Databázové systémy

SQL Server

Petr Voborník

Tento materiál vznikl v rámci realizace projektu
Strategický rozvoj Univerzity Hradec Králové, reg. č. CZ.02.2.69/0.0/0.0/16_015/0002427

EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

verze z 30.10.2022

Obsah

1 Databázové systémy

Pojem *databázové systémy* (DBS) označuje nejenom software pro řízení báze dat (SŘBD) a zprostředkování aplikačního rozhraní (API) pro přístup k datům v nich uložených, ale také celou tu vědu okolo nich. Tedy jaké základní databázové objekty by měly v každé správné databázi existovat, jak mohou být vzájemně provázané, jak by mělo vypadat základní jednotné rozhraní pro přístup a práci s těmito objekty i daty v nich uložených, a jak tyto nástroje používat co nejefektivněji.[1]

Databázových systémů existuje celá řada, každý má svá specifika a může být nejvhodnější variantou pro různé případy využití. Proto je nejen dobré ovládat alespoň jeden z nich důkladně, ale zároveň mít také přehled i o těch ostatních, kdyby náhodou v nějaké situaci bylo vhodnější zvolit je.

V této knize se zaměříme především na databázový systém **SQL Server**, který je použitelným ve většině klasických případů, v základní verzi *Express* je zdarma, a přitom skýtá široké možnosti správy i velmi slušný výkon, který lze při přechodu na placenou verzi ještě několikanásobně navýšit bez nutnosti změny rozhraní pracujícího s verzí nižší.

1.1 Základní databázové objekty

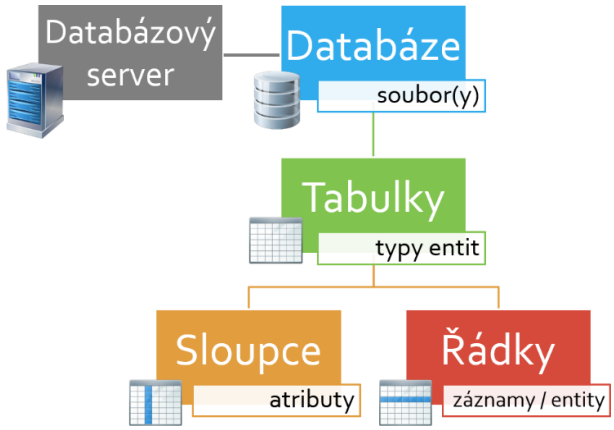
Databázový systém je tedy v základu software, který pro svůj provoz potřebuje i nějaký hardware čili počítač s operačním systémem. V ideálním případě bývá pro náročnější databáze vyhrazen samostatný server, který stíhá zpracovat i tisíce požadavků v řádu sekund, aniž by došlo k jeho zahlcení.

[1] Celou přednášku na téma databázové systémy najdete na programko.net/databaze/dbs

V případě databází, jejichž přetížení hrozí jen nárazově (např. e-shop před Vánocemi, web finančního úřadu pár dní před termínem odevzdání, VŠ systém pro přihlašování na rozvrhy na začátku semestru apod.) a po zbytek roku je jeho zátěž minimální, bývá vhodnější než zřizovat a udržovat vlastní server a konektivitu schopnou zvládnout i tyto špičky, použít cloudový virtuální server (např. Azure), jehož cena je stanovena za použitý výkon, který se dokáže automaticky přizpůsobovat aktuální potřebě.

Nicméně pro provoz databázového systému, který používáme např. pro vývoj aplikace, plně dostačuje i klasický osobní počítač či notebook. Poslední verze SQL Serveru pak kromě operačního systému Windows funguje i pod Linuxem. Zařízení, na kterém databázový systém běží, tak budeme nazývat databázovým serverem, ať se již jedná o cokoli z výše uvedeného.

Na tomto databázovém serveru se tedy nacházejí databáze. U náročnějších systémů může být jen jedna, ale obvykle jich na jednom serveru bývá spravováno více, od jednotek až třeba i po stovky (např. na webhostingu). Databáze se může skládat z jednoho (např. Firebird) či více souborů (např. SQL Server používá 2), některé databázové systémy dokonce používají celou složku souborů, které si vytváří pro každou tabulku zvlášť.



Databáze pak obsahuje jednotlivé tabulky (někdy též typy entit), jež mezi sebou mohou mít nadefinované různé vztahy (reference). Tabulka je samozřejmě tvořena sloupci (atributy) a řádky (záznamy či entitami). Můžeme tak mít třeba tabulku *Knihy*, jejímiž sloupci budou např. *Název*, *Rok vydání*, *Autor*, a řádky tabulky pak budou tvořit jednotlivé knihy, které evidujeme, a jejichž údaje budou rozepsány do příslušných sloupců.

Knihy		
Název	Rok vydání	Autor
Bídníci	1862	Victor Hugo
Osvícení	1977	Stephen King
To	1986	Stephen King
Marťan	2011	Andy Weir

Databázové tabulky musí dodržovat následující pravidla:

1. Každá tabulka má jednoznačné jméno.
2. Každý sloupec v tabulce má jednoznačné jméno.
3. Všechny hodnoty daného sloupce musí být stejného typu.
4. Nezáleží na pořadí sloupců (z pohledu DBS).
5. Nezáleží na pořadí řádků (z pohledu DBS).
6. Tabulka nemůže mít duplicitní řádky.
7. Všechny hodnoty jsou atomické.
8. Každá tabulka musí mít primární klíč.

1.2 Vztahy mezi tabulkami (reference)

Jedna databáze tedy může obsahovat vícero tabulek, obvykle jde o desítky až stovky. Jejich data (záznamy) však v drtivé většině nejsou evidována jen tak sama o sobě, ale bývají vzájemně propojena. Například budeme-li mít výše zmíněnou tabulku pro evidenci knih, bude časem nezbytné jejich autory začít evidovat ve vlastní tabulce, a tyto tabulky vzájemně propojit.

Autoři		
Jméno	Příjmení	Rok narození
Victor	Hugo	1802
Stephen	King	1947
Andy	Weir	1972

Toto rozdělení totiž přináší spoustu výhod, například tak bude možné o autorech evidovat více údajů a zároveň nebude už nezbytné tyto údaje znovu uvádět u každé knihy od téhož autora, a především tyto údaje všude udržovat neustále totožné (co kdyby u jedné knihy někdo zapsal autora jako „Stephen King“ a u druhé „King Stephen“ či „S. King“).

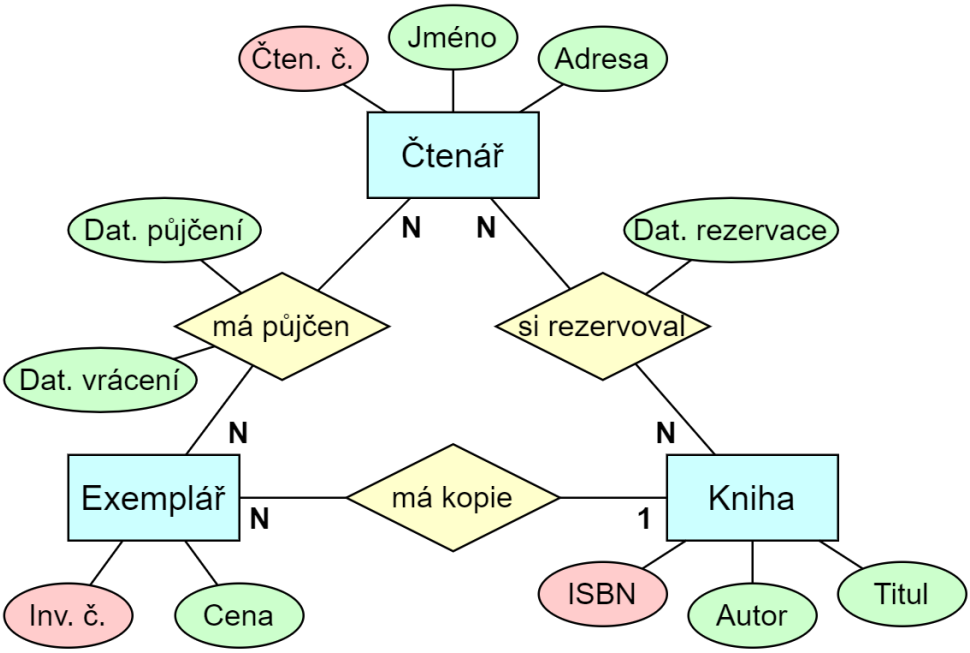
Aby takovéto propojení bylo možné, je třeba do tabulek přidat sloupec, který bude mít v každém řádku jedinečnou hodnotu (jde o tzv. *primární klíč*, *primary key*, zkráceně PK), často nazvaný jako „ID“. Díky němu pak lze do tabulky místo zopakování všech hodnot z řádku druhé tabulky pouze zaevidovat hodnotu tohoto primárního klíče (např. do tabulky knih přidat sloupec s hodnotou ID autora) a příslušný řádek druhé tabulky (autorů) tak lze mezi nimi kdykoli dohledat. Tomuto sloupci s identifikátorem (ID) z jiné tabulky se říká *cizí klíč* (*foreign key*, zkráceně FK).

Autoři			
Id	Jméno	Příjmení	Rok narození
1	Victor	Hugo	1802
2	Stephen	King	1947
3	Andy	Weir	1972

Knihy			
Id	Název	Rok vydání	Id autora
1	Bídníci	1862	1
2	Osvícení	1977	2
3	To	1986	2
4	Marťan	2011	3

1.3 Optimalizace (normalizace) databázového návrhu

Návrh struktury nové databáze obvykle začíná tak, že si sepíšeme typy prvků (entit), které v databázi potřebujeme evidovat (čili tabulky, též typy entit). K nim pak určíme, jaké jejich údaje budeme evidovat (sloupce tabulky, nebo též atributy) a jim zvolíme, či lépe přidáme, primární klíč (**PK**)[2]. Na závěr se zakreslí vztahy mezi tabulkami, u kterých se na jejich koncích určí jejich kardinalita (1 nebo N, např. *1 kniha má N exemplářů*, či *1 čtenář má N výpůjček*). Tím vznikne tzv. Entitně-relační model/diagram (ERM).



[2] V praxi není příliš vhodné jako primární klíč používat žádný z údajů záznamu, jako je např. rodné číslo u osob, ale v mnoha ohledech bývá nejvýhodnější použít automaticky se navyšující číslo (ID). V případě decentralizovaných databází se pak také můžeme setkat s identifikátorem typu GUID, tj. delší náhodná binární hodnota obvykle zobrazovaná v hexadecimálním formátu, jejíž šance na opakované vygenerování jsou tak nízké, že se tato možnost programově ani neřeší.

Místo grafického nákresu lze též použít i jeho textovou reprezentaci. Zde je již vhodné uvést též spojovací tabulky pro vztahy typu M:N, a také na správnou stranu vztahů tabulek zapsat cizí klíče (**FK**, tj. sloupce pro vložení ID záznamu propojené tabulky). Pro názvy tabulek se zde již také může použít množné číslo, neboť např. v tabulce *Čtenáři* budou čtenáři, nikoli jen jeden čtenář.

```
Čtenáři (Č_čt, Jméno, Adresa)
Knihy (ISBN, Autor, Titul)
Exempláře (Inv_č, ISBN, Cena)
Výpůjčky (Inv_č, Č_čt, Datum_půjčení, Datum_vrácení)
Rezervace (ISBN, Č_čt, Datum_rezervace)
```

Tím máme připravený koncept, který buď již rovnou dodržuje tzv. pravidla *normalizace databáze*, nebo jej teprve podle nich ještě upravíme. Těchto pravidel je celkem 6, nicméně potížím, které většina z nich řeší se vyhneme, pokud nebudeme v tabulkách používat složený primární klíč[3] či některou hodnotu ze záznamu, ale právě ono automaticky generované ID. Blíže se tedy podíváme pouze na první a třetí normální formy (NF).

1. NF: Všechny hodnoty musí být atomické

Jednotlivé údaje je tedy třeba „rozsekat na atomy“, čili rozdělit na dále nedělitelné části, do samostatných sloupců tabulky (např. celé jméno rozdělit na jméno a příjmení, popř. druhé jméno, tituly před a tituly za jménem, nebo adresu na ulici, čp, město a PSČ). Hodnoty se pak složit zpět dají vždy, jejich správné je rozdělení ale nelze vždy automatizovat.

[3] Složený primární klíč znamená, že klíčová unikátní hodnota pro každý řádek není složena pouze z jednoho údaje (sloupce), ale ze dvou či více.

3. NF: Všechny neklíčové atributy jsou navzájem nezávislé

Souvisí-li spolu některé dva či více údajů (sloupců) v tabulce, pak si zaslouží zapsat do vlastní tabulky (ty se někdy označují jako *číselníky*) a v původní tabulce budou společně zastoupeny pouze jedním cizím klíčem. Například pokud u osob evidujeme jejich adresy, a to i město a PSČ, které spolu jednoznačně souvisí, pak by bylo vhodnější vytvořit tabulku *Města* (se sloupci *ID*, *Město*, *PSČ*) a do tabulky osob místo nich dát pouze sloupec *ID Města* (vše samozřejmě bez diakritiky a mezer, tedy třeba *IdMesta*, *MestoId*, *ID_Mesta*, *Mesto_ID* apod., záleží na konvenci, kterou si zvolíte a která by pak měla být stejná v celé databázi).

Účelem normalizace databázového návrhu tedy je, vše co nejvíce rozdělit, jednak do sloupců a dále do samostatných tabulek[4]. Ve finále by pak v databázi měl být každý údaj zapsaný pouze jednou. Například bez tabulky měst, bychom název města museli psát ke každé osobě znovu, takže v případě více osob z téhož města by se jeho název v databázi vyskytoval vícekrát. Přitom nelze zcela vyloučit, že by mohl být zapsán různě (překlep, zkrácení, mezery), což by znamenalo, že:

- pomocí různě zapsaných názvů nelze data seskupovat (viz kap. 2.2),
- velikost databáze by zbytečně narůstala, kvůli opakujícím se (redundantním) hodnotám,
- změna či oprava názvu města (či jeho PSČ) by se netýkala jednoho řádku v tabulce města, ale musely by se prohledat všechny řádky v tabulce osob (popř. i v dalších tabulkách s tímto údajem) a opravovat jich více.

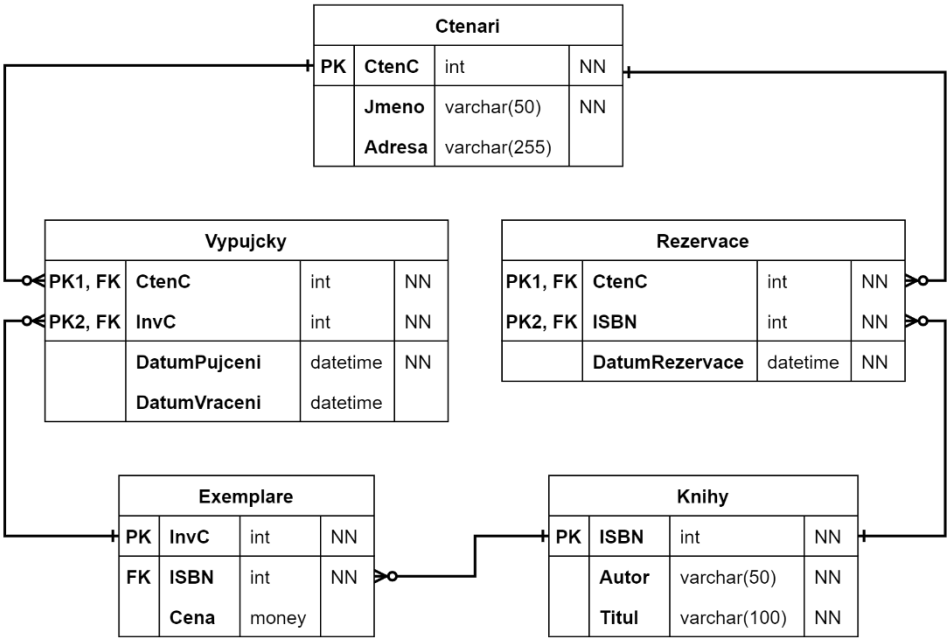
[4] Poslední normální forma přímo říká, že *relaci již není možno bezeztrátově rozložit*.

1.4 Datové modely

Datové modely slouží k vizualizaci databázového návrhu, která nám pomáhá se v něm snáze a rychleji zorientovat, odhalit případné chyby či nedostatky, a především je ve fázi prvotní přípravy databáze nejintuitivnějším způsobem, jak jej vytvářet od nuly.

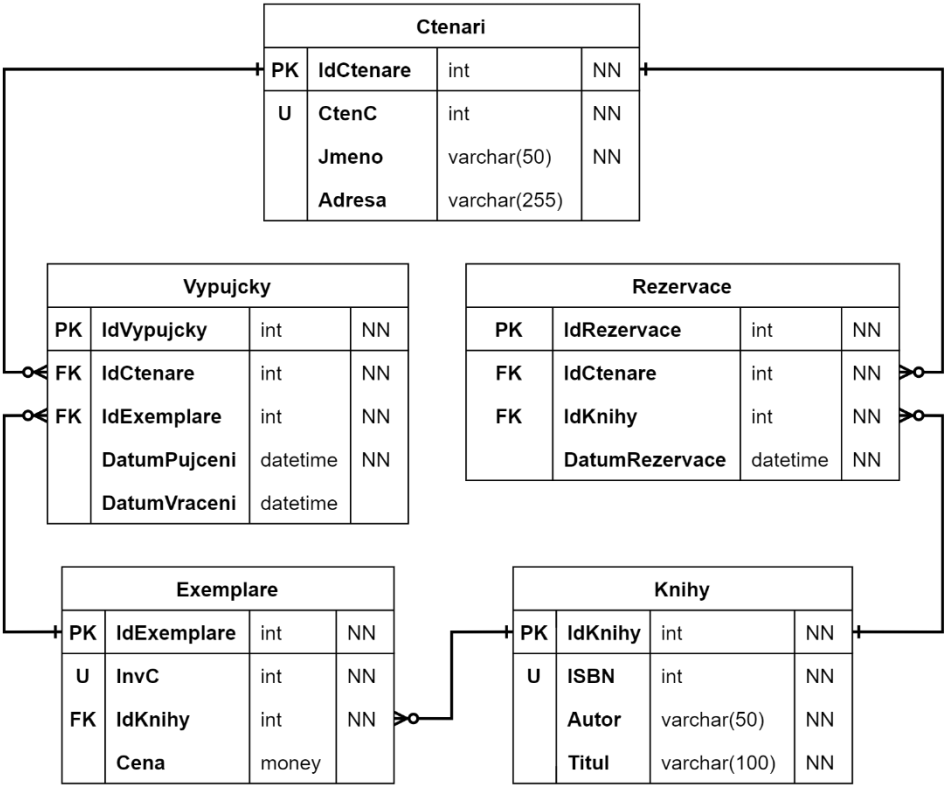
Datový model, minimálně jeho prvotní váze, se dost často může zakreslit od ruky, třeba i na papír, existuje však řada softwarových nástrojů, které umožňují jej efektivně budovat, ladit a spravovat. Většina z nich také dokáže na základě sestaveného modelu vygenerovat DDL skript (viz kap. 5), kterým lze následně databázi snadno vytvořit. Některé nástroje disponují také funkcí pro reverzní inženýrství, tzn. že na základě takovéhoto DDL skriptu, který lze z hotové databáze běžně vyexportovat, dokáže vytvořit datový model.

Datový model obvykle obsahuje jednotlivé tabulky (jejich název), jejich atributy (sloupce), jejich datové typy, jsou-li jejich hodnoty povinné (*not null*, zkráceně NN), označení klíčů (primárních PK a cizích FK), dále pak vztahy mezi tabulkami (reference) a jejich kardinalitu (1:1 nebo 1:N, vztahy typu M:N by zde již měly být zastoupeny samostatnými spojovacími tabulkami).



