1 Úvod do práce s databází

Výstupy

- Zná základní typy databázových systémů
- Orientuje se v dostupných databázových produktech
- Umí instalovat databázový server (Postgresql)
- Umí se prostřednictvím databázového klienta připojit k databázi
- Rozumí základní terminologii relačních databází a uložení dat v databázových tabulkách
- Zná základní datové typy atributů, které jsou používány v relačních databázích
- Umí číst schematické znázornění datového modelu, rozumí problematice datového modelování

Pod pojmem databáze rozumíme softwarovou aplikaci, jejímž primárním úkolem je zajistit jednak ukládání strukturovaných dat, jednak zajistit co možná nejrychlejší a nejefektivnější přístup k těmto datům. Od moderní databáze se taktéž očekává, že zajistí bezpečnost dat, jak z pohledu omezení přístupu k datům pouze pro oprávněné osoby, tak z pohledu zachování integrity dat při víceuživatelském využívání vložených dat.

1.1 Rozdělení databází podle typu

Podle způsobu ukládání dat rozlišujeme několik typů databází:

- *Hierarchická databáze* – tato databáze je založená na hierarchickém modelu. Logické uspořádání dat má stromovou strukturu.
- *Síťová databáze* – tato databáze je založená na síťovém modelu, ve kterém jsou data logicky i fyzicky uspořádána jako uzly rovinného grafu. Každý záznam může být spojený s libovolným počtem dalších záznamů.
- *Objektová databáze* – tato databáze je založena na objektech, jejich zapouzdření a dědičnosti. Místo tabulek jsou zde uloženy přímo objekty, včetně svých vlastností, a místo řádků se ukládají samotné instance objektů.
- *Relační databáze* – tato databáze založená na relačním modelu, v němž jsou data logicky uspořádána do relací, tj. výsledků kartézského součinu nad doménami neboli množinami údajů. Relační databáze je založena na tabulkách, které obvykle chápeme tak, že uchovávají informace o relacích mezi jednotlivými záznamy.

Hierarchické a síťové databáze se poprvé objevují v 60. letech minulého století na sálových počítačích. Jedním z prvních průkopníků databází byl Charles Bachman. Jedním z prvních databázových systémů byl **IMS**, který byl vyvinut firmou IBM pro program letu na Měsíc Program Apollo.

V roce 1970 začínají zveřejněním článku E. F. Codda první **relační databáze**, které pohlížejí na data jako na tabulky. Kolem roku 1974 se vyvíjí první verze dotazovacího jazyka SQL. Vývoj této technologie po 10 letech přinesl výkonově použitelné systémy, srovnatelné se síťovými a hierarchickými databázemi.

V 90. letech 20. století se začínaly objevovat první **objektově** orientované databáze, jejichž filozofie byla přebírána z objektově orientovaných jazyků. Tyto databáze měly podle

předpokladů vytlačit relační systémy. Původní předpoklady se však nenaplnily a vznikla kompromisní **objektově-relační** technologie.

1.2 Databázové produkty

V dnešní době se můžeme setkat s celou řadou databázových systémů od mnoha výrobců. Škála je velmi široká od open source produktů až po velmi drahé komerční produkty. Pokud stojíme pře problémem, který produkt zvolit, měli bychom kromě dostupných finančních prostředků definovat naše požadavky:

- Operační systém, na kterém budeme databázový systém provozovat
- Počet uživatelů, kteří budou k datům přistupovat
- Objem zpracovávaných dat
- Požadavky na výkon
- Požadavky na dostupné funkce
- Požadavky na dostupnost a bezpečnost dat
- Požadavky na technickou podporu dodavatele

Nemusí vždy platit, že komerční produkty jsou kvalitnější než produkty zdarma, pro menší projekty jsou open-source produkty zcela dostačující, komerční produkty najdeme v oblasti kriticky důležitých systémů, kde jsou požadavky na vysoký výkon a vysokou dostupnost a spolehlivost.

Následující abecedně setříděný přehled produktů není zdaleka kompletní, nicméně obsahuje nejrozšířenější databázové produkty.

Cache - Nejrozšířenější objektová databáze, která umožňuje k datům přistupovat také pomocí SQL dotazů. Nabízí transakční zpracování, masivní škálovatelnost, real-time analýzy a webový přístup k databázi. Připravená pro Javu a .NET a certifikovaná na Red Hat a SUSE Enterprise Linux.

DB2 - komerční relační databáze firmy IBM pro aplikace s vysokými nároky na dostupnost a zabezpečení dat.

Firebird - původně známá jako databáze Interbase, která byla vyvíjená firmou Borland, následně uvolněn její zdrojový kód dále vyvíjena open source komunitou jako relační databáze Firebird. Volně dostupná kompaktní databáze instalovatelná na širokém spektru operačních systémů, s podporou transakčního zpracování, uložených procedur i triggerů. Ve srovnání s POSTGRESQL menší nabídka funkcí, výhodou je snadné zálohování a přenos databázového souboru (veškerá data jsou ukládána v jednom souboru).

ObjectDB - Volně dostupná (pro osobní a nekomerční použití) objektová databáze pro Javu. Může pracovat jak v klient-server, tak v embedded režimu.

Objectivity/DB - Jedná se o distribuovanou (data mohou být transparentně replikována na různých serverech), objektovou databázi s nejširším výběrem API – pro C++, Javu, Python, Smalltalk a obecný ODBC. K dipozici je i 64bitová verze. Stáhnout lze 60denní trial, nebo plnou placenou.

ObjectStore - Objektová databáze, která může být použita v C++, nebo Javě. Nabízí robustní systém cachování, transakční zpracování, online zálohy a replikace, škálování nebo podporu clusterů. K dispozici je testovací embedded verze omezená na jeden proces, případně placená plná verze.

MS ACCESS - relační databázová aplikace firmy Microsoft určená pro jednouživatelské aplikace pro prostředí operačního systému Widows, součást kancelářského balíku MS Office.

MS SQL - komerční relační databáze firmy Microsoft určená pro platformu MS Windows a související Microsft technologie (.NET).

MySQL - open source relační databáze, kterou nedávno převzala firma ORACLE. Je velmi rozšířená jako databáze pro webové aplikace. Dokáže zpracovávat velké objemy dat, méně vhodná je pro aplikace vyžadující efektivní transakční zpracování.

ORACLE - komerční relační databáze s podporou objektově relační technologie. Patří ke špičce v oblasti databázových systémů se širou škálou specificky zaměřených modulů (analýza textu, datové sklady, geografická data a další). Databáze je certifikovaná pro systémy MS Windows i vybrané distribuce UNIX a LINUX. Zdarma je dostupná okleštěná a limitovaná verze ORACLE Express.

POSTGRESQL - open source alternativa k databázi ORACLE, volně dostupná pro širokou šálu operačních systémů. Nabízí řadu SQL funkcí, vlastní procedurální jazyk PgSQL pro vytváření uživatelských procedur a triggerů.

1.3 Přístup k databázi

Databázové systémy lze rozdělit na jednouživatelské a víceuživatelské. Ty první jsou obvykle používány pro lokální zpracování dat nebo jako úložný systém pro lokální aplikace. Nepočítá se s tím, že by data využívalo více uživatelů současně. Příkladem může být MS ACCESS. Tato aplikace v sobě obsahuje jak datové úložiště a SQL engine, tak doplňující nástroje pro tvorbu reportů či elektronických formulářů. Vše je dostupné v jedné aplikaci. Oproti tomu víceuživatelské databáze jsou navrženy tak, aby dokázaly zpracovat současnou manipulaci dat několika uživatelů. Jsou rozděleny na tzv. serverovou část a klienta. Server je aplikace, která běží nepřetržitě na vybraném počítači, zajišťuje ukládání dat a vyřizuje SQL příkazy klientů. Klient je samostatná aplikace, pomocí které jsme schopni připojit se k databázovému serveru a ovládat ho pomocí příkazů. Klient nám také zobrazuje z databáze získaná data. Klientských aplikací k jednomu databázovému produktu je často více. Liší se obvykle uživatelským komfortem. Řádkový klient má jen jednoduché uživatelské prostředí v podobě příkazového řádku, kde píšeme SQL dotazy, které klient odesílá serveru. Získaná data klient jednoduše vypíše na obrazovku nebo zapíše do souboru. Grafický klient pak umožňuje prohlížení databáze pomocí menu, SQL příkazy je možné skládat pomocí průvodců, data lze exportovat do různých formátů.

Z výše uvedeného vyplývá, že před tím než začneme s databází pracovat, musíme buď instalovat databázový server, nebo získat potřebné informace o tom, kde je server provozován. Druhým krokem je instalace databázového klienta na náš počítač. Pokud instalujeme databázový server, je obvykle součástí instalačního balíčku i instalace klienta. V případě POSTGRESQL stačí na operačním systému windows spustit instalační soubor pro vybranou verzi (např. postgresql-9.2.4-1-windows-x64.exe), který lze stáhnout na stránkách projektu http://www.postgresql.org. Kromě serveru se nainstaluje řádkový klient **psql** a grafický klient **pgAdmin**.

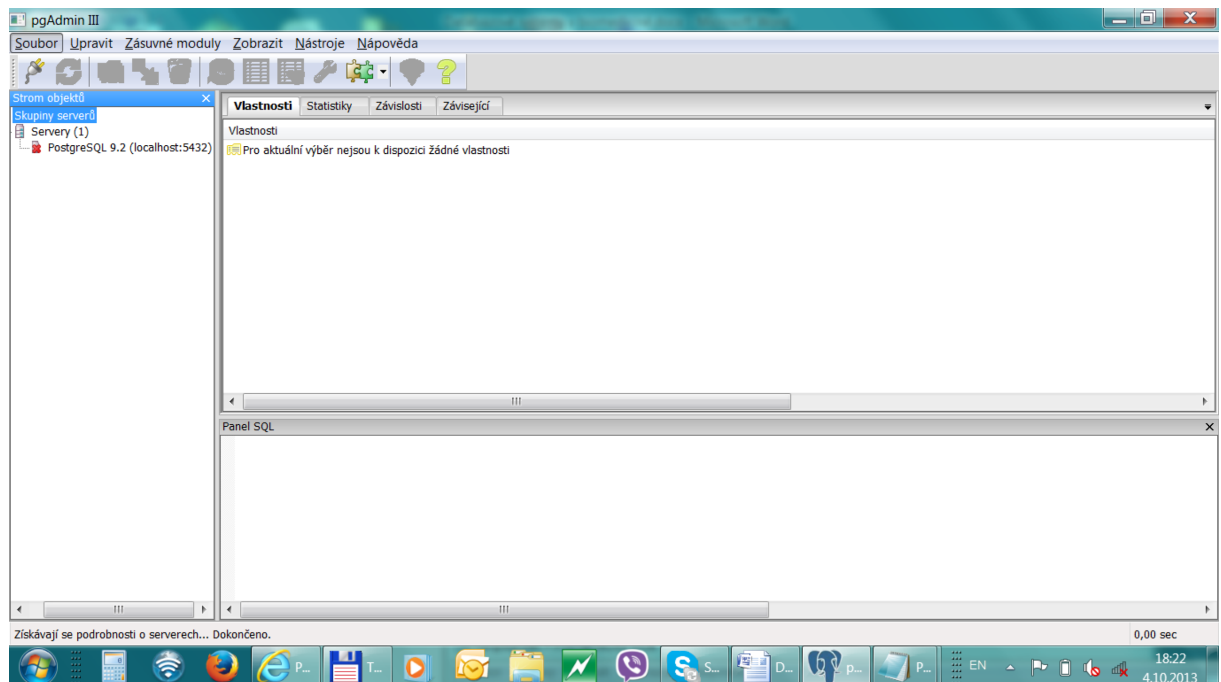
1.3.1 Připojení k POSTGRESQL

K databázi POSTGRESQL se v prostředí MS Windows připojíme spuštěním skriptu runpsql.bat, který součástí instalace klienta. Skript se nás postupně dotáže na:

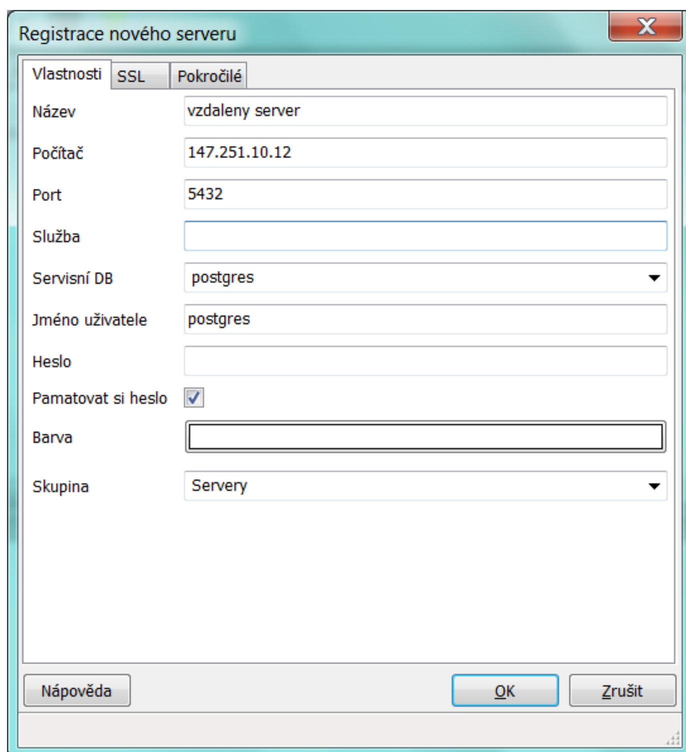
- Jméno nebo IP adresu serveru, kde běží databázový server. Výchozí hodnota je náš počítač (localhost).
- Jméno databáze na daném serveru. Na databázovém serveru můžeme mít uložena data více nezávislých projektů. Při připojování specifikujeme, se kterou datovou strukturou chceme pracovat. Výchozí hodnota je postgres, což je výchozí datová struktura (databáze) na databázovém serveru POSTGRESQL.
- Port, na kterém je databázová aplikace na daném serveru dostupná. Výchozí hodnota je 5432.
- Přihlašovací login. Výchozí účet je postgres, což je hlavní administrátorský účet databázového serveru.
- Heslo. Výchozí heslo k účtu postgres volíme během instalace serveru.

Pokud se připojujeme k jinému než lokálnímu serveru, musí nám administrátor databáze výše uvedené údaje poskytnout.

Pokud se k databázovému serveru připojujeme přes klienta pgAdmin, zadáváme stejné údaje, ale v grafickém režimu. V okně Strom objektů vybereme nejprve server, zadáme heslo a v následujícím kroku zvolíme databázi, se kterou chceme pracovat. Pokud se chceme připojit poprvé ke vzdálenému serveru, zvolíme v menu Soubor => Přidat server.



Obrázek 1 - Aplikace pgAdmin



Obrázek 2 - pgAdmin - registrace nového serveru

Vyplníme potřebné údaje, položku Služba necháváme obvykle prázdnou.

Jakmile jsme připojeni k serveru, zvolíme kliknutím databázi, se kterou chceme pracovat. Seznam databází obsahuje minimálně výchozí databázi postgres. Novou databázi vytvoříme kliknutím pravým tlačítkem na Databáze a výběrem položky Nová databáze. V dialogovém okně stačí vyplnit jméno nové databáze, ostatní položky je možné nechat ve výchozím stavu. Alternativní cestou je vytvoření databáze pomocí SQL příkazu. V řádkovém klientu psql můžeme příkazy SQL zadávat hned po připojení, v pgAdmin nejprve klikneme na vybranou databázi (postgres) a v hlavní liště klikneme na SQL ikonu. Příkaz pro vytvoření databáze v POSTGRESQL vypadá následovně:

```
CREATE DATABASE moje_db
  WITH ENCODING='UTF8'
  CONNECTION LIMIT=-1;
```

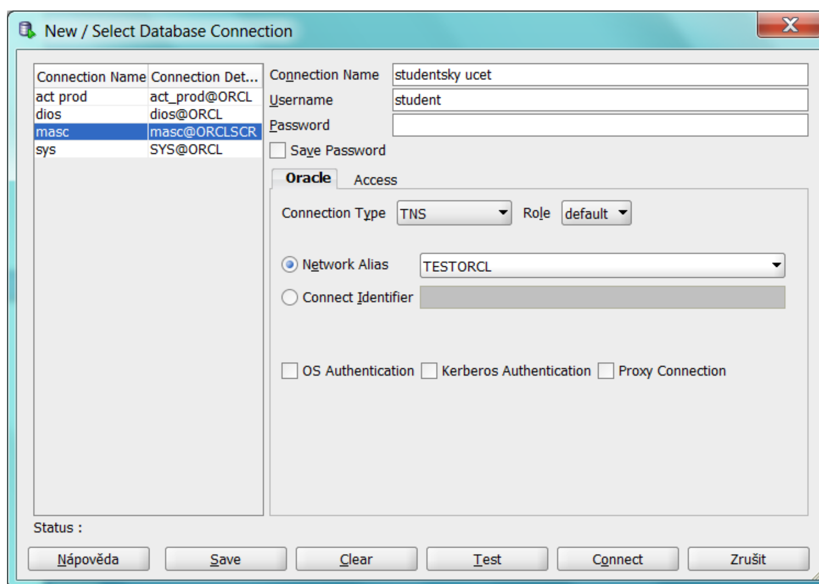
Název nové databáze je "moje_db", encoding označuje znakovou sadu databáze (UTF8 je univerzální sada podporující veškeré světové jazyky), connection limit uvádí maximální počet současně připojených uživatelů (-1 značí neomezený počet).

1.3.2 Připojení k ORACLE

Pro připojení k databázi ORACLE opět potřebujeme nejprve instalovat klienta databáze. Klienta je možné po registraci zdarma stáhnout na http://www.oracle.com. Instalační průvodce nabízí několik variant instalace. Pro připojení k databázi potřebujeme instalovat buď řádkový klient **sqlplus** nebo grafický klient **sqldeveloper.**

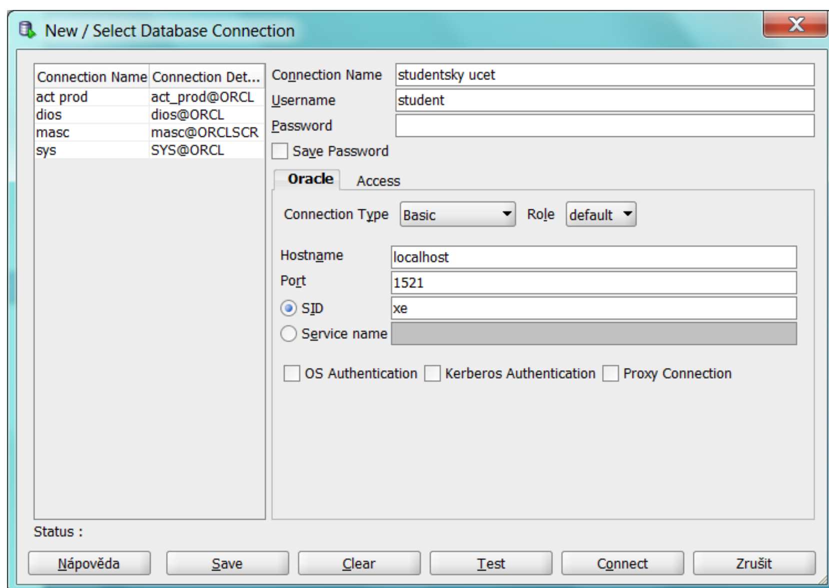
Klient sqlplus se hodí hlavně pro automatické spouštění hotových skriptů a sestav, pro dolování dat a testování SQL příkazů je mnohem vhodnější sqldeveloper, proto se dále zaměříme na něj. Způsobů připojení k ORACLE databázi skrze sqldeveloper je několik typů a se správným výběrem nám obvykle musí poradit administrátor databáze. Potřebujeme znát kromě loginu a hesla buď tzv. network alias, nebo IP adresu serveru, port a tzv. SID. Pokud chceme používat network alias, musíme mít na svém počítači nakonfigurován soubor tnsnames.ora, který najdeme ve složce klienta v podadresáři /network/admin. Tento soubor obsahuje seznam ORACLE databází, ke kterým se můžeme připojit. Soubor nám musí poskytnout administrátor databáze.

Pokud máme možnost připojovat se pomocí Network alias, klikneme v SQLDeveloper v záložce Connections na zelené plus a vytvoříme nové připojení. Definujeme zde název připojení, uživatelské jméno, heslo a v Connection type vybereme možnost TNS. Pokud máme správně konfigurovaný soubor tnsnames.ora, můžeme v roletce Network alias vybrat cílovou databázi. Tlačítkem Save definici připojení uložíme, tlačítkem Connect se připojíme k databázi.



Obrázek 3 - ORACLE Sqldeveloper - konfigurace připojení k databázi přes TNS záznam

Pokud nemůžeme využít tuto variantu připojení, zvolíme variantu Connection type Basic, vyplníme Hostname a položku SID. Obrázek uvádí variantu, pokud se připojujeme k lokální instalaci ORACLE serveru ve verzi Express.



Obrázek 4 - ORACLE Sqldeveloper - konfigurace přímého připojení k databázi

Poznámka: Položku Role ponecháváme na možnosti default kromě případu, že se připojujeme pod administrátorským účtem SYS, pro který je nutné zvolit roli SYSDBA.

Oproti PostgreSQL v databázi ORACLE jako uživatel nevytváříme vlastní databázi. Po přihlášení máme k dispozici vlastní uživatelské schéma (odpovídá "databázi" v PostgreSQL), kde vytváříme vlastní databázové objekty. Pokud jsou na serveru uchovávána data více nezávislých aplikací, bude mít každá tato aplikace vlastní uživatelský účet a s ním spojené schéma. Přistupovat ze svého účtu do jiného schématu můžeme v případě, že nám vlastník schématu udělí patřičná oprávnění.

1.4 Datová struktura relační databáze

Relační databáze ukládají data do tabulek, na které se můžeme dívat jako na dvojrozměrné pole nebo jako na matici. Pojem "relační" pochází z anglického "relation", což je termín z relační algebry, kde označuje výsledek kartézského součinu nad doménami. Přestože relační databáze vycházejí z relační algebry, její znalost pro užívání databáze není nikterak nutná. V této publikaci se termínu relace a doména budeme vyhýbat, místo toho se podíváme na relační databáze z praktického pohledu. Tabulku relační databáze tvoří 1 až n sloupců, kde každý sloupec představuje jeden atribut ukládaného objektu. Záznamy jednotlivých objektů pak tvoří řádky tabulky. Objektem může být cokoliv z reálného světa, co chceme popsat a uložit v tabulce, například student, učitel, pacient, vyšetření, atd. Tabulku vytvoříme tak, že definujeme příkaz SQL jazyka, ve kterém specifikujeme název tabulky a jednotlivé sloupce (atributy). U každého sloupce uvádíme sadu parametrů, přičemž povinné jsou dva: jméno sloupce a datový typ.

1.4.1 **Jména databázových objektů**

Při práci s databází pojmenováváme nejen tabulky a jejich sloupce, ale veškeré vytvářené objekty. Limity pro pojmenovávání se liší mezi databázovými systémy, hlavně pokud jde o maximální délku jména. Pokud se chceme vyhnout problémům se jmény, dodržujeme následující pravidla:

- používáme pouze písmena anglické abecedy a číslice
- jméno začíná vždy písmenem
- místo mezer používáme znak podtržítka "_"
- používáme srozumitelná jména, ale snažíme se omezit jejich délku (max. 30 znaků)

Databáze standardně nerozlišují velikost písmen v názvech a klíčových slovech, doporučuje se proto používat pro jména buď pouze malá písmena, nebo jen velká.

1.4.2 **Datové typy**

Datový typ nám definuje, jaké hodnoty budeme schopni do daného sloupce ukládat. Základní datová typy jsou text, číslo, datum a LOB. LOB je specifický datový typ, který použijeme v případě, že chceme do tabulky ukládat objemná data jako je obrázek, hudba, video nebo velmi dlouhý text. Od těchto základních typů odvozuje každý databázový systém své specifické typy a podtypy. Nejpoužívanější datové typy v systému ORACLE a PostgreSQL popisuje tabulka.

Tabulka 1 - Datové typy

Obecný typ	ORACLE	POSTGRESQL
text	VARCHAR2 (max. délka)	VARCHAR (max. délka)
číslo	NUMBER (číslic, des. míst)	NUMERIC (číslic, des.míst)
Datum	Date, Timestamp	Date, Timestamp
LOB	BLOB, CLOB	Bytea

U textového sloupce definujeme maximální délku textu (1 až limit databáze), který budeme schopni do sloupce uložit. U číselného sloupce definujeme maximální počet číslic (1 až limit databáze) a počet desetinných míst (0 pro celá čísla). Pro LOB definuje ORACLE podtypy BLOB pro ukládání binárních dat (obrázek, video, hudba) a CLOB pro ukládání dlouhého textu (delší než 4096 znaků). Pokud se pokusíme do sloupce tabulky uložit data neodpovídající specifikovanému datovému typu, ohlásí databáze chybu a data se nevloží. Kontrola na správný datový typ je jedna ze základních kontrol, které databáze nabízejí.

