JAZYK SQL – DATABÁZOVÝ ŠTANDARD

V nasledujúcej časti textu sa zoznámime s jazykom SQL, ktorý sa dá označiť za univerzálny jazyk relačných databáz, pretože ho podporujú prakticky všetky moderné databázové systémy. Podobne ako iné počítačové jazyky, je SQL medzinárodným štandardom, uznaným množstvom normatívnych orgánov ako je ISO a ANSI. Keď používame jazyk SQL, musíme použiť správnu syntax. Syntax je skupina pravidiel, pomocou ktorých sa prvky jazyka správne kombinujú. Syntax jazyka SQL je založená na syntaxi angličtiny a používa rovnaké prvky ako syntax jazyka *Visual Basic for Applications* (VBA).

Jazyk SQL je príkazovo orientovaný, pričom **príkazy jazyka SQL formulujeme pomocou klauzúl s príslušnými kľúčovými slovami a parametrami**. **Kľúčové slová** tvoria obvykle v databázovom systéme vyhradené – rezervované slová, takže ich nesmieme používať ako názvy bežných databázových objektov. **Klauzuly** príkazov sa väčšinou musia zapisovať v správnom, predpísanom poradí. **Každý príkaz jazyka SQL musí skončiť bodkočiarkou (;)**.

Okrem týchto základných obmedzení, majú príkazy jazyka voľnú formu, takže jednotlivé jazykové elementy môžeme oddeľovať pomocou jednej alebo viacerých medzier, a dokonca medzi týmito elementmi môžeme zapisovať i koniec riadkov (ale nie vo vnútri elementu). Celkovo môžeme príkazy jazyka SQL rozdeliť do nasledujúcich kategórii:

- **Jazyk pre dopytovanie dát** (*Data Query Language, DQL*). Do tejto skupiny patria **príkazy, ktoré sa dopytujú na databázu, ale v dátach ani v databázových objektoch nerealizujú žiadne zmeny**. Konkrétne ide o jediný príkaz, SELECT. Niektorí výrobcovia túto kategóriu zvlášť neoddeľujú a zaraďujú príkaz SELECT do jazyka DML.
- **Jazyk pre manipuláciu s dátami** (*Data Manipulation Language, DML*). Túto skupinu tvoria **príkazy, ktoré modifikujú dáta uložené v databázových objektoch**, čiže v tabuľkách. Patria sem príkazy INSERT INTO, SELECT... INTO, UPDATE a DELETE.
- **Jazyk pre definíciu dát** *(Data Definition Language, DDL*). V tomto prípade ide o **príkazy, ktoré slúžia na vytváranie a modifikáciu databázových objektov**. Zatiaľ čo príkazy zo skupiny DQL a DML pracovali s dátami v databázových objektoch, manipulujúce príkazy DDL pracujú so samostatnými databázovými objektmi. Môžeme taktiež povedať, že jazyk DDL zaisťuje správu dátových kontajnerov, zatiaľ čo jazyk DML obhospodaruje dáta vo vnútri týchto kontajnerov. Skupinu tvoria príkazy CREATE, ALTER, DROP a UNION.
- **Jazyk pre riadenie dát** (*Data Control Language, DCL*). Tieto **príkazy ovládajú privilégiá (oprávnenia), ktoré má každý užívateľ databázy v jednotlivých databázových objektoch**. Súčasťou tejto kategórie sú príkazy GRANT a REVOKE.

6.1 Dopytovací jazyk SQL

Definície charakteru dát v databáze – tzv. metadát alebo logickej schémy databázy - sa formulujú v špeciálnom jazyku, ktorý okrem tabuliek a stĺpcov umožňuje definovať aj niektoré vzťahy medzi údajmi – tzv. integritné obmedzenia.

O manipuláciu s jednotlivými dátami sa stará ďalší jazyk, v ktorom sa dajú formulovať požiadavky na vkladanie, odstraňovanie a modifikáciu údajov. Obsahuje tiež prostriedky umožňujúce klásť jednoduchý i náročnejší dopyt, t.j. otázku (často sa používa aj anglické slovo query, ale aj české slovo dotaz) na údaje v databáze. Takýto jazyk preto nazývame dopytovací.

SQL je jazykom, ktorý zahŕňa obe tieto funkcie, t.j. je v ňom možné zadefinovať metadáta, a potom pracovať aj s jednotlivými dátami. Na rozdiel od klasických procedurálnych programovacích jazykov typu Pascal, kde program spočíva v opísaní postupu (algoritmu), je SQL tzv. deklaratívnym jazykom – programátor len kladie požiadavku, spravidla ho nezaujíma, ako sa

tento príkaz realizuje. O jeho preklad a vykonanie sa už postará systém riadenia bázy dát. Kvôli jednoduchej kontrole sa riadi zásadou, že príkaz sa buď vykoná celý, alebo sa nevykoná vôbec.

Pomocou dopytov jazyka SQL môžeme získať výsledky, ktoré by nebolo možné vytvoriť bežnými dopytmi, ale je možné ich taktiež využiť pre tvorbu základných dopytov, napríklad výberových. Tým je taktiež často predurčené, kedy dopyty SQL použijeme:

- jedine pomocou SQL je možné definovať poddopyty,
- vnorený SQL dopyt môže byť umiestený v položke pre kritériá návrhového zobrazenia bežného dopytu, ale taktiež priamo ako pole bežného dopytu,
- pomocou SQL môžeme zlúčiť viac tabuliek do jednej tabuľky,
- jazykom SQL je možné priamo definovať alebo meniť štruktúru tabuliek,
- jazyk SQL je možné použiť v programových moduloch a makrách aplikácie Access.

V aplikácii MS Access sú SQL dopyty rozdelené do troch hlavných kategórií:

- **Zjednocovacie dopyty** – sú určené na tvorbu základných dopytov pre výber záznamov a spájanie tabuliek.
- **Odovzdávajúce dopyty** – sú určené iba na odovzdanie dopytu na databázový stroj (pomocou ODBC) a zapisujú sa v syntaxi databázového stroja.
- **Definičné dopyty** – sú určené na prácu s tabuľkami, sú v podstate obdobou akčných dopytov, ale poskytujú väčšie možnosti.

6.2 Jazyk pre dopytovanie dát

➢ Príkaz SELECT

Príkaz SELECT **slúži na načítanie dát z databázy**, pričom sa skladá z nasledujúcich klauzúl: FROM, WHERE, GROUP BY, ORDER BY, HAVING, ktoré si vysvetlíme v nasledujúcej časti textu. Príkaz SELECT je najzákladnejším a najdôležitejším príkazom jazyka SQL. Je to vlastne **ekvivalent výberového dopytu** s tým, že **môže byť použitý ako na tvorbu samotného výberového dopytu, tak aj na tvorbu poddopytu.**

Syntax:

```
SELECT < Tabulka >. Pole < As Alias > FROM Tabulka
```

Príkaz vyberie z tabuľky *Tabulka* pole s názvom *Pole* a premenuje ho na *Alias*. Pre nepovinné položky syntaxe sa v ďalšom texte bude používať symbol <a>. *Napríklad*: SELECT <Tabulka>.Pole znamená, že argument *Tabulka* nie je povinný.

```
SELECT meno, [ročný príjem] FROM [Zoznam hercov];
SELECT meno AS [MENO HERCA] FROM [Zoznam hercov];
```

Názvy tabuliek a ich polí budeme označovať pomocou identifikátorov. Identifikátor slúži na určenie hodnoty, ktorá je s daným poľom spojená. Jednotlivé polia a tabuľky sa medzi sebou oddeľujú čiarkami. Pokiaľ však vyberáme dáta z viacerých tabuliek, je potrebné medzi nimi špecifikovať reláciu pomocou operácie JOIN alebo klauzuly WHERE. V prípade, že identifikátory tabuliek alebo polí obsahujú medzery, zapisujeme ich pomocou operátora hranatej zátvorky. Hranaté zátvorky nie je nutné okolo identifikátora alebo jeho časti zadávať vždy. Pokiaľ sa v identifikátore nevyskytujú medzery alebo iné špeciálne znaky, aplikácia Access zátvorky automaticky pridá pri čítaní výrazu.

```
SELECT * FROM [Zoznam hercov];
```

Symbol hviezdičky **(*) zastupuje všetky polia danej tabuľky**.

```
SELECT Count(*) AS [Počet hercov v divadle] FROM [Zoznam hercov];
```

V príkaze sme použili agregačnú funkciu *Count*, ktorá spočíta počet údajov argumentu, ktorý je v zátvorke tejto funkcie. Pretože sme ako argument zadali *, bude vypočítaný počet všetkých záznamov (stačí aby bolo vyplnené aspoň jedno pole v každom riadku).

Klauzula FROM

Klauzula FROM **určuje zdroj polí dopytu**. Bez tejto klauzuly by príkaz SELECT bol nepoužiteľný. V aktuálnej databáze musia byť názvy tabuliek jedinečné, aby sa mohlo na ich jednotlivé polia príkazom WHERE zacieliť.

```
SELECT priezvisko FROM [Zoznam hercov];
```

Klauzula WHERE

Pomocou klauzuly WHERE **určujeme podmienku obmedzujúcu záznamy, ktoré budú výsledkom dopytu**. Táto klauzula je ekvivalentom kritérií v návrhovom zobrazení dopytov. Bez zápisu klauzuly WHERE vráti dopyt všetky riadky zo zdrojových tabuliek. Pokiaľ klauzulu do dopytu zadáme, vyhodnotí sa klauzula WHERE pre každý riadok dát, a to podľa pravidiel boolovskej (logickej) algebry pomenovanej podľa *George Boolea*. Vo výsledkoch dopytu sa teda zobrazia len tie riadky dát, pre ktoré sa klauzula WHERE vyhodnotí na logickú hodnotu TRUE.

Pri výberových podmienkach sa môžu zadávať aj **predikáty Between, In** a **Like.**

Operátor Like slúži v jazyku SQL na **prehľadávanie znakových stĺpcov**. Konkrétne tak porovnáva znakový reťazec v danom stĺpci s istým vzorovým textom, a vráti hodnotu TRUE pri zhode, respektíve FALSE pri nezhode. Znak otáznik **(?)** pri zápise výrazu slúži ako **pozičný zástupný znak, takže sa na danej pozícii zhoduje s ľubovoľným nájdeným znakom**. Symbol hviezdička **(*) je nepozičným zástupným znakom, pretože sa zhoduje s ľubovoľným počtom znakov, teda s reťazcom ľubovoľnej dĺžky.**

```
SELECT meno, národnosť, vek FROM [Zoznam hercov] WHERE meno Like "*k*";
```

Dopyt zobrazí *meno, národnosť* a *vek* hercov, ktorých meno obsahuje písmeno „k“.

Pri definovaní kriteriálnej podmienky dopytu môžeme využiť aj **operátor hranatých zátvoriek ([])**, ktorý **zastupuje jeden zo znakov zadaných v zátvorkách**. Napríklad zápis kódu

```
SELECT [Zoznam hercov].meno, [Zoznam hercov].stav
FROM [Zoznam hercov]
WHERE ((([Zoznam hercov].stav) Like "slobodn[ý á]"));
```

zobrazí z tabuľky *Zoznam hercov* všetky záznamy hercov, ktorí v poli *stav* obsahujú slovo slobodný, alebo slobodná.

Pri definovaní kriteriálnych podmienok v návrhu dopytu môžeme využiť aj zástupný symbol negácie **(!)** a rozsah **(–)**. Nasledujúca časť zápisu SQL kódu

```
. . . . .
WHERE ((([Zoznam hercov].národnosť) Like "slov[!e]nská"));
```

zobrazí tie záznamy osôb, ktoré v stĺpci *národnosť* obsahujú hodnotu *slovinská*, ale vynechá záznamy so slovom *slovenská*.

Zástupný znak rozsah uvádza rozsah hodnôt. Napríklad prostredníctvom kódu

```
. . . . .
WHERE ((([Zoznam hercov].[rodné číslo]) Like "[5-7]?????/????"));
```

v poli pre *rodné číslo* týmto zápisom nájdeme záznamy všetkých hercov narodených v roku 1950 až 1979.