U větších databází (se stovkami tabulek) se pak datové modely obvykle vykreslují pouze pro určitou související pod-část z nich, přičemž toto rozdělení na SQL Serveru lze zanést i do samotné databáze pomocí tzv. *schémat*. Ta se pak používají v SQL dotazech jako prefixy názvů tabulek (např. **knihovna.KNIHY**), přičemž výchozím schématem je **dbo**, které se v nich jako jediné zapisovat nemusí.

Co se týče uvedené ukázky zjednodušeného modelu knihovny, tak jeho předchozí verze byla spíše akademická. Jako klíče se tam totiž používaly hodnoty přímo ze záznamu a složené primární klíče, což v praxi obvykle přináší nejrůznější problémy. Např. by si stejný čtenář nemohl vypůjčit dvakrát tentýž exemplář knihy, popř. si opakovaně rezervovat tentýž titul, nebo by také nešlo vůbec zaevidovat starší knihu bez ISBN, a v případě jeho chybného zadání by bylo velmi složité jej opravit. Násl. model tak ukazuje vhodnější použití klíčů pro praktické použití DB (model však není normalizován).



Pro následující příklady v jazyce SQL (kap. 2–4) budeme používat mírně upravenou[5] ukázkovou databázi Microsoftu nazvanou *AdwentureWorksLT*, jejíž datový model je následující. Tuto databázi v upravené verzi najdete adrese programko.net/databaze/db/awlt. Její datový model je pak ukazuje následující obrázek.

[5] Upraveno bylo schéma tabulek používaných v dotazech na „dbo“, aby se nemuselo neustále uvádět v SQL dotazech, a hodnoty některých záznamů, aby byla vracena více názorná data u všech zde probíraných typových SQL dotazů.


Address
AddressID int IDENTITY PK
AddressLine1 nvarchar(60)
AddressLine2 nvarchar(60) NULL
City nvarchar(30)
StateProvince nvarchar(50)
CountryRegion nvarchar(50)
PostalCode nvarchar(15)
ModifiedDate datetime
SalesOrderHeader
SalesOrderID int IDENTITY PK
RevisionNumber tinyint
OrderDate datetime
DueDate datetime
ShipDate datetime NULL
Status tinyint
OnlineOrderFlag bit
SalesOrderNumber NULL AK
PurchaseOrderNumber nvarchar(25) NULL
AccountNumber nvarchar(15) NULL
CustomerID int FK
ShipToAddressID int NULL FK
BillToAddressID int NULL FK
ShipMethod nvarchar(50)
CreditCardApprovalCode varchar(15) NULL
SubTotal money
TaxAmt money
Freight money
TotalDue NULL
Comment nvarchar(max) NULL
ModifiedDate datetime
CustomerAddress
CustomerID int PK FK
AddressID int PK FK
AddressType nvarchar(50)
ModifiedDate datetime
Customer
CustomerID int IDENTITY PK
NameStyle bit
Title nvarchar(8) NULL
FirstName nvarchar(50)
MiddleName nvarchar(50) NULL
LastName nvarchar(50)
Suffix nvarchar(10) NULL
CompanyName nvarchar(128) NULL
SalesPerson nvarchar(256) NULL
EmailAddress nvarchar(50) NULL
Phone nvarchar(25) NULL
PasswordHash varchar(128)
PasswordSalt varchar(10)
ModifiedDate datetime
SalesOrderDetail
SalesOrderID int PK FK
SalesOrderDetailID int IDENTITY PK
OrderQty smallint
ProductID int FK
UnitPrice money
UnitPriceDiscount money
LineTotal NULL
ModifiedDate datetime
Product
ProductID int IDENTITY PK
Name nvarchar(50) AK
ProductNumber nvarchar(25) AK
Color nvarchar(15) NULL
StandardCost money
ListPrice money
Size nvarchar(5) NULL
Weight decimal(8, 2) NULL
ProductCategoryID int NULL FK
ProductModelID int NULL FK
SellStartDate datetime
SellEndDate datetime NULL
DiscontinuedDate datetime NULL
ThumbNailPhoto varbinary(max) NULL
ThumbnailPhotoFileName nvarchar(50) NULL
ModifiedDate datetime
ProductModel
ProductModelID int IDENTITY PK
Name nvarchar(50) AK
CatalogDescription text NULL
ModifiedDate datetime
ProductModelProductDescription
ProductModelID int PK FK
ProductDescriptionID int PK FK
Culture nchar(6) PK
ModifiedDate datetime
ProductDescription
ProductDescriptionID int IDENTITY PK
Description nvarchar(400)
ModifiedDate datetime
ProductCategory
ProductCategoryID int IDENTITY PK
ParentProductCategoryID int NULL FK
Name nvarchar(50) AK
ModifiedDate datetime

2 SQL dotazy

Jazyk SQL (*Structured Query Language*) byl vytvořen jako univerzální prostředek pro komunikaci, respektive v jeho základní části pro dotazování se na data uložená v databázových systémech.

2.1 Základní dotazy

Struktura jazyka SQL byla navržena tak, aby se co nejvíce podobala přirozenému anglickému jazyku. Díky tomu, pokud víme, jaké údaje, z jakých tabulek databáze potřebujeme, stačí více méně jen tento požadavek přeložit do angličtiny, případně doladit pořadí této věty dané konvenci.

Například chceme-li *vybrat seznam jmen a příjmení z tabulky osoby seřazený abecedně podle těchto jmen*, stačí tento požadavek již jen drobně doladit a můžeme jej položit přímo databázi. Formálně (zatím česky) by tedy měl znít takto: *vyber jméno, příjmení z osob seřazený podle jmen*, anglicky pak (přeloženy jsou jen klíčová slova, nikoli názvy databázových objektů, tj. tabulek a jejich sloupců) *select Jmeno, Prijmeni from osoby order by Jmeno*. A první platný SQL dotaz máme hotový.

Základní struktura SQL dotazu je tedy následující:

- **select** *sloupce* nebo *
- **from** *tabulka*
  - **order by** *sloupce* [**asc** x **desc**]

Pro výběr všech údajů (sloupců) z tabulky **Product**, vzestupně seřazených dle názvů těchto produktů (sloupec **Name**) by pak SQL dotaz vypadal následovně.

```
1: /* Produkty seřazené dle názvů */
2: select *         -- sloupce zařazené do výběru (* = všechny)
3: from Product     -- tabulka, z níž se mají data vybírat
4: order by Name    -- řazení (nepovinná část dotazu)
```

Pokud chceme vybrat všechny sloupce z tabulky, nemusíme jejich názvy vpisovat a za **select** uvedeme pouze znak hvězdičky (*****), což je právě zástupný znak pro „všechny sloupce“.[6]

Část **order by** není povinná. Pokud není uvedena, jsou data seřazena dle primárního klíče hlavní dotazované tabulky.

Řadit data lze nejen vzestupně (tj. od A do Z, popř. 1...10 čili od nejnižších čísel po nejvyšší, u datumů pak od nejstarších po nejmladší), ale také obráceně, tedy sestupně (tj. od Z do A, u čísel 10…1 čili od nejvyšších po nejnižší, u datumů od nejnovějších po nejstarší). Vzestupně je anglicky *ascending*, sestupně pak *descending*. Zkratky používané v SQL dotazech jsou pak od těchto dvou slov odvozeny, tj. **asc** (*ascending* – vzestupně) a **desc** (*descending* – sestupně). Vzestupná varianta (**asc**) je přitom výchozí (defaultní), takže pokud směr řazení neuvedeme vůbec, použije se právě tento vzestupný směr.

Možné je taky řazení vícestupňové, tzn. že v případě kdy jsou hodnoty, podle kterých se řadí v první úrovni ve dvou či více řádcích stejné (např. osoby mající stejné příjmení), tak jejich pořadí mezi sebou lze určit řazením ve druhé (popř. třetí atd.) úrovně (např. osoby se stejným příjmením řadit dle křestního jména). Tyto úrovně (názvy sloupců) se vypisují za část **order**

[6] Použití * pro výběr všech sloupců místo jejich vyjmenování, může u větších databází zapříčinit jejich pomalejší načítání.

by oddělené čárkou, přičemž u každé z nich zvlášť lze definovat i směr řazení (**asc** či **desc** za název daného sloupce).

Následující kód ukazuje jak výběr pouze dvou sloupců (názvu produktu a jeho katalogové ceny) z tabulky **Product**, přičemž výstupní data řadí dle ceny sestupně (nejdražší položky budou nahoře, nejlevnější dole) a pokud by některé produkty měly stejnou cenu, budou seřazeny abecedně dle názvů vzestupně (do A do Z).

```
1: /* Seznam názvů produktů a jejich katalogové ceny seřazený
      od nejvyšší ceny. Jsou-li dvě ceny shodné, řadí se abecedně
      dle názvu. */
2: select Name, ListPrice
3: from Product
4: order by ListPrice desc, Name
```

Sloupec, podle jehož hodnot se řadí, lze definovat též číslem, které značí index vybíraného (selektovaného) sloupce (1–N), tj. pořadí sloupce uvedeného v sekci **select** (funguje i v případě *). Tato varianta je zvlášť výhodná, pokud je hodnota, podle které chceme řadit, vypočtená (matematickým výrazem či s použitím funkce), protože díky tomu nemusíme zápis tohoto výpočtu znovu celý vypisovat (což by ovšem fungovalo stejně).

Následující kód ukazuje výběr názvu, nákupní ceny produktu, katalogové (prodejní) ceny produktu a nakonec výši zisku z prodeje jednoho kusu tohoto výrobku, vypočítaného jako rozdíl obou předchozích částek. Dle této hodnoty je výstup také sestupně seřazen, takže „nejvýnosnější“ produkty jsou ve výsledku zařazeny jako první.

```
1:  /* Seznam názvů produktů seřazený sestupně dle zisku,
       který produkt přináší. */
2:  select Name, StandardCost, ListPrice, ListPrice – StandardCost
3:  from Product
4:  order by 4 desc  -- nebo order by ListPrice-StandardCost desc
```

Z vybíraných dat lze též vynechat duplicitní řádky (záznamy), tedy aby tam každý řádek výsledku s týmiž hodnotami byl maximálně 1x. Databázová tabulka sice nemůže obsahovat zcela duplicitní řádky již ze své podstaty, nicméně výběr se může týkat pouze některých jejích sloupců, kde již takovouto shodu vyloučit nelze (např. dvě osoby s týmž jménem a příjmením, vybíráme-li pouze tyto dva sloupce). Predikát **distinct** uvedený za slovem **select** zajistí, aby ze všech duplicitních řádků výstupu zůstal vždy právě jeden. Jelikož jsou zcela identické, je jedno, který z nich ponechá.

Následující kód ukazuje, jak získat křestní jména všech zákazníků, tak, aby každé z nich bylo zastoupeno maximálně jednou (např. pro nastavení posílání gratulací k jejich svátkům). Pokud by se vybíralo více hodnot, uvádí se **distinct** i tak pouze jednou za select (před název prvního sloupce).

```
1:  /* Všechna křestní jména všech zákazníků, každé ale jen 1x */
2:  select distinct FirstName -- distinct: stejné řádky jen 1x
3:  from Customer
4:  order by FirstName
```

2.1.1 Podmínky (where)

Podmínky slouží pro filtrování vybíraných záznamů (řádků). Do výsledku jsou tedy zahrnuty pouze ty záznamy, jež splňují zadanou podmínku.

Podmínka se uvádí za klíčovým slovem **where**, které se do SQL dotazu píše za výčet tabulek (za **from ...**). Výsledkem podmínky musí být logická hodnota **true** nebo **false** (*pravda* nebo *nepravda*).

```
1: /* Výběr všech údajů o objednávkách, které mají celkovou
      částku od 60k výše */
2: select *
3: from SalesOrderHeader
4: where TotalDue >= 60000
5: order by TotalDue
```

Podmínka se může skládat z více pod-podmínek složených operátorem **and** (a zároveň) nebo **or** (nebo), rozdělených/spojených případně i pomocí závorek.

```
1: /* Výběr všech zákazníků, kteří se jmenují 'David' a mají
      prostřední jméno 'J.' */
2: select *
3: from Customer
4: where FirstName = 'David' and MiddleName = 'J.'
```

Některé hodnoty záznamů uložených v databázových tabulkách nemusí být vždy definovány a místo konkrétního textu, čísla apod. je uvedeno znamení, že hodnota není vyplněna, což zajišťuje tzv. **null** hodnota. Tato „hodnota“ má ovšem trošku odlišné vlastnosti, pokud je použita ve výpočtech či porovnáních. Kdykoli, jakkoli a cokoli se porovnává s hodnotou **null** (např. `VEK = 18` nebo i `VEK <> 18`, kde **VEK** je **null**), pak výsledek je vždy **false** (nepravda). Je či není-li hodnota **null** je tak třeba testovat zvláštním porovnáním, přičemž pro tento test je zde zvláštní operátor **is** (např. `VEK is null`), popř. **is not** (`VEK is not null`).

Potřebujeme-li tedy zjistit, jestli nějaká hodnota, jež může být **null** (určeno v nastavení sloupce tabulky) je shodná s nějakou konstantou, vystačíme si s jedinou podmínkou, protože při porovnání s null bude výsledek **false**, což je vlastně správně (např. `VEK = 18`). Hledáme-li však hodnoty odlišné od této konstanty, a **null** považujeme také za odlišný, jsou již zapotřebí dvě spojené podmínky (např. `VEK <> 18 or VEK is null`).

Následující kód ukazuje výběr všech zákazníků Davidů, jež mají nějaké (jakékoli) prostřední jméno, tj. nemají u něho hodnotu **null**.

```
1: /* Výběr všech zákazníků, kteří se jmenují David a mají alespoň
      nějaké prostřední jméno */
2: select *
3: from Customer
4: where FirstName = 'David' and MiddleName is not NULL
```

Operátory **and** a **or** nemají stejnou prioritu. Matematicky by se dalo **and** přirovnat k násobení a **or** ke sčítání. Tedy **and** má přednost před **or** (nejdříve násobíme a až poté sčítáme výsledky tohoto násobení). Pokud potřebujeme vyhodnotit spojování podmínek v jiném pořadí, je třeba použít závorek.

Následující příklad ukazuje takovéto spojení podmínek, kdy první musí platit vždy a ze druhé a třetí stačí, pokud platí jen jedna z nich.

```
1: /* Výběr všech položek faktur, kde se prodalo nad 5ks
      a ID produktu bylo 988 nebo 715 */
2: select *
3: from SalesOrderDetail
4: where OrderQty > 5 and (ProductID = 988 or ProductID = 715)
```

Jestliže do výběru chceme zařadit, nebo z něj naopak vyřadit, více záznamů, jejichž sloupec odpovídá nejen jedné, ale více možným konkrétním hodnotám, není třeba pro každou z nich zapisovat zvláštní podmínku a spojovat je všechny operátorem **or**, ale lze využít dalšího operátoru **in** (hodnota nalevo je v seznamu hodnot napravo). Ten umožňuje na pravou stranu výrazu uvést do kulatých závorek více konstantních hodnot oddělených čárkou.

Následující kód ukazuje použití tohoto operátoru **in**.

```
1: /* Prodeje produktů 988 nebo 715 */
2: select *
3: from SalesOrderDetail
4: where ProductID in (988, 715)
```

Ve výčtu hodnot přitom nemusí být pouze číselné hodnoty, ale **in** lze použít pro libovolný typ hodnot (textové, datum atd., viz. násl. ukázka).

```
1: /* Všechna města (každé jen 1x), kde mají zákazníky ze státu
      Texas, Missouri nebo Oregon */
2: select distinct City
3: from Address
4: where StateProvince in ('Texas', 'Missouri', 'Oregon')
5: order by City
```

Hodnoty pro testování **in** operátorem přitom nemusí být jen přímo zadány v dotazu, ale lze je definovat i tzv. vnořeným selektem…

2.1.2 Vnořené dotazy v podmínkách (in)

Dotazy **select** lze do sebe i vnořovat, resp. výsledek jednoho (vnořeného) použít jako data v podmínce druhého (hlavního, tj. toho, do kterého je ten první vnořen). V podmínce **where** se tak můžeme odkázat na seznam hodnot přes operátor **in**, přičemž seznam těchto hodnot pro porovnání získáme

právě přes vnořený **select**. Ten by měl vracet vždy pouze jeden sloupec hodnot stejného typu (vektor).

Následující kód ukazuje výběr názvů produktů, jejichž **ID** se alespoň jednou vyskytlo v tabulce s položkami objednávek (tj. byly alespoň jednou prodány, resp. objednány).

```
1: /* Názvy všech produktů, které se již alespoň 1x prodaly */
2: select Name
3: from Product
4: where ProductID in (
5:   select ProductId
6:   from SalesOrderDetail
7: )
```

Vnořený **select** může data čerpat z jiné tabulky než hlavní dotaz a může také používat vlastní podmínky **where**. Zbytečné až kontraproduktivní je ovšem jeho data řadit (**order by**) nebo činit jedinečnými (**distinct**).

```
1: /* Abecedně seřazené názvy firem zákazníků, kteří někdy
      udělali objednávku nad 50k */
2: select CompanyName
3: from Customer
4: where CustomerID in (
5:   select CustomerID
6:   from SalesOrderHeader
7:   where TotalDue > 50000
8: )
9: order by CompanyName
```

Stejný princip používá i další příklad.

```
1: /* Všechny objednávky doručované do státu England */
2: select *
3: from SalesOrderHeader
4: where ShipToAddressID in (
5:   select AddressID
6:   from Address
7:   where StateProvince = 'England'
8: )
```

2.2 Seskupování dat

Seskupování umožňuje sloučit více řádků tabulky dohromady a to tak, že je buď sloučí do jediného záznamu (řádku), nebo je rozdělí do kategorií definovaných určitým sloupcem/sloupci a pro každou z nich hodnoty jiného sloupce sloučí pomocí tzv. agregační funkce. Kategorizující sloupce musí být vypsány za klíčovým výrazem **group by**. Klauzule **group by** (případně **having**) se zapisují za **where**, před **order by**.

2.2.1 Agregační funkce

Agregační funkce tedy více hodnot slučuje (agreguje) do hodnoty jediné. Základní agregační funkce jsou tyto:

- **sum** – součet
- **avg** – průměr
- **min** – minimum
- **max** – maximum
- **count** – počet

Ukázky v této podkapitole budou pro zjednodušení agregovat data vždy z daného sloupce celé tabulky (všech jejích záznamů). Jejich výstupem tak bude vždy právě jeden záznam s jedním či více sloupci.