Přestože bychom mohli všechna data ukládat jako datový typ text nebo dokonce LOB, nebylo by to efektivní. Jednak jsou čísla efektivněji ukládána než text, jednak pro čísla a datumy nabízí databáze řadu funkcí, které pro textové položky nemůžeme použít.

1.4.3 **Datový model**

Databázi obvykle netvoří jen jedna tabulka, ale obvykle několik, desítky či stovky tabulek. Struktura tabulek databáze, tzv. datový model, odráží modelovanou skutečnost. Tabulky jsou mezi sebou provázány pomocí klíčů. Klíčem označujeme 1 až n sloupců tabulky. Rozlišujeme 2 typy klíčů: primární klíč a cizí klíč. Každá tabulka by měla obsahovat

právě jeden primární klíč a 0 až N cizích klíčů. Primární klíč musí být definován tak, aby obsah sloupce nebo sloupců, které ho tvoří, byl v každém řádku tabulky unikátní. Jinými slovy hodnota primárního klíče, která je uložena v jednom řádku, se nesmí objevit v žádném dalším řádku. Ochranu před vložením duplicity do primárního klíče zajišťuje databáze.

Pomocí cizích klíčů se definují vazby mezi tabulkami. Vazby (relationship) se definují mezi dvojicí tabulek a mohou být třech typů:

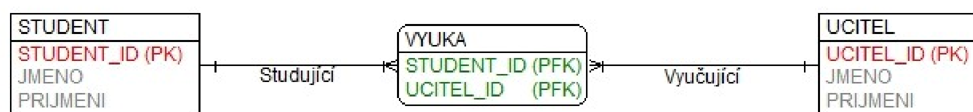
- 1:1
- 1:n
- m:n

Poměry odpovídají počtu řádků, které si ve vázaných tabulkách odpovídají. Vazba 1:1 uvádí, že každému řádků z tabulky A se váže právě jeden řádek z tabulky B. Tato vazba je nejméně častá, protože obvykle není důvod rozdělovat popisovaný objekt do dvou tabulek, místo toho vytvoříme jednu velkou tabulku. Důvodem rozdělení může být limit databáze na počet sloupců jedné tabulky. V tom případě mají obě tabulky stejný primární klíč.

Vazba 1:n je nejčastější vazba, pomocí ní definujeme podřízený vztah tabulky B k tabulce A. Jednomu řádku tabulky A odpovídá 1 až N řádků tabulky B. O tabulce A mluvíme jako o nadřízené nebo rodičovské tabulce, o tabulce B jako o závislé nebo dětské tabulce. Jako klasický příklad vazby 1:n je matka a její děti. Matka může mít více dětí, ale každé dítě má právě jednu matku. Tato vazba se v relační databázi modeluje tak, že primární klíč rodičovské tabulky vložíme do tabulky dětské, kde o něm mluvíme jako o cizím klíči. Dětská tabulka má tak jak vlastní primární klíč tak cizí klíč z rodičovské tabulky. Rodičovská tabulka zůstává nezměněna. Kromě vložení cizího klíče do dětské tabulky definujeme omezení tzv. constraint, čímž databází sdělíme vytvoření vazby. Databáze následně zajistí, že ke každému řádku v dětské tabulce existuje právě jeden řádek v rodičovské tabulce. Při vkládání dat musíme začít vložením řídícího řádku a teprve následně vložit řádek nebo řádky to tabulky dětské. Naopak pokud data mažeme, musíme odstranit nejdřív záznam v dětské tabulce a teprve následně v tabulce řídící.

Třetím typem vazby mezi tabulkami je m:n. Příkladem této situace je například vztah mezi učiteli a studenty. Každý učitel učí více studentů, ale zároveň každý student navštěvuje hodiny několika učitelů. Pokud chceme namodelovat tuto vazbu, musíme vytvořit třetí vazební tabulku, ve které skombinujeme primární klíče tabulky učitelů a studentů. Vazební tabulka tak obsahuje dva cizí klíče, které obvykle tvoří dohromady primární klíč této tabulky. Tímto způsobem dekomponujeme vazbu m:n na dvě vazby 1:n. Nadefinováním tabulek a vazeb mezi nimi vzniká datový model databáze. Jeho schematický nákres ukazuje obrázek.

[1,1]



Obdélníky odpovídají jednotlivým tabulkám s jejich atributy, čáry pak zobrazují vazby mezi nimi. Podle zakončení čar poznáme řídící tabulku od tabulky dětské, u dětské tabulky je zakončení rozvětvené. Někdy je u zakončení přímo připojen popisek, který uvádí, kolik dětských záznamů je pro jeden řídící záznam povoleno.

S návrhem datové struktury databáze (datového modelu) souvisí pojem normalizace. Pod pojmem *normalizace* rozumíme proces zjednodušování a optimalizace navržených struktur

databázových tabulek. Hlavním cílem je navrhnout databázové tabulky tak, aby obsahovaly minimální počet redundantních dat. Správnost navržení struktur lze ohodnotit některou z následujících normálních forem.

1. Nultá normální forma (0NF) - tabulka v nulté normální formě obsahuje alespoň jeden sloupec (atribut), který může obsahovat více druhů hodnot.
2. První normální forma (1NF) - tabulka je v první normální formě, pokud všechny sloupce (atributy) nelze dále dělit na části nesoucí nějakou informaci, neboli prvky musí být atomické. Jeden sloupec neobsahuje složené hodnoty.
3. Druhá normální forma (2NF) - tabulka je v druhé normální formě, pokud obsahuje pouze atributy (sloupce), které jsou závislé na celém klíči.
4. Třetí normální forma (3NF) - tabulka je ve třetí normální formě, pokud neexistují žádné závislosti mezi neklíčovými atributy (sloupci).
5. Čtvrtá normální forma (4NF) - tabulka je ve čtvrté normální formě, pokud sloupce (atributy) v ní obsažené popisují pouze jeden fakt nebo jednu souvislost.
6. Pátá normální forma (5NF) - tabulka je v páté normální formě, pokud by se přidáním libovolného nového sloupce (atributu) rozpadla na více tabulek.

Takto zní oficiální definice normálních forem, v praxi se aplikují první tři, kdy se snažíme v modelu pro každou tabulku definovat primární klíč a sloupce definovat atomicky, tedy tak, aby obsahovaly dále smysluplně nedělitelnou informaci. Jako příklad rozdělení na atomické prvky můžeme uvést například položku bydliště, kterou bychom měli správně rozdělit na atributy (sloupce) ulice, číslo domu, město a PSČ.

Cvičení:

1. Nainstalujte si na svůj počítač server POSTGRESQL pro váš operační systém
2. Vytvořte si vlastní databázi Student.

2 Základy SQL

Výstupy

- Zná základní syntax jednoduchých SQL dotazů
- Umí pomocí SQL prohlížet a zjišťovat strukturu a obsah databázových tabulek
- Umí pomocí SQL vkládat a mazat záznamy v databázi
- Je schopen tvořit sumarizační přehledy dat nad jednou tabulkou
- Rozumí pojmu databázová transakce
- Zná základní příkazy pro tvorbu, změnu a rušení databázových objektů

2.1 Skupiny příkazů

SQL jazyk se výrazně liší od klasických programovacích procedurálních jazyků. Při řešení úlohy neříkáme databází, jak má požadovaný úkol splnit, ale pouze formulujeme příkaz a specifikujeme naše požadavky. Příkazy, které je schopna databáze zpracovat jsou 4 skupiny. Manipulaci s daty obstarávají příkazy ze skupiny DML (Data manipulation language). Pro správu datových struktur a objektů jsou určeny DDL (Data definition language) příkazy. Třetí a čtvrtou skupinou jsou příkazy pro řízení transakcí a oprávnění jednotlivých databázových uživatelů.

2.2 DML příkazy

Nejprve se zaměříme na DML příkazy. Základní DML příkazy jsou 4 a umožňují provádět následující manipulaci s daty.

- SELECT - výběr a zobrazení dat
- INSERT - Vkládání dat
- UPDATE - Změna dat
- DELETE - Smazání dat

2.2.1 SELECT

Jazyk SQL je velice snadný, co se týká slovníku neboli klíčových slov. Nejjednodušší SQL příkaz obsahuje pouhá 2 klíčová slova.

SELECT * FROM jmeno_tabulky

Prvním slovem je jeden ze 4 uvedených příkazů (SELECT), následuje operátor hvězdička (*), kterým říkáme, že chceme získat všechny sloupce tabulky. Druhé klíčové slovo (FROM) uvozuje název tabulky, ze které chceme data získat. Místo operátoru * můžeme zapsat názvy sloupců oddělené čárkou.

SELECT jmeno_sloupce1, jmeno_sloupce2 FROM jmeno_tabulky

Po spuštění toho dotazu nám databáze zobrazí všechny řádky zvolené tabulky. Pokud chceme zobrazení omezit jen na vybrané řádky, použijeme další klíčové slovo WHERE a specifikujeme omezující podmínku. Podmínku tvoří název sloupce, operátor a případně konstanta. Dotaz pak má podobu

SELECT * FROM jmeno_tabulky WHERE jmeno_sloupce1 = 10

nebo

SELECT * FROM jmeno_tabulky WHERE jmeno_sloupce1 = jmeno_sloupce2

První dotaz zobrazí pouze řádky, které mají ve sloupci jmeno_sloupce1 uloženu 10, druhý dotaz zobrazí pouze řádky, které obsahují stejnou hodnotu ve sloupci jmeno_sloupce1 a jmeno_sloupce2. Podmínek je možné specifikovat více, oddělují se pomocí logických operátorů AND a OR.
Zobrazený seznam je možné nechat setřídit dle vybraných sloupců. Jejich seznam specifikujeme na konci dotazu za klíčové slovo ORDER BY. Výchozí je třídění vzestupně, pokud chceme třídit podle některého sloupce sestupně, doplníme klíčové slovo DESC za název sloupce. Názvy sloupců opět oddělujeme čárkou.

SELECT * FROM jmeno_tabulky WHERE jmeno_sloupce1 = jmeno_sloupce2

ORDER BY jmeno_sloupce DESC, jmeno_sloupce2

Kromě výpisu uložených dat umožňuje příkaz SELECT získat základní sumární údaje o obsahu tabulek. K tomuto slouží agregační funkce, které použijeme místo nebo v kombinaci s názvem sloupce. Pro zjištění počtu řádků použijeme funkci COUNT. Tuto funkci lze použít ve třech podobách:

SELECT COUNT(*), COUNT(sloupec1), COUNT (DISTINCT sloupec1) FROM tabulka1

První varianta s hvězdičkou spočítá celkový počet řádků v tabulce1, druhá forma spočítá řádky, které ve sloupci sloupec1 obsahují hodnotu (jsou neprázdné), třetí forma s klíčovým slovem DISTINCT spočítá počet unikátních hodnot ve sloupci sloupec1. Mějme tabulku se třemi řádky:

sloupec1	sloupec2
Ano	1
Ano	2
	3

Výsledek dotazu bude 3, 2, 1, tedy 3 řádky celkem, 2 neprázdné řádky, 1 unikátní hodnota ('Ano').
Mezi agregační funkce patří dále MAX, MIN, AVG, SUM, které pro daný sloupec vypočítají maximum, minimum, aritmetický průměr a celkový součet. Minimum a maximum lze použít pro všechny datové typy (u textových sloupců vrací funkce první a poslední záznam dle abecedy), průměr a sumaci lze počítat pouze nad číselnými datovými typy.

2.2.2 INSERT

Dalším příkazem z rodiny DML je příkaz INSERT pro vkládání záznamů do tabulky. Syntaxe tohoto příkazu je:

INSERT INTO tabulka1 (sloupec1, sloupec2) VALUES (hodnota1, hodnota2)

Tento příkaz vloží do tabulky tabulka1 jeden řádek, přičemž sloupec1 bude obsahovat hodnotu hodnota1 a sloupec2 hodnotu2. Počet sloupců musí odpovídat počtu hodnot. Pokud tabulka obsahuje ještě další sloupce, bude jejich hodnota buď NULL nebo bude rovna výchozí hodnotě sloupce. Výchozí hodnoty sloupců je možné definovat při vytváření tabulky. Připomeňme, že textové hodnoty je nutné uvádět uzavřené v jednoduchých apostrofech ('textová hodnota'). Příkaz INSERT je možné kombinovat s příkazem SELECT, pokud

chceme záznamy místo zobrazení uložit do tabulky. V tomto případě vypadá syntaxe následovně:

INSERT INTO tabulka (sloupec1, sloupec2)

SELECT sloupec3, sloupec4 FROM tabulka2

V tomto případě je klíčové slovo VALUES nahrazeno příkazem SELECT a výsledkem je, že se všechny řádky vybrané příkazem SELECT z tabulky2 uloží do tabulky1. Počet sloupců a jejich datové typy v části SELECT a INSERT musí odpovídat.

2.2.3 UPDATE

Pokud chceme změnit hodnoty záznamů uložených v tabulce, použijeme příkaz UPDATE. Jeho syntaxe je následující:

UPDATE tabulka SET sloupec = hodnota, sloupec2 = hodnota2

Tento příkaz změní hodnotu sloupce na definovanou hodnotu ve všech řádcích tabulky. Místo konstantní hodnoty je možné použít název jiného sloupce či složitější výraz (výpočet). Klíčové slovo SET je povinné, jednotlivá přiřazení jsou oddělena čárkou. Častěji než měnit všechny řádky tabulky potřebujeme měnit hodnoty jen vybraných řádků. V tomto případě musíme doplnit podmínku za klíčové slovo WHERE.

UPDATE tabulka SET sloupec = hodnota WHERE sloupec = hodnota2

Tato varianta změní hodnotu sloupce pouze u těch řádků, kde hodnota sloupce odpovídá hodnotě2. Stejně jako v případě restrikce řádků v příkazu SELECT je možné další podmínky připojovat přes operátory AND nebo OR. Pokud výsledné podmínce neodpovídá žádný řádek, nedojde k chybě, pouze není změněn žádný záznam. SQL dovede v jednom příkaze zaměnit hodnotu dvou sloupců, což standardní procedurální jazyky nedovedou. Je tedy možné napsat:

UPDATE tabulka SET sloupec1 = sloupec2, sloupec2=sloupec1

Tento dotaz korektně zamění hodnoty sloupců.

2.2.4 DELETE

Posledním ze základních DML příkazů je příkaz DELETE, který odstraní záznamy z tabulky. Základní syntaxe je:

DELETE FROM tabulka,

který smaže všechny záznamy v tabulce a jde tedy o velmi destruktivní příkaz. Pokud chceme smazat pouze vybrané záznamy, je nutné stejně jako v případě příkazů UPDATE specifikovat podmínku za klíčové slovo WHERE.

Opomenutí uvedení podmínky za příkazy UPDATE a DELETE je častá začátečnická chyba, která má velmi neblahé důsledky. Proto je nutné s těmito příkazy zacházet vždy velmi opatrně a podmínku testovat nejprve v kombinaci s příkazem SELECT.

2.2.5 Databázové transakce

DML příkazy lze zapouzdřit do tzv. transakcí. V rámci transakce můžeme provést několik DML příkazů, ale teprve po posledním z nich rozhodneme, zda všechny provedené změny budou platné nebo ne. Databáze zajistí, že budou provedeny buď všechny změny, nebo žádná. Pro řízení transakcí existují 2 základní příkazy. Příkaz COMMIT transakci potvrdí, příkaz ROLLBACK transakci zruší. Transakce začíná spuštěním prvního DML příkazu. Změny prováděné v rámci transakce nejsou před spuštěním COMMIT mimo transakci viditelné, jinými slovy, ostatní uživatelé vidí stále stejná data jako před zahájením transakce. Poznámka: transakční režim je nutné v některých databázích či v databázových klientech explicitně aktivovat, například PostgreSQL běží defaultně v autocommit režimu, kdy po každém DML příkazu se automaticky provede COMMIT. Toto chování lze změnit pomocí SET AUTOCOMMIT = off

2.3 DDL příkazy

Pomocí příkazů DDL vytváříme, měníme a rušíme databázové objekty. Slouží k tomu příkazy CREATE, ALTER a DROP. Syntaxe těchto příkazů se liší pro jednotlivé databázové objekty a rozdíly jsou i mezi databázemi. Ukážeme se proto jen základní syntax pro tvorbu a rušení databázových tabulek. Tabulku vytvoříme pomocí příkazů CREATE TABLE. Následuje v závorkách výčet sloupců a jejich specifikace. Sloupce jsou odděleny čárkou, u každého sloupce je nutné definovat minimálně jeho datový typ. Příkaz pro vytvoření jednoduché tabulky v databázi ORACLE může vypadat takto:

```
CREATE TABLE tabulka1 (
        prijmeni VARCHAR2(30),
        datum_narozeni DATE,
        hmotnost NUMBER(3)
)
```

VARCHAR2 je ORACLE datový typ určený pro ukládání textů s variabilní délkou (v závorce je uvedena maximální délka), DATE slouží pro ukládání data a času s přesností na sekundy, NUMBER je datový typ pro ukládání čísel, jejichž maximální velikost a počet desetinných míst určuje parametr/y v závorce.
Názvy datových typů se u databází mírně liší, stejná tabulka pro databázi PostgreSQL by vypadala následovně:

```
CREATE TABLE tabulka1 (
        prijmeni VARCHAR(30),
        datum_narozeni TIMESTAMP,
        hmotnost NUMERIC(3)
)
```

Tabulku zrušíme příkazem DROP TABLE tabulka1 . Pozor, zrušením tabulky nenávratně přijdeme o data, která byla v tabulce uložena. Zachránit nás pak může už jen záloha dat nebo specifické funkce konkrétní databáze (funkce flashback v případě ORACLE). DDL příkazy nelze zařadit do transakce, provádějí se okamžitě a nevratně. Navíc databáze potvrdí všechny

naše dosud nepotvrzené DML příkazy.
Pomocí příkazů ALTER lze obvykle přidávat, přejmenovávat nebo rušit sloupce v tabulce a do určité míry měnit datové typy sloupců. Lze například bez problémů rozšířit v naplněné tabulce maximální délku textového sloupce, nelze už ovšem měnit datumový prvek na číselný.

2.4 Příkazy pro řízení přístupu

Poslední skupinou SQL příkazů jsou příkazy pro řízení přístupových oprávnění. V databázích platí politika, že které objekty vytvořím ty také vlastním a rozhoduji o tom, kdo další k nim bude mít přístup. Oprávnění se udělují na celé objekty, tedy například na celý obsah tabulky. Lze však specifikovat, jaké operace bude moci jiný uživatel nad objektem používat. Typy oprávnění odpovídají DML příkazům. Můžeme tedy zpřístupnit tabulku pro čtení, tím že povolíme příkaz SELECT, ale zakážeme ostatní operace INSERT, UPDATE, DELETE. K udělování oprávnění se používá příkaz GRANT. Syntaxe je

GRANT opravneni ON objekt TO uzivatel

GRANT SELECT ON tabulka1 TO Novak

Po spuštění tohoto příkazu může uživatel Novak spouštět příkazy SELECT nad tabulkou tabulka1. Odejmout přidělená práva můžeme příkazem REVOKE.

`REVOKE opravneni ON objekt` FROM uzivatel

REVOKE SELECT ON tabulka1 FROM Novak

3 Funkce a operátory v SQL

Výstupy:

- Umí použít základní operátory SQL
- Ovládá množinové operátory SQL
- Má přehled a umí využít v databázích dostupné matematické funkce, funkce pro práci s textovými řetězci a funkce pro práci s časovými atributy
- Zná základní agregační funkce SQL

Jazyk SQL nám umožňuje nejen z databáze získávat uložená data, ale zároveň tato data na výstupu modifikovat nejrůznějším způsobem. Slouží k tomu bohatá výbava operátorů a funkcí, kterou dnešní databázové systémy nabízejí. Bohužel standardizace v této oblasti není vysoká, a proto stejné funkce se v jednotlivých systémech jmenují jinak a v některých případech se mírně jinak i chovají.

3.1 Testování funkcí a operátorů

Funkce a operátory se v SQL konstrukcích aplikují na jednotlivé řádky z databáze získaných dat. Pokud si chceme vyzkoušet některou funkci, je nejpřehlednější testovat s co nejjednodušším SQL dotazem a ideálně na jednom záznamu. Nejjednodušší SQL dotaz je:

```
SELECT  1+1 FROM tabulka_s_jednim_radkem
```

Databáze POSTGRESQL umožňuje pro testování funkcí dokonce jednodušší podobu:

```
SELECT 1 + 1
```

Tento příkaz zobrazí jeden řádek s jedním sloupcem s překvapivou hodnotou 2. Tento zápis nelze aplikovat v databázi ORACLE (klíčové slovo FROM je zde vždy povinné), nabízí ale tzv. pseudotabulku DUAL, která obsahuje právě jeden řádek. Prostý součet v ORACLE lze tedy realizovat takto:

```
SELECT 1+1 FROM DUAL
```

V databázi PGSQL jako generátor řádků slouží funkce generate_series(od, do). Tato funkce vrací řádky v intervalu parametrů od do.

```
SELECT 1 + 1 FROM GENERATE_SERIES(1,1)
```

vrátí 1 řádek

```
SELECT cislo + 1  FROM GENERATE_SERIES(5,9) as cislo
```

vrátí 5 řádků s čísly 6 až 10

V dalším textu budou funkce a operátory prezentovány ve variantě pro ORACLE databázi.

V reálné praxi však zpracováváme pomocí funkcí a operátorů data skutečných tabulek. Například zobrazení ceny zboží včetně DPH by vypadalo následovně (předpokládá tabulku s názvem zbozi s číselným sloupcem cena):

```
SELECT cena * 1.21 FROM zbozi
```

3.2 Operátory

3.2.1 Základní operátory

Mezi základní operátory patří operace sčítání (+) a odečítání (-), násobení (*) a dělení (/). Sčítat a odečítat lze číselné konstanty a hodnoty číselných sloupců tabulek. Od datumových sloupců a konstant lze odečítat a přičítat číselné hodnoty, číslo představuje počet přičítaných či odečítaných dnů. Pokud datový sloupec obsahuje i časovou komponentu, lze odečítat a přičítat i desetinné číslo, kdy desetinná část odpovídá části jednoho dne, např.:

```
SELECT datum_narozeni + 5.5 FROM patients
```

přičte k datu narození 5 dnů a 12 hodin (půl dne).

Odečítat lze také dvě data vzájemně, výsledkem je číslo, které odpovídá počtu dnů mezi daty.

Násobit a dělit lze pouze číselné datové typy a číselné konstanty.

Pomocí operátoru lze také spojovat textové řetězce, v databázi ORACLE A PGSQL jde o operátor dvou svislítek (||):

```
SELECT jmeno || ' ' || prijmeni FROM pacient
```

Tento dotaz pojí hodnotu sloupce jména a příjmení a oddělí je mezerou.

Operátory a funkce se používají primárně za klíčovým slovem SELECT nebo při definování podmínek v části WHERE. Jejich vstupem, u funkcí mluvíme o parametrech, jsou buď názvy sloupců tabulky nebo konstanty. Parametry se uvádějí v kulatých závorkách za názvem funkce a oddělují se čárkou. Pokud použijeme funkci v kombinaci s názvem sloupce tabulky, zpracovává funkce či operátor hodnotu každého řádku a výstupem je modifikovaná hodnota pro každý řádek, který SQL dotaz vrátí. O výsledku funkcí mluvíme jako o navrácené hodnotě.

3.2.2 Logické operátory

Mezi operátory patří také výrazy, které využíváme při sestavování složitějších podmínek v části SQL dotazu za WHERE. Jde o tzv. logické operátory: AND , OR a NOT.

Operátory AND a OR spojují dvě podmínky. Pracují tak, že nejprve vyhodnotí pravdivost těchto podmínek (TRUE/FALSE/NULL, pravda/nepravda/NULL) a následně vyhodnotí výsledek dle tabulek.

Tabulka 2 - Logický operátor AND

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

Tabulka 3 - Logický operátor OR

OR	TRUE	FALSE	NULL

TRUE	TRUE		
FALSE	TRUE	FALSE	
NULL	TRUE	NULL	NULL

Operátor NOT má jen jeden parametr, nad kterým provede negaci TRUE => FALSE, FALSE => TRUE.

