiný přístup.
reject	Spojení je odmítnuto. K tomu je dobré vědět, že první „řádek', který vyhovuje požadovanému spojení (podle jména databáze, uživatele a případně IP adresy), rozhoduje o tom, jestli se uživatel připojí nebo ne. Pokud je odmítnut, žádný další řádek už se nekontroluje.
md5	Klientský program musí poslat heslo ve formátu md5.
peer	Ověří, že připojovaný uživatel má stejné uživatelské jméno v operačním systému. Funguje jen pro připojení local.
ident	Podobné jako peer. Funguje ale i pro vzdálené připojení. Uživatelské jméno z operačního systému vzdáleného počítače zjistí kontaktováním „ident serveru" (co je to za server je nad rámec této kapitoly :-). Pokud použijete ident pro spojení local, použije se ve skutečnosti peer autentizace.

O uživatelích, skupinách a rolích

V PostgreSQL před verzí 8.1 jste si mohli založit uživatele (user) a skupiny (group). Uživatel reprezentoval jednoho člověka, který se mohl připojit k databázi. Uživatel mohl být součástí několika skupin. K jedné skupině mohlo být přiřazeno více uživatelů. Skupina i uživatel mohli mít přidělené (granted) práva k databázovým objektům (tabulkám, pohledům, sekvením atd.).

Od verze 8.1 je v Postgresu tzv. **role**. Uživatel se stal rolí s právem LOGIN (právo na připojení k databázi). A skupina je naopak jen role bez práva LOGIN.

Ikdyž máte podle souboru pg_hba.conf právo se připojit, zadáte správně heslo a všechno, tak bez práva LOGIN se stejně nepřipojíte. Připojením k databázi totiž projdete autentizací (ověřením kdo jste), ale právo LOGIN je potřeba pro autorizaci (povolení něco udělat, přihlásit se).

CREATE USER / ROLE

Příkaz CREATE USER slouží k vytvoření nového uživatele, který bude moci k databázím přistupovat. Od Postgresu v. 8.1 je to alias pro příkaz CREATE ROLE *jmeno_role* LOGIN;

Nového uživatele nemůže vytvořit každý, ale pouze ten, kdo k tomu má právo. Po nainstalování PostgreSQL existuje jedinný uživatel, který může vytvářet jiné uživatele, a to uživatel **postgres**.

Uživatel v PostgreSQL není to samé jako uživatel Linuxu, ačkoliv se jejich schoda někdy využívá k autentizaci (viz peer výše). Řekněme, že jste právě po čerstvé instalaci Postgresu a tak chcete vytvořit prvního uživatele tohoto systému, uživatele rimmer. Protože zatím jediný linuxový uživatel, který k tomu má práva, je *postgres*, musíte se připojit k databázi pod tímto uživatelským kontem. Protože neznáte heslo k linuxovému účtu postgres, musíte se přihlásit pomocí programu su.

```
$ su -
Password:
root:~# su - postgres
postgres:~$ psql
psql (9.2.4)
Pro získání nápovědy napište "help".
postgres=#
```

Pokud jste se dostali úspěšně až sem, můžete vytvořit prvního uživatele pomocí příkazu CREATE USER nebo CREATE ROLE. Při vytváření role můžete určit, jaká systémová práva role má mít.

Systémová práva

Právo	Popis
SUPERUSER	Superuživatel, který může všechno. To je případ uživatele postgres.
LOGIN	Možnost připojit se k databázi.
CREATEDB	Možnost vytvořit nové schéma.
CREATEROLE	Možnost vytvořit další role, pravovat je i mazat.
PASSWORD 'heslo'	Toto není právo, ale heslo, pod kterým se uživatel bude přihlašovat, pokud je vyžadována autentizace heslem (md5). Heslo se zadává nezašifrované.

Teď vytvořím uživatele rimmer s právem vytváření nových databází a uživatelů. Nastavím mu heslo na rimmer (opravdu špatné heslo, ale to jen příklad). Existují 2 způsoby, jak to udělat:

```
CREATE USER rimmer WITH CREATEDB CREATEUSER PASSWORD 'rimmer';
CREATE ROLE rimmer WITH LOGIN CREATEDB CREATEUSER PASSWORD 'rimmer';
```

Každý uživatel s právem vytváření dalších uživatelů může neomezeně pracovat s cizími tabulkami (číst je, upravovat, mazat …).

Pokud dáte uživateli právo vytvářet role, má právo i měnit práva rolí (pomocí ALTER ROLE, takže si může sám sobě dát právo vytvářet databáze :-). Alespoň že nemůže měnit superuživatelské účty, na to je potřeba právo SUPERUSER).

Pokud se přihlásíte na počítači jako uživatel stejného jména, budete se nejspíš přihlašovat pomocí peerautentizace, tedy bez nutnosti zadávat heslo. Pokud se ale budete přihlašovat přes síť, bude po vás heslo vyžadováno.

```
petr> psql -U rimmer
psql: FATAL: Peer authentication failed for user "rimmer"
petr> psql -U rimmer -h 127.0.0.1
psql: FATAL: Ident authentication failed for user "rimmer"
petr> psql -U rimmer -h 192.168.1.10
Heslo pro uživatele rimmer:
psql: FATAL: database "rimmer" does not exist
```

Jak vidíte, první dva pokusy o přihlášení nevyšly (linuxový uživatel petr se snažil přihlásit jako databázový uživatel rimmer). Třetí pokus už vyžadoval heslo a proběhl úspěšně. Protože jsem ale nevytvořil databázi rimmer, neměl se uživatel kam připojit. Protože má uživatel právo CREATEROLE, může skoro všechno, takže se může připojit i k libovolné jiné databázi.

```
petr> psql -U rimmer -h 192.168.1.10 rimmer1
Heslo pro uživatele rimmer:
psql (9.2.4)
Pro získání nápovědy napište "help".
rimmer1=#
```

Můžete si taky databázi vytvořit příkazem createdb.

```
petr> createdb -U rimmer -h 192.168.1.10 rimmer
Heslo:
```

Po správném zadání hesla je databáze vytvořena.

Všimněte si, že když se přihlásíte jako superuživatel (postgres, nebo teď rimmer), prompt není ukončen špičatou závorkou >, ale znakem # (sharp).

Uživatel rimmer totiž získal jakési superuživatelské právo (právo CREATEUSER), díky kterému může nakládat s právy ostatních uživatelů. Hned toho využiju k vytvoření uživatele group06.

```
petr> psql -U rimmer -h 192.168.1.10 rimmer
Heslo pro uživatele rimmer:
psql (9.2.4)
Pro získání nápovědy napište "help".
rimmer1=# CREATE USER group06;
CREATE ROLE
```

Seznam existujících rolí můžete získat ze systémové tabulky **pg_roles**. Kupodivu k tomu nemusíte být superuživatel.

```
rimmer=# SELECT rolname, rolcanlogin, rolsuper, rolcreaterole, rolcreatedb,
rolinherit FROM pg_roles;
 rolname  | rolcanlogin | rolsuper | rolcreaterole | rolcreatedb | rolinherit
----------+-------------+----------+---------------+-------------+------------
 postgres | t           | t        | t             | t           | t
 petr     | t           | f        | f             | t           | t
 rimmer   | t           | t        | f             | f           | t
 group06  | t           | f        | f             | f           | t
(4 řádky)
```

Taky můžete použít metapříkaz \du.

ALTER ROLE a DROP ROLE

Příkaz ALTER ROLE slouží ke změně uživatelského účtu postgresu. Jeho použití je stejné jako příkazu CREATE USER. Příkaz DROP USER uživatelský účet smaže.

```
rimmer=# ALTER ROLE group06 WITH CREATEDB PASSWORD 'group06';
ALTER USER
rimmer=# DROP USER group06;
DROP USER
```

Uživatele group06 ještě nemažte, budu ho používat při dalším výkladu.

Pokud byste chtěli práva z role odebrat, pak použijte:

```
ALTER ROLE group06 WITH NOCREATEDB NOCREATEUSER;
```

A nakonec zmíním příkaz pro smazání databáze, který už dobře znáte:

```
petr> dropdb -U rimmer -h 192.168.1.10 rimmer
Heslo:
DROP DATABASE
```

Skupiny

Každá role může fungovat jako skupina. Obvykle role fungující jako skupiny nemají právo LOGIN, ale jinak se od ostatních uživatelů nijak neliší.

Skupině můžete nastavit práva jako uživateli (uživatel a skupina je přeci to samé – role).

Co je ale důležité, skupina může být přidělena (GRANT) uživateli (nebo jiné skupině) jako by to bylo nějaké právo. Všechny práva přidělené „skupině" pak může využívat i uživatel.

Pokud má uživatel nastavené právo **INHERIT**, práva ze skupin jsou ihned aktivní. Pokud toto právno nemá, musí použít příkaz SET ROLE, kterým se přepne do dané role (a má k dispozici pouze práva z této role a žádné jiné).

```
ALTER ROLE group06 WITH INHERIT;
CREATE ROLE dbadmin WITH CREATEDB;
GRANT dbadmin TO group06;
```

V příkladu jsem nejdříve nastavil roli group06 právo INHERIT. Pak jsem vytvořil roli dbadmin s právem vytváření databází. Nakonec jsem tuto roli přidělil roli group06, která od této chvíle může vytvářet databáze bez nutnosti použití příkazu SET ROLE dbadmin; (díky INHERIT).

Skupiny se používají k snadnému přidělování/odebírání skupiny práv mnoha uživatelům. Nejde jen o práva, která jsem zmíňil doposud, ale i o další práva, která popíšu hned dále.

O přístupových právech k databázovým objektům

GRANT (přidání přístupových práv)

Příkaz GRANT slouží k přidělování přístupových práv k databázovým objektům. (Nepleťte si slovo grant se slovem grand - to označuje příslušníka vysoké španělské šlechty :-).

Příkazu GRANT se musí říct jaká přístupová práva se přidělují, kterému uživateli se přidělují a k jaké tabulce (databázovému objektu) se tyto práva přidělují.

V předchozí části jsem si vytvořil dva uživatele: rimmer a group06. V příkladu si uživatel rimmer vytvoří tabulku seznam, ke které bude dávat přístupová práva uživateli group06.

```
-- prihlasen uzivatel rimmer
rimmer=# CREATE TABLE seznam (id SERIAL, nazev VARCHAR(20), cena NUMERIC(5,2));
NOTICE:  CREATE TABLE will create implicit sequence 'seznam_id_seq' for SERIAL
CREATE
```

Přístupová práva můžete přidávat buďto jedné roli (za druhou), nebo jej zpřístupnit všem uživatelům, když použijete speciální roli **PUBLIC**.

Pokud předáváte práva konkrétnímu uživateli/roli, musí tento uživatel existovat.

Možnosti příkazu GRANT zjistíte pomocí metapříkazu \h GRANT. Uvidíte, jaká všechna práva je možné přidělovat. Zajímavá je možnost **ALL**, která přidělí roli všechna práva. Spojením PUBLIC a ALL povolíte všechno všem :-).

V příkladě teď přidělím práva uživateli group06 na čtení (SELECT) tabulky seznam a na vkládání záznamu (INSERT).

Než tak ale učiním, ještě se musím zmínit o jednom důležitém metapříkazu: **\z**. Tento metapříkaz dokáže vypsat přístupová práva k objektům v databázi. Přístupová práva jsou za jménem uživatele/role, ke které se vztahují, a rovnítkem. Jsou označeny jednopísmennými zkratkami, viz následující tabulka. Z ní se také dozvíte jaká že to práva lze nastavovat.