Funkce **sum** (suma) spočítá součet všech numerických hodnot ze sloupce zadaného jako její jediný vstupní parametr. V následující ukázce tedy součet celkových částek (**TotalDue**) všech objednávek (tabulka **SalesOrderHeader**). Případné **null** hodnoty do součtu nezapočítává (přeskakuje je). Pokud je tabulka prázdná (popř. nevyhovuje-li žádný záznam případné podmínce), nebo jsou-li všechny hodnoty **null**, výsledkem této funkce bude také **null**.

```
1: /* Součet celkových částek všech objednávek */
2: select sum(TotalDue)
3: from SalesOrderHeader
```

Funkce **avg** (zkratka z anglického *average*, tj. česky *průměr*) vypočítá aritmetický průměr (součet hodnot vydělený jejich počtem) ze všech hodnot zadaného sloupce. Případné **null** hodnoty do průměru také nezapočítává, tzn. pro výpočet jimi nenavyšuje součet ani počet. Pokud funkce nemá žádné záznamy pro výpočet, nebo jsou-li všechny hodnoty **null**, výsledkem bude opět **null**.

```
1: /* Průměr z celkových částek všech objednávek */
2: select avg(TotalDue)
3: from SalesOrderHeader
```

Dvojice funkcí pro určení extrémní hodnoty **min** a **max** vrací nejnižší či nejvyšší hodnotu v daném sloupci. I tyto funkce přeskakují hodnoty **null**, popř. vrátí takový výsledek, nenajdou-li žádnou platnou hodnotu.

```
1: /* Minimum a maximum z celkových částek všech objednávek */
2: select min(TotalDue), max(TotalDue)
3: from SalesOrderHeader
```

Funkce pro určení počtu záznamů **count** se od těch předchozích trochu odlišuje možnými způsoby svého použití. V základu totiž nepotřebuje znát žádný konkrétní sloupec, a počítá veškeré záznamy v tabulce. V tom případě se jí do závorek jako parametr místo sloupce uvede znak hvězdičky (*****), podobně jako se u příkazu **select** zapsalo, že chceme vybrat všechny sloupce tabulky.

```
1: /* Počet všech objednávek */
2: select count(*)
3: from SalesOrderHeader
4: /* Počet všech zákazníků */
5: select count(*)
6: from Customer
```

Ovšem i funkce **count** lze místo hvězdičky určit sloupec, na jehož hodnoty se má zaměřit. V takovém případě počítá pouze záznamy, které nemají v daném sloupci hodnotu **null**. Dalším rozdílem je také to, že i pokud je tabulka prázdná, nebo jsou všechny hodnoty v testovaném sloupci **null**, tak stejně funkce **count** vrací vždy platnou hodnotu, v těchto případech 0, a nikoli **null**.

```
1: /* Počet všech zákazníků majících prostřední jméno */
2: select count(MiddleName)
3: from Customer
```

S výsledky agregačních funkcí pak lze provádět i základní matematické operace.

```
1: /* Počet všech zákazníků NEmajících prostřední jméno */
2: select count(*) - count(MiddleName)
3: from Customer
```

Určení názvu sloupce, jehož „ne**null**ové“ hodnoty má funkce **count** počítat, lze též opatřit predikátem **distinct**, který podobně jako u příkazu **select** vynechal z výběru veškeré duplicitní záznamy, i zde zajistí, aby každá opakující se hodnota v daném sloupci byla započítána pouze 1x.

```
1: /* Počet jedinečných křestních jmen zákazníků */
2: select count(distinct FirstName)
3: from Customer
```

2.2.2 Aliasy názvů sloupců (as)

Používání agregačních funkcí, ale i jiných výpočtů, či funkcí SQL serveru (viz kap. 3.3) přináší do výstupních hodnot jistý problém, a to, jak ve výstupu takto vypočítané sloupce pojmenovat. Dokud se vybíral pouze konkrétní sloupec z tabulky, sloupec ve výstupu byl pojmenován po něm. To však již u sloupců zaobalených do funkcí neplatí, takže takovéto sloupce název nemají definován a např. v SSMS (SQL Server Management Studio) jsou nadepsány jako „*(No column name)*“.

Pokud aplikace, která data zpracovává přistupuje k výsledkům dotazů přes indexy sloupců, nemusí tento stav ničemu vadit. V případě, že se ovšem data načítají přes názvy sloupců, je nezbytné jim nějaký název přiřadit.

K tomu se používají tzv. aliasy (česky by se také dalo říci *přezdívky*), které lze přiřadit nejen těmto nepojmenovaným sloupcům výstupu, ale i klasickým jednoduchým sloupcům, vybíraným z tabulek bez jakékoli úpravy. Pro jejich použití u názvů sloupců stačí tento alias napsat za fyzický název sloupce, popř. za funkci či výraz, jež výstupní hodnotu počítá. Tento alias by měl navíc být oddělen od této definice klíčovým slovem **as** (*jako*), nicméně většina databázových systémů, včetně SQL Serveru, si s pojmenováním sloupců poradí i bez **as**.

Následující kód ukazuje výběr hned několika hodnot vypočtených z celkové částky objednávek, přičemž každá je pomocí aliasu vhodně pojmenována.

```
1: /* Součet, průměr, minimum, maximum a počet celkových částek
      všech objednávek */
2: select sum(TotalDue) as Soucet,  -- "as" není nezbytné
3:        avg(TotalDue) Prumer,     -- alias funguje i bez něho
4:        min(TotalDue) Minimum,
5:        max(TotalDue) Maximum,
6:        count(*)      PocetObjednavek
7: from SalesOrderHeader
```

2.2.3 Filtr zdrojových záznamů (where)

Filtrování záznamů (řádků tabulky), tedy omezení výpočtů agregačních funkcí pouze na některé záznamy výběru, se provádí stejně jako u prostého výběru těchto dat (viz kap. 2.1.1). Podmínka, kterou musí záznamy splňovat, aby byly do výpočtu zahrnuty, se i zde uvádí za klíčovým slovem **where**.

```
1: /* Suma celkových čátek objednávek za položku s ID 864 */
2: select sum(LineTotal) Soucet
3: from SalesOrderDetail
4: where ProductID = 864
```

Ač je podmínka uvedena až za částí **from**, záznamy jsou vyfiltrovány ještě před samotným výpočtem funkce, takže ta již zpracovává pouze tento podmínce vyhovující pod-výběr.

```
1: /* Počet objednávek s částkou nad 50k */
2: select count(*) Pocet
3: from SalesOrderHeader
4: where TotalDue > 50000
```

I zde lze používat libovolně složité podmínky obsahující jakékoli podporované matematické výpočty či funkce, a tyto podmínky dále spojovat pomocí logických operátorů **and** a **or**.

Následující kód například vybírá pro určení počtu záznamů pouze takové, jejichž datum poslední změny spadá do roku 2006. Vhodně nadefinovaná podmínka zahrnuje i veškeré možné časy krajních dnů (včetně např. těžko zapsatelné poslední sekundy Silvestra 31.12.2016 23:59:59.9999999).

```
1: /* Počet adres upravených v roce 2006 */
2: select count(*)
3: from Address
4: where ModifiedDate >= '2006-01-01'
5:   and ModifiedDate <  '2007-01-01'
```

2.2.4 Seskupování podle jiné hodnoty (group by)

Doposud byl výstupem agregovaných hodnot pouze jediný záznam (řádek) s jedním či více taktéž agregační funkcí vypočítaných hodnot (sloupců). Seskupovat ovšem nemusíme pouze celé tabulky, či jejich podmínkou omezené pod-výběry, ale data lze seskupovat i podle *jiných hodnot*.

Řekněme, že by nás například zajímalo, kolik zákazníků (resp. adres) evidujeme v jednotlivých státech. Pokud bychom zůstali u předchozího způsobu dotazování, museli bychom pro každý stát udělat zvláštní dotaz s funkcí **count** a podmínkou omezující se na daný stát (např. `where StateProvincie = 'Texas'`).

SQL naštěstí disponuje možností, jak počty pro všechny státy získat najednou, tedy jediným dotazem a do jediného výstupu. Ten by měl mít dva sloupce: název státu a počet adres, které se v něm nacházejí. Počet zjistíme

snadno agregační funkcí `count(*)`. Název státu ale nijak seskupovat nechceme, naopak potřebujeme seskupovat *podle* názvu státu. Ten tedy do části **select** zapíšeme tak jak je (**StateProvince**).

Pokud se ale takovýto příkaz (pouze s částí **select** a **from**) pokusíme spustit, databázový server vrátí chybové hlášení, že máme neplatný požadavek v **select** části, nebo nám chybí tzv. **group by** klauzule. Tudíž, buď umažeme z výběru sloupec s názvem státu, či jej také opatříme nějakou agregační funkcí, nebo přidáme onu část začínající příkazem **group by**.

```
1: /* Počet adres v jednotlivých státech */
2: select StateProvince,
3:        count(*) Pocet
4: from Address
5: group by StateProvince
```

V části **group by** (česky *seskup dle*) lze uvést názvy atributů (sloupců), podle jejichž hodnot se mají zbylá vybíraná agregovaná data seskupovat. Seskupením se rozumí sloučení všech záznamů (řádků), které mají v daném sloupci stejnou hodnotu, v tomto případě mají uvedený stejný název státu. Každý název státu v tabulce se vyskytující tak bude mít ve výstupu právě jeden řádek, pro který budou agregovány jeho zbylé hodnoty čili v tomto případě určen počet seskupených záznamů.

Z tohoto příkladu je také patrné, že je naprosto nezbytné, aby názvy států byly pokaždé uvedeny naprosto shodně. Pokud by některý zapisovatel udělal v názvu chybu, ať již záměrně (existuje více variant zápisu téhož), z neznalosti (neví správný zápis názvu), omylem (překlep), či z jiných vlivů (např. mezera před či za slovem, pevná mezera či tabulátor místo té klasické apod., tak ve výstupu tohoto seskupení bude mít takto „přejmenovaný“

stát vlastní řádek s vlastním počtem. I z tohoto důvodu je tedy žádoucí veškeré textové hodnoty dávat do vlastních číselníků a v datových tabulkách se na ně pouze odkazovat cizími klíči přes jejich ID.

Také při víceřádkovém seskupování lze do výběru zařadit libovolný počet funkcí, resp. výstupních hodnot, které pro dané skupiny záznamů počítají různé agregované hodnoty. Dle těchto vypočítaných hodnot lze navíc výstup i seřadit pomocí **order by** (opět až na úplném konci), kde se na výstupní sloupec, dle něhož chceme řadit, lze odkazovat jak plným uvedením dané agregační funkce, tak indexem tohoto sloupce, ale i jeho případným aliasem.

```
1: /* Celkové částky za jednotlivé produkty (ID produktů) */
2: select ProductID,
3:        sum(LineTotal) Celkem,
4:        count(*) Pocet
5: from SalesOrderDetail
6: group by ProductID
7: order by Celkem desc  -- nebo "sum(LineTotal) desc", či "2 desc"
```

Seskupovat lze nejen dle žádného (bez části **group by**) či jediného sloupce zdrojových hodnot, ale může jich být i vícero. V takovém případě je stačí všechny vypsat do části **group by**, v tom pořadí, v jakém má seskupení probíhat (pokud výstup seřadíme pomocí **order by**, je toto pořadí irelevantní).

```
1: /* Celkové částky za jednotlivé objednávky a produkty */
2: select SalesOrderID,
3:        ProductID,
4:        sum(LineTotal) Celkem,
5:        count(*) Pocet
6: from SalesOrderDetail
7: group by SalesOrderID, ProductID
```

Za **group by** může být uvedeno i více seskupovaných parametrů než jen ty, které jsou uvedeny v části select, byť takový výstup bude vracet poněkud neúplná a zavádějící data. Sloupce, které však v části **select** jsou, v **group by** uvedeny být musí, aby bylo možné příkaz vůbec spustit.

```
1: /* Počty adres v jednotlivých regionech a státech */
2: select CountryRegion,
3:        StateProvince,
4:        count(*) Pocet
5: from Address
6: group by CountryRegion, StateProvince
7: order by 1, 3 desc, 2
```

Seskupovat se ovšem nemusí pouze podle hodnot v konkrétních sloupcích, ale tyto hodnoty lze libovolně zkombinovat či přepočítat výrazem či funkcí (nikoli však agregační). Následující příklad používá funkci **month** pro určení čísla měsíce (1–12) z plnohodnotného data a času (v **ModifiedDate**), čímž vytvoří přehled počtů posledních úprav adres v jednotlivých měsících.

```
1: /* Počty změn adres v jednotlivých měsících */
2: select month(ModifiedDate) Mesic,
3:        count(*) PocetZmen
4: from Address
5: group by month(ModifiedDate)
6: order by PocetZmen desc
```

Tento kód určil počty změn bez ohledu na roky, díky čemuž lze třeba identifikovat měsíc či období v roce, kdy se adresy nejčastěji upravují. Pokud bychom tato data chtěli vyjádřit i za konkrétní měsíce v konkrétních rocích, muselo by se do části **group by** přidat na první místo ještě `year(ModifiedDate)`.

2.2.5 Filtr pracující s již agregovanými hodnotami (having)

Vstupní (zdrojové) záznamy (řádky) z tabulek jsme filtrovali pomocí podmínek v části **where** (viz kap. 2.1.1 a 2.2.3). Tyto podmínky však nebylo možné aplikovat na hodnoty, které ze zdrojových dat teprve vzniknou výpočtem přes agregační funkce, tedy na výstupní sloučené řádky. I s tímto požadavkem však SQL počítá.

Výstupní již sloučené řádky lze totiž filtrovat i podmínkou vztaženou k výsledným agregovaným datům. Tato podmínka (či více spojených podmínek) se uvádí do nepovinné části za klíčové slovo **having** (česky *mající*), bezprostředně následující za výčtem seskupovaných sloupců v části **group by** (resp. před **order by** či na konec, neřadí-li se data). Tyto podmínky by tak měly v každé své části používat nějakou agregační funkci (např. `sum(CENA) > 100`).

```
1: /* ID zákazníků, kteří již nakoupili zboží alespoň za 20k */
2: select CustomerID,
3:        sum(TotalDue) Castka
4: from SalesOrderHeader
5: group by CustomerID
6: having sum(TotalDue) >= 20000
7: order by Castka
```

V části **group by**, na rozdíl od **order by**, nelze používat zástupné odkazy na agregované sloupce z části select, jako jsou jejich indexy či aliasy, ale musí být uvedeno plné znění výpočtu dané hodnoty pomocí agregační funkce. V podmínkách nicméně mohou být použity i výpočty, které se v části `select` nevyskytují.

Podmínky pro zdrojová data (**where**) i pro data již seskupená (**having**) lze samozřejmě kombinovat, a v dotazu požívat obě časti současně, popř. v libovolné kombinaci.

```
1: /* Produkty, které vydělaly více než 10k a s ID větším než 800 */
2: select ProductID,
3:        sum(LineTotal) Celkem
4: from SalesOrderDetail
5: where ProductID > 800
6: group by ProductID
7: having sum(LineTotal) > 10000
```

2.3 Omezení počtu záznamů pro výstup

Sestavit SQL dotaz pro požadovaný výběr dat již umíme. Co když ale potřebujeme z tohoto výběru pouze některé záznamy? Můžeme si jej samozřejmě nechat vrátit celý, a přečíst či zpracovat pouze ty řádky výstupu, které potřebujeme, avšak tento přístup není příliš šetrný k síti, přes kterou musí data cestovat, ani k paměti, do které se data, byť třeba jen dočasně načtou.

Naštěstí jsou k dispozici další možnosti, jak můžeme výsledek výběru omezit již v rámci SQL dotazu. Zápis těchto požadavků se ovšem může v různých databázových systémech lišit, zde se zaměříme na syntaxi používanou SQL Serverem.

Nejprve jak tedy získat pouze prvních **N** (v ukázce 5) záznamů.

```
1: /* Jen prvních 5 záznamů */
2: select top 5 *
3: from Customer
4: order by CompanyName
```

Klauzule **top** uvedená hned za příkazem **select** tedy indikuje, že počet záznamů ve výstupu bude omezen na prvních **N**, přičemž hodnota, kolik jich má být, je uvedena bezprostředně za tímto slovem (např. `top 10`), pak teprve následuje výčet vybíraných sloupců. Pokud bychom potřebovali do dotazu zařadit ještě *jedinečnost*, pak predikát **distinct** zapíšeme ještě před klauzuli top (`select distinct top 5 *`).

Jestliže potřebujeme do výstupu vynechat nějaké záznamy z jeho začátku, je třeba použít zcela jiný způsob. Používá se pro to klauzule **offset**, která je v rámci zachování zdání ucelené věty za uvedením počtu přeskakovaných řádků doplněna slovem **rows** (např. `offset 20 rows`, tj. přeskoč 20 řádků) a na rozdíl od **top** se uvádí až na úplném konci dotazu, i za jinak vždy poslední definicí řazení. Tím tedy vynecháme daný počet řádků ze začátku výstupu.

Jakmile je však použit **offset**, nelze již v dotazu použít kolizní **top** pro omezení počtu řádků následujících za tímto přeskočením. Výběr daného počtu záznamů, počínaje určitým záznamem je ovšem velmi důležitá funkcionalita databázových serverů především pro webové aplikace, kde se na přehledy dat používá stránkování (např. třetí stránka o dvaceti záznamech znamená načtení záznamů s pořadím 61–80, tj. `offset 3*20 rows`).

```
1: /* Přeskočení prvních 5 záznamů */
2: select *
3: from Customer
4: order by CompanyName
5: offset 5 rows             -- Prvních 5 přeskoč
```

Pro tyto účely je nicméně k dispozici další nepovinná klauzule, která pro změnu nefunguje bez **offset** (ten bez ní však ano). Je jím **fetch next N rows**

only (*načíst pouze následujících N řádků*) kde **N** určuje, kolik záznamů následujících po těch přeskočených chceme ve výstupu mít (např. `fetch next 20 rows only`). Pokud požadujeme vrácení více řádků, než zbývá záznamů do konce tabulky, budou nám pochopitelně vráceny pouze ty existující v nižším počtu, než bylo požadováno.

```
1: /* Přeskočení prvních 5 záznamů a pak následující 10 */
2: select *
3: from Customer
4: order by CompanyName
5: offset 5 rows              -- Prvních 5 přeskoč
6: fetch next 10 rows only    -- Vrať jen následujících 10
```

2.4 Struktura rozšířeného SQL dotazu

Základní SQL dotaz musí tedy vždy obsahovat části **select** a **from**. Vše ostatní je nepovinné. Pokud však v dotazu použijeme veškeré doposud zmíněné části, bude jeho kompletní struktura následující:

- **select** [**distinct**] [**top** *N*] *sloupce* nebo ***
- **from** *tabulka*
  - **where** *podmínka(y) nad zdrojovými řádky*
  - **group by** *sloupce pro seskupení*
  - **having** *podmínka s agregacemi*
  - **order by** *sloupce, výpočty, indexy či aliasy* [**asc** x **desc**]
  - **offset** *M* **rows**
  - **fetch next** *N* **rows only**

Přitom se stále jedná o poněkud rozvinutější mírně kostrbatou anglickou větu typu: ***vyber*** *křestní jméno, kraj bydliště a počet* ***z*** *osob* ***kde*** *osoba bydlí v ČR,* ***seskupeno*** *dle kraje a jména,* ***mající*** *průměrný počet alespoň 100* ***seřazeno dle*** *kraje a jména* ***přeskoč*** *5* ***řádků*** *a* ***načti pouze*** *10* ***násl. řádků***.

3 Spojování tabulek

Tabulky lze v rámci SQL dotazů spojovat **horizontálně**, tj. spojovat jednotlivé záznamy z více tabulek (zvýší se počet sloupců), nebo **vertikálně**, tj. spojení výsledků více SQL selektů pod sebe (zvýší se počet řádků).

3.1 Horizontální spojování

Častější horizontální spojování lze realizovat mnoha způsoby. V každém se ale v základu rozšíří výčet tabulek v části dotazu **from**, čímž se navýší i počet sloupců z více tabulek, které je možné v různých částech zapojit do dotazu.

3.1.1 Aliasy tabulek

S aliasy sloupců jsme se setkali již dříve (viz kap. 2.2.2). Chystáme-li se do jednoho dotazu zanést dvě tabulky, je na čase se seznámit i s aliasy tabulek. Ty jsou zde důležité v případech, kdy dvě (či více) horizontálně spojovaných tabulek obsahují stejně pojmenované sloupce, na které se v dotazu potřebujeme odkazovat.

Nejprve je třeba zmínit, že vše může fungovat i bez aliasů tabulek. Bez nich je nezbytné u sporných (duplicitních) sloupců uvést i celý název tabulky oddělený tečkou. Vybíráme-li například název (v obou případech **Name**) ze spojení tabulek modelů (**ProductModel**) a kategorií (**ProductCategory**), pak by příkaz začínající `select Name …` nedokázal určit, který z názvů chceme vybrat. Bez aliasů, by v případě že potřebujeme název modelu, začínal tedy dotaz následovně: `select ProductModel.Name ….`

Aliasy nám umožňují předně mnohdy dlouhé názvy tabulek zkrátit, aby kvůli jejich neustálému opakování délka dotazu zbytečně nenarostla a nestala se méně přehlednou. Aliasy tabulek se, podobně jako tomu bylo u sloupců, zapisují bezprostředně za název tabulky v sekci **from**. I zde je

možné použít klíčového slova **as**, avšak také tentokrát vše funguje stejně dobře i bez něj (např. alias **c**: `from Customer as c` či `from Customer c`, a použije se jako `select c.Name …`).