Podmínka vek > 30 AND vek < 50 je pravdivá(splněna) pro hodnoty sloupce vek mezi 30 a 50, podmínka vek < 30 OR vek > 50 je pravdivá pro věk pod 30 nebo nad 50. Podmínka vek < 30 AND vek > 50 není pravdivá nikdy, podmínka vek > 30 OR vek < 50 je pravdivá vždy.

3.2.3 Množinové operátory

Jiným typem operátorů jsou množinové operátory UNION, UNION ALL, INTERSECT a MINUS. Těmito operátory lze spojovat celé SQL dotazy a získávat tak spojené množiny výsledků. Pomocí UNION a UNION ALL můžeme sloučit výsledky dvou dotazů, kdy k výsledným záznamům prvního dotazu se připojí výsledky druhého dotazu:

```
SELECT patient_id FROM ambulance
UNION
SELECT patient_id FROM nemocnice
```

Počet sloupců prvního a druhého dotazu musí být stejný a musí být stejného datového typu. Rozdíl mezi UNION a UNION ALL je v tom, že UNION odstraňuje duplicitní záznamy, zatímco UNION ALL provede prosté sloučení výsledků (množin).

Operátor INTERSECT provede průnik množin a výsledkem jsou pouze ty záznamy, které jsou obsaženy ve výsledku obou dotazů. Operátorem MINUS získáme množinu řádků, které vrátí první dotaz a které zároveň neobsahuje výsledek druhého dotazu.

3.3 Funkce

Funkce lze dělit podle datových typů jejich parametrů na funkce pro textové hodnoty, číselné hodnoty a datumové hodnoty. Speciální kategorií jsou pak funkce pracující s libovolným typem a s prázdnou hodnotou NULL.

3.3.1 Funkce nahrazující NULL

NULL hodnotu lze ve výsledku nahrazovat konstantou nebo hodnotou jiného sloupce pomocí funkcí NVL, NVL2, COALESCE. Funkce NVL a NVL2 lze požít pouze v databázi ORACLE, funkce COALESCE je dostupná v ORACLE i PGSQL.

Funkce NVL vyhodnotí první parametr a pokud je NULL, výsledkem je druhý parametr. Pokud první parametr není NULL, je výsledkem funkce první parametr.

- SELECT NVL(NULL, 0) FROM DUAL - výsledek je 0
- SELECT NVL(5, 0) FROM DUAL - výsledek je 5
- SELECT NVL(cena, 0) FROM zbozi - výsledkem je buď hodnota ve sloupci cena nebo 0, pokud není na daném řádku cena uvedena.

Funkce NVL2 má o parametr víc, druhý parametr je výsledkem funkce v případě, kdy první parametr není NULL, třetí parametr je výsledkem, pokud první parametr je NULL.

- SELECT NVL2(999,0,1) FROM DUAL - výsledek je 0, protože 999 není NULL

Funkce COALESCE má neomezený počet parametrů a jejím výsledkem je první NOT NULL parametr.

- SELECT COALESCE (cena_akce, cena_prodej, cena_nakup, 0) FROM zbozi -

testuje pro každý řádek tabulky zbozi postupně jednotlivé sloupce a vrátí tu hodnotu, která není NULL.

3.3.2 Funkce GREATEST a LEAST

Funkce GREATEST a LEAST patří mezi funkce s neomezeným počtem parametrů a lze je využít pro všechny datové typy, které lze nějakým způsobem třídit. Funkce porovnává všechny zadané parametry a vrací ten největší resp. nejmenší.

SELECT GREATEST (3,6,9,0), LEAST (3,6,9,0) FROM DUAL

Výsledek je jeden řádek a dva sloupce s hodnotami 9 a 0.

3.3.3 Podmíněný výraz CASE a funkce DECODE

3.3.4 Datumové funkce

Nejvýznamnější funkce pro datový typ datum jsou:

- Funkce vracející aktuální datum a čas
- Funkce pro práci s časovým intervalem
- Funkce pro formátování vstupního či výstupního data

Standardní funkcí, která vrátí systémové datum je CURRENT_DATE, v jednotlivých databázových systémech se však může lišit její návratová hodnota. V ORACLE je synonymem funkce SYSDATE, která vrátí aktuální datum i čas s přesností na sekundy. V databáze PGSQL vrací tato funkce pouze datum. Standardní funkcí pro získání aktuálního času je CURRENT_TIME, která však není dostupná ORACLE. Funkce CURRENT_TIMESTAMP je dostupná v ORACLE i PGSQL a vrací datum i čas.

Jak bylo zmíněno výše, pro práci s datumy lze využívat operátory + a - . Takto lze pracovat, pokud pracujeme v jednotkách dnů či týdnů. Problém je, pokud potřebujeme pracovat s přesností na měsíce nebo roky. Kalendářní měsíc má 28 - 31 dnů, rok má 365 nebo 366 dnů. Vyjádřit rozdíl mezi dvěma daty jako počet měsíců nebo let bez zaokrouhlování není tedy zcela triviální. ORACLE nabízí pro tyto úlohy 2 velmi užitečné funkce: ADD_MONTHS a MONTHS_BETWEEN. První umožňuje přičítat či odečítat měsíce k danému datu, druhá vrací počet měsíců mezi dvěma daty. Protože rok má vždy 12 měsíců, je možné s pomocí uvedených funkcí pracovat i s roky. Příklad ukazuje, jak zjistit současný věk osoby, pokud máme v databázi uloženo datum narození. Současný věk odpovídá rozdílu aktuálního data a data narození v měsících, pokud toto číslo podělíme 12 a zaokrouhlíme dolů na celé číslo, dostáváme věk v letech.

SELECT TRUNC (MONTHS_BETWEEN (CURRENT_DATE, date_of_birth)/12) FROM patients

Databáze PGSQL nabízí funkci AGE pro výpočet rozdílu mezi dvěma daty.

SELECT AGE (current_date, date_of_birth) FROM patients

Formátování datumu

Datum a čas je databází interně ukládán jako číslo, které udává počet dnů od databázově specifického výchozího data. Pokud chceme datum zobrazit jako výsledek SQL dotazu, databáze transformuje toto číslo do defaultního formátu. Tento formát nám ale často nevyhovuje. Máme ale k dispozici (ORACLE i PGSQL) formátovací funkci TO_CHAR, která nám umožňuje přesně specifikovat výstupní formát. Funkce má dva parametry, prvním je formátované datum (sloupec tabulky s datumovým typem), druhým je specifikace formátu ohraničená apostrofy. Pro specifikaci formátu se využívají zástupné znaky, seznam nejčastěji používaných uvádí tabulka.

Tabulka 4 - Symboly pro formátování datumu

Symbol	Popis
dd	den měsíce
mm	kalendářní měsíc (1-12)
yyyy	kalendářní rok
hh24	hodiny (0-23)
mi	minuty (0-59)
ss	sekundy (0-59)
ww	číslo týdne v roce

Pokud chceme zobrazit datum a čas dle českých zvyklostí, specifikujeme formát následovně

SELECT TO_CHAR(CURRENT_TIMESTAMP, 'dd. mm. yyyy hh24:mi:ss') FROM DUAL

Výsledek je 20. 7. 2013 10:13:23.

Pokud chceme formát datumu vhodný ke třídění, použijeme

SELECT TO_CHAR(CURRENT_TIMESTAMP, 'yyyy-mm-dd') FROM DUAL

S formátem datumu je problém i při vkládání dat do databáze. Aby databáze byla schopna převést vkládané datum do svého interního formátu, musíme opět přesně specifikovat vkládaný formát. Použijeme k tomu funkci TO_DATE, jejímž prvním parametrem je textový řetězec představující vkládané datum, druhý parametr je opět specifikace formátu. Specifikace formátu je shodná jako v případě funkce TO_CHAR. Příkaz INSERT pro vložení data do tabulky patients, vypadá takto

INSERT INTO patients (date_of_birth) VALUES (TO_DATE('13.3.1950','dd.mm.yyyy'))

Nezkušení uživatelé databáze často funkce TO_CHAR a T0_DATE zaměňují, což může způsobit buď chybu při vykonávání SQL příkazu, v horším případě pak vrácení či vložení chybných dat. Pravidlo je přitom jednoduché, pokud je prvním parametrem textový sloupec nebo text v apostrofech, musí jít vždy o funkci TO_DATE, pokud je parametrem datumový sloupec nebo zmíněné funkce vracející aktuální datum a čas, je na místě funkce TO_CHAR. Funkce TO_CHAR a TO_DATE lze kombinovat, například pokud chceme přeformátovat datumovou konstantu.

SELECT TO_CHAR(TO_DATE('22.3.2000','dd.mm.yyyy'), 'yyyy-mm') FROM DUAL;

Dotaz v příkladu nejprve specifikované datum převede do interního formátu a následně ho funkce TO_CHAR zobrazí ve formátu rok-měsíc.

Extrahovat komponenty data lze také pomocí funkce EXTRACT. Pomocí této funkce můžeme z data získat rok (YEAR), měsíc (MONTH), den (DAY)

SELECT EXTRACT(YEAR FROM TO_DATE('22.3.2000','dd.mm.yyyy')) FROM DUAL;

3.3.5 Textové funkce

Databáze nabízí také řadu funkcí pro práci s textovými řetězci. Mezi nejpoužívanější a nejrozšířenější patří funkce uvedené v tabulce.

Tabulka 5 - Funkce pro práci s textem

Název funkce	Popis funkce
SUBSTR(text, od, počet)	Vrací podřetězec textu dle pozice
INSTR(text, subtext)	Hledání podřetězce v textu, vrací pozici nalezeného podřetězce (pouze ORACLE)
STRPOS(text, subtext)	obdoba INSTR (POSTGRESQL)
LOWER (text), UPPER(text)	Převede text na malá, resp. velká písmena
INITCAP (text)	Převede první písmeno slov na velké písmeno, ostatní na malá
LTRIM (text), RTRIM (text)	Odstranění mezer (nežádoucích znaků) z textu (zleva, zprava)
REPLACE(text, puvodni, nove)	Nahrazení podřetězce za jiný
TRANSLATE(text, nahradit_co, nahradit_cim)	Nahrazení po znacích
LENGTH(text)	Vrací délku textu ve znacích

Pokud chceme z textu získat určitý podřetězec, použijeme funkci SUBSTR. Parametrem je zpracovávaný řetězec, druhým parametrem je pořadí prvního znaku, který chceme extrahovat, třetím nepovinným parametrem je počet znaků, které chceme extrahovat. Pokud chceme zkrátit řetězec na prvních 5 znaků, použijeme funkci SUBSTR následovně:

SELECT SUBSTR('dlouhý text', 1, 5) FROM DUAL;

Funkce, která prohledává text na výskyt specifikovaného podřetězce, se jmenuje v ORACLE INSTR, v PGSQL pak STRPOS. Parametrem je prohledávaný řetězec a hledaný text, výsledkem je pořadí prvního znaku nalezeného řetězce nebo nula, pokud podřetězec není nalezen.

Pokud chceme získat z textu určitou část, která je oddělena definovaným symbolem, použijeme kombinaci funkcí SUBSTR a INSTR.

SELECT SUBSTR('Výsledek:67',INSTR('Výsledek:67',':')+1) FROM DUAL

Funkce INSTR nejprve nalezne symbol ':', vrátí pořadí tohoto znaku, který předá funkci SUBSTR. Ta extrahuje text za tímto znakem. Jednička je připočtena proto, aby výsledek neobsahoval úvodní dvojtečku.

Funkce, která vrátí počet znaků v řetězci, se jmenuje v ORACLE i PGSQL LENGTH. Parametrem je analyzovaný řetězec.

Při práci s textem je často potřebné sjednocení velikosti písmen. Celý text můžeme snadno převést na malá písmena (LOWER) nebo velká písmena (UPPER). Tyto funkce využijeme, pokud budeme chtít pracovat s řetězci bez rozlišení velikosti písmen. Funkci INITCAP využijeme, pokud chceme, aby každé slovo řetězce začínalo velkým písmenem, například v případě vlastních jmen.

SELECT INITCAP('klIMeŠ daNIel') FROM DUAL -- výsledek je Klimeš Daniel

Dvojice funcí LTRIM a RTRIM umožňuje odstranit nežadoucí znaky z levé, resp. z pravé strany řetězce. Defaultně se odstraňují mezery, pokud chceme odstranit specifikované znaky, uvedeme jejich výčet jako druhý parametr.

SELECT RTRIM('text ++++   ', '+ ') FROM DUAL

Dotaz odstraní zprava znaky mezer a plus, výsledkem je 'text'.

K nahrazování znaků v řetězcích slouží funkce REPLACE a TRANSLATE. Pomocí REPLACE nahradíme podřetězec definovaný v druhém parametru za text uvedený jako třetí parametr. Pokud neuvedeme třetí parametr, bude nalezený podřetězec odstraněn. Funkce REPLACE defaultně nahrazuje všechny nalezené podřetězce. Funkce TRANSLATE slouží pro nahrazení jednotlivých znaků. Druhým jejím parametrem je seznam nahrazovaných znaků, třetí parametr pak obsahuje seznam znaků nahrazujících. Nahrazuje se vždy první znak druhého parametru za první znak třetího parametru, druhý za druhý, atd. Pokud chybí třetí parametr, jsou znaky druhého parametru odstraněny. Funkce se využívá, např. pokud chceme odstranit českou diakritiku z textového řetězce.

SELECT TRANSLATE('žluťoučký kůň','žťčýůň','ztcyun') FROM DUAL -- výsledek je 'zlutoucky kun'

3.3.6 Funkce s číselným parametrem

Široká škála funkcí existuje i pro číselné datové typy. Základní přehled uvádí tabulka.

Tabulka 6 - Funkce pro práci s čísly

Název funkce	Popis funkce
SIN, COS, TAN	Goniometrické funkce s jedním parametrem, kterým je úhel v radiánech
ABS(číslo)	Vrací absolutní hodnotu z čísla

POWER(číslo, exponent)	Umocňuje číslo na exponent
SQRT (číslo)	Vrací druhou mocninu z čísla
LN(číslo), LOG (číslo)	Vrací přirozený, dekadický logaritmus čísla
ROUND(číslo, přesnost), CEIL(číslo), TRUNC(číslo, přesnost), FLOOR(číslo)	Zaokrouhluje číslo
MOD	Vrací zbytek po celočíselném dělení

K zaokrouhlování čísel slouží trojice funkcí ROUND, CEIL, TRUNC (resp. FLOOR). Nabízí všechny základní možnosti zaokrouhlování. ROUND zaokrouhluje od 5 nahoru, funkce CEIL zaokrouhluje vždy nahoru, funkce TRUNC nebo FLOOR zaokrouhluje vždy dolů (odříznutí).

SELECT ROUND(5.5), TRUNC(5.5), FLOOR(5.5), CEIL (5.5) FROM DUAL

Výsledkem dotazu je 6, 5, 5, 6

U funkcí ROUND a TRUNC je možné specifikovat druhý parametr a určit jím, na kolik desetinných míst se má zaokrouhlení provést.

Uvedený výčet funkcí je jen základ ze široké nabídky některých databázových systémů. Celou nabídku a podrobnosti k jednotlivým funkcím je třeba vždy hledat v dokumentaci daného databázového systému.

3.3.7 Agregační funkce

Speciální skupinou jsou funkce agregační. Zatímco dosud zmíněné funkce vrací jednu hodnotu pro každý řádek zpracovávaného SQL dotazu, agregační funkce vrací jen jeden řádek, tedy agregují všechny řádky do jedné hodnoty. Seznam standardních funkcí uvádí tabulka

Tabulka 7 - Agregační funkce

Název funkce	Popis funkce
COUNT	Počet řádků, které jsou výsledkem SQL dotazu
AVG(sloupec)	Vypočítá aritmetické průměr sloupce
SUM(sloupec)	Vypočítá sumární součet sloupce
MIN(sloupec)	Vrací minimum sloupce
MAX(sloupec)	Vrací maximum sloupce
STDDEV(sloupec)	Počítá standardní odchylku
MEDIAN(sloupec)	Počítá medián ze sloupce

Funkci COUNT lze použít ve třech variantách:

SELECT COUNT(*), COUNT(date_of_birth), COUNT(DISTINCT date_of_birth) FROM patients

Varianta s hvězdičkou vrací prostý počet řádku SQL dotazu, druhá varianta s názvem sloupce vrací počet řádků s vyplněnou hodnotou daného sloupce (NOT NULL), třetí varianta s klíčovým slovem DISTINCT vrací počet unikátních hodnot v daném sloupci.

Pro výsledek musí platit vždy

COUNT(*) >= COUNT(sloupec) >= COUNT(DISTINCT sloupec)

Funkce AVG, STDDEV a SUM lze použít pouze pro číselné sloupce, sloupce MIN, MAX pro všechny základní datové typy.

Agregační funkce nelze kombinovat s ostatními funkcemi. Nelze:

SELECT COUNT(*), LENGTH (patient_id) FROM patients.

Výjimkou jsou funkce, které vrací jednu hodnotu jako např. CURRENT_DATE. I v tomto případě je však doporučeno aplikovat na tyto funkce funkci agregační:

SELECT COUNT(*), MAX(CURRENT_DATE) FROM patients.

Bez omezení je možné aplikovat standardní funkce na výsledek agregační funkce:

SELECT ROUND(COUNT(*)/10) FROM patients

3.4 Cvičení

Jak extrahujete posledních 5 znaků z textového řetězce?

Jak zjistíte, kolikátý den **v roce** právě je?

4 Pokročilé SQL

Výstupy:

- Připomene si problematiku vytváření vztahů mezi tabulkami
- Umí získat data spojením více tabulek
- Rozumí rozdílu mezi vnitřním a vnějším spojením tabulek
- Zná možnosti zanoření SQL dotazů

Ve druhé kapitole jsme se seznámili se základní konstrukcí SQL dotazů. Víme, jak získat požadované sloupce, jak omezit výpis na určité řádky a jak získat sumární data pomocí agregačních funkcí. V této kapitole si ukážeme agregování dat pomocí klíčových slov GROUP BY a HAVING. Protože databáze jsou jen zřídka tvořeny jednou tabulkou, vysvětlíme si způsob získávání dat z více tabulek pomocí tzv. spojování (joining) tabulek. V

poslední části kapitoly budou vysvětleny možnosti vytváření složitějších zanořených SQL dotazů.

4.1 Seskupování dat

Podívejme se nejprve na možnost pokročilé agregace. Mějme tabulku pacientů se sloupci identifikátor pacienta, pohlaví a datumu narození

```
CREATE TABLE patients
(
    patient_id VARCHAR(10),
    sex VARCHAR(1)  ,
    date_of_birth TIMESTAMP
);
```

Vložíme několik řádků:

```
INSERT INTO patients (patient_id, sex, date_of_birth) VALUES ('pat1','F',TO_DATE('2.4.1940','dd.mm.yyyy'));
INSERT INTO patients (patient_id, sex, date_of_birth) VALUES ('pat2','M',TO_DATE('30.3.1950','dd.mm.yyyy'));
INSERT INTO patients (patient_id, sex, date_of_birth) VALUES ('pat3','F',TO_DATE('13.8.1947','dd.mm.yyyy'));
INSERT INTO patients (patient_id, sex, date_of_birth) VALUES ('pat4','M',TO_DATE('23.11.1987','dd.mm.yyyy'));
INSERT INTO patients (patient_id, sex, date_of_birth) VALUES ('pat5','F',TO_DATE('3.9.1975','dd.mm.yyyy'));
```

Nyní se budeme snažit získat sumární přehled o obsahu tabulky. Zajímá nás, kolik záznamů tabulka obsahuje, kolik je v ní žen kolik mužů a v jakých věkových kategoriích. Počet řádků již zjistit umíme:

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

Snadno také zjistíme, počet unikátních hodnot v jednotlivých sloupcích:

```
SELECT COUNT(DISTINCT patient_id), COUNT(DISTINCT sex), COUNT(DISTINCT date_of_birth) FROM patients ;
```

Z výsledku je vidět, že máme tabulku s 5 řádky, kde sloupce patient_id, date_of_birth obsahují vždy unikátní hodnotu, sloupec sex obsahuje jen 2 unikátní hodnoty, buď F nebo M.

Pokud chceme získat přehled, kolik je v tabulce žen a kolik mužů, můžeme sestavit 2 dotazy:

```
SELECT COUNT(*) FROM patients WHERE sex = 'F';
SELECT COUNT(*) FROM patients WHERE sex = 'M';
```

SQL standard však nabízí elegantnější způsob, jak tento výsledek získat v jediném dotazu. Slouží k tomu klíčové slovo GROUP BY, za kterým specifikujeme název sloupce nebo sloupců, podle kterých chceme data agregovat. GROUP BY se umísťuje v SQL dotazu

za definici podmínky (WHERE), případně za název tabulky, pokud podmínka není specifikována. Počty pacientů dle pohlaví lze získat následovně:

SELECT sex, COUNT(*) FROM patients GROUP BY sex;

Výsledkem jsou 2 řádky (odpovídá počtu unikátních hodnot v agregovaném sloupci). Agregační funkce COUNT počítá řádky zvlášť pro každou kategorii agregovaného sloupce. Použitím klauzule GROUP BY značně omezujeme možnosti výrazů za klauzulí SELECT. Můžeme zde uvést pouze názvy sloupců uvedených za GROUP BY a agregační funkce. Častou chybou je pokus vložit za SELECT název neagregovaného sloupce:

SELECT sex, date_of_birth, COUNT(*) FROM patients GROUP BY sex;

Tento dotaz nedává logický smysl, databáze neví, jaké datum narození má zobrazit (výsledek agregace jsou dva řádky, tabulka ale obsahuje 5 různých dat narození). Je ale možné požadovat pro každé pohlaví nejstaršího a nejmladšího pacienta:

SELECT sex, MIN(date_of_birth) nejstarsi, MAX(date_of_birth) nejmladsi, COUNT(*) FROM patients GROUP BY sex;

Agregovat lze podle více sloupců i podle modifikovaných sloupců. Následující dotaz vrátí přehled počtu pacientů agregovaných přes pohlaví a přes dekádu data narození.

SELECT sex, TRUNC(EXTRACT (YEAR FROM date_of_birth)/10) AS dekada, COUNT(*) FROM patients

GROUP BY sex, TRUNC(EXTRACT (YEAR FROM date_of_birth)/10)

Povšimněte si, že výraz, který tvoří agregovaný sloupec za SELECT, musí odpovídat výrazu za GROUP BY. Výraz TRUNC(EXTRACT (YEAR FROM date_of_birth)/10) nejprve extrahuje rok z data narození, tuto hodnotu podělí 10 a funkcí TRUNC provede zaokrouhlení dolů, čímž dostáváme dekádu narození pacienta.

Co v případě, že bychom chtěli ve výsledku vidět je záznamy s hodnotou COUNT(*) větší než 1? Tuto podmínku nemůžeme specifikovat za klíčové slovo WHERE, protože podmínky za WHERE se aplikují PŘED vlastní agregací na primární data, která do agregace teprve vstupují. Filtrovat agregovaný záznam je možné pomocí HAVING, která se umísťuje za GROUP BY výraz:

SELECT sex, TRUNC(EXTRACT (YEAR FROM date_of_birth)/10) AS dekada, COUNT(*) FROM patients

GROUP BY sex, TRUNC(EXTRACT (YEAR FROM date_of_birth)/10)

HAVING COUNT(*) > 1

Podle výsledku agregačních funkcí je možné i třídit pomocí ORDER BY:

SELECT sex, TRUNC(EXTRACT (YEAR FROM date_of_birth)/10) AS dekada, COUNT(*) FROM patients

GROUP BY sex, TRUNC(EXTRACT (YEAR FROM date_of_birth)/10)

HAVING COUNT(*) > 1

ORDER BY COUNT(*)

Pořadí klíčových slov SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY je dané a nelze je měnit.