Zkratka	Práva	Uživatel může ...
r	SELECT ("read")	použít příkaz SELECT, a také COPY FROM.
w	UPDATE ("write")	použít příkaz UPDATE. Toto právo je třeba k použití sekvencí (nextval, currval, setval)
a	INSERT ("append")	použít příkaz UPDATE, a také COPY TO
d	DELETE	mazat data příkazem DELETE
D	TRUNCATE	mazat data příkazem TRUNCATE
x	REFERENCES	vytvořit novou tabulku s referencí na tuto tabulku
t	TRIGGER	použít příkaz CREATE TRIGGER
arwdDxt	Všechny práva	Může všechno víše zmíněné.

Za přístupovými právy je za lomítkem jméno role, která přístupová práva přidělila.

```
rimmer=# \z
 public | seznam        | tabulka  | rimmer=arwdDxt/rimmer |
 public | seznam_id_seq | sekvence |                       |

rimmer=# GRANT SELECT, INSERT ON seznam TO group06;
GRANT
rimmer=# \z seznam
 public | seznam        | tabulka  | rimmer=arwdDxt/rimmer+|
        |               |          | group06=ar/rimmer     |
```

Teď se k databázi rimmer přihlásím jako uživatel group06 a pokusím se s tabulkou pracovat.

Znovu upozorňuji, že pokud bude mít uživatel právo vytvářet další uživatele (viz ALTER USER), všechna zde popisovaná přístupová práva se budou míjet účinkem.

```
-- prihlasen uzivatel group06
rimmer=> SELECT * FROM seznam;
 id | nazev | cena
----+-------+------
(0 rows)
rimmer=> INSERT INTO seznam (nazev,cena) VALUES ('abcd',25.2);
ERROR:  seznam_id_seq.nextval: you don't have permissions to set sequence
seznam_id_seq
```

To slovíčko rimmer, které je v příkladu, označuje název databáze, nikoliv uživatele.

Vložení záznamu do tabulky selhalo, neboť group06 nemá přístup k sekvenci seznam_id_seq, která se automaticky mění při každém vložení záznamu v tabulce. Uživtel group06 musí dostat právo úpravy k této sekvenci. Teď už to půjde rychle.

```
-- prihlasen uzivatel rimmer
rimmer=# GRANT UPDATE ON seznam_id_seq TO group06;
GRANT
-- prihlasen uzivatel group06
rimmer=> INSERT INTO seznam (nazev,cena) VALUES ('abcd',25.2);
rimmer=> UPDATE seznam SET cena = 0;
ERROR:  seznam: Permission denied.
```

Na INSERT group06 právo má, na UPDATE ne. Zkusím ještě nastavení práva úpravy pro všechny.

```
-- prihlasen uzivatel rimmer
rimmer=# GRANT UPDATE ON seznam TO PUBLIC;
GRANT
rimmer=# \z
                            Access privileges
 Schema |     Name      |   Type   |   Access privileges   | Column access privileges
--------+---------------+----------+-----------------------+--------------------------
 public | seznam        | table    | rimmer=arwdDxt/rimmer+|
        |               |          | group06=ar/rimmer    +|
        |               |          | =w/rimmer             |
 public | seznam_id_seq | sequence | rimmer=rwU/rimmer    +|
        |               |          | group06=w/rimmer      |
(2 rows)
```

Právo pro public je to, co začíná rovnítkem, bez názvu role před ním.

```
-- prihlasen uzivatel group06
rimmer=> UPDATE seznam SET cena = 0;
UPDATE 1
```

REVOKE (odebrání přístupových práv)

Tak, jak příkaz GRANT slouží k přidělování práv, tak příkaz **REVOKE** slouží k jejich odnímání.
Je třeba zase určit jaká práva se odebírají, komu se odebírají (uživatel nebo PUBLIC) a k jakému objektu v databázi se to vztahuje.
Všechna práva, která jsem v předchozím oddíle přidělil k tabulce seznam pro uživatele group06 zase odeberu a pak ještě odeberu právo UPDATE pro všechny (pro PUBLIC).

```
-- prihlasen uzivatel rimmer
rimmer=# REVOKE ALL ON seznam FROM group06;
REVOKE
rimmer=# \z
                            Access privileges
 Schema |     Name      |   Type   |   Access privileges   | Column access privileges
--------+---------------+----------+-----------------------+--------------------------
 public | seznam        | table    | rimmer=arwdDxt/rimmer+|
        |               |          | =w/rimmer             |
 public | seznam_id_seq | sequence | rimmer=rwU/rimmer    +|
        |               |          | group06=w/rimmer      |
(2 rows)

rimmer=# REVOKE UPDATE ON seznam FROM PUBLIC;
REVOKE
rimmer=# \z
                            Access privileges
 Schema |     Name      |   Type   |   Access privileges   | Column access privileges
--------+---------------+----------+-----------------------+--------------------------
 public | seznam        | table    | rimmer=arwdDxt/rimmer |
 public | seznam_id_seq | sequence | rimmer=rwU/rimmer    +|
        |               |          | group06=w/rimmer      |
```

MySQL

V MySQL jsou věci o dost jednoduší. Neexistují žádné skupiny a nedělá se žádný rozdíl mezi připojením na socket nebo přes internet. Uživatel se vždy autentizuje pomocí jména a hesla.
Po instalaci je k dispozici uživatel **root** (přesněji teda root@localhost, viz dále o uživatelských jménech) který má superuživatelská práva.
Nemá nastavené žádné heslo, takže se může jako root přihlásit kdokoliv, kdo má na počítači uživatelský účet (MySQL neporvádí peer autentizaci).
Jméno se skládá z „username" a z IP adresy, ze které se může připojit. Část IP adresy na konci může být nahrazena % (procentem), které znamená, že se zbytek IP adresy nekontroluje, že se může připojit odkudkoliv. Pokud použijete jen %, znamená to „libovolná IP adresa".

Několik příkladů jak můžete vytvořit uživatele v MySQL:

1. `CREATE USER 'rimmer'@'%';`
2. `CREATE USER rimmer@%;` *-- nefunguje, % neni v uvozovkach*
3. `CREATE USER rimmer;`
4. `CREATE USER rimmer@192.168.1.%;`
5. `CREATE USER rimmer@192.168.% IDENTIFED BY 'heslo';`
6. `CREATE USER rimmer@localhost;`
7. `CREATE USER rimmer@127.0.0.1;`

MySQL příkazy zadávané z klienta mysql se obvykle ukládají do souboru ~/.mysql_history. Takže je tu vážné nebezpečí, že se do tohoto souboru uloží i heslo. Soubor by tedy neměl mít právo číst nikdo jiný než vy.

Příklady na 1 a 3 řádce založí téhož uživatele, se jménem rimmer, který se může připojit odkudkoliv bez zadání hesla. Pátý příklad navíc nastaví uživateli heslo.

Ale pozor! Poslední dva příklady neznamenají to samé. Šestý příklad uživatele se může připojit jen prvním způsobem z následujícího příkladu. 7 uživatel jen tím druhým. Protože 'localhost' a '127.0.0.1' nejsou pro MySQL to samé:

 # mysql -u rimmer -h localhost jmeno_databaze
 # mysql -u rimmer -h 127.0.0.1 jmeno_databaze

Heslo lze změnit uživateli příkazem SET PASSWORD:

```
SET PASSWORD FOR 'rimmer'@'192.168.1.%' PASSWORD('heslo');
```

Existující uživatele zjistíte následujícím příkazem:

```
SELECT User, Host FROM mysql.user;
```

Schéma mysql je systémové schéma, které obsahuje různé tabulky a pohledy na data využívané DBMS MySQL.

Smazání uživatele:

```
DROP USER 'rimmer'@'192.168.1.%';
```

Nastavování autorizace k databázovým objektům se dělá pomocí příkazů GRANT a REVOKE.

```
GRANT SELECT ON rimmer1.trest TO 'group06'@'%';
GRANT ALL ON rimmer1.* TO 'rimmer'@'192.168.1.%';
GRANT ALL ON *.* TO 'rimmer2'@'localhost' IDENTIFIED BY 'heslo';
REVOKE ALL ON rimmer1.* FROM 'rimmer'@'192.168.1.%';
```

Speciální právo **ALL** znamená, že povolujete všechno. Hvědička (*) znamená „všechny schémata" nebo „všechny objekty v databázi" (před tečkou je jméno schématu, za tečkou obvykle jméno tabulky).

Pokud uživatel nexistuje, tak jej příkaz GRANT vytvoří, takže mu můžete (ale nemusíte) příkazem GRANTrovnou nastavit i heslo (viz IDENTIFIED BY 'heslo').

V MySQL můžete zobrazit práva přidělená uživateli takto:

```
SHOW GRANTS FOR 'group06'@'%';
```

Se zjištěním nastavených práv pro tabulku je to trochu horší:

```
mysql> SELECT * FROM tables_priv WHERE Table_name = 'trest';
+------+---------+---------+------------+---------------+---------------------+------------+-------------+
| Host | Db      | User    | Table_name | Grantor       | Timestamp           | Table_priv | Column_priv |
+------+---------+---------+------------+---------------+---------------------+------------+-------------+
| %    | rimmer1 | group06 | trest      | root@localhost | 2014-02-04 19:14:30 | Select     |             |
+------+---------+---------+------------+---------------+---------------------+------------+-------------+
1 row in set (0.00 sec)
```

Další způsoby zjišťování nastavených práv najdete v článku How to Get a List of Permissions of MySQL Users.

MySQL defaultně poslouchá spojení na všech portech, takže není potřeba nic nastavovat. Kdyby vás to ale přeci jen zajímalo, podívejte se na volbu bind-address.

Konfigurační soubor pro MySQL najdete obvykle v /etc/mysql/my.cnf.

SQLite

V SQLite se žádná přístupová práva neřeší a řešit nebudou. SQLite je minimalistická databáze a řešení práv či uživetelů je jedna z věcí, které záměrně ignoruje. Jediný způsob, jak nastavit nějaká práva je nastavení práv k databázovému souboru, v Linuxu například pomocí příkazu chmod.

Oracle

V Oracle se dělá všechno tak nějak jinak, složitěji. Ale nebuďme na něj zlý, Oracle byl jeden z prvních SQL DBMS a ostatní se mohli z jeho chyb poučit :-)

V Oracle se při vytvoření uživatele vytvoří automaticky i jeho schéma (se stejným jménem jako je jméno uživatele). V Oracle se rozlišuje mezi rolí a uživatelem (není to to samé).

Po instalaci Oracle máte k dispozici roli SYS a schéma SYS. Toto schéma obsahuje tabulky a pohledy, které jsou nezbytné pro běh vlatního DBMS. Role SYS je systémová role, tu byste měli tak nějak ignorovat a nechat jí DBMS na hraní.

Dále máte po instalaci k dispozici uživatele SYSTEM, jehož heslo jste vytvářeli během instalace. Tento uživatel má právo na roli DBA, která umožňuje téměř všechno. Ten kdo má roli DBA je, nebojím se to říct, administrátor.

Další administrativní role, která stojí za zmínku, je SYSDBA. Ta umožňuje věci jako je změna znakové sady databáze pomocí ALTER DATABASE, CREATE DATABASE, DROP DATABASE, spouštět a vypínat DBMS atp. Tuto roli můžete někomu přidělit (viz SET ROLE dále), pokud mu chcete něco z těchto věcí umožnit.

Uživatel se vytvoří příkazem CREATE USER:

```
-- prihlasen uzivatel SYSTEM
oracle> CREATE USER group06 IDENTIFIED BY group06;
User created.
```

Tímto příkazem se vytvořil uživatel group06 s heslem group06 a i schéma group06.