Operátor In sa používa, keď chcete porovnať hodnoty v stĺpci s viac než jednou hodnotou. **V syntaxi zápisu tohto operátora sa teda zistí, či sa jeho ľavý operand rovná jednej z hodnôt pravého operandu – množine hodnôt v okrúhlych zátvorkách**. Výsledkom zistenia je opäť logická hodnota TRUE alebo FALSE.

(5 **in** {0, 4, 5}) = true (5 **in** {0, 4, 6}) = false (5 **not in** {0, 4, 6}) = true

```
SELECT *
FROM [Zoznam hercov]
WHERE vek IN (32,75);
```

Dopyt zobrazí všetky polia záznamov hercov, ale len tých, ktorých *vek* je buď 32 alebo 75 rokov.

Ako je možné vidieť z nasledujúceho príkladu, musia sa v klauzule WHERE jednotlivé testované podmienky vyhodnotiť vždy na hodnotu pravda alebo nepravda, teda TRUE alebo FALSE. Do podmienok môžeme zapisovať taktiež **relačné operátory =, >, <, <=, >= a <>, ako aj Is Null a Is Not Null, prostredníctvom ktorých SQL kód vyberie všetky záznamy, u ktorých je/nie je v príslušnom stĺpci vyplnený záznam/vyplnený žiadny záznam.** Výraz **Like “” hľadá záznamy obsahujúce v príslušnom stĺpci prázdny reťazec** (pri textových poliach), **resp. nulovú hodnotu** (0; pri číselných poliach).

V prípade, že v jednej klauzule WHERE vyhodnocujeme niekoľko podmienok, môžeme výsledky spojiť pomocou logických operátorov AND, OR a NOT. V zložitých logických výrazoch môžeme zapisovať taktiež (okrúhle) zátvorky, ktoré – na jednej strane – zlepšujú čitateľnosť výrazu a taktiež stanovujú poradie vyhodnotenia jednotlivých podmienok. Pre výrazy s niekoľkými logickými operátormi platia pomerne komplikované pravidlá priority operátorov. Je teda omnoho jednoduchšie si zapamätať, že podmienka alebo vnorený výraz zapísaný do zátvoriek sa vyhodnotí vždy ako prvý.

```
SELECT [Zoznam hercov].meno, [Zoznam hercov].národnosť, [Zoznam hercov].stav, [Zoznam hercov].vek
FROM [Zoznam hercov]
WHERE ((([Zoznam hercov].národnosť) Like "česká") AND (([Zoznam hercov].stav) Is Null)) OR ((([Zoznam hercov].národnosť) Like "")) OR ((([Zoznam hercov].vek) Not In (42)));
```

Výsledkom zápisu SQL kódu je zoznam hercov, ktorých *národnosť* je česká a súčasne nemajú v poli *stav* zadanú žiadnu textovú hodnotu, alebo zoznam tých hercov, ktorí majú v poli *národnosť* zadaný reťazec nulovej dĺžky alebo ich *vek* nie je rovný hodnote 42, čiže majú akýkoľvek vek, len nie 42 rokov.

Dotaz - zložená podmienka vo Where

meno	národnosť	stav	vek
Lasica		ženatý	45
Donutil	česká		70
Dančiak	slovenská	ženatý	68
Kramár	česká	slobodný	32
Lasica	česká	slobodná	32
Pavlíková	slovenská	slobodná	53
Dančiak	slovenská		75

Obrázok 4.1: Výsledok dopytu pri vyhodnotení niekoľkých, súčasne zadaných podmienok.

Jazyk SQL ponúka pri definovaní výberových podmienok taktiež veľmi zaujímavý **predikát Between**, **ktorý vyhľadá interval hodnôt**. Tento interval je pritom uzavretý, takže jeho krajné hranice sú do výsledku dopytu taktiež zahrnuté. Ukážku predikátu Between zachytáva nasledujúca ukážka kódu, kde pomocou neho hľadáme hercov vo veku 45 až 75 rokov.

```
SELECT [Zoznam hercov].meno, [Zoznam hercov].vek
FROM [Zoznam hercov]
WHERE ((([Zoznam hercov].vek) Between 45 And 75));
```

Tu istú podmienku by sme mohli zapísať zložením dvoch čiastkových výrazov:

```
SELECT [Zoznam hercov].meno, [Zoznam hercov].vek
FROM [Zoznam hercov]
WHERE ((([Zoznam hercov].vek)>=45 And ([Zoznam hercov].vek)<=75));
```

Pri definícii výberového dopytu založeného na dvoch zdrojových tabuľkách, ktorých hodnoty sú navzájom závislé, zapisujeme príslušnú reláciu pomocou klauzuly WHERE.

```
SELECT [Užívatelia].[meno], [Vypožičky].[nazov knihy], [Vypožičky].[dátum vrátenia knihy]
FROM Užívatelia, Vypožičky
WHERE Užívatelia.[Cislo preukazky] = Vypožičky.[Meno uzivatela];
```

V tejto ukážke SQL kódu sa vyberie *meno* užívateľa, *názov knihy* a *dátum vrátenia knihy* z príslušných tabuliek (*Užívatelia*, *Vypožičky*) tak, aby sa zároveň použila relácia podľa čísla preukážky.

Agregačné, skupinové funkcie *Count, Avg, Sum, Min, Max* v príkaze SELECT pri svojom výpočte zlučujú väčšie množstvo riadkov dát do jedného výsledku. V nasledujúcom príklade tak pomocou agregačných funkcií *vypočítame priemerný vek* hercov v databáze, *hmotnosť najľahšieho herca* a zároveň vypíšeme *celkový počet hercov* v našej databáze. Pretože v návrhu dopytu chýba klauzula GROUP BY na zoskupenie dát, tvorí celá tabuľka *Zoznam hercov* jednu veľkú skupinu, a výsledná množina tak obsahuje jediný riadok dát.

```
SELECT Round(Avg([vek]),2) AS [Priemerný vek hercov], Min([hmotnosť]) AS [Hmotnosť
najľahšieho herca], Count([Identifikácia]) AS [Počet hercov v databáze]
FROM [Zoznam hercov];
```

K stĺpcu s priemernou hodnotou veku hercov sme ešte doplnili volanie funkcie *Round*, ktorá výsledok zaokrúhli na dve desatinné miesta. Poznamenajme ale, že *Round* nie je agregačná funkcia – ide o funkciu, ktorá iba zaokrúhli hodnotu jedného stĺpca, či iného výrazu. Volaniu určitej funkcie nad výsledkom inej funkcie hovoríme tzv. vnorené volanie funkcií.

Klauzula GROUP BY

Klauzula GROUP BY je ekvivalentom súhrnov v návrhovom zobrazení dopytu. **Pomocou nej zoskupíme rovnaké záznamy**. Pokiaľ použijeme v dopyte agregačnú funkciu, potom táto agregačná funkcia vypočíta súhrnnú hodnotu.

```
SELECT meno, Count(*) AS [Počet menovcov] FROM [Zoznam hercov] GROUP BY priezvisko;
```

Tento riadok programového SQL kódu zoskupí hercov s rovnakým priezviskom, a pre každé priezvisko vypočíta počet výskytov tohto priezviska medzi všetkými hercami.

Pre výpočet priemerného veku (agregačná funkcia *Avg*) hercov v jednotlivých skupinách národností by zápis SQL kódu vyzeral nasledovne:

```
SELECT [Zoznam hercov].národnosť, Avg([Zoznam hercov].vek) AS [Priemerný vek v každej národnosti]
FROM [Zoznam hercov]
```

```
GROUP BY [Zoznam hercov].národnosť;
```

Klauzula HAVING

Klauzula HAVING je čiastočne podobná klauzule WHERE, pretože taktiež obmedzuje počet vybraných záznamov. **Používa sa v súvislosti s agregačnými funkciami a klauzulou GROUP BY**. Klauzula HAVING **obmedzuje už zoskupené záznamy**.

Postup vyhodnotenia príkazu je nasledovný: pri vyhodnotení príkazu sa najskôr odfiltrujú záznamy pomocou WHERE, tie sa zoskupia pomocou GROUP BY, a následne sa z týchto zoskupených záznamov vyberú len niektoré - podľa podmienky v klauzule HAVING.

```
SELECT [Zoznam hercov].meno, Count(*) AS [Počet hercov s týmto menom]
FROM [Zoznam hercov]
WHERE ((([Zoznam hercov].meno) Like "*a*"))
GROUP BY [Zoznam hercov].meno
HAVING (((Count(*))>2));
```

```
SELECT [Tabulka Tovar].[Druh tovaru], Avg(Cena)
FROM [Tabulka Tovar]
GROUP BY [Tabulka Tovar].[Druh tovaru]
HAVING (Avg(Cena)>100);
```

Výsledkom dopytu bude priemerná cena tovaru podľa jeho druhu, ale len tovaru, ktorého priemerná cena je vyššia ako 100.

Klauzula ORDER BY

Podobne ako v Microsoft Access, nie je ani v štandardom jazyku SQL zaručené určité poradie, v ktorom by sa vracali výsledky dopytu, ibaže by sme toto poradie vyslovene nadefinovali v návrhu dopytu. V jazyku SQL zaistíme poradie pre zoradenie záznamov výsledku dopytu pomocou oddeľovacích čiarok v zozname stĺpcov za kľúčovým slovom ORDER BY.

```
SELECT [Zoznam hercov].meno, [Zoznam hercov].vek
FROM [Zoznam hercov]
ORDER BY [Zoznam hercov].meno DESC;
```

Dopyt vypíše *meno* a *vek* hercov z tabuľky *Zoznam hercov,* a zoradí ich podľa mena zostupne (DESC).Východiskovým typom zoradenia je v každom stĺpci vzostupné poradie, ktoré môžeme taktiež v jazyku SQL definovať, a to explicitným zápisom kľúčového slova ASC.

Namiesto názvov stĺpcov môžeme v syntaxi príkazu SELECT uviesť taktiež ich relatívnu pozíciu vo výsledku dopytu. Táto číselná pozícia nemá ale nič spoločné s pozíciou stĺpca v zdrojovej tabuľke. Uvedená alternatíva je vo formálnom SQL skôr „trpená“, pretože pokiaľ tvar dopytu niekedy neskôr zmeníme, môžeme tým veľmi ľahko poprehadzovať aj poradie stĺpcov v zozname SELECT, a zoradenie dát bude následne fungovať nesprávne.

Ak predpokladáme, že poradie polí vo výsledku dopytu je *meno, priezvisko, vek* a *mesto*, nasledujúci zápis vypíše všetky záznamy z tabuľky *Zoznam hercov*, pričom ich zoradí podľa druhého stĺpca (2), čiže *priezvisko* vzostupne (ASC), a potom podľa *veku* zostupne (DESC).

```
SELECT [Zoznam hercov].meno, [Zoznam hercov].priezvisko, [Zoznam hercov].vek, [Zoznam hercov].mesto
FROM [Zoznam hercov]
```

```
ORDER BY 2, [Zoznam hercov].mesto DESC;
```

Predikáty ALL, DISTINCT, DISTINCROW, TOP n zapisujeme pri dodržaní syntaxe **za príkaz SELECT** a ich význam je nasledujúci:

ALL – východisková hodnota. Hovorí, že príkazom SELECT **budeme vyberať všetky záznamy**.

DISTINCT – **vyberie jedinečné záznamy** (do úvahy sa berú iba zobrazené dáta). Vo výsledku dopytu sa teda neobjaví žiadny riadok, ktorý by bol rovnaký ako iný riadok. Nastavenie má ekvivalent vo voľbe vlastnosti *Jedinečné hodnoty* zo zoznamu vlastností na hodnotu áno.

DISTINCTROW – **vyberie jedinečné záznamy.** Na rozdiel od predchádzajúcej voľby sa do jedinečnosti započítavajú aj ostatné stĺpce vybraných tabuliek, aj tých, do výsledku dopytu nezaradených, a teda pri spustení nezobrazených. Nastavenie má ekvivalent vo voľbe vlastnosti *Jedinečné záznamy* zo zoznamu vlastností na hodnotu áno.