4.2 Práce s více tabulkami

Dosud jsme pomocí SQL příkazů pracovali pouze s jednou tabulkou. Databáze jsou však v drtivém počtu tvořeny sadou tabulek, jejichž struktura odpovídá modelované realitě. Tabulky jsou mezi sebou svázané vazbou 1:1 nebo 1:n za pomoci cizích klíčů, viz kapitola datový model. SQL standard umožňuje pracovat s více tabulkami pomocí tzv. join operace. Mluvíme o spojování tabulek. Spojování tabulek je dvojího typu, existuje vnitřní a vnější spojení. Rozdíl nejlépe ilustruje příklad, kdy máme 2 tabulky s vazbou 1: n, tabulka pacientů a tabulka jejich vyšetření, každý pacient může mít 0 až n vyšetření, primárním klíčem v tabulce pacientů je sloupec id_pacienta, který slouží jako cizí klíč v tabulce vyšetření:

Tabulka 8 Tabulka pacientů

patient_id	jmeno	prijmeni
1	Jan	Starý
2	Karel	Nový
3	Olga	Mladá

Tabulka 9 Tabulka vyšetření

patient_id	datum_vysetreni	hmotnost
1	1.2.2012	66
1	1.6.2012	70
2	14.7.2013	69

Tabulka 10Výsledek vnitřního spojení

patient_id	jmeno	prijmeni	datum_vysetreni	hmotnost
1	Jan	Starý	1.2.2012	66
1	Jan	Starý	1.6.2012	70
2	Karel	Nový	14.7.2013	69

Tabulka 11Výsledek vnějšího spojení

patient_id	jmeno	prijmeni	datum_vysetreni	hmotnost
1	Jan	Starý	1.2.2012	66
1	Jan	Starý	1.6.2012	70
2	Karel	Nový	14.7.2013	69
3	**Olga**	**Mladá**		

Výsledkem vnitřního spojení jsou řádky, které existují v obou spojovaných tabulkách, řádky, které existují pouze v jedné z tabulek, jsou vynechány. Oproti tomu vnější spojení umožňuje získat všechny řádky z jedné tabulky a k nim připojit existující řádky v druhé tabulce. Jedna ze spojovaných tabulek je u vnějšího spojení řídící, k níž se dle podmínky váží řádky druhé tabulky.

Pro získání dat z více tabulek slouží v SQL klíčové slovo JOIN, které se umísťuje mezi názvy spojovaných tabulek. Následuje klíčové slovo ON, po kterém je nutné definovat způsob propojení. Pokud použijeme samotné slovo JOIN, provede se vnitřní spojení:

```
SELECT p.id_pacienta, p.jmeno, p.prijmeni, v.datum_vysetreni, v.hmotnost
FROM pacienti p JOIN vysetreni v ON p.id_pacienta = v.id_pacienta
```

Protože pracujeme s více tabulkami, je nutné sloupce identifikovat plným jménem, které se skládá z názvu tabulky a názvu samotného sloupce, který oddělíme tečkou. Místo plného názvu tabulek můžeme definovat zkrácené pojmenování v části FROM, zkratku zapíšeme přímo za název tabulky. Zkratku tabulky pak používáme ve všech částech SQL dotazu. Způsob spojení tabulek je obvykle definováno podmínkou za klíčovým slovem ON. Podmínka definuje, které řádky se spolu mají párovat. Pokud podmínku nedefinujeme, vzniká tzv. kartézský součin, kdy se každý řádek jedné tabulky spojí s každým řádkem druhé tabulky. Výsledná množina má pak m x n řádků, což u větších tabulek může být enormní počet. Toto chování je ve většině případů nežádoucí, proto musíme být při definování podmínky spojení velmi pozorní. Databáze se tak bude chovat i v případě, kdy dle spojovací podmínky odpovídá jednomu řádku první tabulky více řádků v tabulce druhé. Hodnoty řádku první tabulky se kopírují ke každému řádku druhé tabulky. Výsledkem spojení tabulek je n řádků, kde n je u vnitřního spojení v rozsahu 0 až m * n, u vnějšího spojení v rozsahu m až m * n.

Pokud chceme provést vnější spojení tabulek, doplníme před JOIN jedno z dalších klíčových slov: LEFT, RIGH nebo FULL. Pokud chceme, aby řídící byla uvedená první, použijeme klíčové slovo LEFT JOIN, pokud chceme mít řídící druhou tabulku, použijeme RIGHT JOIN. Extrémem je tzv. úplné spojení (full join), jehož výsledkem jsou všechny řádky obou tabulek. Tento typ spojení však použijeme jen zřídka, navíc je pro databáze výkonnostně nejnáročnější. V našem případě použijeme variantu LEFT JOIN:

```
SELECT p.id_pacienta, p.jmeno, p.prijmeni, v.datum_vysetreni, v.hmotnost
FROM pacienti p LEFT JOIN vysetreni v ON p.id_pacienta = v.id_pacienta
```

Alternativou k syntaxi JOIN ON je přímý výčet spojovaných tabulek za FROM a specifikace spojovací podmínky za WHERE. Tato konstrukce může být v případech spojování více tabulek lépe čitelná. Lze ji však standardně použít jen pro vnitřní spojení. Stejný výsledek jako v prvním případě dostaneme i po spuštění této varianty:

```
SELECT p.id_pacienta, p.jmeno, p.prijmeni, v.datum_vysetreni, v.hmotnost
FROM pacienti p, vysetreni v WHERE p.id_pacienta = v.id_pacienta
```

4.2.1 3 a více tabulek

Pokud potřebujeme získat data z více jak dvou tabulek, syntaxe zůstává stejná, pomocí výrazu JOIN připojíme další tabulku. Mějme tabulku RTG vyšetření, na kterých může být zjištěna u daného pacienta 0 až n zlomenin. Tabulka zlomenin je vázána s tabulkou rtg pomocí klíče vysetreni_id.

```
CREATE TABLE rtg
(
vysetreni_id NUMERIC(9),
patient_id VARCHAR(10),
)
```

```
CREATE TABLE zlomeniny
(
zlomenina_id NUMERIC(9),
vysetreni_id NUMERIC(9),
lokalizace VARCHAR(50)
)
```

Naplníme tabulky daty, pacient pat 1 měl 2 vyšetření, jedno negativní, druhé ukázalo 2 zlomené kosti. Pacient pat2 měl jedno vyšetření bez nálezu.

```
INSERT INTO rtg (patient_id, vysetreni_id) VALUES ('pat1', 1);
INSERT INTO rtg (patient_id, vysetreni_id) VALUES ('pat1', 2);
INSERT INTO rtg (patient_id, vysetreni_id) VALUES ('pat2', 3);
INSERT INTO zlomeniny (zlomenina_id, vysetreni_id, lokalizace) VALUES (1, 2, 'femur');
INSERT INTO zlomeniny (zlomenina_id, vysetreni_id, lokalizace) VALUES (2, 2, 'ulna');
```

Vnitrřní spojení bychom provedli takto:

```
SELECT * FROM patients p LEFT JOIN rtg ON p.patient_id = rtg.patient_id LEFT JOIN zlomeniny z ON rtg.vysetreni_id = z.vysetreni_id
```

Výsledkem jsou pouhé 2 řádky, které obsahují popis dvou zlomeni pacienta pat1.

Pokud chceme získat přehled o všech pacientech, jejich RTG vyšetřeních a případně zjištěných zlomeninách musíme obě spojení definovat jako vnější:

```
SELECT * FROM patients p LEFT JOIN rtg ON p.patient_id = rtg.patient_id LEFT JOIN zlomeniny z ON rtg.vysetreni_id = z.vysetreni_id
```

V totmo případě je výsledkem 8 řádků, popis zlomenin je ovšem jen u dvou z nich, v ostatních řádcích ve sloupci popis je hodnota NULL.

Vnitřní spojení více tabulek můžeme provést i bez klauzule JOIN:

```
SELECT * FROM patients p, rtg, zlomeniny z WHERE p.patient_id = rtg.patient_id AND rtg.vysetreni_id = z.vysetreni_id
```

Obdobně postupujeme i při spojování většího množství tabulek. Maximálně možný počet spojovaných tabulek závisí na databázovém systému.

4.3 Vnořené dotazy

Připomeňme si, co s pomocí SQL jazyka již umíme. Získat konkrétní hodnotu zvoleného sloupce, filtrovat konkrétní řádek z libovolné tabulky, pomocí agregačních funkcí a klauzule GROUP BY získat sumární přehled o obsahu zvolených sloupců a umíme propojit záznamy ve více tabulkách pomocí výrazu JOIN ON. Pro čerpání primárních dat z databáze pro další zpracování například ve statistickém software je to zcela dostačující. Zdaleka to ale není vše, co relační databáze a standard SQL nabízejí. Pokročilé SQL začíná možností vnořených (nested) dotazů.

Vnořený dotaz má stejnou strukturu jabo běžný dotaz, pouze je uzavřen v kulatých závorkách a umístěn v nadřazeném dotazu na jednom z těchto míst:

- Na místě výčtu sloupců mezi slovy SELECT a FROM
- Na místě názvu tabulky za FROM
- Jako součást podmínky za slovem WHERE

Za klíčovým slovem SELECT můžeme použít zanořený dotaz, který vrátí právě jeden sloupec a právě jeden řádek. Tuto možnost využijeme, pokud chceme do přehledu či do výpočtu získat výsledek agregační funkce, např. chceme procenticke zastoupení. Představme si tabulku s daty o denní spotřebě léků, ze které chceme získat přehled, kolik procent se vyčerpalo daný den.

```
SELECT datum, mnozstvi, mnozstvi / (SELECT SUM(mnozstvi) FROM spotreba) * 100
FROM spotreba
```

Uvedený dotaz je složen ze dvou částí. Základem je prostý SELECT do tabulky spotreba, odkud získáme data sloupců datum a mnozstvi. Do tohoto dotazu je vložen vnořený dotaz, který pomocí agregační funkce SUM získá celkové spotřebované množství. Tímto číslem dělíme hodnotu každého řádku tabulky spotřeby a násobíme stem, čímž získáme spotřebu daného dne v procentech.

Vnořený dotaz je také možné uvést za klíčové slovo FROM a použít ho tak místo názvu tabulky. Tento postup použijeme při sestavování složitých dotazů, kdy začneme jednodušším dotazem, jehož výsledek použijeme v nadřízeném dotazu k další manipulaci. Tento typ zanoření použijeme také v případě, kdy potřebujeme rychle získat počet řádků, které vrací náš dotaz. Mějme dotaz:

```
SELECT * FROM spotreba WHERE mnozstvi > 100
```

Pokud tabulka spotreba obsahuje tisíce a více řádků, netušíme, kolik řádků dotaz vrátil, dokud nenecháme všechny výsledné řádky zobrazit, což je při ladění dotazů velmi neefektivní. Nejrychlejší způsob, jak získat počet řádků laděného dotazu, je jeho zapouzdření do vnořeného dotazu a aplikace agregační funkce COUNT:

```
SELECT COUNT(*) FROM (
  SELECT * FROM spotreba WHERE mnozstvi > 100)
```

Platí, že vnořené dotazy na pozici za FROM je možné vždy spustit samostatně, tedy nezávisle na nadřízeném dotazu. Zanoření je možné opakovat na další vyšší úrovni. Počet možných zanoření je závislé na limitech daného databázového systému.

Třetím a nejčastějším umístění vnořeného dotazu je v podmínce za WHERE. Zde může být využit jako operand podmínky nebo v kombinaci s výrazem (NOT) EXISTS jako samostatná podmínka. Dotaz vkládáný jako operand může být umístěn buď přímo za operátor (=, <, >, <>), nebo s použitím modifikátoru ANY nebo ALL. Pokud je vnořený dotaz přímo za operátorem, musí dotaz vracet právě jeden sloupec a právě jeden řádek. Typicky se zde používají dotazy s agregační funkcí. Pokud použijeme kromě operátoru také modifikátor, zůstává omezení na jeden sloupec, ale řádků může dotaz vracet 0 až N. Poslední možností je umístění vnořeného dotazů za výraz EXISTS. V této variantě není počet sloupců vnořeného dotazu významný, používá se buď * nebo jakákoliv konstanta (1). Podle počtu vrácených řádků se vyhodnotí pravdivost výrazu EXISTS. Pokud dotaz nevrátí žádný řádek, je výsledek FALSE, pokud vrátí 1 až N řádků, je výsledek výrazu TRUE. Pokud použijeme negaci v podobě výrazu NOT EXISTS je výsledek opačný.

U vnořených dotazů umístěných za WHERE budeme až na výjimky definovat podmínku, která prováže vnořený dotaz s rodičovským dotazem. Mějme dvě tabulky, jedna s názvem student obsahuje jména studentů, druhá tabulka s názvem zkoušky obsahuje informace o složených zkouškách jednotlivých studentů. Pomocí spojení (JOIN) těchto tabulek můžeme získat přehled o absolvovaných zkouškách jednotlivých studentů. Co ale v případě, že chceme získat seznam studentů, kteří doposud žádnou zkoušku nesložili a nemají žádný řádek v tabulce zkouska. Právě v těchto případech využijeme vnořený dotaz s výrazem NOT EXISTS.

SELECT * FROM student WHERE NOT EXIST (SELECT 1 FROM zkouska WHERE student.uco = zkouska.uco)

Všimněme si podmínky student.uco = zkouska.uco. Pokud bychom ji vynechali, dostali bychom neprázdný výsledek jen v případě, kdyby tabulka zkouska byla prázdná. Vložená podmínka zajistí, že se bude tabulka zkouska prohledávat pro každé uco studenta zvlášť. Častou chybou bývá opomenutí nebo chybná definice propojovací podmínky, což má za následek zcela chybný výsledek dotazu. V propojovací podmínce spojujeme sloupce z nadřazeného dotazu se sloupci vnořeného dotazu. Platí, že ve vnořeném dotazu se můžeme odkazovat na všechny sloupce dotazu nadřízeného, ale nikoliv naopak, v nadřízeném dotazu nesmí být žádný odkaz na v něm vnořené dotazy.

V SQL vede ke stejnému výsledku často několik cest. Pokud neřešíme rychlost dotazu, záleží na našich preferencích, kterou cestu zvolíme. Například hledání nejstaršího studenta můžeme řešit minimálně třemi způsoby:

```
SELECT * FROM student WHERE datum_narozeni = (
   SELECT MIN (datum_narozeni) FROM student)
```

Tímto způsobem hledáme studenty, jejichž datum narození se rovná nejmenšímu (nejstaršímu) datu v tabulce.

```
SELECT * FROM student WHERE datum_narozeni <= ALL (
   SELECT datum_narozeni FROM student)
```

Tímto způsobem hledáme studenty, jejichž datum narození je menší nebo rovno než všechny datumy v tabulce. Pokud nemá nikdo menší datum narození než já, jsem nejstarší.

```
SELECT * FROM student ridici WHERE NOT EXIST (
   SELECT 1 FROM student vnoreny
   WHERE ridici.datum_narozeni > vnoreny.datum_narozeni)
```

Pro každý řádek řídícího dotazu je prohledávána tabulka vnořeného dotazu (v našem případě stejná tabulka student), zda obsahuje záznam s menším datem narození. Pokud takový řádek neexistuje je splněna podmínka NOT EXISTS a daný řádek je zobrazen.

Jako složitější příklad můžeme uvést požadavek, kdy chceme vidět jména studentů, kteří již absolvovali alespoň tři zkoušky a všechny na první pokus. Toto je opět příklad, kde k výsledku povede více cest, podívejme se na jednu z nich. Potřebujeme nejprve vybrat ty studenty, kteří mají v tabulce zkouska alespoň 3 řádky s různým kódem předmětu. Pokud se spokojíme s uco studenta, vystačíme si s tabulkou zkouska.

```
SELECT uco, COUNT(DISTINCT predmet) FROM zkouska GROUP BY uco
```

Tento dotaz nám vrátí přehled o počtu zkoušek jednotlivých studentů. Klíčové slovo DISTINCT zajistí, že se bude každý předmět počítat jen jednou. Studenti, kteří doposud žádnou zkoušku nesložili, v seznamu nebudou, protože žádný záznam v tabulce nemají. To nám nevadí, protože nás zajímají pouze studenti s alespoň třemi zkouškami. Abychom

vyfiltrovali studenty s jednou a dvěmi zkouškami, doplníme dotaz o podmínku. Podmínku uvedeme nikoliv za WHERE, ale za klíčové slovo HAVING, protože již pracujeme s agregovaným výsledkem (počet zkoušek v primární tabulce není). Doplněný dotaz vypadá takto:

```
SELECT uco, COUNT(DISTINCT predmet) FROM zkouska GROUP BY uco
HAVING COUNT(DISTINCT predmet) >= 3
```

Nyní ověříme, že v našem seznamu není student s neúspěšnou zkouškou (žádný řádek s F):

```
SELECT uco, COUNT(DISTINCT predmet) FROM zkouska
WHERE NOT EXIST (
   SELECT 1 FROM zkouska vnoreny
   WHERE vnoreny.uco = ridici.uco and vnoreny.znamka = 'F')
GROUP BY uco
HAVING COUNT(DISTINCT predmet) >= 3
```

Pokud se ptáte, proč jsme misto vnořeného dotazu nevložili přímo do původního dotazu za WHERE podmínku znamka <> 'F', uvědomte si, že v tomto případě by nám v seznamu zůstal student, který má 3 a více úspěšných zkoušek a libovolný počet neúspěšných. Jednoduchá podmínka by pouze odfiltrovala jeho neúspěšné pokusy ještě před provedením operace GROUP BY. My však chceme studenty bez F, proto je nutné podmínku prověřit v zanořeném dotazu.

Nyní známe uco hledaných studentů a potřebujeme doplnit jméno. Připojíme k výslednému dotazu tabulku student.

```
SELECT jmeno FROM student JOIN
(SELECT uco, COUNT(DISTINCT predmet) FROM zkouska
WHERE NOT EXIST (
   SELECT 1 FROM zkouska vnoreny
   WHERE vnoreny.uco = ridici.uco and vnoreny.znamka = 'F')
GROUP BY uco
HAVING COUNT(DISTINCT predmet) >= 3
) filtr ON student.uco = filtr.uco
```

Připojení jsme provedli tak, že jsme náš připravený dotaz zanořili a umístili místo názvu tabulky na pozici za JOIN. Za uzavírací závorkou jsme si tento dotaz pojmenovali jako "filtr", abychom mohli definovat spojovací podmínku za ON.

Cvičení:

1. Vložte do tabulky patients další záznam s hodnotou NULL místo data narození. Vyzkoušejte chování všech SQL dotazů uvedených v části seskupovaní dat.
2. Najděte nejstaršího studenta čtvrtým způsobem
3. Přepište všechny varianty a najděte nejmladšího studenta
4. Najděte předmět, ze kterého žádný student nemá F.

5 Analytické a statistické funkce SQL

Výstupy z učení

- Umí používat analytické SQL funkce
- Umí používat statistické funkce SQL

V následující kapitole si popíšeme pokročilé funkce, které nabízí databázový systém ORACLE a POSTGRESQL. Tyto specifické funkce nám umožňují řešit elegantně typ úloh, které jsou pomocí standardního SQL často jen velmi obtížně řešitelné. Jde o úlohy, kde je potřeba:

- Pracovat s pořadím řádků
- Odkazovat se na předchozí či následující řádky ve výsledku dotazu
- Pracovat s agregovanými daty

5.1 Funkce pro určení pořadí řádků ve výsledku - Ranking function

Velmi často potřebujeme v praxi stanovit pořadí záznamů ve výsledku. Pro setřídění výsledků nám standard SQL nám nabízí klíčové slovo ORDER BY umísťované na konec SELECT dotazu. Očíslovat výsledek můžeme v prostředí ORACLE pomocí pseudosloupce ROWNUM.

```
SELECT ROWNUM poradi, jmeno, prijmeni FROM student
```

Pseudosloupec ROWNUM přiřazuje číslo výsledným řádkům. Bohužel přiřazení probíhá ještě před finálním setříděním podle výrazu za ORDER BY, a proto následujícím způsobem pořadí studentů dle abecedy **nezískáme**:

```
SELECT ROWNUM poradi, jmeno, prijmeni FROM student
   ORDER BY prijmeni, jmeno
```

Pokud chceme číslovat až setříděný seznam, musíme dotaz zanořit:

```
SELECT ROWNUM poradi, jmeno, prijmeni
   FROM (
          SELECT jmeno, prijmeni FROM student
          ORDER BY prijmeni, jmeno
   )
```

Na další omezení narazíme, pokud potřebujeme vybrat záznamy podle pořadí, například třetího studenta dle abecedy. V takovém případě musíme podmínku vložit až do třetí vrstvy nadřazeného dotazu. **Nelze napsat**:

```
SELECT ROWNUM poradi, jmeno, prijmeni FROM student
WHERE rownum  = 3
ORDER BY prijmeni, jmeno
```

ani

```
SELECT * FROM (
        SELECT ROWNUM poradi, jmeno, prijmeni FROM student
        ORDER BY prijmeni, jmeno
) WHERE poradi = 3
```

Důvodem je, že databáze pseudosloupec ROWNUM nastavuje, až když řádek splní podmínku za WHERE, první řádek je vždy ROWNUM = 1, a protože tento řádek nesplňuje podmínku ROWNUM = 3, na výstup se nedostane. Databáze tak projde všechny záznamy v tabulce, ale žádný podmínku nesplní. Druhý problém je, že ROWNUM se nastavuje ještě před setříděním přes ORDER BY. Databáze vybere tři řádky z tabulky, které terpve následně setřídí. Výsledkem druhého dotazu je třetí řádek, který databáze našla, ale nikoliv nutně třetí dle příjmení studenta. Náhoda často způsobí, že při letmém testování se může zdát, že dotaz funguje, teprve při hlubší kontrole se ukáže, že jde o chybu.

Správně je:

```
SELECT * FROM (
        SELECT ROWNUM poradi, jmeno, prijmeni FROM (
                SELECT jmeno, prijmeni FROM student
                ORDER BY prijmeni, jmeno
        )
)
WHERE poradi = 3
```

Tedy nejprve setřídit, pak očíslovat a teprve ve třetí vrstvě filtrovat. Jak je vidět tato konstrukce je dost komplikovaná a navíc neřeší variantu, kdy máme studenty se stejným příjmením a tyto bychom chtěli označit stejným pořadovým číslem. Proto je pro tento typ úloh výhodnější použít některou z tzv. ranking function. ORACLE i POSTGRESQL nabízí tři funkce:

- RANK()
- DENSE_RANK()
- ROW_NUMBER()

Funkce se liší způsobem, jakým řádky číslují v případě, kdy se objeví ve výsledku stejné, dle pravidel třídění rovnocenné hodnoty. Funkce RANK() a DENSE_RANK() číslují stejné hodnoty stejným pořadovým číslem. Funkce ROW_NUMBER přiděluje každému řádku unikátní číslo, u shodných hodnot rozhoduje o pořadí náhoda. Rozdíl mezi RANK a DENSE_RANK spočívá v tom, jaké pořadové číslo následuje po sérii shodných řádků. Funkce DENSE_RANK pokračuje nepřerušenou číselnou řadou, funkce RANK přeskočí odpovídající počet čísel. Vše osvětlí následující tabulka.

Tabulka 12 -Srovnání výsledků funkcí RANK, DENSE_RANK a ROW_NUMBER

Příjmení	RANK()	DENSE_RANK()	ROW_NUMBER()
Mladý	1	1	1
Novák	2	2	2

Novák	2	2	3
Novák	2	2	4
Starý	5	3	5

Syntaxe všech tří funkcí je následující:

```
RANK () OVER (ORDER BY sloupec)
```

Za názvem funkce následují prázdné závorky, dále klíčové slovo OVER, za kterým následuje definice třídění, podle kterého chceme určovat pořadí řádků. Definice za ORDER BY v ranking funkci nemá žádnou vazbu k ORDER BY klauzuli na konci celého SQL dotazu. Můžeme určovat pořadí zcela nezávisle na finálním setřídění výsledku.

V případě, kdy potřebujeme stanovit pořadí v jednotlivých kategoriích výsledku, lze výraz OVER dále rozšířit o klauzuli PARTITION BY. Například pokud chceme číslovat pořadí vyšetření jednotlivých pacientů podle data vyšetření.

```
SELECT patient_id, datum_vysetreni,
RANK() OVER (PARTITION BY patient_id ORDER BY datum_vysetreni) poradi
FROM vysetreni
```

Tabulka 13 - Číslování výsledku s klauzulí PARTITION BY

Patient_id	datum_vysetreni	poradi
PAT_1	12. 5. 2012	1
PAT_2	14. 5. 2012	1
PAT_2	23. 9. 2012	2
PAT_2	4. 2. 2013	3
PAT_3	15. 3. 2012	1

Ranking funkce lze v dotazu umístit mezi klíčová slova SELECT a FROM nebo jako výraz pro třídění za závěrečné ORDER BY. Naopak nelze je umístit do podmínky za WHERE či HAVING. Pokud chceme pomocí nich definovat podmínku, musíme použít vnořený SQL dotaz.