Pokus o přihlášení příkazem connect username/passowrd:

```
oracle> connect group06/group06
ERROR:
ORA-01045: user GROUP06 lacks CREATE SESSION privilege; logon denied
```

Pokus selhal, protože uživatel nemá právo CREATE SESSION. Tak mu ho přidělím:

```
-- prihlasen uzivatel SYSTEM
oracle> GRANT CREATE SESSION TO group06;
```

Tak teď už se jako group06 přihlásím. Zkusím vytvořit tabulku:

```
oracle> connect group06/group06
Connected
oracle> CREATE TABLE pokus (id int);
create table pokus (id int)
*
ERROR at line 1:
ORA-01031: insufficient privileges
```

Jak vidíte, uživatel group06 nemá právo na vytvoření tabulky. Nemá vlastně žádná práva, která se mu explicitně nepřiřadí. Takže mu ho přidám:

```
-- prihlasen uzivatel SYSTEM
oracle> GRANT CREATE TABLE TO group06;
-- prihlasen uzivatel GROUP06
create table pokus (id int);
create table pokus (id int)
*
ERROR at line 1:
ORA-01950: no privileges on tablespace 'SYSTEM'
```

Co je to **tablespace**? To je místo, kam se ukládají databázová data. Protože jsem při vytváření uživatele group06 neřekl, do jakého tablespace se mají jeho data ukládat, defaultně se snaží ukládat do tablespace jména SYSTEM.
Tablespace SYSTEM je určené pro systémové věci, takže tam by s uživatelé asi data ukládat neměli. Oracle při instalaci vytvoří automaticky tablespace USERS, který je určený pro uživatele. K tomu se ještě vrátím, teď ukážu příkaz, jak je možné přidělit uživateli nějaké místo v tablespace:

```
-- prihlasen uzivatel SYSTEM
oracle> ALTER USER group06 QUOTA 50m ON SYSTEM;
```

Tímto příkazem jsem přidělil uživateli group06 50 MiB prostoru v tablespace SYSTEM. Neomezené množství prostoru se dá nastavit klíčovým slovem UNLIMITED.

```
-- prihlasen uzivatel GROUP06
oracle> CREATE TABLE pokus (id int);
Table created.
oracle> INSERT INTO pokus VALUES (1);
oracle> DELETE FROM pokus;
1 row deleted.
oracle> DROP TABLE pokus;
Table dropped.
oracle> CREATE TABLE pokus (id int);
oracle> CREATE VIEW wpokus AS select * from pokus;
CREATE VIEW wpokus AS select * from pokus
*
ERROR at line 1:
ORA-01031: insufficient privileges
```

No vida, stačilo právo CREATE TABLE a už můžete s databází docela slušně pracovat. Nemáte ale právo na práci s pohledy (VIEW).
Zadávat každému novému uživateli všechna možná práva je docela otrava (a rádo se na něco zapomene), proto bude lepší přidělit všechna práva nějaké nové ROLI a pak stačí uživateli přidělit jen tuto roli:

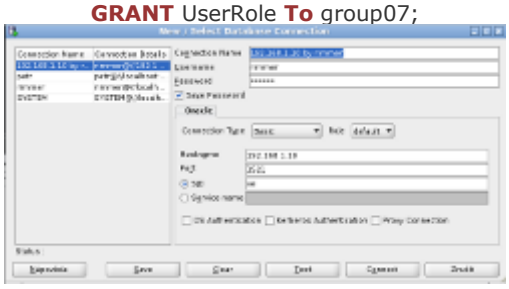
```
-- prihlasen uzivatel SYSTEM
oracle> CREATE ROLE UserRole;
oracle> GRANT CREATE TABLE, CREATE VIEW, CREATE SEQUENCE, CREATE SESSION, CREATE procedure, CREATE syn
onym To UserRole;
oracle> GRANT UserRole To group06;
Grant succeeded.
```

Pokud jste pořád někde přihlášení jako group06, pak se nově nabytá role hned neprojeví. Buď se odhlašte a přihlašte, nebo použijte příkaz SET ROLE.

```
-- prihlasen uzivatel  GROUP06
oracle> CREATE VIEW wpokus AS select * from pokus;
CREATE VIEW wpokus AS select * from pokus
*
ERROR at line 1:
ORA-01031: insufficient privileges
oracle> SET ROLE UserRole;
oracle> CREATE VIEW wpokus AS select * from pokus;
View created.
```

Vytváření uživatele lze zjednodušit, s ohledem na předchozí povídání, takto:

```
-- prihlasen uzivatel SYSTEM
CREATE USER group07 IDENTIFIED BY group07 DEFAULT TABLESPACE users QUOTA UNLIMITED ONusers;
GRANT UserRole To group07;
```



Vzdálené připojení k Oracle v SQL Developeru

K databázi na jiném počítači se můžete připojit (v konzovovém klientovi) příkazem CONNECT. Nastavení v SQL Developeru můžete vidět na obrázku.

```
oracle> connect username/password@host:port /service_name
```

Konkrétně třeba:

```
connect rimmer/rimmer@192.168.1.10:1521 /XE
```

Nezapomeňte, že váš firewall nesmí blokovat TCP port 1521. XE je service name pro databázi Oracle XE.

Heslo uživatele lze změnit takto:

```
ALTER USER group06 IDENTIFIED BY tajneHeslo;
```

Všechny uživatele můžete zjistit následujícím příkazem:

```
oracle> SELECT * FROM all_users;

USERNAME                       USER_ID CREATED
------------------------------ ------- --------
XS$NULL                     2147483638 28.08.11
GROUP06                             50 04.02.14
GROUP07                             51 04.02.14
RIMMER                              49 20.11.13
PETR                                48 31.10.13
APEX_040000                         47 28.08.11
APEX_PUBLIC_USER                    45 28.08.11
FLOWS_FILES                         44 28.08.11
HR                                  43 28.08.11
MDSYS                               42 28.08.11
ANONYMOUS                           35 28.08.11
XDB                                 34 28.08.11
CTXSYS                              32 28.08.11
OUTLN                                9 28.08.11
SYSTEM                               5 28.08.11
SYS                                  0 28.08.11

16 rows selected.
```

Nebo takto:

```
-- prihlasen uzivatel SYSTEM
oracle> SELECT username,  default_tablespace
FROM dba_users
WHERE username in ('RIMMER','GROUP06','GROUP07');

USERNAME             DEFAULT_TABLESPACE
-------------------- ------------------
GROUP06              SYSTEM
GROUP07              USERS
RIMMER               USERS
```

Přidělené role a práva zjistíte takto:

```
-- prihlasen uzivatel SYSTEM
oracle> SELECT * from DBA_ROLE_PRIVS where GRANTEE = 'GROUP07';
GRANTEE  GRANTED_ROLE ADMIN_OPTION DEFAULT_ROLE
-------- ------------ ------------ ------------
GROUP07  USERROLE     NO           YES

oracle> SELECT * from DBA_TAB_PRIVS where GRANTEE = 'GROUP07';

no rows selected
```

Uživatel group07 nemá nastavená žádná práva nad konkrétní tabulkou.

Vytváření uživatelských funkcí

O vytváření funkcí by se mohla napsat samostatná kniha, neboť se v něm, krom jiného, využívá programování. Nechci se pouštět do vysvětování programování (od toho tu jsou jiné tutoriály), proto uvedu jen pár jednoduchých příkladů vytvoření funkce a pokud vás to zaujme, můžete si zbytek dostudovat z oficiální dokumentace.

Toto je (prozatím?) poslední kapitola, takže vám přeji hodně zábavy při využívání všech vašich nabytých znalostí.

- CREATE FUNCTION
- DROP FUNCTION
  - Jazyk SQL
    - Přetěžování funkcí
  - Jazyk PL/pgSQL
  - Jazyk PL/Python
    - MySQL
    - SQLite
    - Oracle
  - Funkce a procedury
    - Pole
    - Balíčky

CREATE FUNCTION

Funkce (přesněji user defined functions) se v PostgreSQL vytvářejí příkazem **CREATE FUNCTION**.

Zjednodušená syntaxe příkazu je následující:

```
CREATE [OR REPLACE] FUNCTION jmeno_funkce ([typ_argumentu [, ]])
RETURNS typ_navratove_hodnoty
AS 'definice funkce'
LANGUAGE jmeno_jazyka;
```

Příkazem **CREATE FUNCTION** jmeno_funkce vytvoříte funkci jen v případě, že funkce tohoto jména ještě neexistuje. Pokud použijete příkaz CREATE **OR REPLACE** FUNCTION jmeno_funkce, bude existující funkce stejného jména přepsána. (Přepsat můžete jenom funkce vytvořené pomocí CREATE FUNCTION, ne standardní funkce postgresu).

K čemu slouží jmeno_funkce je snad jasné. Argument (nebo argumenty) funkce se předávají při použití funkce v závorce, oddělené čárkou. Během deklarace funkce musíte určit typ_argumentu. (Například real, string atp.) Funkce nemusí mít žádné argumenty, pak se při deklaraci a použití funkce musí použít prázdné závorky.

Za závorkami následuje klíčové slovo **RETURNS** a typ návratové hodnoty funkce (real, string atp.). K tomu, k čemu je návratová hodnota, se ještě dostanu (programátoři jistě vědí).

Za klíčovým slovem **AS** je uvedeno tělo funkce. Tělo funkce je v podstatě popis činnosti, kterou má funkce vykonávat. Jak takový popis činosti vypadá, záleží na programovacím jazyku, který k popisu činosti použijete. (V PostgreSQL máte více možností).

Tělo funkce se zadává jako string (fakt, nekecám, taky mě to překvapilo), proto musí být uzavřené v apostrofech (nebo do čeho se text dá uzavřít).

Za klíčovým slovem **LANGUAGE** je název jazyka, který byl použit pro definici těla funkce. Asi vás už napadlo, že je tomu tak proto, že těch jazyků je více. Například 'internal', C, **SQL**, **PL/pgSQL** a další. Běžný uživatel má k dispozici jazyk **SQL** popsaný níže.

Funkce lze **přetěžovat**. To znamená, že je možné vytvořit více funkcí stejného jména, které se ovšem liší počtem a/nebo typem argumentů.

DROP FUNCTION

Protože je možné funkce přetěžovat, musí se příkazu **DROP FUNCTION** předat nejen název funkce, ale i typy argumentů, aby podle nich mohl jednoznačně identifikovat správnou verzi funkce ke smazání.

```
DROP FUNCTION jmeno_funkce ([typ_argumentu [, typ_argumentu, ...]]);
```

Samozřejmě můžete odstranit funkci jen pokud na to máte práva.

Můžete mazat jen uživatelské funkce (user defined functions), tedy funkce vytvořené pomocí CREATE FUNCTION, nikoliv interní funkce Postgresu.

Smaže se ta funkce zadaného jména, která má shodný počet a typ argumetnů.

Jazyk SQL

Začnu příkladem. Vytvořím funkci pomocí jazyka SQL. To znamená, že za klíčovým slovem LANGUAGE bude jazyk SQL a tělo funkce za klíčovým slovem AS **se bude skládat jen z příkazů SQL**. To je asi ten nejjednodušší způsob, jak napsat funkci :-).

Návratovou hodnotou (RETURN) **funkce jazyka SQL je výsledek posledního SQL příkazu.**

Funkce, kterou vytvořím, bude pouze vracet součet dvou čísel typu integer (Tedy návratová hodnota bude typu integer a argumenty funkce budou dvě čísla typu integer).

```
rimmer1=> CREATE FUNCTION soucet (integer, integer)
RETURNS integer
AS '
SELECT $1 + $2;
'
LANGUAGE SQL;
```

V příkladu si všimněte, jak se argumenty v tělu funkce používají. První argument je označen $1, druhý $2 atd. Návratovou hodnotou je výsledek posledního (v příkladu jediného) SELECTu.

```
rimmer1=> SELECT soucet(1,1);
 soucet
--------
      2
(1 řádka)
```

Další ukázka je převzata z dokumentace PostgreSQL. Všimněte si, že místo apostrofů je tělo funkce uzavřeno v $$. (To je v PostgreSQL další způsob, jak je možné ohraničovat text). To je preferovaný způsob, protože umožňuje v tělu funkce používat apostrofy bez nutnosti jejich escapování (pomocí zpětného lomítka).

```
CREATE FUNCTION tp1 (integer, numeric) RETURNS numeric AS $$
UPDATE bank
SET balance = balance - $2
WHERE accountno = $1;
SELECT balance FROM bank WHERE accountno = $1;
$$ LANGUAGE SQL;
```

Funkci se vám nepodaří vytvořit, pokud neexistuje tabulka bank. (Například CREATE TABLE bank (accountno INT, balance NUMERIC);)

Další informace o jazyku SQL najdete v dokumentaci.