TOP n – **vyberie n najvyšších hodnôt**. Tieto hodnoty sú určené utriedením, a môže to teda byť n najnižších hodnôt. Údaj je možné zadať taktiež v percentách (**TOP n PERCENT**).

```
SELECT Vypozicky.[Cislo preukázky] FROM Vypozicky;
```

```
SELECT All Vypozicky.[Cislo preukázky] FROM Vypozicky;
```

Obidva zápisy sú ekvivalentné. Vypíšu zoznam čísel preukážok všetkých tých užívateľov, ktorí majú vypožičanú knihu – v tabuľke *Výpožičky* je o tom záznam. V prípade, že má niekto viac výpožičiek, zobrazí sa číslo preukážky toľkokrát, koľko má výpožičiek.

```
SELECT DISTINCT Vypozicky.[Cislo preukázky] FROM Vypozicky
```

Zápis vypíše zoznam čísel preukážok užívateľov, ktorí majú vypožičanú knihu (v tabuľke *Výpožičky* je o tom záznam). V prípade, že má niekto viac výpožičiek, zobrazí sa číslo preukážky iba jedenkrát.

```
SELECT TOP 10 Vypozicky.[Cislo preukázky] FROM Vypozicky
ORDER BY Vypozicky.[Datum vypožicky];
```

Zápis vypíše zoznam čísel preukážok užívateľov, ktorí majú vypožičanú knihu (v tabuľke *Výpožičky* je o tom záznam). V prípade, že má niekto viac výpožičiek, zobrazí sa číslo preukážky toľkokrát, koľko má výpožičiek. Dopyt však vypíše iba prvých 10 preukážok s najnižším (najstarším) dátumom výpožičky.

- **Definovanie relácií prostredníctvom klauzuly WHERE**

Ako už bolo spomenuté v jednom z príkladov kapitoly 3.1 Formulárové dopyty nad databázou, **musíme v dopyte, ktorý má vrátiť dáta z viacej ako jednej tabuľky, zadefinovať spojenie tabuliek**. **V jazyku SQL to znamená uviesť jednotlivé tabuľky do čiarkami oddeleného zoznamu v klauzule FROM príkazu SELECT.** Samotný jazyk SQL nás ale nijako neupozorní, že by sme mu mali uviesť, akým spôsobom riadky dát z tabuliek prepojiť. Pokiaľ na definíciu tejto podmienky zabudneme, dostaneme karteziánsky súčin.

Uvažujme, že máme zoznam 15 užívateľov (tabuľka *Užívatelia*) a 20 záznamov o vypožičaných knihách v knižnici (tabuľka *Výpožičky*). Uvedené tabuľky *Užívatelia* a *Výpožičky* sú pritom v relácii 1:N. Výsledkom dopytu pri zápise kódu však bude zoznam 300 záznamov.

```
SELECT [Užívatelia].[meno], [Vypožičky].[nazov knihy], [Vypožičky].[dátum vrátenia knihy]
FROM Užívatelia, Vypožičky;
```

Ako sme teda celkom normálnym spojením užívateľov a výpožičiek kníh mohli dostať výstup s 300 záznamami? Odpoveď je jednoduchá: zabudli sme do klauzuly WHERE uviesť špecifikáciu spojenia

tabuliek, a relačný databázový systém vytvoril preto karteziánsky súčin obidvoch tabuliek. To znamená, že spojil každého užívateľa s každou jednotlivou výpožičkou, a vyrobil tak 15 * 20 = 300-riadkový výsledok dopytu.

Chybu z predchádzajúceho príkladu opravíme takým spôsobom, že do dopytu doplníme príslušnú klauzulu WHERE. Podľa nej už databázový systém zobrazí len tie záznamy, kde sa hodnota stĺpca *Meno užívateľa* tabuľky *Výpožičky* (cudzí kľúč) rovná hodnote stĺpca *Číslo preukážky* tabuľky *Užívatelia* (primárny kľúč). Inak povedané, zobrazia sa reálne vypožičané knihy a k nim mená užívateľov, ktorí si danú knihu skutočne vypožičali (počítajme pritom aj so situáciou, že užívateľ si mohol vypožičať viac kníh, ako aj so situáciou, že jedna a tá istá kniha je vypožičaná jednému užívateľovi dvakrát – duplicitná výpožička). Nasledujúcim zápisom SQL kódu teda dostaneme rozumnejší výsledok dopytu, ktorý teraz pozostáva len z dvadsiatich riadkov.

```
SELECT [Užívatelia].[meno], [Vypožičky].[nazov knihy], [Vypožičky].[dátum vrátenia knihy]
FROM Užívatelia, Vypožičky
WHERE Užívatelia.[Cislo preukazky]=Vypožičky.[Meno uzivatela];
```

Kód realizuje štandardné, vnútorné spojenie tabuliek *Užívatelia* a *Výpožičky*. Do výsledku sa vrátili len tie riadky, kde k vybranej knihe bol nájdený užívateľ, ktorý si túto knihu vypožičal. Vnútorné spojenie medzi tabuľkami je možné, okrem klauzuly WHERE, vytvoriť aj pomocou operácie INNER JOIN (pri použití operácie INNER JOIN bude dopyt rýchlejší).

➢ Operácia INNER JOIN

Pomocou operácie INNER JOIN **je možné nadefinovať reláciu medzi tabuľkami**. **Pôjde o vnútornú reláciu, čo znamená, že v dopyte sa zobrazia len tie záznamy, v ktorých kľúč použitý pri tvorbe spojenia tabuliek sa nachádza v obidvoch tabuľkách súčasne.**

Syntax:

```
SELECT Tabulka1.Pole1, Tabulka1.Pole2, Tabulk2.Pole1 FROM Tabulka1 INNER JOIN
Tabulka2  ON Tabulka1.primárny_kľúč  = Tabulka2.cudzí_kľúč;
```

Uveďme si jednoduchú ukážku z databázy zoznamu hercov a hier, v ktorých herci hrajú:

```
SELECT [Zoznam hier].nazov_hry, [Zoznam hier].premiera, [Zoznam hercov].meno
FROM [Zoznam hier] INNER JOIN [Zoznam hercov] ON [Zoznam hier].ID_hry = [Zoznam hercov].ID_hry;
```

Výsledkom dopytu bude zoznam všetkých hercov z tabuľky *Zoznam hercov*, ktorí majú v poli *ID_hry* uvedené číslo hry, v ktorej hrajú. K menu herca budú zobrazené polia *názov hry* a *premiéra* z tabuľky *Zoznam hier*. Herci, ktorí nehrajú v žiadnej hre (t.j. v stĺpci *ID_hry* tabuľky *Zoznam hercov* nie je uvedené žiadne číslo hry), vo výsledku dopytu zobrazení nebudú.

➢ Operácia LEFT JOIN, RIGHT JOIN

Pomocou operácie LEFT JOIN a RIGHT JOIN **môžeme realizovať vonkajšie spojenie medzi tabuľkami.** To znamená, že **z jednej tabuľky budú zobrazené všetky záznamy a k nej zodpovedajúce záznamy z druhej tabuľky, pokiaľ existujú.**

Syntax:

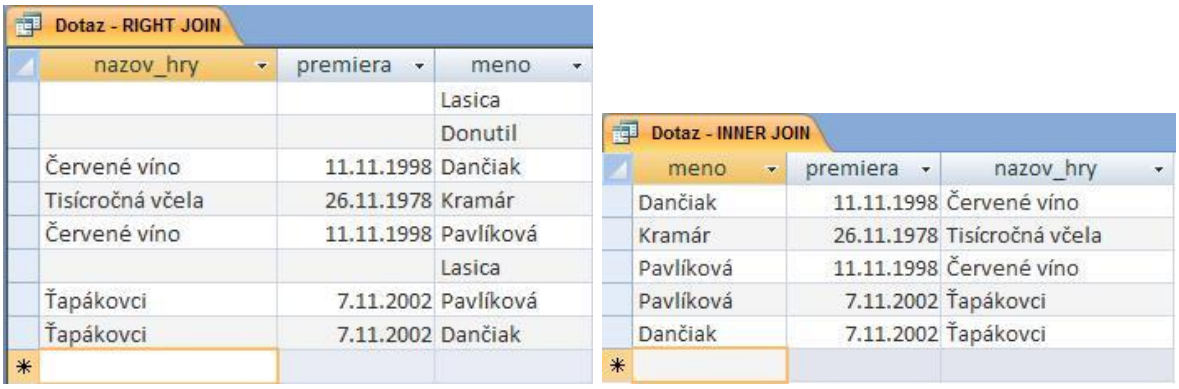
```
SELECT Tabulka1.Pole1, Tabulka1.Pole2, Tabulk2.Pole1 FROM Tabulka1 RIGHT JOIN
Tabulka2  ON Tabulka1.primárny_kľúč  = Tabulka2.cudzí_kľúč;
```

```
SELECT [Zoznam hier].nazov_hry, [Zoznam hier].premiera, [Zoznam hercov].meno
```

```
FROM [Zoznam hier] RIGHT JOIN [Zoznam hercov] ON [Zoznam hier].ID_hry =
[Zoznam hercov].ID_hry;
```

Výsledkom dopytu bude zoznam všetkých hercov z tabuľky *Zoznam hercov* a pokiaľ majú herci v tejto tabuľke v poli *ID_hry* uvedené číslo hry, v ktorej hrajú, budú zobrazené polia *názov hry* a *premiéra* z tabuľky *Zoznam hier*. Pokiaľ herec v žiadnej hre nehrá, t.j. v stĺpci ID_hry tabuľky *Zoznam hercov* nie je uvedené žiadne číslo hry, bude v stĺpcoch *názov hry* a *premiéra* hodnota Null.

Výsledky zobrazenia dopytu pri aplikovaní vnútorného a vonkajšieho prepojenia medzi tabuľkami približuje nasledujúci obrázok (Obrázok 4.2).



Dotaz - RIGHT JOIN

nazov_hry	premiera	meno
		Lasica
		Donutil
Červené víno	11.11.1998	Dančiak
Tisícročná včela	26.11.1978	Kramár
Červené víno	11.11.1998	Pavlíková
		Lasica
Ťapákovci	7.11.2002	Pavlíková
Ťapákovci	7.11.2002	Dančiak

Dotaz - INNER JOIN

meno	premiera	nazov_hry
Dančiak	11.11.1998	Červené víno
Kramár	26.11.1978	Tisícročná včela
Pavlíková	11.11.1998	Červené víno
Pavlíková	7.11.2002	Ťapákovci
Dančiak	7.11.2002	Ťapákovci

Obrázok: 4.2: Výsledok zobrazenia dopytu pri vnútornom (vľavo) avonkajšom (vpravo) spojení (RIGHT JOIN).

- **Deklarácia PARAMETERS**

Pre dopyty spúšťané pravidelne môžeme **použiť deklaráciu PARAMETERS na vytvorenie parametrického dopytu**. Parametrický dopyt pomáha automatizovať proces zmeny kritérií dopytu. Kritériá predstavujú zadané podmienky, ktoré určujú záznamy zahrnuté do množiny výsledkov dopytu alebo filtra. Pomocou parametrického dopytu bude kód musieť poskytnúť parametre vždy, keď sa spustí dopyt.

Syntax:

```
PARAMETERS [názov parametra] dátový typ;
```

Počas spustenia dopytu aplikáciou Access môžeme použiť názov parametra ako reťazec, ktorý je zobrazený v dialógovom okne. Na uzavretie textu, ktorý obsahuje medzery (je zložený z viacerých slov), alebo obsahuje interpunkciu, používame operátor zátvorky ([ ]). Ak deklarácia zahŕňa viacero parametrov, oddeľujeme ich čiarkami. Nasledovný príkaz PARAMETERS zahŕňa dva parametre:

```
PARAMETERS [Zadaj hodnotu platu] Currency, [Zadaj hodnotu veku] Number;
```

Deklarácia PARAMETERS je voliteľná, ak sa však zahrnie, predchádza všetky iné príkazy vrátane príkazu SELECT.