3.1.2 Kartézský součin

Kartézský součin (*Cartesian product*) se dá zjednodušeně popsat jako „*každý s každým*“. Kupříkladu máme-li dvě množiny (tabulky), v první 3 prvky (záznamy) označené jako A, B a C, ve druhé pak 2 prvky (záznamy) označené jako 1 a 2, pak výsledkem jejich kartézském součinu bude seznam všech možných dvojic, tedy A1, A2, B1, B2, C1 a C3, o celkovém počtu 3·2=6 dvojic. A kdybychom k tomu přidali třetí množinu (tabulku) o čtyřech prvcích (záznamech), výstupem kartézského součinu by již bylo 3·2·4=24 trojic.

Tab. 1
A
B
C

x

Tab. 2
1
2

=

Kartézský součin	
A	1
A	2
B	1
B	2
C	1
C	2

Nejjednodušším způsobem, jak požadavek na kartézský součin získat přes SQL dotaz je tabulky vypsat v části **from** (oddělí čárkou a přidělí se jim aliasy) a ve výstupu pak bude každá možná kombinace všech řádků obou (či více) tabulek.

```
1: /* Kartézský součin (každý s každým) Produkty-Kategorie - v1 */
2: select p.Name Produkt,
3:        c.Name Kategorie
4: from Product p,
5:      ProductCategory c
```

Teno SQL dotaz tedy vrátí názvy všech produktů v kombinaci s názvy věch kategorií, což je při 295 produktech a 41 kategoriích celkem 12 095 kombinací, a tedy i řádků výstupu.

Téhož výsledku lze dosáhnout pomocí spojovací funkce pro kartézský součin **cross join** (*křížové spojení*). Jelikož tato varianta horizontálního spojení neobsahuje spojovací podmínku, dalo by se říct, že při základním spojení v podstatě jen toto slovní spojení nahrazuje čárku mezi názvy tabulek v části **from**.

```
1: /* Kartézský součin (každý s každým) Produkty-Kategorie - v2 */
2: select p.Name Produkt,
3:        c.Name Kategorie
4: from Product p
5:      cross join ProductCategory c
```

Při spojování databázových tabulek s několika sty či tisíci záznamy tímto způsobem pak vznikají velmi rozsáhlé výstupy třeba i o milionech řádcích, z nichž nás ale obvykle zajímá pouze malý počet z nich. Ty si pak už jen stačí vyfiltrovat pomocí vhodné spojovací podmínky.

Nejčastěji se jedná o propojení tabulek na základě cizích klíčů, tj. kdy se jedna tabulka (např. produkty) odkazuje na cizím klíčem na primární klíč tabulky jiné (např. kategorie produktů). Do části **where** je tedy třeba přidat podmínku, která výstup omezí jen na ty řádky, které splňují tuto spojovací podmínku, aby nám zůstaly pouze ty kombinace, které spolu souvisí (`a.FK=b.PK`, popř. `b.PK=a.FK`).

```
1: /* Názvy produktů a jejich kategorií - v1 */
2: select p.Name Produkt,
3:        c.Name Kategorie
4: from Product p,
5:      ProductCategory c
6: where c.ProductCategoryID = p.ProductCategoryID
7: order by p.Name
```

Ekvivalentně tutéž podmínku zapíšeme i v případě spojení pomocí **cross join**.

```
1: /* Názvy produktů a jejich kategorií - v2 */
2: select p.Name Produkt,
3:        c.Name Kategorie
4: from Product p
5:      cross join ProductCategory c
6: where c.ProductCategoryID = p.ProductCategoryID
7: order by p.Name
```

Ve výstupu tak zůstanou pouze řádky, kde se hodnota cizího klíče u produktu rovná primárnímu klíči kategorie. Ke každému produktu tak bude přiřazena přesně ta kategorie, do které je zařazen.

Je zde ovšem riziko, že pokud nebude produktu kategorie určena, tedy bude mít ve sloupci **ProductCategoryID** hodnotu **null**, pak porovnání s tímto řádkem produktu pro všechny kategorie vrátí **false** a tento produkt tak nebude zahrnut do výstupních dat. Pokud má tento sloupec nastavenu hodnotu jako povinnou (**not null**), pak takový případ samozřejmě nemůže nastat, nicméně je třeba tento fakt zkontrolovat, nebo počítat s tím, že ve výstupu budou pouze ty produkty, které mají přiřazeny kategorii. V ukázkových datech jsou např. 4 produkty s nepřiřazenou kategorií, takže ve výstupu jich bude pouze 291 řádků.

Analogicky pak můžeme horizontálně spojovat i více tabulek, například k předchozímu výběru přidat ještě název modelu produktu. Výstup lze také seřadit jakýmkoli z dříve zmíněných způsobů nebo jejich kombinací.

```
 1: /* Název produktu, kategorie a modelu - kartézský součin -
       pokud je v těchto vazbách NULL, budou řádky vynechány */
 2: select p.Name Produkt,
 3:        c.Name Kategorie,
 4:        m.Name Model
 5: from Product p,
 6:      ProductCategory c,
 7:      ProductModel m
 8: where c.ProductCategoryID = p.ProductCategoryID
 9:   and m.ProductModelID = p.ProductModelID
10: order by p.Name, Kategorie, 3
```

3.1.3 Vnitřní spojení

Vnitřní spojení (**inner join**, popř. pouze **join**, obě varianty zápisu jsou totožné) je v podstatě ekvivalentem pro kartézský součin (**cross join**) s tím, že po nás bude povinně vyžadovat spojovací podmínku, kterou jsme dříve zapisovali až do části **where**. Použijeme-li pak tutéž podmínku (`a.FK = b.PK`) bude i výsledek zcela shodný, tedy včetně toho, že produkty bez kategorie (`p.ProductCategoryID is null`) budou ve výstupu zcela vynechány.

U spojení vnitřního (i vnějšího) se podmínka zapisuje hned za názvem a aliasem připojované tabulky, kde se zahajuje klíčovým slovem **on**. I zde může být podmínka složena z více částí pomocí **and** či **or**. Podmínka v části **where** je pak již aplikována na výsledek tohoto spojení, které již dále ovlivnit nemůže.

```
1: /* Názvy produktů a jejich kategorií - v3 */
2: select p.Name Produkt,
3:        c.Name Kategorie
4: from Product p
5:      inner join ProductCategory c    -- join = inner join
6:              on c.ProductCategoryID = p.ProductCategoryID
7: order by p.Name
```

3.1.4 Vnější spojení

Vnější spojení (**outer join**) je nejuniverzálnější, protože nevynechává případy, kdy některé záznamy tabulky nemají své „partnery“ v tabulce připojované. Pokud není „partnerský“ záznam nalezen, mají veškeré jeho sloupce hodnoty **null**. Pokud je ovšem v připojované tabulce nalezeno více záznamů vyhovujících spojovací podmínce, záznam z výchozí tabulky bude pro každý z nich duplikován.

V příkazu připojení je tedy třeba také určit, která tabulka je v dotazu ta hlavní (každý její záznam bude ve výstupu vždy minimálně jednou), a která se na ni napojuje (použity budou jen záznamy, které si vyžádá hlavní tabulka přes cizí klíč, popř. **null**). Varianty jsou dvě:

1) **left join** (totéž jako **left outer join**) – hlavní tabulka je nalevo (první zapsaná), k ní připojovaná tabulka je napravo (druhá zapsaná),

```
1: /* Názvy produktů a jejich kategorií, kde kategorie není,
      uvést v jejím názvu NULL */
2: select p.Name Produkt,
3:        c.Name Kategorie
4: from Product p
5:      left join ProductCategory c -- left join = left outer join
6:             on c.ProductCategoryID = p.ProductCategoryID
7: order by p.Name
```

2) **right join** (totéž jako **right outer join**) – hlavní tabulka je napravo (druhá zapsaná), k ní připojovaná tabulka je nalevo (první zapsaná).

Pokud chceme stejný výsledek jako v předchozím příkladě s **left join**, musíme prohodit pořadí tabulek.

```
1: /* Názvy produktů a jejich kategorií, kde kategorie není,
      uvést v jejím názvu NULL */
2: select p.Name Produkt,
3:        c.Name Kategorie
4: from ProductCategory c
5:      right join Product p -- right join = right outer join
6:              on c.ProductCategoryID = p.ProductCategoryID
7: order by p.Name
```

Ponecháme-li však pořadí tabulek beze změny, získáme zcela odlišný a pravděpodobně ne úplně chtěný výstup (názvy kategorií a jejich produktů, kde, pokud produkt nemá kategorii, nebude ve výstupu uveden, pokud v kategorii žádný produkt není, kategorie bude 1x uvedena s názvem produktu **null**).

```
1: /* Názvy kategorií a jejich produktů */
2: select p.Name Produkt,
3:        c.Name Katgorie
4: from Product p
5:      right join ProductCategory c
6:              on c.ProductCategoryID = p.ProductCategoryID
7: order by p.Name
```

Z příkladů pro **left join** a **right join** s týmž výstupem je patrné, že lze obě varianty snadno zaměňovat při prohození pořadí tabulek. Při sestavování dotazuje je pro nás ovšem přirozenější postupovat zleva doprava, tedy ve

směru, ve kterém čteme. Proto se dále zaměříme především na nejuniverzálnější variantu **left join**, s jejíž pomocí lze vyřešit drtivou většinu všech případů spojení tabulek.

Pro úplnost ještě zmiňme možnosti použití plného vnějšího spojení (**full outer join** či **full join**). V něm jsou hlavní obě tabulky, tzn. ve výstupu budou zahrnuty všechny řádky z obou tak, že pokud se je spojovací podmínkou podaří propojit, půjde o jeden řádek ve výstupu. Zbylé řádky, které se propojit nepodařilo (ať již z levé, či pravé tabulky), budou přidány s hodnotami **null** ve sloupcích té druhé tabulky. Např. v ukázkové databázi, kde 4 z 295 produktů kategorii nemají a 4 kategorie neobsahují žádné produkty tak ve výstupu bude 299 řádků.

```
1: /* Názvy všech produktů (i těch bez kategorie) a jejich
      kategorií (všech, i těch bez produktu) */
2: select p.Name Produkt,
3:        c.Name Katgorie
4: from Product p
5:      full join ProductCategory c -- full join = full outer join
6:             on c.ProductCategoryID = p.ProductCategoryID
7: order by p.Name
```

Spojujeme-li více tabulek než jen dvě tabulky, funguje to tak, že se nejprve propojí první dvě a ta třetí se napojuje až ke vzniklému výsledku (popř. tento výsledek ke třetí tabulce v případě *right join*).

```
 1: /* Název produktu, jejich modelu a kategorie */
 2: select p.Name Produkt,
 3:        m.Name Model,
 4:        c.Name Kategorie
 5: from Product p
 6:      left join ProductModel m
 7:             on m.ProductModelID = p.ProductModelID
 8:      left join ProductCategory c
 9:             on c.ProductCategoryID = p.ProductCategoryID
10: order by 1, 2, 3
```

Čtvrtá tabulka se připojuje ke vzniklému propojení tří tabulek atd. Prioritu propojování lze případně změnit pomocí závorek.

```
 1: /* Název produktu, jeho modelu, kategorie a hlavní kategorie */
 2: select p.Name Produkt,
 3:        m.Name Model,
 4:        c.Name Kategorie,
 5:        c2.Name HlavniKategorie
 6: from Product p
 7:      left join ProductModel m
 8:             on m.ProductModelID = p.ProductModelID
 9:      left join ProductCategory c
10:             on c.ProductCategoryID = p.ProductCategoryID
11:      left join ProductCategory c2
12:             on c2.ProductCategoryID = c.ParentProductCategoryID
13: order by 1, 2, 3, 4
```

Pokud má tabulka referenci na sebe samu, lze v ní vytvářet hierarchické struktury, v podstatě neomezenou hloubkou této struktury (např. podobně jako jsou do sebe zanořovány složky na disku počítače). V ukázkové databázi jsou data používající pouze dvouúrovňovou hierarchii kategorií produktů, byť daná struktura umožňuje zapsat i hierarchii hlubší.

```
1: /* Kategorie a jejich nad kategorie */
2: select c2.Name HlavniKategorie,
3:        c.Name Kategorie
4: from ProductCategory c
5:      left join ProductCategory c2
6:             on c2.ProductCategoryID = c.ParentProductCategoryID
7: order by 1, 2
```

V horizontálním spojení tabulek lze také používat seskupování (**group by**) a agregační funkce.

```
1: /* Názvy kategorií a počet jejich podkategorií */
2: select c2.Name NazevKategorie,
3:        count(*) PocePodkategorii
4: from ProductCategory c
5:      left join ProductCategory c2
6:             on c2.ProductCategoryID = c.ParentProductCategoryID
7: group by c2.Name
8: order by 1, 2
```

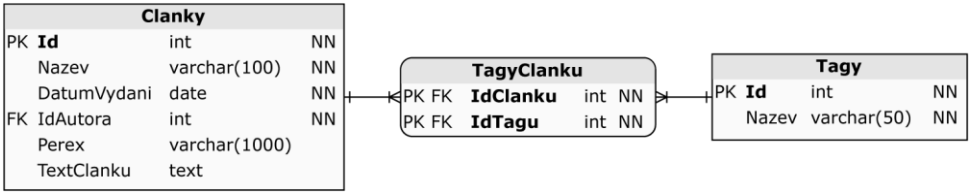
Vnější spojení je obvykle využíváno především pro spojení dat přes cizí a primární klíč ve vztahu s kardinalitou **1:N**, kdy u **left join** je **N** je vlevo a **1** vpravo (k produktu hledáme jednu jeho kategorii, nikoli ke kategorii produkt, kterých v ní může být více), popř. u vztahů **1:1**. Spojování u těchto vztahů tedy začínáme od tabulky, která má početnější kardinalitu.

Vztahy **M:N** (např. články a tagy[7], kdy jeden článek může mít více tagů a jedním tagem může být označeno více článků) se realizují přes spojovací tabulku (např. *TagyClanku*). V těchto případech ale většinou pracujeme jen

[7] Tag, nebo též štítek či značka, u článku označuje kategorie (témata či klíčová slova), do kterých jej lze zařadit (např. domácí zprávy, ekonomika, zahraniční, kultura apod.), přičemž jeden článek může spadat i pod více složek současně.

s jednou stranou tohoto spojení (např. pro konkrétní článek chceme vypsat jeho tagy, nebo pro konkrétní tag chceme seznam článků jím označených). Pokud z nějakého důvodu potřebujeme vypsat vše (např. názvy všech článků a všech jejich tagů), lze se spojováním začít ze kterékoli tabulky s tím, že existují-li nepropojené záznamy (např. články bez tagů nebo tagy bez článků), tak v přehledu mohou různé údaje scházet (místo **left join** je použit znak šipky →):

- **Clanky** → **TagyClanku** → **Tagy** – ve výstupu budou všechny články, z tagů jen ty přiřazené alespoň k jednomu z nich (nepoužité tagy budou chybět).
- **Tagy** → **TagyClanku** → **Clanky** – ve výstupu budou všechny tagy, z článků jen ty s alespoň jedním tagem (články bez tagů budou chybět).
- **TagyClanku** → **Clanky** → **Tagy** (pořadí druhé a třetí tabulky lze zaměnit s týmž výsledkem) – ve výstupu budou jen ty články (resp. tagy), které jsou propojeny s alespoň jedním tagem (resp. článkem), všechny články bez tagů i tagy bez článků budou chybět (stejný výsledek by byl získán i s použitím kartézského součinu a týchž spojovacích podmínek).



V ukázkové databázi jsou tímto způsobem propojeny například zákazníci a adresy nebo modely a jejich popisy. V obou případech jde sice o vztah **M:N** realizovaný přes spojovací tabulku, ale zjevně je počítáno spíše se spojením 1:N (jeden zákazník má více adres, ale jedna adresa není přiřazena

více zákazníků; jeden model má více popisů v různých jazycích, ale týž popis není určen více modelům), které by bylo možné vytvořit i bez spojovacích tabulek.

V případě adres spojovací tabulka navíc určuje typ adresy zákazníka (fakturační/doručovací, což mohlo být zahrnuto i v tabulce **Address**). Pokud potřebujeme pro konkrétní zákazníky určit pouze jednu konkrétní adresu (fakturační), lze použít rozšířenou spojovací podmínku (viz násl. ukázka). Spojovací tabulka však nemá unikátní index na kombinaci ID zákazníka a typu adresy, takže nelze vyloučit, že by adres téhož typu mohlo být pro jednoho zákazníka uloženo více.

```
 1: /* Názvy firem a státy, ve kterých mají hlavní sídlo */
 2: select c.CompanyName,
 3:        a.StateProvince
 4: from Customer c
 5:      left join CustomerAddress ca
 6:             on ca.CustomerID = c.CustomerID and
 7:                ca.AddressType = 'Main Office'
 8:      left join Address a on a.AddressID = ca.AddressID
 9: where a.AddressID is not null
10: order by c.CompanyName
```

Vždy je tedy důležité, ze které ze spojovaných tabulek potřebujeme získat všechny záznamy (a tou začít) a které další budou pouze doplňovat její hodnoty. Jde-li nám o pouze objednané zboží, začneme od detailu objednávek.

```
 1: /* Kód objednávky, název firmy a počet položek v objednávce */
 2: select h.PurchaseOrderNumber,
 3:        c.CompanyName,
 4:        count(*) as PocetPolozek
 5: from SalesOrderDetail d
 6:      left join SalesOrderHeader h
 7:             on h.SalesOrderID = d.SalesOrderID
 8:      left join Customer c
 9:             on c.CustomerID = h.CustomerID
10: group by h.PurchaseOrderNumber, c.CompanyName
```

3.2 Vertikální spojování

Vertikální spojování je realizováno dvěma či více samostatnými SQL dotazy, které jsou propojeny klíčovým výrazem **union all**. Pro oba vertikálně spojované dotazy platí několik pravidel:

- názvy (aliasy) sloupců pro výstup určují jejich názvy (aliasy) v prvním dotazu,
- počet sloupců v obou dotazech musí být stejný,
- není-li v některém ze spojovaných dotazů relevantní nějaký sloupec, který v jiných spojovaných dotazech je, lze jej v selektu doplnit konstantou nebo hodnotou **null**,
- v jednom sloupci nelze kombinovat různé datové typy, pokud to však potřebujeme, je třeba je konvertovat či převést na stejný datový typ (např. datum a číslo – datum lze vyjádřit číslem, popř. obojí může být převedeno na text),
- ve všech spojovaných dotazech lze používat veškeré jeho možnosti (podmínky, horizontální spojování, seskupování apod.), kromě řazení,
- řazení (**order by**), pokud je zapotřebí, musí být až na konci celého dotazu a řadí tak celý (již spojený) výstup.

Díky vertikálnímu spojení dvou v podstatě nezávislých dotazů můžeme do jediného výstupu a téhož sloupce dostat hodnoty záznamů z různých tabulek, což by jiným způsobem efektivně nebylo možné.