Přetěžování funkcí

Funkci soucet z předchozího příkladu přetížím.

O funkci se říká že je přetížená, pokud existuje více funkcí stejného jména, které se odlišují počtem nebo typem argumentů. V příkladu se budou lišit typem argumentů.

Změna typu návratové hodnoty pro přetížení funkce nestačí. Při volání funkce je jasné, kolik a jakého typu jsou argumenty, ale návratová hodnota se z volání funkce odvodit nedá. Takže by se ani nedalo vybrat správnou verzi funkce.

```
rimmer1=> CREATE FUNCTION soucet (float, integer)
RETURNS float
AS '
SELECT $1 + $2;'
LANGUAGE 'sql';
```

Funkce se mohou volat následovně:

```
rimmer1=> SELECT soucet (3,-1),soucet (3.4,-1),soucet(-1,3.4);
 soucet | soucet | soucet
--------+--------+--------
```

```
 2 |   2.4 |     2
(1 row)
```

V prvním případě byla volána funkce soucet(integer,integer) ve druhém případě soucet(float, integer).
~~Protože není definovaná funkce soucet(integer,float), byla v třetím případě použita funkce soucet(integer, integer) a číslo typu float (3.4) bylo Postgresem převedeno na typ integer (3).~~
~~Někdy by podobné automatické konverze mohli být nebezpečné, proto je třeba si na to dávat pozor.~~
UPDATE: Nově se teď v Postgresu třetí volání (s nesprávnými datovými typy) vyvolá chyba. Argumenty musíte explicitně přetypovat:

```
rimmer1=> SELECT soucet(-1, CAST(3.4 AS INTEGER));
 soucet
--------
      2
(1 řádka)
```

Přetěžování funkcí funguje stejně i v dalších jazycích (PL/pgSQL atd.).
Funkce můžete vypsat pomocí metapříkazu \df.

Jazyk PL/pgSQL

Jazyk SQL je omezený pouze na SQL příkazy (SELECT,UPDATE, DELETE atp.).
Jazyk PL/pgSQL naproti tomu umí takové věci, jako deklaraci proměnných, cykly, podmínky, odchytávání výjimek.
Pokud se někdy rozhodnete napsat malinko složitější funkci, pravděpodobně sáhnete pro jazyku PL/pgSQL.
Struktura těla funkce v jazyku PL/pgSQL vypadá takto:

```
[ <<?php ?><label>> ]
[ DECLARE
  declarations ]
BEGIN
  statements
END [ label ];
```

V části DECLARE se deklarují proměnné, mezi BEGIN a END je pak program. (BEGIN/END v tomto případě nemají nic společného s transakcemi). label je libovolný název bloku, který můžete použít pro kvalifikaci proměnných pomcí tečkové notace.

```
CREATE FUNCTION suma(int[]) RETURNS int AS $$
DECLARE
  s int := 0;
  x int;
BEGIN
FOREACH x IN ARRAY $1 LOOP
  s := s + x;
END LOOP;
RETURN s;
END;
$$ LANGUAGE plpgsql;
rimmer1=>  SELECT suma('{3,3,5}');
-- nebo lépe SELECT suma(CAST('{3,3,5}' AS integer[]));
 suma
------
   11
(1 řádka)
```

Jazyk PL/Python

S pl/pgSQL byste si měli vystačit, ale někdy narazíte na to, že je to velice jednoduchý programovací jazyk a vám by se hodil nějaký komplexnější. K tomu můžete využít například perl, nebo python.
Abyste mohli použít jazyk PL/Python, musíte si ho nejdřív pro danou databázi „aktivovat" příkazem CREATE EXTENSION. (Musíte na to taky mít příslušná práva).

```
rimmer1=# CREATE EXTENSION plpythonu;
ERROR:  could not open extension control file "/usr/share/postgresql92/extension/plpythonu.control": Adresář nebo soubor neexistuje
```

Tohle na mě vyzkočilo, když jsem to zkoušel v OpenSUSE. Extension plpythyonu neexistovala, musel jsem si jí extra doinstalovat:

```
root# sudo zypper install postgresql-plpython
```

Teď už to šlo (pod superuživatelským účtem postgres):

```
-- jako uživatel postgres
rimmer1=# CREATE EXTENSION plpythonu;
CREATE EXTENSION
```

Metapříkazem \dL si můžete nechat vypsat aktivní jazyky:

```
rimmer1=> \dL
                                Seznam jazyků
   Jméno   | Vlastník | Důvěryhodný |                Popis
-----------+----------+-------------+-------------------------------------
 plpgsql   | postgres | t           | PL/pgSQL procedural language
 plpythonu | postgres | f           | PL/PythonU untrusted procedural language
(2 řádky)
```

Jazyk plpythonu je označen jako nedůvěryhodný, takže jej běžný uživatel nemůže používat. Buď musí míst superuživatelská práva, nebo musíte změnit důveryhodnost jazyka na true.

```
CREATE FUNCTION pymax (a integer, b integer)
RETURNS integer
AS $$
if a > b:
 return a
return b
```

```
$$ LANGUAGE plpythonu;

ERROR:  permission denied for language plpythonu
```

Jako nedůvěryhodný je plpythonu označen proto, protože s ním může nekalý uživatel narušit běh serveru (například jeho vytížením). Když vím, že uživatelé mého serveru jsou důvěryhodní, změním plpythonu na důvěryhodný:

```
-- jako uživatel postgres
rimmer1=> SELECT * FROM pg_language WHERE lanname= 'plpythonu';
  lanname  | lanowner | lanispl | lanpltrusted | lanplcallfoid | laninline | lanvalidator | lanacl
-----------+----------+---------+--------------+---------------+-----------+--------------+--------
 plpythonu |       10 | t       | f            |         74544 |     74545 |        74546 |
(1 řádka)
rimmer1=# UPDATE pg_language SET lanpltrusted = true WHERE lanname = 'plpythonu';
UPDATE 1
-- ted muze pouzivat python kazdy
```

Jinou možností by bylo nastavit jen konkrétnímu uživateli superuživatelská práva:

```
-- jako uživatel postgres
rimmer1=# ALTER USER petr WITH superuser;
ALTER ROLE
-- ted je petr superuser
```

Příkaz **DO** umožňuje spustit tělo funkce, bez toho, aby se musela funkce definovat. Toho se dá využít pro ověření funkčnosti jazyka:

```
rimmer1=> DO $$
import sys
plpy.info (sys.version_info); return None
$$ LANGUAGE plpythonu;
INFO:  sys.version_info(major=2, minor=7, micro=8, releaselevel='final', serial=0)
KONTEXT:  PL/Python anonymní blok kódu
DO
```

Modul plpy je vždy automaticky importovaný. Obsahuje užitečné metody jako je info, která zobrazí INFO hlášení (klasické pythonovské metody pro výpis, jako je print, nemůžete použít).

MySQL

V MySQL neexistují „jazyky". Máte k dispozici jen jeden „jazyk", který můžete používat s CREATE FUNCTIONpro tělo funkce. Nejvíce se podobá postgresovskému jazyku SQL – prostě jen spouštíte SQL příkazy. MySQL má ve svých SQL příkazech i různé řídící konstrukce (IF, LOOP, WHILE, RETURN) atp., které můžete používat ve funkcích a procedůrách, takže se psaní funkcí vlastně více podobá jazyku pl/pgSQL.

MySQL má kromě CREATE FUNCTION ještě **CREATE PROCEDURE**.

Procedůra se volá pomocí **CALL** nazvev_procedury(); a nemá návratovou hodnotu (může vracet hodnoty pomocí výstupních parametrů, ale to je nad rámec této kapitoly :-).

Funkce naproti tomu mohou vracet hodnotu, mohou být volány jako jakékoliv jiné funkce MySQL v SQL příkazech.

V MySQL jsou výstupem procedury všechny výstupy ze SELECTů spuštěných v těle procedury (ne jen ten poslední, jako v Postgresu). Né každý klient se s tím dokáže poprat, takže se takovým opičárnám raději vyhněte.

Funkce vracejí hodnotu pomocí klíčového slova **RETURN**.

V MySQL se nepíše tělo funkce jako textový řetězec, takže je trochu problém s ukončováním SQL příkazů v těle versus ukončení definice funkce. Obvykle totiž klient (program mysql), když narazí na středník, pošle příkaz serveru. To by způsobilo odeslání začátku funkce a prvního příkazu v těle funkce ukončeného středníkem. Zbytek funkce (resp. další příkazy v těle funkce ukončené středníkem) by se posílaly na server jako další nesouvisející příkazy. Tak tak to prostě nejde.

Obejít se to dá změnou ukončovače (DELIMITER) ze středníku na něco jiného. Obyvkle se používají lomítka**//**.

Argumenty funkce/procedury musí mít jména (v PostgreSQL mohou a nemusí).

```
mysql> DELIMITER //
mysql> CREATE FUNCTION soucet (p1 integer, p2 integer)
RETURNS integer
DETERMINISTIC
BEGIN
RETURN p1 + p2;
END
//
mysql> DELIMITER ;
```

Klíčové slovo **DETERMINISTIC** říká, že funkce pro stejné argumenty vrací vždy stejný výsledek. Další možností je NO DETERMINISTIC. (Tato informace je důležitá pro optimalizaci dotazů.)

MySQL nepodporuje konstrukci CREATE **OR REPLACE** FUNCTION;. (Místo toho umí DROP FUNCTION IF EXISTS.)

V MySQL není možné funkce přetěžovat. Proto se při mazání funkce nemusí uvádět parametry.

```
mysql> SELECT soucet(3,-4);
+--------------+
| soucet(3,-4) |
+--------------+
|           -1 |
+--------------+
1 row in set (0.00 sec)
mysql> DROP FUNCTION soucet;
Query OK, 0 rows affected (0.04 sec)
```

Ještě ukázka vytvoření, použití a smazání procedury:

```
mysql> DELIMITER //
mysql> CREATE PROCEDURE soucet (p1 integer, p2 integer)
DETERMINISTIC
BEGIN
SELECT p1 + p2;
```

```
END
//
mysql> DELIMITER ;
myqsl> CALL soucet(3,-4);
+---------+
| p1 + p2 |
+---------+
|      -1 |
+---------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> DROP PROCEDURE IF EXISTS soucet;
Query OK, 0 rows affected (0.00 sec)
```

SQLite

V SQLite nelze vytvořit uživatelské funkce pomocí CREATE FUNCTION. Můžete si jen doprogramovat funkce pomocí jazyka C, ale to už je nad rámec tohoto tutoriálu.