Nasledujúci príklad predpokladá, že sa poskytnú dva parametre, ktoré sa následne aplikujú ako kritériá na záznamy dát vychádzajúcich z dvoch spojených tabuliek *Zoznam zamestnancov* a *Oddelenia*.

```
PARAMETERS [Zadaj hodnotu platu] Currency, [Zadaj názov oddelenia] Text ( 255 );
SELECT [Zoznam zamestnancov].meno, [Zoznam zamestnancov].priezvisko,
Oddelenia.[nazov oddelenia], [Zadaj hodnotu platu] AS [výška platu zamestnanca]
FROM Oddelenia INNER JOIN [Zoznam zamestnancov] ON Oddelenia.[nazov oddelenia] =
[Zoznam zamestnancov].oddelenie
```

```
WHERE (((Oddelenia.[nazov oddelenia])=[Zadaj názov oddelenia]) AND (([Zadaj hodnotu
platu])<[plat]));
```

- **Vnorený dopyt SELECT**

Veľmi silným prvkom jazyka SQL je tzv. **vnorený dopyt** alebo – inak nazývaný aj – **poddopyt**. Ak už názov napovedá, **ide o príkaz SELECT jazyka SQL, ktorý je obsiahnutý v inom, nadriadenom dopyte SQL (SELECT). Vnorené dopyty využijeme tam, kde potrebujeme najprv zistiť nejakú informáciu, a v závislosti od nej zistiť následne ďalšie informácie.**Podľa syntaktických pravidiel jazyka SQL musí byť poddopyt zapísaný do zátvoriek. Určíme tým poradie vykonávania jednotlivých dopytov.

Varianty postupov tvorby dopytov dopytov:

- vytvorenie nadriadeného dopytu spolu s poddopytom priamo v kóde jazyka SQL;
- SQL dopyt je možné zadať ako podmienku v návrhovom zobrazení hlavného dopytu v položke *Kritériá*.

Pomocou vnoreného dopytu SELECT sa dá riešiť veľa situácií, ktoré by sme inak museli riešiť použitím niekoľkých bežných dopytov, vnorených kaskádovo do seba, čo vo väčšine prípadov znižuje celkovú prehľadnosť zápisu. V niektorých prípadoch by to dokonca nešlo obísť ani týmto spôsobom, alebo len veľmi zložito. Príkladom by mohol byť výsledok získaný sprievodcom na vyhľadanie duplicitných položiek.

```
SELECT Vypožičky.[nazov knihy], Vypožičky.[Meno uzivatela], Vypožičky.[Dátum vrátenia knihy],
Knihy.[vydavateľstvo]
FROM Knihy INNER JOIN Vypožičky ON Knihy.ID = Vypožičky.[nazov knihy]
WHERE (((Vypožičky.[nazov knihy]) IN
(SELECT [nazov knihy] FROM [Vypožičky] As Tmp GROUP BY [nazov knihy] HAVING Count(*)>1 )))
ORDER BY Vypožičky.[nazov knihy];
```

Dopyt vráti záznamy kníh (*názov knihy, meno užívateľa, dátum vrátenia knihy a názov vydavateľstva*) duplicitne vypožičaných rôznym užívateľom, pričom duplicitu zisťuje práve vnorený dopyt SELECT.

Súčasťou predchádzajúceho zápisu kódu je aj **operátor IN**, ktorý **slúži pre jednoduché porovnanie, či sa hodnota stĺpca vľavo vyskytuje medzi hodnotami vrátenými vnoreným dopytom.**

Keby sme napríklad chceli zobraziť názvy kníh (doplnené o mená a priezviská ich autorov) vydaných v rokoch 2001 až 2003, mohli by sme taký dopyt pomocou operátora IN zapísať takto:

```
SELECT názov, meno, priezvisko
FROM Knihy, Autori
    WHERE Knihy.ID = Autori.[ID autora]
    AND [rok vydania] IN (2001, 2002, 2003)
```

Výraz v zátvorke za operátorom IN je množina všetkých hodnôt, s ktorými sa má daná hodnota porovnávať.

Vo väčšine prípadov neuvádzame zoznam hodnôt explicitne, ale vracia nám ich práve definovaný vnorený dopyt. V tejto chvíli si teda môžeme uviesť náš dopyt na názvy kníh vydaných v tých istých rokoch ako napríklad diela autora Sama Chalupku:

```
SELECT názov, meno, priezvisko, vydavateľstvo, Knihy.cena
FROM Knihy, Autori, Vydavateľstvo
WHERE (((Knihy.vydavateľstvo)=[Vydavateľstvo].[ID])
    AND ((Knihy.autor)=[Autori].[ID autora])
```

```
    AND ((Knihy.[rok vydania]) In (SELECT [rok vydania]
      FROM Knihy k, Autori a
      WHERE a.[ID autora] = k.autor
      AND a.meno LIKE 'Samo'
      AND a.priezvisko LIKE 'Chalupka'
)));
```

Všimnite si, že sme vo vnorenom dopyte v časti FROM použili aliasy pre názvy tabuliek (*Knihy k, Autori a*). To preto, že tie isté názvy sa objavujú aj v nadradenom SELECTe. Ak by sme v tom to príklade aliasy nepoužili, chyba by to nebola, dopyt by vrátil očakávané dáta. Niekedy je však použitie aliasov nutné, preto že v niektorých prípadoch bude SQL server potrebovať vedieť, či sa v obmedzení vnoreného dopytu odkazujeme na tabuľku lokálnu alebo globálnu.

Pre prípad definovania vnoreného dopytu budeme opäť pracovať s tabuľkou *Užívateľa* a *Výpožičky*, pričom cieľom dopytu v nasledujúcom príklade je vrátiť názov najstaršej vypožičanej knihy, ktorej názov neobsahuje slovo Flash. Úlohu zrealizujeme operátorom NOT IN s definovaním poddopytov v hlavnom dopyte.

Logický operátor NOT predstavuje negáciu. Týmto spôsobom získame množinu záznamov, ktoré nevyhovujú daným podmienkam. Na ich základe budeme môcť následne zvoliť tie záznamy, ktoré potrebujeme.

Nasledujúci jednoduchý dopyt vracia najdlhšie vypožičanú knihu:

```
(SELECT Min(Vypožičky.[dátum vrátenia knihy]) FROM Vypožičky)
```

Výsledok nasledujúceho jednoduchého dopytu vracia všetky záznamy kníh, ktorých názov obsahuje slovo Flash:

```
(SECECT Vypožičky.[nazov knihy] FROM [Vypožičky] WHERE Vypožičky.[nazov knihy]
Like "*Flash*")
```

Ďalším krokom je získanie množiny všetkých záznamov, ktoré nie sú obsiahnuté vo výsledkoch predchádzajúcich dvoch dopytoch. Tieto dva dopyty budú totiž kritériami hlavného dopytu:

```
NOT IN (SELECT Min(Vypožičky.[dátum vrátenia knihy]) FROM Vypožičky)


NOT IN (SELECT Knihy.[názov] FROM [Knihy] WHERE Knihy.[názov] Like "Flash") Not (Like "Flash")
```

Kompletný SQL kód spolu s výslednou podmienkou, definovanou v klauzule WHERE, zachytáva výpis:

```
SELECT Užívatelia.priezvisko, Knihy.názov AS [názov vypožičanej knihy], Vypožičky.[dátum
vrátenia knihy]
FROM Užívatelia INNER JOIN (Knihy INNER JOIN Vypožičky ON Knihy.ID =
Vypožičky.[nazov knihy]) ON Užívatelia.[Cislo preukazky] = Vypožičky.[meno uzivatela]
WHERE (((Knihy.názov) NOT IN (SELECT Knihy.[názov] FROM [Knihy] WHERE
Knihy.[názov] Like "Flash")) AND ((Vypožičky.[dátum vrátenia knihy])=(SELECT
min(Vypožičky.[dátum vrátenia knihy]) FROM Vypožičky)))
ORDER BY Knihy.názov;
```

V jednoduchých vnorených dopytoch nám obdobným spôsobom budú figurovať aj ostatné agregačné funkcie.

Aplikovanie poddopytov do nadriadených dopytov nám rozširuje klasickú množinu relačných operátorov o **operátory ďalšie, ktoré sa aplikujú z ľavej strany na stĺpec, a z pravej strany na vnorený dopyt vracajúci viac hodnôt. Patria sem operátory IN, ANY (SOME), ALL.** Operátor IN sme si už na príklade demonštrovali, použitie operátorov ANY(SOME) a **ALL** ukazuje nasledujúca syntax:

```
stĺpec relačný_operátor ANY|SOME|ALL (vnorený SELECT);
```

Operátory ANY alebo SOME určujú, že sa relácia vzťahuje na aspoň jednu z hodnôt, ktorú vráti vnorený dopyt, operátor ALL aplikuje reláciu na všetky hodnoty vrátené poddopytom.

V nasledujúcom príklade by sme chceli dopytom zobraziť zoznam všetkých takých kníh od Martina Kukučína, ktorých cena nie je vyššia ako cena ktorejkoľvek knihy od Ľubomíra Feldeka.

```
SELECT názov, autor, vydavateľstvo, cena
FROM Knihy, Autori, Vydavateľstvo
WHERE Knihy.autor = Autori.[ID autora]
  AND Knihy.vydavateľstvo = Vydavateľstvo.ID
  AND meno LIKE 'Samo'
  AND priezvisko LIKE 'Chalupka'
  AND cena < ALL
   (SELECT cena
       FROM Knihy, Autori, Vydavateľstvo
       WHERE Knihy.autor = Autori.[ID autora]
       AND Knihy.vydavateľstvo = Vydavateľstvo.ID
       AND meno LIKE 'Jonáš'
       AND priezvisko LIKE 'Záborský'
     );
```

Hore uvedený dopyt môžeme prepísať použitím operátoru ANY v kombinácii s operátorom NOT (rozdiel bude len v riadku, kde uvádzame stĺpec CENA):

```
...
  AND NOT cena > ANY (SELECT cena ...
...
```

6.3 Jazyk pre manipuláciu s dátami

- **Príkaz SELECT... INTO**

Príkaz **je obdobou vytváracieho dopytu**, to znamená, že vkladá dáta do novej tabuľky.

Syntax:

```
SELECT Pole INTO Tabulka_nova FROM Tabulka_stara
```

```
SELECT Autori.* INTO Autori_zaloha
FROM Autori;
```

Tento zápis vytvorí kompletnú zálohu s názvom *Autori_zaloha* tabuľky *Autori*. Nastavenie štruktúry sa preberie z pôvodnej tabuľky *Autori*.

- **Príkaz INSERT... INTO**

Príkaz je obdobou pridávacieho dopytu, to znamená, že **dokáže pridať ďalšie záznamy do určenej tabuľky**. Príkaz INSERT má dve podoby. Prvá priamo obsahuje požadované hodnoty stĺpcov a druhá tieto hodnoty vyberá príkazom – poddopytom SELECT z inej tabuľky. Na obidva spôsoby vkladania dát sa pozrieme podrobnejšie:

a) Príkaz INSERT s klauzulou VALUES

Syntax:

```
INSERT INTO Tabulka (<Polia>)
VALUES (Hodnoty)
```

V príkaze INSERT s klauzulou VALUES **môžeme pridať jeho volaním iba jeden riadok dát (jeden záznam)**, pretože viac dátových hodnôt nie je možné do tohto príkazu zadať. V časti (<Polia>) píšeme čiarkami oddelený zoznam stĺpcov. Tento zoznam je nepovinný, ale pokiaľ ho v príkaze uvedieme, musí byť vždy zapísaný do zátvoriek. V prípade, že zoznam stĺpcov vynecháme, musíme hodnoty stĺpcov uviesť v správnom poradí, teda presne v tom poradí, v akom sú jednotlivé stĺpce fyzicky usporiadané v tabuľke. To ale taktiež znamená, že príkaz prestane fungovať v prípade, že niekto do tabuľky pridá nový stĺpec (aj keď jeho vypĺňanie nie je povinné), alebo – akonáhle stĺpec odstráni alebo modifikuje. Z uvedených dôvodov je vhodné zoznam stĺpcov zapisovať vždy explicitne, aj keď je s takým príkazom o trochu viac práce. Za zoznamom stĺpcov už nasleduje kľúčové slovo VALUES a zoznam hodnôt zapisovaných do jednotlivých stĺpcov. Aj tento zoznam je oddelený čiarkami, a taktiež musí byť uvedený v zátvorkách. Položky v zozname hodnôt VALUES musia jednoznačne zodpovedať zoznamu stĺpcov v príkaze, prípadne zoznamu stĺpcov v tabuľke alebo v dopyte (pokiaľ sme zoznam stĺpcov explicitne neuviedli).