Následující dotaz například slučuje do výstupu jedinečné názvy firem a názvy produktů. Aby bylo možné rozlišit, ze které tabulky, který řádek pochází, jsou jim přidány textové konstanty identifikující jejich původ, je-li tedy tato informace pro výstup vůbec relevantní.

```
 1: /* Názvy firem a produktů */
 2: select distinct CompanyName Nazev,
 3:        'Firma' Typ -- aliasy v prvním dotazu určí názvy sloupců
 4: from Customer
 5:
 6: union all          -- vertikální spojení dovou dotazů
 7:
 8: select Name,
 9:       'Produkt'    -- konstanta, rozlišující odkud řádek pochází
10: from Product
11:
12: order by 1         -- seřadí až výsledné spojení obou selektů
```

Jak již bylo řečeno, oba spojované dotazy mohou využívat všech dříve (i dále) probraných funkcionalit, včetně seskupování a horizontálního spojování tabulek.

```
 1: /* Názvy kategorií a počet jejich produktů
       s názvy modelů a počtem jejich produktů */
 2: select c.Name Nazev, count(*) Pocet, 'Kategorie' Druh
 3: from Product p
 4:      left join ProductCategory c
 5:             on c.ProductCategoryID = p.ProductCategoryID
 6: group by c.Name
 7:
 8: union all
 9:
10: select m.Name, count(*), 'Model'
11: from Product p
12:      left join ProductModel m
13:             on m.ProductModelID = p.ProductModelID
14: group by m.Name
15:
16: order by Nazev
```

3.3 Vnořené dotazy jako zdroje dat

Výběr dat získaný dotazem nemusí být tím finálním výsledkem, který od databázového serveru dostaneme. Tento výsledek může totiž posloužit jako virtuální zdroj dat pro další dotaz. Potřebujeme-li postupovat takto, pak zdrojový pod-dotaz „zabalíme“ do kulatých závorek a uvedeme jej v sekci **from** u hlavního nad-dotazu, místo názvu zdrojové tabulky.

Jelikož tento mezivýsledek je výsledkem dotazu, a nikoli existující tabulkou, byť by třeba čerpal jenom z jediné, nemá svůj název, který by nad-dotaz mohl při odkazech na něj používat, tudíž musí mít povinně určen alias (v násl. příkladu je jím **x**).

Při vertikálním spojení dvou dotazů například nelze pro výsledek tohoto spojení použít omezení počtu záznamů **top** ani **offset + fetch**. Pokud vý-

sledný výběr použijeme zdroj dat pro nad-dotaz, můžeme jeho řádky jednak vzájemně promíchat jejich seřazením dle zvoleného klíče, a zároveň i vybrat pouze ty řádky, které potřebujeme.

```
1: /* Selekt z výsledku jiného selektu */
2: select top 5 Nazev
3: from (select distinct StateProvince Nazev
4:       from Address
5:       union all
6:       select distinct CountryRegion
7:       from Address) x         -- Vnořený selekt musí mít alias
8: order by 1
```

U vnořených dotazů v části **from** nelze logicky používat odkazy na případné další tabulky z nad-dotazu. K výsledku pod-dotazu však lze horizontálně připojit další tabulky pomocí všech dříve uvedených technik.

Vnořený dotaz tedy může být v části **where** jako zdroj hodnot pro porovnání operátorem **in** (viz kap. 2.1.2), v části **from** jako zdroj dat pro výběr, a pak také v části **select** jako hodnota pro konkrétní sloupec výsledku.

V takovém případě se vnořený pod-dotaz, opět zaobalený do kulatých závorek zapíše do části **select** jako kdyby se jednalo o název sloupce ze zdrojové tabulky, přičemž lze tomuto sloupci přiřadit také alias (viz kap. 2.2.2). V pod-dotazech tohoto typu se, stejně jako u těch ve **where** části, lze odkazovat na zdrojové tabulky z nadřazeného dotazu přes jejich aliasy.

```
1: /* Selekt jako hodnota ve sloupci nadřazené selekt části
      (musí vracet právě 1 záznam s právě 1 sloupcem) */
2: select p.Name,
3:        (select c.Name
4:         from ProductCategory c
5:             where c.ProductCategoryID = p.ProductCategoryID
6:        ) Kategorie
7: from Product p
```

Uvedený příklad by se samozřejmě dal řešit také pomoci klasického horizontálního spojení (**left join**), nicméně v určitých případech má i tento postup své nezastupitelné místo.

4 SQL funkce

Funkce, které je možné v rámci dotazů používat, nejsou z převážné části nijak standardizované, či součástí původní normy SQL jazyka. Většina databázových systémů jich sice méně či více podporuje, některé dokonce pod stejným či podobným názvem, ale mohou se tak systém od systému lišit. V této kapitole si představíme funkce, které nabízí vícero databázových systémů, v syntaxi SQL Serveru. Budete-li tedy některou z nich potřebovat i někde jinde, je důležité vědět, že taková funkce vůbec existuje (popř. může existovat) a její konkrétní název a parametry se již snadno dohledají.

4.1 Funkce pro práci s hodnotou NULL

S hodnotou **null**, která značí, že daná hodnota záznamu není definována, jsme se setkali již dříve (viz kap. 2.1.1). Nyní se podíváme na funkce, které dokáží usnadnit nejčastější problémy, které tato hodnota přináší.

Jednou z potíží s **null** hodnotou je, že jakmile je součástí jakéhokoli výpočtu, výsledkem je vždycky také **null** (např. sčítáme-li něco s nedefinovanou hodnotou, nemůžeme určit ani hodnotu výsledku). Celkem často se přitom předpokládá, že u číselných výpočtů se za null dosadí 0, u slučování textů prázdná textová hodnota `''`, což se nicméně neděje.

```
1: -- NULL se neřeší, kde je MiddleName NULL, bude i FullName NULL
2: select FirstName + ' ' +
3:        MiddleName + ' ' +
4:        LastName as FullName
5: from Customer
```

Existuje však funkce, která tuto záměnu dokáže provést. Její název je **Coalesce** a vstupními parametry je hodnota, která se má otestovat na **null**

a v případě, že definovaná je, tak se použije, a pak hodnota, která se má použít v případě, že první parametr je **null**.

```
1: /* Coalesce – nahrazení případné NULL hodnoty jinou */
2: select FirstName + ' ' +
3:        Coalesce(MiddleName + ' ', '') +
4:        LastName as FullName
5: from Customer
```

V uvedeném příkladu je jako testovaná hodnota výraz sloučení sloupce s prostředním jménem a mezery, která, pokud by byla přičtena až k výsledku funkce **Coalesce**, by duplikovala mezeru za křestním jménem a mezi ním a příjmením by tak byly mezery dvě.

Použití této funkce má samozřejmě význam pouze u sloupců, které nejsou **not null**, tedy mohou vůbec hodnotu **null** obsahovat. Funkce nemusí samozřejmě být nasazena pouze při výpočtech, aby nekazila celkový výsledek, ale může nahrazovat i **null** za nějakou platnou hodnotu.

```
1: /* Dosazení jiné hodnoty za NULL */
2: select MiddleName,
3:        Coalesce(MiddleName, '?')
4: from Customer
```

Máme-li problém opačný, tedy že potřebujeme určitou definovanou hodnotu nahradit za **null**, můžeme použít funkci **Nullif**. Jejím prvním parametrem je testovaná hodnota a druhým hodnota, které pokud se ta první rovná, bude jako výsledek funkce použito **null**. Jsou-li obě hodnoty různé, bude použita hodnota z prvního parametru.

```
1: /* Změna určité hodnoty na NULL (stát England změnit na NULL) */
2: select CountryRegion,
3:        NullIf(StateProvince, 'England') Stat
4: from Address
```

4.2 Podmínky v části select

Pro změnu **null** na jinou konkrétní hodnotu či naopak, k jakožto dosti častým požadavkům na úpravu, existují tedy speciální funkce. K dispozici jsou však i obecnější možnosti, jak konkrétní hodnotu v části **select** upravit, těsně před jejím zařazením do výsledné sestavy.

Nejobecnějším způsobem čili i tím s nejvíce možnostmi, ale na druhou stranu s nejdelším zápisem, je výraz **case** ve své podmínkové variantě. Ten umožňuje zapsat a otestovat nejen jednu, ale v podstatě libovolný počet podmínek (**when**) včetně možnosti, že neplatí ani jedna z nich (**else**) a pro každý tento případ určit hodnotu, která bude v daném sloupci použita do výstupu (**then**).

```
 1: /* Vícenásobná podmínka v selektu */
 2: select Name, Color,
 3:        case
 4:            when Color = 'Red'   then 'Červená'
 5:            when Color = 'Blue'  then 'Modrá'
 6:            when Color = 'Black' then 'Černá'
 7:            when Color = 'White' then 'Bílá'
 8:            else '?'
 9:        end as Barva
10: from Product
```

Syntaxe výrazu **case** je tedy taková, že se v části select místo sloupce zapíše **case**, následuje **when** a za ním první podmínka, tedy výraz, jehož výsledkem je logická hodnota **true** či **false**. Ta (i všechny následující) může být jako vždy

libovolně složitá, pospojovaná pomocí operátorů **and** a **or** z více částí, do kterých lze zapojit i změnu priority jejich vyhodnocování pomocí kulatých závorek.

Za podmínkou následuje klíčové slovo **then** a za ním již jen hodnota, která má být pro daný sloupec a záznam dosazena do výstupu v případě, že podmínka platí. Tato hodnota může být také libovolně sestavena jak z přímého odkazu na libovolný sloupec zdrojových dat, tak i jejich kombinací pomocí dalších výrazů či funkcí.

Poté může následovat další `when` *`podmínka`* `then` *`hodnota`*, tedy druhá (a další) podmínka, která se testuje pouze v případě že ta první (předchozí) splněna nebyla. Na závěr ještě může být použita varianta `else` *`hodnota`*, tedy stanovaní hodnoty, která se má použít, pokud neplatila ani jedna z uvedených podmínek. Vše je zakončeno klíčovým slovem **end**. Tomuto sloupci pak může být přiřazen také alias.

V případě, že u všech podmínek dochází pouze k porovnání téhož sloupce s různými hodnotami, lze zápis výrazu **case** zkrátit. Testovaný sloupec (popř. výraz) se uvede hned za slovo **case** ještě před první **when**, díky čemuž se rozpozná, že jde o tuto variantu **case**. Za jednotlivá **when** se pak již nepíší celé podmínky, ale pouze hodnoty (či výrazy), kterým pokud se testovaný sloupec rovná, tak bude do výstupu použita hodnota za následujícím **then**. I v tomto případě lze použít výchozí hodnotu za **else**, pokud by testovaný sloupec nebyl roven ani jedné z uvedených možností.

```
 1: /* Vícenásobné porovnání */
 2: select Name, Color,
 3:        case Color
 4:            when 'Red'   then 'Červená'
 5:            when 'Blue'  then 'Modrá'
 6:            when 'Black' then 'Černá'
 7:            when 'White' then 'Bílá'
 8:            else '?'
 9:        end as Barva
10: from Product
```

Obě varianty **case** lze, stejně jako jakoukoli další funkci, použít i pro řazení v části **order by**, aniž by týž výraz musel být i v části **select** (to by se na ní stačilo odkázat indexem či aliasem). V uvedených příkladech by tak bylo možné například jednotlivým barvám přiřadit vlastní pořadí dle zvolené stupnice, podle kterých by byly v první úrovni produkty řazeny.

Nepotřebujeme-li vícenásobné porovnání a stačí nám jediná podmínka s možností, že buď platí nebo neplatí, není třeba používat zdlouhavý zápis **case**, byť by to dokázal vyřešit také. Pro tyto případy je zde totiž funkce **iif**, která má tři parametry:

1. logická hodnota (**true**/**false**), obvykle definovaná výrazem s porovnáním,
2. hodnota, kterou funkce vrací, pokud předchozí podmínka platí,
3. hodnota, kterou funkce vrací, pokud předchozí podmínka neplatí.

```
1: /* Jednoduchá podmínka (IF) - IIF(podmínka, hodnota když platí,
      hodnota když neplatí) */
2: select sum(iif(Status  = 5, TotalDue, 0)) Zaplaceno,
3:        sum(iif(Status != 5, TotalDue, 0)) BudouciPrijmy
4: from SalesOrderHeader
```

V ukázce je funkce **iif** použita uvnitř agregační funkce pro součet (**sum**) celkových částek objednávek tak, že dle statusu objednávky vyhodnotí, jestli je již zaplacená (`Status=5`) či nikoli a podle toho její částku (**TotalDue**) do součtu v daném sloupci buď zařadí či nikoli (místo ní vrátí 0, která součet nenavýší).

4.3 Datum a čas

Pro práci s hodnotami typu datum (**date**), čas (**time**) či obojím (**datetime** a další) existuje mnoho funkcí. Nejprve si ale ukážeme, jak z databázového serveru zjistit aktuální datum a čas.

Pokud totiž databázový server používá architekturu *klient-server*, nebo jde o server webový, tedy databáze běží na jiném počítači než aplikace, která k ní přistupuje, může se aktuální datum a čas mezi jednotlivými klienty lišit. Ať už jde o rozdíl několik vteřin z důvodu již dlouho neprovedené synchronizace času s jeho internetovým poskytovatelem, nebo uživatelem čas záměrně posunutý, nemělo by to mít vliv na správné fungování serveru a jeho aplikací. Například pokud by uživatel s opožděným časem odpovídal v diskusním fóru na příspěvek jiného uživatele s časem správným, mohla by se tak jeho odpověď zařadit ještě před původní zprávu.

Je proto žádoucí, aby čas byl pro všechny uživatele stejný, což může zajistit právě databázový server. Ten, i kdyby sám neměl nastavený zcela přesný čas, sjednocoval by jej pro všechny uživatele a záznamy. Při ukládání a úpravě záznamů s aktuálními časovými údaji by tedy tuto hodnotu neměl vyplňovat klient (aplikace/webová stránka), ale až právě databázový server.

Potřebuje-li klientská aplikace znát přesný čas na pro ni centrálním (databázovém) serveru, může jej zjistit pomocí následujícího dotazu, aniž by bylo zapotřebí cokoli vědět o struktuře databáze.

```
1: /* Aktuální datum a čas na DB serveru */
2: select CURRENT_TIMESTAMP AktualniDatumACas
3: from (values (0)) as t(c)   -- Právě jeden virtuální záznam
```

Tento dotaz tedy vrátí jedinou hodnotu (jeden řádek a jeden sloupec) obsahující aktuální datum a čas na serveru s přesností na milisekundy (typ **datetime**), v časové zóně nastavené na serveru. Hodnotu **CURRENT_TIMESTAMP** lze také použít v DML příkazech (viz kap. 6.1) místo konstantních hodnot.

V ukázce je v sekci **from** použit výraz s klíčovým slovem **values** (používá se též při klasickém vkládání nových záznamů **insert**, viz kap. 6.1), který v kulatých závorkách definuje hodnoty záznamu virtuální tabulky (použita je hodnota 0), které je následně povinně přidělen alias **t** (tabulka) a jejím sloupci s nulou alias **c** (column).

Aktuální datum a čas lze v drobných obměnách získat také pomocí následujících funkcí. Lze tak například zvýšit přesnost (typ **datetime2(7)**), získat UTC čas (univerzální „greenwichský“ čas v nulté časové zóně na nultém poledníku), nebo aktuální čas v dané časové zóně včetně hodnoty jejího časového posunu (typ **datetimeoffset(7)**).

```
1: /* Aktuální datum a čas na DB serveru */
2: select
3:   CURRENT_TIMESTAMP,    -- 2022-10-17 20:50:23.320
4:   GetDate(),            -- 2022-10-17 20:50:23.320
5:   GetUtcDate(),         -- 2022-10-17 19:50:23.320
6:   SysDateTime(),        -- 2022-10-17 20:50:23.3214014
7:   SysDateTimeOffset(), -- 2022-10-17 20:50:23.3214014 +01:00
8:   SysUtcDateTime()      -- 2022-10-17 19:50:23.3214014
9: from (values (1)) as t(c)
```

Datum a čas je uložen a zpracováván jako jedna hodnota, nicméně části, ze kterých se skládá, lze z této hodnoty snadno extrahovat pomocí funkce **DatePart** (popř. dalších). Následující dotaz ukazuje jednotlivé varianty použití této funkce pro získání jednotlivých částí data a času.

```
 1: /* Extrakce části z data */
 2: select CURRENT_TIMESTAMP Datum,
 3:        Year(CURRENT_TIMESTAMP) Rok1,  -- podobně Month() a Day()
 4:        DatePart(YEAR, CURRENT_TIMESTAMP) Rok,
 5:        DatePart(MONTH, CURRENT_TIMESTAMP) Mesic,
 6:        DatePart(DAY, CURRENT_TIMESTAMP) Den,
 7:        DatePart(DAYOFYEAR, CURRENT_TIMESTAMP) DenVRoce,
 8:        DatePart(ISO_WEEK, CURRENT_TIMESTAMP) TydenVRoce,
 9:        DatePart(WEEKDAY, CURRENT_TIMESTAMP) DenVTydnuUS, -- DVT
10:        (DatePart(WEEKDAY, CURRENT_TIMESTAMP) + 5) % 7 + 1 DVTCZ,
11:        DatePart(HOUR, CURRENT_TIMESTAMP) Hodina,
12:        DatePart(MINUTE, CURRENT_TIMESTAMP) Minuta,
13:        DatePart(SECOND, CURRENT_TIMESTAMP) Sekunda,
14:        DatePart(MILLISECOND, CURRENT_TIMESTAMP) Milisekund,
15:        DatePart(MICROSECOND, CURRENT_TIMESTAMP) Microekund
16: from (values (0)) as t(c)
```

Další možné hodnoty pro první parametr funkce **DatePart** a možnosti, jak změnit normu pro číslování týdnů v roce či dnů v týdnu lze nalézt v oficiální dokumentaci k této funkci.

Při určení rozdílu mezi dvěma daty lze použít funkci **DateDiff**, které je v první řadě potřeba definovat, v jakých jednotkách má tento rozdíl vyjádřit. K dispozici jsou stejné možnosti jednotek, jako v předchozí ukázce. Výsledek je však vždy celé číslo, do kterého je jako celek započtena i každá započatá jednotka (např. 1 hodina a 5 minut je uvedeno jako rozdíl dvou celých hodin).

```
1: /* Počet dnů od objednání po odeslání */
2: select OrderDate,
3:        ShipDate,
4:        DateDiff(DAY, OrderDate, ShipDate) DnuMezi
5: from SalesOrderHeader
```

Je-li zapotřebí vyjádření rozdílu desetinným číslem (např. 1,5 dne), stačí rozdíl vyjádřit v nižších jednotkách (v hodinách či minutách, záleží na požadované přesnosti) a výsledek vydělit počtem těchto jednotek v jednotce vyšší (např. 36 hodin / 24 = 1,5 dne, nebo také 2 160 minut / (24*60) = 1,5 dne).

4.4 Číselné hodnoty

Také pro práci s číselnými hodnotami je k dispozici celá řada funkcí. Nejprve se zaměříme na generování náhodných hodnot. Za tímto účelem jsou zde dvě základní funkce: **Rand** a **NewID**. Funkce **Rand** (ř. 2) generuje náhodné číslo typu **float** v rozsahu od 0 do 1, pro daný sloupec, jež je však v základu shodné pro každý řádek výstupu. Funkce **NewId** (ř. 3) pak vrací unikátní hodnotu pro každý řádek i sloupec, ta je však typu **uniqueidentifier**, a na číslo ji je v případě potřeby nutné teprve převést.

```
 1: /* Náhodné číslo */
 2: select Rand() Rand1, Rand() Rand2,  -- Náhodné číslo <0, 1)
 3:        NewId() NewId,               -- GUID
 4:        Checksum(NewId()) CheckSum,  -- Kontrolní součet GUID
 5:        -- GUID konvertovaný na číslo
 6:        Convert(varbinary, NewId()) "Convert",
 7:        -- Náhodné číslo, kde SEED je GUID konvertovaný na číslo
 8:        Rand(Convert(varbinary, NewId())) Rand3,
 9:        -- Náhodné číslo, kde SEED kontrolní součet GUID
10:        Rand(CheckSum(NewId())) Rand4
11: from Product
```

Funkce **CheckSum** (ř. 4) z unikátní hodnoty (GUID) vypočítá kontrolní součet, tedy jakýsi číselný hash v rozsahu **int** (tj. cca -2 mld. až +2 mld). Funkce **Convert** (ř. 6) dokáže mezi sebou převádět (konvertovat) kompatibilní datové typy, přičemž **uniqueidentifier** lze například převést na typ **varbinary**, tedy libovolně dlouhé číslo zapsané v binární podobě.