Oracle

Jazyk pl/pgSQL vychází z Oracle jazyka PL/SQL. Z toho vyplývá, že vytváření funkcí v Oracle je hodně podobné. Můžete se podívat do dokumentace PostgreSQL na Porting from Oracle PL/SQL, kde jsou popsány základní rozdíly mezi pl/pgSQL a PL/SQL. Nějaké to programování v Oracle si můžete prohlédnout v kapitole o triggerech.
Teď jen ve stručnosti ukáži, jak by se daly v Oracle řešit příklady popsané v předchozí části o PostgreSQL. Začnu funkcí soucet.

```
CREATE FUNCTION soucet (i integer,j integer)
RETURN integer
IS
BEGIN
RETURN i + j;
END;
/
```

Místo AS se píše v Oracle IS, místo RETURNS je před IS RETURN, tělo se nezadává jako text, argumenty mají jméno (*i* a *j*) a na konci se neurčuje použitý programovací jazyk, protože v Oracle je jen jeden – PL/SQL.
Postgres taky umožňuje argumenty funkce pojmenovat. Přečtěte si dokumentaci :-).
Použití:

```
oracle> SELECT soucet(1,1) FROM dual;

SOUCET(1,1)
-----------
          2
```

Funkce a procedury

Začněme přípravou tabulky a ukázkou funkce tp1:

```
CREATE TABLE bank(balance numeric, accountno integer);
INSERT INTO bank VALUES(1000,1);

CREATE OR REPLACE FUNCTION tp1 (a integer, b numeric)
RETURN numeric
IS
r numeric;
BEGIN
UPDATE bank SET balance = balance - b
WHERE accountno = a;
SELECT balance
INTO r
FROM bank WHERE accountno = a;
RETURN r;
END;
/
```

Pokus o použití:

```
oracle> SELECT tp1(1,600) FROM DUAL;
ORA-14551: cannot perform a DML operation inside a query
```

Oracle nepovoluje spouštět DML (Data modification language) příkazy ve funkcích, tedy v našem případěUPDATE, uvnitř jiného SQL dotazu. Je to něco kvůli transakcím, ale ono obecně je to blbý nápad mít funkci s „side efektem". Obejít se to dá tak, že funkci zavoláte následujícím způsobem:

```
oracle> SET SERVEROUTPUT ON
oracle> DECLARE
r numeric;
BEGIN
r:=tp1(1,600);
DBMS_OUTPUT.put_line (r);
END;
/
400
```

Funkce put_line z balíčku DBMS_OUTPUT vypíše svůj argument na obrazovku. Tedy v případě, že je nastaven SERVEROUTPUT na ON.

Pokud chcete používat DML, většinou se na to v Oracle PL/SQL používají **procedury**. Procedura je to samé jako funkce, jenom nemá návratovou hodnotu a volá se pomocí EXEC (ne SELECTem).

```
CREATE OR REPLACE PROCEDURE tp2 (a integer, b numeric)
IS
BEGIN
UPDATE bank SET balance = balance - b
WHERE accountno = a;
END;
/
```

Spuštění procedury pomocí EXEC:

```
oracle> SELECT * FROM bank;
   BALANCE  ACCOUNTNO
---------- ----------
       400          1

oracle> EXEC tp2(1,200);
PL/SQL procedure successfully completed.

oracle> SELECT * FROM bank;
   BALANCE  ACCOUNTNO
---------- ----------
       200          1
```

Mnohem hezčí, že? :-) Dokonce můžete místo EXEC použít CALL, jako v MySQL.

Pole

Pokud jde o práci s polem, v Oracle je potřeba si nejdřív definovat vlastní typ. Následující příklad vytvoří datový typ se jménem intArrayTyp, který bude obsahovat **až** 200 hodnot typu integer.

```
CREATE OR REPLACE TYPE intArrayTyp AS VARRAY(200) OF integer;
/
```

Datový typ lze smazat příkazem DROP TYPE jmeno_typu;

LOOP se používá v Oracle trochu jinak než v Postgresu, jinak je funkce suma hodně podobná:

```
CREATE OR REPLACE FUNCTION suma(a intArrayTyp)
RETURN integer
IS
s integer := 0;
i integer;
BEGIN
FOR i IN 1..a.count LOOP
s := s + a(i);
END LOOP;
RETURN s;
END;
/
```

S použitím už je to horší. V příkladu vidíte, jak se v oracle vytvoří typ (řádka 4), nastaví se, kolik může obsahovat hodnot (řádka 5) a pak se pole hodnotami naplní (řádky 6 a 7).

Pokud byste vynechali volání na řádce 5, nebo mu zkusili dát čílo větší než našich deklarovaných 200 položek, dojde k chybě.

```
1.  DECLARE
2.  v_t intArrayTyp;
3.  BEGIN
4.  v_t := intArrayTyp();
5.  v_t.extend(2);
6.  v_t(1) := 1;
7.  v_t(2) := 3;
8.  DBMS_OUTPUT.put_line (suma(v_t));
9.  END;
10. /
```

Na výstup se vypíše 4.

Balíčky

Balíčky (packages) jsou databázové objekty, které seskupují funkce, procedury, typy, proměnné atp. Můžete si je představit jako třeba struktury v jazyku C, nebo record v Pascalu. V Postgresu balíčky neexistují, místo nich se doporučuje používat schema. Důvod, proč se tu o balíčcích zmiňuji je hlavně ten, že v Oracle je možné přetěžovat (overload) funkce jen v nich.

Kromě balíčků, které si naprogramujete sami, existují balíčky, které jsou součástí Oracle od jeho instalace. S jedním z nich už jste se setkali – s balíčekem DBMS_OUTPUT, který obsahuje funkci put_line (a spousty dalších).

Dalším takovým balíčkem je balíček STANDARD. Ten má tu zvláštnost, že když chcete zavolat jeho funkce, nemusíte je kvalifikovat jeho jménem. Takže například volání STANDARD.ABS(-5) lze zapat jako ABS(-5).

Všechny objekty z balíčku STANDARD můžete vypsat takto:

```
oracle> SELECT procedure_name
FROM SYS.all_procedures
WHERE object_name = 'STANDARD' AND procedure_name NOT LIKE '%SYS$%'
ORDER BY procedure_name;

PROCEDURE_NAME
------------------------------
ABS
ABS
ABS
ABS
```

```
ACOS
ACOS
ADD_MONTHS
ADD_MONTHS
ASCII
...
383 rows selected.
```

Všiměte si, že funkce ABS je tam několikrát. To právě kvůli tomu, že je přetížená (existuje více verzí s různým počtem a druhem argumentů).

Kvůli optimalizaci rozděluje Oracle vytvoření balíčku do dvou kroků. V prvním se deklaruje, co balíček obsahuje. Deklaruji tedy balíček matika se dvěmi verzemi funkce soucet.

```
CREATE OR REPLACE PACKAGE matika AS
FUNCTION soucet (a integer,b integer) RETURN integer;
FUNCTION soucet (a float, b integer) RETURN float;
end;
/
```

V druhém kroku definuji těla funkcí (BODY).

```
CREATE OR REPLACE PACKAGE BODY matika AS
function soucet (a integer,b integer) return integer
is
begin
return a + b;
end;
function soucet (a float, b integer) return float
is
begin
return a + b;
end;

END;
/
```

Hurá. Teď už je jen použít:

```
oracle> SELECT soucet (3,-1),soucet (3.4,-1),soucet(-1,3.4) FROM dual;

SOUCET(3,-1) SOUCET(3.4,-1) SOUCET(-1,3.4)
------------ -------------- --------------
           2              2              2
```

Aha, tak to nevyšlo. Jen jsem 3x zavolal funkci soucet, kterou jsem definoval dříve bez balíčku. Nesmí se zapomenout na kvalifikaci názvem balíčku.

```
oracle> SELECT matika.soucet (3,-1),matika.soucet (3.4,-1),matika.soucet(-1,3.4) FROMdual;
SELECT matika.soucet (3,-1),matika.soucet (3.4,-1),matika.soucet(-1,3.4) FROM dual
                                                   *
ERROR at line 1:
ORA-06553: PLS-307: too many declarations of 'SOUCET' match this call
```

Tak bohužel, tohle v Oracle nepůjde. Oracle nedokáže rozlišit float od integeru. Přestože dovolil vytvořit funkce, které se od sebe lišily jen v těchto typech parametrů. No, ale abyste mi věřili, že se funkce dají přetěžovat, ukáži příklad s typem varchar2 místo float. Ten už snad od integer odliší.

```
DROP PACKAGE matika;
CREATE OR REPLACE PACKAGE matika AS
function soucet (a integer,b integer) return integer;
function soucet (a varchar2, b integer) return number;
end;
/

CREATE OR REPLACE PACKAGE BODY matika AS
function soucet (a integer,b integer) return integer
is
begin
RETURN a + b;
end;
function soucet (a varchar2, b integer) return number
is
begin
RETURN TO_NUMBER(a) + b;
end;
END;
/
```

Chvilka napětí, jak to dopadne:

```
oracle> SELECT matika.soucet (3,-1),matika.soucet (3.4,-1),matika.soucet(-1,3.4) FROMdual;
MATIKA.SOUCET(3,-1) MATIKA.SOUCET(3.4,-1) MATIKA.SOUCET(-1,3.4)
------------------- --------------------- ---------------------
                  2                     2                     2
```

První pokus ve všech příkladech volal verzi funkce s objema parametry typu integer. Teď zkusím při druhém volání předat jako první parametr text. Nejdřív ale pro jistotu nastavím oddělovače desetinných míst a tisíců, aby náhodou Oracle neočekával číslo v českém formátu (tedy s desetinnou čárkou, místo tečky).

První příkaz z příkladu nastavuje jako oddělovač desetin tečku a jako oddělovač tisíců čárku. Je to podbné, jako nastavování NLS_DATE_LANGUAGE při konverzi data. Tentokrát jsem ale použil nastavení pro celé sezení, nejen pro volání konverzní funkce.

```
oracle> ALTER SESSION SET NLS_NUMERIC_CHARACTERS = ".,";
Session altered.

oracle> SELECT matika.soucet (3,-1),matika.soucet ('3.4',-1),matika.soucet(-1,3.4) FROM dual;
MATIKA.SOUCET(3,-1) MATIKA.SOUCET('3.4',-1) MATIKA.SOUCET(-1,3.4)
------------------- ----------------------- --------------------
                  2                     2.4                    2
```

No vida, toto už vyšlo. Ale neradujte se předčasně.

```
oracle> SELECT matika.soucet (3,'-1') FROM dual;
ORA-06553: PLS-307: too many declarations of 'SOUCET' match this call
```

V tomto posledním příkladu si Oracle zase neporadil. Člověk by čekal, že překonvertuje '-1' na integer (jako to udělal v třetím volání v předchozím příkladě s floatem), ale tentokrát se mu to nějak nelíbí.

Rozdíly mezi Postgresem a Oraclem jsou na první pohled malé, ale někdy mohou být velmi zapeklité. V této kapitole jsem vám ukázal něco málo z možností programování. Pokud vás toto téma zaujalo (a já doufám že ano), nastudujte si zbytek z dokumentace, je toho ještě hodně zajímavého, co můžete objevit :-).

Materializované pohledy

Materializované pohledy jsou databázové objekty, podobně jako „nematerializované“pohledy. Jenom se výsledek SQL dotazu někam uloží. Dotaz na materializovaný pohled je proto mnohem rychlejší (data už jsou připravena), ale zase nemusí být aktuální (data se připravují na požádání). A to je zhruba všechno důležité, co se dá o materializovaných pohledech říci.