Nasledujúci zápis príkazu INSERT pridá do tabuľky *Zoznam režisérov* hodnoty polí *ID, meno* a *vek* režiséra Bednárik.

```
INSERT INTO [Zoznam režisérov] (ID, meno, vek)
VALUES (1, "Bednárik", 75);
```

Po spustení SQL kódu s príkazom INSERT sa zobrazí dialógové okno s upozornením na spustenie dopytu s informáciou o pripojení jedného riadku do cieľovej tabuľky. Kliknutím na tlačidlo *Áno* pokračujte ďalej. Pokiaľ sa následne pozriete na dáta tabuľky *Zoznam režisérov*, uvidíte, že bol do nej pridaný záznam so všetkými súvisiacimi dátami – s *ID* herca, *menom* a *vekom* herca (Obrázok 4.3).

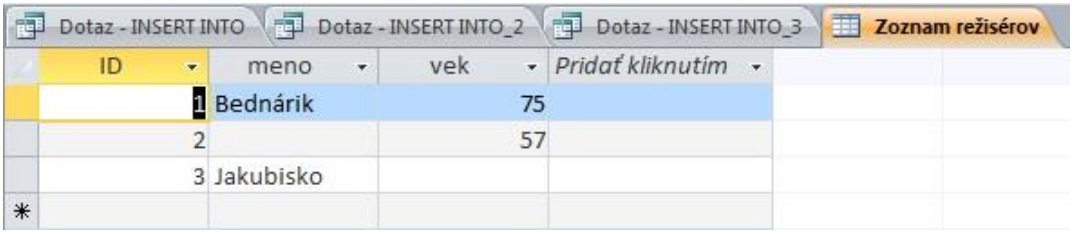
Do cieľovej tabuľky môžeme taktiež doplniť nový záznam s tým, že neuvedieme hodnoty pre každý stĺpec. V takom poradí je však podmienkou uviesť v zátvorke zoznam názvov tých stĺpcov, do ktorých údaje chceme pridať. V prvom prípade sa doplnia hodnoty len do prvého (ID) a tretieho (vek) stĺpca v poradí,

```
INSERT INTO [Zoznam režisérov] (ID, vek)
VALUES (2, 57);
```

a v druhom prípade len do - v poradí prvých dvoch stĺpcov (ID, meno):

```
INSERT INTO [Zoznam režisérov] (ID, meno)
VALUES (3, "Jakubisko");
```

Dátové zobrazenie dopytu po aplikovaní predchádzajúcich troch príkazov INSERT dokumentuje obrázok 4.3.



Obrázok 4.3: Výsledok dopytu pre príkazy INSERT s klauzulou VALUES.

b) Príkaz INSERT s poddopytom (SELECT)

Pomocou príkazu INSERT s poddopytom (vnoreným dopytom SELECT) **vytvoríme podľa každého riadku, načítaného zo zdrojovej tabuľky, alebo dopytu nový riadok v cieľovej tabuľke**. Poddopyt nám tak dáva dohromady údaje, ktoré potom vložíme do novej, cieľovej tabuľky.

Syntax:

```
INSERT INTO Tabulka_1 ( Polia_1)
SELECT  (Polia_2)
FROM    Tabulka_2;
```

Pred spustením tohto kódu je potrebné najprv vytvoriť príkazom CREATE TABLE pomocnú tabuľku *Slovenskí režiséri*.

```
INSERT INTO [Zoznam režisérov] (ID, meno, vek)
SELECT ID, meno, vek
FROM [Slovenskí režiséri];
```

Rovnakým príkazom INSERT je možné vkladať dáta do tabuliek taktiež aj z dopytu, ale za splnenia dvoch podmienok:

- **V prípade, že je v dopyte spojených viac tabuliek**, musí sa príkaz INSERT odvolávať iba na stĺpce z jednej a tej istej tabuľky. Inými slovami, **príkazom INSERT je možné vkladať dáta vždy iba do jednej tabuľky.**
- **Dopyt musí obsahovať všetky povinné stĺpce z podkladovej tabuľky**. V prípade, že tabuľka obsahuje niektoré stĺpce s definovaným obmedzením NOT NULL, ktoré ale nie sú do dopytu zahrnuté, nemôžeme prostredníctvom dopytu definovať hodnoty týchto stĺpcov, a preto ani nemôžeme vložiť nový riadok dát.

Nasledujúci SQL kód príkazu pre vloženie nových záznamov používa ako zdroje dát výsledky dopytu definovaného zo spojených tabuliek. V prípade, že chceme množinu dát pre export obmedziť – v našom prípade by sme chceli vložiť do tabuľky *Vybrané výpožičky* len knihy s cenou od 100 do 200 eur, zapíšeme túto podmienku pre zodpovedajúci stĺpec do klauzuly WHERE.

```
INSERT INTO [Vybrané výpožičky] ([Priezvisko autora], [Názov knihy], [Cena knihy], [Priezvisko užívateľa])
SELECT Autori.priezvisko, Knihy.názov, Knihy.cena, Užívatelia.meno
FROM Autori INNER JOIN (Užívatelia INNER JOIN (Knihy INNER JOIN Vypožičky ON Knihy.ID = Vypožičky.[nazov knihy]) ON Užívatelia.[Cislo preukazky] = Vypožičky.[meno uzivatela]) ON Autori.[ID autora] = Knihy.autor
WHERE (((Knihy.cena) Between 100 And 200));
```

➢ Príkaz UPDATE

Príkaz je obdobou aktualizačného dopytu, čo znamená, že **dokáže hromadne zmeniť (aktualizovať) dátové hodnoty v danom poli, resp. poliach tabuľky alebo dopytu**.

Syntax:

```
UPDATE Tabulka SET Pole=hodnota
```

V syntaxi príkazu uvádzame stĺpce, ktorých hodnoty budú aktualizované, a môžeme do príkazu zapísať taktiež klauzulu WHERE, ktorá obmedzí rozsah aktualizovaných riadkov. Všeobecne nie je klauzula WHERE v príkaze UPDATE povinná, no bez klauzuly WHERE sa príkaz UPDATE pokúša aktualizovať všetky riadky v danej tabuľke alebo v dopyte. **Pre každý aktualizovaný stĺpec musíme v príkaze UPDATE uviesť klauzulu SET s označením stĺpca a určením jeho novej hodnoty.** Touto novou hodnotou môže byť konštanta, názov iného stĺpca alebo akýkoľvek iný výraz, vyhodnotením ktorého získa jazyk SQL zodpovedajúcu hodnotu daného stĺpca.

```
UPDATE Knihy SET Knihy.cena = [Knihy].[cena] * 1.15
WHERE ((Knihy.[názov])="Access");
```

V tabuľke *Knihy* po spustení dopyt zvýši cenu všetkých kníh s názvom Access o 15 %. Len pripomíname, že v prípade aktualizácie znakových hodnôt ich musíme uvádzať v úvodzovkách.

```
UPDATE Užívatelia SET Užívatelia.meno = "František"
WHERE (((Užívatelia.meno) Like "*er*"));
```

Zápis príkazu UPDATE aktualizuje všetky záznamy z tabuľky *Užívatelia*, ktoré v poli meno obsahujú reťazec „er“.

V prípade, že **sa klauzula SET odvoláva na niekoľko stĺpcov súčasne, musíme názvy stĺpcov a ich hodnoty zapísať do čiarkami oddeleného zoznamu.**

```
UPDATE Knihy SET [Knihy].[autor] = "Tajovsky", [Knihy].[názov] ="Ženský zákon"
WHERE ((Knihy.[cena])=100);
```

- **Príkaz DELETE**

Príkaz je obdobou odstraňovacieho dopytu, to znamená, že **odstraňuje z tabuľky jeden alebo viac určených riadkov dát**. Tento príkaz môže odstraňovať dáta aj z dopytu, ale len za podmienky, že je tento dopyt založený len na jednej tabuľke (čiže z dopytov, ktoré zobrazujú dáta s niekoľkých reláciami spojených tabuliek, nie je možné dáta odstraňovať). V príkaze DELETE nešpecifikujeme žiadne názvy stĺpcov, pretože ním pre vybrané záznamy odstránime vždy dáta všetkých stĺpcov súčasne. **Rozsah odstraňovania obmedzíme opäť pomocou klauzuly WHERE.** V prípade, že klauzulu neuvedieme, pokúsi sa príkaz DELETE vymazať všetky riadky zdrojovej tabuľky.

Syntax:

```
DELETE FROM Tabulka;
```

```
DELETE *
FROM [Zoznam režisérov];
```

Aplikovaním uvedeného SQL kódu budú zmazané všetky záznamy z tabuľky *Zoznam režisérov*, pričom štruktúra tabuľky zostane zachovaná.

```
DELETE *
FROM [Zoznam režisérov]
WHERE (([Zoznam režisérov].meno)="Bednárik");
```

Príkazom DELETE zmažeme z tabuľky *Zoznam režisérov* všetky záznamy s menom “Bednárik“.

```
DELETE * FROM Knihy
 WHERE [Knihy].[cena] > (SELECT Avg([Knihy].[cena]) FROM Knihy);
```

Príkazom zmažeme z tabuľky *Knihy* záznamy, ktoré majú hodnotu stĺpca *cena* väčšiu, ako je priemerná cena zo všetkých kníh nachádzajúcich sa v tabuľke.

6.4 Jazyk pre definíciu dát

Na rozdiel od iných typov dopytov, dopyt definujúci údaje sa nepoužíva na načítanie údajov. **Dopyt definujúci údaje používa jazyk definície údajov** (jazyk definície údajov je súčasťou jazyka štruktúrovaného dopytu SQL) **na odstraňovanie, úpravu alebo opätovné vytváranie databázových objektov** (tabuliek, obmedzení, indexov alebo vzťahov), čiže jednotlivých súčastí databázovej schémy. Použite dopytov definujúcich údaje na úpravu databázových objektov môže byť riskantné, pretože akcie nie sú sprevádzané potvrdzujúcimi dialógovými oknami. Ak urobíme chybu, môžeme stratiť údaje, alebo nevratne zmeniť návrh tabuľky.

- **Príkaz CREATE TABLE**

Pomocou príkazu CREATE TABLE **je možné vytvoriť novú tabuľku, vrátane definície jej štruktúry.**

Syntax:

```
CREATE TABLE Tabulka (Pole1 Dátový_Typ <Velkost> <Not Null>, Pole2 …..);
```

Syntax je asi relatívne intuitívna – **zadávame názov tabuľky, a následne v zátvorke definujeme čiarkami oddelený zoznam jednotlivých polí tabuľky, vrátane ich parametrov** (dátový typ, veľkosť, možnosť zadávať prázdne údaje a ďalšie, tu momentálne neuvedené vlastnosti).

```
CREATE  TABLE [Zoznam režisérov2]
(
ID        Number PRIMARY KEY,
meno      String (25) Not Null,
vek       Number,
[počet predstavení] Number
);
```

V jazyku SQL zapisujeme do definície príslušného stĺpca tabuľky kľúčové slovo Not Null alebo Null. Týmto spôsobom máme možnosť stanoviť, či majú byť v stĺpci povolené alebo zakázané hodnoty Null. V prípade, že špecifikáciu Null/Not Null v kóde SQL vynecháme, prevezme sa (v prípade aplikácie MS Access) východiskové nastavenie Null, ktoré znamená, že stĺpec môže hodnoty Null obsahovať.

- **Príkaz ALTER TABLE**

Príkazom ALTER TABLE **dokážeme zmeniť rôzne aspekty definície štruktúry alebo obmedzení už vytvorenej databázovej tabuľky**. Jeho implementácia sa u rôznych výrobcov značne líši, ale vo všeobecnosti sa dá povedať, že pomocou príkazu ALTER TABLE môžeme zrealizovať nasledujúce typy zmien:

- pridať do tabuľky nové stĺpce (**ADD**);
- úprava (napríklad dátového typu) existujúcich stĺpcov alebo tabuľky (**ALTER**);
- odstrániť z tabuľky existujúce stĺpce (**DROP**);

- zmeniť atribúty fyzického uloženia tabuľky;
- pridať, zmeniť alebo odstrániť obmedzenia (**ADD CONSTRAINT, DROP CONSTRAINT**).