5.2 Funkce pro pro přístup k předchozím a následným řádkům - LAG (), LEAD ()

Další speciální operací, která je ve standardním SQL obtížné proveditelná, je práce s jiným než aktuálně zpracovávaným řádkem v setříděném seznamu. Připomeňme, že standardní funkce a operátory jako SUBSTR(), LN(), TRUNC() atd. pracují vždy s hodnotami aktuálně zpracovávaného řádku. Co když ale chceme například porovnat číselnou hodnotu jednoho řádku s předchozím řádkem, například sledujeme u pacientů změnu v počtu leukocytů od předchozího vyšetření. Ve standardním SQL bychom museli pomocí JOIN operace spojit tabulku vysetreni se sebou samou, abychom dostali na jeden řádek hodnotu

předchozího a následného vyšetření. Výrazně snadnější a přehlednější je využití speciálních funkcí LAG() nebo LEAD(). Funkce LAG() nám umožňuje pracovat s předchozími záznamy, funkce LEAD() s následnými záznamy. Syntaxe obou funkcí je shodná.

```
LAG (sloupec1, n, hodnota) OVER (ORDER BY sloupec2)
```

Prvním parametrem je výraz, nejčastěji název sloupce, jehož předchozí nebo následující hodnota nás zajímá. Výraz může obsahovat libovolný operátor či standardní funkci. Druhým parametrem je celé číslo, které udává, o kolik řádků se chceme vrátit nebo posunout vpřed. Třetí nepovinným parametrem je hodnota, kterou chceme, aby funkce vrátila, pokud se posune mimo hranice vybrané množiny řádků (tedy před první nebo za poslední řádek). Výchozí hodnotou třetího parametru je NULL. Následuje výraz OVER s definicí setřídění a případně seskupení zpracovávané množiny výsledků. Touto klauzulí určíme funkci LAG() či LEAD(), co míníme předchozím a následným řádkem. Výraz ORDER BY v klauzuli OVER nijak nesouvisí s finálním setříděním výsledku SQL dotazu, i když pro kontrolu správnosti našeho výsledku bude nejčastěji výraz ORDER BY v klazuli OVER stejný jako na konci celého SQL dotazu.
Mějme tabulku vyšetření se sloupci patient_id, datum_vysetreni a pocet_leukocytu. Zajímá nás změna počtu leukocytů u každého pacienta oproti předchozímu vyšetření.

```
SELECT patient_id, datum_vysetreni, pocet_leukocytu,
   LAG(pocet_leukocytu, 1) OVER
   (PARTITION BY patient_id ORDER BY datum_vysetreni) predchozi_pocet,

   pocet_leukocytu -
   LAG(pocet_leukocytu, 1) OVER
   (PARTITION BY patient_id ORDER BY datum_vysetreni)  zmena

   FROM vysetreni
```

Tabulka 14 - Ukázka výsledku funkce LAG()

patient_id	datum_vysetreni	pocet_leukocytu	predchozi_pocet	zmena
PAT_1	12. 5. 2012	7,4		
PAT_2	12. 5. 2012	5,3		
PAT_2	23. 9. 2012	2,4	5,3	-2,9
PAT_2	4. 2. 2013	3,7	2,3	1,4
PAT_3	15. 3. 2012	1,9		
PAT_3	6. 9. 2012	4,5	1,9	2,6

5.3 Reportovací funkce

Databáze ORACLE i POSTGRESQL nabízí nadstavbu standarního SQL, které se označuje jako window nebo jako reportovací (reporting) funkce. Výraz window znamená, že funkce zpracovávají definovanou podmnožinu výsledku dotazu, tzv. okno (window). V podstatě jde o aplikaci agregačních funkcí na vymezený rozsah řádků, který je nezávislý na výrazu v sekci GROUP BY.

Podívejme se na častý případ, kdy potřebuje znát procentické zastoupení zvolené kategorie v tabulce. Mějme tabulku pacientů se sloupcem označující pohlaví. Chceme získat sumární přehled s procentickým zastoupením žen a mužů. Pro získání počtu jednotlivých kategorií použijeme standardní seskupovací výraz GROUP BY. Abychom ale mohli vyjádřit procentické zastoupení, potřebujeme zároveň celkový počet záznamů, což ve standardním SQL můžeme provést pomocí vnořeného dotazu na pozici sloupce:

```
SELECT pohlavi, COUNT(*) pocet,
        (SELECT  COUNT(*)  FROM pacient) celkem,
        COUNT(*) * 100 / (SELECT  COUNT(*)  FROM pacient) procento
 FROM pacient
GROUP BY pohlavi
```

Tabulka 15 - Výsledek seskupení s procentickým vyjádřením

pohlavi	pocet	celkem	procento
F	80	200	40
M	120	200	60

Pomocí reportovací funkce můžeme stejného výsledku dosáhnout bez vnořeného dotazu:

```
SELECT pohlavi, COUNT(*) pocet,
        SUM (COUNT(*))  OVER ()  celkem,
        COUNT(*) * 100 / SUM (COUNT(*))  OVER ()  procento
 FROM pacient
GROUP BY pohlavi
```

Jak je vidět, jde o aplikaci agregační funkce (SUM) na výsledek jiné agregační funkce (COUNT) s vymezením rozsahu agregace. Rozsah agregace je definován za klíčovým slovem OVER, v našem případě agregujeme přes celou množinu, což je vyjádřeno prázdnými závorkami. Můžeme však chtít vytvořit sumární report, kde bude procento mužů a žen rozvedeno dle státní příslušnosti:

Tabulka 16 - Parciální procentické vyjádření

stat	pohlavi	pocet	celkem	procento
ČR	F	50	160	31,25
ČR	M	110	160	68,75
SR	F	30	40	75
SR	M	10	40	25

Tuto sestavu s parciálními součty získáme drobnou úpravou původního dotazu:

```
SELECT stat, pohlavi, COUNT(*) pocet,
        SUM (COUNT(*))  OVER (PARTITION BY stat)  celkem,
        COUNT(*) * 100 / SUM (COUNT(*))  OVER (PARTITION BY stat)
        procento
```

```
FROM pacient
GROUP BY stat, pohlavi
```

Window funkce nejsou vázány jen na agregační konstrukce s GROUP BY. Lze je použít i v jednoduchých výpisech, kde chceme srovnat konkrétní hodnotu například s průměrem. Mějme tabulku s aplikovanou léčbou konkrétního léku jednotlivým pacientům. Tabulka obsahuje identifikaci pacienta, datum podání a množství podaného léku. V tabulárním reportu chceme srovnávat jednotlivé aplikace s celkovým průměrem v celé tabulce. Tuto sestavu získáme z databáze následovně:

```
SELECT patient_id, datum_podani, davka, AVG(davka) OVER () prumerna_davka
FROM lecba
```

Pokud bychom vynechali klauzuli OVER, hlásila by databáze chybu nesprávného použití agregační funkce. V tomto případě je vše v pořádku a ve čtvrtém sloupci bude ve všech řádcích stejná hodnota, která odpovídá průměrné dávce v celé tabulce lecba.

Window funkce využíváme také při výpočtech kumulativních součtů. Při kumulativním součtu sečítáme všechny hodnoty vybraného sloupce od prvního řádku až po aktuální. Například z tabulky lecba z předchozího příkladu chceme sledovat kumulativní spotřebu léku v čase. Použijeme agregační funkci SUM() doplněnou o klauzuli OVER, ve které specifikujeme pravidlo setřídění:

```
SELECT patient_id, datum_podani, davka,
SUM(davka) OVER (ORDER BY datum_podani) kumulativni_spotreba
FROM lecba
```

Tímto zajistíme, že funkce SUM() agreguje data od prvního záznamu až po aktuální řádek. Jde o implicitně definované agregační okno.

Tabulka 17 - Kumulativní součet

patient_id	datum_podani	davka	kumulativni_spotreba
PAT_1	12. 3. 2012	10	10
PAT_2	18. 4. 2012	20	30
PAT_3	19. 4. 2012	10	40
PAT_3	20. 6. 2012	20	60
PAT_4	2. 9. 2012	30	90

Rozsah agregace (agregační okno) lze specifikovat i explicitně, což umožňuje počítat například klouzavý průměr. Klouzavý průměr je průměrná hodnota vypočítaná v časové řadě v definovaném časovém okně. Obvykle počítáme průměrnou hodnotu z několika předchozích hodnot. Toto explicitní okno definujeme za klauzuli ROWS BETWEEN, za níž můžeme použít některou z následujících možností:

- **UNBOUNDED PRECEDING - všechny předchozí řádky**
- **UNBOUNDED FOLLOWING - všechny následující řádky**
- **CURRENT ROW - aktuálně zpracovávaný řádek**
- **n PRECEDING - n předchozích řádků**
- **n FOLLOWING - n následujících řádků**

Když se vrátíme k příkladu s celkovou průměrnou dávkou, můžeme jej rozšířit o výpočet klouzavého průměru z posledních tří předchozích aplikací následovně:

```
SELECT patient_id, datum_podani, davka, AVG(davka) OVER () celkova_prumerna_davka,
      AVG(davka) OVER (ORDER BY datum_podani
        ROWS BETWEEN 3 PRECEDING AND CURRENT ROW) klouzavy_prumer
FROM lecba
```

Tabulka 18 - Klouzavý průměr

patient_id	datum_podani	davka	klouzavy_prumer
PAT_1	12. 3. 2012	10	10
PAT_2	18. 4. 2012	20	15
PAT_3	19. 4. 2012	10	13,333333
PAT_3	20. 6. 2012	20	15
PAT_4	2. 9. 2012	30	20

Hodnota klouzavého průměru se spočítá jako součet 3 předchozích hodnot plus hodnota v počítaném řádku podělený čtyřmi. U prvních tří řádků se počítá průměr z redukovaného počtu dostupných řádků.

5.4 Statistické funkce

Analytické funkce jsou v databázovém systému ORACLE rozšířeny o základní sadu statistických funkcí. Přehled nejvýznamnějších z nich uvádí tabulka.

Tabulka 19 - Vybrané statistické funkce v ORACLE

Funkce	Popis	Uspořádání
CORR	Pearsonův korelační koeficient	párové
CORR_S	Spearmanův korelační koeficient	párové
STATS_BINOMIAL_TEST	Binomický test	binomické
STATS_F_TEST	F - test	nepárové
STATS_KS_TEST	Kolmogorov-Smirnovův test	nepárové
STATS_MW_TEST	Mann Whitney test	nepárové
STATS_ONE_WAY_ANOVA	ANOVA -analýza rozptylu	nepárové
STATS_T_TEST_*	Varianty T testu	párové i nepárové
STATS_WSR_TEST	Wilcoxnův znaménkový test	párové
REGR_INTERCEPT	α koeficient rovnice lineární regrese	párové

REGR_SLOPE	β koeficient rovnice lineární regrese	párové

Součástí následujícího textu není detailní vysvětlení statistických testů a funkcí ani předpoklady pro adekvátní použití těchto funkcí. Jde o standardní testy a výpočty, jejichž podrobné vysvětlení lze nalézt v publikacích věnovaných statistickému zpracování dat. Vysvětlíme si pouze způsob použití těchto funkcí v SQL dotazech. Použití funkce je závislé na uspořádání vlastního testu, zda se jedná o srovnávání hodnot v páru nebo o hodnocení nepárové. Použití funkcí s párovým uspořádáním je přímočaré, funkcím přímo předáváme názvy hodnocených sloupců. Funkce s nepárovým uspořádáním mají jeden parametr pro vlastní data a druhý pro vysvětlovací (kategorizační) proměnnou.

5.4.1 Výpočet korelace

Použití funkcí CORR a CORR_S pro výpočet korelačního koeficientu je snadné. Funkce vyžadují 2 paremetry, kterými jsou nejčastěji 2 sloupce, mezi nimiž chceme spočítat daný korelační koeficient. Jde o agregační funkce, jejichž výsledkem je jeden řádek. Stejně jako standardní agregační funkce je lze rozšířit o window klauzuli OVER (viz předchozí podkapitola o reportovacích funkcích). Mějme tabulku pacientů se sloupci, které obsahují naměřenou výšku a hmotnost jednotlivých pacientů. Korelační koeficienty mezi výškou a hmotností spočítáme následovně:

```
SELECT CORR (vyska, hmotnost) pearson, CORR_S (vyska, hmotnost) spearman
FROM pacient
```

Koeficienty zvlášť pro muže a pro ženy spolu s koeficientem za celý soubor získáme takto:

```
SELECT pohlavi, CORR (vyska, hmotnost) pearson,
                CORR_S (vyska, hmotnost) spearman,

        , CORR (vyska, hmotnost) OVER  () pearson_vse
FROM pacient
GROUP BY pohlavi
```

Pearsonův korelační koeficient (funkce CORR()) je dostupný taktéž v databázi POSTGRESQL verze 9.1.

5.4.2 Párové statistické testy

Mezi testy s párovým uspořádáním patří párový T-test (funkce STATS_T_TEST_PAIRED()) a Wilcoxnův znaménkový test (funkce STATS_WSR_TEST()). Tyto funkce mají 3 parametry, první dva jsou párově uspořádané vstupní parametry, třetím parametrem je specifikace požadovaného výsledku ve formě jedné z následujících textových konstant:

- STATISTIC - výsledek testové funkce
- ONE_SIDED_SIG - jednostranná míra významnosti
- TWO_SIDED_SIG - oboustranná míra významnosti

Mějme tabulku, kde je uveden počet leukocytů před cytotoxickou léčbou a po cytotoxické léčbě. Jde o klasické párové uspořádání, kde můžeme otestovat významnost změny počtu leukocytů po provedené léčbě. Statistickou významnost získáme následovně:

```
SELECT
STATS_T_TEST_PAIRED (leu_pred_lecbou, leu_po_lecbe, 'TWO_SIDED_SIG')
        t_test,
STATS_WSR_TEST (leu_pred_lecbou, leu_po_lecbe, 'TWO_SIDED_SIG')
        wilcoxon
FROM lecba
```

Opět jde o agregační funkce vracející jeden řádek, rozšíření o klazuli OVER však není ve verzi ORACLE 11g podporováno.

5.4.3 Nepárové statistické testy

Nepárové testy obecně testují proti sobě 2 nezávislé výběry, kde nulová hypotéza je stanovena tak, že oba výběry pochází ze stejné populace a že mezi nimi není statisticky významný rozdíl. Pokud test vyjde statisticky významně, zamítáme tuto nulovou hypotézu. Oproti párovému uspořádání nemusí být velikost vzorku pro oba výběry stejná, N se může lišit. Proto vstupní data těchto funkcí mají odlišný formát než funkce párových funkcí. Prvním parametrem je název sloupce, který kategorizuje vlastní data do 2 vzorků (např. hodnoty ze vprvního výběru mouhou být označeny "A", hodnoty z druhého výběru písmenen "B"). Pokud sloupec obsahuje více jak dvě unikátní hodnoty, ohlásí databáze chybu). Druhý parametr je název sloupce s vlastními daty, třetím parametrem je požadovaný výstup, stejně jako v případě párových testů. Tabulka se vstupními daty pro nepárové testy vypadá následovně:

Tabulka 20 - Vstupní data pro nepárové testy

VZOREK_ID	Hodnota
A	12
A	21
A	17
B	20
B	16
B	13
B	18

Porovnání vzorků A a B pomocí funkce Mann Whitney testu a nepárového T-testu provedeme takto:

```
SELECT STATS_MW_TEST (vzorek_id, hodnota, 'TWO_SIDED_SIG') mw,
        STATS_T_TEST_INDEP (vzorek_id, hodnota, 'TWO_SIDED_SIG') t_test
                FROM tabulka
```

Výsledkem je hladina významnosti p.

5.4.4 Jednofaktorová analýza rozptylu (one way ANOVA)

ANOVA je statistická metoda, která umožňuje porovnání více než 2 vzorků. Jednofaktorová ANOVA představuje nejjednodušší případ analýzy rozptylu, kdy analyzujeme

účinek jednoho faktoru na zkoumanou závislou proměnnou. Databáze ORACLE nabízí pro jednofaktorovou ANOVA analýzu funkci STATS_ONE_WAY_ANOVA(). Vstupní data jsou stejná jako v případě nepárových testů, pouze kategorizační proměnná může obsahovat více než dvě unikátní hodnoty. Liší se také nabídka možností pro třetí parametr, který určuje výstupní hodnotu funkce. Můžeme volit z těchto možností:

Tabulka 21 - Možné výstupy funkce STATS_ONE_WAY_ANOVA

Výstupní hodnota	Popis
SUM_SQUARES_BETWEEN	Suma čtverců mezi skupinami
SUM_SQUARES_WITHIN	Suma čtverců uvnitř skupin
DF_BETWEEN	Stupeň volnosti mezi skupinami
DF_WITHIN	Stupeň volnosti uvnitř skupin
MEAN_SQUARES_BETWEEN	Mean squares mezi skupinami
MEAN_SQUARES_WITHIN	Mean squares uvnitř skupin
F_RATIO	Poměr MSB/MSW
SIG	Míra významnosti

Mějme tabulku pacientů se sloupcem, který určuje stádium onemocnění v době diagnózy a sloupec s celkovým přežitím v měsících od diagnózy. Vliv stádia na přežití pomocí jednofaktorové ANOVA analýzy posoudíme takto:

```
SELECT STATS_ONE_WAY_ANOVA(stadium, preziti, 'F_RATIO') f_ratio,
       STATS_ONE_WAY_ANOVA(stadium, preziti, 'SIG') p_value
FROM patients
```

Opět jde o agregační funkci, výsledkem je tedy jeden řádek s hodnotou testu a statistickou významností.

5.4.5 Binomický test

Pomocí binomického testu můžeme otestovat, zda procentický výskyt zkoumaného jevu odpovídá očekávané frekvenci. Můžeme tak například otestovat, zda procento mužů v naší tabulce pacientů odpovídá očekávaným 50 procentům všech pacientů:

```
SELECT STATS_BINOMIAL_TEST (sex, 'M', 0.5, 'EXACT_PROB') exaktni,
       STATS_BINOMIAL_TEST (sex, 'M', 0.5, 'TWO_SIDED_PROB' ) oboustranna
FROM patients
```

Prvním parametrem je sloupec s kategoriální proměnou, druhý určuje testovanou kategorii, třetí parametrem je očekávaný podíl výskytu dané kategorie, čtvrtým parametrem specifikujeme požadovaný výstup. Kategoriální proměnná musí obsahovat právě dvě unikátní hodnoty. Varianty pro čtvrtý parametr jsou následující:

Tabulka 22 - Možné výstupy funkce STATS_BINOMIAL_TEST

Výstupní hodnota	Popis
TWO_SIDED_PROB	Hodnota oboustranné pravděpodobnosti

EXACT_PROB	Hodnota exaktní pravděpodobnosti
ONE_SIDED_PROB_OR_MORE	Hodnota jednostranné pravděpodobnosti (větší než)
ONE_SIDED_PROB_OR_LESS	Hodnota jednostranné pravděpodobnosti (menší než)

5.4.6 Lineární regrese

Databázový systém ORACLE i POSTGRESQL nám umožňuje snadno provést nad daty lineární regresi a vypočítat alfa a beta koeficienty regresní rovnice. Slouží k tomu funkce REGR_INTERCEPT() a REGR_SLOPE(), které očekávají na vstupu dva parametry, prvním je název sloupce se závislou spojitou proměnou, druhým je sloupec s nezávislou proměnnou.

6 Vyhledávání v textu

Výstupy

- Umí použít operátor LIKE pro jednoduché prohledávání textu
- Rozumí termínu regulární výraz a umí jednodušší výraz sestavit
- Umí sestavit SQL dotaz s regulárním výrazem
- Dokáže pomocí SQL a regulárního výrazu extrahovat potřebnou informaci z textové informace

V této kapitole se seznámíme s databázovými prostředky, které nám umožňují prohledávat textové řetězce, tedy hodnoty uložené ve sloupcích s obecným datovým typem CHAR, VARCHAR, VARCHAR2, případně CLOB. Představen bude jednak operátor LIKE, jednak tzv. regulární výrazy.

6.1 Standardní funkce INSTR a operátor LIKE

Funkci INSTR v databázi ORACLE, respektive STRPOS v případě POSTGRESQL, jsme si představili již ve třetí kapitole. Pro připomenutí tato funkce umožňuje prohledávat text na výskyt specifikovaného podřetězce, kterým může být jeden až N znaků. Výsledkem funkce je pozice nalezeného podřetězce nebo nula v případě, kdy podřetězec nebyl nalezen. Funkce má dva povinné parametry, prohledávaný text a hledaný podřetězec. Ve funkci INSTR v případě potřeby můžeme využít další dva parametry, kterými jsou startovací pozice vyhledávání a pořadí výskytu:

```
SELECT INSTR('strč prst skrz krk', 'r', 5, 2) FROM DUAL
```

Tento příklad hledá druhý výskyt písmena 'r' od páté pozice řetězce 'strč prst skrz krk'. Výsledkem je 13, což je pozice druhého písmena 'r' hledaného od pátého znaku, tedy 'r' ve slově skrz. Výchozí pozice může být i záporné číslo, v tom případě se pozice počítá od konce řetězce a vyhledávání probíhá od konce na začátek:

```
SELECT INSTR('strč prst skrz krk', 'r', -5, 2) FROM DUAL
```

Příkaz provede hledání od pátého znaku od konce, tedy od písmene 'z', směrem k začátku řetězce a výsledkem je pozice 7, tedy písmeno 'r' ve slově prst.

Funkci INSTR můžeme použít i v definici podmínky WHERE:

```
SELECT * FROM tabulka WHERE INSTR(sloupec, 'r', 1, 2)  > 0
```

Tento příklad vrátí řádky tabulky, které ve sloupci obsahují alespoň dvě písmena "r" (hledáme pozici druhého písmena 'r').

Obvyklejším způsobem vyhledávání řádků, které obsahují v textovém sloupci určitý podřetězec, je použití operátoru LIKE a zástupných znaků. Jako zástupný znak se v případě ORACLE databáze používá znak "_" (podtržítko) a znak "%" (procento). Podtržítko nahrazuje právě jeden libovolný znak, procento 0 až N libovolných znaků. Syntaxe operátoru LIKE je následující:

```
sloupec LIKE '%podřetězec%'
```

Operátor LIKE používáme při definici vyhledávacích podmínek za klíčovým slovem WHERE:

SELECT * FROM tabulka WHERE sloupec LIKE '_rk'

Tento příkaz najde všechny řádky, které obsahují ve sloupci třípísmenné slovo končící na "rk".

V případě, kdy potřebujeme vyhledávat v textu samotný zástupný znak, definujeme pro operátor LIKE ještě tzv. ESCAPE znak, který když umístíme před zástupný znak, vrátí zástupnému znaku jeho původní význam. Pokud tedy potřebujeme najít řádky, které obsahují ve sloupci symbol procento, definujeme podmínku následovně:

SELECT * FROM tabulka WHERE sloupec LIKE '%\%%' ESCAPE '\'

Jako ESCAPE znak zde slouží zpětné lomítko, které zbavuje druhý znak procenta jeho funkce zástupného znaku. První a třetí znak procento jsou interpretovány jako zástupné znaky, procento se tedy může ve sloupci vyskytovat kdekoliv (může ho předcházet i následovat libovolné množství jiných znaků).

Pomocí operátoru LIKE bychom se mohli pokusit hledat řetězce obsahující datum následujícím zpsůsobem:

SELECT * FROM tabulka WHERE sloupec LIKE '%__.__.____%'.

Hledáme takto dva znaky, tečku, dva znaky, tečku a čtyři znaky umístěné libovolně v textu. Tento způsob nám však může vrátit mnoho falešně pozitivních výsledků (např. IP adresa 88.45.12.45 bude taká vyhovovat uvedenému vzoru) a naopak mnohé řádky přehlédne (např. 1.2.2000 má pouze jednu číslici před první i druhou tečkou). Pro dosažení lepších výsledků musíme použít tzv. regulární výrazy.

6.2 Regulární výrazy

Regulární výrazy je pokročilá technika prohledávání textu, se kterou se setkáme ve většině programovacích jazyků, v pokročilých textových editorech a taktéž v databázových systémech. Regulárním výrazům jsou věnované samostané publikace, v této kapitole se seznámíme pouze se základními konstrukcemi a se způsobem použití v databázi ORACLE a POSTGRESQL.

6.2.1 Základy regulárních výrazů

Na regulární výraz se můžeme dívat jako na rozšíření operátoru LIKE. Jde o textovou šablonu, která se skládá z:

- hledaných znaků
- zástupných znaků
- kvantifikátorů
- operátorů
- modifikátorů

Zatímco operátor LIKE má pouze 2 zástupné znaky, u regulárních výrazů je nabídka širší. Zástupným znakem můžeme odlišit např. číslici od písmena nebo tzv. bílého znaku (mezera, tabulátor). Přehled základních zástupných znaků uvádí Tabulka 23.