Příprava tabulek

No dobře, můžu o materializovaných pohledech ještě něco napsat. A ukázat pár příkladů. V této kapitole budu ukazovat příklady na tabulkách host,page a referrer, jejichž definice máte pro všechny databáze níže. Zároveň jsem vám připravil skripty pro nahrání dat do tabulek.

```
DROP TABLE IF EXISTS referrer;
DROP TABLE IF EXISTS page;
DROP TABLE IF EXISTS host;
DROP TYPE error_enum;
CREATE TYPE error_enum AS ENUM ('no','yes','connection');


CREATE TABLE host (
id SERIAL PRIMARY KEY NOT NULL,
host VARCHAR(255) NOT NULL UNIQUE,
domain VARCHAR(50),
finded timestamp without time zone,
checked_pages_count integer DEFAULT 0 NOT NULL
);

CREATE TABLE page (
id SERIAL PRIMARY KEY NOT NULL,
page VARCHAR(512) NOT NULL,
port integer DEFAULT 80 NOT NULL,
host_id integer  REFERENCES host(id),
finded timestamp without time zone,
checked boolean DEFAULT false NOT NULL,
checked_at timestamp without time zone,
body text,
finded_in integer,
schema VARCHAR(10) DEFAULT 'http' NOT NULL,
contenttype VARCHAR(50) DEFAULT '' NOT NULL,
charset VARCHAR(50) DEFAULT '' NOT NULL,
base VARCHAR(255) DEFAULT '',
error error_enum DEFAULT 'no' NOT NULL,
CONSTRAINT page_finded_in_fkey FOREIGN KEY (finded_in) REFERENCES page(id),
CONSTRAINT page2_check1 CHECK ((checked_at IS NULL) OR (checked = true))
```

```
);

CREATE TABLE referrer (
id SERIAL PRIMARY KEY NOT NULL,
referrer_id integer NOT NULL REFERENCES page(id),
page_id integer NOT NULL REFERENCES page(id)
);

CREATE UNIQUE INDEX referrer_uix ON referrer (referrer_id, page_id);
```

mview-psql.sql.zip

Popis tabulek

Co tabulky obsahují není pro tuto kapitolu nijak důležité. Ani není důležité jak se používá datový typ enum, nebo drobné rozdíly mezi definicemi tabulek v různých databázích. Jsou to tabulky, které jsem převzal z jednoho projektu, kde jsem se snažil stáhnout obsah celého internetu :-) a náhodou jsem se rozhodl, že je použiji pro tuto kapitolu.

V tabulce host jsou nejdůležitejší sloupce host (například www.sallyx.org) a domain (například cz, org, info atp.).

V tabulce page jsou to soupce page (například /, /sally/, /sally/psql/materializovane-pohledy.phpatp.) a host_id (odkaz na doménu).

Do tabulky referrer se pak zaznamenává, v jaké stránce (referrer_id) byla nalezena jaká stránka (page_id). A samozřejmě vyčtete i obráceně, jaké stránky byly nalezeny ve nějaké stránce, která vás zajímá.

Postgres

Materializovaný pohled se vytváří jako normální pohled, jen uvedete navíc slovíčko MATERIALIZED. A můžete ještě připsat WITH NO DATA, pokud nechcete, aby se pohled hned materializoval (aby se provedl SQL dotaz pohledu a výsledek se kamsi uložil).

Vytvořím pohled, který mi najde stránky, na které je nejčastěji odkazováno.

```
rimmer1=> CREATE MATERIALIZED VIEW most_referred_pages
AS
SELECT referrer.page_id, count(*) AS referred
FROM referrer
GROUP BY referrer.page_id
WITH NO DATA;

SELECT 0
```

Protože jsem použil WITH NO DATA, bude pohled prázdný, resp. neinicializovaný.

```
rimmer1=> SELECT * from most_referred_pages;
ERROR:  materialized view "most_referred_pages" has not been populated
DOPORUČENÍ:  Use the REFRESH MATERIALIZED VIEW command.
```

K naplnění pohledu se používá příkaz REFRESH MATERIALIZED VIEW nazev_pohledu; Po jeho naplnění si mohu zobrazit top 10 nejoblíbenějších stránek.

```
rimmer1=> REFRESH MATERIALIZED VIEW  most_referred_pages;
REFRESH MATERIALIZED VIEW

rimmer1=> SELECT * FROM most_referred_pages ORDER BY referred DESC limit 10;
 page_id | referred
---------+----------
    9776 |       12
    9775 |       11
    9840 |       10
    9990 |       10
    9839 |       10
    9774 |        9
    9841 |        8
    9838 |        8
   10032 |        8
    9935 |        7
```

Nejčastěji odkazovaná stránka je stránka s ID 9776 :-).

Materializované pohledy se automaticky nerefreshují. To znamená, že pokud změním cokoliv v tabulkách, ze kterých se materializovaný pohled vytváří, nic to na jeho obashu nezmění, dokud nezavoláte REFRESH MATERIALIZED VIEW. Tento příkaz obsah materializovaného pohledu smaže, znovu zavolá SQL dotaz a pohled materializuje. Vše je součástí jedné transakce a tak se nemůže stát, že by si někdo v mezičase stihnul přečíst prázdný (zrovna smazaný) pohled před jeho opětovným naplněním.

Materializované pohledy se hodí především tam, kde nepotřebujete aktuální data, ale chcete je rychle. (Například pro statistiky za minulý týden atp.). Případně se tak dají jednoduše replikovat data mezi různými servery (to už je ale nad rámec tohoto kurzu).

Podívejte se, jak si naplánuje Postgres práci pro následující SQL dotaz, který se snaží zjistit podrobnosi o top 10 odkazovaných stránkách:

```
rimmer1=> EXPLAIN
SELECT referred, page.schema, host.host, page.page
FROM most_referred_pages
JOIN page ON page.id = page_id
JOIN host on host.id = host_id
ORDER BY referred DESC
LIMIT 10;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=283.93..283.95 rows=10 width=52)
   ->  Sort  (cost=283.93..289.75 rows=2328 width=52)
         Sort Key: most_referred_pages.referred
         ->  Hash Join  (cost=131.41..233.62 rows=2328 width=52)
```

```
                Hash Cond: (page.host_id = host.id)
             ->  Hash Join  (cost=111.79..181.99 rows=2328 width=41)
                   Hash Cond: (most_referred_pages.page_id = page.id)
                ->  Seq Scan on most_referred_pages  (cost=0.00..35.28 rows=2328width=12)
                ->  Hash  (cost=84.13..84.13 rows=2213 width=37)
                   ->  Seq Scan on page  (cost=0.00..84.13 rows=2213width=37)
             ->  Hash  (cost=13.72..13.72 rows=472 width=19)
```

Žádná sláva. A to je v tabulkách relativně málo záznamů. Je to samý sekvenční čtení. Asi by to chtělo nějaký index. To pro materializovaný pohled není žádný problém :-).

```
rimmer1=> CREATE INDEX referred_ix ON most_referred_pages(referred);
DEBUG:  building index "referred_ix" on table "most_referred_pages"
CREATE INDEX
rimmer1=> explain SELECT referred, page.schema, host.host, page.page FROM most_referred_pages JOIN page ON page.id
= page_id JOIN host on host.id = host_id ORDER BY referred DESC limit 10;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Limit  (cost=0.83..9.00 rows=10 width=52)
   ->  Nested Loop  (cost=0.83..1808.94 rows=2213 width=52)
         ->  Nested Loop  (cost=0.56..1087.51 rows=2213 width=41)
               ->  Index Scan Backward using referred_ix on most_referred_pages  (cost=0.28..117.47 rows=2213 width=12)
               ->  Index Scan using page_pkey on page  (cost=0.28..0.43 rows=1width=37)
                     Index Cond: (id = most_referred_pages.page_id)
         ->  Index Scan using host_pkey on host  (cost=0.27..0.32 rows=1 width=19)
               Index Cond: (id = page.host_id)
(8 řádek)
```

No vida, výsledná odhadovaná doba klesla z 283.95 jednotek na 9.00.

Materializované pohledy můžete používat stejně jako běžné tabulky. Jen je většinou nemůžete přímo updatovat (můžete si napsat INSTEAD OF trigger).

Materializované pohledy jsou novinkou v Postgresu 9.3. Autoři do budoucna slibují další vylepšení. Pokud chcete vědět, co všechno by to mohlo být, čtěte dále v části o Oracle, co on všechno dovede.

MySQL

MySQL materializované pohledy neumí. Tím bych mohl výklad ukončit, ale protože mě materializované pohledy baví, ukáži vám, jak se dají v MySQL simulovat. Stačí k tomu jednoduchý CREATE TABLE ... AS SELECT ... statement.

```
mysql> CREATE TABLE most_referred_pages AS
SELECT referrer.page_id, count(*) AS referred
FROM referrer GROUP BY referrer.page_id
LIMIT 0;
```

LIMIT 0 tu funguje jako WITH NO DATA.

Místo příkazu REFRESH MATERIALIZED VIEW můžete použít tyto příkazy:

```
LOCK TABLE most_referred_pages WRITE,referrer READ;
DELETE FROM most_referred_pages;
INSERT INTO most_referred_pages
SELECT referrer.page_id, count(*) AS referred
FROM referrer GROUP BY referrer.page_id;
UNLOCK TABLES;
```

Ještě vytvořím tabulce indexy, které si zaslouží:

```
CREATE INDEX referred_ix ON most_referred_pages(referred);
ALTER TABLE most_referred_pages ADD FOREIGN KEY (page_id) REFERENCES page(id);
```

A tímto bych mohl výklad ukončit. Ale proč nevyužít příležitosti a nezopakovat si trochu triggery …

Automatický update

Pomocí triggerů můžete aktualizovat tabulku, která slouží jako materializovaný pohled, okamžitě při změně tabulek, na kterých je materializovaný pohled závislý.

Následuje ukázka triggerů, které aktualizují tabulku most_referred_pages při změne tabulky referrer(INSERTem nebo DELETEm).

```
DROP TRIGGER IF EXISTS refresh_mrp_on_insert;
DROP TRIGGER IF EXISTS refresh_mrp_on_delete;
DELIMITER //

CREATE TRIGGER refresh_mrp_on_insert AFTER INSERT ON referrer
FOR EACH ROW
BEGIN
DECLARE was_updated INTEGER;
UPDATE most_referred_pages SET referred = referred + 1 WHERE page_id = NEW.page_id;
SELECT ROW_COUNT() INTO was_updated;
IF (was_updated = 0)
THEN
INSERT INTO most_referred_pages VALUES(NEW.page_id, 1);
END IF;
END;

CREATE TRIGGER refresh_mrp_on_delete AFTER DELETE ON referrer
FOR EACH ROW
BEGIN
UPDATE most_referred_pages SET referred = referred - 1 WHERE page_id = OLD.page_id;
END;
```

```
//
DELIMITER ;
```

Trigger pro UPDATE jsem vynechal, protože tabulku referrer nemá smysl updatovat, ale v reálné aplikaci byste měli alespoň updaty zakázat (viz přístupová práva), abyste měli zajištěnou konzistenci databáze. Trigger pro DELETE taky neřeší případ, že referrer klesne na 0. Mít záznam s 0 a nemít záznam není totéž. Při refreshi celé tabulky by záznam v tabulce nebyl. Jestli to vadí nebo ne, to už je na vás :-). Problém můžete vyřešit jednoduchým příkazem navíc: DELETE ... WHERE page_id = OLD.page_id AND referred = 0;.

Samozřejmě ne vždy je takovýto UPDATE „materializovaného pohledu" možný. Stačilo by, abych vynechal při vytváření tabulky most_referred_pages sloupec page_id a už se nechytáte. A protože se materializované pohledy vytvářejí především pro složitější SQL dotazy, plné JOINů a analytických funkcí, bude to procento tabulek, které nepůjde takto updatovat, poměrně velké.

Musíte taky vzít v úvahu snížení rychlosti updatů do tabulky referrer. Každý INSERT znamená vyvolání triggeru a insert/update do další tabulky most_referred_pages.

Když jsem psal svůj program na stahování internetu, čekal jsem, že internetové připojení bude to, co mě bude brzdit. Ve skutečnosti jsou to právě inserty do databáze, které mě zpomalují nejvíce.

A tímto bych povídání o materializovaných pohledech v MySQL ukončil :) Jen dodám co je vám asi stejně jasné, totiž že stejným způsobem se dá řešit problém automatického updatu i v Postgresu (bez využití skutečných materializovaných pohledů, které, na rozdíl od tabulky, updatovat nejdou).