Syntax:

```
ALTER TABLE Tabulka ADD COLUMN Pole Typ <Velkost> <Not Null>
ALTER TABLE Tabulka ALTER COLUMN Pole Typ <Velkost> <Not Null>
ALTER TABLE Tabulka DROP COLUMN Pole
```

```
ALTER TABLE [Zoznam režisérov]
ADD COLUMN [počet predstavení] Number, [počet hlavných úloh] Number
```

Zápis pridá do tabuľky *Zoznam režisérov* polia *počet predstavení*, a *počet hlavných úloh* súčasne nastaví ich dátový typ na Číslo.

```
ALTER TABLE [Zoznam režisérov]
ALTER COLUMN [počet predstavení] Money
```

Zápis zmení dátový typ poľa *počet predstavení* na Mena.

```
ALTER TABLE [Zoznam režisérov]
DROP COLUMN [počet predstavení], [počet hlavných úloh];
```

Zápis odstráni z tabuľky *Zoznam režisérov* polia s názvom *počet predstavení* a *počet hlavných úloh*.

➢ Príkaz CREATE INDEX

Príkaz CREATE INDEX **vytvorí v tabuľke nad jedným alebo viacerými jej stĺpcami index**. O indexoch sme už čo-to spomínali – ponúkajú možnosť rýchleho vyhľadávania dát v tabuľke podľa hodnôt jedného alebo viacerých stĺpcov. Indexy definované nad cudzími kľúčmi naviac výrazne zvyšujú efektivitu operácie spájania tabuliek. Pri vkladaní alebo odstraňovaní dát z databázovej tabuľky a pri zmene hodnôt indexovaného stĺpca zaisťuje potrebnú údržbu indexu automaticky sám relačný databázový systém. Každý index však zaberá nejaké miesto na disku, a taktiež ich údržba je náročná na výpočtové prostriedky.

Príkaz CREATE INDEX má nasledovnú *syntax*:

```
CREATE [UNIQUE] INDEX názov_indexu
 ON tabuľka (pole1 [DESC][, pole2 [DESC], ...])
 [WITH {PRIMARY | DISALLOW NULL | IGNORE NULL}]
```

Jedinými požadovanými prvkami sú príkaz CREATE INDEX, názov indexu, argument ON, názov tabuľky obsahujúcej polia, ktoré chceme indexovať, a zoznam polí, ktoré sa zahrnú do indexu.

- **Argument DESC spôsobí vytvorenie indexu v zostupnom poradí**, čo môže byť užitočné vtedy, keď často spúšťate dopyty, ktoré vyhľadávajú najvyššie zaradené hodnoty v indexovanom poli, alebo ktoré zoraďujú indexované pole v zostupnom poradí. Index sa predvolene vytvára vo vzostupnom poradí.
- **Argument WITH PRIMARY vytvorí indexované pole (alebo polia) ako primárny kľúč** (primárny kľúč: jedno alebo viacero polí (stĺpcov), ktorých hodnoty jednoznačne označujú každý záznam v tabuľke. Primárny kľúč nepovoľuje hodnoty Null a vždy musí mať jedinečný index. Primárny kľúč sa používa na prepojenie tabuľky s cudzími kľúčmi v iných tabuľkách.) tabuľky.

- **Argument WITH DISALLOW NULL spôsobí to, že index bude vyžadovať zadanie hodnoty pre indexované pole**, t. j. hodnoty Null nie sú povolené.

Nasledujúci príklad vytvorí v tabuľke *Zoznam zamestnancov* index v zostupnom poradí nad stĺpcom *Identifikácia* s jedinečnými hodnotami, a súčasne ho definuje ako primárny kľúč tejto tabuľky. **Preto v prípade, že požadujeme, aby všetky hodnoty indexovaného stĺpca boli jedinečné** (znamená to, že v tomto poli nemôžu mať dva záznamy v tabuľke rovnakú hodnotu), zapíšeme medzi kľúčové slová CREATE a INDEX ešte kľúčové slovo **UNIQUE**.

```
CREATE UNIQUE INDEX Názov_definovaneho_indexu
ON  [Zoznam zamestnancov] (Identifikácia DESC)
WITH PRIMARY;
```

Druhou možnosťou je definovať v tabuľke jedinečné obmedzenie, prostredníctvom ktorého vytvoríme taktiež aj jedinečný index – iba ho vyžiadame nepriamo. Jedinečné indexy sú obyčajne efektívnejšie ako nejedinečné.

➢ Klauzula CONSTRAINT – vytváranie obmedzení a vzťahov

Obmedzenia (Constraints) sú veľmi dôležité, pretože **pomocou ich implementácie realizujeme v databáze aplikačné pravidlá.** Obmedzenie vytvorí logickú podmienku, ktorú musí pole alebo kombinácia polí splniť po vložení hodnôt. Napríklad obmedzenie UNIQUE zabraňuje poľu s obmedzením akceptovať hodnotu, ktorá by duplikovala existujúcu hodnotu pre dané pole (z problematiky definovania indexu už vieme, že rezervované slovo UNIQUE sa používa na nastavenie poľa ako jedinečný kľúč).

Vzťah je typom obmedzenia, ktorý odkazuje na hodnoty poľa alebo kombinácie polí v inej tabuľke, s cieľom určiť, či hodnotu možno vložiť do poľa alebo kombinácie polí s obmedzením. Na označenie toho, že obmedzenie je vzťahom, sa nepoužívajú žiadne špeciálne kľúčové slová.

Je veľmi dôležité všetky obmedzenia vhodne pomenovať, pretože názov obmedzenia sa vypisuje do chybovej správy generovanej v prípade porušenia.

Na vytvorenie obmedzenia sa používa klauzula CONSTRAINT v príkaze CREATE TABLE alebo ALTER TABLE. Existujú dva druhy klauzúl CONSTRAINT. Jedna na vytvorenie obmedzenia jedného poľa a ďalšia na vytvorenie obmedzenia pre viaceré polia.

Klauzula CONSTRAINT vzťahujúca sa na jedno pole sa okamžite riadi definíciou poľa, na ktoré sa vzťahuje, a jej *syntax* je nasledovná:

```
CONSTRAINT názov_obmedzenia {PRIMARY KEY | UNIQUE | NOT NULL |
REFERENCES cudzia_tabuľka [(cudzie_pole)]
[ON UPDATE {CASCADE | SET NULL}]
[ON DELETE {CASCADE | SET NULL}]}
```

a) Referenčné obmedzenie (vytvorenie vzťahu pomocou obmedzenia)

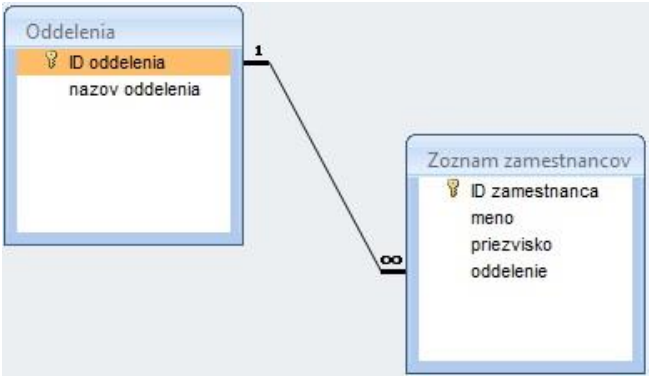
Vo väčšine relačných databáz definujeme referenčné obmedzenie pomocou príkazu **REFERENCES** jazyka SQL. Pozrime sa na príklad referenčného obmedzenia s príkazom ALTER TABLE:

```
ALTER TABLE [Zoznam zamestnancov]
ADD CONSTRAINT Zames_Oddel_ReferencneObmedzenie
FOREIGN KEY (oddelenie)
REFERENCES Oddelenia ([ID oddelenia]);
```

V tomto príklade sme do tabuľky *Zoznam zamestnancov* pridali nové referenčné obmedzenie s názvom ***Zames_Oddel_ReferencneObmedzenie***, podľa ktorého je stĺpec *oddelenie* cudzím

kľúčom, zviazaným so stĺpcom primárneho kľúča *ID oddelenia* v tabuľke *Oddelenia*. **V zápise klauzuly CONSTRAINT môžeme použiť rezervované slová FOREIGN KEY na nastavenie poľa ako cudzieho kľúča**.

Uvedeným spôsobom implementujeme v databáze relácie opísané v rámci jej logického návrhu.

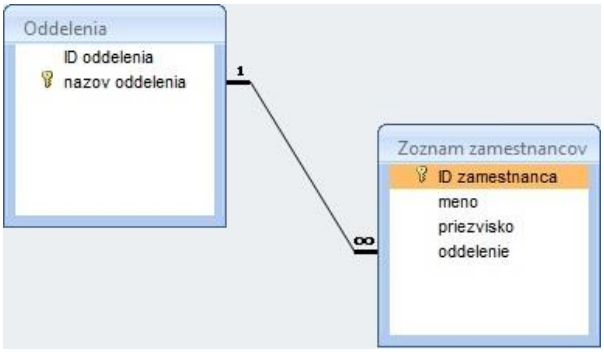


Obrázok 6.4: Grafická vizualizácia vytvoreného vzťahu pomocou klauzuly CONSTRAINT.

Obdobný spôsob vytvorenia relačného vzťahu s názvom ***Zames_Oddel_ReferencneObmedzenie*** medzi tabuľkami *Oddelenia* a *Zoznam zamestnancov* (tentokrát prostredníctvom polí *nazov oddelenia* a *oddelenie*) predstavuje zápis SQL kódu, pričom jeho grafická interpretácia je vizualizovaná na obrázku 6.5. **Rozdiel v porovnaní s predchádzajúcim zápisom je v tom, že pred spustením tohto kódu musí byť už v tabuľke *Oddelenia* definované pole *nazov oddelenia* ako primárny kľúč.**

```
ALTER TABLE [Zoznam zamestnancov]
ALTER COLUMN oddelenie STRING
CONSTRAINT Zames_Oddel_ReferencneObmedzenie
REFERENCES Oddelenia ([nazov oddelenia]);
```

V zápise sa najskôr modifikuje dátový typ poľa *oddelenie* tabuľky *Zoznam zamestnancov* na STRING, a následne sa zrealizuje relačné prepojenie. Zápisom požadujeme, aby sa každá nová hodnota vložená do poľa *oddelenie* tabuľky *Zoznam zamestnancov* zhodovala s hodnotou v poli *nazov oddelenia* tabuľky *Oddelenia*:



Obrázok 6.5: Grafická vizualizácia vytvoreného vzťahu pomocou klauzuly CONSTRAINT.

Rezervované slová PRIMARY KEY môžete použiť na nastavenie jedného poľa alebo množiny polí v tabuľke ako hlavného kľúča (pozri ďalej problematiku s názvom Obmedzenie primárneho kľúča). Všetky hodnoty v hlavnom kľúči musia byť jedinečné a nesmú mať hodnotu Null, pričom tabuľka môže mať iba jeden hlavný kľúč. Ak nastavíme obmedzenie PRIMARY KEY pre tabuľku, ktorá už hlavný kľúč má, databázový systém hlási chybovú správu.