Tabulka 23 - Zástupné znaky v regulárních výrazech

Zástupný znak	Význam
. (tečka)	Jakýkoliv znak

^	Začátek řetězce
$	Konec řetězce
\d	Číslice
\D	Vše kromě číslice
\w	Písmeno, číslice, podtržítko
\W	Doplněk k \w
\s	Bílý znak – mezera, tabulátor
\S	Doplněk k \s

Pokud potřebujeme v textu hledat samotný zástupný znak v původním významu, tedy například tečku, musíme před hledaný znak umístit zpětné lomítko. Toto pravidlo platí pro všechny speciální znaky regulárních výrazů.

```
'^\^\..$'
```

Uvedeným výrazem hledáme tříznakové řetězce, které začínají "^." a libovolným následujícím znakem.

Zástupný znak zastupuje vždy právě jeden znak v prohledávaném řetězci. Toto chování můžeme změnit pomocí tzv. kvantifikátorů, které v regulárním výrazu umístíme těsně za daný zástupný znak. Přehled kvantifikátorů je uveden v Tabulka 24.

Tabulka 24 - Kvantifikátory v regulárních výrazech

Kvantifikátor	**Význam**
*	0 – n opakování ("greedy" chování)
*?	0 – n opakování ("nongreedy" chování)
+	1 – n opakování ("greedy" chování)
+?	1 – n opakování ("nongreedy" chování)
?	0 nebo 1 opakování
{m}	Přesně m opakování
{m,}	m nebo více opakování
{m,n}	Minimálně m, maximálně n opakování

Spojením zástupného znaku "." a kvantifikátoru "*" dostáváme regulární výraz, který pokrývá libovolný textový řetezec. Tuto kombinaci používáme ve dvou variantách:

- "hladová" (greedy)
- "nehladová" (nongreedy)

Pokud použijeme hladovou variantu, bude se hledat shoda s co nejdelším řetězcem, naopak nehladová varianta hledá shodu s co nejkratším řetězcem. Blíže se na tento problém podíváme v další části kapitoly věnované nahrazování podřetězců s použitím regulárních výrazů.

Pokud bychom hledali v textu datum, mohli bychom použít tento regulární výraz:

```
'\d{1,2}\.\d{1,2}\.\d{2,4}'
```

Hledáme jednu až dvě číslice jako den, následuje tečka, jedna až dvě číslice na pozici měsíce, tečka a dvě až čtyři číslice na pozici roku. Pokud by komponenty datumu oddělovaly kromě tečky i mezery, rozšířili bychom výraz o mezeru s otazníkem za každou tečku:

```
'\d{1,2}\. ?\d{1,2}\. ?\d{2,4}'
```

Ani tento výraz však není stále ideální, protože den i měsíc může ve skutečnosti na první pozici obsahovat jen vybrané číslice, konkrétně den 0, 1, 2 nebo 3, měsíc pouze 0 nebo 1. Tento problém nám pomohou řešit tzv. operátory regulárních výrazů. Jejich přehled je uveden v Tabulka 25.

Tabulka 25 - Operátory regulárních výrazů

Operátor	Význam
|	nebo
[abc]	Jeden z uvedených znaků (a nebo b nebo c)
[^abc]	Libovolný znak kromě uvedených (vše kromě a b c)
(abc)	Uzavření skupiny znaků - blok
\1, \2, \3, ...	Odkaz na první, druhý, třetí blok

Pro specifikaci vybraných znaků můžeme využít buď operátor svislítko "|" nebo operátor hranatých zvorek. Upravený výraz pro hledání datumu může vypadat takto:

```
'(0|1|2|3)?\d\.(0|1)?\d\.\d{2,4}'
```

nebo takto:

```
'[0123]?\d\.[01]?\d\.\d{2,4}'
```

V tomto případě jsou obě varianty rovnocenné, rozdíl by byl, pokud bychom kombinovali operátor se zástupným znakem. Zatímco operátor svislítko zástupné znaky interpretuje, operátor hranatých závorek nikoliv. Výraz hledající v textu číslici nebo bílý znak proto musí vypadat takto:

```
(\d|\s)
```

Oproti tomu výraz "[\d\s]" bude hledat v textu znaky "\" nebo "d" nebo "s".

Pomocí kulatých závorek můžeme vybranou skupinu znaků uzavřít do bloku a na ten se pak následně odkazovat pomocí zpětného lomítka a čísla pořadí bloku. Tuto techniku použijeme, pokud hledáme například repetitivní vzor a chceme ho definovat co nejobecněji. Například pokud hledáme opakování tří stejných číslic, můžeme napsat:

```
'(\d)\1\1'
```

Říkáme tím, že chceme najít číslici (\d), za kterou se má opakovat stejný znak (\1). Výraz je podstatně kratší než rovnocenný výraz s použitím operátoru svislítka a výpisem všech variant:

```
'(111|222|333|444|555|666|777|888|999|000)'
```

Pomocí odkazů můžeme hledat text, který začíná dvěmi nečíselnými znaky a končí stejnými znaky v opačném pořadí:

```
'^(\D)(\D).*\2\1$'
```

Poslední komponentou regulárních výrazů jsou **modifikátory**, které mění chování celého procesu vyhledávání. Základním modifikátorem je volba, zda chceme při vyhledávání rozlišovat velikost písmen. Pokud ano, jde o "case sensitive" hledání, pro které se používá znak "c", pokud ne, jde o "case insensitive" označované znakem "i".

6.2.2 Použití regulárních výrazů v databázi ORACLE

Podpora regulárních výrazů v databázi ORACLE zahrnuje funkce vyhledávání řetězců, hledání a extrakci podřetězce i nahrazování podřetězce za jiný text.

Funkce pro vyhledávání řádků v tabulce, které ve sloupci obsahují text odpovídající specifikovanému regulárnímu výrazu, se v databázi ORACLE nazývá REGEXP_LIKE (). Její použití je následující:

SELECT * FROM tabulka WHERE REGEXP_LIKE(sloupec,'reg. vyraz', modifikator)

Hledáme-li v tabulce řádky obsahující ve zvoleném textovém slouci datum, použijeme tento příkaz:

SELECT * FROM tabulka WHERE REGEXP_LIKE (sloupec, '[0123]?\d\.[01]?\d\.\d{2,4}', 'c');

Výsledek funkce REGEXP_LIKE je hodnota true v případě, že daný řádek tabulky obsahuje ve sloupci text odpovídající regulárnímu výrazu.

Regulární výraz můžeme použít i pro vyhledání či extrakci podřetězce z textu. Slouží k tomu funkce, které jsou obdobou textových funkcí SUBSTR() a INSTR(), pouze místo hledaného pevného řetězce používáme regulární výraz. Funkce pro extrakci podřetězce je definována takto:

REGEXP_SUBSTR(text, 'reg. vyraz', hledat_od, vyskyt, modifikator)

Pomocí této funkce extrahujeme ze sloupce podřetězec specifikovaný regulárním výrazem, hledání probíhá od specifikované pozice (třetí parametr), hledá se n-tý výskyt (čtvrtý parametr) při zohlednění modifikátorů specifikovaných posledním parametrem. Povinné parametry jsou první dva. Výsledem funkce je extrahovaný podřetězec nebo NULL.Zatímco funkci REGEXP_LIKE můžeme použít pouze při definování vyhledávací podmínky, ostatní REGEXP funkce můžeme použít v SQL dotazech na všech místech jako standardní funkce.

Extrakci datumu z textu provedeme tak, že stejný regulární výraz použijeme ve funkci REGEXP_LIKE() i REGEXP_SUBSTR():

SELECT REGEXP_SUBSTR(sloupec, '[0123]?\d\.[01]?\d\.\d{2,4}') datum FROM tabulka

WHERE REGEXP_LIKE(sloupec, '[0123]?\d\.[01]?\d\.\d{2,4}')

Pokud chceme získat pouze pozici podřetězce místo samotného podřetězce, použijeme místo funkce REGEXP_SUSBSTR() funkci REGEXP_INSTR(). Její parametry jsou shodné, pouze návratová hodnota je pozice prvního znaku podřetězce nebo nula.

U extrakce podřetězců se vrátíme k pojmům "greedy" a "nongreedy", které jsme zmínili u přehledu kvantifikátorů. Mějme situaci, kdy chceme z textu extrahovat text, který je uveden v závorkách. Text v závorkách může obsahovat libovolné znaky, například výsledek cytogenetického vyšetření.

SELECT REGEXP_SUBSTR('translokace t(9;22)', '\(.*\)') FROM DUAL

Pokud text obsahuje pouze jeden pár závorek, funkce správně vrátí podřetězec "(9;22)". Pokud ale prohledávaný text obsahuje více závorek, projeví se "hladovost" kvantifikátoru "*":

SELECT REGEXP_SUBSTR('translokace t(9;22) (Ph-chromozom) ', '\(.*\)') FROM DUAL

Výsledkem je "(9;22) (Ph-chromozom)", protože výraz ".*" byl roztažen na maximální počet znaků uzavřených mezi první otevírací závorkou a druhou uzavírací závorkou. Pokud chceme získat jen obsah první závorky, musíme použít "nehladový" kvantifikátor "*?":

SELECT REGEXP_SUBSTR('translokace t(9;22) (Ph-chromozom) ', '\(.*?\)') FROM DUAL

Pokud chceme hledaný podřetězec nahradit jiným textem, použijeme funkci REGEXP_REPLACE(). Její syntaxe je následující:

REGEXP_REPLACE(text, reg.výraz, novy_text, hledat_od, vyskyt, modifikator)

Oproti REGEXP_SUBSTR je tu rozdíl v třetím parametru, kterým je text nahrazující nalezený vzor. Parametr "vyskyt" specifukuje, kolikátý nález se má nahradit, pokud uvedeme nulu (výchozí hodnota), nahradí se všechny výskyty. Nahrazovaný text může obsahovat odkazy na bloky specifikované ve vyhledávaném regulárním výrazu. To nám umožní například převést český formát datumu na formát (rok-měsíc-den):

SELECT REGEXP_REPLACE(sloupec, '([0123]?\d)\.([01]?\d)\.(\d{4})', '\3-\2-\1') datum FROM tabulka

WHERE REGEXP_LIKE(sloupec, '[0123]?\d\.[01]?\d\.\d{2,4}')

Poslední významnou funkcí z REGEXP rodiny je REGEXP_COUNT(), která vrací počet nalezených výrazů v prohledávaném textu. Její syntaxe je:

REGEXP_COUNT(text, reg.výraz , hledat_od, modifikator)

Význam parametrů je stejný jako v případě funkce REGEXP_SUBSTR().

6.2.3 Použití regulárních výrazů v databázi POSTGRESQL

V databázi POSTGRESQL najdeme místo funkce REGEXP_LIKE() operátor "~" (vlnka), který provádí porovnání řetězce s regulárním výrazem s ohledem na velikost písmen (case sensitive), zatímco operátor "~*" porovnává shodu bez ohledu na velikost písmen (case insesitive).

SELECT * FROM tabulka WHERE sloupec ~ '[0123]?\d\.[01]?\d\.\d{2,4}'

respektive

SELECT * FROM tabulka WHERE sloupec ~* '[0123]?\d\.[01]?\d\.\d{2,4}'

Funkce REGEXP_SUBSTR() je v POSTGRESQL zastoupena funkcí SUBSTRING(), jejíž syntaxe je následující:

```
SUBSTRING(text, reg.vyraz)
```

Extrakci datumu z textového sloupce bychom tedy v databázi POSGRESQL provedli takto:

SELECT SUBSTRING(sloupec, '[0123]?\d\.[01]?\d\.\d{2,4}') datum FROM tabulka

WHERE sloupec ~ '[0123]?\d\.[01]?\d\.\d{2,4}'

Oproti funkci REGEXP_SUBSTR() je tu užitečná výhoda, že můžeme ze specifikovaného regulárního výrazu extrahovat pouze omezenou část, kterou uzavřeme do kulatých závorek. Zatímco tedy předchozí příklad vrátí celé datum, v následujícím příkladu můžeme drobným doplněním regulárního výrazu získat pouze rok z nalezeného datumu:

SELECT SUBSTRING(sloupec, '[0123]?\d\.[01]?\d\.(\d{2,4})') datum FROM tabulka

WHERE sloupec ~ '[0123]?\d\.[01]?\d\.\d{2,4}'

Shodný název jak v databázi ORACLE tak POSTGRESQL mají funkce pro nahrazení nalezeného podřetězce za jiný text. Jde o funkci REGEXP_REPLACE(), kde rozdíl je pouze ve volitelných parametrech. Syntaxe v POSTGRESQL je:

REGEXP_REPLACE (`text`, `reg.vyraz`, `novy_text` [, `priznaky`])

V prohledávaném textu je nahrazen nalezený vzor za nový text. Pomocí příznaků ovlivňujeme chování funkce. Nejdůležitější příznaky shrnuje Tabulka 26:

Tabulka 26 - Přehled nejvýznamnějších příznaků funkce REGEXP_REPLACE v POSTGRESQL

Příznak	Význam
g	Nahradit všechny výskyty regulárního výrazu
i	Porovnávání bez ohledu na velikost písmen
c	Porovnávání s ohledem na velikost písmen

Pokud chceme v textu zamaskovat čísla hvězdičkou, můžeme použít toto řešení:

SELECT REGEXP_REPLACE ('Rodné číslo 770922/1234', '\d', '*', 'g') FROM GENERATE_SERIES(1,1)

Příznak "g" zajistí, že budou zaměněny všechny nalezené číslice.

6.2.4 Shrnutí

Regulární výrazy jsou velmi mocným nástrojem při operacích s textem. K nevýhodám této techniky patří těžká čitelnost výsledných výrazů a časté hlavu lámající chování složitějších konstrukcí.

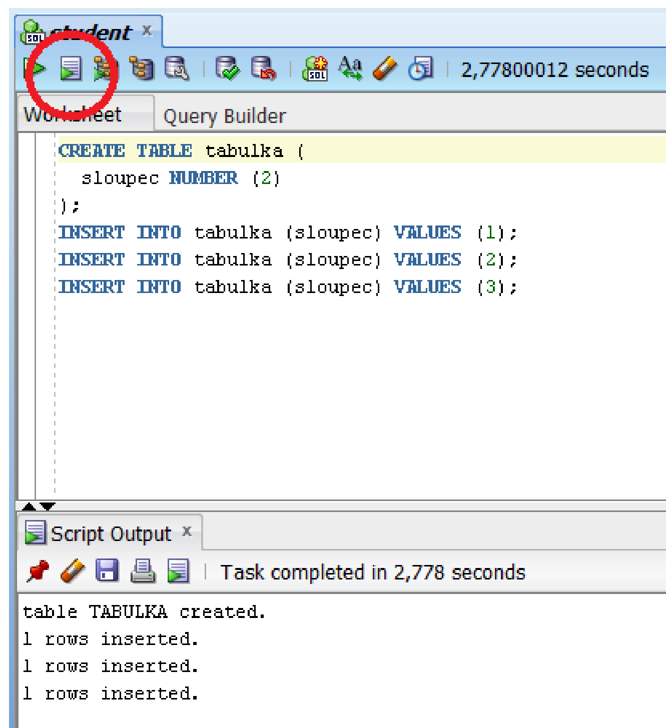
V této kapitole byla popsána základní syntaxe regulárních výrazů a práce s nimi v prostředí ORACLE a POSTGRESQL. Nejde však o vyčerpávající popis, pro další možnosti této techniky je třeba prostudovat dokumentaci k vybranému databázovému systému.

7 SQL skripty, uživatelské procedury a funkce

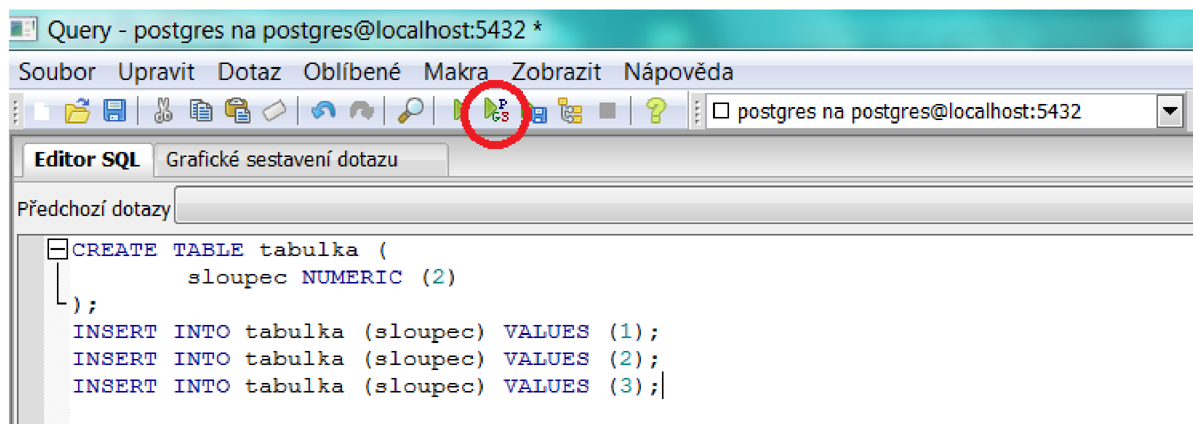
Výstupy

- Chápe, co jsou SQL skripty, SQL procedury a funkce a rozdíl mezi nimi
- Dokáže sestavit a spustit SQL skript
- Rozumí základním prvkům jazyka PL/SQL
- Umí vytvořit jednoduchou uživatelskou proceduru a funkci v jazyce PL/SQL

Jednotlivé SQL příkazy je velmi často potřeba spouštět v menších či větších sériích. Typickým případem je vytváření databázové struktury či manipulace s daty pomocí DML příkazů INSERT, UPDATE, DELETE. Nejjednodušším způsobem seskupení SQL příkazů je vytvoření SQL skriptu. Nejedná se o nic jiného než o seřazení potřebných SQL příkazů za sebou do textového souboru. Jednotlivé příkazy se standardně oddělují středníkem (;). Můžeme kombinovat DDL příkazy (CREATE, ALTER, DROP) s DML příkazy. Na konec skriptu umísťujeme potvrzovací příkaz COMMIT. Spuštění SQL skriptu lze provést různými způsoby. Nejsnazší je spuštění v grafickém klientu, kam do příkazového okna nakopírujeme příslušný skript a stiskneme tlačítko pro spuštění skriptu. Ukázku spuštění jednoduchého skriptu lze vidět na obrázku Obrázek 5 (ORACLE) a Obrázek 6 (POSTGRESQL).



Obrázek 5 - SQL skript v SQLdeveloperu (ORACLE)



Obrázek 6 - SQL skript v pgAdmin (POSTGRESQL)

SQL skript lze spustit také z řádkového klienta. V ORACLE klientovi **sqlplus** spustíme připravený skript, který jsme uložili do souboru script.sql, tím, že napíšeme cestu a název souboru se skriptem a umístíme před něj znak zavináč (@).

SQL > @/oracle/scripts/script.sql

Pokud chceme spustit skript přímo z příkazové řádky operačního systému, spustíme sqlplus s parametrem:

SQLPLUS login/heslo @/oracle/scripts/script.sql

V případě databáze POSTGRESQL je nejen ke spouštění skriptů k dispozici řádkový klient **psql**. Spuštění SQL skriptu z prostředí psql provedeme příkazem \i:

postgres=# \i script.sql

Pokud chceme spustit SQL skript přímo z operačního systému, zavoláme psql následovně:

PSQL -U login -f script.sql

7.1 Uložené procedury a uživatelské funkce

Sekvence DML příkazů můžeme také uložit jako objekt databáze formou uložené procedury nebo uživatelské funkce. Stejně jako tabulky jsou uložené procedury a funkce součástí databáze, mají svého vlastníka a přístup k nim se řídí přiděleným oprávněním. Stejně jako tabulky se vytvářejí příkazem CREATE a ruší příkazem DROP. Rozdíl mezi uloženou procedurou a uživatelskou funkcí není velký. Databáze POSTGRESQL má pouze uživatelské funkce, které ale dokáží totéž co uložené procedury v ORACLE. V databázi ORACLE jedním z rozdílů mezi procedurou a funkcí spočívá ve způsobu spouštění. Zatímco uživatelskou funkci voláme stejně jako standardní funkci (např. SUBSTR() nebo ROUND()) v těle SQL dotazu, uloženou proceduru spouštíme jako procedurální kód, což si ukážeme v této kapitole.

Databázové systémy často kromě SQL jazyka podporují i specifický procedurální jazyk. Pomocí něj lze provádět operace, které v SQL provést nelze nebo které jsou příliš komplikované. Procedurálním jazykem na rozdíl od SQL zpracováváme řádek po řádku pomocí programových smyček a podmíněných výrazů. Procedurální kód nám také umožňuje provádět vstupně/výstupní operace jako je ukládání dat do souboru na disk, odesílání emailem apod. V uložených procedurách a uživatelských funkcích kombinujeme procedurální jazyk s SQL příkazy. Procedurální jazyky jsou ještě více produktově specifické než jazyk SQL. Procedurální jazyk v ORACLE se označuje jako PL/SQL, jazyk v PostgreSQL jako PGSQL. Základní koncept je v obou případech stejný, odlišnosti jsou však v syntaxi, jak si ukážeme dále.

7.2 Základy databázového procedurálního jazyka

Databázový procedurální jazyk vychází ze stejného konceptu a používá stejné prvky jako jakýkoliv standardní procedurální programovací jazyk. V následujícím textu se seznámíme s těmito jeho základními prvky:

- Oddělovače bloku kódu a oddělovač příkazů
- Práce s proměnnými
- Podmíněný výraz
- Programová smyčka
- Volání jiných procedur či funkcí
- Zpracování výjimek

Procedurální kód PL/SQL je vždy ohraničen klíčovými slovy BEGIN na začátku a END na konci. Pro oddělení jednotlivých příkazů se používá středník (;) včetně finálního slova END. Za klíčovým slovem BEGIN se naopak středník nevkládá. BEGIN a END vymezují tzv. blok kódu, který může obsahovat další zanořený blok opět ohraničený slovy BEGIN a END. Stejně jako v SQL funguje symbol "--" k oddělení jednořádkových komentářů, víceřádkové komentáře uzavíráme mezi "/*" a "*/".

```
BEGIN
    --toto je jednořádkový komentář
```

```
/* toto je
   víceřádkový komentář */
END;
```

V procedurálním kódu se využívají tzv. proměnné pro uchování a přenos zpracovávaných hodnot. Proměnná je obdobou sloupce tabulky, má své jméno a datový typ, základní typy proměnných však mohou přenášet v danou chvíli jen jednu hodnotu. Datové typy jsou stejné jako v případě SQL. Proměnné musíme deklarovat ještě před úvodním slovem BEGIN, kde uvedeme název proměnné a její datový typ oddělený mezerou, jednotlivé proměnné oddělujeme středníkem:

```
i NUMBER (5);
str VARCHAR2 (100);
BEGIN
....
END;
```

Proměnné přiřadíme hodnotu pomocí operátoru přiřazení, kterým je ":=". Pomocí standardních operátorů můžeme provádět základní aritmetické operace. Proměnné můžeme využívat na místě konstant v SQL příkazech:

```
i NUMBER (5);
str VARCHAR2 (100);
BEGIN
   i:=1; str := 'text';
   i:= (i-14) * 9875  + 456;
   str := str || ' pripojeny text';
   DELETE FROM tab WHERE sloupec = i AND sloupec2 = str;
   INSERT INTO tab2 (sloupec, sloupec2) VALUES (i, str);
END;
```

Jednou ze základních technik v procedurálním programování je větvení kódu dle definované podmínky pomocí konstrukce IF-THEN-ELSE. V prostředí PL/SQL má podmínková konstrukce následující tvar:

```
IF podminka THEN
   prikaz1;
ELSE
   prikaz2;
END IF;
```

Pokud je podmínka splněna provede se příkaz1 (obecně sada příkazů mezi THEN a ELSE), pokud podmínka platná není, provede se příkaz 2 (sada příkazů mezi ELSE a END).

Část ELSE je nepovinná, naopak při potřebě vyhodnotit postupně více podmínek, můžeme konstrukci rozšířit o ELSEIF (deklarační část proměnných leu a grade je vynechána):

```
IF leu  >= 3000 THEN
    grade  := 'I';
ELSIF leu < 3000 AND leu >= 2000 THEN
    grade := 'II';
  ELSIF leu < 2000 and leu >= 1000 THEN
    grade := 'III';
  ELSE
    grade := 'IV';
  END IF;
```

V konstrukci IF/ELSIF/ELSE se provede kód vždy jen jednoho ramena. Pokud je při průchodu splněna podmínka více než jednoho ramena, provede se pouze první se splněnou podmínkou. Pokud není splněna žádná podmínka, provede se část ELSE. Konstrukci IF/ELSEIF můžeme nahradit za podmíněný výraz CASE, který známe z SQL.