SQLite

SQLite materializovné pohledy nepodporuje, takže se u něj dá napsat to samé, co u MySQL. Jen je potřeba dát si pozor na ty drobné rozdíly mezi databázemi.

SQLite neumí přidat FOREIGN KEY k již existující tabulce, takže si nejdříve musím vytvořit tabulku a pak jí teprve mohu plnit.

```
DROP TABLE IF EXISTS most_referred_pages;
CREATE TABLE most_referred_pages(page_id INT REFERENCES page(id) UNIQUE, referred INT);
CREATE INDEX referred_ix ON most_referred_pages(referred);
```

Co se plnění týče, je to stejné jako u MySQL, jen SQLite nepodporuje LOCK TABLE:

```
DELETE FROM most_referred_pages;
INSERT INTO most_referred_pages
SELECT referrer.page_id, count(*) AS referred
FROM referrer GROUP BY referrer.page_id;
```

A v triggeru neumožňuje používat podmínky IF nebo WHEN, takže trigger AFTER INSERT vypadá trochu jinak. Využívá toho, že jsem udělal sloupec most_referred_pages.page_id unikátní (což jsem vlastně mohl udělat i v MySQL):

```
DROP TRIGGER IF EXISTS refresh_mrp_on_insert;
DROP TRIGGER IF EXISTS refresh_mrp_on_delete;

CREATE TRIGGER refresh_mrp_on_insert AFTER INSERT ON referrer
FOR EACH ROW
BEGIN
UPDATE most_referred_pages SET referred = referred + 1 WHERE page_id = NEW.page_id;
INSERT OR IGNORE INTO most_referred_pages (page_id, referred) VALUES(NEW.page_id, 1);
END;

CREATE TRIGGER refresh_mrp_on_delete AFTER DELETE ON referrer
FOR EACH ROW
BEGIN
UPDATE most_referred_pages SET referred = referred - 1 WHERE page_id = OLD.page_id;
END;
```

INSERT OR IGNORE je rozšíření SQL jazyka v SQLite. Příkaz vloží hodnotu, pokud tomu nezabrání nějaké integritní omezení (např. unikátní hodnota na sloupci page_id). Pokud INSERT selže, chyba se ignoruje (takže trigger, potažmo INSERT, který trigger vyvolal, nebude odvolán).

Oracle

Vítejte v zemi materializovaným pohledům zaslíbené! Oracle už má podporu pro materializované pohledy dlouho a má jí velkou. Podívejte se na syntax pro vytvoření m. pohledu. Syntaxe je rozdělená do mnoha obrázků, aby se na stránku vůbec vešla! Já tu z toho nebudu probírat všechno, ale jen malou část:

```
CREATE MATERIALIZED VIEW view_name
[ BUILD {IMMEDIATE | DEFERRED} ]
[ REFRESH  [{ FAST | COMPLETE | FORCE }] [ON { COMMIT | DEMAND }]
[ START WITH date ] [NEXT interval] ]
AS
SELECT ...;
```

Pro začátek začnu příkladem, kterým jsem začal v úvodní části o Postgresu. Vytvořím materializovaný pohled, který se ihned nezmaterializuje.

```
oracle> CREATE MATERIALIZED VIEW most_referred_pages
BUILD DEFERRED
AS
SELECT referrer.page_id, count(*) AS referred
FROM referrer
GROUP BY referrer.page_id;

oracle> SELECT COUNT(*) FROM most_referred_pages;
COUNT(*)
--------
0
```

BUILD DEFERRED je to, co způsobí, že se materializovaný pohled nematerializuje. Pokud tento výraz vynecháte (nebo nahradíte defaultním BUILD IMMEDIATE), SQL dotaz definující pohled se ihned provede a výsledek se uloží.

Oracle nemá REFRESH příkaz. Místo toho můžete použít balíček DBMS_MVIEW, konkrétně jeho proceduru REFRESH. Této procedůře můžete předat spousty parametrů, důležité jsou tyto dva:

Parametr	Význam
list	Řetězec, čárkou oddělené jména m. pohledů, které chcete aktualizovat.
method	Způsob aktualizace. f = FAST REFRESH, c = COMPLETE REFRESH, ? = FORCE REFERSH. Význam refresh metody popíši dále.

Procedůru můžete spustit dvěma způsoby, oba udělají totéž:

```
BEGIN
  DBMS_MVIEW.REFRESH (
    list   => 'most_referred_pages',
    method => 'c'
  );
END;
/
-- jinak zapsané volání téhož:
EXECUTE DBMS_MVIEW.REFRESH ( list => 'most_referred_pages', method => 'c');
```

Ověření, že to zabralo:

```
oracle> SELECT count(*) FROM most_referred_pages;
  COUNT(*)
----------
      2213
```

Balíček DBMS_MVIEW má i další užitečné procedůry, jako například REFRESH_ALL_MVIVEWS, která aktualizuje všechny materializované pohledy.

Fast, forced a complete refresh

Při vytváření pohledu můžete určit jeden ze tří způsobů refreshe: FAST, COMPLETE nebo FORCE.

REFRESH FAST

Pohled se bude aktualizovat jen podle změn od posledního refreshe v tabulkách, na kterých je pohled závislý. To je samozřejmě rychlejší, než když celý materializovaný pohled smažete a vytvoříte znova. Ale musíte mít někde zaznamenáno, co se změnilo. Takže tabulky, na kterých je pohled závislý (v oracle jim říkají **master tables**), musí své změny logovat.

REFRESH COMPLETE

Materializovaný pohled (resp. jeho data) je celý smazán a vytvořen znova. (Takhle se refreshují m. pohledy v Postgresu.)

REFRESH FORCE

Pokusí se provést FAST REFRESH. Pokud to nejde (třeba proto, že pro master tabulky neexistují logy), provede se REFRESH COMPLETE. Toto je defaultní volba.

Jako příklad zkusím vytvořit m. pohled pages_with_most_referrences s REFRESH FAST.

```
oracle> CREATE MATERIALIZED VIEW pages_with_most_referrences
  REFRESH FAST
  AS
  SELECT referrer_id, count(page_id) AS pages
  FROM referrer
  GROUP BY referrer_id;
ERROR at line 5:
ORA-23413: table "RIMMER"."REFERRER" does not have a materialized view log
```

Já to říkal. Pro REFRESH FAST musí mít master tabulky logování. Tak ho k tabulce referrer přidám. A to pomocí CREATE MATERIALIZED VIEW LOG. Timto příkazem můžete určit co všechno se bude logovat. Pro materializovaný pohled s agregacemi (jako třeba COUNT) musíte zahrnout do logování nové hodnoty (INCLUDING NEW VALUES).

```
oracle>  CREATE MATERIALIZED VIEW LOG ON referrer INCLUDING NEW VALUES;
ERROR at line 1:
ORA-00439: feature not enabled: Advanced replication
```

Tak bohužel, tahle funčknost není ve verzi XE dostupná, takže si to nevyzkoušíme. Ale prostě to někdy někde (v placené verzi) nějak funguje, takže 3x hurá Oracle :-).

Na REFRESH FAST pohledy jsou kladena určitá omezení pro to, z jakých SQL dotazů může být Fast Refresh View vytvořen.

Jestli máte náhodou placenou verzi, tak si pages_with_most_referrences zase smažte a jedeme dál…

```
DROP MATERIALIZED VIEW pages_with_most_referrences;
```

Automatický update

V MySQL jste viděli, jak se může vytvořit automaticky updatovaný pohled pomocí triggerů. V oracle můžete vytvořit materializovaný pohled, který tohle umí sám. Stačí uvést REFRESH ON COMMIT. Pak se pohled automaticky refreshne po commitnutých změnách na jeho master tabulkách.

```
CREATE MATERIALIZED VIEW pages_with_most_referrences
  REFRESH ON COMMIT
  AS
  SELECT referrer_id, count(page_id) AS pages
  FROM referrer
  GROUP BY referrer_id;
```

Pohled se vytvořil a naplnil daty. V databázi ovšem není ještě žádný záznam v tabulce referrer s referrer_id = 11976. Přesvědčíme se o tom na pohledu a pak tento záznam vložíme. Pak by se měl do pohledu automaticky promítnout.

```
oracle> SELECT * fROM pages_with_most_referrences  WHERE referrer_id IN (11976);

no rows selected
oracle> INSERT INTO referrer (id, referrer_id,page_id) VALUES (referrer_id_seq.NEXTVAL, 11976,11975);

oracle> SELECT * FROM pages_with_most_referrences WHERE referrer_id IN (11976);
```

```
no rows selected
```
Tedy až po commitu:
```
oracle> commit;

Commit complete

oracle> SELECT * FROM pages_with_most_referrences WHERE referrer_id IN (11976);

REFERRER_ID      PAGES
----------- ---------
      11976         1
```
Jupíí, funguje! Má to ale své mouchy. Pokud máte m. pohled s REFRESH ON COMMIT ale bez REFRESH FAST (jako v příkladu výše), tak se m. pohled při commitu změn v master tabulkách celý generuje znova. Což obvykle bývá velmi zdlouhavé (proto se m. pohledy vytvářejí v první řadě. (Na těchto testovacích datech to asi nepoznáte, protože tabulka referrer má jen 2500 záznamů a tak se m. pohled pages_with_most_referrences vygeneruje během zlomku sekundy).

Periodický update

Pomocí START WITH date a NEXT interval můžete zařídit, aby se m. pohled aktualizoval automaticky sám po určité době. START WITH date určuje okamžik prvního updatu. Pokud jej neuvedete, ale uvedete NEXT interval, provede se první update okamžitě (stejně, jako když uvedete START WITH SYSDATE). Datum date musí být v budoucnosti.
NEXT interval určuje časový interval, po kterém se bude m. pohled aktualizovat. Pokud zadáte jako interval číslo, bude se interpretovat jako počet dní. 1 je jeden den, 2 jsou dva dny, 1/2 je 12 hodin atd.
Následující příklad vytvoří m. pohled most_referred_pages2 na základě m. pohledu most_referred_pages, ke kterému připojí tabulky host a page. Nastavím u něj, aby se aktualizoval každých 5 minut. A hned vyzkouším, jak to funguje.
```
CREATE MATERIALIZED VIEW most_referred_pages2 REFRESH START WITH SYSDATE NEXT
SYSDATE +1/(24*12)
AS
SELECT page.id AS page_id, page.schema, host.host, page.port, page.page, referred
FROM host
JOIN page ON host.id = host_id
JOIN most_referred_pages ON page.id = page_id
ORDER BY referred DESC;
```
Pohled je vytvořený. Podívejme se nejdřív, kolikrát je podle tohoto pohledu odkazovaná stránka s id 9764.
```
oracle> SELECT * FROM most_referred_pages2 WHERE page_id = 9764;
PAGE_ID SCHEMA       HOST PORT PAGE REFERRED
------- ------ ---------- ---- ---- --------
   9764 http   sallyx.org   80 /       2
```
2x. Tak vložím další odkaz. Pohled most_referred_pages2 je závislý na pohledu most_referred_pages, ten se ale periodicky neaktulizuje (ani automaticky ON COMMIT). Takže, aby se něco změnilo v most_referred_pages2, musím ho nejdřív aktualizovat ručně (ON DEMAND, řekl by oracle databázista).
```
oracle> INSERT INTO referrer (id, referrer_id, page_id) VALUES (referrer_id_seq.NEXTVAL, 9770, 9764);

1 row created.

oracle> COMMIT;

oracle> SELECT * FROM most_referred_pages WHERE page_id = 9764;

PAGE_ID   REFERRED
---------- ----------
      9764          2

oracle> EXECUTE DBMS_MVIEW.REFRESH ( list =>  'most_referred_pages', method =>  'c');

PL/SQL procedure successfully completed.


oracle> SELECT * FROM most_referred_pages WHERE page_id = 9764;

PAGE_ID   REFERRED
---------- ----------
      9764          3

oracle> SELECT * FROM most_referred_pages2 WHERE page_id = 9764;

PAGE_ID    SCHEMA       HOST PORT PAGE REFERRED
---------- --------  ---------- ---- ---- --------
      9764      http   sallyx.org   80 /       2
```
Pohled most_referred_pages jsem aktualizoval, ale most_referred_pages2 je pořád beze změny. Stačí být ale trpělivý a chvilku si počkat (max. 5 minut).
```
-- po 5 minutách
oracle> SELECT * FROM most_referred_pages2 WHERE page_id = 9764;
PAGE_ID    SCHEMA       HOST PORT PAGE REFERRED
---------- --------  ---------- ---- ---- --------
      9764      http   sallyx.org   80 /       3
```
A zase to vyšlo :-)

Závěr

Materializované pohledy jsou šikovná věc. Hodí se především tam, kde potřebujete mít rychle výsledek složitého dotazu, ale nepotřebujete mít aktuální data. Viděli jste, že můžete mít i věčně aktuální materialiozovaný pohled (ideálně REFRESH FAST ON COMMIT v Oracle, nebo tabulka + triggery). Pohledy se taky snadno používají pro jednoduchou replikaci dat, kdy si díky m. pohledu můžete z jedné databáze do druhé (třeba na svém notebooku) přenést jen určitý pohled na data (statistiky za poslední týden atp.).

V Postgresu jsou materializované pohledy novinkou, takže toho ještě moc neumí. (Krom toho co jsem zmínil v tomto tutoriálu snad stojí za zmínku už jen příkaz ALTER MATERIALIZED VIEW.

MySQL a SQLite m. pohledy nepodporují, ale dají se snadno nasimulovat.

Bezkonkurenčně nejlépe je na tom v současné době Oracle. Popsal jsem vám ty nejdůležitější atributy m. pohledů, které můžete nastavit. Kromě toho mají m. pohledy v oracle spoustu dalších vlastností (které můžete nastavovat např. pomocí ALTER MATERIALIZED VIEW. Představil jsem vám balíček DBMS_MVIEW. Kromě toho ale existuje i balíček BMS_REFRESH, který obsahuje další procedury pro práci s m. pohledy. Například vám umožní vytvořit skupiny materializovaných pohledů, které pak můžete refreshnout naráz (v jedné transakci).

V Oracle mohou být jednoduché m. pohledy updatovatelné (DML dotazy nad m. pohledem změní jeho master tabulku). On ten Oracle nebude zase až tak špatná databáze :-). A touto bombou se s vámi zde loučím.

PS: Pro hloubavé čtenáře tu ještě ukáži pár příkazů, které můžete použít pro replikaci dat z jedné databáze (rimmer) do druhé (petr).

```
oracle> connect system/system
oracle> GRANT CREATE DATABASE LINK to petr;
oracle> connect petr/petr
oracle> CREATE DATABASE LINK rimmer_link CONNECT TO rimmer IDENTIFIED BY rimmer USING'xe';
oracle> SELECT count(*) FROM referrer@rimmer_link;
  COUNT(*)
----------
      2502
oracle> CREATE MATERIALIZED VIEW pages_with_most_referrences
AS
SELECT referrer_id, count(page_id) AS pages
FROM referrer@rimmer_link
GROUP BY referrer_id;

Materialized view created.
```

M. pohled vytvářený z tabulky z jiné databáze nemůže mít REFRESH ON COMMIT.

Konfigurace

Když budete mít štěstí, vystačíte si s defaultním nastavením DBMS. Pro případ, že byste štěstí neměli, zmíním tu alespoň pár věcí, se kterými můžete začít při řešení svých problémů. Kompletní konfigurační příručku v jedné kapitole ale nečekejte :-). Tu najdete v online tutoriálu v kapitole III. Server Administration.

- Postgres
  - Log level
  - Konfigurace psql
- MySQL
  - Soubor ~/.my.cnf
- SQLite
- Oracle

PostgreSQL

V kaptiole Instalace PostgreSQL jsem probral jak spustit či vypnout server, případně zjistit jeho status. Dále jak vytvořit uživatele a databázi.

V kapitole Začínáme s PostgreSQL jsem se zmínil o programu pg_dump, který se používá k zálohování a obnově zálohy.

V kapitole o přistupových právech je část věnovaná konfiguračnímu souboru pg_hba.conf, který řídí možnosti připojení k DBMS.

V kapitole Datum a čas jsem probral konfiguraci časových zón a nastavení formátu času.

V kapitole o Vytváření uživatelských funkcí jsem popsal, jak přidat do Postgresu jazyk PL/Python.

Příkaz pro zjištění umístění konfiguračního souboru je SHOW config_file. Musíte mít ale příslušná práva, aby se vám jméno souboru ukázalo:

```
rimmer1=# SHOW config_file;
               config_file
-----------------------------------------
/var/lib/pgsql/data/postgresql.conf
(1 řádka)
```

Všechno co v konfiguračním souboru začíná znakem # je komentář. V souboru je spoustu příkladů toho, co můžete nastavit. Naprostou většinu věcí nebudete muset měnit :-). Co která volba znamená se pak dočtete v dokumentaci.

Pokud byste chtěli vědět, kde se data fyzicky ukládají, tak to vám řekne příkaz SHOW data_directory;.

```
SHOW data_directory;
    data_directory
----------------------
/var/lib/pgsql/data
(1 řádka)
```

Log level

Log level, přesněji proměnná client_min_messages určuje, jaké všechny zprávy se vám při spouštění SQL příkazů budou zobrazovat. Například od verze Postgresu 9.3 se nezobrazují NOTICE o vytvoření implicitních indexech a sequencích (např. při použití typu serial), pokud nenastavíte hodnotu client_min_messagesalespoň na DEBUG1.

```
rimmer1=> SHOW client_min_messages;
 client_min_messages
---------------------
```

notice
(1 řádka)
Možnosti nastavení jsou: DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, INFO, NOTICE, WARNING, ERROR, LOG, FATAL, a PANIC.
rimmer1=> **SET** client_min_messages **TO** DEBUG1;
SET
Příkaz SET client_min_messages TO DEBUG1; můžete zapsat do souboru ~/.psqlrc, abyste jej nemuseli zapisovat při každém spuštění klienta psql.

Konfigurace psql

Klient psql se obvykle snaží při startu spustit soubory /etc/psqlrc a ~/.psqlrc. Uživatelé Linuxu určitě vědí, kde tyto soubory najít, uživatelům Windows úplně neporadím, ale soubor psqlrc hledá klient psql v %APPDATA%\postgresql\psqlrc.conf.

Pokud vás, stejně jako mě, obtěžuje v psql stránkování při výpisu, vložte si do ~/.psqlrc tuto řádku:
\pset pager off
V Linuxu se označuje vlnovkou ~ domovský adresář. Soubor ~/.psqlrc má pro uživatele s domovským adresářem reálnou cestu /home/petr/.psqlrc.
V Linuxu také platí, že všechny soubory které začínají tečkou jsou skryté (defaultně je nevypíše příkaz ls). Proto začínají konfigurační soubory v Linux obvykle tečkou. We Windows jsou pak bez tečky.
Historie SQL příkazů se ukládá do souboru ~/.psql_history (ve Windows %APPDATA%\postgresql\psql_history).

MySQL

Spuštění / vypnutí / restartování serveru MySQL se dělá stejně jako Postgresu. Jen místo Postgres napíšete mysql :).
V kapitole o přístupových právech jsem se zmínil o tom, kde najít informace o zjišťování nastavených přístupových právech a o volbě bind-address.
V kapitole o datu a času se dočtete, jak v MySQL nainstalovat časové zóny.
Nevím jak v MySQL zjistit, z jakého konfiguračního souboru načetl sever své volby, protože konfigurační soubor můžete serveru nastavit při spuštění na příkazové řádce. Můžete ale zjistit defaultní umístění souborů, ze kterých mysqld čte konfiguraci:

```
bash$ /usr/sbin/mysqld --help --verbose | grep "Default options" -A 1
Default options are read from the following files in the given order:
/etc/my.cnf /etc/mysql/my.cnf /usr/etc/my.cnf ~/.my.cnf
```

Další informace můžete zjistit pomocí skriptu my_print_defaults.

```
bash$ my_print_defaults mysql
--default-character-set=utf8
--user=petr
bash$ sudo my_print_defaults mysqld
root's password:
--port=3306
--socket=/var/run/mysql/mysql.sock
--datadir=/var/lib/mysql
--skip-external-locking
--key_buffer_size=32M
--max_allowed_packet=16M
--table_open_cache=128
--sort_buffer_size=512K
--net_buffer_length=16K
--read_buffer_size=256K
--read_rnd_buffer_size=512K
--myisam_sort_buffer_size=8M
--log-bin=mysql-bin
--binlog_format=mixed
--server-id=1
```

Všimněte si, že defaultní hodnoty pro server (mysqld) zjišťuji jako root.
Kde jsou data fyzicky uložená zjistíte následujícím příkazem:

```
mysql> SHOW VARIABLES LIKE 'datadir';
+---------------+-----------------+
| Variable_name | Value           |
+---------------+-----------------+
| datadir       | /var/lib/mysql/ |
+---------------+-----------------+
1 row in set (0.00 sec)
```

Soubor ~/.my.cnf

Soubor ~/.my.cnf se používá pro nastavení konfigurace hned několika programů. Můžete tam nastavit konfiguraci pro server (mysqld). Server čte soubor ~/.my.cnf toho uživatele, který serve spouští (přesněji řečeno, pod kterým uživatelem je server spouštěn).
V tomto souboru si spíš budete nastavovat konfiguraci pro klienta mysql a pro program pro dumpování databáze myqsldump.
Každý program má v souboru svou sekci, která začíná názvem programu uzavřeným v hranatých závorkách:

```
[mysql]
default-character-set=utf8
user=petr
i-am-a-dummy

[mysqldump]
user=petr
```

V příkladu se nastavuje pro klienta defaultní kódování na UTF-8, uživatel na petr a ještě volba "i-am-a-dummy", která vám zabrání například ve spuštění příkazu DELETE bez podmínky WHERE (a v dalších „nebezpečných" příkazech).
Pro program mysqldump je nastaveno jen defaultní uživatelské jméno na petr. (Vzhledem k tomu, že defaultní uživatelské jméno je jméno uživatele operačního systému, asi to nastavovat nebudete muset).

Do tohoto souboru můžete uložit i heslo, ale pak si ho bude moci přečíst každý, kdo získá přístup k vašemu počítači, takže se to nedoproučuje.

O programu mysqldump jsem psal už v kaptiole Začínáme s PostgreSQL.

Historie příkazů se ukládá do ~/.mysql_history.

SQLite

Program SQLite čte konfiguraci ze souboru ~/.sqliterc . Můžete do něj zapsat všechny metapříkazy, které v sqlite3 zobrazíte metapříkazem .help.

```
.headers ON
.mode column
PRAGMA foreign_keys = ON;
```

Historie SQL příkazů se ukládá do souboru ~/.sqlite_history.

Zálohu (dumpování) databáze můžete provést metapříkazem .dump.

Oracle

V kaptiole Začínáme s PostgreSQL se dočtete o dumpování a nahrávání dat.

V kaptiole Datum a čas se dozvíte něco o tom, jak nastavit defaultní formát času.

Pokud potřebujete rekonfigurovat Oracle XE, můžete to zkusit takto:

1. Smažte soubor /etc/sysconfig/oracle-xe nebo v /etc/default/oracle-xe změňteCONFIGURE_RUN=true na CONFIGURE_RUN=false
2. Spusťte (jako administrátor): /etc/init.d/oracle-xe configure

Možná se tím připravíte o všechny data v databázi, takže si je raději nejdříve zálohujte.

Každopádně vám přeji hodně štěstí :-).