Obmedzenia cudzích kľúčov definujú konkrétne akcie, ktoré sa majú vykonať pri zmene hodnoty zodpovedajúceho hlavného kľúča. Na základe zodpovedajúcej akcie vykonanej na hlavnom kľúči v tabuľke, kde je definovaná klauzula CONSTRAINT, môžeme preto určiť, aké akcie sa majú vykonať na cudzej tabuľke. Vytvorme najprv príkazom CREATE TABLE tabuľku s názvom *Oddelenia*:

```
CREATE TABLE Oddelenia
(
[ID oddelenia] INTEGER PRIMARY KEY,
[nazov oddelenia] STRING (20)
);
```

Uvažujme následne o nasledujúcej definícii tabuľky *Zoznam zamestnancov*, ktorá súčasne –okrem vytvorenia novej tabuľky s príslušnými poľami – realizuje vzťah cudzieho kľúča (*oddelenie*) odkazujúceho sa na primárny kľúč *nazov oddelenia* tabuľky *Oddelenia*:

```
CREATE TABLE [Zoznam zamestnancov]
(
[ID zamestnanca] INTEGER PRIMARY KEY,
meno STRING (20),
priezvisko STRING (20),
oddelenie INTEGER,
CONSTRAINT Moja_relácia FOREIGN KEY (oddelenie)
REFERENCES Oddelenia ON UPDATE CASCADE ON DELETE CASCADE
);
```

Na cudzom kľúči sú definované klauzuly ON UPDATE CASCADE a ON DELETE CASCADE. Klauzula **ON UPDATE CASCADE** znamená, že ak sa aktualizuje identifikácia oddelenia (*nazov_oddelenia*) v tabuľke *Oddelenia*, aktualizácia sa kaskádovo rozšíri na tabuľku *Zoznam zamestnancov*. Každá hodnota obsahujúca zodpovedajúcu hodnotu identifikácie oddelenia sa aktualizuje novou hodnotou. Klauzula **ON DELETE CASCADE** znamená, že ak sa z tabuľky *Oddelenia* odstráni určité oddelenie, všetky riadky v tabuľke *Zoznam zamestnancov,* obsahujúce tú istú hodnotu identifikácie oddelenia, sa tiež odstránia.

Uvažujme o nasledujúcej, odlišnej definícii tabuľky *Zoznam zamestnancov*, používajúcej akciu SET NULL namiesto kaskádovej akcie CASCADE:

```
CREATE TABLE [Zoznam zamestnancov] (
[ID zamestnanca] Integer PRIMARY KEY,
meno STRING (20),
priezvisko STRING (20),
oddelenie INTEGER,
CONSTRAINT Moja_relácia FOREIGN KEY (oddelenie)
REFERENCES Oddelenia ON UPDATE SET NULL ON DELETE SET NULL
);
```

Klauzula **ON UPDATE SET NULL** znamená, že ak sa v tabuľke *Oddelenia* aktualizuje identifikácia oddelenia (*nazov oddelenia*), zodpovedajúce hodnoty cudzieho kľúča v tabuľke *Zoznam zamestnancov* sa automaticky nastavia na hodnotu NULL. Podobne znamená klauzula **ON DELETE SET NULL**, že ak je z tabuľky *Oddelenia* odstránené určité oddelenie, všetky zodpovedajúce cudzie kľúče v tabuľke *Zoznam zamestnancov* sa automaticky nastavia na hodnotu NULL.

b) Obmedzenie primárneho kľúča

Obmedzenie primárneho kľúča zaisťuje, že stĺpec alebo stĺpce, určené ako primárny kľúč tabuľky, nebudú nikdy obsahovať dve rovnaké, duplicitné hodnoty. Väčšina relačných databázových systémov vytvára pre zaistenie tohto obmedzenia zvláštny, jedinečný index. Index je pritom špeciálny databázový objekt, ktorý obsahuje hodnoty primárneho kľúča z jedného alebo z viacerých stĺpcov tabuľky, a ku každému z nich ukazovateľ do riadku tabuľky s príslušnou hodnotou kľúča. Takto zostavený index urýchľuje prehľadávanie tabuľky podľa hodnoty kľúča.

My sa ale vrátime k obmedzeniu primárneho kľúča, ktoré v tabuľke *Zoznam zamestnancov* nadefinujeme na pole *Identifikácia* nasledovne:

```
ALTER TABLE [Zoznam zamestnancov]
ADD CONSTRAINT Obmedz_primar_kluca
PRIMARY KEY ([ID zamestnanca])
```

Ak by sme chceli pre tabuľku *Oddelenia* definovať nový primárny kľúč nad poľom *nazov oddelenia* s tým, že by sme súčasne chceli zmeniť aj dátový typ tohto poľa na Text, zapísali by sme do definičného dopytu nasledujúci kód:

```
ALTER TABLE Oddelenia
ALTER COLUMN [nazov oddelenia] TEXT
CONSTRAINT Obmedz_primar_kluca_2
PRIMARY KEY;
```

V situácii, že by sme chceli prostredníctvom obmedzenia primárneho kľúča definovať viacpoložkový primárny kľúč pre tabuľku *Zoznam zamestnancov*, zápis SQL kódu by vyzeral nasledovne:

```
ALTER TABLE [Zoznam zamestnancov]
ADD CONSTRAINT Obmedz_2polozkoveho_primar_kluca
PRIMARY KEY (meno, priezvisko);
```

c) Jedinečné obmedzenie

Okrem primárneho kľúča môžeme v tabuľke vynútiť dodržanie jedinečných hodnôt určitého stĺpca (stĺpcov) aj pomocou "obyčajného", jedinečného obmedzenia bez primárneho kľúča (Obrázok 6.6). **Každá tabuľka môže mať len jeden primárny kľúč, ale jedinečných obmedzení môže mať aj viacej.** Väčšina relačných databázových systémov vytvára opäť pre zaistenie týchto obmedzení jedinečný index.

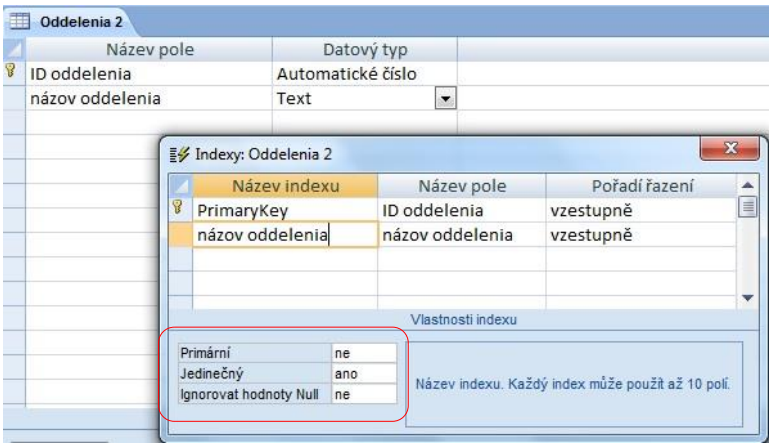
Týmto spôsobom napríklad zaistíme, že v tabuľke *Oddelenia* nebudú zaznamenané dve oddelenia rovnakým názvom.

```
ALTER TABLE Oddelenia
ADD CONSTRAINT Unikatne_Oddelenie
UNIQUE ([názov oddelenia]);
```

To isté jedinečné obmedzenie môžeme potom nasledujúcim spôsobom odstrániť:

```
ALTER TABLE Oddelenia
```

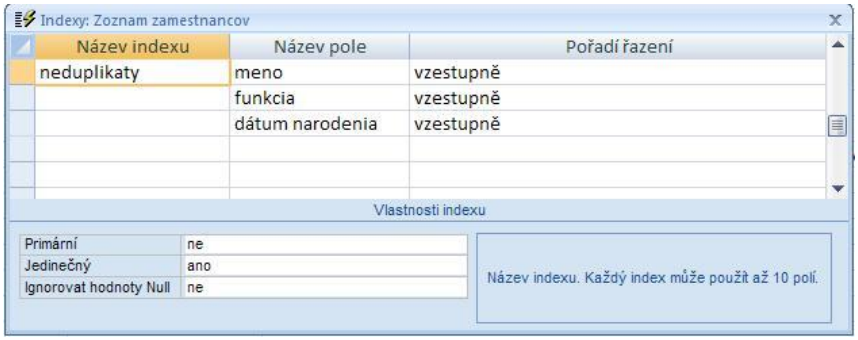
```
DROP CONSTRAINT Unikatne_Oddelenie;
```



Obrázok 6.6: Definovanie jedinečného obmedzenia pre pole *názov oddelenia*.

Predpokladajme, že chceme zaistiť, aby žiadne dva záznamy v tabuľke *Zoznam zamestnancov* nemali rovnakú množinu údajov pre polia *meno*, *funkcia* a *dátum narodenia*. Úlohu môžeme vyriešiť príkazom UNIQUE, ktorý sa vzťahuje na uvedené polia (Obrázok 6.7).

```
ALTER TABLE [Zoznam zamestnancov]
ADD CONSTRAINT neduplikaty
UNIQUE (meno, funkcia, [datum narodenia])
```



Obrázok 6.7: Definovanie jedinečného obmedzenia pre viac polí.

d) Obmedzenie typu CHECK

Kľúčové slovo CHECK označuje „všeobecné“ obmedzenie, ktoré slúži na zaistenie ľubovoľného aplikačného pravidla, definovaného nad jedným stĺpcom tabuľky. Pri akejkoľvek zmene dátovej hodnoty stĺpca sa vyhodnotí podmienka uvedená v tomto obmedzení, a pokiaľ nie je splnená, t.j. nie je rovná hodnote TRUE, zobrazí sa hlásenie o chybe.

Nasledujúci príklad implementuje obmedzenie typu CHECK, ktoré zaistí, že stĺpec *plat* v tabuľke *Platy* zamestnancov bude vždy obsahovať kladnú hodnotu a hodnota v poli oddelenie bude vždy „ekonomické“:

```
ALTER TABLE Platy
ADD CONSTRAINT Kladne_hodnoty_platov
CHECK (plat > 0 AND oddelenie=“ekonomické“)
```

A nasledujúcim príkazom opísané obmedzenie CHECK opäť odstránime:

```
ALTER TABLE Platy
```

```
DROP CONSTRAINT Kladne_hodnoty_platov;
```

➢ Príkaz DROP TABLE

Príkaz DROP jazyka SQL nemá svoj ekvivalent medzi štandardnými dopytmi aplikácie Access. Týmto **príkazom odstránime z databázy existujúcu tabuľku (vrátane jej štruktúry), index alebo zobrazenie**. Pred odstránením tabuľky alebo indexu z tabuľky sa tabuľka musí zatvoriť.

Syntax:

```
DROP {TABLE nazov_tabuľky | INDEX názov_indexu ON názov_tabuľky | VIEW názov_zobrazenia}
```

Odstránenie tabuľky realizujeme zápisom príkazu:

```
DROP TABLE [Zoznam režisérov];
```

Príkaz fyzicky odstráni tabuľku *Zoznam režisérov* z aktuálnej databázy.

Odstránenie indexu s názvom *Moj_index* nad stĺpcom *Identifikácia* (je súčasne definovaný ako primárny kľúč tabuľky) z tabuľky *Zoznam zamestnancov*, ktorý sme vytvorili v jednom z predchádzajúcich príkladov (v príkaze CREATE INDEX), zrealizujeme zápisom príkazu DROP INDEX nasledovne:

```
DROP  INDEX Moj_index ON [Zoznam zamestnancov]
```

Pri odstraňovaní tabuľky môžeme **do príkazu CONSTRAINTS doplniť klauzulu CASCADE**, **ktorá súčasne automaticky odstráni všetky referenčné obmedzenia, ktorých sa daná tabuľka zúčastňuje**. Ak odstránime z databázy tabuľku, odstránia sa súčasne taktiež všetky objekty, ktoré sú od nej závislé (teda indexy a obmedzenia). Dopyty založené na odstránenej tabuľke sa ale vo väčšine relačných databázových systémov neodstraňujú, ale sa označia ako neplatné, takže s nimi nemôžeme pracovať, jedine v prípade, že by sme odstránenú tabuľku opäť vytvorili.

```
DROP TABLE názov_tabuľky CASCADE CONSTRAINTS;
```

➢ Príkaz UNION

Príkaz UNION nemá svoj ekvivalent medzi štandardnými dopytmi aplikácie Access. **Príkaz UNION dokáže zlúčiť výsledky viacerých dopytov do jedného dopytu.** Zlučovacie alebo inak komplikované dopyty nie sú teda návrhovým zobrazením priamo podporované.