Funkce **Rand** může přijmout nepovinný číselný parametr **seed**, který slouží jako základ pro výpočet pseudonáhodné výstupní hodnoty této funkce. Pro stejné **seed** je pak stejná i výstupní hodnota. Pokud není **seed** definován, dosadí za něj server náhodnou hodnotu, která je však pro všechny řádky stejná (jako kdyby bylo v dotazu například napsáno `Rand(123)`), a proto je stejná i výsledná hodnota ve všech řádcích. Dosadíme-li však hodnotu v každém řádku jinou, bude i náhodné číslo pokaždé jiné. Pro tyto účely může posloužit například ID záznamu, kontrolní součet (**CheckSum**) libovolné unikátní hodnoty anebo právě GUID vrácený funkcí **NewId**, převedený na nějaké číslo, ať již konverzí (ř. 8) či hashem (ř. 10).

Náhodnou hodnotu v rozsahu $\langle 0; 1)$ pak již není problém převést na libovolný rozsah a převést třeba i na celé číslo (v ukázce funkcí **Floor**, jež usekne jeho desetiny, resp. vrací největší celé číslo, které je menší nebo rovno tomu zdrojovému desetinnému). Jako zdrojová tabulka je použita **Product**, z níž sice žádná data nečerpáme, ale její záznamy dostatečně (295x) zopakují použití funkce, aby byly patrné rozdíly jednotlivých řádků.

```
1: /* Náhodné číslo od 10 do 50 */
2: select Floor(Rand(CheckSum(NewId())) * 41 + 10)
3: from Product
4: order by 1
```

Funkci vracející náhodné hodnoty lze použít i v části **order by** při řazení záznamů, a do výstupu je tak získat v náhodném pořadí. Pokud navíc omezíme tento výstup na první řádek či jen několik řádků, určíme tak náhodný/é záznam(y).

```
1: /* Náhodné tři produkty */
2: select top 3 Name
3: from Product
4: order by NewId()
```

Je-li navíc potřeba tyto náhodně vybrané záznamy následně seřadit třeba abecedně dle názvu, lze předchozí dotaz použít jako vnořený pro definici zdroje dat dalšího nadřazeného dotazu/výběru, který nad ním požadované již nenáhodné řazení vykoná.

```
1: /* Náhodné tři produkty */
2: select Name
3: from (select top 3 Name
4:       from ProductModel
5:       group by Name      -- Nahrazuje distinct
6:       order by NewId()
7: ) as m
8: order by Name
```

Při práci s desetinnými čísly je také důležité zaokrouhlování. Funkce **Floor** tedy vrací nejbližší menší celé číslo (např. 1,9 => 1, ale -1,9 => -2). Její alternativou je funkce **Celing**, která naopak vrací nejbližší větší celé číslo (např. 1,9 => 2, ale -1,9 => -1).

Pro skutečné zaokrouhlování, nejen na celé číslo, ale na zvolený počet cifer před či za desetinnou čárkou je tu funkce **Round**. Ta potřebuje na vstupu číslo, které má zaokrouhlit, a počet cifer, na který má toto číslo zaokrouhlit.

Pokud je druhý parametr kladné číslo, týká se cifer za (napravo) desetinnou čárkou (např. `Round(123.456, 2)` => 123.46), je-li záporný, jde o cifry před (nalevo) od desetinné čárky (např. `Round(123.456, -1)` => 120).

```
1: /* Zaokrouhlování */
2: select SubTotal,
3:        SubTotal * 0.21 DPH,     --  21%
4:        SubTotal * 1.21 S_DPH,   -- 121% (cena bez DPH + DPH)
5:        Round(SubTotal * 1.21, 2) Zaokrouhleno,
6:        Cast(Round(SubTotal * 1.21, 2) as numeric(10, 2)) DveDes,
7:        Cast(Round(SubTotal * 1.21, -2) as int) ZaokrouhlenoNaSto
8: from SalesOrderHeader
```

Funkce **Round** vrací hodnotu datového typu odvozenou z typu prvního parametru. Zaokrouhlíme-li ale číslo se čtyřmi či více desetinami na dvě, není potřeba nadále zobrazovat číslo se čtyřmi, kde jsou poslední dvě cifry vždy 0, potažmo při zaokrouhlení na číslo celé, nepotřebujeme již desetiny vůbec.

V těchto případech lze funkci pro zaokrouhlení obalit ještě funkcí pro převod datového typu. Lze použít například funkci **Convert**, která hodnoty konvertuje převodovými funkcemi (např. zvládne i převod textu na číslo a obráceně), nebo funkci **Cast**, jež pouze přetypovává hodnotu na jinou kompatibilní (např. desetinné číslo na celé či obráceně, nebo třeba text na delší text).

4.5 Textové řetězce

Nejčastějším případem práce s textovými řetězci je vyhledávání podle nich. Podmínku pro shodu dvou textů zapíšeme klasicky pomocí rovná se (např.

`where Jmeno = 'Jan'`), ale co když známe pouze část tohoto textového řetězce a potřebujeme všechny záznamy, která ji obsahují, začínají na ni, nebo jí končí.

Pro tyto účely je zde operátor **like**, který umožňuje v textovém řetězci použít znak procent (**%**), jež podobně jako v maskách souboru hvězdička (*****), zastupuje jakýkoli počet (0–N) libovolných znaků, který porovnávaný text může na daném místě obsahovat a výsledek porovnání bude stále **true**.

```
1: /* Hledání podle části textu */
2: select LastName
3: from Customer
4: where LastName like 'B%'    -- příjmení začíná na 'B'
5:   and LastName like '%we%' -- příjmení obsahuje  'we'
6: and LastName like '%r'    -- příjmení končí na  'r'
```

Tento způsob porovnávání textů je nicméně náročnější na výkon i čas, než klasické porovnávání (=), protože je nutné zkontrolovat celý obsah každého textu zvlášť. Operátor **like** lze aplikovat pouze na textové sloupce s pevnou délkou (**char**, **nchar**, **varchar**, **nvarchar**), nikoli na sloupce s neomezenou délkou (např. typ **text** či **nvarchar(max)**). Je-li zapotřebí prohledávat i tyto hodnoty, musí se jim aktivovat indexace v katalogu pro fulltextové vyhledávání, na které se pak používají jiné operátory.

Pro úpravu textových hodnot do výstupu či pro jiné operace jsou zde pak, mimo jiné, například funkce použité v následujícím dotazu.

```
1: /* Textové funkce */
2: select LastName,
3:        Left(LastName, 5) Prvnich5znaku,     -- část zleva
4:        Right(LastName, 5) Poslednich5znaku, -- část zprava
5:        SubString(LastName, 2, 3) Stred,     -- od 2. znaku 3
6:        Replace(LastName, 'w', 'vv') W2VV,   -- nahrazení w za vv
7:        Reverse(LastName) Obracene,          -- obrácení pořadí
8:        Len(LastName) PocetZnaku             -- délka textu
9: from Customer
```

5 Data Definition Language (DDL)

Doposud jsme se zabývali pouze SQL dotazy, tedy tím, jak se správně zeptat, abychom získali přesně ta data, která potřebujeme. Kromě čtení dat je však zapotřebí s databází „domluvit“ i další náležitosti, jako třeba tam data ukládat (viz násl. kap. 6), ale předně vůbec nadefinovat, jakou strukturu (tabulky, jejich sloupce, vztahy mezi nimi apod.) má databáze mít.

Pro následující příklady tak opustíme ukázkovou databázi *AdventureWorksLT* a začneme pracovat s databází zcela novou, na počátku prázdnou, jejíž strukturu budeme směřovat k základní evidenci filmové databáze.

5.1 Vytvoření tabulky

Máme-li prázdnou databázi, pak její plnění obvykle zahájíme vytvořením tabulek. I v případě DDL lze vypozorovat paralelu s přirozeným jazykem, neboť například přání „vytvoř tabulku“ realizujeme příkazem **create table**, následovaný názvem této tabulky a výčtem jejích sloupců. Název tabulky je důležité si dobře rozmyslet, protože její případné přejmenování, zvláště existují-li na ni reference, není zrovna triviální záležitostí.

```
1: /* Vytvoření tabulky Reziseri */
2: create table Reziseri (
3:   ID integer primary key not null identity,
4:   Jmeno varchar(20),
5:   Prijmeni varchar(30) not null
6: );
```

Sloupce, resp. jejich názvy, které má tabulka obsahovat, se vypisují do kulatých závorek za názvem tabulky a jsou odděleny čárkou. U každého sloupce musí být uveden minimálně ještě datový typ, kterého budou hodnoty v tomto sloupci uložené. U povinných hodnot, které nemohou nabývat

hodnoty **null** a musí být tedy vždy nějak vyplněny, aby bylo možné záznam uložit, stačí za datový typ dopsat označení **not null**.

Při vytváření tabulky lze také rovnou určit její primární klíč, uvedením slov **primary key** hned za jeho datový typ. Primární klič lze ovšem také definovat také jako omezení tabulky (viz násl. ukázka) anebo dodatečně vlastním příkazem (viz kap. 5.2) pro úpravu tabulky.

V ukázce je u sloupce ID také uvedeno klíčové slovo **identity**. To v případě SQL Serveru u číselného sloupce aktivuje automatické generování hodnoty jako číselné řady. Hodnotu v tomto sloupci pak nelze zadávat ručně, ale doplní se až při uložení záznamu tak, že prvnímu záznamu vyplní 1, druhém 2 atd.[8]

Celá definice ve skutečnosti je `identity(1, 1)`, přičemž parametry 1 a 1 jsou výchozími, takže právě tyto hodnoty jsou do definice dosazeny v případě jejich neuvedení. První z nich určuje **seed**, neboli výchozí hodnotu od které se začne počítat a druhý pak **increment**, tj. hodnotu, o kterou se číselná řada v každém kroku navýší.

Následující příkaz pro vytvoření tabulky **Filmy** navíc ukazuje, jak definovat primární klíč jako omezení tabulky (**constraint**, ř. 7), což oproti předchozí definici umožňuje například definici primárního klíče složeného z více sloupců. Druhé omezení pak přidává cizí klíč (**foreign key**, ř. 8–10) pro sloupec **IdRezisera** na tabulku **Reziseri** a její sloupec **ID**. Ten je navíc nastaven tak, že při smazání (**delete**) režiséra dojde automaticky ke kaskádovitému (**cascade**) smazání i všech filmů, u kterých je jako režisér evidován.

[8] Číslo je záznamu přiřazeno i v případě, že se pokus o uložení záznamu nepodaří (např. není vyplněna nějaká povinná hodnota), což se při dalším pokusu o uložení opakuje, díky čemž nemusí být tato řada souvislá, aniž by muselo dojít k vymazání nějakého záznamu.

```
 1: /* Vytvoření tabulky Filmy */
 2: create table Filmy (
 3:   ID integer not null identity,
 4:   Nazev varchar(100) not null,
 5:   Rok smallint,
 6:   IdRezisera int,
 7:   constraint PK_Filmy primary key clustered (ID),
 8:   constraint FK_Filmy_IdRezisera foreign key (IdRezisera)
 9:     references Reziseri (ID)
10:     on delete cascade
11: );
```

5.2 Úprava tabulky

Jakmile je tabulka vytvořena, příkaz **create** už na ni nelze aplikovat. Dochází tedy k její úpravě, což pro databázové objekty zaštiťuje příkaz **alter** následovaný **table** a jejím názvem. Pro přidání omezení typu primární klíč, pokud by jej tabulka dosud neměla nastavený, pak příkaz úpravy vypadá následovně.

```
1: -- Přidání primárního klíče do tabulky Reziseri pro sloupec ID
2: alter table Reziseri
3: add constraint PK_Reziseri    -- Název klíče
4: primary key (ID);             -- Výčet sloupců složeného PK
```

Pro přidání klíče cizího se pak používá následující příkaz úpravy tabulky. Části on…, definující, co se má stát, dojde-li k výmazu navázaného záznamu či změně hodnoty, přes kterou je provázání realizováno (**ID** v tabulce **Reziseri**, kde změna ovšem nastat nemůže, protože sloupec **ID** má automaticky generovanou hodnotu přes **identity**), jsou nepovinné a nejsou-li explicitně nastaveny, jejich implicitní chování je **no action**, tedy že vymazání režiséra, který má přiřazeny nějaké filmy databáze nepovolí a při pokusu o něj vyvolá

chybové hlášení. Totéž by pak platilo při pokusu o aktualizaci jeho ID, kdyby to bylo možné. K dispozici je ještě třetí varianta **set null**, jež by v těchto případech do sloupce s cizím klíčem nastavila hodnotu null, samozřejmě pouze v případě, že by sloupec nebyl označen jako **not null**.

```
1: /* Přidání cizího klíče do tabulky Filmy pro sloupec IdRezisera
      reference (vazba) na tabulku Reziseri a její sloupec ID */
2: alter table Filmy                    -- Tabulka s FK
3: add constraint FK_Filmy_Reziseri     -- Název klíče
4: foreign key (IdRezisera)             -- Sloupec s klíčem
5: references Reziseri(ID)              -- Navázaná tabulka a její PK
6: on delete cascade  -- Smazat filmy, smaže-li se jejich režisér
7: on update cascade; -- Aktualizovat ID při změně v tab. Reziseri
```

Tabulkám a jejich sloupcům lze přidávat i další omezení. Jedním z nejpoužívanějších je kontrola unikátnosti (resp. index s kontrolou unikátnosti **unique**). Ta je automaticky nastavena i v rámci primárního klíče, kde zajišťuje, aby jeho hodnota nebyla v žádných dvou záznamech tabulky stejná. A stejně tato kontrola funguje i jinde, například v tabulce s uživateli se aplikuje na sloupec s přihlašovacím jménem (login či e-mail), protože to musí být vždy unikátní v rámci celého systému.

V následující ukázce je unikátnost v tabulce **Filmy** složená ze dvou sloupců (**Nazev** a **Rok**), díky čemuž se dva filmy mohou jmenovat stejně, ale pouze pokud měly premiéru v různých letech.

```
1: /* Přidání omezení (indexu/kontroly) unikátnosti - žádné dva
      filmy se v témže roce nesmí jmenovat stejně */
2: alter table Filmy
3: add constraint U_Filmy_Nazev_Rok -- Název omezení
4: unique (Nazev, Rok); -- Sloupce s jedinečnými kombinacemi hodnot
```

Do tabulky lze také dodatečně přidávat další sloupce, pod příkazem **add** bez další specifikace, jelikož sloupec je tím hlavním, co se do tabulky obvykle přidává, proto je v rámci úspor slovo **column** vynecháno. Při přidávání sloupce je samozřejmě nezbytné určit jeho název, datový typ, případně další podrobnosti.

```
1: /* Přidání sloupce Delka do tabulky Filmy */
2: alter table Filmy
3: add Delka time;
```

Pokud by hodnota ve sloupci měla být povinná (**not null**) a tabulka již obsahovala nějaké záznamy, nebude možné takto jednoduše sloupec do tabulky přidat, protože nový sloupec se standardně u všech existujících řádků vyplní výchozí hodnotou **null**, což by tomuto požadavku odporovalo. Proto je potřeba sloupec nejprve přidat bez vynucování zadání hodnoty, jak ukazuje příkaz výše, následně hodnoty všem záznamům nastavit, a až poté sloupec označit jako povinný, jak ukazuje následující příkaz.

```
1: /* Úprava sloupce Delka – přidání NOT NULL kontroly */
2: alter table Filmy
3: alter column Delka time not null;
```

Druhou variantou je sloupci nastavit výchozí hodnotu (**default**), která se ve sloupci nastaví u všech záznamů, kde je hodnota **null**.

```
1: /* Přidání povinného sloupce Delka, který se u stávajících
      záznamů vyplní hodnotu 1 hodina */
2: alter table Filmy
3: add Delka time not null
4: constraint C_Filmy_Delka default '01:00:00';
```

5.3 Odstraňování databázových objektů

Databázové objekty lze pochopitelně nejen vytvářet, ale také mazat (odebírat). Pro tento úkon je stěžení příkaz **drop**. Pokud potřebujeme odebrat sloupec tabulky, jedná se stále o její úpravu a odebrání je až v rámci ní.

```
1: -- Odebrání sloupce Delka z tabulky Filmy (sloupce i dat v něm)
2: alter table Filmy
3: drop column Delka;
```

Odebíráme-li ale celou tabulku, bude příkaz rovnou začínat **drop table** a název tabulky.

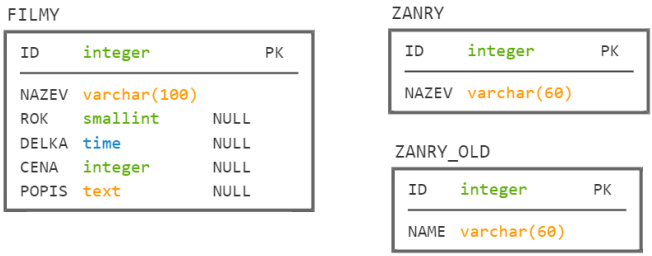
```
1: -- Odebrání tabulky Filmy z databáze (včetně dat v této tabulce)
2: drop table Filmy;
```

Tento příkaz tedy nejen že vymaže data celé tabulky, ale odstraní i celou její přítomnost v databázi. Existuje-li ovšem reference na mazanou tabulku, např. tabulka **Filmy** s cizím klíčem **IdRezisera** odkazující se na tabulku **Reziseri**, tak odstranění tabulky **Reziseri** (na ni je reference) skončí chybovým hlášením, zatímco odstranění tabulky **Filmy** (na ni reference není, to že ona se odkazuje jinam nevadí) by prošlo v pořádku. Po odstranění tabulky **Filmy** (nebo alespoň její reference na režiséry) by již bylo možné odstranit i tabulku **Rezisery**.

Totožně by se pak odebíraly veškeré další databázové objekty (**view**, **trigger**, **index**, **procedure**, **function**, …).

6 Data Manipulation Language (DML)

Manipulace dat bude prezentována na jednoduché databázi, kterou jsme částečně již vytvořili v předchozí kapitole 5. Její kompletní datový model pro tuto kapitolu je pak následující.



6.1 Základní DML příkazy

Základní příkazy vyřešme postupně, dle jejich životního cyklu. Nejprve záznam totiž musíme do tabulky vložit (**insert**), abychom s ním mohli provádět něco dalšího. Pak teprve můžeme jeho hodnoty aktualizovat (**update**), nebo jej celý zase vymazat (**delete**). Vložení jednoho nového záznamu ukazuje následující DML příkaz.

```
1: /* Vložení jednoho nového záznamu */
2: insert into FILMY
3: (ID, NAZEV, ROK, DELKA, CENA)      -- Seznam sloupců pro hodnoty
4: values
5: (1, 'Star Wars', 1977, '02:05:00', 11000000); -- Hodnoty sloupců
```

I zde je snaha o koncept anglické věty, což je například i jeden z důvodů, proč bohužel mají příkazy pro vložení a úpravu dat zcela odlišnou syntaxi. Vložení začíná příkazem `insert into …` (*vlož do …*) a následuje název ta-

bulky, do které se má záznam vložit. Jelikož se záznamy jen vyjímečně přidávají se všemi hodnotami definovanými v tomto příkazu (např. nějaký údaj nemusí být při vkládání znám a zůstane **null**, hodnota ID může být automaticky generována přes **identity**, a tudíž ani ručně definovat nelze, datum a čas poslední úpravy se může doplňovat sám apod.), a také již ze základních pravidel databáze by na pořadí sloupců v tabulce nemělo záležet, je nezbytné v následující části příkazu, do kulatých závorek a oddělené čárkou, explicitně vyjmenovat všechny sloupce tabulky (jejich názvy), jejichž hodnoty hodláme při vkládání definovat. Nesmí zde samozřejmě scházet sloupce, které se automaticky nevyplňují a mají příznak **not null**.