Dalším prvkem procedurálního programování jsou programové smyčky, které nám umožní provádět určitou část kódu opakovaně. Smyček je v PL/SQL několik typů, my si ukážeme jednu z nejvíce využívaných, která umožňuje procházet výsledek SQL dotazu řádek po řádku. Její syntaxe je následující:

```
FOR vektor IN (SELECT_dotaz) LOOP
   prikaz;
END LOOP;
```

Tato smyčka využívá speciální proměnnou (vektor), která se při každém průchodu naplní hodnotami jednoho řádku z výsledné množiny specifikovaného SELECT dotazu. Na jednotlivé hodnoty (sloupce) se odkazujeme jako na sloupce tabulky, pouze místo názvu tabulky používáme název proměnné (vektoru):

```
FOR k IN (SELECT patient_id, sex, date_of_birth FROM patients) LOOP
   IF k.date_of_birth > SYSDATE THEN
          INSERT INTO tabulka_chyb (patient_id, popis_chyby)
          VALUES (k.patient_id, 'Chybné datum narození');
   END IF;
   IF k.sex <> 'M' AND k.sex <> 'F' THEN
          INSERT INTO tabulka_chyb (patient_id, popis_chyby)
          VALUES (k.patient_id, 'Chybné pohlaví');
   END IF;
END LOOP;
```

Uvedená smyčka projde postupně všechny řádky tabulky patients a provádí příkazy mezi klíčovými slovy LOOP a END LOOP, v našem případě zkontroluje správnost vyplnění pohlaví a datumu narození. Zjištěné chyby zapisuje do tabulky tabulka_chyb. Pokud by

tabulka patients byla prázdná, a tedy zdaný SQL dotaz by nevrátil žádný řádek, celý FOR blok by se při běhu programu přeskočil. Vektor v tomto typu smyčky na rozdíl od běžných proměnných nemusíme dopředu deklarovat, za ukončením END LOOP se však již na něj nelze odkazovat. Smyčku lze předčasně ukončit použitím příkazu EXIT, který umístíme do vhodné podmínky.

PL/SQL podporuje i další typy smyček jako je nekonečná smyčka LOOP .. END LOOP, smyčka pro dopředu neurčený počet průchodů WHILE-LOOP či smyčka FOR-LOOP pro předem určený počet průchodů. Detaily lze najít v dokumentaci k databázi ORACLE.

Uvnitř PL/SQL procedury můžeme volat jiné uložené procedury. Buď naše vlastní nebo některou z procedur z bohaté knihovny databázové systému. Proceduru spustíme prostým uvedením jejího jména a případnou specifikací parametrů, které uzavřeme do kulatých závorek. Procedury a funkce mohou být seskupeny do tzv. balíku (package), v tom případě při volání procedury vkládáme před jejich název i název balíku oddělený tečkou. V následujícím příkladě spustíme proceduru PUT_LINE, která je součástí balíku DBMS_OUTPUT. Spustíme ji s textovým parametrem:

BEGIN

DBMS_OUTPUT.PUT_LINE('Všechno špatně');

END;

Balík procedur DBMS_OUTPUT je součástí databázového systému ORACLE. Jeho procedura PUT_LINE vypisuje na obrazovku obsah předaného parametru a slouží nejčastěji k výpisu ladících zpráv uložených procedur a funkcí.

Během provádění procedurálního kódu může dojít k chybě, například když se pokusíme dělit nulou nebo přiřadit do proměnné hodnotu, která neodpovídá jejímu datovému typu. Těmto chybám lze částečně předcházet pomocí podmínkových konstrukcí, ale existuje i jiná varianta řešení chybových stavů. Jde o tzv. zachytávání výjimek a obsloužení chybového stavu. Pokud dojde při běhu PL/SQL k chybě, dojde k přerušení vykonávání kódu na chybovém řádku a generuje se výjimka (exception). Pokud je na konci bloku kódu, kde chyba nastala, sekce pro zpracování výjimek, přesune se vykonávání kódu sem. Každá vzniklá výjimka s sebou nese identifikaci chyby, která ji způsobila. V sekci výjimek pak lze reagovat na jednotlivé druhy chyb:

```
BEGIN
...
...
EXCEPTION
   WHEN ZERO_DIVIDE  THEN
      -- osetreni deleni nulou
   WHEN VALUE_ERROR THEN
      -- osetreni chyb pri konverzi mezi datovymi typy
   WHEN OTHERS THEN
      -- osetreni vsech ostatnich chyb
END;
```

Předefinovaných druhů výjimek je více, detaily lze nalézt v dokumentaci databázového systému. Často však vystačíme pouze s ramenem WHEN OTHERS, kdy chceme pouze zaregistrovat jakoukoliv vzniklou chybu. Reakcí na chybu může být provedení INSERT příkazu do tabulky chyb nebo nastavení výstupních parametrů tak, aby signalizovali volající proceduře vznik chyby. Pokud chyba není ošetřena v sekci výjimek, propaguje se do nadřízeného bloku kódu nebo do volající procedury, kde opět je možné její zpracování. Pokud není zachycena a ošetřena nikde, vykonávání celé procedury je ukončeno s chybovým hlášením. Pokud je výjimka zachycena, pokračuje vykovávání programu za sekcí výjimek za

koncovým END, tedy vykonávání kódu se již nevrátí na původní místo, kde chyba vznikla. Jelikož však bloky kódu mohou být zanořené, můžeme elegantně navázat na chybový stav a pokračovat dále ve vykonávání procedury:

```
BEGIN
  BEGIN
      INSERT INTO tab1....
      INSERT INTO tab2.....
      INSERT INTO tab3....
      DBMS_OUTPUT.PUT_LINE('Všechno OK' );
  EXCEPTION
      WHEN OTHERS THEN
          DBMS_OUTPUT.PUT_LINE('Něco se nepovedlo, ale..' );
  END;
  DBMS_OUTPUT.PUT_LINE('... jedeme dál' );
  ...
END;
```

Pokud v uvedeném příkladu dojde k chybě při provádění některého z INSERT příkazů, přeskočí vykonávání do sekce EXCEPTION a následně pokračuje program dál. Pokud k chybě nedojde, je sekce EXCEPTION přeskočena.

Výjimky generuje databáze při vzniku chyb, je možné ale také výjimku vyvolat cíleně přímo v kódu voláním příkazu RAISE.

7.3 Vytváření uživatelských procedur a funkcí

Jak bylo řečeno, uložené procedury a funkce jsou objekty databáze stejně jako tabulky. Vytváříme je příkazem CREATE PROCEDURE, resp. CREATE FUNCTION. Syntaxe těchto příkazů je následující:

```
CREATE PROCEDURE jmeno_proc (parametry) IS
i NUMBER; -- deklarace proměných
BEGIN
  –tělo procedury
END;


CREATE FUNCTION jmeno_funkce (parametry) RETURN datovy_typ IS
i NUMBER;
BEGIN
 --tělo funkce
  RETURN vysledek;
END;
```

Místo příkazu CREATE můžeme použít CREATE OR REPLACE, který v případě, že procedura či funkce již existuje, provede její nahrazení. Při vytváření funkce na rozdíl od procedury musíme specifikovat datový typ výsledku funkce. Výsledek funkce spočítáme v těle funkce v libovolné proměnné, na konci funkce označíme zvolenou proměnnou jako proměnnou obsahující finální výsledek pomocí příkazu RETURN. Tímto příkazem vykonávání funkce končí. Parametry procedur a funkcí jsou proměnné, pomocí kterých předáváme proceduře či funkci vstupní hodnoty při jejím volání. U procedur mohou parametry sloužit i k předávání výsledků (OUTPUT parametry). Parametrů může být 0 až N, stejně jako interní proměnné mají svůj název a datový typ. Definují se za názvem procedury a funkce a vzájemně jsou odděleny čárkou.

Jako příklad funkce můžeme uvést uživatelskou funkci, která ze dvou datumů spočíta věk, tedy rozdíl datumů v celých rocích.

```
CREATE OR REPLACE FUNCTION age (datum1 DATE, datum2 DATE) RETURN NUMBER
IS
roku NUMBER;
BEGIN
   roku := ABS(TRUNC(MONTHS_BETWEEN(datum1, datum2) / 12));
   RETURN roku;
END;
```

Na vstupní hodnoty použijeme ORACLE funkci MONTHS_BETWEEN, výsledek podělíme dvanácti, odřízneme desetinnou část a funkcí ABS zajistíme, že výsledek je kladné číslo pro případ, že předáme funkci datumy v opačném pořadí.

Tuto funkci můžeme pak následně použít jako jakoukoliv jinou funkci v SQL dotazech:

```
SELECT age(date_of_birth, SYSDATE) vek FROM patients
```

Jako uloženou proceduru si můžeme definovat kód, který provádí hromadné mazání (sérii DELETE příkazů) tabulek v našem schématu.

```
CREATE PROCEDURE uklid () IS
BEGIN
    DELETE FROM tab1;
    DELETE FROM tab2;
    DELETE FROM tab3;
 END;
```

Abychom spustili uloženou proceduru z databázového klienta, musíme její volání vymezit jako PL/SQL blok pomocí BEGIN a END. Spuštění procedury uklid bude vypadat takto:

```
BEGIN
    uklid();
END;
```

7.4 Procedurální jazyk PG/SQL

Procedurální jazyk databáze POSGRESQL se označuje jako PG/SQL nebo také plpgsql a od PL/SQL databáze ORACLE se odlišuje v několika bodech. Předně databáze POSTGRESQL podporuje pouze uživatelské funkce, nikoliv procedury. Nicméně pomocí funkcí lze dosáhnou stejného efektu jako pomocí procedur v PL/SQL.

Funkce PG/SQL definujeme následovně:

```
CREATE FUNCTION nejaka_funkce() RETURNS NUMERIC AS $$
DECLARE
   promenna NUMERIC := 30;
BEGIN
   RAISE NOTICE 'Promenna obsahuje cislo %', promenna;
                     -- % je ve vypisu nahrazeno obsahem promenne
   RETURN promenna;
END;
$$ LANGUAGE plpgsql;
```

Na příkladu lze vidět některé odlišnosti (červeně) v syntaxi oproti PL/SQL. Celý procedurální kód je uzavřen mezi symboly "$$", na konci je pak navíc specifikace použitého jazyka za klíčovým slovem LANGUAGE. Deklarace proměnných je uvozena klíčovým slovem DECLARE. Příkaz RAISE, který v ORACLE slouží výhradně pro vyvolání výjimky, se v POSTGRESQL využívá i k zobrazení zpráv. RAISE NOTICE pouze zobrazí specifikovaný text, zatímco RAISE EXCEPTION vyvolá výjimku.

Operátor přiřazení (:=) je shodná s PL/SQL, stejně jako podmínkové konstrukce IF-ELSIF-ELSE. Podporovány jsou taktéž smyčky LOOP a WHILE. Smyčka FOR pro procházení řádků výsledku SQL dotazu má drobné odlišnosti od verze v PL/SQL:

```
CREATE OR REPLACE FUNCTION smycka() RETURNS integer AS $$
DECLARE
   k RECORD;
BEGIN
   FOR k IN SELECT * FROM patients LOOP
            RAISE NOTICE 'pacient %', k.patient_id;
   END LOOP;
   RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Předně proměnná použitá jako vektor musí být deklarována jako každá jiná proměnná, datový typ pro kurzor je RECORD. Druhý drobnější rozdíl je, že SQL dotaz ve FOR smyčce není třeba uzavírat do závorek.

Uživatelskou funkci voláme jako standardní funkce pomocí SQL dotazů. Pokud chceme jen vynutit její spuštění, použijeme konstrukci:

SELECT nazev_funkce() FROM GENERATE_SERIES(1,1)

nebo zkrácenou variantu

SELECT nazev_funkce()

Tento SQL dotaz zajistí právě jedno spuštění uživatelské funkce.

Co se týká výjimek, je způsob jejich zachytávání shodný s PL/SQL, liší se pouze seznam možných typů systémových výjimek.

8 Export a import dat

Výstupy:

- Umí exportovat data z databázového klienta
- Zná databázová rozhraní označovaná jako ODBC, OLED DB
- Umí načíst data do databáze z textového souboru
- Zná základní ovládání nástroje SQLLDR
- Umí pracovat s externími tabulkami ORACLE

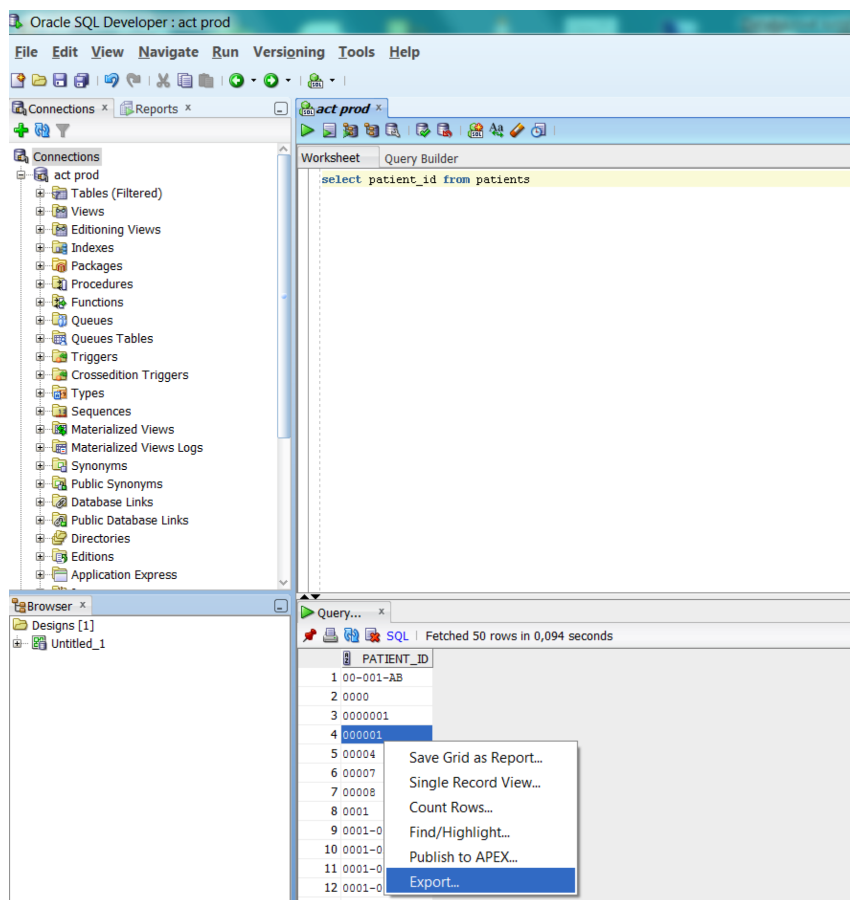
Hlavním úkolem databázového systému je uchovávat data a na vyžádání je poskytovat oprávněným uživatelům. Doposud jsme data pouze prohlíželi ve výstupním okně klienta na našem monitoru. V této kapitole si ukážeme, jak data exportovat ven z databáze, a to buď do souboru, nebo přímo do jiné softwarové aplikace jako je například statistický program. Ukážeme si také, jak data naopak efektivně do databáze vkládat.

8.1 Export dat pomocí databázového klienta

Databázový klient nám prostřednictvím SELECT příkazu zpřístupňuje data uložená v databázi. V základním režimu tato data pouze zobrazuje na monitor. Jednou z věcí, kterou se klienti liší, je nabídka jiného způsobu poskytnutí (exportu) dat.
Grafický klient ORACLE **SQLDeveloper** nabízí následující možnosti exportu dat:

- csv
- delimited
- fixed
- html
- insert
- loader
- pdf
- text (tabulátor)
- XLS
- XML

K této nabídce se dostaneme, pokud v tabulce s výsledkem dotazu klikne pravým tlačítkem myši a zvolíme volbu Export:



Obrázek 7 - Export dat v programu SQLDeveloper

"Html" a **"PDF"** typ exportu jsou určené pro prezentační výstupy, ostatní typy jsou určeny pro další zpracování nebo import do jiné databáze či software. Varianta "**insert**" exportuje data ve formátu INSERT příkazů, které můžeme spustit jako skript a přenést tak data do jiné databáze. Obdobně typ "**loader**" exportuje data ve formátu určeném pro načtení do jiné ORACLE databáze prostřednictvím aplikace SQLLDR, o které si povíme dále v této kapitole. Exporty typu "**csv**" a "**text**" jsou podskupinou exportu "**delimited**". Jde vždy o export dat do textového souboru, kde jsou jednotlivé sloupce exportované tabulky odděleny vybraným oddělovacím znakem. V případě "**csv**" je to čárka, v případě "**text**" je to znak tabulátoru, v případě "**delimited**" exportu si můžeme oddělovací znak zvolit. Varinta "**fixed**" exportuje data taktéž jako textový soubor, místo oddělovače ale mají sloupce vždy stejnou šířku, všechny hodnoty jsou zarovnané mezerami na maximální možnou šířku sloupce. **XLS** export zapíše data ve formátu MS Excel. **XML** typ exportu vytvoří z dat XML soubor. XML formátem se budeme zabývat v samostatné kapitole.

Jednodušší řádkový ORACLE klient **SQLPLUS** umožňuje přesměrovat výstup z monitoru do textového souboru pomocí příkazu SPOOL, za které uvedeme cestu k výstupnímu souboru.

SPOOL jmeno_soboru

Po spuštění tohoto příkazu bude veškerý výstup na monitor zapisován do uvedeného souboru. Ukončení zápisu provedeme příkazem SPOOL OFF.

V PostgreSQL v klientovi pgAdmin je možné výsledek dotazu exportovat přes funkci **Exportovat** v menu programu Soubor. Jde o variantu "delimited" exportu do textového

souboru. V řádkovém klientu můžeme využít příkaz COPY, který uloží obsah tabulky do specifikovaného textového souboru.

COPY patients TO 'C:/vystup.txt'

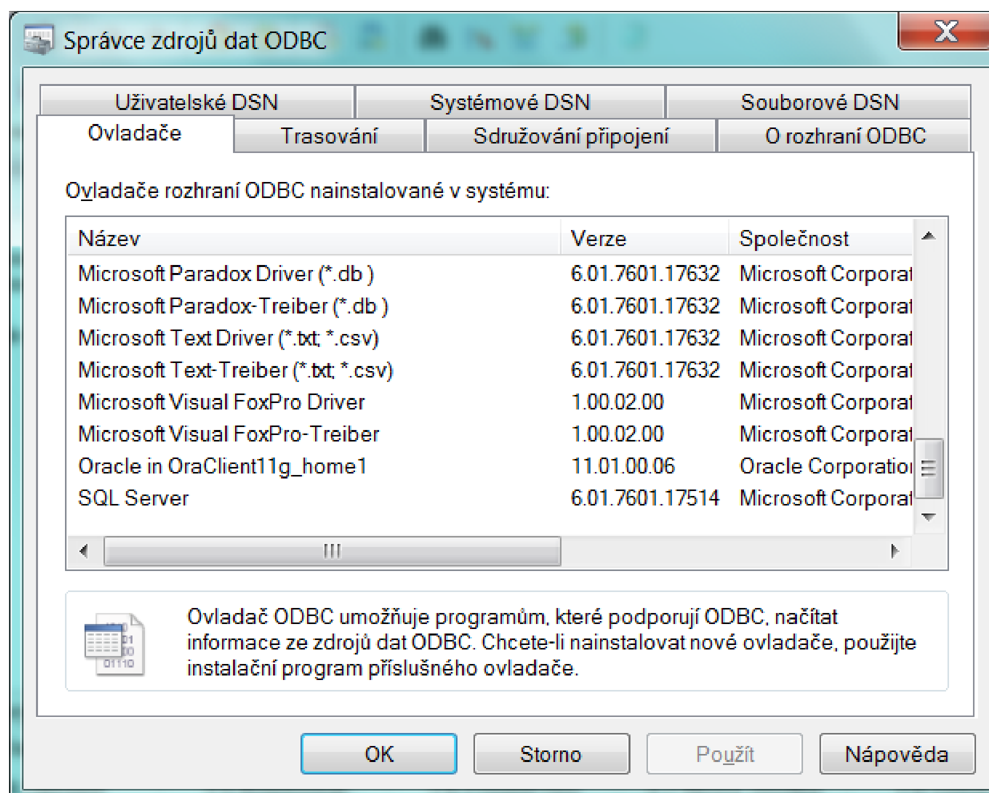
Ve výsledném textovém souboru jsou sloupce odděleny znakem tabulátoru. V uvedeném příkladu je v plném názvu výstupního souboru skutečně normální lomítko, nikoliv obrácené, jak je obvyklé v systému Windows. Pokud chceme exportovat výsledek SQL dotazu, musíme ho uzavřít do kulatých závorek:

COPY (SELECT patient_id FROM patients) TO 'C:/vystup.txt'

8.2 Univerzální databázová rozhraní

8.2.1 ODBC

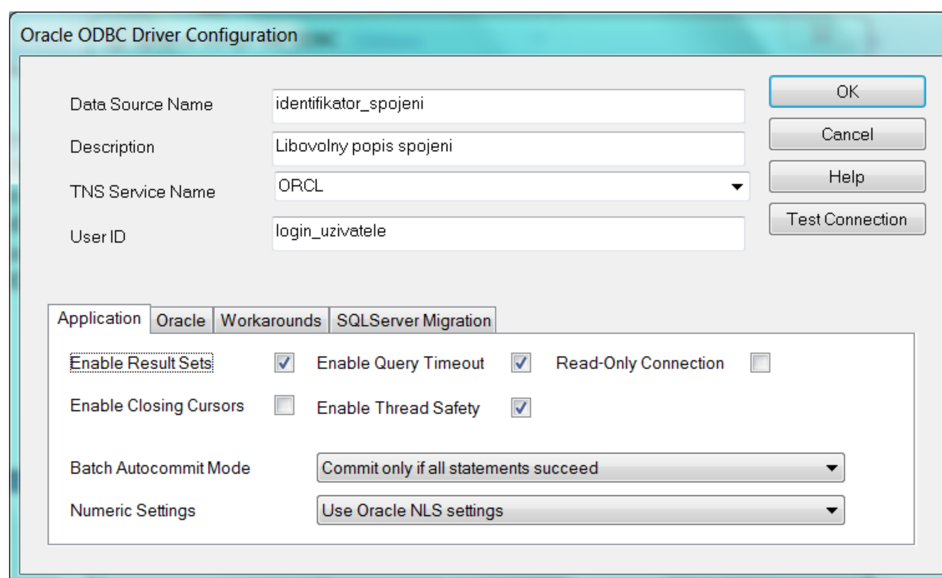
Další možností jak získat data z databáze pro další zpracování je načtení do aplikací přes databázová rozhraní ODBC nebo OLEDB. ODBC je starší rozhraní, které umožňuje načítání dat z databáze do aplikací systému Windows (podporu ODBC najdeme však i v systému Linux) prostřednictvím tzv. ODBC ovladače (driveru), který je obvykle dodáván výrobcem databáze. ODBC standard nám umožní čerpat data stejným způsobem z nejrůznějších databází. Odlišnosti a specifika databázových systémů řeší právě zmíněný ODBC ovladač. ODBC rozhraní je součástí systému Windows. Konfiguraci provádíme přes Nástroje pro administrátory - ODBC rozhraní (Windows 7). Zde v kartě nalezneme seznam instalovaných ovladačů (Obrázek 8).



Obrázek 8 - Správce ODBC zdrojů v MS Windows

Konkrétní konexi do cílové databáze vytvoříme buď v kartě Uživatelské DSN nebo Systémové DSN podle toho, zda chceme, aby spojení bylo dostupné pouze nám nebo všem uživatelům našeho počítače. Ve zvolené kartě klikneme na tlačítko přidat a vybereme ze

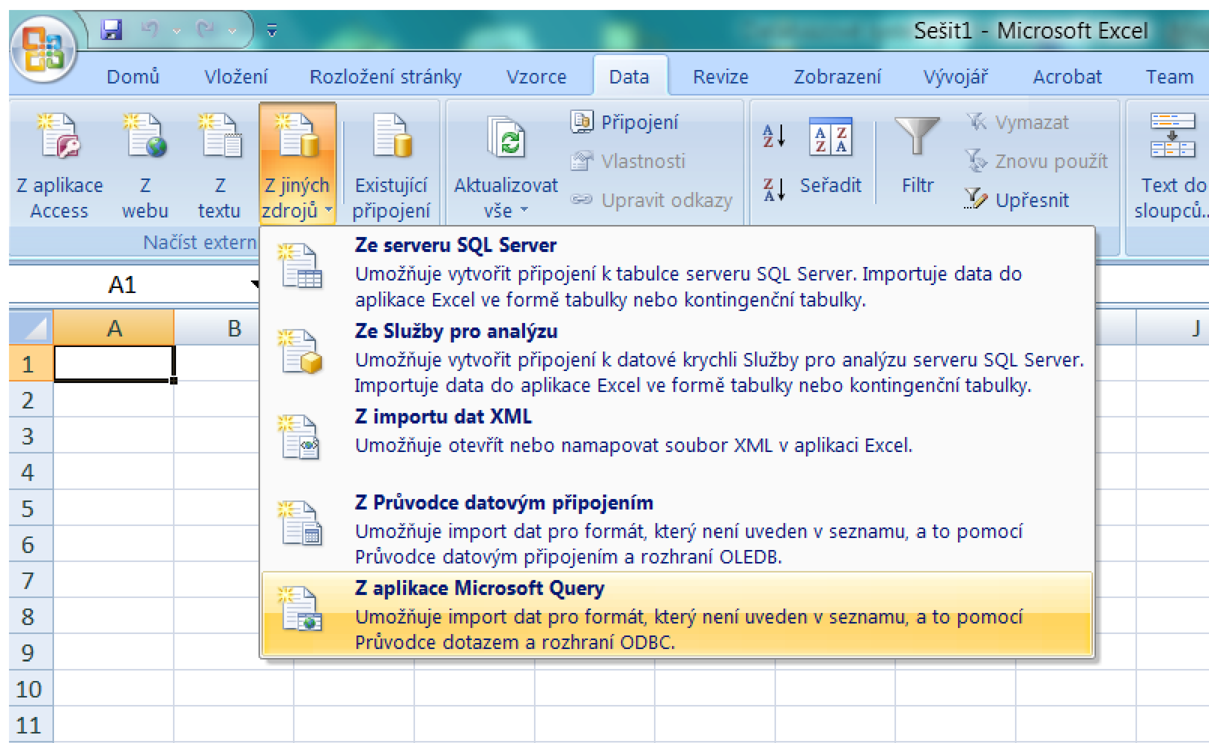
seznamu adekvátní ovladač. Pokud v nabídce požadovaná ovladač nenalezneme, musíme získat jeho instalační soubor a provést instalaci. Po dvojkliknutí na název ovladače se otevře konfigurační okno, které se liší podle zvoleného ovladače. Konfigurační okno ORACLE ODBC ovladače ukazuje Obrázek 9.



Obrázek 9 - Konfigurační okno ORACLE ODBC ovladače

Položkou "Data source name" identifikujeme spojení. Tento název budeme následně používat při připojování do databáze. "Description" je nepovinný popis spojení. "TNS Service name" je specifikum ORACLE databáze a identifikuje nám cílový databázový server. Položka "UserID" nastavíme na databázový login, který budeme při připojení používat. Není to však nutné, login a heslo se standardně specifikují až v momentě připojování do databáze. Ostatní nastavení ORACLE ODBC ovladače můžeme ponechat ve výchozím nastavení. Nadefinované připojení můžeme otestovat stisknutím tlačítka "Test Connection". Po kliknutí na "OK" máme připraveno ODBC zdroj, který můžeme začít používat.

ODBC připojení podporuje například MS Excel. Přes aplikaci Microsoft Query lze načíst data z databáze do zvoleného listu (Obrázek 10). Po kliknutí na tuto volbu je nám nabídnut seznam nadefinovaných ODBC spojení. Zvolíme to, které jsme nadefinovali v předchozím kroku a v následujícím okně vyplníme náš login a heslo do databáze (Obrázek 11). Po úspěšném připojení můžeme definovat SQL SELECT dotaz, jehož výsledek bude přenesen do excelového listu.

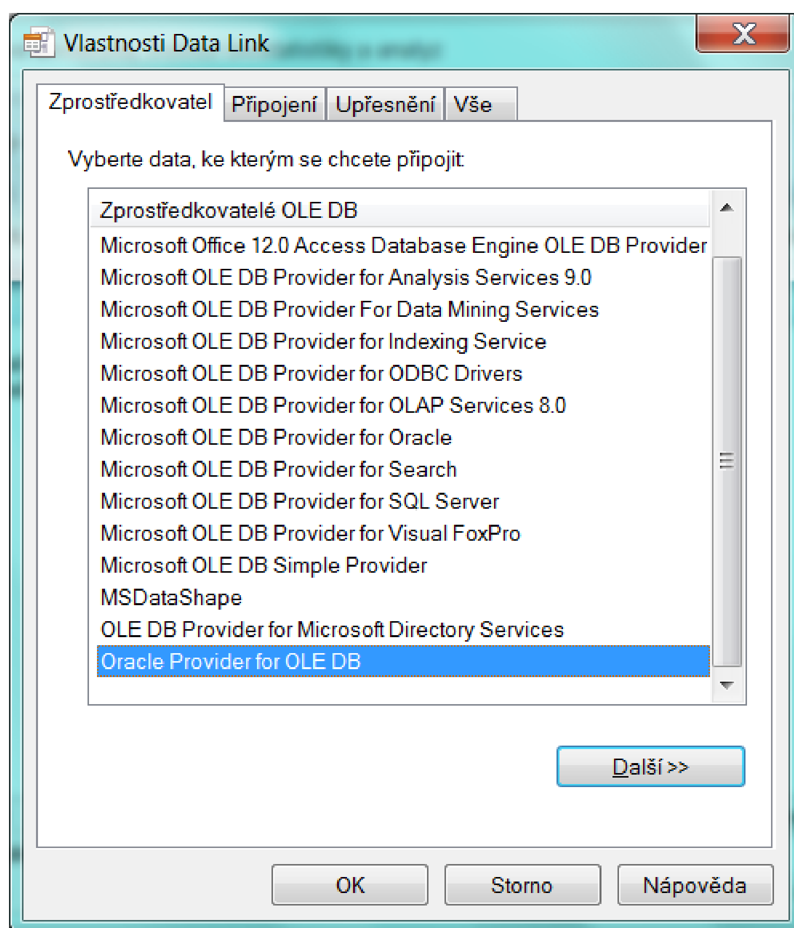


Obrázek 10 - ODBC připojení jako zdroj dat v MS Excel

Obrázek 11 - Připojování do ORACLE databáze přes ODBC spojení

8.2.2 OLEDB

OLEDB je novější obdobou ODBC rozhraní. Opět potřebujeme ovladač pro daný databázový systém, který zajistí, že stejným způsobem přistupujeme k uloženým datům bez ohledu na výrobce databáze. Rozdíl oproti ODBC z uživatelského pohledu je v konfiguraci připojení. To se obvykle provádí v konfiguračním souboru s příponou .udl. Pokud v systému Windows poklepeme na takovýto soubor, objeví se dialog konfigurace OLEDB připojení k databázi. V prvním okně je seznam nainstalovaných OLEDB driveru, další okna se liší podle konkrétního driveru (Obrázek 12).



Obrázek 12 - Okno konfigurace OLEDB připojení

Připojení je možné definovat i bez konfiguračního souboru přímo v cílové aplikaci. OLEDB je modernější, rychlejší a snadněji konfigurovatelné rozhraní než ODBC, nicméně nabídka ovladačů je o něco nižší a ne všechny jsou plně funkční, zvláště ty dodávané třetí stranou. Pro ORACLE je nejvhodnější využívat ovladač přímo od ORACLE.

Také OLEDB připojení je možné použít pro přenos dat do MS Excel, ve verzi 2007 a vyšší najdeme nabídku opět v menu "Data", skupina tlačítek "Načíst externí data", skupina funkcí "Z jiných zdrojů", varianta Průvodce datovým připojením.

8.3 Import dat

8.3.1 Import pomocí INSERT příkazů

Základní metoda vkládání dat do databáze je prostřednictvím příkazu INSERT. Jde o univerzální metodu, která se hodí pro vkládání menšího objemu dat. Insert příkazy lze sestavit v textovém editoru, pomocí programovacích prostředků nebo třeba v aplikaci MS Excel pomocí vzorců. Tuto metodu si ukážeme na jednoduchém příkladu, kdy máme v excelovém souboru číselný sloupec a sloupec s textem. Do třetího sloupce sestavíme vzorec:

Tabulka 27 - Ukázka tvorby INSERT příkazů v MS Excel pomocí vzorců

Číselný sloupec	Textový sloupec	Vzorec
1	text1	="INSERT INTO tabulka (cislo, text) VALUES (" & A2 & ",'" & B2 & "');"

2	text2	="INSERT INTO tabulka (cislo, text) VALUES (" & A3 & ",'" & B3 & "');"
3	text3	="INSERT INTO tabulka (cislo, text) VALUES (" & A4 & ",'" & B4 & "');"

Vzorec v MS Excel začíná vždy znakem "=", pokud vzorec obsahuje text, musí být uzavřen mezi uvozovky. Symbol "&" spojuje text s odkazem na obsah vybraného pole. Odkaz je tvořen písmenem sloupce a číslem řádku (A2 = sloupec A, druhý řádek). Nesmíme zapomenout na to, že příkaz INSERT vyžaduje, aby vkládaná **textová** hodnota byla ohraničena apostrofy. Pomocí vzorců získáme následující složený text INSERT příkazů:

INSERT INTO tabulka (cislo, text) VALUES (1,'text1');

INSERT INTO tabulka (cislo, text) VALUES (2,'text2');

INSERT INTO tabulka (cislo, text) VALUES (3,'text3');

Máme připraven importní skript, který nakopírujeme do databázového klienta a spustíme.

8.3.2 Import dat pomocí SQLLDR (ORACLE)

Metoda vkládání dat pomocí sestavovaných INSERT příkazů je vhodná do cca 10 tisíc záznamů. Pro větší objem dat je vhodnější použít specializovaný software pro import data, tzv. datovou pumpu. Více či méně kvalitních datových pump je ke stažení velké množství, pro práci s databází ORACLE je však nejspolehlivější použít aplikaci **SQLLDR** přímo od firmy ORACLE. Tato aplikace je součástí instalace databázového klienta a patří mezi databázové utility. Je primárně určena k importu dat z textového souboru do databáze. Jde o řádkovou aplikaci, která se ovládá přes kontrolní soubor. Pomocí něj specifikujeme strukturu importovaných dat i cílovou tabulku. Importovat lze textový soubor se sloupci oddělenými specifickým oddělovačem i textové soubory s pevnou strukturou. Možnosti konfigurace aplikace Sqlloader jsou velice široké, podíváme se na nejdůležitější prvky. Mějme textový soubor (zdroj.txt), který obsahuje 3 sloupce, jeden číselný, druhý textový, třetí bude obsahovat datumy, sloupce jsou od sebe odděleny středníkem. Než začneme s importem, musíme nejprve vytvořit cílovou tabulku:

```
CREATE TABLE importovana_data
(
   cislo NUMBER(10,0),
   text VARCHAR2(50),
   datum DATE
);
```

Nejstručnější konfigurační soubor aplikace Sqlloader musí obsahovat minimálně odkaz na zdrojový soubor, cílovou tabulku, specifikaci oddělovače sloupců a seznam importovaných sloupců:

```
LOAD   DATA   INFILE  "C:\zdroj.txt"
APPEND INTO TABLE importovana_data
FIELDS TERMINATED BY ";"
```

(cislo, text, datum DATE "DD.MM.YYYY")

Tímto zápisem v konfiguračním souboru říkáme datové pumpě, aby načetla data ze souboru zdroj.txt, který obsahuje 3 sloupce oddělené středníkem, a zapsala je do tabulky importovana_data. Výčtem sloupců v kulatých závorkách specifikujeme pořadí sloupců ve zdrojovém souboru a u datumu navíc specifikujeme formát. Konfigurační soubor lze dále rozšířit o sekci OPTIONS, pomocí které specifikujeme chování datové pumpy při importu. Nejčastější konfigurovatelné vlastnosti shrnuje Tabulka 28:

Tabulka 28 - Nejčastěji používané konfigurovatelné vlastnosti SQLLDR

Vlastnost	Význam
SKIP=n	Přeskočit na začátku n řádků (např. vynechat záhlaví)
ERRORS=n	Ukončit import po dosažení n chyb
ROWS=n	Provést commit po importu n řádků
DIRECT=true	Volba rychlejší cesty importu

Pokud chceme, aby se při importu ignoroval první řádek v importovaném souboru a aby import skončil při první chybě, vložíme na začátek konfiguračního souboru tento řádek:

OPTIONS (SKIP=1, ERRORS=0)

Je důležité zmínit, že i v případě, že import skončí chybou, bude proveden commit na úspěšně importované řádky. Toto chování je nepříjemné, pokud importujeme data do neprázdné tabulky, kdy při neúspěšném importu nemůžeme jednoduše vše smazat a začít znovu a zároveň nechceme v cílové tabulce duplicity. V tomto případě se doporučuje nastavit vlastnost ERROR na velmi vysoké číslo, aby došlo ke zpracování všech korektních řádků. Chybové řádky budou aplikací zapsány do soubor.bad souboru, čímž vznikne sekundární importní soubor, který pro vyřešení příčin chyb můžeme opět zkusit importovat. Tuto iteraci opakujeme, dokud se nám nepodaří importovat všechny řádky. Druhou možností je provádět import do prázdné pracovní tabulky a teprve z ní pomocí SQL příkazu INSERT SELECT převést data do cílové tabulky. U extrémně velkých souborů, kdy samotný import trvá hodiny a opakování importu je časově neefektivní, si při předčasném ukončení importu pomáháme nastavením parametru SKIP, který nastavujeme za poslední úspěšně importovaný řádek, případně těsně za detekovaný chybový řádek.

Pomocí přepínače DIRECT řídíme metodu importu. Výchozí metoda (DIRECT=false) je metoda INSERT příkazů, kdy SQLLDR ze vstupního souboru sestavuje INSERT příkazy, které spouští. Touto metodou lze bez problémů zpracovat soubory v řádu sta tisíc řádků, ale v případě větších souborů je již tato cesta časově náročná. U velkých souborů se doporučuje využít přímou cestu importu (DIRECT=true). Tato varianta je rychlejší, nicméně nekontrolují se některá integritní omezení a může se tak do databáze dostat řádek, který by při standardní cestě byl odmítnut. U velkých souborů je proto zcela na místě provádět import do pracovních tabulek, zde provést nezbytné kontroly a teprve následně přenášet data do cílových tabulek.

8.3.3 Import přes externí tabulky (ORACLE)

Jako alternativu při importu dat z textového souboru můžeme v případě databáze ORACLE použít techniku označovanou jako externí tabulka (external table). Tato metoda je vhodná, pokud provádíme import dat v definované struktuře opakovaně. Podstatou této metody je, že se díváme na importní textový soubor jako na tabulku, která je přímo součástí

databáze, jde o databázový objekt. Tento typ objektu vytvoříme pomocí příkazu CREATE TABLE ORGANIZATION EXTERNAL. V tomto příkazu popíšeme umístění a strukturu textového souboru a s vytvořeným objektem můžeme následně pracovat jako s běžnou interní tabulkou pomocí příkazu SELECT. Platí zde však jistá omezení, mezi hlavní patří skutečnost, že zdrojový soubor je nutné umístit přímo na databázový server do vyhrazeného adresáře (naproti tomu import přes SQLLDR můžeme provádět odkudkoliv, kde máme nainstalovaného ORACLE klienta). Dále oproti interní tabulce nelze použít příkazy INSERT, UDATE, DELETE pro manipulaci s daty uvnitř externí tabulky. Nad externí tabulkou taktéž nelze vytvářet indexy pro optimalizaci vyhledávání.

Abychom mohli používat externí tabulky, musíme mít nejdříve na serveru vytvořený a zpřístupněný importní adresář. Tuto operaci obvykle provádí správce databáze:

```
CREATE OR REPLACE DIRECTORY
IMPORT_DIR AS
'/home/oracle/import';
GRANT READ, WRITE ON DIRECTORY IMPORT_DIR TO uzivatel;
```

Do tohoto adresáře nakopírujeme zdrojový soubor a přistoupíme k vytvoření objektu externí tabulky. Syntaxe tohoto příkazu připomíná konfigurační soubor SQLLDR aplikace, nicméně není to to samé. Ukážeme si vzor příkazu pro import souboru s fixní velikostí sloupců (pozicový formát) a vzor pro import souboru, kde sloupce jsou odděleny vybraným oddělovačem. V obou případech zdrojový soubor obsahuje 3 sloupce, jeden číselný, druhý textový, třetí je datum. Pro pozicový soubor s názvem data_fix.txt použijeme následující příkaz:

```
CREATE TABLE data_fix
(
  cislo    NUMBER,
  jmeno    VARCHAR2(20),
  datum    DATE
)
  ORGANIZATION EXTERNAL (
    TYPE ORACLE_LOADER
    DEFAULT DIRECTORY IMPORT_DIR
    ACCESS PARAMETERS
      (RECORDS DELIMITED BY NEWLINE
        FIELDS (
          cislo    CHAR(3),
          jmeno    CHAR(6),
          datum    CHAR(10) DATE_FORMAT DATE MASK "dd.mm.yyyy"
        )
      )
   LOCATION ('data_fix.txt')
```

```
);
```

Příkaz obsahuje dvakrát jména importovaných sloupců, liší se datové typy. V prvním případě jde o definici, jak se na data chceme dívat ze vnitř databáze, v druhém případě za klíčovým slovem FIELDS popisujeme situaci ve zdrojovém textovém souboru. Zde jsou všechny prvky pouze text s vymezenou délkou, proto používáme datový typ CHAR s vymezením délky. V případě datumu doplňujeme specifikaci formátu ve stejné notaci jako v případě funkce TO_DATE. Po vytvoření externí tabulky si data můžeme prohlédnout běžným SELECT příkazem:

```
SELECT * FROM data_fix
```

Pokud jsou sloupce ve zdrojovém souboru definovány pomocí oddělovacího znaku, musíme příkaz pro vytvoření externí tabulky doplnit o specifikaci oddělovače:

```
CREATE TABLE data_delimited
(
  cislo    NUMBER,
  jmeno   VARCHAR2(20),
  datum   DATE
)
  ORGANIZATION EXTERNAL (
    TYPE ORACLE_LOADER
    DEFAULT DIRECTORY IMPORT_DIR
    ACCESS PARAMETERS
      (RECORDS DELIMITED BY NEWLINE
       FIELDS TERMINATED BY ";"
        (
          cislo    CHAR(3),
          jmeno   CHAR(6),
          datum   CHAR(10) DATE_FORMAT DATE MASK "dd.mm.yyyy"
        )
      )
    LOCATION ('data_delimited.txt')
  );
```

Vytvoření objektu externí tabulky nám nebrání manipulovat se zdrojovým souborem. Můžeme ho pomocí nástrojů operačního systému bez problémů nahradit a opětovně použít SELECT příkaz pro prohlédnutí dat. Pokud SELECT příkaz skončí chybou, najdeme bližší vysvětlení v log souboru, který se vytvoří na serveru v importním adresáři. Z uvedeného vyplývá, že externí tabulky jsou užitečným nástrojem pro pravidelně opakovaný import dat, nicméně neobejdeme se bez spolupráce s administrátorem databáze, resp. serveru.

8.3.4 **Import do POSTGRESQL**

Textový soubor lze do POSTGRESQL importovat pomocí COPY příkazu ve variantě:

COPY cilova_tabulka FROM 'zdrojovy_soubor'

Podporován je import textového souboru, kde sloupce jsou odděleny specifikovaným jednoznakovým oddělovačem. Ten specifikujeme za klíčovým slovem DELIMITER, výchozím oddělovacím znakem je tabulátor, který nemusíme explicitně specifikovat. V jiných případech je nutné příkaz COPY uvést včetně oddělovače:

COPY patients FROM 'C:/vystup.txt' DELIMITER ';'

Existují ještě další možnosti specifikace formátu zdrojového souboru, nicméně nabídka je nesrovnatelně menší než v případě ORACLE databáze a její datové pumpy SQLLDR. Příkaz COPY podporuje také binární formu exportu a importu, tuto variantu lze použít však pouze při přenosu dat mezi dvěma POSTGRESQL databázemi.

9 Základy XML

Databázové systémy jsou primárně určeny k ukládání dat. Data je však často třeba sdílet, vyměňovat a migrovat z jednoho systému do druhého. Základní možností importu a exportu dat z databáze jsme si ukázali v předchozí kapitole. Jedním z formátů, který můžeme pro export dat použít v klientovi Sqldeveloper je XML formát. Právě tomuto formátu je věnována tato kapitola. Seznámíme se s jeho základy, s nástroji pro jeho zpracování, se souvisejícími technologiemi a v neposlední řadě se způsobem zpracování tohoto formátu v databázi.
XML formát je textový formát, kde přenášené datové položky jsou obklopeny popisnými položkami tzv. tagy:

<nazev>obsah</nazev>

XML formát tak připomíná HTML jazyk webových stránek, nicméně jeden zásadní rozdíl. Názvy tagů v XML jsou určovány autorem dané konkrétní datové struktury nikoliv samotným XML standardem. XML standard ve svém základu standardizuje pouze několik věcí. Mezi ně patří například způsob přenosu speciálních znaků. Speciálními znaky myslíme znaky, které mají v samotém XML standardu speciální význam, např. znaky "<" a ">" ohraničující názvy tagů. Pokud tyto znaky chceme zapsat do XML v původním významu, musíme je nahradit specifikovaným výrazem. Jejich přehled uvádí Tabulka 29.

Tabulka 29 - Vyhrazené znaky v XML a jejich kódování

Vyhrazený znak	Náhradní výraz
< menší než	<
> větší než	>
& ampersand	&
' apostrof	'
" uvozovky	"

Základním prvky XML formátu jsou

- Elementy
- Atributy
- Komentáře
- Instrukce

Element je základní jednotka struktury XML dokumentu. Má své jméno a obsah. Jméno je uzavřeno mezi symboly < a >, např. <jmeno_elementu>. Jménem elementu je ohraničen obsah elementu, mluvíme o otevíracím a uzavíracím tagu. Uzavírací tag se od otevíracího liší pouze lomítkem před jménem elementu. Jednoduchý element včetně obsahu vypadá takto:

<jmeno_elementu>obsah elementu</jmeno_elementu>

Element může být i prázdný, bez obsahu:

<book></book>

Zjednodušený zápis prázdného elementu vypadá takto

```
<book />
```

Jméno elementu nesmí obsahovat mezery a musí začínat písmenem. Bez problémů je možné používat číslice a podtržítko. Znaky s diakritikou ve jménech elementu jsou sice standardem povoleny, nicméně doporučuje se jim vyhýbat stejně jako jiným nealfanumerickým znakům. Rozlišují se velká a malá písmena, <JMENO> je tedy odlišné od <jmeno>

Obsahem elementu může být libovolný text, pozor je třeba dávat jen na speciální znaky, které jsou uvedeny v Tabulka 29.

Text obsahující větší množství vyhrazených znaků je bez nahrazování vložit do XML dokumentu jako CDATA blok. Pokud obsah elementu uvedeme mezi

"<![CDATA[" a " **]]>"**, budou zmíněné speciální znaky brány jako znaky standardní, bez speciálního významu. Do CDATA bloku je možné uvést cokoliv s výjimkou ukončovací sekvence "]]>". CDATA blok nám poslouží třeba v případě, kdy chceme přenášet zdrojový kód webové stránky v HTML.

Element může obsahovat zanořený další element, čímž vzniká charakteristická stromová struktura XML dokumentu.

```
<pacient>
   <vysetreni>
         <datum_vysetreni>27.7.2014</datum_vysetreni>
   </vysetreni>
</pacient>
```

Možnosti zanoření jsou neomezené, platí však, že na nejvyšší úrovni může být jen jeden, tzv. root element. Zakázán je taky překryv elementů. Následující zanoření tedy není povoleno

```
<pacient>
   <vysetreni>
         <lecba>
   </vysetreni>
         </lecba>
</pacient>
```

Elementy mohou obsahovat tzv. atributy, které upřesňují informaci přenášenou v těle elementu. Atributy se zapisují do otevíracího tagu elementu, jejich hodnoty se uvádí v uvozovkách za znakem "=".

```
<nazev_elementu nazev_atributu="hodnota atributu">obsah elementu<nazev_elementu>
<delka_hospitalizace jednotky="dny">9< delka_hospitalizace>
```

V XML dokumentu je možné uvádět komentáře, které se při zpracování ignorují a slouží pouze k uvedení vysvětlivek. Komentáře musí být ohraničeny tímto způsobem

```
<!-- Toto je komentář -->
```

Posledním prvkem, který se obejvuje v XML dokumentech jsou procesní instrukce (Processing instructions). Jak název napovídá, jde o instrukce, které říkají XML procesoru, jak

se zachovat. Procesní instrukce jsou uzavřeny mezi symboly <? a ?>. Každý XML dokoment musí obsahovat na svém začátku tuto procesní instrukci:

<?xml version="1.0"?>

Často tato úvodní instrukce bývá doplněna o znakovou sadu, která je použita pro kódování znaků. Standardem je kódování UTF-8.

<?xml version="1.0" encoding="UTF-8" ?>