Syntax:

```
dotaz_1 UNION dotaz_2 UNION <UNION dotaz_n>
```

```
SELECT meno, vek FROM [Zoznam hercov]
UNION
SELECT meno, vek FROM [Zoznam režisérov]
ORDER BY vek;
```

Príkaz vytvorí tabuľku, ktorá bude pozostávať z *mena* a *veku* hercov z tabuľky *Zoznam hercov* a z *mena* a *veku* režisérov z tabuľky *Zoznam režisérov*. Výsledný zoznam bude utriedený podľa poľa *vek* vzostupne.

```
SELECT Count (*) As Počet, ”Praha” As Mesto FROM ObyvateliaPraha
UNION
SELECT Count (*) As Počet, ”Ostrava” As Mesto FROM ObyvateliaOstrava
```

Výsledkom zápisu budú dva záznamy s dvomi stĺpcami. V prvom stĺpci bude počet záznamov z danej tabuľky (bude mať názov *Počet*) a v druhom stĺpci bude názov mesta (bude mať názov *Mesto*).

V nasledujúcom príklade zostavíme dva podobné dopyty, ktorých výstupy následne zlúčime prostredníctvom príkazu UNION. V prípade prvého dopytu nás budú zaujímať všetky knihy, ktoré vo svojom názve obsahujú slovo Programovanie (Like "Programovanie*"). Predmetom druhého dopytu budú iba tie knihy, ktorých cena je vyššia ako 100 eur. V prípade druhého dopytu sme potrebovali rozšíriť zoznam stĺpcov o pole *cena* – tým však vzniká problém pri zjednocovaní výsledkov, ktoré vyžaduje, aby obidve spájané množiny dát obsahovali rovnaké polia. V prípade, že by sa zoznamy polí obidvoch dopytov líšili, nebolo by možné zjednotenie zrealizovať. Aby sme zabránili zobrazeniu dát z poľa *cena* vo výsledku zjednotenia, jednoducho toto pole neuvedieme do zoznamu zobrazovaných polí, ale len ako kritérium v klauzule WHERE.

```
SELECT Knihy.autor, Knihy.názov, Vydavateľstvo.nazov
FROM Vydavateľstvo INNER JOIN Knihy ON Vydavateľstvo.ID = Knihy.vydavateľstvo
WHERE (((Knihy.názov) Like "Programovanie*"));
UNION
SELECT Knihy.autor, Knihy.názov, Vydavateľstvo.nazov
FROM Vydavateľstvo INNER JOIN Knihy ON Vydavateľstvo.ID = Knihy.vydavateľstvo
WHERE (((Knihy.cena)>100));
```

Použitie príkazu UNION v predchádzajúcom prípade bolo len príkladom možnosti zobrazenia SQL, v praxi by bolo možné tento dopyt vytvoriť oveľa efektívnejšie.

6.5 Jazyk pre riadenie dát

Databázové oprávnenie alebo – povedané aj – privilégium, znamená povolenie na realizáciu určitých operácií v databáze. Databázový užívateľ, ktorý toto oprávnenie udeľuje, sa nazýva oprávňovateľ (garant), zatiaľ čo príjemca oprávnenia sa nazýva oprávnený. Oprávnenia, alebo privilégiá, sa dajú rozdeliť do nasledujúcich dvoch širokých kategórií:

- **Systémové oprávnenia** – tieto oprávnenia povoľujú užívateľovi uskutočňovať rôzne všeobecné činnosti v databáze, ako je napríklad vytvorenie nových užívateľských účtov, alebo pripojení k databáze.
- **Objektové oprávnenia** – oprávnenia k objektom povoľujú užívateľovi realizovať konkrétne operácie nad konkrétnymi objektmi, napríklad výber dát z jednej tabuľky, alebo aktualizáciu dát v druhej tabuľke.

Pre zjednodušenie, inak veľmi prácnej správy oprávnení, väčšina relačných databázových systémov ponúka uloženie množiny definovaných oprávnení do jedného pomenovaného objektu, takzvanej roly. Tieto roly potom môžeme prideliť jednotlivým užívateľom alebo pomenovaným skupinám užívateľov, ktorí z roly zdedia všetky v nej obsiahnuté oprávnenia. Relačný databázový systém, ktorý definíciu rolí podporuje, máva spravidla taktiež určité preddefinované roly.

Užívatelia a skupiny užívateľov sú definovaní na SQL serveri. Ich existencia je nezávislá od aplikácií – v každej aplikácii v jednej databáze vystupujú rovnakí užívatelia a ich skupiny. Na rozdiel od užívateľov a skupín sú roly objekty existujúce vo vnútri aplikácie. V každej aplikácii môžu byť definované iné roly a mimo svoju aplikáciu rola žiadny význam nemá.

Vytvoriť na SQL serveri nového užívateľa zadaného mena realizujeme príkazom CREATE USER. Skupinu užívateľov je možné vytvoriť SQL príkazom **CREATE GROUP**, zrušiť príkazom **DROP GROUP**.

```
CREATE USER novy_uzivatel;
```

```
CALL Set_password (' novy_uzivatel ','nazov_hesla');


CREATE GROUP nova_skupina;
CALL Set_membership(novy_uzivatel, CATEG_USER, ' nova_skupina ', CATEG_GROUP, TRUE);
```

Meno nového užívateľa môže byť dlhé maximálne 31 znakov. Užívateľ tohto mena (*novy_uzivatel*) nesmie doposiaľ na SQL severy existovať. Heslo sa užívateľovi pridá pomocou funkcie **Set_password**.

Nastavenie práv (pridanie alebo odobranie) pre zadaný subjekt (užívateľa, skupinu užívateľov) k tabuľke alebo k záznamom tabuľky realizujeme prostredníctvom SQL príkazov GRANT a REVOKE.

- **Príkaz GRANT**

V prípade, že je v databáze vytvorený objekt, má oprávnenie k jeho použitiu jedine jeho tvorca, a taktiež iba on má právo prideľovať oprávnenie iným užívateľom. **Oprávnenie sa v jazyku SQL prideľuje existujúcemu používateľovi alebo skupine pomocou príkazu GRANT.**

Na udelenie oprávnenia (pripojenia) k tabuľke potrebujú používatelia SELECT (právo čítať všetky stĺpce), INSERT (právo vkladať záznamy), UPDATE (právo prepisovať jednotlivé stĺpce špecifikované v zozname stĺpcov; ak nie je zoznam stĺpcov uvedený, právo prepisovať platí pre všetky stĺpce) alebo DELETE (právo rušiť záznamy) oprávnenia pre túto tabuľku. V zozname oddelených čiarkami vypíšeme názvy oprávnení, ktoré sa majú nastaviť. Ak namiesto oprávnenia, resp. zoznamu oprávnení, použijeme kľúčové slovo ALL, nastavia sa uvedenému užívateľovi všetky možné oprávnenia. Kľúčovým slovom REFERENCES sa definuje právo čítať iba niektoré stĺpce špecifikované v zozname stĺpcov. Ak nie je zoznam stĺpcov uvedený, právo čítať platí pre všetky stĺpce (to isté ako pri zápise SELECT);

Syntax:

```
GRANT {názov_oprávnenia [,názov_oprávnenia, …]}
ON
{TABLE názov_tabuľky  [(názov_stĺpca)] |
OBJECT názov_objektu |
CONTAINER názov_kontajneru }
TO {meno_užívateľa |PUBLIC |názov_role}
[WITH GRANT OPTION];
```

Za klauzulou OBJECT sa ako názov objektu môžeme uviesť ľubovoľný objekt, okrem názvu tabuľky. Jedným z príkladov je uložený dopyt.

V zozname subjektov za klauzulou TO určujeme, ktorých subjektov sa nastavenie oprávnení týka. Na udelenie prístupových práv pre všetkých užívateľov sa použije klauzula PUBLIC. V inom prípade sa nastaví oprávnenie (oprávnenia) tým užívateľom, skupinám alebo rolám, ktorí sú uvedení svojím menom. Ak použijeme mená skupiny užívateľov (ktoré už musia byť predtým definované na serveri), nastavia sa práva tejto skupine, tzn., že užívatelia v nej zaradení dedia práva automaticky. Pokiaľ pri názve subjektu, ktorý má získať oprávnenie, nie je uvedené kľúčové slovo USER, GROUP ani ROLE, jeho meno sa hľadá najprv medzi užívateľmi, potom medzi skupinami.

Pokiaľ subjekt už má určité oprávnenia pridelené, potom oprávnenia uvedené v ďalšom príkaze GRANT sa k nim pridajú.

Dajme užívateľovi Johny oprávnenie v databáze *Knižnica* vymazávať záznamy a editovať *identifikáciu* užívateľa, *meno* a *priezvisko*, ako aj jeho *dátum narodenia* z tabuľky *Užívatelia.*

Definujte užívateľovi Johny aj všetky oprávnenia k jednému vybranému záznamu z tabuľky *Užívatelia*.

```
GRANT DELETE, UPDATE ([Cislo preukazky], [meno], [priezvisko], [ dátum narodenia])
ON Užívatelia
TO Johny
```

```
GRANT ALL
ON Užívatelia
TO USER Johny
WHERE [ID oddelenia]=4
```

Ak sa majú nastavovať oprávnenia k vybraným záznamom, ide o tzv. záznamové oprávnenia, musí sa použiť klauzula WHERE s obmedzujúcou podmienkou na záznamy tabuľky *názov_tabulky,* a súčasne tabuľka musí mať tieto oprávnenia povolené pri návrhu tabuľky. Pre záznamové oprávnenia nie je k dispozícii oprávnenie INSERT.

Väčšina relačných databáz, ktoré podporujú prideľovanie oprávnení, umožňujú oprávňovateľovi delegovať povolenie na udeľovanie oprávnení ďalším užívateľom – inými slovami, užívateľ môže „odovzdať" inému užívateľa právo prideľovať privilégiá ďalším užívateľom. Znamená to zapísať pri systémových oprávneniach klauzulu WITH ADMIN OPTION, respektíve pri objektových oprávneniach klauzulu WITH GRANT OPTION.

- **Príkaz REVOKE**

Pridelené oprávnenia je možné používateľovi alebo skupine odobrať príkazom REVOKE. Oprávnenia sú tie operácie, pre ktoré má užívateľ určitú právomoc. Jednotlivé oprávnenia sú opísané v predchádzajúcej časti textu venovanej klauzule GRANT.

Odobrať oprávnenia môže len užívateľ prideľujúci oprávnenia (príkazom GRANT). Jeden užívateľ môže mať stanovené oprávnenia pre databázový objekt od akéhokoľvek počtu iných užívateľov. Príkaz REVOKE vydaný užívateľom ruší iba oprávnenia skôr stanovené práve týmto konkrétnym užívateľom. Oprávnenia, pridelené všetkým užívateľom s PUBLIC špecifikáciou, môžu byť zrušené iba zrušením oprávnení s uvedením špecifikácie PUBLIC.

Pokiaľ mal daný užívateľ pridelené oprávnenie s klauzulou WITH GRANT OPTION, odobraním jeho oprávnení odoberieme príslušné oprávnenia v kaskáde aj všetkým ostatným užívateľom, ktorým tento užívateľ oprávnenia poskytol.

Syntax:

```
REVOKE { názov_oprávnenia [,názov_oprávnenia, …]}
ON
{TABLE názov_tabuľky  |
OBJECT názov_objektu |
CONTAINTER názov_kontajneru }
FROM { meno_užívateľa [,meno_užívateľa, …]}
```

Všetkým užívateľom v databáze *Knižnica* zabráňte vo vymazaní záznamov z tabuľky *Užívatelia*, ako aj v editácii *identifikácie* užívateľa, *mena* a *priezviska* a jeho *dátumu narodenia*.

```
REVOKE DELETE, UPDATE ([Cislo preukazky], [meno], [priezvisko], [ dátum narodenia])
ON Užívatelia
FROM PUBLIC
```

Ak sa majú nastavovať oprávnenia k vybraným záznamom (záznamové oprávnenia), musí sa použiť klauzula WHERE s obmedzujúcou podmienkou na záznamy tabuľky *názov_tabulky* a súčasne tabuľka musí mať tieto oprávnenia povolené pri návrhu tabuľky. V zápise zoznamu viacerých oprávnení definujeme, aké práva sa majú odobrať, pričom ich pri výpise oddelíme čiarkami. Ak použijeme kľúčové slov ALL, odoberú sa všetky možné oprávnenia k tabuľke.