V základní verzi tohoto příkazu pro vložení jediného záznamu pak následuje klíčové slovo **values** a opět v kulatých závorkách ohraničený výčet hodnot, které se do nového záznamu mají uložit. Hodnoty musí být uvedeny ve stejném pořadí a počtu, jako byl seznam sloupců v předchozí části příkazu. A jelikož se jedná o příkaz, nikoli dotaz, měl by být zakončen středníkem.

Úprava již vloženého záznamu se provádí následovně.

```
1: /* Změna hodnot záznamu */
2: update FILMY set
3:   NAZEV = 'Vetřelci',
4:   ROK = 1986,
5:   DELKA = '02:34:00'
6: where ID = 1; -- Není-li where, změna se provede u všech záznamů
```

Příkaz o úpravu záznamu začíná `update NazevTabulky set …`, (*uprav tabulku nastav …*). Poté následuje výčet sloupců, tentokrát bez závorky, kterým se rovnou nastavují konkrétní hodnoty přiřazením znaménkem rovná se. Jednotlivé změny se oddělují čárkou, která však není za poslední z nich.

Velmi důležitou, byť nepovinnou, součástí příkazu **update** je část **where**. Pokud ta zde totiž není uvedena, promítne se zadaná změna bez jakéhokoli varování na všechny záznamy v tabulce, což většinou nechceme. Podmínka je obvykle koncipována tak, aby byl upraven pouze jeden záznam, což se nejsnáze řeší přes jeho ID, nicméně nic nebrání tomu ji definovat i zcela jinak. Jeden záznam tak může být určen třeba podle jedinečného názvu filmu a jeho roku vydání, nebo záměrně podmínka může postihovat i více záznamů, či ji lze vynechat zcela.

Posledním úkonem, který lze se záznamem provést je jeho trvalé odstranění z tabulky, což řeší následující ukázka.

```
1: /* Vymazání jednoho záznamu */
2: delete from FILMY
3: where ID = 1; -- Není-li where, smažou se záznamy z celé tabulky
```

Příkaz pro výmaz záznamu začíná `delete from …` (*vymaž z …*) a následuje název tabulky, ze které chceme záznam odstranit. Pak je na řadě podmínka (**where**), která vymezí záznam(y), jež se mají vymazat. I zde může být podmínka pojata tak aby postihla jeden konkrétní záznam, vícero záznamů, popř. i žádný z nich. Pokud část **where** vynecháme zcela, nebo jí nesprávně nadefinujeme podmínku (např. `1=1`, tj. vždy podmínka platí), dojde opět k trvalému vymazání všech záznamů v celé tabulce.

Příkazy uvedené v této podkapitole byly pouze základní variantou svého použití. Ta by měla být opět shodná na všech databázových systémech. Příkazy však mají i své rozšířené varianty použití, jimiž se zabývá následující podkapitola.

6.2 Pokročilejší varianty DML příkazů

Příkaz pro vkládání záznamů lze použít nejen tak, aby vložil právě jeden záznam, ale i tím způsobem, že záznamu najednou vloží vícero. Následující příkaz ukazuje, možnost hromadného vložení pomocí v tomto příkazu ručně zadaných hodnot.

```
1: /* Vložení více záznamů v jednom příkazu */
2: insert into ZANRY_OLD
3: (ID, NAME)
4: values
5: (1, 'Sci-Fi'),
6: (2, 'Drama'),
7: (3, 'Krimi'),
8: (4, 'Horor');
```

Používanější variantou pro hromadné vkládání záznamů však většinou bývá, že se jeho zdrojové hodnoty nezadávají ručně, ale kopírují z nějaké již existující tabulky, popř. spojení více tabulek. V tom případě je místo klíčového slova **values** a výčtu hodnot uveden klasický libovolně složitý SQL dotaz (může používat např. seskupování dat, spojování tabulek, funkce atd.). Tento dotaz musí výstupní hodnoty (sloupce) vracet v tom pořadí, jak jsou uvedeny v příkazu **insert**, jinak jeho struktura není nijak více limitována.

```
1: /* Hromadné vložení dat vybraných selectem z jiné tabulky */
2: insert into ZANRY (ID, NAZEV)
3: select ID, NAME from ZANRY_OLD;
```

Následující ukázka již není pouhým příkazem, ale částí kódu, který v sobě obsahuje hned několik pod příkazů. Kód ověří, jestli v tabulce **ZANRY** exis-

tuje nějaký záznam splňující podmínku ID=5. Pokud ano, pak tomuto záznamu pouze upraví jeho název, pokud neexistuje, pak záznam s tímto ID vloží kompletně nový.

```
1: /* Vložení záznamu, ale pokud tam je už (s ID=5)
      tak jen upravit jeho název */
2: if exists (select * from ZANRY where ID = 5)
3: begin
4:   update ZANRY set NAZEV = 'Thriller upravený' where ID = 5;
5: end else begin
6:   insert into ZANRY (ID, NAZEV) values (5, 'Thriller nový');
7: end;
```

Podobného mechanismu se často užívá například při aktualizaci číselníků, kde spíše než podle ID se existence záznamu kontroluje dle jedinečného kódu položky. Např. když do lékárny dorazí aktualizace ceníku léků, přičemž je možné, že do něho přibyly i nějaké nové položky, pak se dle jejich oficiálních kódů ověří, jsou-li již ve stávajícím ceníku lokální databáze prodejny, pokud ano, aktualizuje se jim jen cena, jestli ne, vloží se do ceníku nový lék včetně jeho aktuální ceny.

Dále se zaměříme na další možnosti samotného příkazu pro aktualizaci záznamů **update**. Následující varianta je v podstatě stále tou základní, ale ukazuje dříve nezmíněnou možnost aktualizace nové hodnoty na základě té stávající (čerpat lze i z jiného než aktualizovaného sloupce, např. `CenaProdejni = CenaNakupni * 1.25` pro 25% přirážku).

```
1: /* Navýšení roku u všech filmů o +1 */
2: update FILMY set
3:   ROK = ROK + 1;
```

Další varianta ukazuje jednak možnost použití aliasu pro tabulku (v části **from**), ve které se má záznam aktualizovat, a také získání nastavované hodnoty pomocí vnořeného SQL dotazu z jiné tabulky, který by měl vracet právě jednu hodnotu (jeden sloupec a jeden záznam), v němž se na tento alias lze i odkazovat (obvykle v podmínce).

```
1: /* Update (v1) názvů v ZANRY_OLD názvy z ZANRY se stejnými ID */
2: update o set
3:   o.NAME = (select z.NAZEV from ZANRY z where z.ID = o.ID)
4: from ZANRY_OLD o;
```

Když už víme, že i v příkazu **update** lze použít část **from**, podobně jako v dotazech **select**, nebude už takovým překvapením, že i zde je možné horizontálně propojit více tabulek, například pomocí kartézského součinu.

```
1: /* Totéž (v2) přes kartézský součin - updatuje jen záznamy s ID,
      které jsou v obou tabulkách */
2: update o set
3:   o.NAME = z.NAZEV
4: from ZANRY_OLD o, ZANRY z
5: where o.ID = z.ID;
```

Propojit tabulky lze samozřejmě také pomocí různých variant spojení **join**, například v ukázce použitého levého vnějšího spojení (**left join**).

```
1: /* Totéž (verze 3) přes left join */
2: update o set
3:   o.NAME = z.NAZEV
4: from ZANRY_OLD o
5:      left join ZANRY z on z.ID = o.ID;
```

6.3 Transakce

Transakce umožňují spouštět více DML (ale i DDL) příkazů tak, že se buď vykonají všechny, nebo ani jeden z nich. Zajišťují tak převod databáze z jednoho konzistentního stavu (před svým spuštěním) do druhého (po svém skončení). Díky nim lze také ohlídat souběžnou práci více uživatelů na týchž datech, aby nedošlo k jejich poškození či nevaliditě.[9]

Transakce dodržují tzv. **ACID** vlastnosti, které jsou následující:

- A – **atomicita** – Databázová transakce je jako operace dále nedělitelná (atomární). Provede se buď jako celek, nebo se neprovede vůbec.
- C – **konzistence** – Při a po provedení transakce není porušeno žádné integritní omezení.
- I – **izolovanost** – Operace uvnitř transakce jsou skryty před vnějšími operacemi.
- D – **trvalost** – Změny, které se provedou jako výsledek úspěšných transakcí, jsou skutečně uloženy v databázi, a již nemohou být ztraceny.

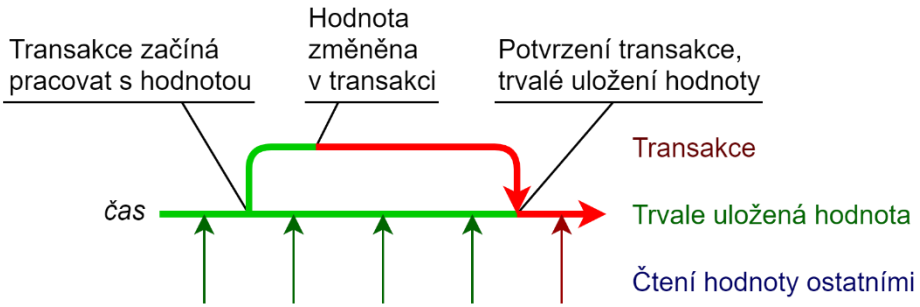
Transakci zahájíme příkazem **begin transaction** a ukončíme buď příkazem **commit**, který veškeré provedené změny potvrdí a trvale uloží do databáze, nebo **rollback**, který veškeré změny „zahodí“, jako kdyby k nim nikdy nedošlo. Následující příklad (viz databáze používaná v násl kap. 7) např. zvyšuje plat všem učitelům o 20 %, přičemž vykonání je zabezpečeno transakcí.

```
1: /* Zvýšení platu všem učitelům o 20 % s použitím transakce */
2: begin transaction;
3:   update UCITELE set
4:     PLAT = PLAT * 1.2;
5: commit;
```

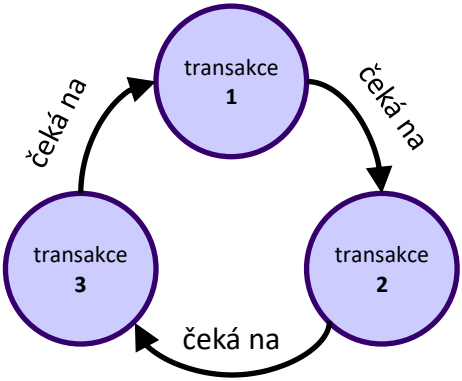
[9] Celou přednášku na téma transakce najdete na programko.net/databaze/transakce

Kdyby během aktualizace platů učitelů došlo k chybě (např. záznam uprostřed zpracování byl uzamčen a nemohl by být aktualizován), mohlo by toto navyšování bez transakce skončit tak, že někteří učitelé budou mít plat navýšen a zbytek nikoli. Příkaz bychom tak již spustit znovu nemohli, jelikož by se někomu plat navýšil již podruhé, a nezbývalo by než jednotlivé záznamy kontrolovat a případně upravovat ručně. Díky transakci se však buď plat navýší všem, a až tento stav se do databáze trvale zapíše, nebo v případě chyby se nezmění nic a bude možné zkusit příkaz pustit znovu.

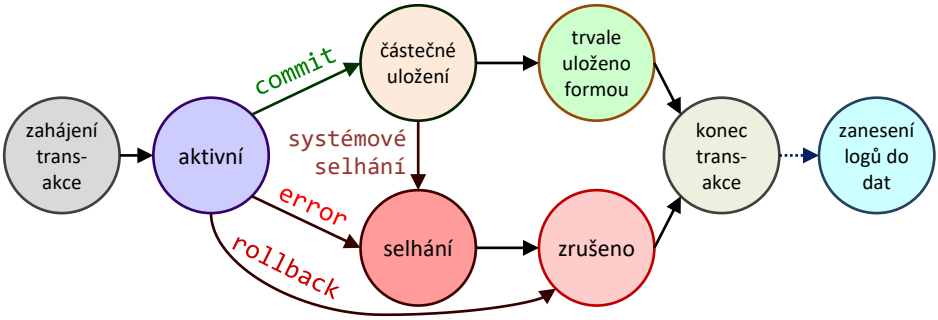
Během trvání transakce jsou veškeré úpravy v záznamech před ostatními uživateli či procesy skryty, a ti tak mohou číst pouze původní hodnoty záznamu (není-li uzamčen i pro čtení), které měl v okamžiku jeho zařazení do transakce, a to až dokud není tato ukončena.



Kromě záměrného ukončení transakce pomocí příkazů **commit** či **rollback**, může také dojít k jejímu automatickému ukončení databázovým systémem. K tomu dochází jednak, když nastane tzv. **timeout**, tedy čas vyhrazený na trvání transakce vypršel, například z důvodu programové chyby v aplikaci, která k databázi přistupovala. Další možností je tzv. **uváznutí**, při kterém na sebe vzájemně čekají dvě či více transakcí tak, že ani jedna nemůže pokračovat dále, dokud ta druhá neuvolní uzamčený záznam, a zároveň ta čeká na to samé u nich.



Tyto stavy uváznutí by měl databázový systém odhalit a některou z transakcí ukončit, aby ty ostatní mohly pokračovat a ta ukončená následně mohla pokus opakovat. Postup průběhu a možné způsoby zakončení transakce ukazuje následující diagram.

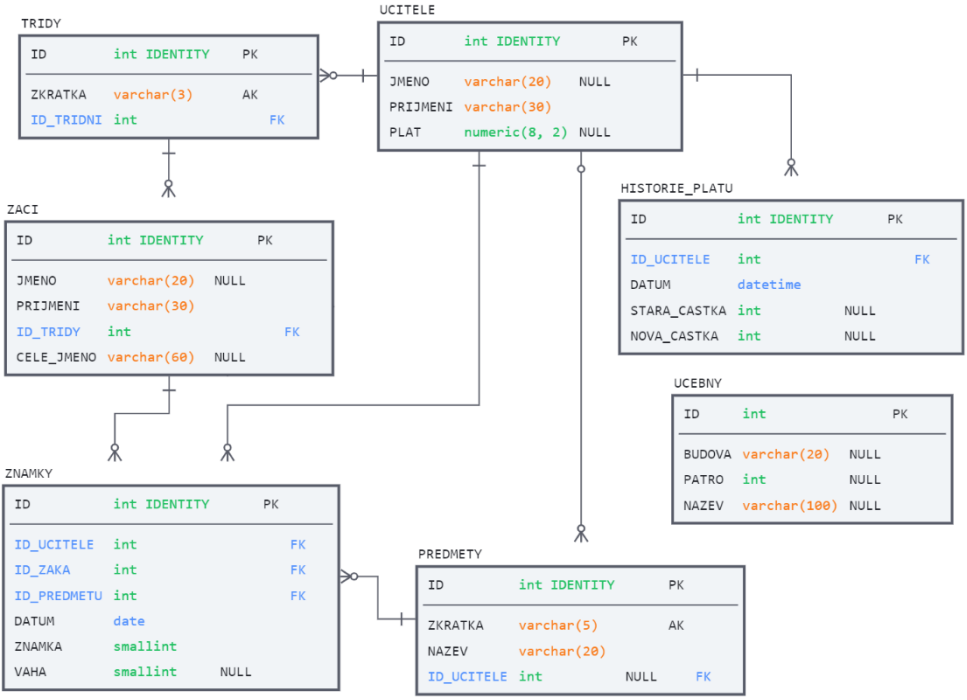


7 Další databázové objekty

Hlavními objekty v databázích jsou bezesporu tabulky. Nicméně se nejedná o jediné druhy objektů, se kterými se tam lze setkat. V této kapitole si ukážeme některé z těch, které se po tabulkách používají nejčastěji.

Tyto další typy objektů se skládají především z bloků kódu, obsahujících kromě SQL, DML či DDL, také další rozšířené příkazy, které v podstatě utváření jednoduchý programovací jazyk. V něm je možné používat např. proměnné, podmínky, cykly, výstup do „konzole“ atd. smísené právě třeba s SQL dotazy. Na SQL Serveru je tento jazyk označován jako T-SQL (Transact-SQL) a s jeho obdobou se lze setkat i v jiných databázových systémech (např. PL/SQL v Oracle nebo PSQL ve Firebirdu).

Příklady v této kapitole budou vycházet z jednoduché databáze školy, kterou najdete adrese programko.net/databaze/db/skola a jejíž datový model vypadá následovně.



7.1 Bloky kódu

Bloky T-SQL kódu lze používat i zcela samostatně, takže si některé konstrukce můžeme nejprve vyzkoušet na nich. Následující části kódu tedy stačí tak jak jsou, zapsat stejným způsobem, jako veškeré dříve prováděné SQL/DDL/DML příkazy a lze je tak přímo spustit.

Následující blok spuštěný v klasickém dotazovacím okně (*Query*) nástroje *SQL Server Management Studio* zopakuje 10x výpis čísla (od 0 do 9) pod sebe do okna *Messages*.

```
1: -- Výpis hodnot 0-9
2: declare @i int = 0;  -- Deklarace a nastavení proměnné i
3: while @i < 10        -- Cyklus opakující se dokud platí podmínka
4: begin                -- Začátek bloku cyklu
5:   print @i;          -- Výpis hodnoty do okna Messages v SSMS
6:   set @i += 1;       -- Navyšování počítalda i o +1
7: end;                 -- Konec bloku cyku
```

Z ukázky jsou patrné některé základní syntaktické principy jazyka T-SQL. Například deklarace proměnné se značí klíčovým slovem **declare**, proměnné jsou rozlišeny zavináčem (**@i**) před svým názvem (podobně jako např. v PHP znakem dolaru), změnu hodnoty proměnné je třeba uvést klíčovým slovem **set**, podmínka cyklu **while** není ani v závorce (jako např. v C#) ani zakončena klíčovým slovem (opakuj či **do**, jako např. v Pascalu), začátek (**begin**) a konec (**end**) se naopak používají podobně jako v **Pascalu** atd.

Další příkazy lze pozorovat také v následující ukázce, která tentokrát již pracuje se záznamy tabulky. Konkrétně postupně prochází jednotlivé záznamy tabulky **UCITELE**, načítá jejich příjmení ze sloupce **PRIJMENI** do proměnné **@Prijmeni** a vypisuje je do výstupního okna *Messages*.

```
-- Výpis příjmení učitelů
declare @Prijmeni varchar(30);            -- Deklarace proměnné
declare UciteleCursor cursor for          -- Deklarace kurzoru
  select PRIJMENI from UCITELE order by 1 -- definováno dotazem
open UciteleCursor;                       -- Otevření kurzoru
fetch next from UciteleCursor into @Prijmeni; -- Načtení 1. záz.
while (@@FETCH_STATUS = 0)    -- Opakovat dokud nebude konec
begin
  print @Prijmeni;            -- Výpis příjmení do výstupu
  fetch next from UciteleCursor into @Prijmeni; -- Čtení dalšího
end;
close UciteleCursor;          -- Uzavření kurzoru
deallocate UciteleCursor;     -- Uvolnění kurzoru z paměti
GO
```

V ukázce je tedy vidět, jakým způsobem je třeba postupovat pro čtení hodnot z tabulky. SQL dotaz se zde nepoužívá přímo, ale deklaruje se do tzv. kurzoru (**cursor**), jež pak lze v různých částech kódu opakovaně používat pro postupné načítání jednotlivých záznamů z výstupu tohoto dotazu. Tento přístup tak umožňuje obdobu proudového čtení dat, tedy postupného načítání a zpracování výstupních dat řádek po řádku, bez nutnosti načíst do paměti vše najednou.

Nastavený kurzor je tedy nejprve třeba otevřít (**open**). Poté lze cyklicky projít jednotlivé záznamy vrácené dotazem kurzoru pomocí `fetch next from` *`kurzor`* `into` *`proměnné`* (načti další záznam do proměnné), kde proměnné (je-li jich víc) je čárkami oddělený seznam deklarovaných proměnných, do nichž se načtou hodnoty ze sloupců (v pořadí určeném v dotazu kurzoru) aktuálního záznamu. Tento příkaz zároveň posune kurzor na další záznam, takže při příštím volání se bude číst ten. To, zdali se tento posun na další záznam povedl či nikoli, k čemuž dojde v případě, že bylo dosaženo posledního záznamu vráceného dotazem kurzoru, lze poznat že interní (má před

názvem hned dva zavináče) proměnné **@@FETCH_STATUS**, který má hodnotu nula pouze dokud se posun na další záznam podařil. Po zpracování všech záznamů je kurzor ještě potřeba uzavřít (**close**) a uvolnit z paměti (**deallocate**).

Načíst záznamy z tabulky pro zpracování T-SQL kódem již tedy umíme, tak si ještě vyzkoušíme jejich zápis. Následující příklad je rozšířením toho předchozího, který načítal příjmení učitelů. To se děje i zde, ovšem místo vypisování těchto názvů do výstupního okna příkazem **print**, se tentokrát budou jejich křestní jména přidávat do tabulky **UCEBNY**. Navíc před tento název přidáme text „Učitel: “, a zároveň, jelikož tato tabulka neměla definovanou sekvenci pro automatické generování hodnoty PK, tak přidáme i vlastní kód, který určí následující unikátní číslo řady ve sloupci **ID**.

```
-- Přepis jmen učitelů do tabulky UCEBNY s prefixem 'Učitel: '
declare @id int;                          -- Deklarace proměnné
declare @Jmeno varchar(20);               -- Deklarace proměnné
declare UciteleCursor cursor for          -- Deklarace kurzoru
  select JMENO from UCITELE order by 1 -- definovaného dotazem
open UciteleCursor;                           -- Otevření kurzoru
fetch next from UciteleCursor into @Jmeno; -- Načtení 1. záz.
while (@@FETCH_STATUS = 0)          -- Opakovat dokud nebude konec
begin
  set @id = (select max(ID) from UCEBNY) + 1; -- Hodnota pro PK
  insert into UCEBNY (ID, NAZEV)                -- Vložení názvu
    values (@id, 'Učitel: ' + @Jmeno);     -- do tab. ZANRY
  fetch next from UciteleCursor into @Jmeno; -- Další zazn.
end;
close UciteleCursor;                  -- Uzavření kurzoru
deallocate UciteleCursor;             -- Uvolnění kurzoru z paměti
GO
```

Pro vkládání nových dat lze tedy volat klasický příkaz **insert**, do jehož hodnot (**values**) dodáme hodnoty z proměnných, kam jsme je předtím načetli z jiné tabulky, popř. je poté ještě nějak upravili. Hodnotu pro PK sloupec **ID** zde získáme tak, že načteme maximum ze všech stávajících hodnot v tomto sloupci, kterou navýšíme o 1.

7.2 Pohledy

Pohled (**view**) je v databázi jakousi virtuální tabulkou, která ovšem neuchovává žádná vlastní data, pouze je čerpá z dat tabulek ostatních, tak jako klasický dotaz **select**. A SQL dotaz je zároveň právě tím hlavním, co obsah pohledu definuje.

Pro vytvoření pohledu je třeba určit jeho jednoznačný název, který nesmí být shodný s názvem jiného pohledu či tabulky, a výčet názvů sloupců, které pohled bude vracet, přičemž jejich datové typy se v této hlavičce pohledu blíže nespecifikují. Dále následuje již pouze SQL dotaz (viz kap. 2), jehož jedinou podmínkou je, že musí vracet tytéž sloupce (ve stejném počtu a pořadí), jako bylo definováno v hlavičce pohledu. Tento SQL dotaz, definující obsah pohledu, může být libovolně složitý, nesmí však obsahovat část pro řazení (**order by**), jelikož to se provádí až nad daty tohoto pohledu a v případě požadavku na jiný způsob řazení by se tak data musela přerovnávat vícekrát. V pohledu se také nepoužívají žádné T-SQL příkazy.

```
 1: /* Pohled průměrující známky za třídu a předmět (dle zkratek) */
 2: create or alter view PRUMERY_ZNAMEK (TRIDA, PREDMET, PRUMER)
 3: as
 4: select t.ZKRATKA,
 5:        p.ZKRATKA,
 6:        sum(z.ZNAMKA * coalesce(z.VAHA, 1)) /
               cast(sum(coalesce(z.VAHA, 1)) as float)
 7: from ZNAMKY z
 8:      left join ZACI s on s.ID = z.ID_ZAKA
 9:      left join TRIDY t on t.ID = s.ID_TRIDY
10:      left join PREDMETY p on p.ID = z.ID_PREDMETU
11: group by t.ZKRATKA, p.ZKRATKA;
12: GO
```

Tímto příkazem je pohled vytvořen (**create**) nebo aktualizován (**alter**), existoval-li již pod tímto názvem dříve. Pro čtení funguje podobně, jako klasická tabulka, jejíž data však pochopitelně nelze měnit. Data pohledu lze například načíst klasickým SQL dotazem, kde v části **from** uvedeme právě název tohoto pohledu.

```
1: /* Výběr dat z pohledu */
2: select *
3: from PRUMERY_ZNAMEK
4: where TRIDA = '2B'
5: order by TRIDA, PREDMET;
```

Takovýto SQL dotaz na data z pohledu lze samozřejmě omezit pouze na určité sloupce (**select**), doplnit libovolnou podmínkou (**where**), záznamy dále seskupovat (**group by**), spojit je s jinou tabulkou či pohledem (**join**) a výsledek libovolně seřadit (**order by**).

Pohledy tak umožňují uložit složitější anebo často se opakující SQL dotazy jako databázový objekt (jejich zpracování si tak databáze může předem nachystat a pak provádět rychleji), s plnou kontrolou a ohlídáním jejich validnosti a provázanosti (např. pak nelze jen tak smazat tabulku či její sloupce, na které se dotaz pohledu odkazuje), nebo rozložit složité dotazy na části a ty pak dalším dotazem či pohledem (jednotlivé pohledy mohou čerpat data i z jiných pohledů) zase poskládat, ať již horizontálně (**join**), vertikálně (**union all**) nebo obojí, do jediného výstupu.

7.3 Spouště

Spoušť (ve smyslu např. „spoušť u pistole“, v originálu **trigger**), označuje část kódu zapsanou v T-SQL jazyce, která se spustí, když dojde k určité události ovlivňující data určené tabulky.

Už víme, že s daty tabulky lze provádět tři základní operace (viz kap. 6): vložení nového záznamu (**insert**), úprava existujícího záznamu (**update**) a vymazání záznamu (**delete**). Zároveň lze rozlišovat také okamžik před (**before**) a po (**after**) některé z těchto tří událostí. A právě takto je pro každou tabulku možné definovat šest variant, kdy se spoušť (**trigger**), resp. její T-SQL kód, může automaticky spustit.

SQL Server ovšem nepoužívá variantu před (**before**), ale používá *namísto* (**instead of**), která se liší v tom, že není spuštěna před vykonáním dané operace, ale namísto ní. Pokud tedy má tabulka definovanou spoušť typu **instead of** pro některou ze tří operací, musí zároveň v této spoušti mít i kód pro její vykonání, jelikož tabulka ji automaticky již provádět nebude. Vkládaná či aktualizovaná data lze ve spoušti načíst z proměnné **INSERTED** (při mazání **DELETED**), se kterou se pracuje jako s virtuální tabulkou (její záznamy tvoří vkládané/upravované/mazané záznamy), i kdyby dotčený záznam byl jen jeden jediný (pak to bude tabulka s jedním záznamem). DML

příkazy volané z této spouště již pak znovu tuto spoušť nespustí. Takový postup je sice trochu více pracný oproti variantě **before**, používané v některých jiných DBS, ale zároveň to umožňuje mít nad celým procesem mnohem lepší kontrolu.

Před vložením (nebo *namísto* vložení, resp. při vkládání pod vlastní kontrolou) nového záznamu (**before insert** či **instead of insert**) můžeme například do vkládaných dat doplnit chybějícím atributům výchozí hodnoty, popř. nějaké údaje dohledat či dopočítat (např. další číslo v řadě dokladů pro daný rok, aktuální DPH dle data dokladu a číselníku DPH, vlastní řešení auto inkrementace ID apod.). V okamžiku před vložením lze zároveň ještě vložení samotnému zabránit (např. z důvodu nevalidity některých hodnot, duplicity záznamu, **null** v atributu, kde v této kombinaci ostatních údajů být nesmí apod.), buď jeho přeskočením (při variantě **instead of insert**) nebo vyvoláním výjimky (chybového přerušení operace, což je jediná možnost při variantě **before insert**).

Totéž platí i pro úpravu dat (**before update** či **instead of update**). Při aktualizaci záznamu je nicméně navíc možné, přečíst si jeho stávající hodnoty a porovnat je s těmi upravovanými, což při vkládání samozřejmě možné nebylo.

Co se týče mazání záznamu (**before delete** či **instead of delete**), tak zde je poslední možnost takovému výmazu zabránit. Díky tomu lze např. otestovat i vztahy, které klasickými databázovými metodami definovat nelze, a rozšířit tak možnosti kontroly konzistence i nad rámec integritních omezení pomocí vlastního T-SQL kódu.

Spouště aktivované až po události (**after insert**, **after update**, **after delete**) pak již operaci samotné zabránit, ani ovlivnit její průběh, nemohou. Lze však s jejich pomocí například zajistit logování (uložení informace do speciální

tabulky, že v daný okamžik, určitý uživatel s konkrétním záznamem provedl jistou operaci), přidání navazujících záznamů (např. po uzamčení příjemky či dodacího listu se na sklad zapíše příjem/výdej daného množství určených položek), nebo aktualizaci redundantních dat (např. aktualizovat celkový stavu skladu, aby se tato hodnota nemusela neustále znovu počítat ze všech pohybů na skladě od poslední inventury, což by mohlo značně zrychlit práci s takovým systémem).

Jedna spoušť může obsluhovat i více událostí jedné tabulky, je s tím však potřeba v jejím kódu počítat, pokud se některé reakce mají pro různé události lišit.

První příklad ukazuje spoušť, která nahrazuje používání **identity** pro automatické doplňování hodnoty **ID**. Spoušť je tedy *namísto vložení*, při kterém pouze se tato hodnota mění (určuje). Není-li sloupci **ID** nastaveno používání **identity**, nebude se automaticky jeho hodnota doplňovat, což lze vyřešit buď jejím důsledným doplňováním v rámci každého insertu, nebo třeba právě takovouto spouští. Ta tedy namísto obyčejného vložení, provede úpravu hodnoty **ID** tak, že nalezne maximální **ID** ve stávajících datech (pokud žádná nejsou, použije 0), k ní přičte 1, a zároveň upraví i případnou hodnotu **null** ve sloupci **BUDOVA**, kterou nahradí otazníkem (?).

```
 1: /* Spoušť pro automatické generování ID tabulky UCEBNY */
 2: create or alter trigger T_UCEBNY_BI_ID
 3:    on UCEBNY
 4:    instead of insert   -- instead of insert = místo insertu
 5: as
 6: begin
 7:   insert into UCEBNY
 8:   (ID, BUDOVA, PATRO, NAZEV)
 9:   select coalesce((select max(ID)
                    from UCEBNY), 0) + 1,  -- určí následující ID
10:          coalesce(BUDOVA, '?'),   -- změní NULL budovy na znak ?
11:    PATRO, -- informace o patře a názvu se použijí bez úprav
12:    NAZEV
13:   from INSERTED; -- INSERTED je virtuální tabulka obsahující
                      data, která měla být dotabulky vložena
                      (či v ní změněna změněna, jde-li o update)
14: end
15: go -- uloží výše provedené změny (vložení spouště) do DB
```

Kód spouště tedy začíná příkazem **create or alter trigger**, čili *vytvoř nebo uprav spoušť*, díky čemuž lze stejný příkaz použít, ať již spoušť se stejným názvem existuje či nikoli a dle potřeby se buď vytvoří a přidá do databáze jako nová, nebo se aktualizuje kód té stávající[10]. Dále se definuje databázová tabulka (**on**), změnu jejíchž dat má spoušť nahrazovat, a následuje výčet těchto událostí (v tomto případě **instead of insert**, tedy *namísto vložení*). Poté, za klíčovým slovem **as**, následuje vymezení bloku T-SQL kódu pomocí začátku (**begin**) a konce (**end**), mezi nimiž je tedy obslužný kód,

[10] I z těchto důvodů je dobré používat jednotnou konvenci pro pojmenovávání veškerých databázových objektů, aby nemohlo omylem dojít k duplikaci názvu žádných dvou z nich. Zde je použito schéma „T_<NázevTabulky>_<ZkratkaAkce>_<CoSpoušťŘeší>“.

který provede vlastní vložení záznamu do tabulky. Vše je zakončeno příkazem **go**, který potvrdí změnu databáze, tedy přidání či úpravu spouště a její aktivaci.

Následující ukázka obsahuje kód spouště, která má provádět vlastní kontrolu integrity nad rámec běžně definovatelných pravidel. Konkrétně tedy hlídá, aby jeden učitel byl třídním učitelem v maximálně jedné třídě (0-1).

```
/* Spoušť hlídající, aby učitel nebyl třídní ve více třídách */
create or alter trigger T_TRIDY_BI_TRIDNI
  on TRIDY
  instead of insert
as
begin
  -- uložení ID třídního nově vkládané třídy do proměnné
  declare @tridni int = (select top 1 ID_TRIDNI from INSERTED);
  if (@tridni is not null)   -- je-li třídní definován
  begin
    -- počet tříd, kde už je tento učitel třídním
    declare @pocet int = (
        select count(*)
        from TRIDY t
        where t.ID_TRIDNI = @tridni);
    -- vyvolání výjimky, což zabrání dalšímu průběhu ukládání
    if (@pocet > 0) -- počet > 0 => učitel už je třídním jinde
      throw 100000, 'Tento učitel je již třídním jinde', 1;
  end;
  -- uložení vkládaných dat do tabulky
  insert into TRIDY
  (ZKRATKA, ID_TRIDNI)
  select ZKRATKA, ID_TRIDNI from INSERTED;
end
go
```

Spoušť pro zjednodušení řeší pouze vkládání nové třídy (nikoli její update) a počítá s právě jedním vkládaným záznamem (nikoli hromadným insertem). Pokud je tedy vkládané třídě přiřazen třídní, který už je třídním v jiné již dříve vložené třídě, vložení se neprovede a dojde k vyvolání výjimky.

Následující příklad ukazuje, jak spoušť, která při vkládání (*insert*) či aktualizaci (*update*) záznamu v tabulce **ZACI** dopočítává hodnotu **CELE_JMENO**. Tato spoušť je tentokrát volána až po (*after*) uložení dat, takže řeší pouze jejich dodatečnou aktualizaci. Vkládaná nebo aktualizovaná data čerpá v obou případech z virtuální tabulky **INSERTED**.

```
 1: /* Spoušť dopočítávající hodnotu CELE_JMENO v tabulce ZACI
       při každé změně dat */
 2: create or alter trigger T_ZACI_AIU_CELE_JMENO
 3:    on ZACI
 4:    after insert, update   -- trigger se spouští po vkládání
                              -- (insert) i po úpravě (update) dat
 5: as
 6: begin
 7:   update z set
 8:     z.CELE_JMENO = i.JMENO + ' ' + i.PRIJMENI -- výpočet c. jm.
 9:   from ZACI z  -- ukládat do tabulky ZACI
10:        left join INSERTED i on i.ID = z.ID  /* propojení
                                s virtuální tabulkou ukládaných dat */
11:   where i.ID is not null
12: end
13: go
```

Následující spoušť má logovat změny platu všech učitelů do tabulky **PLATOVA_HISTORIE**. Je tak opět spouštěna až po (*after*) vložení či aktualizaci dat v tabulce **UCITELE**, avšak na základě těchto změn tentokrát vkládá data do tabulky jiné. Při aktualizaci dat přitom potřebuje znát nejen nově ukládanou hodnotu, ale i tu původní, kterou lze čerpat z tabulky **DELETED**. Ta

při update obsahuje původní hodnoty záznamu, přičemž čerpat z ní lze i při mazání záznamů.

```
 1: -- Spoušť logující změny platu učitelů do tab. PLATOVA_HISTORIE
 2: create or alter trigger T_UCITELE_AIU_PLAT
 3:    on UCITELE
 4:    after insert, update
 5: as
 6: begin
 7:   insert into HISTORIE_PLATU   -- Při změně ji vloží do hitroie
 8:     (ID_UCITELE, DATUM, STARA_CASTKA, NOVA_CASTKA)
 9:   -- Hodnoty, které budou uloženy do tabulky s historií platu
10:   select i.ID, CURRENT_TIMESTAMP, d.PLAT, i.PLAT
11:   from INSERTED i
12:        left join DELETED d on d.ID = i.ID /* DELETED – virtuální
               tabulka s původními daty (starý plat), která mají být
               přepsána daty z INSERTED (nový plat) */
13:   where d.PLAT is null or d.PLAT != i.PLAT /* změnu logovat, jen
              pokud se plat skutečně změnil (nebo se vkládá nový) */
14: end
15: go
```

7.4 Uložené procedury

Tzv. *uložené procedury* (*stored procedures*) zjednodušeně řečeno kombinují pohledy a spouště. Jako pohledy mohou vracet data a za tímto účelem být volány v rámci SQL dotazů, zároveň však nejsou definovány pouze dotazem, ale mohou obsahovat libovolný T-SQL kód jako spouště. Na rozdíl od nich však nereagují automaticky na žádné události či manipulaci s daty, ale pro jejich spuštění je nezbytné je vždy zavolat. To lze buď v rámci SQL dotazu, například jako zdroj dat v sekci **from** (místo klasické databázové tabulky), nebo zcela samostatně příkazem **exec** (jako *execute* např. `exec SP_DOPLN_A_VYPIS_CELA_JMENA_ZAKU 1;`).

```
 1: -- Procedura doplní celé jméno všem žákům a vrátí je
 2: create or alter procedure SP_DOPLN_A_VYPIS_CELA_JMENA_ZAKU
 3:   @Poradi bit = 0 -- Vstupní proměnná určí pořadí celého jména
 4: as
 5: begin
 6:   begin transaction; -- Update provedeme v transakci
 7:     update ZACI set
 8:       CELE_JMENO = iif(@Poradi = 0, -- Jakým stylem spojit?
 9:         JMENO + ' ' + PRIJMENI,  -- @Poradi=0: 'Jmeno Prijmeni'
10:         PRIJMENI + ' ' + JMENO); -- @Poradi=1: 'Prijmeni Jmeno'
11:   commit;
12:   select CELE_JMENO from ZACI;   -- Výstup z procedury
13: end
14: go
```

Předchozí procedura provede v rámci explicitně volané transakce aktualizaci dat v tabulce **ZACI**, v níž jim nastaví hodnotu do sloupce **CELE_JMENO** a výsledné hodnoty vrátí jako jednosloupcový výstup. Má jeden nepovinný (jelikož mu je při jeho deklaraci nastavena výchozí hodnota 0) vstupní parametr **@Poradi** (ř. 3), kterým lze určit, jak má být celé jméno složeno (pro 0 je to 'Jan Novák' pro 1 je to 'Novák Jan'). Tato procedura nebude fungovat současně se spouští **T_ZACI_AIU_CELE_JMENO** ze str. 91, která dělá totéž v pevně daném pořadí a změny procedury by potlačila (přepsala v **after update** části).

Tento materiál vznikl v rámci realizace projektu
Strategický rozvoj Univerzity Hradec Králové, reg. č. CZ.02.2.69/0.0/0.0/16_015/0002427

EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY